***Relazione progetto di Laboratorio di Ottimizzazione, Intelligenza Artificiale e Machine Learning***

A cura di Cacchi Alessandro

Matricola: 001080687

E-mail istituzionale: [alessandro.cacchi@studio.unibo.it](mailto:alessandro.cacchi@studio.unibo.it)

Il progetto presentato è lo sviluppo di un’architettura di rete neurale volta a risolvere un problema di classificazione multi-classe. Utilizza un'architettura di rete neurale con modelli pre-addestrati (ResNet50 e EfficientNet) e tecniche di preprocessing per migliorare le performance.

**\_Dataset\_**

In questo caso è stato creato un sistema di classificazione delle immagini in quattro categorie principali:

* **buildings 🡪** edifici e palazzi
* **mountain 🡪** paesaggi di montagna
* **sea 🡪** paesaggi di mare
* **street 🡪** strade/vie anche residenziali

**Bilanciamento del dataset:** Ho fatto un controllo preliminare per verificare il bilanciamento tra le classi, per evitare che il modello si concentri su una categoria a scapito delle altre.

In questo caso, come si può notare dal grafico a barre prodotto, il dataset è perfettamente bilanciato:

Immagine che contiene testo, schermata, Rettangolo, diagramma

Descrizione generata automaticamente

**\_Data Trasforms \_**

**Preprocessing delle immagini:** Ho preprocessato le immagini tramite le trasformazioni fornite dalla libreria **torchvision**.

* **Ridimensionamento**: ho scalato tutte le immagini a una dimensione di 224x224 pixel.
* **Normalizzazione**: ho scalato i valori dei pixel usando le medie e deviazioni standard di ImageNet, per allinearsi alle reti preaddestrate.

Ho anche applicato tecniche di **data augmentation** per migliorare le capacità del modello di generalizzare su immagini nuove e non viste. Le trasformazioni come **flipping orizzontale**, **rotazione casuale** e **alterazioni di luminosità e colore** (con *ColorJitter*) hanno aiutato a simulare vari scenari reali.

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

Tecniche utilizzate:

* **Flipping orizzontale (*RandomHorizonalFlip*):** 
  + Viene applicato con una probabilità del **50%** per ribaltare orizzontalmente le immagini.
* **Rotazione casuale (*RandomRotation*):**
  + Introduce rotazione casuale con un angolo massimo di 15 gradi.
* **Variazioni di colore (*ColorJitter*):**
  + Introduce variazioni casuali di **Luminosità**, **contrasto**, **saturazione** e **tonalità**.
* **Ridimensionamento (*Resize*):**
  + ho scalato tutte le immagini a una dimensione di 224x224 pixel.
* **Normalizzazione dei pixel (*Normalize*):**
  + ho scalato i valori dei pixel usando le medie e deviazioni standard di ImageNet, per allinearsi alle reti preaddestrate.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Test Split**: La funzione utilizza train\_test\_split di scikit-learn per separare una porzione di dati per il test set (test\_size). La suddivisione è stratificata rispetto alla classe delle etichette *- stratify=labels['class'] -* per garantire che la distribuzione delle classi rimanga uniforme.
* **Train/Validation Split**: Divide il rimanente set di dati (train/validation) in due parti, secondo la proporzione specificata da *- val\_size / (1 - test\_size) –*

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Dataset Creation**:
  + Crea tre oggetti PlaceDataset (dataset personalizzato) per training, validation e test, applicando le trasformazioni appropriate.
* **DataLoader Creation**:
  + Crea un DataLoader per ciascun dataset con *-* **batch\_size** *–* (numero di campioni per batch) e *–* **shuffle** *–* (mescola i dati durante il training, ma non durante la validazione o il test).

**\_Modelli\_**

Nel progetto ho implementato e confrontato due modelli di deep learning: **ResNet-50** e **EfficientNet-B0**, entrambi preaddestrati su ImageNet.

Nel training dei modelli ho deciso di inserire una tecnica di early stopping per interrompere il training nel momento in cui il modello non mostra più miglioramenti sulla perdita di validazione per un numero consecutivo di epoche definito dal parametro patience.

Quindi, dopo che sono stati calcolati la perdita e l'accuratezza sul dataset di validazione, viene controllato se la perdita di validazione dell'epoca corrente (val\_loss) è migliorata rispetto alla migliore perdita registrata finora (best\_loss).

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Risultati del training dei due modelli**  
**ResNet 50**

Epoch 1/50, Train Loss: 1.0851, Train Acc: 58.33%, Val Loss: 0.7460, Val Acc: 75.00%

Salvato miglior modello: ResNet50\_best.pth

Epoch 2/50, Train Loss: 0.4653, Train Acc: 91.67%, Val Loss: 0.4087, Val Acc: 83.93%

Salvato miglior modello: ResNet50\_best.pth

Epoch 3/50, Train Loss: 0.2026, Train Acc: 95.24%, Val Loss: 0.3307, Val Acc: 87.50%

Salvato miglior modello: ResNet50\_best.pth

………………………………………………Epoch dalla 4 alla 8………………………………………….………….

Epoch 9/50, Train Loss: 0.1015, Train Acc: 97.62%, Val Loss: 0.2933, Val Acc: 94.64%

Early stopping triggered.

Salvato ultimo modello: ResNet50\_last.pth

**EfficientNet:**

Epoch 1/50, Train Loss: 1.3132, Train Acc: 38.69%, Val Loss: 1.2621, Val Acc: 46.43%

Salvato miglior modello: EfficientNet\_best.pth

Epoch 2/50, Train Loss: 1.1466, Train Acc: 63.69%, Val Loss: 1.1130, Val Acc: 75.00%

Salvato miglior modello: EfficientNet\_best.pth

Epoch 3/50, Train Loss: 1.0024, Train Acc: 80.36%, Val Loss: 0.9931, Val Acc: 76.79%

Salvato miglior modello: EfficientNet\_best.pth

………………………………………………Epoch dalla 4 alla 32……………………………………….………….

Epoch 33/50, Train Loss: 0.1048, Train Acc: 98.81%, Val Loss: 0.2538, Val Acc: 91.07%

Salvato miglior modello: EfficientNet\_best.pth

………………………………………………Epoch dalla 34 alla 37…………………………………….………….

Epoch 38/50, Train Loss: 0.0318, Train Acc: 100.00%, Val Loss: 0.2640, Val Acc: 92.86%

Early stopping triggered.

Salvato ultimo modello: EfficientNet\_last.pth

Come possiamo vedere

* **ResNet50:** Dopo un rapido miglioramento iniziale, la **Val Loss** si stabilizza intorno a 0.2674 (epoca 4), ma successivamente aumenta leggermente. Questo porta a un early stopping all'epoca 9, quando la **Val Loss** è 0.2933 e la **Val Acc** è 94.64%. La stabilità del training è buona, ma l'early stopping viene attivato rapidamente poiché il modello non migliora ulteriormente.
* **EfficientNet:** Il training è più lungo e la **Val Loss** continua a migliorare fino all'epoca 33, quando raggiunge 0.2538. Il modello mostra una maggiore stabilità nel mantenere alte performance nelle epoche successive, anche se non migliora sempre in modo consistente. L'early stopping viene attivato all'epoca 38, indicando che il modello ha avuto più tempo per perfezionarsi.

Ho anche configurato il **config.json** con vari parametri, tra questi:

* **batch\_size** determina il numero di campioni elaborati in ciascun passaggio durante il training, influenzando sia la velocità di addestramento che la stabilità dell'ottimizzazione.
* Il numero di **epochs** stabilisce il massimo numero di iterazioni sull'intero dataset, ma l'**early stopping** garantisce che non tutte le epoche vengano necessariamente utilizzate, fermando il training quando non ci sono miglioramenti significativi.
* **learning\_rate** controlla la velocità con cui il modello aggiorna i suoi pesi durante l'ottimizzazione.
* **patience** indica il numero massimo di epoche consecutive senza miglioramenti sulla perdita di validazione prima di attivare l'early stopping e interrompere il training.
* Per regolare dinamicamente il learning rate, vengono utilizzati **step\_size** e **gamma**, che insieme definiscono ogni quanto tempo e di quanto il learning rate viene ridotto, contribuendo a una convergenza più stabile verso il minimo globale della funzione di perdita.
* La divisione del dataset, controllata da **test\_size** e **val\_size**, è fondamentale per valutare le performance del modello. La suddivisione in training, test e validazione garantisce che il modello venga addestrato su un insieme di dati e valutato su un altro, preservando la capacità di generalizzazione.
* **random\_state** garantisce che i risultati siano riproducibili, mantenendo costanti le suddivisioni del dataset e il comportamento di altre operazioni randomiche.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**\_Valutazione dei modelli\_**

Tramite sklearn ho generato il report con la funzione *classification\_report*.

**ResNet**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | recall | F1-score | support |
| Sea | 1.00 | 0.79 | 0.88 | 14 |
| Street | 0.76 | 0.93 | 0.84 | 14 |
| Buildings | 0.92 | 0.86 | 0.89 | 14 |
| Mountain | 0.93 | 1.00 | 0.97 | 14 |
|  |  |  |  |  |
| Accuracy |  |  | 0.89 | 56 |
| Macro avg | 0.91 | 0.89 | 0.89 | 56 |
| Weighted avg | 0.91 | 0.89 | 0.89 | 56 |

**EfficientNet**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | recall | F1-score | support |
| Sea | 1.0 | 0.93 | 0.96 | 14 |
| Street | 0.92 | 0.86 | 0.89 | 14 |
| Buildings | 0.87 | 0.93 | 0.90 | 14 |
| Mountain | 0.93 | 1.00 | 0.97 | 14 |
|  |  |  |  |  |
| Accuracy |  |  | 0.93 | 56 |
| Macro avg | 0.93 | 0.93 | 0.93 | 56 |
| Weighted avg | 0.93 | 0.93 | 0.93 | 56 |

* **Precision:** Indica la percentuale di predizioni corrette tra tutte le predizioni fatte per una determinata classe.

In entrambi i casi nella classe “sea”, la precisione è 1.00, il che significa che tutte le immagini predette come "sea" erano effettivamente corrette.

* **Recall:** Misura la capacità del modello di trovare tutte le istanze di una determinata classe (sensibilità).

Nel caso di ResNet, nella classe "sea", il recall è 0.79, il che significa che il modello ha trovato il 79% di tutte le immagini effettivamente appartenenti alla classe "sea";

Mentre con EfficientNet si ha un recall di 0,86, leggermente incrementato rispetto a ResNet

* **F1-score:** È la media armonica tra precision e recall, utile quando c'è uno squilibrio tra di loro.

Nel caso di ResNet la classe "sea" ha un F1-score di 0.88, che è una misura bilanciata delle sue performance.

Mentre con EfficientNet si ha un F1-score di 0,96, leggermente incrementato rispetto a ResNet

* **Support:** Indica il numero totale di esempi reali per ciascuna classe.

tutte le classi hanno un supporto di 14, il che significa che ci sono 14 immagini per ciascuna classe nel set di test.

* **Macro avg**: EfficientNet ottiene una macro-media più alta in tutte le metriche (0.93 contro 0.91 per precision, e 0.93 contro 0.89 per recall e F1-score). Questo indica che EfficientNet è più consistente nelle prestazioni tra tutte le classi.
* **Weighted avg**: Anche in termini di media ponderata, EfficientNet ottiene punteggi superiori (0.93 contro 0.89), confermando che gestisce meglio la distribuzione dei dati nel dataset.

**EfficientNet** dimostra una maggiore capacità di generalizzazione rispetto a **ResNet50**, specialmente nelle classi con maggiore difficoltà come "Sea" e "Buildings". Sebbene ResNet abbia mostrato buone prestazioni complessive, EfficientNet risulta più accurato.

**\_Confusion Matrix\_**

Un’altra valutazione che ho svolto è stata quella di calcolare la **confusion matrix** tramite la funzione confusion\_matrix sempre di **sklearn**. Viene generata una **heatmap** con le predizioni del modello sull’asse delle x e le etichette reali sull’asse delle y. Con essa posso identificare le classi più confuse del modello, evidenziando errori frequenti

**ResNet:**

Immagine che contiene schermata, Rettangolo, diagramma, quadrato

Descrizione generata automaticamente

**EfficientNet:**

Immagine che contiene schermata, Rettangolo, diagramma, quadrato

Descrizione generata automaticamente

Come si può vedere **EfficientNet** mostra un miglioramento rispetto a ResNet50, specialmente per la classe **sea**, con una riduzione degli errori (da 3 errori a 1).

La classe **mountain** viene classificata perfettamente con entrambi i modelli.

ResNet50 ottiene presstazioni leggermente migliori per la classe street, con un solo errore contro i 2 di EfficientNet.

EfficientNet sembra avere un vantaggio in termini di precisione complessiva, specialmente per la classe "sea" e "buildings".

**\_immagini dimostrative delle predizioni\_**

**\_Mayn.py\_**

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteIl codice che gestisce l’addestramento e la valutazione dei due modelli è “diviso” in più parti. Iniziando con il caricamento e la validazione delle configurazioni:

* Ho utilizzato file di configurazione (**config.json**) e uno schema di validazione (**config\_schema.json**) per centralizzare i parametri.
* La validazione tramite **config\_schema.json** assicura che i parametri rispettino le specifiche attese.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteSuccessivamente ho impostato un seed per la riproducibilità. Infatti, il seed garantisce che gli esperimenti siano ripetibili, così da evitare risultati casuali che avrebbero potuto confondere l’analisi

* **Trasformazioni:** Definisce trasformazioni per dati di addestramento e test.
* **Suddivisione:** Divide il dataset in training, validation e test.
* **Dataloader:** Crea i dataloader per iterare sui dati.

Le trasformazioni (get\_transforms) standardizzano i dati, migliorando l’apprendimento del modello.

La suddivisione (split\_data) separa i dati in modo controllato per evitare contaminazioni tra training e test.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Configurazione di perdita, ottimizzatori e scheduler

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**nn.CrossEntropyLoss:** È una funzione di perdita. Dice al modello quanto i suoi output si discostano dai valori reali, guidando l’ottimizzatore nella direzione giusta per migliorare le predizioni.

**optim.Adam:** L’ottimizzatore Adam (Adaptive Moment Estimation) è una versione avanzata della discesa del gradiente

* resnet\_model.parameters(): Specifica che i parametri del modello devono essere aggiornati.
* lr=config["learning\_rate"]: Il learning rate (tasso di apprendimento) definisce quanto grandi devono essere gli aggiornamenti dei parametri.

Utilizzando gli ottimizzatori aggiorno i pesi del modello per ridurre la funzione di perdita.

Poi utilizzo anche uno scheduler per evitare che un learning rate troppo alto può faccia oscillare il modello senza convergere mentre un learning rate troppo basso rallenti l’apprendimento

**StepLR:** È un tipo di scheduler che regola il learning rate in modo graduale dopo un numero fisso di epoche.

* **step\_size:** Dopo quante epoche ridurre il learning rate.
* **gamma:** Fattore di riduzione. In questo caso, gamma=0.5, il learning rate viene moltiplicato per 0.5

Quindi:

* **Funzione di perdita (criterion):** Calcola l’errore del modello.
* **Ottimizzatori (resnet\_optimizer e efficientnet\_optimizer):** Utilizzano il gradiente della perdita per aggiornare i pesi del modello.
* **Scheduler (resnet\_scheduler e efficientnet\_scheduler):** Modificano dinamicamente il learning rate per migliorare l’efficienza dell’ottimizzatore.

Addestramento

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Training:** Addestro entrambi i modelli utilizzando il dataloader, la funzione di perdita e gli ottimizzatori.
* **Scheduler:** Aggiorno il learning rate dopo ogni epoca.
* Il parametro **patience=config["patience"]** serve per il meccanismo di **early stopping**, cioè interrompere l'addestramento se le performance sul validation set non migliorano per un certo numero di epoche consecutive

Infine, abbiamo la valutazione e la visualizzazione

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Valutazione:** Valuta le performance dei modelli sul test set.
* **Visualizzazione:** Mostra predizioni dei due modelli.