Relazione progetto Computational Learning
Convolutional Neural Network

Studente: Massimiliano Sirgiovanni
Matricola: 7077251
Email: massimiliano.sirgiovanni@stud.unifi.it

L'obiettivo di questo progetto è la **costruzione di un modello per la creazione di Convolutional Neural Network** che sia il più generale possibile.
Dunque, tale **generalizzazione** dovrà prendere forma a livello di **iperparametri**, di **dimensioni dell'input** e **dell'output**, in modo tale che questi fattori possano essere impostati, a piacimento, durante l'inizializzazione della rete neurale.
Inoltre, la rete verrà **testata**, *seguendo i principi visti a lezione*, su un **dataset di immagini** reali, *CIFAR10*.

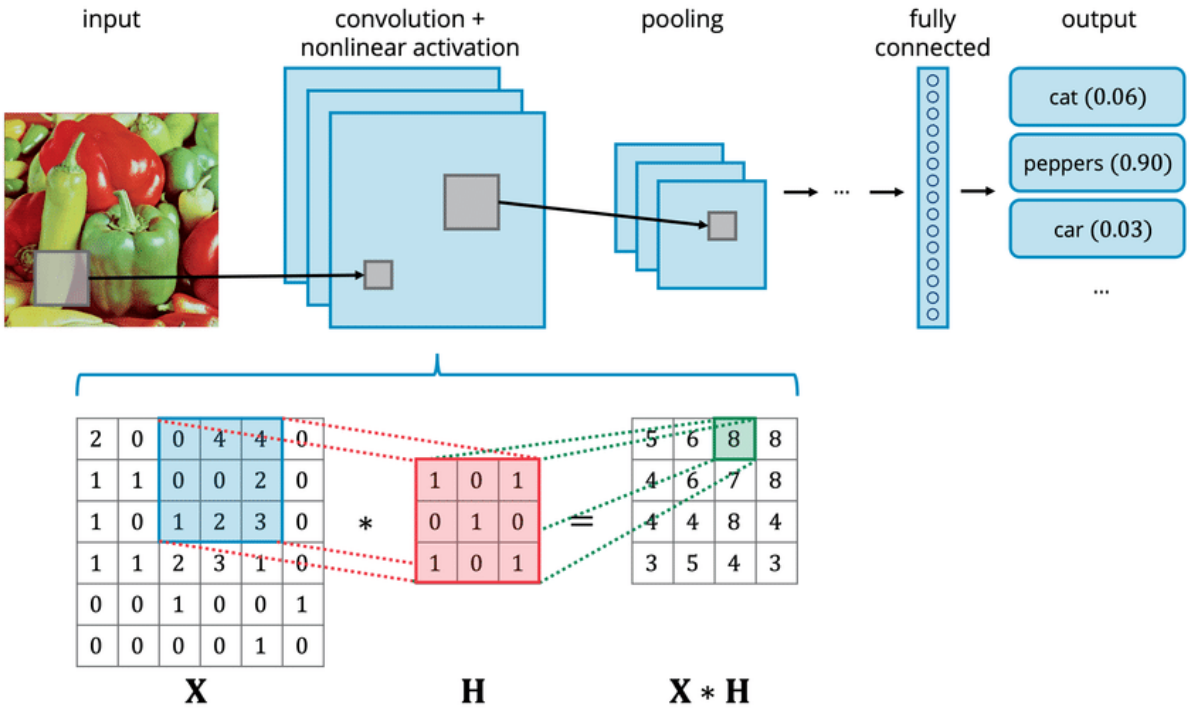
La sezione relativa alla configurazione dell'ambiente per l'esecuzione ed alle librerie necessarie si trova nella sezione finale della relazione, tuttavia è raggiungibile anche tramite il seguente link → ***Guida all'esecuzione***

Struttura di una Convolutional Neural Network

Una Convolutional Neural Network è una rete formata da, almeno, uno o più **Convolutional Layer** e da uno o più **Fully Connected Layer**.
Nelle righe seguenti, verrà fatto un veloce ripasso sulla struttura di una Convolutional Neural Network come passo introduttivo alla sua implementazione.
Un **Convolutional Layer** è formato, a sua volta, da:

1. **Convolutional Filter**;
2. Applicazione di una **Feature map**;
3. Operatore di **Pooling**.

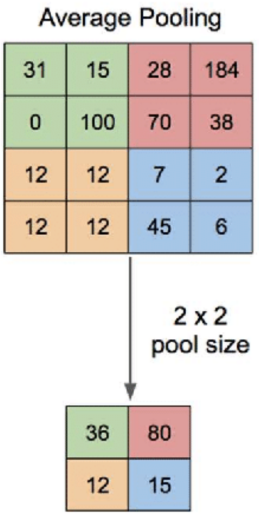
Il **Convolutional Filter** prende come input un tensore *(che rappresenta un'immagine o l'output di un altro Layer Convoluzionale)* e restituisce una trasformazione di quest'ultimo. Più nello specifico, il Convolutional Filter applica dei pesi, *denominati filtro*, a varie zone dell'input.
Il **filtro** può essere visto come una **finestra che si muove sull'input**, di dimensioni fissate e che si sposta di un numero di celle fissate.
La grandezza del filtro, **kernel size,** ed il numero di celle da saltare, **stride,** per muoversi sono degli **iperparametri** della CNN.



Per **evitare** che vi sia **perdita di informazione**, durante l'applicazione di un filtro, è possibile aggiungere del **padding** all'immagine.
Il padding non è altro che una trasformazione sull'input, che comporta l'aggiunta di un determinato numero di celle attorno all'immagine.
Indipendentemente dal numero di canali posseduti dall'input, dopo l'applicazione di un filtro, questi vengono fusi tra di loro.
Tuttavia, **i filtri** che possono essere **applicati all'input possono essere anche più di uno**. Dal numero di filtri applicati all'input dipenderà il numero di canali dell'output.

Come secondo passo, *all'interno di un Convolutional Layer*, è necessaria l'applicazione di un **operatore non lineare**, elemento per elemento, all'output del Convolutional Filter.
Si effettua tale operazione per gestire anche situazioni in cui il problema da risolvere dovesse essere **non linearmente separabile**, situazione piuttosto comune, e, dunque, non può essere risolto con la sola applicazione di un modello lineare, *come il Convolutional Filter*.

Infine, all'interno di un Convolutional Layer, può essere applicato un operatore di **Pooling**. Questo ha l'obiettivo di **ridurre** ulteriormente **le dimensioni dell'input**. Come accade nel Convolutional Filter, viene applicato un **filtro** *(rappresentabile come una finestra)* all'input. Ma, *al contrario dell'operatore precedente*, il filtro **consiste nell'applicazione di una funzione** e non nell'utilizzo di pesi. Questa funzione prende in **input un numero indefinito di elementi**, variano a seconda della grandezza del filtro, e **restituisce un singolo valore**.



Le funzioni più usate sono: il **massimo**, la **media**, la **selezione randomica** e la **norma L2**.

D'altra parte, il **Fully Connected Layer** è, solitamente, formato da un **operatore di compressione**, *detto Flatten*, e da un **operatore lineare** che restituisce la probabilità di appartenenza alle varie classi su cui operiamo.

Implementazione della rete - Forward

Una volta descritto teoricamente, in breve, il funzionamento di una Convolutional Neural Network e la sua struttura, è possibile capire meglio la sua implementazione.
La realizzazione di tale modello è stata svolta seguendo l'ordine posto in via teorica, sviluppando le varie componenti della CNN separatamente, per poi collegarle in una seconda fase.
Per ogni componente è stata realizzata una diversa **classe** *(o usata la corrispondente classe nel pacchetto pytorch)*.
Ognuna di queste è una sottoclasse della classe astratta denominata ***"Model"***.
Tale superclasse contiene esclusivamente **tre metodi astratti**, che dunque dovranno essere implementati nelle sottoclassi:

- ***__call__(self, *args)*** ← *Usato per eseguire le operazioni di forward*;
- ***parameters(self)*** ← *Restituisce i parametri usati dal modello*;
- ***__str__(self)*** ← *Permette di ottenere una stampa leggibile del modello*.

Il principio alla base della scelta di questi metodi è stato il tentare di rendere compatibili le classi realizzate a mano e quelle importate da pytorch, in modo che non vi fossero particolari problemi nella loro combinazione per la rete neurale da allenare.

Convolutional Layer

> Convolutional Filter

É stata creata la classe ***ConvolutionalFilter*** per la rappresentazione dell'operatore omonimo.
Nel costruttore vengono inizializzati diversi **attributi** per la classe:

- **kernel_size** *(dato in input al costruttore)*;
- **stride** *(dato in input al costruttore)*;
- **numChannels** *(dato in input al costruttore, il valore di default è 3);*
- **numFilters** *(dato in input al costruttore, il valore di default è 1)*;
- **filters** *(inizializzati in maniera casuale)*;
- **namePadding** *(dato in input al costruttore, utile solo per al stampa)*;
- **dimPadding** *(dato in input al costruttore, valore di default: 'auto')*;
- **paddingFunction** *(dato in input al costruttore)*.

I **filtri corrispondono** alla rappresentazione della **matrice dei pesi**, che l'operatore di convoluzione deve applicare all'input. La loro inizializzazione avviene tramite la seguente istruzione:

```
self.filters = nn.Parameter(th.randn(numFilters, numChannels,
self.kernel_size[0], self.kernel_size[1]))
```

La funzione **randn,** di pytorch, permette di creare un **tensore contenente elementi randomici**, campionati da una distribuzione con media zero e varianza uguale ad uno. Come input, tale funzione, richiede le **dimensioni del tensore**.
Il filtro viene condiviso da tutte le immagini contenute nel dataset.
Dunque, le **dimensioni della matrice dei pesi** dipendono dalla **kernel size**, data in input alla rete convoluzionale, dal **numero dei canali** dell'input e dal **numero di filtri** che si desidera applicare all'input, non dal numero di record nel dataset.

All'interno del metodo ***__call__()*** viene gestito l'utilizzo del **padding**. Nello specifico, viene applicata una **trasformazione sull'input**, aggiungendo, *se richiesto*, un certo quantitativo fissato di **padding**. Le dimensioni del padding possono essere fornite in input, all'interno di una tupla, oppure si può utilizzare una procedura che calcola il quantitativo di padding necessario a seconda dei parametri utilizzati.
Per questo secondo caso, la formula usata è la seguente:

$$P = \frac{K-1}{2}$$

dove *K* rappresenta la kernel size.
I Convolutional Filter possono prevedere anche **filtri di dimensioni non quadrate** e tale possibilità è stata preventivata nel modello implementato. Infatti, l'attributo ***kernel_size*** prende come input una tupla di due elementi.
Per quanto concerne il **calcolo della quantità padding**, nel caso di **filtri non quadrati**, viene calcolato il quantitativo di padding per entrambe le dimensioni, con la medesima formula vista sopra:

$$P.W = \frac{K.W-1}{2} \qquad P.H = \frac{K.W-1}{2}$$

dove *K.W* è la larghezza del filtro e *K.H* è l'altezza del filtro.
Ovviamente, tale formula verrà utilizzata solo se non vengono specificate dall'utente le dimensioni del padding.
Sono stati implementate quattro diverse **opzioni per il padding**, *viste a lezione*:

- **Zero** Padding;
- **Same** Padding;
- **Casual** Padding;
- **Valid** Padding *(Non viene aggiunto padding all'immagine)*.

La scelta del tipo è rimandata all'utente nel momento della creazione della CNN, qualora non venisse specificato, il tipo di **default è Zero Padding**.

La funzione principale, che implementa la forward del Convolutional Filter è denominata ***__forward_fun__*** e prende in input il dataset delle immagini, *sotto forma di tensore*, su cui si vuole applicare il filtro.
L'output avrà tante istanze quante ve ne sono in input, un numero di canali pari al numero di filtri richiesti dall'utente ed, inoltre, altezza e larghezza, calcolate utilizzando le formule:

$$W' = \frac{W+2P-K}{S} + 1; \quad H' = \frac{H+2P-K}{S} + 1;$$

dove *W* ed *H* sono rispettivamente larghezza ed altezza dell'input, *P* è il padding aggiunto all'immagine, *K* è la kernel size ed *S* è il valore della stride.
Una volta calcolate queste ultime due dimensioni, è possibile effettuare l'operazione di **sliding della finestra**.
Questa operazione è stata **vettorizzata** grazie all'utilizzo di due metodi di pytorch, **unfold e fold**.
Il primo prende in input un tensore, la kernel size della finestra e la stride da applicare e restituisce in output un tensore che raggruppa gli elementi seguendo le direttive dello sliding.
Il secondo effettua l'operazione inversa, partendo da un tensore raggruppato consente di ritrasformarlo seguendo le direttive date in input.
Dunque, per poter effettuare le operazioni richieste, si è effettuata la **trasformazione** descritta al **database in input** ed ai **pesi**, di seguito si può osservare il caso dell'applicazione sui pesi:

```
unfoldedF = nn.functional.unfold(self.filters, self.kernel_size,
stride=self.stride)
```

Dato che la funzione **unfold fonde i canali in uno solo**, mantenendo però le informazioni intatte, sono state utilizzate le funzioni **split e cat** per risuddividere i canali del tensore, in questo modo:

```
unfoldedF = th.cat(th.split(unfoldedW.unsqueeze(1),
self.kernel_size[0]*self.kernel_size[1], 2), 1).unsqueeze(0)
```

Una volta fatta la medesima operazione per l'input, è sufficiente **moltiplicare i due tensori** e **sommare tutti gli elementi contenuti nella finestra**, ottenendo un singolo valore, **e sommare questi valori sui canali**.
Per ottenere il tensore di output è sufficiente usare la funzione fold, utilizzando le dimensioni calcolate precedentemente come output_size:

```
conv_output = nn.functional.fold(conv_output, (dim_new[0], dim_new[1]), (1,
1))
```

A questo punto, il tensore contenuto della variabile conv_output corrisponderà al valore di uscita del Convolutional Filter.
La funzione ***parameters()*** restituisce i pesi dell'operatore convoluzionale all'interno di una lista.

Infine, la funzione ***__str__()*** definisce una stringa per la rappresentazione di una specifica istanza del Convolutional Filter. Tale rappresentazione contiene i valori dei parametri dati in input dall'utente.

> Operatore non lineare

Come consentito dalle specifiche di progetto, è stato utilizzato un oggetto della **classe ReLu(),** *importata dal pacchetto torch.nn*.
A questo oggetto viene, semplicemente, dato in input il risultato del Convolutional Filter, a cui viene applicata la **feature map** accennata nella teoria. La feature map viene applicata **elemento per elemento**, dunque l'output dell'oggetto ReLu(), una volta richiamato il metodo ***__call__()***, sarà sempre un tensore delle medesime dimensioni del suo input.

> Pooling

L'applicazione della funzione di Pooling, d'altra parte, è stata implementata a mano, *come il Convolutional Filter*. Anche in questo caso è stata creata una **classe**, *sottoclasse di Model*, denominata semplicemente ***"Pooling"***.
Gli attributi della classe sono i seguenti:

- **kernel_size** *(dato in input al costruttore)*;
- **stride** *(dato in input al costruttore)*;
- **nameFunction** *(dato in input al costruttore, utile solo per al stampa)*;
- **poolingFunction** *(dato in input al costruttore)*;
- **namePadding** *(dato in input al costruttore, utile solo per al stampa)*;
- **dimPadding** *(dato in input al costruttore, valore di default: 'auto')*;
- **paddingFunction** *(dato in input al costruttore)*.

Il funzionamento dell'operatore di pooling è molto simile al filtro convoluzionale, tuttavia non vi sono pesi, ma, di contro, **viene applicata una funzione che**, ***dati in input diversi valori*****, ne restituisce uno solo**.
La **trasformazione** da fare sull'input è la medesima di quella vista per l'operatore convoluzionale, tramite l'utilizzo delle funzioni ***unfold, split e cat***.
Si richiama poi la funzione di pooling selezionata dall'utente, le opzioni implementate sono le seguenti:

- **Max Pooling**;
- **Average Pooling**;
- **Random Pooling**;
- **L2-Norm Pooling**.

Nella maggior parte dei casi si tratta di **applicare all'input la funzione di pytorch** corrispondente all'operazione richiesta sulla terza dimensione, che rappresenta le sottomatrici del tensore individuate dalla finestra.
Per quanto riguarda il **random pooling**, d'altra parte, si è costruito un **tensore di indici**, per selezionare delle posizioni randomiche da scegliere. Successivamente, si

è ottenuto il tensore in output applicando la funzione *"th.take_along_dim"* al tensore in input ed al tensore contenente gli indici, solo per la terza dimensione.
L'utente può selezionare la funzione che più gradisce nel momento di istanziamento dell'oggetto.

La funzione ***parameters()***, in questo caso, restituisce una **lista vuota**, poiché non vi sono parametri da aggiornare nel pooling.
Il metodo ***__str__()***, d'altra parte, restituisce una rappresentazione, sotto forma di stringa, dell'oggetto che contiene i valori, assegnati dall'utente, ai vari parametri della classe.

Fully Connected Layer

Il Fully Connected Layer, *nella sua forma più semplice*, applica all'output di un Convolutional Layer due operatori:

- **Flatten**;
- **Linear**;

come anticipato nella sezione teorica.
L'**operazione di flattening** è stata realizzata tramite l'ausilio di una **classe di pytorch**, denominata ***Flatten()***.
D'altra parte, l'operatore lineare è stato implementato da zero. Come per gli altri modelli, è stata realizzata una sottoclasse di Model, che sovrascrive i suoi tre metodi astratti.
Gli **attributi** della nuova classe, chiamata ***Linear***, sono i seguenti:

- ***input_feature;***
- ***output_feature;***
- ***bias;***
- ***weight;***
- ***paramBias.***

I primi tre attributi vengono passati al **costruttore** della classe come input.
Input_feature rappresenta la **grandezza dell'input** mentre **output_feature quella dell'output**. La **variabile bias è un booleano**, *di default posto a False*, che comunica al modello se si desidera inserire una b diversa da zero nell'equazione:

$$y = x*W' + b$$

Gli attributi weight e paramBias sono i parametri del modello e rappresentano, nella formula di sopra, le variabili W e b. Se bias è posto a False, paramBias è uguale a zero.
Dato che gli unici due parametri restituiti dal modello sono quelli sopracitati, la funzione ***parameters()*** restituisce, semplicemente, una lista contenente questi due parametri o una lista contenente solo weight, se bias è posto a False.

Per l'applicazione della ***forward***, semplicemente si applica la formula dell'operatore lineare all'input, con i pesi ed il bias precedentemente definiti:

```
output = input.matmul(self.weight.t())+ self.bias
```

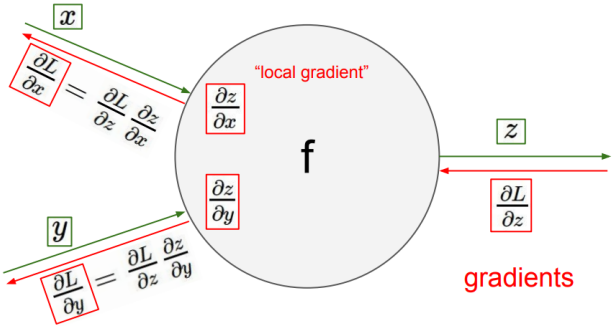
La funzione matmul serve per effettuare il prodotto tra la **matrice di input e la matrice dei pesi**. Ovviamente, l'operatore lineare viene applicato a tutte le immagini presenti nel database.

> Costruzione di una rete

Per comporre i modelli, è stata definita un'altra sottoclasse di Model, denominata ***Sequential***, che è stata realizzata, sostanzialmente, seguendo il design pattern Composite. Questa classe possiede come attributi privati la **lista dei modelli** di cui è composta ed un flag, ***long_print***, utile solo per la stampa.
Un oggetto di questa classe può contenere sia istanze di sottoclassi di Model, ma anche modelli importati da pytorch.
Infatti, nella sottoclasse Sequential sono stati implementati i tre metodi astratti della classe Model più alcuni metodi per il **monitoraggio delle epoche** e per la gestione della procedura di **early stopping** *(le cui ragioni saranno illustrate nelle successive sezioni della relazione)*.
Il metodo **parameters()** itera su tutti i modelli contenuti nell'oggetto di classe Sequential, sui quali applica, a sua volta, il metodo parameters(). Tutti i parametri dei modelli vengono poi inseriti in un'unica lista che sarà l'output della funzione.
Il metodo ***__call__()*** lavora in maniera simile, poiché itera su tutti i modelli e richiama lo stesso metodo __call__() su di essi. Differentemente dal metodo precedente, però, **l'output di ogni modello viene poi utilizzato come input di quello successivo**, realizzando così la forward.
Per quanto concerne la creazione di una stringa per la presentazione dell'oggetto, vi sono **due modalità di stampa**.
Qualora il **flag long_print** sia **diverso da uno**, *di default è posto a zero,* durante la stampa l'oggetto di classe Sequential semplicemente richiamerà il metodo ***__str__()*** sui modelli contenuti al suo interno, concatenando insieme i risultati.
D'altra parte, se il flag viene posto a **uno**, allora, prima di ottenere le stampe dei modelli sottostanti, viene stampato il **numero di epoche** per cui il modello è già stato allenato, l'epoca in cui l'**accuratezza nel Validation Set** è risultata **massima** ed il valore di tale accuratezza.
Inoltre, vengono mantenute delle liste per l'errore in training ed in validation, utili per effettuare il plotting dell'errore ed osservare il suo comportamento tra le epoche.
Viene anche salvato il **modello migliore** trovato fino a quel momento *(con migliore accuratezza in Validation),* il quale dovrà poi essere utilizzato come modello finale, per predire le immagini.

Implementazione della rete - Backward

Per la realizzazione della backward nella Convolutional Neural Network implementata è stato possibile l'utilizzo delle librerie, disponibili in rete, per la gestione dei parametri della rete.

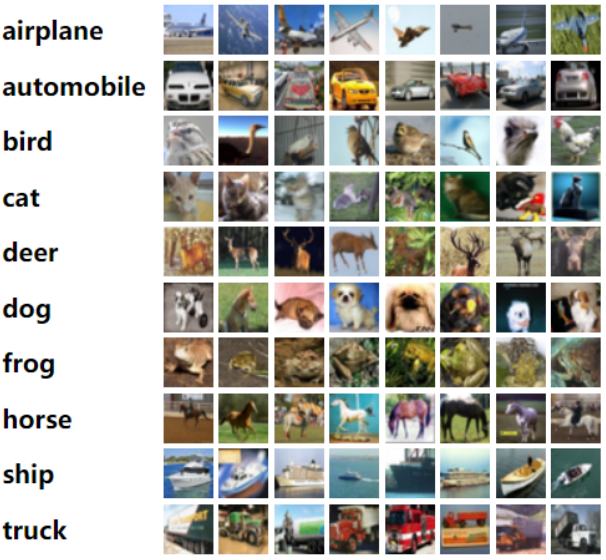


Inoltre, per l'**aggiornamento dei parametri** è stata utilizzata la **libreria optim**, con la classe ***Adam()***, che prende come input i parametri del modello che si desidera allenare.
Questa classe implementa l'algoritmo Adam, che **ottimizza l'aggiornamento dei parametri** in modelli di machine learning.

Dunque, tramite la funzione ***backward()*** sono stati calcolati i gradienti dei parametri utilizzati dalla rete.
Successivamente, per l'aggiornamento effettivo, si può usare la funzione ***step()*** sull'oggetto di classe optim.Adam().

Struttura del dataset

Il dataset utilizzato per l'apprendimento del modello è **CIFAR10**. Questo contiene **50000 immagini a colori** di **dimensione 32x32**.
In realtà, il dataset conterrebbe 60000 immagini, di cui 10000 adibite al testing. Tuttavia, dato che è condizione del progetto il fatto di non poter usare sistemi che dividono automaticamente il dataset, si è deciso di usare solo il training set per gli scopi del progetto.
Nel dataset vi sono **dieci classi** distinte di immagini, rappresentanti mezzi di trasporto o animali.
Più nello specifico, le classi nel dataset sono osservabili nella figura di fianco.



Le classi sono tra loro **mutualmente esclusive**, ovvero ogni immagine appartiene ad una e una sola classe.
Il dataset è stato caricato direttamente dal pacchetto ***torchvision.datasets***.

Prima di lavorare materialmente sulle immagini, come è buona prassi, si è applicata una **normalizzazione** a queste ultime.

Come visto nella teoria, vi sono diversi tipi di normalizzazione. Per il progetto si è scelta la normalizzazione più usata in rete, la ***normalizzazione min-max***.
In questo caso, **ogni pixel è rappresentato da un numero** compreso nell'intervallo ***[0, 1]***. Per effettuare questa normalizzazione sul nostro dataset è stato sufficiente eseguire la seguente istruzione:

```
X = th.tensor(trainset.data/255)
```

Il tensore è poi stato ridimensionato per fare in modo che abbia le dimensioni corrette *(sono state usate le stesse dimensioni di pytorch)*:

[50000, 3, 32, 32]

Metodo di apprendimento

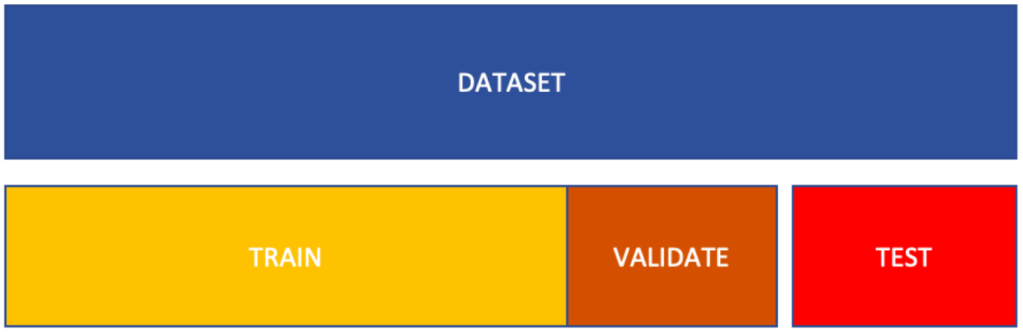
Per l'allenamento dei modelli, è stata usata la funzione ***trainCNN***, *definita a lezione*.
Questa funzione, prende in input il **modello da allenare**, **l'oggetto di tipo optim.Adam** per l'aggiornamento dei parametri, il **training set**, il **validation set** ed il **numero di epoche** che si intende effettuare.
Dunque, il training set viene diviso in **batch**, di dimensione fissata, ed ad ogni batch viene applicato il modello.
Durante una **singola epoca**, si effettua un'iterazione per batch, nelle quali vengono aggiornati i parametri del modello, secondo il valore riportato dalla loss function.
Infine, si applica il modello al **validation set** per osservare l'accuratezza ottenuta su quest'ultimo dal modello.
Viene, poi, stampato a schermo il valore della loss function per tutte le bach e l'accuratezza del modello sul validation set.
Si ripetono tante volte quante sono le epoche richieste.

Al metodo descritto, sono state fatte delle **modifiche**, *rispetto a quello visto a lezione*, dettate dai **considerevoli tempi di esecuzione** per l'allenamento dei modelli *(in alcuni casi si arriva sull'ordine delle ore per epoca)*.
Infatti, in caso di spegnimento dell'elaboratore o di malfunzionamento, per evitare di perdere tutti gli aggiornamenti fatti, **l'istanza del modello viene salvata su un file esterno**, ogni volta che termina un'epoca.
Per salvare le istanze è stata usata una libreria di Python, denominata ***pickle***, che consente proprio di salvare e caricare variabili ed oggetti di Python su file esterni, in modo tale da conservare tali informazioni in maniera non volatile.
Un'altra modifica, *derivata anch'essa dai tempi di esecuzione poco permissivi*, consiste nel **salvataggio del numero di epoche all'interno del modello** *(come anticipato in riferimento alla classe Sequential)* e nella possibilità di ripartire, in una seconda esecuzione, con il training dall'epoca completata nell'ultima esecuzione.
Per fare questo, si è semplicemente assegnato un limite inferiore, maggiore o uguale di zero, al ciclo for che itera sulle epoche:

```
for e in range(my_model.getTrainedEpochs(), n_epochs):
```

Inoltre, sono state effettuate delle modifiche per aggiungere la funzionalità dell'**Early Stopping** al training, che consente di **interrompere il training** per **evitare l'overfitting** della rete. Innanzitutto, è stato aggiunto come argomento della funzione trainCNN la ***patience***, con valore di default pari a dieci. La patience non è altro che il **numero di epoche** che devono essere eseguite **prima di terminare il training** di un modello.
Per tenere traccia del numero di epoche trascorse durante la procedura di Early Stopping, è stata usata una variabile, ***indexPatient***. Questa, viene decrementata ogni qual volta l'accuratezza di un'epoca non è la migliore trovata fino ad ora. Questa variabile viene resettata, *ovvero riportata al valore della patience*, se si trova un valore di accuratezza in validation che consente di uscire dalla procedura di Early Stopping.
Se, consecutivamente, si trovano un numero di modelli pari al valore di patience che si comportano nel secondo modo, allora il training si ferma e viene restituito il modello migliore trovato finora.
Il modello trovato **prima dell'avvio dell'Early Stopping** viene messo da parte e salvato all'interno dell'oggetto di classe Sequential, per poter essere riutilizzato in seguito, come modello finale, qualora venisse interrotto il training per la procedura di Early Stopping.
Per **riprendere la procedura di Early Stopping** anche qualora l'esecuzione dovesse essere interrotta, sono stati **salvati**, all'interno della classe Sequential, il numero di **epoche eseguite**, i valori di **accuratezza** e **training error** ottenuti durante la corrente procedura di Early Stopping.

Alla funzione di training è stato dato in input il dataset importato, *CIFAR10*, diviso secondo il metodo **holdout**.
Si è utilizzato un approccio di tipo holdout piuttosto che la cross validation.
Dunque, dal dataset di partenza sono stati ricavati il **training set**, con grandezza pari al **60% del dataset originale**, il **validation set ed il test set**, entrambi con una grandezza del **20% rispetto al dataset originale**.
Successivamente, è stata costruita una permutazione randomica degli indici, tramite una funzione di pytorch, *randperm()*.
Il procedimento utilizzato è simile a quello visto a lezione, con l'aggiunta della divisione per il test set.



Model selection

Per Model Selection si intende la **scelta del modello** rispetto ai vari iperparametri ed alle varie caratteristiche che questo può avere.
Come anticipato, si è utilizzato un approccio di tipo **holdout**, dunque, si utilizzerà il **validation set per la model selection**, infine, per la **model assessment verrà utilizzato il test set**.
Nel caso delle CNN, le possibilità da studiare e sperimentare sono veramente tante. Infatti, un ruolo fondamentale spetta alla **scelta degli iperparametri**, ovvero la ***kernel size*** e la ***stride*** sia per l'operatore Convoluzionale che per il Pooling. Inoltre, è possibile selezionare diverse **funzioni per il Pooling** e diverse **tecniche di Padding**.
D'altra parte, è anche possibile **concatenare più Layer Convoluzionali**, ottenendo, solitamente, un modello più accurato. Ovviamente, più è complessa la rete e maggiore è il lavoro da eseguire per effettuare una Model Selection appropriata.
Per ottenere dei risultati interessanti ed appropriati, si è scelto di effettuare model selection su reti aventi **tre Layer Convoluzionali**.
Si è deciso di effettuare della **model selection sulla kernel size e sulla stride del primo Layer Convoluzionale**.
Per quanto concerne la scelta della **funzione di Pooling**, *secondo alcune ricerche effettuate online*, le due funzioni più usate ed efficaci sono *"max"* e *"average"*.
Tra le due, è **preferibile** la funzione che restituisce il **massimo**, poiché **conserva le caratteristiche più importanti della feature map**.
Effettuando altre ricerche su internet, come intuibile, è venuto fuori che l'**utilizzo del padding è da preferire** all'allenamento diretto sull'immagine, senza trasformazioni. Questo perché, ovviamente, consente di **utilizzare kernel di diverse dimensioni** senza doversi preoccupare delle dimensioni dell'input. Altri vantaggi del padding sono: permette di **realizzare reti più profonde** *(che però non verranno affrontate in questo caso)* e aumenta le performance mantenendo l'informazione presente ai bordi dell'immagine.
Di seguito, dalla teoria, si è a conoscenza del fatto che lo **zero padding non è adatto** nel momento in cui viene utilizzata la funzione max come pooling.
Dunque, rimangono come scelte il Casual Padding ed il Same Padding. Tra i due si è **preferito usare il Same Padding**, poiché rappresenta caratteristiche dell'immagine stessa e non introduce ulteriore randomicità nella rete.
Fissate queste caratteristiche della rete, è possibile procedere alla model selection sul primo Convolutional Filter.
Come anticipato, si è scelto di fare Model Selection sulla kernel size e la stride, considerandole dipendenti tra loro. Sono stati dunque selezionati **quattro valori** ragionevoli **di kernel size e cinque di stride.** Sono state allenate assieme solo **kernel size che fossero minori o uguali della stride considerata**.
Inoltre, sono state utilizzate solo **kernel size e stride quadrate**, considerato che le immagini in input sono quadrate la cosa è parsa piuttosto ragionevole.

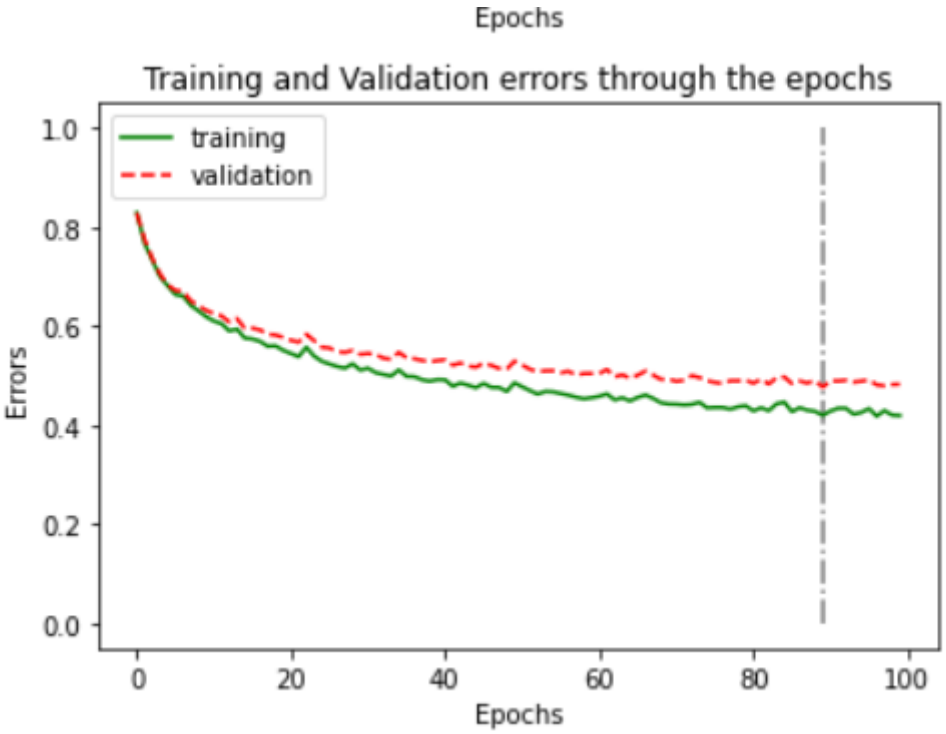
I valori scelti sono:

Kernel Size	*2x2*	*3x3*	*4x4*	*5x5*	
Stride	*1x1*	*2x2*	*3x3*	*4x4*	*5x5*

Considerato il vincolo posto, i modelli allenati saranno:

- **cinque** per il modello con **kernel size = 5x5**;
- **quattro** per il modello con **kernel size = 4x4**;
- **tre** per il modello con **kernel size = 3x3**;
- **due** per il modello con **kernel size = 2x2**.

Per un totale di **quattordici modelli**.
Per il numero di epoche massimo da utilizzare per allenare questi modelli, si è deciso di dare un'occhiata al grafico di alcuni modelli già allenati per diverse epoche:



Si può osservare che la curva inizia a stabilizzarsi intorno all'ottantesima epoca epoca *(questo comportamento è stato verificato su diversi modelli)*.
Dunque, si è deciso di settare il **numero massimo di epoche** raggiungibili dai modelli a **cento**, *ma con un ulteriore vincolo*.
Infatti, le cento epoche rappresentano solo un limite massimo per i modelli, settato per fare in modo che l'esecuzione della model selection non continui per troppo tempo. Tuttavia, ci si aspetta che **la maggior parte dei modelli si fermi molto prima** di raggiungere questo numero, poiché è stata settata la **patience per il training a cinque epoche**, ovvero, la procedura di **Early Stopping** durerà solo per cinque epoche.

In questo modo, si dovrebbe riuscire a trovare il momento in cui la validation accuracy del modello tende a stabilizzarsi.

In una situazione reale, sarebbe necessario effettuare una **model selection su più seed** per poi fare una media dei risultati e selezionare il modello migliore.
Per ragioni di tempistica non sono stati usati più seed, ma si è **costruito il codice,** per la model selection, **tenendo conto di questa possibilità**, permettendo di testare più seed semplicemente inserendoli in un'apposita lista.
Il cambio del seed influenza solo i modelli ed i loro parametri, non la suddivisione del dataset in training, validation e test set, per la quale vi è un seed globale.
Per la prova effettuata si è utilizzato il ***seed 12446***.
I modelli sono stati allenati per il seguente **numero di epoche**:

K\S	**1x1**	**2x2**	**3x3**	**4x4**	**5x5**
2x2	*41*	*66*	*X*	*X*	*X*
3x3	*48*	*35*	*71*	*X*	*X*
4x4	*39*	*58*	*53*	*61*	*X*
5x5	*25*	*81*	*71*	*46*	*51*

Le celle della tabella che contengono una X indicano i modelli che non sono stati allenati, poiché la stride risultava essere superiore alla kernel size.
Le **Validation accuracy** ottenute sono le seguenti:

K\S	**1x1**	**2x2**	**3x3**	**4x4**	**5x5**
2x2	*0.4630*	***0.4922***	*X*	*X*	*X*
3x3	*0.4655*	*0.4372*	*0.4033*	*X*	*X*
4x4	*0.4719*	*0.4504*	*0.4056*	*0.3818*	*X*
5x5	*0.4129*	*0.4279*	*0.4369*	*0.3762*	*0.3288*

In questo caso, il modello migliore risulta essere il modello avente **kernel size e stride pari a 2x2**.
Dunque, questo modello, **già allenato per sessantasei epoche**, viene salvato come modello scelto in modo da terminare il training ed, infine, per essere usato nella fase di Model Assessment.

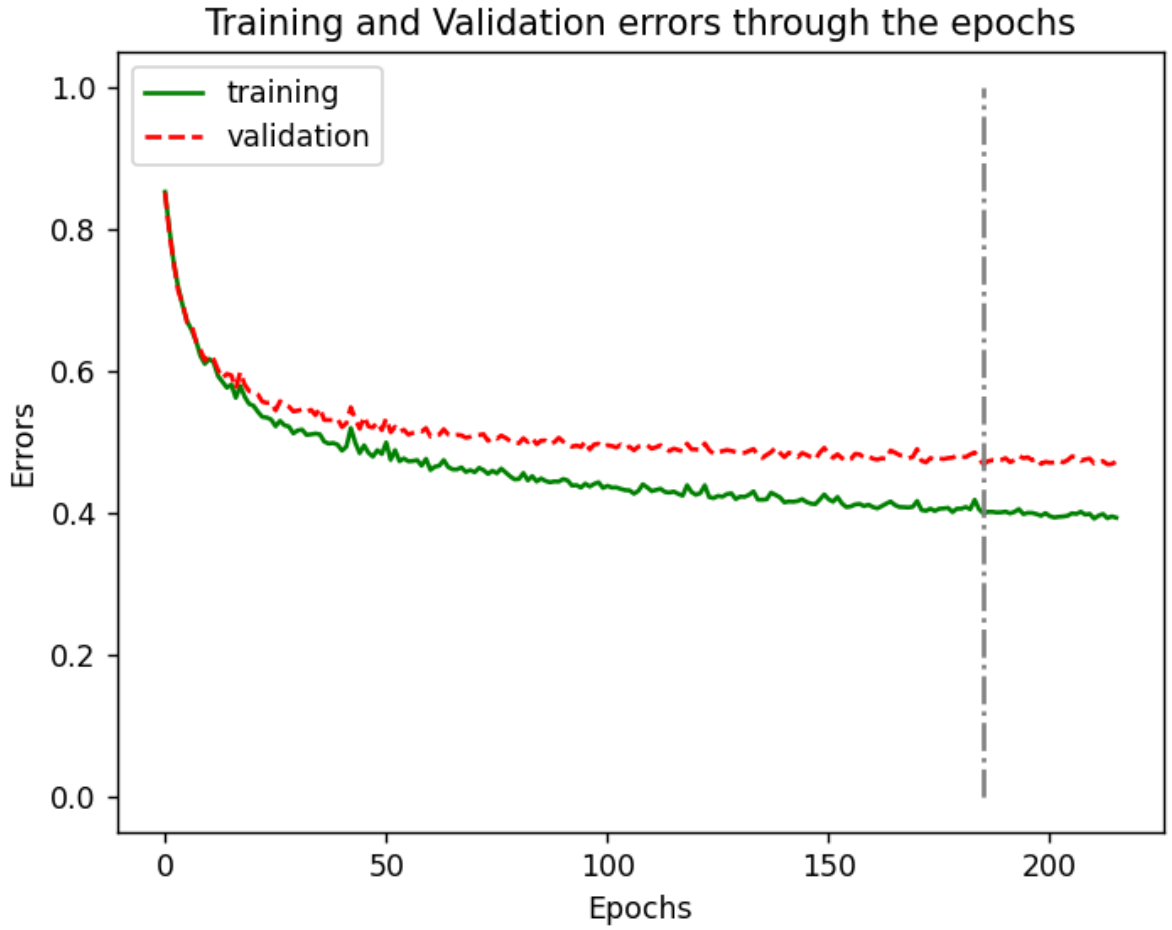
Model Assessment

Durante la procedura di Model Selection, il modello selezionato assume il seguente comportamento in **training e validation**:

Naturalmente, per le ultime cinque epoche, dato che il training in model selection è stato terminato dalla procedura di Early Stopping, il Validation error tenderà a salire.
Tuttavia, il **valore di patience dato in model selection,** cinque, **è un valore scelto deliberatamente basso**, per evitare una model selection troppo lunga.
I **valori reali di patience** per il training su un modello, solitamente, vanno da **dieci a cento**, a seconda del caso di interesse.
Dunque, per **continuare l'allenamento** del modello, si è deciso di **aumentare il valore della patience**, da cinque a **trenta**. Inoltre, come **limite massimo di epoche** è stato posto il valore **cinquecento**, per evitare di trovare situazioni in cui il modello non termini mai l'esecuzione. Tuttavia, è stato appositamente scelto un valore molto alto proprio con l'idea che l'esecuzione venga terminata dalla procedura di Early Stopping.
A questo punto, proseguendo con il training, il modello raggiungerà le **216 epoche**, ottenendo un'accuratezza in **Validation** che è pari a:

0.5315

Graficamente, l'errore in training ed in validation ottenuto durante le varie epoche appare in questo modo:



Ovvero, si nota che l'errore in Validation oscilla intorno ad un medesimo valore, mentre il training error continua la sua discesa, anche se più lentamente.
Questa è una situazione piuttosto comune, considerando che i parametri si adattano ai dati passati in training. Probabilmente, proseguendo per diverse epoche l'allenamento del modello si cadrebbe facilmente nell'overfitting.
Come ultima istanza, si può passare alla **fase di testing**. Il modello viene, dunque, applicato al testing set.
L'**accuratezza sul test set** è la seguente:

0.5296

con un **errore di generalizzazione** di circa: ***0.47040000000000004***
D'altra parte, l'**errore sul training set** del modello scelto è il seguente:

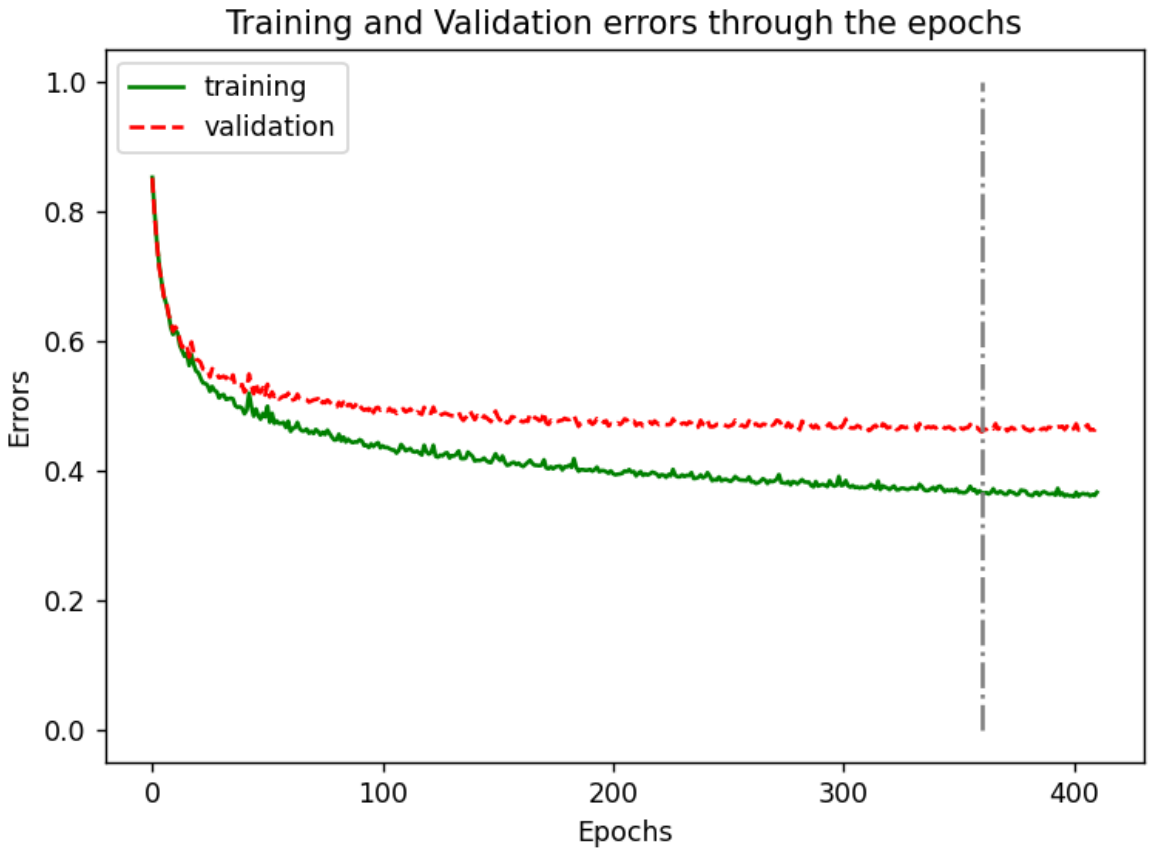
0.3993333333333333

Come è da aspettarsi, l'errore in training è inferiore rispetto all'errore di generalizzazione, considerato che il training set viene usato per la costruzione dei parametri. Non si tratta, ovviamente, di un caso di overfitting, considerato che l'errore in testing ed in training sono piuttosto simili.

Non sembra trattarsi nemmeno di una situazione di underfitting, nonostante la rete non sia estremamente performante, ma considerata la presenza di dieci classi, un'accuratezza di più del 50% del modello *(in training del 60%)* indica comunque che la rete riesce a classificare correttamente un discreto numero di immagini.
Sicuramente, è anche possibile proseguire l'allenamento della rete, aumentando il valore della patience. Ad esempio, è stata fatta una prova aumentando la patience al valore cinquanta, ed, effettivamente, si è ottenuta un'accuratezza in validation leggermente superiore:

0.5401

Graficamente si nota maggiormente il comportamento precedentemente descritto:

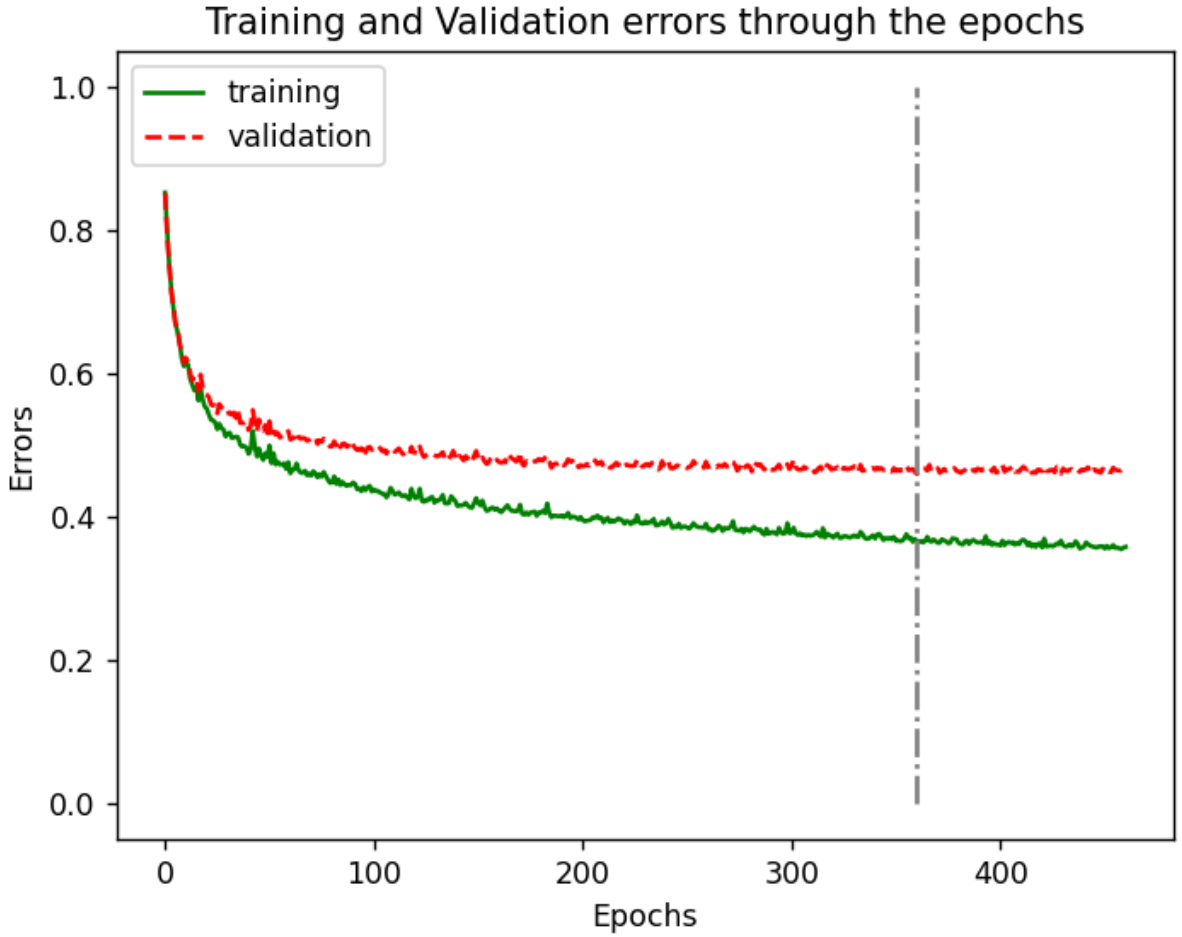


Per quanto concerne l'errore di generalizzazione, si ottiene il seguente risultato, applicando il modello al test set:

0.46130000000000004

Dunque, inferiore rispetto a quello trovato in precedenza e così diminuisce anche l'errore in training, raggiungendo il valore: **0.3661333333333333**
Si potrebbe aumentare ulteriormente il valore della patience, per osservare se è possibile trovare modelli ancora più accurati, dato che, comunque, la procedura di Early Stopping permette di evitare situazioni di overfitting.
Ad esempio, eseguendo il training con patience uguale a cento, il modello restituito in output rimane il medesimo, poiché il training non riesce ad uscire dalla procedura

di Early Stopping anche dopo cento epoche. Di seguito si può osservare il grafico di quest'ultima esecuzione:



Guida all'esecuzione

Le **librerie utilizzate** per questo progetto, e dunque che devono essere installate qualora non siano già presenti su dispositivo utilizzato, sono:

- *torch*
- *torch.nn*
- *torch.optim*
- *torchvision*
- *matplotlib.pyplot*
- *numpy*
- *colorama*
- *os.path*
- *abc*
- *sys*
- *torch.utils.data*
- *tqdm*
- *pickle*

Il progetto è stato **diviso in più file**, per rendere il codice più ordinato. Il **file per l'esecuzione è *main.py***, in cui è presente una piccola interfaccia testuale per permettere all'utente di eseguire le varie parti del codice separatamente.

- *main.py*
- *manageData.py*
- *manageFiles.py*
- *models.py*
- *modelSelection.py*
- *plotting.py*
- *testing.py*
- *training.py*
- *trainingAndTestingFunctions.py*

I file ***manageFiles.py*** e ***trainingAndTestingFunctions.py*** contengono solo delle **funzioni utili**, rispettivamente per **salvare informazioni** nella memoria non volatile e per il **training ed il testing dei modelli**.
Nel file ***models.py*** sono contenute le **classi** che implementano i modelli precedentemente descritti, ovvero: ***Model, Sequential, ConvolutionalFilter, Pooling e Linear***.
L'**esecuzione** effettiva del progetto è **divisa in quattro fasi**, ognuna rappresentata da un file diverso:

1. ***Gestione dei dati e preprocessing*** → *manageData.py*
2. ***Model Selection*** → *modelSelection.py*
3. ***Training del modello scelto*** → *training.py*
4. ***Testing del modello scelto*** → *testing.py*

Le funzioni definite in questi file sono messe in esecuzione tramite il file ***main.py***, permettendo così di eseguire il progetto per intero o separatamente.
Importante per l'esecuzione è la **gestione dei file salvati**, che devono essere inseriti in **specifiche cartelle con specifici nomi**. La struttura dell'ambiente è la seguente:

- ***Cartella principale****:*
  - ***savedObjects****:*
    - ***models****:*
      - *file contenenti i modelli*
      - ***choosedModel****:*
        - *file con il modello selezionato tramite model selection*
    - ***datasets****:*
      - *training, validation e test set*
  - *file contenenti codice python (.py)*

Qualora le cartelle non fossero nelle corrette posizioni vi potrebbero essere errori nel codice. Per questo, nel file main.py, è stata adibita la **creazione automatica delle cartelle** necessarie, qualora non siano già presenti.
Come esempio, si osservi la creazione della cartella ***/savedObjects***:

```
if not exists("./savedObjects"):
    makedirs("./savedObjects")
```

L'**esecuzione del codice per intero**, con tanto di Model Selection, senza la presenza di modelli già allenati *(resi disponibili nel codice consegnato)* può impiegare qualche ora. Infatti, i modelli forniti sono stati allenati separatamente e successivamente inseriti nella medesima cartella.
Tuttavia, **qualora si volesse effettuare un'esecuzione da zero**, senza modelli già allenati, è sufficiente **eliminare la cartella *****/savedObjects*** *(o rinominarla)* ed eseguire il file ***main.py***.

Per modificare i parametri in input al file main.py, è sufficiente **cambiare le variabili globali presenti all'inizio del file**:

```
modelSelectionEpochs = 100
trainingEpochs = 500
modelSelectionPatience = 5
trainingPatience = 30
finalModelName=f"{seed}final{trainingPatience}Patience"
modelSelectionSeeds = [seed]
modelSelectionKernels = [2, 3, 4, 5]
modelSelectionStrides = [1, 2, 3, 4, 5]
```