modelling_f1_mlp

June 11, 2023

[ ]:
```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score, recall_score, accuracy_score

from tqdm import tqdm
```

[ ]:
```
device = ('cuda'
          if torch.cuda.is_available()
          else 'mps'
          if torch.backends.mps.is_available()
          else 'cpu')

print(f'Using {device} device')
```

```
Using cuda device
```

[ ]:
```
#torch.set_default_device(device)
```

[ ]:
```
df = pd.read_csv('F1Stats_withWeather_filtered.csv', index_col=0)
df = df.sort_values(by=['date'])

df['rain'] = df['rain'].astype(int)

df
```

[ ]:

	year	date	GP_name	raceId	circuitId	driverId	\
0	2011	2011-03-27	Australian Grand Prix	841	1	1	\
18	2011	2011-03-27	Australian Grand Prix	841	1	17	
17	2011	2011-03-27	Australian Grand Prix	841	1	2	
16	2011	2011-03-27	Australian Grand Prix	841	1	3	
15	2011	2011-03-27	Australian Grand Prix	841	1	4	
…	…	…	…	…	…	…	
4680	2022	2022-11-20	Abu Dhabi Grand Prix	1096	24	844	
4679	2022	2022-11-20	Abu Dhabi Grand Prix	1096	24	842	
4697	2022	2022-11-20	Abu Dhabi Grand Prix	1096	24	832	
4687	2022	2022-11-20	Abu Dhabi Grand Prix	1096	24	839	
4698	2022	2022-11-20	Abu Dhabi Grand Prix	1096	24	855	

	constructorId	driver	constructor	position	…	\
0	1	Lewis Hamilton	McLaren	2	…	\
18	9	Mark Webber	Red Bull	5	…	
17	4	Nick Heidfeld	Renault	12	…	
16	131	Nico Rosberg	Mercedes	17	…	
15	6	Fernando Alonso	Ferrari	4	…	
…	…	…	…	…	…	
4680	6	Charles Leclerc	Ferrari	2	…	
4679	213	Pierre Gasly	AlphaTauri	14	…	
4697	6	Carlos Sainz	Ferrari	4	…	
4687	214	Esteban Ocon	Alpine F1 Team	7	…	
4698	51	Guanyu Zhou	Alfa Romeo	12	…	

	pitDuration	GP_name.1	constructor_dnf	driver_dnf	\
0	0.000000	Australian Grand Prix	0	0	\
18	0.000000	Australian Grand Prix	0	0	
17	0.000000	Australian Grand Prix	0	0	
16	0.000000	Australian Grand Prix	0	1	
15	0.000000	Australian Grand Prix	0	0	
…	…	…	…	…	
4680	25.458667	Abu Dhabi Grand Prix	0	0	
4679	23.265667	Abu Dhabi Grand Prix	0	0	
4697	24.168333	Abu Dhabi Grand Prix	0	0	
4687	24.156500	Abu Dhabi Grand Prix	0	0	
4698	24.721500	Abu Dhabi Grand Prix	0	0	

	constructor_reliability	driver_confidence	weather	rain	snow	cloudy
0	0.000000	0.000000	12.2	0	False	16
18	1.000000	0.000000	12.2	0	False	16
17	1.000000	0.000000	12.2	0	False	16
16	1.000000	0.000000	12.2	0	False	16
15	1.000000	0.000000	12.2	0	False	16
…	…	…	…	…	…	…
4680	0.975610	1.000000	31.3	0	False	4

```
4679                0.929412          0.954545     31.3     0  False      4
4697                0.975904          1.000000     31.3     0  False      4
4687                0.922078          1.000000     31.3     0  False      4
4698                0.908046          1.000000     31.3     0  False      4

[4699 rows x 28 columns]
```

[ ]:
```
df.dtypes
```

[ ]:
```
year                       int64
date                      object
GP_name                   object
raceId                     int64
circuitId                  int64
driverId                   int64
constructorId              int64
driver                    object
constructor               object
position                   int64
quali_pos                  int64
wins                     float64
points                   float64
constructorPoints        float64
constructorWins          float64
driver_home               object
statusId                   int64
pitStops                 float64
pitDuration              float64
GP_name.1                 object
constructor_dnf            int64
driver_dnf                 int64
constructor_reliability  float64
driver_confidence        float64
weather                  float64
rain                       int64
snow                        bool
cloudy                     int64
dtype: object
```

0.1 Create our training and testing data

[ ]:
```
def position_index_5_classes(x):
    if x < 4:
        return 0 # 1
    if x < 8:
        return 1 # 2
    if x < 12:
```

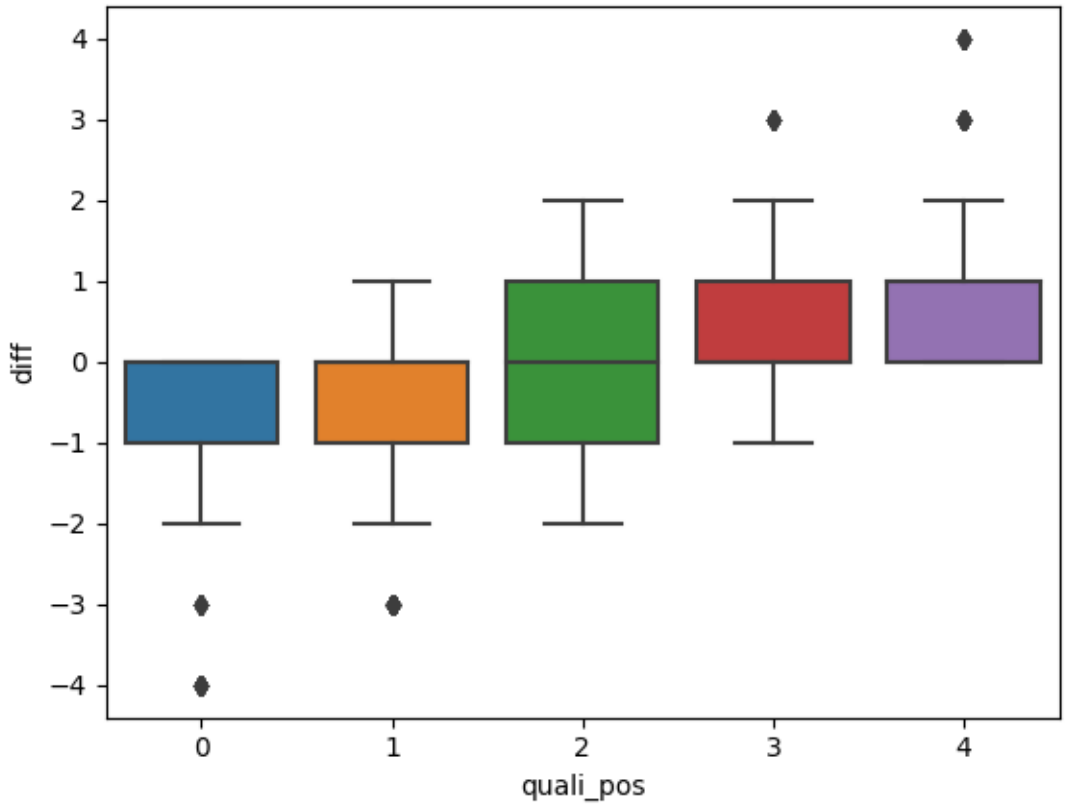
```
        return 2 # 3
    if x < 16:
        return 3 # 4
    else:
        return 4 # 5

def position_index_20_classes(x):
    if x >= 20:
        return 19
    else:
        return x - 1
```

[ ]:
```
df_3_classes = df.copy()
df_3_classes['position'] = df_3_classes['position'].
 ↪apply(position_index_5_classes)
df_3_classes['quali_pos'] = df_3_classes['quali_pos'].
 ↪apply(position_index_5_classes)

df_3_classes['diff'] = df_3_classes['quali_pos'] - df_3_classes['position']
```

[ ]:
```
print(f"Value counts in percentage:\n {df_3_classes['diff'].
 ↪value_counts(normalize=True)}")
```

```
Value counts in percentage:
 diff
 0    0.465205
 1    0.239838
-1    0.133645
 2    0.068525
-2    0.048095
-3    0.021707
 3    0.012769
-4    0.007661
 4    0.002554
Name: proportion, dtype: float64
```

[ ]:
```
sns.boxplot(df_3_classes, x='quali_pos', y='diff')
plt.show()
```



[ ]:
```
def get_positions_of_other_drivers_in_race(df, race_id, driver_id):
    # Get all driver IDs in the specific race
    competing_driver_ids = df[df['raceId'] == race_id]['driverId'].unique()

    # Remove the specific driver ID from the list
    competing_driver_ids = competing_driver_ids[competing_driver_ids !=␣
↪driver_id]

    # Select the columns for the competing driver IDs
    competing_columns = [0 for i in range(20)]

    for competing in competing_driver_ids:
        quali_pos = df[df['driverId'] == competing]['quali_pos'].values[0]
        encoded_value = str(competing) + '|' + str(quali_pos)

        df.loc[(df['raceId'] == race_id) & (df['driverId'] == driver_id),␣
↪f'competing_{quali_pos+1}'] = encoded_value
```

[ ]:
```
# Create an empty DataFrame with 20 columns
compet_columns = [f'competing_{i}' for i in range(1, 21)]
compet_df = pd.DataFrame(columns=compet_columns)

df_3_classes_compet = pd.concat([df_3_classes, compet_df], axis=1)

unique_race_ids = df_3_classes_compet['raceId'].unique()
count = 0

for i in unique_race_ids:
    for j in df[df['raceId'] == i]['driverId'].unique():
        count += 1
        get_positions_of_other_drivers_in_race(df_3_classes_compet, i, j)
```

[ ]:
```
from sklearn.preprocessing import LabelEncoder

# Perform label encoding
df_3_classes_compet_2 = df_3_classes_compet.copy()
for quali_pos in range(1, 21):
    encoder = LabelEncoder()
    df_3_classes_compet_2[f'competing_{quali_pos}'] = encoder.
 ↪fit_transform(df_3_classes_compet_2[f'competing_{quali_pos}'])
```

[ ]:
```
x_variables_names =␣
 ↪['constructorId','driverId','quali_pos','wins','points','constructorPoints','constructorWin
x_variables_names_compet = x_variables_names + [f'competing_{i}' for i in␣
 ↪range(1, 21)]
```

[ ]:
```
X = df_3_classes[x_variables_names]
y = df_3_classes['position']
```

[ ]:
```
print(X.shape)
print(y.shape)
```

```
(4699, 11)
(4699,)
```

[ ]:
```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,␣
 ↪shuffle=False)

X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.int32)

X_val_tensor = torch.tensor(X_val.values, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val.values, dtype=torch.int32)
```

change the dataset to grab for the lookback the rows of the same driver in past races, maybe add

the information of the position of all other drivers in a race to the row of a driver

[ ]:
```
# class LSTMDataset(Dataset):
#     def __init__(self, data, target, lookback):
#         self.data = data
#         self.target = target
#         self.lookback = lookback

#     def __len__(self):
#         return len(self.data) - self.lookback

#     def __getitem__(self, idx):
#         data_items = self.data[idx:idx+self.lookback]
#         target_item = self.target[idx+self.lookback - 1] # value to be␣
 ↪predicted, we predict the last value of the lookback window, not the next␣
 ↪one
#         return data_items, target_item

class LSTMDataset(Dataset):
    def __init__(self, data, x_variables, y_variables, lookback):
        self.data = data.sort_values(by=['date']).copy()
        self.data = self.data.reset_index(drop=True)
        self.lookback = lookback

        #Convert column names to numeric indices
        self.x_indices = [data.columns.get_loc(col) for col in x_variables]
        self.y_indices = [data.columns.get_loc(col) for col in y_variables]

        target_tensor = torch.tensor(target_items[0], dtype=torch.float32)

        return data_tensor, target_tensor
```

[ ]:
```
train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = torch.utils.data.TensorDataset(X_val_tensor, y_val_tensor)
# train_dataset = LSTMDataset(X_train_tensor, y_train_tensor, 100)
# test_dataset = LSTMDataset(X_val_tensor, y_val_tensor, 100)

# lstm
# train_data, test_data = train_test_split(df_3_classes_compet_2, test_size=0.
 ↪3, shuffle=False)
# train_dataset = LSTMDataset(train_data, x_variables_names_compet,␣
 ↪['position'], 4) # train_data
# test_dataset = LSTMDataset(test_data, x_variables_names_compet, ['position'],␣
 ↪4) # test_data

train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

[ ]:
```
class MLP_driver_constructor(nn.Module):
    def __init__(self, inputs, outputs, hidden_layers):
        # hidden layers = [32, 64, 64, 64, 128, 128, 64, 64]
        super(MLP_driver_constructor, self).__init__()

        self.hidden_layers = nn.ModuleList([])

        prev_input = inputs

        for i in hidden_layers:
            self.hidden_layers.append(nn.Linear(prev_input, i))
            self.hidden_layers.append(nn.GELU())
            prev_input = i

        self.output_layer = nn.Linear(prev_input, outputs)

    def forward(self, x):
        for i in self.hidden_layers:
            x = i(x)
        x = self.output_layer(x) # logits

        return x
```

[ ]:
```
class LSTM_driver_constructor(nn.Module):
    def __init__(self, inputs, outputs, lstm_layers, lstm_size, hidden_layers):
        super(LSTM_driver_constructor, self).__init__()

        self.lstm_layers = lstm_layers

        prev_input = lstm_size

        self.lstm = nn.LSTM(inputs, prev_input, self.lstm_layers,␣
↪batch_first=True, dropout=0.2, bidirectional=False)

        self.hidden_layers = nn.ModuleList([])

        for i in hidden_layers[1:]:
            self.hidden_layers.append(nn.Linear(prev_input, i))
            self.hidden_layers.append(nn.GELU())
            prev_input = i

        self.output_layer = nn.Linear(prev_input, outputs)

    def forward(self, x):
        # Initialize hidden and cell states
        # h0 = torch.zeros(self.lstm_layers, x.size(0), self.hidden_size).to(x.
↪device)
```

```
        # c0 = torch.zeros(self.lstm_layers, x.size(0), self.hidden_size).to(x.
↪device)

        # Forward pass through LSTM layer
        out, _ = self.lstm(x)

        # Reshape output and pass through fully connected layer
        out = out[:, -1, :]

        for i in self.hidden_layers:
            out = i(out)

        out = self.output_layer(out) # logits

        return out
```

[ ]:
```
def l1_regularizer(model, lambda_l1=0.01, device='cpu'):
    l1_reg = torch.tensor(0.).to(device)
    for name, param in model.named_parameters():
        if 'weight' in name:
            l1_reg += lambda_l1 * param.abs().sum()
    return l1_reg

def l2_regularizer(model, lambda_l2=0.01, device='cpu'):
    l2_reg = torch.tensor(0.).to(device)
    for name, param in model.named_parameters():
        if 'weight' in name:
            l2_reg += lambda_l2 * param.pow(2).sum()
    return l2_reg
```

[ ]:
```
class EarlyStopper:
    def __init__(self, model, patience=1, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf
        self.model = model

    def early_stop(self, validation_loss):
        if validation_loss < self.min_validation_loss:
            self.min_validation_loss = validation_loss
            self.counter = 0

            # Save the entire model
            torch.save(self.model, 'model_mlp_5_classes.pt')
        elif validation_loss > (self.min_validation_loss + self.min_delta):
            self.counter += 1
```

```
            if self.counter >= self.patience:
                return True
        return False

class Trainer:
    def __init__(self, model, train_dataloader, val_dataloader, criterion,␣
 ↪optimizer, device, num_epochs=100):
        self.model = model
        self.train_dataloader = train_dataloader
        self.val_dataloader = val_dataloader
        self.criterion = criterion
        self.optimizer = optimizer
        self.num_epochs = num_epochs
        self.device = device
        self.history_loss = []
        self.history_val_loss = []
        self.history_f1 = []
        self.history_recall = []
        self.early_stopper = EarlyStopper(self.model, patience=20, min_delta=0.
 ↪2)

        self.lambda_l2 = 0.001


    def to_device(self, data):
        if isinstance(data, (list, tuple)):
            return [self.to_device(x) for x in data]
        return data.to(self.device)

    def train_batch(self, inputs, labels):
        inputs = self.to_device(inputs)
        labels = self.to_device(labels)

        self.model.train()
        self.optimizer.zero_grad()
        outputs = self.model(inputs)
        loss = self.criterion(outputs, labels.long())

        # l2 regularization
        l2_reg = l2_regularizer(self.model, self.lambda_l2, self.device)
        loss += l2_reg

        loss.backward()
        self.optimizer.step()
        return loss.item(), outputs

    def validate_batch(self, inputs, labels):
```

```
        inputs = self.to_device(inputs)
        labels = self.to_device(labels)

        self.model.eval()
        with torch.no_grad():
            val_outputs = self.model(inputs)
            loss = self.criterion(val_outputs, labels.long())

            # l2 regularization
            l2_reg = l2_regularizer(self.model, self.lambda_l2, self.device)
            loss += l2_reg

            return loss.item(), val_outputs

    def train_epoch(self):
        total_loss = 0
        total_correct = 0
        total_samples = 0
        y_true = []
        y_pred = []

        for inputs_batch, labels_batch in self.train_dataloader:
            loss, outputs = self.train_batch(inputs_batch, labels_batch)
            total_loss += loss

            _, predicted = torch.max(outputs, dim=1)
            total_correct += (predicted.detach().cpu() == labels_batch.detach().
↪cpu()).sum().item()
            total_samples += labels_batch.size(0)
            y_true.extend(labels_batch.tolist())
            y_pred.extend(predicted.tolist())

        avg_loss = total_loss / len(self.train_dataloader)
        accuracy = total_correct / total_samples
        f1 = f1_score(y_true, y_pred, average='weighted')
        recall = recall_score(y_true, y_pred, average='weighted')

        return avg_loss, accuracy, f1, recall

    def validate_epoch(self):
        val_loss = 0
        val_correct = 0
        val_samples = 0
        y_true = []
        y_pred = []

        for val_inputs, val_labels in self.val_dataloader:
```

```
            loss, val_outputs = self.validate_batch(val_inputs, val_labels)
            val_loss += loss

            _, val_predicted = torch.max(val_outputs, dim=1)
            val_correct += (val_predicted.detach().cpu() == val_labels.detach().
↪cpu()).sum().item()
            val_samples += val_labels.size(0)
            y_true.extend(val_labels.tolist())
            y_pred.extend(val_predicted.tolist())

        avg_val_loss = val_loss / len(self.val_dataloader)
        val_accuracy = val_correct / val_samples
        val_f1 = f1_score(y_true, y_pred, average='weighted')
        val_recall = recall_score(y_true, y_pred, average='weighted')

        return avg_val_loss, val_accuracy, val_f1, val_recall

    def train(self):
        self.model.to(self.device)

        with tqdm(total=self.num_epochs, desc="Training", unit="epoch") as␣
↪progress_bar:
            for epoch in range(self.num_epochs):
                # Training phase
                avg_loss, accuracy, f1, recall = self.train_epoch()

                # Validation phase
                avg_val_loss, val_accuracy, val_f1, val_recall = self.
↪validate_epoch()

                # Append the loss, F1 score, and recall to history
                self.history_loss.append(avg_loss)
                self.history_val_loss.append(avg_val_loss)
                self.history_f1.append(f1)
                self.history_recall.append(recall)

                # Update the progress bar
                progress_bar.set_postfix(loss=avg_loss, accuracy=accuracy,␣
↪f1=f1, recall=recall,
                                         val_loss=avg_val_loss,␣
↪val_accuracy=val_accuracy, val_f1=val_f1, val_recall=val_recall)
                progress_bar.update(1)

                if self.early_stopper.early_stop(avg_val_loss):
                    print("Early stopping")
                    break
```

```
    def predict(self, dataloader):
        self.model.to(self.device)
        self.model.eval()
        predictions = []
        true_labels = []

        with torch.no_grad():
            for val_inputs, val_labels, in dataloader:
                val_inputs = self.to_device(val_inputs)
                outputs = self.model(val_inputs)
                _, predicted_labels = torch.max(outputs, dim=1)
                predicted_labels = predicted_labels.cpu().numpy()
                val_labels = val_labels.cpu().numpy()
                predictions.extend(predicted_labels.tolist())
                true_labels.extend(val_labels.tolist())

        return predictions, true_labels
```

[ ]:
```
# Create the model instance
model = MLP_driver_constructor(len(x_variables_names), 5, [64, 64, 64, 128,␣
 ↪128, 256, 128, 64, 64, 32])

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-3)
```

[ ]:
```
model_trainer = Trainer(model, train_dataloader, test_dataloader, criterion,␣
 ↪optimizer, device, num_epochs=1000)
```

[ ]:
```
model_trainer.train()
```

```
Training:   0%|          | 2/1000 [00:01<09:16,  1.79epoch/s, accuracy=0.387,
f1=0.382, loss=1.44, recall=0.387, val_accuracy=0.33, val_f1=0.294,
val_loss=1.6, val_recall=0.33]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[174], line 1
----> 1 model_trainer.train()

Cell In[171], line 131, in Trainer.train(self)
    128 with tqdm(total=self.num_epochs, desc="Training", unit="epoch") as␣
 ↪progress_bar:
    129     for epoch in range(self.num_epochs):
    130         # Training phase
--> 131         avg_loss, accuracy, f1, recall = self.train_epoch()
    133         # Validation phase
```

```
    134         avg_val_loss, val_accuracy, val_f1, val_recall = self.
 ↪validate_epoch()

Cell In[171], line 85, in Trainer.train_epoch(self)
     82 y_pred = []
     84 for inputs_batch, labels_batch in self.train_dataloader:
---> 85     loss, outputs = self.train_batch(inputs_batch, labels_batch)
     86     total_loss += loss
     88     _, predicted = torch.max(outputs, dim=1)

Cell In[171], line 59, in Trainer.train_batch(self, inputs, labels)
     56 loss += l2_reg
     58 loss.backward()
---> 59 self.optimizer.step()
     60 return loss.item(), outputs

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/
 ↪optimizer.py:280, in Optimizer.profile_hook_step.<locals>.wrapper(*args,␣
 ↪**kwargs)
    276         else:
    277             raise RuntimeError(f"{func} must return None or a tuple of␣
 ↪(new_args, new_kwargs),"
    278                                f"but got {result}.")
--> 280 out = func(*args, **kwargs)
    281 self._optimizer_step_code()
    283 # call optimizer step post hooks

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/
 ↪optimizer.py:33, in _use_grad_for_differentiable.<locals>._use_grad(self,␣
 ↪*args, **kwargs)
     31 try:
     32     torch.set_grad_enabled(self.defaults['differentiable'])
---> 33     ret = func(self, *args, **kwargs)
     34 finally:
     35     torch.set_grad_enabled(prev_grad)

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/adamw
 ↪py:171, in AdamW.step(self, closure)
    158     beta1, beta2 = group["betas"]
    160     self._init_group(
    161         group,
    162         params_with_grad,
   (…)
    168         state_steps,
    169     )
--> 171     adamw(
    172         params_with_grad,
    173         grads,
```

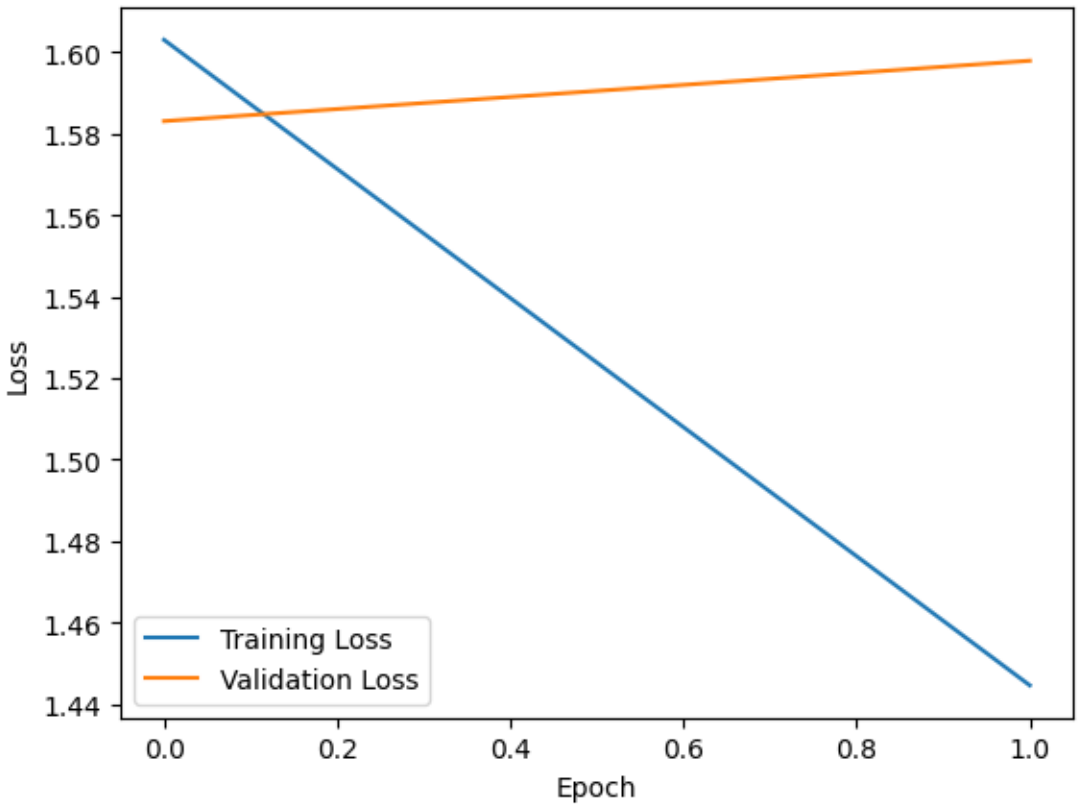
```
    174         exp_avgs,
    175         exp_avg_sqs,
    176         max_exp_avg_sqs,
    177         state_steps,
    178         amsgrad=amsgrad,
    179         beta1=beta1,
    180         beta2=beta2,
    181         lr=group["lr"],
    182         weight_decay=group["weight_decay"],
    183         eps=group["eps"],
    184         maximize=group["maximize"],
    185         foreach=group["foreach"],
    186         capturable=group["capturable"],
    187         differentiable=group["differentiable"],
    188         fused=group["fused"],
    189         grad_scale=getattr(self, "grad_scale", None),
    190         found_inf=getattr(self, "found_inf", None),
    191     )
    193 return loss

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/adamw
 ↪py:321, in adamw(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,␣
 ↪state_steps, foreach, capturable, differentiable, fused, grad_scale,␣
 ↪found_inf, amsgrad, beta1, beta2, lr, weight_decay, eps, maximize)
    318 else:
    319     func = _single_tensor_adamw
--> 321 func(
    322     params,
    323     grads,
    324     exp_avgs,
    325     exp_avg_sqs,
    326     max_exp_avg_sqs,
    327     state_steps,
    328     amsgrad=amsgrad,
    329     beta1=beta1,
    330     beta2=beta2,
    331     lr=lr,
    332     weight_decay=weight_decay,
    333     eps=eps,
    334     maximize=maximize,
    335     capturable=capturable,
    336     differentiable=differentiable,
    337     grad_scale=grad_scale,
    338     found_inf=found_inf,
    339 )
```

```
File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/adamw
 ↪py:553, in _multi_tensor_adamw(params, grads, exp_avgs, exp_avg_sqs,␣
 ↪max_exp_avg_sqs, state_steps, grad_scale, found_inf, amsgrad, beta1, beta2,␣
 ↪lr, weight_decay, eps, maximize, capturable, differentiable)
    549 bias_correction2 = [1 - beta2 ** _get_value(step) for step in␣
 ↪device_state_steps]
    551 step_size = _stack_if_compiling([(lr / bc) * -1 for bc in␣
 ↪bias_correction1])
--> 553 bias_correction2_sqrt = [_dispatch_sqrt(bc) for bc in bias_correction2]
    555 if amsgrad:
    556     # Maintains the maximum of all 2nd moment running avg. till now
    557     torch._foreach_maximum_(device_max_exp_avg_sqs, device_exp_avg_sqs)

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/adamw
 ↪py:553, in <listcomp>(.0)
    549 bias_correction2 = [1 - beta2 ** _get_value(step) for step in␣
 ↪device_state_steps]
    551 step_size = _stack_if_compiling([(lr / bc) * -1 for bc in␣
 ↪bias_correction1])
--> 553 bias_correction2_sqrt = [_dispatch_sqrt(bc) for bc in bias_correction2]
    555 if amsgrad:
    556     # Maintains the maximum of all 2nd moment running avg. till now
    557     torch._foreach_maximum_(device_max_exp_avg_sqs, device_exp_avg_sqs)

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/optim/
 ↪optimizer.py:53, in _dispatch_sqrt(x)
     52 def _dispatch_sqrt(x: float):  # float annotation is needed because of␣
 ↪torchscript type inference
---> 53     if not torch.jit.is_scripting() and isinstance(x, torch.Tensor):
     54         return x.sqrt()
     55     else:

File ~/miniconda3/envs/torch-gpu/lib/python3.10/site-packages/torch/
 ↪_jit_internal.py:1102, in is_scripting()
   1098 for i in range(2, 7):
   1099     globals()[f"BroadcastingList{i}"] = BroadcastingList1
-> 1102 def is_scripting() -> bool:
   1103     r"""
   1104     Function that returns True when in compilation and False otherwise.␣
 ↪This
   1105     is useful especially with the @unused decorator to leave code in yo r
   (…)
   1119                 return unsupported_linear_op(x)
   1120     """
   1121     return False

KeyboardInterrupt:
```

[ ]:
```
def print_confusion_matrix(y_true, y_pred, classes):
    cm = confusion_matrix(y_true, y_pred)
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure(figsize=(20, 15))
    sns.heatmap(cm_normalized, annot=True, fmt=".2f", cmap="Blues", ␣
 ↪xticklabels=classes, yticklabels=classes, annot_kws={"size": 15})
    plt.xlabel("Predicted", fontsize=15)
    plt.ylabel("Actual", fontsize=15)
    plt.title("Normalized Confusion Matrix", fontsize=20)
```

[ ]:
```
def plot_loss_curves(history_loss, history_val_loss):
    sns.lineplot(x=range(len(history_loss)), y=history_loss, label='Training␣
 ↪Loss')
    sns.lineplot(x=range(len(history_val_loss)), y=history_val_loss,␣
 ↪label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

[ ]:
```
plot_loss_curves(model_trainer.history_loss, model_trainer.history_val_loss)
```



[ ]:
```
model_best = torch.load("model_mlp_5_classes_v3.pt")
model_best_trainer = Trainer(model_best, train_dataloader, test_dataloader,␣
 ↪criterion, optimizer, device, num_epochs=500)
```

[ ]:
```
test_predictions, test_true = model_best_trainer.predict(test_dataloader)

print(f"f1 score: {f1_score(test_true, test_predictions, average='weighted')}")
print(f"recall: {recall_score(test_true, test_predictions,␣
 ↪average='weighted')}")
print(f"accuracy: {accuracy_score(test_true, test_predictions)}")

print_confusion_matrix(test_true, test_predictions, [1, 2, 3, 4, 5])
```

```
f1 score: 0.4651658337675834
recall: 0.47304964539007094
accuracy: 0.47304964539007094
```