| **Subject:** Applied Natural Language Processing  **Class:** MIS 6342.001  **Assignment:** MultiLabel Classification  **Name:** Sai Prakash Ravilisetty  **UTD ID:** SXR210170 |
| --- |

**MultiLabel Classification**

**Add docstrings to Class Trainer (Trainer\_v2.py file) and suggest improvements using ChatGPT**

**Code -**

from pathlib import Path

from torchmetrics import Metric

import torch

from datetime import datetime

import matplotlib.pyplot as plt

import torch.nn as nn

import random

import numpy as np

**Explanation:**

This code is a Python script that imports several modules including pathlib, torchmetrics, torch, datetime, matplotlib.pyplot, and torch.nn. It also defines a class called Metric, which is imported from the torchmetrics module.

* pathlib: This module provides an object-oriented interface to the file system. It is used to manipulate file and directory paths.
* Metric: This is a class from the torchmetrics module. It provides a base class for defining custom metrics in PyTorch.
* torch: This module provides the main functionality for creating and training deep learning models in PyTorch.
* datetime: This module provides classes for working with dates and times.
* matplotlib.pyplot: This module provides a MATLAB-like plotting framework for creating visualizations.
* nn: This module provides a set of predefined layers and functions for building neural networks in PyTorch.
* random: This module provides functions for generating random numbers and sequences.
* numpy: This module provides support for large, multi-dimensional arrays and matrices, along with a large library of mathematical functions.

**Improvements:**

Add error handling: It is a good practice to add error handling to the code to prevent unexpected errors and crashes. For example, adding try-except blocks to handle exceptions or errors in the code.

**Code -**

class Trainer:

def \_\_init\_\_(self, model, optimizer, criterion, device):

# input to constructor

self.model = model

self.optimizer = optimizer

self.criterion = criterion

self.device = device

self.model.to(self.device)

# set using function set loader

self.train\_loader = None

self.val\_loader = None

# set using function set metric

self.train\_metric = None

self.val\_metric = None

# set using function set early stopping

self.early\_stopping\_step = None

# set using set checkpoint function

self.save\_best = None

self.save\_every\_n\_epochs = None

self.save\_last\_epoch = None

self.timestamp = None

# set using set gradient clipping function

self.clipping = None

# updated during training

self.best\_score = None

self.total\_epochs = 0

self.total\_train\_steps = 0

self.total\_val\_steps = 0

self.best\_epoch = 0

self.train\_losses = []

self.val\_losses = []

self.train\_metrics = []

self.val\_metrics = []

self.early\_stopping\_counter = 0

self.early\_stop = False

# self.learning\_rates = []

**Explanation:**

This code defines a Trainer class that can be used to train a PyTorch model. The class has several instance variables that can be set using various functions defined in the class.

The Trainer class constructor takes four input parameters:

1. model: This is a PyTorch model that will be trained. It should be an instance of a subclass of nn.Module.
2. optimizer: This is the optimizer that will be used to update the model parameters during training. It should be an instance of a subclass of torch.optim.Optimizer.
3. criterion: This is the loss function that will be used to calculate the loss between the predicted output of the model and the ground truth labels. It should be an instance of a subclass of nn.Module.
4. device: This is the device (CPU or GPU) on which the model and tensors will be stored. It should be a string representing the device, such as "cpu" or "cuda:0".

Here's an explanation of the instance variables:

* model: The PyTorch model to be trained.
* optimizer: The PyTorch optimizer used to update the model's parameters during training.
* criterion: The PyTorch loss function used to calculate the loss during training.
* device: The device (CPU or GPU) on which to perform the calculations.
* train\_loader: The PyTorch DataLoader used to load the training data.
* val\_loader: The PyTorch DataLoader used to load the validation data.
* train\_metric: The PyTorch Metric used to compute the performance of the model on the training data.
* val\_metric: The PyTorch Metric used to compute the performance of the model on the validation data.
* early\_stopping\_step: The number of epochs to wait before stopping the training process if there is no improvement in the validation metric.
* save\_best: Whether to save the best model (based on the validation metric) during training.
* save\_every\_n\_epochs: Save the model every n epochs during training.
* save\_last\_epoch: Whether to save the model at the end of the last epoch.
* timestamp: The timestamp when the training process started.
* clipping: The value used for gradient clipping during training.
* best\_score: The best validation metric score obtained during training.
* total\_epochs: The total number of epochs trained.
* total\_train\_steps: The total number of steps taken during training.
* total\_val\_steps: The total number of steps taken during validation.
* best\_epoch: The epoch with the best validation metric score.
* train\_losses: The training loss for each epoch.
* val\_losses: The validation loss for each epoch.
* train\_metrics: The training metric score for each epoch.
* val\_metrics: The validation metric score for each epoch.
* early\_stopping\_counter: The number of epochs since the last improvement in the validation metric.
* early\_stop: Whether to stop the training process early if there is no improvement in the validation metric.

Overall, the Trainer class provides a framework for training a PyTorch model, with several customizable options for early stopping, checkpointing, and gradient clipping.

**Improvements:**

1. Use Python's built-in logging module to log information about the training process. This will make it easier to track the progress of the training, especially when training for many epochs.
2. Implement a learning rate scheduler to adjust the learning rate during training. This can help improve the performance of the model by allowing it to converge more quickly.

**Code -**

def set\_early\_stopping(self, patience=5, delta=0):

def early\_stopping\_step(val\_loss):

if self.best\_score is not None:

if val\_loss >= self.best\_score - delta:

self.early\_stopping\_counter += 1

print(

f'EarlyStopping counter: {self.early\_stopping\_counter} out of {patience}')

if self.early\_stopping\_counter >= patience:

self.early\_stop = True

else:

self.early\_stopping\_counter = 0

self.early\_stopping\_step = early\_stopping\_step

**Explanation:**

The set\_early\_stopping method is used to set a function that will be called during the validation phase of training to determine whether training should be stopped early to prevent overfitting. The function takes one input parameter, val\_loss, which is the validation loss calculated for the current epoch.

The method takes two optional input parameters:

1. patience: This is the number of consecutive epochs during which the validation loss must not improve by at least delta for training to be stopped. If the validation loss does not improve for patience consecutive epochs, training is stopped early to prevent overfitting. The default value is 5.
2. delta: This is the minimum amount by which the validation loss must improve in order for it to be considered an improvement. This can be used to prevent training from stopping too early due to small fluctuations in the validation loss. The default value is 0.

The set\_early\_stopping method defines an inner function, early\_stopping\_step, which takes the val\_loss as input and updates the Trainer object's early\_stopping\_counter attribute based on whether the validation loss has improved by at least delta compared to the best validation loss seen so far (self.best\_score). If the validation loss has not improved by at least delta, the early\_stopping\_counter is incremented. If the early\_stopping\_counter reaches the patience value, training is stopped by setting the early\_stop flag to True.

The early\_stopping\_step function also prints the current progress of the early stopping procedure, including the current early\_stopping\_counter and the patience value.

**Improvements:**

One improvement that could be made to the set\_early\_stopping method is to allow for more flexibility in the early stopping criteria. For example, instead of just stopping training when the validation loss does not improve for a certain number of epochs, the method could be modified to stop training if there is no improvement in some other metric, such as the validation accuracy or F1 score. Additionally, the method could be extended to allow for different stopping criteria based on different metrics, or to allow for more complex stopping criteria that depend on multiple metrics or other factors.

**Code -**

def set\_loaders(self, train\_loader, val\_loader=None):

self.train\_loader = train\_loader

self.val\_loader = val\_loader

**Explanation:**

The set\_loaders method sets the data loaders used for training and validation. The train\_loader parameter should be a PyTorch DataLoader object that provides batches of training data, while the val\_loader parameter (which is optional) should be a similar object providing validation data. These data loaders will be used to iterate over the training and validation data during training.

The train\_loader should typically be set to a DataLoader object that loads the training dataset in batches, shuffles the data, and applies any necessary data preprocessing or augmentation. The val\_loader can be set to a similar DataLoader object that loads the validation dataset. It is not required to use a validation set during training, but doing so can help to prevent overfitting and improve model generalization.

Once the data loaders have been set using this method, they can be used during training to iterate over the training and validation data, typically using a for loop that iterates over batches of data and performs forward and backward passes through the model on each batch.

**Code -**

def set\_metric(self, train\_metric: Metric, val\_metric: Metric = None):

self.train\_metric = train\_metric

self.val\_metric = val\_metric

**Explanation:**

The set\_metric method sets the evaluation metric(s) to be used during training and validation. The train\_metric parameter should be a PyTorch Metric object that defines the evaluation metric(s) to be used during training. This could be a classification metric such as accuracy or F1 score, or a regression metric such as mean squared error.

The val\_metric parameter is optional and allows for a separate evaluation metric to be used during validation if desired. If val\_metric is not provided, then the train\_metric will be used for both training and validation evaluation.

During training, after each epoch, the model's performance will be evaluated using the train\_metric and val\_metric if provided. The results of these evaluations will be stored in the train\_metrics and val\_metrics lists, respectively, which can be accessed later for analysis or plotting.

Overall, this method allows for flexible evaluation of the model during training and validation and allows the user to specify custom evaluation metrics as needed.

**Code -**

def set\_checkpoint(self, save\_path, save\_best=True, save\_every\_n\_epochs=None, save\_last\_epoch=False):

self.timestamp = datetime.now().strftime("%Y-%m-%d\_%H-%M-%S")

self.save\_path = save\_path

self.save\_best = save\_best

self.save\_every\_n\_epochs = save\_every\_n\_epochs

self.save\_last\_epoch = save\_last\_epoch

**Explanation:**

The set\_checkpoint method is used to set up the checkpointing functionality for the trainer. The checkpointing allows saving the model state at different points during the training process so that it can be reloaded later to continue the training or make predictions.

The input parameters for this method are:

* save\_path: the path to save the checkpoints to.
* save\_best: a boolean indicating whether to save only the best checkpoint based on the validation loss.
* save\_every\_n\_epochs: an integer indicating after how many epochs the model should be checkpointed. If None, the model will not be saved periodically.
* save\_last\_epoch: a boolean indicating whether to save the last epoch checkpoint regardless of its validation loss.

The method sets the timestamp attribute to the current date and time formatted as a string, which will be used in the names of the checkpoint files. It also sets the save\_path, save\_best, save\_every\_n\_epochs, and save\_last\_epoch attributes to the input parameters, which will be used to determine the checkpointing behavior during training.

**Code -**

def set\_gradient\_clipping(self, clip\_type, clip\_value, norm\_type=2):

if clip\_type.lower() == 'value':

self.clipping = lambda: nn.utils.clip\_grad\_value\_(

self.model.parameters(), clip\_value=clip\_value

)

elif clip\_type.lower() == 'norm':

self.clipping = lambda: nn.utils.clip\_grad\_norm\_(

self.model.parameters(), clip\_value, norm\_type

)

else:

raise ValueError(

"Invalid clip\_type provided. Use 'value' or 'norm'.")

**Explanation:**

This function sets the gradient clipping method to be used during training. Gradient clipping is a technique used to prevent the gradients from becoming too large or too small during training, which can lead to unstable training or vanishing/exploding gradients.

The function takes in three parameters: clip\_type, clip\_value, and norm\_type. clip\_type specifies the type of gradient clipping to be used, which can be either 'value' or 'norm'. clip\_value is the maximum allowed value for the gradients if clip\_type is 'value', or the maximum allowed norm for the gradients if clip\_type is 'norm'. norm\_type is the type of norm to be used for the 'norm' clipping method.

The function sets the self.clipping attribute to a lambda function that applies the chosen gradient clipping method to the model parameters. If an invalid clip\_type is provided, the function raises a ValueError.

Note that this function does not actually apply gradient clipping during training. Instead, the self.clipping function is called within the training loop at the appropriate time to perform the clipping.

**Code -**

def \_get\_batch\_metric(self, outputs, targets, multilabel, metric, threshold):

if multilabel:

outputs = (torch.sigmoid(outputs) > threshold).float()

else:

outputs = torch.argmax(outputs, dim=1)

batch\_metric = metric(outputs, targets)

**Explanation:**

This method is used to compute the metric for a batch of outputs and targets. It takes the following parameters:

* outputs: the output predictions from the model for the batch
* targets: the true target values for the batch
* multilabel: a boolean indicating whether the problem is multilabel or not
* metric: the metric function used to compute the metric score
* threshold: a threshold value used for multilabel classification, indicating the probability value above which a class is considered positive

If multilabel is True, the sigmoid function is applied to the output predictions and then a threshold comparison is made with the threshold value to obtain binary predictions. If multilabel is False, the argmax function is applied to the output predictions to obtain the class with the highest probability.

Finally, the batch metric score is computed by applying the metric function to the batch of predictions and targets. The computed score is returned.

**Improvements:**

One potential improvement for this method is to add error handling for the case where the metric parameter is None. Currently, if metric is None, the method will raise a TypeError when trying to call the function metric(outputs, targets). A possible solution for this is to add a check for metric before calling it, and set batch\_metric to None in case metric is None

**Code -**

def \_backward(self, loss, ):

loss.backward()

if callable(self.clipping):

self.clipping()

self.optimizer.step()

self.optimizer.zero\_grad()

**Explanation:**

This method takes a single argument, loss, which is a scalar tensor representing the loss value calculated by the model for a batch of data.

The method first calls the backward() method of the loss tensor, which computes the gradients of the loss with respect to all the parameters of the model. These gradients are accumulated in the respective parameter tensors.

Next, the method checks if the clipping attribute of the class is callable, which indicates that gradient clipping has been enabled. If so, it calls the clipping function, which clips the gradients to prevent them from becoming too large and causing numerical instability during optimization.

Finally, the method calls the step() method of the optimizer associated with the model, which updates the model parameters using the accumulated gradients. It then resets the gradients to zero using the zero\_grad() method of the optimizer, in preparation for the next batch of data to be processed.

**Improvements:**

The code looks good as is. However, you may consider adding some exception handling to handle cases where the optimizer doesn't have a zero\_grad() method or if there's an error in clipping the gradients. You may also want to consider passing the retain\_graph parameter to loss.backward() to ensure that the computation graph is retained in case multiple backward passes are needed.

**Code -**

def \_move\_inputs\_targets\_to\_gpu(self, inputs, targets):

if isinstance(inputs, tuple):

inputs = tuple(input\_tensor.to(self.device)

for input\_tensor in inputs)

else:

inputs = inputs.to(self.device)

targets = targets.to(self.device)

return inputs, targets

**Explanation:**

The \_move\_inputs\_targets\_to\_gpu method is a helper method used to move the inputs and targets tensors to the GPU device. This method takes two arguments: inputs and targets, which are the inputs and targets tensors, respectively.

The method first checks if the inputs argument is a tuple. If so, it iterates over the tuple and moves each input tensor to the GPU device. If inputs is not a tuple, it simply moves the input tensor to the GPU device. Finally, the targets tensor is moved to the GPU device.

This method is useful when training a neural network on a GPU device. Moving tensors to the GPU can speed up the training process by allowing computations to be performed in parallel.

**Improvements:**

One improvement that could be made to this method is to add a check to see if the device is already set to GPU. If the device is already set to GPU, the method could skip moving the tensors to the GPU and return them as is. This would prevent unnecessary data transfers between the CPU and GPU.

**Code -**

def \_log\_print\_epoch\_loss\_metric(self, train\_loss, train\_metric, val\_loss, val\_metric, epoch,

num\_epochs, dt\_train, dt\_valid):

if self.train\_metric:

print(

f"Epoch {epoch + 1}/{num\_epochs} - Train Loss: {train\_loss:.4f}, "

f"Train Metric: {train\_metric:.4f}, Train Time: {dt\_train}")

else:

print(

f"Epoch {epoch + 1}/{num\_epochs} - Train Loss: {val\_loss:.4f}, Train Time: {dt\_train}")

if self.val\_loader is not None:

if self.val\_metric:

print(

f"Epoch {epoch + 1}/{num\_epochs} - Val Loss: {val\_loss:.4f}, "

f"Val Metric: {val\_metric:.4f}, Val Time: {dt\_valid}")

else:

print(

f"Epoch {epoch + 1}/{num\_epochs} - Val Loss: {val\_loss:.4f}, Val Time: {dt\_valid}")

# print(f"Current Learning rate is {self.learning\_rates[-1]}")

print()

**Explanation:**

This function prints out the training and validation loss/metric and the time taken to complete each epoch during training. It takes in the following parameters:

* train\_loss: the training loss value for the current epoch
* train\_metric: the training metric value for the current epoch (if applicable)
* val\_loss: the validation loss value for the current epoch (if validation data is provided)
* val\_metric: the validation metric value for the current epoch (if applicable and if validation data is provided)
* epoch: the current epoch number
* num\_epochs: the total number of epochs to be run
* dt\_train: the time taken to complete the training epoch
* dt\_valid: the time taken to complete the validation epoch (if validation data is provided)

The function first checks if train\_metric and val\_metric are not None, indicating that metrics are being used for evaluation. It then prints out the training and validation loss/metric for the current epoch, as well as the time taken to complete each epoch. If val\_loader is not provided, only the training loss/metric and time are printed.

The function ends by printing a newline character for formatting purposes.

**Code -**

def \_run\_batch(self, inputs, targets, metric, multilabel, threshold, training):

inputs, targets = self.\_move\_inputs\_targets\_to\_gpu(

inputs, targets)

with torch.set\_grad\_enabled(training):

outputs = self.model(inputs)

loss = self.criterion(outputs, targets)

if training:

self.\_backward(loss, )

if metric is not None:

batch\_metric = self.\_get\_batch\_metric(

outputs, targets, multilabel, metric, threshold)

if training:

self.total\_train\_steps += 1

else:

self.total\_val\_steps += 1

return loss

**Explanation:**

This is a private method \_run\_batch that performs a forward and backward pass for a given batch of inputs and targets. The method also calculates the loss and metric for the batch (if metric is provided).

Here's a breakdown of the steps performed in the method:

1. The method first moves the inputs and targets to the device specified during initialization, typically a GPU device.
2. The forward pass is performed by passing the inputs to the model and obtaining the predicted outputs.
3. The loss is calculated by comparing the predicted outputs with the actual targets using the criterion specified during initialization.
4. If the method is called during training, the backward pass is performed by calling the private method \_backward and passing the loss as an argument.
5. If a metric is specified, the method calls the private method \_get\_batch\_metric to compute the metric using the predicted outputs and actual targets.
6. The method keeps track of the total number of training or validation steps performed so far.
7. Finally, the method returns the computed loss.

Overall, this method is used by the train and validate methods to perform forward and backward passes for each batch of data during training or validation.

**Code -**

def \_run\_epoch(self, loader, training=True, multilabel=False, threshold=0.5):

if training:

self.model.train()

metric = self.train\_metric

else:

self.model.eval()

metric = self.val\_metric

epoch\_loss = 0.0

num\_samples = 0

for i, (inputs, targets) in enumerate(loader):

loss = self.\_run\_batch(

inputs, targets, metric, multilabel, threshold, training)

epoch\_loss += loss.item() \* targets.size(0)

num\_samples += targets.size(0)

epoch\_loss /= num\_samples

if metric is not None:

epoch\_metric = metric.compute().item()

metric.reset()

else:

epoch\_metric = None

return epoch\_loss, epoch\_metric

**Explanation:**

This is a method \_run\_epoch in a PyTorch model training wrapper class that runs an epoch of training or validation on a given data loader. It takes in several parameters:

* loader: a PyTorch DataLoader object representing the dataset to be used for training/validation
* training: a boolean flag indicating whether this epoch is for training or validation
* multilabel: a boolean flag indicating whether the problem is a multilabel classification problem or not (default is False)
* threshold: a threshold value to be used for multilabel classification (default is 0.5)

The method first sets the model to training or evaluation mode based on the training parameter, and sets the appropriate metric to use (either the training metric or validation metric) based on the mode. Then, it loops through the data loader, and for each batch, it calls the \_run\_batch method to compute the loss and perform backpropagation if in training mode. The epoch loss is then updated by adding the batch loss multiplied by the number of samples in the batch. After looping through all batches, the epoch loss is divided by the total number of samples to obtain the average loss for the epoch.

If a metric is provided, it computes the metric value for the epoch using the compute method of the metric object and resets the metric object. Finally, the method returns the epoch loss and metric value (if applicable).

**Code -**

def save\_checkpoint(self, suffix=''):

save\_dir = Path(self.save\_path)

save\_dir.mkdir(parents=True, exist\_ok=True)

checkpoint\_path = save\_dir / f"checkpoint\_{self.timestamp}{suffix}.pt"

checkpoint\_data = {

'total\_epochs': self.total\_epochs,

'total\_train\_steps': self.total\_train\_steps,

'total\_val\_steps': self.total\_val\_steps,

'model\_state\_dict': self.model.state\_dict(),

'optimizer\_state\_dict': self.optimizer.state\_dict(),

'val\_loss': self.best\_score,

}

# Add losses and metrics history to the checkpoint

checkpoint\_data['train\_losses'] = self.train\_losses

checkpoint\_data['val\_losses'] = self.val\_losses

checkpoint\_data['train\_metrics'] = self.train\_metrics

checkpoint\_data['val\_metrics'] = self.val\_metrics

torch.save(checkpoint\_data, checkpoint\_path)

**Explanation:**

This method saves a checkpoint of the training progress. It creates a directory (if it doesn't already exist) to store the checkpoint file and saves the following information in a dictionary:

* total\_epochs: the total number of epochs trained
* total\_train\_steps: the total number of training steps taken
* total\_val\_steps: the total number of validation steps taken
* model\_state\_dict: the state dictionary of the trained model
* optimizer\_state\_dict: the state dictionary of the optimizer used to train the model
* val\_loss: the best validation loss achieved during training
* train\_losses: a list of training losses for each epoch
* val\_losses: a list of validation losses for each epoch
* train\_metrics: a list of training metrics (if specified) for each epoch
* val\_metrics: a list of validation metrics (if specified) for each epoch

The method then saves this dictionary to a file in the specified directory with a name indicating the timestamp and, optionally, a suffix.

**Improvements:**

1. Instead of saving the checkpoint to a single file, you can split it into multiple files to make it more modular. For example, you can save the model's state dict to one file, optimizer's state dict to another file, and losses/metrics history to another file.
2. You can also add more information to the checkpoint such as the current epoch, current learning rate, and any other relevant information that might be useful later during inference or analysis.

**Code -**

def load\_checkpoint(self, checkpoint\_path):

checkpoint = torch.load(checkpoint\_path)

self.model.load\_state\_dict(checkpoint['model\_state\_dict'])

self.optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

self.total\_epochs = checkpoint['total\_epochs']

self.total\_train\_steps = checkpoint['total\_train\_steps']

if 'total\_val\_steps' in checkpoint:

self.total\_val\_steps = checkpoint['total\_val\_steps']

if 'val\_loss' in checkpoint:

self.best\_score = checkpoint['val\_loss']

if 'train\_losses' in checkpoint:

self.train\_losses = checkpoint['train\_losses']

if 'val\_losses' in checkpoint:

self.val\_losses = checkpoint['val\_losses']

if 'train\_metrics' in checkpoint:

self.train\_metrics = checkpoint['train\_metrics']

if 'val\_metrics' in checkpoint:

self.val\_metrics = checkpoint['val\_metrics']

print(f"Loaded checkpoint from '{checkpoint\_path}'.")

**Explanation:**

This method loads a checkpoint from a specified path and sets the model state, optimizer state, and several other attributes of the trainer object based on the checkpoint data.

It first loads the checkpoint data using torch.load and then sets the model state and optimizer state using the load\_state\_dict method of the model and optimizer objects, respectively.

Then, it sets the total\_epochs, total\_train\_steps, total\_val\_steps, best\_score, train\_losses, val\_losses, train\_metrics, and val\_metrics attributes of the trainer object based on the corresponding keys in the checkpoint data dictionary.

Finally, it prints a message to indicate that the checkpoint has been loaded successfully.

Overall, this method is responsible for resuming training from a saved checkpoint.

**Improvements:**

It could be improved by adding error handling in case the checkpoint file is not found or has an incorrect format. For example, you could use a try-except block to catch the possible exceptions thrown by the torch.load() function and print an error message if necessary.

**Code -**

def train(self, num\_epochs, multilabel=False, threshold=0.5):

assert self.train\_loader is not None, "Train loader must be set before calling train()"

if self.val\_loader is None:

print(

'Validation loader is not set. The trainer will only execute training Loop')

if all(value is None for value in [self.save\_best, self.save\_every\_n\_epochs, self.save\_last\_epoch]):

print('Not saving any checkpoint')

for epoch in range(num\_epochs):

t0 = datetime.now()

train\_loss, train\_metric = self.\_run\_epoch(

self.train\_loader, training=True, multilabel=multilabel, threshold=threshold)

dt\_train = datetime.now() - t0

self.train\_losses.append(train\_loss)

if self.train\_metric:

self.train\_metrics.append(train\_metric)

if self.val\_loader is not None:

t0 = datetime.now()

val\_loss, val\_metric = self.\_run\_epoch(

self.val\_loader, training=False, multilabel=multilabel,

threshold=threshold)

dt\_valid = datetime.now() - t0

self.val\_losses.append(val\_loss)

if self.val\_metric:

self.val\_metrics.append(val\_metric)

if callable(self.early\_stopping\_step):

self.early\_stopping\_step(val\_loss)

if self.early\_stop:

print("Early stopping triggered")

break

if self.best\_score is None or val\_loss < self.best\_score:

self.best\_score = val\_loss

self.best\_epoch = self.total\_epochs + 1

if self.save\_best:

self.save\_checkpoint(suffix=f'\_best')

# saving checkpoint

if self.save\_every\_n\_epochs and (epoch + 1) % self.save\_every\_n\_epochs == 0:

self.save\_checkpoint(

suffix=f'\_epoch\_{self.total\_epochs + 1}')

if self.save\_last\_epoch:

self.save\_checkpoint(

suffix=f'\_last')

self.\_log\_print\_epoch\_loss\_metric(train\_loss, train\_metric, val\_loss, val\_metric, epoch,

num\_epochs, dt\_train, dt\_valid)

self.total\_epochs += 1

**Explanation:**

This code defines a train method for a Trainer class. This method is responsible for training a neural network model for a given number of epochs, using a specified train loader and an optional validation loader. The method uses the \_run\_epoch method to perform training and validation for each epoch.

The method first asserts that a train loader has been set before calling the train method. If a validation loader is not set, the method prints a message indicating that only training will be performed.

If no checkpoint saving options are set (save\_best, save\_every\_n\_epochs, and save\_last\_epoch are all None), the method prints a message indicating that no checkpoints will be saved.

The method takes in three parameters - num\_epochs (integer), multilabel (boolean), and threshold (float) - that control the number of training epochs, the type of label encoding (multi-class or multi-label), and the threshold for binary classification (if applicable).

The method then loops over the specified number of epochs and performs the following steps:

* Starts a timer using datetime.now() to measure the time taken to train the epoch
* Calls the \_run\_epoch method with training=True to perform training on the train loader
* Calculates the time taken to train the epoch by subtracting the start time from the current time (datetime.now() - t0)
* Appends the train loss to the train\_losses list
* If self.train\_metric is not None, appends the train metric to the train\_metrics list
* If a validation loader is set, performs the following steps:
  + Starts a timer using datetime.now() to measure the time taken to validate the epoch
  + Calls the \_run\_epoch method with training=False to perform validation on the validation loader
  + Calculates the time taken to validate the epoch by subtracting the start time from the current time (datetime.now() - t0)
  + Appends the validation loss to the val\_losses list
  + If self.val\_metric is not None, appends the validation metric to the val\_metrics list
  + If self.early\_stopping\_step is callable, calls it with the validation loss. If self.early\_stop is True, prints a message indicating that early stopping has been triggered and breaks out of the epoch loop.
  + If the best\_score is None or the validation loss is lower than self.best\_score, sets self.best\_score to the validation loss, sets self.best\_epoch to the current epoch number plus one, and if self.save\_best is True, calls self.save\_checkpoint with a suffix of \_best to save the current checkpoint as the best checkpoint.
* If self.save\_every\_n\_epochs is not None and the current epoch number plus one is a multiple of self.save\_every\_n\_epochs, calls self.save\_checkpoint with a suffix of \_epoch\_{self.total\_epochs + 1} to save the current checkpoint for this epoch.
* If self.save\_last\_epoch is True, calls self.save\_checkpoint with a suffix of \_last to save the current checkpoint as the last checkpoint.
* Calls \_log\_print\_epoch\_loss\_metric to log and print the train and validation losses and metrics for the epoch.
* Increments self.total\_epochs by one.

**Improvements:**

* Instead of using the assert statement to check if self.train\_loader is not None, it might be better to use an if statement and raise an exception if it is None. This will make the error message more informative and allow the user to handle the error more gracefully.
* The if all(value is None for value in [self.save\_best, self.save\_every\_n\_epochs, self.save\_last\_epoch]) statement can be simplified to if not any([self.save\_best, self.save\_every\_n\_epochs, self.save\_last\_epoch]):. This will make the code more readable and easier to understand.
* In the line self.best\_epoch = self.total\_epochs + 1, it might be better to use the epoch variable instead of self.total\_epochs + 1. This will make the code more consistent and easier to understand.
* It might be better to move the if callable(self.early\_stopping\_step) statement outside of the loop and check it only once. This will improve the performance of the code.
* In the line self.\_log\_print\_epoch\_loss\_metric(train\_loss, train\_metric, val\_loss, val\_metric, epoch, num\_epochs, dt\_train, dt\_valid), it might be better to pass the arguments as a dictionary instead of positional arguments. This will make the code more readable and easier to understand.

**Code -**

def plot\_history(self):

epochs = range(1, len(self.train\_losses) + 1)

plt.figure()

plt.plot(epochs, self.train\_losses, label="Train")

plt.plot(epochs, self.val\_losses, label="Validation")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.show()

if self.train\_metrics[0] is not None:

plt.figure()

plt.plot(epochs, self.train\_metrics, label="Train")

plt.plot(epochs, self.val\_metrics, label="Validation")

plt.xlabel("Epochs")

plt.ylabel("Metric")

plt.legend()

plt.show()

**Explanation:**

The plot\_history method of a Trainer class is responsible for plotting the training and validation losses and metrics (if available) over the training epochs. The method first initializes the epochs range as a list of integers from 1 to the length of the train\_losses list. It then creates a new figure using plt.figure(), plots the training losses using plt.plot(), and labels the plot axes and the plot line with appropriate text using plt.xlabel(), plt.ylabel(), and plt.legend(), respectively. The show() method is then called to display the plot. If the train\_metrics list is not empty, the same steps are repeated for the metric plot. Overall, this method provides a convenient way to visualize the training progress of the model during the training loop.

**Code -**

def predict(self, loader, return\_targets=False, multilabel=False, threshold=0.5):

self.model.to(self.device)

self.model.eval()

predictions = []

targets\_list = []

with torch.no\_grad():

for inputs, targets in loader:

inputs, targets = self.\_move\_inputs\_targets\_to\_gpu(

inputs, targets)

outputs = self.model(inputs)

if multilabel:

outputs = (torch.sigmoid(outputs) > threshold).float()

else:

outputs = torch.argmax(outputs, dim=1)

predictions.append(outputs.cpu())

targets\_list.append(targets.cpu())

predictions\_tensor = torch.cat(predictions, dim=0)

targets\_tensor = torch.cat(targets\_list, dim=0)

if return\_targets:

return predictions\_tensor, targets\_tensor,

else:

return predictions\_tensor

**Explanation:**

The predict() method takes in a data loader and some optional arguments, and returns the predictions made by the trained model on the given data loader.

The method first moves the model to the device specified during initialization and sets the model to evaluation mode. It then loops through the data loader and makes predictions for each batch of inputs. The method then appends the predictions and targets to the predictions and targets\_list lists, respectively.

If multilabel is set to True, the method applies a threshold to the output probabilities using sigmoid function and converts them to binary predictions. Otherwise, it computes the class indices corresponding to the highest probability predictions using argmax() method.

Finally, the method concatenates the predictions and targets list along the batch dimension to create the output tensors. If return\_targets is True, the method returns both the predictions and targets tensors. Otherwise, it returns only the predictions tensor.

Overall, the predict() method provides a convenient way to generate predictions on a data loader using a trained model.

**Improvements:**

1. Consider adding some input validation code to the beginning of the method. For example, you can check that the loader argument is not None, and that multilabel is a boolean.
2. You may want to handle the case where self.train\_metric or self.val\_metric is None, since these attributes are not always set by the user. For example, you can add an if statement to check if the corresponding list is empty before plotting the metrics.

**Code -**

def sanity\_check(self, num\_classes):

for inputs, targets in self.train\_loader:

inputs, targets = self.\_move\_inputs\_targets\_to\_gpu(

inputs, targets)

self.model.eval()

# Forward pass

outputs = self.model(inputs)

loss = self.criterion(outputs, targets)

print(f'Actual loss: {loss}')

break

print(f'Expected Theoretical loss: {np.log(num\_classes)}')

self.model.train()

**Explanation:**

This method performs a sanity check to ensure that the model's loss is close to the theoretical value for a given number of classes. Here are the steps taken by the method:

1. It loops over the batches in the train loader and retrieves the first batch of data and targets.
2. It moves the data and targets to the GPU if available.
3. It sets the model to evaluation mode.
4. It passes the inputs to the model and retrieves the outputs.
5. It calculates the loss using the criterion and the retrieved outputs and targets.
6. It prints the actual loss.
7. It prints the expected theoretical loss, which is calculated as the natural logarithm of the number of classes.
8. It sets the model back to train mode.

**Improvements:**

1. You could add more checks to ensure that the model outputs and targets have the correct shapes and types.

**Code -**

@staticmethod

def set\_seed(seed=42):

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

np.random.seed(seed)

random.seed(seed)

**Explanation:**

This method sets the random seed for the PyTorch, NumPy, and Python's random module to ensure reproducibility of results.

* torch.backends.cudnn.deterministic = True ensures that the convolution algorithms used by PyTorch's backend (cuDNN) produce deterministic results.
* torch.backends.cudnn.benchmark = False disables the cuDNN benchmark mode, which dynamically selects the best convolution algorithm for a given input size, resulting in non-deterministic behavior. Setting it to False ensures that the same algorithm is used for all inputs.
* torch.manual\_seed(seed) sets the random seed for the CPU version of PyTorch.
* torch.cuda.manual\_seed(seed) sets the random seed for the GPU version of PyTorch.
* np.random.seed(seed) sets the random seed for NumPy.
* random.seed(seed) sets the random seed for Python's built-in random module.

Setting the same seed ensures that the same sequence of random numbers is generated every time the code is run, which is useful for debugging and ensuring reproducibility.