

Modello ResNet per la classificazione del dataset SVHN

Studentessa: Melissa Selmanhaskaj

Ambiente: Conda (anaconda3) – Python

Obiettivo: Classificazione di immagini

Modello: ResNet-31

Introduzione

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.

La rete 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.

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 e 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.

Tecniche utilizzate

- **Ottimizzatore (Adam):** Regola dinamicamente il learning rate per ogni parametro della rete. 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):** Misura la differenza tra le probabilità predette e quelle reali. 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):** Riduce il learning rate ogni 5 epoche moltiplicandolo per 0.1. 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 ($\text{gamma}=0.1$), quindi riduce del 10%. Questo aiuta il modello a convergere meglio man mano che l'allenamento procede.

Modifiche Avanzate Implementate

Durante lo sviluppo del progetto, ho implementato tre importanti modifiche che migliorano significativamente il processo di ricerca e validazione dei risultati:

- **TensorBoard per il tracciamento di esperimenti:** ho integrato TensorBoard per visualizzare in tempo reale l'andamento del training. Questo strumento registra e visualizza la loss di training e validation, l'accuracy della validazione, e il learning rate durante l'addestramento. Grazie a TensorBoard, è possibile monitorare i progressi del modello in real-time e analizzare facilmente i grafici di convergenza del modello, facilitando la diagnosi di eventuali problemi durante il training.

- **Transfer learning con resNet50 pretrained:** ho implementato una versione alternativa del modello sfruttando il Transfer Learning con ResNet50 pretrained su ImageNet, direttamente da `torchvision.models`. Questo approccio permette di caricare i pesi preaddestrati su milioni di immagini e di riutilizzare le feature apprese per il nostro compito specifico. La configurazione è facilmente selezionabile nel `config.json` tramite il flag `"transfer_learning"`, con l'opzione di congelare il backbone o eseguire fine-tuning completo.

- **Riproducibilità degli esperimenti:** per garantire che gli esperimenti siano completamente riproducibili, ho impostato i seed per tutte le fonti di randomicità: seeds per torch, torch.cuda, numpy e la libreria random di Python. Inoltre, ho utilizzato il parametro `random_state` in tutti i metodi di scikit-learn (come `train_test_split`). Questo assicura che gli stessi risultati possano essere ottenuti in esecuzioni successive, fondamentale per la validazione scientifica.

Dataset

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

Esempio:



Poiché il dataset è molto grande, ho anche implementato una funzione **"Reduce-Dataset"** per velocizzare i test utilizzando una porzione ridotta del dataset.

Riproducibilità degli esperimenti

Per garantire la riproducibilità dei risultati e permettere la replicazione degli esperimenti, ho implementato diverse misure di controllo della casualità:

- **Seed fisso:** Ho impostato un seed (valore 42) per tutti i generatori di numeri casuali utilizzati nel progetto, inclusi PyTorch, NumPy, Python random e CUDA. Questo viene fatto attraverso la funzione `set seeds()` che viene chiamata all'inizio di ogni esecuzione.
- **Divisione deterministica del dataset:** La divisione tra training e validation set viene effettuata utilizzando il parametro `random state=seed` nella funzione `train_test_split` di *scikit-learn*, garantendo che la suddivisione sia sempre identica.
- **DataLoader deterministico:** Ho configurato i DataLoader di PyTorch con un generatore seeded (`generator=torch.Generator().manual_seed(seed)`), assicurando che l'ordine di presentazione dei batch sia riproducibile.

Queste misure consentono di ottenere risultati consistenti eseguendo gli stessi esperimenti con le stesse configurazioni, facilitando il debugging e la validazione del modello.

Esperimenti e Risultati

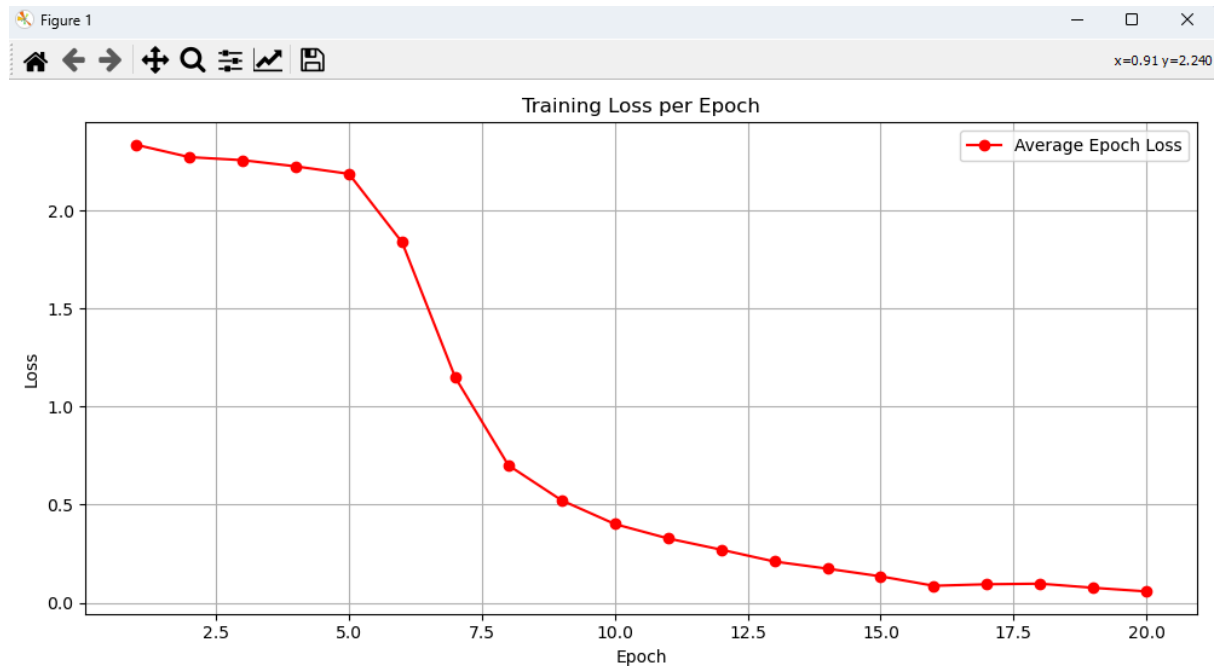
Durante i test ho sperimentato con vari iper-parametri:

- Con batch size di 32 o 64 ho raggiunto un'accuratezza dell'86% dopo 15-20 epoche.
- Riducendo il batch size a 16 ho osservato una velocità di apprendimento migliore, (la loss diminuiva nelle prime epoche) ma accuratezza meno stabile che faticava a

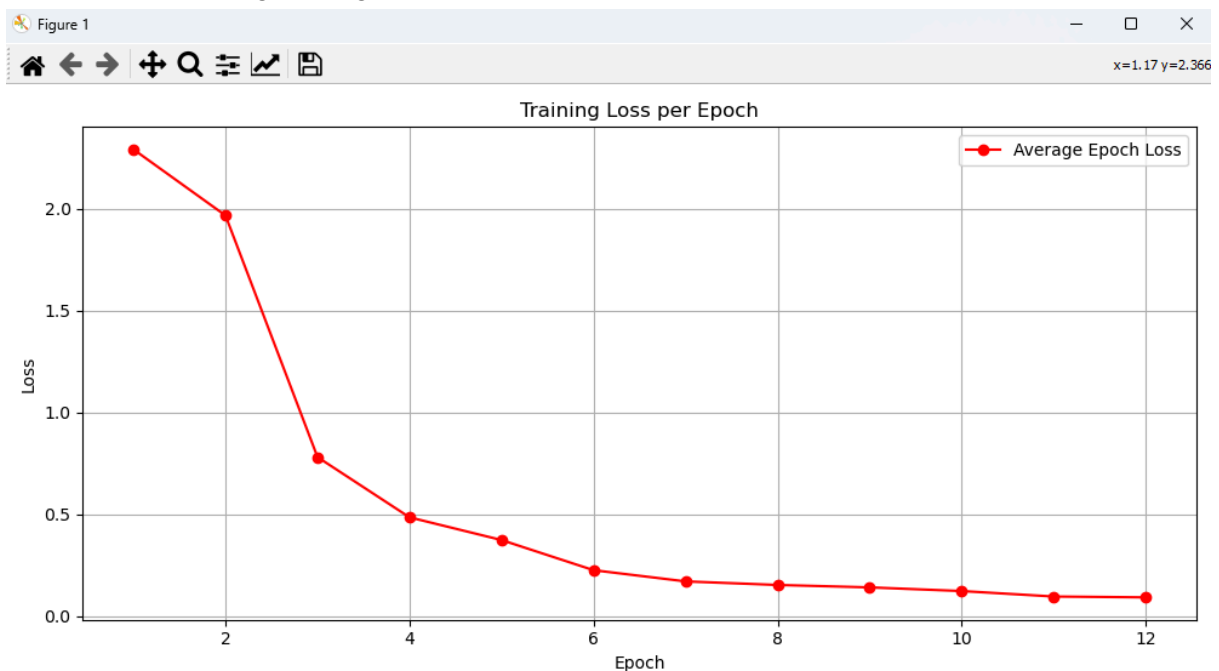
migliorare.

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

Il miglior setup ha incluso: learning rate di **0.0013**, batch size di **16**, scheduler con **step 10** e **gamma 0.1**, ottenendo una **loss vicina allo zero** e una **accuratezza del 92%**.



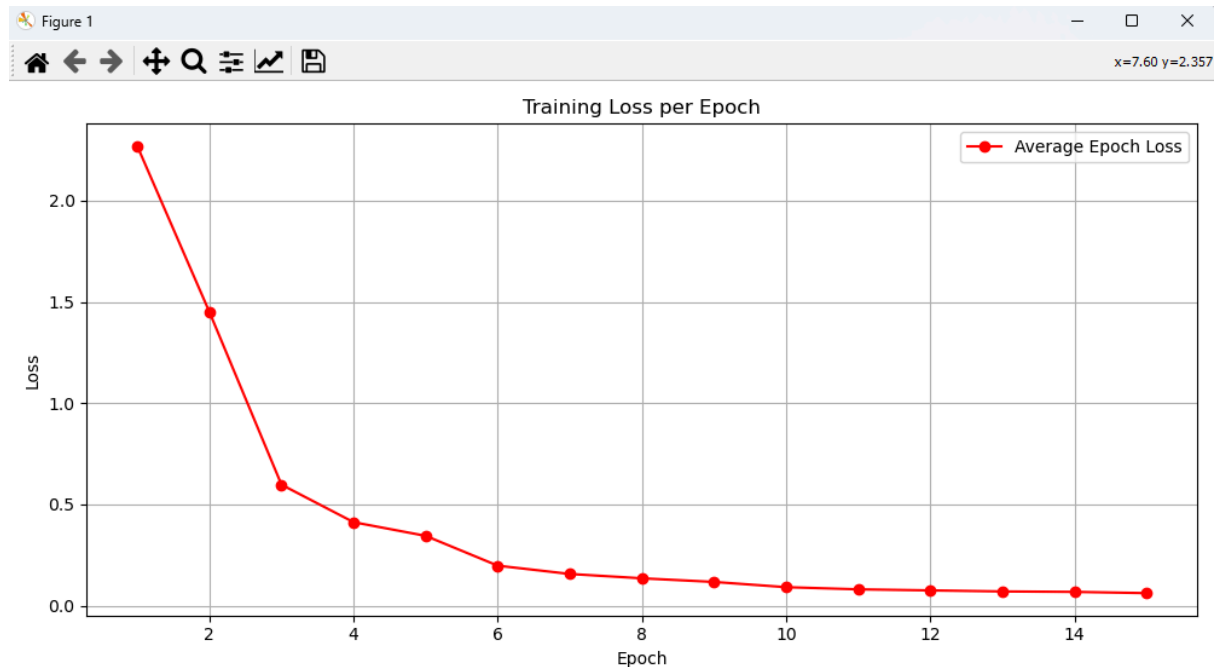
Per ottimizzare ulteriormente il processo di apprendimento, ho implementato uno scheduler, che riduce il learning rate ogni 5 epoche secondo una percentuale prestabilita.



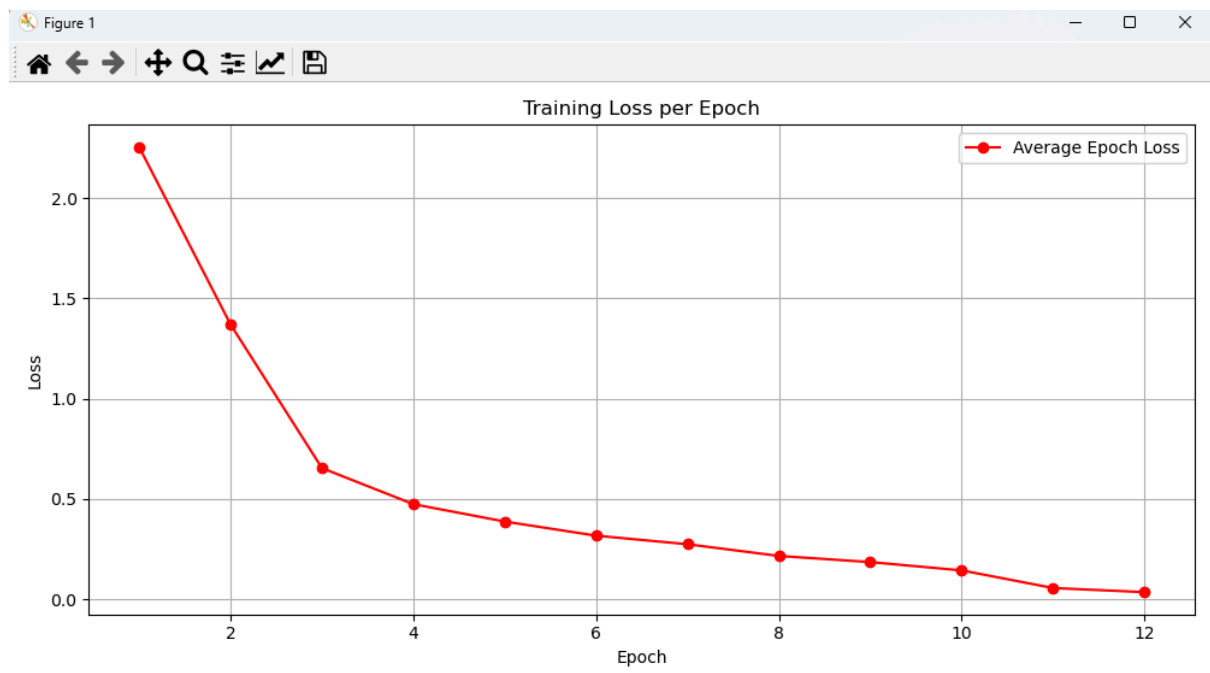
Il valore ottimale del learning rate è risultato essere 0.0014. Con questo valore, ho ottenuto buoni risultati iniziali in termini di loss, stabilizzandosi intorno alla 12esima epoca.

Tuttavia, l'accuratezza ha raggiunto solo circa il 92%.

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

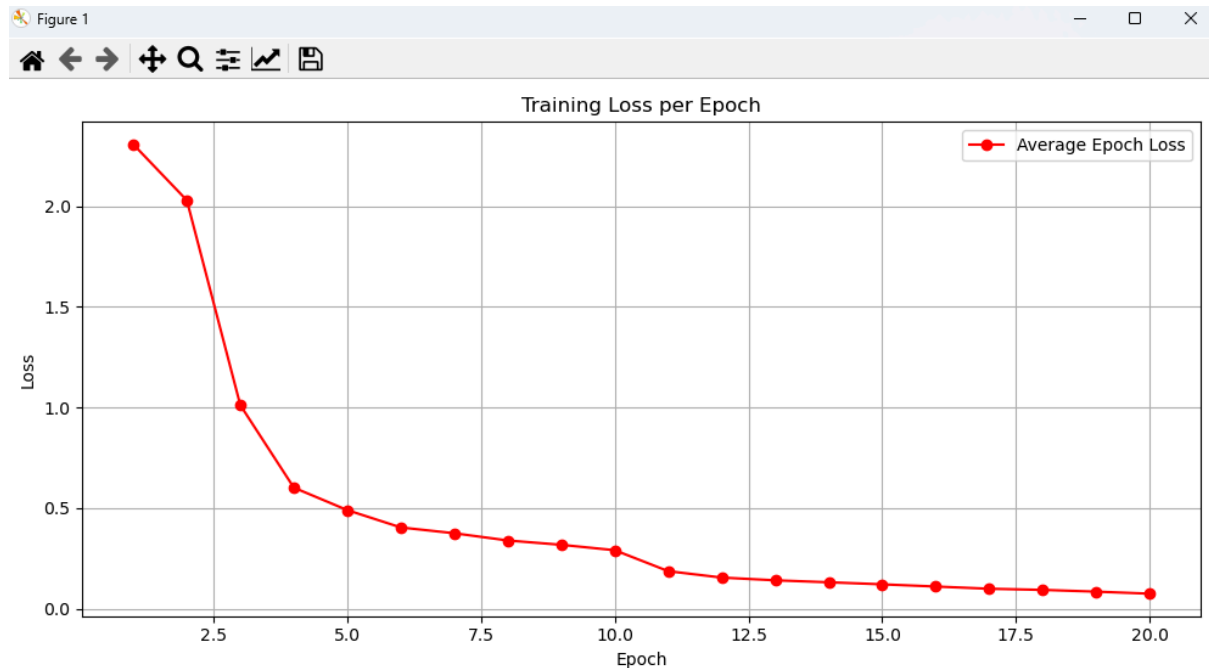


Il miglior risultato ottenuto è stato con un learning rate di 0.0013, un batch size di 16, e uno scheduler con step di 10 e gamma di 0.1. Questo setup è stato l'unico ad avvicinarsi a una loss ottimale, con una loss media vicina allo zero.



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

Qui sotto è riportato il grafico del modello utilizzato come modello finale. 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.

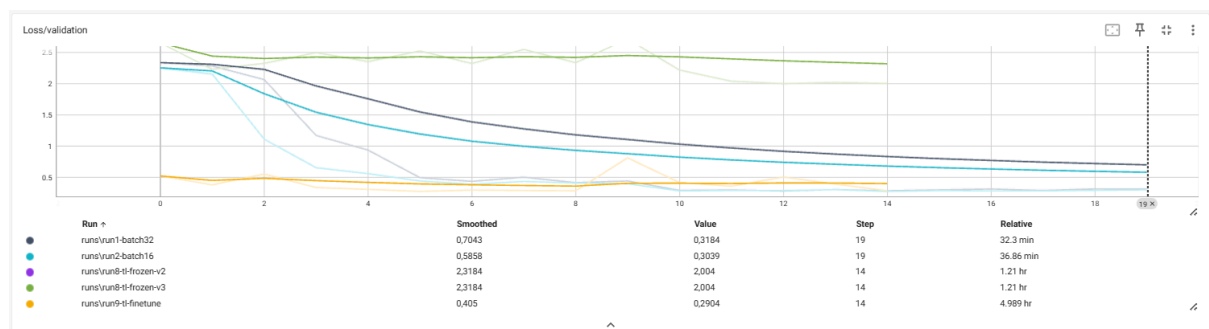


Monitoraggio con TensorBoard

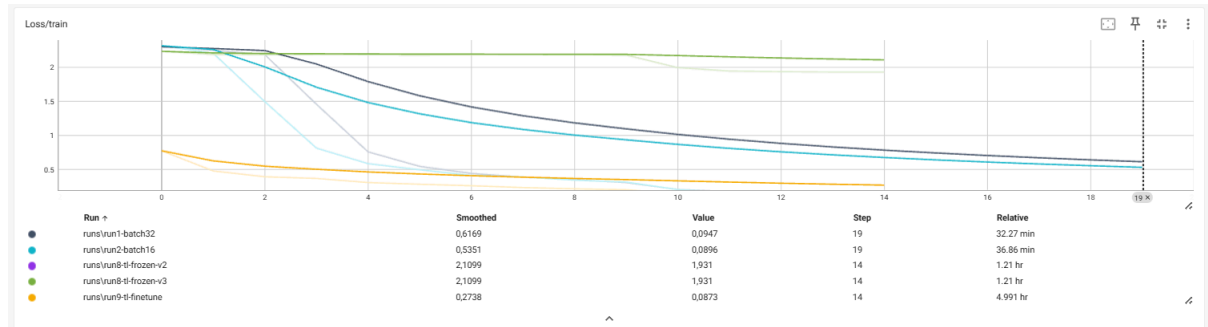
Per tenere traccia dell'andamento dell'addestramento e analizzare le metriche di performance, ho integrato TensorBoard nel progetto. TensorBoard è uno strumento di visualizzazione che permette di monitorare in tempo reale l'evoluzione delle metriche durante il training.

Metriche tracciate:

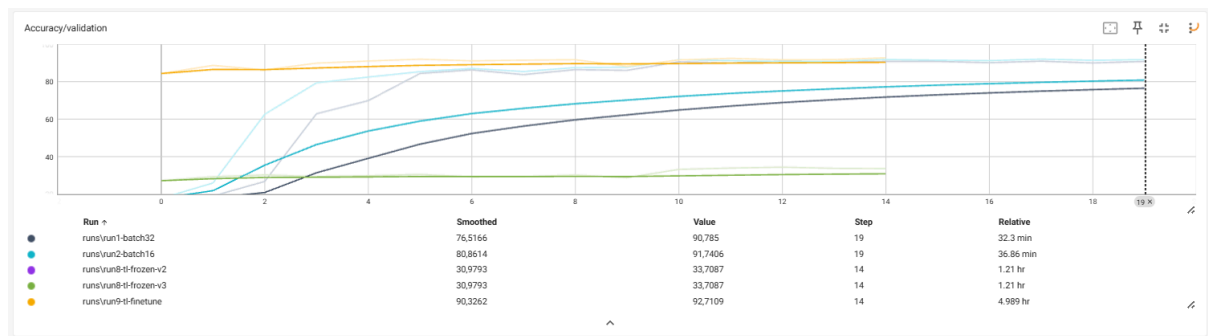
- **Loss di training:** Monitora la perdita calcolata sui dati di training ad ogni epoca, permettendo di osservare come il modello riduce l'errore sui dati visti durante l'addestramento.



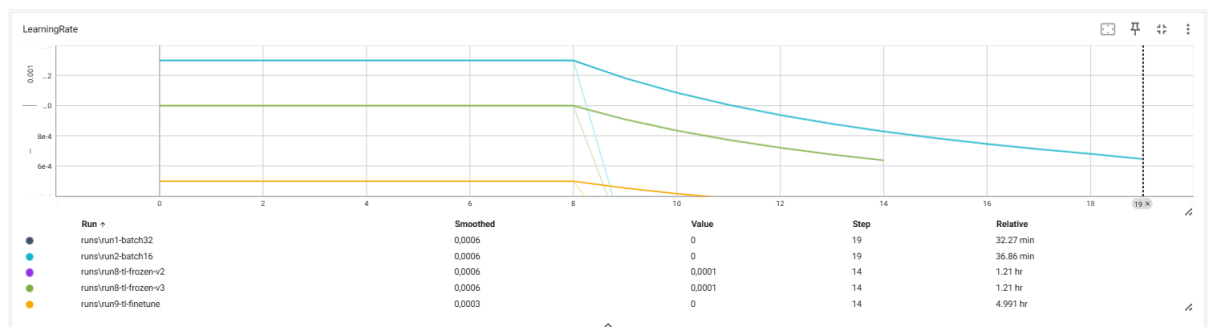
- **Loss di validation:** Traccia la perdita sui dati di validation, fondamentale per individuare eventuali fenomeni di overfitting (quando il modello impara troppo bene i dati di training ma generalizza male su dati nuovi).



- **Accuratezza di validation:** Registra la percentuale di predizioni corrette sui dati di validation, fornendo una misura diretta delle prestazioni del modello.



- **Learning Rate:** Visualizza l'andamento del learning rate durante l'addestramento, particolarmente utile quando si utilizza uno scheduler che modifica dinamicamente questo valore.



Implementazione: Ho utilizzato *SummaryWriter di PyTorch*, che salva automaticamente le metriche in file di log nella *directory runs*. Ogni esperimento viene salvato in una sottocartella dedicata (es. run1-batch32, run2-batch16, run8-tl-frozen, run9-tl-finetune), permettendo di confrontare facilmente diverse configurazioni.

Utilizzo: Per visualizzare i grafici, è sufficiente eseguire il comando `python -m tensorboard.main --logdir=logs` e aprire il browser all'indirizzo indicato. Questo strumento è stato fondamentale per identificare il miglior setup di iper-parametri e per comprendere il comportamento del modello durante l'addestramento.

Esperimenti con Transfer Learning

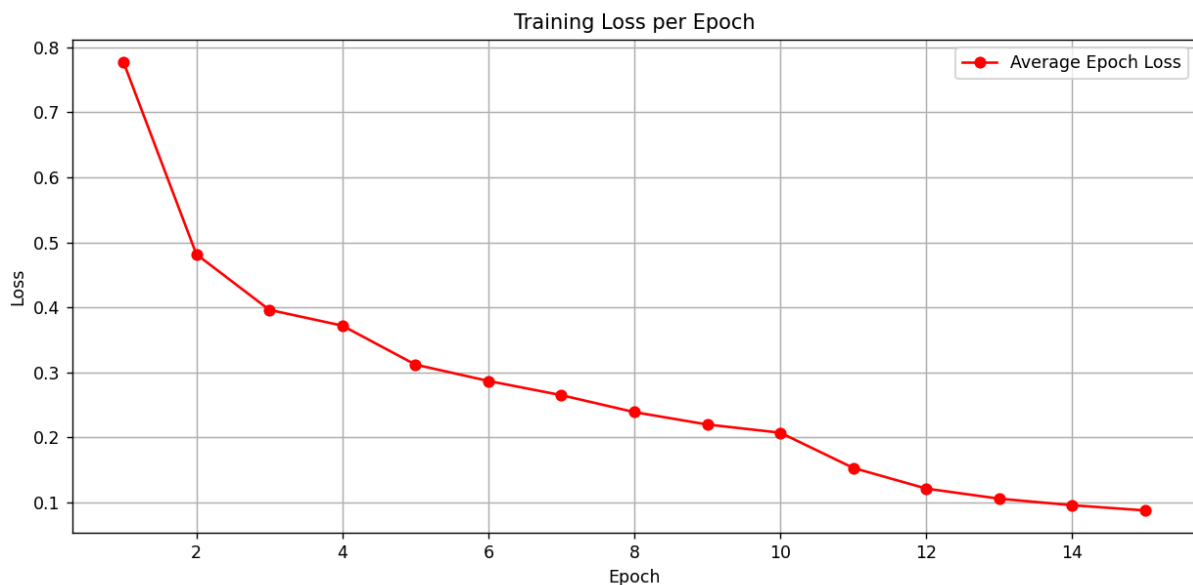
Oltre agli esperimenti con la ResNet custom addestrata da zero, ho esplorato l'utilizzo del **Transfer Learning utilizzando una ResNet-50 pre-addestrata su ImageNet**. Il Transfer Learning è una tecnica che consente di sfruttare le conoscenze acquisite da un modello su un dataset ampio (ImageNet con milioni di immagini) e riutilizzarle per un compito diverso (classificazione di cifre SVHN).

Esperimento 1 - Backbone Congelato (Run 8): Nel primo esperimento ho utilizzato la ResNet-50 pre-addestrata mantenendo tutti i layer convoluzionali congelati (parametro *freeze_backbone=True*), addestrando solamente il layer finale fully-connected adattato alle 10 classi del dataset SVHN.

- **Configurazione:** Learning rate 0.001, batch size 16, 12 epoche (early stopping attivato)
- **Risultati:** Validation accuracy ~30.6%, validation loss ~2.18
- **Analisi:** I risultati sono stati deludenti. Il backbone congelato, addestrato su immagini naturali di ImageNet, non è riuscito ad estrarre feature significative per le cifre SVHN. Le feature apprese su oggetti, animali e scene naturali non si trasferiscono efficacemente al riconoscimento di cifre stilizzate su sfondi urbani.

Esperimento 2 - Fine-Tuning Completo (Run 9): Nel secondo esperimento ho sbloccato l'intero modello (parametro *freeze_backbone=False*), permettendo a tutti i layer di adattarsi al nuovo compito tramite fine-tuning.

- **Configurazione:** Learning rate 0.0005 (ridotto per evitare di "dimenticare" le conoscenze pre-addestrate), batch size 16, 15 epoche, dataset completo (senza riduzione)
- **Risultati:** Validation accuracy ~92.7%, validation loss ~0.29, training loss ~0.087
- **Analisi:** Il fine-tuning ha permesso alla rete di adattare progressivamente le feature estratte dai layer convoluzionali al dominio specifico delle cifre SVHN. L'accuratezza raggiunta è comparabile e leggermente superiore a quella del modello custom addestrato da zero, dimostrando che il Transfer Learning, se applicato correttamente, può essere efficace anche su domini visivamente diversi.



Conclusioni sul Transfer Learning:

Gli esperimenti hanno dimostrato che il semplice utilizzo di un backbone congelato non è sufficiente quando il dominio target (cifre SVHN) è molto diverso dal dominio sorgente (ImageNet).

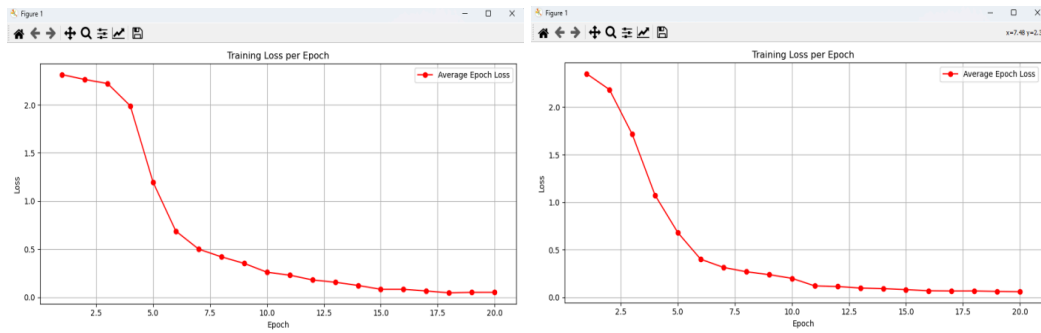
Il fine-tuning completo permette al modello di adattarsi efficacemente, raggiungendo performance eccellenti (~92.7%) comparabili al modello custom.

Il Transfer Learning può essere una strategia vincente quando si dispone di dataset di dimensioni limitate o risorse computazionali ridotte, a patto di permettere l'adattamento dei pesi pre-addestrati.

Problematiche riscontrate

- **Incoerenza tra diversi test anche con stesse configurazioni.** Durante i vari test, non sono mai riuscita 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.



- **Difficoltà iniziale nel replicare un modello di successo.** Nonostante utilizzassi le stesse configurazioni, i risultati non erano sempre consistenti.
- **La gestione dello scheduler a volte non ha portato reali miglioramenti.** 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 codice:** automatizzare il salvataggio dei modelli, scegliere quale modello testare, creare uno storico in JSON. Sarebbe utile creare file più autonomi, con salvataggi più efficienti e la possibilità di scegliere quale modello utilizzare per la fase test. Inoltre sarebbe ottima la creazione automatica di uno storico in formato JSON di tutti i salvataggi.
- **In termini di rete:** valutare reti più profonde o architetture alternative, come DenseNet o MobileNet. 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.

Struttura del progetto

La struttura del progetto si divide in tre file principali:

- **Net.py:** Contiene la vera e propria struttura neurale (ResNet custom e ResNetTransfer).
- **Net_runner.py:** Serve per addestrare e testare la rete, gestendo anche il logging su TensorBoard.
- **Start.py:** Avvia il ciclo di train o test, semplificando il processo grazie ai parametri

configurabili all'inizio. Implementa i seed per la riproducibilità.

- Config.json: Contiene tutti i parametri configurabili (learning rate, batch size, numero di epoche, scheduler, early stopping, transfer learning).

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).
- Logs/runs: Contiene i file di TensorBoard per la visualizzazione dei grafici.
- Other_models/Other_img: Directory dedicate rispettivamente a contenere altri modelli e immagini (non rinominate).

Conclusioni

Il progetto ha dimostrato che una **ResNet-31** può classificare efficacemente le cifre del dataset SVHN, raggiungendo un'accuratezza del **92%**. Le tre modifiche implementate (TensorBoard, Transfer Learning e Riproducibilità) hanno significativamente migliorato il processo di ricerca e validazione dei risultati.

La riproducibilità garantita dai seed ha risolto i problemi di incoerenza nei test precedenti e l'integrazione di TensorBoard ha permesso un monitoraggio dettagliato del processo di addestramento, facilitando l'identificazione delle configurazioni ottimali.

Rimangono margini di miglioramento, in particolare nell'utilizzo del Transfer Learning e nella sperimentazione di architetture alternative. Per il futuro si suggerisce l'utilizzo estensivo dell'approccio Transfer Learning e l'introduzione di dataset più complessi o personalizzati.