RELAZIONE PROGETTO DI MACHINE LEARNING

Giorgio Guarnieri, 575295

Roberto Magno Mazzotta, 575368

Vittorio Salta, 575883

1 Semestre 2023/2024

INDICE

CAPITOLO 1: INTRODUZIONE

1.1 Obiettivo del Progetto

Abbiamo scelto la seconda traccia, usando Spettrogrammi di target spaziali, con l'obiettivo di "proporre almeno due metodi di ML/DL per la classificazione degli oggetti presenti negli spettrogrammi in 3 classi (oggetto piccolo, oggetto medio, oggetto grande), valutando le prestazioni dei metodi scelti a confronto e discutendo i risultati ottenuti in termini di matrici di confusione e accuratezza."

1.2 Librerie

Per questo progetto abbiamo usato il framework PyTorch, in particolare le librerie torch e torchvision. Abbiamo usato anche le librerie pandas, numpy, seaborn, matplotlib e sklearn.

1.3 Hardware

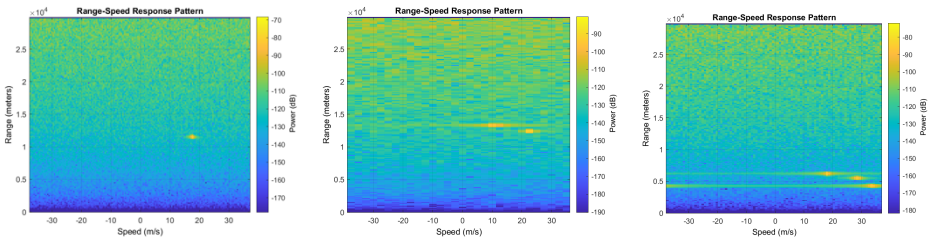
Abbiamo utilizzato un MacBook Pro M1 con MPS(metal platform sysyem) , modulo di PyTorch che ha recentemente introdotto per accelerare l'addestramento tramite la GPU.

https://pytorch.org/docs/stable/notes/mps.html

https://developer.apple.com/metal/pytorch/

1.4 Impostazione del Dataset

In questo progetto, lo scopo è quello di classificare spettrogrammi di target spaziali in tre categorie distinte, basate sulla loro grandezza(oggetto piccolo, oggetto medio e oggetto grande).



Oggetto classificato piccolo *Oggetto classificato medio* *Oggetto classificato grande*

Il dataset a nostra disposizione comprende 600 immagini, suddivise equamente in 200 per ciascuna classe. Il primo problema che abbiamo riscontrato è stata la carenza di dati, in quanto i modelli di apprendimento automatico richiedono grandi quantità di dati, circa 600 per classe, per essere addestrati in modo efficace.

Inoltre, la nostra analisi preliminare ha rivelato che le metriche di performance ottenute facendo "trainare" le reti neurali sul nostro dataset non raggiungono livelli soddisfacenti, sottolineando la necessità di approcci più sofisticati: ad esempio maggiori featuring di data augmentation, ma anche l'utilizzo di una rete pre-addestrata su tipologie diverse di dati, sempre inerenti alla classificazione di oggetti.

Un aspetto cruciale da considerare è l'importanza del colore negli spettrogrammi. Gli spettrogrammi, essendo rappresentazioni visive di segnali spaziali, contengono la maggior parte delle informazioni all'interno del colore. Pertanto, ogni manipolazione del dataset, come l'augmentation dei dati, deve mirare a preservare l'integrità di queste informazioni cromatiche.

1.5 Modifiche Dataset

Ci siamo orientati inizialmente verso l'uso di modelli di rete neurale convoluzionale (resnet34) pre-addestrati, sfruttando il potere dell'apprendimento trasferibile. Questo approccio ci permette di sfruttare la conoscenza acquisita dai modelli su dataset più ampi e diversificati, superando così le limitazioni del nostro dataset relativamente piccolo. In particolare, abbiamo esplorato l'uso di varianti della rete ResNet, notoriamente efficaci nel riconoscimento di pattern visivi complessi, adattando le loro architetture per adattarsi specificamente alla nostra attività di classificazione di spettrogrammi.

Successivamente abbiamo deciso di utilizzare una rete convoluzionale semplice e con pochi strati , dove abbiamo eseguito l'addestramento sui dati di train, utlizzando tecniche di data augmentation.

Durante l'ispezione iniziale del dataset, è emerso un aspetto critico: ciascuno spettrogramma includeva contorni e altre informazioni visive che si sono rivelate irrilevanti per il nostro obiettivo di classificazione. Questi elementi superflui rischiavano di introdurre rumore nei dati e di deviare l'attenzione del modello dalle caratteristiche essenziali per la distinzione delle classi basate sulla grandezza. Per affrontare ciò , abbiamo adottato una strategia di pre-elaborazione mirata a isolare le aree rilevanti degli spettrogrammi, rimuovendo i contorni e le informazioni irrilevanti.

Il processo è stato eseguito come segue:

Conversione in Tensori utilizzando: `transforms.ToTensor()`

(passaggio è essenziale per preparare le immagini per il processamento mediante la nostra rete neurale).

Ritaglio dell'Immagine: Utilizzando l'indicizzazione dei tensori di PyTorch abbiamo selezionato un sottoinsieme di pixel che esclude i bordi esterni:

```
t[:, 49:584, 102:708]
```

Questo intervallo di indici è stato determinato attraverso l'analisi visiva degli spettrogrammi e ha permesso di enfatizzare le regioni di interesse.

Visualizzazione:

Per assicurarci che i ritagli fossero appropriati, abbiamo visualizzato gli spettrogrammi ritagliati usando : `plt.imshow(t.permute(1,2,0))`

che riorganizza l'ordine delle dimensioni del tensore per soddisfare i requisiti della funzione di visualizzazione di Matplotlib. Con `plt.axis('off')` abbiamo rimosso gli assi per una visualizzazione più pulita, e `plt.tight_layout()` è stato utilizzato per ottimizzare la disposizione degli elementi nel plot. La rimozione di informazioni non pertinenti attraverso il ritaglio ha migliorato la focalizzazione del modello sulle caratteristiche salienti degli spettrogrammi. Successivamente, abbiamo proceduto con tecniche di data augmentation, selezionando con cura quelle che non alterano la composizione cromatica delle immagini, per preservare l'integrità delle informazioni cruciali per la classificazione.

CAPITOLO 2: PREPROCESSING

Abbiamo inizialmente definito una classe dataset con tre attributi:

- la path dell'immagine (`image_paths`)
- le trasformazioni da applicare alle immagini (`transform`)
- augmentation

Abbiamo poi definito una funzione `__getitem__` che accetta un indice e restituisce l'immagine corrispondente e la sua label, successivamente l'immagine viene convertita in formato 'rgb' e ritagliata tramite la funzione 'crop'.

Definiamo la funzione `get_train_valid_loader`, che prepara i DataLoader per il training e la validation. Accetta i parametri `train_size`, `batch_size`, e `num_workers`, con valori di default impostati rispettivamente a 0.8, 32 e 0.

La funzione inizia ottenendo i percorsi delle immagini di addestramento usando `get_images_paths` con `train_data_path`. Mischia casualmente i percorsi delle immagini e li divide in due set: uno per l'addestramento e uno per la validazione, basati sulla percentuale train_size (80%).

Definiamo due tipi di trasformazioni: `base_transform` (comprende ridimensionamento, conversione in tensori, e normalizzazione)e `augumentation_transform` (comprende rotazioni casuali, ritagli casuali, capovolgimenti orizzontali e leggeri cambiamenti nella luminosità, contrasto e saturazione). Creiamo due dataset (`train_dataset` e `valid_dataset`) usando la classe Dataset definita in precedenza, applichiamo augmenattion solo per il dataset di addestramento. Infine, crea i DataLoader per l'addestramento e la validazione (`train_loader` e `valid_loader`) con le opzioni specificate (dimensione del batch, numero di lavoratori, e se mischiare i dati o meno). Le trasformazioni e le augmentation dei dati aiutano a generalizzare meglio il modello e a prevenire l'overfitting.

Come per la funzione `get_train_valid_loader`, anche la funzione `get_test_loader` è creata per preparare il DataLoader per il set di dati di test. Accetta tre parametri con valori di default: `batch_size` impostato a 32, `num_workers` che indica il numero di core della cpu che si vogliono utilizzare , shuffle impostato a True, che indica se i dati devono essere mescolati o meno.

La funzione prepara un DataLoader per il set di test applicando trasformazioni standardizzate alle immagini: le ridimensiona a 224x224 pixel, le converte in tensori e le normalizza con valori medi e deviazioni standard. Le normalizzazioni e il ridimensionamento delle immagini sono cruciali per assicurarsi che i dati input siano in un formato adeguato per la rete neurale.

```
count_images_per_class(TRAIN_DATA_PATH)
```

```
0    158
1    162
2    160
dtype: int64
```

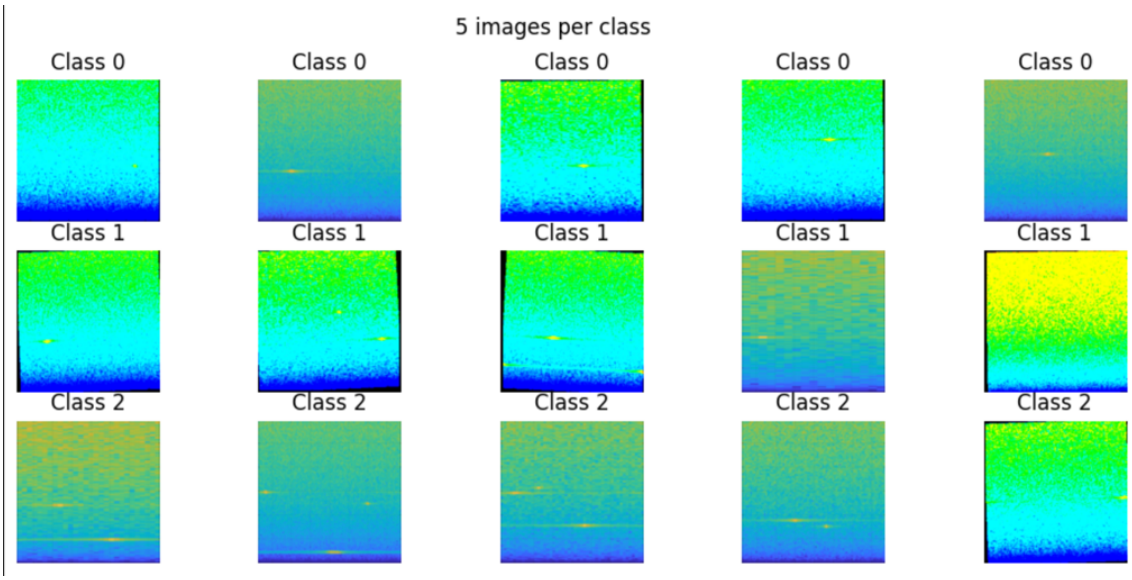
figura 1: funzione TRAIN_DATA_PATH

```
count_images_per_class(TEST_DATA_PATH)
```

```
0    40
1    40
2    40
dtype: int64
```

figura 2: funzione TEST_DATA_PATH

Concluso il preprocessing ci troviamo con tre classi di dati, etichettate con label tra 0, 1 e 2 con un numero uniforme di immagini in ciascuna classe nel dataset di addestramento e nel set di dati di test. Tutto ciò è importante per evitare bias del modello durante la fase di addestramento.



CAPITOLO 3: MODELLI TRAIN E TEST

UTILIZZO RETE ResNet-34 Pretrained:

La nostra rete "ResNet34Pretrained" è una versione personalizzata del noto modello di rete neurale convoluzionale ResNet-34. Questo modello è inizialmente caricato con pesi pre-addestrati tramite `torchvision.models.resnet34`, sfruttando l'apprendimento trasferibile da *ImageNet*, un vasto dataset di immagini.

Il cuore della personalizzazione sta nel ridefinire l'ultimo strato *fully connected* della rete. Utilizziamo il numero di filtri dell'ultimo strato lineare originale di ResNet-34 (`nr_filters`) per configurare i nostri strati personalizzati:

```
nr_filters = self.model.fc.in_features
self.model.fc = nn.Sequential(
nn.Linear(nr_filters, 256),
nn.ReLU(),
nn.Dropout(0.2),
nn.Linear(256, 128),
nn.ReLU(),
nn.Dropout(0.2),
nn.Linear(128, 64),
nn.ReLU(),
nn.Dropout(0.2),
nn.Linear(64, 3),
nn.LogSoftmax(dim=1)
)
```

Applichiamo una serie di strati lineari (`nn.Linear`) che riducono progressivamente la dimensione dei dati, da `nr_filters` a 256, poi a 128, e infine a 64. Questo permette una graduale e controllata estrazione di caratteristiche rilevanti per la classificazione.

Ogni strato lineare è seguito da una funzione di attivazione ReLU (`nn.ReLU`). ReLU è scelta per la sua efficacia nell'introdurre non-linearità nel modello, permettendo una rappresentazione più ricca delle caratteristiche apprese. Inoltre, ReLU aiuta a mitigare il problema della scomparsa dei gradienti durante l'addestramento, accelerando la convergenza.

Includiamo strati di dropout (`nn.Dropout`) con una probabilità di 0.2 dopo ogni ReLU. Questo aiuta a prevenire l'overfitting, specialmente quando si lavora con un dataset relativamente piccolo come quello degli spettrogrammi.

Abbiamo personalizzato l'ultimo strato (*fully connected*) del modello ResNet-34, rimpiazzandolo con una sequenza di strati lineari, ReLU, Dropout, e LogSoftmax. Questo strato finale è stato adattato per la specifica attività di classificazione in tre classi.

Nel metodo forward, l'input x passa attraverso la rete ResNet-34 pre-addestrata e successivamente attraverso la sequenza di strati lineari e di attivazione che abbiamo definito. Questo processo assicura che i dati siano processati efficacemente, sfruttando sia le caratteristiche apprese da ImageNet sia le specifiche modifiche per la nostra task di classificazione.*

Utilizziamo `nn.CrossEntropyLoss`, una scelta standard per compiti di classificazione multiclasse.

Abbiamo scelto ottimizzatore l'Adam, rispetto a Stochastic Gradient Descent (SGD) perchè :

1- Adatta il tasso di apprendimento per ciascun paramentro , mentre SGD rimane fisso.
2- Adam tende ad essere più efficente nella fase iniziale, convergendo più rapidamente rispetto a SGD, questo può essere vantaggioso quando si lavora con dataset complessi e di dimensioni relativamente limitate
3- Adam è generalmente migliore nel maneggiare minimi locali e punti sella grazie alla sua combinazione di momentum e scaling adattivo del gradiente. Questo può essere particolarmente utile negli spettrogrammi, dove le variazioni di intensità e texture possono creare superfici di errore complesse.

*Nota: Inizialmente si considerava di congelare i pesi del modello base, in modo che durante l'addestramento, i pesi del modello pre-addestrato non vengono aggiornati, consentendo solo ai pesi degli strati aggiuntivi di cambiare. Personalizzazione dell'Ultimo Strato (Fully Connected): `nr_filters = self.model.fc.in_features` capta il numero di filtri in entrata dell'ultimo strato lineare di ResNet-34.

Utilizzo rete CNN:

Dopo numerosi tentativi di addestrare una rete neurale con tale dataset, abbiamo constatato che le metriche di accuratezza e precisione che si ottengono con una rete molto semplice ed essenziale, sono migliori di molte reti con più strati e livelli nascosti dunque abbiamo optato per una rete convoluzionale con elementi essenziali.

Architettura della Rete CustomCNN

La rete "CustomCNN" si basa su una struttura convoluzionale per l'elaborazione delle immagini. Le sue caratteristiche principali sono :

```
class CustomCNN(nn.Module):
	def __init__(self, num_classes):
		super(CustomCNN, self).__init__()
		*


*
# Define the first convolutional layer
self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=7)
self.pool = nn.MaxPool2d(2)

# Define the second convolutional layer
self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5)

# Define the third convolutional layer
self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)

# Assuming the input dimensions are (3, 32, 32), the output dimensions after
convolutions are (128, 32, 32)
# This assumes no pooling layers are used. If pooling is used, dimensions will change
accordingly.
self.fc1 = nn.Linear(64 * 25 * 25, 128) # First linear layer
self.fc2 = nn.Linear(128, num_classes) # Second linear layer
```

- **Primo Strato Convolutivo**: Utilizziamo nn.Conv2d con 3 canali in ingresso, 16 canali in uscita e un kernel di 7x7. Questo strato è progettato per estrarre le caratteristiche di base dall'immagine.

- **Pooling**: Dopo ogni strato convolutivo, applichiamo il MaxPooling (`nn.MaxPool2d` con una finestra di 2x2) per ridurre le dimensioni spaziali, mantenendo le informazioni importanti.

- **Secondo e Terzo Strato Convolutivo**: Continuiamo con altri due strati convolutivi, aumentando progressivamente il numero di filtri (32 e poi 64) e riducendo la dimensione del kernel (5x5 e 3x3). Questi strati servono per catturare caratteristiche più complesse.

- **Strati Fully Connected**: Dopo l'appiattimento dell'output convoluzionale, utilizziamo due strati lineari (`nn.Linear`). Il primo trasforma l'output in una dimensione di 128, mentre il secondo lo mappa al numero di classi desiderato.

- **Attivazione e Dropout**: Ogni strato lineare è seguito da una funzione di attivazione ReLU (nn.ReLU) per introdurre non-linearità, e un dropout (`nn.Dropout`) con probabilità di 0.25 per ridurre l'overfitting.

- **Metodo Forward**: Nel metodo forward, definiamo la direzione del flusso dei dati attraverso la rete, applicando sequenzialmente i suddetti strati per elaborare l'input.

CONFRONTO TRA ARCHITETTURE

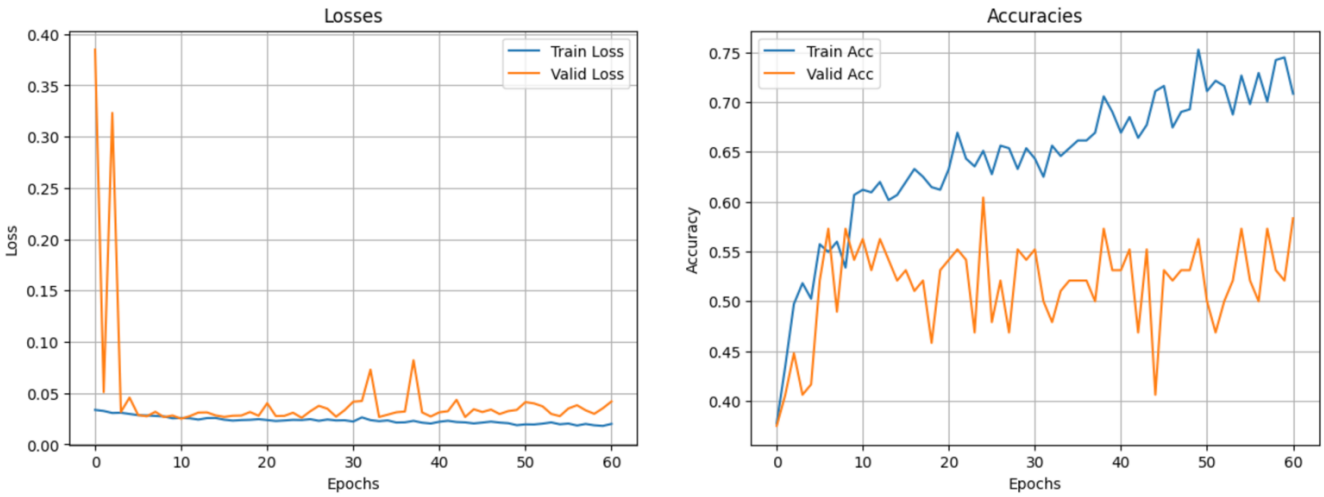
Per comprendere perché utilizzare la rete ResNet-34 per classificare spettrogrammi in tre classi e come confrontarla con una CNN personalizzata è importante considerare le caratteristiche e le differenze tra le due reti:

ResNet-34

(rete molto profonda con 34 strati). È stata progettata per gestire compiti di visione artificiale complessi e può beneficiare di un'ampia varietà di caratteristiche apprese durante l'addestramento su grandi set di dati come ImageNet. Inoltre, l'uso di una rete pre-addestrata come ResNet-34 permette di sfruttare il trasferimento di apprendimento. Questo significa che la rete ha già imparato a riconoscere molte caratteristiche visive generiche che possono essere utili anche per classificare spettrogrammi. Modificando l'ultimo strato completamente connesso, ResNet-34 può essere adattata per classificare spettrogrammi in tre classi. Questo approccio è utile se gli spettrogrammi condividono somiglianze con le immagini naturali (ad esempio, in termini di pattern e texture).

CNN Personalizzata:
La rete più semplice e con meno strati. Questo potrebbe renderla meno potente in termini di capacità di apprendimento rispetto a ResNet-34, ma può essere più veloce da addestrare e più facile da ottimizzare per compiti specifici. Essendo stata progettata specificamente per tale compito, questa rete può essere più adatta a catturare caratteristiche specifiche degli spettrogrammi che non sono presenti in set di dati di immagini naturali come ImageNet. Avere una rete personalizzata ci dà controllo completo sulla sua architettura. Abbiamo potuto sperimentare con diverse configurazioni di strati e parametri per ottimizzare le prestazioni sulla tua specifica attività. In conclusione, ResNet-34 è utile se non hai un grande set di dati per l'addestramento, grazie al trasferimento di apprendimento. ResNet-34 può richiedere più risorse computazionali a causa della sua profondità. Una rete personalizzata può essere ottimizzata specificamente per la classificazione di spettrogrammi.

CAPITOLO 4: RISULTATI



Il grafico mostra due tracciati, uno per le perdite (Losses) e uno per le accuratezze (Accuracies), entrambi in funzione del numero di epoche durante l'addestramento.
Ogni tracciato ha due curve: una per il set di addestramento (Train) e una per il set di validazione (Valid).

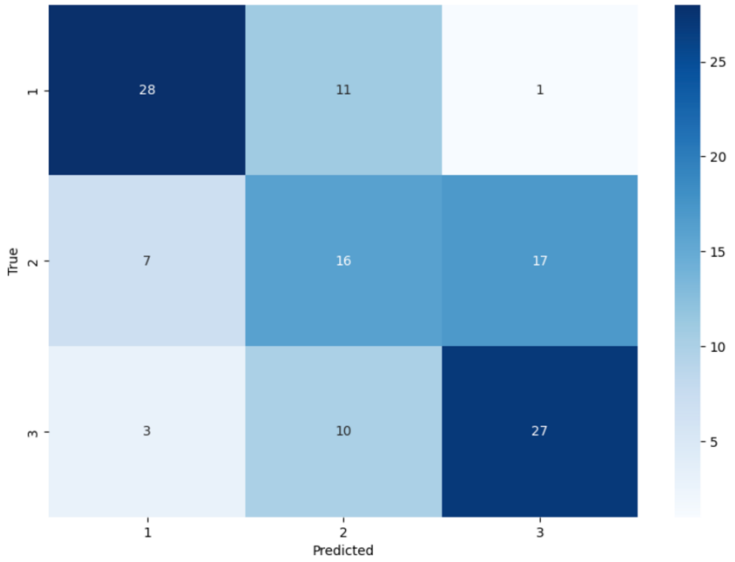
Perdite (Losses):
Sul lato sinistro, il grafico mostra che la perdita di addestramento (Train Loss, in blu) inizia ad un valore alto e diminuisce rapidamente nelle prime epoche, stabilizzandosi vicino allo zero. La perdita di validazione (Valid Loss, in arancione) mostra una leggera variazione, ma nel complesso, rimane significativamente più bassa rispetto al picco iniziale e sembra stabilizzarsi con una leggera tendenza al decremento.

Accuratezze (Accuracies):
Sul lato destro, la train accuracy (in blu) inizia intorno al 45% e aumenta gradualmente, raggiungendo una tendenza ascendente fino ad oltre il 70%.
Al contrario, la valid accuracy (in arancione) inizia vicino al 40% e fluttua notevolmente con il passare delle epoche, mostrando un trend generale di crescita ma con molta più varianza rispetto all'accuratezza di addestramento. Il comportamento ideale è quello in cui entrambe le curve di perdita diminuiscono e le curve di accuratezza aumentano con un andamento simile sia per il set di addestramento che per quello di validazione.

In conclusione, possiamo dedurre che il modello sembra stabilizzarsi, poiché le perdite di addestramento e di validazione diminuiscono e le accuratezze migliorano con il passare delle epoche. Non sembra esserci overfitting poiché le valid losses non aumentano e la valid accuracy non diminuisce man mano che aumentano le epoche. Non sembra esserci nemmeno underfitting, in quanto, anche si di poco, le metriche di prestazione migliorano nel tempo.

RISULTATI IN MATRICI DI CONFUSIONI E ACCURATEZZA

1- Resnet34Pretrained



Accuracy: 0.5917

Precision: 0.5898

Recall: 0.5917

DISCUSSIONE MATRICE Resnet34Pretrained:

La matrice di confusione presentata è riferita alla rete neurale Resnet34Pretrained:

Il modello ha correttamente predetto 28 oggetti come classe 0 (oggetti piccoli), ma ne ha erroneamente predetto 11 come classe 1 (oggetti medi) e 1 come classe 2 (oggetti grandi).
Ci sono stati 7 falsi negativi per la classe 0 e 17 per la classe 2, con solo 16 corrette predizioni per la classe 1.
Per la classe 2, il modello ha fatto discretamente bene , con 27 predizioni corrette, ma ha confuso 3 istanze di classe 2 per classe 0(oggetti piccoli) e 10 per classe 2 (oggetti grandi).

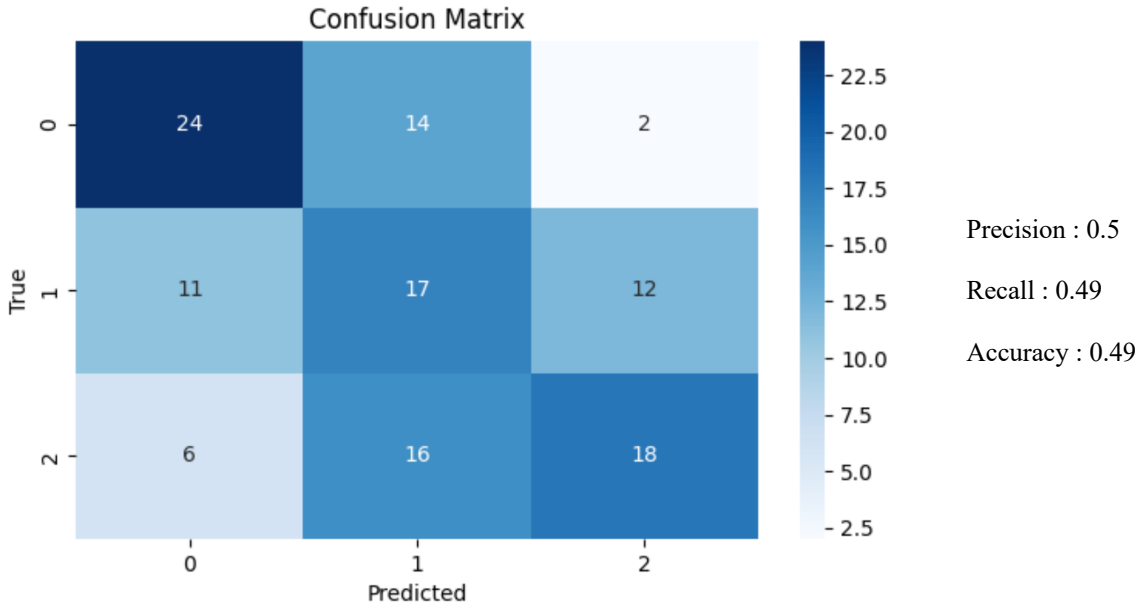
Le metriche di performance sono le seguenti:
Accuracy: 0.5917. Questo valore indica che il 59.17% delle predizioni complessive sono corrette, solitamente non sarebbe un valore accetabile , ma considerando che il caso peggiore è al 33% in questo caso , essendo una classificazione in 3 classi , l'informazione è che ha appreso discretamente.

Precision: 0.5898. Questo valore indica che, quando il modello predice una certa classe, è corretto il 58.98% delle volte. La precisione è una misura di quanto siano affidabili le predizioni positive del modello.

Recall: 0.5917. Questo indica che il modello è in grado di identificare correttamente il 59.17% di tutte le istanze positive reali all'interno della classe. Il recall è una misura di quanto sia completo l'insieme di predizioni positive.

In conclusione, questo modello mostra un equilibrio tra precisione e recall, indicando che non è fortemente sbilanciato verso falsi positivi o falsi negativi.
Tuttavia, l'accuracy generale non è molto alta, suggerendo che potrebbe non essere facile distinguere le classi etreme con quella di mezzo.

2-CNN:



DISCUSSIONE MATRICE CNN:

Oggetti Piccoli (Classe 0): Il modello ha correttamente identificato 24 oggetti piccoli, ma ha scambiato 14 oggetti piccoli per oggetti di dimensioni medie e 2 per oggetti grandi.
Oggetti Medi (Classe 1): 17 oggetti medi sono stati correttamente classificati, ma 11 sono stati confusi con oggetti piccoli e 12 con oggetti grandi. Questo indica che il modello ha difficoltà a distinguere tra oggetti di dimensioni medie e quelle degli altri due gruppi.
Oggetti Grandi (Classe 2): Il modello ha riconosciuto correttamente 18 oggetti grandi. Tuttavia, ha etichettato erroneamente 6 oggetti grandi come piccoli e 16 come medi. Questo suggerisce una tendenza del modello a sottovalutare la dimensione degli oggetti grandi, confondendoli più frequentemente con oggetti di dimensioni medie.
Le metriche di performace sono:
Precisione: 0.5. Questo valore basso implica che quando il modello predice una certa dimensione, è corretto solo la metà delle volte. Ciò può indicare una confusione significativa tra le categorie, specialmente tra oggetti medi e grandi.
Recall: 0.49. Il modello è in grado di individuare meno della metà degli oggetti per ogni dimensione corretta. In termini pratici, ciò significa che molti oggetti non vengono identificati correttamente in base alla loro dimensione.
Accuratezza: 0.49. Questo valore indica che solo il 49% delle predizioni totali del modello su tutte le dimensioni è corretto, suggerendo che le capacità del modello di classificare correttamente le dimensioni degli oggetti sono limitate.