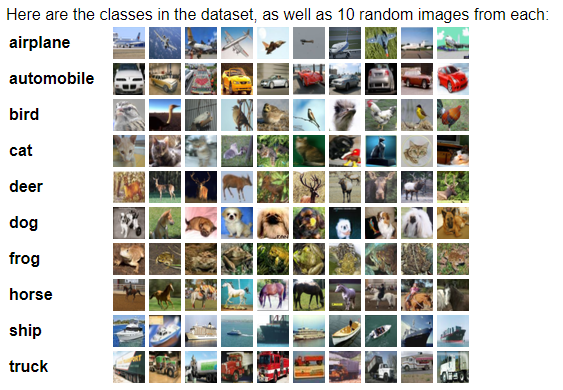
***Report Progetto - “Laboratorio di Ottimizzazione, Intelligenza Artificiale e Machine Learning”***

# **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-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 mediante l'uso di tecniche di regolarizzazione, come il dropout e il weight decay.
* 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 preprocessing 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
* **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 sonorta 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 corretteza del modello
* **main.py**: script principale da cui avvviare l'addestramento del modello tramite comando "python main.py".
* **image\_test.py**: script che permettte di verificare/testare il funzionamento del modello sottoponendogli le immagini presenti nella directory "testing\_images/"
* **last\_output.py**: scritp 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**

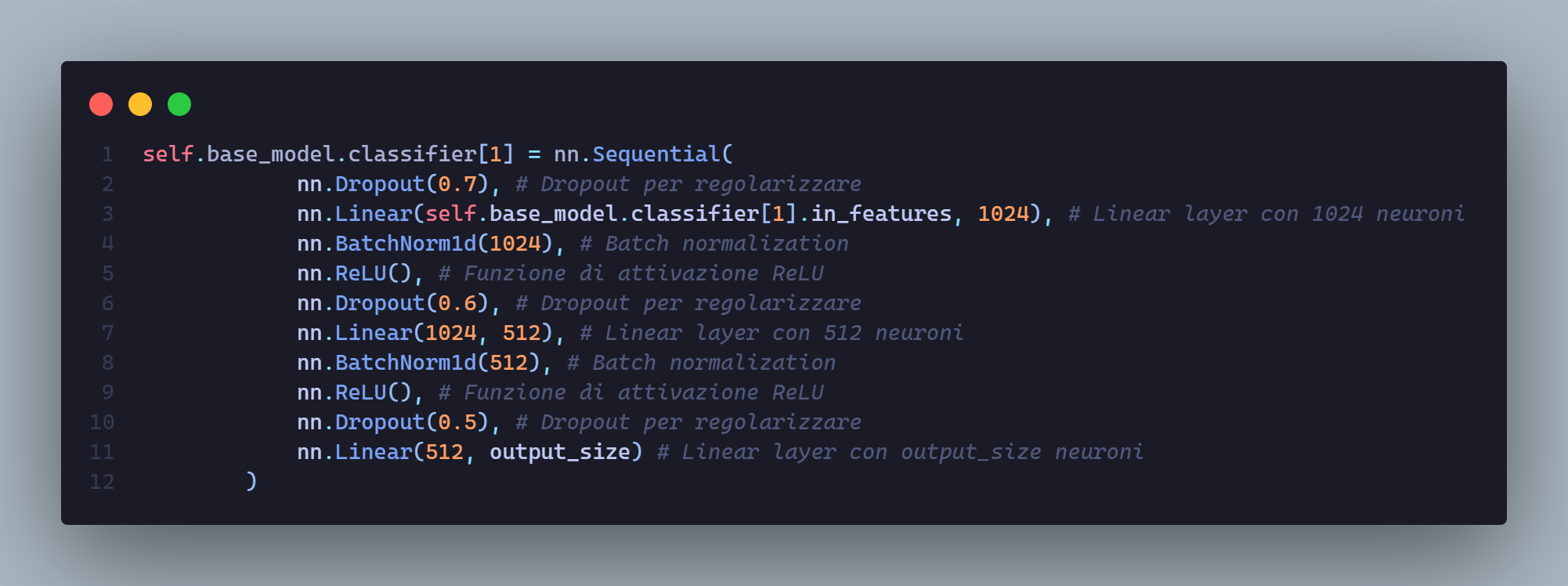
**Architettura del Modello**  
Il file model.py definisce una rete neurale personalizzata basata sul modello pre-addestrato **EfficientNet-B2** fornito dalla libreria **torchvision**. Il codice sfrutta l'architettura pre-esistente di EfficientNet-B2, la modifica per adattarla alle esigenze specifiche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

**EfficientNet-B2** è una versione ottimizzata di EfficientNet, progettata per bilanciare tra accuratezza e efficienza computazionale. È particolarmente utile in scenari con limitazioni di risorse

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

**Modifiche e Ottimizzazioni**  
Per adattare EfficientNet-B2 al nostro 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 (output\_size) e aggiunge nuovi livelli per migliorare la capacità di apprendimento del modello:

* **Dropout**: Utilizzato per prevenire l'overfitting, spegne casualmente alcune unità durante l'allenamento.
* **Linear Layer (Fully Connected Layer)**: Aggiunge un layer completamente connesso per ridurre la dimensionalità e migliorare la capacità del modello di catturare caratteristiche complesse.
* **Batch Normalization**: Viene applicata per normalizzare l'output di un livello e velocizzare l'allenamento stabilizzando la distribuzione delle attivazioni.
* **ReLU**: Funzione di attivazione rettificata, introduce non-linearità per consentire al modello di apprendere rappresentazioni più complesse.



**Caricamento Dati**

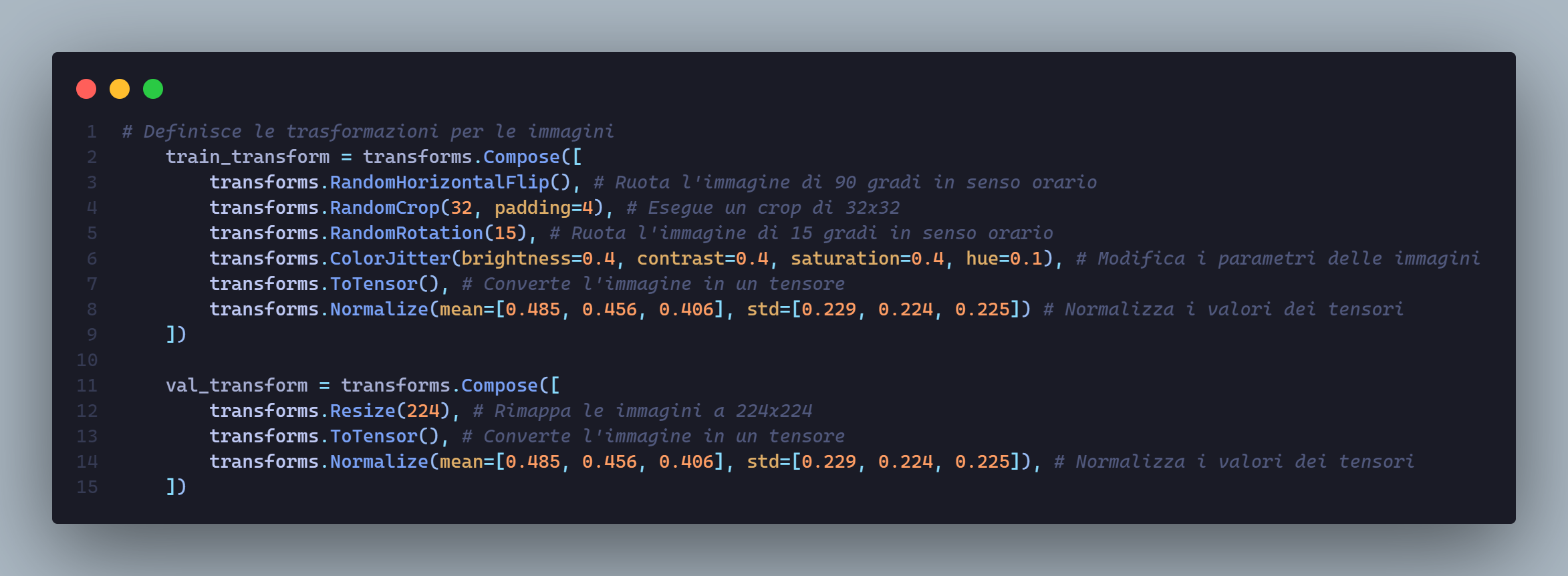
Il file data\_loader.py contiene la logica per preparare e caricare i dati per l'addestramento e la validazione.

**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.



# **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.

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 train.

Per ogni batch di dati:

* Viene calcolata la loss e aggiornati i pesi del modello tramite il back-propagation.
* 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. 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.]*



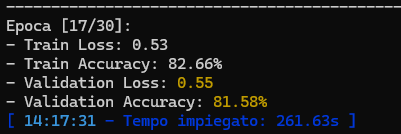
**Struttura e Funzionalità degli Iper-parametri utilizzati**

1. **Sezione data**:
   * *batch\_size***: 512**:
     + Questo parametro definisce il numero di campioni che il modello elabora prima di aggiornare i pesi. Un batch size di 512 indica che il modello eseguirà il backpropagation e l'aggiornamento dei pesi dopo aver elaborato 512 esempi. Un batch size maggiore può velocizzare l'addestramento grazie all'efficienza computazionale, ma richiede più memoria GPU.
   * *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. Questo aiuta 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. 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 è un iperparametro fondamentale che determina la velocità con cui il modello aggiorna i suoi pesi durante l'addestramento. Un valore di 0.003 è relativamente alto, il che permette al modello di apprendere rapidamente, ma potrebbe necessitare di un attento monitoraggio per evitare di saltare oltre i minimi locali.
   * *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. Con un valore di 3, se la loss di validazione non migliora per tre epoche consecutive, l'addestramento viene fermato.
   * *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**, un modello noto per l'ottimizzazione tra accuratezza e risorse computazionali.
   * *input\_size***: 3072**:
     + Indica la dimensione dell'input che il modello riceve. Questo parametro deve essere coerente con la struttura dei dati di input (ad es. immagini ridimensionate o caratteristiche estratte), in questo caso immagini 32x32.
   * *output\_size***: 10**:
     + Definisce il numero di classi per il compito di classificazione. Un valore di 10 indica 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.

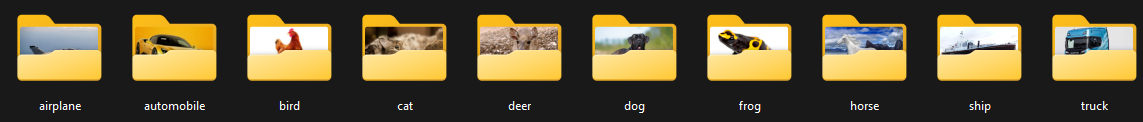
L'integrazione di maggiori tecniche di data augmentation ha portato a una maggiore robustezza del modello, riducendo il gap tra l'accuracy su training e validation set.



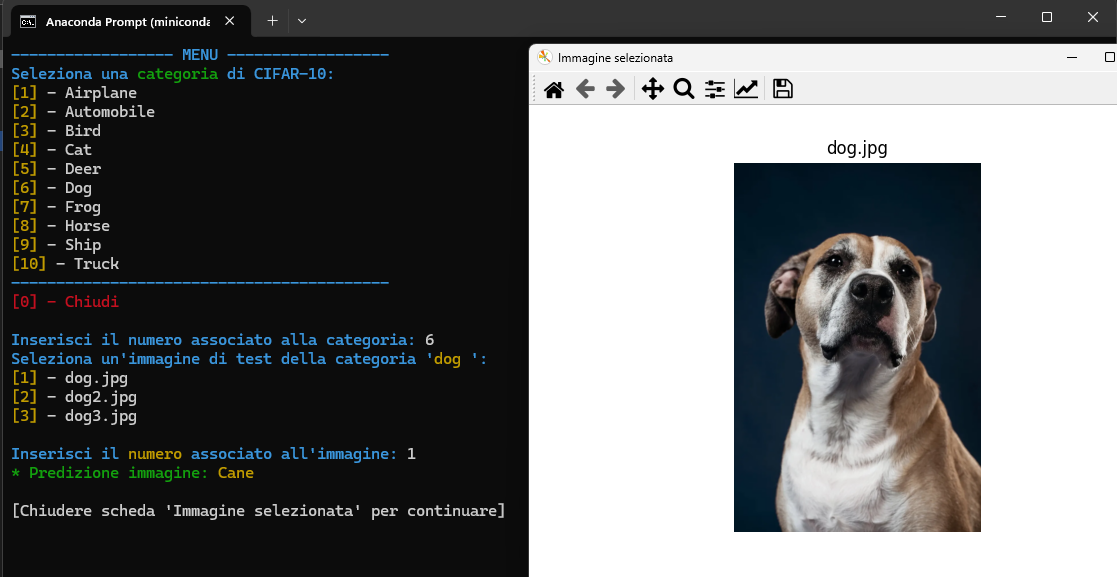
**Confronto con Modelli Baseline**  
Il modello ha mostrato prestazioni superiori rispetto a un semplice modello CNN addestrato da zero, dimostrando l'efficacia del transfer learning e delle tecniche di regolarizzazione utilizzate.

# **Test e Verifica**

**Validazione su Immagini Esterne**  
è possibile testare il modello utilizzando immagini esterne 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**  
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…

Esse suggerivano la necessità di ulteriori tecniche di miglioramento della discriminazione tra classi simili.

# **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: Rethinking Model Scaling for Convolutional Neural Networks](https://arxiv.org/abs/1905.11946)
* CIFAR-10 and CIFAR-100 Datasets
* Documentazione PyTorch (<https://pytorch.org/>)
* Altri articoli e risorse rilevanti...