HW 5 – Multi-class Classification Using Dense Embeddings

Data Pre-Processing:

No Preprocessing is required. You are provided with cleaned text in column : Cleaned_text

Split the Data to create a train, test, and validation split.

- Use 80% observations for the train set, 10% for the validation set, and 10 % for the test set.

Hyperparameters and Network:

- For creating vocab use a min frequency of 5
- Your Network should have the following layers

Embeddinglayer → Hidden_Layer1 → Dropout_Layer1 → BatchNorn_Layer1 → Hidden_Layer2 → DropoutLayer2 → BatchNorm_Layer2

- Embedding dimension - 300
- Neurons in Hidden_Layer 1 - 200
- Neurons in Hidden_Layer 2- 100
- Dropout probability for Dropout_Layer1 → 0.5
- Dropout probability for Dropout_Layer2 → 0.5
- Loss Function → CrossEntropy
- Batch_size - 256
- Learning_rate - 0.2
- Number of epochs = 20
- Use early stopping with the patience of 5
- Weight_decay = 0
- Activation function for hidden layer = ReLU
- Optimizer - SGD (Make sure you use SGD and not Adam)

Import libraries

```
%%capture
!pip install wandb --upgrade -qq
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
     Mounted at /content/drive
```

```
# Import random function

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchtext.vocab import  vocab

import wandb
import spacy
#import custom_preprocessor as cp

import random
from datetime import datetime
import numpy as np
from pathlib import Path
import pandas as pd
import joblib
from collections import Counter

from pathlib import Path

from sklearn.model_selection import train_test_split
```

```
# Fix seed value
SEED = 2345
random.seed(SEED)
```

```
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

data_folder = Path('/content/drive/MyDrive/Lec10/HW5/Data')


save_model_folder = Path('/content/drive/MyDrive/NLP/models')
```

We will be using W&B for visualization.

```
# Login to W&B
wandb.login()
```

```
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
True
```

Train/Test/Valid Dataset for Preprocessed data

```
cleaned_file = data_folder/ 'multiclass_hw_cleaned.csv'
```

```
# Preprocessed data
cleaned_data = pd.read_csv(cleaned_file, index_col=0)
```

```
print(f'Shape of cleaned data set is : {cleaned_data.shape}')
```

```
Shape of cleaned data set is : (188878, 5)
```

```
cleaned_data.head()
```

	Title	Body	cleaned_text	Tags	Tag_Number_final
0	detail disclosure indicator on UIButton	<p>Is there a simple way to place a detail dis...	detail disclosure indicator uibutton simple wa...	iphone	8
1	hello world fails to show up in emulator	<p>I followed Hello World tutorial exactly. E...	hello world fail emulator follow hello world t...	android	4
2	Why is JSHint throwing a "possible strict viol...	<p>Trying to validate some Javascript in JsHin...	jshint throw possible strict violation line tr...	javascript	3
	Programmatically Make Bound	<p>I'm trying to make a data bound	programmatically bound column		

```
# We are interested in cleaned_text and Tag_Number_final columns
X, y = cleaned_data['cleaned_text'].values, cleaned_data['Tag_Number_final'].values
```

```
 # 80% observations for  train, 10%  for validation, and 10 % for test

 # Step1 - Split entire dataset into 90% train and 10% test
 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
 # Step1 - Split 90% train further into 10% test. So finally remaining is 80% train
 X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.1, random_state=42)
```

Custom Dataset Class

```
class CustomDataset(torch.utils.data.Dataset):

    def __init__(self, X, y):
        self.X = np.array(X)
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        text = self.X[idx]
        labels = self.y[idx]
```

```
        sample = (text, labels)

        return sample
```

```
trainset = CustomDataset(X_train,y_train)
validset = CustomDataset(X_valid,y_valid)
testset = CustomDataset(X_test,y_test)
```

```
trainset.__getitem__([11])
```

```
(array(['download audio file non direct link want implement download manager let form post method site click submit button site
       dtype=object), array([8]))
```

Create Vocab

```
def create_vocab(dataset, min_freq):
  counter = Counter()
  for (text, _) in dataset:
    counter.update(str(text).split())
  my_vocab = vocab(counter, min_freq=min_freq)
  my_vocab.insert_token('<unk>', 0)
  my_vocab.set_default_index(0)
  return my_vocab
```

```
# create vocab based on trainset using a min frequency of 5
stack_exchange_vocab = create_vocab(trainset, min_freq = 5)
```

```
len(stack_exchange_vocab)
```

```
85311
```

```
stack_exchange_vocab.get_itos()[0:5]
```

```
['<unk>', 'line', 'execute', 'javascript', 'get']
```

Collate_fn for Data Loaders

```
# Creating a lambda function objects that will be used to get the indices of words from vocab
text_pipeline = lambda x: [stack_exchange_vocab[token] for token in str(x).split()]
label_pipeline = lambda x: int(x)
```

```
'''
We know that input to the embedding layers are indices of words from the vocab.
The collate_batch() accepts batch of data and gets the indices of text from vocab and returns the same
We will include this collate_batch() in collat_fn attribute of DataLoader.
So it will create a batch of data containing indices of words and corresponding labels.
But for EmbeddingBag we need one more extra parameter, that is offset.
offsets determines the starting index position of each bag (sequence) in input.
'''
def collate_batch(batch):
    label_list, text_list, offsets = [], [], [0]
    for (_text, _label) in batch:
         label_list.append(label_pipeline(_label))
         processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
         text_list.append(processed_text)
         offsets.append(processed_text.size(0))
    label_list = torch.tensor(label_list, dtype=torch.int64)
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    return text_list, label_list, offsets
```

Check Data Loader

```
batch_size=2
check_loader= torch.utils.data.DataLoader(dataset=trainset,
                                          batch_size=batch_size,
```

```
                                        shuffle=True,
                                        collate_fn=collate_batch,
                                        num_workers=4)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 4 worker pr
  cpuset_checked))
```

```
for text, label, offsets in check_loader:
  print(label, text, offsets)
  break
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 4 worker pr
  cpuset_checked))
tensor([6, 0]) tensor([  978,   525,  5937,  4948,   661,  1237,     9,    86,   412,   337,
            8,    18,    18,   978,   525,  5937,  4948,   123,   234,    90,
          316,     0,   416,  4616,  1512,   395,  1259,  2010,   395, 14092,
           12,     0,  1259,  2010,   395,  2097,    12,     0,    63,  4231,
         5785,  4229,   163,    90,    86,    90,  1598,    66,   829,   820,
          130,   116,  4229,  1824,  3113,   179,   103,   115,   157,     8,
          534,   879,   810,  4008,   180,   559,  1258,  9883,   164,  2167,
        38507,   595,   559,     0,   559,   224,    44,   278,   278,  2058,
          278,  1467,   278,  1467, 10053,   278,  1257,   278,  2159, 20668,
            0,   278,   531,  2110,     0,   278,   141,   216,   559,   965,
           45,  1331,  9852,    40,   782,   829,  6737,  2167,   559,   108,
          649,   559,  7644,   180,  1399,   519,  2167, 38507,   830,   595,
          830,     4,  1253,    18,  2167, 38507,   180,  2375,     9,   180,
          559,  9883,   164,  2167,   325,  1436,  5902,   830,   100,   595,
         1258, 38507, 55429,   557]) tensor([ 0, 63])
```

Model

```
class MLPCustom(nn.Module):
  def __init__(self, vocab_size, hidden_sizes_list, dprobs_list, batchnorm_binary, output_dim, non_linearity):



    self.vocab_size = vocab_size


    self.hidden_sizes_list = hidden_sizes_list # hidden_sizes = [emb_dim, hidden_dim1, hidden_dim2, .....hidden_dimn] # n + 1 element
    self.dprobs_list = dprobs_list # dpropb =[prob1, prob2....probn] # n elements
    self.batchnorm_binary  = batchnorm_binary  # True or False
    self.output_dim = output_dim
    self.non_linearity = non_linearity


    super().__init__()

    # embedding_layer
    self.embedding_layer = nn.EmbeddingBag(self.vocab_size, self.hidden_sizes_list[0])

    # Creation of 3 empty lists of layers in pytorch (ModuleLists)
    # hidden layers
    self.hidden_layers = nn.ModuleList()

    # dropout layers
    self.dropout_layers = nn.ModuleList()

    # batchnorm layers
    self.batchnorm_layers = nn.ModuleList()

    for k in range(len(self.hidden_sizes_list)-1):
      self.hidden_layers.append(nn.Linear(self.hidden_sizes_list[k], self.hidden_sizes_list[k+1]))
      self.dropout_layers.append(nn.Dropout(p=self.dprobs_list[k]))

      if self.batchnorm_binary:
        self.batchnorm_layers.append(nn.BatchNorm1d(self.hidden_sizes_list[k+1], momentum=0.9))

    self.output_layer = nn.Linear(self.hidden_sizes_list[-1], self.output_dim)



  def forward(self, input, offsets):
    x = self.embedding_layer(input, offsets)
    for  k in range(len(self.hidden_sizes_list)-1):
```

```
      x =  self.non_linearity(self.hidden_layers[k](x))
      if self.batchnorm_binary:
        x = self.batchnorm_layers[k](x)
      x= self.dropout_layers[k](x)

    x = self.output_layer(x)
    # we are not using softmax function in the forward passs
    # nn.crossentropy loss (which we will use to define our loss) combines  nn.LogSoftmax() and nn.NLLLoss() in one single class
    return x
```

Training Functions

Training Epoch

```
def train(train_loader, model, optimizer, loss_function, log_batch, log_interval, grad_clipping, max_norm):

  """
  Function for training the model in each epoch
  Input: iterator for train dataset, initial weights and bias, epochs, learning rate.
  Output: final weights, bias, train loss, train accuracy
  """
  # initilalize variables as global
  # these counts will be updated every epoch
  global example_ct_train
  global batch_ct_train

  # Training Loop loop
  # Initialize train_loss at the he start of the epoch
  running_train_loss = 0
  running_train_correct = 0

  # put the model in training mode
  model.train()

  # Iterate on batches from the dataset using train_loader
  for inputs, targets, offsets in train_loader:

    # move inputs and outputs to GPUs
    inputs = inputs.to(device)
    targets = targets.to(device)
    offsets = offsets.to(device)

    # Forward pass
    output = model(inputs, offsets)
    loss = loss_function(output, targets)

    # Correct prediction
    y_pred = torch.argmax(output, dim = 1)
    correct = torch.sum(y_pred == targets)

    example_ct_train +=  len(targets)
    batch_ct_train += 1

    # set gradients to zero
    optimizer.zero_grad()

    # Backward pass
    loss.backward()

    # Gradient Clipping
    if grad_clipping:
      nn.utils.clip_grad_norm_(model.parameters(), max_norm=max_norm, norm_type=2)

    # Update parameters using their gradient
    optimizer.step()

    # Add train loss of a batch
    running_train_loss += loss.item()

    # Add Corect counts of a batch
    running_train_correct += correct

    # log batch loss and accuracy
    if log_batch:
```

```
        if ((batch_ct_train + 1) % log_interval) == 0:
          wandb.log({f"Train Batch Loss  :": loss})
          wandb.log({f"Train Batch Acc :": correct/len(targets)})


    # Calculate mean train loss for the whole dataset for a particular epoch
    train_loss = running_train_loss/len(train_loader)



    # Calculate accuracy for the whole dataset for a particular epoch
    train_acc = running_train_correct/len(train_loader.dataset)

    return train_loss, train_acc
```

Validation/Test Epoch

```
  def valid(loader, model, optimizer, loss_function, log_batch, log_interval):

    """
    Function for training the model and plotting the graph for train & valid loss vs epoch.
    Input: iterator for train dataset, initial weights and bias, epochs, learning rate, batch size.
    Output: final weights, bias and train loss and valid loss for each epoch.
    """

    # initilalize variables as global
    # these counts will be updated every epoch
    global example_ct_valid
    global batch_ct_valid

    # Validation loop
    # Initialize train_loss at the he strat of the epoch
    running_valid_loss = 0
    running_valid_correct = 0

    # put the model in evaluation mode
    model.eval()

    with torch.no_grad():
      for inputs, targets, offsets in loader:

        # move inputs and outputs to GPUs
        inputs = inputs.to(device)
        targets = targets.to(device)
        offsets = offsets.to(device)

        # Forward pass
        output = model(inputs, offsets)
        loss = loss_function(output,targets)

        # Correct Predictions
        y_pred = torch.argmax(output, dim = 1)
        correct = torch.sum(y_pred == targets)

        # count of images and batches
        example_ct_valid +=  len(targets)
        batch_ct_valid += 1

        # Add valid loss of a batch
        running_valid_loss += loss.item()

        # Add correct count for each batch
        running_valid_correct += correct

        # log batch loss and accuracy
        if log_batch:
          if ((batch_ct_valid + 1) % log_interval) == 0:
            wandb.log({f"Valid Batch Loss  :": loss})
            wandb.log({f"Valid Batch Accuracy :": correct/len(targets)})


      # Calculate mean valid loss for the whole dataset for a particular epoch
      valid_loss = running_valid_loss/len(valid_loader)

      # scheduler step
      # scheduler.step(valid_loss)
      # scheduler.step()
```

```
    # Calculate accuracy for the whole dataset for a particular epoch
    valid_acc = running_valid_correct/len(valid_loader.dataset)

  return valid_loss, valid_acc
```

Model Training Loop

```
def train_loop(train_loader, valid_loader, model, loss_function, optimizer, epochs, device, patience, early_stopping,
               file_model):

  '''
  model: specify your model for training
  criterion: loss function
  optimizer: optimizer like SGD , ADAM etc.
  train loader: function to carete batches for training data
  valid loader : function to create batches for valid data set
  file_model : specify file name for saving your model. This way we can upload the model weights from file. We will not to run model


  '''
  # Create lists to store train and valid loss at each epoch

  train_loss_history = []
  valid_loss_history = []
  train_acc_history = []
  valid_acc_history = []
  delta = 0
  best_score = None
  valid_loss_min = np.Inf
  counter_early_stop=0
  early_stop=False


  # Iterate for the given number of epochs
  for epoch in range(epochs):
    t0 = datetime.now()
    # Get train loss and accuracy for one epoch

    train_loss, train_acc = train(train_loader, model, optimizer, loss_function,
                                  wandb.config.LOG_BATCH, wandb.config.LOG_INTERVAL,
                                  wandb.config.GRAD_CLIPPING, wandb.config.MAX_NORM)
    valid_loss, valid_acc = valid(valid_loader, model, optimizer, loss_function,
                                    wandb.config.LOG_BATCH, wandb.config.LOG_INTERVAL)

    dt = datetime.now() - t0

    # Save history of the Losses and accuracy
    train_loss_history.append(train_loss)
    train_acc_history.append(train_acc)
    valid_loss_history.append(valid_loss)
    valid_acc_history.append(valid_acc)

    if early_stopping:
      score = -valid_loss
      if best_score is None:
        best_score=score
        print(f'Validation loss has decreased ({valid_loss_min:.6f} --> {valid_loss:.6f}). Saving Model...')
        torch.save(model.state_dict(), file_model)
        valid_loss_min = valid_loss

      elif score < best_score + delta:
        counter_early_stop += 1
        print(f'Early stoping counter: {counter_early_stop} out of {patience}')
        if counter_early_stop > patience:
          early_stop = True


      else:
        best_score = score
        print(f'Validation loss has decreased ({valid_loss_min:.6f} --> {valid_loss:.6f}). Saving model...')
        torch.save(model.state_dict(), file_model)
        counter_early_stop=0
        valid_loss_min = valid_loss

      if early_stop:
```

```
        print('Early Stopping')
        break

    else:

      score = -valid_loss
      if best_score is None:
        best_score=score
        print(f'Validation loss has decreased ({valid_loss_min:.6f} --> {valid_loss:.6f}). Saving Model...')
        torch.save(model.state_dict(), file_model)
        valid_loss_min = valid_loss

      elif score < best_score + delta:
        print(f'Validation loss has not decreased ({valid_loss_min:.6f} --> {valid_loss:.6f}). Not Saving Model...')

      else:
        best_score = score
        print(f'Validation loss has decreased ({valid_loss_min:.6f} --> {valid_loss:.6f}). Saving model...')
        torch.save(model.state_dict(), file_model)
        valid_loss_min = valid_loss



    # Log the train and valid loss to W&B
    wandb.log({f"Train epoch Loss :": train_loss, f"Valid epoch Loss :": valid_loss })
    wandb.log({f"Train epoch Acc :": train_acc, f"Valid epoch Acc :": valid_acc})



    # Print the train loss and accuracy for given number of epochs, batch size and number of samples
    print(f'Epoch : {epoch+1} / {epochs}')
    print(f'Time to complete {epoch+1} is {dt}')
    # print(f'Learning rate: {scheduler._last_lr[0]}')
    print(f'Train Loss: {train_loss : .4f} | Train Accuracy: {train_acc * 100 : .4f}%')
    print(f'Valid Loss: {valid_loss : .4f} | Valid Accuracy: {valid_acc * 100 : .4f}%')
    print()
    torch.cuda.empty_cache()

  return train_loss_history, train_acc_history, valid_loss_history, valid_acc_history
```

Model Training

Meta data

```
hyperparameters = dict(

    HIDDEN_SIZES_LIST = [300] + [200] + [100],  # 300 = embed_dim, 200- hidden_dim1 , 100 -hidden_dim2
    DPROBS_LIST = [0.5] + [0.5], # 0.5 - dropout after first hidden layer layer, 0.8 dropout after  second hidden layer
    BATCHNORM_BINARY = True,


    VOCAB_SIZE = len(stack_exchange_vocab),
    OUTPUT_DIM = 10,

    EPOCHS = 20,

    BATCH_SIZE = 256,
    LEARNING_RATE = 0.2,
    DATASET="stack_exchange",
    ARCHITECTUREe="Embed_2_hidden_layers",
    LOG_INTERVAL = 25,
    LOG_BATCH = True,
    FILE_MODEL = save_model_folder/'stack_exchange_2_hidden_layers.pt',
    GRAD_CLIPPING = False,
    MAX_NORM = 0,
    MOMENTUM = 0,
    PATIENCE = 5,
    EARLY_STOPPING = True,
    SCHEDULER_FACTOR = 0,
    SCHEDULER_PATIENCE = 0,
    WEIGHT_DECAY = 0
   )
```

```
# Activation function for hidden layer
non_linearity = F.relu
```

Initialize wandb

```
wandb.init(name = 'Embed_2_hidden_layers', project = 'NLP_MLP_Stack_Exchange', config = hyperparameters)
```

wandb: Currently logged in as: arulchak (use `wandb login --relogin` to force relogin)
Tracking run with wandb version 0.12.14
Run data is saved locally in /content/wandb/run-20220410_221011-2rohzf2j
Syncing run **Embed_2_hidden_layers** to Weights & Biases (docs)
Display W&B run

```
# add non_linearity to config file
wandb.config.NON_LINEARITY = non_linearity
```

Specify Dataloader, Loss_function, Model, Optimizer, Weight Initialization

```
# Fix seed value
from datetime import datetime
SEED = 2345
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

# Data Loader
train_loader = torch.utils.data.DataLoader(trainset, batch_size=wandb.config.BATCH_SIZE, shuffle = True,
                                           collate_fn=collate_batch, num_workers = 4)
valid_loader = torch.utils.data.DataLoader(validset, batch_size=wandb.config.BATCH_SIZE, shuffle = False,
                                           collate_fn=collate_batch,  num_workers = 4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=wandb.config.BATCH_SIZE,   shuffle = False,
                                          collate_fn=collate_batch,  num_workers = 4)

# cross entropy loss function
loss_function = nn.CrossEntropyLoss()

# use GPUs
#device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device ='cpu'

# model
model_stack_exchange = MLPCustom(wandb.config.VOCAB_SIZE, wandb.config.HIDDEN_SIZES_LIST, wandb.config.DPROBS_LIST, wandb.config.BATC
           wandb.config.OUTPUT_DIM, non_linearity)
model_stack_exchange.to(device)

def init_weights(m):
  if type(m) == nn.Linear:
      torch.nn.init.kaiming_normal_(m.weight)
      torch.nn.init.zeros_(m.bias)

# apply initialization recursively  to all modules
model_stack_exchange.apply(init_weights)

# Intialize stochiastic gradient descent as optimizer
optimizer = torch.optim.SGD(model_stack_exchange.parameters(), lr = wandb.config.LEARNING_RATE, weight_decay=wandb.config.WEIGHT_DECA

wandb.config.optimizer = optimizer
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 4 worker pr
  cpuset_checked))
```

Sanity Check

- Check the loss without any training. For Cross entropy the expected value will be log(number of classes)

```
# Fix seed value
SEED = 2345
random.seed(SEED)
```

```
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

for input, targets, offsets in train_loader:

  # move inputs and outputs to GPUs
  input = input.to(device)
  targets = targets.to(device)
  offsets = offsets.to(device)
  model_stack_exchange.eval()
  # Forward pass
  output = model_stack_exchange(input, offsets)
  loss = loss_function(output, targets)
  print(f'Actual loss: {loss}')
  break

print(f'Expected Theoretical loss: {np.log(2)}')
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will create 4 worker pr
  cpuset_checked))
Actual loss: 2.3519251346588135
Expected Theoretical loss: 0.6931471805599453
```

Train Model and Save best model

```
wandb.watch(model_stack_exchange, log = 'all', log_freq=25, log_graph=True)
```

```
wandb: logging graph, to disable use `wandb.watch(log_graph=False)`
[<wandb.wandb_torch.TorchGraph at 0x7f3c272c2950>]
```

```
example_ct_train, batch_ct_train, example_ct_valid, batch_ct_valid = 0, 0, 0, 0
train_loss_history, train_acc_history, valid_loss_history, valid_acc_history = train_loop(train_loader, valid_loader, model_stack_exc
                                                                                          wandb.config.EPOCHS, device,
                                                                                          wandb.config.PATIENCE, wandb.config.EARLY_S
                                                                                          wandb.config.FILE_MODEL)
```

```
Train Loss:  0.5840 | Train Accuracy:  81.4682%
Valid Loss:  0.5229 | Valid Accuracy:  83.2284%

Validation loss has decreased (0.522871 --> 0.509175). Saving model...
Epoch : 12 / 20
Time to complete 12 is 0:01:28.939615
Train Loss:  0.5685 | Train Accuracy:  82.0650%
Valid Loss:  0.5092 | Valid Accuracy:  83.8814%

Early stoping counter: 1 out of 5
Epoch : 13 / 20
Time to complete 13 is 0:01:29.036023
Train Loss:  0.5603 | Train Accuracy:  82.2133%
Valid Loss:  0.5097 | Valid Accuracy:  83.8932%

Validation loss has decreased (0.509175 --> 0.503600). Saving model...
Epoch : 14 / 20
Time to complete 14 is 0:01:29.435028
Train Loss:  0.5495 | Train Accuracy:  82.5846%
Valid Loss:  0.5036 | Valid Accuracy:  84.0226%

Validation loss has decreased (0.503600 --> 0.497045). Saving model...
Epoch : 15 / 20
Time to complete 15 is 0:01:29.668937
Train Loss:  0.5394 | Train Accuracy:  82.8493%
Valid Loss:  0.4970 | Valid Accuracy:  84.0755%

Validation loss has decreased (0.497045 --> 0.483569). Saving model...
Epoch : 16 / 20
Time to complete 16 is 0:01:29.106426
Train Loss:  0.5325 | Train Accuracy:  83.0997%
Valid Loss:  0.4836 | Valid Accuracy:  84.4638%

Validation loss has decreased (0.483569 --> 0.475371). Saving model...
Epoch : 17 / 20
Time to complete 17 is 0:01:30.383227
Train Loss:  0.5240 | Train Accuracy:  83.3585%
Valid Loss:  0.4754 | Valid Accuracy:  84.8403%

Early stoping counter: 1 out of 5
Epoch : 18 / 20
Time to complete 18 is 0:01:29.783428
```

```
Train Loss:  0.5166 | Train Accuracy:  83.5834%
Valid Loss:  0.4806 | Valid Accuracy:  84.4991%

Validation loss has decreased (0.475371 --> 0.466566). Saving model...
Epoch : 19 / 20
Time to complete 19 is 0:01:29.797050
Train Loss:  0.5071 | Train Accuracy:  83.8056%
Valid Loss:  0.4666 | Valid Accuracy:  85.0109%

Validation loss has decreased (0.466566 --> 0.464801). Saving model...
Epoch : 20 / 20
Time to complete 20 is 0:01:30.722929
Train Loss:  0.5013 | Train Accuracy:  84.1213%
Valid Loss:  0.4648 | Valid Accuracy:  85.0932%
```

Add Visulaization

Add Loss plot

```
import matplotlib.pyplot as plt
# Plot the train loss and test loss per iteration
fig = plt.figure(0)
plt.plot(train_loss_history, label = 'train loss')
plt.plot(valid_loss_history, label = 'valid loss')
plt.legend()

# Log the plot to W&B
wandb.log({"train-test loss per epoch": fig})
```

```
/usr/local/lib/python3.7/dist-packages/plotly/matplotlylib/renderer.py:613: UserWarning:

I found a path object that I don't think is part of a bar chart. Ignoring.
```

Accuracy and Predictions

Now we have final values for weights and bias after training the model. We will use these values to make predictions on the test dataset.

Function to get predictions

```
def get_acc_pred(data_loader, model):
  """
  Function to get predictions for a given test set and calculate accuracy.
  Input: Iterator to the test set.
  Output: Prections and Accuracy for test set.
  """
  model.eval()
  with torch.no_grad():
    # Array to store predicted labels
    predictions = torch.Tensor()
    predictions = predictions.to(device)

    # Array to store actual labels
    y = torch.Tensor()
    y = y.to(device)
    # Iterate over batches from test set
    for inputs, targets, offsets in data_loader:

      # move inputs and outputs to GPUs
      inputs = inputs.to(device)
      targets = targets.to(device)
      offsets = offsets.to(device)

      # Calculated the predicted labels
      output = model(inputs, offsets)

      # Choose the label with maximum probability
      indices = torch.argmax(output, dim = 1)
```

```
        # Add the predicted labels to the array
        predictions = torch.cat((predictions, indices))

        # Add the actual labels to the array
        y = torch.cat((y, targets))

      # Check for complete dataset if actual and predicted labels are same or not
      # Calculate accuracy
      acc = (predictions == y).float().mean()

    # Return array containing predictions and accuracy
    return predictions, acc
```

Load saved model from file

```
model_nn = MLPCustom(wandb.config.VOCAB_SIZE, wandb.config.HIDDEN_SIZES_LIST, wandb.config.DPROBS_LIST, wandb.config.BATCHNORM_BINARY
           wandb.config.OUTPUT_DIM, non_linearity)
model_nn.to(device)
model_nn.load_state_dict(torch.load(wandb.config.FILE_MODEL))
```

```
<All keys matched successfully>
```

```
print(wandb.config.FILE_MODEL)
```

```
/content/drive/MyDrive/NLP/models/stack_exchange_2_hidden_layers.pt
```

```
# Get the prediction and accuracy for the test dataset
predictions, acc_test = get_acc_pred(test_loader, model_nn)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning:

This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which is sm
```

```
# Print Accuracy for test dataset
print(acc_test * 100)
```

```
tensor(85.1493)
```

We have obtained 85 % accuracy on test dataset.

Visualizations on Predictions

Now, we will make some visualizations for the predictions that we obtained.

We will construct a `confusion matrix` which will help us to visualize the performance of our classification model on the test dataset as we know the true values for the test data.

```
# Get an array containing actual labels
testing_labels = testset.y
```

```
# Define the values for classes
classes = ['c#', 'java', 'php', 'javascript', 'android', 'jquery', 'c++', 'python', 'iphone', 'asp.net']

# Log a confusion matrix to W&B
wandb.log({"conf_mat" : wandb.plot.confusion_matrix(
                        probs = None,
                        y_true = testing_labels,
                        preds = predictions.to('cpu').numpy(),
                        class_names = classes)})
```

```
wandb.finish()
```

Waiting for W&B process to finish... **(success).**

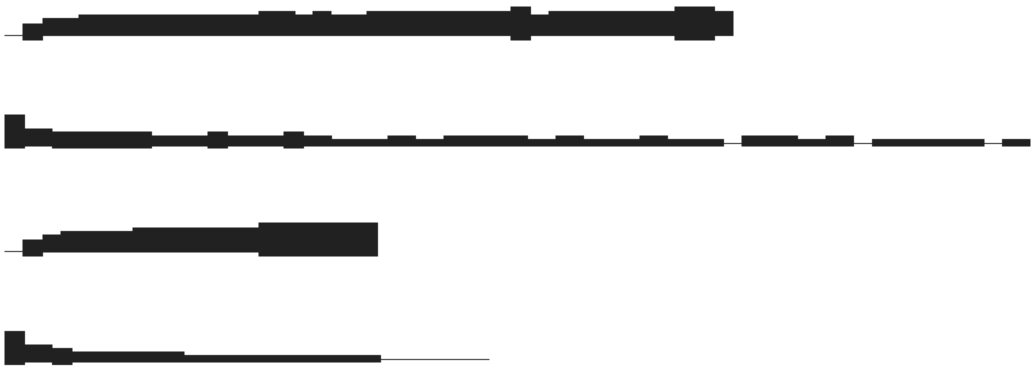
Run history:

Train Batch Acc :

Train Batch Loss :

Train epoch Acc :

Train epoch Loss :



Run summary:

Train Batch Acc :	0.85547
Train Batch Loss :	0.47444
Train epoch Acc :	0.84121
Train epoch Loss :	0.50129
Valid Batch Accuracy :	0.85547
Valid Batch Loss :	0.49636
Valid epoch Acc :	0.85093
Valid epoch Loss :	0.4648

✓ 4s completed at 5:47 PM