Università degli Studi di Padova

Dipartimento di Scienze Statistiche

Corso di Laurea Triennale in

Statistica per le Tecnologie e le Scienze

RELAZIONE FINALE

Segmentazione di istantanee fotografiche del cuoio capelluto tramite l'utilizzo di ResNet

Relatore: Prof. Massimo Melucci

Dipartimento di Ingegneria dell'Informazione

Laureando: Paolo Magagnato

Matricola N. 2001306

Anno Accademico 2022/2023

Sommario

ABSTRACT

Oggigiorno le reti neurali convoluzionali profonde giocano un ruolo essenziale in molti sistemi grazie alla capacità di avere un’efficacia crescente all’aumentare del numero di strati presenti, utili per l’identificazione di concetti sempre più complessi e astratti. Queste tipologie di reti vengono principalmente utilizzate nella classificazione di immagini, tramite l’identificazione di pattern e forme contenute all’interno di esse.

Vogliamo quindi sfruttare questa tecnologia per poter costruire un’applicazione software che sia in grado di poter riconoscere alcuni problemi relativi al cuoio capelluto in maniera automatica e valutarne il funzionamento.

1. Introduzione

1.1 Analisi di una fotografia

Ricavare informazioni da una fotografia è una sfida che hanno accolto molte aziende informatiche in questi ultimi anni per poter trarre vari benefici, dalla possibile ricerca attraverso l’uso di immagini di un determinato argomento (per esempio, in botanica, l’identificazione di una specie di una pianta) al riconoscimento di contenuti sensibili per poter censurare o impedire che determinato materiale mediatico possa circolare senza filtri in Internet.

Per poter superare questo ostacolo, nel mondo della Computer Science, sono state utilizzate delle particolare strutture denominate *kernel*.

1.2 Kernel

Il *kernel*, o filtro, è una matrice $n_{lunghezza} \times n_{altezza}$ (tipicamente quadrata con dimensione 3×3, 5×5 o 7×7) che, oltre ad analizzare il singolo pixel analizza l'area intorno ad esso; un filtro è perciò caratterizzato da $n_{lunghezza} \times n_{altezza}$ valori, uno per ogni cella dello stesso. Inoltre, il filtro può anche espandersi tridimensionalmente (perciò avere dimensione $n_{lunghezza} \times n_{altezza} \times n_{profondità}$).

I filtri, opportunamente impostati, non solo permettono di dare effetti alle istantanee, ma si possono sfruttare anche per mettere in risalto le componenti che vogliamo siano catturate, come, ad esempio, bordi o contorni: questo lavoro di confronto tra l'immagine e il filtro è oggi nota con il termine "convoluzione". Per convoluzione si intende quell'operazione in cui i pixel dell'immagine analizzati dal filtro vengono moltiplicati per i pesi dello stesso e successivamente sommati per ottenere il pixel con la modifica apportata dal filtro.

1.3 Reti neurali

Sebbene all'inizio questi filtri si preimpostassero, col tempo ha sempre preso più il sopravvento l'idea che si potessero regolare in

maniera automatica in base al problema da risolvere. Per fare ciò, i ricercatori si sono appoggiati alle reti neurali [1].

Lo scopo della rete neurale è perciò quello di trovare i filtri ottimali (partendo da dei filtri con valori casuali) affinché le varie immagini siano correttamente classificate: in questo modo la rete neurale sfrutta i kernel per identificare dei pattern in maniera graduale, andando a prendere come input la matrice di dati presente allo strato precedente e restituendo nell'output la matrice di dati a cui è stato applicato il filtro, che passa allo strato successivo. É così possibile ottenere una matrice di dati molto più ristretta contenente le principali componenti di un'immagine che vengono infine passate allo strato di output per ottenere la classe predetta.

1.4 Scelta del modello

La creazione e il successivo apprendimento legato a questa tipologia di reti neurali sono già stati realizzati da aziende informatiche che, in alcune sfide negli anni passati, si sono occupate in maniera specifica di questi temi. Data l'elevata onerosità a livello computazionale e la richiesta di particolari conoscenze a livello implementativo, queste reti sono già state messe a disposizione degli utenti dalle principali librerie in Python, uno dei linguaggi cardine nel mondo del data science.

Da qui in poi la scelta della rete neurale spetta all'usufruitore: in tale progetto, come modello adatto all'insieme di dati e alla casistica, si è optato per il modello ResNet.

1.5 ResNet

ResNet nasce nel 2015 da alcuni dipendenti e ricercatori dell'azienda Microsoft come modello per migliorare la precisione sfruttando una rete neurale convoluzionale profonda. Questa tipologia di rete, rispetto a quella semplice, presenta però un grave problema noto come "*vanishing gradient*", ossia l'azzeramento del gradiente. Questo dilemma porta al mancato aggiornamento dei pesi e dei bias dei primi strati, rimanendo quindi invariati e portando a un errore crescente nella valutazione durante il training, con conseguente minore efficacia.

Per ovviare a questa criticità, gli ideatori di ResNet si sono appoggiati a una serie di connessione aggiuntive che saltano due o tre (in base al modello ResNet) strati di convoluzione successivi e propagando in avanti la matrice di input della rete a quel passo: questi collegamenti sono definiti con il termine di *skip connection*. Con queste nuove connessioni, la rete diventa un agglomerato di blocchi residuali (da qui il nome ResNet), che, dopo aver svolto i loro calcoli, aggiungono il risultato alle convoluzioni svolte precedentemente. Tramite questa costruzione della rete neurale, il modello mantiene un punto di riferimento

con i risultati degli strati antecedenti e in questo modo, durante la fase di apprendimento, il gradiente, tramite back-propagation, passa attraverso queste *skip connection* e non si azzera all'arrivo dei primi strati [2].

Per la rete neurale ResNet sono state costruite diverse versioni in base al numero di strati costituiti. Per questo insieme di dati, si è deciso di implementare due modelli di ResNet: ResNet-34 e ResNet-50.

2. Dataset

Per poter svolgere le analisi durante lo svolgimento del progetto nell'azienda BOSS, è stato utilizzato un dataset contenente 19959 immagini suddivise in 23 categorie, ciascuna delle quali rappresenta un disturbo o una patologia riguardante i capelli o il cuoio capelluto.

Le immagini sono state ottenute tramite un dispositivo denominato Mic-Fi, un microscopio portatile in grado di poter scattare istantanee. Questa modalità peculiare di acquisizione di immagini non ha però reso possibile ampliare e diversificare l'insieme di dati già in possesso con fonti esterne in quanto la ricerca di queste tipologie di immagini nelle piattaforme web risulterebbe troppo complicata e dispendiosa economicamente data la loro unicità.

3. Obiettivo

Lo scopo del progetto di questo stage è quello di individuare un modello in grado di segmentare le fotografie del cuoio capelluto e successivamente valutarne l'efficacia confrontando le previsioni generate da tale modello con quelle effettive.

4. Dichiarazione dei modelli

Considerato il dataset a disposizione, si è voluto testare quale modello è più preciso tra ResNet-34 e ResNet-50 nella valutazione delle fotografie.

Per implementare tali reti neurali, si è fatto uso del linguaggio Python e, in particolare, della libreria PyTorch, che contiene al suo interno i principali strumenti riguardanti questa tipologia di modelli.

Per prima cosa, affinché PyTorch possa fare eventuali elaborazioni in maniera più efficiente, è stato dichiarato un oggetto che si occupa di compiere le operazioni necessarie per le elaborazioni, in particolar modo quelle vettoriali, come sotto riportato:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Successivamente, si è passati alla dichiarazione dei modelli sopra citati, che sono stati salvati in apposite variabili, e per fare ciò, è stato applicato il seguente codice:

```
resnet50_ft = models.resnet50(weights = models.ResNet50_Weights.IMAGENET1K_V1)
resnet34_ft = models.resnet34(weights = models.ResNet34_Weights.IMAGENET1K_V1)
```

Dal nome dei parametri delle funzioni chiamate si nota che questi modelli sono stati costruiti e addestrati per classificare le immagini di un altro dataset chiamato ImageNet, un'immensa collezione di immagini generiche (più di un milione) categorizzato in mille classi nelle quali non sono però presenti quelle del dataset del progetto. Tuttavia, per categorizzare le istantanee ottenute con Mic-Fi è stato sfruttato una particolare tipologia di addestramento: il *transfer learning*.

4.1 Transfer learning

Il *transfer learning* è una modalità d'apprendimento dove un modello già addestrato viene appositamente riaddestrato su un nuovo dataset, ossia dei nuovi dati vengono aggiunti e i parametri ottimali della rete neurale vengono ricalcolati. In questo modo è possibile ottenere una classificazione diretta delle immagini associando le classi che sono a disposizione al posto di quelle di default definite da ImageNet.

4.2 Pre-processing

Per fare in modo che le immagini di Mic-Fi vengano correttamente categorizzate, è stato necessario innanzitutto modificare lo strato di output affinché contenesse le 23 categorie del dataset del progetto invece delle 1000 categorie di ImageNet. Questa variazione è stata apportata prima che al modello venisse applicata la funzione di transfer learning ed è di seguito riportata:

```
num_features_real = 23
num_ftrs_50_ft = resnet50_ft.fc.in_features
resnet50_ft.fc = nn.Linear(num_ftrs_50_ft, num_features_real)
num_ftrs_34_ft = resnet34_ft.fc.in_features
resnet34_ft.fc = nn.Linear(num_ftrs_34_ft, num_features_real)
```

Dopo aver modificato lo strato di output, è stato necessario per prima cosa dividere il dataset in un insieme di training, che sarebbe poi passato ai modelli per ricalcolare i pesi, e un insieme di test, che sarebbe stato in seguito utilizzato per compiere analisi sui due modelli e ottenere dei risultati sulle loro performance. Si è deciso di non implementare, invece, la cross-validazione poiché è altamente onerosa a livello computazionale per le risorse hardware a disposizione e richiede molto tempo per essere applicata.

Successivamente, le foto contenute nei due insiemi sono state collocate nelle rispettive macro-cartelle, una cartella “train” e una cartella “test”, più precisamente nelle 24 sottodirectory delle varie

categorie presenti in ciascuna di esse. Questo processo è stato eseguito dal codice che segue:

```
final_destination = "C:\\Users\\PiEmme\\Downloads\\valutation"
#Creo repository per training set e test set
os.mkdir(final_destination)
data_dir = 'C:\\Users\\PiEmme\\Downloads\\archive'
reps = os.listdir(data_dir)
tmp_dataset = datasets.ImageFolder(data_dir)
#Divisione del dataset in training set e test set
train_list = []
test_list = []
for j in range(0, num_features_real):
    tmp = []
    for i in range(0, len(tmp_dataset)):
        ie = tmp_dataset.imgs[i]
        if(ie[1] == j):
            tmp.append(ie)
    train, test = train_test_split(tmp, train_size = 0.79, random_state=123)
    for x in train:
        train_list.append(x)
    for y in test:
        test_list.append(y)
valutation = ['train', 'test']
list_set = {"train" : train_list, "test": test_list}
for set in valutation:
    first = True
    for x in reps:
        #Creazione della macro-cartella
        if(first):
            cartella_destinazione = final_destination + "\\" + set
            os.mkdir(cartella_destinazione)
            first = False
        #Creazione delle sottodirectory e successivo riempimento
        cartella_destinazione += "\\" + x
```

```
        for image in list_set[set]:
            img = image[0]
            if(x in img):
                # Estrai il nome del file dall'array
                nome_image = os.path.basename(img)
                # Crea il percorso completo di destinazione
                percorso_destinazione = os.path.join(cartella_destinazione,  nome_image)
               # Copia l'immagine nella cartella di destinazione
               shutil.copy(img, percorso_destinazione)
      cartella_destinazione = final_destination + "\\" + set
data_dir = final_destination
```

Nel successivo step si è passati alla trasformazione delle immagini, dato che i modelli ResNet34 e ResNet50 accettano un'immagine con grandezza 224 × 224. Tuttavia, non ci si è limitati a un semplice ritaglio, ma alla seguente trasformazione per il training set:

```
  transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.5407, 0.4138, 0.3821], [0.1912, 0.1643, 0.1587])
  ])
```

Il motivo che si cela dietro a questa tipologia di trasformazione è l'applicazione di un'ulteriore elaborazione dei dati, che consente alla rete neurale di percepire molte più informazioni dalle immagini di training e ciò è stato svolto attraverso l'uso di due funzioni: il *random crop*, che ha tagliato l'immagine alla

dimensione 224 × 224 partendo da un punto casuale, e il *random horizontal flip*, che l'ha capovolta in maniera randomica. La principale conseguenza di queste trasformazioni è che il modello diventa più performante e percepisca le caratteristiche indipendentemente dall'orientamento e dall'ubicazione di quest'ultime nell'immagine. Infine, sono stati normalizzati i valori RGB delle diverse istantanee, utilizzando media e deviazione standard ottenute dalle immagini dell'intero dataset.

Per il test set è stata usata, invece, la trasformazione sotto riportata, che si limita a ridimensionare l'immagine, tagliarla centrata in una figura 224 × 224 e normalizzare i valori RGB della stessa:

```
transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.5407, 0.4138, 0.3821], [0.1912, 0.1643, 0.1587])
])
```

Infine, sono stati caricati i modelli sul dispositivo prima riportato.

```
resnet50_ft = resnet50_ft.to(device)
resnet34_ft = resnet34_ft.to(device)
```

4.3 Implementazione

Nello step successivo, è stata implementata la funzione sottostante, che si è occupata del *transfer learning*:

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=50):
    since = time.time()
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')
        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0
        for epoch in range(1,num_epochs+1):
            #Ogni epoch ha una fase di training e una di test
            for phase in ['train', 'test']:
                if phase == 'train':
                        model.train()  # Imposta il modello in modalità “training”
                else:
                        model.eval()   # Imposta il modello in modalità “valutazione”
                running_loss = 0.0
                running_corrects = 0
                for inputs, labels in dataloaders[phase]:
                        inputs = inputs.to(device)
                        labels = labels.to(device)
                        #Azzera i gradienti dei parametri
                        optimizer.zero_grad()
                        #Traccia la storia solo se in fase di training
                        with torch.set_grad_enabled(phase == 'train'):
                                outputs = model(inputs)
                                _, preds = torch.max(outputs, 1)
                                loss = criterion(outputs, labels)
                                if phase == 'train':
                                    loss.backward()
                                    optimizer.step()
                        #Statistiche
                            running_loss += loss.item() * inputs.size(0)
                            running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                        scheduler.step()
```

```
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')
            #Copia profonda del modello
            if phase == 'test' and epoch_acc > best_acc:
                best_acc = epoch_acc
                torch.save(model.state_dict(), best_model_params_path)
        time_elapsed = time.time() - since
        print(f'Training concluso in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
        print(f'Best Acc: {best_acc:4f}')
        #Carica il modello con i miglior pesi
        model.load_state_dict(torch.load(best_model_params_path))
    return model
```

Tramite tale funzione sono stati inseriti i seguenti parametri:

- *model*, il modello della rete neurale opportunamente caricato nel dispositivo di PyTorch;
- *criterion*, il criterio con cui la funzione di perdita viene calcolata tra i valori predetti e quelli reali;
- *optimizer*, il tipo di ottimizzatore (Stochastic Descent Gradient, un ottimizzatore a livello computazionale meno oneroso dell'algoritmo di discesa del gradiente ma con simile effetto, o Adam, altra tipologia di ottimizzatore più preciso del precedente);
- *scheduler*, la funzione che modifica la velocità d'apprendimento durante la fase di addestramento, andandola a diminuire progressivamente dopo un determinato numero di *epoch* prestabilite;

- *num_epoch*, il numero di epoche (o *epoch*) per il training della rete neurale necessarie per ottenere la classificazione più ottimale. Questo parametro è particolarmente delicato soprattutto se il dataset preso in considerazione è ampio: infatti, anche con 25 *epoch*, il tempo per il training può durare fino a un'intera giornata.

La definizione completa di tutti questi parametri è molto delicata e richiede sufficiente pratica per capire quali sono i parametri ideali per il dataset. In questo progetto, le variabili precedente elencate sono state così impostate:

```
criterion = nn.CrossEntropyLoss()
optimizer_50 = optim.SGD(resnet50_ft.parameters(), lr=0.001,
momentum=0.875)
exp_lr_scheduler_50 = lr_scheduler.StepLR(optimizer_50, step_size=10,
gamma=0.1)
optimizer_34 = optim.SGD(resnet34_ft.parameters(), lr=0.01, momentum=0.6)
exp_lr_scheduler_34 = lr_scheduler.StepLR(optimizer_34, step_size=10,
gamma=0.1)
num_epoch = 25
```

Dopo aver opportunamente addestrato le reti neurali tramite transfer learning, attraverso la chiamata delle funzioni sopra citate, i modelli ResNet-34 e ResNet-50 erano pronti per la successiva fase di valutazione.

5. Valutazione

Dopo l'addestramento di questi due modelli, è stato possibile analizzare quanto il sistema fosse efficace nella classificazione e

per farlo sono stati usati due strumenti statistici: la matrice di confusione e la curva ROC. Prima di utilizzare queste metriche, è stato necessario definire una funzione che fosse in grado di restituire le classi predette e le probabilità associate alla scelta di quelle classi, qua sotto definita:

```
def visualize_model(model, num_images=len(image_datasets["test"])):
    was_training = model.training
    model.eval()
    images_so_far = 0
    predicted_labels = []
    true_labels = []
    predicted_probabilities = []
    k = 0
    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['test']):
            inputs = inputs.to(device)
            labels = labels.to(device)
            output = model(inputs)
            _, preds = torch.max(output, 1)
            probabilities = F.softmax(output, dim=1)
            for j in range(inputs.size()[0]):
                images_so_far += 1
                predicted_labels.append(int(preds[j]))
                true_labels.append(int(labels[j]))
                predicted_probabilities.append(probabilities[j].cpu())
                if images_so_far == num_images:
                    model.train(mode=was_training)
                    return true_labels, predicted_labels, np.array(predicted_probabilities)
```

5.1 Matrice di confusione

Dato l'insieme di test, è stato possibile creare una matrice di confusione in cui mettere in corrispondenza le vere classi, che sono conosciute a priori, con le classi predette dal nostro modello per ogni fotografia, usando il codice che segue:

```
total = len(image_datasets["test"])
cm_custom_34 = confusion_matrix(true_labels_custom_34, pred_labels_custom_34)
cm_custom_50 = confusion_matrix(true_labels_custom_50, pred_labels_custom_50)
cm_display_custom_34 = ConfusionMatrixDisplay(cm_custom_34)
cm_display_custom_50 = ConfusionMatrixDisplay(cm_custom_50)
fig, ax = plt.subplots(figsize=(15, 15))
ax.set_title(f"""Matrice di confusione per il modello ResNet-34 custom
Somma degli elementi correttamente individuati: {sum(diag(cm_custom_34))}
Rapporto di elementi correttamente individuati rispetto al totale:
{sum(diag(cm_custom_34))/total:.2f}""")
cm_display_custom_34.plot(ax = ax)
fig, ax = plt.subplots(figsize=(15, 15))
ax.set_title(f"""Matrice di confusione per il modello ResNet-50 custom
Somma degli elementi correttamente individuati: {sum(diag(cm_custom_50))}
Rapporto di elementi correttamente individuati rispetto al totale:
{sum(diag(cm_custom_50))/total:.2f}""")
cm_display_custom_50.plot(ax = ax)
```

Con questo codice, è stato poi possibile mostrare le seguenti immagini:

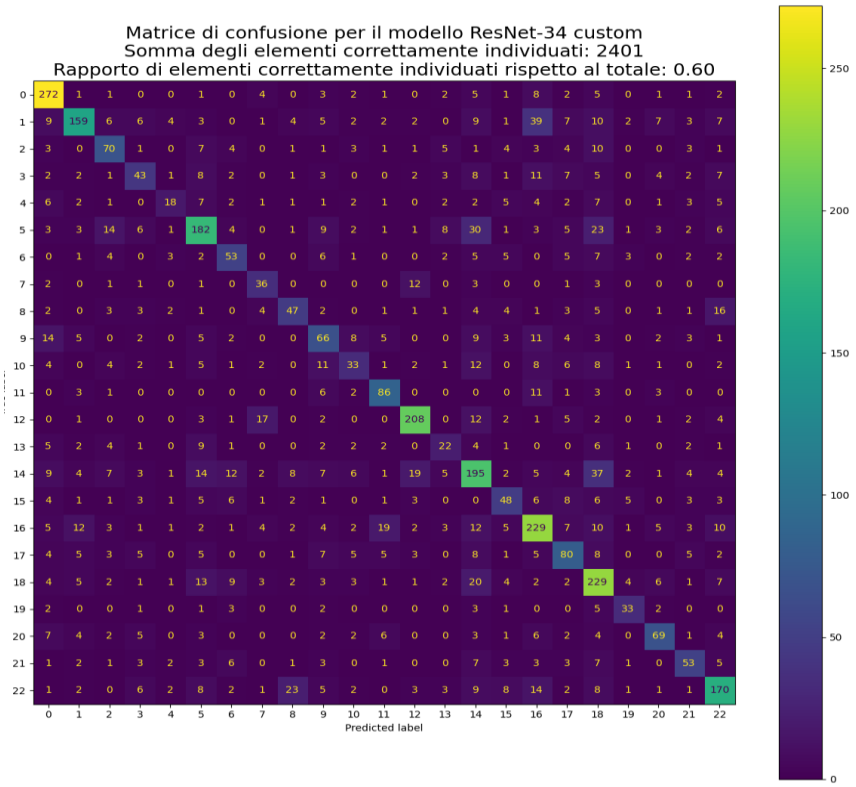

Figura 1: Matrice di confusione ottenuta con la rete neurale ResNet-34

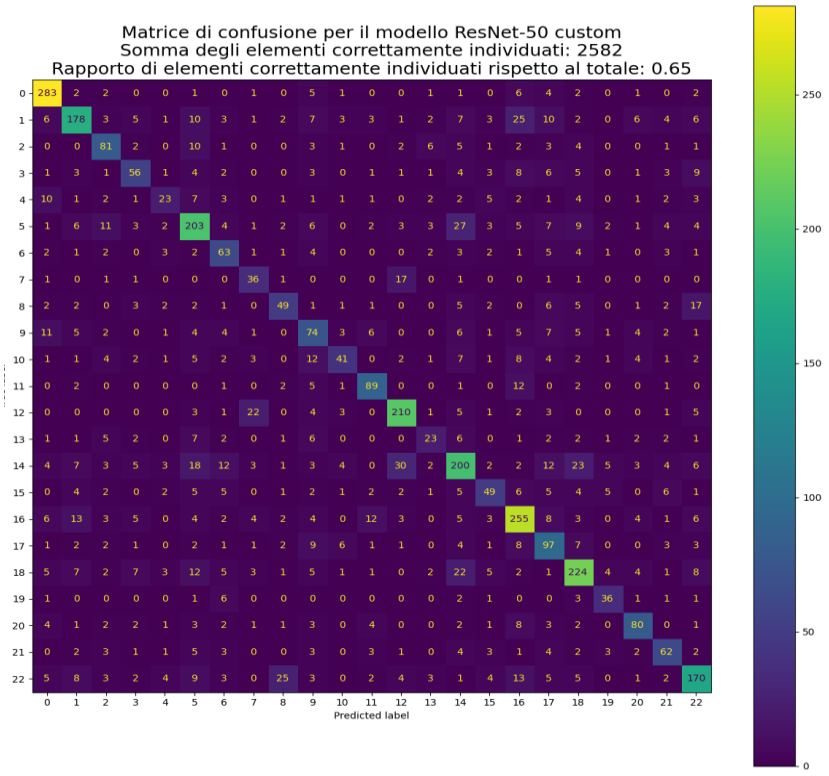

Figura 2: Matrice di confusione ottenuta con la rete neurale ResNet-50

5.2 Curva ROC

La curva ROC (o Receiving Operator Charateristic) è uno strumento statistico che confronta la sensibilità con il complemento della specificità per un determinato classificatore binario. Tuttavia, in questo progetto è stato usato un classificatore multiclasse, il cui risultato è il numero della classe a cui appartiene la nostra immagine. È stato possibile comunque calcolare la curva ROC mediante la creazione di un ciclo il cui numero di iterazioni fosse pari a quello delle categorie presenti: a ogni passo, è stato definito un classificatore binario in cui è stato dato un valore pari a 1 nel caso in cui l'immagine facesse parte di quella categoria (definito dal numero dell'iterazione) e 0 in caso contrario. Questo classificatore è stato applicato sui valori predetti sfruttando i valori veri per poter calcolare il tasso di veri positivi (denominato sensibilità) e il tasso di falsi positivi (corrispondente al complemento della specificità, ossia il tasso di veri negativi) per ogni rispettiva categoria. Ottenuti i tassi, è stato possibile calcolare la curva ROC, utile soprattutto quando possiamo scegliere la soglia per la quale una previsione è vera o meno. Il sistema però nel nostro caso restituiva direttamente la classe con la probabilità associata a essa, senza alcuna scelta sulla soglia. Nonostante ciò, è stato possibile valutare queste reti neurali calcolando l'AUC o Area Under the Curve, ossia l'area che

definisce la curva ROC e che misura la capacità predittiva di un modello in un intervallo che va da 0 a 1: maggiore è l'AUC, maggiore è il numero di previsioni correttamente individuate. Il risultato finale è stato l'ottenimento di 23 curve ROC diverse che corrispondono alla prestazione del modello per ogni singola categoria. Il codice per il calcolo e il risultato di queste metriche è qui riportato:

```
roc_curves = []
for i in range(num_features_real):
    binary_true_labels_custom_34 = [1 if label == i else 0 for label in true_labels_custom_34]
    binary_predicted_labels_custom_34 = [1 if label == i else 0 for label in predicted_labels_custom_34]
    y_score_custom_34 = list(predicted_probabilities_custom_34[:,i])
    binary_true_labels_custom_50 = [1 if label == i else 0 for label in true_labels_custom_50]
    binary_predicted_labels_custom_50 = [1 if label == i else 0 for label in predicted_labels_custom_50]
    y_score_custom_50 = list(predicted_probabilities_custom_50[:,i])
    # Calcola la curva ROC per la classe 'i'.
    with torch.no_grad():
        fpr_custom_34,tpr_custom_34, _ =roc_curve(binary_true_labels_custom_34, y_score_custom_34)
        fpr_custom_50,tpr_custom_50, _ =roc_curve(binary_true_labels_custom_50, y_score_custom_50)
    # Calcola l'area sotto la curva ROC (AUC) per la classe 'i'.
    roc_auc_custom_34 = auc(fpr_custom_34, tpr_custom_34)
    roc_auc_custom_50 = auc(fpr_custom_50, tpr_custom_50)
    # Aggiungi la curva ROC alla lista.
    roc_curves.append([(fpr_custom_34, tpr_custom_34, roc_auc_custom_34),(fpr_custom_50, tpr_custom_50, roc_auc_custom_50)])
```

```
figures = {'ROC Curve for Class '+ str(i+1): roc_curves[i] for i in
range(num_features_real)}
fig, axeslist = plt.subplots(ncols=4, nrows=6, figsize = (25,25))
for ind,title in enumerate(figures):
    # Disegna la curva ROC per la classe 'i'
    tmp = axeslist.ravel()[ind]
    roc = figures[title]
    tmp.set_title(title, fontsize = 30, fontweight="bold")
    tmp.plot(roc[0][0], roc[0][1], color='darkorange', lw=2, label=f'ROC curve classic
(AUC = {roc[0][2]:.3f})')
    tmp.plot(roc[1][0], roc[1][1], color='blue', lw=2, label=f'ROC curve custom (AUC =
{roc[1][2]:.3f})')
    tmp.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    tmp.axis(xmin = -0.05, xmax = 1.05)
    tmp.axis(ymin = -0.05, ymax = 1.05)
    tmp.set_xlabel('False Positive Rate', fontsize = 20)
    tmp.set_ylabel('True Positive Rate', fontsize = 20)
    tmp.legend(loc='lower right', prop = {'size': 14})
    fig.tight_layout(pad = 3)
ax = axeslist.ravel()[23]
ax.set_axis_off()
```

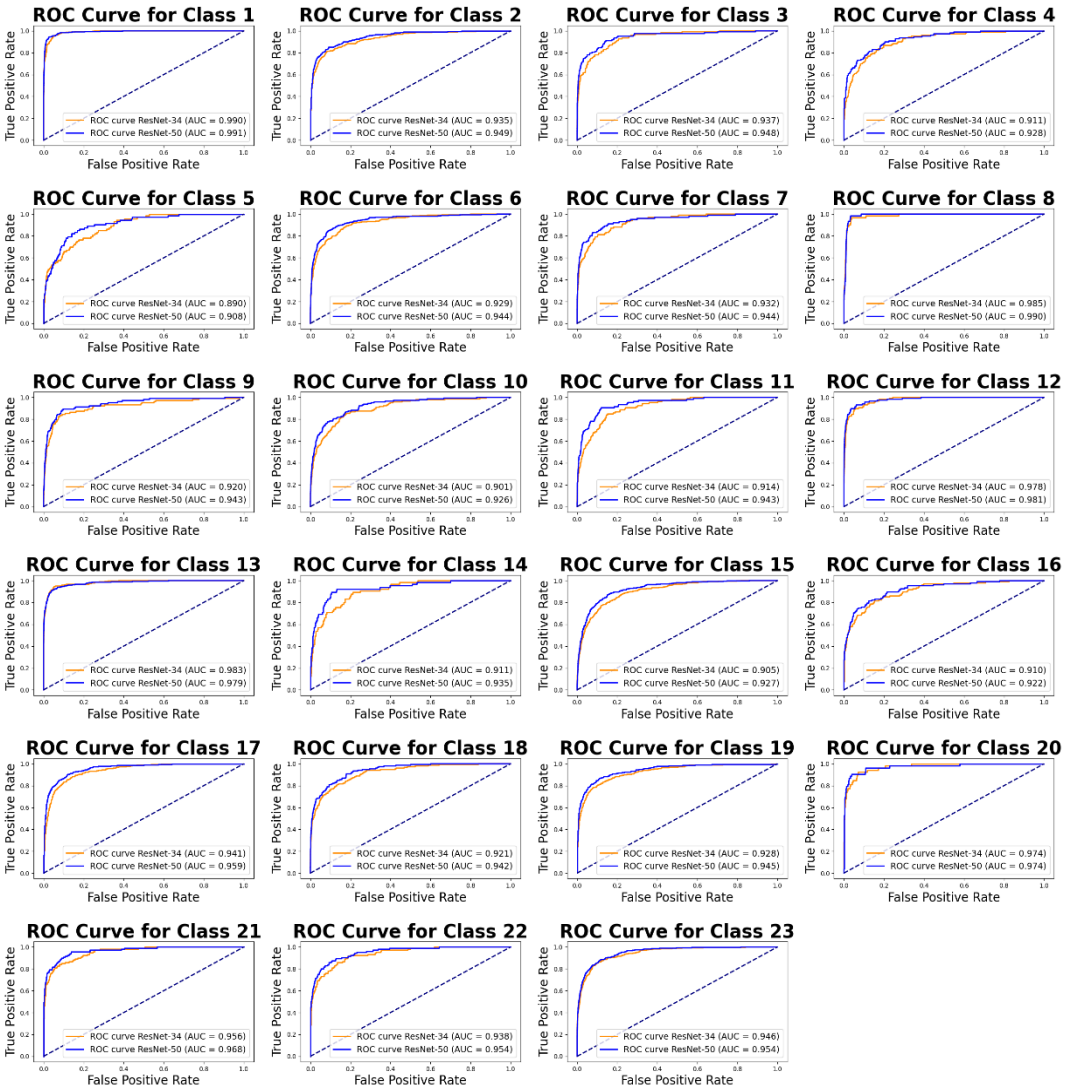

Figura 2: Curve ROC di ogni categoria per i modelli ResNet-34 e ResNet50

6. Conclusioni

Sebbene i modelli siano entrambi abbastanza precisi dato che superano il 60% nella corretta classificazione, è stata notata una precisione maggiore con la rete neurale ResNet-50 rispetto a ResNet-34, dato lo scarto di poco più di 150 immagini corrette, pari al 5% in più. Dalle immagini relative alle curve ROC, si conferma la deduzione precedente mostrando anche più chiaramente come ad ogni categoria l'AUC sia maggiore per ogni

classe, fatta eccezione per una. Questa maggiore efficacia si deve alla maggiore profondità (e corretta costruzione e implementazione) della rete ResNet-50 rispetto alla controparte ResNet-34. La scelta può però cambiare nel caso si preferisca un modello più leggero e con un minor tempo per compiere ulteriori riaddestramenti sullo stessa tipologia di immagini, dove, invece, ResNet-34 eccelle rispetto a ResNet-50.

Un'altra strada possibile è anche quella di usare altre tipologie di reti neurali convoluzionali (GoogLeNet tra tutte grazie alla sua rapida velocità d'addestramento e gli ottimi risultati a livello di precisione), però nessun modello riesce a essere preciso tanto quanto ResNet, in particolar modo con le sue versioni più profonde come ResNet-101 e ResNet-152 [3].

7. Bibliografia e sitografia

[1] Giovanni Nardini, "https://www.ai4business.it/intelligenza-artificiale/rete-neurale-convolutive-cosa-e/."

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Dec. 2015, [Online]. Available: http://arxiv.org/abs/1512.03385

[3] Aqeel Anwar, "https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96."