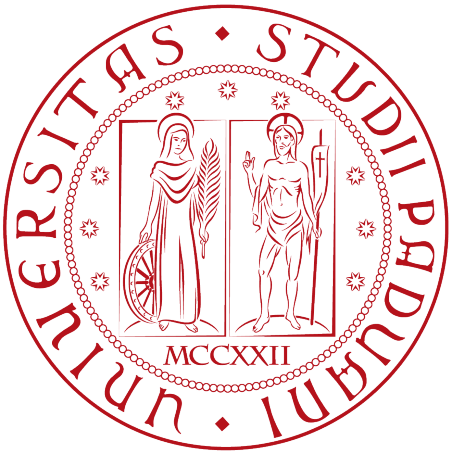
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Applicazione di algoritmi di machine learning per il riconoscimento di sushi integrato in una applicazione Android

Tesi di laurea

Relatore

Prof.essa Ombretta Gaggi

Laureando

Samuele Sartor

N. di Matricola: 1187588

Anno Accademico 2021-2022

Samuele Sartor: *Applicazione di algoritmi di machine learning per il riconoscimento di sushi integrato in una applicazione Android,* Tesi di laurea, © Settembre 2022.

Non si può mai attraversare l'oceano se non si ha il coraggio di perdere di vista la riva.

— Cristoforo Colombo

Ringrazio tutti i docenti dell'università, in particolare la Professoressa Ombretta Gaggi per l'aiuto e il sostegno fornitomi durante la stesura della relazione di stage.
Ringrazio anche l'azienda SyncLab s.r.l. per avermi dato l'opportunità di svolgere il mio stage universitario allargando il mio bagaglio di esperienze e competenze.
Infine desidero ringraziare con affetto la mia famiglia e amici più e meno storici per avermi sostenuto in questo viaggio.

Padova, Settembre 2022 Samuele Sartor

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di 336 ore totali nel periodo compreso tra il 19 maggio e il 15 luglio 2022, dal laureando Samuele Sartor presso l'azienda SyncLab s.r.l .

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Azienda e progetto

1.1 L'azienda

Sync Lab nasce come software house nel 2002 a Napoli con lo scopo di fornire soluzioni per il settore dell'*Information and Communications Technology* (ICT). Si è evoluta rapidamente entrando nei campi di sanità, industria, energia, telecomunicazione, finanza e trasporti & logistica.

Figura 1.1: Logo azienda SyncLab.

Attualmente Sync Lab ha più di 150 clienti diretti e finali, con un organico aziendale di non meno di 200 dipendenti distribuiti tra le 5 sedi dislocate in tutta Italia: Napoli, Roma, Milano, Verona e Padova.

L'offerta di consulenza specialistica, sintesi dell'esperienza maturata in circa 20 anni nel settore IT e la forte competenza in diversi domini tecnologici aiutano i clienti ad adattarsi ai nuovi trend e alle sfide che ne derivano, quali: GDPR, Big Data, Cloud Computing, IoT, Mobile e Cyber Security.

1.2 Problema affrontato

La realtà lavorativa presente in questa azienda permette la formazione in quasi tutti i campi dell'informatica applicata, dallo sviluppo web app, alla cyber sicurezza fino all'utilizzo di modelli di intelligenza artificiale. Per questo motivo Sync Lab si pone come artefice di svariate tipologie di soluzioni informatiche e combinazioni di tecnologie distinte, come il progetto da me realizzato che vuole rispondere al seguente problema reale.
Nelle ore di punta molti ristoranti incontrano rallentamenti nel processo di consegna dei piatti ai clienti semplicemente perché si presentano troppi ordini allo stesso tempo e l'organizzazione in cucina diventa seriamente complessa da gestire, specialmente nei locali con formula "*all you can eat*" dove ogni cliente ordina svariati piatti contemporaneamente. Un ulteriore problema è rappresentato dal fatto che, quando si ordina in gruppo, è molto comune dimenticare alcuni dei propri ordini o pensare di aver ordinato piatti altrui.
La soluzione potrebbe essere un'applicazione che permetta di ordinare i piatti in modo preciso e individuale tramite il loro numero specifico del menu.Tale applicazione consentirebbe poi di aggiungere i propri ordini a quelli di amici e colleghi, assegnandoli al proprio tavolo e con il proprio nominativo individuale. In tal modo i camerieri, avendo direttamente accesso alla lista degli ordini dei clienti, riuscirebbero a organizzarsi meglio per le consegne dei piatti. L'applicazione consentirebbe anche ai cuochi l'accesso diretto agli ordini dei clienti lasciando ai camerieri più tempo per servire ai tavoli.
L'obiettivo di questo progetto sarebbe l'implementazione di un servizio software per l'organizzazione degli ordini e la loro identificazione nel minor tempo possibile. L'azienda mira alla creazione di un'applicazione di *object detection*, per il riconoscimento dei piatti, integrata in un'**applicazione per dispositivi portatili** connessa ad una **piattaforma web**, per l'organizzazione degli ordini.

Tale progetto è stato scomposto in tre esperienze di stage per laureandi: in primo luogo, due studenti dell'Università di Padova hanno condotto lo sviluppo della **piattaforma web** tramite il linguaggio React Native, altamente compatibile con ogni piattaforma e sistema operativo, da computer a dispositivi mobili; in seguito, io ho allenato l'**intelligenza artificiale** per il riconoscimento dei piatti di sushi; infine, uno studente dell'Università di Milano avrà il compito di produrre l'**applicativo mobile** tramite l'unione della piattaforma web con l'intelligenza artificiale per lo sviluppo del prodotto finale.

La soluzione del progetto da me realizzata è stata suddivisa in quattro fasi:

- Ricerca delle tecnologie corrispondenti al corrente stato dell'arte del *deep learning*$_{\mathbf{G}}$ nel campo dell' *object detection* con lo scopo di trovare i migliori **modelli di reti neurali** per l'identificazione di cibo e nello specifico sushi, ma che fornisca anche librerie dedicate o compatibili per l'utilizzo nel progetto in cui dovrà essere utilizzata;

- Composizione di un **dataset** per l'allenamento e test della **rete neurale**, ricerca immagini corrispondenti a vari tipi di sushi e etichettatura di questi ultimi all'interno delle immagini definendo coordinate, dimensioni e classe delle *label*$_{\mathbf{G}}$;

- Allenamento e sperimentazione di varianti della rete neurale per il raggiungimento di una precisione elevata e stabile riguardo i risultati ottenuti in fase di test del

dataset per la rilevazione corretta di classi e *bounding boxes*$_{\mathbf{G}}$ all'interno delle immagini;

- Integrazione della rete neurale in un **applicativo Android**$_{\mathbf{G}}$ per monitorare le prestazioni dei dispositivi mobili senza l'ausilio della connessione ad una **piattaforma web** per maggiore versatilità anche per l'utilizzo *offline* o con poca disponibilità di banda.

1.3 Obiettivi

Nella seguente sezione sono riportati gli obiettivi decisi assieme al tutor aziendale riguardo il corrente progetto.
Per gli obiettivi obbligatori, desiderabili e facoltativi è stato usato uno specifico codice per identificarli:

[Priorità][Numero incrementale]

La priorità può assumere tre valori: O (obbligatorio), D (desiderabile) e F (facoltativo). Di seguito al codice è riportato il titolo dell'obiettivo.

Requisito	Titolo	Descrizione
O01	Autoformazione	Acquisizione di competenze nell'ambito di *machine learning*$_{\mathbf{G}}$, *deep learning*$_{\mathbf{G}}$ e *computer vision*$_{\mathbf{G}}$.
O02	Rispetto delle tempistiche	Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma.
O03	Implementazione *computer vision*$_{\mathbf{G}}$	Portare a termine l'implementazione di una soluzione di intelligenza artificiale con il supporto di tecniche geometriche e *computer vision*$_{\mathbf{G}}$.
O04	Implementazione *deep learning*$_{\mathbf{G}}$	Portare a termine l'implementazione di una soluzione di intelligenza artificiale eventualmente con il supporto di tecniche di *deep learning*$_{\mathbf{G}}$ e *advanced computer vision*$_{\mathbf{G}}$. Superando l'80% rispetto a quanto definito in fase di progettazione.

Tabella 1.1: Tabella del tracciamento dei requisti obbligatori.

Requisito	Titolo	Descrizione
D01	Completamento totale del progetto	Portare a termine l'implementazione di una soluzione di intelligenza artificiale eventualmente con il supporto di tecniche di *deep learning*$_{\mathbf{G}}$ e *advanced computer vision*$_{\mathbf{G}}$. Superando il 100% rispetto a quanto definito in fase di progettazione.

Tabella 1.2: Tabella del tracciamento dei requisti desiderabili.

Requisito	Titolo	Descrizione
F01	Integrazione su piattaforma web o app	Implementare una piattaforma (web/app) che utilizzi il modello di intelligenza artificiale allenato o le tecniche di *computer vision*$_{\mathbf{G}}$ utilizzate.

Tabella 1.3: Tabella del tracciamento dei requisti facoltativi.

1.4 Struttura della tesi

Il presente documento vuole illustrare il progetto di stage precedentemente descritto secondo la seguente struttura:

- **Capitolo 1, tecnologie utilizzate**: essendo un progetto principalmente orientato verso la ricerca e autoformazione sulle ultime tecnologie sviluppate nel campo del *machine learning*$_{\mathbf{G}}$ e *computer vision*$_{\mathbf{G}}$, il compito più importante e dispendioso per ottenere una soluzione ideale è stato la ricerca e confronto tra le varie tecnologie alternative presenti;
- **Capitolo 2, applicazione tecnologie**: questo capitolo intende esporre il funzionamento pratico delle tecnologie adottate e il software generato per il completamento del prodotto;
- **Capitolo 3, risultati ottenuti**: vengono evidenziati i risultati ottenuti secondo le metriche di valutazione per le prestazioni dei modelli di intelligenza artificiale per l'*object detection* e gli esperimenti nella fase di allenamento della rete neurale utilizzata;
- **Capitolo 4, conclusioni**: infine sono stati revisionati i requisiti del progetto per la definizione della progressione nel loro completamento, assieme alla dichiarazione delle competenze e abilità acquisite nel corso del progetto.

1.5 Convenzioni tipografiche

Nella redazione del presente documento sono state usate le seguenti convenzioni tipografiche:

- Le occorrenze dei termini presenti nel glossario saranno seguiti da una G formattata come pedice, esempio deep learning$_{\mathbf{G}}$;
- I termini in lingua diversa dall'italiano sono posti in *corsivo*;
- Termini o tecnologie di particolare rilevanza sono posti in **grassetto**.

Capitolo 2

Tecnologie utilizzate

In questo capitolo verranno analizzate le tecnologie utilizzate e visionato il confronto con le alternative.

2.1 Rete neurale artificiale

I progressi tecnologici e le risorse dei calcolatori si sono evoluti esponenzialmente nel tempo, ma persistono problemi non banali per le macchine che non possono essere ovviati senza ragionamenti derivanti da una mente umana come il riconoscimento di oggetti o la decisione di una risposta. Questi problemi risultano infinitamente complicati da eseguire per un calcolatore, per questo è nata la ricerca delle reti neurali artificiali.

Verranno illustrati i componenti e funzionamenti su cui si basano le reti neurali prima di discutere le ricerche effettuate per trovare il modello di intelligenza artificiale ottimale per questo progetto.

2.1.1 Introduzione alle reti neurali

2.1.1.1 Il percettrone

Il cervello umano è un complesso sistema neurobiologico in grado di elaborare informazioni, composto fondamentale da neuroni.
I processi elettrochimici che hanno sede all'interno di queste cellule, processano e modulano le informazioni provenienti da altri neuroni per poi inviare nuovi segnali ai neuroni connessi.
Tuttavia un calcolatore non dispone di queste abilità, per questo motivo nel corso degli anni si è cercato di identificare tramite studi psicologici e neurologici un modello matematico in grado di simulare il neurone che potesse essere eseguito in un calcolatore.

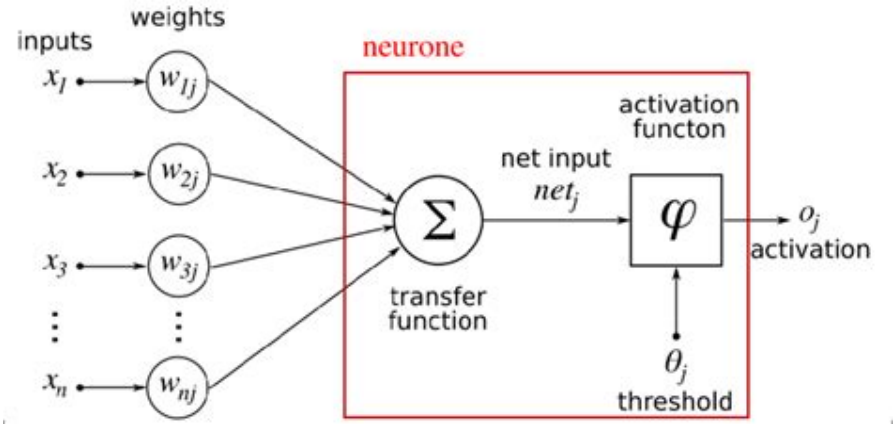


Figura 2.1: Schema computazionale di un Percettrone, come descritto da F. Rosenblatt.

Il Percettrone, schematizzato nella figura 2.1, è un tipo di classificatore binario che associa un insieme di *input* ad un *output* scalare y (di tipo reale) che viene calcolato come:

$$y = \theta(\sum_{i=1}^{n} w_i x_i - b)$$

Dove $\sum_{i=1}^{n} w_i x_i - b$ è la somma pesata delle corrispondenti x_i del vettore degli *input*, dove il peso è definito da w_i e θ rappresenta una funzione chiamata **funzione di attivazione**.
Il valore b, detto $bias_{\text{G}}$, permette di far slittare la funzione di attivazione a destra o a sinistra per interpretare al meglio i dati in *input*. Questo valore può risultare critico per l'apprendimento poiché valori incompatibili con le funzioni di attivazione dei neuroni potrebbero portare a risultati fallaci.

2.1.1.2 L'architettura di una rete neurale artificiale

L'architettura delle reti neurali è derivata dal modello matematico che definisce il percettrone. Infatti, una architettura basilare è rappresentata da una sequenza di strati interconnessi composti da percettroni. Ognuno di essi assumerà il valore fornito della sua funzione di attivazione dato come *input* il risultato del calcolo della somma pesata degli *input* composti dai valori di ogni percettrone dello strato precedente pesati, per la connessione sinaptica che li unisce con il percettrone corrente. Di strato in strato viene calcolato il valore di ogni percettrone finchè non viene raggiunto lo strato di *output* che fornirà, tramite i valori raggiunti dai suoi percettroni, informazioni interessanti rispetto al caso per cui viene usata la rete neurale.
L'esempio di una semplice rete neurale viene illustrato nella figura 2.2.

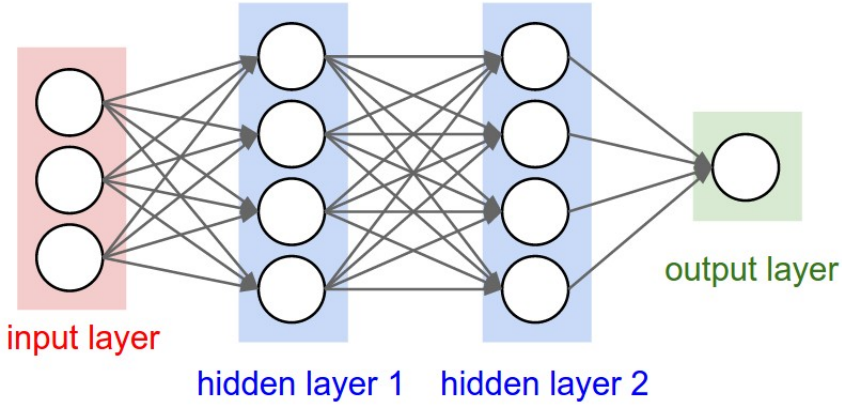


Figura 2.2: Schema di una semplice rete neurale con due strati nascosti.

Esistono due principali tipologie di reti neurali:

- **Reti Feed-Forward**: la classica rete neurale, rappresentata da vari strati, o *layer*, completamente interconnessi dal precedente al successivo;
- **Reti Ricorrenti (RNN)**: sono reti che utilizzano connessioni cicliche, dagli strati successivi a quelli precedenti oppure nello stesso strato, per permettere uno stato stabile tra gli *input*, utilizzate principalmente per elaborare dati sequenziali, come in una linea temporale. Un esempio di rete neurale ricorrente viene illustrato in figura 2.3.

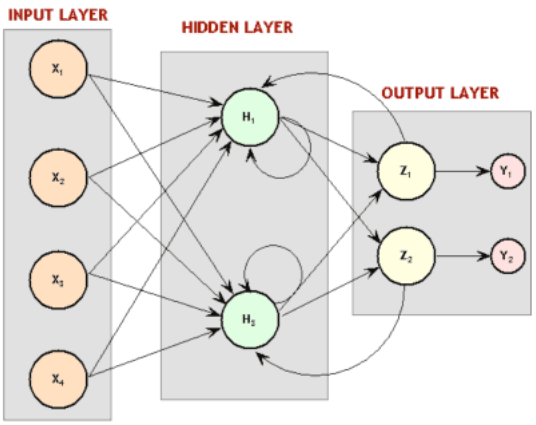


Figura 2.3: Schema di una rete neurale con uno strato nascosto ricorrente a se stesso e allo strato di *output*.

2.1.1.3 Apprendimento della rete neurale

2.1.1.3.1 Apprendimento

Per permettere di comprendere l'errore prodotto dalle predizioni di una rete neurale in fase di allenamento viene utilizzata una formula che calcola la differenza tra il valore predetto e quello reale, la più utilizzata è:

$$MSE = \frac{1}{n}(\sum_{i=1}^{n} y_i - \bar{y}_i)^2$$

Dove n rappresenta il numero di campioni da predire, y_i è il valore reale i-esimo e $\bar{y}_i$ è il valore predetto i-esimo, questa formula viene chiamata *mean squared error* (MSE) o errore quadratico medio.
Utilizzando questa formula di errore con il modello matematico del percettrone, definito nella sezione 2.1.1.1, al posto di $\bar{y}_i$ si ottiene:

$$Loss(w) = \frac{1}{n}(\sum_{i=1}^{n} y_i - \theta(\sum_{i=1}^{n} w_i x_i - b))^2$$

Questa formula definisce l'errore di perdita per cui l'obiettivo è di individuare il w ottimale per cui *Loss* sia minima. Questo calcolo viene fatto tramite il calcolo del *gradient descent*, o discesa di gradiente.

2.1.1.3.2 Gradient descent

Definita una formula f **multi-variabile**, il gradiente Δ_f viene rappresentato da un vettore contenente le derivate parziali di f per ognuna delle sue variabili.

Trovato un punto x, il gradiente indica l'inclinazione del **grafo multi-dimensionale** rappresentante la funzione f, permettendo di riconoscere per ogni variabile il grado di crescita, fornendo la scelta di quale direzione seguire sulla funzione per raggiungere un minimo.
Per comprendere meglio questo concetto prendiamo in considerazione il caso illustrato in figura 2.4 dove f è composta da due variabili, θ_0 e θ_1 che rappresentano le dimensioni del piano, x e y, e la funzione di perdita, $J(\theta_0, \theta_1)$ rappresenta l'asse z. L'obiettivo è il raggiungimento del valore minimo della funzione di perdita trovando di conseguenza i valori delle variabili a cui corrisponde.

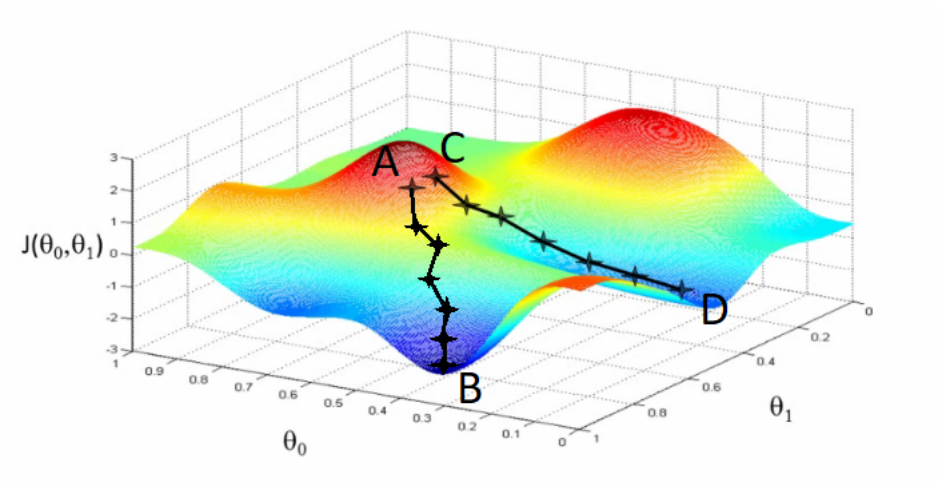


Figura 2.4: Grafo di discesa del gradiente con due percorsi differenti.

Come si può osservare questo procedimento è di natura stocastica, per due punti relativamente vicini come A e C i risultati della discesa del gradiente risultano estremamente differenti e distanti. Per questo sono stati sviluppati metodi stocastici come il calcolo del *learning rate*, trattato nella prossima sezione, dinamico per eseguire la risalita del gradiente in modo da poter trovare altre discese da percorrere per raggiungere minimi locali sempre minori.

2.1.1.3.3 Backpropagation

Il processo di apprendimento di una rete neurale si scompone in due fasi principali: la prima chiamata *forward propagation* definisce la propagazione dell'informazione nella rete neurale dall'*input* inserito nel primo strato fino al raggiungimento di un *output*; la seconda parte viene chiamata *backpropagation*, l'*output* predetto viene confrontato con quello reale e l'errore viene propagato all'indietro, come dice il nome, per ogni *layer* della rete in modo da aggiustare i pesi sinaptici approssimando la predizione della rete neurale finchè non riuscirà a produrre l'*output* desiderato. L'aggiornamento dei pesi viene eseguito tramite la seguente formula:

$$W_i = W_{i-1} - \alpha \frac{d}{dw} Loss(w)$$

Dove W_i viene chiamato passo i-esimo e rappresenta il vettore dei pesi sinaptici che partono dal *layer* precedente all'i-esimo connessi al *layer* i-esimo. Dunque i pesi del passo i-esimo vengono aggiornati assumendo il valore dei pesi del passo precedente a cui viene sottratta la derivata dell'errore stimato moltiplicata per un valore α chiamato *learning rate*.
Il *learning rate* viene utilizzato per regolare la velocità di apprendimento della rete in modo da raggiungere i minimi locali di errore, ma come mostrato in figura 2.5 un buon *learning rate* deve essere scelto in modo da evitare i due principali problemi che ne derivano:

- ***Overshooting***: per un *learning rate* troppo elevato ne corrisponde una velocità di apprendimento altrettanto elevata, ciò può causare l'aggiustamento dei pesi specializzato maggiormente per l'*input* corrente perdendo i progressi ottenuti con i precedenti, dal punto di vista matematico viene saltato il minimo locale;

- ***Undershooting***: se invece la velocità di apprendimento è troppo lenta ciò che può accadere è un rallentamento nell'apprendimento seguito dal suo stesso blocco nel primo minimo locale trovato.

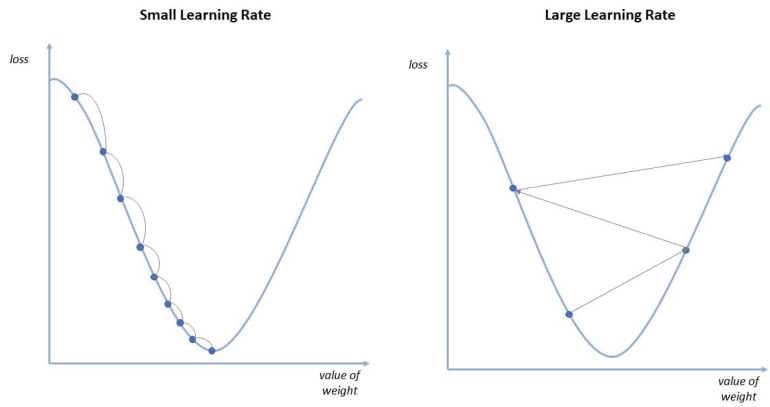


Figura 2.5: Grafi illustranti la differenza tra un *learning rate* basso e uno alto, causando rispettivamente *undershooting* e *overshooting*.

2.1.1.4 Applicazione in computer vision

L'applicazione del *machine learning*$_{\mathbf{G}}$ nel settore della *computer vision*$_{\mathbf{G}}$ ha portato allo sviluppo di diversi campi di ricerca e utilizzo di modelli di reti neurali per il riconoscimento di oggetti, arrivando a produrre la varietà di *output* illustrati in figura 2.6, i quali vengono prodotti dalle seguenti tecnologie:

- ***Object recognition***: si tratta del primo metodo di *machine learning*$_{\mathbf{G}}$ sviluppato per il riconoscimento di oggetti, permette di riconoscere gli oggetti nelle immagini ricevendo come *output* della rete neurale i nomi delle classi e la probabilità che corrispondano alle classi predette;

- ***Object detection***: evoluzione dell'*object recognition*, permette di riconoscere svariati oggetti per immagine definendo anche la loro posizione e dimensione con le *bounding boxes*$_{\mathbf{G}}$;

- ***Semantic segmentation***: lo stadio successivo del riconoscimento di oggetti ha portato all'assegnazione di una classe per ogni pixel dell'immagine;

- ***Instance segmentation***: l'evoluzione di *semantic segmentation*, oltre ad assegnare una classe per ogni oggetto si assicura di discernere i differenti oggetti l'uno dall'altro.

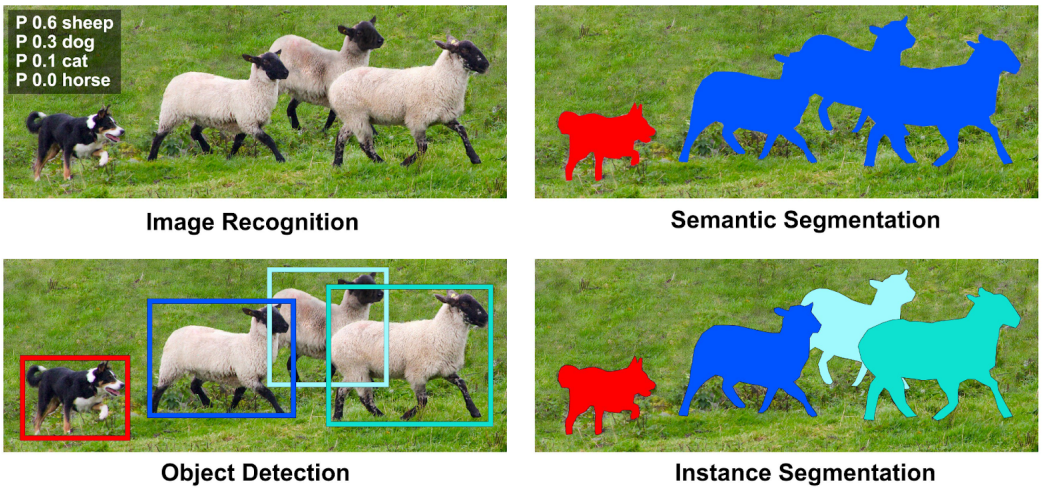


Figura 2.6: Confronto tra gli *output* prodotti per le tecniche di: *object recognition*, *semantic segmentation*, *object detection*, *instance segmentation*.

2.1.1.5 Architettura di una rete neurale per l'object detection

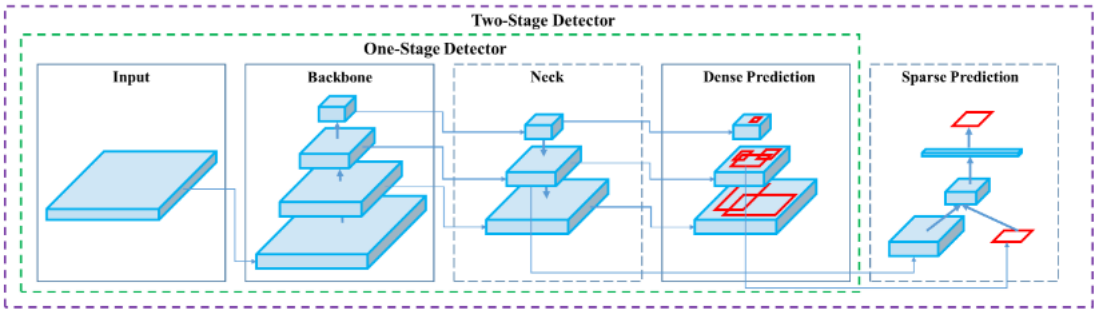


Figura 2.7: Architettura di una rete neurale performante *object detection*.

Come illustrato in figura 2.7, l'architettura dei modelli di *object detection* è suddivisa in tre principali strati: ***Backbone***, ***Neck*** e ***Head***.

2.1.1.5.1 Backbone

Backbone, lo strato portante dell'intera architettura, serve per ridurre la dimensionalità dell'immagine data in *input*, al fine di facilitare l'elaborazione dello strato successivo e la conseguente estrazione delle $feature_{G}$ importanti per l'identificazione degli oggetti. Il suo funzionamento si focalizza sull'estrazione dei *pattern* simili a quelli appresi durante il processo di allenamento, al fine di ridurre i tempi computazionali fornendo i riscontri trovati tra l'immagine e i *pattern* nel formato più facilmente elaborabile dagli strati di neuroni presenti in ***neck***, lo strato successivo della rete neurale.

Se si utilizza un modello già allenato è possibile applicare la tecnica di *transfer learning*$_{\mathbf{G}}$, generando un *dataset* specifico al caso d'uso. Quanto descritto è possibile tramite il congelamento dello strato di backbone, concentrando l'apprendimento negli strati successivi. Questo stesso processo può essere applicato per il *fine tuning*$_{\mathbf{G}}$ che permette di raggiungere una precisione maggiore del modello in fase di predizione non supervisionata.

2.1.1.5.2 Neck

Neck è costituito dagli strati di neuroni responsabili dell'elaborazione dell'immagine pre-processata da **backbone**, tramite gli *hyper-parametri* qui presenti, al fine di estrarre le *feature*$_{\mathbf{G}}$ necessarie per l'identificazione delle classi.

2.1.1.5.3 Head

Lo strato head viene utilizzato per l'elaborazione delle *feature*$_{\mathbf{G}}$ provenienti da **neck** per identificare la classe a cui appartengono con maggiore probabilità, traducendola nel punteggio di confidenza, definito in percentuale; l'*output* di questo strato consiste in un vettore di dimensione corrispondente alle *bounding boxes*$_{\mathbf{G}}$ massime predicibili e per ognuna di esse sono presenti le coordinate assieme ai punteggi di confidenza per tutte le classi con cui è stato allenato il modello.

2.1.2 Ricerca del modello di intelligenza artificiale

Sono state analizzati i seguenti modelli e tecnologie di estrazione delle *feature*$_{\mathbf{G}}$:

- **Fast R-CNN**;
- **Faster R-CNN**;
- **OpenCV HOG**;
- **OpenCV SURF**;
- **Inception**;
- **SSD**;
- **YOLO**.

Inizialmente sono stati considerati i modelli che utilizzano architetture **R-CNN** (*Recurrent Convolutional Neural Network*), ma le basse performance nei tempi di allenamento e rilevazione hanno costretto a cercare delle alternative. Successive ricerche, riguardanti modelli creati per il riconoscimento di cibo e sushi, hanno portato alla valutazione dell'implementazione di estrattori di *features*$_{\mathbf{G}}$ come **SURF** superato da **HOG**, scoperto in un secondo momento, per la qualità delle *features*$_{\mathbf{G}}$ estratte. Adottando l'architettura della rete neurale **inception**, come struttura delle rete neurale

finale, nel suo strato di **backbone** verrebbe applicato **HOG** per l'estrazione delle *features*$_G$ di forme e parallelamente ad esso una variante di *fisher vector*$_G$ per l'estrazione di *features*$_G$ della gamma cromatica dei tipi di pesce presenti nel sushi. Ma la **mAP** (*mean average precision*) non era sufficiente per dei modelli che utilizzano **inception** con **HOG**, ciò è comprensibile essendo queste soluzioni datate; la più recente è stata *FoodCam* creata nel 2009 la cui architettura di rete neurale è stata aggiornata l'ultima volta attorno al 2015.
Dunque si è giunti allo studio della rete neurale **SSD** che permette di avere un tempo di allenamento e elaborazione più corti rispetto alle alternative appartenenti alla classe delle **R-CNN** grazie al suo principio "*Single Shot Detector*". Con un principio simile **YOLO** "*You Only Look Once*" sono state prese in considerazione per il confronto finale.
Dopo ulteriori ricerche si è evinto che le reti SSD e YOLO sono molto simili, visto che utilizzano una tecnologia analoga per la struttura dell'architettura e che le performance dipendono dall'obiettivo di impiego. Perciò, in specifici casi SSD raggiunge prestazioni e precisione migliori di YOLO e viceversa in altri casi. Per il confronto sono stati riassunti i pro e i contro di entrambe nella tabella 2.1.

SSD	**YOLO**
Single Shot Detector	*You Only Look Once*
Esegue una rete convoluzionata una sola volta per ogni immagine di *input* e elabora la mappa delle *feature*$_G$.	Essendo un progetto *open source*$_G$ può essere allenata con *transfer learning*$_G$ facilmente data la varietà di progetti presenti.
È la scelta migliore per l'elaborazione di video nei quali la precisione risulta modesta.	È considerata l'alternativa migliore per il rapporto tra prestazioni di precisione e velocità di allenamento ed elaborazione degli *input*.
Quando gli oggetti sono piccoli tende a sbagliare la predizione.	Riesce a riconoscere gli oggetti anche quando sono piccoli con un margine di errore ridotto rispetto alle sue alternative.
Tende ad essere considerata l'alternativa più veloce nell'elaborazione delle immagini in *input*.	Utilizzata, e con potenzialità di implementazione, per modelli di intelligenza artificiale utilizzati in campi radicalmente differenti con prestazioni finali simili come per guida autonoma e riconoscimento del cancro.

Tabella 2.1: Tabella del confronto tra SSD e YOLO.

Riguardo al caso d'uso di questo progetto è stato scelto YOLO data la sua capacità di riconoscere piatti da elementi piccoli e vari considerando anche le sue prestazioni per ottenere un modello ottimale da utilizzare in un applicativo mobile e le librerie *open source*$_G$ presenti nel progetto di cui fa parte: g*OpenCV*; un gruppo di librerie per *real-time computer vision*$_G$ utili in svariati casi, nel progetto corrente si è rivelato utile principalmente per l'elaborazione delle immagini in modo da ottimizzarle prima che vengano date in *input* alla rete neurale e per poter eseguire la rete neurale su dispositivi mobili grazie alla versione convertita a libreria *SDK*$_G$ per l'utilizzo in **Java** su Android$_G$ di tutte le librerie compilate in **C++**.

Infine la ricerca ha portato allo studio delle varie versioni di YOLO, il quale grazie all'applicazione di HOG in un'architettura di *deep learning*$_G$ estrae *features*$_G$ notevolmente più ricche rispetto alle reti neurali alternative.
Essendo le ultime versioni (v6 e v7) ancora in fase di sviluppo e con poca documentazione non sono state considerate come mature per essere utilizzate nel progetto. Per questo motivo sono state confrontate le versioni di YOLO fino alle varianti del modello v5.
L'unico modello valida alternativa alla versione 5 è **PP-YOLO** basato su **YOLOv4** risulta infatti dotato di una velocità di elaborazione leggermente maggiore dei modelli di **YOLOv5**, ma presenta una precisione minore.
Per questo motivo si è preferito considerare l' utilizzo delle varietà di YOLOv5 che sono: *nano*, *small*, *medium*, *large* e *extra large*. La loro grandezza definisce la quantità di neuroni e conseguenti connessioni sinaptiche per gli strati nascosti della rete neurale i quali invece rimangono di numero inalterato al variare della dimensione.
Dati i limiti di risorse dell'hardware sono state utilizzate per l'allenamento le varietà dalla *nano* alla *medium* con volontà futura di sperimentare anche *large* e *extra large*.

2.1.3 Evoluzione di YOLO

Le versioni di YOLO verranno definite come YOLOv_ dove _ rappresenta il numero della versione.

2.1.3.1 YOLOv1

YOLO nasce dall'idea di Joseph Redmon nel 2015 come progetto *open source*$_G$ a licenza libera per una rete neurale integrante le maggiori caratteristiche che rendevano le alternative R-CNN lo stato dell'arte nel campo dell' *object detection*, ma con prestazioni migliori.
La prima versione del modello riesce a riconoscere oggetti da sorgente video a 45 FPS (*Frames Per Second*) e superiori, con una scheda video di ultima generazione di quel tempo, ottenendo un mAP (*mean Average Precision*) doppio rispetto alle altre reti come Fast R-CNN.
Il problema con le R-CNN è che dividono l'immagine in tante regioni indicate come ROI (*Region of Interest*) e di queste si verifica singolarmente se all'interno di ognuna ci sia o meno una *feature*$_G$, impiegando una quantità considerevole di tempo per l'analisi di una singola immagine.
In YOLO invece l'analisi viene fatta su tutta l'immagine; questo tipo di approccio viene chiamato *Single Shot Detection*.

YOLO si basa su un principio semplice ma efficace: legge l'immagine di *input* frazionandola tramite una griglia di dimensioni $N * N$. Viene aumentata così la precisione di riconoscimento delle classi grazie all'allenamento e successivo utilizzo su dettagli degli oggetti da riconoscere piuttosto che sull'oggetto completo. Grazie a questo metodo il riconoscimento può avvenire anche tramite immagini ritraenti solo parti degli oggetti classificati.
Ogni cella della griglia può prevedere un solo oggetto per volta e un numero limitato (B) di *bounding boxes*$_{\mathbf{G}}$. Queste ultime sono riquadri di dimensioni variabili rispetto all'oggetto interessato che viene riconosciuto all'interno dell'immagine.

Ogni *bounding box*$_{\mathbf{G}}$ racchiude cinque elementi: x, y, w, h e il punteggio di confidenza dell'oggetto rilevato, dove:

- x e y definiscono il centro della *bounding box*$_{\mathbf{G}}$ con un valore tra 0 e 1, dove 1 è la dimensione corrispondente dell'immagine, larghezza o altezza;

- w e h sono le dimensioni di larghezza e altezza della *bounding box*$_{\mathbf{G}}$ sempre relative all'intera immagine, rappresentate con lo stesso principio di x e y;

- Il punteggio di confidenza è il valore in percentuale che il modello produce relativamente alla 'convinzione' del riconoscimento di uno specifico oggetto e quanto sia accurato rispetto all'oggetto stesso.

Lo strato di neuroni finale dell'architettura è un tensore$_{\mathbf{G}}$ di dimensione $(N, N, B*5+C)$ dove C è il numero di classi con cui è stato addestrato il modello.
L'architettura della rete è di tipo CNN composta da 24 *layer* seguiti da 2 layer completamente interconnessi come mostrato nella figura 2.8.

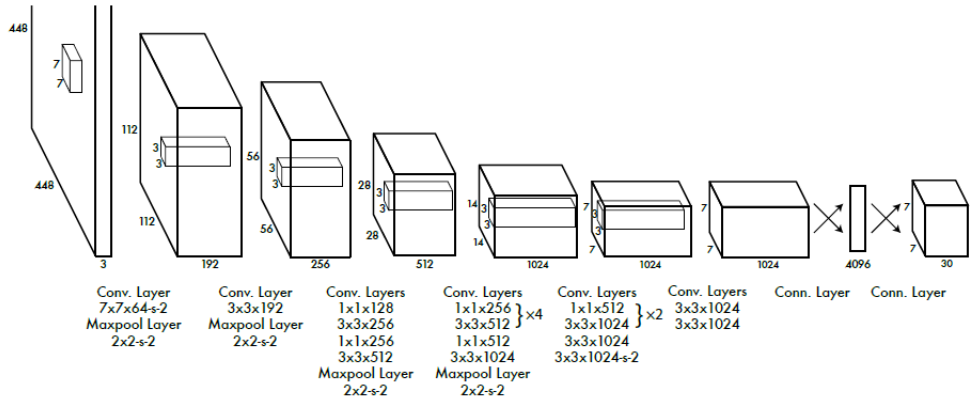


Figura 2.8: Architettura della prima versione di YOLO.

Inoltre, la griglia è di grandezza variabile per permettere il riconoscimento di oggetti di varie dimensioni, più celle sono presenti più piccoli saranno gli oggetti in grado di essere riconosciuti dal modello. Questo metodo presenta dei problemi palesi, nel momento in cui lo stesso oggetto viene riconosciuto in più celle adiacenti si presenteranno diverse *bounding boxes*$_{\mathbf{G}}$ sovrastanti come *output*, questo problema viene ovviato grazie alla formula **IoU** *(Intersection Over Union)*:

$$IoU = \frac{Areadiintersezione}{Areaunita}$$

In questo modo, più alto sarà il valore della IoU presso la *bounding box*$_{\mathbf{G}}$ corretta definita dall'addetto all'allenamento, maggiore sarà il punteggio di riconoscimento, questo processo continua fino a quando le aree riconosciute inizieranno a sovrapporsi fornendo una precisione maggiorata. Lo stesso metodo viene applicato per il riconoscimento non supervisionato considerando la *bounding box*$_{\mathbf{G}}$ corretta come quella con confidenza maggiore, unendo le predizioni dello stesso oggetto nelle celle adiacenti per definire la *bounding box*$_{\mathbf{G}}$ completa.

YOLO fa anche uso del metodo di **NMS** *(Non Maximum Suppression)* che esclude le *bounding box*$_{\mathbf{G}}$ con confidenza minore lasciando solo quella dal punteggio maggiore per evitare duplicazioni di riconoscimento dello stesso oggetto.

Tuttavia la rete YOLO non è immune da problemi: infatti non riesce a rilevare correttamente oggetti piccoli molto vicini come nel caso di stormi di uccelli o file di formiche, non essendo capace di estrapolare il riconoscimento di tutti i singoli animali. Ad ogni modo il risultato è nettamente superiore rispetto alle sue concorrenti perché riesce a generalizzare meglio e ad adattarsi per una varietà molto vasta di ambienti.

2.1.3.2 YOLOv2

Quasi un anno dopo dal suo debutto, venne rilasciata una nuova versione della rete YOLO che introduceva i seguenti miglioramenti:

- ***Batch Normalization***: e' stato introdotto normalizzando gli *input* dei *layer* di convoluzione$_{\mathbf{G}}$, aumentando di 2 punti percentuale la mAP, eliminando il *dropout*$_{\mathbf{G}}$ dalla rete.

- **Classificatori a risoluzione maggiore**: la prima versione di YOLO riduceva la dimensione dell'immagine per l'addestramento a 224x224 pixel per poi utilizzare immagini del doppio della dimensione per il rilevamento. Nella nuova versione viene introdotta un'operazione di *tuning* (aggiustamento prestazioni) successivo all'allenamento con immagini di 448x448 pixel in modo da ottimizzare i pesi sinaptici per migliorare la precisione di rilevamento.

- **Diminuzione della dimensione dell'immagine in *input***: per quanto riguarda il rilevamento non supervisionato si è ridotta la dimensione dell'immagine di *input* da 448x448 a 416x416 pixel in modo da ottimizzare il centro dell'immagine quando la riduzione dell'immagine viene fatta a un trentaduesimo della risoluzione (rispetto alla risoluzione media delle foto del tempo). Data l'alta probabilità che le immagini in *input* presentino al centro dell'immagine il centro dell'oggetto da riconoscere, questo cambiamento permette di migliorare la precisioni di questi

casi ponendo il centro dell'oggetto al centro della cella centrale della griglia che YOLO utilizza per suddividere l'immagine di *input*.

- ***Anchor boxes***$_{\mathbf{G}}$: sono stati rimossi i due strati completamente interconnessi presenti alla fine della rete, che si occupavano di prevedere la posizione delle *bounding boxes*$_{\mathbf{G}}$, per essere sostituiti da *layer* utilizzanti il principio delle *anchor boxes*$_{\mathbf{G}}$. Questo è dato dal fatto che non tutti gli oggetti presentano la stessa forma e quindi le *anchor boxes*$_{\mathbf{G}}$, definite per le dimensioni ottimali di ogni oggetto, facilitano il riconoscimento di oggetti differenti. Inoltre questo approccio ha permesso di riconoscere più oggetti nella stessa cella.

- ***Multi-scale Training***: non essendoci più i due strati completamente interconnessi alla fine della rete non abbiamo più vincoli sulla dimensione dell'immagine. Quindi, ogni 10 epoche, la rete durante la fase di allenamento utilizza immagini di dimensione diversa da quella standard per abituare la rete a performare il rilevamento su immagini con dimensioni diverse.

- ***Dimension Cluster e Direct Location prediction***: per individuare il numero appropriato di *bounding boxes*$_{\mathbf{G}}$ è stato usato l'algoritmo *k-mean clustering*$_{\mathbf{G}}$ che ha portato al miglior risultato con $k = 5$. Grazie a questo metodo la posizione delle *bounding boxes*$_{\mathbf{G}}$ viene abbinata a quella delle *anchor boxes*$_{\mathbf{G}}$ nelle quale sono stati rilevati gli oggetti interessati.

- ***Fine-Grained Features***: sfruttando l'approccio *passthrough* ovvero il collegamento dell'*output* di un *layer* di strato n-esimo con quelli precedenti, si riesce a ricostruire un'immagine meno deteriorata (grazie al lavoro dei *layer* di convoluzione$_{\mathbf{G}}$) così da rilevare anche oggetti più piccoli e sgranati nell'immagine di *input* data la sua scarsa risoluzione.

Tutte queste innovazioni, e altre minori, hanno portato a un notevole incremento delle prestazioni e della precisione, come si può notare dallo schema presente in figura 2.9.

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?								✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figura 2.9: La rete YOLOv2 con le migliorie apportate e il suo risultato sul *dataset* VOC 2007.

Un altro importante cambiamento è stato il passaggio dal framework di *backbone* implementato come variazione di **GoogleNet** a **Darknet-19**, un framework *open source*$_{\mathbf{G}}$ per reti neurali scritto in C e CUDA$_{\mathbf{G}}$, ottimizzato quindi per dispositivi di elaborazione grafica NVIDIA. Si è giunti così a migliorare l'accessibilità, i tempi di allenamento e ciò ha attirato parecchi ricercatori in grado di allenare modelli complessi nel loro computer privato.
Per diversi mesi YOLOv2 ha rappresentato lo stato dell'arte nel settore dell' *object detection* riuscendo ad ottenere prestazioni fino a quel momento mai raggiunte nel settore.

2.1.3.3 YOLOv3

Con il continuo sviluppo di nuove tecniche e l'aiuto di una comunità sempre più popolosa, ben presto YOLOv2 fu resa obsoleta da altre reti come **RetinaNet** e ***Light-Head R-CNN*** che la superarono sia in precisione che velocità.
I nuovi metodi inclusero l'utilizzo dei ***residual block***$_{\mathbf{G}}$ introdotti da ResNet, di ***upsampling*** ovvero portare l'immagine ad una risoluzione maggiore con migliori dettagli e l'utilizzo delle ***skip connections***, ovvero la capacità di fornire l'*output* dei *layer* della rete neurale agli *input* di strati successivi a quello immediatamente prossimo.
Due anni dopo YOLOv2 sorse YOLOv3 che incorpora tutte queste nuove tecniche, dimostrando ancora una volta la potenzialità delle reti *Single Shot Detector* e le capacità del *framework* Darknet evolutosi a **Darknet-53**. Rispetto alla versione precedente cambia l'architettura del *framework* di *backbone* Darknet: YOLOv2, infatti, si basava su **Darknet-19** ovvero una rete composta da 19 *layer* di convoluzione$_{\mathbf{G}}$ più altri 11 per il rilevamento di oggetti con un totale di 30 *layer*. Invece la variante **Darknet-53** porta l'aumento dei *layer* di convoluzione$_{\mathbf{G}}$ a 53 e altrettanti *layer* di rilevamento con un totale di 106 *layer*, permettendo un miglioramento generale della precisione soprattutto sugli oggetti di piccole dimensioni.

L'introduzione di questi *layer* ha portato ad un rallentamento delle prestazioni rispetto alla sua versione precedente, ma guadagnando precisione la terza versione è riuscita a superare le concorrenti RetinaNet e SSD sulla metrica mAP50, ovvero accettando *bounding boxes*$_{\mathbf{G}}$ con confidenza superiore al 50%.
Aumentando però la richiesta di precisione, arrivando al 75% di confidenza si poteva notare che la precisione effettiva di YOLOv3 non riusciva a raggiungere quella ottenuta con RetinaNet, ma comunque riusciva ad essere più veloce, comportando un modesto *trade-off* tra precisione e velocità di rilevamento.

2.1.3.4 YOLOv4

Dopo altri due anni arrivò la prima versione di YOLO non supervisionata dal suo creatore Joseph Redmon in conseguenza a problematiche morali legate ad utilizzi militari e poco etici. Alexey Bochkovskiy partecipe al percorso intrapreso da YOLO a partire dalle prime versioni divenne il supervisore di YOLOv4.
Questa versione portò innovazioni per raggiungere il nuovo stato dell'arte nel campo dell'*object detection* grazie all'evoluzione dell'architettura del *framework* di *backbone* ora composto da:

- ***Bag of freebies***;
- ***Bag of specials***;

- **CSPDarknet-53**.

2.1.3.4.1 Bag of freebies

La *bag of freebies* è un insieme di metodi che aumentano il costo di allenamento e le sue strategie facendo restare basso il costo di elaborazione della rete. Principalmente si tratta di strategie di ***data augmentation*** il cui scopo è quello di aumentare la variabilità delle immagini in modo da raggiungere una generalizzazione più efficace in ambienti e contesti diversi per il riconoscimento degli oggetti, alcune di queste sono:

- **Distorsione Fotometrica**: creazione di nuove immagini tramite la modifica di luminosità, hue, contrasto, saturazione e rumore per creare varianti della stessa immagine;
- **Distorsione Geometrica**: si tratta di creare alternative di un'immagine girandola, rendendola speculare, ridimensionandola in dimensioni casuali e ritagliandola;
- **MixUp**: combinazione di due immagini, in modo da creare una nuova immagine, tramite la loro sovrapposizione con l'ausilio di pesi casuali per entrambe, dove il peso varia da 0 per totale trasparenza a 1 per totale nitidezza. Nell'immagine finale saranno presenti le classi corrispondenti a quelle dell'immagine con peso maggiore presentando del rumore prodotto dall'immagine con peso minore;
- **CutMix**: sezionamento di due parti casuali, ma identiche per due immagini così da scambiarle tra di esse creando due nuove immagini ricche di rumore.

2.1.3.4.2 Bag of specials

La *bag of specials* è composta da metodi che aumentano leggermente il costo di elaborazione della rete neurale per permettere un significativo miglioramento della precisione, sono principalmente modifiche dell'architettura della rete neurale, di cui la più importante innovazione:

- **Funzione di Attivazione *Mish***: una funzione innovativa di attivazione regolarizzata autonomamente non monotona definita come:
$$f(x) = x \tanh(softplus(x))$$
La sua funzione è quella di migliorare le prestazioni di allenamento impedendo il fenomeno di saturazione della funzione di attivazione **ReLU** (*Rectified Linear Unit*) causata da alti $bias_{\mathbf{G}}$ negativi che impediscono l'aggiustamento dei pesi sinaptici in fase di *backpropagation*.

2.1.3.4.3 CSPDarknet-53

Dalla versione precedente che implementa Darknet-53, CSPDarknet-53 ne è una variante che implementa l'architettura ricorrente tra i *layer* di neuroni nascosti ispirandosi alle nuove reti ricorrenti che definirono lo stato dell'arte del settore. L'acronimo **CSP** definisce la nuova architettura Darknet: *Cross Stage Partial*, ispirata dall'architettura DenseNet la quale definisce gli *input* di ogni strato nascosto concatenando gli *output* del *layer* precedente assieme ai suoi *input* e a quelli di tutti gli strati di neuroni precedenti come rappresentato di seguito:

$$x_1 = w_1 * x_0$$
$$x_2 = w_2 * [x_0, x_1]$$
$$\vdots$$
$$x_n = w_n * [x_0, x_1, ..., x_{n-1}]$$

Dove $*$ si riferisce all'operatore di convoluzione$_{\mathbf{G}}$, x_i e w_i rappresentano rispettivamente gli *output* e i pesi sinaptici del *layer* i-esimo.
L'architettura **CSP** si differisce da DenseNet perchè divide l'*input* i-esimo in $x_{0'}$ e $x_{0''}$ andando a fornire come *input* del *layer* i-esimo $x_{0''}$ concatenato con gli *output* degli strati precedenti, mentre $x_{0'}$ verrà concatenato all'*input* del *layer* successivo come illustrato nelle seguenti equazioni:

$$x_k = w_k * [x_{0''}, x_1, ..., x_{k-1}]$$
$$x_i = w_i * [x_{0''}, x_1, ..., x_k]$$
$$x_n = w_n * [x_{0'}, x_i]$$

In questo modo differenti *layer* apprendono ripetutamente le informazioni copiate nel gradiente.

2.1.3.5 YOLOv5

Rilasciata da Ultralytics un mese dopo la quarta versione, YOLOv5 non è una versione ufficiale di YOLO e manca di documentazione, ma porta innovazione nell'evoluzione della ormai famosa rete neurale e vuole essere comunque *open source*$_{\mathbf{G}}$ a licenza libera come le sue versioni precedenti.
Utilizzando il *framework* di *backbone* della versione precedente convertito da **Darknet** a **PyTorch** la nuova versione di YOLO permette un allenamento nettamente più veloce grazie ai tensori ottimizzati di PyTorch altamente compatibili con i driver CUDA$_{\mathbf{G}}$ per elaboratori grafici Nvidia. Questa conversione permette la modifica ed evoluzione della rete neurale in modo estremamente più semplificato per gli esperti del settore. Rispetto al *framework* Darknet PyTorch è molto più utilizzato e semplice da comprendere. Inoltre viene consentita la modifica tramite i file di configurazione *.yaml* per ottimizzare l'intera architettura rispetto al caso in analisi. Invece fino alla versione precedente era possibile trovare nella cartella Darknet del progetto i file *.cfg* che permettevano di ottimizzare le caratteristiche durante la fase di apprendimento, non consentendo però dirette modifiche all'architettura.

Le innovazioni principali portate da questa versione consistono in:

- ***Mosaic***: un nuovo metodo di ***data dugmentation***, come dice il nome, consiste nella creazione di mosaici come immagini uniche incorporando molteplici immagini presenti nel *dataset* aggiungendo il rumore generato dai diversi *background* per permettere un apprendimento più discriminante. Le immagini così composte, ridotte di risoluzione, permettono un allenamento più preciso nella rilevazione di classi estremamente piccole;

- **Algoritmi genetici**: sono algoritmi nativamente sviluppati dal concetto di selezione naturale tramite la suddivisione in geni delle possibili soluzioni e ibridazione delle migliori, affinché dopo svariate generazioni possa nascere una soluzione ottimale. In YOLO gli algoritmi genetici vengono utilizzati per ottimizzare

l'apprendimento tramite la ricerca di *hyper-parametri* migliori per la rilevazione delle classi definite nel *dataset*;

- **Riduzione del problema di sensibilità della griglia**: fino alla precedente versione l'equazione utilizzata per il calcolo dell'errore delle *bounding boxes*$_{\mathbf{G}}$ subiva sporadici errori nel calcolo dell'altezza e larghezza delle stesse. Questo causava il rischio di perdere il progresso nell'apprendimento avvenuto fino a quel momento per l'operazione di *backpropagation* che andava a correggere un errore non effettivo. Il calcolo dell'errore delle *bounding boxes*$_{\mathbf{G}}$ è definito dalla distanza di quelle fornite dall'operatore umano rispetto a quelle predette dalla rete neurale. La riduzione di questo problema è stata apportata tramite la ridefinizione dei calcoli per le coordinate e dimensioni delle *bounding boxes*$_{\mathbf{G}}$ predette, ciò non corregge definitivamente il problema, ma permette di continuare l'apprendimento tramite la riduzione del gradiente per la *backpropagation* che non va più ad avere un effetto eccessivo sui parametri della rete neurale.

2.2 Creazione del Dataset

Il problema maggiore nell'implementazione di una soluzione di *machine learning*$_{\mathbf{G}}$ è trovare un *dataset* sufficientemente ampio e vario per la fase di training del modello. Oggigiorno sono presenti milioni di immagini gratuite in *dataset* localizzati su svariati domini online, ma sono ancora molto limitati per varietà di classi e indicizzazione.

Numerose fonti sono state utilizzate e consultate per poter realizzare un *dataset* contenente un numero sufficiente di tipologie di sushi, tale da poter allenare il modello e ottenere performance adeguate. Tra questi:

- **Kaggle**: una filiale di Google e comunità di *data scientists* fornisce un servizio di *cloud computing* che utilizza il formato *notebook jupyter* per l'allenamento di modelli di *machine learning*$_{\mathbf{G}}$, contiene più di 50'000 datasets pubblici;
- **Free image**: è un servizio che fornisce immagini *royalty free* perché vengano utilizzate liberamente;
- **Weight and biases**: è un servizio di *repository* per modelli di reti neurali che presenta molteplici artefatti di *dataset* riguardanti il sushi, senza distinzione tra i vari tipi, ma almeno con le bouding box già presenti;
- **Google immagini**: molti tipi di sushi e piatti orientali non erano presenti in nessun *dataset* pubblico, dunque l'unico metodo disponibile è stato quello di crearne uno con le scarse immagini indicizzate da Google.

2.3 Cloud computing per l'allenamento

Per l'allenamento di modelli di *machine learning*$_{\mathbf{G}}$, due sono i principali provider che forniscono un servizio gratuito di *cloud computing* con elaboratori grafici dedicati: **google colaboratory** e **gradient di paperspace**. In questo progetto, si è optato per quest'ultimo servizio (paperspace) per le risorse superiori a disposizione. I due servizi sono confrontanti per le risorse fornite nella tabella 2.2.

Service Provider	**Google colaboratory**	**Paperspace gradient**
CPU	2 core	8 core
RAM	12 GB	30 GB
GPU	Tesla K80 12GB VRAM (ma solo il 5% di VRAM disponibile per il piano gratuito)	quadro m4000 8GB VRAM
Storage	100 GB	5 GB

Tabella 2.2: Tabella del confronto tra Google collaboratory e Gradient di paperspace.

I due principali candidati per l'utilizzo dei documenti computazionali ***jupyter notebook*** e annesso *cloud computing* a piano gratuito sono stati Google colaboratory, usato nel corso di introduzione all'apprendimento automatico, e successivamente a svariate ricerche il servizio gradient di paperspace.

Google colaboratory presenta senza ombra di dubbio l'interfaccia grafica più accessibile e intuitiva presente per questo tipo di servizio, dopotutto è stata Google a creare il formato *jupyter notebook*, ma presenta dei limiti computazionali: YOLOv5 nano (il modello di YOLOv5 che richiede meno calcolo computazionale) impiega 5 minuti per allenare una epoca utilizzando come *input* un *dataset* di 200 immagini, una quantità di tempo spropositata considerando che sotto le 150 epoche un modello di questo tipo non è allenato sufficientemente neanche per permettere un processo di *transfer learning*$_{\text{G}}$ efficiente. Questo problema sorge dal fatto che nonostante siano dedicate delle schede video **Tesla K80** con **12GB di VRAM** dedicata, esse vengano condivise tra molti utenti contemporaneamente rendendo disponibile solo il 5% dello spazio di memoria presente, e quindi riduce drasticamente le prestazioni.

Esistono moltissimi provider di questo servizio, come kaggle citato in precedenza, ma presentano tutti limiti notevoli confrontati con Google colaboratory; nonostante ciò dopo una lunga ricerca ho optato per lo strumento gradient disponibile nel sito di paperspace che rende disponibile a piano gratuito una **quadro m4000** con **8GB di VRAM** dedicata interamente destinata ad un singolo utente per un massimo di 6 ore a sessione, questo ha permesso di ridurre i tempi a 15 secondi per epoca, e con il *dataset* completo un minuto.

Il limite di gradient è lo spazio disponibile per memorizzare i dati che è di soli **5GB** portando l'esecuzione a fallire non appena i modelli salvati tra le epoche di allenamento saturano questo spazio. La soluzione qui proposta è quella di ridurre al minimo la dimensione delle immagini del *dataset* e salvare i modelli ogni 10 epoche di allenamento. Lo spazio di archiviazione è limitato se confrontato ai **100GB** di Google colaboratory i quali però vengono cancellati automaticamente al fine della sessione, mentre su gradient ciò non avviene, rendendo notevolmente più veloce la ripresa dell'allenamento senza la

necessità di scaricare nuovamente la libreria di YOLO e il *dataset* sulla macchina in remoto.

Per quanto riguarda l'interfaccia grafica e le possiblità di interazioni *input*, gradient è notevolmente limitato in confronto a Google colaboratory, ma nonostante la sua maggiore complessità di utilizzo è sufficientemente intuitivo, l'interfaccia grafica è illustrata in figura 2.10.

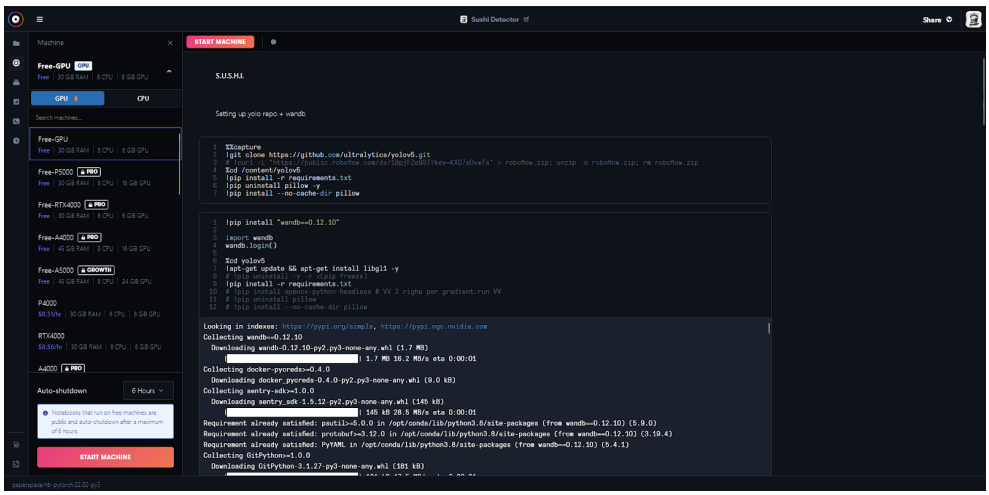


Figura 2.10: Schermata di Gradient per l'esecuzione di *script* in formato *jupyter notebook*.

2.4 Repository

Per permettere un accesso facilitato del *dataset* e salvaguardia delle epoche di allenamento della rete neurale è stata necessaria la ricerca di un servizio web di *repository* in grado di gestire queste necessità. Principalmente il confronto è avvenuto tra:

- **Roboflow**;
- **Weight and Biases**;
- **Comet**;
- **Valohai**;
- **Neptune**.

Questi servizi sono molto simili, permettono la gestione delle epoche prodotte dal modello in allenamento e la segnalazione di problemi in questo processo per un ottimale *workflow* di gruppi di ricercatori, permettono inoltre di salvare i *dataset* creati e generano autonomamente i grafi rappresentanti le prestazioni raggiunte durante l'allenamento. Nello specifico **Valohai** si contraddistingue per la possibilità di implementare strumenti per automatizzare il processo di *continuous delivery* al fine di testare sempre l'ultima versione del modello caricato con un *dataset* non incluso in quello di allenamento, per stimare più precisamente le prestazioni rappresenta un'ottima opzione per ricercatori esperti.

Anche se quasi tutti sono a pagamento presentano una prova gratuita di un mese in cui si possono familiarizzare, ma essendo l'esperienza di stage riguardante un periodo di due mesi e il budget dell'azienda limitato per un servizio secondario come questo, la scelta è stata ristretta verso l'unica *repository* completamente gratuita: **Weight and Biases**.

2.4.1 Weight and Biases

Il link alla repository con dataset e grafi di confronto tra YOLOv5 *nano*, *small* e *medium* è presente nella bibliografia.

Repository dedicata a progetti di *machine learning*$_{\mathbf{G}}$ permette la conservazione di *dataset* e il tracciamento dell'allenamento del proprio modello tramite file di configurazione semplici da implementare.

YOLO è stata configurata nel progetto *open source*$_{\mathbf{G}}$ della versione 5 così da essere utilizzabile collegandone una *repository* weight and biases direttamente nel comando di allenamento della rete neurale, effettuando l'accesso al proprio account e specificando il nome del progetto esistente su weight and biases direttamente da riga di comando.

2.4.1.1 Artefatti

Gli artefatti servono per contenere le informazioni riguardanti il progetto presente nella *repository*, sono dei seguenti tipi.

2.4.1.1.1 Dataset

I *dataset* di allenamento, validazione e test sono salvati in artefatti distinti, ma secondo lo stesso formato all'interno di una cartella chiamata *Data* saranno presenti due cartelle distinte per le immagini e file omonimi contenenti le *label*$_{\mathbf{G}}$.
Weight and biases inoltre crea per ogni *dataset* un file di tipo **json** contenente gli indirizzi delle immagini e *label*$_{\mathbf{G}}$, salvati sul loro server, elaborato dal servizio web per fornire un'interfaccia grafica che mostra le miniature delle immagini con le rispettive *label*$_{\mathbf{G}}$ all'interno di una tabella assieme al tipo di classi presenti, l'id dell'artefatto e il nome dell'immagine per rendere l'interazione con il *dataset* estremamente fruibile, come illustrato in figura 2.11.

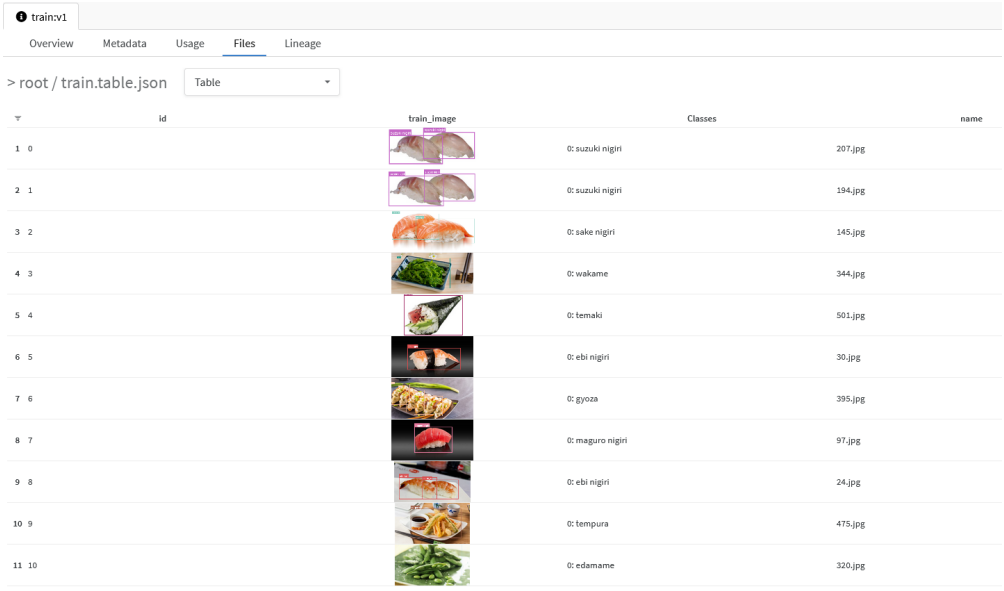


Figura 2.11: Schermata di weight and biases per la visualizzazione dei dataset.

2.4.1.1.2 Modelli per epoca

L'intervallo specificato di epoche per il salvataggio del modello crea una copia in locale e una presente su weight and biases negli artefatti *Model* sotto il nome della run a cui appartiene, questo file viene conservato con il nome di *best* e l'estensione specifica del modello in allenamento, per YOLO corrisponde a .pt.

2.4.1.1.3 Evaluation

In questo gruppo di artefatti viene salvata una cartella contenente le immagini con le predizioni effettuate dal modello, salvato nel gruppo di artefatti indicati nella sezione 2.4.1.1.2, e successivamente integrate in un file json come quello dell'artefatto di *dataset* assieme ai punteggi di confidence per ogni classe identificata, come illustrato in figura 2.12.

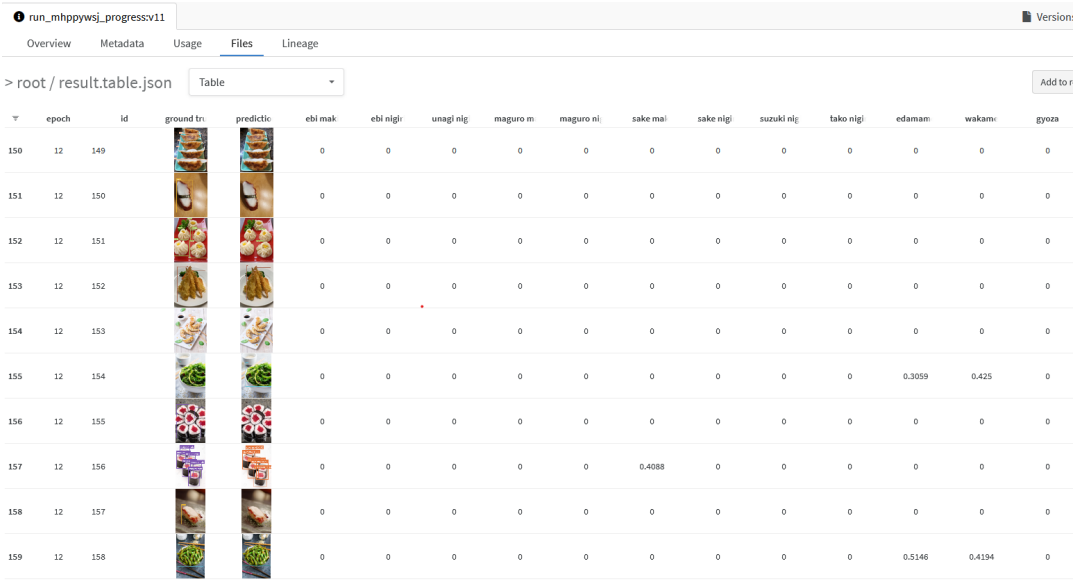


Figura 2.12: Schermata di weight and biases per la visualizzazione delle $label_G$ predette rispetto a quelle definite nel *dataset*.

2.4.1.1.4 Run

Questi artefatti sono i più importanti perchè contengono lo stato di avanzamento degli allenamenti. Collegando *dataset* e modelli salvati permette di riesumare l'allenamento in caso di fallimento del software o hardware direttamente riferendosi al percorso della *run* interessata.

2.4.1.1.5 Grafici

Successivamente all'allenamento di un modello per l'*object detection* vengono creati automaticamente i grafici di valutazione delle metriche di *precision*, *recall*, *precision recall curve* e *F1 score* assieme alla matrice di confusione (verranno spiegate nella sezione 4.1).

Capitolo 3

Applicazione delle tecnologie

In questo capitolo verranno illustrati i metodi e comandi per l'applicazione delle tecnologie adottate.

3.1 YOLO

Per allenare un modello YOLO sarà necessario creare un *dataset* etichettato mettendo in una cartella due cartelle dove la prima contiene le immagini e la seconda i file di testo .txt omonimi alle immagini corrispondenti contenenti le $label_G$ definite secondo il formato:

ID_classe X_centro Y_centro Larghezza Altezza

Le dimensioni dovranno essere in proporzione a quelle dell'immagine dunque tra 0 e 1.

3.1.1 File YAML di configurazione e percorsi dataset

Successivamente serve creare un documento *.yaml* contenente l'indirizzo delle cartelle dove si trovano i *dataset* di *training*, *validation* e *test*; la variabile intera Nc rappresenta il numero di classi; il vettore *names* è composto dalle $label_G$ delle classi nello stesso ordine specificato dai file di testo per ID_classe, da 0 a Nc, come mostra il seguente esempio di codice:

```
train : datasets/train
val: datasets/val

nc: 5

names: [ 'car', 'motorcycle', 'person', 'dog', 'cat' ]
```

3.1.2 Download e installazione repository YOLOv5

A questo punto serve scaricare la *repository* in python di YOLOv5 fornita da ultralytics su github con il comando:

```
#\> git clone https://github.com/ultralytics/yolov5.git
```

E installare le librerie necessarie all'esecuzione di YOLO:

```
#\> pip install -r requirements.txt
```

In alcuni computer non è possibile far funzionare la libreria **OpenCV-python** per problemi di dipendenze circolari con le altre librerie per cui è consigliabile installare **OpenCV-python-headless**, assicurandosi di installare una versione compatibile con l'architettura del computer, con il seguente comando:

```
#\> pip uninstall \g{\textit{OpenCV}}-python -y

#\> pip install \g{\textit{OpenCV}}-python-headless
```

3.1.3 Allenamento

Per iniziare l'allenamento bisogna indicare le caratteristiche dello stesso, specificando il modello di rete neurale, tra le varianti di YOLOv5 presenti, tramite i file di configurazione yaml:

- **Nano**: yolov5n.yaml ;
- **Small**: yolov5s.yaml ;
- **Medium**: yolov5m.yaml ;
- **Large**: yolov5l.yaml ;
- **Extra large**: yolov5x.yaml .

Per iniziare l'allenamento ci si serve del seguente comando:

```
#\> python train.py \
#       --cfg $config_yaml \
#       --data $dataset_yaml_path \
#       --epochs $Num_epochs \
#       --project $wandb_project_name \
#       --bbox_interval 1 \
#       --save-period 10 \
#       --upload_dataset
```

Dove le variabili che iniziano con **$** indicano rispettivamente:

- **config_yaml**: il percorso locale del file di configurazione della rete neurale, indicato per scegliere le tipologie di YOLOv5 da *nano* a *extra large*;
- **dataset_yaml_path**: il percorso locale del file di configurazione del *dataset*, indicato in precedenza;
- **Num_epochs**: il numero di epoche in cui la rete neurale si allenerà, raggiunta l'ennesima epoca indicata l'allenamento terminerà;
- **wandb_project_name**: nome del progetto creato su weight and biases dove verranno salvati i pesi della rete neurale come file di modello, ciò avviene ogni intervallo di epoche indicate in –save-period.

Gli ultimi 2 campi del comando precedente servono per un apprendimento preciso, dato l'intervallo minimo delle bbox, e stabile, salvando i pesi del modello ogni 10 epoche. L'appendice del comando upload_dataset viene utilizzata solo durante la prima istanza di allenamento.

Esiste la possibilità di eseguire *transfer learning*$_G$ e *fine tuning*$_G$ di un modello pre allenato tramite una nuova sessione di allenamento con il comando precedente, l'unica variante necessaria è la specifica dell'opzione:

```
# --weights $pretrained_model_path
```

La variabile pretrained_model_path indica il percorso locale del modello già allenato.

Se il processo di allenamento si interrompe è possibile riprenderlo dall'ultima epoca salvata tramite il seguente comando, notare che il percorso della run è disponibile nell'artefatto salvato su weight and biases:

```
#\> python train.py --resume wandb-artifact://{$crashed_run_path}
```

Dove la variabile crashed_run_path indica il percorso della run di allenamento fallita, si trova all'interno del progetto, indicato inizialmente come repository, nei dettagli dell'artefatto della *run* interessata.

3.1.4 Test di rilevamento con modello allenato

Infine per testare il modello allenato è possibile farlo tramite questo comando in un ambiente python:

```
# from IPython.display import Image

#\> python detect.py --weights $model_path \
#    --img 640 \
#    --conf $confidence_threshold \
#    --source $image_path

# Image(filename='runs/detect/exp/$image_name', width=600)
```

Dove le variabili indicate con $ rappresentano:

- **model_path**: percorso locale del modello allenato;
- **confidence_threshold**: valore di confidenza per cui le predizioni sono accettate, tra 0 e 1;
- **image_path**: percorso locale dell'immagine su cui effettuare la classificazione;
- **image_name**: nome dell'immagine che verrà salvata con le *label*$_G$ predette integrate.

3.2 Dataset

Come citato in precedenza il *dataset* è stato creato per la maggior parte da immagini trovate tramite query contenenti i nomi specifici del piatto di sushi in giapponese su Google Immagini, e una parte minore da repository di *dataset* e modelli di intelligenze

artificiali nello specifico weight and biases e kaggle.
Successivamente è stato necessario definire le $label_{\mathbf{G}}$ delle classi scelte in modo da poter allenare la rete neurale.
Le $label_{\mathbf{G}}$ possono essere contrassegnate con l'ausilio di un software specifico o tramite un servizio web dedicato. In questo progetto è stata adottata la seconda opzione, nello specifico è stato usato il servizio gratuito fornito da **hasty**, di seguito ne sono definite le caratteristiche e la modalità d'uso.

3.2.1 Hasty

Servizio online per l'etichettatura di label nelle immagini molto semplice e intuitivo, l'interfaccia grafica è illustrata in figura 3.1.

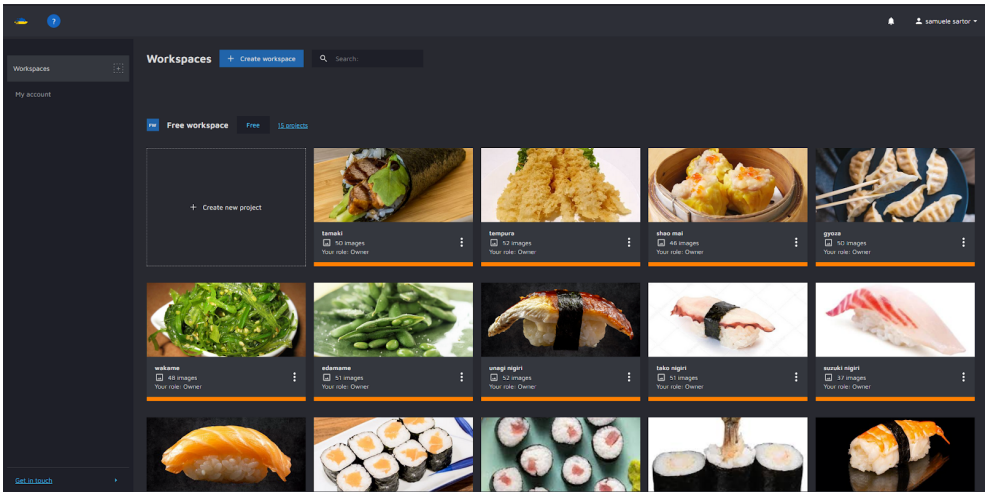


Figura 3.1: Schermata principale di Hasty dove si possono osservare tutti i progetti creati.

Dopo aver eseguito l'upload delle immagini, previa creazione del progetto dedicato, è possibile sfruttare il credito gratuito per generare centinaia di classi per le $label_{\mathbf{G}}$, uno dei pochi servizi per cui serve effettivamente il credito (30 credito iniziale, 0.04 richiesto per classe, utilizzabile per 750 classi totali).
Non appena 10 immagini vengono etichettate aggiornando lo stato di queste immagini da "*to review*" a "*in progress*", le stesse immagini verranno utilizzate per l'allenamento di una rete neurale dedicata al riconoscimento automatico delle specifiche classi al fine di velocizzare il lavoro di etichettatura. Ogni immagine successiva, etichettata come "*to review*" verrà dedicata al *fine tuning*$_{\mathbf{G}}$ dell'intelligenza artificiale di riconoscimento delle classi; nella figura 3.2 viene mostrato come è composta l'interfaccia per queste operazione, ne sono evidenziati il campo dello stato e il bottone per l'aiuto del riconoscimento delle classi.

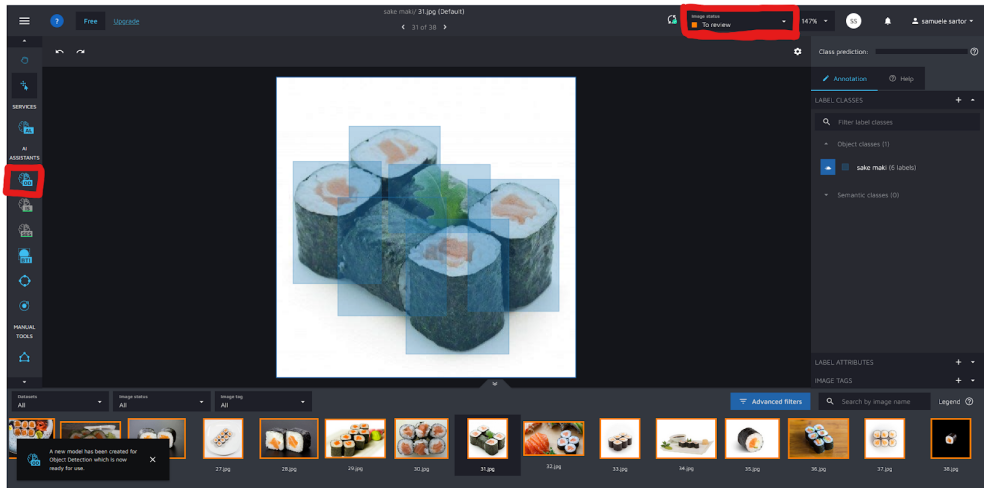


Figura 3.2: Schermata di Hasty per l'inserimento delle label.

Il servizio di riconoscimento autonomo funziona sorprendentemente bene per i *dataset* di *object detection*, ma può eventualmente essere utilizzato anche nel caso di *dataset* per *semantic segmentation* e *instance segmentation* a cui corrispondono i bottoni rappresentati sopra **SES** e **IS**. Ciò rappresenta una notevole opportunità per risparmiare enormi quantità di tempo data l'elevata difficoltà nella definizione di *label*$_G$ per questi differenti metodi di riconoscimento autonomo di oggetti.

Per l'estrazione delle **label** è disponibile all'interno del progetto l'opzione *project -> export data* o all'esterno di esso direttamente *export*, disponibile tra le opzioni che permette l'esportazione in vari formati delle *label*$_G$ riconosciute, dal formato hasty dedicato alle pure immagini con label disegnata. Il formato consigliato per YOLO è **COCO json**, uno standard utilizzato dai tempi dei primi modelli di *machine learning*$_G$ per l'*object detection* e *recognition* dedicato al *dataset* MS COCO (*Microsoft Common Objects in Context*), composto inizialmente da 164 mila immagini e ora 328 mila, utilizzato ancora oggi per una valutazione oggettiva dei nuovi modelli.

3.2.2 coco2yolo

Essendo COCO json un formato molto diffuso, esistono varie librerie per la conversione delle sue *label*$_G$ nei vari formati utilizzati dai diversi modelli. coco2yolo permette di eseguire la conversione delle *label*$_G$ nel formato YOLO.
Le *bounding box*$_G$ di COCO sono definite in pixel dai due angoli in alto a sinistra e in basso a destra (x_1, y_1, x_2, y_2); dunque per ricavare le coordinate di tutti gli angoli della *bounding box*$_G$ basta trovare gli ultimi due angoli simmetrici a quelli presenti nel formato COCO combinando x_1 con y_2 e x_2 con y_1. YOLO richiede un diverso formato basato su proporzioni quindi valori tra 0 ed 1 dove 1 è la dimensione x o y massima dell'immagine in questione e richiede il punto centrale della *bounding box*$_G$ assieme a larghezza ed altezza (x, y, w, h), salvate in file di formato .txt omonimi alle rispettive immagini.

Questa libreria funziona in modo egregio senza avere bisogno di scrivere uno *script*

dedicato per la conversione che pur essendo semplice di certo richiede del tempo per essere progettato e sviluppato.

Per iniziare la conversione serve eseguire lo *script* coco2yolo tramite python via linea di comando e specificare la posizione del file COCO json come illustra il comando qui di seguito:

```
#\> python coco2yolo -ann-path ./annotations/instances_train.json
```

Successivamente verranno richiesti gli indirizzi di due cartelle, la prima è quella contenente le immagini del *dataset* e la seconda è la cartella di destinazione per le immagini e i file di testo .txt contenenti i dati delle *bounding box*$_{\text{G}}$ relativi alle immagini omonime.
Infine viene richiesto di inserire i nomi delle classi nell'ordine in cui si vuole che vengano collegate agli indici da 0 a N dove N equivale al numero delle classi meno uno.

3.2.3 Dataset finale

Il *dataset* ultimato, illustrato in figura 3.3, rappresenta le seguenti 15 classi definite con i nomi originali giapponesi dei piatti, la loro traduzione o interpretazione è scritta successivamente ad ognuna:

- **Tamaki**: cono di alghe contenente sushi e verdure;
- **Tempura**: pesce impanato fritto;
- **Shao mai**: ravioli aperti di carne e piselli cotti al vapore;
- **Gyoza**: ravioli di pesce o frutti di mare e verdure cotti al vapore;
- **Wakame**: insalata di alghe omonime;
- **Edamame**: baccello di legumi giapponese cotto al vapore;
- **Unagi nigiri**: riso con anguilla;
- **Tako nigiri**: riso con polpo;
- **Suzuki nigiri**: riso con branzino;
- **Sake nigiri**: riso con salmone;
- **Maguro nigiri**: riso con tonno;
- **Ebi nigiri**: riso con gambero;
- **Sake maki**: roll di riso avvolto da un alga contenente salmone;
- **Maguro maki**: roll di riso avvolto da un alga contenente tonno;
- **Ebi maki**: roll di riso avvolto da un alga contenente gambero.

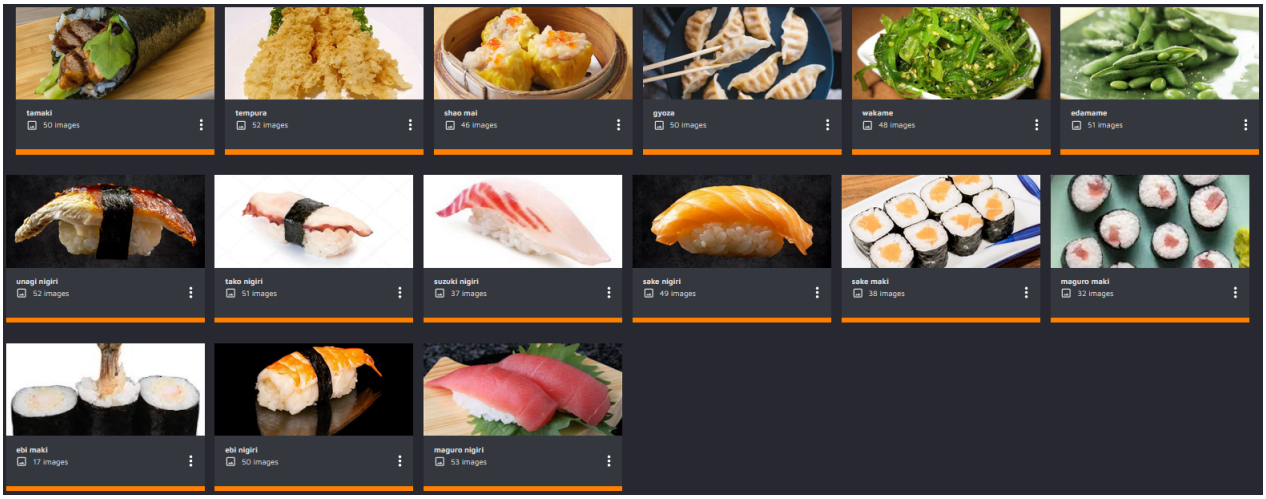


Figura 3.3: Illustrazione del *dataset* completo creato.

3.3 Sviluppo della Mobile app

3.3.1 Funzionalità e dettagli

L'applicazione è stata sviluppata in **Java** per l'ambiente Android$_G$ compatibile a partire dalla versione 6.0 .

La schermata iniziale dell'applicazione illustrata in figura 3.4 è composta da un menù a tendina che permette la selezione dei tre modelli di YOLOv5 allenati: *nano*, *small* e *medium*. Scelta la versione che si vuole testare, bisogna premere il bottone in fondo alla schermata per iniziare a catturare immagini con la fotocamera del proprio dispositivo Android$_G$.

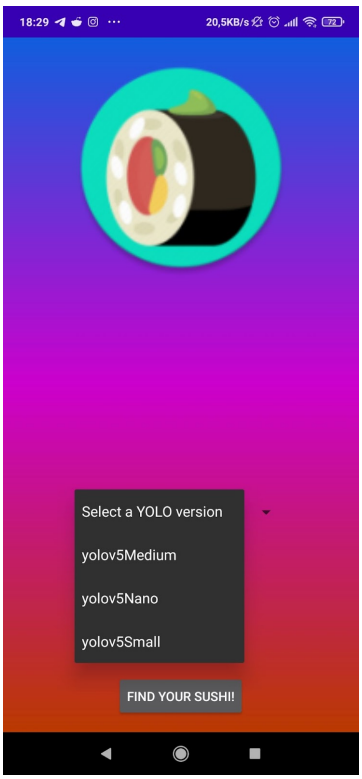


Figura 3.4: Schermata iniziale dell'applicazione creata.

Nel momento in cui si portano a termine i passaggi sopra definiti si arriva alla schermata chiamata **CameraActivity** dove sono visualizzati, a schermo intero, i frame catturati dalla videocamera del cellulare. È possibile a questo punto scegliere tra due opzioni: YOLO e GAMMA.
Il primo elabora tramite il modello selezionato in precedenza i frame catturati mentre il secondo corregge la luminosità per una migliore predizione del modello.

3.3.2 CameraActivity

Per permettere l'utilizzo del modello YOLO allenato, ho utilizzato la libreria *open source*$_{\mathbf{G}}$ di *OpenCV*$_{\mathbf{G}}$ la quale rende disponibile per ogni versione rilasciata la libreria Android$_{\mathbf{G}}$ SDK$_{\mathbf{G}}$ in **Java** precompilata in **C++** dalla versione originale di *OpenCV*$_{\mathbf{G}}$. Per l'attività e la lettura dei frame dalla fotocamera ho implementato la classe **CVCameraViewListener2** la quale estende la classe **CameraActivity** nativa della libreria di Java per Android$_{\mathbf{G}}$.

L'*output* viene usato per disegnare sul frame il testo delle *label*$_{\mathbf{G}}$ e il rettangolo delle *bounding box*$_{\mathbf{G}}$ rispettivamente tramite le funzioni **putText** e **rectangle** della classe **Imgproc** integrata nella libreria di *OpenCV*$_{\mathbf{G}}$. Il risultato è illustrato in figura 3.5.

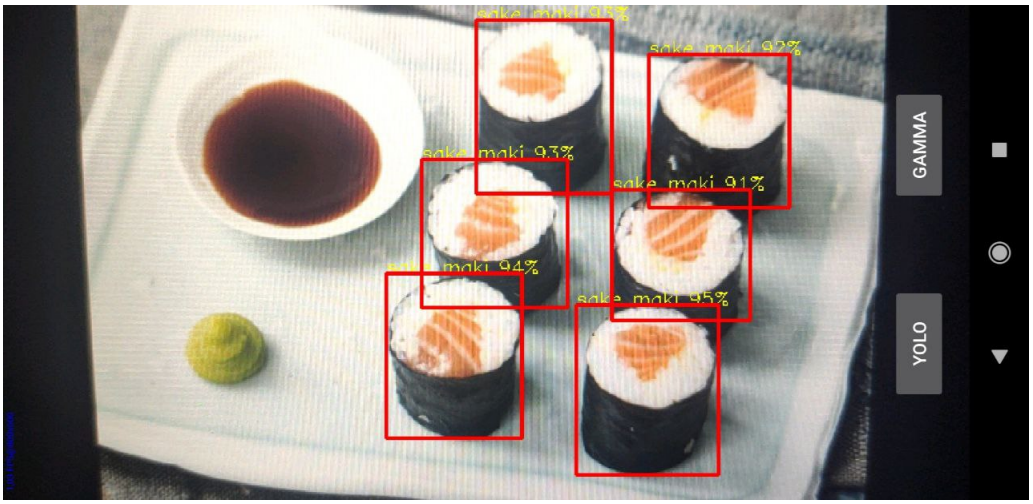


Figura 3.5: Schermata dell'applicazione creata che riconosce dei roll di salmone.

3.3.2.1 GAMMA

Per favorire la classificazione dei tipi di sushi ho utilizzato una tecnica di trasformazione della matrice **RGB** dei frame in formato **HSV** per calcolare la media della luminosità di tutti i pixel così da utilizzarla come valore indicativo per alterare la luminosità di ogni pixel, arrivando ad aumentare la luminosità dell'intero frame. Questo metodo non è il migliore per l'aumento della luminosità, dato che un'immagine maggiormente scura diventa solo più scura, ma è utile quando è presente una fonte di illuminazione, così il resto dell'immagine si adegua ad essa creando un formato più omogeneo e quindi migliore per la predizione della rete neurale.

3.3.2.2 YOLO

Per la valutazione della rete neurale ho trasformato il file **.pt** in **.onnx**, così che fosse compatibile con la libreria di *OpenCV*$_{\mathbf{G}}$, tramite il comando invocabile nella cartella di YOLOv5 utilizzando lo script **export.py**:

```
#\> python export.py --weights $model_path --include onnx
```

Dove la variabile model_path deve indicare il percorso locale del modello da convertire. Il modello è stato salvato nella cartella *assets* del progetto Android$_{\mathbf{G}}$ e letto tramite un **BufferedInputStream** in un array di byte, trasformato in **MatOfByte**, una matrice di byte classe della libreria di *OpenCV*$_{\mathbf{G}}$, per poi essere fornita come *input* alla funzione **Dnn.readNetFromONNX()** per avere in ritorno l'oggetto di tipo **Net** grazie al quale poter eseguire il *forward* degli *input* equivalenti ai frame catturati sotto forma di **MatOfByte**.

L'*output* in formato di matrice di 2 dimensioni $25200 * 20$ corrisponde per il primo indice al numero massimo di *bounding box*$_{\mathbf{G}}$ rilevabili, e per il secondo indice ai valori di confidenza delle 15 classi per cui il modello è stato allenato, i valori x e y del centro della *bounding box*$_{\mathbf{G}}$, la larghezza e l'altezza della *bounding box*$_{\mathbf{G}}$.

Capitolo 4

Risultati ottenuti

In questo capitolo verranno analizzati i risultati ottenuti riguardo al modello di intelligenza artificiale allenato.
I principali vantaggi comportano la possibilità di utilizzare tecniche di deep learning$_G$ open source$_G$ per l' object detection grazie a YOLO, Weight and biases fornisce una repository gratuita, Hasty fornisce credito gratuito per 750 classi etichettabili, Gradient fornisce la potenza di calcolo sufficiente per allenare modelli complessi in poche ore e la libreria di OpenCV$_G$ implementata nell'applicazione mobile è sufficientemente veloce per processare un frame in un secondo.
Mentre gli svantaggi principali sono la precisione carente causata da un datset limitato, il rilevamento in real time che non può essere raggiunto data l'assenza di elaboratori grafici con driver compatibili per le intelligenze artificiali nei dispositivi portatili e Gradient è in grado di allenare i modelli di YOLOv5 da nano fino a medium non supportando l'allenamento del modello *large*.

4.1 Metriche dell' *object detection*

Verranno illustrate brevemente di seguito le metriche utilizzate nel processo di allenamento delle reti neurali, prima di discutere delle prestazioni del modello allenato.

4.1.1 Matrice di confusione

Un classificatore binario, allenato solo per due classi A e B, può dare solo quattro risultati:

- ***True positive***: corretta classificazione di un membro della classe A (la classe positiva) come A;
- ***True negative***: corretta classificazione di un membro della classe B (la classe negativa) come B;
- ***False positive***: incorretta classificazione di un membro della classe B come A;
- ***False negative***: incorretta classificazione di un membro della classe A come B.

Notare che le classi A e B servono solo per facilitare l'illustrazione, sono due nomi di classe che possono essere invertiti e cambiati con qualsiasi altro nome, l'unica costante

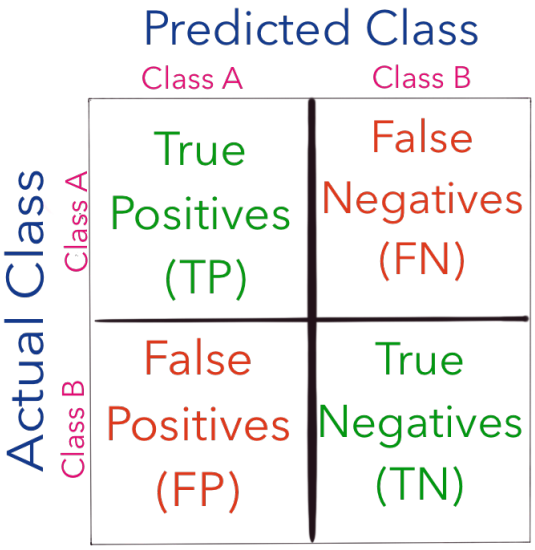


Figura 4.1: Matrice di confusione base.

sta nel fatto che A rappresenta la classe osservata e B qualsiasi altra classe negativa rispetto alla corretta classificazione di A.

Avendo compreso il principio della matrice di confusione si è pronti per capire come siano definite le metriche.

4.1.2 Precision

La *precision* è il rapporto tra le classi correttamente riconosciute su tutte le classi riconosciute come positive; in altre parole rappresenta la percentuale di tutte le predizioni corrette della classe positiva.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

La sua utilità viene definita dalla capacità di comprendere quanto ci si può fidare del classificatore quando ti dice che una predizione appartiene alla classe corretta.

4.1.3 Recall

La *recall*, invece, è il rapporto tra le classi riconosciute come positive su tutte le classi veramente positive. Definisce la percentuale delle classi riconosciute come positive che effettivamente sono positive.

$$Precision = \frac{TruePositives}{TruePositives + FalseNegatives}$$

La sua utilità sta nel definire quanto ci si può fidare del classificatore per trovare tutte le classi corrette.

4.1.4 F1 score

Per comprendere come viene convertito il rapporto tra *precision* e *recall* in un unico valore, in modo da automatizzare l'apprendimento di una rete neurale, va definito prima di tutto come queste due metriche operano nel momento in cui si valorizza esclusivamente una rispetto all'altra:

- **Massima *precision***: il modello sarà molto preciso nel riconoscimento delle classi, ma riconoscerà solo quelle che valuta come sicuramente positive, avendo difficoltà nel riconoscere molte altre classi positive;
- **Massima *recall***: il modello classificherà quasi tutte le classi positive, ma riconoscerà come tali anche classi negative o rumore.

Precision e *recall* se valorizzate al massimo vanno ad escludersi mutualmente per questo vengono rappresentate in un grafo chiamato *precision-recall curve* in modo da osservare come il modello si comporta al variare dei valori delle due. Si possono comprendere le prestazioni di precisione del modello riconoscendo quanto la curva del grafo si elevi sopra la diagonale definita unendo i punti di *precision* e *recall* massime, l'obiettivo sarebbe raggiungere un equilibrio ideale di *precision* e *recall* massimizzate, ma non massime.

La necessità di ottenere un valore unico per la valutazione dei modelli ha portato alla creazione di un'altra metrica chiamata *F1 score* la quale rappresenta il bilanciamento tra *precision* e *recall*. Dunque se *F1* raggiunge un valore alto allora sia *precision* che *recall* saranno definite da un valore alto e viceversa per valori bassi.
Definita come segue:

$$F1 = 2\frac{Precision * Recall}{Precision + Recall}$$

F1 score risulta utile per rappresentare la valutazione delle performance della rete neurale, ma non praticabile nella fase di addestramento della rete neurale, per cui è stata creata una metrica che valuta in modo omogeneo e costante la progressione di *precision* e *recall*: l' *average precision*.

4.1.5 Mean Average Precision (mAP)

Anche l'*average precision* (**AP**) riassume la curva di *precision recall* in un unico valore rappresentate la media della precisione del modello. L'AP viene calcolata come illustrato nella seguente equazione usando un ciclo iterato per il numero di **IoU** trovati (definito precedentemente nella sezione 2.1.3.1) che somma tutti i rapporti di *precision recall* calcolando la differenza tra la *recall* corrente e quella successiva per poi moltiplicare tale valore per la *precision* corrente. In altre parole l'AP è una somma pesata delle *precision* di ogni *bounding box*$_{\mathbf{G}}$, composta da IoU, corrispondente dove il peso viene definito dalla *recall* crescente.

$$AP = \sum_{k=0}^{n-1} [Recalls_k - Recalls_{k+1}] * Precisions_k$$
$$Recalls_n = 0, Precisions_n = 1$$
$$n = Numero\ di\ IoU\ bounding\ boxes_{\mathbf{G}}\ totali$$

Infine viene calcolata la media delle AP di ogni classe in modo da ottenere la metrica che rappresenta in un unico valore la precisione di predizione del modello per l'intero *dataset*, **mAP** (*mean average precision*) definita dalla seguente formula:

$$mAP = \frac{1}{n}\sum_{k=1}^{n} AP_k$$
$$AP_k = \text{L'AP della classe k-esima}$$
$$n = \textit{Numero di classi}$$

4.1.6 Metriche per la valutazione delle *bounding boxes*$_G$ predette

Infine per comprendere il processo di perfezionamento nella predizione delle *bounding boxes*$_G$ delle classi di oggetti nel campo dell'*object detection* è necessario comprendere queste tre metriche principali:

- ***bounding box*$_G$ regression loss**: chiamato anche box_loss in YOLO viene calcolato tramite l'errore quadratico medio tra le *bounding box*$_G$ predette e quelle effettive;
- **log loss**: chiamato obj_loss in YOLO rappresenta l'errore nella classificazione della classe corretta identificando altre classi nella stessa area definite come negative, questo valore viene calcolato tramite la *binary cross entropy*.
- **cls_loss**, simile a log loss definisce l'errore nella classificazione della classe corretta con tutte le altre classi identificandole separatamente, calcolato tramite *cross entropy*.

4.2 Risultati ottenuti

I test di allenamento, condotti per 200 epoche utilizzando una combinazione di tutte le tecniche e tecnologie fornite dalle *bag of Freebies* e *bag of specials*, hanno evidenziato le differenze di *performance* dei tre modelli di YOLOv5 allenati (*nano*, *small* e *medium*) secondo i punteggi raggiunti per le metriche di valutazione. Dalla figura 4.2 si può osservare come le metriche valorizzanti gli errori di posizionamento e classificazione, durante la predizione delle *label*$_G$ per il *dataset* di validazione, abbiano portato alla convergenza dell'errore prodotto dai tre modelli verso le ultime epoche anche se è evidente che *small* producesse un errore minore durante l'intero allenamento. Per comprendere perchè l'allenamento dei tre modelli abbia portato a questi risultati è necessario studiare la figura 4.3 dove sono rappresentate le prestazioni dei modelli secondo la metrica di **mAP**. Questa metrica mostra come *nano* abbia ottenuto i risultati peggiori conducendo l'allenamento a discendere verso una precisione minore durante molteplici epoche raggiungendo il punteggio di 0.73 mAP; questo comportamento denota carenza di percettroni e connesioni sinaptiche per la definizione di hyper parametri polivalenti per i vari tipi di classi forzando la rete neurale a "dimenticare" i *pattern* di alcune classi per generalizzarne altre. Si nota che *medium* ha raggiunto il punteggio di 0.82 mAP ottenuto il comportamento contrario, cioè ha avuto più picchi di risalita seguiti da discese notevoli che denota la presenza di troppi percettroni e connessioni sinaptiche i quali hanno causato la definizione di hyper parametri estremamente complessi per il riconoscimento di specifiche classi causando nelle epoche successive la generalizzazione degli hyper parametri per il riconoscimento di altre classi perdendo la precisione raggiunta per le specifiche classi apprese in precedenza. Invece *small* ha condotto una salita stabile nelle prime 50 epoche per poi aggiustare i propri hyper parametri e divenire stabile dopo l'epoca 120.

Questo studio ha evidenziato come il modello *small* sia il più adatto per il numero di $feature_{\mathbf{G}}$ maggiormente compatibile con il *dataset* creato, invece *nano* subisce $underfitting_{\mathbf{G}}$ causato dalla carenza di percettroni e connessioni sinaptiche, mentre *medium* presenta $overfitting_{\mathbf{G}}$ per l'eccessiva quantità di $feature_{\mathbf{G}}$.

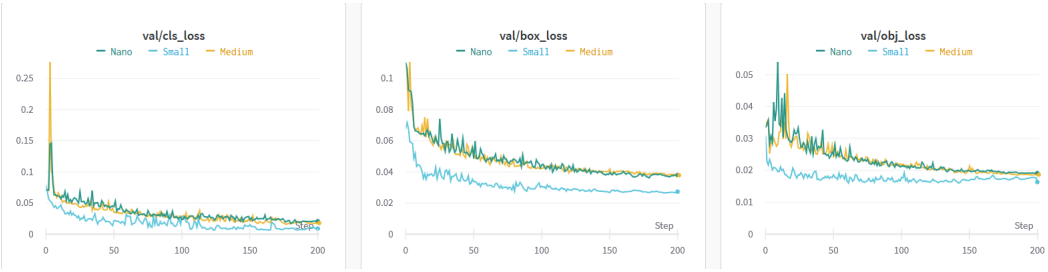


Figura 4.2: Grafici di valutazione dei modelli allenati, nano, medium e small, secondo le metriche di precisione per le $bounding\ boxes_{\mathbf{G}}$ definite in 4.1.6.

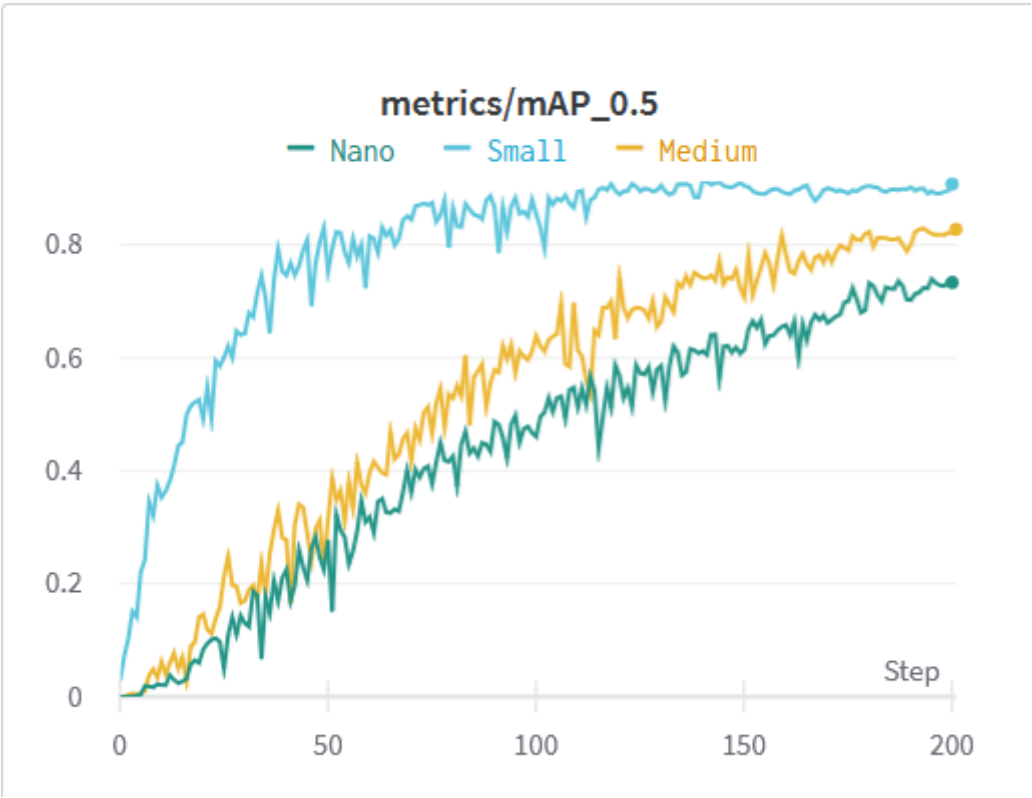


Figura 4.3: Grafico di valutazione della mAP per i modelli allenati nano, medium e small.

Probabilmente con un *dataset* più vasto, la versione medium potrebbe raggiungere prestazioni migliori e per il *dataset* completo di ogni tipologia di sushi potrebbe essere ottimale addirittura il modello *large*.

Ho creato un PoC (*proof of concept*) per dimostrare la velocità di elaborazione della rete neurale processando ogni frame catturato dalla camera del cellulare quando viene premuto il bottone YOLO presente nell'interfaccia grafica della seconda schermata dell'applicazione Android$_G$, CameraActivity. Inoltre in questa schermata in alto a sinistra vengono mostrati i frame per secondo, come illustrato in figura 4.4 da cui si può notare come vengano ridotti drasticamente da 30, per la sola cattura dei frame, a 2-5 processandoli secondo i diversi modelli di YOLO. Tale riduzione avviene anche per la correzione della luminosità, premendo il bottone GAMMA, dato che deve processare l'intera matrice dei pixel di ogni frame due volte. Se la correzione della luminosità viene eseguita assieme all'elaborazione della rete neurale la frequenza dei frame caleranno arrivando attorno ad 1 per secondo. Questo dimostra come senza un servizio di *cloud computing* sia pressoché impossibile creare un'applicazione che esegua la rilevazione in tempo reale, ma può essere catturata un'immagine singola per volta dal dispositivo Android$_G$ per eseguire in pochi secondi la rilevazione nel frame catturato. La libreria di *OpenCV*$_G$ permetterebbe inoltre di catturare più frame per interpolarli in un'immagine più dettagliata e con risoluzione maggiore. Ciò potrebbe aiutare per acquisire immagini meno sfocate e mosse in modo che la rete neurale possa riconoscere i piatti di sushi con precisione migliore.

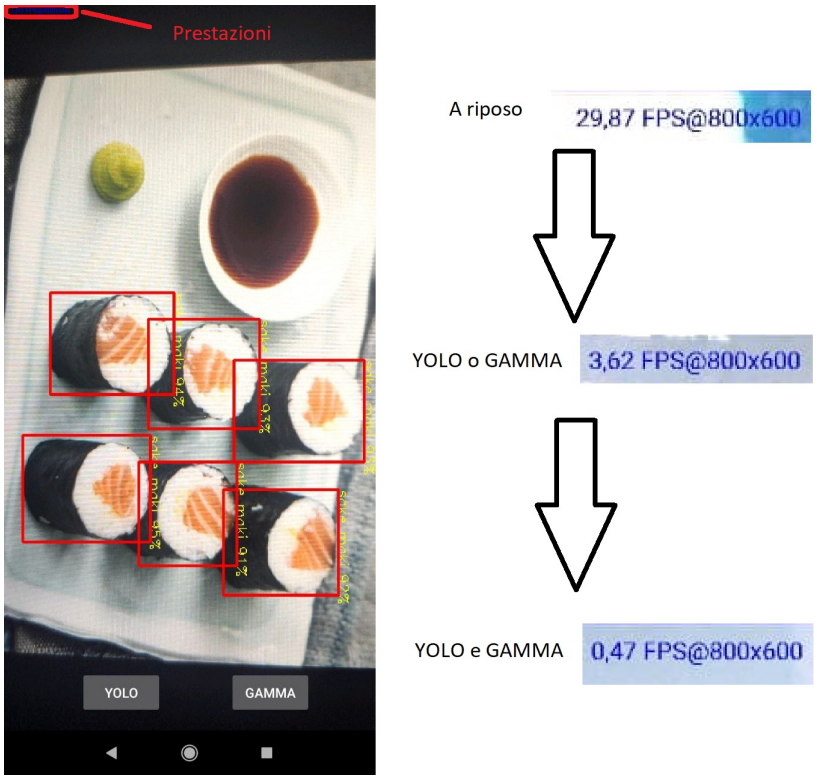


Figura 4.4: Visualizzazione dei frame per secondo e risoluzione delle immagini catturate per gli stati possibili, a riposo, con una funzionalità attiva e con entrambe attive.

La capacità di riconoscimento di sushi è stata riscontrata ottimale impostando i parametri di **confidenza a 0.9** e **IoU a 0.55** per cui la rete neurale raggiunge le seguenti prestazioni: con immagini scattate vicine al piatto e per una quantità inferiore a 7 pezzi di sushi è pressoché del 90-95% nella fase di validazione, come illustrato nella figura 4.3, e si avvicina al 60-70% nel *live testing*, ma nel momento in cui si provano a rilevare più pezzi di sushi nella stessa immagine o quest'ultima viene scattata troppo lontana dal piatto, la precisione diminuisce notevolmente. Ciò è dovuto dalla scarsità di immagini nel *dataset* nonostante sia composto da 50 per tipo, per un totale superiore a 700. Sarà necessario introdurre tipologie più miste e con molti pezzi per immagine. L'efficacia viene dimostrata dal riconoscimento della classe ***shao mai***, il raviolo orientale, il quale è presente maggiormente in gruppi superiori a 8 ravioli per immagine. Per questo motivo l'acquisizione dei suoi dettagli, essendo i singoli ravioli sia grandi che piccoli, avviene in modo molto più scrupoloso, arricchendo lo strato *neck* della rete neurale di svariati *hyper parametri* per il successivo riconoscimento di questa classe. Il modello allenato è in grado di riconoscere i singoli *shao mai* anche in immagini che li ritraggono da distanti, dunque sono rappresentati a bassa risoluzione, per lo stesso principio di allenamento adottato per questa classe è possibile raggiungere la stessa precisione con qualsiasi tipo di sushi e piatto orientale.

In figura 4.5 viene illustrato il grafo di *precision recall curve* del modello YOLO *small* per tutte le classi definite nel *dataset*.

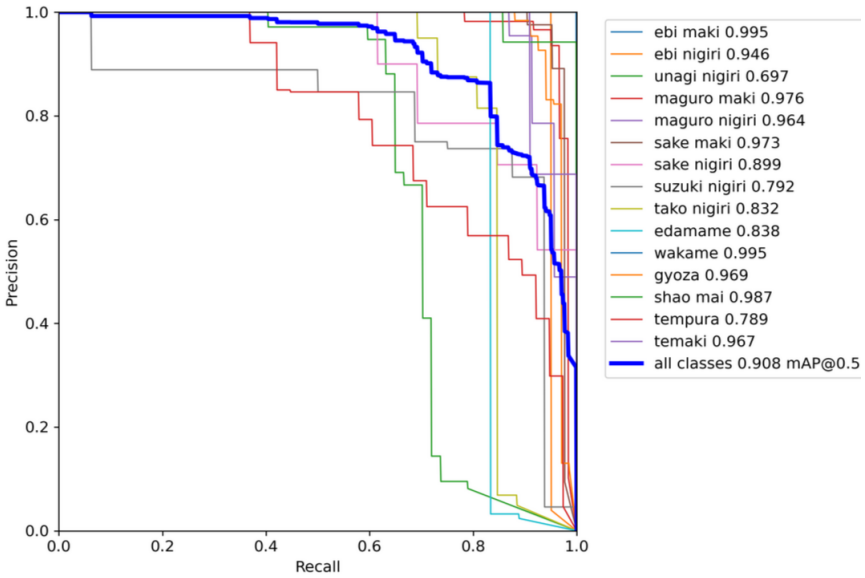


Figura 4.5: Grafico di valutazione *precision recall curve* del modello *small* allenato per ogni classe.

E il grafo della curva dell'*F1 score* rispetto alla confidenza, per una lettura facilitata, è illustrato in figura 4.6.

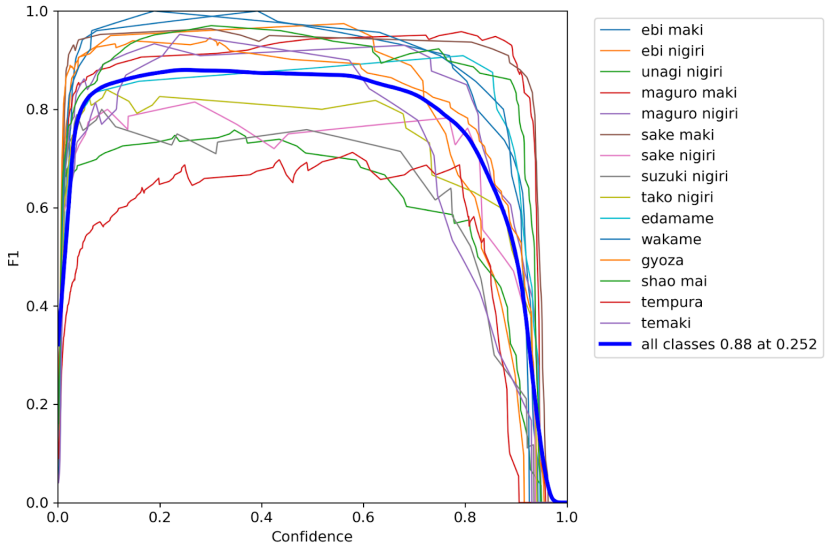


Figura 4.6: Grafico di valutazione *F1 score* del modello *small* allenato per ogni classe.

L'abilità del modello allenato nel discernere le differenti classi viene definita dalla matrice di confusione, tale matrice rappresentante le percentuali di "confusione" per cui la rete neurale ha la probabilità di confondere una classe rispetto all'altra del *dataset* creato. Per il modello YOLO *small* allenato è rappresentata in figura 4.7.

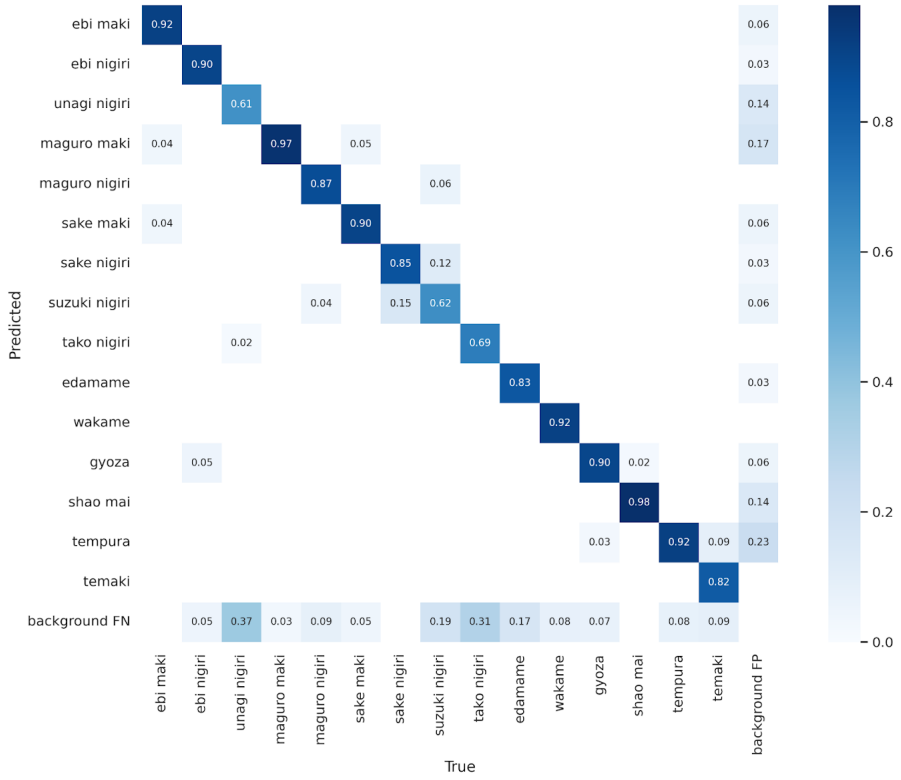


Figura 4.7: Matrice di confusione per tutte le classi predicibili dal modello *small* allenato.

La matrice di confusione dimostra come l'allenamento abbia condotto ad una sensibilizzazione sufficiente per ridurre al minimo la rilevazione di classi falsamente positive. Si è raggiunto un massimo del 15% di probabilità nel confondere tipi di sushi estremamente simili come i roll di salmone e tonno i quali differiscono solo per la tonalità del colore del pesce al loro interno.

Capitolo 5

Conclusioni

5.1 Obiettivi del prodotto finale realizzato

Lo scopo di questo progetto di stage è quello di verificare e valutare l'applicazione di tecniche di *computer vision*$_{G}$, supportate da tecniche di *machine learning*$_{G}$, nell'utilizzo di strumenti di *object detection* e nell'utilizzo di algoritmi di *machine learning*$_{G}$ per il riconoscimento di immagini. In particolare, si voleva realizzare un modello per il riconoscimento di piatti di sushi, mostrando performance e limiti di questa architettura. Purtroppo questi modelli "intelligenti" sono pre allenati, quindi necessitano di una quantità di dati, in fase di training. Non sempre, però, i database presenti in rete sono sufficienti per tale scopo.

Lo stage è stato concluso con la realizzazione di tutti gli obiettivi e con la realizzazione di un PoC (*Proof of Concept*) per lo studio delle performance massime raggiungibili dai dispositivi portatili Android$_{G}$ tramite il processamento delle immagini catturate dalla telecamera in real time al fine di definire se sia possibile processare i frame catturati direttamente offline tramite i dispositivi Android$_{G}$ o se sia necessaria l'implementazione di servizi di *cloud computing* per fornire un servizio performante.

5.2 Obiettivi raggiunti

Nella tabella 5.1 sono illustrati i requisiti, indicati nella sezione 1.3, e il loro stato d'avanzamento a progetto concluso.

Requisito	Titolo	Stato d'avanzamento
O01	Autoformazione	Completato
O02	Rispetto delle tempistiche	Completato.
O03	Implementazione *computer vision*$_{\mathbf{G}}$	Completato.
O04	Implementazione *deep learning*$_{\mathbf{G}}$	Completato.
D01	Completamento totale del progetto	Completato.
F01	Integrazione su piattaforma web o app	Parzialmente completato, a causa del tempo richiesto è stato possibile creare solo un PoC per valutare le prestazioni su dispositivi mobile, senza l'ausilio di una piattaforma web.

Tabella 5.1: Tabella del tracciamento sull'avanzamento dei requisiti.

5.3 Conoscenze e abilità acquisite

Le conoscenze con cui ho iniziato questo progetto erano pregresse negli anni di studio universitari e precedenti. Conoscevo la programmazione delle applicazioni Android$_{\mathbf{G}}$ nel dettaglio avendo intrapreso parecchie esperienze di questo genere anche per progetti didattici. Per quanto riguarda il campo delle reti neurali e *object detection*, avevo condotto degli esperimenti innescati dalla mia curiosità e seguito un paio di corsi per comprendere come la psicologia avesse ispirato la nascita delle reti neurali e il funzionamento delle più basilari applicazioni di *machine learning*$_{\mathbf{G}}$. Per questo ho studiato il funzionamento delle reti neurali dello stato dell'arte dell'*object detection* durante lo sviluppo di questo progetto riuscendo a raggiungere una consapevolezza strutturata grazie ai metodi di ricerca e studio appresi negli anni universitari. Alla fine dell'esperienza di stage ho potuto avere anche un'idea definita sul funzionamento dell'architettura di *deep learning*$_{\mathbf{G}}$ che YOLO incorpora.

Appendice A

Glossario

A

Anchor box
Tecnica di *object detection* per facilitare la predizione di oggetti multipli in un'unica immagine. Le *anchor boxes* si specializzano nel riconoscimento delle specifiche classi definite nel *dataset* e forniscono potenziali *buonding boxes* per tutte le occorrenze rilevate, successivamente queste buonding boxes verranno verificate per l'appartenenza alla specifica classe e solo quelle con confidenza sufficiente comporranno l'output della rete neurale. Questo metodo permette di risparmiare tempo nella fase di rilevazione.

Android
Android è un sistema operativo basato sul kernel Linux per sistemi embedded, principalmente viene installato in dispositivi portatili, ma può essere trovato in televisioni, smart cars e altri dispositivi.

B

Bias
Il *bias* algoritmico viene utilizzato nelle reti neurali per fornire un valore di *default* ad ogni strato, equivale ad un percettrone anche se il suo valore è fissato e non deciso dall'*output* dello strato precedente. Un *bias* troppo elevato porta la rete neurale ad effettuare le rilevazioni delle classi solo per *input* estremamente simili a quelli presenti nel *dataset*, mentre un *bias* troppo basso conduce la rete neurale a generalizzare le predizioni sbagliando le classi predette o rilevandone dove non dovrebbero esserci.

Bounding box
La bounding box definisce il riquadro di rilevamento di un oggetto.

Tale formato viene utilizzato dalle reti neurali di *object detection*, si compone delle dimensioni e coordinate del riquadro stesso, principalmente le coordinate definiscono l'angolo in alto a sinistra o destra della bounding box, ma in casi come YOLO definiscono il centro.

C

Computer vision
Computer vision è un settore interdisciplinario che ricerca metodi per l'estrazione di informazioni da immagini o video da parte di un computer o un sistema automatizzato come un'intelligenza artificiale.

Convoluzione
Nel campo delle intelligenze artificiali si intende con convoluzione l'applicazione di connessioni sinaptiche ricorrenti in una rete neurale da uno strato successivo al precedente o all'interno di uno stesso strato.

CUDA
CUDA (Compute Unified Device Architecture) si compone dell'architettura e dei driver per l'utilizzo parallelo degli elaboratori grafici NVIDIA a partire dal 2007, permettendo ai programmatori di scrivere applicazioni capaci di eseguire il calcolo parallelo delle schede video utile per lo sviluppo di videogiochi, l'elaborazione di grandi quantità di dati, l'allenamento e l'utilizzo di reti neurali.

D

Deep learning
Il *deep learning* è una specializzazione del *machine learning* che raggiunge maggiori prestazioni e flessibilità tramite la rappresentazione dell'ambiente osservato come una gerarchia annidata di concetti. Ogni concetto viene definito in relazione a concetti più semplici e rappresentazioni astratte elaborate in concetti meno astratti. Viene utilizzato nell'*object detection* per l'estrazione di paramentri al fine di perfezionare la predizione delle singole classi di oggetti.

Dropout
Il *dropout* rappresenta nel settore delle reti neurali il fenomeno di annullamento della propagazione dell'informazione o l'aggiunta di rumore destabilizzante, principalmente viene causato dalle imperfezioni dei modelli matematici utilizzati come funzioni di attivazione nei percettroni.

F

Feature
Le *features* nelle reti neurali identificano le informazioni astratte passate tra gli strati di neuroni come caratteristiche rilevanti corrispondenti a quelle apprese in fase di allenamento riguardanti le classi da riconoscere.

Fine tuning
Il *fine tuning* è il processo finale dell'allenamento di un modello di intelligenza artificiale, consiste in una nuova fase di allenamento con informazioni non presenti nel *dataset* dell'allenamento originale in modo da valutare le prestazioni del modello e perfezionare eventuali imprecisioni delle predezioni.

Fisher vector
Il metodo chiamato *fisher vector* esegue l'estrazione di specifiche *feature* da un'immagine, estrapolandone i vettori dei valori di deviazione media e covarianza di deviazione, estratte da ogni componente calcolato tramite *Gaussian Mixture Model* (GMM) e altre *feature* estratte dal descrittore di *feature* locale. Utilizzato principalmente per l'estrazione dei dettagli della forma di oggetti, alcune variazioni permettono l'estrazione di altre caratteristiche come i colori.

K

K-mean clustering
L'algoritmo di *k-mean clustering* suddivide il *dataset* di allenamento in k gruppi, o cluster, dove ogni elemento è presente esclusivamente in un unico gruppo. Successivamente vengono calcolati i valori di predizione di ogni elemento per spostarli nel gruppo che contiene elementi simili. Alla fine del processo il *dataset* viene ordinato in uno spazio multidimensionale, dove le dimensioni sono dagli hyper parametri dello strato *neck* della rete neurale, per una maggiore comprensione per gli umani delle caratteristiche delle classi e la generazione di metriche di valutazione delle nuove predizioni per le reti neurali.

L

Label
La *label* contiene una *bounding box*, dunque le dimensioni e coordinate del riquadro in cui è contenuto un oggetto all'interno di un'immagine, assieme

all'etichetta della classe corrispondente.

M

Machine learning
Machine learning è un campo dell'intelligenza artificiale che si occupa della creazione di sistemi per permettere l'apprendimento simulato nei calcolatori tramite la ripetizione di simulazioni basate su dati raccolti riguardanti il comportamento finale atteso o gli elementi di un ambiente da riconoscere.

O

Open source
Open source indica un sistema di sviluppo software decentralizzato dove i file sorgente sono condivisi gratuitamente.

OpenCV
OpenCV è una libreria software multipiattaforma nel campo della *computer vision*. Nata come libreria *open source* in un centro di ricerca russo della Intel.

Overfitting
Nell'ambito delle intelligenze artificiali l'*overfitting* si riferisce al caso in cui i risultati predetti da una rete neurale si specializzano troppo verso specifici casi presenti nel *dataset* di allenamento risultando in una rete neurale che non riesce a predire tutti i casi di allenamento e quindi neanche i casi reali.

R

Residual block
Il metodo *residual block* si basa sulla creazione di reti neurali in cui l'informazione passa da uno strato di neuroni ad un altro successivo saltando degli strati con una connessione sinaptica diretta senza passare per dei percettroni intermedi. Ciò permette di evitare la degradazione dell'informazione in reti neurali che trattano concetti estremamente complessi in cui si rischia di perdere informazioni importanti nell'elaborazione tra gli strati di neuroni.

S

SDK
Software Development Kit (SDK) sono delle librerie dedicate per lo sviluppo di applicazioni mobile contenenti inoltre documentazione, API, IDE, un debugger e alcuni esempi di codice per l'implementazione.

T

Tensore
I tensori sono strutture di dati utilizzate nell'algebra lineare per velocizzare le operazioni di vettori e matrici, hanno dato l'idea per la fondazione di ***TensorFlow*** che ha creato le prime reti neurali basate sul principio dei tensori per sfruttare al massimo le potenzialità del calcolo parallelo delle schede video.

Transfer learning
Il *transfer learning* corrisponde al processo della trasmissione delle informazioni contenute in una rete neurale già allenata perchè riesca a riconoscere classi simili rispetto a quelle con cui è stata allenata, ciò avviene tramite una nuova fase di allenamento con un *dataset* contenente le classi interessate. Questo metodo può essere utilizzato ad esempio per la creazione di modelli che riconoscono i diversi tipi di veicoli a quattro ruote i quali possono essere basati su una rete neurale che riconosce solo automobili.

U

Underfitting
Nell'ambito delle intelligenze artificiali l'*underfitting* riguarda i casi in cui la rete neurale interessata generalizza troppo tramite le sue predizioni divenendo meno precisa.

Bibliografia

Repository:

Modello di intelligenza artificiale (weight and biases): https://wandb.ai/santoss/sushi/
Applicazione Android (Github): github.com/ApocalypseTH/SyncLab_Sushi_Android_App

Tecnologie e servizi:

YOLOv1: arxiv.org/pdf/1506.02640.pdf
YOLOv2: arxiv.org/pdf/1612.08242v1.pdf
YOLOv3: arxiv.org/pdf/1804.02767v1.pdf
YOLOv4: arxiv.org/pdf/2004.10934v1.pdf
YOLOv5: github.com/ultralytics/yolov5.git
Kaggle: www.kaggle.com
Free images: www.freeimages.com
Roboflow: roboflow.com
Comet: www.comet.com
Valohai: valohai.com
Neptune: neptune.ai
Weight and biases: wandb.ai
Google colaboratory: colab.research.google.com
Paperspace gradient: learn.paperspace.com/product/gradient
Hasty: app.hasty.ai
Coco2yolo: github.com/tw-yshuang/coco2yolo
OpenCV: opencv.org