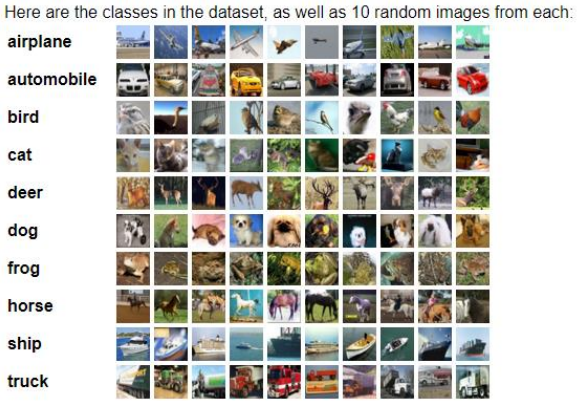
Report Progetto - "Laboratorio di Ottimizzazione, Intelligenza Artificiale e Machine Learning"

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, si intende migliorare la precisione del modello su questo dataset relativamente piccolo.

Obiettivi Specifici

- Implementare un modello di classificazione delle immagini utilizzando la rete EfficientNet-B2 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.

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-processamento 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/**: directory contenente la definizione del modello:
  - **model.py**: file contenente la struttura del modello basato su EfficientNet
  - **best_model.pth**: salvataggio del modello migliore in fase di addestramento
- **notebooks/**: notebook .ipynb di analisi preventiva dei dati del dataset.
- **runs/**: archivio dei run di allenamento visualizzabili tramite tensorboard
- **utils/**: funzioni di utilità valide per tutto il programma:
  - **early_stopping.py**: file di gestione di modalità di early_stopping in caso di mancati miglioramenti delle prestazioni di accuracy e loss per un predeterminato periodo massimo di epoche chiamato “patience”.
  - **alert.py**: utilità sonora in fase di terminazione allenamento
  - **beep.py**: utilità sonora in fase di miglioramento modello
  - **clear_console.py**: utilità per ripulire la console e garantire un'esperienza di esecuzione più completa e fluida.
  - **console_output_manager.py**: utilità per sopprimere/consentire gli output a terminale per evitare "fastidiosi" messaggi come future-warnings, ecc...
  - **save_epoch_output.py**: utilità che scrive su un file z.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**: file 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*" *(posizionandosi prima nella root del repository)*

Descrizione del modello

Architettura del Modello

Il file *model.py* definisce una rete neurale personalizzata basata sul modello pre-addestrato **EfficientNet-B2** fornito dalla libreria **torchvision**. Il codice sfrutta l'architettura pre-esistente e la modifica per adattarla alle esigenze specifiche del progetto, implementandone una versione personalizzata tramite la classe CustomModel.

EfficientNet-B2 è una versione ottimizzata di EfficientNet, progettata per bilanciare tra accuratezza e efficienza computazionale ed è quindi particolarmente adatto a scenari con limitazioni di risorse *(come il mio caso...)*

Batch Normalization e **Dropout** sono due componenti utili per migliorare la stabilità e la capacità di generalizzazione del modello. In particolare:

- Dropout: impedisce che il modello si adatti troppo ai dati di addestramento (overfitting).
- Batch Normalization: accelera l'addestramento e stabilizza il modello normalizzando l'input di ogni livello.

Modifiche e Ottimizzazioni

Per adattare EfficientNet-B2 a questo task, la parte finale del modello (il classificatore) è stata modificata per adattarsi al compito specifico che il modello deve svolgere.

Originariamente, EfficientNet-B2 è addestrato per classificare immagini in 1000 categorie (come in ImageNet). Qui, il codice lo modifica per adattarsi a un numero diverso di classi (definito dall'output_size = 10) e aggiunge nuovi livelli per migliorare la capacità di apprendimento:

- **Dropout**: Disattiva casualmente alcune unità durante l'allenamento per prevenire l'overfitting.
- **Linear Layer (Fully Connected Layer)**: Aggiunge un livello completamente connesso per ridurre la dimensionalità e catturare caratteristiche complesse.
- **Batch Normalization**: Normalizza l'output di un livello per accelerare l'allenamento e stabilizzare le attivazioni.
- **ReLU**: Funzione di attivazione che introduce non-linearità, permettendo al modello di apprendere rappresentazioni più complesse.

```
self.base_model.classifier[1] = nn.Sequential(
            nn.Dropout(0.7), # Dropout per regolarizzare
            nn.Linear(self.base_model.classifier[1].in_features, 1024), # Linear layer con 1024 neuroni
            nn.BatchNorm1d(1024), # Batch normalization
            nn.ReLU(), # Funzione di attivazione ReLU
            nn.Dropout(0.6), # Dropout per regolarizzare
            nn.Linear(1024, 512), # Linear layer con 512 neuroni
            nn.BatchNorm1d(512), # Batch normalization
            nn.ReLU(), # Funzione di attivazione ReLU
            nn.Dropout(0.5), # Dropout per regolarizzare
            nn.Linear(512, output_size) # Linear layer con output_size neuroni
        )
```

Caricamento Dati

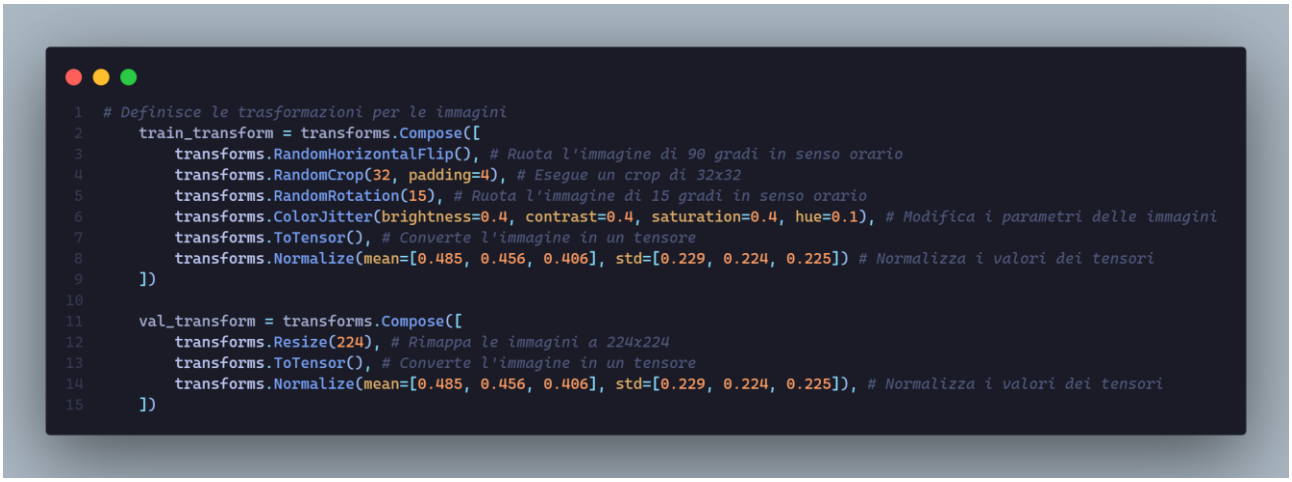
Il file *data_loader.py* contiene la logica per preparare e caricare i dati per l'addestramento e la validazione.

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.
  - **RandomCrop()**: Esegue un ritaglio casuale delle immagini dopo aver aggiunto un padding.
  - **RandomRotation()**: 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 sul ridimensionamento e normalizzazione delle immagini:
  - **Resize(224)**: Ridimensiona le immagini a 224x224 pixel, una dimensione tipica per modelli come EfficientNet.
  - **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
    ])
```

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. In particolare:

- **AdamW**: serve ad aggiornare i pesi del modello durante l'addestramento, riducendo il rischio di overfitting grazie alla regolarizzazione L2 (una tecnica che penalizza i pesi di un modello durante l'addestramento)
- **Scheduler ReduceLROnPlateau**: riduce automaticamente il tasso di apprendimento quando le prestazioni del modello non migliorano per un certo numero di epoche

Inoltre, ho implementato la tecnica di early-stopping per interrompere l'allenamento nel caso di mancati miglioramenti significativi delle prestazioni di validation loss 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 back-propagation, la quale serve ad aggiornare i pesi del modello calcolando e propagando all'indietro l'errore dai 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, inoltre 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.

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

```
# Itera sul dataset di training
        for i, (inputs, labels) in enumerate(train_loader):
            # Calcola la perdita e fa backpropagation
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Aggiorna valori di perdita e accuracy
            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        # Calcola la perdita media e l'accuratezza
        train_loss = running_loss / len(train_loader)
        train_accuracy = 100 * correct / total

        # Valuta il modello sul dataset di validazione
        val_loss, val_accuracy = evaluate(model, val_loader, criterion)
```

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*: **0.20**:
     - 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 l'80% sarà usato per l'addestramento vero e proprio.

2. **Sezione training**:
   - *epochs*: **30**:
     - Definisce il numero massimo di epoche durante l'addestramento, cioè un passaggio completo attraverso l'intero set di dati di addestramento. L'early stopping può interrompere l'addestramento prima del raggiungimento delle 30 epoche, se le condizioni lo permettono.
   - *learning_rate*: **0.003**:
     - Il learning rate determina la velocità con cui il modello aggiorna i suoi pesi durante l'addestramento. Un valore di 0.003 permette al modello di apprendere rapidamente. In caso esso sia troppo elevato e causi problemi, verrà modificato dalla funzione di scheduling del LR.
   - *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*: **0.005**:
     - È una soglia minima di miglioramento nella loss di validazione che deve essere superata affinché l'early stopping non intervenga. Questo previene l'interruzione del training in presenza di piccoli miglioramenti che potrebbero essere rilevanti.

3. **Sezione model**:
   - *architecture*: **"efficientnet_b2"**:
     - Specifica l'architettura del modello da utilizzare. In questo caso, si utilizza **EfficientNet-B2**.
   - *input_size*: **3072**:
     - Indica la dimensione dell'input che il modello riceve. Questo parametro deve essere coerente con la struttura delle immagini di input, in questo caso 32x32 px.
   - *output_size*: **10**:
     - Definisce che il modello è progettato per classificare i dati in 10 categorie distinte.

Risultati

Prestazioni del Modello

Come si può osservare eseguendo lo script "*python last_output.py*", il quale è utile per poter visualizzare la migliore prestazione ottenuta durante l'ultimo allenamento, il modello ha raggiunto una precisione del **81.58%** sul validation set ed un valore di perdita (loss) di circa **0.55**, con un miglioramento rispetto all'approccio senza transfer learning.

L'integrazione di maggiori tecniche di data augmentation ha inoltre portato ad una maggiore robustezza del modello, riducendo il gap tra l'accuracy su training e validation set.

```
----------------------------------------
Epoca [17/30]:
- Train Loss: 0.53
- Train Accuracy: 82.66%
- Validation Loss: 0.55
- Validation Accuracy: 81.58%
[ 14:17:31 - Tempo impiegato: 261.63s ]
```

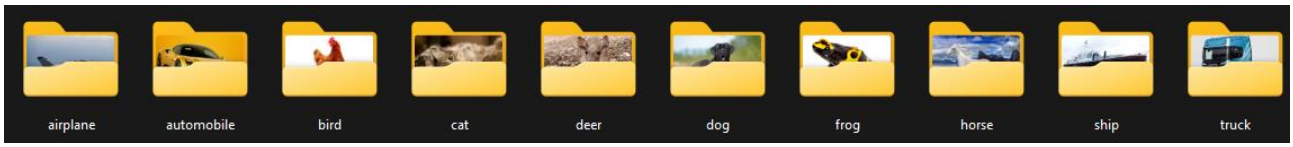
Confronto con Modelli Baseline

Il modello ha mostrato prestazioni superiori rispetto a un semplice modello CNN addestrato da zero, dimostrando l'efficacia del transfer learning e delle tecniche di regolarizzazione utilizzate.

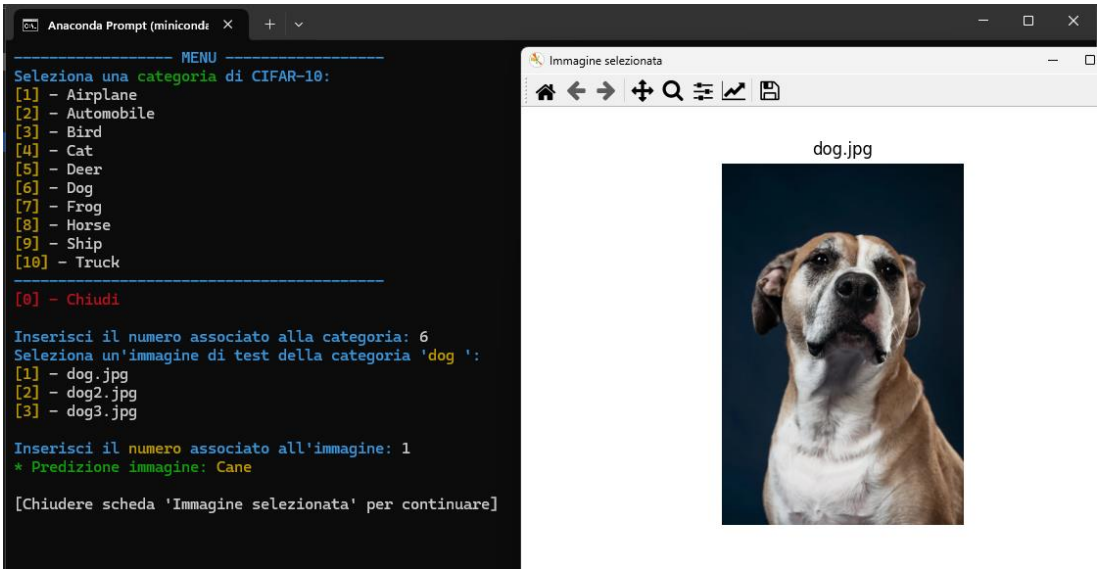
Test e Verifica

Validazione su Immagini Esterne

È possibile testare il modello utilizzando immagini esterne, 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 modello, 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