Report Progetto - "Laboratorio di Ottimizzazione, Intelligenza Artificiale e Machine Learning"

Introduzione

Contesto e Motivazione

La **classificazione** delle immagini è un compito fondamentale nel campo della visione artificiale, con numerose applicazioni pratiche, dal riconoscimento facciale all'analisi delle immagini mediche. In questo progetto, ho affrontato il problema della classificazione delle immagini utilizzando il dataset **CIFAR-10**, che contiene 60.000 immagini 32x32 suddivise in **10 classi** distinte.

(https://www.cs.toronto.edu/~kriz/cifar.html)

Scopo del Progetto

L'obiettivo del progetto è implementare e ottimizzare un **modello** di rete neurale in grado di classificare le immagini del dataset CIFAR-10. Utilizzando tecniche avanzate e un modello pre-addestrato su **ImageNet**, intendo migliorare la precisione del modello su questo dataset relativamente piccolo.

Obiettivi Specifici

- Implementare un modello di classificazione delle immagini utilizzando una rete EfficientNet-B0 **pre-addestrata** su ImageNet.
- **Adattare** l'architettura del modello per renderla compatibile con le classi del dataset CIFAR-10.
- **Ottimizzare** il processo di allenamento
- Utilizzare **data-augmentation** per migliorare la capacità del modello di generalizzare su dati non visti.
- **Valutare** e **migliorare** le prestazioni del modello attraverso un processo iterativo di allenamento e validazione.

Descrizione del progetto

Struttura del Progetto

Il progetto è organizzato in diverse **directory**, ognuna delle quali contiene **file** specifici per determinati aspetti del workflow:

- **config/**: file di configurazione:
  - **config.json**: iper-parametri di configurazione
  - **config_schema.json**: file di strutturazione e tipizzazione dati
- **data/**: caricamento e pre-processing dei dati:
  - **data_loader.py**: file di caricamento dei dati che scarica il dataset in caso di prima esecuzione e si occupa di pre-processing
- **models/**: definizione del modello:
  - **best_model.pth**: salvataggio del modello migliore in fase di allenamento
- **notebooks/**: analisi dei dati.
- **runs/**: archivio dei run di allenamento utili per visualizzazione tramite TensorBoard
- **utils/**: funzioni di utilità:
  - **early_stopping.py**: file di gestione di modalità di early-stopping in caso di mancati miglioramenti delle prestazioni di accuracy e loss
  - **alert.py**: utilità sonora in fase di terminazione allenamento
  - **beep.py**: utilità sonora in fase di miglioramento modello
  - **clear_console.py**: utilità per pulire la console e garantire un'esperienza di esecuzione più "pulita"
  - **console_output_manager.py**: utilità per gestire sopprimere/permettere gli output a terminale per evitare "fastidiosi" messaggi come future-warnings, ...
  - **save_epoch_output.py**: utilità che scrive su un file .txt il risultato migliore dell'ultimo ciclo di allenamento eseguito
  - **time_manager.py**: utilità per gestire il concetto di tempo all'interno del programma
- **testing_images/**: archivio di immagini con il quale verificare la correttezza del modello
- **main.py**: script principale da cui avviare l'addestramento del modello tramite comando "*python main.py*".
- **image_test.py**: script che permette di verificare/testare il funzionamento del modello sottoponendogli le immagini presenti nella directory "testing_images/", utilizzabile tramite comando "*python image_test.py*".
- **last_output.py**: script che permette di leggere il file .txt contenente l'output migliore dell'ultimo addestramento, in modo tale da avere un'idea precisa delle prestazioni
- **environment.yaml**: ambiente contenente tutte le dipendenze necessarie per poter eseguire il codice in ambiente CONDA. Per eseguire il file di installazione eseguire il comando "*conda env create -f environment.yaml*"

Descrizione del modello

Architettura del Modello

Il file *model.py* definisce una **rete neurale** personalizzata basata sul modello pre-addestrato **EfficientNet-B0** fornito dalla libreria **torchvision**. Il codice sfrutta l'architettura pre-esistente di EfficientNet-B0 la modifica per adattarla alle esigenze specifiche del progetto e ne implementa una versione personalizzata con la classe **CustomModel**.

EfficientNet-B0 è una versione **ottimizzata** di EfficientNet, progettata per bilanciare tra accuratezza e **efficienza** computazionale. È particolarmente utile in scenari con limitazioni di risorse. *(come il mio caso)*

Batch Normalization e **Dropout**: L'inserimento di questi due componenti nei **livelli finali** migliora la **stabilità** e la capacità di **generalizzazione** del modello. Dropout impedisce che il modello si adatti troppo ai dati di addestramento.

Modifiche e Ottimizzazioni

La parte finale del modello, il **classificatore**, è stata modificata per adattarsi al compito specifico che il modello deve svolgere. Originariamente, EfficientNet-B0 è addestrato per classificare immagini in **1000 categorie** (come in ImageNet). Qui, il codice lo modifica per adattarsi a un numero diverso di classi (*output_size=10*).

Dopo alcuni tentativi con alcune architetture come EfficientNet-B[1-3], ResNet18, DenseNet121, VGG-16 e MobileNet-V2, tutte con prestazioni non convincenti a causa di eccessiva complessità operazionale o tempi di addestramento troppo elevati, ho scelto EfficientNet-B0.

```python
class CustomModel(nn.Module):
    def __init__(self, output_size):
        super(CustomModel, self).__init__()
        # Carica il modello EfficientNet-B2 pre-addestrato
        self.base_model = efficientnet_b0(weights=EfficientNet_B0_Weights.DEFAULT)

        # Modifica l'ultimo layer per il numero di output desiderato
        self.base_model.classifier[1] = nn.Linear(self.base_model.classifier[1].in_features, output_size)

    def forward(self, x):
        return self.base_model(x)
```

Caricamento Dati

Il file *data_loader.py* contiene la logica per **preparare** e **caricare** i dati per l'addestramento e la validazione, in particolare prevede:

Trasformazioni del training set:

- La funzione definisce una serie di trasformazioni per **aumentare la varietà** del dataset durante l'addestramento, migliorando la capacità del modello di **generalizzare**:
  - **RandomHorizontalFlip()**: Capovolge casualmente le immagini orizzontalmente.
  - **RandomVerticalFlip()**: Capovolge casualmente le immagini verticalmente.
  - **RandomRotation(15)**: Ruota casualmente le immagini di un massimo di 15 gradi.
  - **ColorJitter()**: Modifica casualmente la luminosità, il contrasto, la saturazione e la tonalità delle immagini.
  - **ToTensor()**: Converte le immagini in tensori PyTorch.
  - **Normalize()**: Normalizza le immagini con valori di media e deviazione standard specifici per ogni canale (usati comunemente per i modelli pre-addestrati su ImageNet).

Trasformazioni del validation set:

- Le trasformazioni per il validation set sono più **semplici** e si concentrano principalmente sulla **normalizzazione** delle immagini:
  - **Resize(224)**: ridimensionamento dell'immagine per adattamento
  - **ToTensor** e **Normalize**: Convertire e normalizzare le immagini, come fatto per il training set.

```python
# Definisce le trasformazioni per le immagini
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(), # Ruota l'immagine di 90 gradi in senso orario
    transforms.RandomCrop(32, padding=4), # Esegue un crop di 32x32
    transforms.RandomRotation(15), # Ruota l'immagine di 15 gradi in senso orario
    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1), # Modifica i parametri delle immagini
    transforms.ToTensor(), # Converte l'immagine in un tensore
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Normalizza i valori dei tensori
])

val_transform = transforms.Compose([
    transforms.Resize(224), # Rimappa le immagini a 224x224
    transforms.ToTensor(), # Converte l'immagine in un tensore
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]), # Normalizza i valori dei tensori
])
```

[Per garantire ripetibilità negli addestramenti ho inserito un seed manuale]

```python
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size], generator=torch.Generator().manual_seed(42))
```

Processo di allenamento

Definizione dell'Allenamento
Il processo di **allenamento** è stato gestito utilizzando il framework PyTorch. Ho utilizzato l'ottimizzatore **AdamW** per aggiornare i pesi del modello, combinato con uno **scheduler** di learning rate per adattare dinamicamente la velocità di apprendimento al verificarsi di situazioni definite "**plateau**" e la funzione di loss di classificazione "**CrossEntropyLoss**". In particolare:

- **AdamW**: serve ad aggiornare i pesi del modello durante il training, riducendo il rischio di overfitting grazie alla regolarizzazione L2, una tecnica che penalizza i pesi di un modello durante l'addestramento.

  Ho eseguito anche un test con optimizer **AdaGrad**: il cui scopo è quello di scalare il LR per ogni parametro, accumulandone il quadrato dei gradienti, per poi aggiornarlo in rapporto a questo accumulatore.

- **ReduceLROnPlateau**: riduce automaticamente il tasso di apprendimento di un determinato "*factor*" quando le prestazioni attuali non migliorano per un certo numero di epoche definito "*patience*" *(si definisce "plateau" una situazione in cui il modello rimane più o meno stazionario invece che migliorare).*

- **CrossEntropyLoss**: misura la differenza tra le probabilità previste dal modello e le etichette vere in un problema di classificazione, penalizzando previsioni lontane dalla realtà per migliorare l'accuratezza.

Inoltre, ho implementato la tecnica di **early-stopping** per interrompere l'allenamento nel caso di mancati miglioramenti significativi delle prestazioni entro un numero di epoche predefinito (patience), prevenendo cosi situazioni di overfitting.

La funzione ***train()*** è il cuore del processo di addestramento e include tutte le operazioni necessarie per iterare attraverso le epoche di addestramento e validazione. Per ogni epoca, il modello viene impostato in modalità di allenamento e per ogni batch di dati:

- Viene calcolata la loss e aggiornati i pesi del modello tramite il back-propagation, la quale serve ad aggiornare i pesi del modello calcolando e propagando all'indietro l'errore dei livelli finali a quelli iniziali.
- Vengono aggiornati gli accumulatori per la loss e l'accuratezza.

Al termine di ogni epoca, il modello viene valutato sul set di validazione utilizzando la funzione ***evaluate()***, che restituisce la loss e l'accuratezza sul validation set e il learning rate scheduler viene aggiornato in base alla validazione.

Il criterio di early-stopping viene verificato alla fine di ogni epoca e se soddisfatto, l'addestramento si interrompe prematuramente.

Inoltre all'interno dello stesso file è presente un metodo per salvare la miglior versione del modello raggiunta come checkpoint da cui riavviare successivamente l'addestramento in caso di arresto intermedio.

[Se l'addestramento viene interrotto manualmente con Ctrl+C, viene gestita l'eccezione KeyboardInterrupt per terminare l'addestramento in modo ordinato.]

UPDATE – Dopo l'introduzione dei metodi per visualizzare su TensorBoard la Confusion Matrix, sono nati problemi per quanto riguarda la cattura della KeyboardInterrupt e non sono stato in grado di risolverli nonostante i molteplici tentativi.

Struttura e Funzionalità degli Iper-parametri utilizzati

1. **Sezione data**:
   - *batch_size*: **512**:
     - Questo parametro definisce il numero di **campioni** che il modello elabora prima di aggiornare i pesi.
   - *validation_split*: **0.2**:
     - Indica la **percentuale** di dati di addestramento che viene riservata per la validazione. In questo caso, il **20%** dei dati totali verrà utilizzato per valutare le prestazioni del modello durante l'addestramento, mentre il **80%** sarà usato per l'addestramento vero e proprio, aiutando a monitorare l'overfitting.
2. **Sezione training**:
   - *epochs*: **30**:
     - Definisce il **numero massimo** di epoche durante l'addestramento. Un'epoca rappresenta un **passaggio completo** attraverso l'intero set di dati di addestramento.
   - *learning_rate*: **0.003**:
     - Il learning rate è un iper-parametro fondamentale che determina la **velocità** con cui il modello **aggiorna** i suoi **pesi** durante l'addestramento.
   - *patience*: **3**:
     - Definisce il numero di epoche di **tolleranza** che l'early-stopping concede senza miglioramenti significativi nella loss di validazione prima di **interrompere** l'addestramento.
   - *delta*: **0.01**:
     - È una **soglia minima** di miglioramento nella loss di validazione che deve essere superata affinché l'early-stopping non intervenga.
3. **Sezione model**:
   - *architecture*: **"efficientnet_b0"**:
     - Specifica l'**architettura** del modello da utilizzare. In questo caso, si utilizza **EfficientNet-B0**, un modello noto per l'**ottimizzazione** tra accuratezza e risorse computazionali.
   - *input_size*: **3072**:
     - Indica la **dimensione** dell'input che il modello riceve, in questo caso immagini di dimensioni **32x32** pixel.
   - *output_size*: **10**:
     - Definisce il numero di **classi** distinte per il compito di classificazione

Risultati

Prestazioni del Modello

Le prestazioni ottenute dal modello sono visualizzabili a termine di ogni addestramento eseguendo lo script "*python last_epoch.py*", il quale legge il contenuto del file "*epoch_output.txt*", fornendo dati relativi ad accuracy e loss raggiunte.

```
Training...: 100%| | 79/79 [02:59<00:00,  2.28s/it]
Epoca [22/30]:
- Train Loss: 0.36
- Train Accuracy: 87.59%
- Validation Loss: 0.49
- Validation Accuracy: 83.27%
[ 21:37:09 - Tempo impiegato: 199.94s ]
Early stopping counter: 5 / 5

Early stopping triggered: Fine allenamento

Early stopping attivato

[ Tempo totale impiegato: 1h 12m 8s ]
```

Se si sceglie invece di impiegare AdaGrad come ottimizzatore, le prestazioni che si raggiungono sono inferiori, ma più stabili e con meno tendenza ad overfitting:

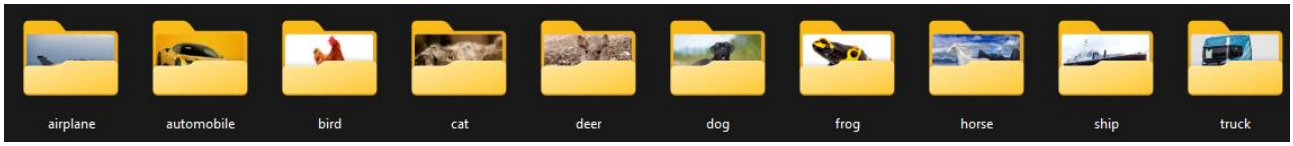
```
Training...: 100%| | 157/157 [02:53<00:00,
Epoca [30/30]:
- Train Loss: 0.61
- Train Accuracy: 78.40%
- Validation Loss: 0.66
- Validation Accuracy: 77.09%
[ 16:12:53 - Tempo impiegato: 193.99s ]
Validation loss diminuita (0.68 --> 0.66), salvataggio nuovo modello migliore...


[ Tempo totale impiegato: 1h 34m 58s ]
```

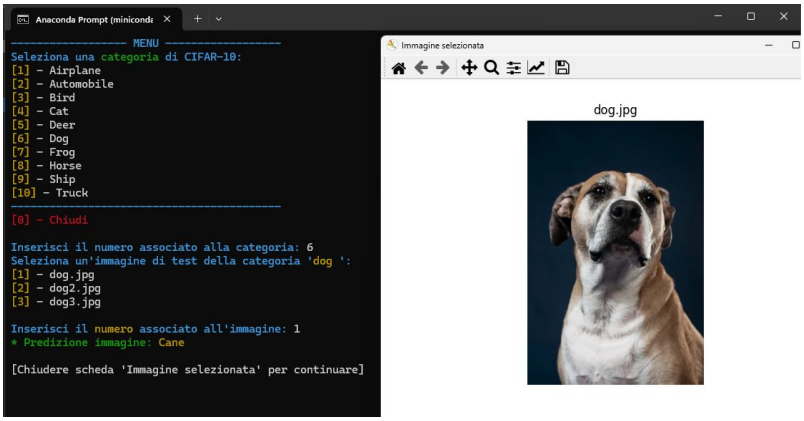
Test e Verifica

Validazione su Immagini Esterne

È possibile testare il modello utilizzando immagini esterne presenti, per esempio quelle situate nella directory ***testing_images/***, che non facevano parte del dataset CIFAR-10. Il modello ha mostrato buone capacità di generalizzazione, classificando correttamente le immagini anche al di fuori del set di training.



Per poter effettuare queste verifiche è sufficiente digitare a terminale lo script "*python image_test.py*", per avviare il menù di selezione delle immagini.



Analisi degli Errori

Durante il ciclo di sviluppo del progetto, e i vari test con una serie di modelli differenti, una parte delle immagini è stata classificata erroneamente, suggerendo aree di miglioramento. Gli errori più comuni includevano immagini di classe simile ad esempio:

- cani vs. gatti
- camion vs. barche
- ecc...

Conclusioni

Sintesi del Progetto

Il progetto ha dimostrato l'efficacia del **transfer learning** nel migliorare le prestazioni di un modello di classificazione delle immagini su un dataset relativamente piccolo come CIFAR-10. La struttura modulare del codice e l'utilizzo di tecniche avanzate di ottimizzazione hanno permesso di ottenere risultati significativi.

Riferimenti e fonti

Bibliografia

- EfficientNet: https://arxiv.org/abs/1905.11946
- CIFAR-10: https://www.cs.toronto.edu/~kriz/cifar.html
- Documentazione PyTorch: https://pytorch.org/
- Altre risorse