



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Deep Learning Application

## CNN and deep network

*Edoardo Sarri*

7173337

Giugno 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	Main . . . . .	4
1.2	Moduli Python . . . . .	4
<b>2</b>	<b>Analisi Dataset</b>	<b>5</b>
2.1	MNIST . . . . .	5
2.2	CIFAR-10 . . . . .	5
2.3	CIFAR-100 . . . . .	6
<b>3</b>	<b>Multilayer Perceptron</b>	<b>8</b>
3.0.1	Scelte . . . . .	8
3.1	Analisi senza residual connection . . . . .	8
3.2	Analisi con residual connection . . . . .	9
<b>4</b>	<b>Convolutional Neural Network</b>	<b>12</b>
4.1	Analisi senza residual connection . . . . .	12
4.2	Analisi con residual connection . . . . .	13
<b>5</b>	<b>Fine Tune A Pre-Trained Model</b>	<b>15</b>
5.0.1	Scelte . . . . .	15
5.1	Baseline . . . . .	15
5.2	Fine tune . . . . .	16
5.3	Early stopping . . . . .	17
5.4	Optimizer . . . . .	18
5.5	Iperparametri . . . . .	19
5.6	Dropout e generalizzazione . . . . .	20

# Elenco delle figure

2.1	Distribuzione del train set . . . . .	5
2.2	t-SNE in due dimensioni MNIST e CIFAR-10 . . . . .	6
2.3	Esempio CIFAR-10 RGB . . . . .	6
2.4	Esempio CIFAR-100 RGB . . . . .	7
2.5	t-SNE in due dimensioni CIFAR-10 e CIFAR-100 . . . . .	7
3.1	Loss e Accuracy MLP_4, MLP_13 e MLP_23. . . . .	9
3.2	Loss e Accuracy MLP_33. . . . .	9
3.3	Analisi sulla somma delle norme dei parametri. . . . .	10
3.4	Loss e Accuracy MLP_res. . . . .	10
3.5	Training time in second (s). . . . .	10
3.6	Somma delle norme dei parametri nei modelli con residual connection. . . . .	11
4.1	Esperimenti CNN non profonde senza residual connection . . . . .	12
4.2	Esperimento CNN profonda senza residual connection . . . . .	13
4.3	Esperimenti CNN con residual connection . . . . .	13
4.4	Training time in CNN . . . . .	14
5.1	Fine tune con stesso learning rate. . . . .	16
5.2	Fine tune con learning rate decrescente . . . . .	17
5.3	Esperimento con early stopping con patience=5 . . . . .	18
5.4	Esperimento con diversi ottimizzatori e lr=5e-4 . . . . .	18
5.5	Esperimento con diversi ottimizzatori e lr=1e-5 . . . . .	19
5.6	Ricerca degli iperparametri . . . . .	20
5.7	Ricerca degli iperparametri: esperimenti migliori. . . . .	20
5.8	Esperimento con dropout. . . . .	21

# 1 Introduzione

In questo primo capitolo viene spiegata l'organizzazione del progetto.

## 1.1 Main

Il main del progetto è rappresentato dal Jupyter notebook `main.ipynb`. È stata presa questa decisione per aumentare la leggibilità delle funzioni chiamate per arrivare ai risultati. La struttura del notebook infatti è la stessa di questa relazione.

Questo documento è pesato per non far eseguire il notebook a chi è interessato ai risultati. Ogni grafico stampato dal codice, derivato dai log su W&B [3] (framework usato durante tutto il progetto per tracciare gli esperimenti) o risultato dei vari esperimenti è riassunto in questa relazione. Con questa idea i vari file `.py` sono utili non per capire cosa è stato fatto, ma come è stato fatto.

## 1.2 Moduli Python

Le funzioni chiamate nel `main.py` sono importate da quattro moduli:

- `myModel.py`  
Contiene i modelli usati nel progetto e sono classi che estendono `nn.Module`. Spesso si utilizzano gli stessi blocchi di base, definiti in `modelBlock.py`.
- `coreFunction.py`  
Contiene le funzioni responsabili dell'addestramento, dell'estrazione dell'accuracy, della pipeline usata durante tutto il progetto e del features extractor.
- `utilityFunction.py`  
Contiene funzioni ausiliarie per rendere il codice più leggibile riutilizzabile. Sono funzioni chiamate più volte e che svolgono compiti precisi.

## 2 Analisi Dataset

Quando si usa un dataset è sempre bene capire con cosa si lavora. In questo capitolo si analizzano i dataset utilizzati, cioè MNIST, CIFAR-10 e CIAFR-100.

Come possiamo osservare dalla Figura 2.1 i due CIAFR sono dataset perfettamente bilanciati, mentre MNIST è abbastanza bilanciato. Per questo motivo durante la fase di analisi sarà valutata solamente l'accuracy, senza considerare altre metriche come la precision, la recall e la F1.

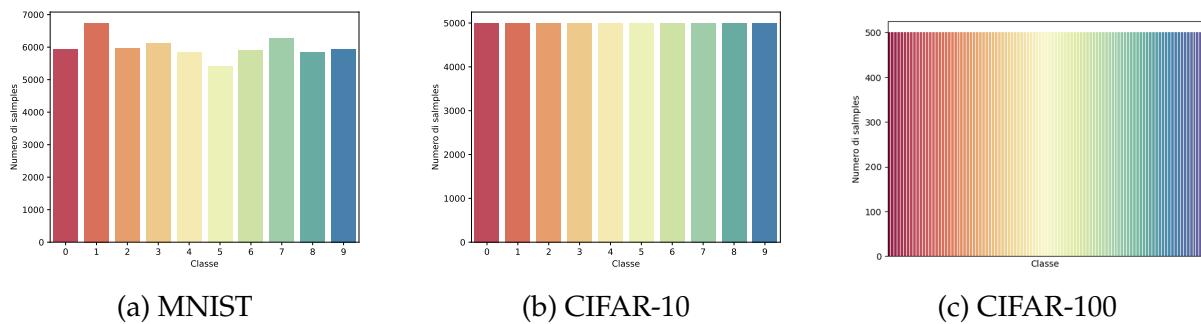


Figura 2.1: Distribuzione del train set

### 2.1 MNIST

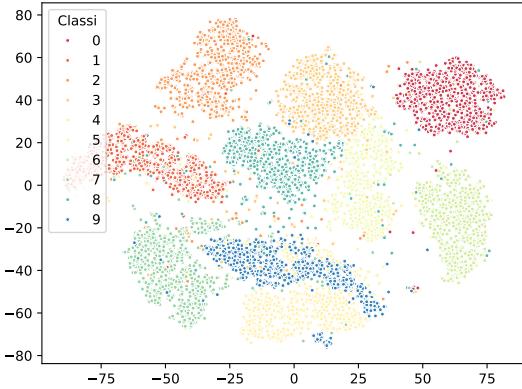
MNIST è un dataset relativamente semplice, i cui elementi rappresentano immagini di numeri scritti a mano. Il compito di classificazione sta nel decidere a quale delle possibili 10 classi appartiene un elemento. Ogni istanza ha una dimensione  $28 \times 28$  con un solo canale in scala di grigi. Ogni pixel è un numero naturale da 0 a 255, dove 0 indica il nero e 255 il bianco. È composto da 70K samples totali, già divisi in un train set da 60K samples e un test set da 10K.

Per rappresentare la sua semplicità (soprattutto rispetto a CIFAR-10) si sono rappresentati i dati in due dimensioni tramite l'algoritmo t-SNE, come si vede in Figura 2.2a.

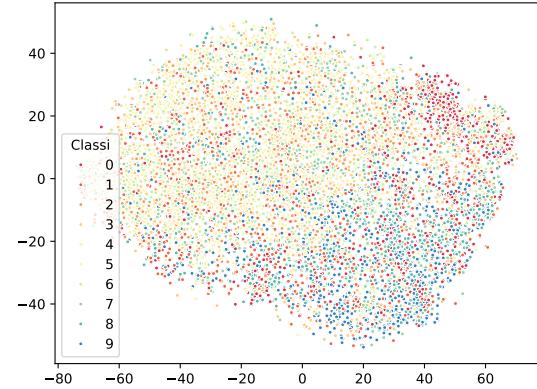
Durante l'importazione gli elementi del dataset sono stati trasformati in tensori e normalizzati usando la media e la varianza.

### 2.2 CIFAR-10

CIFAR-10 è un dataset più complesso rispetto a MNIST, i cui elementi rappresentano immagini di oggetti. Il compito di classificazione anche in questo caso è quello di predire una delle possibili 10 classi. Ogni elemento del dataset è un'immagine  $32 \times 32$



(a) MNIST



(b) CIFAR-10

Figura 2.2: t-SNE in due dimensioni MNIST e CIFAR-10

con tre canali, come mostrato in Figura 2.3; è quindi un’immagine RGB, dove per ogni canale 0 rappresenta il nero e 255 il bianco. È composta da 60K samples totali, dove 50K sono dedicati al train set e 10K al test set. Il test set è perfettamente bilanciato con 5K immagini per classe.

Anche in questo caso per mettere in evidenza la difficoltà della classificazione rispetto a MNIST si è usato t-SNE su due dimensioni, come si vede in Figura 2.2b.

Durante l’importazione, oltre alle operazioni eseguite per MNIST, i dati sono stati anche scalati in  $128 \times 128$ . In questo modo si sono potute eseguire più di 3 operazioni di pooling (supponendo di dimezzare la dimensionalità ad ogni operazione).

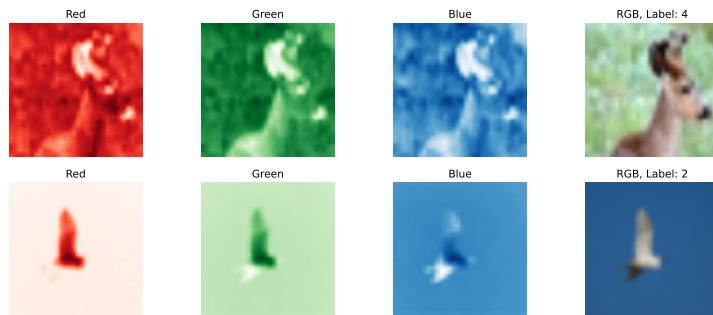


Figura 2.3: Esempio CIFAR-10 RGB

## 2.3 CIFAR-100

CIFAR-100 ha esattamente la stessa struttura di CIFAR-10 (Figura 2.4), ma con piccole e sostanziali differenze che rendono più complesso il lavoro di predizione (Figura 2.5) [2]:

1. Le classi sono gerarchiche. Abbiamo 20 super classi, ognuna delle quale è raffinata in 5 sotto classi, per un totale di 100 classi.

2. Siccome il numero di samples è lo stesso, ci sono poche immagini per ogni classe. In particolare, per ogni classe, ci sono 500 immagini per il train set e 100 per il test set.

Come in CIFAR-10 il formato delle immagini è sempre  $32 \times 32 \times 3$  (RGB). Queste sono state scalate  $128 \times 128$  per gli stessi motivi elencati sopra.

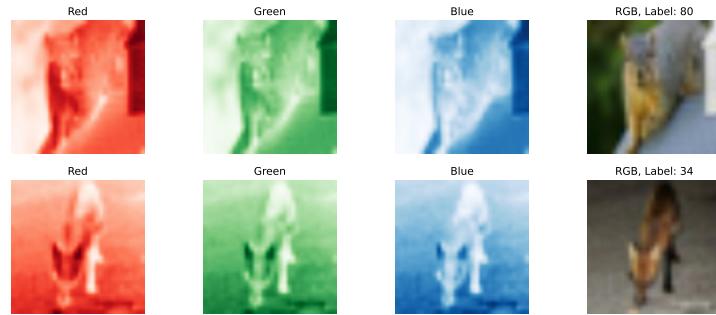


Figura 2.4: Esempio CIFAR-100 RGB

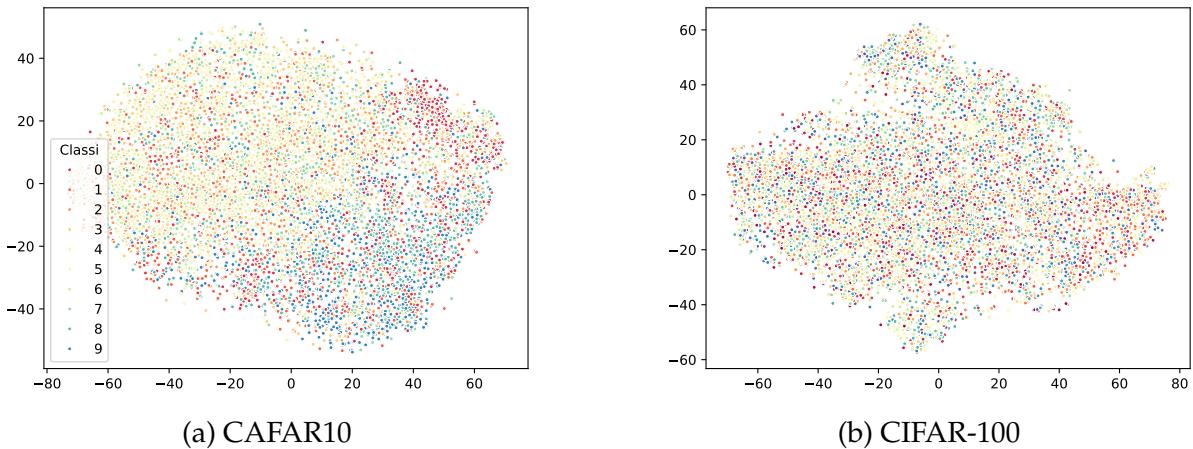


Figura 2.5: t-SNE in due dimensioni CIFAR-10 e CIFAR-100

# 3 Multilayer Perceptron

In questo capitolo e nel Capitolo 4 sono stati replicati, in piccola scala, i risultati ottenuti nel paper di ResNet [1]. Si è voluto quindi dimostrare che andare più in profondità non vuol dire in ogni caso aumentare le prestazioni del modello. Inoltre si è poi notato come aggiungendo le residual connection si ottengono risultati migliori anche per le reti più profonde.

Per svolgere il compito di classificazione è stato usato il dataset MNIST, presentato nella Sezione 2.1.

## 3.0.1 Scelte

Ho deciso di non valutare utilizzare un validation set, ma di basarmi solo sul train e test set. Il mio obiettivo non era infatti quello di valutare le prestazioni di un modello in relazione a più iperparametri oppure di migliorare un dato modello per ottenere performance sempre migliori, ma solo di capire se e a quale profondità la rete smetteva di imparare. Un validation set infatti è utile per non includere nell'addestramento informazioni su dati che saranno poi usati in fase di testing; in questo caso infatti non riusciremo a capire quanto bene il modello generalizza.

Nel caso del MLP senza residual block sono stati usati 4 modelli, con un numero crescente di layer (ogni uno di tipo `nn.Linear`): 4, 13, 23 e 33. I modelli testati con le residual connection sono invece 5: sono stati aggiunti due modelli molto profondi per valutare le prestazioni su un numero di layer (molto) elevante.

In tutti gli esperimenti il numero all'interno dei nomi che vedremo nei grafici è relativo al numero di layer del modello.

## 3.1 Analisi senza residual connection

Partiamo dall'osservare la Figura 3.1, relativa alla loss e l'accuracy delle reti con 4, 13 e 23 layer lineari. Si può osservare come le performance del modello siano inversamente proporzionali alla sua profondità. Il modello più profondo infatti ha ottenuto i risultati peggiori sia nell'accuracy sul test set che nella loss sul train set. Già questo è un primo risultato che verifica la tesi. Inoltre se osserviamo bene MLP\_23 possiamo vedere che nelle prime epoche è nettamente peggiore rispetto agli altri.

Analizziamo ora i grafici in Figura 3.2 relativi a MLP\_33, il multilayer perceptron più profondo testato. È chiaro come la rete non stia imparando nulla. La loss è

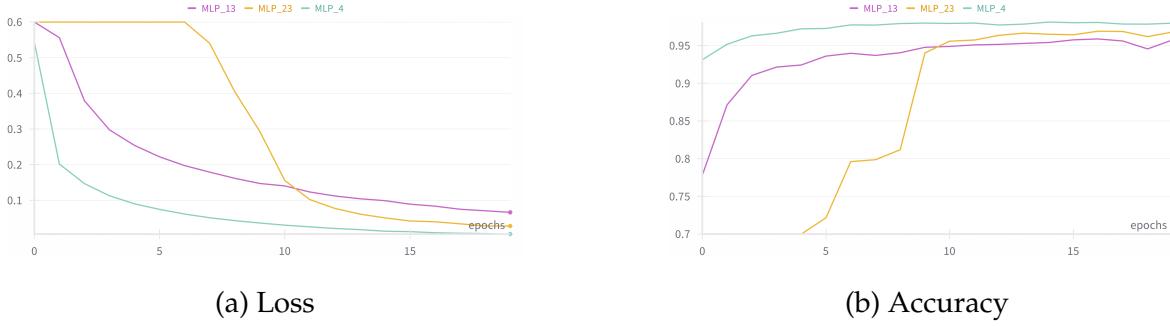


Figura 3.1: Loss e Accuracy MLP\_4, MLP\_13 e MLP\_23.

molto alta (circa 2,3) e non riesce minimamente a convergere verso l’ottimo; l’accuracy è pari a 0,1, che equivale a lanciare un dato con 10 facce.

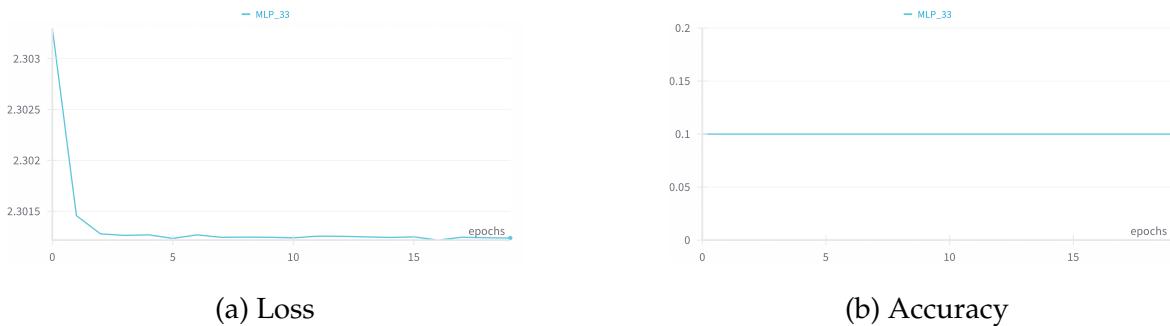


Figura 3.2: Loss e Accuracy MLP\_33.

Un’analisi interessante è quella relativa alle norme dei gradienti delle varie reti. Quello che ho fatto è stato sommare per ogni parametro la norma del suo gradiente alla fine di ogni epoca.

In Figura 3.3a si osservano i gradienti dei modelli meno profondi, quelli con 4, 13 e 23 layer. Si può notare come le prestazioni siano inversamente proporzionali alla norma; probabilmente dovrebbero essere fatti altri esperimenti e non si può dire che il risultato sia definitivo, però di sicuro è qualcosa di interessante. Se invece osserviamo la Figura 3.3b vediamo il risultato sul modello con 33 layer; in questo caso è chiaro che i risultati relativi all’accuracy nell’esperimento di questo modello sono dovuti al problema del vanishing gradient.

## 3.2 Analisi con residual connection

L’utilizzo delle residual connection, come si può osservare dai risultati in Figura 3.4, aiutano la rete a non perdere totalmente le prestazioni. Anche con moltissimi (forse troppi) layer le prestazioni sono tutte equivalenti. Se si guardano i dati nel dettaglio

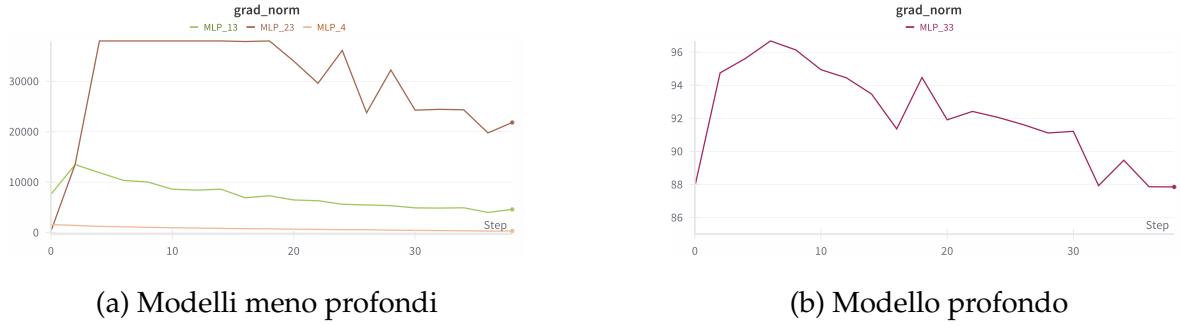


Figura 3.3: Analisi sulla somma delle norme dei parametri.

possiamo forse dire che, anche se di veramente poco e forse solo nelle prime epoche, i modelli più profondi sono addirittura migliori, cosa impensabile senza i residui.

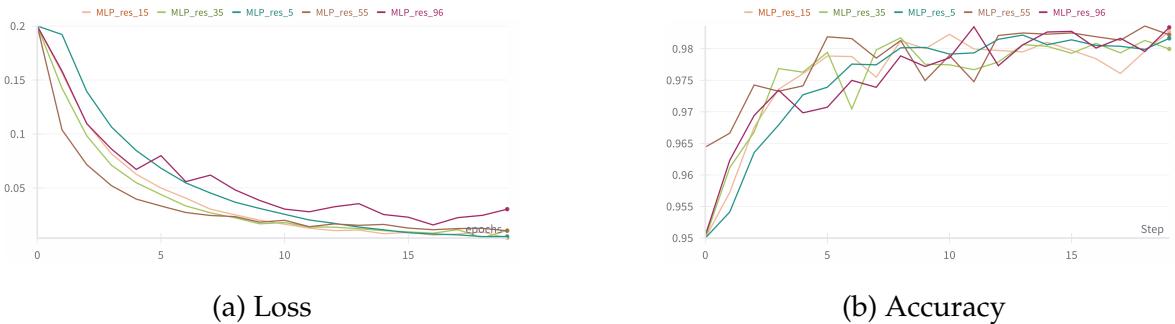


Figura 3.4: Loss e Accuracy MLP\_res.

Ho pensato che i tempi di addestramento a parità di layer potessero essere diversi, a favore dei modello che usano le residual connection. Ho ottenuto però un risultato che non conferma questa ipotesi, come mostrato in Figura 3.5. Questo però potrebbe anche essere dovuto alla semplicità del modello e alla loro (anche se piccola) differenza nel numero di layer. Per questo motivo sarà fatta la stessa analisi su modelli più profondi e complessi nel prossimo capitolo.

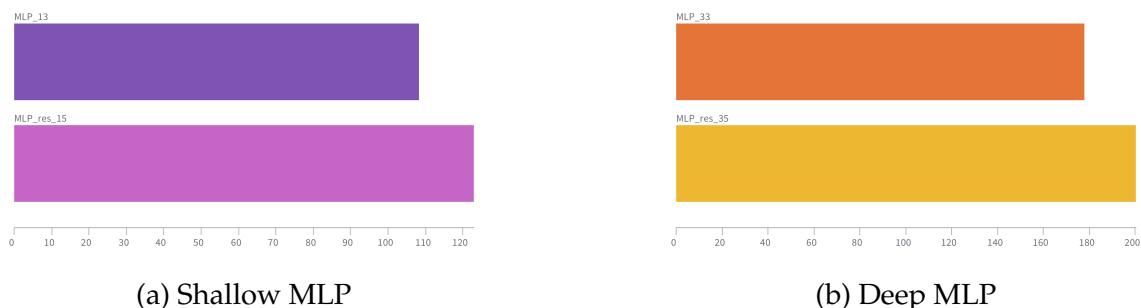


Figura 3.5: Training time in second (s).

L'utilità delle residual connection si vede chiaramente anche sulle osservazioni che possiamo fare sui gradienti dei parametri. Se osserviamo infatti la Figura 3.6 vediamo una situazione totalmente diversa rispetto ai modelli senza residual connection: andare in profondità cambia di poco la somma delle norme dei gradienti.



Figura 3.6: Somma delle norme dei parametri nei modelli con residual connection.

# 4 Convolutional Neural Network

Le analisi svolte in questo capitolo seguono la stessa linea del Capitolo 3. Si è voluto dimostrare che andare più in profondità con una CNN non significa migliorare le prestazioni; aggiungere troppi hidden layer porta il modello a non apprendere nulla. Si è dimostrato infine che le residual connection migliora le cose e mitigano i problemi.

In questo contesto il dataset di riferimento è stato CIFAR-10, già analizzato nella Sezione 2.2. Il motivo di questa scelta è, data la maggior complessità del dataset in questione, la possibilità di ottenere risultati abbastanza veritieri. MNIST infatti è molto semplice e sbagliare qualcosa e fare male è veramente molto difficile.

Le scelte relative a non utilizzare il validation set e a non valutare la loss sul test set sono esattamente le stesse messe in evidenza nel capitolo precedente; è infatti importante ricordare che l’obiettivo non è quello di ottenere alte performance.

Anche in questo caso i nomi degli esperimenti sono relativo al numero di layer del modello analizzato.

## 4.1 Analisi senza residual connection

Se analizziamo inizialmente CNN\_10 e CNN\_13 in Figura 4.1, possiamo osservare l’opposto rispetto ai modelli MLP. Finché il numero di layer non è troppo elevato andare più in profondità migliora sia la loss sul train set che l’accuracy sul test set.

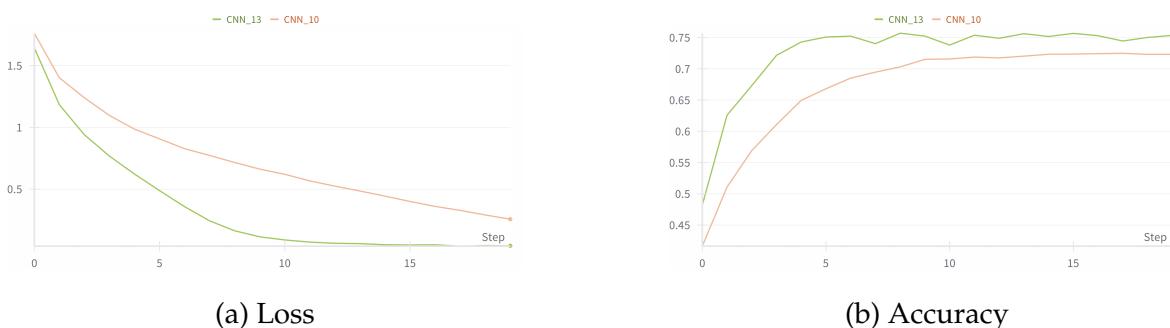


Figura 4.1: Esperimenti CNN non profonde senza residual connection

Quando però la rete diventa troppo profonda, come per CNN\_30 in Figura 4.2, si osserva lo stesso problema dei MLP: il modello non impara niente e la loss rimane molto alta. È chiaro che il problema sia sempre quello del vanishing gradient.

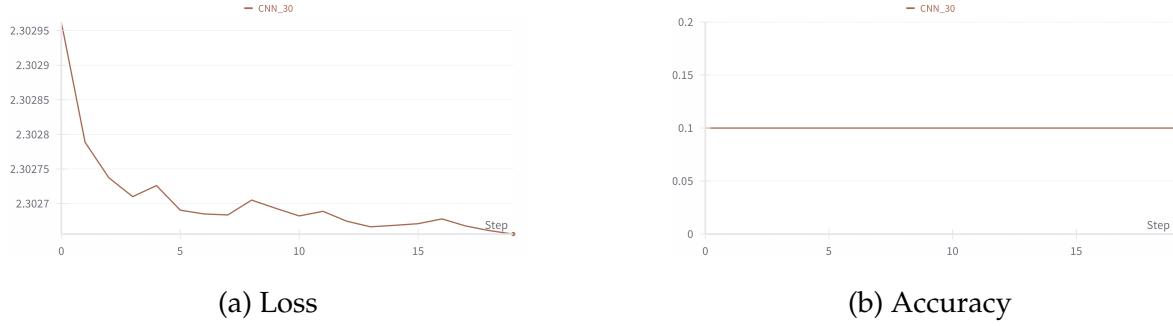


Figura 4.2: Esperimento CNN profonda senza residual connection

## 4.2 Analisi con residual connection

Come nel caso nel MLP anche qua sono stati introdotte le residual connection per le varie CNN analizzate precedentemente. Le reti in questione sono esattamente le stesse a cui abbiamo aggiunto i residui tra blocchi convoluzionali.

Possiamo osservare contemporaneamente CNN\_res\_16, CNN\_res\_32 e CNN\_res\_56 nella Figura 4.3. In questo caso possiamo notare in modo ancora più evidente l'utilità delle residual connection nel mitigare (in questo caso risolvere) il problema del vanishing gradient. I modelli in questione sono tutti confrontabili, sia secondo la loss sul train set sia per l'accuracy sul test set.

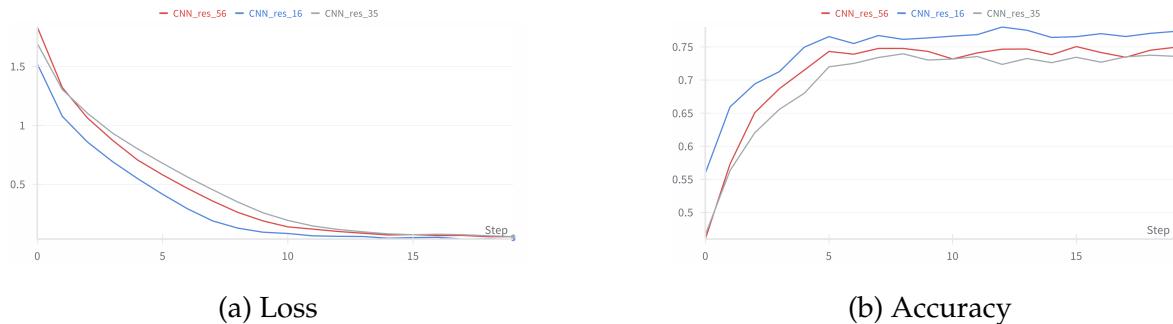


Figura 4.3: Esperimenti CNN con residual connection

Come ho fatto nel capitolo precedente ho analizzato i tempi di addestramento per le coppie di reti con e senza residual connection, cioè per CNN\_13 vs CNN\_res\_16 e per CNN\_30 vs CNN\_res\_35. Pensavo che per addestrare una rete con i residui fosse necessario meno tempo, ma il risultato, come mostrato in Figura 4.4, non ha confermato questa mia idea.

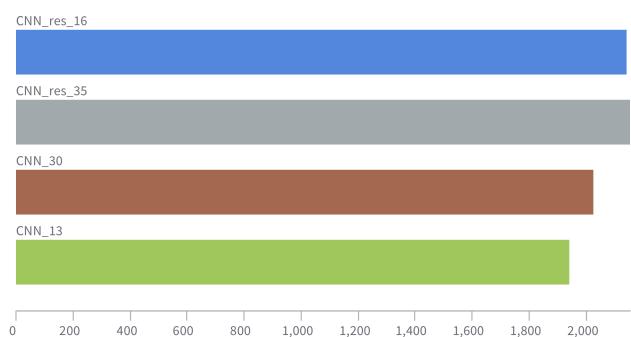


Figura 4.4: Training time in CNN

# 5 Fine Tune A Pre-Trained Model

L'obiettivo di questa parte è stato quello di cercare di ottenere le migliori performance possibili da una CNN su un dataset diverso da quello su cui la rete è stata pre addestrata.

È stata prima stabilita una baseline con un classico classificatore standard.

## 5.0.1 Scelte

Fino a questo momento l'obiettivo non era basato sulle prestazioni; come abbiamo già spiegato nei Capitoli 3 e 4 l'architettura delle reti usate è stata definita senza un qualche razionale che incrementasse una qualche metrica, ma solo per dimostrare che la profondità non è direttamente proporzionale alle prestazioni. Per questo motivo non ho preso una delle reti definite in precedenza, ma ho usato un modello noto, già definito e strutturato per massimizzare le prestazioni: ResNet-18.

Il dataset scelto per eseguire gli esperimenti è CIFAR-100. Come già analizzato nella Sezione 2.3, questo è più complesso per quanto riguarda tipo e specializzazione delle classi, e quindi rende più complesso il compito. Visto l'obiettivo si è deciso di usare un validation set: il train set di partenza , composto da 50K elementi, è stato suddiviso in un train set di 45K elementi e un validation set di 5K.

Durante i vari cicli di addestramento il test set è stato dimenticato (come se non ci fosse). Solo alla fine è stato poi utilizzato per vedere come i migliori modelli generalizzano su dati mai visti.

## 5.1 Baseline

Per stabilire una baseline su CIFAR-100 sono state valutate le performance di due modelli di classificazione classici: SVM e Logist Regression.

L'addestramento di questi due classificatori è stato fatto sulle features estratte da ResNet-18 su CIFAR-100. I pesi di ResNet sono ricavati dall'addestramento su CIFAR-10. Per l'estrazione di features è stato fondamenta rimuovere l'ultimo layer lineare presente in ResNet; in questo modo le features estratte sono il risultato dell'ultimo average pooling della rete.

Per l'addestramento della ResNet è stato necessario sostituire l'ultimo layer lineare, che ha dimensionalità dell'output 1000, con uno che avesse una dimensione in uscita

di 10 classi (le 10 classi di CIAFR-10). Con questo approccio l'accuracy ottenuta dai due classificatori è stata di circa 0,37.

## 5.2 Fine tune

Come primo tentativo per aumentare le performance ho cercato di fare fine tuning del modello addestrato su CIFAR-10. Per prima cosa, prima di procedere all'addestramento su CIFAR-100, ho dovuto modificare l'ultimo layer lineare, quello che si occupa della classificazione, in modo che la dimensione dell'output sia 100 e non 10.

Ho provato a congelare i pesi di tutti i layer tranne quelli del nuovo layer lineare (`nn.Linear()`), in quello che è stato definito l'esperimento `fine_tune_1`; in questo modo tutti i pesi restano gli stessi, tranne quelli che sono responsabili della classificazione vera e propria. Con questa soluzione l'accuracy registrata è stata inferiore alla baseline, con un valore di circa 0,29.

Successivamente ho scongelato i pesi del quarto layer. In un primo momento l'idea era quella di abbassare il valore della loss, portandola da  $5 \cdot 10^{-4}$  a  $10^{-5}$ . Osservando però le curve della loss sul train set e dell'accuracy dell'addestramento precedente, si può notare come non siamo ancora arrivati ad un asintoto, per cui ho deciso di lasciarlo invariato. Come si mostra però dalla Figura 5.1, questo approccio ha portato ad avere overfitting; si nota infatti che la loss sul train set è praticamente a 0.

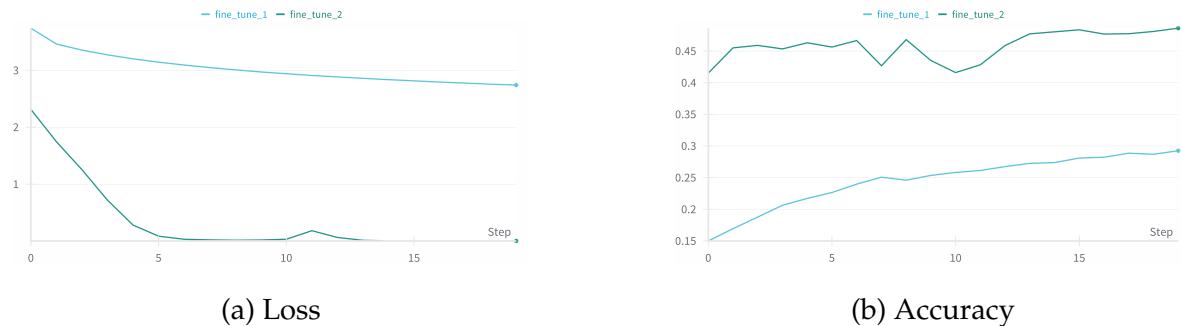


Figura 5.1: Fine tune con stesso learning rate.

Per cercare di mitigare il problema ho abbassato il learning rate quando ho addestrato i layer precedenti all'ultimo (esperimenti `fine_tune_x_lr`), ma con un learning rate di un ordine inferiore. I risultati, mostrati in Figura 5.2 sono stati più soddisfacenti: la loss ha una discesa molto più regolare; l'accuracy ha raggiunto i livelli di quella dell'esperimento `fine_tune_2` ma probabilmente la generalizzazione è migliore. L'ultima osservazione che si può fare è sui valori alti della loss: si può notare che la curva di discesa (così come quella di salita dell'accuracy) non ha raggiunto un asintoto; mol-

to probabilmente con più epochi il modello si daterebbe meglio ai dati e avremmo risultati migliori.

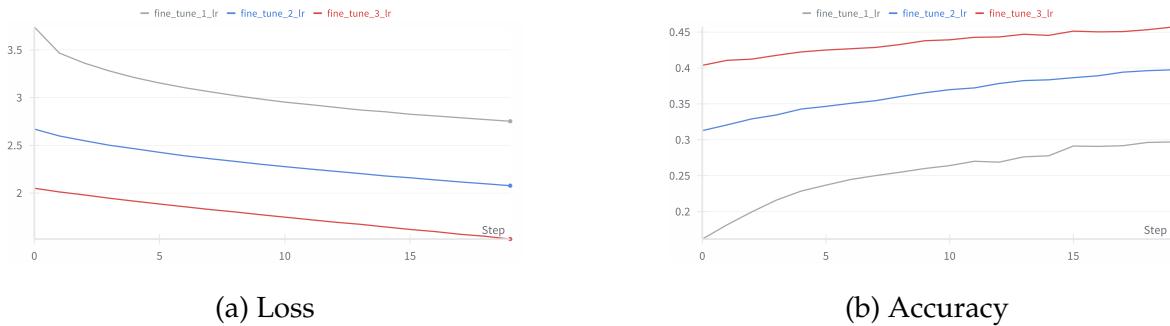


Figura 5.2: Fine tune con learning rate decrescente

Non ho scongelato altri layer perché altrimenti ci saremmo allontanati dal nostro obiettivo. Se scongelassimo tutti i layer non staremmo facendo altro che un classico addestramento di ResNet su CIFAR-100 a partire da pesi non inizializzati casualmente; quello che vogliamo invece è ottimizzare le prestazioni di un modello già addestramento.

### 5.3 Early stopping

Nel seguito continuiamo ad addestrare il modello di ResNet variando solo i pesi degli ultimi due layer più quello di classificazione. Manteniamo quindi congelati i pesi dei primi layer della rete.

Per cercare di migliorare la generalizzazione del modello ho applicato la tecnica di early stopping. L'idea è quella di monitorare l'accuracy sul validation set e di interrompere l'addestramento se non migliora di una data tolleranza per un certo numero di epochi (la "patience").

Per vedere l'early stopping in azione senza utilizzare moltissime epochi e non avere tempi di addestramento lunghissimi (in pratica solo per avere un esperimento fittizio), ho dovuto tenere un learning rate alto ( $5 \cdot 10^{-3}$ ) e una patience bassa (5). Osservando la Figura 5.3 si può notare esattamente questo: la loss sul train set raggiunge un valore molto basso; l'accuracy inizia a oscillare senza però migliorare.

Questo è un esempio fittizio e probabilmente poco utile, ma sicuramente fa capire quanto questa tecnica possa diventare utile per fermare l'addestramento sotto determinate condizioni.

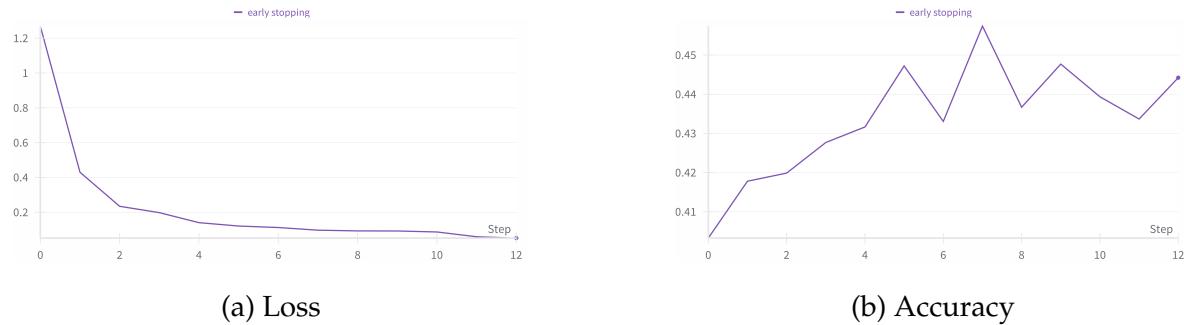


Figura 5.3: Esperimento con early stopping con patience=5

## 5.4 Optimizer

Ho cercato di variare l'algoritmo di ottimizzazione per cercare di capire quale si adattasse al problema. In tutti gli addestramenti è stata usata la tecnica di early stopping implementata in precedenza. Gli ottimizzatori testati sono stati:

- Adam
  - GSD
  - AdamW, che migliora la generalizzazione del modello con la tecnica weight decay.
  - RMSprop, che migliora SGD con la scelta del learning rate dinamicamente basandosi su una media dei gradienti.

Un primo approccio è stato quello di usare un learning rate di  $5 \cdot 10^{-4}$ , ma non ha funzionato bene. Come si può vedere dalla Figura 5.4 tutte le funzioni di ottimizzazione sono state soggette ad overfitting, tranne SGD che è anche l'unico che ha attivato l'early stopping (ma ha comunque avuto prestazioni non soddisfacenti).

Questo risultato ha evidenziato quando già avevamo capito dalla prima parte della Sezione 5.2: quando si scongelano solo alcuni layer conviene procedere più lentamente, tramite un learning rate basso, per avvicinarsi in modo più preciso alla soluzione.

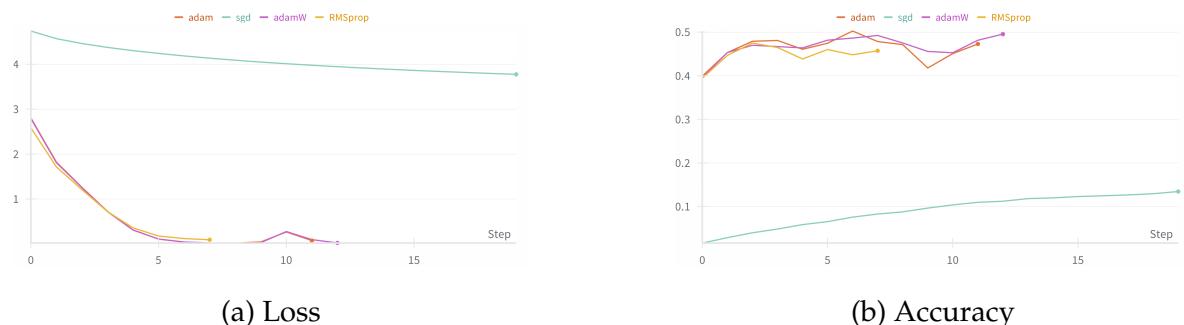


Figura 5.4: Esperimento con diversi ottimizzatori e  $lr=5e-4$

Ho allora abbassato il learning rate a  $10^{-5}$  e i risultati sono stati più soddisfacenti. Come si osserva dalla Figura 5.5 le prestazioni di SGD sono ancora di molto inferiori agli altri sia confrontando la loss sul training set che l'accuracy sul validation set. Gli altri 3 ottimizzatori sono invece confrontabili: quello che ha avuto risultati migliori è stato Adam classico. Questo sarà anche l'ottimizzatore utilizzato nelle fasi successive.

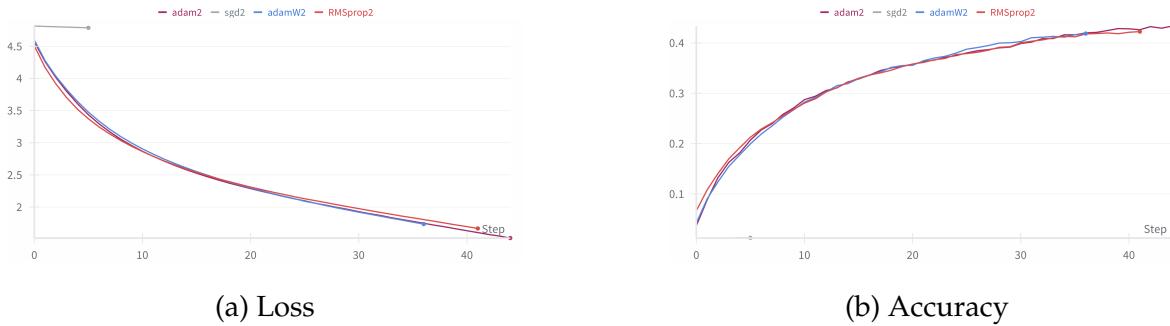


Figura 5.5: Esperimento con diversi ottimizzatori e lr=1e-5

Quello che rimarebbe da fare è testare i vari algoritmi con learning rate a metà tra quelli testati. Questo però è fatto nella ricerca degli iperparametri nella sezione successiva.

## 5.5 Iperparametri

Cerchiamo di trovare i migliori iperparametri del modello. Gli iperparametri presi in considerazione sono:

- Learning rate

L'idea è quella di considerare un learning rate che possa andare da  $10^{-2}$  fino a  $10^{-6}$ . Questa variabilità non è decisa in modo discreto, ma è lasciato allo sweeper di W&B, che decide il valore campionandolo all'interno dell'intervallo.

- Batch size

Come possibili valori ho considerato 32, 64, 128 e 256.

Come si vede dalla Figura 5.6 stati eseguiti 12 esperimenti.

Se osserviamo gli esperimenti che sono andati meglio (4, 6 e 10) in Figura 5.7 possiamo osservare che il learning rate è dello stesso ordine di grandezza ( $\mathcal{O}(10^{-3})$ ), mentre il batch size è 64 in due casi e 128 nell'altro. Nel prossimo paragrafo addestreremo il modello con learning rate e batch size dell'esperimento 6, il migliore registrato.

	toasty-sweep-12		Add notes	edoard	6h ago	34m 46s	msnmhtrg	128	-	-	0.0000024	0.18825	3.52845
	lively-sweep-11		Add notes	edoard	6h ago	29m 9s	msnmhtrg	256	-	-	0.0034013	0.47513	0.10202
	worthy-sweep-10		Add notes	edoard	7h ago	15m 41s	msnmhtrg	128	-	-	0.0027407	0.4959	0.13731
	dark-sweep-9		Add notes	edoard	7h ago	34m 48s	msnmhtrg	128	-	-	0.0000263	0.44755	1.3166
	wild-sweep-8		Add notes	edoard	8h ago	44m 45s	msnmhtrg	16	-	-	0.0000037	0.35205	2.5624
	drawn-sweep-7		Add notes	edoard	9h ago	36m 4s	msnmhtrg	64	-	-	0.0000057	0.33478	2.58771
	still-sweep-6		Add notes	edoard	9h ago	16m 14s	msnmhtrg	64	-	-	0.0093064	0.50697	0.12357
	easy-sweep-5		Add notes	edoard	9h ago	20m 13s	msnmhtrg	16	-	-	0.0097322	0.48185	0.32046
	usual-sweep-4		Add notes	edoard	10h ago	14m 27s	msnmhtrg	64	-	-	0.0095642	0.49561	0.16794
	crimson-sweep-3		Add notes	edoard	10h ago	44m 48s	msnmhtrg	16	-	-	0.0000037	0.35782	2.5581
	major-sweep-2		Add notes	edoard	11h ago	15m 40s	msnmhtrg	128	-	-	0.0058932	0.48748	0.14808
	sleek-sweep-1		Add notes	edoard	11h ago	14m 29s	msnmhtrg	64	-	-	0.0012879	0.48273	0.16231

Figura 5.6: Ricerca degli iperparametri

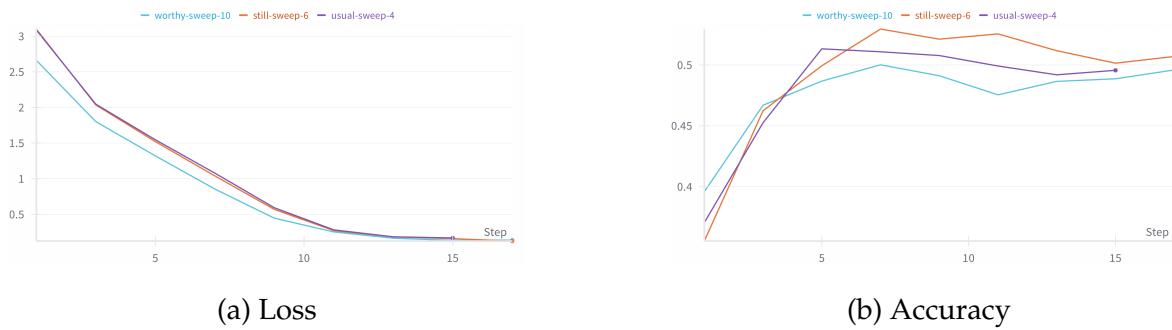


Figura 5.7: Ricerca degli iperparametri: esperimenti migliori.

## 5.6 Dropout e generalizzazione

Vediamo ora come il miglior modello che abbiamo ottenuto si comporta nell’addestramento e come generalizza sui dati mai visti del test set. Gli iperparametri utilizzati sono quelli ricavati alla Sezione 5.5.

In Figura 5.8 si possono osservare le metriche ottenute durante l’addestramento: la loss sul train set e l’accuracy sul validation set. Possiamo notare come l’accuracy ottenga usando il dropout durante l’addestramento sia la migliore riscontrata fino a questo momento.

L’accuracy ottenuta sul test set è stata invece di circa 0,51; questo risultato è verificabile solo eseguendo il codice. Ovviamente è normale che sul test set l’accuracy è più bassa rispetto a quella sul validation set. Però il fatto che non sia comunque troppo inferiore ci fa capire che il modello generalizza abbastanza bene.

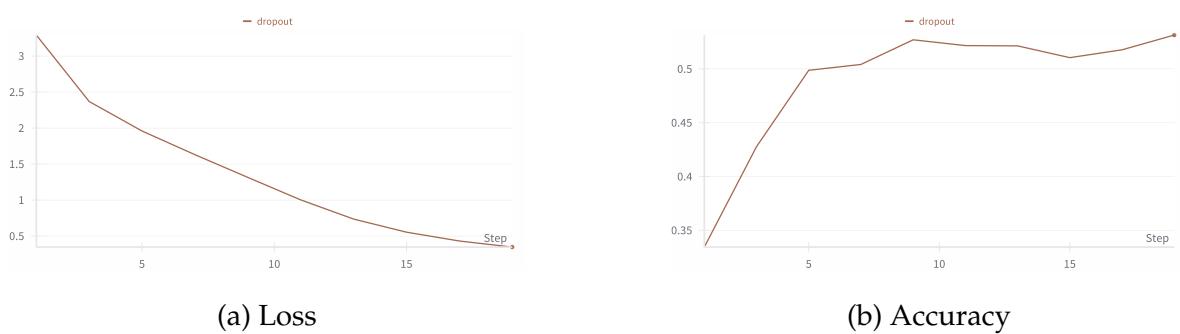


Figura 5.8: Esperimento con dropout.

# Bibliografia

- [1] Kaiming He et al. *Deep Residual Learning for Image Recognition*. <https://arxiv.org/pdf/1512.03385.pdf>. Data ultima visualizzazione: 2025-03-23. CoRR, 2015.
- [2] Alex Krizhevsky. *The CIFAR-100 dataset*. <https://www.cs.toronto.edu/~kriz/cifar.html>. Data ultima visualizzazione: 2025-03-27. Alex Krizhevsky, 2009.
- [3] *Weight & Biases*. <https://wandb.ai/site/>.