2023/24

MiniContest Machine Learning 3

RICCARDO ROMANO

Università degli Studi di Napoli Federico II

2023/24

Sommario

ANALISI MINI CONTEST3 – MACHINE LEARNING

Descrizione del Dataset

Il set di dati è costituito da campioni .PNG che rappresentano sei diverse condizioni atmosferiche e di illuminazione della scena nel contesto delle ferrovie intelligenti. In particolare, tutte le immagini sono state generate utilizzando RoadRunner di MathWorks, simulando una telecamera RGB montata davanti al treno, senza ostacoli sui binari.

File

Di seguito sono elencati i file presenti all'interno del Dataset fornito per la prova:

- TrainingSet.zip - l'insieme di immagini per il set di addestramento, diviso in sottocartelle per classe
- TestSet.zip - l'insieme di immagini per il set di test
- SampleSubmission.csv - un file di presentazione di esempio nel formato corretto
- SampleFeatureExtractor.zip - un flusso di lavoro di Knime che mostra come estrarre alcune caratteristiche testuali e profonde.
- Features_Textural.zip - un insieme di caratteristiche testuali pre-estratte per entrambi i dataset di allenamento e di test.

Le etichette delle classi sono le seguenti:

- **CL** - sta per Clean (pulito) ed è associata all'etichetta 0
- **BR** - Sta per Bright Right ed è associata all'etichetta 1
- **DA** - Sta per Buio ed è associata all'etichetta 2
- **RA** - Sta per Pioggia ed è associato all'etichetta 3
- **SF** - Sta per Sun Flare (interferenza diretta della luce solare) ed è associato all'etichetta 4
- **SH** - Sta per Ombre ed è associato all'etichetta 5

Rete Neurale Utilizzata: Xception

Xception è un tipo di architettura di rete neurale convoluzionale (CNN) utilizzata per il riconoscimento delle immagini. È stata introdotta da François Chollet di Google AI nel 2017. Il nome "Xception" deriva dall'idea di

"Extreme Inception", in quanto si basa sull'architettura Inception (che utilizza moduli di convoluzione multipla) e implementa un concetto di separazione delle convoluzioni.

L'idea principale di Xception è quella di utilizzare convoluzioni profonde e separabili. Invece di combinare convoluzioni spaziali e di profondità, Xception utilizza convoluzioni profonde separate, cioè convoluzioni spaziali seguite da convoluzioni di profondità. Questo approccio aiuta a ridurre il numero di parametri e a migliorare l'efficienza computazionale della rete, consentendo comunque ottime prestazioni nel riconoscimento delle immagini.

Xception è stata addestrata su un'ampia varietà di immagini provenienti da dataset standard di visione artificiale. Questi dataset includono spesso immagini provenienti da categorie diverse, come ImageNet, un dataset di milioni di immagini suddivise in oltre mille categorie.

Le reti neurali, incluse quelle come Xception, spesso beneficiano di un addestramento su dataset diversificati e rappresentativi per poter generalizzare meglio a una vasta gamma di immagini del mondo reale.

ImageNet è uno dei dataset più comunemente utilizzati per l'addestramento di reti di questo tipo, anche se possono essere addestrate su dataset personalizzati o su insiemi di dati specifici per determinate applicazioni.

In sostanza, Xception è progettato per estrarre caratteristiche da immagini in modo efficiente e preciso, utilizzando una struttura che massimizza la profondità senza aumentare eccessivamente il numero di parametri, il che lo rende una scelta popolare in alcune applicazioni di visione artificiale.

Analisi del Codice

Di seguito è riportata l'analisi del codice delle parti più particolari, infatti sono stati evitati quei lati di codice più semplici come l'importazione del Dataset mediante Kaggle, i vari import o l'unzip.

Split Dataset in Train e Validation

Questo codice Python è progettato per suddividere un dataset di immagini in set di addestramento (training set) e set di validazione (validation set), seguendo una proporzione specifica.

Ecco una spiegazione linea per linea:

1. **Import dei moduli**: Importa i moduli necessari (*os* per operazioni di sistema, *shutil.copyfile* per copiare file e *random* per la generazione di numeri casuali).
2. **Definizione del percorso del dataset**: Specifica il percorso della cartella principale del dataset.
3. **Definizione delle etichette**: Elenca le etichette delle cartelle presenti nel dataset.
4. **Definizione delle proporzioni**: Specifica le proporzioni desiderate per la divisione del dataset (70% per training, 30% per validation).
5. **Creazione della cartella per il dataset suddiviso**: Crea una nuova cartella chiamata "dataset_diviso" nella stessa posizione del dataset originale, se non esiste già.
6. **Impostazione del seed per la riproducibilità**: Utilizza un seed fisso per garantire la stessa divisione dei dati ogni volta che il codice viene eseguito.
7. **Iterazione attraverso le etichette**: Per ogni etichetta nel dataset:
    a. *Raccogli i file*: Ottiene la lista di tutti i file nella cartella corrispondente all'etichetta attuale.
    b. *Mischia i file*: Mescola casualmente l'ordine dei file utilizzando un seed fissato in precedenza.
    c. *Calcola le dimensioni dei set*: Calcola il numero di file necessario per ciascun set (training e validation) in base alle proporzioni specificate.
    d. *Crea le cartelle per i set*: Crea le cartelle per i set di training e validation all'interno della cartella "dataset_diviso".

e. *Copia i file nei rispettivi set*: Copia i file nelle cartelle appropriate per i set di training e validation.

8. **Stampa dei risultati**: Stampa il numero di file presenti in ciascuna cartella di training e validation per ogni etichetta, mostrando la distribuzione dei dati dopo la divisione.
9. **Messaggio di completamento**: Stampa un messaggio che conferma il successo della suddivisione del dataset.

A cosa serve dunque?

Questo codice serve a suddividere un dataset di immagini in due insiemi principali: un training set e un validation set. Il training set viene utilizzato per addestrare un modello di machine learning, mentre il validation set viene impiegato per valutare le prestazioni del modello durante il processo di addestramento.

Il codice prende un dataset di immagini organizzato in cartelle etichettate, ad esempio immagini di diverse categorie o classi. Utilizza proporzioni predefinite (come il 70% per il training e il 30% per la validazione) per dividere ogni classe di immagini in due sottoinsiemi: uno per il training e uno per la validazione.

Questo processo di divisione è importante per garantire che il modello di machine learning non solo impari dai dati disponibili, ma anche che sia in grado di generalizzare bene su dati non visti, aiutando a evitare il fenomeno dell'overfitting ai dati di addestramento.

In breve, questo codice è un passo preliminare fondamentale per preparare e organizzare i dati in modo ottimale prima di addestrare un modello di machine learning su un dataset di immagini.

Definizione di Train Loop Validation

La definizione di tale funzione è così riportata:

```
def train_loop_validation(dataloaders, startEpoch, numEpochs, model_conv, criterionCNN, optimizer_conv, best_acc, best_loss, best_epoca, outputPath)
```

Tale cella di codice rappresenta un ciclo di addestramento di un modello di rete neurale convoluzionale (CNN) utilizzando i dati presenti nei *dataloaders* suddivisi in training e validation set. Il ciclo si sviluppa attraverso diverse iterazioni (***epoche***) di addestramento per cercare il modello migliore basato sulle prestazioni sul set di validation.

Ecco una spiegazione delle principali sezioni e operazioni all'interno del codice:

- **Ciclo di addestramento e validazione:**
  - Il ciclo *for epochs in range(startEpoch, numEpochs + 1)* esegue un numero specificato di epoche di addestramento, iniziando dalla *startEpoch* e terminando a *numEpochs*.
  - Il modello viene impostato in modalità di addestramento con *model_conv.train()*.
  - Viene iterato attraverso il set di training *(`dataloaders['train']`)* in batch.
  - Per ogni batch di dati, vengono eseguiti i seguenti passaggi:
    - Le immagini e le etichette vengono caricate sulla GPU *(`inputs = inputs.type(torch.FloatTensor).cuda()`* e *`labels = labels.cuda()`).*
    - Il modello effettua una previsione *(`y = model_conv(inputs)`)* e calcola la perdita (***loss***) rispetto alle etichette reali.
    - La loss viene utilizzata per ottimizzare i pesi del modello con *`lossCNN.backward()`* e *`optimizer_conv.step()`*.
    - Vengono calcolati la loss media e l'accuratezza per l'intero set di training per ogni epoca.
    - I pesi del modello vengono salvati in un file *`.pth`* ad ogni iterazione.

- **Valutazione sul set di validation:**
  - Dopo ogni epoca di addestramento, il modello viene impostato in modalità di valutazione con *`model_conv.eval()`.*
  - Si esegue un loop attraverso il set di validation *(`dataloaders['val']`)* in batch.
  - Per ogni batch, si calcola la perdita e l'accuratezza sul set di validation.
  - Viene calcolata la perdita media e l'accuratezza per l'intero set di validation per ogni epoca.

- **Salvataggio dei migliori pesi e dei risultati:**
  - Viene confrontata l'accuratezza e la perdita ottenute sul set di validation con i valori precedentemente registrati *(`best_acc` e `best_loss`).*
  - Se i nuovi valori di accuratezza e perdita sono migliori di quelli precedenti, i pesi del modello vengono salvati come i migliori fino a quel punto.
  - I risultati vengono stampati a schermo, mostrando la perdita e l'accuratezza sia per il set di training che per il set di validation.
  - I risultati (loss e accuracy) vengono salvati in file di testo separati per training e validation, mentre i migliori pesi del modello vengono salvati in un file separato *(`best_model_weights.pth`).*

- **Salvataggio dei checkpoint:**
  - Viene salvato un file *check_point.mat* contenente le informazioni sui migliori risultati ottenuti fino a quel momento.

In sintesi, questo codice addestra un modello CNN su un dataset, valutandone le prestazioni su un set di validation per trovare il modello con la migliore accuratezza e perdita su questi dati di validazione. I migliori pesi del modello vengono salvati periodicamente durante l'addestramento per poter riprendere l'addestramento da un certo punto in caso di interruzioni o per confrontare diversi modelli.

A cosa serve questo codice?

Questo codice esegue un ciclo di addestramento e validazione di un modello di rete neurale convoluzionale. Durante ogni epoca di addestramento, il modello viene allenato sui dati di training e valutato sui dati di validation per monitorare le prestazioni.

La principale funzione di questo codice è:

1. **Addestramento del modello:**
   a. Utilizza i dati di training per aggiornare i pesi del modello.
   b. Calcola la perdita e l'accuratezza del modello sui dati di training.
   c. Salva i pesi del modello ad ogni iterazione.
2. **Validazione del modello:**
   d. Valuta le prestazioni del modello sui dati di validation senza modificare i pesi del modello.
   e. Calcola la perdita e l'accuratezza del modello sui dati di validation.
3. **Controllo e salvataggio dei migliori risultati:**
   f. Monitora le metriche di perdita e accuratezza ottenute durante la validazione.
   g. Salva i pesi del modello se vengono ottenuti risultati migliori rispetto ai migliori finora registrati.
4. **Registrazione dei risultati e checkpoint:**
   h. Salva i risultati delle metriche di perdita e accuratezza su file di testo separati per training e validation.
   i. Salva un checkpoint contenente informazioni sui migliori risultati finora ottenuti.

In sintesi, questo codice gestisce l'addestramento e la valutazione di un modello CNN, cercando di migliorare le sue prestazioni durante le varie epoche di addestramento e mantenendo traccia dei migliori risultati ottenuti fino a quel momento. È utile per sviluppare e ottimizzare modelli di machine learning basati su dati di immagini, consentendo di monitorare e registrare i progressi del modello durante l'addestramento.

Settaggio degli Iperparametri

Di seguito è riportato il settaggio degli iperparametri presi in considerazione all'interno del minicontest.

```
#rendere l'esecuzione deterministica
torch.manual_seed(0)
np.random.seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

#Update 11/12: Diminuito valore del learning rate per avere grafici più precisi
learning_rate = 0.00001
batch_size = 32
set_classes_number = 6
num_epoch = 20
```

Questo codice configura l'ambiente di esecuzione per garantire la riproducibilità e la coerenza dei risultati nell'addestramento di modelli di reti neurali utilizzando il framework PyTorch.

- **Rendimento dell'esecuzione deterministica:**
  - *`torch.manual_seed(0)`:* Imposta il seed per i generatori di numeri casuali di PyTorch per garantire che la generazione di numeri casuali all'interno del framework sia deterministica quando si utilizzano funzioni che coinvolgono la casualità, come l'inizializzazione dei pesi del modello o la creazione di dati casuali.
  - *`np.random.seed(0)`:* Imposta il seed per il generatore di numeri casuali di NumPy per garantire la stessa deterministica quando si utilizzano funzioni di NumPy insieme a PyTorch.
  - *`torch.backends.cudnn.deterministic = True`:* Fissa la deterministica nell'utilizzo delle funzionalità di ottimizzazione di cuDNN (CUDA Deep Neural Network library), garantendo la riproducibilità nell'addestramento quando si utilizza la GPU.
  - *`torch.backends.cudnn.benchmark = False`:* Disabilita la modalità di benchmark di cuDNN che ottimizza automaticamente le operazioni, ma potrebbe causare variazioni nelle prestazioni e rendere non deterministico l'addestramento.

- **Impostazioni di addestramento:**
  - *`learning_rate = 0.00001`:* Specifica il tasso di apprendimento (learning rate) utilizzato nell'ottimizzazione durante l'addestramento del modello. Un learning rate basso può consentire aggiornamenti più piccoli dei pesi del modello ad ogni iterazione, ma può richiedere più tempo per l'addestramento.
  - *`batch_size = 32`:* Indica la dimensione del batch di dati utilizzati durante l'addestramento del modello. Ogni batch conterrà 32 esempi dal dataset.
  - *`set_classes_number = 6`:* Rappresenta il numero di classi o categorie presenti nel problema di classificazione a cui si sta lavorando. In questo caso, ci sono 6 classi.
  - *`num_epoch = 20`:* Definisce il numero di epoche, ovvero il numero di volte in cui l'intero dataset viene utilizzato per addestrare il modello.

A cosa serve dunque?

Il codice stabilisce condizioni specifiche per l'esecuzione dell'addestramento di un modello di rete neurale utilizzando PyTorch, garantendo la determinismo nei risultati e impostando i parametri chiave per l'addestramento, come il tasso di apprendimento, la dimensione del batch e il numero di epoche.

Divisione Training e Validation in cartelle separate

Questo codice è parte di un processo di caricamento e trasformazione di dati di immagini per l'addestramento di un modello di rete neurale convoluzionale (CNN) utilizzando PyTorch.

Ecco cosa fa:

1. **Definizione delle trasformazioni dei dati:**
   - `transforms.Compose`: È una funzione di PyTorch che consente di concatenare diverse trasformazioni da applicare alle immagini. Le trasformazioni specificate vengono applicate in sequenza.
   - Per il set di training (`**'train'**`), le trasformazioni includono:
     i. ***`transforms.RandomResizedCrop(224)`:*** Ritaglia casualmente e ridimensiona l'immagine a una dimensione di 224x224 pixel.
     ii. ***`transforms.RandomHorizontalFlip()`:*** Applica casualmente un'operazione di ribaltamento orizzontale all'immagine.
     iii. ***`transforms.ToTensor()`:*** Converte l'immagine in un tensore PyTorch.
     iv. ***`transforms.Normalize(mean, std)`:*** Normalizza i canali dell'immagine in base a una media e deviazione standard specificate. Questi valori sono tipicamente basati su una normalizzazione predefinita per le immagini.
   - Per il set di validation (`**'val'**`), le trasformazioni includono:
     i. ***`transforms.Resize(256)`:*** Ridimensiona l'immagine a 256x256 pixel.
     ii. ***`transforms.CenterCrop(224)`:*** Esegue un ritaglio centrato dell'immagine a una dimensione di 224x224 pixel.
     iii. Le altre trasformazioni rimangono le stesse del set di training.

2. **Caricamento dei dati:**
   - `data_dir`: Specifica il percorso della cartella contenente il dataset diviso tra training e validation.
   - `image_datasets`: Carica i dataset di immagini utilizzando `datasets.ImageFolder`, che organizza le immagini in base alla struttura delle cartelle, dove ogni sottocartella rappresenta una classe. Applica le trasformazioni definite in `data_transforms`.
   - `dataloaders`: Utilizza `torch.utils.data.DataLoader` per creare dei dataloader per il training e la validation, consentendo di caricare i dati in batch, mischiarli e utilizzare più processi (`num_workers`) per il caricamento parallelo dei dati.

3. **Altre informazioni sul dataset:**
   - `dataset_sizes`: Memorizza la dimensione di ciascun dataset (training e validation).
   - `class_names`: Ottiene i nomi delle classi presenti nel set di training.

5. **Utilizzo della GPU (se disponibile):**

- `device`: Contiene il dispositivo su cui verrà eseguito l'addestramento, preferibilmente la GPU se disponibile, altrimenti la CPU.

6. **Stampa delle classi:**
   - Stampa i nomi delle classi presenti nel set di training (`class_names`) e nel set di validation (`image_datasets['val'].classes`). Il codice per il set di test è attualmente commentato.

A cosa serve dunque?

Questo codice serve per preparare e caricare dati di immagini in un formato adatto per addestrare modelli di reti neurali convoluzionali utilizzando PyTorch.

In breve, questo codice è fondamentale per la preparazione dei dati prima dell'addestramento di un modello di rete neurale per la classificazione di immagini. Trasforma, organizza e carica i dati in un formato adatto all'addestramento del modello, rendendo il processo più efficiente e facilitando la creazione di reti neurali che possono imparare dai dati di immagini forniti.

Perché devo effettuare quelle trasformazioni?

Le trasformazioni dei dati sono cruciali nell'ambito dell'apprendimento automatico e in particolare nell'addestramento di modelli di reti neurali convoluzionali per diverse ragioni:

- **Uniformità dei dati:** Le trasformazioni assicurano che tutti i dati siano coerenti in termini di formato, dimensione e scala. Le reti neurali richiedono spesso che i dati siano standardizzati in modo uniforme per garantire che il modello apprenda in modo efficiente e consistente.

- **Preparazione dei dati:** Le immagini possono variare notevolmente in termini di dimensioni, orientamento, luminosità, etc. Le trasformazioni consentono di standardizzare tali variazioni e rendere i dati più facilmente interpretabili per il modello.

- **Riduzione dell'overfitting:** Le trasformazioni come il ritaglio casuale, il ribaltamento e la normalizzazione possono aiutare a generare più dati artificiali o a migliorare la capacità del modello di generalizzare su dati nuovi e non visti, riducendo tale rischio.

- **Compatibilità con l'architettura del modello:** Alcune reti neurali richiedono dati di input di dimensioni specifiche o devono essere alimentate con dati normalizzati in un certo modo. Le trasformazioni sono utili per adattare i dati al formato richiesto dal modello.

- **Miglioramento delle prestazioni:** Trasformazioni appropriate possono migliorare l'efficienza dell'addestramento del modello, accelerando la convergenza del processo di apprendimento e migliorando le prestazioni del modello sui dati di test.

In sintesi, le trasformazioni dei dati sono cruciali perché preparano e standardizzano i dati in modo da renderli adatti all'addestramento del modello. Questo processo è fondamentale per migliorare la capacità del modello di generalizzare su nuovi dati e ridurre problemi come l'overfitting.

Creazione del modello con rete Xception

Prima di procedere con la creazione del modello, per utilizzare la rete convoluzionale Xception in Python, è necessario installare *Timm*.

```
1. !pip install timm
```

Timm, acronimo di "PyTorch Image Models", è una libreria progettata per semplificare l'utilizzo di modelli avanzati di deep learning nell'ambito dell'elaborazione delle immagini utilizzando PyTorch, una libreria di machine learning.

Questo codice crea e configura un modello di rete neurale utilizzando l'architettura Xception da TensorFlow Hub e lo adatta per l'addestramento su un nuovo problema di classificazione di immagini utilizzando PyTorch.

Ecco una spiegazione passo per passo:

- **Importazione delle librerie:**
  - *`import timm`:* Importa la libreria "timm".

- **Definizione del modello:**
  - *`networkName = 'xception'`:* Specifica il nome dell'architettura della rete neurale (in questo caso, Xception).
  - *`WeightPath`:* Definisce il percorso in cui verranno salvati i pesi del modello durante l'addestramento.
  - *`model_conv = timm.create_model("xception", pretrained=True)`:* Utilizza la libreria "timm" per creare un modello Xception preaddestrato su grandi dataset come ImageNet.
  - *`num_ftrs = model_conv.fc.in_features`:* Ottiene il numero di feature nel layer fully connected (fc) del modello Xception.
  - *`model_conv.fc = nn.Linear(num_ftrs, set_classes_number)`:* Sostituisce l'ultimo layer fully connected con uno nuovo non addestrato con un numero di output corrispondente al numero di classi nel nuovo problema (`set_classes_number`).

- **Configurazione dell'addestramento:**
  - *`model_conv = model_conv.cuda()`:* Muove il modello sulla GPU (se disponibile) per l'addestramento.
  - *`criterion = nn.CrossEntropyLoss()`:* Definisce la funzione di loss da ottimizzare durante l'addestramento, in questo caso la Cross Entropy Loss, comunemente utilizzata per problemi di classificazione.
  - *`optimizer_conv = optim.Adam(model_conv.parameters(), lr=learning_rate)`:* Specifica l'ottimizzatore Adam per aggiornare i pesi del modello durante l'addestramento.

- **Caricamento dei pesi precedenti (se disponibili):**
  - *`checkpoint_path_train`:* Percorso per i pesi del modello salvati durante l'addestramento.
  - `os.path.exists(checkpoint_path_train)`: Controlla se esiste un checkpoint dei pesi del modello salvati.

    Se esiste, carica i pesi salvati per riprendere l'addestramento da dove era stato interrotto. Altrimenti, inizia un nuovo addestramento.

A cosa serve dunque?

Questo codice serve a creare e configurare un modello di rete neurale utilizzando l'architettura Xception, preaddestrata su un ampio dataset di immagini, per l'addestramento su un nuovo problema di classificazione di immagini. Configura l'addestramento del modello utilizzando la Cross Entropy Loss come funzione di perdita e l'ottimizzatore Adam. Controlla se ci sono pesi preaddestrati salvati per riprendere l'addestramento da dove era stato interrotto o inizia un nuovo addestramento.

Inizio fase di Training

```
1. startEpoch =1
2. best_acc = 0
3. best_loss=0
4. best_epoca = 0
5.
6.  train_loop_validation(dataloaders, startEpoch, num_epoch, model_conv, criterion, optimizer_conv, best_acc, best_loss, best_epoca, WeightPath)
```

- **dataloaders**: Questo è un insieme di dataloaders che contiene i dati per il training e la validazione. Questi dataloaders vengono utilizzati per caricare i dati durante l'addestramento del modello.

- **startEpoch = 1**: Questo indica da quale epoca iniziare l'addestramento. L'epoca rappresenta una singola iterazione completa dell'intero dataset durante il processo di addestramento del modello.

- **num_epoch:** Rappresenta il numero totale di epoche per cui si desidera addestrare il modello.

- **model_conv**: Questo è il modello di rete neurale su cui verrà effettuato l'addestramento.

- **criterion**: Rappresenta la funzione di perdita utilizzata durante l'addestramento del modello. Nell'esempio precedente, è stata definita come Cross Entropy Loss.

- **optimizer_conv**: Questo è l'ottimizzatore (nel caso specifico, l'Adam optimizer) utilizzato per aggiornare i pesi del modello durante l'addestramento.

- **best_acc, best_loss, best_epoca**: Queste variabili memorizzano le migliori metriche di accuratezza e perdita ottenute durante l'addestramento del modello. Inizialmente, sono impostate a 0, indicando che non sono state ancora calcolate.

- **WeightPath**: Questo è il percorso in cui vengono salvati i pesi migliori del modello durante l'addestramento.

Valutazione Accuracy e creazione grafici di Loss

Questo frammento di codice legge i file contenenti i dati di loss e accuracy durante l'addestramento di un modello e crea dei grafici per visualizzare le variazioni di tali metriche nel corso delle epoche.

Ecco come funziona:

- **Inizializzazione delle liste vuote:**
  - `lossModel_Train`, `lossModel_val`, `accModel_Train`, `accModel_val`: Queste sono liste vuote che verranno riempite con i dati di loss e accuracy letti dai file.

- **Lettura dei file e caricamento dei dati:**
  - Viene aperto un file per la loss dei dati di training (``'lossTrain.txt'`) e per la loss della validation (``'lossVal.txt'`). Ogni riga del file contiene un valore di loss.
  - I valori vengono letti riga per riga, convertiti in numeri float e aggiunti alle liste `lossModel_Train` e `lossModel_val`.

- **Creazione del grafico per la Loss:**
  - Viene creato un grafico per visualizzare l'andamento della loss durante il training e la validation nel corso delle epoche.
  - Sull'asse x sono rappresentate le epoche, sull'asse y la loss.
  - I valori di loss per il training e la validation vengono rappresentati sui grafici con colori diversi (``'r'` per il training e ``'g'` per la validation).
  - Infine, il grafico viene salvato come immagine con il nome ``'LossTrainVal.png'` nel percorso specificato da `WeightPath`.

- **Lettura dei file e caricamento dei dati di accuracy:**
  - Similmente a quanto fatto per la loss, vengono letti i file contenenti i valori di accuracy del training e della validation (``'AccTrain.txt'` e ``'AccVal.txt'`) e i valori letti vengono aggiunti alle liste `accModel_Train` e `accModel_val`.

- **Creazione del grafico per l'Accuracy:**
  - Viene creato un secondo grafico per visualizzare l'andamento dell'accuracy durante il training e la validation nel corso delle epoche.
  - Anche qui, sull'asse x sono rappresentate le epoche, sull'asse y l'accuracy.
  - I valori di accuracy per il training e la validation vengono rappresentati con colori diversi (``'r'` per il training e ``'g'` per la validation).
  - Infine, il grafico viene salvato come immagine con il nome ``'AccTrainVal.png'` nel percorso specificato da `WeightPath`.

Grafici

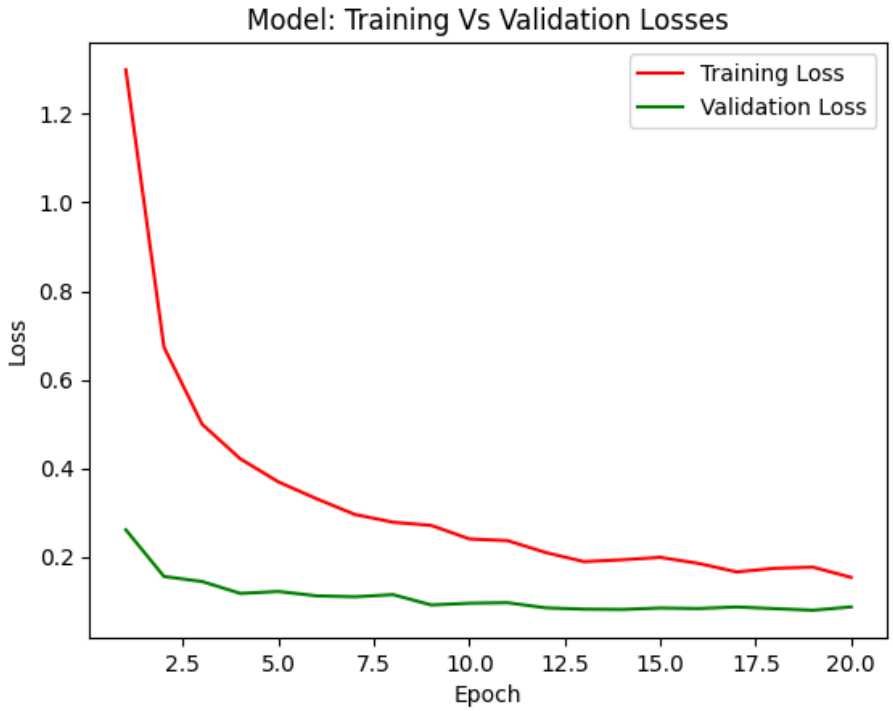

Figura 1: Grafico Loss

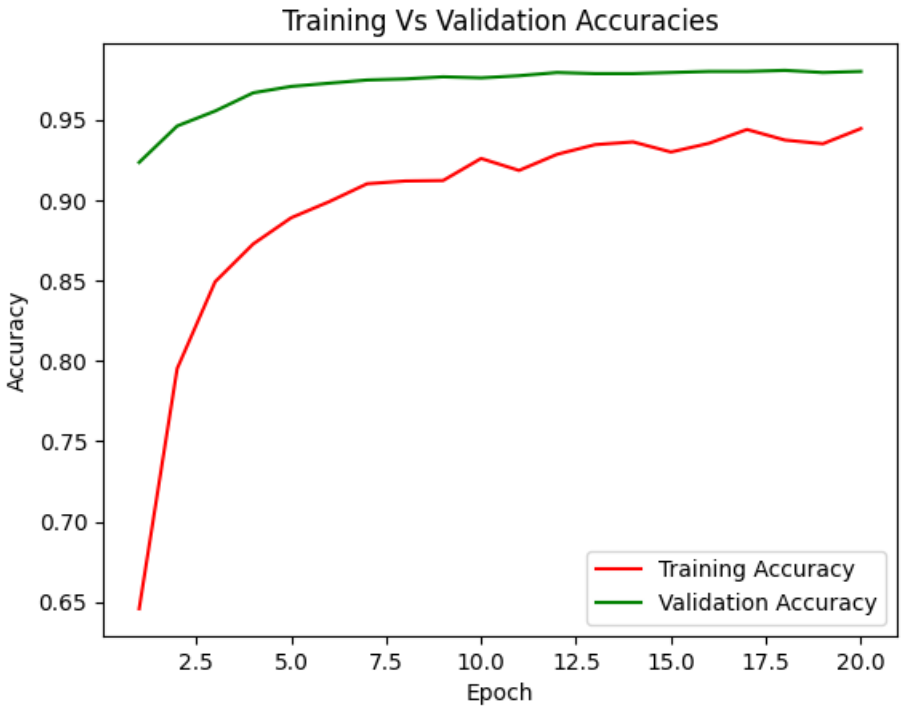

Figura 2: Grafico Accuracy

A cosa serve dunque?

Questo codice serve a visualizzare graficamente l'andamento della loss e dell'accuracy del modello durante il processo di addestramento e validazione. Legge i valori di loss e accuracy salvati in file di testo e crea due grafici separati:

- **Grafico della Loss:**
  - Mostra come varia la loss nel corso delle epoche per il training (`lossTrain.txt`) e la validation (`lossVal.txt`).
  - Due linee separate per la loss del training e della validation sono rappresentate in colori diversi (rosso per il training, verde per la validation).
  - Il grafico viene salvato come immagine.

- **Grafico dell'Accuracy:**
  - Visualizza come cambia l'accuracy durante le epoche per il training (`AccTrain.txt`) e la validation (`AccVal.txt`).
  - Due linee separate per l'accuracy del training e della validation sono rappresentate in colori diversi (rosso per il training, verde per la validation).
  - Il grafico viene salvato come immagine.

Questi grafici forniscono una rappresentazione visuale dell'apprendimento del modello nel corso delle epoche, permettendo di valutare come loss e accuracy cambiano durante l'addestramento e la validazione. Essi sono utili per identificare l'efficacia del modello, valutarne la performance e individuare eventuali problemi come overfitting o underfitting.

Caricamento modello preaddestrato & preparazione modello per inferenza

Questo codice svolge due operazioni, ovvero il caricamento di un modello preaddestrato, utilizzando i pesi salvati in una run precedente & la preparazione del modello per l'inferenza.

Ma cos'è questa **inferenza**?

Nell'ambito del machine learning e dell'intelligenza artificiale, l'inferenza si riferisce al processo in cui un modello addestrato applica le conoscenze acquisite durante l'addestramento per fare predizioni o trarre conclusioni su nuovi dati. Quando un modello è addestrato, apprende relazioni, pattern e caratteristiche nei dati di training. L'inferenza rappresenta il passaggio successivo, in cui il modello applica queste conoscenze per effettuare previsioni su dati precedentemente non visti, cercando di generalizzare la sua capacità di comprendere e risolvere problemi.

Dunque, l'inferenza è la fase in cui il modello mette in pratica la sua capacità di generalizzazione e applica le sue conoscenze per fare predizioni o compiere azioni su nuovi dati, senza eseguire ulteriori aggiustamenti o modifiche ai parametri del modello.

```
# Carica il modello addestrato
model_path = WeightPath + 'best_model_weights.pth'
checkpoint = torch.load(model_path)
#model_conv.load_state_dict(checkpoint['model_state_dict']) # Utilizzare se si vuole caricare i pesi

# Verifica che la chiave 'state_dict' sia presente
if 'state_dict' in checkpoint:
    state_dict = checkpoint['state_dict']
else:
    state_dict = checkpoint
```

```
11.
12. # Adatta le chiavi del dizionario in base alla struttura del modello
13. adjusted_state_dict = {}
14. for key, value in state_dict.items():
15.     new_key = key.replace('module.', '')  # Rimuove 'module.' se presente (dipende dalla modalità di salvataggio)
16.     adjusted_state_dict[new_key] = value
17.
18. # Carica lo stato del modello
19. model_conv.load_state_dict(adjusted_state_dict)
```

- Caricamento di un modello addestrato:
  - Carica i pesi di un modello precedentemente addestrato (modello Xception in questo caso) da un percorso specifico (WeightPath + 'best_model_weights.pth').

- Preparazione del modello per l'inferenza:
  - Adatta le chiavi del dizionario dei pesi del modello per essere compatibili con la struttura del modello stesso.
  - Carica lo stato addestrato del modello.

➢ Ma cosa sono queste **chiavi**?

Le chiavi a cui si fa riferimento qui sono gli identificatori delle varie parti o componenti del modello salvate nei pesi durante il processo di addestramento. Nei modelli di reti neurali profonde, i pesi vengono salvati solitamente come un dizionario che associa nomi o chiavi a tensori che rappresentano i pesi di specifici strati o componenti del modello. Quando si salva un modello, è comune associare a ciascun tensore di peso un nome univoco che identifica quel particolare tensore all'interno del modello. Questi nomi o identificatori, chiamati chiavi, vengono utilizzati quando si salvano e si caricano i pesi.

➢ Perché bisogna **adattarle** al modello attuale?

A volte possono esserci differenze nelle convenzioni di denominazione o nei nomi delle chiavi tra modelli o versioni diverse dello stesso modello, a seconda di come è stato implementato il salvataggio dei pesi. Pertanto, quando si carica un modello preaddestrato in un modello attuale, potrebbero essere necessari dei passaggi per adattare le chiavi dei pesi del modello preaddestrato in modo che siano compatibili con la struttura del modello attuale.

In sintesi, adattare le chiavi dei pesi significa effettuare eventuali modifiche o manipolazioni necessarie alle etichette o ai nomi delle parti del modello memorizzate nei pesi, in modo che corrispondano alla struttura del modello in cui vengono caricati. Questo garantisce che i pesi vengano assegnati alle giuste parti del modello durante il caricamento senza generare errori dovuti a discrepanze nelle etichette dei pesi.

Trasformazioni, Classificazione delle Immagini e creazione del file CSV

- In questa parte di codice, inizialmente viene specificato un percorso contenente nuove immagini da classificare *(/content/TestSet)*.
- Vengono applicate le trasformazioni di preprocessing alle immagini utilizzando le stesse trasformazioni usate per l'addestramento.

```
# Applica le trasformazioni
normalize = transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
data_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    normalize
])
```

- Si esegue l'inferenza sulle nuove immagini utilizzando il modello addestrato.
- Si mappano i risultati delle predizioni delle classi a etichette numeriche specifiche.

```
# Lista dei risultati
results = []

# Effettua la classificazione
for img_path in new_images:
    img = Image.open(img_path).convert('RGB')
    img_tensor = data_transforms(img).unsqueeze(0)
    img_tensor = img_tensor.cuda()

    with torch.no_grad():
        output = model_conv(img_tensor)
        _, pred = torch.max(output, 1)
        class_name = class_names[pred.item()]

    # Estrai il nome del file (senza percorso)
    file_name = os.path.basename(img_path)

    # Estrai l'indice numerico dal nome del file
    index = int(file_name.split('.')[0])

    # Estrai l'estensione dal nome del file
    extension = file_name.split('.')[-1]

    # Mappa il nome della classe alla sua etichetta numerica
    class_label = {'CL': 0, 'BR': 1, 'DA': 2, 'RA': 3, 'SF': 4, 'SH': 5}[class_name]

    # Aggiungi il risultato alla lista
    results.append({'rowID': f'{index}.{extension}', 'Class': class_label})
```

Dopo aver effettuato la classificazione, si procede con la creazione del file CSV per effettuare la submission:

```
# Scrivi i risultati in un file CSV

csv_path = '/content/result' + networkName + '.csv'
with open(csv_path, 'w', newline='') as csvfile:
    fieldnames = ['rowID', 'Class']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    # Scrivi l'intestazione
    writer.writeheader()

    # Scrivi i dati
    writer.writerows(results)

print(f'I risultati sono stati scritti in: {csv_path}')
```

Dubbio: Congelamento dei pesi degli strati pre-addestrati

Perché bisognerebbe congelare i pesi?

Non congelando i pesi degli strati preaddestrati e permettendo l'aggiornamento di tutti i parametri del modello durante l'addestramento, il modello potrebbe dimenticare o sovrascrivere le informazioni apprese durante il preaddestramento. Ciò potrebbe avvenire perché il gradiente proveniente dall'errore calcolato durante la retropropagazione verrebbe applicato a tutti i parametri, inclusi quelli che avevano già appreso delle buone rappresentazioni durante il preaddestramento. In pratica, il fine-tuning con il congelamento degli strati preaddestrati mira a sfruttare le conoscenze generali acquisite durante il preaddestramento e a personalizzare solo una parte del modello per il nuovo compito specifico.

Se **non congelassi** gli strati preaddestrati, si potrebbe ancora ottenere risultati accettabili, ma ciò dipende da diversi fattori, come:

- la quantità di dati di addestramento disponibili
- la complessità del modello
- la natura specifica del compito di predizione meteorologica

In generale, il congelamento degli strati preaddestrati è spesso una pratica consigliata quando si utilizzano modelli preaddestrati, poiché aiuta a preservare le informazioni utili già apprese durante il preaddestramento.

Perché non vengono congelati invece?

Ecco alcuni punti chiave in aiuto alla tesi del **non congelamento**:

- **Quantità di Dati di Addestramento:**
  Disponendo di una quantità considerevole di dati di addestramento specifici per il contest di ambito meteorologico, si ha la possibilità di permettere al modello di adattarsi completamente a questi dati senza il rischio di dimenticare le informazioni generali apprese durante il preaddestramento.

- **Complessità del Modello:**
  Il tuo modello è relativamente semplice, dunque la sua capacità di estrarre caratteristiche generali è limitata, e quindi l'adattamento completo ai nuovi dati potrebbe essere benefico.

- **Natura del Compito:**
  La natura specifica del compito di previsione meteorologica richiede una maggiore flessibilità del modello per adattarsi a pattern complessi nei dati meteorologici.

- **Risultati Sperimentali:**
  Eseguendo esperimenti comparativi tra il modello con il congelamento degli strati e il modello senza, è stato possibile notare un lieve cambiamento tra i due; si evince che il modello senza congelamento è meno rapido ma performa meglio evitando comunque l'overfitting.

- **Apertura al Cambiamento:**
  La scelta di non congelare i pesi è volta anche a sperimentare diverse configurazioni e strategie aprendo possibili modifiche basate sulla valutazione dei risultati.