Applicazioni e limiti della classificazione di immagini con reti neurali convoluzionali in dispositivi mobili

Laureando

Fabio Maiani

Relatore

Carlo Fantozzi

Corso di Laurea Magistrale in Ingegneria Informatica

Data

19/04/2016

Anno Accademico

2015/2016

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

Questa tesi descrive la progettazione di un'applicazione per dispositivi Android che sia in grado di riconoscere oggetti eterogenei inquadrati dalla fotocamera del dispositivo, e successivamente archiviarli ed elaborarli in base alla categoria di appartenenza. Questa procedura costituisce un tentativo di generalizzazione dell'architettura di molte applicazioni esistenti; la tesi fornisce inoltre un proprio esempio di applicazione concreta di questo tipo. Per classificare gli oggetti presenti nelle foto si utilizzano le Deep Neural Network (DNN), uno strumento molto potente con cui attualmente si raggiunge lo stato dell'arte negli ambiti dell'object recognition, object detection e object segmentation. Si è scelto di selezionare alcune DNN già esistenti, sulla base delle loro performance e dimensioni, e di allenarle su un dataset appositamente costruito; sono state inoltre effettuate specifiche analisi per sincerarsi che le risorse necessarie, in particolare CPU ed energia, per un loro utilizzo in locale sul dispositivo mobile siano sufficienti.

Introduzione generale

Il Machine Learning è la scienza che permette che dei computer effettuino determinate azioni senza che essi siano stati esplicitamente programmati per eseguirle e rappresenta una delle aree fondamentali dell'intelligenza artificiale.

Il Deep Learning costituisce un ramo del Machine Learning e si basa su un insieme di algoritmi che tentano di modellare delle astrazioni di alto livello in dati; generalmente questi algoritmi prevedono molteplici stadi di processamento, aventi una struttura spesso complessa e normalmente questi stadi sono composti da una serie di trasformazioni non lineari. Come si vedrà più avanti, le convoluzioni sono largamente utilizzate nel Deep Learning, soprattutto per applicazioni di computer vision, che costituisce il campo di interesse di questa tesi. Le architetture utilizzate spesso sono tutt'altro che semplici.

Nel Deep Learning ci sono ancora molte incognite. La teoria che spiega perchè esso funzioni così bene è attualmente ancora incompleta e probabilmente non esiste nessun libro o guida che sia migliore di un'esperienza diretta.

Nonostante questo, le Deep Neural Network (DNN) negli ultimi anni stanno riscuotendo un grande successo, attirando l'interesse di molti. Per questo motivo nella tesi si farà uso di questo potente strumento per tentare di risolvere problemi di object recognition nell'ottica della computer vision, nello specifico per classificare oggetti eterogenei presenti all'interno di immagini.

L'utilizzo delle DNN è previsto all'interno di una specifica applicazione per dispositivi mobili, la quale verrà illustrata nel successivo capitolo. Al giorno d'oggi esistono molte applicazioni per dispositivi mobili che permettono, inquadrando un determinato oggetto con la fotocamera, di ricavare automaticamente delle informazioni da esso e di usarle di conseguenza per determinati scopi. Ad esempio con l'applicazione Evernote Scannable ([2]) è possibile, inquadrando dei documenti, riconoscere in automatico la tipologia del documento, il quale viene opportunamente ritagliato e salvato nel dispositivo stesso. In pratica è come possedere uno scanner sempre a portata di mano: in questo modo l'utente può salvare tutte le informazioni volute e

successivamente visionarle o condividerle.

Un'applicazione molto simile ad Evernote Scannable, sempre in questo contesto, è Microsoft Office Lens ([3]): anche qui semplicemente inquadrando un documento, questo viene ridimensionato e salvato su OneNote, un'altra applicazione di Microsoft per la gestione generali di annotazioni ed appunti. Qui inoltre è presente anche un motore OCR (Optical Character Recognition) per il riconoscimento automatico del testo eventualmente presente all'interno dell'immagine acquisita: in questo modo è possibile cercare, copiare o modificare alcune parole presenti in esso.

Un altro esempio è rappresentato dall'attuale applicazione della Banca Unicredit ([4]), con la quale è possibile pagare una bolletta semplicemente inquadrandola con il proprio smartphone. L'applicazione infatti riconosce in automatico il codice QR presente nella bolletta e da esso ricava tutti i dati necessari inserendoli in automatico nel dispositivo, predisponendo dunque il pagamento con l'utente che deve solo confermare la correttezza dei dati ricavati.

Gli esempi riportati rappresentano solo una piccola parte dell'intero insieme di applicazioni attualmente esistenti in contesti simili. Risulta chiaro che questa categoria di applicazioni, le quali analizzano in maniera automatica dei dati per poi utilizzarli per degli scopi specifici, è fortemente in espansione in questi ultimi anni. Proprio in base a questo si è deciso di progettare un'applicazione dello stesso ambito, ma più generale, ovvero un'applicazione che sia in grado di riconoscere automaticamente una discreta varietà di oggetti eterogenei presenti nella vita reale di un'utente e di ricavare ed organizzare i dati ottenuti. Nel prossimo capitolo si parlerà infatti proprio di quanto appena detto.

Capitolo 1

Schema generale dell'applicazione

In questo capitolo iniziale verrà presentato lo schema complessivo dell'applicazione che si prevede di realizzare e la progettazione formulata in questa tesi.

1.1 Schema dell'applicazione

L'idea generale di questa applicazione è la seguente: date delle immagini in input, ricavare dati significativi da esse in maniera automatica, ovvero senza che un utente effettui un qualsiasi tipo di intervento.

L'applicazione deve inoltre svolgere le sue elaborazioni in locale, non semplicemente ricavare dei dati ed inviarli ad un server esterno per un'ulteriore elaborazione, come ad esempio l'applicazione realizzata da H. Ghasemzadeh et al. [26]: in quest'ultimo caso si utilizzano svariati sensori che rilevano ed inviano i dati ad un gateway, il quale comunica con un server esterno che effettua le principali elaborazioni e prende decisioni in base ai dati ottenuti. Le applicazioni di questo tipo necessitano di una connessione dati sempre attiva (oppure il WiFi), il che comporta a lungo andare un considerevole consumo della batteria del dispositivo, oltre alla possibilità di qualche ritardo o perdita di dati. Un'elaborazione locale invece potrebbe potenzialmente evitare questi problemi, anche se bisogna considerare altri aspetti come ad esempio i tempi computazionali e lo stesso consumo energetico, che potrebbero diventare eccessivi.

Considerando elaborazioni locali, l'applicazione deve quindi riconoscere inizialmente cosa rappresenta una determinata immagine, poi selezionare in essa un'area d'interesse da cui verranno estratte le informazioni volute ed infine estrarre da tale selezione i dati. I dati finali verranno poi utilizzati per gli scopi d'interesse relativi all'ambito in cui l'applicazione verrebbe utilizzata.

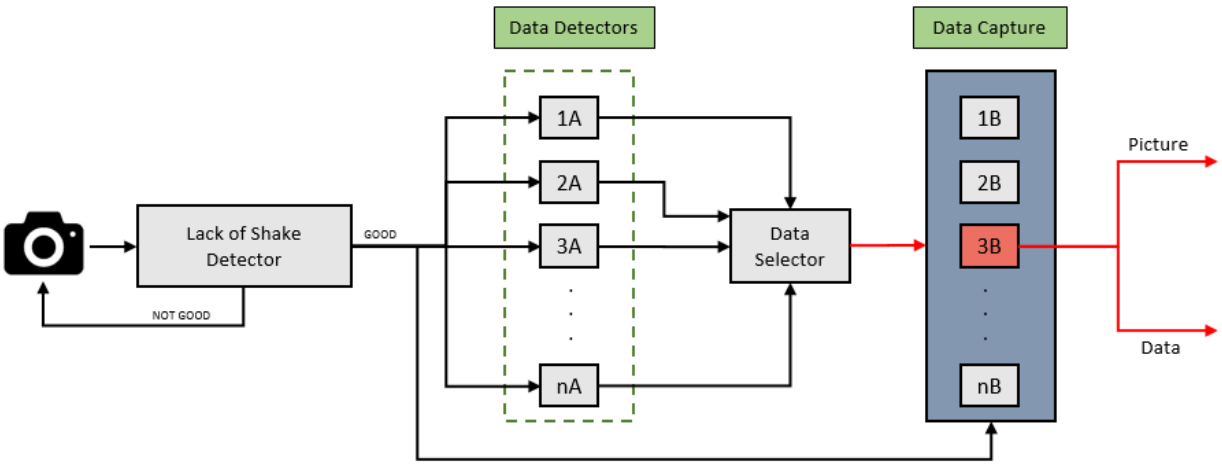


Figura 1.1: Apprendimento supervisionato

Lo schema generale dell'applicazione viene mostrato nella figura 1.1.

L'acquisizione iniziale dell'immagine si suppone che venga effettuata tramite la fotocamera di uno smartphone o di un tablet.

L'immagine precedentemente acquisita viene poi processata da un modulo chiamato *Lack of Shake Detector*, che come si può intuire serve semplicemente per verificare se l'immagine acquisita risulta essere mossa o non perfettamente definita. Nel caso fosse così, questo modulo richiede all'utente di inquadrare di nuovo l'obbiettivo per un ulteriore acquisizione con qualità migliore. Quando l'immagine viene considerata accettabile si può passare alla fase successiva.

Il primo grande blocco dell'applicazione riguarda l'insieme dei *moduli A* mostrati in figura 1.1; la loro unione costituisce il blocco dei *Data Detectors*. L'insieme di questi moduli ha lo scopo di identificare l'oggetto rappresentato dall'immagine acquisita, al termine di una loro elaborazione. Si parla dunque di un problema di classificazione e quindi della predizione di una specifica classe all'interno di un dominio discreto e finito di classi presenti in un determinato dataset; più in generale in questo caso si parla di un problema di object recognition.

Una volta identificato l'oggetto rappresentato nell'immagine, il *Data Selector* procede, se necessario, a selezionare solamente l'area d'interesse all'interno dell'immagine, area da cui si prevede di ricavare informazioni utili. Qui si parla dunque di un problema di object detection.

Selezionata l'area d'interesse, la fase successiva, costituita dall'insieme dei *moduli B* e detta *Data Capture*, prevede l'estrazione effettiva dei dati d'interesse. Sulla base della classificazione effettuata in precedenza, si prevede

di attivare uno specifico modulo per estrarre i dati relativi alla tipologia di classe predetta. Il collegamento che si vede nella figura 1.1 fra l'insieme dei *moduli B* e la parte precedente ai *moduli A* indica che l'immagine stessa deve essere passata anche alla parte di *Data Capture*, in quanto essa stessa rappresenta un'informazione utile che deve essere successivamente archiviata. In questo contesto, molto probabilmente sarà necessario utilizzare apposite librerie in base a che tipo di dati bisogna estrarre (ad esempio una libreria se vi sono dei codici QR o a barre, una libreria per OCR, ecc.).

Il risultato del modulo attivato (nella figura 1.1 ad esempio è il *3B*), che comprende dunque l'immagine ed i dati ricavati, verrà successivamente organizzato ed usato per altri scopi dipendenti dal campo in cui l'applicazione viene utilizzata.

1.2 Progettazione dell'applicazione

In questo paragrafo verrà presentata la complessiva fase di progettazione, tralasciando ancora per il momento i dettagli implementativi, rimandati al capitolo 7.

Tralasciando momentaneamente la realizzazione del modulo *Lack Of Shake Detector*, dato che non dovrebbero esserci grossi problemi nella sua realizzazione, il primo grande ostacolo è presente nella realizzazione dei *Data Detectors*. Ci sono molti strumenti che permettono di identificare un oggetto in un'immagine, come ad esempio il modello Bag Of Visual Words [8], tramite i descrittori di feature SIFT o SURF, oppure l'utilizzo delle Support Vector Machine (SVN); tuttavia in questi ultimi anni c'è un nuovo strumento che sembra essere molto promettente e che viene sempre più utilizzato: le Deep Neural Networks [9], in particolare una specifica tipologia di esse nota come Convolutional Neural Networks (CNN), nome che deriva dalle operazioni di convoluzione in esse presenti. Sono stati effettuati diversi studi di confronto fra le DNN ed i metodi prima citati, come ad esempio il lavoro di Phillip Fischer et al. [10], ed è stato visto che con le DNN si ottengono complessivamente dei risultati migliori, in termini di classificazioni corrette.

La scelta dell'utilizzo delle reti neurali convoluzionali come strumento di classificazione delle immagini dell'applicazione presentata nei precedenti paragrafi, ha portato tuttavia una notevole dose di informazioni da dover assimilare. Per questo motivo, nel corso dello studio fatto, si è deciso di puntare a realizzare solo una parte del progetto inizialmente concepito e cioè il primo dei due grossi blocchi di cui esso è costituito, relativo all'identificazione automatica dell'oggetto rappresentato dall'immagine acquisita (si veda figura 1.1), cercando di realizzarla al meglio in modo che in futuro sia possibile un

suo potenziale utilizzo per sviluppare il resto dello schema iniziale, o almeno un'altra parte di esso. Tutte le informazioni raccolte credo risultaranno comunque utili a questo fine, anche in caso di un qualche cambio di direzione nel procedimento.

L'idea quindi è quella di tentare un utilizzo locale su un dispositivo Android delle reti neurali convoluzionali per classificare le immagini di uno specifico dataset e di verificare il loro funzionamento.

1.2.1 Prerequisiti necessari

Per poter avanzare nel progetto è necessario essere già a conoscenza di una serie di concetti. In particolare è necessario avere compreso come funzionano in generale le reti neurali convoluzionali (capitolo 3), per poter in questo modo ideare o modificare la propria rete necessaria allo scopo.

Risulta poi indispensabile aver familiarizzato con un framework che utilizza queste reti, che in questa tesi risulta essere Caffe (capitolo 4): essere a conoscenza di come funziona tale framework risulta vitale per qualsiasi operazione si debba effettuare sulla propria rete, anche per il suo allenamento.

Dato che un dispositivo mobile ha la caratteristica di essere limitato, per quanto riguarda l'energia a sua disposizione per effettuare operazioni, dalla sua batteria, sarà necessario effettuare delle considerazioni progettuali sul consumo delle risorse che la tipologia di reti scelte necessita di avere per un'implementazione locale; risulta chiaro che sarà necessario stabilire con precisione che tipo di rete neurale convoluzionale creare e successivamente testare, al fine di evitare il sorgere di problemi legati appunto alle risorse disponibili sul dispositivo. Studiare i parametri di ciascun layer, l'architettura generale di una CNN ed il framework scelto per l'implementazione finale risultano dunque essenziali in questo contesto.

Prima di iniziare l'effettiva implementazione del progetto, sarebbe opportuno fare dei veloci test con operazioni simili a quelle da realizzare, ad esempio effettuare dei test su alcuni semplici dataset, al fine di apprendere direttamente sul campo le strategie necessarie per ottenere dei buoni risultati e cercare di evitare possibili errori futuri.

1.2.2 Procedura generale per avere una buona CNN

Innanzitutto, qualunque forma la CNN abbia, è possibile suddividere la realizzazione di una CNN in due sostanziali fasi: training della rete ed il suo successivo test. E' bene tenere presente che i tempi della prima fase non risultano essere troppo importanti per lo scopo di questo progetto, in quanto l'allenamento di una rete, indipendentemente dalla piattaforma dove essa

andrebbe utilizzata alla fine, verrebbe sempre effettuata su un computer e costituirebbe solamente una fase preliminare per l'ottenimento di un buon modello per la classificazione. Ciò che invece è di vitale interesse è proprio la seconda fase.

L'idea iniziale è quindi quella di creare, basandosi sulle conoscenze assimilate, allenare e testare una CNN inizialmente su un computer e, una volta ottenuti dei risultati soddisfacenti, tentare il test della rete ottenuta anche sulla piattaforma Android.

1.2.3 Modalità di training

Come si vedrà nel capitolo 4, una CNN, una volta definita, può essere allenata ed utilizzata in due modalità, ovvero tramite CPU o GPU. Nel secondo caso sono necessari i driver CUDA di NVIDIA ed ovviamente una GPU compatibile con tali driver. Purtroppo nel computer a disposizione per effettuare il training della rete non è presente una scheda grafica che soddisfi i requisiti richiesti. Quindi per allenare una rete sarà possibile utilizzare solo la modalità CPU, decisamente più lenta rispetto all'altra modalità. Essendo i tempi di training non eccessivamente importanti per il progetto, questo potrebbe non essere un grosso problema, tuttavia bisognerà vedere quanto tempo è effettivamente necessario per un allenamento soddisfacente, tenendo cioè conto del fatto che molto probabilmente sarà necessario cambiare alcuni parametri più volte, riavviando quindi l'allenamento, per avere dei buoni risultati.

Dato che inizialmente si procederà ad effettuare dei test sempre sullo stesso computer, risulta ovvio che anche la seconda fase (il test della rete) verrà effettuata in modalità CPU. Successivamente, quando si tenterà di importare la rete su Android, si potrà verificare se entrambe le modalità sono disponibili o meno. Attualmente non ci sono molte informazioni di Caffe su questa piattaforma, quindi per il momento è impossibile sapere cosa succederà al momento della sua importazione (potrebbe anche non funzionare in generale).

Utilizzo dell'operazione di fine-tuning

Caffe mette a disposizione due metodologie per allenare una rete: un training completo della rete oppure effettuare un'operazione di fine-tuning. La seconda operazione è descritta nel paragrafo 4.7 e presenta molti aspetti interessanti. L'unico problema è che con essa non è praticamente possibile effettuare modifiche sostanziali alle reti considerate.

Tuttavia molti programmatori che utilizzano il framework Caffe consigliano vivamente un'operazione di questo tipo per coloro che non posseggono

un dataset eccessivamente grande ed inoltre non dispongono di un hardware troppo avanzato. Infatti con questa operazione sarebbe possibile avere a disposizione i modelli delle reti più avanzate e performanti del momento, costruite e testate nel corso di mesi o anni da parte di team specializzati aventi a disposizione abbondanti risorse economiche. Con i tempi e le risorse a disposizione, credo che difficilmente sarebbe possibile creare una rete più performante delle loro. E' vero che, per il progetto in questione, potrebbe bastare anche una rete con prestazioni inferiori, ma riflettendo più in generale, se fosse possibile applicare una rete del genere, si potrebbero ottenere buoni risultati anche con dataset molto diversi fra loro e magari anche con dataset abbastanza insidiosi, con ad esempio alcune classi che si assomigliano. Il fatto poi che questo aiuti anche con i tempi di allenamento, in quanto l'allenamento tramite fine-tuning di una rete è molto più veloce, non può che fare comodo, dato che avendo a disposizione solo la modalità CPU per allenare una rete, per quanto modesta essa sia, sarebbero necessari diversi giorni o anche settimane, tenendo conto che bisogna trovare la giusta configurazione dei parametri.

All'università di Standford, in un corso dedicato allo studio delle CNN, all'interno di una presentazione ([11]) viene dedicata un'intera sezione all'utilizzo della tecnica di fine-tuning. Qui sotto vengono riassunti i vantaggi che secondo tale presentazione porta il fine-tuning:

- un'ottimizzazione più robusta. Con un fine-tuning si parte di base con un'inizializzazione già buona dei parametri, anzi spesso ottima dato che i modelli a disposizione sono stati allenati con i dataset più complessi attualmente esistenti, aventi un numero molto elevato di classi. Le feature che sono state dunque costruite sono già molto robuste ed è sufficiente adattarle (approssimarle) al nuovo dataset, che normalmente possiede un numero di immagini e di classi inferiore rispetto ai dataset utilizzati per la creazione di questi modelli. Questo fatto porta spesso il fine-tuning ad avere prestazioni superiori su determinati dataset minori, rispetto alla possibilità di allenare una rete da zero, come è mostrato in Ken Chatfield et al. [21] e in R. B. Girshick et al. [24]; quest'ultimo riguarda solo l'ambito dell'object detection, ma la funzione del fine-tuning rimane comunque la stessa.

- necessità di meno dati. Il fine-tuning risulta ottimo per i dataset di piccole dimensioni e funziona discretamente anche sui dataset aventi un numero limitato di immagini per ogni classe. E' tuttavia consigliato, se possibile, avere un dataset con un buon numero di immagini per ogni classe e magari con una qualità decente: questi fattori aiutano sempre la classificazione.

- allenamento più veloce. Questo è scontato ma comunque fondamentale per chi non possiede un hardware adeguato e quindi non vuole perdere troppo tempo per la fase di training.

Come si evince sempre dalla presentazione, attualmente l'operazione di fine-tuning costituisce lo stato dell'arte negli ambiti dell'object recognition, dell'object detection e dell'object segmentation.

Inoltre non vi sarebbero problemi di reperimento dei modelli pre-allenati delle reti, in quanto il Model Zoo di Caffe offre una vasta scelta di questi.

Per i motivi fin qui citati è stato scelto di applicare la tecnica di fine-tuning per ottenere il modello finale da importare sulla piattaforma di Android.

1.2.4 CNN da considerare

Tuttavia con la scelta dell'utilizzo dell'operazione di fine-tuning emerge un nuovo problema e cioè su quale rete applicare il fine-tuning. Sarà dunque necessario studiare quali sono le reti più adatte allo scopo del progetto.

A questo proposito, l'analisi che verrà illustrata nel capitolo 3 risulterà essere molto utile, in quanto per tutte le reti che si vedranno sarà possibile reperire un modello pre-allenato sul Model Zoo di Caffe e di conseguenza effettuare un fine-tuning su tali reti.

Bisognerà però tenere conto, oltre alle prestazioni stesse delle reti in termini di accuracy, anche del possibile eccessivo consumo di risorse che esse potrebbero utilizzare. Per questo motivo verranno svolte delle analisi e delle considerazioni, che saranno presentate nel capitolo 6. Lo stesso capitolo mostrerà che le tre reti più interessanti per il progetto risultano essere AlexNet, NIN e GoogLeNet.

Si dovrà dunque procedere con un'operazione di fine-tuning per ciascuna di queste reti, fino ad ottenere un buon modello per ognuna di esse.

Si è deciso di procedere con più reti invece di una sola, in quanto si hanno in questo modo più possibilità di successo, oltre al fatto di poter confrontare pregi e difetti di ciascuna rete. In caso di dubbio sul fatto di selezionare o meno una rete, conviene forse considerarla ed escluderla successivamente dopo un suo test effettivo su Android.

1.2.5 Costruzione del dataset

Per lo scopo del progetto, prima ancora di allenare le reti, sarà necessario creare un proprio dataset. L'operazione di fine-tuning permette di poter utilizzare meno dati in input in generale, ma comunque per ogni categoria presente nel dataset dovranno essere presenti diverse centinaia di immagini,

cercando di non avere un numero di elementi troppo diverso fra tutte le classi, per non creare situazioni di sbilanciamento fra esse. Per rendere la rete più robusta sarà necessario disporre di immagini che rappresentino l'oggetto da classificare con angolazioni diverse, con sfondi diversi e magari anche con situazioni di luminosità (anche presenza di ombre) differenti.

Per ciascuna categoria del dataset, le immagini raccolte dovranno essere suddivise in un training set ed un validation set; quest'ultimo solitamente contiene una minima parte del totale delle immagini di una classe. E' importante tuttavia fare attenzione al fatto che le immagini presenti nel validation set, non siano presenti anche sul training set, dato che altrimenti non si avrebbero dei dati di accuracy sensati perchè non si testerebbe l'effettiva capacità della rete di classificare immagini mai viste prima. Sarebbe infatti opportuno, sempre al fine di migliorare il modello finale della rete, avere nel validation set immagini con sfondi mai utilizzati nel training set: in questo modo si suppone che i dati finali di accuracy rispecchino di più la realtà in cui l'applicazione finale dovrebbe essere utilizzata.

Per ogni classe di immagine, la situazione ottimale sarebbe quella di avere nel training set sia immagini rappresentanti l'oggetto in questione con uno sfondo e sia immagini rappresentanti solamente l'intero oggetto, senza nessuno sfondo: in questo modo la rete, durante la fase di allenamento, diventerebbe più robusta, riuscendo a distinguere in maniera migliore l'oggetto di interesse anche in situazioni più complicate. Risulta poi ovvio che per quanto riguarda le immagini con lo sfondo, conviene avere una certa varietà di sfondi, per rendere la rete più flessibile.

Può essere utile inoltre preparare un ulteriore modesto insieme di immagini, che rappresentino tutte le categorie del dataset, per un test finale delle reti. Tali immagini dovrebbero rappresentare la realtà con cui normalmente esse verrebbero acquisite con l'applicazione finale, ad esempio una persona che fa una foto ad un documento sul tavolo oppure che tiene il documento in mano.

1.2.6 Allenamento delle reti

Una volta selezionate le reti e costruito il dataset, si procederà con un'operazione di fine-tuning su ogni rete. Sarà necessario individuare all'incirca quante iterazioni sono necessarie per ottenere un buon modello senza overfittare troppo la rete. Non è detto che tale numero sia lo stesso con ogni rete, per cui bisognerà prestare attenzione ad ognuna di esse. Contemporaneamente bisognerà prestare attenzione ad trovare una giusta combinazione dei parametri del solver (questo termine verrà chiarito nel capitolo 4), anche qui per evitare overfitting oppure una non corretta applicazione del fine-tuning.

Per quando riguarda le modifiche alla definizione della reti si dovranno applicare quelle standard indicate nel paragrafo 4.7 dedicato all'utilizzo della tecnica di fine-tuning con Caffe.

Una volta in possesso di un buon modello finale per ciascuna rete (buona accuracy sul validation set e su un eventuale test set), si potrà eventualmente passare alla fase successiva. In caso contrario bisognerà cambiare la configurazione del solver oppure procedere con più iterazioni per verificare se ci sono ulteriori miglioramenti.

1.2.7 Porting del framework

Ottenuti i modelli, nel caso le performance di questi fossero buone, si potrà procedere al test delle reti su Android. Sarà tuttavia prima necessario analizzare le librerie che il framework Caffe necessita per effettuare un suo porting su Android. Essendo Caffe completamente scritto in C++, molto probabilmente sarà richiesto l'utilizzo di Android NDK, strumento utile proprio al fine di importare del codice nativo sulla piattaforma Android. Sarà senz'altro utile verificare se esistono altri tentativi di porting del medesimo framework, per evitare di perdere un'eccessiva quantità di tempo su questa fase. Anche dei porting non funzionanti si potrebbero rilevare utili per comprendere in che maniera importare le librerie necessarie per un corretto funzionamento di Caffe su Android.

1.2.8 Applicazione per il test delle reti

Come testbed per lo schema generale dell'applicazione precedentemente introdotto si è scelto di utilizzare il programma PIPPI ([5], [6]), ovvero il Programma di Intervento Per la Prevenzione dell'Istituzionalizzazione. Il programma PIPPI ha lo scopo di monitorare il benessere psicologico dei bambini che stanno attraversando situazioni familiari complicate, come ad esempio frequenti traslochi, problemi economici e altro ancora. Gli operatori sociali che operano nel programma PIPPI sono gli addetti a ricavare tutte le informazioni necessarie, tramite l'utilizzo di opportuni schemi, moduli e questionari. Si intuisce che all'interno di questo programma gli operatori sociali hanno la necessità di digitalizzare ed organizzare di continuo una certa varietà di immagini rappresentanti proprio gli strumenti cartacei appena menzionati. Dato che si sta in generale parlando di immagini rappresentanti oggetti eterogenei, questa risulta essere una situazione reale ideale per lo scopo dell'applicazione che si vuole realizzare. Tramite l'applicazione ideata si potrebbe infatti semplicemente scattare una foto ad ognuno di essi e di conseguenza l'oggetto nella foto verrebbe riconosciuto ed archiviato in base

alla sua categoria di appartenenza in maniera automatica, operazione che risulterebbe molto utile e comoda per gli operatori sociali. Per questi motivi è stato considerato il programma finora descritto per effettuare i test dell'applicazione voluta e proprio per esso è stato costruito un apposito dataset che verrà illustrato successivamente.

Nel caso in cui il porting del framework andasse a buon fine, l'idea è quella di creare una semplice applicazione basilare che permetta di classificare immagini appartenenti alle categorie del proprio dataset, cioè al dataset creato per il programma PIPPI in questo caso, con le reti neurali convoluzionali scelte per tale scopo. Non ci si focalizzerà dunque sull'estetica della UI, dato che lo scopo di tale applicazione è solo quello di testare le reti. Infatti si ricorda che l'intero processo costituisce solo una parte dell'intero progetto inizialmente ideato e quindi rappresenta solo una parte della vera e propria applicazione finale. Per questo motivo risulta momentaneamente di scarso interesse l'aspetto della UI, dato che potenzialmente manca ancora buona parte del lavoro. La semplice applicazione da realizzare di fatto potrebbe costituire un modulo di una qualsiasi applicazione Android che necessitasse di un'operazione di object recognition, relativa ad un suo dataset di interesse (sarà necessario un allenamento della rete con quel dataset ovviamente). In caso di un'elaborazione troppo lunga di un'immagine, potrebbe essere necessario lavorare in background con un altro thread su Android, dato che non è ovviamente possibile bloccare il main thread per un arco di tempo troppo lungo.

Un aspetto che tornerebbe molto utile sarebbe quello di separare la fase di inizializzazione di una rete, cioè il caricamento del suo modello, dall'effettivo suo utilizzo per le predizioni. In questo modo sarebbe possibile mantenere in memoria i dati ricavati dalla fase di inizializzazione ed utilizzarli ogni volta per classificare le immagini, senza dover quindi ricaricare il modello. Questo sicuramente farebbe risparmiare tempo, ma bisogna vedere se il sistema Android permette di mantenere in memoria una discreta quantità di dati senza uccidere l'applicazione.

Risulta poi ovvio che, al fine di poter utilizzare le reti in questione, sarà necessario caricare in qualche modo i file necessari per il corretto funzionamento delle reti. Nello specifico questi file sono il file *deploy.prototxt*, il file *synset.txt* ed il file *.caffemodel*, che saranno presentati sempre nel capitolo 4. Dato che in questo caso si vogliono testare ben 3 reti, conviene caricare tutti i file necessari di ogni rete, organizzandoli magari in apposite cartelle, e successivamente tramite il codice dell'app selezionare la rete che si vuole utilizzare, modificando i path per raggiungere i file di una specifica rete. In questo modo è sufficiente una veloce ricompilazione del codice per cambiare la rete da utilizzare nei test.

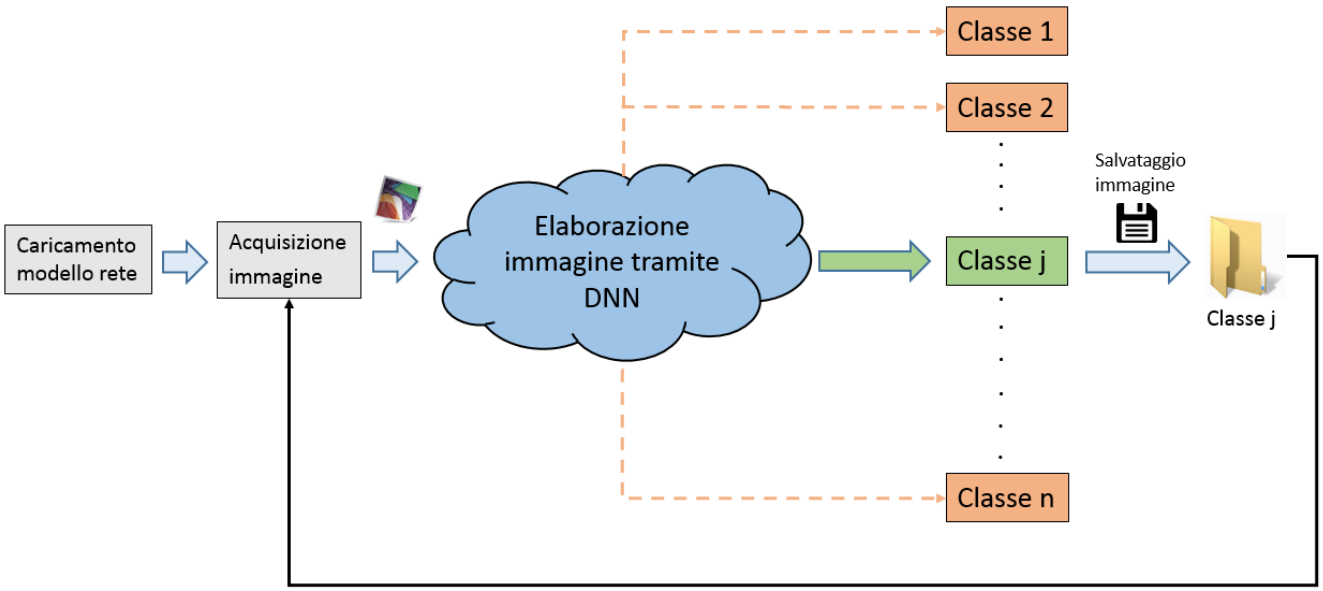


Figura 1.2: Schema generale dell'app per il test delle reti

In figura 1.2 viene mostrato lo schema generale che l'app basilare deve seguire.

1.3 Schema riassuntivo

A titolo di comprensione, verrà ora mostrato uno schema riassuntivo dell'intera fase di progettazione fin qui illustrata (figura 1.3).

1.4 Introduzione ai capitoli successivi

Si è dunque deciso di considerare le CNN per affrontare la prima parte dello schema globale dell'applicazione visto in figura 1.1. Per utilizzarle è ovviamente necessario prima di tutto conoscere le nozioni elementari di una rete neurale, dato che molti concetti sono gli stessi o molto simili, e successivamente capire come funzionano nello specifico le CNN. I capitoli 2 e 3 servono esclusivamente per questo scopo.

Il capitolo 4 sarà poi dedicato all'introduzione di uno dei più famosi framework per le CNN, utilizzato in questa tesi, chiamato Caffe. Qui verrà mostrato nel dettaglio com'è composta una CNN, la definizione delle varie

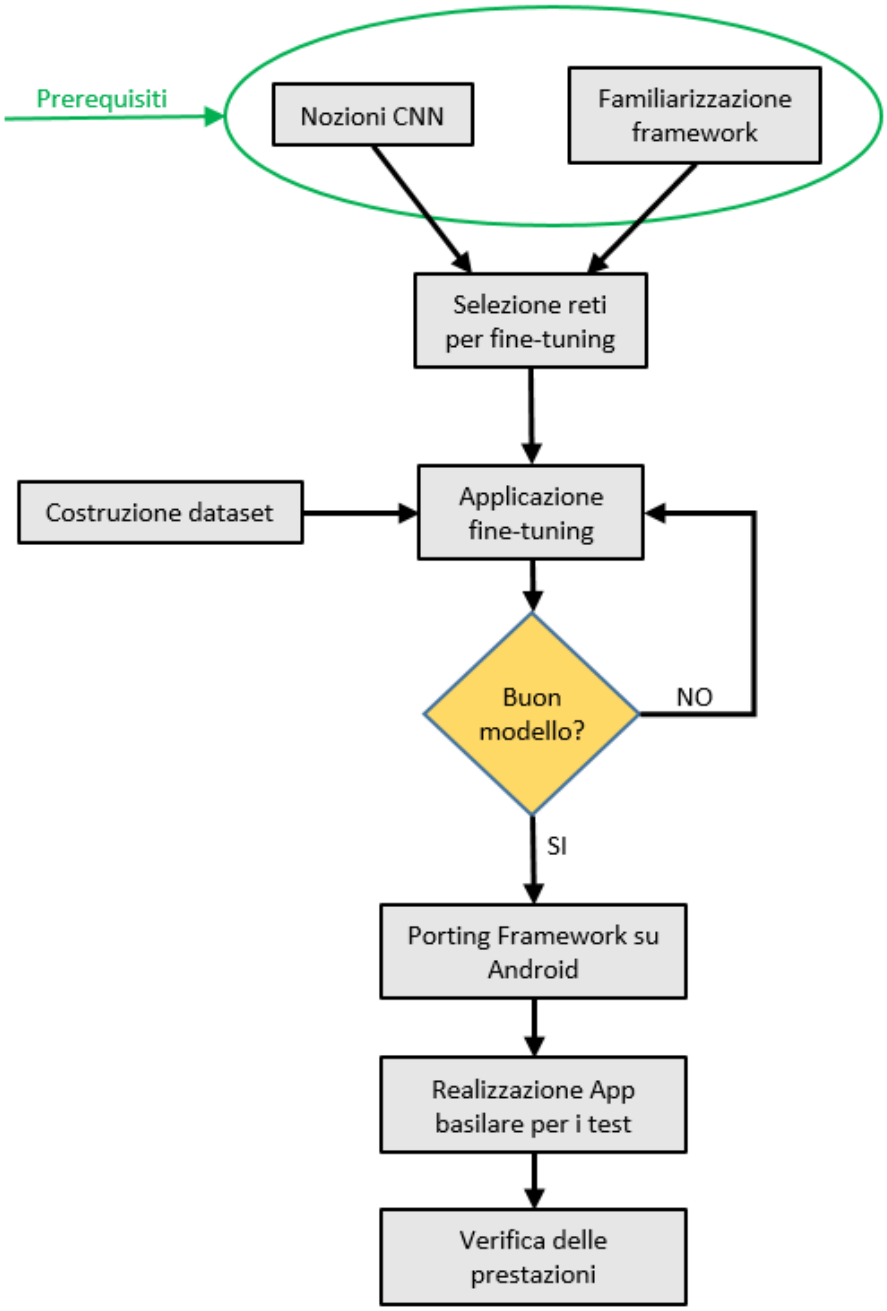


Figura 1.3: Schema generale del progetto

tipologie di layer ed in che modo allenare una CNN, il tutto utilizzando il framework in questione.

Il capitolo 5 introdurrà invece le reti potenzialmente più interessanti per i fini del progetto, reti che risultano essere fra le più famose ed efficienti CNN attualmente esistenti, in quanto quasi tutte hanno vinto almeno una fra le varie competizioni annuali a cui i vari team di sviluppo di CNN partecipano. Questo capitolo risulterà utile anche per effettuare una possibile selezione delle reti considerate.

Come già detto all'inizio di questo capitolo, tutte le elaborazioni dell'applicazione devono essere effettuate localmente. Bisognerà quindi capire se è possibile usare le reti neurali convoluzionali su un dispositivo mobile e, nel caso fosse così, capire se un loro utilizzo, oltre ad essere utile ai fini della classificazione da eseguire, risulta fattibile in termini di tempi di elaborazione e di consumo energetico. A questo proposito verrà dedicato in seguito l'intero capitolo 6.

L'ultimo capitolo, il capitolo 7, verrà dedicato invece ai vari dettagli implementativi del progetto e alla validazione delle reti selezionate sul dispositivo Android.

Per il momento ci si concentrerà su quanto appena detto, lasciando la realizzazione del resto dell'applicazione di figura 1.1 per una eventuale fase successiva.

Capitolo 2

Le reti neurali

Verranno ora brevemente presentate le principali caratteristiche delle reti neurali, un potente strumento utilizzato nel campo dell'intelligenza artificiale. Ci si focalizzerà sulle loro principali caratteristiche, in modo tale da avere un quadro generale introduttivo dei concetti che risultano comunque necessari per comprendere al meglio le reti neurali convoluzionali, le quali verranno illustrate nel capitolo successivo.

2.0.1 Un po' di storia

Di per sè le reti neurali non sono per nulla recenti, basti pensare che il primo modello di neurone artificiale fu proposto addirittura nel 1943 ([13]); nei successivi anni ci sono stati ulteriori studi e miglioramenti, fra cui l'invenzione del modello del percettrone da parte di Rosenblatt, fra gli anni '50 e gli anni '60, e l'introduzione del famoso algoritmo di retropropagazione dell'errore nel 1986, utile per allenare le proprie reti. Tuttavia negli anni '90, l'introduzione di altre metodologie come le support vector machine ed altri classificatori lineari avevano gradualmente offuscato la popolarità delle reti neurali, le quali venivano considerate complicate, costose in termini computazionali e con prestazioni non eccezionali rispetto ai nuovi metodi. L'avvento del **Deep Learning**, verso la fine degli anni 2000, ha invece fatto rinascere uno spiccato interesse per esse.

Il vero impatto del Deep Learing in larga scala iniziò nell'ambito del riconoscimento vocale (speech recognition) attorno al 2010, quando due dipendenti della Microsoft Research, Li Deng e Geoffrey Hinton [7], si accorsero che utilizzando grandi quantità di dati per il training, i quali venivano dati in pasto a delle deep neural network progettate con degli output layer piuttosto grandi, si producevano delle percentuali di errore decisamente inferiori rispet-

to agli stati dell'arte in quel campo di allora, come ad esempio il Gaussian Mixture Model/Hidden Markov Model (GMM-HMM).

Le nuove scoperte nel campo dell'hardware, come segnalato ad esempio da Yann LeCun et al. [12], hanno senz'altro contribuito all'aumento di interesse per il Deep Learning. In particolare, le nuove sempre più potenti GPU sembrano essere adatte a svolgere gli innumerevoli calcoli matematici di matrici e vettori presenti nel Deep Learing. Le GPU attuali infatti permettono di ridurre i tempi di allenamento di una DNN dall'ordine di settimane a quello di giorni.

2.1 Nozioni basilari

Nel campo del machine learning e della scienza cognitiva, le reti neurali artificiali (Artificial Neural Network, ANN) sono una famiglia di modelli ispirati alle reti neurali biologiche; in pratica il sistema nervoso centrale degli animali, in particolare il cervello. Esse vengono utilizzate per stimare o approssimare funzioni che possono dipendere da un numero elevato di input, molti dei quali spesso non noti. Le reti neurali artificiali vengono generalmente presentate come dei sistemi di "neuroni" fra loro interconnessi, tra i quali avviene uno scambio di messaggi. Ciascuna connessione ha un relativo peso associato, detto **weight**; il valore dei weight è regolabile in base all'esperienza in questione e ciò rende le reti neurali uno strumento adattabile ai vari tipi di input e con la capacità di apprendere.

2.1.1 Paradigma di apprendimento per reti neurali

A questo punto è bene conoscere il concetto di paradigma di apprendimento in relazione alle reti neurali. Per paradigma di apprendimento si intende la metodologia con cui si addestra la rete di neuroni. Operata questa scelta si passa a decidere secondo quali regole variano i singoli weight, cioè la legge di apprendimento. I paradigmi di apprendimento per le reti neurali si possono suddividere in:

- apprendimento supervisionato,
- apprendimento non supervisionato,
- apprendimento per rinforzo.

L'apprendimento supervisionato, che segue lo schema della figura 2.1, prevede l'utilizzo di un set di esempi ovvero di un training set, composto dalle coppie X_k e Y_{dk}, rispettivamente il k-esimo ingresso e la k-esima uscita

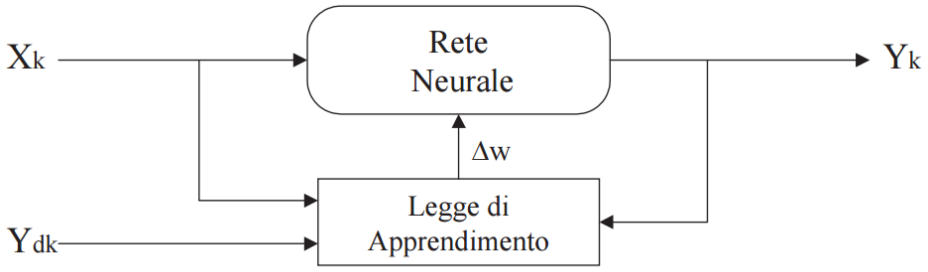


Figura 2.1: Apprendimento supervisionato

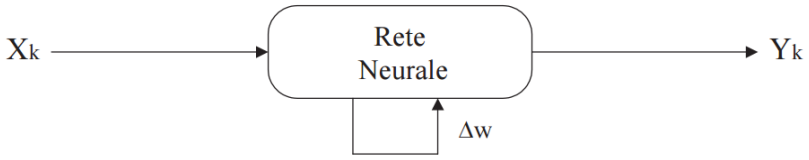


Figura 2.2: Apprendimento non supervisionato

desiderata. L'uscita reale, Y_k, viene confrontata con quella desiderata e si aggiustano i weight in modo tale da minimizzare la loro differenza. Il training set viene ciclicamente considerato finché Y_k e Y_{dk} risultano essere molto simili. Le reti più comuni di questo tipo, che sono anche le più famose, sono quelle multistrato, le quali verranno illustrate fra poco. Le figure di questo capitolo ed anche di quello successivo derivano da un tutorial e da un corso sulle reti neurali convoluzionali entrambi dell'università di Standford ([14] e [15]).

Nell'apprendimento non supervisionato, il cui schema generale è presente nella figura 2.2, la rete aggiusta i weight in modo autonomo, senza avere alcuna indicazione sull'output. Appartengono a questa categoria le Reti di Kohonen (SOM, Self-Organizing Maps).

Nell'apprendimento per rinforzo vi è un opportuno algoritmo che ha lo scopo di individuare un certo modus operandi, a partire da un processo d'osservazione dell'ambiente esterno; ogni azione ha un impatto sull'ambiente e quest'ultimo produce una retroazione che guida l'algoritmo stesso nel processo d'apprendimento. In pratica un agente riceve delle informazioni dall'ambiente esterno (ciò rappresenta il rinforzo), e prende una determinata decisione; in base al risultato immediato delle sue azioni l'agente può essere premiato o meno.

Nel corso di questa tesi verranno considerate solo le reti neurali che seguono il paradigma di apprendimento supervisionato.

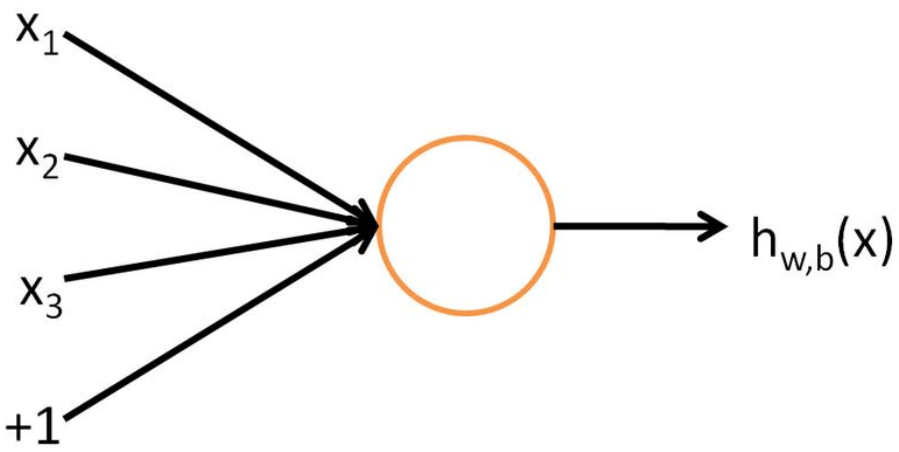


Figura 2.3: Un neurone artificiale

2.1.2 Un neurone artificiale

La più semplice rete neurale che può esistere è quella costituita da un solo neurone e viene mostrata nella figura 2.3.

Considerando la figura 2.3 un neurone può essere interpretato come un'unità computazionale, la quale prende in input gli ingressi x_1, x_2, x_3 e produce come output $h_{W,b}(x)$, detta **attivazione** del neurone. Si può notare anche un ulteriore input, il quale vale costantemente 1; il suo ruolo verrà spiegato fra poco. In realtà le cose sono leggermente più complesse, come mostrato nella figura 2.4. Un singolo neurone riceve in ingresso un valore numerico (la somma pesata di diversi input), ed esso può attivarsi, elaborando il valore ricevuto e calcolandosi la sua attivazione tramite una specifica funzione, oppure può rimanere inattivo a seconda che venga superata o meno la sua soglia di attivazione. Pertanto, il neurone sarà caratterizzato da una funzione di attivazione e da una soglia di attivazione. Data una determinata funzione di attivazione in certi casi potrebbe essere difficile se non impossibile ricavare con essa alcuni valori di output: potrebbero non esistere delle combinazioni di valori di input e di weight per quella funzione che producano un potenziale output voluto. E' necessario dunque l'utilizzo di un ulteriore coefficiente b noto come **bias**, il cui scopo è quello di permettere la traslazione della funzione di attivazione del neurone in modo da poter ottenere con certi input e weight tutti i potenziali output voluti. L'input costante $+1$ visto in figura 2.3 ha esattamente questo scopo.

Di norma si adottano diverse funzioni di attivazione, a seconda del ruolo che il neurone e la rete neurale sono destinati a svolgere. Alcune funzioni di attivazione più comuni sono:

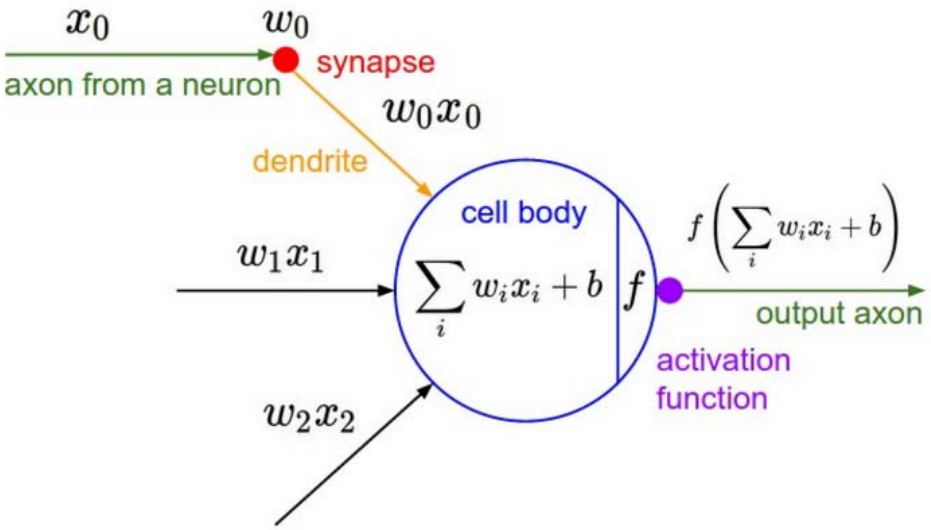


Figura 2.4: Un neurone artificiale nel dettaglio

- la sigmoide:
$$f(x) = \frac{1}{1 + e^{-x}}$$
- il gradino:
dato a $\in \mathbb{R}$
$$f(x) = \begin{cases} 0 & x \leq a \\ 1 & x > a \end{cases}$$
- la tangente iperbolica:
$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} = tanh(x)$$

Solitamente è vantaggioso considerare, come funzione di attivazione, la sigmoide oppure la tangente iperbolica, in quanto questo semplifica in maniera significativa l'algoritmo di backpropagation, algoritmo che verrà illustrato successivamente e mediante il quale la rete apprende dall'insieme di dati, il training set, che le viene fornito.

2.1.3 Tipologie comuni di reti neurali

Si possono distinguere due tipi fondamentali di reti neurali: le **reti neurali feedforward** e **le reti neurali ricorrenti (RNN)** (dette anche **reti neurali feedback**). Le prime hanno come proprietà il fatto che le connessioni

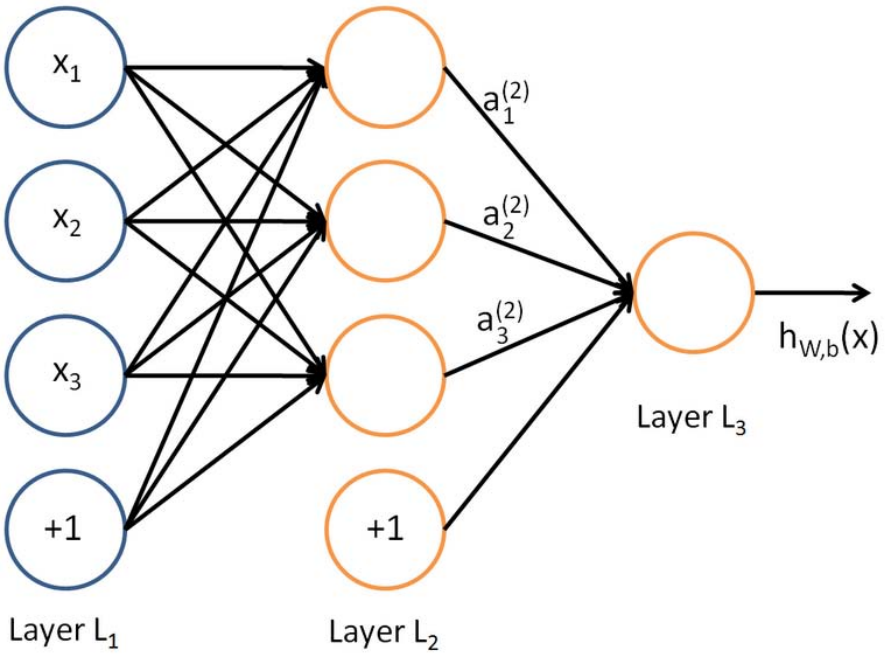


Figura 2.5: Rete stratificata con 3 layer

fra i loro neuroni non formano mai un ciclo e quindi le informazioni al loro interno viaggiano in una sola direzione.

Al contrario, nelle reti neurali ricorrenti le connessioni fra i neuroni formano un ciclo diretto, avente cioè una direzione; questo crea un vero e proprio stato interno della rete, il quale permette ad essa di possedere un comportamento dinamico nel corso del tempo. Le RNN possono utilizzare la loro memoria interna per processare vari tipi di input.

E' possibile inoltre distinguere tra reti completamente connesse (**fully connected**), in cui ogni neurone è connesso a tutti gli altri, e **reti stratificate** in cui i neuroni sono organizzati in strati, detti **layer**. Nelle reti stratificate tutti i neuroni di un layer sono connessi con tutti i neuroni del layer successivo, nel verso che va dall'ingresso all'uscita (non ci sono cicli, sono reti feedforward). Non esistono connessioni né tra neuroni di uno stesso layer né tra neuroni di layer non adiacenti. Una rete stratificata con tre layer è mostrata nella figura 2.5.

Il layer più a sinistra nella figura 2.5, con i neuroni denotati da cerchi blu, viene in generale definito **input layer** mentre il layer più a destra, con un unico neurone arancione, costituisce invece l'**output layer**. Infine il layer centrale viene detto **hidden layer** per via del fatto che i suoi valori non hanno nessun collegamento con il training set. In questo esempio vi è un solo

hidden layer, ma potenzialmente possono essere presenti molte più unità di hidden layer.

Si noti inoltre la presenza di due neuroni bias, uno per ciascun layer (output layer escluso), i quali non hanno nessun input, hanno un valore costante +1 e, come da prassi nelle reti stratificate, sono connessi solamente a tutti i neuroni del layer successivo.

Per queste reti è stato dimostrato che la formazione di regioni decisionali complesse è possibile solo se la funzione di uscita dei neuroni è non lineare.

2.2 Allenamento di una rete neurale

Uno dei metodi più noti ed efficaci per il training di reti neurali è il cosiddetto algoritmo di retropropagazione dell'errore (error backpropagation), il quale modifica sistematicamente i weight delle connessioni tra i neuroni, in modo tale che la risposta della rete si avvicini sempre di più a quella desiderata. L'addestramento di una rete neurale di questo tipo avviene in due diversi stadi: **forward propagation** e **backward propagation**. Sostanzialmente nella prima fase vengono calcolate tutte le attivazioni dei neuroni della rete, partendo dal primo e procedendo fino all'ultimo layer. Durante questa fase i valori dei weight sinaptici sono tutti fissati; alla prima iterazione si avranno dei valori di default per essi. Nella seconda fase la risposta della rete, o meglio l'uscita reale, viene confrontata con l'uscita desiderata ottenendo così l'errore della rete. L'errore calcolato viene propagato nella direzione inversa rispetto a quella delle connessioni sinaptiche, ovvero in senso opposto alla prima fase. Al termine della seconda fase, sulla base degli errori appena calcolati, i weight vengono modificati in modo da minimizzare la differenza tra l'uscita attuale e l'uscita desiderata. L'intero procedimento viene poi iterato, ripartendo con una forward propagation. Le prossime sezioni illustreranno più nel dettaglio le due fasi ed il conseguente aggiornamento dei parametri.

2.2.1 Forward Propagation

Considerando ancora la rete di figura 2.5, si vedrà ora un semplice esempio di forward propagation. Sia n_l il numero di layer della rete, nel nostro caso abbiamo n_l=3, e si etichetti ciascun layer l come L_l in modo tale che L_1 sia l'input layer ed L_{n_l} sia l'output layer. La rete considerata ha dunque i parametri (W,b)=($W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$); si considerino inoltre le seguenti notazioni:

- W^{l}_{ij} è il weight associato alla connessione fra il neurone j nel layer l ed il neurone i nel layer $l+1$ (notare l'ordine degli indici), per $1 \leq i \leq 3$, $1 \leq j \leq 3$, $1 \leq l \leq 3$.
- $W^{(l)}$ rappresenta l'intera matrice costituita dai weight fra i neuroni del layer l ed il successivo layer *l+1*
- a^{l}_{i} costituisce l'ouput, cioè l'attivazione, del neurone i nel layer l. Risulta ovvio nel nostro caso che $a^{l}_{i}=x_i$, per $1 \leq i \leq 3$, $1 \leq l \leq 3$, ovvero i nostri input della rete
- b^{l}_{i} è il bias associato al neurone i nel layer $l+1$, per $1 \leq i \leq 3$, $1 \leq l \leq 3$
- s_l corrisponde al numero di neuroni nel layer l, senza contare i neuroni bias, per $1 \leq i \leq 3$. Questa quantità verrà utilizzata successivamente.

Nell'esempio si ha $W^{(1)} \in \mathbb{R}^{3x3}$ e $W^{(2)} \in \mathbb{R}^{1x3}$. Si noti il fatto che i neuroni bias non hanno nessun input, dato che in ogni caso il loro output è sempre $+1$. L'output della rete in questione può essere ora espresso come mostrato nelle equazioni (2.1), dove la funzione $f(.)$ indica sempre la funzione di attivazione scelta per i neuroni.

$$\begin{aligned} a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\ a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\ a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\ h_{W,b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}) \end{aligned} \tag{2.1}$$

In particolare si può notare il fatto che, esattamente come accennato nella sezione 2.1.2, risulta $h_{W,b}(x) = f(W^{T}\vec{x}) = f(\sum_{i=1}^{3} W_i x_i + b)$, ovvero si ha una somma pesata (prodotto membro a membro fra ciascuna riga della matrice W ed il vettore $\vec{x}$ di input) degli input, includendo anche il bias per i motivi prima citati. Complessivamente questo processo viene detto **forward propagation** o **forward pass**.

2.2.2 Backward propagation

Tipicamente, la backward propagation utilizza un algoritmo d'apprendimento supervisionato, spesso detto **backpropagation algorithm**. Come già accennato, esso permette di modificare i weight delle connessioni con lo scopo di minimizzare il valore una certa funzione di costo C. Considerando un

singolo campione $(x^{(i)}, y^{(i)})$ del training set, utilizzando la stessa notazione vista finora, la funzione di costo rispetto ad un singolo esemplare del training set si può scrivere nel seguente modo:

$$C(W, b; x^{(i)}, y^{(i)}) = \frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \tag{2.2}$$

dove $\mathrm{h_{W,b}(x^{(i)})}$ è l'output restituito dalla rete, $\mathrm{x^{(i)}}$ è l'input e $\mathrm{y^{(i)}}$ è l'output che noi desideriamo e che fa parte del training set. In generale però, il training set è un insieme di m coppie di valori $(\mathrm{x^{(i)}},\mathrm{y^{(i)}})$, con i=1,...,m. La funzione di costo completa diventa di conseguenza:

$$\begin{aligned} C(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} C(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} (W_{ji}^{(l)})^2 = \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} (W_{ji}^{(l)})^2 \end{aligned} \tag{2.3}$$

Il primo termine della formula (2.3) è la media della somma dei quadrati degli errori; per errore si intende appunto la differenza fra uscita reale e uscita desiderata. Il secondo termine è un termine di regolarizzazione e viene detto **weight decay**: esso tende a diminuire l'ammontare dei weight ed aiuta a prevenire situazioni di overfitting. Da notare il fatto che il weight decay non viene applicato ai termini di bias, in quanto ciò solitamente non comporta grandi cambiamenti alla rete. Il **parametro di weight decay** λ controlla la relativa importanza dei due termini.

Prima di allenare una rete neurale, è bene inizializzare ciascun parametro $W_{ij}^{(l)}$ ad un piccolo valore casuale vicino allo 0 (non tutti a 0 altrimenti si rischierebbe che tutti i neuroni ottengano la medesima attivazione) e successivamente scegliere un algoritmo di ottimizzazione per minimizzare la funzione (2.3). Solitamente si utilizza l'algoritmo della discesa del gradiente (**gradient-descent**).

L'operazione di backpropagation, o meglio la backward propagation, comincia ovviamente dall'ultimo layer della rete, (l'output layer) ed, esattamente come il nome suggerisce, si propaga all'indietro fino al primo layer (l'input layer), calcolando per ciascun neurone di ogni layer (input layer escluso) un termine di errore che serve per dare una misura di quanto quel neurone viene ritenuto responsabile di un potenziale errore nell'output della rete. La backward propagation viene eseguita solamente dopo una forward propagation, vista in precedenza, nella quale vengono calcolati tutti i parametri W e b.

2.2.3 Aggiornamento dei weight

Utilizzando i termini di errore calcolati nella backward propagation si possono calcolare le derivate parziali, cioè i gradienti, rispetto a W e b della funzione di costo C. Tramite queste derivate, tutti i weight possono infine essere aggiornati.

Una singola iterazione del gradient-descend aggiorna i parametri W e b come mostrato nelle formule (2.4).

$$\begin{aligned} W_{ij}^{l} &= W_{ij}^{l} - \alpha \frac{\partial}{\partial W_{ij}^{l}} C(W, b) \\ b_{i}^{(l)} &= b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} C(W, b) \end{aligned} \tag{2.4}$$

dove C(W,b) è data dalla formula (2.3) vista in precedenza, mentre α è un parametro noto come **learning rate** e, come si può intuire dal nome, serve per specificare il grado di apprendimento, o meglio di aggiornamento, rispetto ai parametri W e b. Risulta inoltre ovvio a questo punto che la funzione di costo deve essere differenziabile.

L'intero processo di forward e backward propagation con aggiornamento successivo dei weight viene poi iterato, migliorando in questo modo di volta in volta i parametri W e b, con l'obbiettivo di minimizzare la funzione di costo.

Capitolo 3

Le reti neurali convoluzionali

Le reti neurali convoluzionali (CNN) sono di fatto delle reti neurali artificiali. Esattamente come queste ultime infatti, anche le reti neurali convoluzionali sono costituite da neuroni collegati fra loro tramite dei rami pesati (weight); i parametri allenabili delle reti sono sempre quindi i weight ed i bias allenabili. Tutto quanto detto in precedenza sull'allenamento di una rete neurale, cioè forward/backward propagation e aggiornamento dei weight, vale anche in questo contesto; inoltre un'intera rete neurale convoluzionale utilizza sempre una singola funzione di costo differenziabile. Tuttavia le reti neurali convoluzionali fanno la specifica assunzione che il loro input abbia una precisa struttura di dati, come ad esempio un'immagine nel nostro caso, e ciò permette ad esse di assumere delle specifiche proprietà nella loro architettura al fine di elaborare al meglio tali dati. Ad esempio di poter effettuare delle forward propagation più efficienti in modo da ridurre l'ammontare di parametri della rete.

3.1 Differenze principali fra CNN e ANN

Le normali reti neurali stratificate con un'architettura fully connected, dove cioè ogni neurone di ciascun layer è collegato a tutti i neuroni del layer precedente (neuroni bias esclusi), in generale non scalano bene con l'aumentare delle dimensioni delle immagini. Se prendiamo ad esempio un noto dataset presente in rete chiamato CIFAR-10, dove 10 indica il numero di categorie o classi presenti, ciascuna immagine ha dimensioni 32 x 32 x 3 (larghezza 32, altezza 32 e 3 canali del colore). Dunque ciascun neurone completamente connesso nel primo hidden layer avrebbe 32*32*3=3,072 weight. Questa quantità sarebbe ancora gestibile, ma se invece prendiamo un'immagine 200 x 200 x 3, che non è comunque una dimensione così grande, lo stesso neuro-

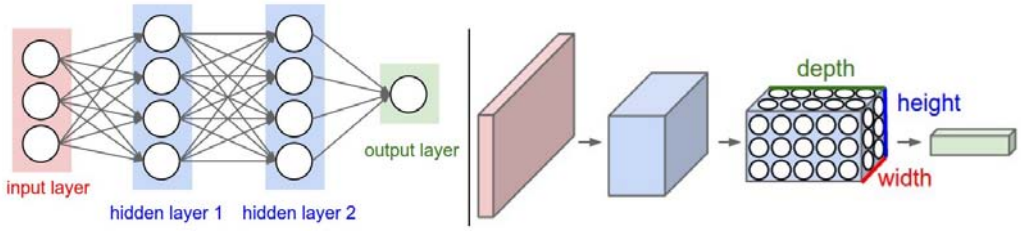


Figura 3.1: A sinistra una ANN fully connected. A destra una CNN.

ne di prima avrebbe 200*200*3=120,000 weight. Questo solo per un singolo neurone, ma considerando poi l'intera rete la cosa diventerebbe di certo ingestibile. A differenza delle normali reti neurali, i layer di quelle convoluzionali hanno i neuroni organizzati in tre dimensioni: larghezza, altezza e profondità. Ad esempio un'immagine di input di CIFAR-10 costituisce un volume di input di dimensioni 32 x 32 x 3. Inoltre, come si vedrà più nel dettaglio successivamente, i neuroni di un layer sono connessi solo ad una piccola regione del layer precedente, invece che a tutti i neuroni come in un'architettura fully connected. Questa è forse la principale caratteristica che contraddistingue una CNN da una normale rete neurale stratificata con un'architettura fully connected.

Per CIFAR-10 l'output layer avrà dimensioni 1 x 1 x 10 perchè alla fine della rete l'immagine di partenza verrà ridotta ad un vettore di score per le 10 classi presenti nel dataset, organizzato lungo la dimensione della profondità.

La figura 3.1 dovrebbe chiarire graficamente quanto detto finora.

Riassumendo dunque: ciascun layer di una rete neurale convoluzionale trasforma un volume 3D di input in un volume 3D di output; quest'ultimo costituisce l'insieme delle attivazioni dei neuroni di tale layer, tramite una determinata funzione di attivazione differenziabile.

3.2 I principali tipi di layer di una CNN

In una rete neurale convoluzionale esistono diversi tipi di layer, ciascuno avente una propria specifica funzione. Alcuni di questi hanno dei parametri allenabili (weight e bias) mentre altri layer implementano semplicemente una funzione fissata. Esistono svariate tipologie di layer, ma i più utilizzati ed indispensabili sono quelli che verranno affrontati fra poco. Altri tipi di layer più specifici verranno presentati nel capitolo successivo quando verrà introdotto il framework scelto ed utilizzato per le reti convoluzionali.

3.2.1 Convolutional layer

Come si può intuire questo è il principale tipo di layer: l'utilizzo di uno o più di questi layer in una rete neurale convoluzionale è indispensabile. I parametri di un convolutional layer in pratica riguardano un insieme di filtri allenabili. Ciascun filtro è spazialmente piccolo, lungo le dimensioni di larghezza ed altezza, ma si estende per l'intera profondità del volume di input a cui viene applicato. Durante la forward propagation si trasla, o più precisamente si convolve, ciascun filtro lungo la larghezza e l'altezza del volume di input, producendo una activation map (o feature map) bidimensionale per quel filtro. Man mano che si sposta il filtro lungo l'area dell'input, si effettua un'operazione di prodotto membro a membro, cioè un prodotto scalare, fra i valori del filtro e quelli della porzione di input al quale è applicato. Intuitivamente, la rete avrà come obbiettivo quello di apprendere dei filtri che si attivano in presenza di un qualche specifico tipo di feature in una determinata regione spaziale dell'input. L'accodamento di tutte queste activation map, per tutti i filtri, lungo la dimensione della profondità forma il volume di output di un layer convoluzionale. Ciascun elemento di questo volume può essere interpretato come l'output di un neurone che osserva solo un piccola regione dell'input e che condivide i suoi parametri con gli altri neuroni nella stessa activation map, dato che questi valori provengono tutti dall'applicazione dello stesso filtro.

Attualmente le cose potrebbero non apparire ancora del tutto chiare. Vediamo dunque più dettagliatamente i concetti appena espressi, suddividendoli per meglio comprendere l'intero processo.

Proprietà di connettività locale

Era già stato menzionato in precedenza il fatto che, avendo a disposizione degli input di considerevoli dimensioni come appunto delle immagini, connettere ciascun neurone di un layer con tutti i neuroni del layer precedente (o del volume di input) nella pratica non è conveniente. Infatti qui ciascun neurone è connesso solo ad una piccola regione locale del volume di input. L'estensione spaziale (sempre larghezza e altezza) di questa regione è un parametro del layer convoluzionale e viene detto **receptive field** del neurone. E' bene ricordare quanto già detto in precedenza: l'estensione lungo l'asse della profondità delle regione considerata è sempre uguale alla profondità del volume di input. Ciò significa che rispetto alla profondità non si esclude nulla dell'input per ciascuna regione locale, limitata invece in larghezza ed altezza. Un semplice esempio può chiarire questi concetti: prendiamo la solita immagine di CIFAR-10, quindi un volume 32 x 32 x 3; se il receptive field è

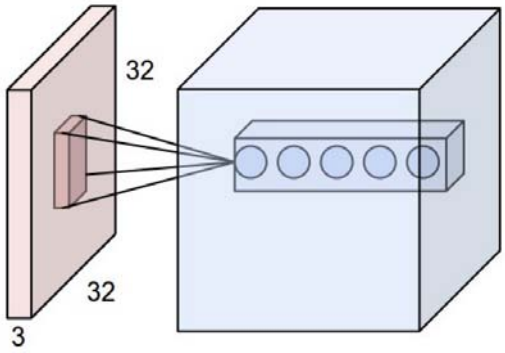


Figura 3.2: Layer convoluzionale applicato ad un'immagine di CIFAR-10

di dimensioni 5 x 5, allora ciascun neurone del layer convoluzionale avrà dei weight associati ad una regione locale 5 x 5 x 3 del volume di input, per un totale di 5*5*3=75 weight, numero decisamente inferiore rispetto a quanto visto nella sezione precedente (3072 weight) per un neurone in un'architettura fully connected. Nell'esempio di figura 3.2 si può osservare quanto appena detto; si può intuire inoltre anche un'altra cosa dalla figura: il fatto che un certo numero di neuroni lungo l'intera profondità del layer convoluzionale osservano tutti la stessa regione dell'input; si noti che il fatto che ci siano 5 neuroni nella figura non c'entra nulla con le dimensioni del receptive field.

Organizzazione dello spazio

Fin qui si è parlato della connettività di ciascun neurone di un layer convoluzionale rispetto al volume di input. In questa sottosezione si parlerà invece della quantità di neuroni presenti nel volume di output di un layer convoluzionale e di come essi sono organizzati. In questo caso vi sono 3 parametri:

- il parametro di **depth** del volume di output è quello che sostanzialmente controlla il numero di neuroni nel layer convoluzionale che sono connessi alla stessa regione locale dell'input. In particolare, un insieme di neuroni di questo tipo viene detto, per ovvi motivi, **depth column**. Il raggruppamento di neuroni osservato prima nella figura 3.2, i quali osservano tutti la stessa regione locale dell'input, costituisce una depth column.

- il parametro di **stride** serve per specificare in che maniera allocare le depth column in tutto lo spazio, cioè larghezza e altezza, dell'input. Può essere visto anche come il passo con cui spostare il receptive field

dei filtri lungo lo spazio dell'input. Ad esempio se si ha uno stride pari a 1, ciò significa che ci sarà una depth column di neuroni ogni 1 unità spaziale di distanza dalla regione locale di applicazione di un'altra depth column. Questo potrebbe portare ad un intenso overlapping delle regioni ed anche ad un incremento notevole del volume di output. L'utilizzo invece di uno stride maggiore può portare ad una riduzione dell'overlapping fra le regioni e dunque a dei volumi spazialmente più piccoli. Tutto questo dipende sempre ovviamente dalle dimensioni di ciascuna regione locale, ovvero del receptive field.

- l'ultimo parametro è lo **zero-padding**. Spesso può essere utile effettuare un'operazione di padding, con degli zeri in questo caso, spazialmente lungo i bordi del volume di input. Il parametro in questione stabilisce la taglia del padding: ad esempio con uno zero-padding pari a 0 vuol dire che in pratica non si ha nessun padding; con uno zero-padding pari a 1 significa che tutto il mio volume di input avrà, per ciascuna dimensione, un bordo di grandezza 1 di zeri e così via. Il parametro di zero-padding permette di controllare anche la taglia del nostro volume di output. In molti casi non viene utilizzato, cioè viene settato a 0, mentre in molti altri casi viene utilizzato per fare in modo che i valori assegnati ai vari parametri che specificano le dimensioni dei volumi dei vari layer lungo la rete siano risultino validi per le operazioni che si devono effettuare. La sottosezione successiva chiarirà quest'ultimo caso.

Utilità dell'operazione di zero-padding

Ormai dovrebbe essere abbastanza chiaro il fatto che i parametri di un layer convoluzionale devono essere settati molto accuratamente. Ad esempio bisogna stare attenti al fatto che gli spostamenti dei receptive field vadano a coprire l'intera area spaziale dell'input, al fatto che i receptive field non si sovrappongano troppo, o peggio al fatto non si possa effettuare l'intera convoluzione rispetto alle dimensioni dell'input in quanto le dimensioni del receptive field e lo stride scelti non permettono il normale svolgimento del processo finora spiegato. Una formula molto spesso utilizzata, che lega i valori di dimensioni dell'input, del receptive field e dello stride è $\frac{W-F+2P}{S} + 1$, dove:

- W è la larghezza dell'input,
- F è la dimensione di un lato del receptive field; dato che la sua area è sempre un quadrato, la si calcola di conseguenza,

- P è il parametro di zero-padding,
- S è il parametro di stride.

Tale formula dà come risultato il valore della larghezza, in questo caso, del nostro volume di output; finora si era parlato solo di profondità del volume di output, che è data dal parametro di depth come visto in precedenza. Risulta ovvio che se il valore della larghezza non è intero allora i parametri vanno aggiustati. Analoga cosa per quanto riguarda l'altezza del volume di output (basta sostituire nella formula W con H, dove H è l'altezza dell'input). Da questo fatto si intuisce un'altra cosa: le immagini che vengono date come input ad una rete neurale sono tutte spazialmente quadrate; ciò semplifica di molto i conti e permette di non dover ogni volta pensare ad una formula diversa per l'elaborazione di un'immagine (si avrebbero inoltre problemi per stabilire l'area dei receptive field, i quali potrebbero non essere più quadrati anche loro per poter coprire tutta l'area di input); inoltre solitamente si cerca di evitare di avere in ingresso immagini troppo grandi per ovvi motivi. Infatti normalmente dopo aver preparato un dataset, dove potenzialmente ci possono essere immagini di tutte le dimensioni, si procede ad una semplice operazione di resize di ciascuna immagine; quest'ultimo processo è ovviamente esterno, fatto cioè prima di avviare l'allenamento della rete. Una volta fatto questo, le immagini sono pronte per essere date in pasto alla rete.

Sempre nella medesima formula, l'unico parametro "libero" diciamo è proprio quello dello zero-padding: esso in certi casi può aiutare a far tornare i conti nella formula vista in modo che il risultato sia un numero intero. In generale, settare lo zero-padding con la formula $P = \frac{F-1}{2}$ (sempre se F è dispari, altrimenti si toglie il -1) quando si ha uno stride S=1 assicura che il volume di input ed il volume di output avranno la stessa taglia spaziale, cioè la stessa larghezza e altezza; la profondità dipende dal parametro di depth.

Si vedrà ora un esempio che riprende tutto quanto detto finora. Prendendo in esame una delle tante immagini che riceve in input una delle più famose CNN, chiamata AlexNet [16], abbiamo in questo caso un volume 227 x 227 x 3. Nel primo layer, che è un layer convoluzionale, si hanno i parametri F=11, S=4, P=0. Dato che $\frac{227-11}{4} + 1 = 55$ e che il parametro di depth, per comodità chiamato K, è pari a 96, il volume di output di tale layer convoluzionale avrà dimensioni 55 x 55 x 96. Ciascuno dei 55*55*96 neuroni in questo volume è connesso ad un receptive field 11 x 11 x 3 dell'area spaziale del volume di input. Inoltre tutti i 96 neuroni in ciascuna depth column sono connessi alla medesima regione 11 x 11 x 3 dell'input, ma ovviamente ciascuno possiede dei weight diversi.

Condivisione dei parametri

Prendendo ancora in esame l'ultimo esempio appena visto sopra, solo in un singolo convolutional layer si hanno 55*55*96=290,400 neuroni, e ciascuno di questi ha 11*11*3=363 weight ed 1 bias, per un totale di 364 parametri. Complessivamente si hanno dunque (290,400)*364=105,705,600 parametri. Chiaramente questo numero, relativo ad un singolo layer, è enorme. Per ridurre un così alto numero di parametri si fa una ragionevole assunzione: se una determinata feature è utile in una posizione (x_1, y_1) allora essa risulta utile anche in un'altra posizione (x_2, y_2). In pratica, chiamando una singola "sezione" bidimensionale lungo l'asse di profondità del volume di output con **depth slice** (ad esempio un volume 55 x 55 x 96 ha 96 depth slice, ciascuna ovviamente di dimensioni 55 x 55), si farà in modo che tutti i neuroni di ciascuna depth slice (attenzione a non confondere con depth column) utilizzino gli stessi weight e bias. Con questo schema in mente, lo stesso layer convoluzionale di prima ha ora solo 96 insiemi unici di weight (un insieme per ciascuna depth slice) e 96 bias, per un totale di 96*11*11*3=34848 weight unici e un numero totale di parametri, aggiungendo i 96 bias, pari a 34944. I 55*55 neuroni di ciascun depth slice utilizzano dunque gli stessi parametri. Nello specifico, durante la fase di backward propagation ogni neurone calcolerà il gradiente dei suoi weight, ma questi gradienti saranno aggiunti attraverso ciascun depth slice ed aggiorneranno solo un singolo insieme di weight per ciascun depth slice. Da notare il fatto che se tutti i neuroni in una singola depth slice utilizzano lo stesso vettore di weight, allora la forward propagation del layer convoluzionale può essere calcolata in ciascun depth slice come una convoluzione dei weight di un neurone con il volume di input; da questo fatto deriva il nome del layer in questione. Per via di questo, di norma ci si riferisce agli insiemi di weight con il termine filtri o, più comunemente, **kernel**, e con ognuno di questi si effettua un'operazione di convoluzione con l'input. Il risultato di questa convoluzione è una activation map (seguendo il solito esempio, di taglia 55 x 55), e tutte le activation map per ciascun filtro diverso vengono raggruppate assieme lungo la dimensione della profondità formando il volume di output (secondo l'esempio di taglia 55 x 55 x 96). In figura 3.3 a titolo di esempio vengono mostrati i kernel, ciascuno dei quali rappresenta un insieme di weight appresi con l'allenamento, per il primo layer convoluzionale della rete AlexNet [16].

Ciascuno dei 96 kernel mostrati, come già illustrato, ha dimensioni 11 x 11 x 3 e ciascuno di essi è condiviso dai 55*55 neuroni di una depth slice. L'assunzione fatta con la tecnica della condivisione dei parametri è relativamente ragionevole: se l'individuazione, ad esempio, di una linea orizzontale è importante in una certa locazione dell'immagine, dovrebbe teoricamente

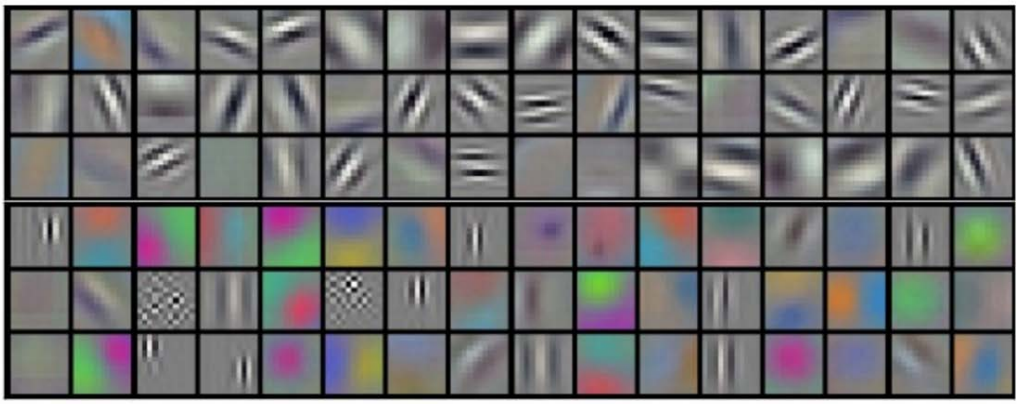


Figura 3.3: Esempio dei 96 kernel di un layer convoluzionale di AlexNet

essere importante anche in una qualsiasi altra locazione dell'immagine. Non è dunque necessario dover imparare di nuovo ad individuare una linea orizzontale in ognuna delle diverse locazioni 55*55 del volume di output del layer convoluzionale.

Riassumendo, un layer convoluzionale:

- accetta in ingresso un volume W_1 x H_1 x D_1
- richiede il settaggio dei seguenti parametri
  - numero di kernel K (il parametro depth)
  - il lato dell'area del receptive field F
  - lo stride S
  - l'ammontare dello zero-padding P
- produce un volume di output di dimensioni W_2 x H_2 x D_2, dove
  - $W_2 = \left(\frac{W_1 - F + 2P}{S}\right) + 1$
  - $H_2 = \left(\frac{H_1 - F + 2P}{S}\right) + 1$
  - $D_2 = K$
- con la tecnica della condivisione dei parametri, si hanno $F * F * D_1$ weight per ogni kernel, per un totale di $(F * F * D_1) * K$ weight e K bias
- nel volume di output l'i-esimo depth slice (di taglia W_2 x H_2) è il risultato dell'operazione di convoluzione dell'i-esimo kernel sul volume di input con uno stride S e con la successiva aggiunta dell'i-esimo bias.

L'operazione di convoluzione

Per meglio comprendere matematicamente l'operazione di convoluzione, viene qui riportato un semplice esempio. In questo caso abbiamo un volume di input di dimensioni $W_1 = 5, H_1 = 5, D_1 = 3$ ed i parametri del layer convoluzionale sono $K = 2, F = 3, S = 2, P = 1$. In pratica si hanno 2 kernel di dimensione 3 x 3, i quali vengono applicati all'input con uno stride di 2. Dato che $\left(\frac{5-3+2}{2}\right) + 1 = 3$, il volume di output avrà dimensioni 3 x 3 x 2. La figura 3.4 mostra tutto questo ovviamente in 2D. Si ricorda che la profondità del volume di input e dei kernel è la stessa, mentre risulta invece essere uguale al numero di kernel per quanto riguarda il volume di output. Sempre nella stessa figura, viene mostrato, tramite delle celle evidenziate in ciascuna matrice, il calcolo di un singolo elemento del volume di output. Il valore 2 evidenziato nel volume di output è calcolato nel seguente modo: viene calcolata la somma dei prodotti (prodotto membro a membro) fra la prima matrice blu evidenziata del volume di input, che rappresenta il receptive field, e la prima matrice del kernel W_0 rossa ad essa collegata (si ottiene 0+3+0=3); analoga cosa viene poi fatta con gli altri due kernel applicati (si ottiene rispettivamente 0 e -2). Si sommano i 3 risultati ottenti in precedenza (3+0-2=1) e si aggiunge ad essi anche il valore del parametro di bias (1+1=2). Il processo poi è analogo per tutti gli altri elementi del volume di output; l'intera prima matrice verde di output, che rappresenta la prima depth slice, è data dall'applicazione del primo kernel W_0 ai vari receptive field, traslati di volta in volta, mentre la seconda matrice verde di output deriva dal medesimo processo però utilizzando il kernel W_1 e rappresenta la seconda depth slice del volume di output. Si noti inoltre la presenza dello zero-padding di taglia 1 nel volume di input.

Backward propagation di un layer convoluzionale

Quanto visto finora per un layer convoluzionale riguarda solo la sua funzione durante la forward propagation. Tuttavia, essendo questo tipo di layer uno di quelli che possiede dei parametri allenabili (weight dei kernel ed i bias) è previsto anche un lavoro addizionale durante la backward propagation. La backward propagation di un layer convoluzionale è comunque molto simile a ciò che è stato visto nel paragrafo 2.2.2. Se chiamiamo k_{ij} gli elementi dei kernel (solo per comodità, ma rimangono comunque i weight di questo tipo

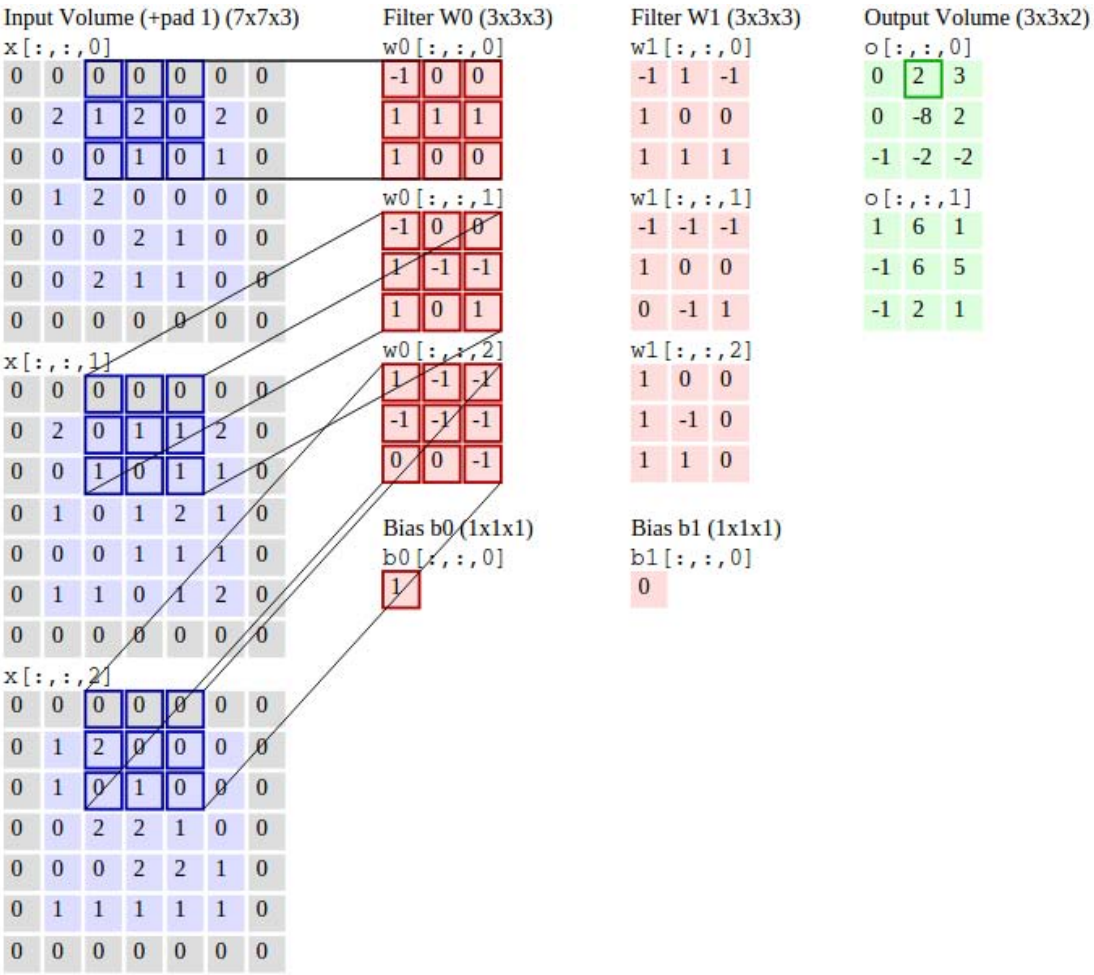


Figura 3.4: Operazione di convoluzione

di layer), la formula con cui essi si aggiornano sono le seguenti:

$$\begin{aligned} k_{ij}^{(l)} &= k_{ij}^{(l)} - \alpha \frac{\partial}{\partial k_{ij}^{(l)}} C(W,b) \\ b_{i}^{(l)} &= b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} C(W,b) \end{aligned} \tag{3.1}$$

in linea con quanto visto con le formule (2.4), dove C(W,b) è sempre la funzione di costo utilizzata in tutta la rete ed α è il learning rate impostato per la rete.

Utilizzo dei layer convoluzionali in una CNN

La capacità di una rete neurale convoluzionale può variare in base al numero di layer che essa possiede. Oltre a possedere altri tipi di layer, una CNN può avere più layer dello stesso tipo. Raramente infatti si può trovare ad esempio un solo layer convoluzionale, a meno che la rete in questione non sia estremamente semplice. Di solito una CNN possiede una serie di layer convoluzionali: i primi di questi, partendo dall'input layer ed andando verso l'output layer, servono per ottenere feature di basso livello, come ad esempio linee orizzontali o verticali, angoli, contorni vari, ecc; più si scende nella rete, andando verso l'output layer, e più le feature diventano di alto livello, ovvero esse rappresentano figure anche piuttosto complesse come dei volti, degli oggetti specifici, una scena, ecc. In sostanza dunque più layer convoluzionali possiede una rete e più feature dettagliate essa riesce ad elaborare.

3.2.2 ReLU layer

Innanzitutto ReLU è l'abbreviazione di "Rectified Linear Units". Questo tipo di layer è molto comune in una rete neurale convoluzionale e viene utilizzato più volte all'interno della stessa rete, molto spesso dopo ciascun layer convoluzionale. La sua funzione principale è quella di incrementare la proprietà di non linearità della funzione di attivazione, accennata nel paragrafo 2.1, senza andare a modificare i receptive field di un layer convoluzionale. Una funzione molto adatta a questo scopo, e per questo è quella utilizzata dai ReLU layer, è $f(x) = max(0, x)$. Si possono comunque specificare altri tipi layer con il medesimo scopo ma che utilizzano una funzione diversa, come si vedrà anche più avanti, ad esempio $f(x) = tanh(x)$, $f(x) = |tanh(x)|$, oppure la sigmoide $f(x) = (1 + e^{-x})^{-1}$. Rispetto alle funzioni di tangente iperbolica, i ReLU layer permettono di allenare una rete molto più velocemente e con prestazioni simili, altro motivo per cui sono largamente utilizzati. Come si può

intuire questi tipo di layer non ha nessun parametro impostabile, semplicemente viene eseguita una funzione fissa. Oltre a non avere nessun parametro settabile, questi layer non hanno nemmeno parametri allenabili. I layer senza parametri allenabili hanno una backward propagation più semplice: vengono retropropagati gli errori calcolati fino a quel momento, che arrivano dal layer successivo, passandoli al layer precedente a quello considerato.

3.2.3 Pooling layer

Un altro tipo di layer indispensabile in una rete neurale convoluzionale è senz'altro il pooling layer. Questi layer vengono periodicamente inseriti all'interno di una rete per ridurre la dimensione spaziale (larghezza ed altezza) delle attuali rappresentazioni, cioè dei volumi in uno specifico stadio della rete; ciò serve per ridurre il numero di parametri ed il tempo computazionale della rete, ed inoltre tiene sotto controllo l'overfitting. Un pooling layer opera su ciascun depth slice del volume di input in maniera indipendente, andando a ridimensionarlo spazialmente. Per il ridimensionamento si utilizza una semplice funzione, come ad esempio un'operazione di MAX oppure di AVE (il massimo o la media di un determinato insieme di elementi). I pooling layer hanno alcuni parametri settabili:

- il lato F dell'estensione spaziale della selezione quadrata che verrà di volta in volta considerata sull'input in ogni suo depth slice
- il parametro di stride S

Si può notare una certa somiglianza con i parametri di un layer convoluzionale: questo perchè anche qui si ha una sorta di receptive field che viene spostato di volta in volta, con un passo specificato dal parametro di stride, su ciascun depth slice del volume di input. In realtà oltre a questo, la situazione in questo caso è completamente diversa, dato che qui non vi è alcuna operazione di convoluzione. Tuttavia esattamente come in un layer convoluzionale anche qui c'è una relazione fra i parametri dell'estensione spaziale F e lo stride S (qui non c'è lo zero-padding), dato che bisogna comunque riuscire a coprire l'intera area di ciascun depth slice. La relazione è infatti molto simile a quella osservata in precedenza: considerando la larghezza W_1 di un volume di input, il volume di output di un pooling layer avrà larghezza $\left(\frac{W_1-F}{S}+1\right)$. In figura 3.5 è mostrato un esempio di pooling con F=2 ed S=2 tramite la quale la dimensione spaziale viene dimezzata, mentre in figura 3.6 viene illustrato un esempio di operazione di MAX pooling applicata ad un singolo depth slice.

Riassumendo, un pooling layer:

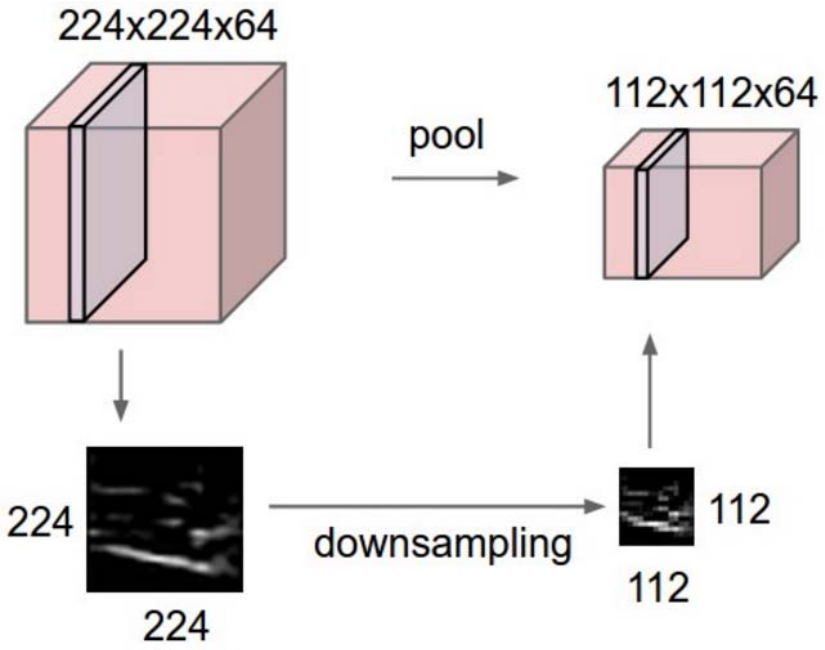


Figura 3.5: Effetti di un'operazione generale di pooling

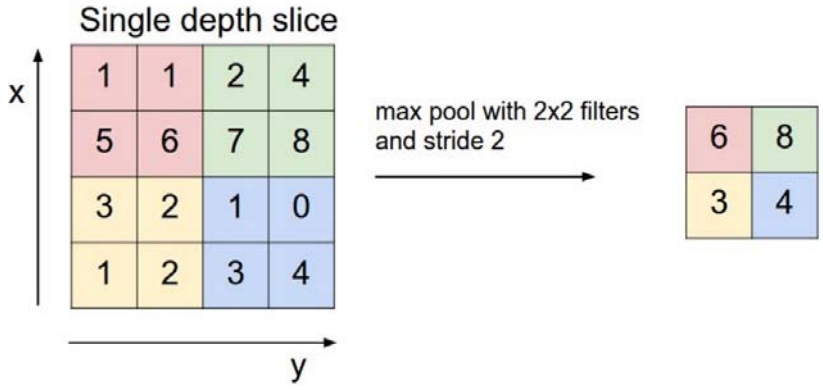


Figura 3.6: Esempio di un'operazione di MAX pooling

- accetta in ingresso un volume W_1 x H_1 x D_1
- richiede il settaggio dei seguenti parametri
  - il lato F dell'estensione spaziale della selezione quadrata
  - lo stride S
- produce un volume di output di dimensioni W_2 x H_2 x D_2, dove
  - $W_2 = \left(\frac{W_1 - F}{S}\right) + 1$
  - $H_2 = \left(\frac{H_1 - F}{S}\right) + 1$
  - $D_1 = D_2$

Si noti il fatto che un pooling layer non intacca in alcun modo la dimensione della profondità di un volume di input. Come già detto, a differenza di un layer convoluzionale non c'è lo zero-padding. Ciò non vuol dire che non si possa usare, solo che non è pratica comune utilizzarlo in questo tipo di layer in quanto in generale non apporta alcun miglioramento in questo caso. Rispetto ad un ReLU layer, un pooling layer possiede alcuni parametri settabili, ma non possiede però alcun parametro allenabile, a differenza di un layer convoluzionale: infatti, una volta settati i parametri, si procede con l'applicazione di una funzione fissa. Per questo, esattamente come un layer ReLU, un pooling layer svolge le sue mansioni solo nelle fasi di forward propagation, mentre nelle fasi di backward propagation retropropaga gli errori e basta. Potrebbe sembrare che esistano svariate tipologie di pooling, in base a come vengono settati i relativi parametri. In realtà ne esistono fondamentalmente solo 3 di veramente utili, con alcuni casi particolari oltre ad essi:

1. MAX pooling con F=2 ed S=2
2. MAX pooling con F=3 ed S=2 (detto anche **overlapping pooling**)
3. AVE pooling con F pari al lato di un intero depth slice del volume di input ed S=1 (detto anche **average global pooling**)

Innanzitutto, come si può notare, in un layer di pooling di norma il parametro F non supera mai il valore di 3, in quanto considerare aree di input troppo grandi porterebbe ad una perdita di informazioni troppo elevata e quindi ad una conseguente diminuzione delle prestazioni della rete. Il terzo dei tre casi elencati è un caso speciale che è esente da questa regola e che verrà spiegato più avanti.

Per quanto riguarda il parametro S invece, esso normalmente può valere 1 o 2: innanzitutto perchè avere un valore di stride maggiore del lato della

selezione quadrata significherebbe perdere informazioni, dato che ogni volta alcuni dati della matrice di input non verrebbero considerati; si potrebbe teoricamente impostare F=3 ed S=3, ma nella pratica questo non viene mai fatto, probabilmente perchè considerando un'area più grande su cui effettuare il pooling è meglio avere un po' di overlapping per non perdere anche qui troppe informazioni. Ci sono casi speciali comunque, come ad esempio F=3 ed S=1 ma in questo modo si rischia di avere troppo overlapping. Si può in ogni caso applicare comunque una scelta di questo genere: è il caso ad esempio di GoogLeNet [18], dove però vi sono delle condizioni particolari riguardanti l'architettura della rete che si vedranno più avanti. Il caso F=1 ed S=1 invece non ha senso perchè vorrebbe dire che sostanzialmente si lascia completamente inalterato il volume di input: tanto vale non applicare il pooling a questo punto.

Detto questo, nel primo dei tre casi, si ha sostanzialmente un MAX pooling senza overlapping fra le aree selezionate dell'input, e che complessivamente va a dimezzare le dimensioni spaziali dei vari depth slice. Le reti VGG [19] utilizzano esclusivamente questa tecnica.

Nel secondo caso invece si ha invece un MAX pooling con overlapping. Negli ultimi anni si è visto che questa tecnica pare leggermente più efficiente rispetto al primo caso, come sostiene Alex Krizhevsky et. al. [16], uno dei primi ad utilizzare questa tecnica di pooling. Anche la rete NIN [17] ed in generale GoogLeNet [18] utilizzano la stessa tecnica.

Come preannunciato, l'ultimo caso di pooling è molto particolare: l'average global pooling viene utilizzato solo ed esclusivamente come ultimo layer prima del layer di output, cioè della classificazione finale. L'area di input su cui agisce comprende un intero depth slice, in quanto per allora si dovrà avere un volume avente una profondità pari al numero di classi che si vogliono distinguere, cioè il numero di depth slice del volume deve essere uguale al numero di classi. Ad ogni modo si vedrà più dettagliatamente la funzione di questo tipo di pooling nel capitolo 5, quando verranno presentate le varie reti utilizzate. Per ora è sufficiente sapere che esiste anche questo tipo di pooling e che è stato utilizzato per la prima volta nella rete NIN [17] ed è stato successivamente utilizzato anche da GoogLeNet [18].

3.2.4 Normalization layer

Non sempre, ma a volte un'operazione di normalizzazione dei dati può aumentare le performance delle reti: ecco perchè a volte all'interno della definizione di una rete neurale convoluzionale si può trovare un layer di normalizzazione, anche se negli ultimi anni si è visto che il loro contributo è piuttosto limitato. Si ricordi comunque che l'utilizzo di questi layer non è assolutamente

necessario, in quanto nessun layer richiede di avere dei dati normalizzati. Questi layer sono stati proposti al fine di tentare di simulare alcuni schemi di inibizione del cervello umano. A tal proposito sono state proposte svariate funzioni; un layer in particolare chiamato LRN (Local Responce Normalization) viene utilizzato spesso nel framework considerato in questa tesi e lo si vedrà in dettaglio nel capitolo successivo.

Questa tipologia di layer ha alcuni parametri settabili (il loro numero e tipologia dipende dalla funzione di normalizzazione scelta) ma nessun parametro allenabile, quindi anche questi layer agiscono solamente nella forward propagation, mentre nella backward propagation diffondono l'errore all'indietro.

3.2.5 Fully Connected layer

Questo tipo di layer è esattamente uguale ad uno qualsiasi dei layer di una classica rete neurale artificiale con architettura fully connected: semplicemente in un layer Fully Connected (FC) ciascun neurone è connesso a tutti i neuroni del layer precedente, nello specifico alle loro attivazioni. Dunque l'attivazione di un neurone di un layer FC può essere calcolata come con le formule (2.1) del paragrafo 2.2.1, ovvero con il prodotto fra la matrice dei weight e la matrice di input seguito dall'aggiunta di un bias.

Questo tipo di layer, a differenza di quanto visto finora nelle reti neurali convoluzionali, non utilizza la proprietà di connettività locale: un FC layer è connesso all'intero volume di input e quindi, come si può immaginare, si avranno moltissime connessioni. L'unico parametro impostabile di questo tipo di layer è il numero di neuroni K che lo costituiscono. Ciò che fa sostanzialmente un FC layer è quello di collegare i suoi K neuroni con tutto il volume di input e di calcolare l'attivazione di ciascuno dei suoi K neuroni. Il suo output sarà infatti un singolo vettore 1 x 1 x K, contenente le attivazioni calcolate. Il fatto che dopo l'utilizzo di un singolo FC layer si passi da un volume di input, organizzato in 3 dimensioni, ad un singolo vettore di output, in una singola dimensione, fa intuire che dopo l'applicazione di un FC layer non si può più utilizzare alcun layer convoluzionale. La funzione principale dei FC layer nell'ambito delle reti neurali convoluzionali è quello di effettuare una sorta di raggruppamento delle informazioni ottenute fino a quel momento, esprimendole con un singolo numero (l'attivazione di uno dei suoi neuroni), il quale servirà nei successivi calcoli per la classificazione finale.

Solitamente si utilizza più di un layer FC in serie e l'ultimo di questi avrà il parametro K pari al numero di classi presenti nel dataset. I K valori finali saranno infine date in pasto all'output layer, il quale, tramite una specifica

funzione probabilistica, effettuerà la classificazione. In verità negli FC layer ci sono altri parametri sono impostabili ma non è strettamente necessario specificarli; essi riguardano solamente la distribuzione iniziale dei valori dei weight e dei bias; se non settati, si procede con un'inizializzazione di default.

Come si può intuire, in questi layer vi sono dei parametri allenabili (i weight ed i bias) e quindi nella backward propagation tramite il solito backpropagation algorithm e con il metodo del gradient-descent (formule (2.4)) vengono aggiornati i parametri in questione.

3.3 Struttura generale di una CNN

Si può ora avere un'idea generale di come sia strutturata una semplice rete neurale convoluzionale: partendo da delle immagini come input date in ingresso ad un input layer, vi saranno una certa serie di layer convoluzionali, intervallati da ReLU layer e, quando sono necessari, da dei layer di normalizzazione e di pooling, ed infine vi sarà un'ultima serie di FC layer prima dell'output layer. In realtà, come si vedrà nel capitolo 5, negli ultimi studi recenti si è scoperto che i FC layer non sono così indispensabili, ma per il momento si assuma che questa sia la struttura di una normale rete neurale convoluzionale. La rete AlexNet [16], creata nel 2012, e le reti VGG [19] hanno esattamente questa struttura; la prima è diventata famosa perchè è stata una delle prime reti neurali convoluzionali ad avere prestazioni molto buone sui dataset più avanzati attualmente esistenti. Sempre nel capitolo 5 si vedrà nello specifico la struttura di questa rete ed anche di molte altre.

Capitolo 4

Il framework Caffe

Esistono vari framework che permettono di creare, allenare e testare delle reti neurali convoluzionali come ad esempio Torch, Theano e Caffe. Nella tesi si è scelto di utilizzare Caffe per motivi che verranno fra poco spiegati. In questo capitolo verrà dunque presentato tale framework dandone una panoramica generale e illustrando alcuni layer più specifici utilizzati da Caffe che non sono stati presentati nel capitolo 3. Sempre nel corso del capitolo, verranno descritte alcune procedure effettuate con il framework utili per lo scopo del progetto finale, come ad esempio come creare ed utilizzare su Caffe un proprio dataset, oppure come effettuare un'operazione di fine-tuning. Buona parte di queste operazioni non sono ben definite sul sito di Caffe e per questo si è deciso di esprimere la relativa esperienza direttamente affontata, sperando che possa aiutare chiunque debba cimentarsi nell'utilizzare lo stesso framework. Caffe permette di affrontare molteplici problemi; in questa tesi tuttavia ci si concentrerà solo sull'utilizzo di Caffe legato alle reti neurali convoluzionali per problemi di object recognition.

4.1 Introduzione a Caffe

Caffe è un framework per il deep learning scritto in linguaggio C++: è stato sviluppato all'interno del Berkeley Vision and Learning Center (BVLC), durante il lavoro di dottorato di Yangqing Jia all'università di Berkeley. Caffe è stato rilasciato con la licenza BSD 2-Clause ed è open-source.

4.1.1 Motivazioni dell'utilizzo di Caffe

Verranno ora precisati i motivi dell'utilizzo del framework Caffe. I vantaggi principali si possono raggruppare in 4 categorie:

1. **Architettura incoraggiante**. L'architettura che Caffe possiede è adatta ad inspirare innovazione, nuove applicazioni. I modelli e le ottimizzazioni, come si vedrà più avanti, sono definiti tramite accurate configurazioni senza hard-coding, ovvero lasciando un certo grado di libertà all'utente nelle sue scelte durante le fasi di impostazione, utente che non deve per forza conoscere specificatamente come è costituito il codice sottostante. Caffe, esattamente come altri framework, permette di svolgere elaborazioni sia tramite CPU che tramite GPU; indipendentemente dal framework, per il secondo caso serve una scheda video adatta allo scopo. Lo switch fra una modalità e l'altra su Caffe è particolarmente semplice: un semplice flag booleano (CPU_ONLY) all'interno del makefile serve a specificare quale modalità usare.

2. **Codice estensibile**. All'inizio (poco dopo la sua nascita) Caffe era ovviamente abbastanza limitato riguardo alle sue possibilità di utilizzo all'interno del campo del deep learning. Tuttavia solo nel primo anno, più di 1000 sviluppatori si sono messi al lavoro per espandere ed ottimizzare Caffe, dando un contributo più che significativo. Ciò è stato possibile solo grazie alla facilità di estensione del codice di Caffe, per come è strutturato. Grazie a tutti gli sviluppatori contribuenti, attualmente il framework risulta essere uno stato dell'arte nel campo nel campo del deep learning, sia per quanto riguarda i modelli delle reti per esso disponibili che per il codice. Ad esempio è stato realizzato un codice per far funzionare in maniera ancora più efficiente Caffe tramite i driver CUDA di NVIDIA: in questo modo, oltre ad avere modelli di reti innovativi e all'avanguardia si possiede anche un'elaborazione dei dati fra le più efficienti in termini di velocità computazionale.

3. **Velocità**. L'ultimo esempio dovrebbe far capire che, grazie alla sua velocità Caffe è perfetto per effettuare esperimenti di ricerca. Con una GPU NVIDIA k40 con 12 GB di memoria, può processare più di 60 milioni di immagini al giorno. Ciò significa circa 5 ms per l'intero processamento di una singola immagine.

4. **Comunità**. Attualmente Caffe è utilizzato in moltissimi progetti di ricerca, prototipi di start-up, ed anche in applicazioni industriali di larga scala nel campo della computer vision, speech e del multimedia. Allo scopo di aiutare ad affrontare determinati problemi, sia per i meno esperti che per i veterani, Caffe dispone di gruppo su Google-Groups chiamato "Caffe users" a cui per iscriversi è necessaria solamente una propria mail personale. Chiunque può aprire una nuova discussione, specificando la propria situazione e chiedendo un possibile aiuto

a chi magari ha già avuto un'esperienza simile. In alternativa, dato che l'intero progetto di Caffe si trova attualmente su Github, ci si può confrontare anche tramite esso.

4.1.2 Utilizzo dei protocol buffer

Per utilizzare Caffe è necessario capire che cos'è un **protocol buffer**. I protocol buffer costituiscono un meccanismo flessibile, efficiente ed automatico per la serializzazione di strutture di dati. Si può vedere un protocol buffer come una sorta di file XML, ma in realtà più piccolo, veloce e semplice. E' possibile definire in che maniera strutturare i propri dati e, una volta fatto questo, si può utilizzare un apposito codice, precedentemente ideato, per scrivere e leggere nella struttura creata; per il codice è possibile utilizzare una certa varietà di linguaggi di programmazione. In particolare, si definisce la propria struttura di dati definendo dei *protocol buffer message* all'interno di un file *.proto*. Un protocol buffer message è una piccola unità logica che contiene una serie di coppie (nome-valore). Per dare un'idea, la figura 4.1 mostra il contenuto di semplice file ".proto" che definisce un message contenenti le informazioni di una persona.

Dalla figura 4.1 si nota che il formato di un message è piuttosto semplice: ciascun message possiede uno o più campi numerati unicamente ed ogni campo ha un suo nome ed un sua tipologia, ovvero può essere un intero, un float, un booleano ecc., oppure, come nell'esempio appena mostrato, anche un altro message come *PhoneNumber*. Si intuisce dunque che i message si possono anche nidificare, permettendo in questo modo di creare strutture di dati gerarchiche. Si possono inoltre specificare quali parametri sono obbligatori (*required*), quali opzionali (*optional*) e quali possono essere ripetuti più volte (*repeated*). Nell'esempio una persona potrebbe avere più di un numero di telefono). Una volta definiti i propri message si avvia il compilatore dei protocol buffer per lo specifico linguaggio di programmazione voluto, passandogli il file *.proto* in modo che si generino le classi per l'accesso ai dati con lo specifico linguaggio desiderato, ad esempio Java, C++ oppure Python. Tali classi forniscono sia un semplice accesso ad ogni campo specificato, come ad esempio *name()* e *set_name()* per quanto riguarda la lettura e la scrittura di un nome di una persona, sia metodi per serializzare i dati in output oppure che permettono operazioni di parsing. L'esecuzione di questo processo sull'esempio di figura 4.1 creerà dunque una classe *Person* con i campi specificati. Nell'esempio di figura 4.2 e 4.3 vengono mostrati degli esempi di codice, in cui si è scelto C++ come linguaggio di programmazione, che si può scrivere dopo aver creato la classe *Person* (il primo è una scrittura, il secondo è una lettura di dati).

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Figura 4.1: Esempio file .proto

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Figura 4.2: Esempio codice C++ per scrittura dati

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Figura 4.3: Esempio codice C++ per lettura dati

Si possono aggiungere nuovi campi ai formati dei propri message senza andare ad intaccare la loro compatibilità con altri script che utilizzano ancora una loro vecchia versione. I vecchi file binari semplicemente ignorano i nuovi campi durante il parsing dei parametri. Questo risulta molto utile, in quanto con Caffe capita spesso di avere qualche parametro aggiuntivo in molte situazioni. I protocol buffer costituiscono dunque per Caffe una semplice interfaccia, con cui gli utenti definiscono in maniera relativamente semplice i dati delle proprie reti e allo stesso modo posso modificare, aggiungere, rimuovere dati senza preoccuparsi di dover ricompilare Caffe ogni volta. Nella sezione successiva si vedranno alcuni esempi di definizione di semplici reti tramite l'utilizzo dei protocol buffer.

4.2 Anatomia di una rete di Caffe

Caffe definisce la struttura di una rete neurale convoluzionale utilizzando i protocol buffer, esplicitando ogni singolo layer con i suoi parametri, partendo dall'input layer ed andando fino all'output layer. Nello specifico, nell'intero processo di utilizzo della rete in cui i dati "viaggiano" all'interno di essa, sia con la forward propagation che con la backward propagation, Caffe salva, trasmette e manipola le informazioni tramite **blob**. Le blob sono gli array standard che costituiscono l'interfaccia di memoria utilizzata dal framework per organizzare i dati. Come si può intuire, un **layer** invece costituisce l'unità fondamentale dell'intero modello di una rete, anche per quanto riguarda la sua computazione. Una **rete** si ottiene dunque da una collezione di layer fra loro connessi. Una blob descrive sostanzialmente come le informazioni vengono salvate e comunicate fra un layer ed un altro, come mostrato in figura 4.4, in cui la prima blob fornisce l'input al layer mentre la seconda ne contiene l'output.

I parametri relativi a ciascun layer vengono dichiarati all'interno della definizione dei layer stessi, mentre i parametri relativi all'allenamento generale

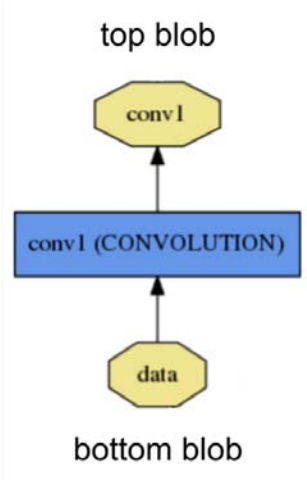


Figura 4.4: Esempio con due blob

della rete vengono definiti separatamente, in un altro protocol buffer. Ciò serve per rendere indipendente la definizione di una rete rispetto alla sua configurazione, o meglio il **solving**, e rendere inoltre più modulare il tutto. Si vedrà tutto questo nel dettaglio più avanti.

4.2.1 Blob

La struttura generale di una blob è quella di un array di dimensioni N x K x H x W, dove:

- N viene detto parametro di **batch size**. Esso rappresenta in pratica, il numero di elementi di input, cioè di immagini in questo caso, che vengono processate nella rete di volta in volta (importante per l'allenamento di una rete). Questo parametro va settato in base alla memoria che si ha a disposizione sul proprio computer (memoria della scheda video se si lavora in modalità GPU, oppure la RAM che si lavora in modalità CPU). La variazione di questo parametro non intacca le prestazioni della rete in termini di accuratezza della classificazione. Certamente un valore più basso di batch size provocherà una fase di training più lenta, perchè si processano appunto meno immagini alla volta, ma ciò è in linea col fatto che si possiede dunque un hardware più limitato.

- K rappresenta il numero di canali delle immagini o la profondità di un volume all'interno della rete.

- H è l'altezza delle immagini o di un volume all'interno della rete.

- W è la larghezza delle immagini o di un volume all'interno della rete.

Questo è il formato standard delle blob, anche se poi in realtà in certi casi assumono forme anche diverse. Ad esempio nei FC layer invece di essere in 4D sono semplicemente bidimensionali, in quanto ciò è sufficiente per rappresentare le informazioni per essi. Le dimensioni delle blob variano a seconda del tipo e della configurazione di un determinato layer. Ad esempio per un layer convoluzionale con 96 kernel, con un receptive field 11 x 11 che agiscono su un input di profondità 3, la blob risultante ha dimensioni 96 x 3 x 11 x 11. Per un layer FC avente 1000 neuroni, ciascuno dei quali è collegato ad un volume di 1024 neuroni, la blob ha dimensioni 1000 x 1024.

4.2.2 Setup, forward e backward di un layer

Un qualsiasi layer possiede tre stadi computazionali:

- una fase iniziale di **setup**, eseguita una sola volta durante la fase di inizializzazione dell'intera rete. In questa fase il layer inizializza i suoi parametri e le sue connessioni con altri layer.

- una fase di **forward** (durante la forward propagation della rete), in cui il layer, dato un input che arriva dal layer precedente, si calcola un output e lo passa al layer successivo.

- una fase di **backward** (durante la backward propagation della rete), in cui il layer retropropaga l'errore della rete, calcolando il gradiente del suo input e spedendolo al layer precedente. Nel caso il layer avesse dei parametri allenabili, esso calcola i gradienti anche dei suoi parametri e li salva internamente.

Gli ultimi due stadi sono in linea con quanto visto nei paragrafi 2.2.1, 2.2.2 e 2.2.3; quindi un layer si trova in maniera alterna in uno di questi due stadi, in base all'attuale processamento della rete. In realtà su Caffe esistono due implementazioni diverse delle funzioni per gli stadi di forward e backward di ciascun layer: uno per la modalità CPU ed uno per la modalità GPU. Se non viene implementato il codice per la modalità GPU, un layer passa in automatico al codice per la modalità CPU.

Su Caffe è inoltre possibile definire un proprio layer personalizzato. Per fare questo, bisogna modifica il file *Caffe.proto* ed aggiungere l'identificativo del nuovo layer, specificandone anche i relativi parametri; successivamente è necessario implementare il codice per la fase di forward e di backward del layer (è sufficiente farlo solo per la modalità CPU) ed infine ricompilare interamente Caffe. In realtà esistono ormai praticamente tutti i layer di cui

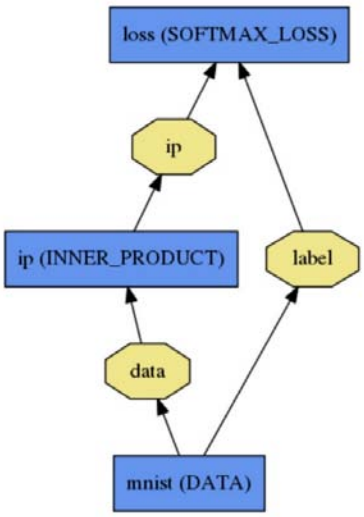
normalmente si ha bisogno, ma potrebbe tuttavia capitare qualche caso in cui si necessita di una visualizzazione alternativa o di un qualche tipo particolare di funzione.

4.2.3 Definizione di una rete con Caffe

Una rete è un grafo (per la precisione un DAG, Directed Acyclic Graph) i cui nodi sono i layer e le blob. Durante la definizione di una rete in un protocol buffer, Caffe controlla che tutte le connessioni fra i layer siano specificate correttamente, in maniera cioè da garantire una corretta forward e backward propagation. Tipicamente un rete inizia con un data layer, che costituisce l'input layer, il quale carica le immagini dalla memoria e finisce con un loss layer, che costituisce l'output layer, il quale calcola la funzione di costo, o funzione obbiettivo, utile per la classificazione e per la backward propagation futura. Entrambi i layer menzionati, verranno illustrati più accuratamente in uno dei paragrafi successivi.

Come esempio di definizione di rete si osservi la figura 4.5. A sinistra è mostrata l'architettura di un semplice classificatore di regressione logistica (uno dei modelli statistici più utilizzati negli ultimi anni in questo ambito), mentre a destra è presente la relativa definizione in un protocol buffer.

La keyword *type* specifica ovviamente la tipologia di un layer.*Softmax* serve semplicemente ad indicare il nome il tipo di funzione utilizzata dal loss layer, cioè dal classificatore; la funzione di Softmax verrà esplicitata nel paragrafo di questo capitolo relativo alla funzione di loss. *Inner_Product* invece è il termine che Caffe utilizza per indicare un normale FC layer. Si noti come nella definizione siano presenti i tre costrutti dei tre layer; in ognuno di essi le keyword *top* e *bottom* specificano intuitivamente le connessioni del grafo della rete. Alcuni layer (in questo esempio nessuno) possono avere lo stesso valore in *top* e *bottom*; questo significa che la blob di input è la stessa blob di output e che quindi la computazione avviene diciamo localmente. Questo è il classico caso dei ReLU layer, i quali non modificano mai la dimensione del volume avuto in input perchè semplicemente calcolano una funzione fissa su esso. Dunque la dimensione della blob rimane la stessa sia prima che dopo l'operazione e quindi la stessa blob può essere riutilizzata. Nel primo layer, il data layer, viene specificato il parametro di batch size, visto in precedenza. Tale parametro deve essere sempre specificato nel layer di input, che in questo caso è di tipo *Data*, ma in realtà ne esistono diversi tipi come si vedrà più avanti. Il secondo layer è un semplice FC layer con 2 soli neuroni, il quale calcola le loro attivazioni. Infine l'ultimo layer, il loss layer, calcola il valore della funzione di costo specificata, in base agli input ricevuto. E' importante osservare il fatto che nell'input layer i dati e le relative label vengano caricati



(a) Il grafo della rete

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

(b) Definizione della rete

Figura 4.5: Una semplice rete di Caffe

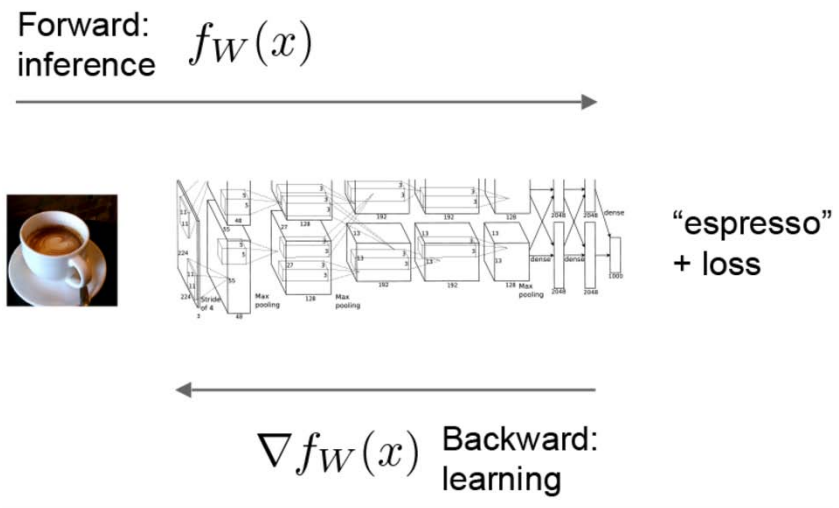


Figura 4.6: Allenamento di una rete con Caffe

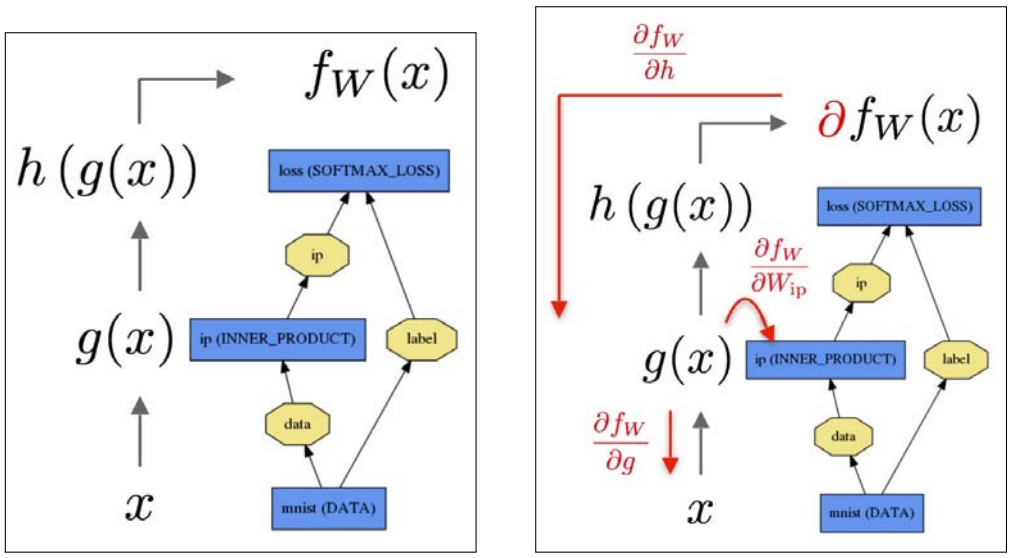
in due blob separate. La separazione avviene per come è stato costruito il dataset, altra cosa che si affronterà fra poco. La blob dei dati viene processata, mentre quella delle label no: viene semplicemente passata al loss layer finale. Una volta terminata la definizione di una rete, è necessario allenarla con uno specifico training set. Il processo di allenamento, in termini teorici, è esattamente lo stesso di una normale rete neurale con i weight ed i bias allenabili, come già visto in precedenza. In figura 4.6 viene riassunto quando già detto nel capitolo 2.1 sul processo di allenamento di una rete con il metodo di retropropagazione dell'errore, e qui ripreso come training di una rete neurale convoluzionale con Caffe.

Le figure 4.7(a) e 4.7(b) mostrano invece nello specifico le due suddette fasi, applicate alla stessa rete considerata nell'esempio precedente. Si noti come, nella backward propagation, il layer FC sia l'unico che aggiorni i propri parametri, calcolando le loro derivate, dato che è l'unico layer nella rete a possedere dei parametri allenabili.

Successivamente, in una sottosezione di questo capitolo, verranno mostrati i comandi necessari per allenare una rete.

4.2.4 I tipi di input accettati da Caffe

Era già stato visto, nel paragrafo 4.2.3, un esemplare di data layer, ovvero della tipologia di layer che su Caffe esegue la funzione di un normale input layer, prelevando le immagini di input. In realtà esistono più tipologie di data layer, le quali verranno ora presentate. Vale sempre il fatto che in ogni



(a) Forward propagation (b) Backward propagation

Figura 4.7: Fasi di allenamento di una rete con Caffe

data layer, qualunque sia stato scelto per la costruzione di una rete, deve essere specificato il parametro di batch size. I data layer più utilizzati su Caffe hanno le seguenti keyword (valori del parametro *type*):

- **Data**. Il più comune ed utilizzato con Caffe e già affrontato con la figura 4.5(b). Questo tipo di layer prende in input dei database di immagini, in particolare con i formati LEVELDB oppure LMDB (Lightning Memory−Mapped Database). Il layer non effettua in automatico la conversione (da dataset di immagini a database organizzato in coppie chiave-valore), la quale deve essere già stata fatta in precedenza. Caffe mette a disposizione un apposito script per questo scopo ed in seguito verrà mostrato come effettuare questa procedura.

- **MemoryData**. In questo caso i dati vengono letti direttamente da memoria, senza effettuarne una copia. E' necessario in questo caso specificare la dimensione dei dati che vengono letti di volta in volta (batch size). Per il resto la struttura è molto simile ad un layer di tipo *Data*.

- **HDF5Data**. Permette di leggere i dati in formato HDF5 (Hierarchical Data Format). Questo tipo di formato è spesso utilizzato in Matlab.

- **ImageData**. In questo caso le immagini vengono lette e copiate una ad una da memoria. Bisogna sempre specificare quante immagini leg-

```
layer {
  name: "data"
  type: "ImageData"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  image_data_param {
    source: "data/flickr_style/train.txt"
    batch_size: 50
    new_height: 256
    new_width: 256
  }
}
```

Figura 4.8: Esempio di image data layer

gere alla volta e soprattutto specificare un resize per ciascuna di queste; quest'ultimo non è obbligatorio, ma è fortemente consigliato per il corretto funzionamento della rete. Un esempio di questo tipo di layer è mostrato in figura 4.8. In questo caso si ha un esempio di data layer più complesso, in quanto è presente anche un'operazione di preprocessing di *mean subtraction* che verrà affrontata fra poco. Per ora è sufficiente osservare che con il layer in esempio vengono lette 50 immagini alla volta e che le keyword *new_height* e *new_width* specificano le nuove dimensioni, cioè il resize, di ciascuna immagine, ovvero 256 x 256. *source* specifica il percorso delle immagini su disco. In realtà è prassi comune, come in questo caso, definire un file di testo contenente i path di ciascuna immagine, specificandone uno per ogni riga del file.

Sempre nella figura 4.8 si noti il parametro *crop_size*. Questo parametro viene sempre specificato nelle reti più efficienti ed indica le dimensioni di una **crop**, ovvero di un "ritaglio" dell'immagine originale; in questo caso la crop avrebbe dimensioni 227 x 227. Spesso di una singola immagine vengono prese più crop, nello specifico una crop centrale, 4 crop ciascuna contenente un angolo dell'immagine e per tutte queste crop vengono prese anche le rispettive crop specchiate. Le versioni specchiate possono essere attivate o disattivate tramite il flag '*mirror*, che nell'esempio citato è settato a *true*. Se non viene specificato nulla sul numero di crop, Caffe di base considera solo una singola crop centrale. Nell'esempio si hanno dunque 2 crop 227 x 227, ovvero la normale centrale e la specchiata, per ogni elaborazione di una singola immagine. L'ultimo parametro, che per ora è ignoto, è *phase*, ma per

non complicare troppo le cose verrà specificato più avanti. Tutte le operazioni appena menzionate non riguardano solo un image data layer, bensì tutti i tipi di data layer visti. L'esempio visto è semplicemente un input layer di Caffe leggermente più complesso ma con parametri che vengono molto spesso utilizzati e quindi risulta utile per cominciare a capire alcune operazioni che verranno meglio introdotte più avanti.

4.2.5 Operazione di mean subtraction

Assieme al resize di un'immagine, l'operazione di mean subtraction risulta essere l'unico tipo di preprocessing che di solito si effettua con Caffe. Il fatto di non fare molte operazioni di preprocessing non è un limite, bensì un vanto: riuscire ad ottenere buoni risultati senza quasi praticamente fare un fase preliminare di operazioni che aiutino l'intero processo di classificazione è sicuramente un'ottima cosa. Un'ulteriore fase esterna di preprocessing teoricamente dovrebbe solo aiutare di più, anche se certamente rallenterebbe un po' l'intero processo. L'operazione di mean subtraction è comunemente usata per rendere più facile l'identificazione dei pixel che costituiscono l'oggetto che si vuole classificare all'interno di un'immagine, selezionandoli rispetto ai pixel dello sfondo (background subtraction). Questa operazione non è obbligatoria, ma risulta essere spesso molto utile e non appesantisce quasi per nulla il processo di allenamento di una rete, in quanto il lavoro più lungo, che dipende dal numero di immagini presenti nel dataset, viene effettuato a parte all'inizio e solo un'unica volta. Nello specifico, viene calcolata la media, pixel per pixel, di ciascuno dei 3 canali delle immagini del dataset (di norma solo del training set, in quanto le immagini del test set dovrebbero essere completamente ignote) e successivamente, all'interno dell'allenamento della rete questa volta, tali medie vengono sottratte ai pixel delle varie immagini. L'operazione risulta utile nei casi in cui ci siano background piuttosto simili: in questo caso dopo l'operazione di mean subtraction molti pixel rappresentanti il background dovrebbero avere valore 0, o perlomeno vicino ad esso, e potrebbero dunque essere esclusi. Esistono vari tool che permettono il calcolo iniziale delle medie; Caffe tuttavia mette a disposizione uno script per questo scopo chiamato *make_imagenet_mean.sh*. Imagenet è uno dei più importanti ed avanzati dataset che attualmente esistono e lo scopo iniziale di questo script era quello di calcolare le medie relative a tale dataset. Basta però modificare qualche parametro all'interno dello script per avere il calcolo delle medie su un qualsiasi dataset si voglia, ed ora verrà mostrato come fare; eventualmente si può anche fare una copia dello script, rinominare la copia dopo averla modificata ed eseguire il nuovo script, non cambia nulla. L'immagine 4.9 mostra l'intero script in questione.

```
#!/usr/bin/env sh
# Compute the mean image from the imagenet training lmdb
# N.B. this is available in data/ilsvrc12

EXAMPLE=examples/imagenet
DATA=data/ilsvrc12
TOOLS=build/tools

$TOOLS/compute_image_mean $EXAMPLE/ilsvrc12_train_lmdb \
  $DATA/imagenet_mean.binaryproto

echo "Done."
```

Figura 4.9: Calcolo delle medie per l'operazione di mean subtraction con Caffe

Come si può vedere esso risulta molto semplice, in quanto esso richiama un file binario (*compute_image_mean*) già presente nel framework di Caffe. Lo script in questione vuole sostanzialmente 3 parametri:

1. il percorso dell'eseguibile *compute_image_mean*. Si può lasciare lo stesso path se ci si trova attualmente all'interno della cartella di Caffe. In caso contrario, oppure nel caso il file fosse stato spostato, è bene inserire un path assoluto più specifico.

2. il percorso di dove sono presenti i file LMDB del training set precedentemente creati. Caffe in questo caso permette l'utilizzo dell'operazione con solo questo tipo di file, i più utilizzati ed efficienti. Per altri tipi di input bisogna crearsi un apposito programma oppure utilizzare un tool esterno.

3. il percorso di dove salvare il file di output con le medie calcolate

Il file di output ha estensione *.binaryproto* ed in pratica è un protocol buffer in forma binaria, dunque non apribile e modificabile. Questo file può essere incluso in un data layer, esattamente come si può vedere dalla figura 4.8. In alternativa, se si utilizza un tool esterno, oppure è stato creato un apposito programma, ed in sostanza si hanno a disposizione i valori voluti, è possibile specificarli direttamente come mostrato in figura 4.10.

4.2.6 La funzione di loss

Nel paragrafo 2.2.2 era stato introdotto il concetto di funzione di costo, o funzione obbiettivo, di una rete neurale, una funzione che si deve minimizzare

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: false
    crop_size: 224
    mean_value: 104
    mean_value: 117
    mean_value: 123
  }
  data_param {
    source: "/home/fabio/caffe/data/tune/lmdb/train_lmdb"
    batch_size: 32
    backend: LMDB
  }
}
```

Figura 4.10: Valori delle medie di ciascun canale all'interno del data layer

al fine di allenare la rete neurale in maniera accurata. Su Caffe questo tipo di funzione viene detta funzione di loss, ma il concetto rimane comunque lo stesso: una funzione di loss specifica l'obbiettivo dell'apprendimento della rete, mappando determinate configurazioni di parametri (i weight ed i bias della rete) con uno specifico valore scalare che in pratica specifica quanto questi parametri non sono buoni; in paratica più quest'ultimo valore è alto e più i parametri non sono buoni. Dunque anche qui l'obbiettivo è quello trovare una configurazione di weight e bias che minimizzi il valore di uscita della funzione di loss. In ogni rete di Caffe è dunque presente un loss layer, uno dei quali è già stato mostrato in figura 4.5(b). In tale figura si ha un layer di tipo *SoftmaxWithLoss*, che in pratica è un loss layer che fa uso di una delle più utilizzate funzioni di loss per le classificazioni, ovvero la funzione di Softmax, mostrata nelle formule (4.1).

$$\begin{aligned} y_i &= \frac{e^{a_i}}{\sum_{j=1}^{L} e^{a_j}} \\ a_i &= \sum_{k=0}^{d} W_{ik} x_k \end{aligned} \tag{4.1}$$

Nello specifico:

- le a_k vengono dette posterior probabilities, cioè probabilità secondarie; quelle primarie vengono dette prior probabilities. Sono un concetto teorico della probabilità condizionale.
- i rappresenta l'i-esimo neurone del layer di output, cioè del loss layer,
- L rappresenta il numero totale di neuroni del loss layer,
- W_{ki} sono sempre i weight,
- x_i sono sempre gli input che riceve il loss layer (il vettore di dimensione d),
- le y_i sono le attivazioni degli L neuroni del loss layer.

La forma di questa funzione di output, garantisce che gli L valori di output diano somma 1. La funzione di loss ovviamente viene calcolata solo durante le fasi di forward propagation della rete. Ad un loss layer solitamente vengono date in input due blob: una contenente un vettore di dimensioni pari al numero totale di classi ed un'altra contenenti l'insieme delle label delle classi. E' potenzialmente possibile definire più di un loss layer all'interno di una rete,

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

Figura 4.11: Esempio di loss layer

```
loss := 0
for layer in layers:
  for top, loss_weight in layer.tops, layer.loss_weights:
    loss += loss_weight * sum(top)
```

Figura 4.12: Calcolo della loss totale

anche se succede raramente. E' il caso ad esempio della rete GoogLeNet [18] in cui vi sono ben 3 loss layer. Quando questo accade, solitamente nei loss layer si specifica un ulteriore parametro detto *loss_ weight*: questo parametro stabilisce quanto conta il risultato della funzione di loss di un determinato loss layer. In pratica se viene definito questo parametro, la loss finale del layer non sarà il normale valore risultante della funzione, bensì quel valore moltiplicato per il parametro di *loss_ weight*; quindi se si vuole abbassare l'importanza di un loss layer è bene specificare un *loss_ weight* compreso fra 0 e 1, in caso contrario si imposta un *loss_ weight* maggiore di 1. Specificare un *loss_ weight* pari a 1 equivale a lasciare inalterata l'importanza del loss layer; di default tale parametro vale 1. In figura 4.11 viene mostrato un semplice esempio di loss layer.

Di base Caffe assume che un qualsiasi layer avente nel campo *type* il suffisso *Loss* contribuisca al calcolo della funzione di loss, con *loss_ weight* pari a 1, nella forward propagation. Tuttavia è possibile fare in modo che anche altri layer contribuiscano al calcolo specificando in essi sempre il campo *loss_ weight*, anche se di solito non si fa mai.

L'algoritmo per il calcolo della loss totale dovrebbe ormai essere abbastanza immaginabile ed è mostrato in figura 4.12.

4.2.7 Le interfacce di Caffe

Caffe dispone di ben tre interfacce per gli utenti: una da terminale, dove tramite appositi comandi si eseguono direttamente gli script, una comple-

tamente realizzata in Python, chiamata pycaffe, ed una specifica per chi utilizza Matlab. Tralasciando l'ultima interfaccia, dato che Matlab non viene mai utilizzato nel corso di questa tesi, attualmente pycaffe risulta essere a mio avviso leggermente instabile (probabilmente per un problema riguardante gli aggiornamenti delle librerie di Python necessarie) e quindi si è scelto di utilizzare l'interfaccia standard da terminale, in quanto risulta essere sicura, veloce e la quale agisce direttamente sul codice C++ di Caffe.

4.3 Costruzione di un dataset con Caffe

In questa sezione verrà spiegato come creare, preparare e dare in input un dataset con Caffe. E' stato scelto di mostrare la procedura per la creazione di un dataset in un formato LMDB, in quanto è la più elaboriosa e soprattutto perchè è quella utilizzata in tutte le reti che verranno presentate nei successivi capitoli. Creare dataset per altri tipi di layer, come ad esempio per un image data layer, risulta essere comunque molto simile alla procedura che verrà illustrata, ed anche più semplice. Un piccolo appunto iniziale: la teoria del Data Mining insegna che nel caso si avesse un dataset in cui esiste una minoranza di classi aventi un numero di elementi molto più grande rispetto alle classi rimanenti, ci si trova davanti ad un caso di sbilanciamento delle classi del dataset. Una classificazione con questo tipo di dataset tende solitamente a favorire le classi più presenti e Caffe non è da meno. E' dunque fortemente consigliato costruire un dataset in cui il numero di elementi di ciascuna classe, che non deve essere necessariamente uguale, non sia troppo diverso per ogni classe. Nel caso non si riuscisse a fare ciò, la tecnica dell'oversampling, che consiste nella ripetizione degli elementi delle classi minoranti, può senz'altro aiutare, anche se potrebbe andare ad incrementare l'overfitting.

4.3.1 Il file synsetwords.txt

Finora per dataset si è inteso solo un insieme di immagini. In realtà oltre ad esse, per ciascuna immagine, sia del training set che del test set, è necessario specificare una propria label. Tale label serve per identificare a quale classe appartiene ogni immagine. Infatti, come già visto sempre in figura 4.5(b), un data layer riceve in ingresso una blob con i dati delle immagini ed una blob con le label di tali immagini. In che maniera si organizzano queste label e come fare in modo che Caffe capisca le associazioni immagine-label? Questo è esattamente lo scopo di questo paragrafo. Su Caffe, per ogni rete, esiste un file chiamato *synsetword.txt*. All'interno di questo file vanno specificate

```
0 consenso-informato
1 disegno
2 ecomappa
3 modulo-730
4 triangolo-bambino
5 triangolo-operatore
6 triangolo-vuoto
```

Figura 4.13: Esempio di file synsetwords.txt

le classi che sono presenti nel proprio dataset, assegnando a ciascuna classe una specifica label, esattamente come si vede in figura 4.13.

Il formato di ciascuna riga è "*label nome_classe_corrispondente*", e deve essere rispettato questo ordine e formato. In questo caso come label, per semplicità sono stati utilizzati dei semplici numeri interi univoci, ma in realtà si può immettere una stringa qualunque; lo scopo di una label è quello di identificare una classe in maniera veloce e breve, tramite una sorta di codice. Il file appena illustrato può anche essere rinominato, facendo poi attenzione al fatto che questo comporterà qualche modifica in più nel processo di allenamento.

4.3.2 Creazione file specifici per training e validation set

Una volta creato il file *synsetwords.txt*, bisognerà creare i file di testo che verranno dati in input al data layer (come visto in figura 4.8, col percorso specificato dalla keyword *source*), dove in ogni riga vi sarà una stringa di questo genere: "*nome_immagine label_classe_di_appartenenza*". Anche in questo caso il formato va rispettato. Si parla di file in plurale, perchè è necessario creare due file di questo tipo, uno per il training set ed uno per il test set, in questo caso chiamato anche validation set, dato che le label corrette sono comunque note e servono appunto per convalidare se il classificatore ha classificato correttamente una determinato gruppo di immagini. Nell'esempio di figura 4.14 viene mostrato una parte di uno di questi file relativo ad un training set.

Caffe non mette questa volta a disposizione un tool per creare i suddetti file. Il file *synsetwords.txt* è facilmente realizzabile, almeno per dataset con un numero piuttosto limitato di classi ed anche se non si fosse in questo caso converrebbe comunque specificarlo a mano, mentre gli altri due possono essere molto lunghi e pesanti da realizzare. Per questo motivo si è deciso di

```
dis_train_991.jpg 1
dis_train_992.jpg 1
dis_train_993.jpg 1
dis_train_994.jpg 1
dis_train_995.jpg 1
dis_train_996.jpg 1
dis_train_997.jpg 1
dis_train_998.jpg 1
dis_train_999.jpeg 1
eco_train_1.jpg 2
eco_train_10.jpg 2
eco_train_100.jpg 2
eco_train_101.jpg 2
eco_train_102.jpg 2
eco_train_103.jpg 2
eco_train_104.jpg 2
eco_train_105.jpg 2
```

Figura 4.14: Parte del file per i path di un training set

creare un semplice programma in C++ (*imageName.cpp*), tramite cui l'utente, specificando il numero di classi del suo dataset e quante relative immagini vi sono per ciascuna classe, ottiene uno di questi file in maniera automatica. L'operazione va ripetuta una volta per il training set ed una seconda volta per il validation set, quindi è necessario avere due cartelle, chiamate *train* e *val* in questo caso, con tutte le rispettive immagini all'interno. In figura 4.14 è stato utilizzato proprio questo programma.

In tutte le figure mostrate finora i nomi delle varie immagini sono stati modificati in base alla loro classe di appartenenza per rendere più facile il controllo della correttezza dei file creati (guardando il nome del file e label rispettiva si riesce facilmente a capire se la loro relazione è corretta) ed anche per il corretto funzionamento del piccolo programma creato. Infatti il programma legge all'interno di una cartella specificata tutti i file presenti, assegnando ai primi N_0 file l'etichetta 0, ai successivi N_1 la label 1 e così via. Perchè tutto questo abbia senso è necessario che ci sia un ordine fra i file letti e le label assegnate, cioè i file devono essere tutti raggruppati in base alla loro classe e tutti i raggruppamenti devono rispettare l'ordine specificato nel file *synsetwords.txt*. Per fare questo è necessario rinominare tutti i file, scegliendo un nome molto simile fra due immagini della stessa classe (ad esempio *modulo_1* e *modulo_2*) e rispettando con i nominativi scelti per ciascuna classe l'ordine del synset file (è sufficiente scegliere nominativi che in ordine alfabetico abbiano lo stesso ordine del synset file). Rinominare grandi quantità di file può essere noioso ed anche un grosso spreco di tempo. Per questo, prima di creare tutti i file finora incontrati, è stato creato

```
#!/bin/bash

COUNTER=1;

cd consenso_informato/train;

for f in * ; do
        mv "$f" "cons_train_$COUNTER.jpg" ;
        let COUNTER=COUNTER+1;
        done;

cd ../val;
COUNTER=1;

for f in * ; do
        mv "$f" "cons_val_$COUNTER.jpg" ;
        let COUNTER=COUNTER+1;


done;
```

Figura 4.15: Parte script per rinominazione file

un altro piccolo script per questa funzione, di cui una parte è mostrata in figura 4.15. La parte di script mostrata permette di rinominare i file di una singola classe, ma per effettuare la stessa operazione sulle altre classi il resto del codice è molto simile: basta cambiare il nome della cartella e dei file. In quest'ultimo trovare un'ordine alfabetico che rispetti l'ordine del synset file. Fare attenzione al fatto che, nel caso lo si volesse utilizzare, questo script funziona solo se si hanno tutte immagini *.jpg* nel proprio dataset. Nel caso si possedesse qualche immagine con un qualche formato diverso, è comunque sufficiente rimuovere provvisoriamente tali immagini dal dataset (basta fare una veloce ricerca con *.png* ad esempio) ed organizzarle in altre cartelle raggruppandole per estensione, applicare lo script al dataset con le immagini tolte per ottenere il solito risultato, modificare lo script mettendo l'estensione di immagini voluta ed applicarlo al raggruppamento di immagini con la stessa estensione e così via.

Come si può notare, il programma procede ordinatamente, ovvero scrivendo prima tutte le righe delle immagini della prima classe, poi quelle della seconda e così via fino all'ultima, sempre seguendo l'ordine del synset file. Di norma è meglio "randomizzare un po' i dati"; per fare questo è stato creato un semplice script che effettua l'operazione di mescolamento dei dati su entrambi i file creati (per il training ed il validation set), mostrato in figura 4.16.

Il risultato che si ottiene è visibile in figura 4.17.

```
#!/bin/bash

sort -R orderTest.txt >> randomTest.txt;
sort -R orderTrain.txt >> randomTrain.txt;
```

Figura 4.16: Script per mescolare i dati

```
trBambino_val_38.jpg 4
cons_val_23.jpg 0
dis_val_28.jpg 1
trOperatore_val_17.jpg 5
dis_val_1.jpg 1
mod_val_34.jpg 3
eco_val_39.jpg 2
trOperatore_val_20.jpg 5
eco_val_7.jpg 2
eco_val_14.jpg 2
cons_val_36.jpg 0
mod_val_13.jpg 3
trOperatore_val_1.jpg 5
trVuoto_val_27.jpg 6
trBambino_val_37.jpg 4
trBambino_val_23.jpg 4
trVuoto_val_41.jpg 6
trVuoto_val_12.jpg 6
mod_val_2.jpg 3
cons_val_12.jpg 0
cons_val_38.jpg 0
eco_val_34.jpg 2
```

Figura 4.17: Path delle immagini in ordine casuale

Nel caso si volesse utilizzare ad esempio un image data layer, il processo di costruzione fin qui è esattamente lo stesso, l'unica differenza è che questo tipo di layer è che in ciascuna riga dei file di testo per il training ed il validation set, ci sia il path assoluto di ciascuna immagine. Per ottenere questo è sufficiente modificare il programma *imageName* in modo che per ogni immagine non salvi solo il suo nome, ma l'intero suo path (nome compreso). Fatto questo il processo di creazione di un dataset per questo tipo di layer termina qui; non è così invece per un dataset in formato LMDB, come si vedrà fra poco.

4.3.3 Creazione formati LMDB

Fortunatamente, per creare dei dati con questo tipo di formato, Caffe fornisce uno script chiamato "*crate_ imagenet.sh*". Anche in questo caso, come per la mean subtraction, questo file era utilizzato per creare i dati in formato LMDB per il dataset ImageNet. Modificando opportunamente lo script si può tuttavia fare lo stesso su un qualsiasi proprio dataset. In figura 4.18 e 4.19 viene mostrato lo script originale dove vengono evidenziate le aree in cui apportare le modifiche.

La prima area da modificare riguarda i percorsi della cartella del training set e del validation set. La seconda area riguarda il resize: è bene settare a true il flag "RESIZE", in modo che venga in automatico effettuato un ridimensionamento 256 x 256 di tutte le immagini. Le ultime due aree di modifica specificano i percorsi dei file di testo del training set e del validation set creati in precedenza e dove salvare i file di output *.lmdb* per ciascuno di essi. Una volta modificato il tutto secondo le specifiche, basta avviare lo script ed attendere i risultati; il processo potrebbe richiedere diversi minuti, a seconda di quante immagini si hanno nel dataset. Terminata l'elaborazione, i file *.lmdb* sono pronti per essere utilizzati da una qualsiasi rete avente un input layer di tipo *Data*, come mostrato in figura 4.20. Dalla stessa immagine si nota che nel campo *source* è sufficiente mettere il path della cartella che contiene i due file *.lmdb*.

4.4 I layer più comuni in Caffe

Per meglio comprendere le reti che verranno affrontate nei successivi capitoli, in questa sezione saranno mostrati i principali layer solitamente utilizzati con Caffe. Alcuni di questi layer sono già stati visti nello specifico nel capitolo precedente, come ad esempio il layer convoluzionale, e quindi in questo caso ci si limiterà a mostrarne la struttura su Caffe, senza spiegarne la funzione

```
#!/usr/bin/env sh
# Create the imagenet lmdb inputs
# N.B. set the path to the imagenet train + val data dirs

EXAMPLE=examples/imagenet
DATA=data/ilsvrc12
TOOLS=build/tools

TRAIN_DATA_ROOT=/path/to/imagenet/train/
VAL_DATA_ROOT=/path/to/imagenet/val/

# Set RESIZE=true to resize the images to 256x256. Leave as false if images have
# already been resized using another tool.
RESIZE=false
if $RESIZE; then
  RESIZE_HEIGHT=256
  RESIZE_WIDTH=256
else
  RESIZE_HEIGHT=0
  RESIZE_WIDTH=0
fi

if [ ! -d "$TRAIN_DATA_ROOT" ]; then
  echo "Error: TRAIN_DATA_ROOT is not a path to a directory: $TRAIN_DATA_ROOT"
  echo "Set the TRAIN_DATA_ROOT variable in create_imagenet.sh to the path" \
       "where the ImageNet training data is stored."
  exit 1
fi

if [ ! -d "$VAL_DATA_ROOT" ]; then
  echo "Error: VAL_DATA_ROOT is not a path to a directory: $VAL_DATA_ROOT"
  echo "Set the VAL_DATA_ROOT variable in create_imagenet.sh to the path" \
       "where the ImageNet validation data is stored."
  exit 1
fi
```

Figura 4.18: Prima parte script “create_imagenet.sh”

```
echo "Creating train lmdb..."

GLOG_logtostderr=1 $TOOLS/convert_imageset \
    --resize_height=$RESIZE_HEIGHT \
    --resize_width=$RESIZE_WIDTH \
    --shuffle \
    $TRAIN_DATA_ROOT \
    $DATA/train.txt \
    $EXAMPLE/ilsvrc12_train_lmdb

echo "Creating val lmdb..."

GLOG_logtostderr=1 $TOOLS/convert_imageset \
    --resize_height=$RESIZE_HEIGHT \
    --resize_width=$RESIZE_WIDTH \
    --shuffle \
    $VAL_DATA_ROOT \
    $DATA/val.txt \
    $EXAMPLE/ilsvrc12_val_lmdb

echo "Done."
```

Figura 4.19: Seconda parte script “create_imagenet.sh”

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
```

Figura 4.20: Esempio data layer con dati lmdb

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4          # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```

Figura 4.21: Esempio di layer convoluzionale con Caffe

che rimane esattamente la stessa. Verranno invece tralasciati i data layer, dato che sono già stati visti nel paragrafo 4.2.4 di questo capitolo.

4.4.1 Convolutional layer

Niente di nuovo in questo caso, a parte alcuni specifici parametri generali che Caffe utilizza, riguardanti non solo un layer convoluzionale e che fra poco verranno spiegati. In figura 4.21 ne viene mostrata la struttura su Caffe. Il layer convoluzionale risulta essere sicuramente il più complesso da definire su Caffe, per via di tutti i suoi parametri e del fatto che possiede anche una fase di backward propagation.

Il piccolo costrutto denotato con *weight_filler* serve per l'inizializzazione dei weight all'inizio dell'allenamento. Come era già stato detto nel paragrafo 2.2.2, è bene che i weight non siano tutti settati a 0, perchè questo vorrebbe dire che tutti i kernel (i filtri) apprenderebbero le stesse informazioni; è meglio inizializzarli con valori molto prossimi allo 0, ma diversi fra loro. Nell'esempio i weight vengono inizializzati seguendo la distribuzione di una normale Gaussiana con media 0 e deviazione standard 0.01. Analoga funzione per quanto riguarda i bias (*bias_filler*) che però possono invece essere settati tutti a 0 in maniera fissa. Il campo *num_output* rappresenta il numero di kernel utilizzati, ovvero il parametro di depth K visto nel paragrafo

3.2.1 per il layer convoluzionale; *kernel_ size* indica il lato del receptive field di ogni kernel.

I campi *lr_ mult* e *decay_ mult*, che si possono notare sempre in figura 4.21, sono invece una novità. Non sono parametri specifici di un layer convoluzionale, ma bensì parametri per un qualunque layer che abbia una backward propagation, cioè che possegga dei parametri allenabili, anche se poi, come si vedrà meglio avanti, ciò ha effetti anche nelle forward propagation. Non sono nemmeno obbligatori, ma molto spesso aiutano a configurare ogni layer di questo tipo nella maniera che si vuole: intuitivamente essi fungono da fattore moltiplicativo per i parametri generali di learning rate, visto nel paragrafo 2.2.2 con la formula (2.4), e di weight decay, uno dei parametri presenti nella funzione di costo nella formula (2.3). Entrambi questi parametri sono già stati presentati in precedenza e verranno comunque ripresi di nuovo anche nella sezione 4.5 di questo capitolo. Ad ogni modo, lo scopo dei campi menzionati è quello di poter regolare, settando opportuni valori per essi, il comportamento di un layer durante la sua backward propagation. Se ad esempio per un layer si settasse, sia per i weight che per i bias, un valore di *lr_ mult* più grande rispetto a quello degli altri layer, ciò vorrebbe dire che quel layer avrà un learning rate più alto e quindi avrà un aggiornamento più incisivo dei propri parametri nella fase di backward propagation. Analoga cosa per quanto riguarda *decay_ mult*: anch'esso permette di modificare la backward propagation di un layer, solo che in questo caso si avranno effetti sul calcolo dei gradienti, in quanto si avranno dei risultati diversi per la funzione di loss. Settare *decay_ mult* a 0 significa annullare l'intero fattore di weight decay per il calcolo della loss di quel layer (il secondo fattore nella formula (2.3)) e quindi di avere risultati diversi nel calcolo dei gradienti della stessa funzione di loss della formula (2.4). Come si può notare, non esiste una maniera specifica per dichiarare se, in questo contesto, ci si sta riferendo ai weight o ai bias. Semplicemente il primo costrutto *param* indica i weight, mentre il secondo indica i bias. Ovviamente questi campi, se utilizzati, riguardano solamente i layer aventi parametri allenabili.

4.4.2 ReLU layer

Qui non cambia praticamente nulla rispetto a quanto detto nel paragrafo 3.2.2. I ReLU layer sono i più utilizzati, non solo su Caffe, ma in generale per quanto riguarda le reti neurali convoluzionali negli ultimi anni, visto che risultano molto efficienti in termini di velocità computazionale, eseguendo comunque bene la loro funzione di incremento di non linearità delle attivazioni. La figura 4.22 illustra un esempio di ReLU layer su Caffe.

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

Figura 4.22: Esempio di ReLU layer con Caffe

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

Figura 4.23: Esempio di Sigmoid layer con Caffe

Esattamente come specificato nel paragrafo 3.2.2, esistono altri tipi di funzioni diverse rispetto a quella che implementano i ReLU layer, potenzialmente utilizzabili per lo stesso obbiettivo. Fra poco infatti se ne vedranno alcuni esempi.

4.4.3 Sigmoid layer

Questo tipo di layer ha la stessa funzione di un ReLU layer, ma viene utilizzata la funzione della sigmoide per tale scopo. In figura 4.23 ve ne è un esempio.

4.4.4 TanH layer

Anche qui, tali layer hanno lo stesso scopo dei ReLU layer, ma si utilizza la funzione della tangente iperbolica. In figura 4.24 ve ne è un esempio.

Esisterebbero altri tipi di layer con le medesime funzioni di un ReLU layer, ma quelli illustrati fin qui sono i più famosi ed utilizzati. Per visionare tutti i layer di questo tipo è possibile comunque consultare la sezione del catalogo dei layer di Caffe, anche se attualmente non risulta essere propriamente aggiornata.

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

Figura 4.24: Esempio di TanH layer con Caffe

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3 # pool over a 3x3 region
    stride: 2      # step two pixels (in the bottom blob) between pooling regions
  }
}
```

Figura 4.25: Esempio di layer di pooling con Caffe

4.4.5 Pooling layer

Non vi sono novità nemmeno in questo caso. La figura 4.25 mostra un esempio di layer di pooling su Caffe.

I parametri sono esattamente gli stessi visti nel paragrafo 3.2.3. MAX specifica che in questo esempio si utilizza la funzione di massimo. Nel caso ci fosse stato AVE allora si utilizzerebbe la funzione di media. Si noti che in questo caso non vi sono tutti i campi visti in precedenza (*weight_filler*, *lr_mult*, ecc.) in quanto un layer di pooling non possiede parametri allenabili. Vale la stessa cosa per tutti i successivi layer che si vedranno.

4.4.6 LRN layer

Un Local Response Normalization layer è un layer che Caffe utilizza per effettuare operazioni di normalizzazione. Un esempio è presente in figura 4.26.

La formula di normalizzazione utilizzata è la seguente:

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(1 + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^i_{x,y})^2\right)^\beta} \tag{4.2}$$

- α e β sono due parametri impostabili nel layer che rappresentano il fattore di scala e l'esponente nella formula,

```
layer {
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
```

Figura 4.26: Esempio di un layer di normalizzazione con Caffe

- $a^i_{x,y}$ rappresenta l'attivazione di un neurone calcolata applicando l'i-esimo kernel nella posizione (x,y), alla quale si solito è stata applicata anche un'operazione di ReLU o funzioni simili viste prima,
- N è il numero totale di kernel,
- n, che è impostabile nel layer, rappresenta le dimensioni della regione locale in cui applicare l'operazione di normalizzazione, da cui il nome Local Response Normalization,
- $b^i_{x,y}$ rappresenta il valore normalizzato risultante di $a^i_{x,y}$.

4.4.7 Loss layer

Già incontrato nella sezione 4.2.3 di questo capitolo e mostrato in figura 4.11. Come già detto, una qualsiasi rete su Caffe deve possedere almeno un loss layer, per il calcolo della funzione di loss. Esistono vari tipi di loss layer, i quali si differenziano in base al tipo di funzione che essi utilizzano. Il più utilizzato è sicuramente quello con la funzione di Softmax (formula (4.1)), ma si possono potenzialmente usare anche altri tipi di funzioni come ad esempio una qualche variante della sigmoide oppure l'Infogain, concetto presente nella teoria del Data Mining.

4.4.8 SoftMax layer

Esiste tuttavia anche un tipo di layer softmax non utilizzato per il calcolo della funzione di loss. Il suo scopo principale è quello, sempre utilizzando la solita funzione, di classificare una più immagini esterne al dataset, come si vedrà nel paragrafo 4.9.

4.4.9 Accuracy layer

Questo tipo di layer serve per calcolare l'accuracy della rete in fase di testing. Per accuracy si intende il rapporto fra il numero di immagini classificate correttamente nel validation set ed il numero totale di immagini presenti nel validation set, come mostrato nella seguente formula.

$$\text{Accuracy} = \frac{\text{n°immagini classificate correttamente nel validation set}}{\text{Totale immagini validation set}}$$

Questa è la metrica principale con cui Caffe misura la bontà di una rete: più l'accuracy è alta e più significa che la rete funziona bene. A dire il vero si parla in questo caso di top-1 accuracy, cioè la percentuale di immagini del validation set in cui la classe corretta ha la probabilità più alta rispetto a tutte quelle delle altre classi, sempre rapportato sul totale delle immagini del validation set.

Spesso viene tuttavia utilizzata anche la top-5 accuracy, ovvero la percentuale di immagini del validation set in cui la classe corretta possiede una fra le 5 probabilità più alte. Questa metrica ha poco senso per dataset aventi poche classi: se ad esempio si possiede un dataset con 7 classi in totale, risulta quasi ovvio che si avrà una top-5 accuracy pari a 1 o quasi, in quanto, a meno che la rete non funzioni proprio male, sicuramente la classe corretta sarà fra le prime cinque. La metrica invece assume valore quando si hanno dataset con un grande numero di classi, come ad esempio 1000 classi. La top-1 accuracy anche in questo caso risulta sempre utile per capire la bontà della rete, ma anche se non si possedesse una top-1 accuracy alta, avere una top-5 accuracy alta indica che comunque la rete non funziona proprio male, perchè in molti casi è molto vicina a classificare la classe corretta, che nel secondo esempio avrebbe una probabilità fra i primi 5 posti su un totale di 1000. In figura 4.27 viene mostrato un esempio di un layer di accuracy che calcola in questo caso la top-5 accuracy.

Ovviamente se si vogliono entrambi i tipi di accuracy è necessario definire due layer di accuracy, uno per ciascun tipo. Per avere la top-1 accuracy, guardando sempre figura 4.27, è sufficiente rimuovere il costrutto *accuracy_param*. Fin qui si è parlato di top-1 e top-5, perchè sono le più utilizzate, ma volendo è possibile definire una top-x accuracy, specificando il valore x (ovviamente intero positivo) nel campo *top_k*.

4.4.10 Inner Product layer

Inner Product è semplicemente il nome con cui Caffe si riferisce ad un layer FC. Per il resto nulla di nuovo. Un esempio è presente in figura 4.28.

```
layer {
  name: "loss1/top-5"
  type: "Accuracy"
  bottom: "loss1/classifier_myNet"
  bottom: "label"
  top: "loss1/top-5"
  include {
    phase: TEST
  }
  accuracy_param {
    top_k: 5
  }
}
```

Figura 4.27: Esempio di un layer di accuracy con Caffe

```
layer {
  name: "fc7"
  type: "InnerProduct"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
```

Figura 4.28: Esempio di un layer FC con Caffe

num_ output rappresenta il numero di neuroni di cui è costituito il layer e quindi anche il suo output (paragrafo 3.2.5). Notare che vi sono i soliti parametri aggiuntivi per il semplice fatto che questo è uno dei layer che, come quello convoluzionale, possiede dei parametri allenabili.

4.4.11 Dropout layer

Questo layer si usa per l'applicazione dell'omonima tecnica di Dropout. Essa solitamente viene utilizzata per diminuire l'overfitting all'interno di una rete e per questo i layer di dropout spesso vengono posizionati dopo i layer che possiedono un grande quantità di parametri allenabili, come ad esempio i FC layer. Questa tecnica permette di settare a 0, e quindi di escludere, l'attivazione di una certa percentuale dei neuroni del layer precedente. La probabilità che l'attivazione di un neurone venga settata a 0 viene indicata dal parametro *dropout_ ratio* all'interno del layer, tramite un numero compreso fra 0 e 1: in pratica l'attivazione di un neurone viene tenuta con probabilità *dropout_ ratio* e viene scartata, cioè settata a 0, con probabilità (1-*dropout_ ratio*). I neuroni interessati dall'operazione non influiscono dunque durante la forward propagation e nemmeno durante la successiva backward propagation di un certo input. In questo modo per ogni input la rete possiede effettivamente un'architettura leggermente diversa sui FC layer precedenti un dropout layer (alcune connessioni sono attive ed altre no, in maniera differente ogni volta), anche se tutte queste architetture posseggono gli stessi weight. Questa tecnica riduce la possibilità che un neurone faccia affidamento sulla presenza di altri neuroni, in quanto questo non è garantito. E' dunque costretto ad apprendere feature più robuste che risultino utili anche con collegamenti con altri neuroni diversi. Attenzione che utilizzare la tecnica di Dropout non significa cambiare le dimensioni della rete ogni volta, le quali rimangono sempre le stesse, ma solo settare a 0 alcuni parametri in sostanza. In figura 4.29 è mostrato un esempio della sua definizione con Caffe.

4.4.12 Concat layer

Questo tipo di layer viene raramente utilizzato, in quanto spesso, se la rete è grande e non viene gestita opportunamente la presenza di questi layer, vi possono essere grossi problemi in termini di quantità di dati elaborati dalla rete. Come si vedrà nei successivi capitoli, la rete GoogLeNet [18] è l'unica fra quelle presentate a fare uso di questi layer, ma per essi ha dovuto inventarsi uno specifico stratagemma per limitare la quantità di dati all'interno della rete.

```
layer {
  name: "drop7"
  type: "Dropout"
  bottom: "fc7"
  top: "fc7"
  dropout_param {
    dropout_ratio: 0.5
  }
}
```

Figura 4.29: Esempio di un dropout layer con Caffe

```
layer {
  name: "inception_4e/output"
  type: "Concat"
  bottom: "inception_4e/1x1"
  bottom: "inception_4e/3x3"
  bottom: "inception_4e/5x5"
  bottom: "inception_4e/pool_proj"
  top: "inception_4e/output"
}
```

Figura 4.30: Esempio di un concat layer con Caffe

La funzione di questi layer comunque è piuttosto semplice: prendono in input un certo numero di blob e restituiscono come output una singola blob di dimensioni pari alla somma delle dimensioni delle blob avute in input. L'esempio è in figura 4.30: qui si ha un concat layer che prende in input 4 blob.

4.5 Il solver

Il solver è un termine che su Caffe indica uno specifico file, ovvero *solver.prototxt*, nel quale vengono definiti tutti parametri necessari per l'allenamento di una rete neurale convoluzionale. Non sono i parametri specifici di un singolo layer, bensì dei parametri generali per l'allenamento della rete.

4.5.1 Il campo *phase*

Prima di illustrare un solver dettagliatamente è necessario fare un piccolo appunto riguardo ad uno dei parametri incontrati precedentemente e che non è ancora stato spiegato, ovvero il parametro *phase*. Questo parametro

può assumere solo due valori, TRAIN e TEST. L'allenamento di una rete con Caffe è infatti costituito da fasi alterne di TRAIN e TEST. In particolare:

- si effettuano un certo numero di iterazioni cercando di diminuire la funzione di loss il più possibile (fase di TRAIN, un certo numero di forward + backward propagation),
- si effettua un test generale sul validation set per vedere l'accuratezza attuale con i parametri ottenuti (fase di TEST, solo forward propagation per ciascuna immagine nel validation set),
- se l'accuratezza ottenuta non soddisfa le aspettative si ripete l'intero procedimento, altrimenti ci si ferma.

Infatti in realtà in ciascuna rete non esiste un solo data layer, bensì due: uno che carica i dati dal training set, e che avrà *phase* = TRAIN, ed un altro che carica i dati dal validation set, con *phase*= TEST. In questo modo durante le fasi di TRAIN lavora solo il primo dei due, mentre nelle fasi di TEST entra in gioco il secondo. Si avranno quindi due valori del parametro batch size, uno relativo al training set ed un altro al validation set, come mostrato in figura 4.31.

Il campo *phase*, impostabile per un qualsiasi layer, permette dunque di specificare se un determinato layer entra in funzione solo in una determinata fase. Ad esempio un layer che calcola l'accuratezza ha senso utilizzarlo solo nella fase di TEST. Se in un layer non viene specificato nessun campo *phase*, come in quasi tutti i layer centrali, allora quel layer viene utilizzato in entrambe le fasi di allenamento.

4.5.2 I parametri di un solver

I parametri che si possono impostare in un solver, all'interno del protocol buffer *solver.prototxt*, sono i seguenti:

- **net** specifica il percorso del file dove è stata definita la rete,
- **base_lr** indica il valore del parametro di learning rate iniziale per la rete; si veda il paragrafo 2.2.2 per riprendere le informazioni di questo parametro),
- **lr_policy** è la tipologia, cioè il tipo di funzione, con cui il learning rate decresce nel corso dell'allenamento di una rete. La motivazione per cui il learning rate deve decrescere è per il semplice fatto che inizialmente una rete ha molto da apprendere perchè sicuramente non possiede dei

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {phase: TRAIN}
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {phase: TEST}
  transform_param {
    mirror: false
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_val_lmdb"
    batch_size: 50
    backend: LMDB
  }
}
```

Figura 4.31: I due data layer le fasi di TRAIN e TEST

```
// The learning rate decay policy. The currently implemented learning rate
// policies are as follows:
//    - fixed: always return base_lr.
//    - step: return base_lr * gamma ^ (floor(iter / step))
//    - exp: return base_lr * gamma ^ iter
//    - inv: return base_lr * (1 + gamma * iter) ^ (- power)
//    - multistep: similar to step but it allows non uniform steps defined by
//      stepvalue
//    - poly: the effective learning rate follows a polynomial decay, to be
//      zero by the max_iter. return base_lr (1 - iter/max_iter) ^ (power)
//    - sigmoid: the effective learning rate follows a sigmod decay
//      return base_lr ( 1/(1 + exp(-gamma * (iter - stepsize))))
//
// where base_lr, max_iter, gamma, step, stepvalue and power are defined
// in the solver parameter protocol buffer, and iter is the current iteration.
```

Figura 4.32: Le policy di Caffe

buoni valori di weight e bias, ma più avanza nel suo allenamento più i valori in questione che trova dovrebbero essere accurati e quindi ci si discosta di meno da essi durante il loro aggiornamento. Questo comporta il fatto di dover abbassare il learning rate. In figura 4.32 sono mostrate le possibilità che Caffe mette a disposizione per questo parametro, tratte da uno dei file C++ che definiscono il framework. La più utilizzata è sicuramente la policy *step* dove in pratica il learning rate viene decrementato di una certa quantità dopo un certo numero di iterazioni fissato (la quantità con cui viene decrementato diminuisce nel corso dell'allenamento), ma alcune particolari reti, come GoogLeNet [18], utilizzano invece la policy *poly* in cui ad ogni iterazione il learning rate si abbassa di una piccola quantità. Queste sono le due sole policy che si vedranno in questa tesi.

- **gamma** è un valore necessario solo nel caso di una policy di tipo *step*. E' il fattore che determina il decremento del learning rate dopo un certo numero di iterazioni. Impostare ad esempio gamma = 0.1, significa moltiplicare il learning rate attuale per 0.1, ovvero dividerlo per 10.

- **stepsize** è un valore necessario solo nel caso di una policy di tipo *step*. Rappresenta il numero di iterazioni dopo le quali ridurre il learning rate.

- **display** indica dopo ogni quante iterazioni verranno visualizzate le informazioni relative alla rete (valore della funzione di loss).

- **power** è un valore necessario solo nel caso di una policy di tipo *poly*. Rappresenta la potenza a cui è elevata la formula con cui decrementare il learning rate dopo ogni iterazione.

- **test_interval** stabilisce dopo quante iterazioni fare un test (sull'intero validation set) per avere un'idea dell'accuratezza ottenuta con i parametri attuali. In base al risultato ottenuto si può decidere se continuare l'allenamento oppure fermarlo prima che raggiunga un numero *max_iter* di iterazioni.

- **test_iter** rappresenta il numero di batch del validation set utilizzate per il testing. Il valore di questo parametro va settato secondo la formula

$$(\textbf{test\_iter}) * (\textbf{val\_batch\_size}) = \textbf{n}^\circ \textbf{ immagini nel validation set}$$

dove *val_batch_size* rappresenta il valore del parametro di batch size impostato per il validation set. In realtà spesso settare i parametri di *test_iter* e *val_batch_size* perchè sia valida la suddetta formula è difficile, essendo possibile impostare solo valori interi. E' sufficiente comunque che:

$$(\textbf{test\_iter}) * (\textbf{val\_batch\_size}) \geq \textbf{n}^\circ \textbf{ immagini nel validation set}$$

perchè tanto una volta terminate le immagini nel validation set, si ricomincia daccapo riprendendo la prima immagine di queste. L'importante è non avere

$$(\textbf{test\_iter}) * (\textbf{val\_batch\_size}) < \textbf{n}^\circ \textbf{ immagini nel validation set}$$

in quanto si andrebbe ad escludere qualche dato dal validation set, oppure

$$(\textbf{test\_iter}) * (\textbf{val\_batch\_size}) >> \textbf{n}^\circ \textbf{ immagini nel validation set}$$

perchè si avrebbe un'inutile elaborazione extra.

- **max_iter** rappresenta il numero massimo di iterazioni che la rete deve fare nel corso del suo allenamento. In questo caso non esiste una formula per sapere come impostare questo parametro. E' utile sapere comunque che

$$\textbf{n° immagini di apprendimento} = (\textbf{max\_iter}) * (\textbf{train\_batch\_size})$$

dove *train_batch_size* è il batch size per il training set, e che

$$\textbf{n° epoch} = \frac{\textbf{n° immagini di apprendimento}}{\textbf{n° immagini training set}}$$

Un epoch è semplicemente un'unità di misura per l'allenamento di una rete neurale convoluzionale, poco utilizzata a dire il vero su Caffe. Si può pensare di allenare la propria rete per un certo numero di epoch e verificare al termine se l'allenamento ha prodotto buoni risultati o meno. Nel secondo caso una possibilità è quella di aumentare il numero di epoch dell'allenamento e se necessario modificare qualche altro parametro.

- **momentum** indica il peso dell'aggiornamento precedente dei parametri durante l'algoritmo di gradient-descent. Solitamente si lascia sempre il valore 0.9, in quanto è stato visto che in questo modo si ottengono buoni risultati.

- **weight_decay** è il parametro di regolazione della formula (2.3) della funzione di costo. Anche qui il valore di solito viene lasciato intorno ad un valore fisso, in questo caso però circa 0.005 o valori poco distanti.

- **snapshot** indica dopo quante iterazioni, in maniera ciclica, viene salvato un temporaneo modello della rete. In realtà vengono salvati due file: uno con estensione *.caffemodel* che rappresenta effettivamente il modello della rete con i parametri attuali, ed un file *.solverstate* che invece verrà visto nel dettaglio più avanti.

- **snapshot_prefix** indica il percorso dove Caffe deve salvare i suoi modelli nel corso dell'allenamento

```
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

Figura 4.33: Esempio di solver

- **solver_mode** specifica in che modalità Caffe deve allenare la rete. Può assumere ovviamente solo uno dei valori fra GPU e CPU.

In figura 4.33 viene mostrato un esempio di solver con policy *step* (è il solver della rete AlexNet [16]).

4.6 Allenamento di una rete con Caffe

Ora che si è a conoscenza di come costruire un dataset per Caffe (paragrafo 4.3) e ci si è fatti un'idea di cosa sia e come funzioni un solver (paragrafo 4.5) è possibile capire come allenare una rete con Caffe. Verrà dunque ora presentato un semplice esempio per fare questo. I file interessati nell'operazione sono i seguenti:

- ***train_val.prototxt*** è il file contenente l'intera definizione della rete neurale convoluzionale; un insieme dunque di definizioni dei vari layer con i loro parametri, che insieme costituiscono la rete. Il file può essere rinominato come si vuole; in questa sezione è stato scelto di lasciare i nominativi standard utilizzati da Caffe. Il file possiede di default questo particolare nome perchè include in esso sia le operazioni della fase di train che quelle di test, grazie al parametro *phase*. Non molto tempo fa non esisteva tale parametro e si era quindi costretti ad utilizzare due file separati, uno per ciascuna fase (*train.prototxt* e *val.prototxt*); l'unione dei due file ha portato al nome standard in questione.

- ***mean_file.binaryproto***. Questo file è opzionale e sua presenza dipende se esso è richiesto o meno nella definizione dei data layer in

train_ val.prototxt. Per il resto le sue funzioni sono già state rese note in precedenza.

- ***solver.prototxt*** è il solver che specifica come allenare la rete
- ***nomerete_ iter_ x.caffemodel*** è uno dei modelli che Caffe salva dopo 'x iterazioni specificate dal parametro *snapshot* del solver. Verrà salvato un file di questo tipo ogni x iterazioni fino al raggiungimento del numero massimo di iterazioni specificate dal parametro *max_ iter* del solver. Questi file binari contengono l'insieme di tutti i parametri ottenuti dalla rete nel momento in cui si effettua lo snapshot. Se l'utente da terminale digita CTRL+C, in automatico viene salvato uno di questi file al termine dell'iterazione corrente, interrompendo successivamente l'intero allenamento. Questo è uno dei file che saranno successivamente necessari per le operazioni di classificazione future, esterne al dataset attuale.
- **nomerete_iter_x.solverstate**. Il salvataggio di uno di questi file avviene sempre simultaneamente al salvataggio di un file *.caffemodel*. Questo file serve solamente nel caso l'allenamento venisse interrotto. Le operazioni di allenamento spesso sono esageratamente lunghe e quindi l'utente può decidere di voler interrompere l'allenamento per magari riprenderlo in un secondo momento dando il seguente comando:

```
./build/tools/caffe train --solver=models/bvlc_reference_caffenet/solver.prototxt
snapshot=models/bvlc_reference_caffenet/caffenet_iter_10000.solverstate
```

specificando l'ultimo file *.solverstate* salvato in precedenza ed il solver. Intuitivamente lo scopo di questi file è quello di tenere in memoria i parametri attuali del solver ad una certa iterazione, ad esempio l'iterazione a cui si era arrivati, il learning rate attuale, ecc.).

Da terminale, all'interno della cartella di installazione di Caffe, per allenare una rete dall'inizio si può ora dare il comando:

```
./build/tools/caffe train --solver=models/bvlc_reference_caffenet/solver.prototxt
```

specificando solo il percorso del solver. Dopo aver dato questo comando, da terminale si dovrebbe notare una serie di output che specificano i dati caricati da ciascun layer della rete, come ad esempio in figura 4.34. Dopodichè la rete comincerà le sue iterazioni, seguendo i parametri indicati dal solver.

L'allenamento di una rete da zero ("from scratch" come si dice spesso) richiede un numero solitamente molto alto di iterazioni, dipendenti comunque

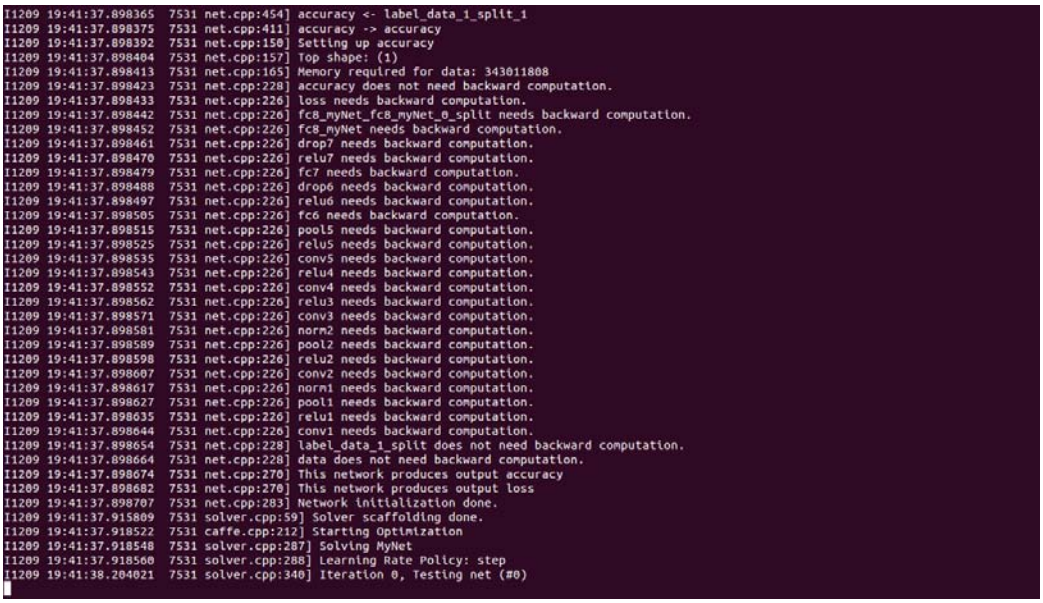


Figura 4.34: Inizio del training di una rete

dal dataset di cui si dispone, cioè da quante immagini ci sono, la loro qualità, quanto sono diverse fra loro le classi nel dataset, ecc. Si intuisce dunque che per allenare una rete ci possono volere molti giorni, settimane anche nel caso si allenasse con la modalità CPU, in quanto risulta decisamente molto più lenta rispetto alla modalità GPU. E' bene sempre ricordare che si può comunque interrompere l'allenamento e riprenderlo successivamente con la procedura prima illustrata. E' necessario tenere sotto controllo l'andamento del valore risultante della funzione di loss, perchè esso può dare un'idea di come sta procedendo l'allenamento (si ricorda che tale valore dovrebbe abbassarsi man mano che le iterazioni aumentano). Tuttavia non è sempre così facile, perchè ovviamente non tutto funziona bene la prima volta e nel caso fosse così ci si potrebbe ritenere fortunati. Basta anche un semplice parametro non settato adeguatamente e l'intero allenamento può non dare i risultati sperati. Al fine di controllare un po' questi errori, riporterò in una delle prossime sezioni alcune procedure che possono risultare utili a chi volesse cominciare ad utilizzare Caffe per allenare una propria rete neurale convoluzionale.

4.7 Operazione di fine-tuning

L'operazione di fine-tuning è una fra quelle più in voga negli ultimi anni con Caffe e non solo. Effettuare un'operazione di fine-tuning significa prendere

una rete che è già stata costruita ed uno dei suoi modelli risultanti, apportare le opportune modifiche dei parametri dei vari layer in modo che possa adattarsi al proprio dataset, impostare un determinato learning rate in ciascun layer di tale rete e dare successivamente il comando (il tag -gpu può essere omesso nel caso si voglia operare in modalità CPU):

./build/tools/caffe train -solver models/finetune_flickr_style/solver.prototxt -weights models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel -gpu 0

Come si può notare, il comando è molto simile a quello dato per l'allenamento classico di una rete, con la differenza che qui bisogna specificare oltre al solver anche il percorso del modello, cioè del file *.caffemodel*, della rete su cui si effettua il processo di fine-tuning. Tuttavia prima di dare il comando descritto sopra, è necessaria una fase preliminare di preparazione dei dati, la quale verrà ora descritta.

4.7.1 Modifiche al file train_val.prototxt

Innanzitutto è necessario apportare le seguenti modifiche del file *train_val.prototxt* della rete su cui si fa il fine-tuning:

1. il nome della rete, definito nella prima riga del file dal campo *name*
2. il percorso, ed il formato se necessario, dei file di input nel data layer
3. il nome dell'ultimo layer che possiede dei parametri allenabili
4. se necessario impostare dei parametri di batch size, sia per il training che per il validation set, in base alle caratteristiche del proprio hardware
5. il valore del campo *num_output* dell'ultimo layer che possiede dei parametri allenabili. E' necessario inserire in questo caso un valore pari al numero di classi presenti nel proprio dataset
6. per ciascun layer avente parametri allenabili, impostare opportuni valori, sia per i weight che per i bias, nei campi *lr_mult* e *decay_mult*.

Per quanto riguarda l'ultimo punto, di norma si impostano valori molto bassi per tutti i layer a parte l'ultimo, sempre fra quelli con parametri allenabili; per l'ultimo di questi layer si impostano invece parametri di *lr_mult* molto più alti. Questa scelta è motivata dal fatto che si vuole fare in modo che i primi layer, i quali teoricamente possiedono già degli ottimi parametri in quanto allenati in un dataset molto grande e complesso come ad esempio

ImageNet, modifichino molto lentamente i loro parametri per il nuovo dataset, mentre per quanto riguarda l'ultimo layer, quello rinominato, è come se fosse un nuovo layer e quindi non possiede dei propri parametri importati dal modello specificato nello script precedente e deve perciò svolgere molto più lavoro rispetto agli altri layer. Ci si potrebbe chiedere perchè è necessario avere come ultimo layer uno nuovo. La spiegazione è semplice: nell'ultimo layer avente parametri allenabili viene specificato il numero di output di tale layer e questo di norma corrisponde al numero di classi all'interno del proprio dataset; l'ultimo layer normalmente dà in output un valore per ciascuna delle le classi disponibili nel dataset e quindi l'intera operazione è da rifare nel caso si possegga un dataset con un numero di classi diverso da quello della rete originale su cui si sta effettuando il fine-tuning. Ci si potrebbe inoltre chiedere se è possibile definire altri nuovi layer, oltre a quello finale, per la rete selezionata. La risposta è si, ma solamente se rispettano la stessa architettura della rete. In conclusione è potenzialmente possibile rinominare tutti i layer che si vuole, ma non si può ad esempio aggiungerne o rimuoverne uno all'interno della rete lasciando inalterati gli altri. Ciò comporterebbe una sostanziale modifica dell'architettura della rete, oltre al fatto di dover sistemare i parametri di tutti gli altri layer perchè tornino i conti con i volumi nella rete; quindi quando Caffe andrà a caricare il file *.caffemodel*, il quale è stato creato secondo lo schema precedente della rete, verrà generato un errore. Invece semplicemente rinominando un layer lo si esclude dal caricamento dei rispettivi parametri per il fine-tuning: Caffe in pratica controlla quali layer hanno lo stesso nome delle rete precedente, per i quali carica i dati, e quali invece no, di cui non carica i dati. E' in teoria possibile invece, anche se nessuno a quanto fare lo fa, lasciare inalterati una serie dei primi layer della rete selezionata e creare un'intera seconda parte della rete, modificando questa volta anche una piccola parte dell'architettura, ma solitamente non conviene fare questo, perchè si avrebbe comunque una buona parte della rete da allenare da zero e la quale sarebbe comunque condizionata da come è costruita la prima parte della rete non modificata. E' più conveniente in questo caso crearsi una propria rete e allenarla da zero. Uno dei più famosi lavori di fine-tuning fatto con Caffe si chiama *Flickr Style* ([23]): partendo dalla rete [16], è stato considerato un sottoinsieme delle immagini del dataset ImageNet di circa 80000 immagini; le immagini in questione sono state accuratamente selezionate ed organizzate in 20 classi (ImageNet ne possiede 1000), dove ciascuna di questa classe rappresenta uno stile di immagini, come ad esempio vintage, malinconica, minimale ecc. Questo particolare caso è un esempio di fine-tuning applicato ad un dataset tutt'altro che semplice, infatti l'accuracy finale non risulta essere altissima. Eppure solo con qualche migliaio di iterazioni si sono ottenuti degli ottimi risultati. Per ottenere gli

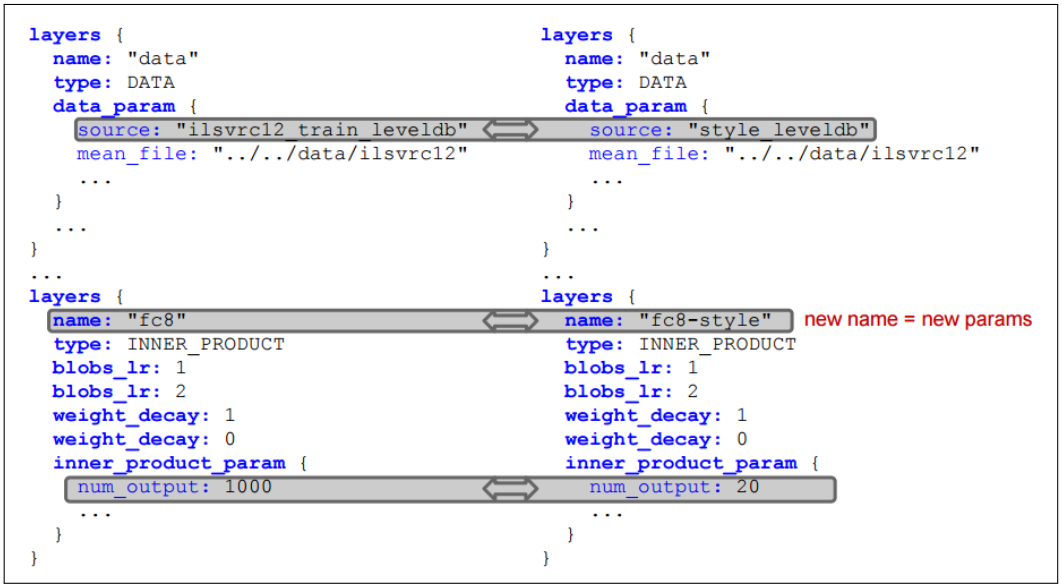


Figura 4.35: Modifiche nella definizione di una rete per effettuare un fine-tuning

stessi risultati allenando una rete da zero probabilmente ci sarebbero volute decine, anche centinaia, di migliaia di iterazioni in totale. In figura 4.35 viene riassunto quanto detto finora, applicato nel caso del Flickr Style.

4.7.2 Modifiche al solver

Terminate le modifiche del file dove era definita la rete originale è necessario ora modificare anche il file originale *solver.prototxt* della medesima rete. Solitamente effettuare un'operazione di fine-tuning di una rete richiede molto meno tempo, e quindi molte meno iterazioni, rispetto ad un normale allenamento. Di conseguenza vanno sicuramente abbassati i relativi parametri (*test_interval*, *max_iteration*, *snapshot* e relativo percorso, ecc.). Va inoltre abbassato anche il learning rate iniziale (*base_lr*), in quanto la rete ha molte meno cose da apprendere e meno iterazioni per farlo. Anche il parametro *test_iter* va modificato in base alla formula vista nel paragrafo 4.5. Infine se il file di *train_val.prototxt* è stato rinominato, spostato o altro è necessario specificare il percorso del file creato in precedenza nel campo *net*. I rimanenti parametri solitamente si lasciano inalterati. Terminate anche le modiche del solver è ora possibile procedere con il fine-tuning, dando il comando specificato all'inizio di questa sezione.

4.7.3 Il model zoo

Per l'operazione di fine-tuning può essere molto utile consultare l'intera sezione di GitHub che Caffe ha creato al fine di condividere i modelli delle reti create da vari utenti, chiamata Model Zoo (`https://github.com/BVLC/caffe/wiki/Model-Zoo`). In questo "zoo dei modelli" vengono resi disponibili tutti i modelli più famosi e performanti, insieme ad alcuni dei file necessari per il fine-tuning; a volte non ci sono tutti, ma l'importante è che ci sia il modello risultante della rete; gli altri file si possono creare nel caso mancassero. Tramite il Model Zoo, i ricercatori e gli ingegneri possono dunque condividere i loro lavori. Non tutti i problemi nello zoo riguardano casi di classificazione; c'è un po' di tutto: casi di classificazioni in larga scala o meno, casi semplici di regressione (concetto di Data Mining dove il dominio di un problema di classificazione non è discreto), applicazioni per la robotica e altro ancora, spaziando su tutto ciò che Caffe permette di fare. Chiunque può quindi cercare un modello che gli interessa e, nel caso lo trovasse, lo può scaricare ed utilizzare per i propri scopi. Analogamente è possibile caricare un proprio modello per un dataset specifico.

4.8 Consigli durante il monitoraggio dell'allenamento di una rete

Una volta avviato l'allenamento, o il fine-tuning, di una rete, dopo un determinato numero di iterazioni specificato dal parametro *display* del solver, sul terminale appariranno le informazioni che mostrano il valore attuale della funzione di loss. Il normale e corretto andamento di questi valori dovrebbe essere quello in cui la funzione di loss, all'aumentare delle iterazioni, diminuisce il proprio valore. Non è detto che migliori nell'arco di poche iterazioni (anzi a volte può anche leggermente aumentare), ma l'importante è che complessivamente dopo ad esempio qualche centinaia di iterazioni si abbia un valore di loss un po' più basso di quello di partenza. Se ciò non accade, vuol dire che qualcosa non va: nel migliore dei casi potrebbe essere semplicemente dovuto al fatto che alcuni parametri non sono stati settati in maniera adeguata; nel peggiore dei casi potrebbe essere un problema che riguarda il dataset costruito, ad esempio troppe poche informazioni, immagini di scarsa qualità o altro ancora. In questo caso bisogna intervenire direttamente sul dataset. Nel primo dei due casi si può tentare di procedere in uno dei seguenti modi (non sempre funzionano, ma spesso possono aiutare):

- se il valore di loss invece di diminuire aumenta, provare ad abbassare il valore di base del learning rate del solver. In alternativa incrementare

il suo fattore di diminuzione sempre nel solver o in casi estremi anche cambiare il suo tipo di policy (ultima cosa da fare dopo aver provato di tutto). Se ci si sente abbastanza sicuri si può anche tentare di manovrare il valore del learning rate direttamente all'interno di ciascun layer (molto utile nel caso di un fine-tuning) andando ad incrementare i parametri di *lr_ mult*.

- se il valore di loss ad un certo punto non decresce più, oppure decresce ma molto e troppo lentamente, provare in questo caso ad aumentare il learning rate, sempre con una delle metodologie descritte nel punto precedente, ovviamente nel caso opposto stavolta.

4.9 Classificazione di un'immagine con Caffe

Terminato l'intero processo di allenamento (o di fine-tuning) si avrà dunque a disposizione un file *.caffemodel* necessario per la successiva classificazione di immagini rientranti nelle categorie del dataset, ma che non sono presenti in esso. Per questo scopo Caffe mette a disposizione un piccolo programma scritte in C++ chiamato *classification.cpp*. L'eseguibile di questo programma prende in input i seguenti parametri:

- il protocol buffer ***deploy.prototxt*** che verrà illustrato fra poco.
- il modello risultante dall'allenamento della rete (***.caffemodel***).
- il file binario per l'operazione di mean subtraction. In questo caso è obbligatorio per come è stato scritto il piccolo programma di classificazione, ma con opportune modifiche si può fare in modo di farne a meno. Caffe lo ha probabilmente reso obbligatorio di base perchè spesso quest'operazione aiuta molto e non appesantisce troppo il programma.
- il file ***synsetwords.txt*** per avere i nomi e le label da assegnare alle varie classi
- il percorso dell'immagine da classificare.

Il file *deploy.prototxt* serve solo nelle predizioni di immagini esterne: in pratica è un file quasi uguale al file *train_ val.prototxt* (mancano alcuni parametri che in questo caso non sono necessari perchè non si sta allenando una rete) che serve per estrarre i dati, secondo la struttura della rete, dal suo modello binario e con essi effettuare una successiva classificazione. Nella sua creazione è dunque necessario prestare attenzione che nei due file il nome, la

```
layer {
  name: "prob"
  type: "Softmax"
  bottom: "fc8"
  top: "prob"
}
```

Figura 4.36: Esempio di un softmax layer nel file di deploy

loro tipologia, i rispettivi parametri di ciascun layer ed il loro ordine siano esattamente gli stessi, fatta eccezione per i layer qui sotto descritti. Le uniche cose che si possono cambiare sono:

- il tipo di data layer utilizzato, purchè effettui le stesse operazioni di resize o prenda una crop delle stesse dimensioni del data layer con cui la rete è stata allenata. E' sufficiente inserire un parametro relativo appunto ad uno di questi due dati e si può tranquillamente usare un altro data layer.

- il tipo di output layer. In questo caso non ha senso calcolare un'accuracy o una loss. Si utilizza un semplice layer di output, come ad esempio un layer Softmax (mostrato in figura 4.36), che calcoli e distribuisca le probabilità fra le classi specificate. Si noti che in questo caso il layer di output non prende in input una blob con le label: infatti le label sono in questo caso caricate esternamente dal file di synset e vengono utilizzate, dal codice di Caffe, solo alla fine per l'output su terminale. Qui infatti non c'è un training set o un validation set: la vera classe dell'immagine da classificare è ignota. Il file di synset, contenente le label, serve solamente per l'output della classe predetta dalla rete.

In realtà, esattamente come era stato visto con il fine-tuning, nel file di deploy si possono omettere interamente alcuni layer a partire dalla fine della rete; non è invece possibile escludere alcuni layer interni in quanto si distruggerebbe la struttura della rete stessa e Caffe segnalerebbe un errore proprio perchè i valori dei parametri non risulterebbero più validi per poter effettuare una valida forward pass della rete. Eliminando invece ad esempio solo l'output layer finale, è comunque possibile effettuare una forward pass, solo che ovviamente cambierà l'output finale, che risulterà essere quello dell'ultimo layer presente nella rete. Si ricorda che qui non ci sono backward pass e quindi anche se manca un layer alla fine della rete, essa effettua comunque la sua normale elaborazione dell'immagine fino alla fine della sua sequenza di layer.

Anche in questo caso non è possibile aggiungere nuovi layer che modifichino la sostanziale architettura della rete, cosa che ad esempio un loss layer, o un data layer diverso, ma con gli stessi parametri di dimensioni delle immagini, infatti non fanno. In un prossimo futuro sembra che Caffe unirà il file di deploy a quello di training, esattamente come in precedenza era stato fatto per le fasi di training e di testing durante l'allenamento: si è passati da *train.prototxt* e *val.prototxt* a *train_ val.prototxt* e in futuro si passerà dunque a *train_ val_ deploy.prototxt*, aggiungendo un qualche tipo di parametro che distingua l'allenamento dalle predizioni esterne, esattamente come era stato raccontato all'inizio della sezione 4.6 con il parametro *phase*; basterebbe anche solo aggiungere una possibilità all'insieme di valori che questo parametro può assumere.

Per chiunque avesse fatto un'operazione di fine-tuning si può procedere in questo caso in due modi: se non si possiede il file di deploy è necessario crearlo da zero, rispettando i vincoli precisati sopra; se invece si dispone già anche di un file di deploy è sufficiente modificare in esso il nome della rete, il nome dell'ultimo layer con parametri allenabili ed il suo numero di output (il numero di classi), facendo attenzione a mettere gli stessi valori messi nel file di training, altrimenti verrà generato un errore in fase di caricamento del modello. Quando si avrà un file di deploy completo è importante ricordarsi di aggiungere anche qui i vari parametri di *lr_ mult* e *decay_ mult*. Questi parametri devono essere gli stessi utilizzati nel file di training della rete perchè, anche se in questo caso non si hanno delle backward propagation, i parametri letti dal modello devono avere gli stessi valori di quando sono stati utilizzati durante il training, ergo devono essere moltiplicati per gli stessi fattori, altrimenti si avrebbero numeri diversi. Quest'ultima regola non riguarda solo il fine-tuning, ma la costruzione generale del file di deploy.

Una volta predisposto tutto è sufficiente dare da terminale, sempre dall'interno della cartella di Caffe, un comando simile al seguente:

```
./build/examples/cpp_classification/classification.bin
/home/fabio/Scrivania/Reti/Googlenet/deploy.prototxt
/home/fabio/Scrivania/Risultati_test/Googlenet/modelli/_iter_2000.caffemodel
data/ilsvrc12/imagenet_mean.binaryproto /home/fabio/Scrivania/Reti/synset_words.txt
/home/fabio/Scrivania/dataset/consenso_informato/val/cons_train_1.jpg
```

In figura 4.37 viene mostrato l'output del suddetto comando.

```
---------- Prediction for /home/fabio/Scrivania/cons_train_1.jpg ----------
0.9999 - "0 consenso-informato"
0.0001 - "3 modulo-730"
0.0000 - "5 triangolo-operatore"
0.0000 - "2 ecomappa"
0.0000 - "4 triangolo-bambino"
```

Figura 4.37: Esempio output predizione immagine esterna con Caffe

Capitolo 5

Analisi delle CNN più avanzate

In questo capitolo verranno osservate nel dettaglio le reti neurali convoluzionali più famose ed avanzate del momento. Il loro studio si è rilevato fondamentale per portare avanti il lavoro, dato che come si è visto nel paragrafo 1.2.3 è stato scelto di utilizzare la tecnica del fine-tuning come metodo di allenamento di una rete neurale convoluzionale. Le motivazioni verranno illustrate sempre nel suddetto capitolo. La tecnica del fine-tuning prevede di utilizzare un modello pre-allenato di una rete esistente e questo richiede appunto un'analisi per selezionare le migliori reti adatte allo scopo. In questo capitolo verranno mostrate le architettue generali delle reti studiate, con infine un confronto delle performance di accuratezza delle predizioni che queste reti hanno registrato in questi ultimi anni su uno dei dataset più complessi attualmente esistenti, chiamato ImageNet.

Il capitolo successivo si focalizzerà invece su altre considerazioni, non riguardanti le performance di classificazione.

Le reti qui considerate verranno più volte riprese nel corso di tutti i capitoli successivi. Il loro studio si è inoltre rilevato utile per capire le strategie utilizzate in questi ultimi anni per migliorare ed ottimizzare sempre di più le accuracy di questi potenti strumenti.

5.1 AlexNet

La rete AlexNet [16] è una fra le prime reti neurali convoluzionali che ha riscosso un enorme successo. Vincitrice della competizione ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) del 2012, questa rete è stata la prima ad ottenere risultati più che buoni su un dataset molto complicato come ImageNet, utilizzando una sorta di standard per la costruzione di reti neurali convoluzionali che la vecchia rete LeNet-5 aveva definito, già ac-

cennato nel relativo paragrafo 3.3: la struttura classica di una rete neurale convoluzionale prevede una serie di layer convoluzionali seguiti da un'altra serie di FC layer. Vediamo ora più specificamente la struttura della rete.

5.1.1 Architettura di AlexNet

Qui vi sono 8 layer con parametri allenabili: una serie di 5 layer convoluzionali seguita da 3 FC layer. Ogni layer convoluzionale è seguito da un layer ReLU e opzionalmente anche da un layer di MAX pooling, soprattutto all'inizio della rete, per ridurre le dimensioni spaziali dei volumi. Tutti i layer di pooling hanno regione di estensione 3 x 3 ed uno stride pari a 2: ciò vuol dire che si utilizza sempre un overlapping pooling. Tale scelta è dovuta al fatto che questo tipo di pooling incrementa leggermente le prestazioni della rete rispetto al normale pooling senza overlapping.

Sempre nella parte iniziale della rete, fra un layer di pooling e il successivo layer convoluzionale, sono stati utilizzati un paio di layer di normalizzazione LRN: dopo alcuni test è stato visto che essi tendono a diminuire l'errore della rete.

Dei 3 FC layer, i primi due posseggono 4096 neuroni, mentre l'ultimo possiede 1000 unità, corrispondenti al numero di classi presenti sul dataset di ImageNet. Dato il numero enorme di connessioni possedute dai FC layer, in mezzo ad ogni coppia di essi è stato aggiunto un dropout layer con un ratio 0.5, cioè metà delle attivazioni dei neuroni ogni volta non viene considerata. E' stato notato in questo caso che l'utilizzo della tecnica di dropout, non solo velocizza l'elaborazione di una singola iterazione, ma permette di prevenire piuttosto bene l'overfitting della rete. Senza il dropout i creatori della rete sostengono che la rete originale avesse un overfitting troppo alto.

Ovviamente, come in tutte le reti di Caffe, sono presenti un data layer, un loss layer ed un layer di accuracy. Nell'anno della sua creazione il data layer accettava in ingresso immagini 256 x 256 (si effettua come al solito un'operazione di resize) e come dimensioni di crop aveva 224 x 224. Successivamente la rete è stata leggermente modificata, in quanto le dimensioni di crop sono adesso 227 x 227, modificando dunque leggermente i parametri successivi nella rete per far tornare i conti (probabilmente solo lo zero-padding del primo layer convoluzionale perchè tutte le altre dimensioni risultano essere uguali alla versione precedente).

5.1.2 Esempio di forward propagation con AlexNet

Si vedrà ora nel dettaglio una singola fase di forward propagation per rendere l'idea della moltitudine di operazioni all'interno di una rete neurale convolu-

zionale che in questo caso risulta anche essere architetturalmente piuttosto semplice. Infatti per le successive reti che verranno presentate, sarà omessa questa parte, dato che le cose si complicano abbastanza e si rischierebbe di creare troppa confusione con i conteggi. Ovviamente è sufficiente verificare direttamente le dimensioni di un modello finale di una rete per sapere le sue precise dimensioni, cosa che è stata fatta con le reti più complesse, ma avendo in questo caso un'architettura semplice può essere utile capire come si arriva ad avere un file del modello di determinate dimensioni.

Una singola fase di forward propagation di AlexNet implica dunque le seguenti principali operazioni: il primo layer convoluzionale accetta in input un'immagine 227 x 227 x 3 e ad essa applica ben 96 kernel, ognuno di dimensioni 11 x 11 x 3 con uno stride di 4 e nessun zero-padding, ottenendo un volume 55 x 55 x 96 (per i conti si riguardi paragrafo 3.2.1 e tutte le successive sottosezioni interne ad esso).

Il secondo layer convoluzionale prende in input un volume a cui sono state applicate le funzioni di ReLU, normalizzazione ed anche un overlapping pooling; l'ultima di queste operazioni trasforma il volume precedente in uno di dimensioni 27 x 27 x 96 (si veda paragrafo 3.2.3). Tale volume viene dato in input al secondo layer convoluzionale che lo convolve con 256 kernel di dimensioni 5 x 5 x 96 ed uno stride pari a 1 e uno zero-padding di 2, ottenendo un volume 27 x 27 x 256 (si noti che uno stride di 1 in pratica lascia inalterate le dimensioni spaziali, aumentando solo la profondità dovuta al numero di kernel). Successivamente, dopo le operazioni di ReLU e di normalizzazione, viene applicato un altro layer di pooling identico al precedente e si ottiene un volume 13 x 13 x 256.

Il terzo layer convoluzionale possiede 384 kernel con un receptive field 3 x 3 x 256 ed uno stride di 1 ed uno zero-padding di 1: risulta un volume di 13 x 13 x 384. Questa volta non c'è un altro pooling e nemmeno una normalizzazione, bensì solo un layer ReLU che non intacca le dimensioni.

Il quarto layer convoluzionale possiede altri 384 kernel di dimensioni 3 x 3 x 384, sempre con stride 1 ed uno zero-padding di 1: si ottiene un volume uguale a quello di prima, ovvero 13 x 13 x 384. Anche stavolta, dopo il layer convoluzionale, c'è solo un'operazione di ReLU.

Il quinto layer convoluzionale ha 256 kernel di dimensioni 3 x 3 x 384, con stride 1 e zero-padding 1: il volume risultante ha dimensioni 13 x 13 x 256. Questa volta oltre ad un layer ReLU vi è successivamente anche uno dei soliti layer di pooling che trasforma il volume di prima in uno di taglia 6 x 6 x 256.

A questo punto il primo dei 3 FC layer, possedente 4096 neuroni, effettua il suo normale lavoro, cioè i vari prodotti e somme per ottenere le attivazioni

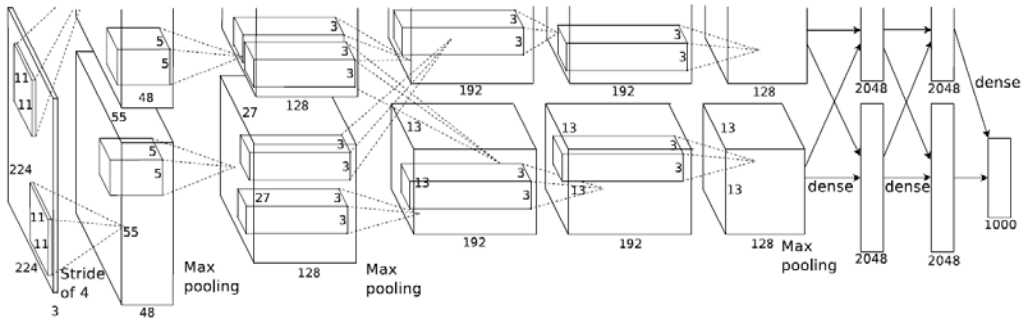


Figura 5.1: Architettura di AlexNet

dei suoi 4096 neuroni, ottenendo un volume (anche se in realtà ha solo una dimensione) 1 x 1 x 4096.

Analoga cosa con il secondo layer FC, avente sempre 4096 unità, e successivamente con il terzo ed ultimo layer FC, il quale possedendo solo 1000 unità, il numero di classi nel dataset, produce in output un vettore delle stesse dimensioni, cioè 1 x 1 x 1000.

Se si guarda tuttavia il paper di AlexNet [16], si nota che le cose in realtà non sono esattamente come descritte sopra, anche se sono comunque equivalenti. L'architettura della rete è stata in questo caso divisa in due rami perchè nei vari test sono state utilizzate due GPU in parallelo. Specificando il parametro *group*, in questo caso pari a 2, è teoricamente possibile con Caffe suddividere il lavoro in più unità GPU dimezzando la profondità di ciascun volume, cioè il numero di depth slice considerati, esattamente come mostrato in figura 5.1.

Ciò però comporta alcuni rischi, dovuti al fatto che la comunicazione fra i due rami di elaborazione delle due GPU non avviene sempre, perchè se fosse così non converrebbe utilizzare questo approccio, ma solo in determinati punti della rete. Nello specifico solo i kernel del terzo layer convoluzionale sono connessi a tutte le feature create dal secondo; tutti gli altri layer convoluzionali sono collegati solamente alle feature dei layer precedenti presenti nella loro GPU locale. Quindi con questo metodo si ottiene sicuramente un grande aumento di velocità computazionale, ma tuttavia si può perdere un po' di generalità, in quanto le feature non sono sempre analizzate tutte insieme. Oltre a questo è ovviamente necessario sviluppare un apposito codice, CUDA in questo caso, adatto a questo scopo, dato che a quel tempo ancora non esisteva. Nonostante tutto ciò, i risultati non sono comunque andati male.

La rete originale eseguiva esattamente le operazioni descritte accuratamente prima; tuttavia si è poi visto che la rete risultava troppo pesante da

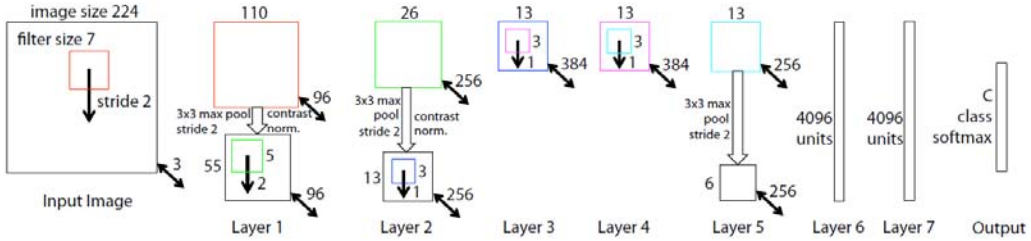


Figura 5.2: Rete AlexNet densa

allenare e quindi si è optato per un aumento del parallelismo come appena visto. La limitazione più grande derivava probabilmente dalla quantità di memoria che una GPU possedeva 3 anni fa (nei test erano state utilizzate due NVIDIA GTX 580 aventi 3GB di memoria ciascuna) rispetto alla quantità di dati che con la loro rete si voleva processare di volta in volta per avere tempi abbastanza ragionevoli.

Avviando la rete in modalità CPU non sono sicuro del fatto che Caffe suddivida il lavoro fra i vari processori presenti, assumendo che nel computer per effettuare il training ce ne sia più di uno. Ad ogni modo è sufficiente considerare la rete originale, senza parallelismo, per evitare eventuali problemi: il training risulterà un po' più lungo, ma i risultati saranno comunque gli stessi, anzi teoricamente si dovrebbero avere feature migliori avendo sempre a disposizione tutti i dati sulla stessa memoria. Spesso infatti anche altri team o utenti utilizzano una versione di AlexNet "densa", ovvero senza la partizione in due rami per le due GPU vista in precedenza, come mostrato in figura 5.2.

Come si può notare alcuni parametri mostrati in figura 5.2 non sono esattamente tutti gli stessi rispetto a quanto detto finora. Questo perchè la figura rappresenta una vecchia versione di AlexNet. Col passare del tempo tale versione è stata leggermente migliorata, apportando proprio alcune modifiche ai parametri della rete. All'interno di questa tesi quando si parlerà di AlexNet ci si riferirà sempre all'ultima versione attualmente disponibile, la più accurata, che è quella con i parametri illustrati finora all'interno di questo paragrafo.

5.1.3 Esempio di calcolo dimensioni modello di AlexNet

Le dimensioni di un modello (file *.caffemodel*) di una rete derivano dalla quantità di parametri allenabili che essa possiede. Verrà mostrato nel dettaglio come, una volta progettata una rete, calcolare all'incirca le dimensioni del modello risultante. Esattamente come per la sottosezione precedente, questo

tipo di calcoli viene mostrato solo in questo caso perchè la rete in questione ha un'architettura relativamente semplice. Viene invece omesso per le reti che si vedranno dopo, in quanto la loro architettura potrebbe rendere questi calcoli complicati e confusionari, oltre al fatto che il procedimento sarebbe in generale sempre lo stesso e quindi diventerebbe tutto ripetitivo.

Vediamo dunque i dettagli del calcolo:

- il primo layer convoluzionale applica 96 kernel di dimensioni 11 x 11 x 3. Si avranno dunque 96*11*11*3 weight e 96 bias, per un totale di 34,944 parametri.

- il secondo layer convoluzionale applica 256 kernel di dimensioni 5 x 5 x 96. Dunque si hanno 256*5*5*96 weight e 256 bias, per un totale di 614,656 parametri.

- il terzo layer convoluzionale applica 384 kernel di dimensioni 3 x 3 x 256. Quindi 384*3*3*256 weight e 384 bias, per un totale di 884,736 parametri.

- il quarto layer convoluzionale applica 384 kernel di dimensioni 3 x 3 x 384. Quindi 384*3*3*384 weight e 384 bias, per un totale di 1,327,488 parametri.

- il quinto layer convoluzionale applica 256 kernel di dimensioni 3 x 3 x 384. Risultano 256*3*3*384 weight e 256 bias, per un totale di 884,992 parametri.

- il primo layer FC possiede 4096 unità e viene collegato ad un volume 6 x 6 x 256. Risultano 4096*6*6*256 weight e 4096 bias, per un totale di 37,752,832 parametri

- il secondo layer FC possiede 4096 unità e viene collegato ad un volume 1 x 1 x 4096. Risultano 4096*1*1*4096 weight e 4096 bias, per un totale di 16,781,312 parametri

- il terzo layer FC possiede 1000 unità e viene collegato ad un volume 1 x 1 x 4096. Risultano 1000*1*1*4096 weight e 1000 bias, per un totale di 4,097,000 parametri

Come si può notare si ha un numero enorme di parametri allenabili, in totale 62,377,960 (più di 62 milioni di parametri), di cui più di 58 milioni (58,631,144 per l'esattezza) di essi derivanti solo dai 3 FC layer. Se si moltiplica il numero totale di parametri trovato per un fattore 4, dimensioni in

byte di un float (ringrazio Michael Figurnov che su Caffe Users mi ha ricordato di aggiungere questo fattore al calcolo), otteniamo il numero di Byte che in totale il modello di AlexNet occupa in memoria, ovvero 249,511,840. Dividendo tale numero per 1024*1024 si ottiene ovviamente il medesimo valore in MegaByte e cioè più o meno 238 MB, di cui circa 224 MB dai FC layer e i rimanenti 14 MB dai layer convoluzionali.

Risulta quindi ovvio che in generale i FC layer sono i layer che possiedono il più grande numero di parametri allenabili, nonostante la loro funzione sia comunque più semplice rispetto a quella di un normale layer convoluzionale.

5.2 Reti VGG

VGG [19] è il nome di un team di persone che hanno presentato le proprie reti neurali durante la competizione ILSVRC-2014. Si parla di reti al plurale in quanto sono state create più versioni della stessa rete, ciascuna possedente un numero diverso di layer. In base al numero di layer n con weight che una di queste reti possiede, ognuna di esse viene solitamente chiamata *VGG-n*. Esistono poi altre versioni delle reti VGG menzionati in Ken Chatfield et al. [21], ma verranno tralasciate in quanto molto simili.

Un particolare comune a tutte le versioni è il seguente: tutte queste reti risultano essere più "profonde" rispetto a quella di AlexNet. Per "profonde", si intende il fatto che sono costituite da un numero di layer con parametri allenabili maggiore di AlexNet, in questo caso da 11 a 19 layer allenabili in totale. Spesso si considerano solo i layer allenabili dato che sono quelli che incidono di più nell'elaborazione e nelle dimensioni del modello, come visto nel paragrafo precedente. La struttura complessiva rimane tuttavia molto simile: sono sempre presenti una serie iniziale di layer convoluzionali ed una serie finale di FC layer, quest'ultima esattamente uguale a quella di AlexNet. Ciò che cambia dunque è il numero di layer convoluzionali utilizzati, ed ovviamente i loro parametri. La figura 5.3 illustra tutte le varianti costruite dal team.

Ogni colonna, partendo da sinistra e andando verso destra, mostra una determinata rete VGG, dalla meno profonda alla più profonda. I termini in grassetto mostrano cosa è stato aggiunto in ciascuna versione rispetto a quella precedente. Non sono mostrati nella figura i ReLU layer, ma nelle reti ne esiste uno dopo ogni layer convoluzionale, come sempre.

Tutti i layer di pooling effettuano un pooling senza overlapping, cioè con estensione 2 x 2 e stride 2, ed utilizzano la funzione di MAX nelle varie selezioni.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figura 5.3: Le versioni delle reti VGG

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figura 5.4: Totale dei parametri delle reti VGG (in milioni)

Tutti i layer convoluzionali utilizzano uno stride pari a 1. Si noti il fatto che rispetto ad AlexNet non vi sono layer convoluzionali con un receptive field piuttosto grande: qui tutti i receptive field hanno dimensioni 3 x 3, fatta eccezione per un paio di layer convoluzionali in VGG-16 aventi un receptive field 1 x 1. Si ricorda che un layer convoluzionale avente stride pari ad 1 non modifica le dimensioni spaziali dell'input, mentre modifica il valore della profondità che diventa uguale al numero di kernel utilizzati. Dunque in pratica i layer convoluzionali delle VGG non intaccano mai le dimensioni di larghezza e altezza dei volumi di input; solo i layer di pooling lo fanno.

L'idea di utilizzare una serie di layer convoluzionali con un receptive field più piccolo, che comunque complessivamente alla fine simulano un singolo layer convoluzionale con un receptive field più grande, è motivata dal fatto che in questo modo si utilizzano più layer ReLU invece di uno solo, incrementando dunque maggiormente la non linearità della funzione di attivazione e rendendola dunque più discriminante; inoltre serve per ridurre il numero di parametri utilizzati. Si consideri ad esempio una serie di 3 layer convoluzionali con receptive field 3 x 3 x C, con C kernel in totale per ognuno: l'ammontare dei parametri per questa serie è pari a $3 * (3 * 3 * C * C) = 27C^2$. Con un singolo layer convoluzionale avente C kernel, ognuno di dimensioni 7 x 7 x C, come quello di AlexNet, si avrebbero invece $1 * (7 * 7 * C * C) = 49C^2$ parametri, molti di più rispetto a prima.

L'utilizzo invece dei layer convoluzionali 1 x 1 con stride 1 è una strategia per aumentare anche in questo caso la non linearità della funzione di attivazione senza intaccare le dimensioni spaziali dei volumi e nemmeno la dimensione della profondità dato che ciascuna serie di layer convoluzionali ha lo stesso numero di kernel. Di fatto dunque le dimensioni dei volumi rimangono esattamente le stesse fino all'applicazione di un layer di pooling; poi si procede con una nuova serie di layer convoluzionali con un numero maggiore di kernel. L'aumento della non linearità deriva dal fatto che dopo ogni layer convoluzionale c'è sempre un layer ReLU.

Nonostante forse si possa pensare che questa rete, con i vari stratagemmi visti, non possegga molti parametri, la tabella in figura 5.4 dice il contrario: essa mostra per ciascuna versione il numero totale approssimato di parametri allenabili posseduti (in milioni).

Queste reti vengono considerate una sorta di evoluzione di AlexNet, in

quanto, complessivamente e a parità di dataset, hanno prestazioni superiori in confronto ad essa. Il concetto principale dimostrato con le reti VGG (e non solo) è che più una rete neurale convoluzionale è profonda e più le sue prestazioni aumentano. E' necessario però disporre di un hardware sempre più potente, altrimenti l'allenamento di una rete diventerebbe improponibile. Per le VGG sono state utilizzate ben 4 NVIDIA Titan Black con 6 GB di memoria ciascuna. Le VGG hanno dunque prestazioni migliori, ma necessitano di una notevole potenza hardware per l'allenamento ed inoltre utilizzano un numero di parametri molto elevato (figura 5.4): il modello ottenuto da VGG-19 ad esempio pesa circa 550 MB (dimensioni raddoppiate rispetto ad AlexNet). Le reti VGG più piccole hanno comunque un modello di circa 507 MB.

5.3 Network In Network

Ormai si dovrebbe avere intuito che in una rete neurale convoluzionale vi sono dei layer convoluzionali che effettuano dei prodotti fra i loro kernel ed i receptive field dell'input e che l'intero processo è sempre seguito da una funzione di attivazione non lineare (layer ReLU) prima del successivo layer convoluzionale. Questo perchè i filtri , cioè i kernel, dei layer convoluzionali sono lineari. Gli autori delle rete Network In Network (NIN) [17], che verrà ora illustrata, sostengono che l'utilizzo di un normale filtro convolutivo, il quale si può considerare un modello lineare generalizzato, abbia un livello di astrazione basso, cioè non rappresenti al meglio il classico sistema biologico a cui le reti neurali si ispirano. In pratica una convoluzione lineare è sufficiente, per quanto riguarda l'astrazione, quando gli input sono linearmente separabili. Tuttavia generalmente le rappresentazioni che ottengono un livello piuttosto alto di astrazione sono delle funzioni non lineari: dalla teoria classica delle reti neurali si ha infatti che la formazione di regioni decisionali complesse è possibile solo se la funzione di uscita dei neuroni è non lineare. Secondo questo team, sostituire il modello di convoluzione lineare con un sistema più potente di approssimazione ad una funzione non lineare può innalzare il livello di astrazione della rete. Nella rete NIN il modello finora preso in considerazione viene sostituito con una micro-rete (ecco il motivo del nome della rete stessa), la quale di fatto costituisce un sistema generale di approssimazione ad una funzione non lineare. Nello specifico come micro-rete è stato scelto in questo caso il multilayer perceptron. NIN inoltre sostituisce la serie di FC layer con un global average pooling, tecnica che verrà vista fra poco nel dettaglio. La figura 5.5 mostra le sostanziali modifiche che NIN ha apportato ad una classica CNN.

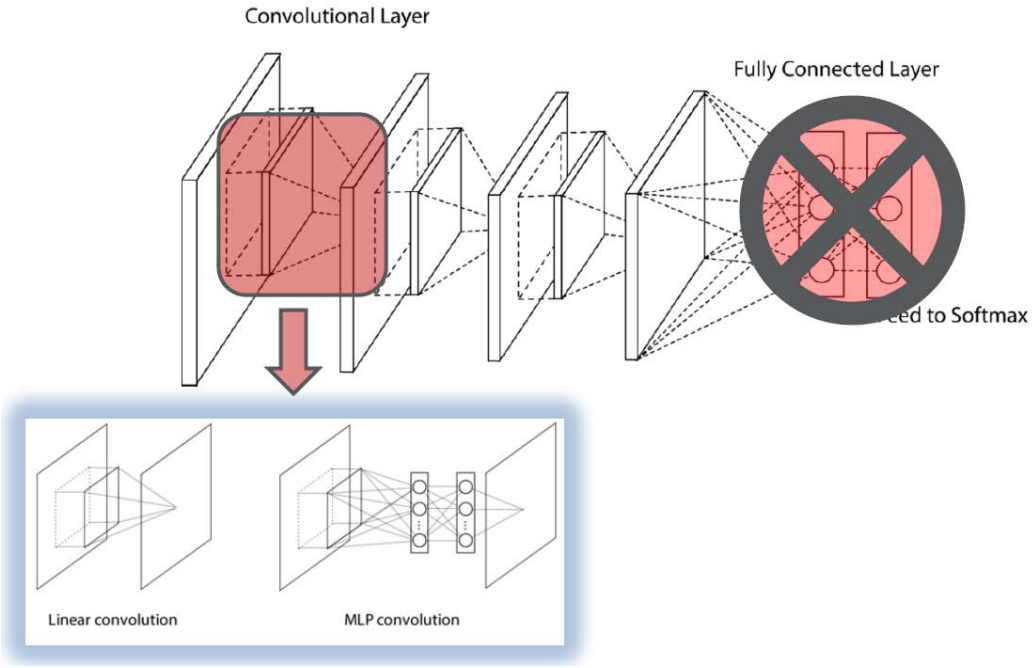


Figura 5.5: Le novità di NIN

5.3.1 Utilizzo del multilayer perceptron

Il multilayer perceptron (MLP) è una rete neurale artificiale di tipo feedforward che, esattamente come tutte le reti di questo tipo, mappa un insieme di dati di input su uno specifico insieme di valori di output. Un MLP ha la caratteristica di possedere più layer di neuroni ed inoltre ciascun layer è completamente connesso (Fully Connected) a quello successivo. Fatta eccezione per i neuroni di input, ogni neurone possiede una funzione di attivazione non lineare. Costituisce una variante più evoluta del linear perceptron, in quanto un MLP è in grado di distinguere dati non linearmente separabili. La figura 5.6 fornisce un esempio di MLP.

5.3.2 MLP Convolutional layer

Il MLP è stato scelto come micro-rete perchè esso rappresenta una funzione di approssimazione universale, ma soprattutto perchè è compatibile con la struttura generale di una rete neurale convoluzionale, la quale viene allenata tramite un algoritmo di backpropagation, cosa che fa anche il MLP. Questo nuovo tipo di layer viene chiamato MLP convolutional layer e potenzialmente dovrebbe sostituire i normali layer convoluzionali finora utilizzati. La figura 5.7 mostra graficamente come si modificate le cose: a sinistra viene mostrato

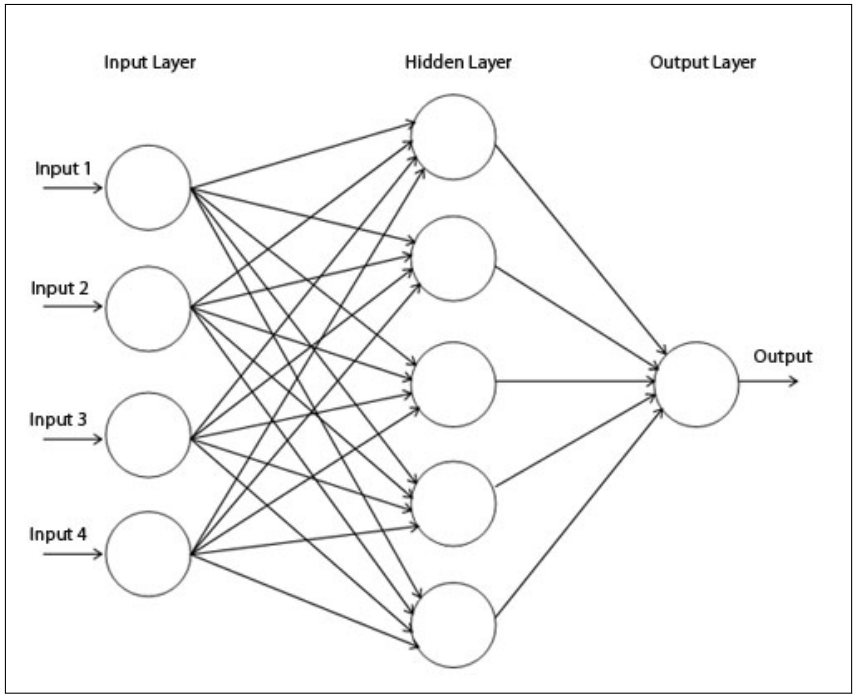


Figura 5.6: Esempio di MLP

un normale layer convoluzionale, che utilizza la normale convoluzione lineare, mentre a destra viene mostrato un MLP convolutional layer, che fa appunto uso di un MLP al posto di una convoluzione lineare.

In pratica è come passare da un calcolo del tipo:

$$f_{i,j,k} = max(w_k x_{i,j}, 0) \tag{5.1}$$

visto per un normale layer di ReLU, al seguente calcolo:

$$f_{i,j,k} = max(w_k x_{i,j} + b_k, 0) \tag{5.2}$$

Finora si sta però solo parlando di concetti teorici; nella pratica un MLP convolutional layer, è una composizione di più layer, definiti nel seguente ordine:

- un primo layer convoluzionale normale, con determinati valori di stride e receptive field,
- un primo layer di ReLU,
- un secondo layer convoluzionale con receptive field 1 x 1 e stride 1,
- un secondo layer di ReLU,

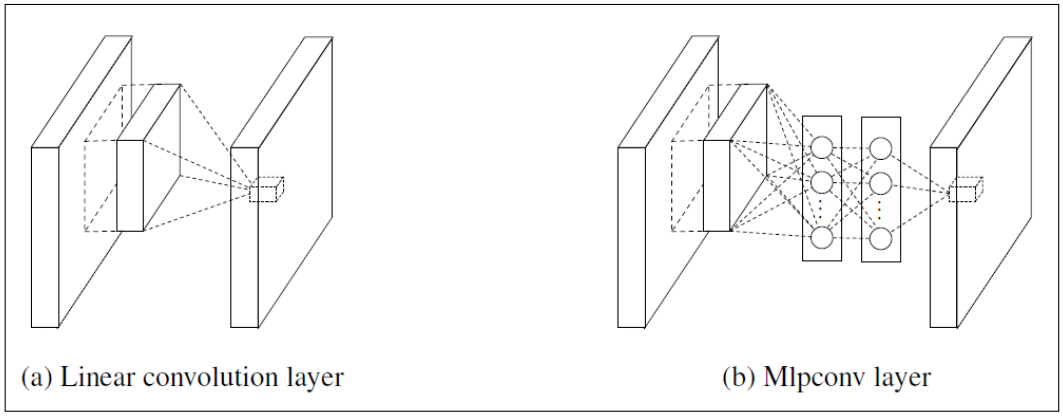


Figura 5.7: I due tipi di convoluzione

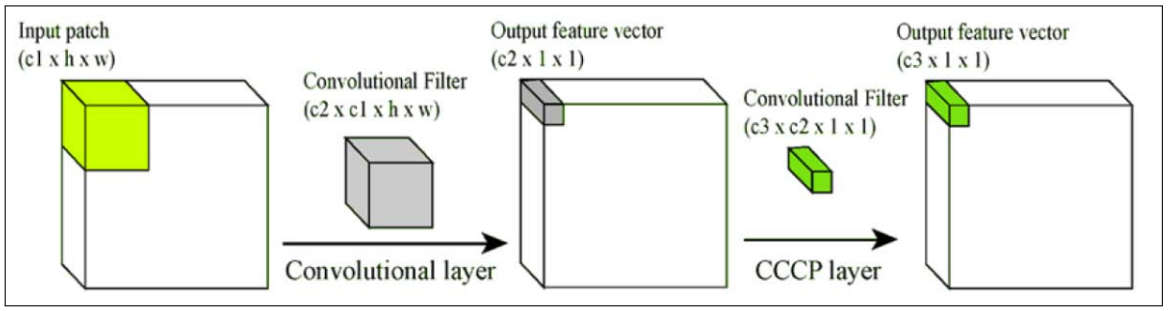


Figura 5.8: MLP convolutional layer nel dettaglio

- un terzo layer convoluzionale con receptive field 1 x 1 e stride 1,
- un terzo layer di ReLU,
- un layer di pooling con estensione spaziale 3 x 3, stride 2 e operazione di MAX (overlapping pooling).

Secondo il team questo modello, una serie di layer convoluzionali di questo tipo (1 x 1 con stride 1) con un pooling finale, costituisce una buona approssimazione ai concetti teorici finora citati.

L'ultimo layer di pooling dell'intera rete non è della stessa tipologia appena vista e verrà spiegato nel prossimo paragrafo.

I particolari layer convoluzionali aventi receptive field 1 x 1 e stride 1, vengono spesso etichettati con il nome di CCCP layer (Cascade Cross Channel Parametric layer). Nella rete NIN esistono sempre 2 CCCP layer fra un layer convoluzionale ed il successivo, ma il team argomenta che è potenzialmente possibile inserirne anche di più. Le figure 5.8 e 5.9 riassumono quanto detto finora.

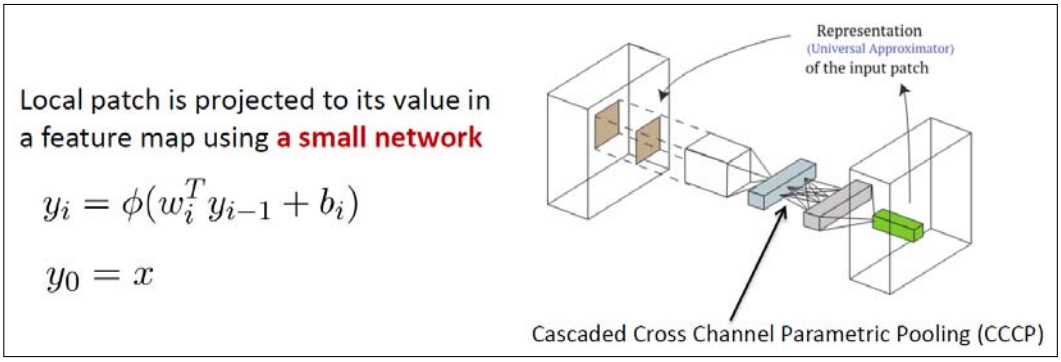


Figura 5.9: La micro-rete di NIN

5.3.3 Global average pooling

Già accennato nel paragrafo 3.2.3, questo tipo di pooling è stato introdotto per la prima volta proprio con la rete NIN. Il suo scopo principale è quello di sostituire i classici FC layer. L'idea è quella di generare una feature map, cioè una depth slice, per ciascuna categoria o classe nel dataset nell'ultimo MLP convolutional layer. Invece di utilizzare un FC layer si effettua un'operazione di media sui valori di ogni feature ed il risultante vettore viene direttamente dato in pasto al loss layer per il calcolo della funzione di loss o al Softmax layer per la predizione finale.

Nella pratica il global average pooling comporta l'utilizzo di un layer di pooling, con operazione AVE, dopo l'ultimo MLP convolution layer della rete e subito prima del layer di output. L'estensione spaziale del layer è pari all'estensione spaziale del volume di output dell'ultimo MLP convolutional layer, ovvero alle dimensioni di altezza e larghezza di ciascuna depth slice di quel volume. Ad esempio se l'uscita dell'ultimo MLP convolutional layer è un volume 6 x 6 x 1000, dove 1000 è il numero di classi nel dataset, allora l'ultimo layer di pooling avrà un'estensione spaziale 6 x 6 e stride 1; il risultato sarà dunque un volume 1 x 1 x 1000, contenente i valori medi calcolati per ciascuna depth slice, che verrà passato all'output layer successivo.

Un vantaggio del global average pooling rispetto ai FC layer consiste nel fatto che esso rafforza le corrispondenze fra feature map e le categorie del dataset (figura 5.10). Oltre a questo, col global average pooling non ci sono parametri allenabili come con i FC layer e quindi si previene l'overfitting che invece normalmente si crea con i FC layer, ed anche si risparmia memoria.

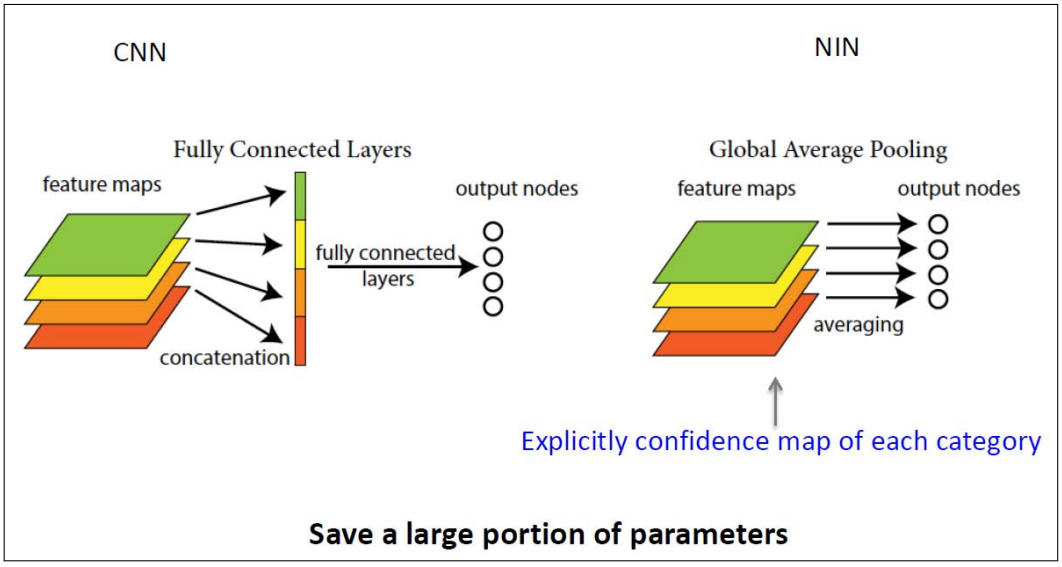


Figura 5.10: Il global average pooling

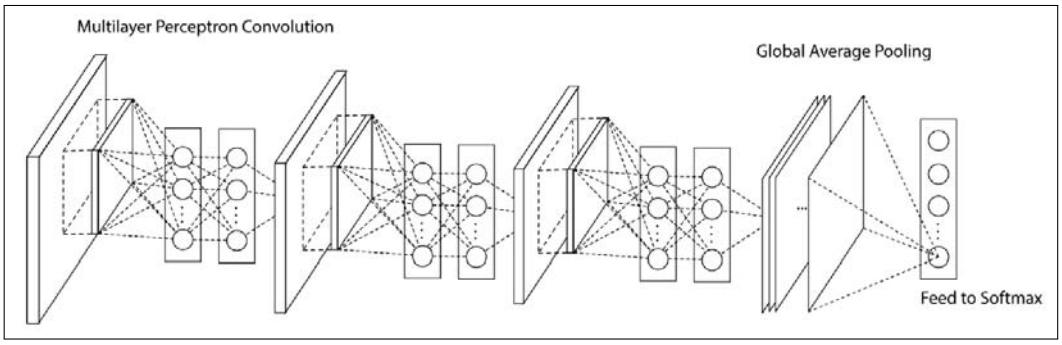


Figura 5.11: Architettura originale di NIN (3 MLP convolutional layer)

5.3.4 Architettura di NIN

L'architettura finale di NIN dovrebbe ormai essere abbastanza chiara: non è altro che una serie di MLP convolutional layer, al termine della quale vi è un global average pooling. La rete NIN è costituita nello specifico da 4 MLP convolutional layer. Al termine del terzo MLP layer è stato aggiunto un dropout layer, per limitare ulteriormente l'overfitting. In origine tuttavia la rete aveva un MLP convolutional layer in meno, come mostrato in figura 5.11. NIN è stata progettata nel 2013, nella successiva competizione ILSVRC del 2014 è stato aggiunto il MLP convolutional layer in più.

L'attuale versione, ottimizzata prima di essere presentata alla competizione ILSVRC dell'anno successivo è presente nella figura 5.12. Questa è la

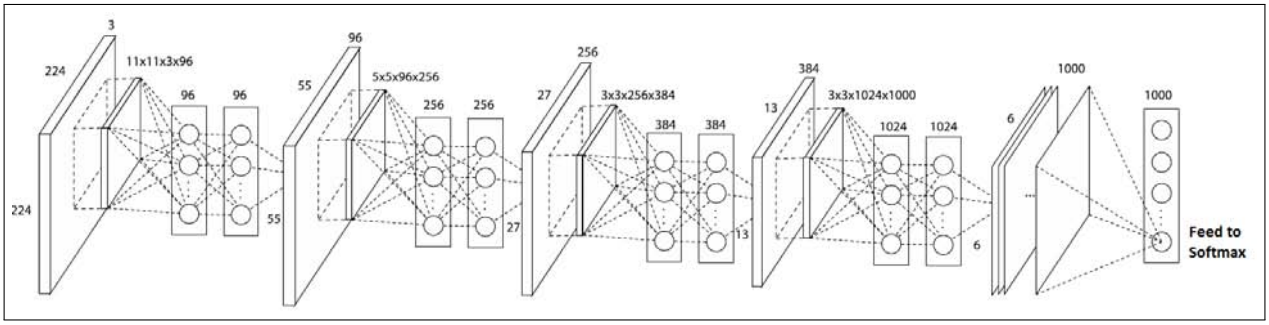


Figura 5.12: Architettura attuale di NIN (4 MLP convolutional layer)

	Parameter Number	Performance	Time to train (GTX Titan)
AlexNet	60 Million (230 Megabytes)	40.7% (Top 1)	8 days
NIN	7.5 Million (29 Megabytes)	39.2% (Top 1)	4 days

Figura 5.13: I parametri utilizzati da NIN

versione che verrà considerata nella tesi.

I 4 attuali layer convoluzionali principali, cioè i primi layer di ciascun MLP layer, hanno le seguenti caratteristiche:

- il primo possiede 96 kernel con un receptive field 11 x 11 e stride 4.
- il secondo possiede 256 kernel con un receptive field 5 x 5 e stride 1.
- il terzo possiede 384 kernel con un receptive field 3 x 3 e stride 1.
- il quarto possiede 1024 kernel con un receptive field 3 x 3 e stride 1.

Si noti che come al solito il numero di kernel aumenta scendendo lungo la rete, mentre le dimensioni del receptive field diminuiscono.

Il numero di classi su NIN viene specificato nell'ultimo layer convoluzionale prima del global average pooling.

NIN dunque utilizza un numero molto minore di parametri rispetto ad AlexNet ed alle VGG, come mostrato in figura 5.13, principalmente per l'assenza dei FC layer.

Il modello introdotto da NIN è stato utilizzato da molti altri team ed è inoltre stato testato su altre reti già esistenti. Potrebbe rappresentare una sorta di nuovo standard per le reti neurali convoluzionali, per avere comunque buone prestazioni utilizzando un numero inferiore di parametri; si tenga conto che non tutti i team sono comunque interessati a questo.

5.4 GoogLeNet

L'ultima rete analizzata è chiamata GoogLeNet ([18]), costruita nel 2014 da un team di dipendenti di Google e chiamata così in onore di una delle più vecchie, ovvero LeNet. GoogLeNet è sicuramente la rete che possiede l'architettura più complessa fra quelle viste, ma ciò è dovuto agli obbiettivi che il team voleva raggiungere. Come si era visto con le reti VGG, il metodo più diretto per aumentare le performance di una CNN è quello di aumentare le loro dimensioni lungo la profondità ("going deeper"). Tuttavia per GoogLeNet l'aumento è inteso sia come incremento di profondità, cioè il numero di livelli che la rete possiede, e sia come incremento della larghezza della rete, ovvero il numero di layer presenti su uno stesso livello. GoogLeNet si è cimentata dunque anche nella sfida di aumentare la larghezza di una rete, cosa che prima nemmeno si ipotizzava per il semplice fatto che non di riuscirebbe a gestire l'enorme quantità di dati che si verrebbe a creare.

Con un aumento della larghezza della rete, si intende la possibilità di svolgere su uno stesso volume di input all'interno della rete, diversi tipi convoluzioni, o meglio utilizzare ogni volta un layer convoluzionale con dei kernel di dimensioni diverse. Nel caso di GoogLeNet, essa prevede di effettuare tre tipi di convoluzione su un determinato volume di input:

- una convoluzione con dei kernel di dimensioni 1 x 1
- una convoluzione con dei kernel di dimensioni 3 x 3
- una convoluzione con dei kernel di dimensioni 5 x 5

Le feature ottenute da ciascuna convoluzione vengono poi successivamente tenute tutte in considerazione nei layer successivi. Si dovrebbe quindi cominciare a capire perchè prima si parlava di un'enorme quantità di dati: per ogni livello della rete, invece di avere i dati di un singolo layer convoluzionale, si hanno i dati di ben 3 layer convoluzionali; oltre a questo c'è da tenere conto del fatto che è necessario anche un tempo maggiore per effettuare tutte le operazioni di ciascun layer convoluzionale, in particolare per le convoluzioni con i kernel di dimensioni maggiori (3 x 3 e 5 x 5).

Ad ogni modo, l'aumento delle dimensioni, sia in profondità che in larghezza, di una rete porta due svantaggi significativi:

- innanzitutto reti di grandi dimensioni utilizzano un numero sempre maggiore di parametri e ciò può rendere una rete più incline all'overfitting, soprattutto nel caso in cui nel dataset non vi fossero molte immagini

- inoltre l'aumento delle dimensioni di una rete porta ad una drammatica crescita di potenza computazionale necessaria per l'allenamento. Ad esempio, se si hanno due layer convoluzionali consecutivi, un aumento uniforme del numero dei kernel utilizzati porta ad un aumento quadratico della computazione totale. Dato che nella realtà la potenza computazionale disponibile è sempre finita, è preferibile una distribuzione efficiente delle risorse rispetto ad un aumento sconsiderato delle dimensioni della rete, anche quando l'obbiettivo è quello di aumentare la qualità dei risultati.

L'approccio fondamentale per andare incontro ai due problemi citati è quello di passare da un'architettura completamente connessa ad un'architettura sparsa, anche all'interno delle convoluzioni.

Per questo motivo anche GoogLeNet non fa uso degli enormi FC layer nel modo in cui AlexnNet e nelle reti VGG: GoogLeNet ne utilizza solo uno alla fine e fra l'altro il team sostiene che non è nemmeno assolutamente necessario per il corretto funzionamento della rete, ma è stato aggiunto per tentare di migliorare ancora di più la rete stessa. Ad ogni modo, questo FC layer è completamente connesso ad un volume molto piccolo quando viene utilizzato, e quindi non possiede moltissimi parametri. GoogLeNet infatti prende spunto da NIN ed utilizza i layer convoluzionali 1 x 1 con stride 1 (CCCP layer su NIN), seguiti dai soliti layer di ReLU. Tuttavia su GoogLeNet i layer convoluzionali 1 x 1 hanno anche un altro scopo: vengono principalmente utilizzati come moduli per la riduzione della dimensioni prima di effettuare convoluzioni computazionalmente più costose come quelle con i kernel 3 x 3 o 5 x 5, per rimuovere eventuali "colli di bottiglia" all'interno della rete. In questo modo è stato possibile aumentare non solo la profondità della rete, ma anche la sua larghezza, senza un significativo calo delle performance.

5.4.1 L'inception layer

Ci si potrebbe chiedere perchè utilizzare diversi tipi di convoluzioni sullo stesso input. La risposta è che non sempre con una singola convoluzione, per quanto accuratamente siano stati studiati i suoi parametri, è possibile ricavare delle feature utili al fine di effettuare un'accurata classificazione. Generalmente infatti con alcuni input funzionano meglio convoluzioni con dei kernel di piccole dimensioni, mentre con altre si ottengono risultati migliori con altri tipi di kernel. Per questo motivo probabilmente il team di GoogLeNet vuole tentare di considerare più alternative all'interno della loro rete.

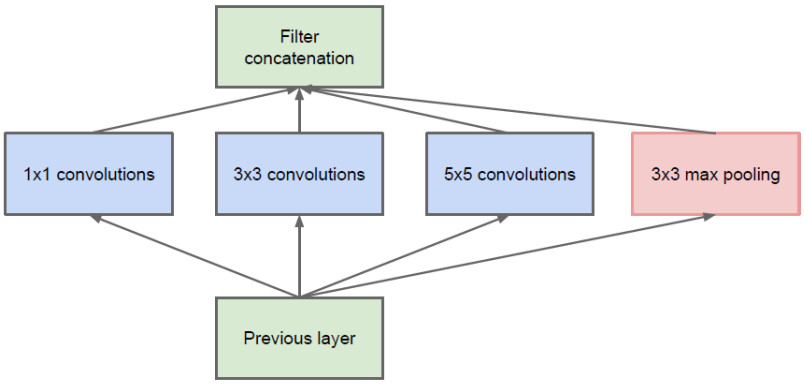


Figura 5.14: Versione naive di un inception layer

Come già detto prima, GoogLeNet utilizza ben 3 tipi di layer convoluzionali nello stesso livello della rete, cioè sono in parallelo, per questo scopo: si tratta di un layer 1 x 1, un layer 3 x 3 e un layer 5 x 5.

Il risultato complessivo di questa architettura locale di 3 layer paralleli è la combinazione di tutti i loro valori output, concatenati in un singolo vettore di output che costituirà l'input del layer successivo. Ciò viene fatto grazie all'uso di un concat layer.

In realtà nella stessa struttura locale, oltre ai 3 layer convoluzionali paralleli è stato aggiunto, sempre in parallelo, anche un pooling layer dato che si è visto che le operazioni di pooling risultano essenziali per il successo generale di una CNN negli ultimi anni.

L'insieme di questi 4 layer paralleli, i quali uniscono i loro risultati in un successivo concat layer, viene detto Inception layer: il nome deriva da un noto film da cui è stato ricavato un famoso meme chiamato "We need to go deeper", con il messaggio in questo caso relativo ovviamente alle dimensioni di una deep neural network. L'inception layer viene mostrato in figura 5.14. All'interno dell'inception layer sono sempre presenti i layer ReLU dopo ciascuna convoluzione, come per NIN.

5.4.2 Architettura generale

Esattamente come NIN era il risultato dell'utilizzo di più MLP convolutional layer, GoogLeNet è il risultato della concatenazione di più inception layer successivi, con qualche accorgimento in più che si vedrà fra poco.

Purtroppo, una volta che più inception layer sono stati assemblati insieme, il team ha visto che la rete presentava un grave problema: l'utilizzo di un inception layer del tipo di figura 5.14, in una rete di dimensioni piutto-

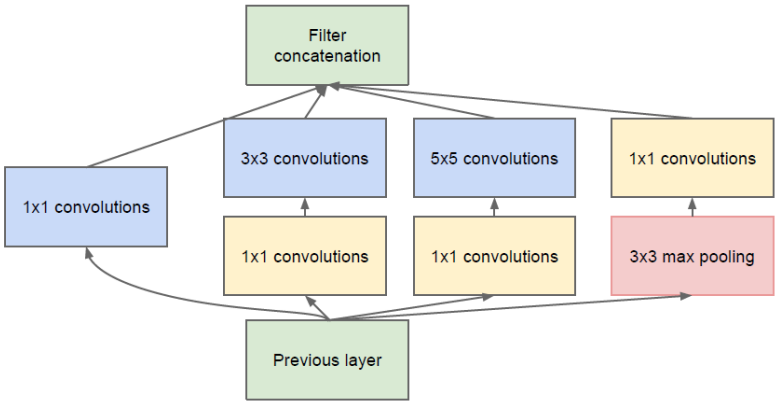


Figura 5.15: Versione finale di un inception layer

sto grandi, risultava essere proibitivo per via della quantità di dati che si accumulavano nella rete. L'utilizzo ad esempio di un numero anche abbastanza modesto di kernel per le convoluzioni 5 x 5 risultava essere troppo costoso in questo senso. Inoltre l'aggiunta, tramite il concat layer, di tutti i dati che derivavano dal ramo dove era presente il layer di pooling peggiorava ancora di più la situazione: si ricorda che l'operazione di pooling riduce le dimensioni spaziali di un volume ma non modifica la sua profondità. Con i risultati di tutti i tipi di layer convoluzionali ed in più con i risultati del layer di pooling tutti assemblati insieme si otteneva un raggruppamento di dati troppo grande. Questo ammasso di dati andava poi dato in input di volta in volta all'inception layer successivo e la cosa diventava sempre di più ingestibile; si pensi che normalmente, come è stato visto finora, la dimensione della profondità dei volumi aumenta man mano che si procede verso il basso della rete.

La soluzione a questo problema ha portato ad una modifica della struttura di un inception layer: tramite riduzioni di dimensioni e un'operazione di proiezione applicati al suo interno nei punti dove i dati risultavano essere più numerosi, il problema è stato arginato. La sua risoluzione non ha inoltre richiesto una diminuzione della larghezza dell'inception layer, in quanto è come se i dati fossero stati compressi solo in alcuni punti critici. Per la riduzione delle dimensioni, prima delle convoluzioni 3 x 3 e 5 x 5 è stato aggiunto, uno per ciascun ramo, un layer convoluzionale 1 x 1 seguito come al solito da un layer ReLU. Analoga cosa dopo il layer di pooling: un layer convoluzionale 1 x 1 effettua una sorta di proiezione dei dati ottenuti. Il nuovo e definitivo inception layer è mostrato in figura 5.15.

All'interno della rete finale, occasionalmente dopo un inception layer può

esserci un ulteriore singolo livello costituito da un solo layer di pooling, con il solito scopo di ridurre le dimensioni spaziali dei volumi. Tutti i pooling descritti finora, anche quelli all'interno degli inception layer, effettuano operazioni di MAX. Dopo l'ultimo inception layer tuttavia, esattamente come NIN, è presente un global average pooling prima del loss layer che ha le stesse funzioni di quanto visto per la rete precedente. In questo caso però, l'implementazione di GoogLeNet differisce leggermente da quella di NIN, in quanto dopo il global average pooling è presente anche un layer FC lineare prima del loss layer. Questa scelta è stata presa con lo scopo di permettere un più semplice adattamento e fine-tuning della rete ad insiemi di label diversi, dipendenti dal dataset utilizzato. Il team di GoogLeNet sostiene comunque che l'utilizzo del FC layer non intacca in maniera significativa le prestazioni della rete.

Anche in questa rete l'utilizzo di un qualche layer di dropout risulta essere comunque essenziale: ne viene utilizzato uno prima del FC layer.

Spesso tuttavia si parla di FC layer al plurale, perchè su GoogLeNet in realtà non esiste un singolo loss layer, bensì 3. Oltre alla presenza di un loss layer finale come in tutte le normali reti, GoogLeNet possiede altri due layer di loss, collocati in due posizioni di profondità intermedie della rete, distanziati l'uno dall'altro. La rete di GoogLeNet è molto profonda e quindi può risultare utile il fatto che le feature prodotte, anche in livelli intermedi, della rete siano molto discriminanti. L'aggiunta di due classificatori ausiliari ha come obbiettivo proprio questo. Inoltre essi, durante le fasi di backward propagation, aumentano il segnale del gradiente che viene propagato verso l'inizio della rete, permettendo una migliore diffusione che invece prima, senza il loro utilizzo, secondo il team risultava essere un discreto problema. Durante il training il valore di loss di questi due classificatori ausiliari viene aggiunto al valore di loss totale, cioè quello dell'ultimo layer di loss della rete, ma essi nella somma hanno un peso minore: entrambi nella loro definizione infatti possiedono un parametro *loss_weight* (paragrafo 4.2.6) settato a 0.3, il che significa che il loro valore di loss viene moltiplicato per quel corrispondente valore. In fase di testing invece queste piccole reti ausiliarie vengono ignorate. Si parla di reti ausiliarie proprio perchè effettivamente lo sono: prima di ognuno dei due loss layer ausiliari sono presenti anche due layer FC preceduti da un layer convoluzionale 1 x 1 e da un global average pooling.

L'appendice A fornisce i dettagli dell'architettura complessiva di GoogLeNet.

Un aspetto importante di questa architettura consiste nel permettere l'aumento significativo del numero di unità, layer dopo layer, senza che però questo aumento porti ad un'incontrollabile complessità computazionale. Oltre a questo, il design della rete è in linea con il fatto che le informazioni

dovrebbero essere processate in varie scale e successivamente aggregate in modo che nello stage successivo le feature si possano astrarre da diverse scale simultaneamente.

5.5 Confronto prestazioni sul dataset ImageNet

Nel paragrafo 4.4.9 si era parlato di top-1 e top-5 accuracy. Queste sono le metriche con cui solitamente si misurano le prestazioni di queste reti nelle varie competizioni di classificazione; analogamente vengono utilizzate le misure di top-1 e top-5 error. Queste metriche sono ovviamente l'esatto contrario delle accuracy: error = 1 - accuracy, cioè il rapporto tra il numero di immagini non classificate correttamente sul validation set ed il totale delle immagini presenti nel validation set. Quindi l'indice di bontà di una rete deriva dal fatto che essa possegga un errore complessivo il più basso possibile. Il motivo per cui vengono analizzati sia il top-1 che il top-5 error è sempre lo stesso citato nel paragrafo 4.4.9: di norma queste reti nelle competizioni lavorano su dataset aventi un numero di classi molto elevato; il top-5 error è un indice per indicare che se anche la rete non classifica bene molte immagini magari non è così pessima come si potrebbe pensare, in quanto per molte immagini potrebbe non essere lontana da una corretta classificazione.

Un dataset comune a tutte le reti analizzate finora, sul quale i vari team hanno testato le loro performance nelle competizioni ILSVRC dei vari anni, è quello di ImageNet. Questo dataset possiede 1000 classi e un totale di più di un milione di immagini al suo interno, per l'esattezza circa 1.2 milioni di immagini nel training set e circa 50,000 immagini nel validation set. Le classi non sono tutte ben distinte fra loro: molte possiedono immagini molto simili e sono dunque più difficilmente distinguibili. L'esempio di figura 5.16 mostra due diverse classi di Imagenet: nonostante si possa pensare che entrambe le immagini rappresentino la stessa razza di cane, un husky, in realtà a sinistra è presente un Siberian Husky, mentre a destra è presente un'altra razza di cane che non è un Husky ma che gli assomiglia molto.

Il dataset, come si può capire, è stato creato in modo da rendere più difficile ed avvincente la sfida fra i vari team e le loro reti. Durante queste competizioni vengono inoltre solitamente fornite anche un buon numero di immagini non presenti nel dataset al fine di testare le effettive performance della rete, una volta allenata.

La tabella 5.1 riporta i risultati delle reti viste finora sul dataset ImageNet. Tutti i risultati riportati considerano solo 1 crop centrale per ciascuna immagine e sono relativi all'ambito della sola classificazione; le varie competizioni sono di solito divise in più task, ad esempio classification, classification+localization, detection ecc., così una rete può ottenere risultati in ambiti anche piuttosto diversi. Per quanto riguarda le reti VGG è stata inserita solo VGG-19, in quanto risulta essere la migliore fra tutte le reti VGG,

(a) Siberian husky (b) Eskimo dog

Figura 5.16: Due immagini di due classi diverse su Imagenet

	Top-1 accuracy	**Top-5 accuracy**
AlexNet	57,1%	80,2%
NIN	59,4%	/
GoogleNet	68,7%	88,9%
VGG-19	74,5%	92,0%

Tabella 5.1: Prestazioni reti analizzate su ImageNet

anche se la più pesante. L'utilizzo, da parte di tutte le reti, di 10 crop per ciascuna immagine (4 ai lati, una centrale e le versioni specchiate di esse) dovrebbe far aumentare leggermente tutte le performance, anche se non di molto.

Tutti questi dati sono ricavabili sulla pagina di GitHub di ciascuna delle reti, dove solitamente è presente il modello scaricabile di ciascuna rete avente le prestazioni illustrate. Nella tabella non è presente la top-5 accuracy di NIN in quanto non è stato trovato tale dato. Teoricamente però, dato che la sua top-1 accuracy è leggermente migliore di quella di AlexNet, lo stesso dovrebbe valere per la top-5 accuracy. Notare il fatto che VGG-19 risulta essere la migliore fra tutte, in termini di accuratezza.

Capitolo 6

Considerazioni progettuali sul consumo di risorse

Nel capitolo precedente sono state osservate dettagliatamente le reti neurali convoluzionali più potenti e famose degli ultimi anni. Di ciascuna rete sono state inoltre riportate le precise prestazioni di classificazione sullo stesso dataset Imagenet, dataset costruito appositamente per verificare la bontà delle reti più avanzate.

Tuttavia bisogna tenere conto anche di altri aspetti oltre a quello delle performance in termini di accuratezza delle predizioni. Lo scopo del progetto infatti prevede l'utilizzo di una rete neurale convoluzionale in un dispositivo mobile, il quale risulta essere limitato dalle risorse che esso possiede: nonostante negli ultimi anni il divario fra questi device ed i normali computer si stia riducendo sempre di più, bisogna pensare che le reti finora viste normalmente vengono allenate ed utilizzate su dei computer con un hardware di ultimissima generazione; fortunatamente questi potenti computer hanno come scopo principale quello di allenare nel minor tempo possibile una rete, non quello di utilizzarla sul campo, cosa per cui sarebbe sicuramente sufficiente anche un hardware inferiore. Infatti il tempo di allenamento di una di queste reti è enorme rispetto al tempo per effettuare i test.

Dato che per quanto riguarda il progetto, il tempo di allenamento di una rete non costituirebbe un grosso problema (nei limiti del possibile con i tempi a disposizione), la cosa che invece risulta essenziale è capire se queste reti possono potenzialmente essere adatte a funzionare su un dispositivo Android, il quale possiede un hardware decisamente inferiore rispetto ai supercomputer menzionati prima.

Si tratta quindi di capire se ad esempio i tempi di elaborazione di un'immagine con queste reti non siano eccessivamente lunghi, in modo da poter effettivamente pensare ad un loro utilizzo.

Il dispositivo dispone inoltre di una memoria piuttosto limitata rispetto un normale computer: il sistema Android, nel caso un'applicazione necessitasse di troppa memoria per il suo normale funzionamento, potrebbe ucciderla per preservare le risorse ad esso necessarie per mantenere una situazione stabile del sistema. Bisogna dunque considerare anche questo aspetto: utilizzare una rete che magari risulta essere troppo pesante, in termini di memoria, non è decisamente consigliato.

Un altro aspetto importante da analizzare riguarda il consumo energetico di queste reti una volta applicate ed utilizzate sul dispositivo: se il consumo fosse eccessivo, il loro utilizzo in locale sarebbe probabilmente sconsigliato, in quanto ci sarebbero problemi di utilizzo per via del veloce scaricamento della batteria. Si tratta quindi di tentare di analizzare il consumo energetico di queste reti utilizzate in locale sul dispositivo, in modo da poter evidenziare eventuali limiti di questo approccio.

Ai fini di evidenziare potenziali limiti sull'utilizzo delle reti neurali convoluzionali su un dispositivo mobile, verranno illustrati in questo capitolo i risultati di alcuni test riguardanti le tematiche finora descritte e le relative considerazioni che ne derivano con i dati ottenuti. I test sono stati effettuati sia su un PC, avente una CPU quad-core Intel Core i5-2550K, 3.8 GHz e 16 GB (4x4) DDR3 1333 MHz di RAM e sia su un tablet Nexus 7 avente una CPU 1.51 GHz quad-core Krait 300, con 2 GB DDR3L 1600 MHz di RAM.

Tutti i dettagli implementativi vengono rimandati al capitolo 7.

6.1 Considerazioni sulle dimensioni dei modelli

Il problema più grande in questo contesto riguarda le reti VGG: esse posseggono un numero elevatissimo di parametri e quindi molto probabilmente risulterebbero troppo pesanti per le generali caratteristiche hardware di un dispositivo mobile. Le reti VGG infatti producono tutte un modello di dimensioni superiori ai 500 MB e tale modello, per classificare delle immagini all'interno del dispositivo mobile, deve essere caricato almeno una volta in memoria e successivamente mantenuto in essa per effettuare le successive predizioni. Potrebbe anche essere ricaricato per ogni singola predizione, ma date le dimensioni sicuramente non conviene fare questo. Quindi in sostanza il peso del modello risulterebbe di certo un problema. Oltre a questo c'è da aggiungere il fatto che queste reti, o meglio alcune di esse, risultano essere fra le più profonde attualmente esistenti, il che fa pensare anche ad un tempo di forward pass, per predirre la classe di un'immagine, piuttosto lungo, dato che in queste reti non ci si è sicuramente focalizzati sul rendere un po' più efficienti alcune operazioni (a differenza di GoogLeNet). Le reti VGG sono

reti che hanno dimostrato di funzionare molto bene e di anno in anno, con l'uscita di un hardware sempre più potente, vengono migliorate; il loro limite è quindi direttamente proporzionale all'hardware attualmente disponibile. Ad esempio quando sarà disponibile una nuova GPU più potente di quelle attualmente in circolazione, sarà possibile aggiungere ulteriori layer alla reti, aumentandone dunque la profondità e molto probabilmente anche le loro prestazioni. Questa filosofia non risulta essere sicuramente la più adatta per gli scopi del progetto.

Quindi, per i motivi finora citati, queste reti sono state escluse dai successivi test, in quanto le speranze di un loro possibile utilizzo su un dispositivo Android sono veramente basse. Inoltre perfino il loro allenamento con la tecnica del fine-tuning risultava essere quasi improponibile con il computer a disposizione.

Un discorso simile si potrebbe fare per AlexNet: il suo modello infatti pesa circa 240 MB. Tuttavia le dimensioni sono già dimezzate rispetto alle reti VGG, ed inoltre questa rete detiene un'architettura molto più semplice rispetto alle suddette reti: possiede un numero decisamente inferiore di layer convoluzionali e ciò potrebbe far sì che le predizioni, una volta caricato il modello, siano abbastanza veloci. Come è stato visto nel paragrafo 5.5, nonostante la semplicità di AlexNet, le sue performance di classificazione, seppur più basse rispetto alle reti VGG, sono comunque buone. Si ricorda che infatti AlexNet ha vinto la competizione ILSVRC del 2012. Quindi tale rete viene considerata complessivamente "testabile" su Android, con la consapevolezza che comunque il peso del suo modello potrebbe creare problemi.

Le rimanenti due reti viste in precedenza, NIN e GoogLeNet, sembrano invece essere le più adatte allo scopo finale: entrambe hanno un'architettura piuttosto complessa, soprattutto GoogLeNet, la quale infatti è la rete più profonda fra tutte quelle analizzate, ma possiedono un modello relativamente piccolo (NIN di circa 30 MB, GoogLeNet di circa 50 MB), il che significa che utilizzano un numero limitato di parametri. Inoltre esse hanno prestazioni più che buone, in particolare GoogLeNet, migliori in generale anche di AlexNet. Come già accennato prima, GoogLeNet è molto profonda come rete e questo potrebbe portare ad un tempo di predizione più elevato; tuttavia in questa rete sono state fatte diverse ottimizzazioni allo scopo di renderla più efficiente e le dimensioni limitate del suo modello e le prestazioni ottime la rendono una rete più che appetibile per lo scopo del progetto.

Nella tabella 6.1 vengono evidenziate le precise dimensioni del modello di ciascuna delle reti viste. Si ricorda che le dimensioni di un modello sono direttamente proporzionali al numero di parametri utilizzati dalla rete.

C'è un ulteriore appunto da fare: le dimensioni dei modelli dovrebbero comunque leggermente ridursi, in quanto le dimensioni finora viste riguarda-

Rete	Dimensioni modello (in MB)
AlexNet	238
NIN	29
GoogLeNet	51
VGG-19	550

Tabella 6.1: I modelli delle reti con un dataset di 1000 classi

no i modelli ottenuti dalle reti con il dataset ImageNet avente 1000 classi e quindi tutti gli ultimi layer con parametri allenabili di ciascuna rete posseggono 1000 unità. I dataset normalmente utilizzati per lo scopo finale hanno un numero di classi di norma decisamente inferiore e quindi di conseguenza anche i relativi layer hanno meno unità. Si parla di leggera riduzione, perchè in tutti i casi i layer interessati non sono certamente quelli che pesano di più sulle dimensioni finali del modello, dato che il numero di neuroni, e quindi di parametri, che essi possiedono sono una piccola percentuale rispetto al totale. In conclusione dunque le reti che sono state viste come potenzialmente applicabili su Android sono AlexNet, NIN e GoogLeNet. Per quanto riguarda AlexNet esiste un livello di incertezza maggiore, ma ciò non significa che non possa funzionare.

6.2 Tempi computazionali: fasi principali di una predizione

L'idea è quella di cominciare ad ipotizzare quale fra le tre reti selezionate possa risultare la più adeguata a funzionare su Android, controllando in questo caso i tempi necessari per la predizione di un'immagine da parte di tutte le reti.

6.2.1 Misure su PC

Le misure riportate in questa sezione riguardano il PC con le caratteristiche menzionate all'inizio di questo capitolo.

Qui, con la tabella 6.2, vengono mostrati i tempi delle principali operazioni all'interno dell'intero script di classificazione; come si può notare dalla tabella, alcune risultano avere un tempo irrisorio, ma per scrupolosità sono state ugualmente considerate in questa prima fase.

Misurazione temporale	**AlexNet**	**NIN**	**GoogLeNet**
Caricamento immagine input (s)	0.018	0.017	0.021
Caricamento modello della rete (s)	1.771	0.239	0.158
Resize + mean subtraction (s)	0.002	0.001	0.001
Caricamento delle label (s)	2.1e-05	2.1e-05	2.2e-05
Forward pass (s)	0.172	0.132	0.585
Ordinamento vector finale (s)	6e-05	6.8e-05	6.5e-05
Predizione complessiva (s)	1.963	0.390	0.766

Tabella 6.2: Risultati test singola predizione su PC

Tutti i dati sono i risultati della media di 5 tentativi di classificazione della medesima immagine. Si è scelto di utilizzare sempre la stessa immagine, sia per questi test su PC sia per quelli che successivamente verranno mostrati riguardanti il dispositivo Android, in quanto una rete, indipendentemente dall'immagine ricevuta in input, effettua sempre le medesime operazioni su un'immagine. Si ricorda infatti che ogni immagine data in input viene ridimensionata prima di essere elaborata, quindi tutte le immagini risultano essere delle stesse dimensioni prima della loro elaborazione: ciò vuol dire che il numero di operazioni effettuate sono sempre le stesse per ciascuna di esse e quindi usare la stessa immagine rispetto ad utilizzarne diverse, non dovrebbe influenzare i calcoli che la rete esegue.

Il primo tempo riguarda semplicemente il caricamento dell'immagine di input in memoria.

Il secondo tempo presente nella tabella riguarda sostanzialmente il tempo di caricamento del modello, ovvero il file binario *.caffemodel* contenente i parametri della rete, necessario per inizializzare l'intera rete.

Il terzo tempo contiene l'intero preprocessing eseguito normalmente da Caffe, ovvero il resize dell'immagine originale in una di dimensioni 256 x 256 ed il processo di mean subtraction sull'immagine originale.

Il quarto tempo riguarda il caricamento delle label e delle classi dal file *synsetword.txt*.

Il quinto tempo include l'intera forward propagation dell'input sulla rete fino al calcolo finale delle probabilità per ogni classe.

Il sesto tempo riguarda semplicemente il tempo per il riordinamento, secondo un ordine decrescente, del vector contenente le probabilità calcolate in

precedenza.

Il settimo tempo è il tempo totale della predizione di un'immagine, cioè è circa la somma di tutti i tempi precedenti, in quanto può differire leggermente perchè qui è stato considerato proprio l'intero codice dello script e non la somma delle pure singole operazioni.

Con questi tempi si possono fare alcune riflessioni: per quanto riguarda il caricamento della rete in memoria (il primo tempo), AlexNet risulta essere la peggiore in termini di tempo. Ciò è comprensibile visto il peso del modello di questa rete di circa 227 MB. Le altre due reti migliorano parecchio avendo modelli molto più piccoli (NIN circa 26 MB e GoogLeNet circa 41 MB) nonostante abbiano un'architettura più complessa. Non è del tutto chiaro perchè in generale GoogLeNet ottenga un tempo leggermente migliore rispetto a NIN, dato che possiede sia un modello leggermente più grande e sia una struttura più complessa e quindi in teoria avente più dati da caricare nel complesso.

Per quanto riguarda il tempo di classificazione, AlexNet risulta essere migliore rispetto a GoogLeNet. Ciò ha senso, in quanto in questa fase si effettua una forward propagation della rete ed essendo l'architettura di AlexNet molto più semplice rispetto a quella di GoogLeNet: quest'ultima possiede molti più layer, anche considerando solo quelli con parametri allenabili, rispetto ad AlexNet ed è dunque plausibile che la forward pass di AlexNet sia più veloce di quella di GoogLeNet. E' vero che AlexNet ha milioni di parametri in più rispetto a GoogLeNet, dovuti all'utilizzo dei FC layer su volumi di neuroni molto grandi, ma ciò incide molto sulle dimensioni del modello, e quindi sul suo caricamento, e sulla backward pass in fase di train, dove questi parametri vanno ogni volta aggiornati. In una singola forward pass il fatto di avere un grande numero di parametri a quanto pare incide meno rispetto al fatto di avere un numero decisamente alto di layer, e quindi di operazioni diverse da effettuare, come GoogLeNet.

E' vero però che AlexNet probabilmente richiede l'utilizzo di un'elevata quantità di memoria ed anche se il modello si potesse, una volta caricato, tenere in memoria senza doverlo ricaricare ogni volta, si avrebbe comunque una buona percentuale di essa utilizzata solo ed esclusivamente a questo scopo.

NIN risulta essere la migliore di tutti in questo caso ed anche qui tutto sembra sensato: NIN ha un'architettura più complessa di AlexNet ma meno complessa di quella di GoogLeNet, anche se a spese di un leggero calo di prestazioni; inoltre possiede un numero infinitamente minore di parametri rispetto ad AlexNet. L'insieme di questi due fatti (numero basso di parametri e rete non troppo complessa) porta, a mio avviso, al leggero miglioramento temporale rispetto ad AlexNet.

Misurazione temporale	AlexNet	NIN	GoogLeNet
Caricamento modello della rete	16.738	2.067	1.986
Forward pass	1.091	1.079	2.198

Tabella 6.3: Risultati test singola predizione su Android

Tutti gli altri tempi indicati nei file sono stati presi per scrupolosità, ma come si può vedere non influiscono in maniera significativa sul tempo totale di predizione.

6.2.2 Misure su un dispositivo Android

Esattamente come visto nella sezione precedente, verranno ora illustrati i risultati, in termini di tempo di esecuzione, dei vari test delle tre reti selezionate sul tablet a disposizione. Verranno prese in considerazione solo le due fasi più importanti di una singola predizione, cioè quelle temporalmente più dispendiose, ovvero il tempo di caricamento del modello di una rete ed il tempo della sua forward pass. Gli altri tempi, come visto in precedenza nella tabella 6.2, non risultano influire pesantemente sul tempo complessivo di una predizione, che è appunto principalmente governato dai due fattori menzionati, in quanto la loro somma porta ad avere circa il tempo di un'intera predizione infatti; pertanto i tempi poco influenti sono stati esclusi da questa analisi.

La tabella 6.3 riporta i risultati delle misurazioni ottenute con le tre CNN; ogni dato è la media risultante di 5 diverse esecuzioni, tutte applicate sempre alla stessa immagine, la medesima usata anche per i test di questi tempi sul PC.

La prima cosa che si nota è che in generale su Android tutti i tempi sono aumentati, rispetto alle medesime misure prese su PC, in linea con quanto ci si aspettava, dato che un computer generalmente ha un hardware più potente di un dispositivo Android.

Tuttavia confrontando le prestazioni di ciascuna rete all'interno della stessa operazione, sostanzialmente le relazioni d'ordine in termini di tempo speso, sono rimaste praticamente le stesse. Infatti AlexNet risulta essere ancora la peggiore per quanto riguarda il caricamento del modello della rete, mentre le altre due hanno tempi decisamente migliori, con GoogLeNet in leggero vantaggio su NIN; la forward pass di GoogLeNet risulta ancora la peggiore,

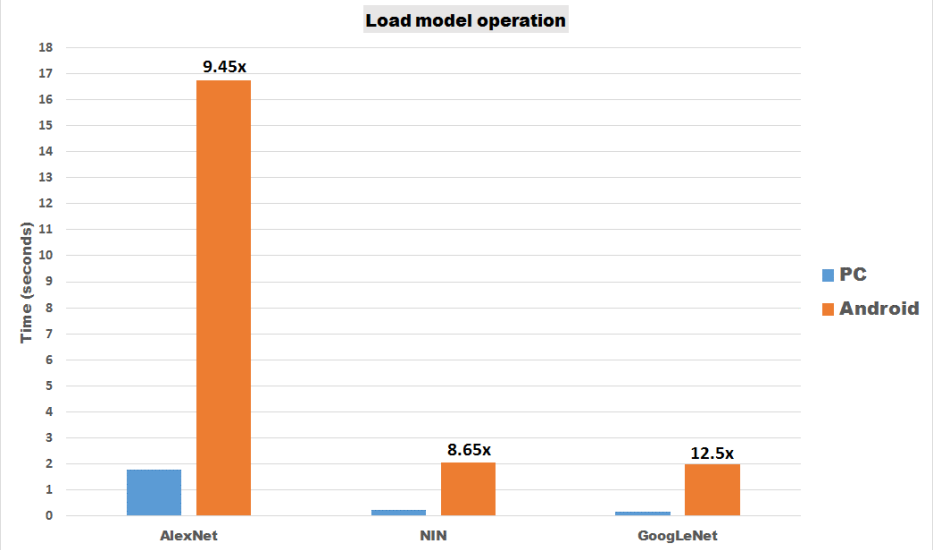


Figura 6.1: Confronto tempi PC e Android nel caricamento del modello

mentre AlexNet e NIN si comportano molto meglio, con NIN leggermente più veloce di AlexNet.

Le figure 6.1 e 6.2 danno un'idea di quanto i tempi siano aumentati nella due operazioni considerate.

Con questi dati risulta chiaro che AlexNet non risulta di certo essere le rete più indicata ad essere utilizzata su Android: il suo tempo medio di una forward pass è molto buono, ma per caricare un modello sono necessari quasi 17 secondi; un'infinità di tempo insomma nonostante si sia visto che il modello può essere caricato una sola volta per tutte le immagini da predire.

GoogLeNet si comporta già molto meglio, anche se ad ogni predizione impiega circa 2 secondi per una forward pass, più o meno il doppio di AlexNet e NIN.

NIN al contrario risulta essere una buona via di mezzo: tempo di caricamento di un modello molto buono e la migliore fra le 3 in termini di tempo di forward pass; come già visto nella tabella 7.2, le sue prestazioni, considerando l'accuracy, sono leggermente inferiori rispetto a GoogLeNet, ma in questo caso sembra essere un compromesso accettabile. Bisogna tenere conto poi del fatto che molto dipende dal dataset con cui è stata allenata la rete: con un dataset diverso le reti potrebbero avere prestazioni diverse, anche se guardando ad esempio la tabella 5.1 in cui come dataset è stato considerato ImageNet, le relazioni d'ordine anche qui si mantengono.

In conclusione NIN potenzialmente sembra essere la CNN più adatta, fra le tre selezionate, a funzionare su Android.

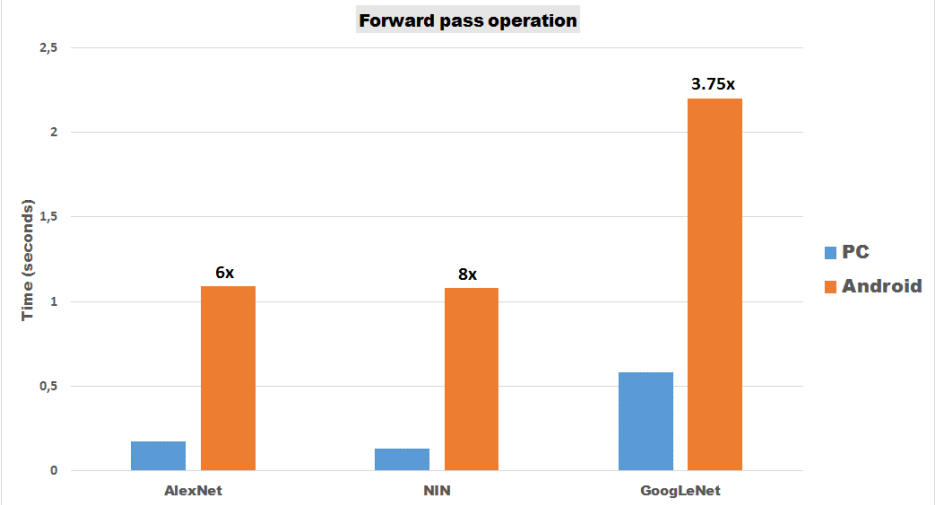
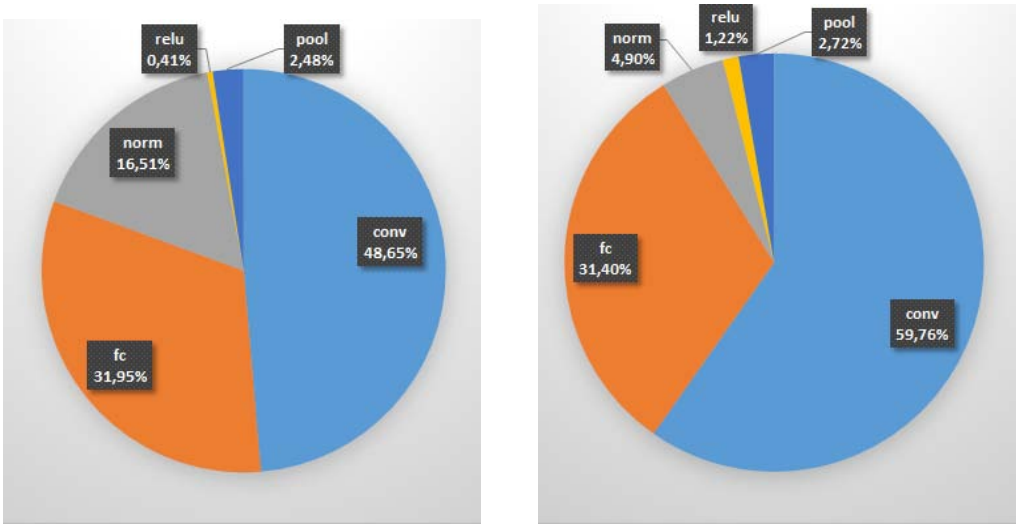


Figura 6.2: Confronto tempi PC e Android di una singola forward pass

Un'altra cosa degna di nota riguarda le differenze fra i tempi ricavati su PC ed i tempi ricavati su Android: nonostante ci si aspettasse un loro aumento, quest'ultimo non è eccessivamente grande. I tempi analizzati in questa sezione fanno capire che l'utilizzo in locale di reti neurali convoluzionali su un dispositivo Android, con reti anche abbastanza complesse, sembra potenzialmente fattibile, dato che i tempi non sono troppo elevati. Il fatto poi di poter utilizzare reti di questo calibro probabilmente risulterà utile per la qualità della classificazione all'interno del progetto.

6.3 Tempi computazionali: apporto dei singoli layer

Dopo essersi fatti un'idea generale sui tempi in media spesi per effettuare un'intera singola predizione di un'immagine sia su PC che su Android, per approfondire ulteriormente si è voluto anche misurare in media quanto tempo viene speso in ciascun layer di ognuna delle reti finora considerate durante una singola forward pass. In questo modo è possibile individuare quali sono i layer temporalmente più onerosi nel complesso ed avere in questo modo un quadro più generale ed approfondito sulle prestazioni complessive di ogni rete, durante il procedimento di classificazione. Come appena accennato, i tempi in questione riguardano una singola forward pass, la quale costituisce solo un parte di una singola predizione (si guardi la tabella 6.2) anche se, escludendo il caricamento del modello, temporalmente la comprende quasi



(a) PC (b) Android

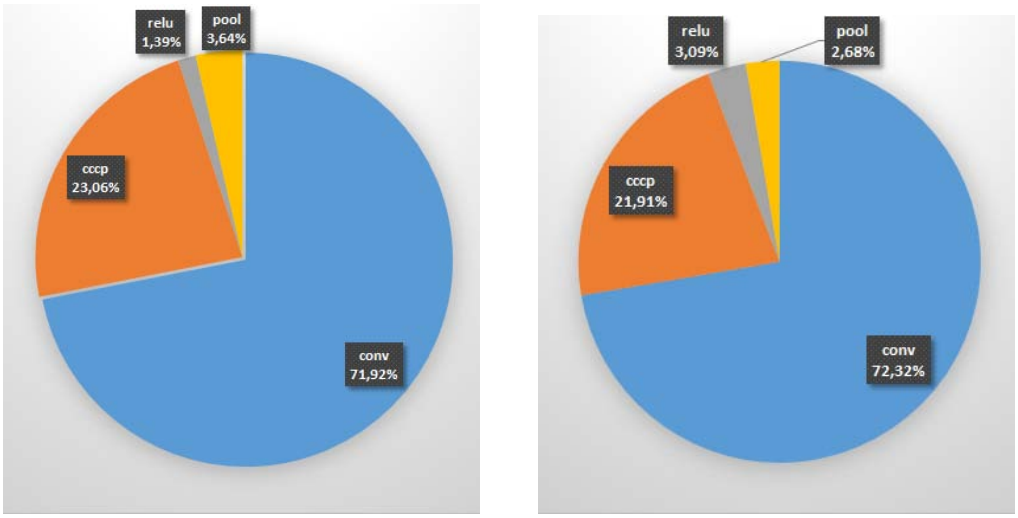
Figura 6.3: Suddivisione tempi su AlexNet

tutta; sono dunque esclusi da questa analisi tutti gli altri tempi illustrati nella tabella 6.2. Le analisi sono state effettuate come al solito sia su PC che sul tablet Android.

Tutti i dati, sia quelli rilevati su PC che quelli rilevati sul tablet Android, sono stati successivamente inseriti su Excel, tramite il quale è stato possibile visionarli nel dettaglio e costruire dei grafici che esprimessero, per ciascuna rete e piattaforma, la suddivisione dei tempi di una singola forward pass in base alle tipologie di operazioni, cioè di layer, effettuate all'interno di essa. Si è scelto di suddividere i tempi in base alla loro tipologia, cioè alla loro specifica funzione, perchè il numero di layer complessivi di tutte le reti è piuttosto alto, soprattutto con GoogLeNet, e quindi costruire un grafico che indichi l'apporto di ogni singolo layer non sarebbe sensato, in quanto diventerebbe tutto incomprensibile. Risulta più utile invece capire quanto tempo viene dedicato ad una specifica tipologia di operazioni, dato dalla somma degli apporti di tempo di tutti i layer aventi la stessa specifica tipologia, durante una singola predizione.

Le figura 6.3 mostra quanto ottenuto con la rete AlexNet.

Per quanto riguarda AlexNet, è facile vedere che le suddivisioni su PC e su Android si assomigliano abbastanza; c'è una leggera differenza sui tempi riguardanti le normalizzazioni, in quanto le normalizzazioni su PC detengono una percentuale leggermente più alta rispetto a quella sul device Android. Si noti inoltre che in entrambi i casi, un buon 30% del tempo di una forward pass



(a) PC (b) Android

Figura 6.4: Suddivisione tempi su NIN

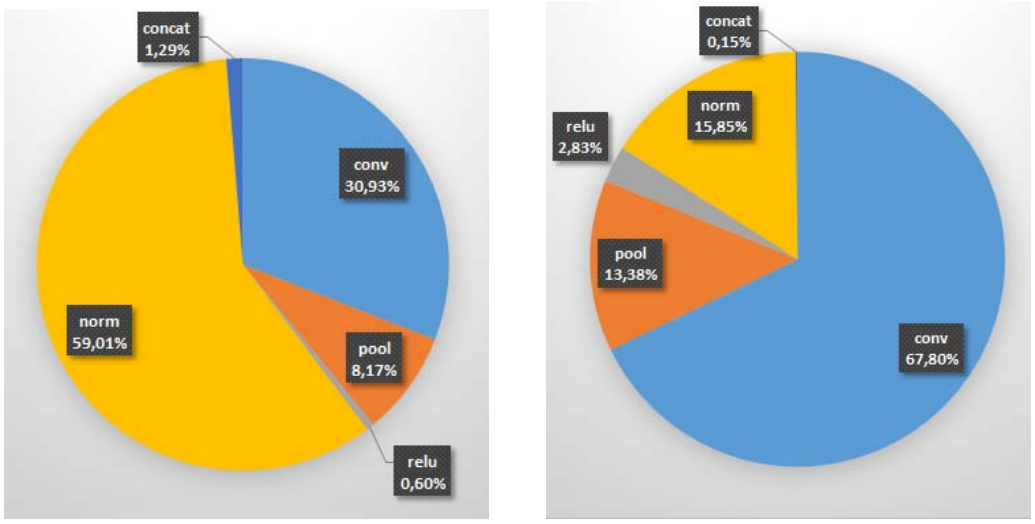
di AlexNet è riservato ai FC layer, in particolare (non si vede nella figura) ma più del 20% è dato solo dal primo di questi layer; singolarmente esso è il layer con la percentuale di tempo maggiore fra tutti. Ciò è probabilmente dovuto all'enorme quantità di parametri che questo layer possiede (si veda il paragrafo 5.1.3).

Proseguendo con NIN, i dati ottenuti per essa sono mostrati nella figura 6.4.

Qui i tempi fra le due piattaforme sono veramente simili. Non ci sono operazioni di normalizzazione, dato che NIN non ha nessun layer di questo tipo e nemmeno FC layer. I CCCP layer all'interno degli MLP convolutional layer, nonostante siano sempre dei layer convoluzionali sono stati comunque separati dalla percentuale di tali layer, in quanto costituiscono un caso particolare di questo tipo di layer e volendo sono la caratteristica che contraddistingue NIN dalle reti come AlexNet. Nel caso si volesse considerare il tempo complessivo totale dovuto alle operazioni di convoluzione, basta sommare le percentuali del grafico *conv* e *cccp*, ottenendo dunque una percentuale superiore al 90% in entrambi i casi.

Infine la figura 6.5 mostra cosa succede con GoogLeNet.

In questo caso si può invece notare una netta differenza fra i due grafici, principalmente dovuta alla normalizzazione effettuata su PC. Il grafico relativo ad Android infatti ha caratteristiche piuttosto simili a tutti i grafici osservati in precedenza, ovvero con la percentuale più alta detenuta dalle operazioni di convoluzione; su PC invece la normalizzazione si prende la fet-



(a) PC (b) Android

Figura 6.5: Suddivisione tempi su GoogLeNet

ta più grande. Questo fatto ha concentrato l'interesse sul funzionamento di questo tipo di layer su PC e per questo sono state fatte ulteriori analisi, le quali verranno illustrate fra poco.

La cosa fondamentale che si può concludere dall'analisi di tutti questi dati è che le operazioni di convoluzione sono le operazioni computazionalmente più costose nelle reti neurali convoluzionali. Vista la loro complessità questo dato era prevedibile (anzi, è noto in letteratura).

I FC layer possono avere una fetta di tempo considerevole per le reti aventi un'architettura simile a quella di AlexNet, in quanto in questi casi esiste sempre un FC layer con un grande numero di parametri, il che implica più tempo speso sia per i calcoli che per operazioni di caricamento di dati in memoria. Nelle reti come GoogLeNet i FC ci sono, ma vengono utilizzati quando il volume di input è molto piccolo, quindi si ha un numero molto inferiore di parametri e di conseguenza meno tempo speso per la loro elaborazione.

I CCCP layer su NIN si prendono anche loro una buona porzione di tempo sul totale, ma d'altra parte sono sempre dei layer convoluzionali e quindi anche questo ha senso.

Per quanto riguarda tutti gli altri layer con percentuali molto piccole (ReLU, Concat, poi altri addirittura non inseriti perchè aventi percentuali troppo piccole come Dropout, Softmax ecc.), le loro basse percentuali sono dovute per certi casi alla scarsa presenza di layer del loro tipo nella rete, ma

principalmente alla semplicità delle loro funzioni, come ad esempio i layer ReLU.

Rimane da investigare il comportamento dei normalization layer, dato che a quanto pare con la rete GoogLeNet su PC hanno uno strano comportamento.

6.3.1 La normalizzazione di GoogLeNet su PC

Ricordando che Caffe utilizza gli LRN layer per normalizzare i dati, i quali fanno uso della formula 4.2 mostrata in precedenza, visti i dati ricavati per essi con GoogLeNet su PC, si è voluto partizionare ulteriormente i tempi all'interno di un layer di questo tipo per capire quale operazioni risultassero essere onerose e causare il fenomeno visto nella figura 6.5.

Normalmente in questo capitolo sono stati omessi i dettagli implementativi, ma in questo unico caso sarà fatta un'eccezione, dato che è necessario capire come si comporta il framework Caffe per comprendere i successivi grafici.

Per effettuare l'ulteriore suddivisione è stato necessario visionare e modificare leggermente il file *lrn_layer.cpp*, all'interno del framework di Caffe inserendo anche qui delle istruzioni per misurare i tempi voluti.

Innanzitutto è bene sapere che su Caffe c'è una fase preliminare di setup in cui vengono inizializzati tutti i layer della rete, normalization layer compresi. C'è un'apposita funzione all'interno del codice di un normalization layer che fa questo: effettua in pratica un check per vedere se esistono layer nella rete di questo tipo. Quindi i tempi di setup dei due normalization layer presenti su GoogLeNet avvengono cronologicamente prima dell'esecuzione dei layer di normalizzazione stessi. Dopodichè considerando una singola forward pass, il frammento di codice riguardante il calcolo della normalizzazione in modalità CPU (perchè all'interno vi è anche la parte riguardante la modalità in GPU), viene eseguito per ogni normalization layer presente nella rete quando diciamo "è il suo turno" durante la forward pass. Quindi seguendo sempre il nostro esempio, cioè la rete GoogLeNet, il codice viene eseguito esattamente due volte, con i parametri ricavati dal setup di ciascun layer.

Suddividerò ora in che modo vengono eseguite le primitive, o meglio i piccolissimi pezzi di codice, tenute in considerazione:

- **setup** è il setup iniziale del normalization layer eseguito per tutti i layer, normalization layer compresi, prima della loro esecuzione (come detto prima),
- **init** rappresenta l'inizializzazione dei valori per effettuare la normalizzazione,

- **caffe_sqr** identifica il calcolo di tutti i quadrati della formula 4.2,
- **first channel scale** indica il calcolo di una scala per il primo canale, cioè la prima depth slice del volume di input,
- **copy previous scale** indica la copiatura della scala calcolata precedentemente,
- **add head** identifica l'aggiunta di un offset; qui è presente la primitiva *caffe_axpy*. Questa primitiva teoricamente serve ad effettuare, operazioni del tipo $(\alpha * X[i]) + Y[i]$, dove X[i] ed Y[i] sono gli elementi di due vettori, uno contenente i quadrati calcolati in precedenza e l'altro i valori della scala, mentre α è uno dei parametri impostabili nel layer,
- **subtract tail** identifica la sottrazione di un offset, anche qui utilizzando *caffe_axpy* ma con parametri diversi al suo interno,
- **caffe_powx** effettua l'elevamento a potenza con esponente β della formula 4.2. In realtà essendo l'elevamento a potenza al denominatore nella formula in questione, viene effettuato un elevamento a potenza con esponente $-\beta$ per poter poi effettuare un prodotto invece di una divisione,
- **caffe_mul** effettua il prodotto fra i valori calcolati da *caffe_powx* ed i valori di input del layer.

Alcune delle operazioni considerate vengono effettuate solo una volta per l'intera esecuzione di un normalization layer, mentre altre si ripetono più volte. In particolare:

- **init**, **caffe_sqr**, **first channel scale**, **caffe_powx**, **caffe_mul** vengono eseguite una volta per ciascun layer di normalizzazione
- **copy previous scale**, **add head**, **subtract tail** vengono eseguite per tutti canali del volume di input, per ogni normalization layer. Il numero di canali che ci sono in totale dipende dal volume di input del layer precedente. Possono essere qualche decina o anche qualche centinaia. Stampando questi tempi per ciascuna iterazione, cioè per ogni canale, viene a crearsi un output di più di 2000 righe. Per questo motivo si è preferito, per ciascun layer di normalizzazione, considerare il totale del tempo speso, ovvero per tutti i canali, di ciascuna di queste operazioni.

Misurazione temporale	**Tempo (s)**
setup	0.000043
init	0.002309
sqr	0.000118
first channel scale	0.000021
copy prev. scale	0.000060
add head	0.000078
subtract tail	0.000080
powx	0.012986
mul	0.000120
Total time 1st normalization	0.015815

Tabella 6.4: Dati primo normalization layer di GoogLeNet su PC

Misurazione temporale	**Tempo (s)**
setup	0.000012
init	0.006183
sqr	0.000371
first channel scale	0.000023
copy prev. scale	0.000213
add head	0.000277
subtract tail	0.000227
powx	0.325750
mul	0.000373
Total time 2nd normalization	0.333429

Tabella 6.5: Dati secondo normalization layer di GoogLeNet su PC

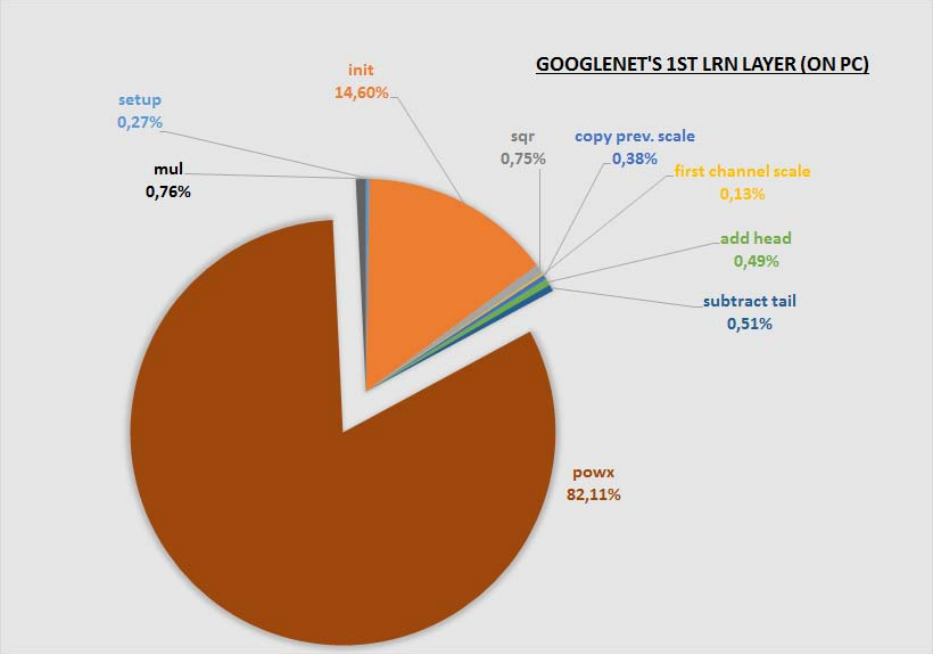


Figura 6.6: Ripartizione tempi primo normalization layer di GoogLeNet

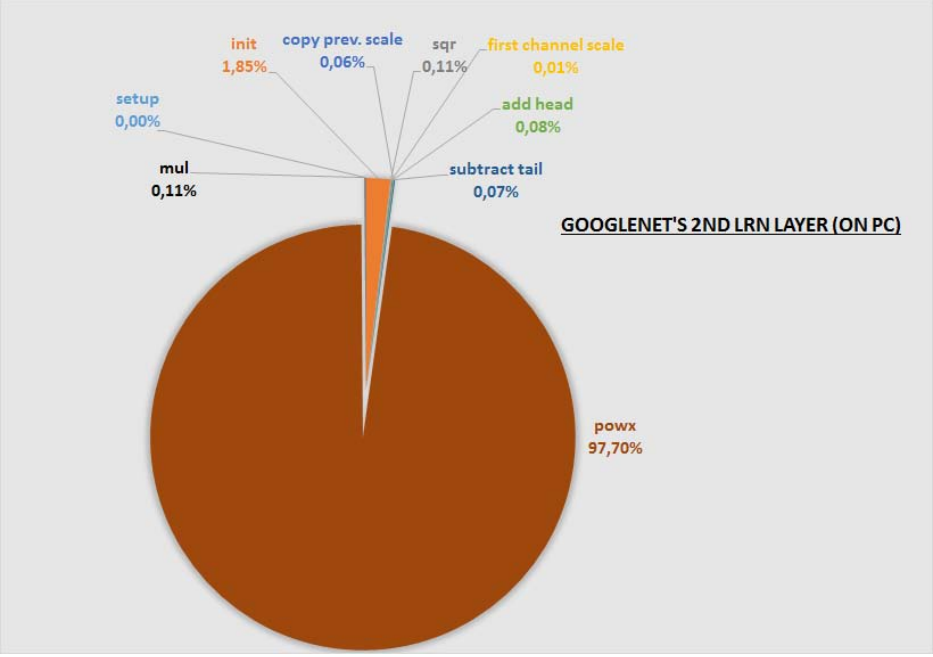


Figura 6.7: Ripartizione tempi secondo normalization layer di GoogLeNet

Layer	**Tempo PC (s)**	**Tempo Android (s)**
normalization layer 1	0.016	0.095
normalization layer 2	0.335	0.252

Tabella 6.6: Confronto normalization layer di GoogLeNet fra PC e Android

I risultati ottenuti per ciascuno dei layer di normalizzazione di GoogLeNet su PC sono mostrati nelle tabelle 6.4, 6.5 e nelle figure 6.6, 6.7.

Si evince dunque che più dell'80% del tempo di ciascuna esecuzione di un normalization layer di GoogLeNet su PC è dedicato all'operazione di elevamento a potenza con esponente β per la formula di normalizzazione 4.2. Addirittura sul secondo normalization layer questa operazione ha un tempo di circa 0.32 secondi su un totale di 0.33 secondi del layer. Si suppone che durante questo calcolo entrino in gioco altri fattori come ad esempio il non accurato utilizzo della memoria.

Considerando inoltre che su PC un'intera forward pass di GoogLeNet impiega in media 0.585 secondi (si veda tabella 6.2), si può notare che buona parte della percentuale di normalizzazione della figura 6.5(a) è dovuta al secondo normalization layer. Quindi l'operazione di elevamento a potenza rimane decisamente la più onerosa in entrambi i casi, ma nel primo normalization layer il suo tempo è molto più contenuto. Il tempo più contenuto probabilmente in parte è dovuto al fatto che, al momento dell'applicazione della prima normalizzazione, il volume in input risulta avere una profondità, cioè un numero di canali, nettamente inferiore rispetto al volume che si ha durante la seconda normalizzazione. Questo fatto lo si intuisce anche dal numero di righe di output ottenute all'interno del ciclo sui canali. Tuttavia confrontando i tempi medi ricavati in precedenza di ciascun normalization layer di GoogLeNet fra Android e PC (nella tabella 6.6 sono riportati i dati d'interesse), si vede che nella prima normalizzazione, nonostante *caffe_powx* costituisca un buon 80-90% del tempo totale per la normalizzazione stessa, il tempo su PC risulta comunque decisamente più piccolo rispetto a quello su Android, nello specifico 0.02 circa su PC e 0.09 circa su Android, cosa che invece non accade nella seconda normalizzazione. Quindi si ipotizza che con l'aumentare del numero di canali, dato che c'è un balzo di diverse centinaia di unità fra la prima e la seconda normalizzazione, questa operazione non sia diciamo propriamente efficiente su PC.

6.4 Considerazioni sul consumo energetico

Nei dispositivi mobile una delle limitazioni principali è rappresentata dal limite di energia che una batteria può fornire per svolgere tutte le funzioni desiderate. In questa sezione verranno illustrate delle stime di consumo per una singola predizione di un'immagine da parte delle reti finora considerate.

6.4.1 Metodo di misurazione

Il miglior metodo per misurare l'effettivo apporto di energia consumato durante una specifica funzione sarebbe quello di effettuare i rilevamenti direttamente sull'hardware del dispositivo durante lo svolgimento della funzione in questione; tuttavia il dispositivo Nexus 7, su cui sono stati effettuati tutti i precedenti test, possiede una batteria non estraibile e per questo motivo si è dovuto procedere con un test non invasivo.

Il metodo utilizzato prevede la misurazione della differenza di potenziale presente nel circuito, circuito che comprende il dispositivo collegato ad un normale alimentatore, durante l'operazione per cui si vuole effettuare un'analisi di tipo energetico. Il dispositivo deve essere a piena carica, in modo da limitare le variazioni di tensione che avvengono durante il processo di carica. Per effettuare le rilevazioni è necessario inoltre inserire una resistenza in serie al cavo di alimentazione. Il valore di questa resistenza non deve essere troppo grande, altrimenti potrebbe influenzare pesantemente il circuito: secondo K. M. Saipullah et al. [33] l'ammontare di potenza dissipata in tale resistenza non deve superare l'1-2% dell'intera potenza fornita dall'alimentatore. Si intuisce dunque che la resistenza deve essere molto piccola e per questo motivo è stato necessario metterne più di una in parallelo.

Una volta ottenuti i valori della differenza di potenziale v(t) in vari istanti temporali durante lo svolgimento del fenomeno, dividendo tali valori per la resistenza R applicata si ottengono i valori della corrente istantanea i(t), seguendo la legge di Ohm $I = \frac{V}{R}$.

Moltiplicando i valori della corrente per la tensione fornita dall'alimentatore, alla quale è stata sottratta la differenza di potenziale misurata sulla resistenza per escludere la potenza dissipata su di essa, si ottengono i valori della potenza istantanea p(t), seguendo la legge di Joule $P = V * I$.

A questo punto è sufficiente integrare la potenza sul tempo di durata del fenomeno per ottenere l'energia consumata da esso. Per il calcolo dell'integrale sono state utilizzate le formule del metodo dei trapezi, utilizzate sui dati raccolti in un foglio Excel.

Per il rilevamento della tensione è stato utilizzato un oscilloscopio Agilent DSO3102A fornito dall'università. L'oscilloscopio concesso permette di

	AlexNet	**NIN**	**GoogLeNet**
Energia consumata in una singola foward pass (J)	4.05	3.65	5.63

Tabella 6.7: Valori energetici di una singola predizione

salvare le proprie misurazioni in un supporto USB: è possibile il salvataggio delle misurazioni in un file *.csv* ed il salvataggio della forma d'onda in una classica immagine bitmap.

6.4.2 Applicazione del metodo

Per l'applicazione del suddetto metodo si è scelto di mettere una resistenza equivalente di 0.09 Ohm, costituita da 3 resistenze da 0.27 Ohm messe in parallelo. Su queste resistenze dovrebbe essere dissipato al massimo il 2% della potenza totale del circuito. Le resistenze sono state saldate sul cavo di alimentazione con una classica saldatura a stagno. Effettuando una misurazione diretta è stato visto che la resistenza in questione è pari a circa a 0.12 Ohm, aumentata per via di altri fenomeni quali ad esempio la resistenza del cavo stesso.

La tensione di alimentazione, misurata con un multimetro, è pari a 5.16 V.

Durante la misurazione, sono state ovviamente limitate le sorgenti di interferenza: il dispositivo Android è stato messo in modalità aereo e nessun'altra app era mantenuta in memoria. Lo schermo durante le prove è rimasto acceso, di conseguenza ai dati energetici ricavati è stata sottratto il valore medio di energia consumata dal dispositivo con lo schermo acceso e nessun altra attività per avere dati che riguardino puramente il fenomeno d'interesse. Nella pratica ciò si riconduce a sottrarre l'area mostrata in figura 6.8.

6.4.3 Risultati ottenuti

La tabella 6.7 riporta i valori energetici ottenuti durante una singola predizione di un'immagine, espressi in Joule, con ciascuna delle reti analizzate.

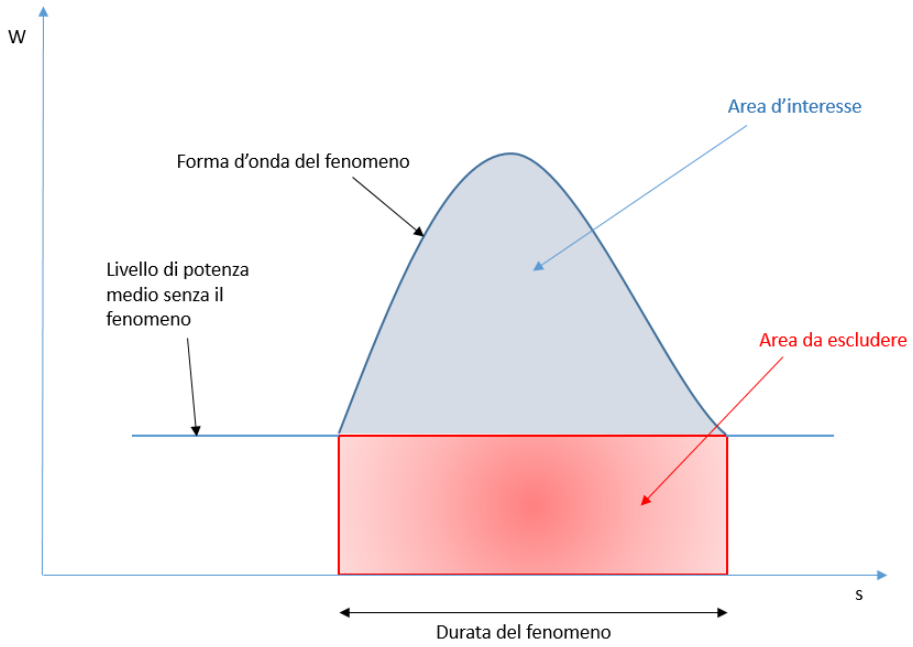


Figura 6.8: Calcolo dell'area riguardante l'energia d'interesse

6.4.4 Considerazioni sul consumo energetico

Il fenomeno, in base alle forme d'onde analizzate, dura circa 3 secondi per AlexNet e NIN, mentre dura circa 4 secondi per GoogLeNet. Questo fatto è sensato, in quanto se si osserva la tabella 6.3 si nota che AlexNet e NIN hanno un tempo di forward pass piuttosto simile, mentre GoogLeNet ci mette circa un secondo in più, probabilmente perchè GoogLeNet è decisamente più profonda delle altre due. Il consumo di energia sembra dunque essere collegato alla durata della forward pass di una rete.

Come rete neurale convoluzionale, NIN è leggermente più profonda di AlexNet, ma non possiede tuttavia nessun FC layer: come è stato visto in precedenza, normalmente questi layer presenti in reti con la struttura tipica di AlexNet, ovvero una serie di layer convoluzionali e una serie di FC layer, posseggono una pesante quantità di parametri e ciò molto probabilmente incide sul consumo energetico, in quanto potrebbero essere necessarie diverse operazioni legate ad un utilizzo di memoria più intensivo. Questa potrebbe essere una possibile spiegazione del fatto che in generale NIN consumi meno rispetto ad AlexNet.

Sul Nexus 7 utilizzato vi è una batteria da circa 3,950 mAh, circa 15 Wh, 54 KJ. Considerando quindi la batteria completamente carica, avendo a disposizione quindi circa 54KJ, e guardando i consumi di una singola

predizione di ciascuna delle reti, emergono i seguenti risultati:

- con AlexNet si stima di poter effettuare circa 13,300 predizioni
- con NIN si stima di poter effettuare circa 14,800 predizioni
- con GoogLeNet si stima di poter effettuare circa 9,600 predizioni

Si nota dunque che, nonostante il dispositivo sia limitato dal punto di vista energetico, esso permette comunque di effettuare un buon numero di predizioni e con delle reti piuttosto complesse. Bisogna inoltre tenere conto del fatto che il dispositivo è del 2013 e che molto probabilmente testato in un altro device più recente, magari anche con un hardware più avanzato, le stime potrebbero anche migliorare. Ciò fa riflettere sul fatto che il divario fra dispositivi mobili e PC si sta assottigliando sempre di più: negli ultimi anni sono stati fatti dei passi da gigante nel campo del mobile ed il futuro sembra proprio essere legato a questi dispositivi.

Quanto visto sembra inoltre indicare che una classificazione fatta in locale è più conveniente, per quanto riguarda il consumo energetico, rispetto ad una possibile classificazione fatta in remoto. Secondo Fatemeh Jalali et al.[34], effettuare l'upload di una singola immagine di circa 1.1 MB sono necessari ben 40 J tramite 4G e 23 J con il WiFi. L'immagine considerata in tutti i test visti finora pesa circa 800 KB, quindi non molto inferiore rispetto all'altra considerazione appena fatta, ma come si può notare il consumo energetico stimato risulta essere decisamente inferiore.

In conclusione il ridotto consumo energetico di un'elaborazione locale, cosa prevista dal progetto, risulta essere un punto a favore rispetto ad altre applicazioni simili allo progetto stesso, aventi magari lo stesso scopo ma senza l'utilizzo diretto di una rete neurale convoluzionale sul proprio dispositivo.

Capitolo 7

Implementazione del progetto

In questo capitolo verranno illustrati tutti i dettagli implementativi riguardanti la realizzazione del progetto. Alcune procedure relative al framework Caffe, sono già state spiegate nel relativo capitolo, e quindi in quei casi, per non essere ripetitivi, si segnalerà semplicemente il paragrafo contenente le informazioni per tali procedure.

7.1 Test iniziali con il framework

7.1.1 Installazione del framework

Esiste un'intera sezione del sito di Caffe dedicata alla sua installazione (`http://caffe.berkeleyvision.org/installation.html`). In base al sito l'installazione di Caffe è stata testata correttamente con i sistemi operativi Ubuntu, Red Hat e OSX. teoricamente è possibile installare Caffe sia in sistemi a 32 bit che a 64, tuttavia sono stati riscontrati diversi problemi durante l'installazione in un sistema a 32 bit. E' caldamente consigliato di installare sul proprio computer un sistema a 64 bit, dato che coloro che hanno progettato ed anche le persone che lavorano pesantemente con esso, dispongono di questo tipo di architettura. Fare inoltre attenzione al makefile: attualmente non risultano essere ben aggiornati e quindi nel caso si utilizzasse una qualche nuova versione di una delle tante librerie, di cui Caffe necessita per il suo corretto funzionamento, potrebbero sorgere problemi e quindi diventerebbe necessario effettuare alcune piccole modifiche al makefile. In questo caso ci sono stati alcuni problemi per quanto riguarda le versioni di OpenCV e della libreria Protobuf: nel primo caso, come accennato è stato sufficiente effettuare qualche modifica al makefile, mentre nel secondo caso, non trovando una soluzione, si è preferito scaricare ed installare una versione leggermente meno

aggiornata della libreria, ma più stabile. Per l'intero studio è stato utilizzato il sistema operativo Ubuntu 14.04.03 a 64 bit e come interfaccia di Caffe, come già specificato nel relativo capitolo, è stata utilizzata quella da riga di comando, tramite terminale (la più veloce e sicura) ed è stata utilizzata solo la modalità CPU, in quanto la scheda video posseduta non risulta essere compatibile con i driver CUDA.

7.1.2 I primi test e la tecnica del fine-tuning

Una volta studiato ed analizzato il framework di Caffe sono iniziati i primi effettivi test. Come specificato al termine del paragrafo 1.2.1, per prendere confidenza con il framework e per evitare possibili errori futuri, questi primi test non sono stati fatti sul dataset finale, il quale in questa fase doveva ancora essere creato, bensì su altri dataset, velocemente assemblati o scaricati con il solo scopo di fare pratica. In questa primissima fase tuttavia, non si possedevano ancora tutte le conoscenze illustrate nel capitolo 5: semplicemente sono stati effettuati dei test per cominciare a vedere come si comporta questa tipologie di reti neurali.

I primissimi test sono stati effettuati prendendo come dataset un piccolo sottoinsieme di classi del dataset Caltech-256. Nonostante sia stato chiarito che in questo progetto si è utilizzata la tecnica del fine-tuning per allenare una rete, in questi test essa non è stata applicata, perchè al momento di attuazione non era ancora chiaro l'utilizzo della suddetta tecnica. Era stata costruita una piccola rete, con una classica semplice struttura del tipo layer convoluzionali + FC layer, di dimensioni molto ridotte e si era semplicemente tentato di allenarla da zero, settando una determinata configurazione dei parametri, secondo le conoscenze possedute in quel momento. Con questo metodo non si sono mai ottenuti dei buoni risultati: forse non erano ancora state sviluppate le competenze necessarie, ma molto probabilmente il principale problema era il fatto che le iterazioni fatte nella fase di allenamento (qualche migliaio, tenendo il computer acceso per una giornata) erano troppo poche per una rete allenata da zero. Infatti il valore della funzione di loss diminuiva ma in maniera lentissima e partendo da valori piuttosto elevati. Quindi si comprese che questo tipo di allenamento risultava essere troppo costoso in termini di tempo: si poteva anche tentare di allenare una rete per decine di migliaia di iterazioni, ma niente garantiva che al termine dell'allenamento la rete avrebbe avuto buone prestazioni. Raramente in questi casi capita che un allenamento vada bene al primo tentativo. E' sufficiente il settaggio non corretto anche di un solo parametro per dover ripetere l'intero allenamento. Con un hardware adeguato, i tempi si sarebbero certamente ridotti ed il tutto avrebbe potuto diventare fattibile, ma purtroppo non era

questo il caso. Dato il limitato hardware a disposizione bisognava cercare un'altra strategia. Dopo un ulteriore approfondimento la nuova strategia sembrava avere trovato un nome: fine-tuning di una rete. Tempi molto ridotti di allenamento, sicurezza per via dell'utilizzo di reti robuste, famose per le loro prestazioni, e la presenza di un loro modello già allenato con numero molto elevato di iterazioni, rendevano questa tecnica molto interessante. Quindi è stata studiata nel dettaglio ed implementata sullo stesso dataset menzionato prima: a parità di tempo, rispetto ai test precedenti, i risultati sono stati decisamente migliori dei precedenti. Come rete è stata inizialmente scelta la rete AlexNet, in quanto risultava essere architetturalmente la più semplice; le altre reti al tempo risultavano ancora ignote oppure non ancora completamente comprensibili.

7.1.3 Ulteriori test su un dataset provvisorio

Trovata la tecnica per allenare una rete, successivamente è stato creato un primo provvisorio dataset. La procedura per la creazione di un dataset per Caffe, universale per tutti i test effettuati, è la stessa indicata nel paragrafo 4.3, precedentemente illustrata. Come si può notare da tale paragrafo, sono stati realizzati alcuni script ad-hoc per velocizzare ed ottimizzare l'organizzazione e la costruzione dei file necessari a Caffe per leggere le immagini del dataset.

Per facilitare le predizioni il dataset possedeva 4 classi di immagini molto diverse fra loro. Le classi erano:

1. **disegno**. La classe conteneva immagini di disegni fatti da dei bambini, trovate su Internet.

2. **modulo 730**. La classe conteneva immagini del modulo 730, anche queste reperite su Internet.

3. **monitor**. La classe conteneva immagini di monitor, in parte trovate su Internet, in parte prese da altri dataset minori.

4. **volto**. La classe conteneva immagini di volti di persone, prese da alcuni dei molti dataset utilizzati per il riconoscimento facciale che si trovano in rete.

Le classi dei disegni e del modulo 730 contenevano circa 200 immagini ciascuna, dato che per esse era difficile reperirne di più su Internet, mentre le altre contenevano ognuna circa 800 immagini. C'era dunque uno sbilanciamento nel numero di elementi delle classi e questo è noto dal Data Mining

che può essere un problema perchè le classi più frequenti vengono tendenzialmente favorite nella classificazione. Per evitare problemi era necessario o rimuovere immagini dalle classi più numerose oppure aggiungere ulteriori immagini nelle classi minori. Rimuovere immagini tuttavia significava dare globalmente meno informazioni alla rete e quindi si è preferito stampare alcune immagini, armarsi di uno smartphone e scattare un buon numero di foto per le classi minoranti. Alla fine tutte le classi risultavano avere un numero di elementi piuttosto simile.

Con circa 1000 iterazioni, il fine-tuning di Alexnet su questo dataset ha prodotto una top-1 accuracy del 95% circa, confermando i buoni risultati di questa tecnica con un numero relativamente basso di iterazioni, anche se il dataset considerato era abbastanza semplice, con poche classi e tutte ben distinte fra loro, ed inoltre era stato fornito di un buon numero di immagini; quest'ultimo fattore è risultato essere molto importante. Oltre a questo, il nuovo modello della rete ottenuto dal fine-tuning è stato testato, con esiti quasi tutti positivi, su circa 50 immagini esterne al dataset.

7.2 Il dataset PIPPI

Visti i buoni risultati ottenuti con il dataset precedente, si è deciso successivamente di costruire un dataset più difficile, ovvero avente alcune classi piuttosto simili fra loro. Il dataset costruito contiene in parte una tipologia di dati che viene utilizzato dall'applicazione per il programma PIPPI (si veda paragrafo 1.2.8), sviluppata all'Università di Padova. Si voleva avere questa volta una situazione ancora più reale per l'applicazione di quanto visto finora e direttamente collegata all'intero scopo della tesi, ovvero la classificazione, tramite reti neurali convoluzionali, di immagini su un dispositivo mobile, immagini che in questo contesto riguardano l'applicazione del programma PIPPI (con la quale ipoteticamente potrebbe essere un giorno essere integrato il classificatore automatico realizzato).

Questo dataset considerato risulta essere quello finale per lo scopo progetto.

7.2.1 Struttura e proprietà del dataset

Il nuovo dataset possiede 7 classi:

- **consenso informato**. Un particolare tipo di modulo utilizzato dall'applicazione per il programma PIPPI. La figura 7.1 mostra una delle immagini nel dataset di questa classe.

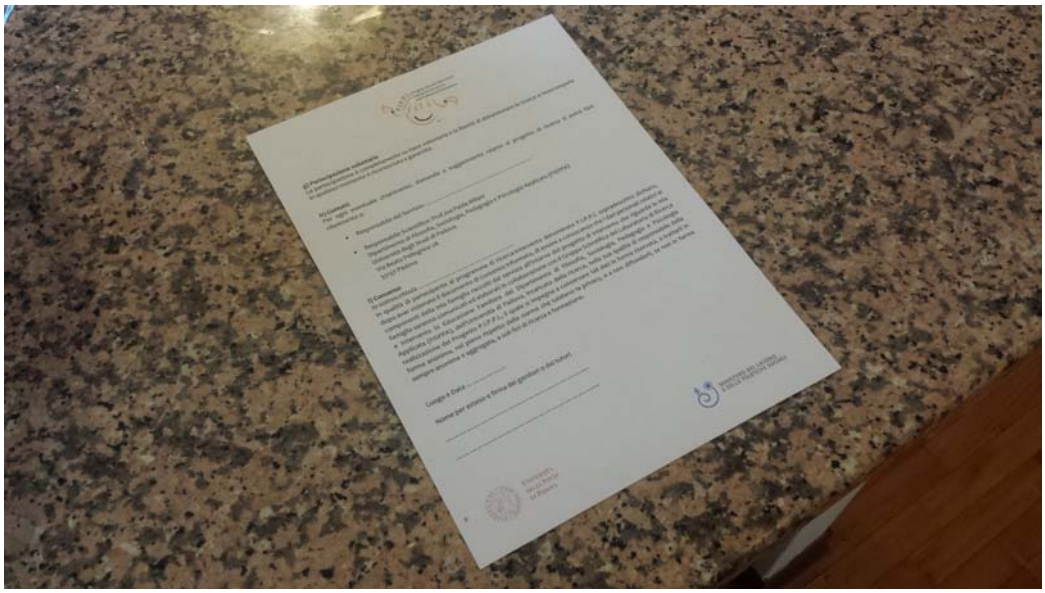


Figura 7.1: Esempio di consenso informato

- **disegno**. Stessa classe che era presente nel dataset precedente. E' stata conservata in quanto risulta essere in linea con gli interessi dell'applicazione PIPPI.
- **ecomappe**. Un particolare tipo di disegno che ha a che fare con l'applicazione del programma PIPPI. In figura 7.2 è presente un esempio.
- **modulo 730**. Stessa classe del dataset precedente. E' stata lasciata per rendere più difficile la selezione fra questo modulo e quello del consenso informato.
- **triangolo bambino**. Un particolare tipo di disegno che che ha a che fare con l'applicazione del programma PIPPI. Si guardi figura 7.3 per vederne un esempio. I piccoli riquadri colorati vengono detti "nuvole".
- **triangolo operatore**. Disegno che differisce leggermente dal triangolo bambino per via delle diverse parole presenti all'interno delle nuvole (figura 7.4).
- **triangolo vuoto**. Disegno che differisce leggermente dai due precedenti in quanto all'interno delle nuvole non vi è alcuna scritta stampata. Si veda figura 7.5.

Come preannunciato, in questo dataset sono presenti gruppi di classi simili fra loro, il che rende il compito della classificazione più difficile rispetto ai test

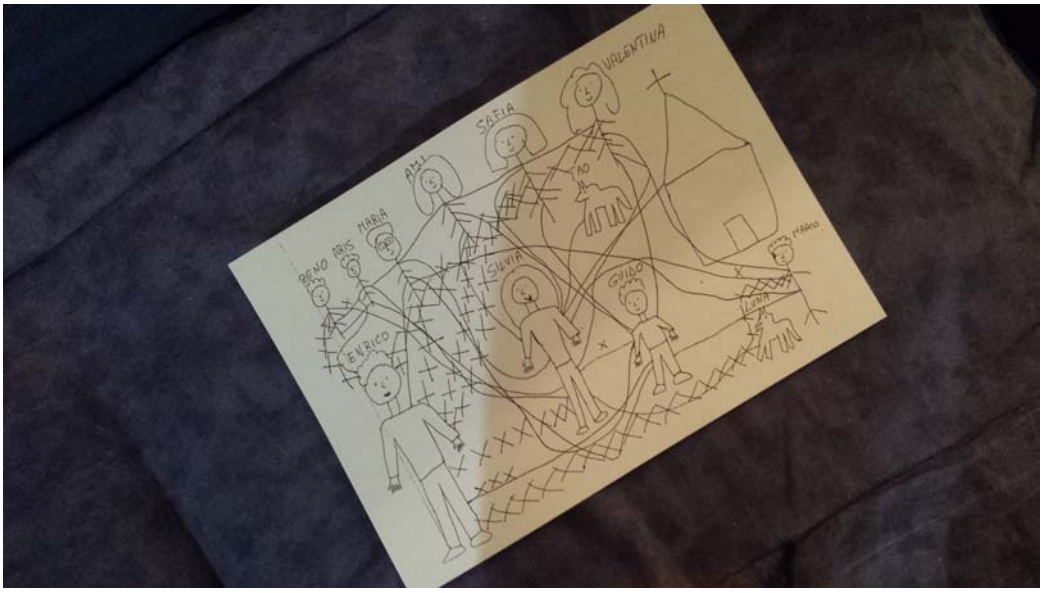


Figura 7.2: Esempio di ecomappa

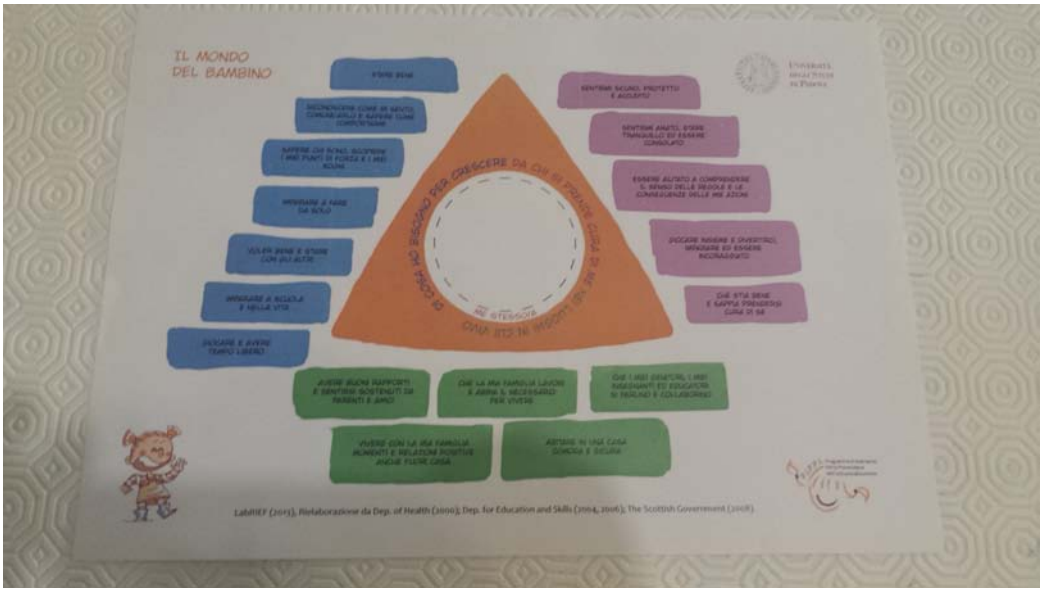


Figura 7.3: Esempio di triangolo bambino

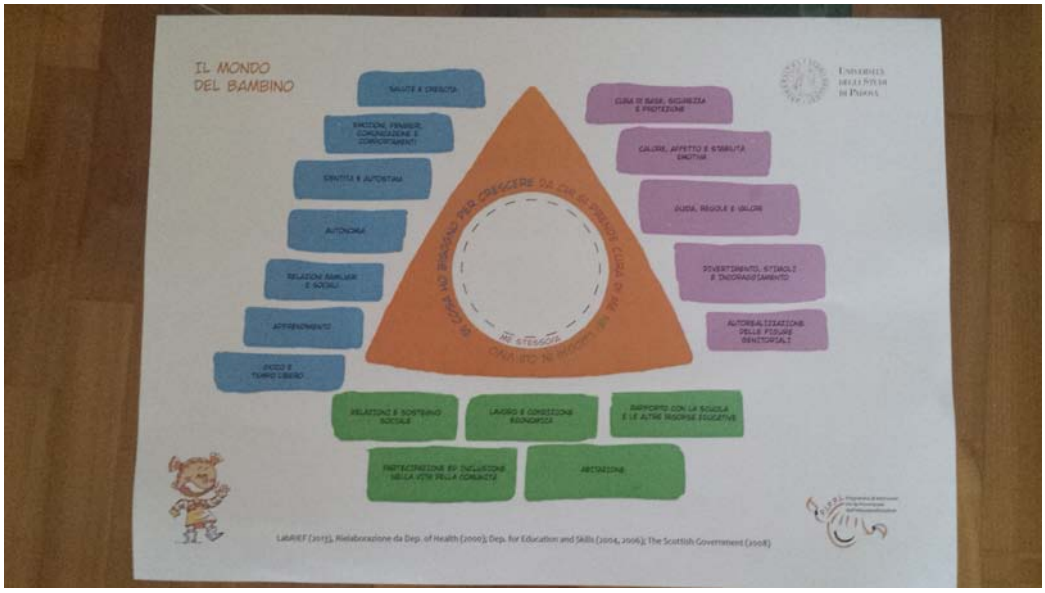


Figura 7.4: Esempio di triangolo operatore

Figura 7.5: Esempio di triangolo vuoto

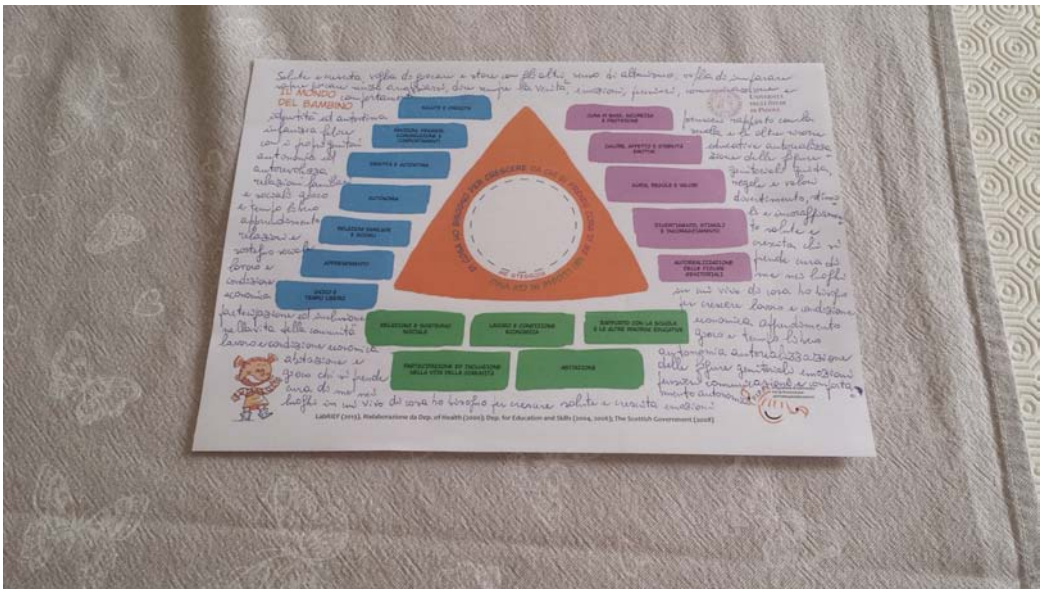


Figura 7.6: Esempio di triangolo operatore compilato

precedenti. In particolare le tre classi riguardanti i triangoli sono veramente molto simili fra loro e quindi di sicuro alzano di molto il livello di difficoltà nella distinzione fra di loro.

Ogni classe di questo dataset è stata dotata di circa 1000 immagini nel training set e circa 50 immagini nel validation set: le classi disegno e modulo 730 sono state semplicemente estese rispetto a prima, aggiungendo più foto; le rimanenti classi sono invece state costruite da zero, effettuando quindi circa 1000 foto per ognuna di esse. Il materiale (consenso informato, ecomappe, triangoli vari) è stato gentilmente fornito dal professor Carlo Fantozzi e dal Dott. Marco Ius. I triangoli, di tutti e tre i tipi, in situazioni normali sono compilati o da un bambino o da un operatore. Per questo motivo alcuni di essi sono stati compilati a mano, in modo che la rete di adatti anche a questa situazione. La figura 7.6 mostra un triangolo operatore compilato. Inoltre per essi sono state considerate sia delle stampe a colori che in bianco e nero: nel dataset sono dunque presenti foto di entrambe le tipologie, dato che nella realtà possono sussistere entrambi i casi.

Purtroppo non avendo a disposizione molte immagini di ecomappe, la relativa classe, nonostante possegga comunque quasi 1000 immagini, non risulta essere varia come invece dovrebbe essere, come ad esempio la classe dei disegni. Un possibile miglioramento del dataset potrebbe essere apportato nel caso si avesse quindi a disposizione più immagini di questa categoria.

Per ogni classe presente nel dataset, gli sfondi utilizzati nel validation set non sono mai stati utilizzati nei corrispondenti training set. Questo è stato fatto con lo scopo di rendere la classificazione all'interno del validation set più neutrale, non cioè in alcun modo influenzabile dai vari sfondi con cui la rete è stata allenata. Ciò in teoria dovrebbe rafforzare il riconoscimento dell'oggetto voluto all'interno delle immagini.

Per ciascuna classe inoltre, sono state scattate foto da molteplici angolazioni e con molteplici distanze dall'obbiettivo, al fine di rendere più robusta la rete e dato che non si può sapere con precisione con che angolatura un utente che utilizza uno smartphone o un tablet può inquadrare gli oggetti in questione.

Nel dataset sono presenti anche situazioni di luminosità differenti, ed anche situazioni in cui vi sono ombre direttamente sulla superficie degli oggetti, per tentare di abituare la rete a classificare anche questi casi.

7.3 Allenamento delle reti

Una volta ultimato il dataset PIPPI, si è proseguito con l'allenamento di ciascuna delle tre reti selezionate, tramite l'operazione di fine-tuning, su tale dataset. Per poter effettuare questo allenamento è stato necessario modificare alcuni parametri di ciascuna rete, sia nella loro definizione che nei file di solver. L'intera procedura eseguita per ciascuna rete è la stessa illustrata precedentemente nel paragrafo 4.7.

Per ottenere dei buoni modelli sono state necessarie 2000 iterazioni per AlexNet, 1000 per NIN e 1000 per GoogLeNet. In realtà sono state effettuate fino a 3000 iterazioni per ciascuna rete, controllando la relativa accuracy di tanto in tanto. Con un numero più elevato di iterazioni rispetto a quelli prima indicati, è stato visto che l'accuracy cominciava a diminuire, probabilmente perchè cominciava ad aumentare l'overfitting. A parità di iterazioni, GoogLeNet è stata la rete che ha richiesto un maggior tempo di allenamento, quasi il doppio rispetto alle altre due reti, principalmente a causa della sua complessa architettura.

Ottenuti dei modelli presumibilmente buoni, questi sono stati testati su un ulteriore piccolo test set , appositamente creato per verificare le prestazioni di ciascuna rete. I dati relativi alle performance vengono rimandati al capitolo successivo.

7.4 Porting di Caffe su Android

Dopo aver testato con successo Caffe su PC, le stesse reti viste finora sono state testate con esito positivo anche sulla piattaforma Android. L'analisi e la selezione delle reti fatta in precedenza ha sicuramente aiutato ad escludere eventuali reti che sarebbero risultate troppo pesanti per funzionare su un dispositivo mobile, in quanto esso comunemente risulta avere caratteristiche hardware inferiori rispetto ad un normale PC e quindi in generale ci si aspetta un aumento dei tempi analizzati finora.

Il problema di utilizzare Caffe su Android è che non c'è molto materiale reperibile a riguardo; infatti attualmente non esiste una versione ufficiale di Caffe per questa piattaforma, bensì solo alcuni makefile creati da singoli individui come ad esempio il progetto dell'utente *darrenl tzutalin* ([28]) oppure dell'utente Shiro Bai ([27]), i quali hanno generosamente messo a disposizione il loro lavoro, a patto di segnalare, nel caso se ne facesse uso, il loro Copyright. Certamente poi ognuna di queste versioni solitamente presenta diversi problemi di funzionamento, in quanto sicuramente non sono state sufficientemente controllate e testate, dato che i loro autori non ne avevano probabilmente la necessità. La difficoltà stà dunque nell'individuare, nel corso della ricerca, quale fra i porting esistenti risulta essere il più completo e nel tentare di modificarlo per fare come minimo funzionare il porting e magari successivamente migliorarlo nel complesso.

L'alternativa sarebbe stata quella di creare da zero un makefile generale per l'intero porting di Caffe, ma ciò avrebbe richiesto una buona conoscenza di tutte le librerie di cui Caffe ha bisogno, oltre ad una buona dose di tempo per assemblarle tutte assieme e testare il loro funzionamento. Per questi motivi è stato preferito prima cercare di verificare la presenza di un qualche porting già esistente, per poterlo così studiare, testare e nel caso qualcosa non funzionasse tentare alcune modifiche per avere un esito opposto. Fortunatamente, dopo varie ricerche e test negativi di makefile, ne è stato individuato uno che, con modifiche minime, ha permesso di utilizzare Caffe su Android. Il porting in questione appartiene ad un progetto del 2015, creato da Shiro Bai, chiamato *caffe-android-lib*. Nella sezione sottostante riporto le parti cruciali del porting assieme alle piccole modifiche effettuate per avere un corretto funzionamento.

Al fine di poter effettivamente far funzionare Caffe su Android, è stato necessario individuare quali sono le librerie necessarie per un suo corretto funzionamento su tale piattaforma, sempre in modalità CPU. La modalità GPU richiederebbe ulteriori librerie e, nonostante credo che sicuramente nel corso del tempo si riuscirà ad avere un porting anche in questa modalità (alcuni sostengono che bisognerà riscrivere buona parte del codice di Caffe per

questo scopo), al momento non risulta esserci nulla di veramente funzionante in questo contesto. Inoltre, dato che anche sul PC è stato possibile testare solo la modalità CPU, così sarà possibile successivamente fare un confronto fra le due piattaforme utilizzando la stessa modalità.

Tornando alle librerie per la modalità CPU, guardando svariati tentativi di porting è risultato che esse sono:

- le librerie Boost ([29])
- le librerie Eigen ([30])
- le librerie OpenCV ([31])
- la libreria Protobuf ([32])

Il porting selezionato non fa altro che scaricare una determinata versione di ciascuna delle librerie selezionate ed organizzarle all'interno di una cartella. Sempre all'interno della medesima cartella è necessario che siano presenti anche i file sorgenti di Caffe (possono essere direttamente scaricati da Internet, oppure copiati dal proprio PC nel caso si avesse Caffe già installato).

Oltre ad i file sorgenti di Caffe, nel porting sono presenti due ulteriori script C++. chiamati *caffe_mobile.cpp* e *caffe_jni.cpp*: il primo dei due file contiene il codice con le chiamate dirette alle apposite funzioni di Caffe; i metodi contenuti in *caffe_mobile.cpp* saranno invocati da *caffe_jni.cpp*; all'interno di *caffe_jni.cpp* ci sono sostanzialmente tre metodi (più altri piccoli metodi ausiliari), che sono i principali metodi che possono essere chiamati da Android tramite la JNI:

- **enableLog()** serve per collegare l'output che Caffe normalmente farebbe apparire nel terminale di Linux direttamente sul LogCat di Android. Questo metodo risulta utile ad un programmatore, una volta importato Caffe su Android positivamente, per capire se ci sono eventuali errori riguardanti il framework stesso.
- **loadModel()** serve il caricamento del modello di una rete
- **predictImage()** esegue il restante codice per effettuare una predizione di un'immagine data in input

I file appena menzionati sono in pratica l'equivalente dello script *classification.cpp* visto nella sezione 4.9. Tralasciando il metodo *enableLog()*, una cosa da notare è che, a differenza dello script *classification.cpp*, qui si ha una separazione fra il caricamento del modello di una rete e la successiva predizione. In questo modo è possibile separare le due fasi all'interno del codice

Java del progetto Android: basta effettuare le chiamate ai metodi in questione quando se ne ha la necessità e dunque risulta possibile personalizzare ancora di più l'applicazione, valutando in quali istanti conviene effettuare le due operazioni. In questo modo si rimane in linea con quanto detto a questo riguardo nella fase di progettazione.

Dato che Caffe è scritto in C++ e così pure i metodi per la classificazione di un'immagine appena visti, è ovviamente necessario utilizzare Android NDK per il porting su Android. Si ricorda che Android NDK serve per importare su Android del codice nativo utilizzando la Java Native Interface (JNI), esattamente come è stato fatto in questo caso. E' indispensabile dunque comprendere come esso funziona in generale: la documentazione di Android (`http://developer.android.com/tools/sdk/ndk/index.html`) fornisce a mio avviso una spiegazione esaustiva a questo riguardo. Per utilizzare Android NDK, come si può vedere sempre dalla documentazione Android, è necessario definire due ulteriori file, fortunatamente già presenti all'interno del porting, ovvero *Android.mk* ed *Application.mk*; il secondo in realtà in generale è opzionale, ma in questo caso risulta essenziale.

Il primo di essi serve per specificare quali sono e dove si trovano le librerie ed i file in codice nativo da importare su Android; aprendolo si può notare una divisione in due moduli, indicati dalla keyword *LOCAL_MODULE*, ovvero "*caffe*" e "*caffe_jni*". Questi due moduli creano due diverse librerie *.so*, in particolare **caffe.so** e **caffe_jni.so**: la prima contiene il codice generale di Caffe, ovvero i principali file sorgenti, i file dei layer, ecc., mentre la seconda riguarda invece i file *caffe_jni.cpp* e *caffe_mobile.cpp* visti prima. Qui è stato necessario apportare alcune modifiche, in quanto la versione di Caffe utilizzata di default dal porting risultava allora un po' datata e quindi alcuni file sono stati aggiunti ed altri sostituiti, in linea con le continue modifiche che sono state fatte nel corso del tempo al framework di Caffe, framework in continua evoluzione. Alcuni layer di base non erano presenti, come ad esempio alcuni tipi di data layer: questo probabilmente perchè l'autore ha inserito giustamente solo i layer principali di cui ha fatto uso a suo tempo; ad ogni modo è sufficiente avere i relativi script sul proprio PC e integrarli su *Android.mk* per poterli utilizzare (è facile vedere dove inserirli una volta aperto il file). Nel caso si facesse uso di un qualche layer non presente in questo file, Caffe naturalmente genererà un errore.

Il file *Application.mk* specifica invece ulteriori informazioni sull'architettura in cui le librerie *.so* dovranno funzionare. E' stato dunque necessario modificare anche questo file, indicando la versione della piattaforma Android e di NDK Toolchain corrette.

Uno script in Python, *build.py*, raggruppa tutte le operazioni descritte finora; è dunque sufficiente eseguire tale file, indicando solamente il path

della cartella principale dove è presente Android NDK per effettuare sequenzialmente tutto quanto visto fino ad adesso, ottenendo direttamente le due librerie *.so* finali. All'interno di questo file, sono state effettuate diverse modifiche, la maggior parte delle quali sempre riguardanti la specifica delle versioni corrette delle varie librerie e la correzione dei vari path per identificarle. Attualmente come versione di OpenCV è stato lasciata la 2.4.11, la più avanzata con cui Caffe non dà alcun tipo di problema. Molto probabilmente funzionerebbe anche la versione 3.0 o la 3.1, ma questo richiederebbe, oltre ad un ulteriore modifica dell'ultimo file Python visto, anche di una buona parte dei makefile di Caffe stesso, in quanto non aggiornati per queste versioni di OpenCV, all'interno delle quali sono cambiate abbastanza cose rispetto alle versioni precedenti, come ad esempio la collocazione di alcune specifiche funzioni che Caffe usa. Per evitare ulteriori problemi e rallentamenti, momentaneamente si è deciso di lasciare questa versione. In futuro, con un po' di pazienza, non dovrebbe comunque essere impossibile aggiornare il tutto ad una versione più avanzata di OpenCV.

Nel repository condiviso con il professore in cui sono stati collocati tutti file necessari per l'utilizzo di Caffe su Android, è stato comunque creato un file README in cui vengono descritti nel dettaglio tutti i passi necessari per far funzionare Caffe su Android, alcuni dei quali sono già stati visti fin qui mentre i rimanenti verranno illustrati fra poco.

7.5 Applicazione Android

All'interno dello stesso progetto di porting, è presente anche una "bozza" di applicazione Android per chiunque volesse testare il funzionamento di Caffe importato su Android. Di base essa non funzionava ed è stata dunque modificata ed arricchita con ulteriori aggiunte al codice. Il risultato finale è un'applicazione che fondamentalmente permette di fare due cose:

1. scattare una foto e successivamente classificare tramite Caffe l'immagine risultante di tale foto

2. caricare dall'archivio del dispositivo Android un'immagine e classificarla con Caffe

Come risultato l'applicazione mostra sullo schermo l'immagine classificata assieme ad un'etichetta, nello specifico una TextView, che indica la classe risultante dal processo di classificazione. Chiamarla applicazione è forse un po' esagerato, in quanto praticamente essa è costituita da una singola activity, oltre ai file/librerie descritti nella sezione precedente; più specificatamente

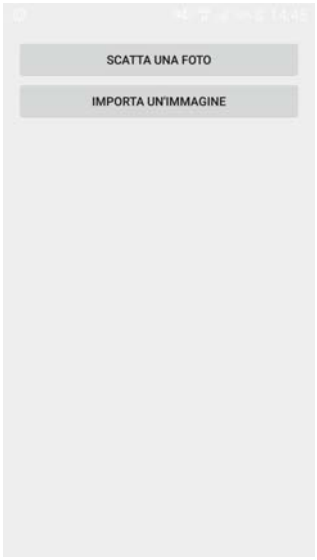


Figura 7.7: Interfaccia basilare app

essa costituisce una bozza di possibile modulo per una qualsiasi applicazione esistente che richieda un'operazione di questo tipo. Tale modulo può poi essere ulteriormente ampliato, aggiungendo altri moduli che effettuano altre operazioni successive al riconoscimento di un'immagine, come ad esempio il riconoscimento di testo all'interno dell'immagine, esattamente come visto nelle pianificazioni iniziali. Per questi motivi, in linea con quanto detto durante la fase di progettazione, non si è dato molto peso alla grafica dell'applicazione stessa, in quanto ci si vuole focalizzare sul corretto funzionamento di Caffe su Android ed analizzare in seguito le prestazioni delle reti. L'interfaccia utente può essere applicata in maniera esaustiva una volta che saranno state effettuate svariate scelte, come ad esempio il dataset su cui si deve costruire il modello con Caffe che indica sostanzialmente anche l'ambito di utilizzo dell'applicazione stessa, la rete con cui effettuare le predizioni e cosa fare inoltre una volta che la predizione viene completata. L'activity si presenta dunque con un'interfaccia utente minima, ovvero uno sfondo bianco con due soli pulsanti, uno per ciascuna delle due modalità descritte prima, ed eventuali messaggi su schermo, cioè dei Toast, per notificare cosa sta facendo l'applicazione. La figura 7.7 mostra quanto appena detto.

7.5.1 Organizzazione file necessari

Innanzitutto, per il corretto funzionamento dell'applicazione stessa, è necessario che alcuni file relativi ad una specifica rete di Caffe siano presenti sul

dispositivo Android. I file in questione sono gli stessi visti su PC per effettuare la predizione di un'immagine esterna con una rete convoluzionale, ovvero il file del solver, il file di deploy, il file del modello della rete allenata (*.caffemodel*) e, se utilizzato, anche il file *.binaryproto* per l'operazione di mean subtraction, anche se volendo è possibile integrare le informazioni per il mean subtraction direttamente sul file di deploy, senza utilizzare un ulteriore file.

E' inoltre necessario il file di synset, contenente le informazioni delle classi del dataset.

Tutti questi file possono essere allocati sul dispositivo in una locazione a piacere, basta poi indicare il path corretto all'interno del codice dell'applicazione al momento del loro utilizzo. Di default il file di synset, che è indipendente dal tipo di rete utilizzata, viene inserito all'interno di una cartella chiamata *assets* all'interno dell'applicazione stessa, mentre tutti gli altri file relativi alla rete vengono raggruppati all'interno di un'altra singola directory che dovrà poi essere copiata sul path di default "*/sdcard/caffe_mobile/nome_directory_rete*" (questo path è comunque configurabile). Un semplice metodo per inserire i file della rete all'interno del dispositivo è quello di utilizzare il tool adb dal terminale: dando ad esempio il comando

adb shell mkdir -p /sdcard/caffe_mobile/

si crea sul dispositivo la cartella *caffe_mobile* sul percorso voluto, mentre con il comando

adb push GoogLeNet/ /sdcard/caffe_mobile/GoogLeNet

si può ad esempio copiare l'intero contenuto della cartella *GoogLeNet* creata sul PC, dentro alla quale si suppone ci siano tutti i file della rete descritti prima, sul relativo path del dispositivo indicato dal comando. Dato che si prevedeva di testare tutte e tre le reti selezionate, sono state dunque aggiunte tre directory, una per ciascuna rete, all'interno del path "*/sdcard/caffe_mobile*", in modo da avere già tutti i dati pronti; modificando poi leggermente i path all'interno dell'applicazione si può dunque accedere alla rete voluta.

Oltre a tutto questo, è ovviamente necessario includere nell'applicazione le librerie *.so* precedentemente create.

7.5.2 Funzionamento dell'applicazione

Una volta predisposto tutti i file necessari, si può avviare l'applicazione. La prima cosa obbligatoria da fare consiste nell'importare le librerie *.so*. Il codice per fare questo si può trovare nella documentazione Android e la figura 7.8 ne porta un relativo esempio.

```
//importazione delle librerie .so, create precedentemente con Android NDK
   static {
       System.loadLibrary("caffe");
       System.loadLibrary("caffe_jni");
   }
```

Figura 7.8: Importazione librerie .so

All'avvio dell'applicazione, quindi all'interno del metodo *onCreate()* del ciclo di vita di un'app di Android, il modello della rete scelta viene caricato tramite il metodo *loadModel()*, indicando come parametri i path dei file della rispettiva rete. Al termine di questa operazione verrà mostrato un Toast che indica l'esito positivo del caricamento dei dati.

Sempre in automatico all'interno del metodo *onCreate()* viene inoltre letto il file di synset, memorizzando su un array le informazioni delle classi da esso specificate. Il file di synset deve essere organizzato con lo stesso formato standard che si utilizza su Caffe, ovvero ogni riga deve essere del tipo *etichetta_classe nome_classe*, altrimenti l'applicazione andrà in crash quando tenterà di estrarre i dati da questo file. Il file di synset è un normale file di testo, quindi non ci sono grossi problemi per ricavare le informazioni che esso contiene tramite Android.

Dopodichè, come già visto, l'utente ha a disposizione due pulsanti per scattare una foto oppure caricare un'immagine da file. Nel primo caso, un intent fa partire l'applicazione della fotocamera del dispositivo, mentre nel secondo caso tramite un altro intent viene aperta una galleria delle immagini presenti sul dispositivo. Con entrambe le modalità, l'immagine finale viene momentaneamente salvata sul dispositivo all'interno di una cartella temporanea chiamata *Temp*. Lo scopo di questo è quello di poter salvare a parte l'URI di questa immagine, indipendentemente da che modalità l'utente abbia scelto, il quale potrà poi essere mantenuto anche in caso di distruzione dell'activity. L'URI di tale immagine dovrà poi essere passato al metodo *predictImage()*.

Quest'ultimo metodo, riguardante la predizione di un'immagine con Caffe e cioè il tempo di forward pass della rete, richiede un tempo abbastanza considerevole su un dispositivo Android (come visto nel capitolo 6), dell'ordine di qualche secondo. E' dunque necessario su Android effettuare un'elaborazione in background, durante la quale all'utente viene notificato un messaggio di attesa per elaborazione in corso. L'applicazione infatti utilizza una classe interna privata chiamata *CNNTask* che estende *AsynchTask*, la quale serve proprio per questo scopo. Si ricorda che all'interno di una classe che estende *AsynchTask* devono essere presenti i metodi *doInBackground()*, che

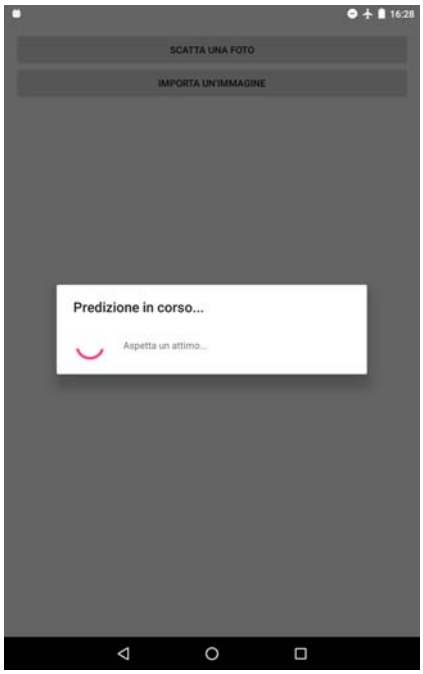


Figura 7.9: Immagine in elaborazione

contiene il codice per l'elaborazione in background, ed il metodo *onPostExecute()*, che contiene invece il codice da eseguire al termine dell'elaborazione in background. Come si può intuire il metodo della JNI *predictImage()* viene invocato all'interno del metodo *doInBackground()*, mentre il metodo *onPostExecute()* riceve in ingresso un intero che rappresenta l'indice della classe risultante predetta dalla rete; tale indice è relativo all'array contenente le informazioni del file di synset letto in precedenza. Sempre all'interno di *onPostExecute()* viene chiamato un ulteriore metodo nominato *onTaskCompleted()* per effettuare le ulteriori elaborazioni che verranno fra poco descritte, passando sempre l'intero con l'indice della classe predetta. Come già accennato, durante questo intero processo, all'utente viene mostrato un classico messaggio di attesa (figura 7.9).

Il metodo *onTaskCompleted()* utilizza l'intero fornitogli per ricavare le informazioni prese inizialmente dal file di synset all'interno dell'array da esso ricavato, in pratica la label della classe predetta. Queste informazioni vengono successivamente mostrate tramite una TextView sulla UI, assieme all'immagine scelta per la predizione; si veda figura 7.10. Inoltre in automatico viene creata una cartella con lo stesso nome della classe finale predetta, dentro alla quale viene salvata l'immagine appena elaborata. Nel caso la cartella fosse già presente, magari perchè è già stata fatta una predizione

avente come esito la stessa classe, semplicemente viene aggiunta la nuova immagine all'interno della medesima cartella. Le immagini salvate in queste cartelle vengono nominate secondo lo standard solitamente utilizzato dalla fotocamere, ovvero aggiungendo al nome del file la data e l'ora in cui tale immagine è stata salvata. In questo modo si evitano eventuali problemi di sovrascrittura di file.

Al termine di una predizione l'utente può ovviamente effettuarne un'altra, sempre secondo una delle due modalità accennate. I passaggi sono gli stessi descritti finora.

Un'ulteriore aggiunta riguarda il fatto che il modello della rete convoluzionale scelta viene caricato una sola volta, non ad ogni singola predizione. Questo permette di risparmiare molto tempo, in quanto l'utente può effettuare più predizioni sequenzialmente senza dover ricaricare ogni volta il pesante file *.caffemodel*, oppure analogamente nel caso l'app non fosse più in foreground, quando essa vi farà ritorno, se non è stata chiusa dal sistema per necessità di risorse, non sarà necessario ricaricare il modello. Nel caso invece l'app venisse chiusa completamente, dal sistema oppure dall'utente, sarà necessario un nuovo caricamento del modello. La verifica viene fatta tramite un semplice booleano che identifica il fatto che il modello sia già stato caricato in precedenza o no.

L'applicazione è stata testata e risulta funzionante sulla versione 6.0.1 di Android presente in un tablet Nexus 7 gentilmente fornito dall'università di Padova. L'applicazione è stata infatti predisposta anche per il nuovo sistema di richiesta dei permessi che viene adottato su Android a partire dalla versione 6.0. I permessi necessari per un corretto funzionamento sono *CAMERA* e *WRITE_EXTERNAL_STORAGE* per ovvi motivi.

7.6 Modifiche per effettuare ulteriori analisi

Nel capitolo 6 sono stati mostrati i risultati di varie analisi effettuate tramite dei test delle reti sia su PC che su Android. In questo paragrafo verranno brevemente riassunti i dettagli relativi alle modifiche che hanno permesso l'ottenimento dei relativi dati, ovvero i vari tempi ricavati.

I dati della tabella 6.2, i risultati sono stati ottenuti ottenuti applicando alcune modifiche al file *classification.cpp* presente all'interno del framework Caffe: in vari punti del codice sono stati aggiunte apposite istruzioni per ricavare i tempi del sistema in determinati istanti; effettuando le giuste differenze fra due misurazioni (tempo fine operazione - tempo inizio operazione) è stato possibile ottenere i tempi delle durate delle varie operazioni, salvandoli su file per ogni predizione eseguita. Un'operazione analoga è stata effettuata

Figura 7.10: Predizione finale dell'app

anche per il ricavo dei dati della tabella 6.3, solo che questa volta le istruzioni sono state inserite nel file *caffe_mobile.cpp*, il quale come già detto contiene le chiamate dirette al codice di Caffe che si volevano analizzare, in particolare quelle relative al caricamento del modello ed alla forward pass della rete per la predizione.

Per quanto riguarda i dati relativi ai tempi dei singoli layer di una rete, mostrati nel paragrafo 6.3, all'interno del framework di Caffe è presente uno script chiamato *net.cpp*; in questo script è presente il metodo *ForwardFromTo()*, il quale intuitivamente riceve in ingresso il numero di layer della rete, o meglio i due valori che identificano il layer minimo ed layer massimo. Nel metodo *ForwardFromTo()*, tramite un ciclo for, vengono fatte tutte le operazioni di forward di ciascun singolo layer, nell'ordine con cui sono stati definiti all'interno della rete. E' dunque possibile in questo caso misurare, per ciascuna operazione di forward, la porzione di tempo spesa in ciascun layer durante una singola predizione, impostando opportune istruzioni per le misurazioni temporali all'interno del codice esattamente come è stato fatto per la rilevazione dei tempi delle fasi principali.

Su PC non ci sono stati problemi, dato che era possibile salvare di volta in volta i dati su un file esterno; su Android invece il discorso cambia perchè non è possibile salvare un file direttamente sul dispositivo dalle librerie *.so*. Fortunatamente grazie al metodo *enableLog()* è possibile vedere l'output di Caffe direttamente sul Logcat di Android: è necessario modificare leggermente il metodo *enableLog()* in *caffe_jni.cpp* e impostare il canale *stdout* invece di quello di default che risulta essere *stderr*. Impostando dunque delle visualizzazioni di dati, cioè dei cout, è stato dunque possibile visionare i dati sul Logcat per poi copiarli ed elaborarli successivamente. Vi è stato tuttavia un ulteriore problema: per reti come GoogLeNet, in cui ci sono più di un centinaio di layer in totale, e di conseguenza altrettante righe di output, nel passaggio dell'output dal codice nativo al Logcat qualche dato andava perso, forse per via dei tanti dati visualizzati insieme ad altre operazioni di Caffe nell'arco di un paio di secondi. Ad ogni modo per risolvere si è deciso di procedere in questo modo: invece di stampare a video il tempo di ciascun layer, esso veniva salvato in un array appositamente creato. Al termine del ciclo for della forward pass, è stato creato un ulteriore ciclo for esclusivamente per visualizzare i tempi ricavati. In questo modo tutti gli altri messaggi relativi all'elaborazione di Caffe non erano più presenti, in quanto era già terminata, ed il problema si è risolto: tutti i dati risultavano essere sempre reperibili. Per inciso, ogni singola modifica apportata al framework importato su Android, necessita ovviamente di una ricompilazione con Android NDK per ottenere delle nuove librerie *.so* da importare all'interno dell'applicazione.

Per ottenere i dati più approfonditi riguardanti i layer di normalizza-

zione, è stato invece necessario inserire delle rilevazioni all'interno del file *lrn_layer.cpp* presente all'interno del framework Caffe, analizzando le porzioni di codice d'interesse, le stesse viste e spiegate nel paragrafo 6.3.1.

7.7 Validazione e performance

In questo ultimo paragrafo vengono presentati i risultati ottenuti, in termini di accuratezza delle predizioni, da parte delle tre reti selezionate sul dataset PIPPI, appositamente costruito. Vengono inoltre fatte ulteriori considerazioni sull'utilizzo delle reti neurali convoluzionali, basandosi sui dati e sui risultati ottenuti nel corso dell'intero progetto.

7.7.1 Performance delle reti con il dataset PIPPI

Il modello pre-allenato di AlexNet, appositamente inserito nel Model Zoo di Caffe per un'operazione di fine-tuning, risulta essere già stato allenato per ben 358,000 iterazioni; per NIN è stato scaricato il relativo modello già allenato con circa lo stesso numero di iterazioni di quello di AlexNet, mentre il modello di GoogLeNet è già stato pre-allenato con addirittura 2,400,000 iterazioni. I risultati ottenuti dalle tre reti vengono presentati nella tabella 7.2. Tutti i dati presenti nella tabella riguardano la top-1 accuracy, dato che sia per lo scopo della tesi che per il numero relativamente basso di classi nel dataset, la top-5 accuracy non risulta essere un dato significativo. L'accuracy, come già accennato nella sua definizione nel paragrafo 4.4.9, riguarda ovviamente il validation set del dataset PIPPI e dunque è relativa all'utilizzo di Caffe su PC: essa serve per capire se il modello ottenuto con una specifica rete risulta essere buono o meno; dato che i dati in questione risultavano essere buoni, si è deciso di proseguire successivamente con il test diretto sul dispositivo mobile, i cui risultati verranno mostrati fra poco.

Esattamente come ipotizzato, le dimensioni dei modelli sono leggermente diminuite, dato che si è passati da 1000 a 7 classi. Nella stessa tabella vengono riportate le effettive dimensioni finali dei modelli.

Come già detto, per ogni rete sono state effettuate 3000 iterazioni. Ci si è fermati a questo numero, perchè oltre ad esso nessuna rete dava prestazioni migliori, anzi spesso l'accuracy calava, come si vede nella tabella.

Si nota che, nonostante il dataset sia significativamente più difficile rispetto a quello precedente, i risultati del fine-tuning di ciascuna rete risultano essere molto buoni. AlexNet trae qualche vantaggio con 1000 iterazioni in più (da 1000 a 2000), aumentando leggermente la sua accuracy; le altre due reti invece peggiorano leggermente, per via dell'overfitting, in quanto avendo

	Top-1 accuracy su validation set			**Dimens. modello (MB)**
	1000 iter.	***2000 iter.***	***3000 iter.***	
AlexNet	0,91	0,95	0,945	227
NIN	0,97	0,95	0,92	26
GoogLeNet	0,98	0,977	0,96	41

Tabella 7.2: Risultati test su dataset PIPPI

architetture più complesse molto probabilmente apprendono meglio più velocemente rispetto ad AlexNet e quindi con 1000 iterazioni in più si rischia di overfittare sui dati.

I risultati complessivamente di per sè sono molto buoni, ma ciò è principalmente dovuto alla classificazione delle 4 classi che non rappresentano triangoli: gli esemplari di queste 4 classi sono classificati praticamente tutti correttamente, quindi le reti distinguono molto bene le classi piuttosto diverse fra loro ed anche classi non particolarmente diverse come ad esempio quelle dei moduli 730 e consenso informato. Se si analizzano invece solamente le classi dei 3 tipi di triangoli li c'è sicuramente molta più incertezza: quasi tutti gli errori di classificazione derivano infatti dalla non corretta classificazione di una determinata tipologia di triangolo, ed in molti casi, anche se la classificazione risulta corretta, la differenza fra la probabilità più elevata e quella successiva è minima, anche dell'ordine di 0,1 a volte. Ci sono ovviamente anche casi inversi in cui invece la classificazione è di poco errata. Le predizioni errate sono praticamente sempre tipologie diverse di triangoli, ovvero gli errori non riguardano la considerazione di altre classi come ad esempio quella del modulo del consenso informato e dei disegni.

Si intuisce dunque che, nonostante il numero elevato di immagini, nonostante l'utilizzo di modelli pre-allenati molto robusti ed un successivo fine-tuning, le reti faticano a distinguere un tipo di triangolo da un altro. Anche considerando complessivamente solo i triangoli, non se la cavano comunque male, ad esempio GoogLeNet sbaglia la classificazione di poco più di 10 immagini, su un validation set di 316 immagini, di cui circa 150 di esse sono triangoli, tuttavia in molte classificazioni le probabilità più alte sono molto vicine, come già detto. Ciò vuol dire che se ad esempio si prendessero tre campioni, ognuno costituito da una foto dalla stessa tipologia di triangolo, dove per ciascun campione le foto acquisite sono simili ma comunque tutte diverse fra loro, c'è il rischio di ottenere risultati diversi per ognuno dei 3

	Classificazioni corrette			
	Cons.Informato	**Disegno**	**Ecomappa**	**Modulo 730**
AlexNet	$\frac{18}{20}$	$\frac{18}{20}$	$\frac{19}{20}$	$\frac{18}{20}$
NIN	$\frac{18}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{18}{20}$
GoogLeNet	$\frac{19}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{19}{20}$

Tabella 7.3: Classificazione immagini tramite dispositivo Android (parte 1)

campioni. In sostanza si potrebbero avere classificazioni diverse anche per immagini della stessa classe: è sufficiente infatti variare magari leggermente l'inquadratura, oppure la distanza della camera rispetto all'oggetto per avere un possibile risultato diverso, sempre di norma all'interno dell'insieme dei triangoli. Quindi il comportamento delle reti nel distinguere i triangoli è abbastanza imprevedibile, nonostante tendenzialmente la classificazione risulti corretta.

Bisogna tenere presente inoltre che le immagini date in pasto alle reti vengono ridimensionate a 256 x 256 e successivamente Caffe prende una crop centrale 227 x 227. Se si prova a confrontare le immagini di due tipi di triangoli diversi, in particolare un triangolo bambino ed un triangolo operatore, di queste dimensioni si vedrà che anche ad occhio umano la classificazione risulta essere veramente difficile, motivo che giustifica a mio avviso le difficoltà delle reti per questi casi. Solo una conoscenza semantica posseduta dalla rete potrebbe aiutare, ma ciò non rientra nello scopo della tesi.

Il test diretto con il dispositivo Android, effettuato tramite l'acquisizione di un certo numero di immagini per ogni categoria del dataset con la fotocamera, rispecchia grossomodo quanto detto finora. Globalmente le performance si abbassano, proprio a causa delle predizioni molto variabili sulle classi dei triangoli. Le tabelle 7.3 e 7.4 mostrano i risultati ottenuti, indicando per ciascuna classe quante immagini sono state classificate correttamente sul totale di immagini acquisite nella medesima categoria. Per ciascuna rete sono state utilizzate le stesse immagini acquisite, dato che una volta salvate è possibile riutilizzarle importandole dalla galleria, impostando l'utilizzo della rete voluta tramite il codice dell'app.

Mentre nella tabella 7.3 c'è una situazione abbastanza uniforme, nella

	Classificazioni corrette		
	T. Bambino	**T. Operatore**	**T. Vuoto**
AlexNet	$\frac{10}{20}$	$\frac{16}{20}$	$\frac{13}{20}$
NIN	$\frac{13}{20}$	$\frac{15}{20}$	$\frac{14}{20}$
GoogLeNet	$\frac{16}{20}$	$\frac{12}{20}$	$\frac{16}{20}$

Tabella 7.4: Classificazione immagini tramite dispositivo Android (parte 2)

tabella 7.4 come preannunciato vi è invece una situazione piuttosto instabile. Infatti, se ad esempio si provasse ad acquisire delle altre immagini sui triangoli e a verificare di nuovo quante vengono predette correttamente, molto probabilmente si otterrebbero dei valori diversi per ciascuna classe, proprio perchè la rete fa fatica ad identificare un tipo di triangolo da un altro e quindi con immagini leggermente diverse potrebbe prevalere una classe invece di un'altra. Di norma comunque, come già accennato, viene sempre predetto un triangolo; raramente viene predetta una delle altre 4 classi rimanenti.

7.7.2 Validazione del progetto

Il porting di Caffe e la piccola applicazione per effettuare delle classificazioni di immagini acquisite, hanno permesso la realizzazione del progetto, che si ricorda essere solo una parte dello schema iniziale di figura 1.1.

Complessivamente i risultati sembrano essere buoni: le reti hanno un'elevata accuratezza nella predizione di classi, cosa che è stata vista anche testando direttamente l'applicazione. Ci sono alcuni problemi con l'identificazione di una particolare categoria di triangoli, come accennato prima, che però rappresentano un caso molto particolare che è stato considerato per vedere effettivamente il comportamento delle reti in casi reali piuttosto complicati.

AlexNet purtroppo, come era stato previsto, ha un tempo di inizializzazione molto alto dovuto alla dimensione del suo modello, ma ha però un ottimo tempo di predizione: in circa un secondo infatti è in grado di classificare una specifica immagine. Riguardo a quest'ultimo tempo, NIN risulta essere la migliore in termini di velocità computazionale, anche se come si evince dalla tabella 7.2, non risulta essere però la più accurata nelle predi-

zioni, ruolo ricoperto da GoogLeNet. Queste ultime due reti hanno anche un tempo di inizializzazione molto più basso rispetto ad AlexNet, dell'ordine di circa un paio di secondi per entrambe (tabella 6.3). Un tempo del genere risulta decisamente più accettabile rispetto a quanto visto per AlexNet. Per determinare quale rete sia la più adatta bisognerebbe poi fare un compromesso, ovvero decidere che è sufficiente utilizzare NIN, in quanto ha un'accuracy comunque buona, nel caso si volesse spendere meno tempo nella predizione (circa un secondo), oppure utilizzare GoogLeNet, più accurata ma anche più lenta, dato che la sua predizione dura circa 2 secondi. La scelta dipenderà anche dalla rimanente parte del progetto di figura 1.1 ancora da effettuare.

Ad ogni modo, i dati raccolti nel capitolo 6 e nella parte iniziale di questo capitolo, dimostrano che è potenzialmente possibile utilizzare le reti neurali convoluzionali localmente su un dispositivo mobile, in quanto non solo sono un ottimo strumento di classificazione, ma in più il loro consumo di risorse non sembra essere eccessivo. Tutto questo però, consci del fatto che l'utilizzo di queste reti presenta ancora alcuni limiti, legati principalmente proprio alle risorse disponibili sul dispositivo mobile. Bisogna poi presente che il dispositivo utilizzato negli esperimenti della tesi risale al 2013. I nuovi dispositivi mobili, dotati di un hardware sempre più potente, potrebbero benissimo utilizzare queste reti in tempi anche inferiori. Il divario fra le risorse disponibili presenti in questi device e quello degli attuali PC, sta diminuendo sempre di più: i dati stessi ricavati nel corso di questo progetto dimostrano questo.

Sempre considerando le risorse del dispositivo, l'utilizzo di queste reti in locale sembra essere accettabile anche a livello di consumo energetico, infatti le stime sostengono che il consumo non è eccessivo e sembra addirittura minore rispetto a quello di un upload di immagini con una qualsiasi rete dati, cosa abbastanza comune al giorno d'oggi.

E' vero però che nel caso si volesse un tempo di elaborazione molto più rapido, il che probabilmente significherebbe utilizzare un hardware più potente, i livelli di consumo energetico potrebbero tuttavia diventare meno sostenibili. Il fatto però che nuove tipologie di hardware appositamente progettate per l'utilizzo locale di queste reti sui dispositivi mobili, sono state presentate in questi ultimi mesi, sembra dare speranza a quanto detto finora. Ad esempio un nuovo chip, chiamato Eyeriss ([35]), è stato presentato a Febbraio da parte di alcuni ricercatori del MIT: questo chip permetterebbe l'utilizzo di reti neurali convoluzionali con tempi di elaborazione minori e consumo di energia limitato. Molte applicazioni che attualmente utilizzano questo tipo di reti, necessitano che l'utente sia in qualche modo collegato ad Internet, per effettuare l'upload di dati che vengono elaborati esternamente; questo chip potrebbe fare in modo invece di utilizzare sempre localmente queste reti e quindi di non necessitare ogni volta di utilizzare una connessione dati, il tutto

senza consumare velocemente la batteria del dispositivo. Potenzialmente si potrebbero inoltre utilizzare reti ancora più "deep" come ad esempio le reti VGG, o altre reti che verranno ideate nel corso del tempo, cosa che invece al momento non conviene fare per via dell'enorme quantità di parametri che esse possiedono, ottenendo forse in questo modo risultati ancora migliori nell'accuratezza delle predizioni.

Conclusioni

In questa tesi è stata presentata una possibile implementazione locale delle reti neurali convoluzionali in un dispositivo mobile, legate al progetto di un'applicazione Android per l'apprendimento automatico di informazioni. La scelta per cui si è deciso di fare questo riguarda il fatto che questo tipo di dispositivi rappresenta molto probabilmente il futuro, in quanto quasi chiunque ormai ne possiede almeno uno, chi per necessità di lavoro e chi per semplice svago o altro tipo di attività. Se i dispositivi mobili sono il futuro della tecnologia, le reti neurali convoluzionali sembrano essere il futuro per molti settori della computer vision e non solo, specialmente per problemi di object recognition, detection e segmentation. Aziende del calibro di Google si sono già accorte da tempo di questo ed infatti le stanno già utilizzando in molte delle loro applicazioni. Nel corso degli anni, viste le loro prestazioni, molto probabilmente si diffonderanno ancora di più e sicuramente verranno ulteriormente migliorate, ed i nuovi harware, come ad esempio il chip menzionato in precedenza, permetteranno un loro utilizzo locale scavalcando alcuni limiti che attualmente posseggono, legati alla disponibilità di risorse di un attuale dispositivo mobile di fascia comune. Nella tesi lo scopo del loro utilizzo riguarda una specifica applicazione, applicazione che dovrebbe potenzialmente servire ad automatizzare alcuni compiti che attualmente richiedono ancora un'intervento umano esterno. Per completare l'idea iniziale rimarrebbe da svolgere la seconda parte dello schema 1.1, che si lascia per eventuali sviluppi. Di per sè comunque, lo studio effettuato nel corso di questa tesi può comunque rivelarsi utile a chiunque voglia cimentarsi nel ramo del Deep Learning, con alcuni dati aggiuntivi riguardanti un possibile utilizzo di Convolutional Neural Networks sulla piattaforma Android.

Appendice A

Architettura complessiva di GoogLeNet

In figura A.1 viene mostrata una tabella riassuntiva dell'intera rete; vengono omesse le due reti ausiliarie di GoogLeNet, contenenti i due ulteriori output layer.

Le figure A.2, A.3, A.4, A.5, A.6 mostrano invece l'intera architettura della rete. Si noti che la parte iniziale della rete non possiede più layer in parallelo, bensì rappresenta un classico approccio di tutte le CNN viste finora. A partire dal primo inception layer aumenta anche la larghezza della rete.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figura A.1: Tabella riassuntiva di GoogLeNet

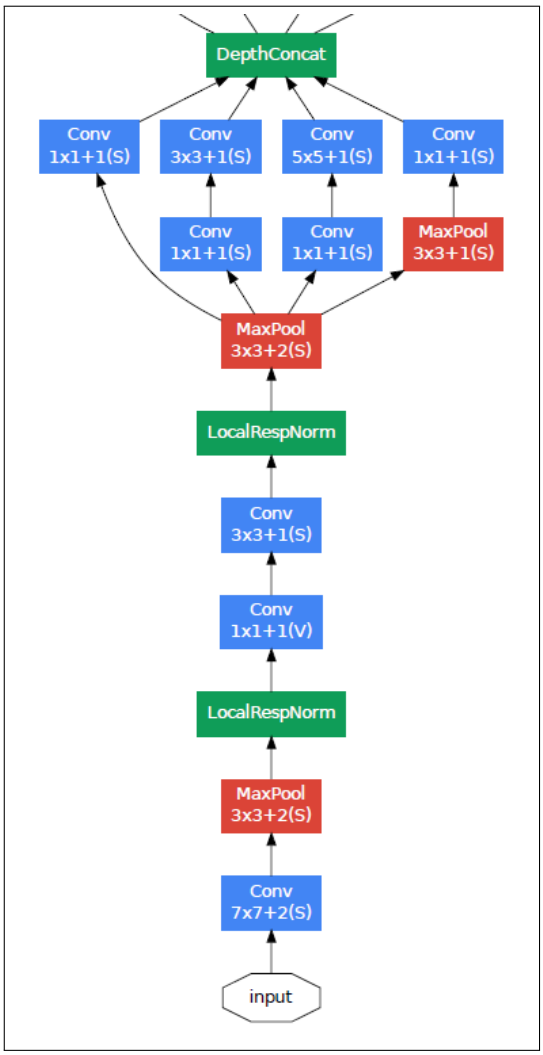


Figura A.2: Architettura GoogLeNet (parte 1/5)

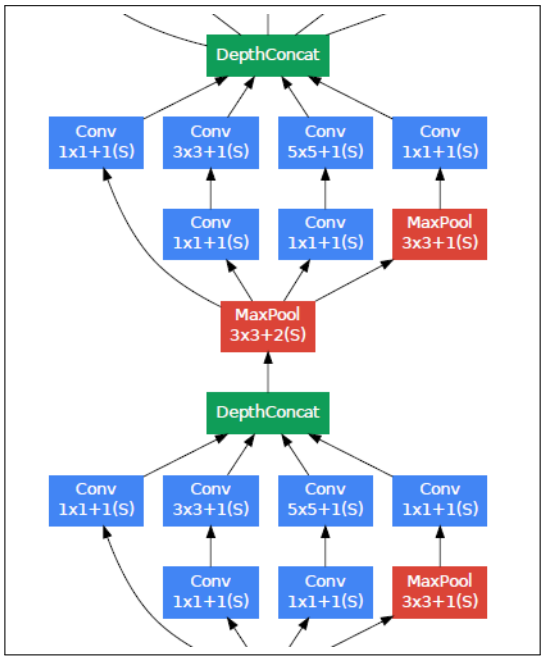


Figura A.3: Architettura GoogLeNet (parte 2/5)

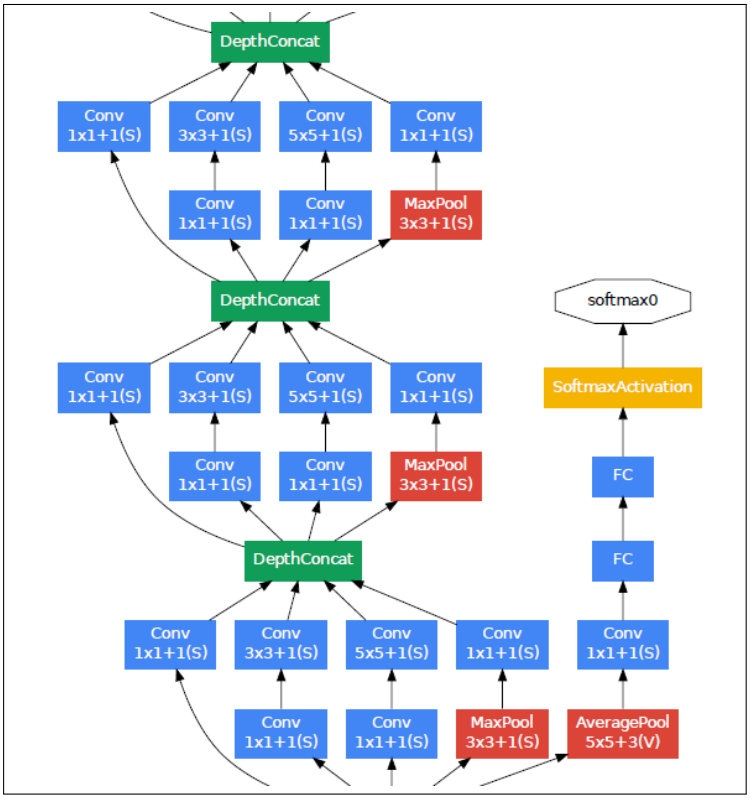


Figura A.4: Architettura GoogLeNet (parte 3/5)

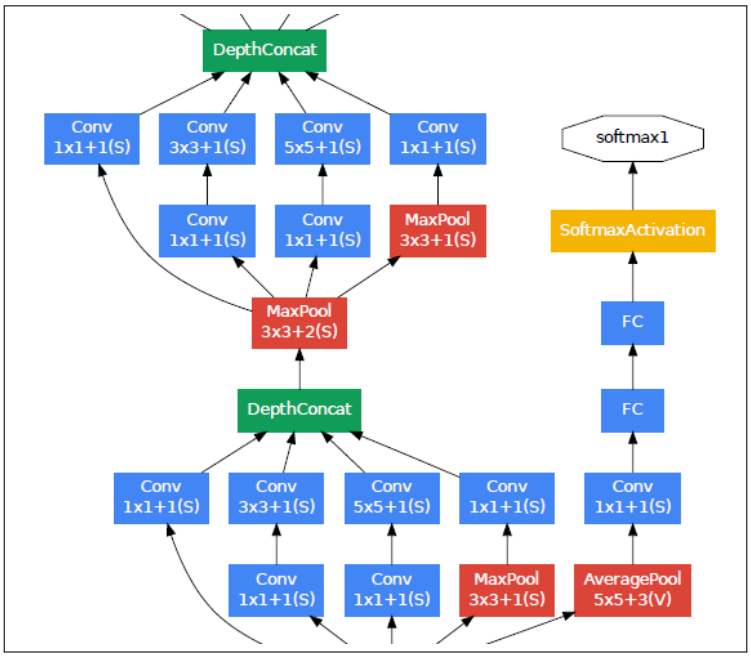


Figura A.5: Architettura GoogLeNet (parte 4/5)

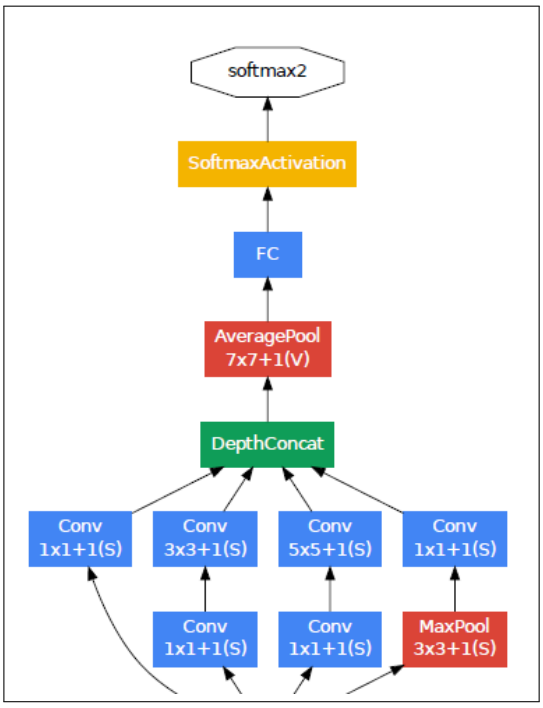


Figura A.6: Architettura GoogLeNet (parte 5/5)

Bibliografia

[1] Y. Jia. Caffe: An open source convolutional architecture for fast feature embedding. http://caffe.berkeleyvision.org/, 2013.

[2] Evernote Scannable App. https://evernote.com/intl/it/products/scannable/

[3] Microsoft Office Lens. https://www.microsoft.com/it-it/store/apps/office-lens/9wzdncrfj3t8

[4] App Mobile Banking Unicredit. https://www.unicredit.it/it/privati/serviziinnovativi/tuttaunaltrastoria_app.html

[5] Programma di Intervento Per la Prevenzione dell'Istituzionalizzazione (PIPPI). https://elearning.unipd.it/progettopippi/

[6] Laboratorio di Ricerca e Intervento in Educazione Familiare (LabRIEF). http://labrief.fisppa.unipd.it/pippi/

[7] Geoffrey Hinton, Li Deng, Dong Yu, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath George Dahl, and Brian Kingsbury. "*Deep Neural Networks for Acoustic Modeling in Speech Recognition*". IEEE Signal Processing Magazine, pages 82-97, 2012.

[8] Jun Yang, Yu-Gang Jiang, Alexander G. Hauptmann, Chong-Wah Ngo. "*Evaluating Bag-of-Visual-Words Representations in Scene Classification*". Proceedings of the international Workshop on Workshop on Multimedia information Retrieval , pages 197-206.

[9] Deep Learning and Deep Neural Networks. https://en.wikipedia.org/wiki/Deep_learning.

[10] Philipp Fischer, Alexey Dosovitskiy, Thomas Brox. "*Descriptor Matching with Convolutional Neural Networks: a Comparison to SIFT*". CoRR, abs/1405.5769, 2014.

[11] "*This Business of Brewing: Caffe in Practice*", from the course of Convolutional Neural Networks for Visual Recognition of the Standford University. `http://vision.stanford.edu/teaching/cs231n/slides/evan.pdf`.

[12] Yann LeCun, Yoshua Bengio, Geoffrey Hinton. "*Deep learning*". Nature, Vol 521, 2015. `https://dx.doi.org/10.1038%2Fnature14539`

[13] Artificial Neuron. `https://en.wikipedia.org/wiki/Artificial_neuron`

[14] UFLDL Tutorial, Standford University. `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`.

[15] Convolutional Neural Network for Visual Recognition, Standford University. `http://cs231n.github.io/convolutional-networks/`.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. "*Imagenet classification with deep convolutional neural networks*". In Advances in Neural Information Processing Systems 25, pages 1106−1114, 2012.

[17] Min Lin, Qiang Chen, and Shuicheng Yan. "*Network in network*". CoRR, abs/1312.4400, 2013.

[18] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich,A. "*Going deeper with convolutions*". CoRR, abs/1409.4842, 2014.

[19] Karen Simonyan, Andrew Zisserman. "*Very deep convolutional networks for large scale image recognition*". Published as a conference paper at ICLR 2015.

[20] Matthew D. Zeiler and Rob Fergus. "*Visualizing and understanding convolutional networks*". David J. Fleet, Tom as Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I, volume 8689 of Lecture Notes in Computer Science, pages 818-833. Springer, 2014.

[21] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, Andrew Zisserman. "*Return of the Devil in the Details: Delving Deep into Convolutional Nets*", 2014.

[22] Song Han, Huizi Mao, William J. Dally. "*Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding*". Under review as a conference paper at ICLR 2016.

[23] Sergey Karayev, Matthew Trentacoste, Helen Han, Aseem Agarwala, Trevor Darrell, Aaron Hertzmann, Holger Winnemoeller. "*Recognizing Image Style*". University of California, Berkeley, Adobe, 2014.

[24] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In Proc. CVPR, 2014.

[25] Emily Blem, Jaikrishnan Menon, Karthikeyan Sankaralingam. "*Power struggles: revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures*" University of Wisconsin - Madison In Proc. IEEE, 2013.

[26] Hassan Ghasemzadeh, Navis Amini, Ramyar Saeedi, Majid Sarrafzadeh. "*Power-Aware computing in wearable sensor networks: an optimal feature selection*". IEEE Transactions on mobile computing, VOL.14, NO.4, April 2015.

[27] Caffe-android-lib by Shiro Bai. `https://github.com/sh1r0`.

[28] Android-Object-Detection by Darrenl Tzutalin. `https://github.com/tzutalin/Android-Object-Detection`.

[29] Boost libraries. `http://www.boost.org/`.

[30] Eigen libraries. `http://eigen.tuxfamily.org/index.php?title=Main_Page`.

[31] OpenCV libraries. `http://opencv.org/`.

[32] Protobuf library. `https://developers.google.com/protocol-buffers/`.

[33] Khairul Muzzammil Saipullah, Ammar Anuar, Nurul Atiqah Ismail and Yewguan Soo "*Measuring power consumption for image processing in android smartphone*". Department of Computer Engineering, Faculty of Electronic and Computer Engineering, University Teknikal Malaysia Melaka, Melaka, Malaysia.

[34] Fatemeh Jalali, Chrispin Gray, Arun Vishwanath, Robert Ayre, Tansu Alpcan, Kerry Hinton, Rodney S. Tucker "*Energy Consumption of Photo Sharing in Online Social Networks*". Centre for Energy-Efficient Telecommunications (CEET), University of Melbourn.

[35] Yu-Hsin Chen, Tushar Krishna, Joel Emer, Vivienne Sze. "*Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks*". International Solid-State Circuits Conference (ISSCC), pages 262-263, San Francisco, CA, USA, 2016.

Ringraziamenti

Vorrei ringraziare innanzitutto il Prof.Carlo Fantozzi per avermi sempre seguito, ascoltato e consigliato nell'arco dell'intero svolgimento di questa tesi, dedicandomi molto del suo tempo e fornendomi anche molto materiale utile in vari punti della tesi.

Ringrazio poi la mia famiglia e ai miei parenti (un saluto speciale al nonno) per avermi sostenuto nel corso di tutti questi anni di studio e per avermi sempre fornito un consiglio nei momenti di bisogno.

Ringrazio tutti i miei compagni di studio, incontrati nell'arco del mio percorso universitario, per tutti i bei momenti passati assieme.

Un ringraziamento è d'obbligo anche ad i miei amici, per tutte le sere che hanno sopportato i miei racconti nel corso di questo lavoro.