FuseMedML – Guida rapida all'uso

v. 2022-1

Benfenati Domenico, Carandente Vincenzo

10 maggio 2022

Sommario

FuseMedML è un framework open-source basato su python progettato per migliorare la collaborazione e accelerare le scoperte nei dati medici attraverso tecnologie avanzate di Machine Learning. La versione iniziale è basata su PyTorch e si concentra sul deep learning nell'imaging medico. Questo manuale fornisce una panoramica di come ne sono definite le funzionalità e come è possibile utilizzarle.

Indice

1 Panoramica su FuseMedML

FuseMedML è un *framework* di collaborazione eccezionale che consente di ripetere un esperimento o riutilizzare alcune delle capacità originariamente scritte per progetti diversi, il tutto con il minimo sforzo. Utilizzando FuseMedML, è possibile scrivere componenti generici che possono essere facilmente condivisi tra i progetti in modo plug & play, semplificando la condivisione e la collaborazione.

Il design unico del software del framework fornisce molti vantaggi, rendendolo un framework ideale per la ricerca e lo sviluppo di deep-learning nell'imaging medico. In particolare, i vantaggi riguardano:

- Sviluppo rapido, flessibile e scalabile;
- Incoraggiamento alla condivisione, e alla collaborazione;
- Valutazioni standard;
- Competenza nell'imaging medico;
- Interoperabilità tra framework differenti

2 Installazione

Il miglior modo per installare FuseMedML è effettuare la clonazione della repository Github e renderla editabile tramite pip:

1. Duplicare la repository tramite `!git clone https://github.com/IBM/fuse-med-ml.git`
2. Rendere editabile `pip` tramite il comando `!pip install -e .`

> É necessario riavviare il runtime dopo l'esecuzione per rendere effettive le modifiche di `pip`

Un'alternativa, è semplicemente installare usando PyPI, tramite il comando `!pip install fuse-med-ml`.
FuseMedML supporta Python 3.6 o successivo e PyTorch 1.5 o successivo. L'intera lista delle dipendenze è consultabile tramite l'opportuno file all'interno dei documenti nella repository al seguente link

3 Overview sulle funzionalità

3.1 Terminologia

- **batch_dict**: Dizionario che aggrega dati e risultati durante un ciclo di esecuzione per un batch. Ciò che si ottiene è un dizionario "gerarchico", che include tre branch: `'data'`, `'model'`, `'losses'`. Tale dizionario permette di disaccoppiare i componenti del training. Per esempio la parte di codice `'classification_loss'` non interagisce direttamente con le zone data o model, ma viene specificata una stringa, che sarà una chiave del dizionario batch_dict, che serve per specificare sia:
  1. l'etichetta di classificazione: `'data.gt.global'`;

2. le predizioni del modello logits: `'model.head_0.logits'`.

- **epoch_results**: Dizionario che viene creato dal manager e include i valori di metriche e di loss, così come definiti dal template e calcolati per ogni specifica epoca.
- **Classi Base** **Fuse*Base**: Classi astratte dell'oggetto che formano insieme il framework Fuse.
- **Classi Default** **Fuse*Default**: Classi che forniscono un implementazione generica delle classi Base equivalenti. Tali implementazioni di default vengono in aiuto in molte situazioni, ed è possibile poi specificare delle implementazioni alternative a seconda dei casi.
- **sample_descriptor**: ID univoco che rappresenta i campioni trattati.

3.2 Accesso ai dizionari di output

Per accedere facilmente ai dati memorizzati nei dizionari di output, si usa FuseUtilsHierarchicalDict tramite la seguente operazione:

```
FuseUtilsHierarchicalDict.get(batch_dict, 'model.output.classification')
```

l'output sarà il batch_dict['model']['output']['classification']

3.3 Esempio di approccio disaccoppiato

```
FuseMetricAUC(pred_name='model.output.classifier_head',
              target_name='data.gt.gt_global.tensor')
```

FuseMetricAUC leggerà i tensori richiesti per calcolare l'AUC da batch_dict. Le chiavi rilevanti del dizionario sono pred_name e target_name. Questo approccio permette di scrivere una metrica generica che è completamente indipendente dal modello e dall'estrattore di dati. Inoltre, permette di riutilizzare facilmente questo oggetto in modo plug & play senza aggiungere codice extra. Tale approccio ci permette anche di usarlo più volte nel caso in cui abbiamo più teste/compiti.
FuseMedML include versioni pre-implementate delle classi astratte che possono essere usate in (quasi) ogni contesto. Tuttavia, se necessario, possono essere sostituite dall'utente senza influenzare gli altri componenti.
Di seguito è riportato un elenco delle principali classi astratte e del loro scopo:

3.4 Funzioni sui Dati

- **Modulo:** FuseDataSourceBase

  Scopo: Semplice oggetto che genera degli identificatori unici dei campioni, detti Descrittori. Questa classe è di solito specifica del progetto, ed è quindi necessaria l'implementazione da parte dell'utente. Semplici implementazioni sono però già incluse in FuseMedML.

  Esempio di implementazione: FuseDataSourceDefault legge una tabella in formato DataFrame, includendo due colonne: (1) descrittori del campione (2) fold di appartenenza.

Emette un elenco di descrittori di campioni per le fold richieste. Un esempio di descrittore di campione potrebbe essere un percorso di un file DICOM che identifica univocamente un campione.

- **Modulo:** `FuseProcessorBase`

  Scopo: Estrae e pre-elabora un singolo campione o parte di esso tramite i suoi descrittori. Questa classe è di solito specifica del progetto, ed è quindi necessaria l'implementazione da parte dell'utente. Semplici implementazioni sono però già incluse in FuseMedML.

  Esempio di implementazione: Dato un percorso ad un descrittore di un campione, carica i dati dell'immagine, ridimensiona, normalizza i valori dei pixel, ritaglia, e converte in un Tensore PyTorch.

- **Modulo:** `FuseCacheBase`

  Scopo: Memorizza il campione pre-elaborato per un rapido recupero di esso.

  Esempio di Implementazione: Le opzioni di cache su disco sono direttamente integrate all'interno dei moduli `FuseDatasetDefault` e `FuseDatasetGeneration`

- **Modulo:** `FuseAugmentorBase`

  Scopo: Esegue una pipeline di operazioni di data augmentation.

  Esempio di implementazione: É possibile applicare un oggetto in grado di effettuare operazioni di perturbazione, augmentation affine di tipo 2D/3D, ecc. Rif: `FuseAugmentorDefault`

- **Modulo:** `FuseDatasetBase`

  Scopo: Implementazione di un dataset Pytorch compresod di utility aggiuntive. A differenza un un classico dataset PyTorch, un dataset FuseMedML restituisce un dizionario, includendo ogni campione del dataset all'interno di esso, e indicando quest'ultimo con una 'label'. É possibile fare ugualmente un wrapping di un dataset PyTorch nativo, tramite il modulo `FuseDatasetWrapper`

  Esempio di implementazione: `FuseDatasetDefault` è un'implementazione generica di un dataset, che supporta operazioni di caching, processing, data augmentation, data_source, ecc.

- **Modulo:** `FuseVisualizerBase`

  Scopo: Strumento di debug, che permette la visualizzazione dell'input prima e dopo la data augmentation.

  Esempio di implementazione: `FuseVisualizerDefault` permette di visualizzare un immagine 2D

3.5 Funzioni sui Modelli

FuseMedML include tre tipologie di oggetti di tipo Model:

- `Model` : oggetto che include l'intero modello end to end. FuseMedML Model è un modello PyTorch che riceve come input `batch_dict`, aggiunge gli output del modello al dizionario `batch_dict['model']` e restituisce `batch_dict['model']`. Il modello PyTorch può essere facilmente convertito in modello in stile FuseMedML usando un wrapper `FuseModelWrapper`

- BackBone : oggetto che estrae caratteristiche spaziali da un'immagine. Backbone è un modello PyTorch che riceve come input un tensore o una sequenza di tensori e restituisce un tensore o una sequenza di tensori.

- Head : un oggetto che mappa caratteristiche della predizione e di solito include strati di pooling e strati densi/convonvoluzionali 1x1. Head riceve come input batch_dict e restituisce batch_dict.

Tutti questi oggetti ereditano direttamente dal modulo torch.nn.Module

3.6 Funzioni sulle Lossess

- **Modulo:** FuseLossBase

  Scopo: Oggetto che calcola l'intera loss o una parte di essa secondo una funzione predefinita.

  Esempio di implementazione: Il modulo FuseLossDefault richiama una data funzione di loss come torch.nn.functional.cross_entropy per il calcolo della loss.

3.7 Funzioni sulle Metriche

- **Modulo:** FuseMetricBase

  Scopo: Oggetto che calcola il valore/i valori della metrica definita. Raccoglie i dati rilevanti per il calcolo della metrica da batch_dict, e alla fine di un'epoca calcola la metrica. Può restituire un singolo valore o un dizionario che contiene diversi valori.

  Esempio di implementazione: FuseMetricAUC è una sottoclasse che calcola l'Accuracy Under Curve sfruttando la curva ROC.

3.8 Funzioni di Gestione

- **Modulo:** FuseManagerBase

  Il gestore FuseManagerDefault ha la responsabilità di utilizzare tutti i componenti forniti dall'utente e di addestrare un modello di conseguenza. Per mantenere la flessibilità, il gestore supporta degli elementi detti **callback** che possono influenzare dinamicamente il suo stato e i dizionari: batch_dict e epoch_results. Per riferimenti vedi FuseCallback. Un esempio di callback pre-implementato è FuseTensorboardCallback che è responsabile della scrittura dei dati sia di allenamento che di convalida nei logger di tensorboard nella cartella model_dir È anche possibile modificare il comportamento del gestore sovrascrivendo funzioni come `handle_batch()` o in alternativa implementare un nuovo gestore. Il gestore fornisce anche una funzione chiamata `infer` che ripristina da model_dir (la funzione del manager `train` memorizza le informazioni in questa directory) gli oggetti richiesti ed esegue l'inferenza sui descrittori dei campioni richiesti.

3.9 Funzioni di Analisi

- **Modulo:** FuseAnalyzerDefault

La responsabilità del modulo è quella di valutare un modello addestrato. L'Analyzer prende in input un file di inferenza, generato da `manager.infer()`. Tale file include i descrittori dei campioni e le loro previsioni, ma potrebbe anche includere gli obiettivi della ground truth e i metadati su ogni campione. In caso contrario, dovrebbe essere fornito un processore o un set di dati per estrarre l'obiettivo dato un descrittore del campione.

4 Template di utilizzo

4.1 Import di librerie

```
# ---- LIBRERIE PYTHON ----
import os
import logging

# ---- LIBRERIE PYTORCH ----
import torch.optim as optim
from torch.utils.data.dataloader import DataLoader

# ---- LIBRERIE FUSEMEDML ----
# ---------- Dati ------------
from fuse.data.sampler.sampler_balanced_batch import FuseSamplerBalancedBatch
from fuse.data.visualizer.visualizer_default import FuseVisualizerDefault
from fuse.data.augmentor.augmentor_default import FuseAugmentorDefault
from fuse.data.dataset.dataset_default import FuseDatasetDefault

# ---------- Modello ------------
from fuse.models.model_default import FuseModelDefault

# ---------- Manager ------------
from fuse.managers.callbacks.callback_tensorboard import FuseTensorboardCallback
from fuse.managers.callbacks.callback_metric_statistics import
↪  FuseMetricStatisticsCallback
from fuse.managers.callbacks.callback_time_statistics import
↪  FuseTimeStatisticsCallback
from fuse.managers.manager_default import FuseManagerDefault

# ---------- Utilities ------------
from fuse.utils.utils_debug import FuseUtilsDebug
from fuse.utils.utils_logger import fuse_logger_start
import fuse.utils.gpu as FuseUtilsGPU

# ---------- Analisi ------------
from fuse.analyzer.analyzer_default import FuseAnalyzerDefault
```

4.2 Fase di Addestramento

La fase di addestramento di un modello tramite FuseMedML passa per la definizione di 6 blocchi fondamentali:

1. Data;
2. Model;
3. Losses;
4. Metrics;
5. Callbacks;
6. Manager.

4.2.1 Modalità di Debug

Per supervisionare le fasi di esecuzione è possibile definire una modalità di Debug tramite la funzione di FuseMedML FuseUtilsDebug. Tale funzione gestisce un file log secondo una modalità in input a tale funzione: `default`, `fast`, `debug`, `verbose`, `user`.

```
mode = 'TODO' # TODO: indica la tua modalità di debug
debug = FuseUtilsDebug(mode)
```

4.2.2 Definizione dei path

Vanno successivamente definite le cartelle dove verranno inseriti modelli, dati e risultati della fase di inferenza. Dove viene scritto `'TODO'` vanno inserite le cartelle che si desidera creare. Il parametro `force_reset_model_dir` se impostato a `True` permette di resettare forzatamente la cartella di modello ad ogni iterazione.

```
ROOT = 'TODO'

PATHS = {'model_dir': os.path.join(ROOT, 'TODO'),
         'force_reset_model_dir': True,
         'cache_dir': os.path.join(ROOT, 'TODO'),
         'inference_dir': os.path.join(ROOT, 'TODO'),
         'eval_dir': os.path.join(ROOT, 'TODO')
         }
```

4.2.3 Definizione dei parametri di training

Si vanno a definire i parametri da utilizzare per la fase di training, dove compare `TODO` è necessario inserire i parametri specifici.

```
# ---- PARAMETRI SUI DATI ----
TRAIN_COMMON_PARAMS = {}
TRAIN_COMMON_PARAMS['data.batch_size'] = 'TODO-intero'
TRAIN_COMMON_PARAMS['data.train_num_workers'] = 'TODO-intero'
TRAIN_COMMON_PARAMS['data.validation_num_workers'] = 'TODO-intero'
TRAIN_COMMON_PARAMS['data.augmentation_pipeline'] = [
    'TODO'
    # TODO: definisci qui la pipeline delle operazioni di data augumentation
    # Sugg: puoi usare la pipeline di default all'interno del file
    ↪  Fuse.data.augmentor.augmentor_toolbox.aug_image_default_pipeline
]

# ---- PARAMETRI DI GESTIONE ----
TRAIN_COMMON_PARAMS['manager.train_params'] = {
    'num_epochs': 'TODO-intero',
    'virtual_batch_size': 'TODO-intero',  # Numero di batch in un batch virtuale
    'start_saving_epochs': 'TODO-intero',  # Prima epoca da cui salvare i pesi
    'gap_between_saving_epochs': 'TODO-intero'  # Numero di epoche tra i salvataggi

    # Ogni 5 epoche vengono salvati i pesi, partendo dalla decima epoca
}
TRAIN_COMMON_PARAMS['manager.best_epoch_source'] = {
    'source': 'TODO',  # Specifica la metrica da utilizzare (accuracy, AUC, ...)
```

```
    'optimization': 'TODO',  # Obiettivo, può essere max o min
    'on_equal_values': 'TODO'    # better/worse - indica cosa fare in caso di epoche
    ↪  con pari metrica, in questo caso considera la migliore
}
TRAIN_COMMON_PARAMS['manager.learning_rate'] = 'TODO'
TRAIN_COMMON_PARAMS['manager.weight_decay'] = 'TODO'
TRAIN_COMMON_PARAMS['manager.resume_checkpoint_filename'] = None  # Se è None, carica
↪  i checkpoint precedenti
```

4.2.4 Funzione di Training

```
def train_template(paths: dict, train_common_params: dict):
```

La costruzione della funzione di addestramento consta di differenti fasi:

1. **Fase 1: Definizione di un Logger opzionale per il debug**

   Un **Logger** è un oggetto che genera informazioni di log su tre vie:

   - sulla console;
   - su un file, salvandovi le informazioni prodotte sulla console;
   - output verboso, usato per le operazioni di debug.

```
    fuse_logger_start(output_path=paths['model_dir'],
    ↪  console_verbose_level=logging.INFO)
    lgr = logging.getLogger('Fuse')
    lgr.info('Fuse Train', {'attrs': ['bold', 'underline']})

    lgr.info(f'model_dir={paths["model_dir"]}', {'color': 'magenta'})
    lgr.info(f'cache_dir={paths["cache_dir"]}', {'color': 'magenta'})
```

2. **Fase 2: Definizione delle classi per i dati**

   Si procede alla costruzione dei DataLoader (torch.utils.data.DataLoader) per le fasi di train e validation. Vengono usati i seguenti componenti generici di Fuse:

   - **Sorgente Dati:** `FuseDataSourceBase`

     Classe che fornisce un elenco di descrittori dei campioni

   - **Processor:** `FuseProcessorBase`

     Gruppo di classi che estrae dei sample da un descrittore. Il Processor si divide in due parti (1) 'input' (2) 'gt'. Entrambi sono usati per il train, ma la parte 'input' viene usata per la fase di inferenza.
     La parte 'input' viene aggregata in `batch_dict['data.input.<NomeProcessor>.*']`, mentre la parte 'gt' viene aggregata in `batch_dict['data.gt.<NomeProcessor>.*']`.

     Le classi Processor disponibili sono:

     – `FuseProcessorCSV`: Leggo da un file CSV, la funzione `call()` ritorna un campione come dizionario;

- FuseProcessorDataFrame: Leggo da un Dataframe o da un Dataframe pickle, e la funzione `call()` ritorna un campione come dizionario;
- FuseProcessorDataFrameWithGT: Leggo sia da un Dataframe che da un Dataframe pickle, la funzione `call()` ritorna il valore di un tensore come dizionario;
- FuseProcessorRand: Genero una ground trouth randomica, utile per i test.

- **Dataset:** FuseDatasetBase

  Le classi Dataset disponibili sono:

  - FuseDatasetDefault: Implementazione generica;
  - FuseDatasetGenerator: Implementazione da usare quando si generano semplici campioni in una volta sola (per esempio, patch di una singola immagine);
  - FuseDatasetWrapper: wrapper per un dataset di Pytorch che converte ogni campione in un dizionario.

- **Augmentor:** FuseAugmentorBase

  La classe, opzionale, si occupa di fare data augmentation tramite una pipeline di operazioni.

- **Visualizzatore:** FuseVisualizerBase

  Classe opzionale per la visualizzazione dei dai prima di effettuare le operazioni di augumentation. Le classi disponibili sono:

  - FuseVisualizerDefault: Visualizzatore per immagini 2D;
  - Fuse3DVisualizerDefault: Visualizzatore per immagini 3D;
  - FuseVisualizerImageAnalysis: Visualizzatore di immagini per analisi.

- **Campionatore:** Sampler

  Classe che recupera l'elenco dei campioni da utilizzare per ogni batch. Le classi disponibili sono:

  - FuseSamplerBalancedBatch: Campionatore che genera dei batch bilanciati. Supporta il bilanciamento delle classi secondo una probabilità o un peso.

```
# ==========================
# ---- DATI DI TRAINING ----
# ==========================
lgr.info(f'Train Data:', {'attrs': 'bold'})

## --- SORGENTE DATI ---
train_data_source = FuseDataSourceDefault()

## --- PROCESSOR ---
# TODO - Crea istanze delle classe FuseProcessorBase e aggiungile ai dizionari
input_processors = {}
gt_processors = {}

## --- AUGMENTOR (opzionale) ---
augmentor = FuseAugmentorDefault(augmentation_pipeline=
                    train_common_params['data.augmentation_pipeline'])

# --- VISUALIZZATORI (opzionale) ---
# TODO - inserisci le stringhe per il tuo problema
```

```
    visualiser = FuseVisualizerDefault(image_name='TODO', label_name='TODO')

    # --- DATASET ---
    train_dataset = FuseDatasetDefault(cache_dest=paths['cache_dir'],
                                       data_source=train_data_source,
                                       input_processors=input_processors,
                                       gt_processors=gt_processors,
                                       augmentor=augmentor,
                                       visualizer=visualiser)

    lgr.info(f'- Load and cache data:')
    train_dataset.create()
    lgr.info(f'- Load and cache data: Done')

    ## Suggerimenti aggiuntivi:
    # 1. Per conoscere la struttura di dati risultante stampa train_dataset[0]
    # 2. visualizza gli input con train_dataset.visualize(sample_index)
    # 3. visualizza i dati aumentati e non con
    ↪  'train_dataset.visualize_augmentation(sample_index)'
    # 4. Controlla il sommario delle statistiche con train_dataset.summary()

    ## --- CAMPIONATORE ---
    lgr.info(f'- Create sampler:')
    sampler = FuseSamplerBalancedBatch(dataset=train_dataset,
                                       balanced_class_name='TODO',
                                       num_balanced_classes='TODO',
                                       batch_size=
                                       ↪  train_common_params['data.batch_size'],
                                       balanced_class_weights=None)

    lgr.info(f'- Create sampler: Done')

    # Suggerimenti aggiuntivi:
    # 1. Non è necessario bilanciare secondo le etichette di classificazione,
    ↪  qualsiasi valore categorico andrà bene.
    # 2. Usa balanced_class_name per selezionare il valore categorico
    # 3. Non è necessario bilanciare equamente tra le classi.
    # 4. Utilizzare balanced_class_weights per specificare il numero di campioni
    ↪  richiesti in un batch per ogni classe

    ## --- DATALOADER ---
    train_dataloader = DataLoader(dataset=train_dataset,
                                  shuffle=False, drop_last=False,
                                  batch_sampler=sampler,
                                  collate_fn=train_dataset.collate_fn,
                                  num_workers=
                                  train_common_params['data.train_num_workers'])
    lgr.info(f'Train Data: Done', {'attrs': 'bold'})

    # ============================
    # --- DATI DI VALIDAZIONE ---
    # ============================
    lgr.info(f'Validation Data:', {'attrs': 'bold'})

    ## --- SORGENTE DATI ---
    validation_data_source = FuseDataSourceDefault()
```

```
## --- DATASET ---
validation_dataset = FuseDatasetDefault(cache_dest=paths['cache_dir'],
                                        data_source=validation_data_source,
                                        input_processors=input_processors,
                                        gt_processors=gt_processors,
                                        augmentor=None,
                                        visualizer=visualiser)

lgr.info(f'- Load and cache data:')
validation_dataset.create()
lgr.info(f'- Load and cache data: Done')

## --- DATALOADER ---
validation_dataloader = DataLoader(dataset=validation_dataset,
                                   shuffle=False,
                                   drop_last=False,
                                   batch_sampler=None,
                                   batch_size=
                                   ↪  train_common_params['data.batch_size'],
                                   num_workers= train_common_params
                                   ↪  ['data.validation_num_workers'],
                                   collate_fn=validation_dataset.collate_fn)
lgr.info(f'Validation Data: Done', {'attrs': 'bold'})
```

3. **Fase 3: Creazione del modello**

Si procede poi alla creazione un modello Pytorch (`torch.nn.Module`) usando componenti Fuse come:

- `FuseModelDefault` – componente generico che supporta una singola backbone con head multiple;
- `FuseBackbone` – scheletro generico di un modello;
- `FuseHead*` – implementazione generica di un head.

Gli output del modello saranno aggregati in `batch_dict['model.*']`, mentre gli output dei singoli head saranno aggregati in `batch_dict['model.<NomeHead>*']`. Altri modelli implementati sono:

- `FuseModelEnsemble` – esegue più sotto-modelli sequenzialmente;
- `FuseModelMultistream` – rete neurale convoluzionale con flussi di elaborazione multipli e head multiple.

```
lgr.info('Model:', {'attrs': 'bold'})
# TODO - definisci un modello Fuse oppure crealo da zero
model = FuseModelDefault(
    conv_inputs=(('data.input.input_0.tensor', 1),),
    backbone='TODO',  # Vedi anche: FuseBackboneInceptionResnetV2
    heads=['TODO']  # Vedi anche: FuseHeadGlobalPoolingClassifier,
    ↪  FuseHeadDenseSegmentation
)
lgr.info('Model: Done', {'attrs': 'bold'})
```

> Per rendere più semplice l'uso dei modelli tramite FuseMedML, si consiglia di sfruttare il relativo wrapper Fuse per modelli Pytorch, in modo da usufruire dei modelli pre-allenati presenti nella relativa libreria, e poterli convertire semplicemente in oggetti Fuse tramite la relativa classe wrapper `FuseModelWrapper`.

4. **Fase 4: Definizione delle funzioni di loss**

   Si crea il dizionario degli elementi di loss; ogni elemento è una sottoclasse di FuseLossBase. La perdita complessiva del modello sarà calcolata come somma pesata delle loss indicate in tale dizionario. Il valore della loss media del lotto per epoca sarà inclusa in `epoch_result['losses.<NomeLoss>']`, e la loss totale in `epoch_result['losses.total_loss']`. Le classi disponibili sono:

   - `FuseLossDefault` – wrapper di una funzione di perdita PyTorch con una API Fuse;
   - `FuseLossSegmentationCrossEntropy` – calcolo della cross entropy per locazione (`"dense"`) in una mappa di attivazione di classe (`"segmentation"`).

```
losses = {
    # TODO: aggiungi qui le loss (istanze della classe FuseLossBase)
}
```

5. **Fase 5: Definizione delle metriche**

   Si crea il dizionario delle metriche da calcolare durante la fase di addestramento, sia sul dataset di training che su quello di validazione. Le possibili metriche sono:

   - `FuseMetricAccuracy` – accuracy delle predizioni;
   - `FuseMetricAUC` – Area sotto la curva caratteristica operativa del ricevitore;
   - `FuseMetricBoundingBoxes` – metrica sui Bounding boxes;
   - `FuseMetricConfidenceInterval` – Wrapper metrico per gli intervalli di confidenza di un altra metrica;
   - `FuseMetricConfusionMatrix` – matrice di confusione sulle predizioni;
   - `FuseMetricPartialAUC` – Area parziale sotto la curva caratteristica operativa del ricevitore;
   - `FuseMetricScoreMap` – mappa degli score sulle predizioni.

```
metrics = {
    # TODO: aggiungi qui le metriche come istanze delle classi che preferisci
}
```

6. **Fase 6: Definizione dei callbacks** Un **callback** è un oggetto che può eseguire azioni in varie fasi di training; in ogni fase permette di manipolare i dati presenti nei dizionari `batch_dict` o `epoch_results`.

```
callbacks = [
        # Suggerimento: aggiungi dei callback addizionali qui
        # --- CALLBACK DI DEFAULT ----
        FuseTensorboardCallback(model_dir=paths['model_dir']),  # salva le
        ↪  statistiche su tensorboard
        FuseMetricStatisticsCallback(output_path=paths['model_dir'] +
        ↪  "/metrics.csv"),  # salva le statistiche in un file csv
        FuseTimeStatisticsCallback(
            num_epochs=train_common_params['manager.train_params']['num_epochs'],
            ↪  load_expected_part=0.1)  # profilatore di tempi
    ]
```

7. **Fase 7: Definizione di un Manager per l'addestramento**

Infine è necessario istanziare un Manager Fuse per la gestione delle fasi di training del modello, e alcuni parametri necessari per tale gestore.

```
    lgr.info('Train:', {'attrs': 'bold'})

    # --- OTTIMIZZATORE ---
    optimizer = optim.Adam(model.parameters(),
        lr=train_common_params['manager.learning_rate'],
        weight_decay=train_common_params['manager.weight_decay'])

    # --- SCHEDULER ---
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer)

    # --- CREAZIONE DEL MANAGER ---
    manager = FuseManagerDefault(output_model_dir=paths['model_dir'],
                                 force_reset=paths['force_reset_model_dir'])

    # --- SET DEGLI OGGETTI AL MANAGER ---
    manager.set_objects(net=model,
                        optimizer=optimizer,
                        losses=losses,
                        metrics=metrics,
                        best_epoch_source=
                        ↪  train_common_params['manager.best_epoch_source'],
                        lr_scheduler=scheduler,
                        callbacks=callbacks,
                        train_params=train_common_params['manager.train_params'])

    # --- CARICAMENTO DEI CHECKPOINT ---
    if train_common_params['manager.resume_checkpoint_filename'] is not None:
        manager.load_checkpoint(checkpoint=
        ↪  train_common_params['manager.resume_checkpoint_filename'],
        mode='train')

    # ==========================================
    # --- AVVIO DEL PROCESSO DI ADDESTRAMENTO ---
    # ==========================================
    manager.train(train_dataloader=train_dataloader,
                  validation_dataloader=validation_dataloader)

    lgr.info('Train: Done', {'attrs': 'bold'})
```

4.3 Fase di Inferenza

4.3.1 Definizione dei path

```
INFER_COMMON_PARAMS = {}
INFER_COMMON_PARAMS['infer_filename'] = os.path.join(PATHS['inference_dir'],
↪  'validation_set_infer.pickle')
INFER_COMMON_PARAMS['checkpoint'] = 'best'
```

4.3.2 Funzione di Inferenza

```
def infer_template(paths: dict, infer_common_params: dict):
    # --- LOGGER ---
    fuse_logger_start(output_path=paths['inference_dir'],
    ↪  console_verbose_level=logging.INFO)
    lgr = logging.getLogger('Fuse')
    lgr.info('Fuse Inference', {'attrs': ['bold', 'underline']})
    lgr.info(f'infer_filename={infer_common_params["infer_filename"]}', {'color':
    ↪  'magenta'})

    # --- SORGENTE DATI ---
    # TODO: Crea l'istanza di FuseDataSourceBase
    infer_data_source = 'TODO'
    lgr.info(f'experiment={infer_common_params["experiment_filename"]}', {'color':
    ↪  'magenta'})

    # --- CREAZIONE DEL MANAGER ---
    manager = FuseManagerDefault()
    # TODO - definire le chiavi di batch_dict che saranno salvate in un file
    output_columns = ['TODO']

    # ========================================
    # --- AVVIO DEL PROCESSO DI INFERENZA ---
    # ========================================
    manager.infer(data_source=infer_data_source,
                  input_model_dir=paths['model_dir'],
                  checkpoint=infer_common_params['checkpoint'],
                  output_columns=output_columns,
                  output_file_name=infer_common_params['infer_filename'])
```

4.4 Fase di Analisi

4.4.1 Definizione dei path

```
ANALYZE_COMMON_PARAMS = {}
ANALYZE_COMMON_PARAMS['infer_filename'] = INFER_COMMON_PARAMS['infer_filename']
ANALYZE_COMMON_PARAMS['output_filename'] = os.path.join(PATHS['analyze_dir'],
↪  'all_metrics')
ANALYZE_COMMON_PARAMS['num_workers'] = 4
ANALYZE_COMMON_PARAMS['batch_size'] = 2
```

4.4.2 Funzione di Analisi

```
def analyze_template(paths: dict, analyze_common_params: dict):
    # --- LOGGER ---
    fuse_logger_start(output_path=None, console_verbose_level=logging.INFO)
    lgr = logging.getLogger('Fuse')
    lgr.info('Fuse Analyze', {'attrs': ['bold', 'underline']})

    # --- PROCESSOR DI TIPO GT ---
    # TODO : includi tutti i processori gt necessari per la fase di analisi
    gt_processors = {
        'TODO',
    }

    # --- METRICHE ---
    # TODO : aggiungi le metriche necessarie al dizionario
    metrics = {
        'TODO'
    }

    # --- CREAZIONE DELL'ANALIZZATORE ---
    analyzer = FuseAnalyzerDefault()

    # ====================================
    # --- AVVIO DEL PROCESSO DI ANALISI ---
    # ====================================
    analyzer.analyze(gt_processors=gt_processors,
                     data_pickle_filename=analyze_common_params['infer_filename'],
                     metrics=metrics,
                     output_filename=analyze_common_params['output_filename'],
                     num_workers=analyze_common_params['num_workers'],
                     batch_size=analyze_common_params['num_workers'])
```

4.5 Esecuzione delle fasi

```
if __name__ == "__main__":
    # --- DEFINIZIONE NUMERO DI GPU NECESSARIE ---
    NUM_GPUS = 1

    force_gpus = None  # Se None sfrutta una GPU libera qualsiasi
    FuseUtilsGPU.choose_and_enable_multiple_gpus(NUM_GPUS, force_gpus=force_gpus)

    # ========================================
    # --- DEFINIZIONE DELLE FASI DA ESEGUIRE ---
    # ========================================
    RUNNING_MODES = ['TODO']  # Opzioni: 'train', 'infer', 'analyze'
    # a seconda del parametro eseguirai una fase dell'algoritmo, o addestramento, o
    ↪ inferenza o analisi finale. Nel vettore puoi anche mettere tutte le fasi
    ↪ insieme

    # --- MODALITA' DI ADDESTRAMENTO ---
    if 'train' in RUNNING_MODES:
        train_template(paths=PATHS, train_common_params=TRAIN_COMMON_PARAMS)
```

```
    # --- MODALITA' DI INFERENZA ---
    if 'infer' in RUNNING_MODES:
        infer_template(paths=PATHS, infer_common_params=INFER_COMMON_PARAMS)

    # --- MODALITA' DI ANALISI ---
    if 'analyze' in RUNNING_MODES:
        analyze_template(analyze_common_params=ANALYZE_COMMON_PARAMS)
```