

Modello ResNet per classificazione del dataset SVHN

Studente: Filippo Lutterotti

Ambiente: Conda (anaconda3) – Python

Obiettivo: Classificazione di immagini

Modello: ResNet-30

L'obiettivo principale di questo progetto è stato addestrare un modello ResNet in grado di riconoscere e classificare immagini di numeri civici. Attraverso l'apprendimento supervisionato e l'uso di tecniche di *data augmentation*, il progetto ha mirato a migliorare la capacità delle reti neurali convoluzionali di generalizzare efficacemente su nuovi compiti di *computer vision*.

ResNet

Ho implementato una rete di tipo ResNet a 31 livelli, progettata per classificare le immagini in 10 classi (numeri da 0 a 9). La ResNet (*Residual Network*) è una tipologia di rete neurale convoluzionale ampiamente utilizzata per la classificazione di immagini, come nel mio caso.

La caratteristica distintiva della ResNet è l'uso di blocchi residui, che consentono al modello di apprendere in modo più efficiente, riducendo il problema del *vanishing gradient*. Questo problema si verifica nelle reti quando i gradienti, utilizzati per aggiornare i pesi durante l'addestramento, diventano troppo piccoli, impediscono alla rete di apprendere correttamente.

Esistono diverse versioni di ResNet, che possono variare in complessità: versioni più semplici con meno livelli o versioni più complesse con un maggior numero di livelli per una migliore classificazione.

Alcune caratteristiche che hanno aiutato l'apprendimento:

-**Ottimizzatore (Adam):** Adam è un algoritmo di ottimizzazione efficace per gestire problemi con gradienti sparsi e rumorosi (rumore, inteso come cosa o oggetto in più e disturbante per l'analisi e training). Questo ottimizzatore regola dinamicamente il tasso di apprendimento per ogni parametro della rete, migliorando la convergenza.

-**Criterion (CrossEntropyLoss):** La *Cross Entropy Loss* è una funzione di perdita comunemente usata per problemi di classificazione multi classe. Misura la differenza tra la distribuzione delle probabilità predetta dal modello e la distribuzione reale, penalizzando le previsioni errate.

-**Scheduler (StepLR):** Lo scheduler regola il *learning rate* durante l'allenamento. In questo caso, il *StepLR* riduce il tasso di apprendimento ogni 5 epoche (impostazione di `step_size=5`), moltiplicandolo per 0.1 ($\gamma=0.1$), quindi riduce del 10%. Questo aiuta il modello a convergere meglio man mano che l'allenamento procede.

Analisi e Test

Per i miei esperimenti, ho utilizzato il dataset SVHN (*Street View House Numbers*), che contiene immagini di piccole dimensioni (32x32 pixel) raffiguranti numeri civici. Il dataset, composto da circa 72.000 immagini per il training e 36.000 per il test, è stato scaricato tramite torchvision.

Ecco un esempio di immagine:

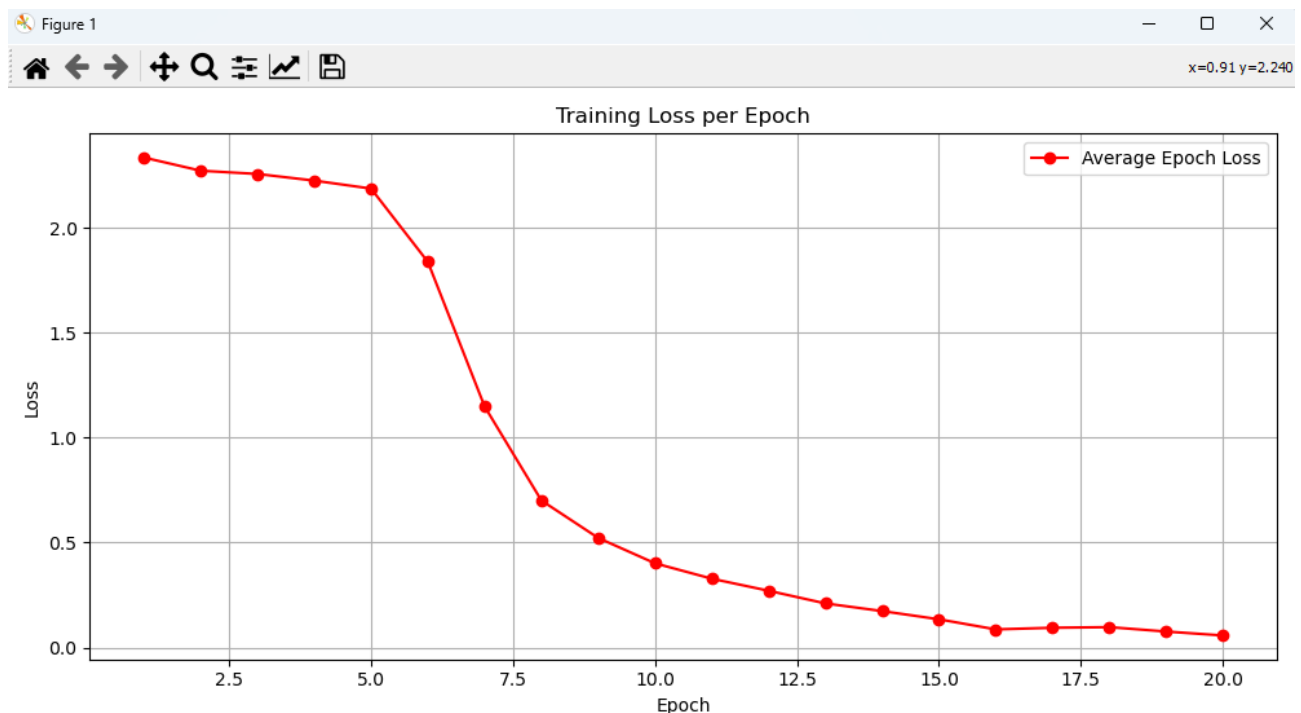


Poiché il dataset è molto grande, ho implementato un metodo denominato "Reduce-Dataset", che mi ha permesso di utilizzare solo una piccola porzione del dataset per velocizzare i test.

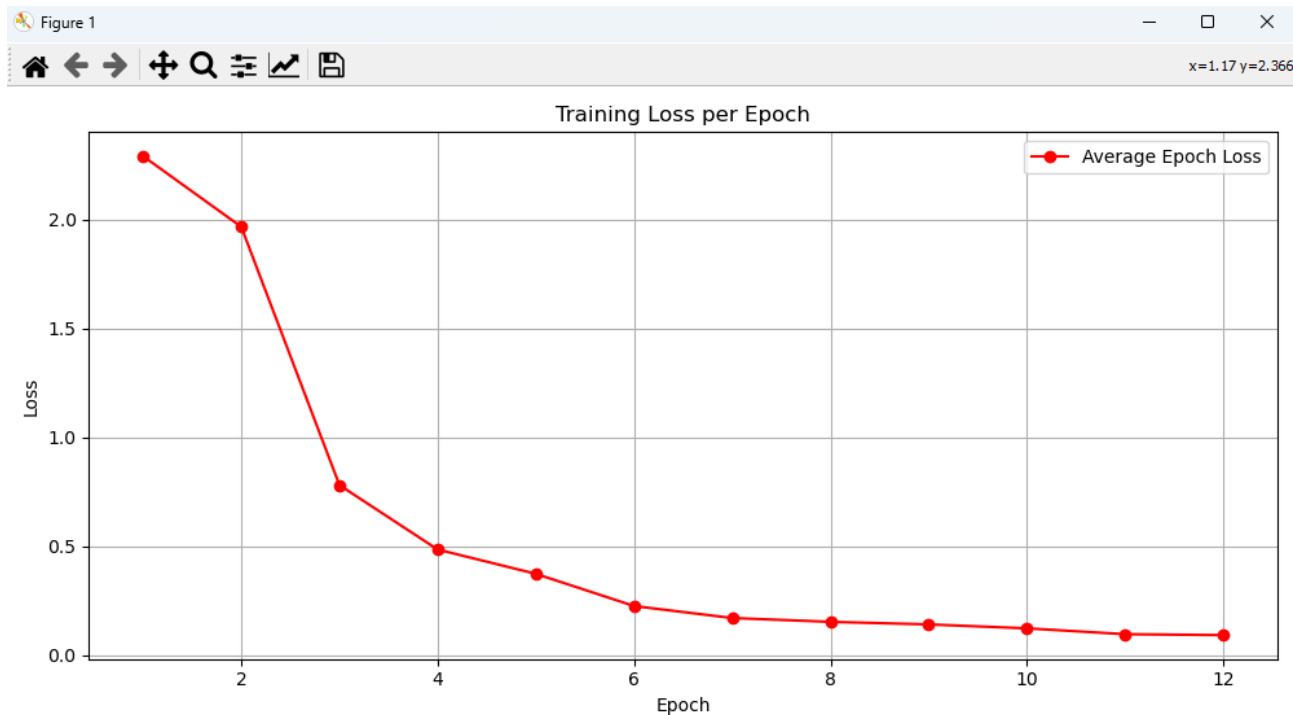
Durante i test, ho sperimentato con vari iper-parametri per ottenere una precisione accettabile durante la fase di valutazione. Nei primi test, con un *batch size* di 32 o 64, ho raggiunto un'accuratezza dell'86% dopo 15-20 epoche.

Successivamente, ho ridotto il *batch size* a 16, osservando una velocità di apprendimento migliore (la *loss* diminuiva nelle prime epoche), anche se l'accuratezza faticava a migliorare.

Ho quindi provato ad aumentare il *learning rate*, ma questo ha peggiorato i valori di *loss* senza portare significativi miglioramenti in termini di accuratezza.

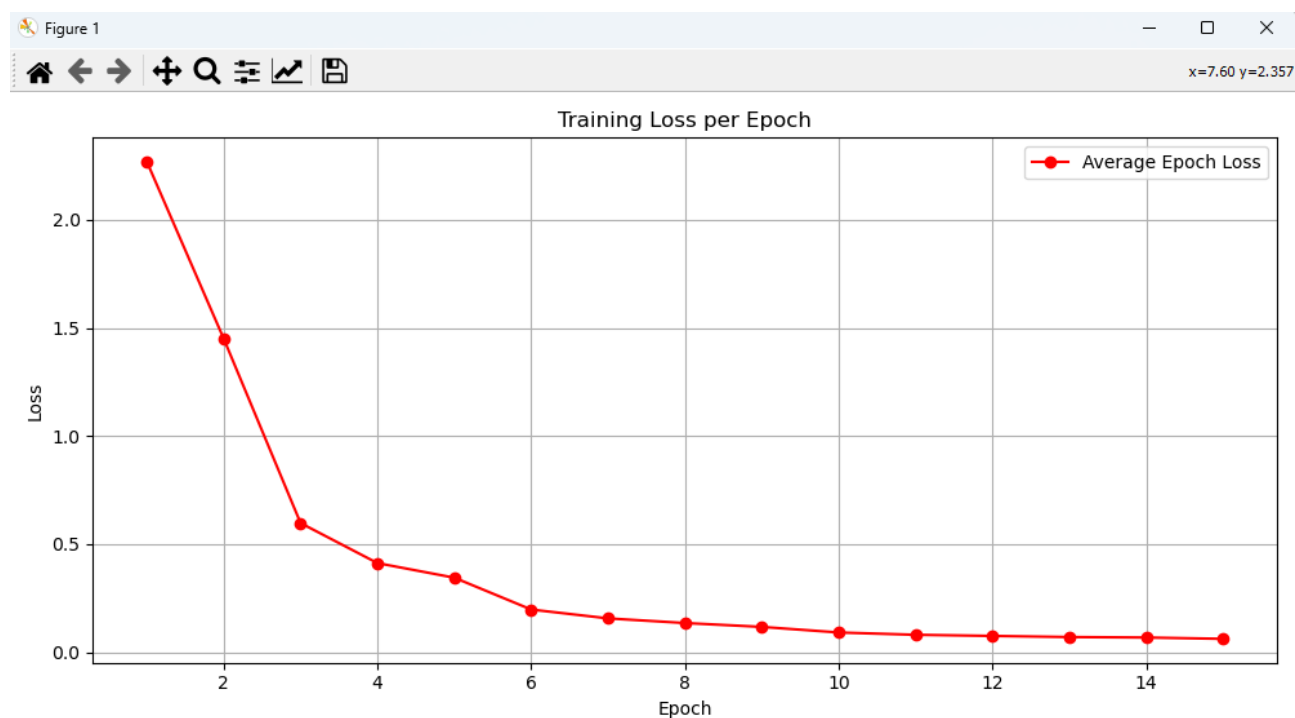


Per ottimizzare ulteriormente il processo di apprendimento, ho implementato uno **scheduler**, che riduce il *learning rate* ogni 5 epoche secondo una percentuale prestabilita.

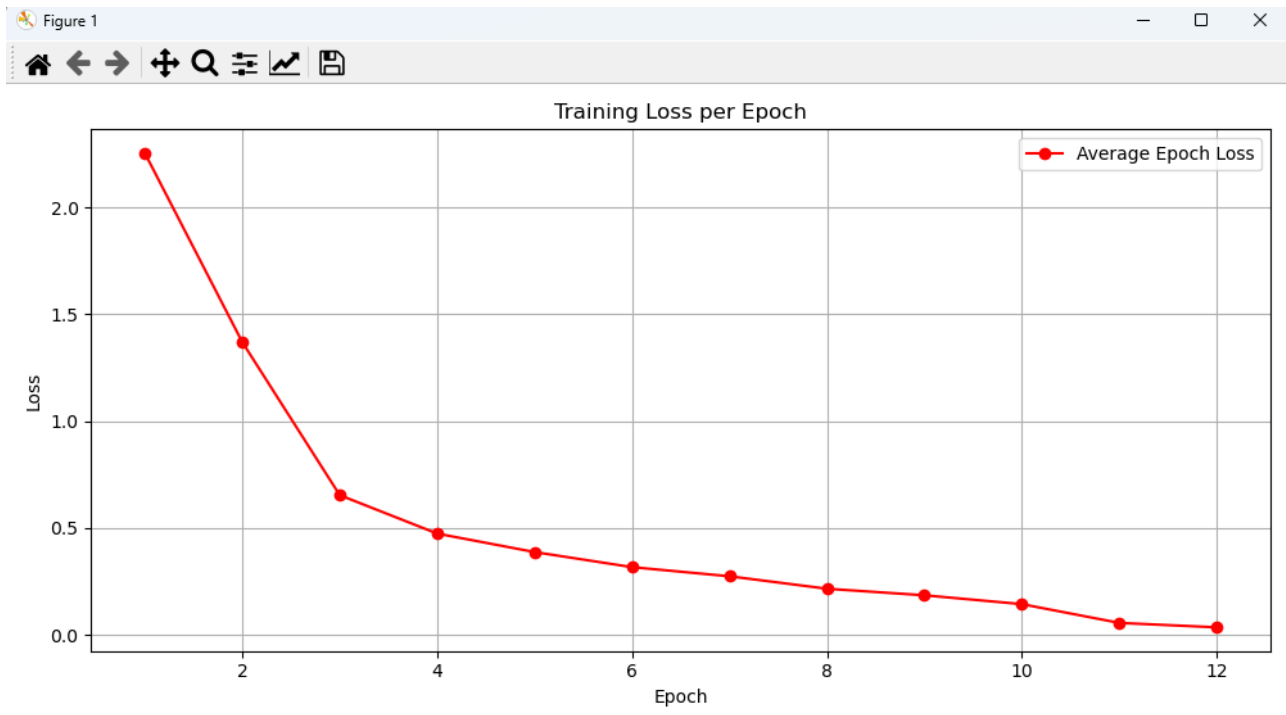


Il valore ottimale del *learning rate* è risultato essere 0.0014. Con questo valore, ho ottenuto buoni risultati iniziali in termini di *loss*, stabilizzandosi intorno alla 12esima epoca. Tuttavia, l'accuratezza ha raggiunto solo circa il 92%.

Sono riuscito ad avere un grafico *loss* migliore tramite l'utilizzo di un *learning rate* di 0.0013

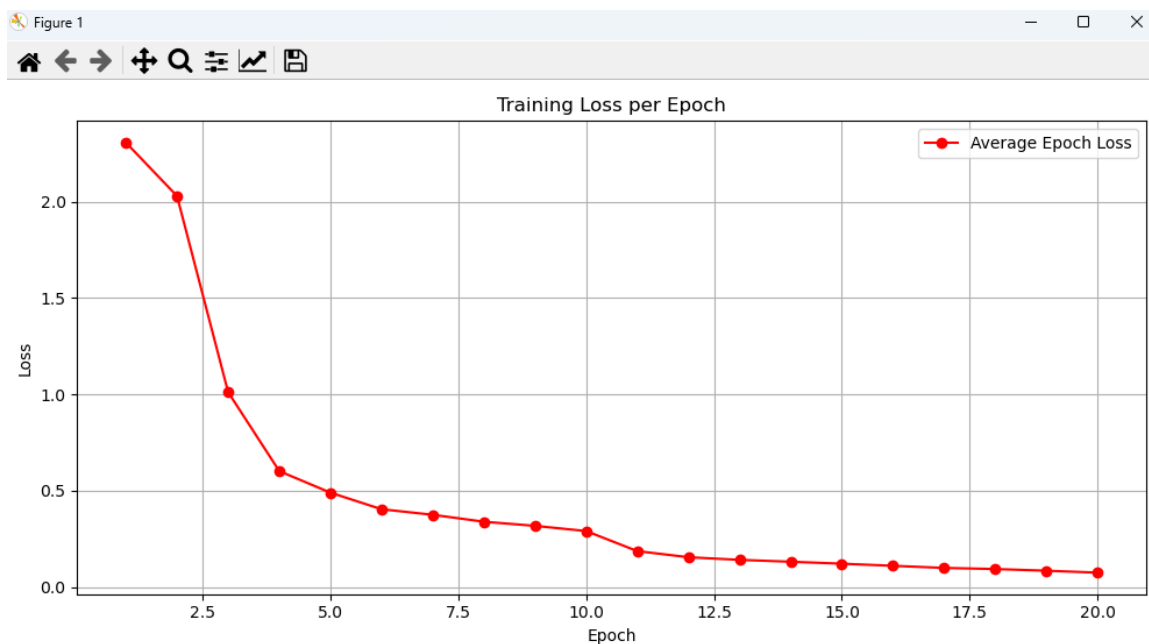


Il miglior risultato ottenuto è stato con un *learning rate* di 0.0013, un *batch size* di 16, e uno *scheduler* con step di 10 e gamma di 0.1. Questo setup è stato l'unico ad avvicinarsi a una *loss* ottimale, con una *loss* media vicina allo zero.



Peccato che non ho potuto né salvare il modello né replicarlo.

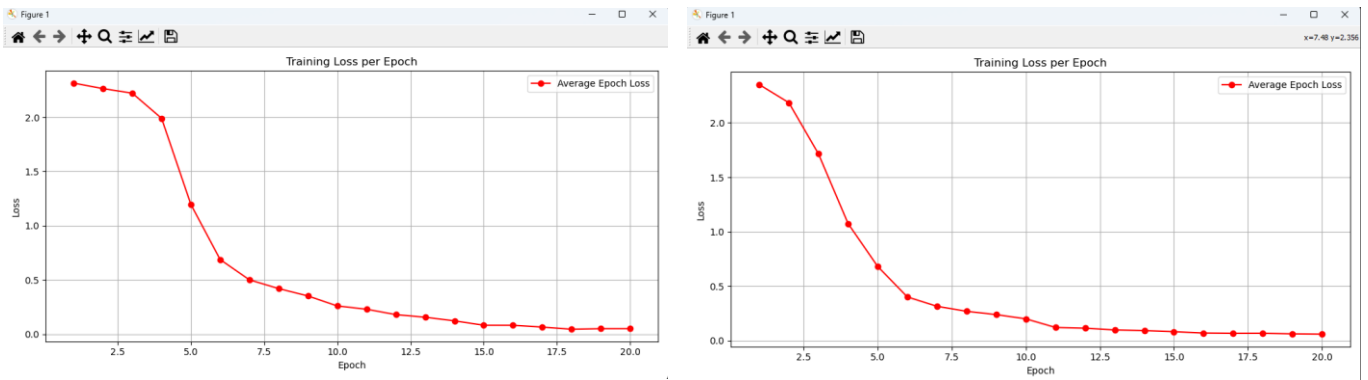
Qui sotto è riportato il grafico del modello utilizzato come modello finale, salvato nella cartella **Out**. I parametri utilizzati sono i seguenti: *batch size* di 16, *learning rate* di 0.0014, *reduce_dataset* di 0.1, *scheduler* con step a 10 e gamma a 0.1.



Problematiche

Durante i vari test, non sono mai riuscito a trovare una soluzione permanente e completamente soddisfacente. È possibile che l'uso dello *scheduler* sia stato inefficace o gestito in modo subottimale, anche se ha portato a lievi miglioramenti in alcuni casi.

Ad esempio, nell'immagine a sinistra è mostrata una variante con 20 epoche, un *batch size* di 16, e lo stesso *learning rate* di 0.002.



Ho riscontrato alcune difficoltà nel replicare un modello di successo ottenuto in precedenza. Nonostante avessi utilizzato le stesse configurazioni, i risultati non erano sempre consistenti.

Probabilmente, la scelta del dataset non è stata ideale. Avrei potuto creare un mio dataset personalizzato, eseguire dei test e introdurre gradualmente delle difficoltà. Ad esempio, utilizzare un modello pre-addestrato per riaddestrarlo ulteriormente, oppure fornire una spiegazione più dettagliata del dataset, arricchita con grafici, e successivamente complicare il dataset stesso.

Miglioramenti attuabili

- **In termini di file e progetto:** Sarebbe utile creare file più autonomi, con salvataggi più efficienti e la possibilità di scegliere quale modello utilizzare per la fase test. Inoltre sarebbe ottimo la creazione automatica di uno storico in formato JSON di tutti i salvataggi.
- **In termini di rete:** Un'opzione potrebbe essere quella di creare una rete più grande, senza modificare troppo gli iperparametri, o valutare un cambio di architettura. Utilizzare un'altra architettura potrebbe aiutarmi a comprendere meglio il problema, anche se la ResNet dovrebbe essere una buona scelta di base.

Informazioni aggiuntive

La struttura del progetto si divide in tre file principali:

- **Net.py:** Contiene la vera e propria struttura neurale.
- **Net_runner.py:** Serve per addestrare e testare la rete.
- **Start.py:** Avvia il ciclo di *train* o *test*, semplificando il processo grazie ai parametri configurabili all'inizio.

Le directory del progetto sono organizzate come segue:

- **Data:** Contiene i dataset in formato .mat. (sarà creata da torchvision, dopo aver scaricato i file il parametro DOWNLOAD può essere messo in false)
- **Out:** Questa directory conterrà l'ultimo modello addestrato, ma nel vostro caso, il modello migliore. Il metodo di test carica il modello da qui. (ATTENZIONE: Se viene addestrato un nuovo modello sovrascriverà quella già presente in out)
- **Images:** Contiene le immagini generate durante alcuni test, tutte rinominate seguendo una logica precisa. Esempio: 20(epoca)x16(batch) 0.1 (% di riduzione) 0.0014 (learning rate) sch5 (scheduler con step a 5) 0.1 (gamma dello scheduler) 92 (% di accuracy) rot/rotation (se era presente la rotazione delle immagini nel dataset).
- **Other_models/Other_img:** Directory dedicate rispettivamente a contenere altri modelli e immagini (non rinominate).