UNIVERSITÀ
DEGLI STUDI
FIRENZE

Report Progetto

Esame di Computational Learning

Implementazione di una Rete Neurale Convoluzionale per la classificazione di immagini

Autore:
Alberto Biliotti

Matricola:
7109894

1 Introduzione

1.1 Descrizione del task

L'obbiettivo del lavoro è quello di implementare una rete neurale convoluzionale (CNN) in grado di classificare immagini a colori di dimensione 32x32 appartenenti a dieci classi distinte, utilizzando il dataset Cifar-10.[1]. È stata usata in particolare la libreria "PyTorch" di Python per implementare i singoli moduli che compongono tale rete.

Figure 1: Esempio di immagine di un gatto appartenente al dataset

1.2 Ambiente di lavoro

Per fare ciò è stato utilizzato un computer portatile con un processore Intel i7 di undicesima generazione, 16 GB di memoria RAM a disposizione ed una scheda grafica NVIDIA GeForce RTX 3060 con 6 GB di memoria dedicata. La versione di Python che ho utilizzato è la 3.10.12 mentre la versione di PyTorch è la 2.1.1 con il supporto a CUDA versione 11.8.

1.3 Caratteristiche del dataset

Il dataset Cifar-10 contiene 60000 immagini a colori di aeroplani, automobili, navi, camion, rane, cavalli, uccelli, cervi, cani e gatti. Tali immagini sono contenute in sei file pickle distinti che è possibile importare su Python, in formato dizionario, grazie al metodo "unpickle" definito sul sito da cui è possibile scaricare il dataset [1]. Su tale sito viene inoltre affermato che ogni classe di immagini ha la stessa frequenza, per cui non dobbiamo preoccuparci di bilanciare le classi.

Figure 2: Esempio di immagine di un aereo appartenente al dataset

2 Preprocessing e splitting dei dati

2.1 Caricamento dei dati

Dopo aver importato il dataset nel progetto Python, è stato necessario trasformare le immagini e le etichette della classe di appartenenza in tensori Pytorch. Per fare ciò ho prima separato le immagini dalle relative etichette, ho separato per ciascuna immagine i canali corrispondenti ai colori rosso verde e blu e li ho inseriti in un array Numpy di dimensioni 3x32x32, ho quindi inserito tutte le immagini in una lista che ho trasformato infine in un tensore Pytorch.

2.2 Divisione dei dati

Una volta trasformato il dataset in un unico tensore, ho effettuato lo splitting dei dati. Prima di fare ciò ho impostato un seme per la generazione di numeri casuali in modo da poter riprodurre esattamente in ciascuna esecuzione le divisioni del dataset che andrò ad effettuare.
Ho scelto di dividere il dataset in tre sottoinsiemi, il primo per il training dei modelli, il secondo per la validazione dei modelli durante la model selection, ed il terzo, chiamato insieme di test, per la stima dell'errore di generalizzazione, che fornirò solo al modello scelto dopo aver effettuato la model selection.
Ho scelto di assegnare il 30% delle immagini all'insieme di test, mentre la restante parte è stata assegnata al 80% al training test (56% del totale) e la restante parte è rimasta all'insieme di validazione.
Ho quindi scelto di non implementare la cross-validation per i modelli che implementerò in quanto, come vedremo nelle sezioni successive, alcuni modelli che andrò a provare impiegano parecchio tempo per essere allenati e dover ripetere più volte la fase di training ad ogni iterazione della cross validation, anche con un numero di fold limitato, avrebbe richiesto troppo tempo.

2.3 Normalizzazione dei dati

Una volta ottenuto l'insieme di training è stato quindi deciso di normalizzare i pixel delle immagini attraverso la normalizzazione Z-score, calcolando per ciascun canale la media dei valori dei pixel e la relativa deviazione standard per poi sottrarre la media e dividere per la deviazione standard, ottenendo così dei dati con media zero e deviazione standard uno.
Per evitare di usare impropriamente gli insiemi di validazione e di test ho scelto di calcolare la media e la deviazione standard, con cui andrò a normalizzare tutti i dati, solo delle immagini presenti nell'insieme di training.
In questo modo avremo che, almeno per l'insieme di training, la media del valori dei pixel di ciascun canale sarà pari a zero mentre la deviazione standard avrà valore uno, in maniera tale che i valori dei pixel di ciascun canale siano confrontabili tra loro, migliorando così le prestazioni del modello che andrò ad implementare.

3 Implementazione della Rete Neurale Convoluzionale

Una volta preparate e separate le immagini è possibile implementare tramite PyTorch i moduli che andremo ad usare per comporre la nostra rete neurale, in particolare prima introdurrò i moduli che abbiamo visto a lezione, mentre in seguito esporrò anche gli altri metodi giustificando la ragione per cui sono stati introdotti.
Ciascun modulo che andrò a definire è stato implementato come una classe che eredita dalla super classe "module" di PyTorch, di cui ho ridefinito il costruttore ed il metodo "forward".

3.1 Convoluzione

Volendo implementare una rete neurale convoluzionale, mi è sembrato naturale partire implementando il livello convolutivo della rete. Ho quindi definito una classe in cui il costruttore prende in input il numero di canali di input, il numero di canali di output, la dimensione del filtro convolutivo (quadrato), lo stride ed il kernel.
Una volta fatto ciò il vettore dei parametri è generato come una matrice quadridimensionale, in cui, per ogni canale di output, sono contenuti tutti i relativi filtri convolutivi per i canali in input con la dimensione del kernel scelta in precedenza.
Ho deciso di inizializzare il filtro con valori campionati da una distribuzione uniforme con estremi $-\sqrt{\frac{1}{C_{in}*{k_{size}}^2}}$ e $\sqrt{\frac{1}{C_{in}*{k_{size}}^2}}$, dove C_{in} è il numero di canali in input e k_{size} la dimensione del kernel, in quanto è lo stesso metodo usato dalla libreria PyTorch [2], che effettivamente offre delle prestazioni migliori rispetto ad altre scelte di inizializzazione oltre ad essere più semplice rispetto ad altre soluzioni proposte in letteratura.
Per quanto riguarda il padding invece prima calcolo se è necessario, e in caso affermativo di quanto, controllando se la formula $\frac{W-F+2P}{S}+1$ restituisce un numero intero, dove W è la dimensione della matrice in input, F quella del filtro ed S lo stride, mentre P rappresenta la quantità di padding richiesta, che partendo da 0 incremento finché la formula descritta in precedenza non restituisce un valore intero. Ho quindi creato il metodo "padd_img" che, dato il tensore contenente la batch di immagini su cui stiamo lavorando e il padding richiesto, genera una tensore casuale (con valori estratti da una normale standard) di dimensioni pari al tensore originale ma con le matrici di ogni canale di dimensione pari a quella originale più due volte la quantità di padding richiesto. Una volta fatto ciò inserisco al centro di ciascuna delle matrici che compongono il tensore la matrice del canale corrispondente e ritorno il tensore così generato.
Una volta definito il filtro con cui andrò a lavorare posso implementare l'operazione di convoluzione vera e propria ridefinendo il metodo forward.
Sono prima partito da un metodo che fosse il più rapido possibile in fase di forward senza tenere conto della backpropagation. Ho quindi implementato un metodo che "appiattiva" ciascun canale delle immagini appartenenti alla batch in input in un vettore monodimensionale. Successivamente trasformava ciascun canale della matrice filtro contenente i parametri in una matrice composta da righe, di dimensione pari al numero di pixel delle immagini in input, che erano tutte a valori nulli tranne nelle posizioni in cui era presente la finestra della convoluzione. In questa maniera "shiftando" volta volta la stessa riga riuscivo a simulare lo spostamento della finestra in base allo stride e a comporre una matrice che, una volta trasposta e post-moltiplicata per la matrice delle immagini "appiattite", ritornava i valori ottenuti dalla convoluzione per ogni singolo canale. A questo punto era sufficiente sommare i valori ottenuti per i canali di input e assemblare i valori così ottenuti in una nuova matrice quadrata per ogni canale di output.
Tale metodo era anche più rapido dell'implementazione di PyTorch nella fase di forward, tuttavia era estremamente più lento nella fase di backpropagation, in quanto andavo a intervenire pesantemente sui parametri, ossia i filtri, prima di moltiplicarli per la matrice dei dati, ciò comportava che il calcolo del gradiente per ogni parametro richiedesse molto tempo, andando ad inficiare sul tempo di esecuzione.
Per risolvere questo problema ho quindi cercato di toccare il meno possibile matrice dei filtri, andando invece a modificare la batch di dati in input. Ispirandomi sempre all'implementazione di Pytorch, ho quindi definito un metodo che, partendo dal tensore quadridimensionale della batch di immagini in input, similmente al metodo unfold di Pytorch, restituisca un tensore tridimensionale che, per ogni immagine, contiene una matrice bidimensionale della matrice originale che ha come colonne gli elementi appartenenti alla stessa finestra convolutiva che andranno moltiplicati per i relativi pesi contenuti nel kernel per poi essere sommati tra loro per ottenere il risultato di una singola iterazione della convoluzione.
Una volta ottenuta tale matrice tridimensionale posso quindi andare a trasporne le ultime due dimensioni per poter così effettuare un prodotto matriciale con la matrice dei filtri, "appiattita" anch'essa, ma ancora divisa in base ai canali di uscita e successivamente trasposta, ottenendo così i valori risultanti dalla convoluzione che, opportunamente ri-trasposti e ridimensionati, compongono la matrice risultante.
Il metodo così implementato risulta decisamente più lento della versione precedente in fase di forward, ma in backpropagation è nettamente più rapido in quanto ho dovuto solamente "appiattire" per canale di uscita i filtri

convoluzionali per poi trasporli e infine moltiplicarli, senza dover quindi intervenire pesantemente sui parametri del modello come avveniva nella prima implementazione.

3.2 ReLU

Ho deciso di inserire come funzione che garantisse la non-linerità del nostro modello ricorrendo alla funzione ReLU che restituisce 0 se il numero in input è negativo e il numero stesso altrimenti.
Per implementare la funzione ReLU (rectified linear unit) è stato sufficiente applicare la funzione max di PyTorch, che valuta la funzione massimo in ciascuna posizione del tensore, e restituire quindi il massimo tra gli elementi del tensore e 0.

3.3 Pooling

Ho scelto di implementare il pooling massimale in quanto è la tecnica di pooling più comunemente utilizzata per l'implementazione di reti neurali convoluzionali. Per fare ciò ho definito il metodo "max_pool", che, dati in input il tensore con le immagini e la dimensione del filtri, desiderata (lo stride sarà pari a tale valore, in maniera che le finestre non si sovrappongano), applica la funzione max ad ogni sotto-matrice di dimensione desiderata e ricompone la matrice sotto-campionata desiderata.

3.4 Multi Layer Perceptron

Essendo il mio task di classificazione, dovrò avere anche nei livelli finali uno o più Multi Layer Perceptron. Per implementarlo ho definito la classe MLP, il cui costruttore prende in input la dimensione dei tensori in input e il numero di valori che deve restituire in output. I due vettori dei parametri, ossia i pesi W e il bias b, sono inizializzati in maniera casualmente campionando, in maniera simile a quanto fatto per il layer convoluzionale, da una distribuzione uniforme con estremi $-\frac{1}{num\_input}$ e $\frac{1}{num\_input}$, tale inizializzazione è uguale a quella effettuata dal modulo "Linear" di Pytorch[3].
Per il forward invece è sufficiente moltiplicare il tensore contenente la batch di immagini in input per il vettore dei pesi trasposto e sommare il vettore dei bias e ritornare i valori così ottenuti.

3.5 Batch Normalizzation

Ho scelto di implementare la batch normalization in quanto permette di rendere l'allenamento della rete più rapido e stabile durante fase di training, riducendo alcuni problemi tipici delle reti neurali come il "Internal Covariate Shift" [4].
La classe che ho implementato prende quindi in input nel costruttore il numero di canali delle immagini nella batch, un valore (piccolo) eps per evitare di dividere per 0, ed un valore per il momento con cui andrò ad aggiornare media e varianza mobili durante la fase di training.
In pratica la Batch Normalizzation consiste quindi in un layer che, in fase di training, prima normalizza la batch di dati in input per ogni canale, sommando prima il valore eps alla varianza e moltiplica il risultato elemento per elemento per una matrice di pesi W e ci somma un'ulteriore matrice di bias b. Tali valori rappresentano due ulteriori tensori di parametri da imparare. Oltre a questo durante il training aggiorno costantemente due vettori di media e varianza mobili, in cui la media e la varianza dei canali della batch attualmente in input viene pesata in base al valore del momentum specificata in input nel costruttore e sommata ai valori di media e varianza mobili ricavati nelle precedenti epoche.
In fase di validazione o test, invece di usare la media e la varianza delle immagini nella batch per normalizzare i dati, userò la media e la varianza mobili che abbiamo calcolato durante il training senza ovviamente aggiornare tali valori.
In questo modo dovrei quindi riuscire sperabilmente ad ottenere più stabilita e velocità in fase di training.

3.6 Dropout

Volendo infine aggiungere un meccanismo di regolarizzazione in maniera da tale da ridurre il rischio di over-fitting, ho deciso di implementare anche il Dropout, che, in una rete neurale convoluzionale, durante la fase di addestramento, dato in input un iper-parametro p, pone a 0 alcuni pixel dell'immagine in input estraendo da una bernoulliana con probabilità p, in maniera tale da far allenare la rete ogni volta con immagini in input leggermente modificate. Durante la fase di test o validazione questo procedimento non viene effettuato. I valori che non sono posti a zero sono anche divisi per (1-p) prima di essere restituiti per tentare di regolarizzare ulteriormente la rete.

3.7 Parallelizzazione su GPU

Avendo a disposizione una scheda grafica Nvidia, ho deciso di sfruttarla per parallelizzare almeno una parte del mio codice per poter avere un'esecuzione più rapida. Per fare ciò ho dovuto installare la versione di PyTorch con CUDA. Nel codice controllo se effettivamente la GPU è rilevata, in caso contrario il codice viene comunque eseguito completamente in CPU. Se la GPU è rilevata correttamente sposto tutti i parametri del modello e la batch di immagini sui cui lavoro nella memoria della scheda grafica ed eseguo lì tutti i prodotti tra matrici in maniera ottimizzata.
Facendo ciò riduco di circa un quarto il tempo di esecuzione necessario per il training di ciascuna rete, anche se ciò dipende molto anche dal numero di immagini presenti nella batch e dalla dimensione della rete stessa.

4 Model Selection

Avendo definito i moduli che mi permettono di costruire la mia rete neurale convoluzionale, posso passare alla fase di model selection, in cui andrò a definire la mia funzione di loss da minimizzare e l'algoritmo di ottimizzazione, oltre ad introdurre diversi strategie come l'early stopping. Successivamente testerò diversi modelli, con struttura ed iper-parametri differenti, per poi cercare la configurazione che massimizza l'accuratezza sul validation set.

4.1 Funzione di Loss

Visto che il mio task è di classificazione non binaria, la funzione di loss più adatta è la Cross Entropy Loss, la cui implementazione in Pytorch comprende anche l'applicazione della funzione di log-softmax per normalizzare i dieci valori in uscita dalla rete. Tale funzione diventerà quindi la funzione obbiettivo da minimizzare durante il training.

4.2 Algoritmo di ottimizzazione

Una volta scelta la funzione obbiettivo da minimizzare, ho dovuto scegliere un algoritmo di ottimizzazione per la discesa del gradiente. Invece di ricorrere al "tradizionale" algoritmo stocastico SGD ho preferito utilizzare il metodo AdAM (Adaptive Moment Estimation), che garantisce solitamente prestazioni migliori in quanto usa strategie più sofisticate come il momentum o il learning rate adattivo[5].
Avendo scelto un algoritmo di ottimizzazione che implementa diverse strategie, avrei di conseguenza anche tanti iper-parametri da allenare, come il learning rate iniziale o il valore del decadimento per il momentum. Tuttavia ho preferito lasciare i valori di default in quanto il tempo di addestramento della rete, nonostante le varie ottimizzazioni, è abbastanza lungo e quindi non posso permettermi di validare tutti questi iperparametri, visto che ho cose molto più importanti da validare come la struttura della rete stessa.

4.3 Minibatch Training

Ho deciso di evitare di aggiornare i parametri dopo aver visto l'intero insieme di training, in quanto ciò comporterebbe un deterioramento delle prestazioni del modello sopratutto in termini di tempo di esecuzione (visto che uso anche la GPU), ma in parte anche perchè ciò dovrebbe aiutare ridurre il numero di epoche necessarie per il training del modello se scegliamo un numero di immagini per minibatch adeguato.
Introducendo tale meccanismo però non avrò più la garanzia che la loss diminuisca dopo ogni epoca, inoltre ciò introduce un nuovo iper-parametro da allenare che è la dimensione della batch. Ho deciso quindi di validare due diversi valori di tale iper-parametro, ossia 128 e 256.

4.4 Early stopping

Un ultimo meccanismo di regolazione da introdurre per evitare che la nostra rete vada in overfitting è l'early stopping. Tale meccanismo consente di interrompere il training della nostra rete se l'accuratezza che otteniamo nella fase di validazione della rete dopo ogni epoca non aumenta significativamente dopo un numero di epoche da noi scelto, in tal modo evitiamo che il modello si alleni eccessivamente sui dati di training, perdendo così la capacità di generalizzazione desiderata. Ho scelto di utilizzare l'accuratezza e non la loss di validazione in quanto più intuitiva come metrica di valutazione e di successivo confronto tra modelli.
I parametri del modello con più accuratezza sul validation set sono quindi salvati in un file e, una volta attivato l'early stopping o terminato il numero di epoche massimo, ripristinati. Questo passaggio è stato abbastanza complicato da implementare in quanto all'inizio non avevo definito la media e la varianza mobili nella batch normalization come parametri (che non richiedono comunque il calcolo del gradiente), quindi non venivano esportati nel file e di conseguenza i risultati che ottenevo con il modello finale erano pessimi.
Il numero di epoche dopo il quale far terminare il training se l'accuratezza di validazione non incrementa è quindi un ulteriore iper-parametro della nostra rete (chiamato pazienza) che andrebbe validato, tuttavia ho preferito assegnare valore tre a tale iper-parametro in quanto, altrimenti, con valori più grandi, rischierei che l'esecuzione del training del nostro modello impieghi veramente troppo tempo per essere completata, in quanto rischio di aumentare sensibilmente il numero di epoche necessarie per completare l'allenamento del mio modello.

4.5 Struttura della rete

L'aspetto più delicato della model selection rimane però scegliere la struttura del modello più adatta al mio task. Essendo una scelta così delicata, ho deciso di prendere spunto da un'architettura già esistente, ossia la

VGGNet – VGG16 [6], implementata per la prima volta nel 2014, sopratutto per quanto riguarda la scelta dei valori di alcuni iper-parametri. Ho scelto quindi di usare in ogni modello che andrò a controllare in fase di validazione, per ogni livello convoluzionale della rete un kernel 3X3 con stride 1 e un pooling massimale con una finestra di dimensione 2X2, in questa maniera non avrò bisogno di applicare il padding alle immagini che rischierebbe di compromettere le prestazioni della mia rete.
Inoltre dovrò anche controllare se inserire nella rete il dropout e la batch normalization, e nel caso del dropout, devo anche decidere un valore per l'iper-parametro relativo alla probabilità di porre a zero uno degli elementi delle matrici in input.

4.6 Scelta delle reti da validare

Ho scelto di validare sette modelli differenti, che si distinguono tra loro per struttura o valore di alcuni iper-parametri. In particolare il primo è un modello molto semplice con tre livelli composti da un modulo di convoluzione, uno che applica la ReLu ed uno per il pooling, terminando quindi con un semplice Multy Layer Perceptron. Tale rete, partendo da immagini 32X32 con 3 canali in ingresso, arriva al layer fortemente connesso con matrici di 16 canali di dimensione 2X2, che vengono appiattite in un vettore di 64 elementi. Tale modello è utile solamente per osservare in maniera rapida che effettivamente la mia rete riesce ad apprendere dai dati in ingresso senza dover attendere il completamento di una rete di maggiori dimensioni, infatti mi aspetto che questa rete non riuscirà ad avere prestazioni paragonabili alle altre che introdurrò in seguito.
Per il resto delle reti ho deciso di prendere ancora spunto dall'architettura VGGNet – VGG16, pur riducendone il numero di livelli visto che l'architettura originale conteneva circa 140 milioni di parametri ed era pensata per immagini di dimensione 224x224. Infatti nel secondo modello da validare ho inserito sempre tre livelli composti però questa volta da due moduli convolutivi divisi da una ReLU e che terminano con un ultimo modulo di pooling massimale, tranne per l'ultimo livello che non presenta il pooling ma è connesso direttamente al livello fortemente connesso finale poiché ho già ridotto le mie immagini ad una dimensione 3X3. Il livello fortemente connesso invece presenta tre Multi Layer Perceptron, separati tra loro da ReLU, che ritornano infine i valori associati a ciascuna delle dieci classi. Quindi, partendo da immagini 32X32 con tre canali, arrivo, prima di appiattire le matrici, ad avere, per ciascuna immagine, 256 canali con matrici di dimensione 3x3.
Il resto delle reti mantiene sostanzialmente la stessa struttura ma ciascuna introduce gli altri moduli che abbiamo implementato in precedenza.
Nel terzo modello introduciamo infatti la batch normalizzation dopo la prima convoluzione presente in ciascun livello. Nel quarto modello invece introduciamo solo il dropout alla fine del secondo e del terzo livello e prima dell'ultimo MLP, con gli iper-parametri relativi alla probabilità di porre un pixel a zero impostati rispettivamente a 0.05, 0.1 e 0.1.
La quinta rete invece combina sia batch normalization che dropout, con gli stessi iper-Parametri e struttura visti per le reti precedenti.
Per tutti i modelli visti fin'ora ho scelto una dimensione di batch pari a 128 immagini. Per la sesta rete invece ho mantenuto invariata la struttura vista nel modello precedente ma ho incrementato la dimensione della batch a 256 immagini.
Nell'ultima rete invece ho aggiunto al precedente modello anche come layer inziale un dropout con iper-parametro della probabilità di oscurare un pixel a cui ho assegnato valore 0.02
Per ciascuna di queste reti ho impostato un limite massimo di 50 epoche, anche se tale numero di epoche non viene mai raggiunto in quanto l'early stopping interviene solitamente molto prima.

4.7 Risultati

La metrica di confronto tra i modelli è l'accuratezza ottenuta sull'insieme di validazione.
Come prevedibile, data la sua semplicità, la prima rete raggiunge appena un'accuratezza del 58%, tuttavia come possiamo vedere dal grafico riportato nella figura 3 il modello è riuscito effettivamente ad imparare e a generalizzare qualcosa anche se in minima parte.
Il secondo modello, ossia quello senza batch normalizzation e dropout raggiunge invece un'accuracy del 69% in validazione, mentre la terza rete con la batch normalizzation raggiunge il 75%. La quarta, con soltanto il dropout, invece ha prestazioni peggiori perché raggiunge il 72% di accuratezza. La rete completa con dimensione di batch impostata a 128 raggiunge invece il 77%, mentre quella con dimensione di batch di 256 circa il 76% di accuratezza in validazione. Come notiamo dai grafici 4 e 5, aumentando il numero di immagini nella batch l'accuratezza in validazione non cresce più in maniera "liscia" ma cresce in maniera più "irregolare" rispetto a prima, nonostante l'accuratezza finale rimanga abbastanza simile.
L'ultima rete invece non riesce a battere le prestazioni ottenute dalla quinta, che si conferma quindi la miglior scelta, anche se di poco rispetto alla sesta.

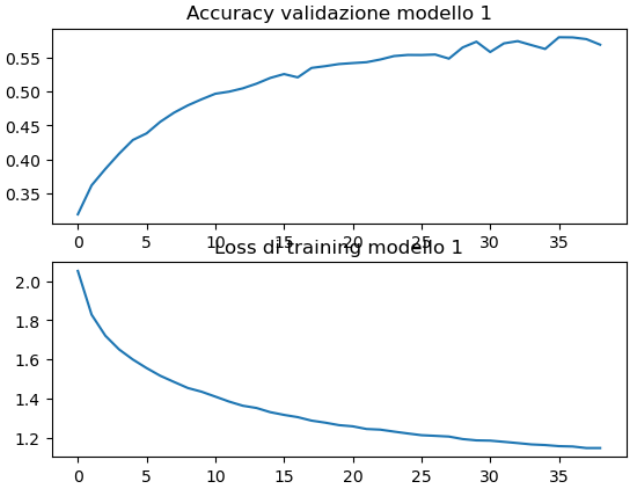

Figure 3: Andamento dell'accuracy sul validation set e della loss di training per la prima rete al passare delle epoche

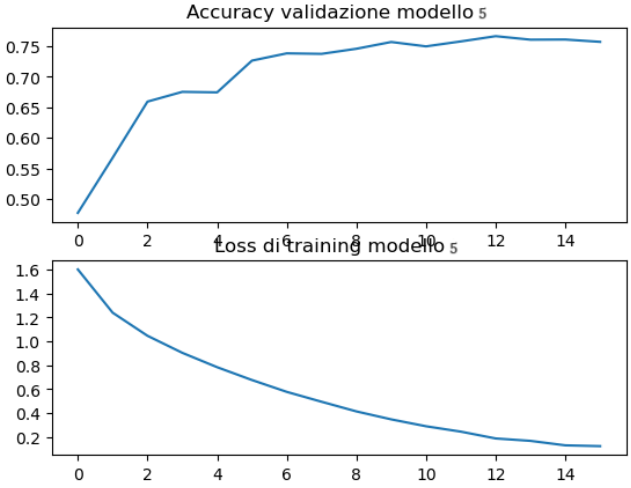

Figure 4: Andamento dell'accuratezza sul validation set e della loss di training per la quinta rete

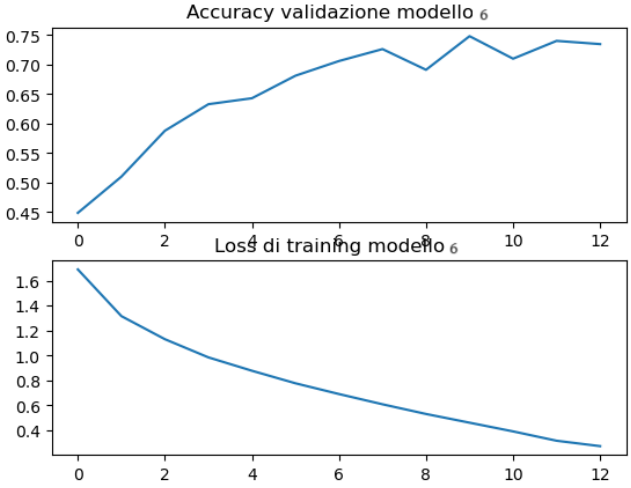

Figure 5: Andamento dell'accuratezza sul validation set e della loss di training per la sesta rete

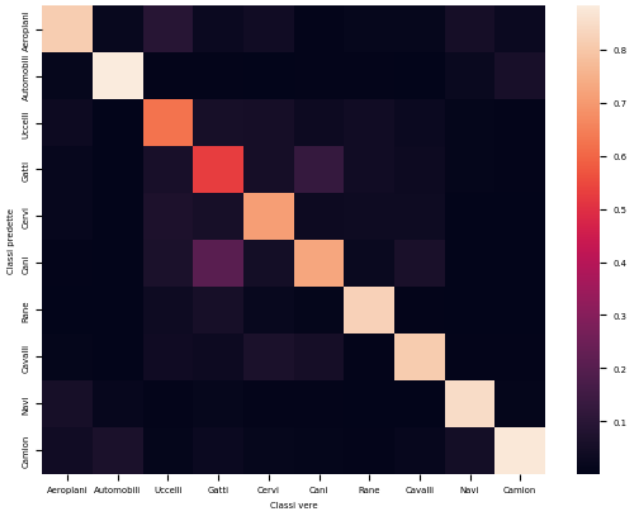

Figure 6: Heatmap dei risultati della rete sull'insieme di test

Avendo quindi scelto la quinta rete, essa sarà quella che andrò ad utilizzare nella prossima fase di model assessment per stimare l'errore di generalizzazione della rete.

5 Model Assessment

In quest'ultima fase andrò a verificare le prestazioni della mia rete sull'insieme di test che finora non mai considerato, stimando l'errore di generalizzazione attraverso la proporzioni di immagini classificate in maniera non corretta.

5.1 Prestazioni della rete sul test set

La rete scelta riesce quindi a classificare correttamente più del 76% dei dati di test, con un errore di generalizzazione di meno del 24%. Tale risultato è più che buono avendo implementato la rete "a mano" ed avendo impiegato reti non un numero di parametri non troppo elevato rispetto ad altre architetture usate attualmente. Il fatto che l'accuratezza sull'insieme di validazione e quella sull'insieme di test siano relativamente vicine è dovuto al grande numero di dati che ho a disposizione nei due insiemi, infatti tali insiemi sono composti da migliaia di immagini, anche se l'insieme di validazione è grande meno della metà di quello di test.

5.2 Commento ai risultati

Possiamo inoltre notare, riportato nella figura 6, l'heatmap che illustra in base allam gradazione di colore, per ogni classe, la proporzione di immagini correttamente classificate e quelle che invece sono state incorrettamente assegnate ad altre classi. Notiamo come le classi di immagini che la rete ha avuto più difficoltà a riconoscere sono i cani ed i gatti, che vengono spesso confusi dalla rete, forse per il fatto che sono entrambi animali domestici e quindi la rete potrebbe far affidamento, almeno in parte, anche sulla presenza di un ambiente domestico nelle immagini per classificare entrambe queste classi di immagini. Anche gli uccelli hanno creato un po' difficoltà alla rete, ma per il resto le immagini delle altre classi sembrano essere classificate in maniera più che soddisfacente.

6 Conclusioni

6.1 Struttura del progetto implementato

Le cose viste fin'ora sono implementate in quattro file nei quali implemento tutti i moduli ed i meccanismi visti nelle sezioni precedenti. In particolare nel file "moduli.py" ho implementato tutti i moduli come la convoluzione, il pooling e così via, nel file "modelli.py" ho invece costruito ciascuna rete che ho validato nella model selection, nel file "training_validation.py" ho implementato i metodi usati appunto per effettuare il training e la validazione di ciascuna rete dati in input i rispettivi dataset e la rete da allenare/testare insieme ai valori degli iper-parametri, mentre nel file "gestione_dati.py" ho implementato i metodi per importare, trasformare in tensori e separare le

immagini.
Oltre a questi quattro file ho implementato anche il file "main.py" che, se lanciato, esegue in automatico tutti i meccanismi di split dei tre dataset, normalizzazione dei dati, model selection e model assessment.

6.2 Tempo di esecuzione

Sul mio computer il tempo per completare il processo di training di tutte le reti e la fase di test della rete scelta impiega più di cinque ore, motivo per il quale ho scelto di creare un file a parte, chiamato "model_assessment.py" in cui alleno solo il modello scelto ignorando il resto dei modelli che ho dovuto allenare nella fase di model selection, in maniera da poter verificare i miei risultati senza dover ripetere tutti i passaggi di allenamento e validazione delle reti che invece sono implementati nel file "main.py".
Se il codice viene eseguito esclusivamente in CPU il tempo impiegato, almeno nel mio computer, quadruplica, per cui diventa quasi impossibile riuscire ad eseguire il codice in quanto impiegherebbe quasi venti ore per allenare e selezionare ciascuna rete e per testare quella selezionata.

6.3 Riproducibilità

All'inizio dei file "main.py" e "model_assessment.py" ho impostato dei semi per la riproducibilità, in tal modo posso lanciare un esecuzione avendo la sicurezza che l'insieme di test rimanga sempre lo stesso e che quindi possa accedere a tali dati solo per fare assessment della rete.
Nonostante ciò l'esecuzione del mio codice non è deterministica, infatti posso avere risultati leggermente diversi in ciascuna esecuzione, tuttavia in generale ciò non inficia sulle scelte e sulle prestazioni della mia rete, che rimangono sempre abbastanza simili a quelle che ho indicato in questa relazione. L'unico problema è che essendo la quinta e la sesta rete molto simili tra loro, è possibile che in model selection venga scelta la sesta invece della quinta rete come migliore, tuttavia questo accade abbastanza raramente.

6.4 Eseguire il codice

Come accennato in precedenza, è possibile eseguire sia il file "main.py" per riprodurre tutti i procedimenti visti come lo splitting dei dati, la normalizzazione, la model selection e l'assessment, che il file "model_assessment.py" per riprodurre tale operazione più velocemente senza dover ripetere la model selection.
Purtroppo eseguire il codice da terminale non permette di visualizzare i grafici delle accuracy di validazione e della loss di training al passare delle epoche e nemmeno la matrice di confusione sopra riportata. Per poterli visualizzare è necessario quindi utilizzare un notebook Jupyter, come quello che ho incluso nella cartella del progetto rinominato "Visual.ipynb". Purtroppo il grafico della loss di training e dell'accuratezza di validazione dell'ultimo modello vengono sovrapposti, però nel notebook sono riuscito a riportare singolarmente la matrice di confusione generata durante il model assessment.

6.5 Problematiche e commenti finali

A causa dei mezzi a mia disposizione e dei limiti temporali che non mi consentono di eseguire il codice per troppe ore di fila, visto che mentre alleno le reti non posso usare nessun altro programma sul computer in quanto tale processo satura quasi completamente la memoria RAM del mio computer, non ho potuto implementare reti troppo complesse e non ho potuto validare ciascun iper-parametro, come visto in precedenza, né tanto meno sono stato in grado di sfruttare la cross-validation per validare i miei modelli in maniera più adeguata.
Nonostante ciò sono comunque soddisfatto dei risultati ottenuti anche se in rete si trovano modelli che riescono a raggiungere prestazioni nettamente superiori alla mia, tuttavia tali modelli sono spesso allenati su macchine ben più potenti della mia e in parte anche pre-allenati o che comunque utilizzano tecniche come la data augmentation che ho preferito non implementare per semplicità.

References

[1] A. Krizhevsky. "Cifar-10 and cifar-100 datasets." (2009), [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[2] "Conv2d-pytorch 2.1 documentation." (2023), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

[3] "Linear-pytorch 2.1 documentation." (2023), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html.

[4] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[5] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, *On empirical comparisons of optimizers for deep learning*, 2020. arXiv: 1910.05446 [cs.LG].

[6] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].