UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato in Cognitive Computing Systems

# Visual Recognition per l'identificazione e la classificazione dei tumori cerebrali

Anno Accademico 2022/2023

Docente

**Paolo Maresca**

Studenti

**Ferdinando Simone D'Agostino (M63001274)**
**Simone D'Orta (M63001283)**

# Indice

# Introduzione

L’obiettivo della relazione è quello di documentare il processo di progettazione e di implementazione di un sistema di Visual Recognition che ha lo scopo di classificare due diversi tipi di tumore cerebrale a partire dalle immagini di risonanza magnetica (RMI). In particolare, le due classi di tumore cerebrale che si intende classificare sono le metastasi (MET) e il glioblastoma (GBM). La realizzazione di un sistema in grado di distinguere in maniera accurata le due classi potrebbe essere un utile supporto per gli specialisti durante la fase di diagnosi.

Nel primo capitolo sarà presentata una descrizione più accurata del problema e degli strumenti che sono stati introdotti per la soluzione. In particolare, saranno mostrati i due dataset a disposizione (VOL e MULTI) e il framework open-source Fuse-Med-ML sviluppato da IBM.

Nel secondo capitolo sarà presentata l’implementazione del classificatore addestrato con il primo dataset.

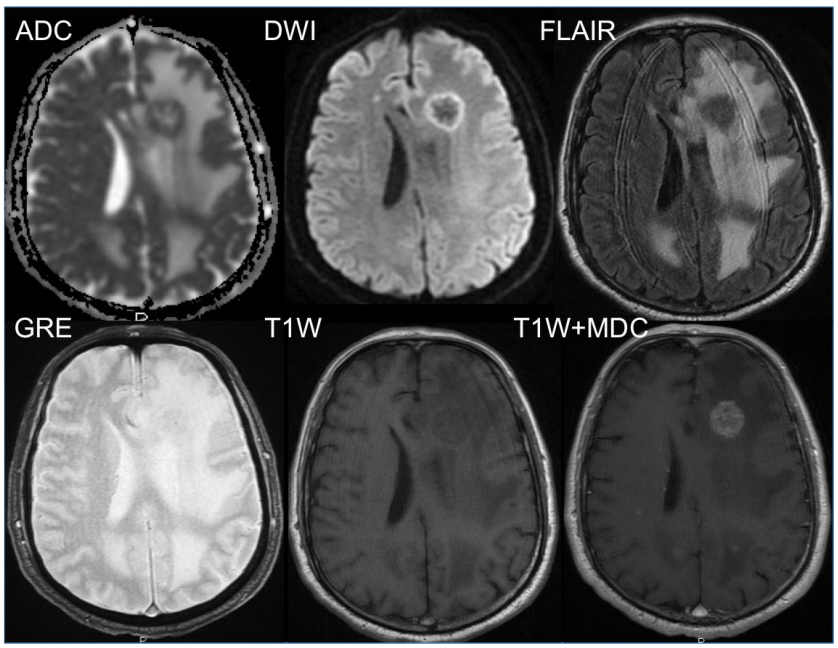
Nel terzo capitolo sarà presentata l’implementazione del classificatore addestrato con il secondo dataset.

# Capitolo 1 – Presentazione del problema

Nel seguente capitolo si presentano i dataset analizzati e gli strumenti utilizzati per l'implementazione del classificatore, ossia Fuse-Med-ML e Scikit-learn.

## 1.1 Presentazione dei dataset

Sono state analizzate retrospettivamente 1412 immagini RMN volumetriche post-contrastografiche ottenute a scopo preoperatorio per neuronavigazione (RMN VOL), insieme a 1692 immagini RMN multimodali (RMN MULTI) utilizzate per scopi diagnostici. Le immagini RMN multimodali sono state suddivise in base alla sequenza di acquisizione, comprese le pesature RMN come "ADC", "DWI", "FLAIR", "GRE", "T1W" e "T1W+MDC".



Queste immagini compongono un totale di 282 campioni effettivi. In entrambi i dataset, i campioni sono stati classificati in due categorie, che corrispondono alle due etichette di interesse: GBM e MET.

Tutte le immagini sono in formato JPEG. Sono state escluse dalla successiva analisi i pazienti con artefatti significativi o comorbidità cerebrali. L'analisi preliminare delle immagini e l'inclusione/esclusione dei pazienti sono state effettuate da un neuroradiologo specializzato in neuroimaging oncologico. È presente un lieve sbilanciamento in favore delle immagini etichettate come GBM, ma è importante notare che questo squilibrio nel numero di pazienti tra le due condizioni riflette la diversa distribuzione epidemiologica dei due tipi di tumore.

## 1.2 Fuse-Med-ML e Scikit-Learn

Fuse-Med-ML è un framework *open-source* sviluppato da IBM basato su Pytorch il cui obiettivo è supportare lo sviluppo di applicazione cognitive nell'ambito della ricerca medica. Lo scopo principale di questo framework è incoraggiare il riuso del codice nell'ambito della medicina, la cui assenza nel passato ha spesso costretto gli sviluppatori a dover scrivere il codice senza possedere delle basi di riferimento solide e, dunque, ad allungare di molto i tempi di implementazione.

La caratteristica fondamentale di Fuse-Med-ML è la sua struttura flessibile. I dati sono collezionati in strutture dati gerarchiche che sono accessibili come dei comuni dizionari Python. Questo consente agli sviluppatori di maneggiare agevolmente i dati *multimodali*, ossia i dati ai quali è possibile accedere in più modalità differenti.

```
from fuse.utils import NDict

sample_ndict = NDict()
sample_ndict['input.mri'] = # ...
sample_ndict['input.ct_view_a'] = # ...
sample_ndict['input.ct_view_b'] = # ...
sample_ndict['groundtruth.disease_level_label'] = # ...
```

In questo esempio è stato dichiarato un dizionario al cui interno è salvato un dato in input.

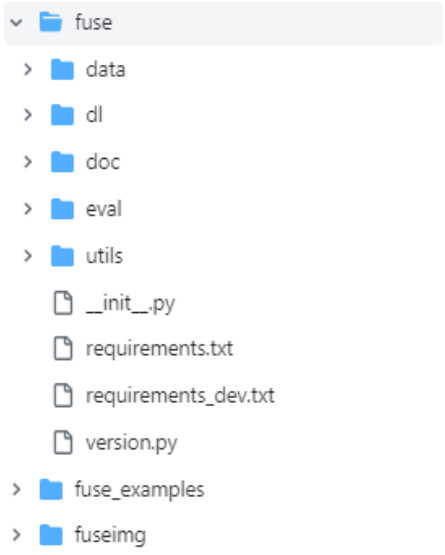
Tale dato ha più caratteristiche (“ct_view_a”, “ct_view_b”, etc.), che sono memorizzate in maniera gerarchica e alle quali è possibile facilmente accedere utilizzando la stessa struttura dati utilizzando una semplice path key (e.g, *input.ct_view_a* oppure, più in generale, *a.b.c.d.etc*).

Un altro vantaggio del framework è il disaccoppiamento dei componenti. Questo permette di tenere ben separati i dati e il modello e inoltre consente all’utente di definire chiavi separate per l’input e per l’output che accedono a strutture dati differenti. Per esempio:

```
MetricAUCROC(pred="model.output.scores", target="data.label")
```

Questo codice calcola la metrica AUCROC a partire dalle predizioni e dal target che sono memorizzati in due strutture gerarchiche diverse *(model.* e data.*).*

Il framework è organizzato in più package. Di particolare interesse sono i package *data* e il package *eval*.



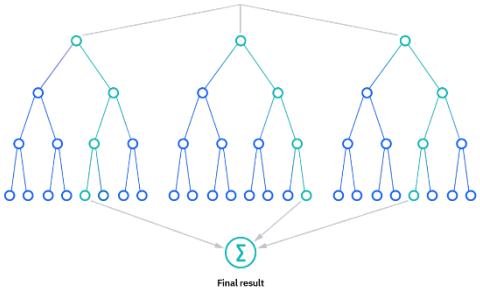

[Fonte: https://github.com/BiomedSciAI/fuse-med-ml/tree/master/fuse]

Il package data contiene al suo interno una serie di definizioni utili al *pre-processing* dei dati. In particolare, esso contiene al suo interno il sotto-package *datasets*, che definisce le implementazioni delle classi utilizzate per il caricamento e la gestione dei dataset, come DatasetDefault e DatasetWrapSeqToDict. Durante lo sviluppo di questo elaborato si è fatto uso in particolare di quest’ultima classe, che consente di convertire un dizionario (formato tipicamente utilizzato per i dataset Pytorch) nella rappresentazione interna di Fuse-Med-ML.

Il package *eval* è invece molto utile perché contiene il sotto-package *metrics*, che definisce le classi relative a tutte le metriche che si utilizzano comunemente per valutare un modello di Machine Learning/Deep Learning. In particolare, nel corso del progetto sono state utilizzate le classi MetricAccuracy (che definisce ovviamente la metrica dell'accuratezza), MetricROC/MetricAUCROC (che definiscono le curve ROC e la metrica AUCROC) e MetricConfusion, che definisce la matrice di confusione e tutte le metriche da essa derivanti, per esempio *Precision, Recall, F1-Score.*

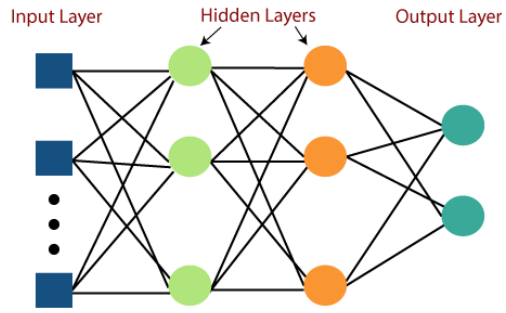
Nella fase finale dell'elaborato, in particolare durante l'implementazione del classificatore relativo al dataset MULTI, l'utilizzo di Fuse-Med-ML è stato integrato con la libreria *scikit-learn* di Python. Si tratta di una libreria open-source molto popolare tra gli sviluppatori di Machine Learning e Deep Learning, in quanto permette di utilizzare modelli molto performanti come il Random Forest e il Multi Layer Perceptron.

Il Random Forest è un algoritmo che combina l'output di più alberi di decisione per ottenere un unico risultato.




(Fonte: https://www.ibm.com/it-it/topics/random-forest)

Il Multi Layer Perceptron è un modello di rete neurale costituito da più strati, ognuno dei quali è composto da più nodi che formano un grafo diretto.

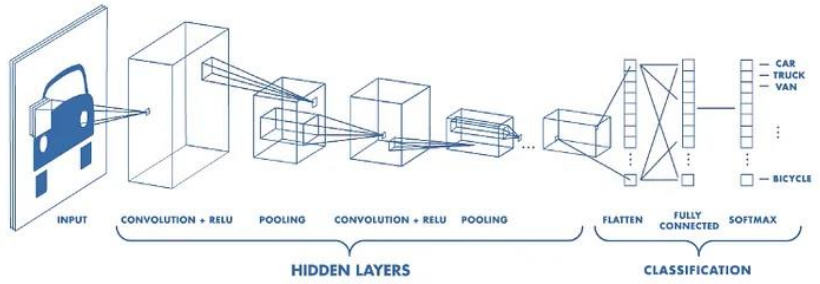



(Fonte: Javatpoint)

Il loro utilizzo sarà descritto più nel dettaglio nel terzo capitolo.

## 1.3 Reti convoluzionali e reti pretrainate

Le reti convoluzionali neurali (CNN) sono utilizzate efficacemente nell'ambito del deep learning per classificare dati che si presentano in forma matriciale, come ad esempio le immagini del nostro dataset. Le CNN sono progettate per riconoscere pattern e strutture complesse nelle immagini, imitando in modo intuitivo il modo in cui il cervello umano elabora visivamente le informazioni. Le CNN sono costituite da diversi strati convoluzionali, strati di pooling e strati completamente connessi.

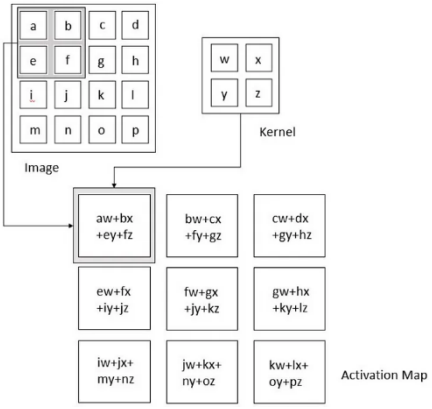



[Fonte: https://miro.medium.com/v2/resize:fit:720/format:webp/1*kkyW7BR5FZJq4_oBTx3OPQ.png]

Il livello di convoluzione è il blocco fondamentale della CNN e produce la maggior parte del carico computazionale della rete.

Questo livello esegue un prodotto scalare tra due matrici, in cui una matrice è l'insieme di parametri apprendibili, noto anche come kernel, e l'altra matrice è una porzione ristretta dell'immagine.

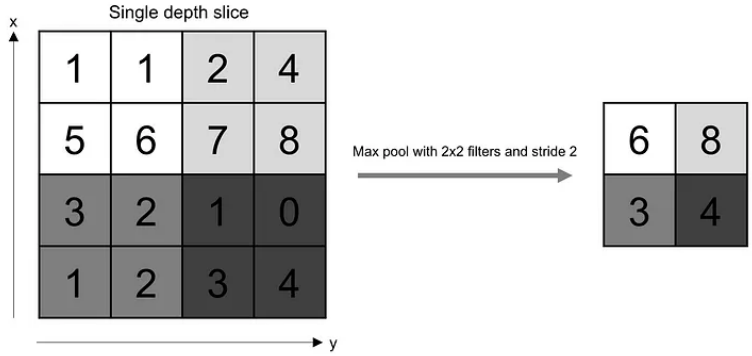
Il kernel scivola lungo l'altezza e la larghezza dell'immagine, producendo in output una nuova rappresentazione bidimensionale dell'immagine, nota come mappa di attivazione o mappa delle caratteristiche.




[Fonte: https://miro.medium.com/v2/resize:fit:720/format:webp/1*r13ZUdVTQwVuhDPmo3JKag.png]

I livelli di pooling nelle reti CNN riducono progressivamente la dimensione delle mappe delle caratteristiche generando una versione ridotta delle stesse. Ciò è fatto per ridurre la complessità computazionale e il numero di parametri nella rete, nonché per mantenere le caratteristiche più salienti dell'immagine.

Il pooling opera su una finestra (ad esempio, una finestra di 2x2) e ne calcola una statistica di riepilogo, come il valore massimo (max pooling) o la media (average pooling), all'interno di quella finestra. Successivamente, la finestra viene spostata sulla mappa delle caratteristiche con uno stride definito e viene applicato nuovamente il calcolo della statistica di riepilogo.




[Fonte: https://miro.medium.com/v2/resize:fit:720/format:webp/1*sK7oP1m129V_oNGSsHIm_w.png]

Oltre a ridurre la dimensione spaziale delle mappe delle caratteristiche, il pooling aiuta a ridurre l'overfitting e ad aumentare la robustezza del modello, poiché riduce il rischio di memorizzare caratteristiche ridondanti o rumore.

Infine, i livelli completamente connessi aggregano le informazioni estratte dalle mappe delle caratteristiche e le utilizzano per effettuare la classificazione finale.

Per ottenere la classificazione migliore possibile, uno degli strumenti fondamentali è stato l’utilizzo di reti pretrainate.

Le reti pretrainate rappresentano una risorsa preziosa nel campo del deep learning e delle reti neurali convoluzionali. Questi modelli sono stati addestrati su grandi dataset contenenti milioni di immagini, consentendo loro di apprendere rappresentazioni di alto livello delle caratteristiche visive. Ciò significa che le reti pretrainate sono in grado di riconoscere

automaticamente bordi, forme, texture e altre caratteristiche rilevanti presenti nelle immagini. Utilizzare reti pretrainate presenta diversi vantaggi:

- **Risparmio tempo ed energia evitando la necessità di addestrare un modello da zero.** Le reti pretrainate rappresentano un punto di partenza solido e offrono una base solida per l'apprendimento delle nuove immagini o compiti specifici.
- **Sono disponibili in diverse varianti, addestrate su dataset di dimensioni e complessità diverse.** Ciò consente di selezionare la rete che si adatta meglio alle caratteristiche delle immagini del proprio problema specifico.
- **Possono essere adattate attraverso il trasferimento di apprendimento**, che permette di personalizzarle e ottimizzarle per un determinato compito, aggiungendo uno o più strati e raffinandone i pesi.
- **Consentono di affrontare problemi di disponibilità di dati.** Se si dispone solo di un numero limitato di dati di addestramento, le reti pretrainate possono essere utilizzate come punto di partenza per migliorare le prestazioni del modello su un dataset più piccolo, sfruttando le loro conoscenze apprese su un ampio insieme di dati.

# Capitolo 2 – VOL

Nel seguente capitolo è presentata l'implementazione del classificatore addestrato con il dataset VOL. Gli strumenti implementativi utilizzati sono quelli definiti nel primo capitolo. L'ambiente di sviluppo utilizzato è Google Colab, a cui è stato collegato un account Google Drive in cui sono state caricate le immagini e in cui sono stati salvati i risultati dei modelli.

## 2.1 Esplorazione e preprocessing del dataset

Il dataset VOL è organizzato nelle tre seguenti cartelle: **trainingMRI_VOL, validationMRI_VOL, testingMRI_VOL**, che contengono rispettivamente i dataset di training, testing e validazione del modello. Ognuna delle tre cartelle contiene le immagini MRI etichettate nel modo seguente:

*VOL_n_classe_fase.formato*, dove:

- **n** è un numero progressivo che identifica univocamente l'immagine all'interno della sequenza, espresso con 3 cifre decimali.
- **classe** è l'etichetta dell'immagine (GBM oppure MET).
- **fase** è la fase di utilizzo dell'immagine (training, validazione, testing).
- **formato** è il formato dell'immagine, ossia JPEG.

Esempio: ***VOL_001_GBM_training.jpg*** indica la prima immagine del dataset di training, etichettata come GBM.

A questa struttura sono stata apportate le seguenti modifiche:

1) Si sono rinominate le tre cartelle in **training_VOL**, **validation_VOL** e **testing_VOL**, in quanto l'attributo MRI è stato considerato ridondante.
2) Per ognuna delle tre cartelle, si sono create le due sottocartelle **GBM** e **MET** e si sono quindi raggruppate le immagini sulla base delle loro etichette. Per cui, il path

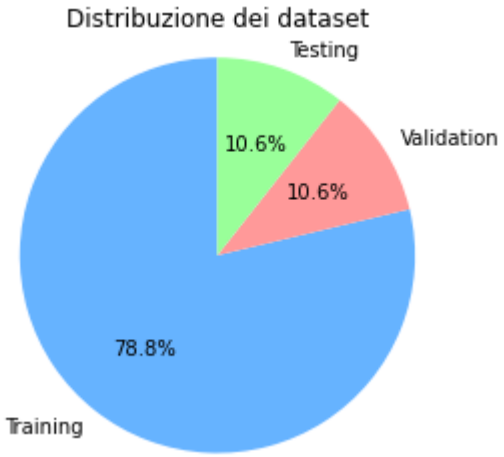
*training_VOL/GBM* contiene tutte le immagini etichettata come GBM e un discorso analogo vale per il path *training_VOL/MET*.

3) Le tre cartelle sono state inserite in una cartella su Google Drive denominata **DATASET**.

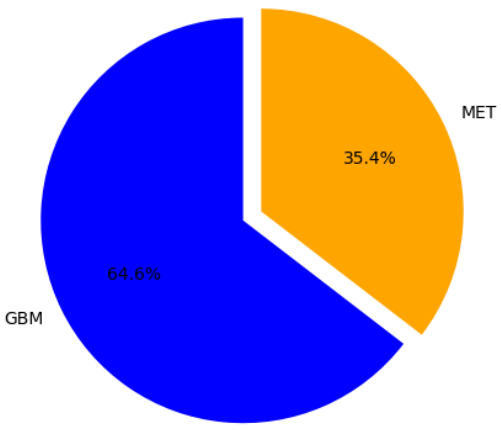
A questo punto è possibile importare i tre dataset in Fuse-Med-ML. Prima di tutto, si setta la directory *data_dir* con il percorso contenete i dati:

```
data_dir = '/content/drive/MyDrive/PROGETTO_CCS/DATASET'
```

Successivamente, si importano i tre dataset training_VOL, validation_VOL e testing_VOL utilizzando la funzione *datasets.ImageFolder()* messa a disposizione da *Pytorch* e indicizzandoli in un dizionario. Il dataset di training contiene circa l'80% delle immagini, quello di validation un ulteriore 10% e quello di testing il restante 10% circa.




Innanzitutto, analizzando il numero di istanze per ogni classe nel dataset di training, si può notare come esso sia sbilanciato in favore della classe GBM. È importante notare che questo sbilanciamento riflette la diversa distribuzione epidemiologica dei due tipi di tumore.



Successivamente sono state applicate delle trasformazioni alle immagini, in modo da poter addestrare i modelli in maniera più efficace. Innanzitutto, è stato necessario applicare un **ridimensionamento** delle immagini per due motivi:

1) non tutte le immagini all'interno dei dataset hanno la stessa dimensione;
2) i modelli pretrainati richiedono immagini di dimensioni prefissate per poter lavorare correttamente. Nel caso della maggior parte dei modelli utilizzati, la dimensioni richiesta era *224 x 224 pixel*.

Dopo il ridimensionamento, le immagini sono state trasformate in **tensori**: si tratta di un passaggio obbligato al fine di dare i dati in input ai modelli.

Successivamente, si può opzionalmente eseguire la **normalizzazione** dell'immagine, utilizzando dei valori specifici oppure dei valori consigliati dagli sviluppatori dei modelli pretrainati. In questo caso, si sono testati due tipi diversi di normalizzazione: una è quella consigliata dal MNIST, mentre l'altra è quella consigliata per Resnet/VGG. Si sono ottenuti risultati migliori utilizzando la prima normalizzazione. Il codice delle trasformazioni è il seguente:

```
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
    #transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
```

A questo punto, ognuno dei tre dataset va convertito nella rappresentazione dei dataset interna al framework Fuse-Med-ML grazie alla classe DatasetWrapSeqToDict. Il costruttore di questa classe prende in input il nome del dataset, il dataset e una lista di sample keys, che sono le chiavi per accedere ai campioni: in questo caso 'data.image' permette di accedere all'immagine, mentre 'data.label' permette di accedere all'etichetta dell'immagine (GBM oppure MET). Per esempio, si mostra il codice per convertire il dataset di training nella rappresentazione di Fuse-Med-ML:

```
torch_train_dataset = {x: datasets.ImageFolder(os.path.join(data_dir,
x),transform) for x in ['training_VOL', 'testing_VOL', 'validation_VOL']}
```

```
train_dataset = DatasetWrapSeqToDict(name='training_VOL',
dataset=torch_train_dataset['training_VOL'], sample_keys=('data.image',
'data.label'))
train_dataset.create()
```

Il framework Fuse-Med-ML mette a disposizione la classe BatchSamplerDefault per effettuare il bilanciamento dei batch. Questa classe permette di definire il numero delle classi da bilanciare (in questo caso, due) e i pesi con cui effettuare il bilanciamento. Il sampler si occuperà di campionare casualmente le immagini in modo che in ogni batch le due classi siano bilanciate come richiesto.

```
sampler = BatchSamplerDefault(
    dataset=train_dataset,
    balanced_class_name="data.label",
    num_balanced_classes=2,
    batch_size=train_params["data.batch_size"],
    balanced_class_weights=None,
)
```

A questo punto è possibile caricare in memoria i dati utilizzando la classe DataLoader che prende in input, tra gli altri parametri, il dataset e il sampler.

```
train_dataloader = DataLoader(
    dataset=train_dataset,
    batch_sampler=sampler,
    collate_fn=CollateDefault(),
    num_workers=train_params["data.train_num_workers"],
)
```

Ragionamenti analoghi si possono applicare al dataset di validazione e, in una fase successiva, a quello di testing.

## 2.2 Definizione dei modelli

Per quanto riguarda i modelli, sono stati utilizzate alcune delle reti CNN che performano al meglio sul dataset ImageNet.

È stato realizzato un confronto tra nove differenti modelli pretrainati: ResNet18, ResNet50, ResNet101, ResNet152, VGG16, VGG19, EfficientNet_B0, EfficientNet_B7, EfficientNet_V2_L. Tutti questi modelli appartengono a **tre grandi famiglie** di CNN:

- **ResNet.** La caratteristica principale di ResNet è l'uso di blocchi residui, che consentono di superare il problema della scomparsa del gradiente, che si verifica quando la retropropagazione del gradiente attraverso molti strati causa una graduale

diminuzione del suo valore. I blocchi residui introducono delle "skip connections" che permettono di saltare uno o più strati. Ciò facilita l'ottimizzazione e l'addestramento di reti particolarmente profonde.

- **EfficientNet**. La caratteristica principale di EfficientNet risiede nella scalabilità composta (compound scaling). Invece di aumentare casualmente la profondità della rete, la larghezza dei canali e la risoluzione degli input, la scalabilità composta modifica uniformemente ogni dimensione con un certo insieme fisso di coefficienti di scalatura. L'obiettivo è quello di trovare un equilibrio ottimale tra la complessità del modello e la sua capacità di apprendimento.
- **VGG**. La caratteristica principale di VGG è la sua semplicità e omogeneità. Essa è caratterizzata da una serie di strati convoluzionali seguiti da strati di pooling e strati fully connected alla fine. La particolarità è che tutte le convoluzioni sono di dimensione 3x3 con stride 1 e padding 1, e i max-pooling sono di dimensione 2x2 con stride 2. Questa struttura regolare consente a VGG di apprendere a partire da immagini molto complesse.

Il codice per la definizione del modello è il seguente. Ovviamente bisogna modificarlo opportunamente a seconda della rete che si intende definire.

```
def create_model():
    torch_model = models.resnet152(pretrained=True) # Definisci modello
    model = ModelWrapSeqToDict(
        model=torch_model,
        model_inputs=["data.image"],
        post_forward_processing_function=perform_softmax,
        model_outputs=["model.logits.classification",
"model.output.classification"],
    )
    return model

model = create_model()
```

Pytorch mette a disposizione la possibilità di stampare l'architettura delle reti e un riassunto di tale struttura (numero di parametri addestrabili, numero di parametri non addestrabili e Mult-Adds, ossia il numero totale di moltiplicazioni e addizioni eseguite dalla rete).

```
from torchsummaryX import summary
summary(model, torch.zeros((64, 3, 224, 224)))
```

A scopo illustrativo, mostriamo l'architettura di alcune CNN utilizzate, ossia quelle che sono risultate le migliori in termini di Accuracy.

**Resnet152** (parte dell'architettura è omessa per brevità):

| Layer | Kernel Shape | Output Shape | Params | Mult-Adds |
|---|---|---|---|---|
| 0_features.0.Conv2d_0 | [3, 32, 3, 3] | [64, 32, 112, 112] | 864.0 | 10838016.0 |
| 1_features.0.BatchNorm2d_1 | [32] | [64, 32, 112, 112] | 64.0 | 32.0 |
| 2_features.0.SiLU_2 | - | [64, 32, 112, 112] | NaN | NaN |
| 3_features.1.0.block.0.Conv2d_0 | [32, 32, 3, 3] | [64, 32, 112, 112] | 9216.0 | 115605504.0 |
| 4_features.1.0.block.0.BatchNorm2d_1 | [32] | [64, 32, 112, 112] | 64.0 | 32.0 |
| ... | ... | ... | ... | ... |
| 952_features.8.BatchNorm2d_1 | [1280] | [64, 1280, 7, 7] | 2560.0 | 1280.0 |
| 953_features.8.SiLU_2 | - | [64, 1280, 7, 7] | NaN | NaN |
| 954_avgpool | - | [64, 1280, 1, 1] | NaN | NaN |
| 955_classifier.Dropout_0 | - | [64, 1280] | NaN | NaN |
| 956_classifier.Linear_1 | [1280, 1000] | [64, 1000] | 1281000.0 | 1280000.0 |

| | Totals |
|---|---|
| Total params | 60.192808M |
| Trainable params | 60.192808M |
| Non-trainable params | 0.0 |
| Mult-Adds | 11.513702336G |

**EfficientNet_V2_L** (parte dell'architettura è omessa per brevità):

| Layer | Kernel Shape | Output Shape | Params | Mult-Adds |
|---|---|---|---|---|
| 0_features.0.Conv2d_0 | [3, 32, 3, 3] | [64, 32, 112, 112] | 864.0 | 10838016.0 |
| 1_features.0.BatchNorm2d_1 | [32] | [64, 32, 112, 112] | 64.0 | 32.0 |
| 2_features.0.SiLU_2 | - | [64, 32, 112, 112] | NaN | NaN |
| 3_features.1.0.block.0.Conv2d_0 | [32, 32, 3, 3] | [64, 32, 112, 112] | 9216.0 | 115605504.0 |
| 4_features.1.0.block.0.BatchNorm2d_1 | [32] | [64, 32, 112, 112] | 64.0 | 32.0 |
| ... | ... | ... | ... | ... |
| 952_features.8.BatchNorm2d_1 | [1280] | [64, 1280, 7, 7] | 2560.0 | 1280.0 |
| 953_features.8.SiLU_2 | - | [64, 1280, 7, 7] | NaN | NaN |
| 954_avgpool | - | [64, 1280, 1, 1] | NaN | NaN |
| 955_classifier.Dropout_0 | - | [64, 1280] | NaN | NaN |
| 956_classifier.Linear_1 | [1280, 1000] | [64, 1000] | 1281000.0 | 1280000.0 |

| | Totals |
|---|---|
| Total params | 118.515272M |
| Trainable params | 118.515272M |
| Non-trainable params | 0.0 |
| Mult-Adds | 12.231444832G |

**Resnet18** (parte dell'architettura è omessa per brevità)**:**

| Layer | Kernel Shape | Output Shape | Params | Mult-Adds |
|---|---|---|---|---|
| 0_conv1 | [3, 64, 7, 7] | [64, 64, 112, 112] | 9408.0 | 118013952.0 |
| 1_bn1 | [64] | [64, 64, 112, 112] | 128.0 | 64.0 |
| 2_relu | - | [64, 64, 112, 112] | NaN | NaN |
| 3_maxpool | - | [64, 64, 56, 56] | NaN | NaN |
| 4_layer1.0.Conv2d_conv1 | [64, 64, 3, 3] | [64, 64, 56, 56] | 36864.0 | 115605504.0 |
| ... | | | | |
| 55_layer4.1.Conv2d_conv2 | [512, 512, 3, 3] | [64, 512, 7, 7] | 2359296.0 | 115605504.0 |
| 56_layer4.1.BatchNorm2d_bn2 | [512] | [64, 512, 7, 7] | 1024.0 | 512.0 |
| 57_layer4.1.ReLU_relu | - | [64, 512, 7, 7] | NaN | NaN |
| 58_avgpool | - | [64, 512, 1, 1] | NaN | NaN |
| 59_fc | [512, 1000] | [64, 1000] | 513000.0 | 512000.0 |

| | Totals |
|---|---|
| Total params | 11.689512M |
| Trainable params | 11.689512M |
| Non-trainable params | 0.0 |
| Mult-Adds | 1.814078144G |

## 2.3 Definizione delle metriche

L'ultimo passaggio da eseguire prima di procedere con il training dei modelli e il tuning degli iperparametri è la definizione delle metriche da utilizzare per la valutazione del modello nelle fasi di training, validazione e testing. In particolare, si sono scelte le seguenti metriche per le fasi di training e validazione:

- **Accuracy.** Rappresenta la percentuale di previsioni corrette del modello. Formalmente è il rapporto tra il numero di previsioni corrette e il numero di previsioni totali.

  Nei classificatori binari, supponendo di aver definito due classi (classe Positiva e classe Negativa) vale la seguente espressione:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

  dove TP sono i True Positives (veri positivi), TN sono i True Negatives (veri negativi), FP sono i False Positives (falsi positivi) e FN sono i False Negative (falsi

negativi). Questa metrica è stata scelta come parametro per stabilire quale fosse il miglior modello in quanto descrive in maniera efficace le performance della rete con un unico valore numerico intuitivo.

- **Precision**. La precisione rappresenta la percentuale di istanze classificate correttamente come positive (TP) rispetto a tutte le istanze classificate come positive dal modello (TP + FP). Permette quindi di avere un'informazione su qual è la percentuale di elementi classificati come positive ad essere stata effettivamente classificata correttamente. Vale la seguente espressione:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (sensitivity)**. La recall misura la percentuale di istanze positive correttamente identificate dal modello rispetto a tutte le istanze effettivamente positive.
  È anche detto "tasso dei veri positivi" in quando indica la capacità del modello di

$$\text{Recall} = \frac{TP}{TP + FN}$$

  individuare correttamente tutti i veri positivi rispetto a tutti i casi positivi reali. Ossia, individua qual è la percentuale dei valori effettivamente positivi che sono stati individuati correttamente.

- **F1-Score.** È una metrica che si utilizza spesso in quanto rappresenta una sintesi tra le metriche di precision e recall. Si ottiene calcolando la media armonica tra precision e recall.

$$F1\ score = 2 \cdot \frac{Precision * Recall}{Precision + Recall}$$

  Si tratta quindi di un valore compreso tra 0 e 1 che è vicino a 1 quando sia Precision sia Recall sono alti, mentre è vicino a 0 quando sia Precision sia Recall sono bassi.
  Il codice per la definizione delle metriche è il seguente:

```
train_metrics = OrderedDict(
    [
        ("operation_point",
MetricApplyThresholds(pred="model.output.classification")),
```

```
        ("accuracy",
MetricAccuracy(pred="results:metrics.operation_point.cls_pred",
target="data.label")),
        ("confusion", MetricConfusion(pred="model.output.classification",
target="data.label"))
    ]
)
validation_metrics = copy.deepcopy(train_metrics)
```

Per quanto riguarda le metriche utilizzate in fase di valutazione del modello, si utilizzano quelle precedenti con l'aggiunta di:

- **Curva ROC.** La curva ROC un grafico che mostra le prestazioni di un modello di classificazione al variare delle soglie di classificazione. Sull'asse delle ascisse c'è il TPR (True Positive Rate, che coincide con la Recall). Sull'asse delle ordinate c'è il TNR (True Negative Rate). Una curva ROC traccia TPR rispetto a FPR a soglie di classificazione diverse. Abbassando la soglia di classificazione, vengono classificati più elementi positivi, aumentando così i falsi positivi e i veri positivi. In sostanza, la curva ROC dà informazioni sul trade-off tra la capacità del modello di identificare correttamente i veri positivi (alta TPR) e la sua tendenza a generare falsi positivi (alta FPR). Nel punto (0,0) la Recall è 0, dunque nessun positivo è classificato correttamente. Al contrario, nel punto (1,1) la Recall è 1, quindi tutti i positivi sono classificati correttamente a discapito dell'aumento del tasso dei falsi positivi FPR.
- **AUC.** E' l'area sottesa alla curva ROC. Rappresenta una misura quantitativa delle prestazioni del modello e riassume le informazioni della curva ROC in un unico valore. Tale valore varia tra 0 e 1, dove 1 rappresenta il classificatore perfetto (ossia, non sbaglia mai predizione) mentre 0 rappresenta un classificatore che sbaglia tutte le predizioni. Un valore di 0.5 indica invece una classificazione puramente casuale.

```
metrics = OrderedDict(
    [
        ("operation_point",
MetricApplyThresholds(pred="model.output.classification")),
        ("accuracy",
MetricAccuracy(pred="results:metrics.operation_point.cls_pred",
target="data.label")),
        (
            "roc",
            MetricROCCurve(
                pred="model.output.classification",
```

```
                target="data.label",
                class_names=class_names,
                output_filename=os.path.join(paths["inference_dir"],
"roc_curve.png"),
            ),
        ),
        ("auc", MetricAUCROC(pred="model.output.classification",
target="data.label", class_names=class_names)),
        ("confusion", MetricConfusion(pred="model.output.classification",
target="data.label",metrics=("precision", "sensitivity", "f1")))
    ]
)
```

## 2.4 Tuning degli ipeparametri

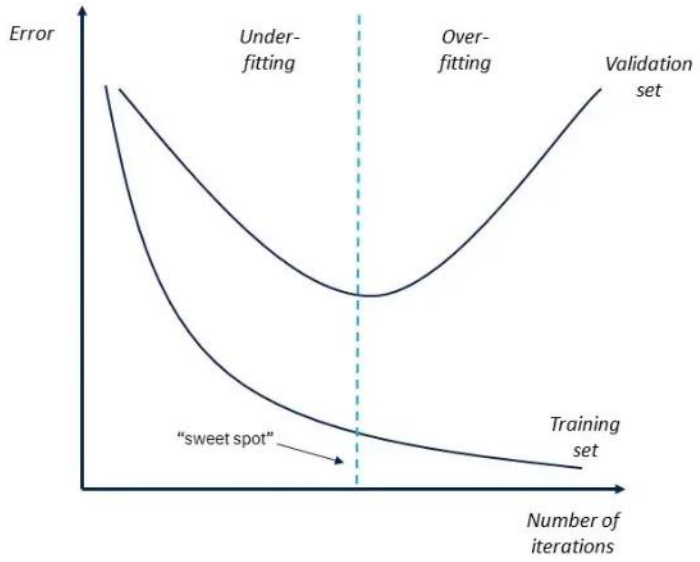
Dopo aver definito correttamente le metriche, si possono addestrare i vari modelli e valutarli sulla base dei valori ottenuti. Come già accennato nel capitolo precedente, sono state utilizzate delle reti neurali convoluzionali pretrainate e per ogni modello è necessario effettuare il tuning di numerosi iperparametri. Per farlo è stato utilizzato un validation set composto da circa 150 campioni. Sperimentalmente questi sono risultati i migliori iperparametri:

| **PARAMETRO** | **VALORE** |
|---|---|
| Batch Size | 64 |
| Epochs | 50 |
| Learning Rate | $10^{-4}$ |
| Weight Decay | $10^{-3}$ |
| Optimizer | Adam |
| Loss Function | Cross Entropy |

- **Batch Size**: Il batch size rappresenta il numero di campioni di addestramento che vengono utilizzati per calcolare il gradiente e aggiornare i pesi del modello in una singola iterazione. Un batch più grande può accelerare il processo di addestramento sfruttando la parallelizzazione dei calcoli, ma richiede più memoria. D'altro canto, un batch più piccolo può fornire una migliore generalizzazione e una maggiore stabilità

durante l'addestramento. È stato scelto un numero di campioni pari a 64 per ogni batch, in questo modo il tempo di addestramento non risulta troppo elevato, ma allo stesso tempo riusciamo ad ottenere un'ottima generalizzazione.

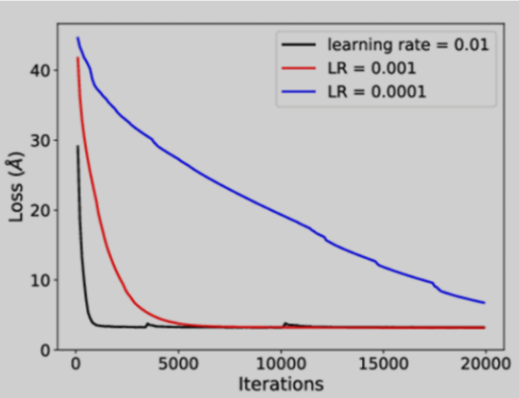
- **Numero di Epoche**: Il numero di epoche indica quante volte l'intero dataset di training viene presentato alla rete durante l'addestramento. Ad ogni epoca, il modello esegue un'iterazione completa su tutti i dati di addestramento. Un numero maggiore di epoche permette al modello di apprendere più a fondo le caratteristiche dei dati, ma può portare anche a un rischio di overfitting se il modello memorizza troppo i dati di addestramento.




[Fonte:https://databasecamp.de/wp-content/uploads/image-9.png]

Il numero di epoche scelte nel nostro caso è pari a 50, si tratta di un numero scelto sperimentalmente in quanto alcuni modelli necessitano all'incirca di 30-40 epoche per raggiungere le migliori performance. In ogni caso, per non incorrere nell'overfitting, è calcolata l'accuracy sul set di validazione in modo tale da salvare il nuovo modello prodotto a fine epoca solamente se risulta migliore dei precedenti.

- **Learning Rate**: Il learning rate (tasso di apprendimento) determina la dimensione dei passi che vengono fatti durante l'aggiornamento dei pesi del modello durante l'addestramento. Un learning rate più grande può accelerare l'addestramento, ma può anche portare a salti troppo grandi nell'ottimizzazione e a una convergenza inefficace. Al contrario, un learning rate troppo piccolo può rallentare l'addestramento o far rimanere il modello bloccato in minimi locali.




[https://www.researchgate.net/profile/Eric-Wilson-18/publication/343310960/figure/fig5/AS:918988120936449@1596115250187/Effect-of-learning-rate-on-LSTM-training-on-SMD-trajectories.png]

Il learning rate è uno degli iperparametri più importanti e attraverso sperimentazione è stato scelto un valore pari a 0.0001. In questo modo, il modello riesce a convergere efficacemente entro le 50 epoche.

- **Decadimento dei Pesi**: Il decadimento dei pesi (weight decay), noto anche come regolarizzazione L2, è una tecnica che introduce un termine additivo alla funzione di perdita durante l'addestramento che penalizza i pesi più grandi. Questo aiuta a limitare la complessità del modello, favorendo pesi più piccoli e contribuendo a evitare l'overfitting. Sperimentalmente si è scelto di adottare un valore pari a 0.001.
- **Optimizer**: L'optimizer è l'algoritmo che durante ogni epoca modifica i pesi della nostra rete in modo tale da minimizzare (o massimizzare) la funzione di loss scelta. Gli optimizer più utilizzati sono: Adam, Stochastic gradient descent (SGD), Batch Gradient Descent (BGD). Sperimentalmente Adam è risultato il più performante.
- **Funzione di loss**: La funzione di loss compara i valori di target e le predizioni misurando quindi le prestazioni della rete. Per problemi di classificazione la binary cross-entropy è la funzione in assoluto più utilizzata.

## 2.5 Valutazione e analisi comparata dei modelli

La fase finale è la valutazione dei modelli sulla base delle metriche definite e l'analisi comparata per stabilire quale modello è il migliore sulla base dei risultati ottenuti sul dataset di test. Per prima cosa si calcolano le predizioni sul test set.

```
# setting dir and paths
create_dir(paths["inference_dir"])
infer_file = os.path.join(paths["inference_dir"],
infer_common_params["infer_filename"])
checkpoint_file = os.path.join(paths["model_dir"],
infer_common_params["checkpoint"])

# creating a dataloader
testing_dataset = DatasetWrapSeqToDict(name='testing_VOL',
dataset=torch_train_dataset['testing_VOL'], sample_keys=('data.image',
'data.label'))
testing_dataset.create()
testing_dataloader = DataLoader(dataset=testing_dataset,
collate_fn=CollateDefault(), batch_size=2, num_workers=2) #
testing_dataloader

# load pytorch lightning module
model = create_model()
pl_module = LightningModuleDefault.load_from_checkpoint(
    checkpoint_file, model_dir=paths["model_dir"], model=model,
map_location="cpu", strict=True
)

# set the prediction keys to extract (the ones used be the evaluation
function).
pl_module.set_predictions_keys(
    ["model.output.classification", "data.label"]
)  # which keys to extract and dump into file

# create a trainer instance
pl_trainer = pl.Trainer(
    default_root_dir=paths["model_dir"],
    accelerator=infer_common_params["trainer.accelerator"],
    devices=infer_common_params["trainer.num_devices"],
)

# predict
predictions = pl_trainer.predict(pl_module, testing_dataloader,
return_predictions=True)

# convert list of batch outputs into a dataframe
```

```
infer_df = convert_predictions_to_dataframe(predictions)
save_dataframe(infer_df, infer_file)
```

Successivamente vengono determinati i valori di tutte le metriche a partire dalle predizioni e dalle label.

```
# create evaluator
evaluator = EvaluatorDefault()

# run eval
results = evaluator.eval(
    ids=None,
    data=os.path.join(paths["inference_dir"],
eval_common_params["infer_filename"]),
    metrics=metrics,
    output_dir=paths["eval_dir"],
    silent=False,
)
```

**Il modello ResNet152 è risultato il migliore in assoluto.** Di seguito si riporta il riassunto delle epoche di addestramento del modello sui dataset di training e di validazione:

```
Stats for epoch: 0 (Currently the best epoch for source validation.metrics.accuracy!)
-----------------------------------------------------------------------------------------
|                            | Best Epoch (0)             | Current Epoch (0)          |
-----------------------------------------------------------------------------------------
| train.losses.cls_loss      | 3.2696                     | 3.2696                     |
-----------------------------------------------------------------------------------------
| train.losses.total_loss    | 3.2696                     | 3.2696                     |
-----------------------------------------------------------------------------------------
| train.metrics.accuracy     | 0.5805                     | 0.5805                     |
-----------------------------------------------------------------------------------------
| validation.losses.cls_loss | 0.8292                     | 0.8292                     |
-----------------------------------------------------------------------------------------
| validation.losses.total_loss| 0.8292                    | 0.8292                     |
-----------------------------------------------------------------------------------------
| validation.metrics.accuracy | 0.8800                    | 0.8800                     |
-----------------------------------------------------------------------------------------
Stats for epoch: 1 (Best epoch is 0 for source validation.metrics.accuracy)
-----------------------------------------------------------------------------------------
|                            | Best Epoch (0)             | Current Epoch (1)          |
-----------------------------------------------------------------------------------------
| train.losses.cls_loss      | 3.2696                     | 0.2475                     |
-----------------------------------------------------------------------------------------
| train.losses.total_loss    | 3.2696                     | 0.2475                     |
-----------------------------------------------------------------------------------------
| train.metrics.accuracy     | 0.5805                     | 0.9313                     |
-----------------------------------------------------------------------------------------
| validation.losses.cls_loss | 0.8292                     | 0.5606                     |
-----------------------------------------------------------------------------------------
| validation.losses.total_loss| 0.8292                    | 0.5606                     |
-----------------------------------------------------------------------------------------
| validation.metrics.accuracy | 0.8800                    | 0.8467                     |
-----------------------------------------------------------------------------------------
```

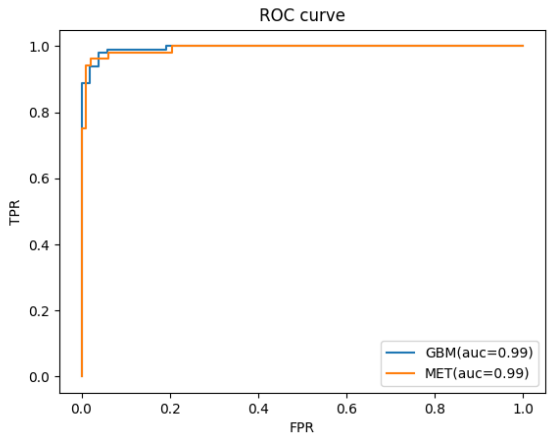
…

```
Stats for epoch: 49 (Best epoch is 8 for source validation.metrics.accuracy)
-------------------------------------------------------------------------------------------
|                              | Best Epoch (8)              | Current Epoch (49)          |
-------------------------------------------------------------------------------------------
| train.losses.cls_loss        | 0.0089                      | 0.0045                      |
-------------------------------------------------------------------------------------------
| train.losses.total_loss      | 0.0089                      | 0.0045                      |
-------------------------------------------------------------------------------------------
| train.metrics.accuracy       | 0.9961                      | 0.9984                      |
-------------------------------------------------------------------------------------------
| validation.losses.cls_loss   | 0.2635                      | 0.3241                      |
-------------------------------------------------------------------------------------------
| validation.losses.total_loss| 0.2635                      | 0.3241                      |
-------------------------------------------------------------------------------------------
| validation.metrics.accuracy  | 0.9467                      | 0.9467                      |
-------------------------------------------------------------------------------------------
```

Nel file *metrics.txt* sono invece salvati i valori calcolati sul **dataset di test** durante la fase di **evaluation**.

```
Metric accuracy:

------------------------------------------

0.9733333333333334

Metric confusion.precision:

------------------------------------------

0.9615384615384616

Metric confusion.sensitivity:

------------------------------------------

0.9615384615384616

Metric confusion.f1:

------------------------------------------

0.9615384615384616
```

Le curve ROC si trovano invece nella cartella *infer_dir:*



Per tutti gli altri modelli si riportano di seguito i valori delle metriche calcolate per il test set nella fase di evaluation:

| MODELLO | ACCURACY | PRECISION | RECALL | F1 |
|---|---|---|---|---|
| ResNet18 | 96.00% | 97.91% | 90.38% | 94.00% |
| ResNet50 | 95.33% | 90.90% | 96.15% | 93.45% |
| ResNet101 | 94.67% | 97.82% | 86.53% | 91.83% |
| ResNet152 | 97.33% | 96.15% | 96.15% | 96.15% |
| VGG16 | 94.67% | 92.30% | 92.30% | 92.30% |
| VGG19 | 94.67% | 94.00% | 90.38% | 92.15% |
| EfficientNet_B0 | 92.00% | 95.71% | 92.30% | 88.89% |
| EfficientNet_B7 | 94.00% | 89.09% | 94.23% | 91.58% |
| EfficientNet_V2_L | 96.67% | 97.96% | 92.30% | 95.05% |

# Capitolo 3 - MULTI

Nel seguente capitolo è presentata l'implementazione del classificatore addestrato con il dataset MULTI.

## 3.1 Esplorazione e preprocessing del dataset

Il dataset MULTI è organizzato nelle tre seguenti cartelle: **trainingMRI_MULTI, validationMRI_MULTI, testingMRI_MULTI**, che contengono rispettivamente i dataset di training, testing e validazione del modello. Rispetto ai dataset precedenti, la particolarità è che in questo caso ogni campione è costituito da 6 immagini differenti che appartengono alle 6 categorie ADC, DWI, FLAIR, GRE, MDC, T1W. Di conseguenza 1612 immagini contenute nei tre dataset corrispondono di fatto a 282 campioni effettivi. Ognuna delle tre cartelle contiene delle ulteriori sottocartelle, le quali a loro volta contengono un singolo campione costituito dalle 6 immagini. I nomi delle sottocartelle seguono il pattern:

*MRI_MULTI_CLASSE_fase_n*, dove:

- **classe** è l'etichetta dell'immagine (GBM oppure MET).
- **fase** è la fase in cui utilizzare il dataset (training, testing, validation).
- **n** è un numero progressivo che identifica univocamente l'immagine all'interno della sequenza, espresso con 3 cifre decimali.

Esempio: ***MRI_MULTI_GBM_training_001*** indica il primo campione del dataset di training, etichettata come GBM e contente 6 ulteriori immagini.

I nomi delle 6 immagini contenute nelle sottocartelle seguono invece il pattern:

*categoria_classen.formato*, dove:

- **categoria** è il nome della categoria a cui appartiene l'immagine.
- **classe** è l'etichetta dell'immagine (GBM oppure MET).

- **n** è un numero progressivo che identifica univocamente l'immagine all'interno della sequenza, espresso con 3 cifre decimali.
- **formato** è il formato dell'immagine, ossia JPEG.

Esempio: ***ADC_GBM0218.jpg*** indica l'immagine numero 218, etichettata come GBM e appartenente alla categoria ADC.

Alla struttura del dataset sono state apportate le seguenti modifiche:

- Per ognuno dei tre dataset (testing, training, validation) sono state create **sei** cartelle, ognuna associata a una singola categoria e contenente immagini appartenenti esclusivamente a tale categoria. Per esempio, la cartella **training_MULTI_ADC** contiene tutte le immagini del dataset di training associate alla categoria ADC, mentre la cartella **testing_MULTI_GRE** contiene tutte le immagini del dataset di testing associate alla categoria GRE, e così via. In totale si sono ottenute quindi 18 cartelle.
- Per ognuna delle 18 cartelle risultanti, si sono create due sottocartelle associate alle etichette dei dati GBM e MET e si sono quindi suddivise le immagini sulla base della loro label. Per esempio, il path *training_MULTI_ADC/GBM* contiene tutte le immagini del dataset di training associate alla categoria ADC ed etichettate come GBM, e così via.
- Le tre cartelle principali sono state inserite in una cartella su Google Drive denominata **DATASET_MULTI.**

L'ottenimento della nuova struttura è stato possibile grazie ad uno script Python che ha automatizzato il processo di creazione delle nuove cartelle. Al fine di ottenere la struttura definitiva a partire dal dataset originale, è opportuno seguire i passaggi:

1) Creare all'interno delle cartelle training_MULTI, testing_MULTI, validation_MULTI le due cartelle GBM e MET.
2) Spostare le cartelle delle immagini etichettate come GBM nella cartella GBM e le cartelle delle immagini etichettate come MET nella cartella MET.
3) Eseguire lo script Python che effettua la suddivisione in cartelle e corregge alcuni errori di formattazione presenti:

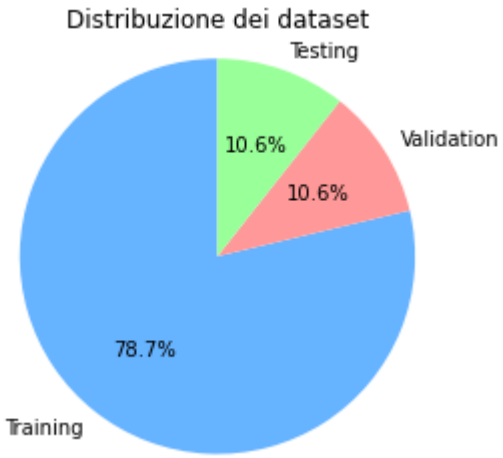
```
import os
import shutil
from pathlib import Path
```

```
# Percorso della cartella principale
root_dir = "DATASET_MULTI"

# Elenco delle categorie di immagini
categories = ["ADC", "DWI", "FLAIR", "GRE", "MDC", "T1W"]

# Itera su tutte le categorie
for category in categories:
    # Percorso della cartella di input
    input_dir = os.path.join(root_dir, "validation_MULTI")
    # Percorso della cartella di output
    output_dir = os.path.join(root_dir, "validation_MULTI_" + category)

    # Crea la cartella di output se non esiste già
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Elenco delle sottocartelle (GBM e MET)
    subfolders = ["GBM", "MET"]

    # Itera su tutte le sottocartelle
    for subfolder in subfolders:
        # Percorso della sottocartella di input
        subfolder_input_dir = os.path.join(input_dir, subfolder)
        # Percorso della sottocartella di output
        subfolder_output_dir = os.path.join(output_dir, subfolder)

        # Crea la sottocartella di output se non esiste già
        if not os.path.exists(subfolder_output_dir):
            os.makedirs(subfolder_output_dir)

        # Cerca le immagini corrispondenti alla categoria nella struttura delle
cartelle di input
        for root, dirs, files in os.walk(subfolder_input_dir):
            for filename in files:
                if filename.endswith((".jpg", ".jpeg")):
                    # Converte il nome della categoria e il nome del file in
lowercase
                    category_uppercase = category.upper()
                    filename_uppercase = filename.upper()
                    if category_uppercase in filename_uppercase:
                        src_path = os.path.join(root, filename)
                        dst_filename = Path(filename_uppercase).stem + ".jpg"
                        dst_path = os.path.join(subfolder_output_dir, dst_filename)
                        shutil.copy(src_path, dst_path)
```

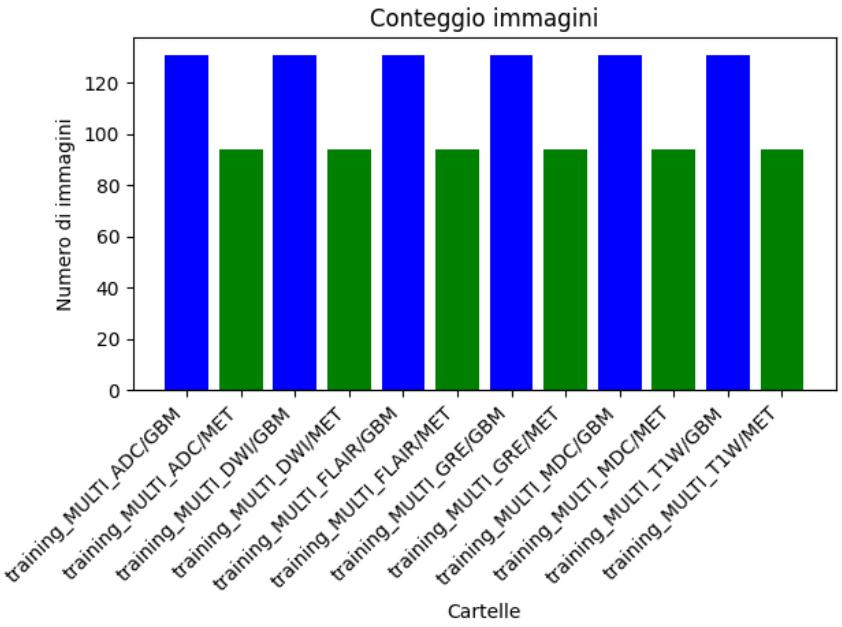
4) Spostare le cartelle ottenute nella cartella DATASET_MULTI su Google Drive.

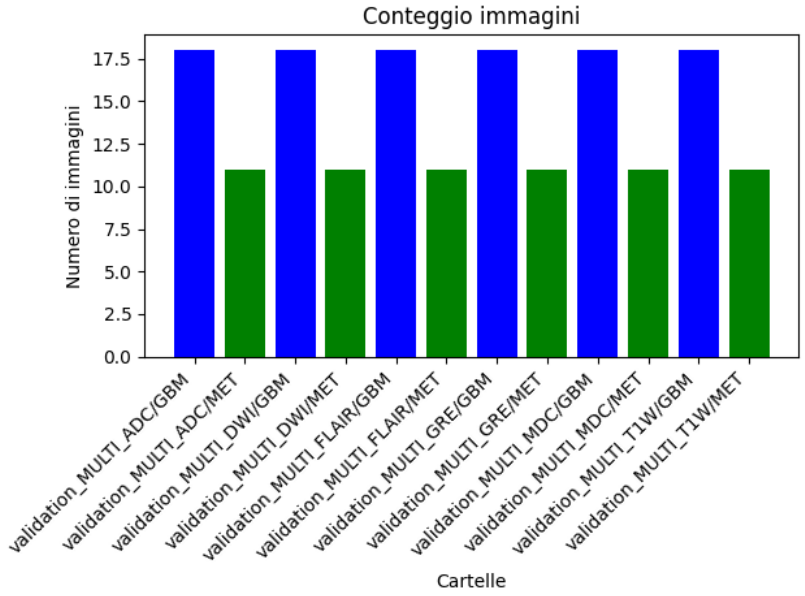
Prima di proseguire, è importante notare che il campione numero 155 presenta due copie

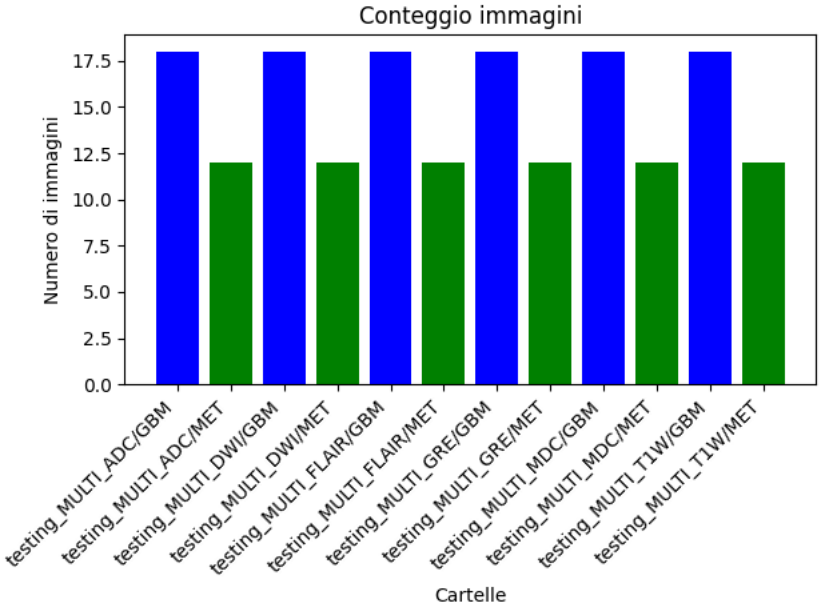
identiche dell'immagine ADC_MET0155 (rispettivamente denominate ADC_MET0155.jpg e ADC_MET0155 2.jpg). La seconda immagine è stata eliminata. Successivamente, si importano i tre dataset training_VOL, validation_VOL e testing_VOL utilizzando la funzione datasets.ImageFolder() messa a disposizione da Pytorch e indicizzandoli in un dizionario. Il dataset di training contiene circa l'80% delle immagini, quello di validation un ulteriore 10% e quello di testing il restante 10% circa.



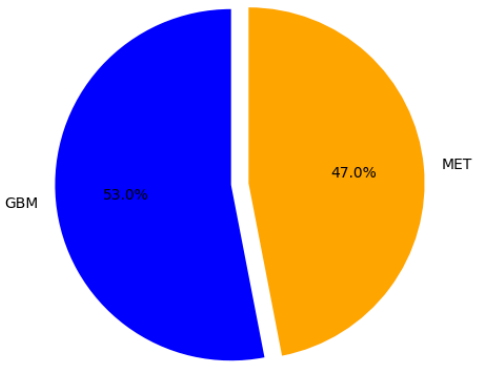

Inoltre, analizzando più approfonditamente il dataset si può notare che ogni campione possiede esattamente una singola immagine per ogni categoria, per cui non ci sono errori nel dataset.






Analizzando le percentuali di valori etichettati come GBM e come MET, si può concludere che il dataset è lievemente sbilanciato in favore della classe MET, anche se lo



sbilanciamento è inferiore rispetto al caso del dataset MULTI.

Successivamente sono state applicate delle trasformazioni alle immagini, in modo da poter addestrare i modelli in maniera più efficace. Innanzitutto, è stato necessario applicare un **ridimensionamento** delle immagini per due motivi:

1. non tutte le immagini all'interno dei dataset hanno la stessa dimensione;
2. i modelli pretrainati richiedono immagini di dimensioni prefissate per poter lavorare correttamente. Nel caso della maggior parte dei modelli utilizzati, la dimensioni richiesta era *224 x 224 pixel.*

In questo caso, unitamente al ridimensionamento si è provato ad eseguire anche altre trasformazioni della morfologia delle immagini, vale a dire il Center Crop e il Random Crop. La prima operazione produce delle immagini di dimensioni *224 x 224 pixel* tagliando l'immagine originale a partire dal centro. La seconda invece produce delle immagini di dimensioni *224 x 224 pixel* ritagliando un'area casuale dell'immagine. Tali operazioni dovrebbero rendere il dataset più robusto a variazioni della posizione dell'oggetto principale all'interno dell'immagine. Tuttavia, i test empirici svolti hanno mostrato che non c'è stata alcuna variazione apprezzabile nelle performance dei modelli.

Dopo il ridimensionamento, le immagini sono state trasformate in **tensori**: si tratta di un passaggio obbligato al fine di dare i dati in input ai modelli.

Successivamente, si può opzionalmente eseguire la **normalizzazione** dell'immagine, utilizzando dei valori specifici oppure dei valori consigliati dagli sviluppatori dei modelli pretrainati. In questo caso, si sono testati due tipi diversi di normalizzazione: una è quella consigliata dal MNIST, mentre l'altra è quella consigliata per Resnet/VGG. Si sono ottenuti risultati migliori utilizzando la prima normalizzazione. Il codice delle trasformazioni è il seguente:

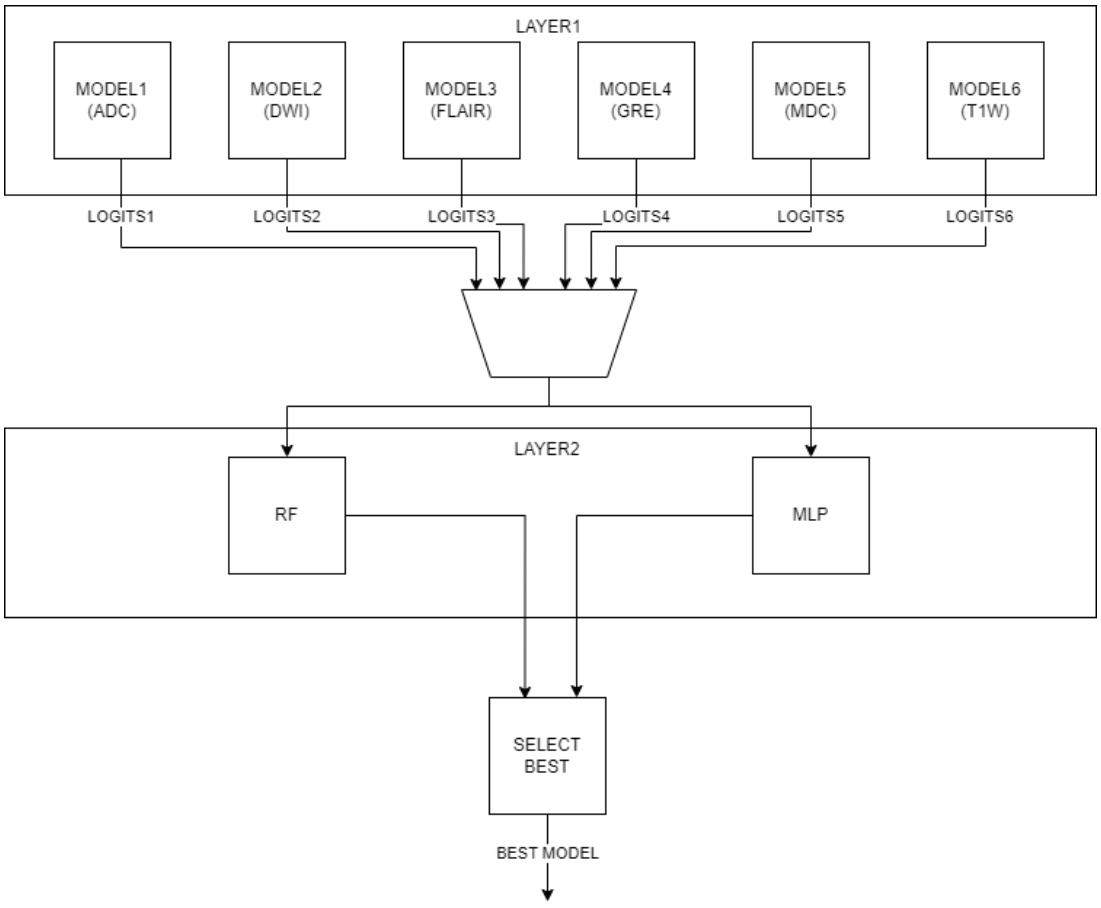
```
transform = transforms.Compose([
    transforms.Resize((224,224)),
    #transforms.CenterCrop(224),
    #transforms.RandomCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
    #transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
```

A questo punto si può passare alla fase di progettazione dell'architettura della rete, che in

questo caso è più complessa rispetto a quella descritta del capitolo precedente.

## 3.2 Struttura della rete

Ricordiamo che ogni campione è caratterizzato da sei immagini RMN di tipo “ADC”, "DWI", "FLAIR", "GRE", "T1W" e "MDC". Un modo efficace di riuscire ad utilizzare tutti i dati a nostra disposizione è quello di creare una rete a due livelli. Il primo livello è formato da sei reti CNN che ricevono in ingresso una delle sei immagini del campione. Il secondo livello riceve l’output del penultimo layer (**logits**) di ognuna delle reti del primo livello. Il classificatore del secondo livello effettuerà le predizioni finali sulla base dei valori dei logits ricevuti in input. In particolare, nel secondo livello i migliori classificatori per generare predizioni accurate, sono risultati il Random Forest e il MultiLayer Perceptron.




### 3.2.1 Livello 1

In questo livello sono state addestrate sei reti CNN che ricevono in ingresso un tipo di immagine RMN. Per fare ciò è necessario iterare su ogni tipo di immagine utilizzando un

for ed andare ad addestrare il relativo modello. Per quanto riguarda la scelta degli iperparametri, essi sono rimasti invariati rispetto al capitolo precedente, eccetto per i valori di:

- Batch Size: è stato necessario ridurre il valore della batch size in quanto si hanno a disposizione meno campioni del caso precedente. In particolare, si sono testati valori pari a 4, 8, 64.
- Numero di epoche: si è ridotto il numero di epoche in quanto i modelli convergono molto più velocemente rispetto al caso precedente a causa del minore numero di campioni. Si sono testati i valori 5, 10, 20.

```
 categories = ['ADC', 'DWI', 'FLAIR', 'GRE', 'MDC', 'T1W']
test_result_acc = {}
predictions_training = {}
predictions_validation = {}
predictions_testing = {}
df_training = {}
df_validation = {}
df_testing = {}

for category in categories:
  ROOT = '/content/drive/MyDrive/PROGETTO_CCS'
  model_dir = os.path.join(ROOT, "MULTI/model_dir/" + category)
  PATHS = {
      "model_dir": model_dir,
      "cache_dir": os.path.join(ROOT, "MULTI/cache_dir/" + category),
      "inference_dir": os.path.join(ROOT, "MULTI/infer_dir/" + category),
      "eval_dir": os.path.join(ROOT, "MULTI/eval_dir/" + category),
  }

  paths = PATHS

  TRAIN_COMMON_PARAMS = {}

  ### Data ###
  TRAIN_COMMON_PARAMS["data.batch_size"] = 4
  TRAIN_COMMON_PARAMS["data.train_num_workers"] = 2
  TRAIN_COMMON_PARAMS["data.validation_num_workers"] = 2

  ### PL Trainer ###
  TRAIN_COMMON_PARAMS["trainer.num_epochs"] = 5
  TRAIN_COMMON_PARAMS["trainer.num_devices"] = 1
  TRAIN_COMMON_PARAMS["trainer.accelerator"] = "gpu" if use_gpu else "cpu"
  TRAIN_COMMON_PARAMS["trainer.ckpt_path"] = None  #  path to the checkpoint
you wish continue the training from
```

```
  ### Optimizer ###
  TRAIN_COMMON_PARAMS["opt.lr"] = 1e-4
  TRAIN_COMMON_PARAMS["opt.weight_decay"] = 0.001

  train_params = TRAIN_COMMON_PARAMS
  ## Training Data
  # Create dataset
  data_dir = '/content/drive/MyDrive/PROGETTO_CCS/DATASET_MULTI'
  transform = transforms.Compose([
      transforms.Resize((224,224)),
      #transforms.CenterCrop(224),
      #transforms.RandomCrop(224),
      transforms.ToTensor(),
      transforms.Normalize((0.1307,), (0.3081,))])
      #transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
  torch_train_dataset = {x: datasets.ImageFolder(os.path.join(data_dir,
x),transform) for x in ['training_MULTI_' + category, 'testing_MULTI_' +
category, 'validation_MULTI_' + category]}
  train_dataset = DatasetWrapSeqToDict(name='training_MULTI_' + category,
dataset=torch_train_dataset['training_MULTI_' + category],
sample_keys=('data.' + category + 'image', 'data.' + category + 'label'))
  train_dataset.create()

  # Create Fuse's custom sampler
  sampler = BatchSamplerDefault(
      dataset=train_dataset,
      balanced_class_name="data." + category + "label",
      num_balanced_classes=2,
      batch_size=train_params["data.batch_size"],
      balanced_class_weights=None,
  )

  # Create dataloader
  train_dataloader = DataLoader(
      dataset=train_dataset,
      batch_sampler=sampler,
      collate_fn=CollateDefault(),
      num_workers=train_params["data.train_num_workers"],
  )

  ## Validation data
  # Create dataset
  validation_dataset = DatasetWrapSeqToDict(name='validation_MULTI_' +
category, dataset=torch_train_dataset['validation_MULTI_' + category],
sample_keys=('data.' + category + 'image', 'data.' + category + 'label'))
  validation_dataset.create()

  # dataloader
  validation_dataloader = DataLoader(
```

```
        dataset=validation_dataset,
        batch_size=train_params["data.batch_size"],
        collate_fn=CollateDefault(),
        num_workers=train_params["data.validation_num_workers"],
    )

    ## MODEL ##
    import torchvision.models as models

    def create_model():
        torch_model = models.efficientnet_v2_l(pretrained=True)
        # wrap basic torch model to automatically read inputs from batch_dict
and save its outputs to batch_dict
        model = ModelWrapSeqToDict(
            model=torch_model,
            model_inputs=["data." + category + "image"],
            post_forward_processing_function=perform_softmax,
            model_outputs=["model.logits." + category + "classification",
"model.output." + category + "classification"],
        )
        return model
    model = create_model()

    ## loss
    losses = {
        "cls_loss": LossDefault(
            pred="model.logits." + category + "classification", target="data."
+ category + "label", callable=F.cross_entropy, weight=1.0
        ),
    }

    ## METRICS
    train_metrics = OrderedDict(
        [
            ("operation_point", MetricApplyThresholds(pred="model.logits." +
category + "classification")),  # will apply argmax
            ("accuracy",
MetricAccuracy(pred="results:metrics.operation_point.cls_pred",
target="data." + category + "label")),
        ]
    )
    validation_metrics = copy.deepcopy(train_metrics)  # use the same metrics
in validation as well

    best_epoch_source = dict(monitor="validation.metrics.accuracy",
mode="max")

    # create optimizer
    optimizer = optim.Adam(model.parameters(), lr=train_params["opt.lr"],
weight_decay=train_params["opt.weight_decay"])
```

```
  # create scheduler
  lr_scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer)
  lr_sch_config = dict(scheduler=lr_scheduler,
monitor="validation.losses.total_loss")

  # optimizer and lr sch - see pl.LightningModule.configure_optimizers
return value for all options
  optimizers_and_lr_schs = dict(optimizer=optimizer,
lr_scheduler=lr_sch_config)

  # create instance of PL module - FuseMedML generic version
  pl_module = LightningModuleDefault(
      model_dir=paths["model_dir"],
      model=model,
      losses=losses,
      train_metrics=train_metrics,
      validation_metrics=validation_metrics,
      best_epoch_source=best_epoch_source,
      optimizers_and_lr_schs=optimizers_and_lr_schs,
  )
  pl_module.set_predictions_keys(
    ["model.logits." + category + "classification","model.output." +
category + "classification", "data." + category + "label"]
  )  # which keys to extract and dump into file

  # create lightning trainer
  pl_trainer = pl.Trainer(
      default_root_dir=paths["model_dir"],
      max_epochs=train_params["trainer.num_epochs"],
      accelerator=train_params["trainer.accelerator"],
      devices=train_params["trainer.num_devices"],
  )

  # train
  pl_trainer.fit(pl_module, train_dataloader, validation_dataloader,
ckpt_path=train_params["trainer.ckpt_path"])
    # set the prediction keys to extract (the ones used be the evaluation
function).

  train_dataloader_pred = DataLoader(dataset=train_dataset,
collate_fn=CollateDefault(), batch_size=2, num_workers=2) #
testing_dataloader
  validation_dataloader_pred = DataLoader(dataset=validation_dataset,
collate_fn=CollateDefault(), batch_size=2, num_workers=2) #
testing_dataloader
  predictions_training[category] = pl_trainer.predict(pl_module,
train_dataloader_pred, return_predictions=True)
  predictions_validation[category] = pl_trainer.predict(pl_module,
validation_dataloader_pred, return_predictions=True)
```

```
  # convert list of batch outputs into a dataframe
  df_training[category] =
convert_predictions_to_dataframe(predictions_training[category])
  df_validation[category] =
convert_predictions_to_dataframe(predictions_validation[category])


  ## INFER
  INFER_COMMON_PARAMS = {}
  INFER_COMMON_PARAMS["infer_filename"] = "infer_file.gz"
  INFER_COMMON_PARAMS["checkpoint"] = "best_epoch.ckpt"
  INFER_COMMON_PARAMS["trainer.num_devices"] =
TRAIN_COMMON_PARAMS["trainer.num_devices"]
  INFER_COMMON_PARAMS["trainer.accelerator"] =
TRAIN_COMMON_PARAMS["trainer.accelerator"]

  infer_common_params = INFER_COMMON_PARAMS

  # setting dir and paths
  create_dir(paths["inference_dir"])
  infer_file = os.path.join(paths["inference_dir"],
infer_common_params["infer_filename"])
  checkpoint_file = os.path.join(paths["model_dir"],
infer_common_params["checkpoint"])

  # creating a dataloader
  testing_dataset = DatasetWrapSeqToDict(name='testing_MULTI_' + category,
dataset=torch_train_dataset['testing_MULTI_' + category],
sample_keys=('data.' + category +'image', 'data.' + category + 'label'))
  testing_dataset.create()
  testing_dataloader = DataLoader(dataset=testing_dataset,
collate_fn=CollateDefault(), batch_size=2, num_workers=2) #
testing_dataloader

  # load pytorch lightning module
  model = create_model()
  pl_module = LightningModuleDefault.load_from_checkpoint(
      checkpoint_file, model_dir=paths["model_dir"], model=model,
map_location="cpu", strict=True
  )

  # set the prediction keys to extract (the ones used be the evaluation
function).
  pl_module.set_predictions_keys(
      ["model.logits." + category + "classification","model.output." +
category + "classification", "data." + category + "label"]
  )  # which keys to extract and dump into file

  # create a trainer instance
  pl_trainer = pl.Trainer(
```

```
      default_root_dir=paths["model_dir"],
      accelerator=infer_common_params["trainer.accelerator"],
      devices=infer_common_params["trainer.num_devices"],
  )

  # predict
  predictions_testing[category] = pl_trainer.predict(pl_module,
testing_dataloader, return_predictions=True)

  # convert list of batch outputs into a dataframe
  df_testing[category] =
convert_predictions_to_dataframe(predictions_testing[category])
  save_dataframe(df_testing[category], infer_file)

  ## EVAL
  EVAL_COMMON_PARAMS = {}
  EVAL_COMMON_PARAMS["infer_filename"] =
INFER_COMMON_PARAMS["infer_filename"]

  eval_common_params = EVAL_COMMON_PARAMS

  class_names = ['GBM', 'MET']

  # metrics
  metrics = OrderedDict(
      [
          ("operation_point", MetricApplyThresholds(pred="model.output." +
category + "classification")),  # will apply argmax
          ("accuracy",
MetricAccuracy(pred="results:metrics.operation_point.cls_pred",
target="data." + category + "label")),
          (
              "roc",
              MetricROCCurve(
                  pred="model.output." + category + "classification",
                  target="data." + category + "label",
                  class_names=class_names,
                  output_filename=os.path.join(paths["inference_dir"],
"roc_curve.png"),
              ),
          ),
          ("auc", MetricAUCROC(pred="model.output." + category +
"classification", target="data." + category + "label",
class_names=class_names)),
      ]
  )

  # create evaluator
  evaluator = EvaluatorDefault()
```

```
  # run eval
  results = evaluator.eval(
      ids=None,
      data=os.path.join(paths["inference_dir"],
eval_common_params["infer_filename"]),
      metrics=metrics,
      output_dir=paths["eval_dir"],
      silent=False,
  )

  print("Done!")

  test_result_acc[category] = results["metrics.accuracy"]
```

Successivamente al ciclo for sono costruiti tre dataframe (training, validation e testing), contenenti i logits delle sei reti addestrate e sulle quali sono state effettuate le predizioni per tutti e tre i set di dati.

```
import pandas as pd
  # Crea una lista con i nomi delle categorie
categories = ['ADC', 'DWI', 'FLAIR', 'GRE', 'MDC', 'T1W']
lista = [df_training, df_testing, df_validation]
lista_output = []
lista_logits = []

for dataframe in lista:

  # Supponiamo che tu abbia i 6 dataframe separati per categoria
  df_adc = dataframe['ADC']  # DataFrame per la categoria ADC
  df_dwi = dataframe['DWI']  # DataFrame per la categoria DWI
  df_flair = dataframe['FLAIR']  # DataFrame per la categoria FLAIR
  df_gre = dataframe['GRE']  # DataFrame per la categoria GRE
  df_mdc = dataframe['MDC']  # DataFrame per la categoria MDC
  df_t1w = dataframe['T1W']  # DataFrame per la categoria T1W

  # Crea una lista vuota per le colonne del nuovo dataframe
  columns = ['id']

  # Aggiungi le colonne desiderate al nuovo dataframe
  for category in categories:
      columns.append(f'{category} Prob0')
      columns.append(f'{category} Prob1')

  columns.append('data.label')

  # Crea un nuovo dataframe con le colonne desiderate
  df_out = pd.DataFrame(columns=columns)

  columns_logits = ['id']
  for category in categories:
```

```
    for i in range(len(dataframe[category])):
      columns_logits.append(f"{category}{i}")
  columns_logits.append('data.label')
  df_logits = pd.DataFrame(columns=columns_logits)

  # Itera sulle righe del primo dataframe (usiamo df_adc come esempio)
  for index, row in df_adc.iterrows():
      # Estrai l'id del dataframe
      id_value = row['id']
      df_out.at[index, 'id'] = id_value
      df_logits.at[index, 'id'] = id_value

      label_value = row[f'data.ADClabel']
      df_out.at[index, 'data.label'] = label_value
      df_logits.at[index, 'data.label'] = label_value

  # Aggiungi i valori delle colonne desiderate dai dataframe per ogni
categoria
  for category in categories:
      # Estrai i valori dalla colonna
'model.output.{CATEGORIA}classification'
      for index, row in dataframe[category].iterrows():
        classification_list = row[f'model.output.{category}classification']
        first_value = classification_list[0]
        second_value = classification_list[1]
        df_out.at[index, f'{category} Prob0'] = first_value
        df_out.at[index, f'{category} Prob1'] = second_value
  df_out['data.label'] = pd.to_numeric(df_out['data.label'],
errors='coerce') # trasforma label in numerico
  lista_output.append(df_out)


  # logits
  for category in categories:
    for index, row in dataframe[category].iterrows():
      logits_list = row[f'model.logits.{category}classification']
      for i in range(len(logits_list)):
        df_logits.at[index,f'{category}{i}'] = logits_list[i]
  df_logits['data.label'] = pd.to_numeric(df_out['data.label'],
errors='coerce') # trasforma label in numerico
  df_logits = df_logits.copy()
  lista_logits.append(df_logits)

df_training_output = lista_output[0]
df_testing_output = lista_output[1]
df_validation_output = lista_output[2]

df_training_logits = lista_logits[0]
df_testing_logits = lista_logits[1]
df_validation_logits = lista_logits[2]
```

Le reti pretrainate provate sono le tre che hanno performato meglio sul dataset VOL, dunque ResNet151, ResNet18 ed EfficientNet_V2_L. In più, si è voluto testare anche il modello VGG16, essendo il modello precedentemente utilizzato nell'abstract.

Di seguito sono riportati in forma tabellare i valori di accuracy che si otterrebbero utilizzando solamente un tipo di immagine per effettuare le predizioni:

| MODELLO | ADC | DWI | FLAIR | GRE | MDC | T1W |
|---|---|---|---|---|---|---|
| ResNet18 | 83.33% | 83.33% | 76.67% | 80.00% | 66.67% | 73.33% |
| ResNet152 | 90.00% | 80.00% | 86.67% | 86.67% | 83.33% | 60.00% |
| EfficientNet_V2_L | 90.00% | 70.00% | 83.33% | 86.67% | 93.33% | 80.00% |
| VGG16 | 93.33% | 76.66% | 80.00% | 80.00% | 90.00% | 76.67% |

### 3.2.2 Livello 2

Nel secondo livello è addestrato un classificatore in grado di ricevere in input tutti e sei i logits del livello precedente e generare in output una predizione. In particolare, i due migliori classificatori sono risultati il Random Forest ed il MultiLayer Perceptron (MLP).

Il **Random Forest** è un algoritmo di apprendimento automatico utilizzato principalmente per problemi di classificazione e regressione. Si basa sul concetto di ensemble learning, che combina le previsioni di più modelli per ottenere risultati più accurati e stabili.

In particolare, il Random Forest crea un insieme di alberi decisionali, noti come "alberi casuali", ogni albero è addestrato su un sottoinsieme casuale dei dati di addestramento e utilizza solo un sottoinsieme casuale delle feature disponibili. Questa tecnica di campionamento casuale, chiamata bagging, aiuta a ridurre la varianza e il rischio di overfitting.

Gli iperparametri presi in considerazione sono stati due: la profondità massima di ogni albero (max_depth) ed il numero massimo di alberi che vengono creati (n_estimators).

Impostare una profondità massima limita la complessità dell'albero e impedisce che si adatti troppo ai dati di addestramento, ma potrebbe portare a predizioni meno accurate.

Più stimatori vengono utilizzati, più complessa diventa la previsione aggregata e più aumenta anche il tempo di addestramento e di previsione del modello. L'aumento del

numero di stimatori può contribuire a ridurre la varianza dell'ensemble e migliorare la stabilità delle previsioni.

```
# Seleziona le colonne features
features = df_training_logits.columns[1:-1]  # Escludi la colonna 'id' e
'data.label'
# Seleziona la colonna target
target = 'data.label'
train_df, test_df, valid_df, train_labels, test_labels, valid_labels =
[df_training_logits[features], df_testing_logits[features],
df_validation_logits[features], df_training_logits[target],
df_testing_logits[target], df_validation_logits[target]]

# Iperparametri
n_estimators = [50, 100, 500, 1000]
max_depth = [None, 10, 20]
# Valori migliori da salvare
best_accuracy = 0
best_model = None
best_estimator = 0
best_depth = None

# Ciclo su tutti gli iperparametri per trovare la migliore combinazione e
salvarla
for estimator in n_estimators:
    for depth in max_depth:
        # Crea il modello di Random Forest
        random_forest = RandomForestClassifier(n_estimators=estimator,
max_depth=depth)
        # Addestra il modello sul training set
        random_forest.fit(train_df, train_labels)
        # Effettua la predizione sul test set
        predictions = random_forest.predict(valid_df)
        # Valuta l'accuratezza del modello
        accuracy = accuracy_score(valid_labels, predictions)
        # Salva solo il modello con l'accuratezza più alta
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_model = random_forest
            best_estimator = estimator
            best_depth = depth

# Salva il modello con l'accuratezza più alta
model_random_forest_path = os.path.join(ROOT,
"MULTI/RandomForest/random_forest_model.pkl")
with open(model_random_forest_path, 'wb') as file:
    pickle.dump(best_model, file)

# Effettua la predizione sul dataset di test utilizzando il modello migliore
```

```
predictions = best_model.predict(test_df)
# Salva le predizioni
predictions_path = os.path.join(ROOT, "MULTI/RandomForest/predictions.npy")
np.save(predictions_path, predictions)
# Calcola le metriche associate al modello migliore
accuracy = accuracy_score(test_labels, predictions)
recall = recall_score(test_labels, predictions)
precision = precision_score(test_labels, predictions)
f1 = f1_score(test_labels, predictions)

# Salva le metriche
metrics_path = os.path.join(ROOT, "MULTI/RandomForest/metrics.txt")
with open(metrics_path, 'w') as file:
    file.write(f'Accuracy: {accuracy}\n')
    file.write(f'Recall: {recall}\n')
    file.write(f'Precision: {precision}\n')
    file.write(f'F1-score: {f1}\n')
    file.write('\nParametri migliori:\n')
    file.write(f'Estimator: {best_estimator}\n')
    file.write(f'Max depth: {best_depth}\n')
```

L'**MLP** è un tipo di rete neurale artificiale feedforward ed è una delle architetture più comuni utilizzate nell'apprendimento profondo. È chiamato "multilayer" perché è composto da più strati di neuroni, e "perceptron" perché ogni neurone nel MLP è modellato sulla base del concetto di perceptron.

L'MLP è costituito da almeno tre strati di neuroni: uno strato di input, uno o più strati intermedi chiamati strati nascosti e uno strato di output. Ogni neurone in uno strato è collegato a tutti i neuroni nello strato successivo tramite pesi, che vengono aggiustati da un algoritmo di ottimizzazione durante il processo di addestramento per ottenere la migliore performance del modello. Tramite il set di validazione è stato selezionato il miglior algoritmo di ottimizzazione tra i seguenti: Adam, Stochastic Gradient Descent (SGD), Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS).

Le funzioni di attivazione applicabili ai neuroni sono tante, in questo caso sono state provate la ReLU (Rectified Linear Unit), Logistic (Funzione logistica), Tanh (Tangente iperbolica) e Identity (Identità). Con Hidden Layer Size si possono variare il numero di livelli intermedi e il numero di neuroni per livello. Per esempio, la notazione (50,) indica un solo livello intermedio composto da 50 neuroni.

```
# Iperparametri
hidden_layer_sizes = [(50,), (100,), (50, 50), (100, 100)]
activations = ['relu', 'logistic', 'tanh', 'identity']
solvers = ['adam', 'sgd', 'lbfgs']

best_accuracy = 0
best_model = None
best_hidden_size = (50,)
best_activation = 'relu'
best_solver = 'adam'

# Ciclo su tutti gli iperparametri per trovare la migliore combinazione e
salvarla
for hidden_size in hidden_layer_sizes:
    for activation in activations:
        for solver in solvers:
            mlp = MLPClassifier(hidden_layer_sizes=hidden_size,
activation=activation, solver=solver, random_state=42, max_iter=1000)
            mlp.fit(train_df, train_labels)
            predictions = mlp.predict(valid_df)
            accuracy = accuracy_score(valid_labels, predictions)

            # Salva solo il modello con l'accuratezza più alta
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_model = mlp
                best_hidden_size = hidden_size
                best_activation = activation
                best_solver = solver

# Salva il modello con l'accuratezza più alta
model_mlp_path = os.path.join(ROOT, "MULTI/MLP/mlp_model.pkl")
with open(model_mlp_path, 'wb') as file:
    pickle.dump(best_model, file)

# Effettua la predizione sul dataset di test usando il modello migliore
predictions = best_model.predict(test_df)
# Salva le predizioni
predictions_path = os.path.join(ROOT, "MULTI/MLP/predictions.npy")
np.save(predictions_path, predictions)
# Calcola le metriche associate al modello migliore
accuracy = accuracy_score(test_labels, predictions)
recall = recall_score(test_labels, predictions)
precision = precision_score(test_labels, predictions)
f1 = f1_score(test_labels, predictions)

# Salva le metriche
metrics_path = os.path.join(ROOT, "MULTI/MLP/metrics.txt")
```

```
with open(metrics_path, 'w') as file:
    file.write(f'Accuracy: {accuracy}\n')
    file.write(f'Recall: {recall}\n')
    file.write(f'Precision: {precision}\n')
    file.write(f'F1-score: {f1}\n')
    file.write('\nParametri migliori:\n')
    file.write(f'Hidden Layer Size: {best_hidden_size}\n')
    file.write(f'Activation: {best_activation}\n')
    file.write(f'Solver: {best_solver}\n')
```

In entrambi i casi, sia per il Random Forest che per l'MLP, i modelli che sono risultati i migliori sono salvati permanentemente in un file.pkl. Inoltre, sono calcolate e salvate sempre in un file di testo tutte le metriche relative al miglior modello.

Di seguito si riporta la tabella con i risultati ottenuti utilizzando il Random Forest sul secondo livello e gli iperparametri che sono risultati migliori per ogni modello utilizzato sul primo livello.

| MODEL | ACC | REC | PREC | F1 | ESTIMATORS | MAX DEPTH |
|---|---|---|---|---|---|---|
| ResNet18 | 83.33% | 58.33% | 100% | 73.68% | 50 | 10 |
| ResNet152 | 90.00% | 75.00% | 100% | 85.71% | 50 | 10 |
| EfficientNet V2_L | 96.67% | 91.67% | 100% | 95.65% | 50 | 20 |
| VGG16 | 90.00% | 83.33% | 90.90% | 86.96% | 50 | 10 |

Di seguito, invece, si riporta la tabella con i risultati ottenuti utilizzando il MLP sul secondo livello e gli iperparametri che sono risultati migliori per ogni modello utilizzato sul primo livello.

| MODEL | ACC | REC | PREC | F1 | HIDDEN LAYERS SIZE | ACTIV | SOLV |
|---|---|---|---|---|---|---|---|
| ResNet18 | 86.67% | 66.67% | 100% | 80.00% | (50,) | TANH | SGD |
| ResNet152 | 80.00% | 75.00% | 100% | 85.71% | (50,) | RELU | SGD |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| EfficientNet V2_L | 93.33% | 83.33% | 100% | 90.90% | (50,) | RELU | LBFGS |
| VGG16 | 86.67% | 75.00% | 90.00% | 81.81% | (50,) | RELU | SGD |

L’analisi dei risultati ottenuti mostra che la migliore combinazione è la seguente:

- Livello 1: EfficientNet_V2_L con i seguenti iperparametri:
    - Batch Size: 4
    - Numero di epoche: 5
    - Learning Rate: $10^{-4}$
    - Weight Decay: $10^{-3}$
    - Optimzer: Adam
    - Loss Function: Cross Entropy
- Livello 2: Random Forest con i seguenti iperparametri:
    - Numero di stimatori: 50
    - Profondità massima: 20

# Conclusioni

Le analisi effettuate per entrambi i dataset portano a trarre le seguenti conclusioni:

1) Per il primo dataset (VOL), il modello risultato migliore in termini di accuracy è ResNet152, con un valore di 97.3%. Il secondo modello migliore è EfficientNet_V2_L con una accuracy di 96.67%. Tutti gli altri modelli hanno prestazioni comparabili e si attestano attorno al 94-96%.
2) Per il secondo dataset (MULTI), la coppia di modelli risultata migliore in termini di accuracy è (EfficientNet_V2_L, RandomForest) con una accuracy di 96.67%.

A valle dei risultati si ritiene che il modello EfficientNet_V2_L sia l'opzione migliore in quanto raggiunge ottime prestazioni con entrambi i dataset. Tutti i modelli testati raggiungono in ogni caso dei buoni valori di accuracy e sono fondamentalmente comparabili.

Alla luce di queste considerazioni, si può affermare che l'utilizzo di modelli di classificazione risulta un ottimo strumento per la diagnosi dei tumori cerebrali a supporto degli specialisti.

Per quanto riguarda gli sviluppi futuri, si ritiene che i risultati possano ancora migliorare aumentando le dimensioni del dataset, soprattutto nel caso del dataset MULTI, che allo stato attuale presenta un numero troppo ristretto di campioni.

# Bibliografia


1) Camilla Russo, Paolo Maresca, Alfredo Marinelli, *Abstract_Ital-IA23_CamillaRusso.pdf*
2) Camilla Russo, Paolo Maresca, Alfredo Marinelli, *Abstract_ITAL-IA2022_Camilla Russo.pdf*
3) Camilla Russo, Paolo Maresca, Alfredo Marinelli, *Abstract_MELECON22_final.pdf*
4) *https://github.com/topics/fusemedml*