```
!pip install transformers
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from datetime import datetime
from pathlib import Path
import pandas as pd
import torchtext.data as ttd
```

```
Collecting transformers
  Downloading https://files.pythonhosted.org/packages/99/84/7bc03215279f603125d844bf8
     |████████████████████████████████| 1.4MB 12.5MB/s
Requirement already satisfied: dataclasses; python_version < "3.7" in /usr/local/lib/
Collecting tokenizers==0.9.4
  Downloading https://files.pythonhosted.org/packages/0f/1c/e789a8b12e28be5bc1ce2156c
     |████████████████████████████████| 2.9MB 48.9MB/s
Collecting sacremoses
  Downloading https://files.pythonhosted.org/packages/7d/34/09d19aff26edcc8eb2a01bed8
     |████████████████████████████████| 890kB 46.5MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: filelock in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: packaging in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from sa
Requirement already satisfied: click in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: joblib in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: pyparsing>=2.0.2 in /usr/local/lib/python3.6/dist-pack
Building wheels for collected packages: sacremoses
  Building wheel for sacremoses (setup.py) ... done
  Created wheel for sacremoses: filename=sacremoses-0.0.43-cp36-none-any.whl size=893
  Stored in directory: /root/.cache/pip/wheels/29/3c/fd/7ce5c3f0666dab31a50123635e6fb
Successfully built sacremoses
Installing collected packages: tokenizers, sacremoses, transformers
Successfully installed sacremoses-0.0.43 tokenizers-0.9.4 transformers-4.0.0
```

Loading Dataset

We will use The 20 Newsgroups dataset Dataset [homepage](#):

Scikit-learn includes some nice helper functions for retrieving the 20 Newsgroups dataset-- https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html. We'll use them below to retrieve the dataset.

Also look at results fron non- neural net models here : https://scikit-learn.org/stable/auto_examples/text/plot_document_classification_20newsgroups.html#sphx-glr-auto-examples-text-plot-document-classification-20newsgroups-py

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelera
  print('and then re-execute this cell.')
else:
  print(gpu_info)
```

```
Wed Dec  2 07:19:09 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.38       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    23W / 300W |      0MiB / 16130MiB |      0%      Default |
|                               |                      |                 ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

```
from sklearn.datasets import fetch_20newsgroups

train = fetch_20newsgroups(subset='train',
                           remove=('headers', 'footers', 'quotes'))

test = fetch_20newsgroups(subset='test',
                          remove=('headers', 'footers', 'quotes'))
```

```
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 MB)
```

```
print(train.data[0])
```

```
I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
```

```
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
```

```
print(train.target[0])
```

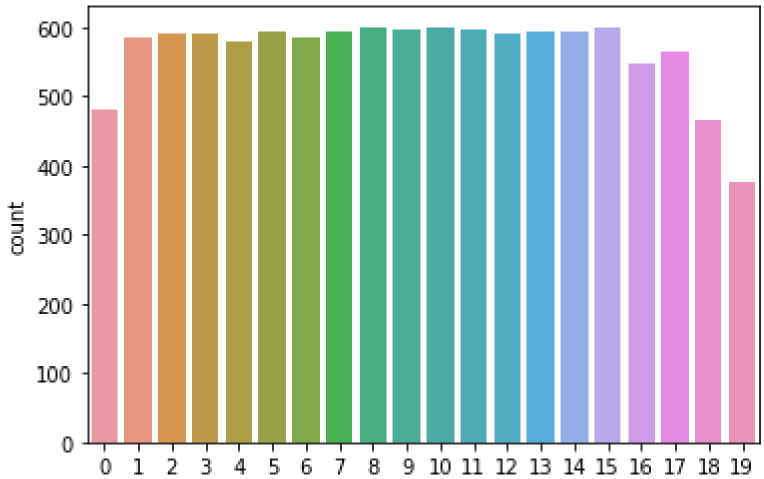
```
7
```

```
train.target_names
```

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

```
import seaborn as sns

# Plot the number of tokens of each length.
sns.countplot(train.target);
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass
  FutureWarning
```



BERT with 128 features and truncating at end

```
rom transformers import RobertaTokenizer

 Load the BERT tokenizer.
rint('Loading BERT tokenizer...')
okenizer = RobertaTokenizer.from_pretrained('roberta-base', do_lower_case=True)
```

```
Loading BERT tokenizer...
```

Downloading: 100% 899k/899k [00:01<00:00, 826kB/s]

Downloading: 100% 456k/456k [00:00<00:00, 933kB/s]

```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []

# For every sentence...
for sent in train.data:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        truncation=True, #Truncate the sentences
                        max_length = 128,           # Pad & truncate all sentence
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(train.target)

# Print sentence 0, now as a list of IDs.
print('Original: ', train.data[0])
print('Token IDs:', input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
```

```
  FutureWarning,
Original:  I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
Token IDs: tensor([    0,   100,    21,  8020,   114,  1268,    66,    89,   115, 36:
          225,   162,    15,    42,   512,    38,   794, 50118,   627,    97,
          183,     4,    85,    21,    10,   132,    12, 11219,  1612,   512,
            6,  1415,     7,    28,    31,     5,   628,  1191,    29,    73,
        50118, 23099,  1510,    29,     4,    85,    21,   373,    10, 23868,
         2614,     4,    20,  4259,    58,   269,   650,     4,    96,  1285,
            6, 50118,   627,   760, 20314,    21,  2559,    31,     5,  1079,
            9,     5,   809,     4,   152,    16,  1437, 50118,  1250,    38,
          216,     4,   318,  1268,    64,  1137,  1794,    10,  1421,   766,
            6,  3819, 21634,     6,   107, 50118,  1116,   931,     6,   147,
           42,   512,    16,   156,     6,   750,     6,    50,  3046,  8574,
           47, 50118, 11990,    15,    42, 33205,   546,   512,     6,  2540,
          364,    12,  6380,     4,     2,     1,     1,     1])
```

```
test_input_ids = []
test_attention_masks = []

# For every sentence...
for sent in test.data:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        truncation=True, #Truncate the sentences
                        max_length = 128,           # Pad & truncate all sentence
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    test_input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    test_attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
test_labels = torch.tensor(test.target)
```

```
# Print sentence 0, now as a list of IDs.
print('Original: ', test.data[0])
print('Token IDs:', test_input_ids[0])
```

```
/usr/local/lib/python3.6/dist-packages/transformers/tokenization_utils_base.py:2142:
  FutureWarning,
Original:  I am a little confused on all of the models of the 88-89 bonnevilles.
I have heard of the LE SE LSE SSE SSEI. Could someone tell me the
differences are far as features or performance. I am also curious to
know what the book value is for prefereably the 89 model. And how much
less than book value can you usually get them for. In other words how
much are they in demand this time of year. I have heard that the mid-spring
early summer is the best time to buy.
Token IDs: tensor([    0,   100,   524,    10,   410, 10985,    15,    70,     9,
         3092,     9,     5,  7953,    12,  5046, 13295,   858,   705, 22244,
            4, 50118,   100,    33,  1317,     9,     5, 10611,  6324,   226,
         3388,   208,  3388,   208,  3388,   100,     4,  9918,   951,  1137,
          162,     5, 50118, 32278, 36806,    32,   444,    25,  1575,    50,
          819,     4,    38,   524,    67, 10691,     7, 50118, 27066,    99,
            5,  1040,   923,    16,    13, 33284,  2816,  4735,     5,  8572,
         1421,     4,   178,   141,   203, 50118,  1672,    87,  1040,   923,
           64,    47,  2333,   120,   106,    13,     4,    96,    97,  1617,
          141, 50118, 28431,    32,    51,    11,  1077,    42,    86,     9,
           76,     4,    38,    33,  1317,    14,     5,  1084,    12, 36741,
        50118, 23099,  1035,    16,     5,   275,    86,     7,   907,     4,
            2,     1,     1,     1,     1,     1,     1,     1])
```

```
from torch.utils.data import TensorDataset, random_split

# Combine the training inputs into a TensorDataset.
dataset = TensorDataset(input_ids, attention_masks, labels)
test_dataset = TensorDataset(test_input_ids, test_attention_masks, test_labels)

# Create a 90-10 train-validation split.

# Calculate the number of samples to include in each set.
train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size

# Divide the dataset by randomly selecting samples.
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

print('{:>5,} training samples'.format(train_size))
print('{:>5,} validation samples'.format(val_size))
print('{:>5,} test samples'.format(len(test_dataset)))
```

```
10,182 training samples
1,132 validation samples
7,532 test samples
```

```
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

# The DataLoader needs to know our batch size for training, so we specify it
# here. For fine-tuning BERT on a specific task, the authors recommend a batch
# size of 16 or 32.
batch_size = 8
```

```
# Create the DataLoaders for our training and validation sets.
# We'll take training samples in random order.
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )

# For validation the order doesn't matter, so we'll just read them sequentially.
validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batches sequenti
            batch_size = batch_size # Evaluate with this batch size.
        )

test_dataloader = DataLoader(
            test_dataset,  # The training samples.
            sampler = RandomSampler(test_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )
```

Training just the Fully Connected Classifier layer by freezing the bert model weights

```
from transformers import  RobertaModel

bert_model = RobertaModel.from_pretrained('roberta-base')
```

Downloading: 100% 481/481 [00:00<00:00, 1.12kB/s]

Downloading: 100% 501M/501M [00:09<00:00, 50.3MB/s]

```
bert_model
```

```
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
```

```
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): RobertaPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
```

```
# Define the model
class linear(nn.Module):

  def __init__(self, bert_model, n_outputs, dropout_rate):

    super(linear, self).__init__()

    self.D = bert_model.config.to_dict()['hidden_size']
    self.bert_model = bert_model
    self.K = n_outputs
    self.dropout_rate=dropout_rate

    # embedding layer
    #self.embed = nn.Embedding(self.V, self.D)
```

```
    #self.embed = nn.Embedding(self.V, self.D)


    # dense layer
    self.fc = nn.Linear(self.D , self.K)

    # dropout layer
    self.dropout= nn.Dropout(self.dropout_rate)

  def forward(self, X):

    with torch.no_grad():
      embedding = self.bert_model(X)[0][:,0,:]

    #embedding= self.dropout(embedding)

    output = self.fc(embedding)
    output= self.dropout(output)

    return output


n_outputs = 20
dropout_rate = 0.5


#model = RNN(n_vocab, embed_dim, n_hidden, n_rnnlayers, n_outputs, bidirectional,
model = linear(bert_model, n_outputs, dropout_rate)
model.to(device)
```

```
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (10): RobertaLayer(
        (attention): RobertaAttention(
          (self): RobertaSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): RobertaSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): RobertaIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): RobertaOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (11): RobertaLayer(
        (attention): RobertaAttention(
          (self): RobertaSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
```

```
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): RobertaPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (fc): Linear(in_features=768, out_features=20, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

```
print(model)
```

```
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
```

```
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): RobertaPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (fc): Linear(in_features=768, out_features=20, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

```
for name, param in model.named_parameters():
  print(name, param.shape)
```

```
bert_model.encoder.layer.8.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.8.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.8.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.8.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.8.output.dense.bias torch.Size([768])
bert_model.encoder.layer.8.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.8.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.key.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.key.bias torch.Size([768])
bert_model.encoder.layer.9.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.9.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.9.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.9.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.9.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.9.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.9.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.9.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.9.output.dense.bias torch.Size([768])
bert_model.encoder.layer.9.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.9.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.10.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.10.attention.self.key.weight torch.Size([768, 768])
```

```
bert_model.encoder.layer.10.attention.self.key.bias torch.Size([768])
bert_model.encoder.layer.10.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.10.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.10.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.10.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.10.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.10.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.10.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.10.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.10.output.dense.bias torch.Size([768])
bert_model.encoder.layer.10.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.10.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.query.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.query.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.key.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.key.bias torch.Size([768])
bert_model.encoder.layer.11.attention.self.value.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.self.value.bias torch.Size([768])
bert_model.encoder.layer.11.attention.output.dense.weight torch.Size([768, 768])
bert_model.encoder.layer.11.attention.output.dense.bias torch.Size([768])
bert_model.encoder.layer.11.attention.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.11.attention.output.LayerNorm.bias torch.Size([768])
bert_model.encoder.layer.11.intermediate.dense.weight torch.Size([3072, 768])
bert_model.encoder.layer.11.intermediate.dense.bias torch.Size([3072])
bert_model.encoder.layer.11.output.dense.weight torch.Size([768, 3072])
bert_model.encoder.layer.11.output.dense.bias torch.Size([768])
bert_model.encoder.layer.11.output.LayerNorm.weight torch.Size([768])
bert_model.encoder.layer.11.output.LayerNorm.bias torch.Size([768])
bert_model.pooler.dense.weight torch.Size([768, 768])
bert_model.pooler.dense.bias torch.Size([768])
fc.weight torch.Size([20, 768])
fc.bias torch.Size([20])
```

```
import random

seed = 123

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

learning_rate = 0.001
epochs=10

# STEP 5: INSTANTIATE LOSS CLASS
criterion = nn.CrossEntropyLoss()

# STEP 6: INSTANTIATE OPTIMIZER CLASS

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Freeze embedding Layer

#freeze embeddings
#model.embed.weight.requires_grad  = False
```

```
# STEP 7: TRAIN THE MODEL

train_losses= np.zeros(epochs)
valid_losses= np.zeros(epochs)


for epoch in range(epochs):

  t0= datetime.now()
  train_loss=[]

  model.train()
  for batch in train_dataloader:

    # forward pass
    output= model(batch[0].to(device))
    loss=criterion(output,batch[2].to(device))

    # set gradients to zero
    optimizer.zero_grad()

    # backward pass
    loss.backward()
    optimizer.step()
    train_loss.append(loss.item())

  train_loss=np.mean(train_loss)

  valid_loss=[]
  model.eval()
  with torch.no_grad():
    for batch in validation_dataloader:

      # forward pass
      output= model(batch[0].to(device))
      loss=criterion(output,batch[2].to(device))

      valid_loss.append(loss.item())

    valid_loss=np.mean(valid_loss)

  # save Losses
  train_losses[epoch]= train_loss
  valid_losses[epoch]= valid_loss
  dt= datetime.now()-t0
  print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}    Valid Loss: {
```

```
   Epoch 1/10, Train Loss: 2.9532    Valid Loss: 2.8360, Duration: 0:00:31.186350
   Epoch 2/10, Train Loss: 2.8261    Valid Loss: 2.7139, Duration: 0:00:31.107693
   Epoch 3/10, Train Loss: 2.7307    Valid Loss: 2.6224, Duration: 0:00:31.100038
   Epoch 4/10, Train Loss: 2.6755    Valid Loss: 2.5402, Duration: 0:00:31.128046
   Epoch 5/10, Train Loss: 2.6331    Valid Loss: 2.4827, Duration: 0:00:31.148078
   Epoch 6/10, Train Loss: 2.6037    Valid Loss: 2.4380, Duration: 0:00:31.155191
   Epoch 7/10, Train Loss: 2.5736    Valid Loss: 2.3783, Duration: 0:00:31.159997
   Epoch 8/10, Train Loss: 2.5548    Valid Loss: 2.3535, Duration: 0:00:31.015302
```

```
Epoch 9/10, Train Loss: 2.5245    Valid Loss: 2.3185, Duration: 0:00:31.188327
Epoch 10/10, Train Loss: 2.5225    Valid Loss: 2.2842, Duration: 0:00:31.334590
```

```
# Accuracy- write a function to get accuracy
# use this function to get accuracy and print accuracy
def get_accuracy(data_iter, model):
  model.eval()
  with torch.no_grad():
    correct =0
    total =0

    for batch in data_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      correct+= (batch[2].to(device)==indices).sum().item()
      total += batch[2].shape[0]

    acc= correct/total

    return acc


train_acc = get_accuracy(train_dataloader, model)
valid_acc = get_accuracy(validation_dataloader, model)
test_acc = get_accuracy(test_dataloader ,model)
print(f'Train acc: {train_acc:.4f},\t Valid acc: {valid_acc:.4f},\t Test acc: {te
```

```
Train acc: 0.5217,       Valid acc: 0.4947,      Test acc: 0.4734
```

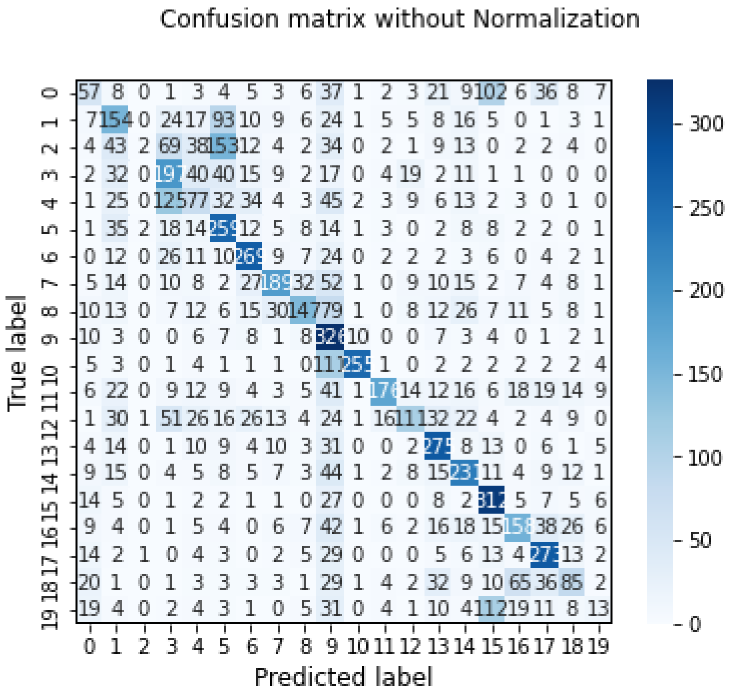
```
# Write a function to get predictions

def get_predictions(test_iter, model):
  model.eval()
  with torch.no_grad():
    predictions= np.array([])
    y_test= np.array([])

    for batch in test_iter:

      output=model(batch[0].to(device))
      _,indices = torch.max(output,dim=1)
      predictions=np.concatenate((predictions,indices.cpu().numpy()))
      y_test = np.concatenate((y_test,batch[2].numpy()))

  return y_test, predictions


y_test, predictions=get_predictions(test_dataloader, model)
```

```
# Confusion Matrix
cm=confusion_matrix(y_test,predictions)
cm
```

```
array([[ 57,   8,   0,   1,   3,   4,   5,   3,   6,  37,   1,   2,   3,
         21,   9, 102,   6,  36,   8,   7],
       [  7, 154,   0,  24,  17,  93,  10,   9,   6,  24,   1,   5,   5,
          8,  16,   5,   0,   1,   3,   1],
       [  4,  43,   2,  69,  38, 153,  12,   4,   2,  34,   0,   2,   1,
          9,  13,   0,   2,   2,   4,   0],
       [  2,  32,   0, 197,  40,  40,  15,   9,   2,  17,   0,   4,  19,
          2,  11,   1,   1,   0,   0,   0],
       [  1,  25,   0, 125,  77,  32,  34,   4,   3,  45,   2,   3,   9,
          6,  13,   2,   3,   0,   1,   0],
       [  1,  35,   2,  18,  14, 259,  12,   5,   8,  14,   1,   3,   0,
          2,   8,   8,   2,   2,   0,   1],
       [  0,  12,   0,  26,  11,  10, 269,   9,   7,  24,   0,   2,   2,
          2,   3,   6,   0,   4,   2,   1],
       [  5,  14,   0,  10,   8,   2,  27, 189,  32,  52,   1,   0,   9,
         10,  15,   2,   7,   4,   8,   1],
       [ 10,  13,   0,   7,  12,   6,  15,  30, 147,  79,   1,   0,   8,
         12,  26,   7,  11,   5,   8,   1],
       [ 10,   3,   0,   0,   6,   7,   8,   1,   8, 326,  10,   0,   0,
          7,   3,   4,   0,   1,   2,   1],
       [  5,   3,   0,   1,   4,   1,   1,   1,   0, 111, 255,   1,   0,
          2,   2,   2,   2,   2,   2,   4],
       [  6,  22,   0,   9,  12,   9,   4,   3,   5,  41,   1, 176,  14,
         12,  16,   6,  18,  19,  14,   9],
       [  1,  30,   1,  51,  26,  16,  26,  13,   4,  24,   1,  16, 111,
         32,  22,   4,   2,   4,   9,   0],
       [  4,  14,   0,   1,  10,   9,   4,  10,   3,  31,   0,   0,   2,
        275,   8,  13,   0,   6,   1,   5],
       [  9,  15,   0,   4,   5,   8,   5,   7,   3,  44,   1,   2,   8,
         15, 231,  11,   4,   9,  12,   1],
       [ 14,   5,   0,   1,   2,   2,   1,   1,   0,  27,   0,   0,   0,
          8,   2, 312,   5,   7,   5,   6],
       [  9,   4,   0,   1,   5,   4,   0,   6,   7,  42,   1,   6,   2,
         16,  18,  15, 158,  38,  26,   6],
       [ 14,   2,   1,   0,   4,   3,   0,   2,   5,  29,   0,   0,   0,
          5,   6,  13,   4, 273,  13,   2],
       [ 20,   1,   0,   1,   3,   3,   3,   3,   1,  29,   1,   4,   2,
         32,   9,  10,  65,  36,  85,   2],
       [ 19,   4,   0,   2,   4,   3,   1,   0,   5,  31,   0,   4,   1,
         10,   4, 112,  19,  11,   8,  13]])
```

```
# Write a function to print confusion matrix
# plot confusion matrix
# need to import confusion_matrix from sklearn for this function to work
# need to import seaborn as sns
# import seaborn as sns
# import matplotlib.pyplot as plt
# from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(y_true,y_pred,normalize=None):
  cm=confusion_matrix(y_true,y_pred,normalize=normalize)
  fig, ax = plt.subplots(figsize=(6,5))
  if normalize == None:
    fmt='d'
    fig.suptitle('Confusion matrix without Normalization', fontsize=12)

  else :
    fmt='0.2f'
```

```
    fig.suptitle('Normalized confusion matrix', fontsize=12)

  ax=sns.heatmap(cm,cmap=plt.cm.Blues,annot=True,fmt=fmt)
  ax.axhline(y=0, color='k',linewidth=1)
  ax.axhline(y=cm.shape[1], color='k',linewidth=2)
  ax.axvline(x=0, color='k',linewidth=1)
  ax.axvline(x=cm.shape[0], color='k',linewidth=2)

  ax.set_xlabel('Predicted label', fontsize=12)
  ax.set_ylabel('True label', fontsize=12)


plot_confusion_matrix(y_test,predictions)
```



Finetuning the pre-trained Roberta Model

```
from transformers import RobertaForSequenceClassification, AdamW, RobertaConfig

# Load BertForSequenceClassification, the pretrained BERT model with a single
# linear classification layer on top.
model = RobertaForSequenceClassification.from_pretrained(
    "roberta-base", # Use the 12-layer Robera model, with an uncased vocab.
    num_labels = 20, # The number of output labels
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.
)
```

```
Some weights of the model checkpoint at roberta-base were not used when initializing
- This IS expected if you are initializing RobertaForSequenceClassification from the
- This IS NOT expected if you are initializing RobertaForSequenceClassification from
```

```
Some weights of RobertaForSequenceClassification were not initialized from the model
You should probably TRAIN this model on a down-stream task to be able to use it for p
```

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda', index=0)
```

```
# Tell pytorch to run this model on the GPU.
model.to(device)
```

```
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
```

```
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
    (classifier): RobertaClassificationHead(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.1, inplace=False)
      (out_proj): Linear(in_features=768, out_features=20, bias=True)
    )
  )
```

Just for curiosity's sake, we can browse all of the model's parameters by name here.

In the below cell, I've printed out the names and dimensions of the weights for:

1. The embedding layer.
2. The first of the twelve transformers.
3. The output layer.

```
# Get all of the model's parameters as a list of tuples.
params = list(model.named_parameters())

print('The BERT model has {:} different named parameters.\n'.format(len(params)))

print('==== Embedding Layer ====\n')

for p in params[0:5]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))

print('\n==== First Transformer ====\n')

for p in params[5:21]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))

print('\n==== Output Layer ====\n')

for p in params[-4:]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
```

```
The BERT model has 201 different named parameters.

==== Embedding Layer ====

roberta.embeddings.word_embeddings.weight                 (50265, 768)
roberta.embeddings.position_embeddings.weight               (514, 768)
roberta.embeddings.token_type_embeddings.weight               (1, 768)
roberta.embeddings.LayerNorm.weight                             (768,)
roberta.embeddings.LayerNorm.bias                               (768,)

==== First Transformer ====

roberta.encoder.layer.0.attention.self.query.weight         (768, 768)
```

```
roberta.encoder.layer.0.attention.self.query.bias              (768,)
roberta.encoder.layer.0.attention.self.key.weight          (768, 768)
roberta.encoder.layer.0.attention.self.key.bias                (768,)
roberta.encoder.layer.0.attention.self.value.weight        (768, 768)
roberta.encoder.layer.0.attention.self.value.bias              (768,)
roberta.encoder.layer.0.attention.output.dense.weight      (768, 768)
roberta.encoder.layer.0.attention.output.dense.bias            (768,)
roberta.encoder.layer.0.attention.output.LayerNorm.weight        (768,)
roberta.encoder.layer.0.attention.output.LayerNorm.bias        (768,)
roberta.encoder.layer.0.intermediate.dense.weight         (3072, 768)
roberta.encoder.layer.0.intermediate.dense.bias               (3072,)
roberta.encoder.layer.0.output.dense.weight               (768, 3072)
roberta.encoder.layer.0.output.dense.bias                      (768,)
roberta.encoder.layer.0.output.LayerNorm.weight                (768,)
roberta.encoder.layer.0.output.LayerNorm.bias                  (768,)

==== Output Layer ====

classifier.dense.weight                                    (768, 768)
classifier.dense.bias                                          (768,)
classifier.out_proj.weight                                  (20, 768)
classifier.out_proj.bias                                        (20,)
```

4.2. Optimizer & Learning Rate Scheduler

Now that we have our model loaded we need to grab the training hyperparameters from within the stored model.

For the purposes of fine-tuning, the authors recommend choosing from the following values (from Appendix A.3 of the BERT paper):

> - **Batch size:** 16, 32
> - **Learning rate (Adam):** 5e-5, 3e-5, 2e-5
> - **Number of epochs:** 2, 3, 4

We chose:

- Batch size: 32 (set when creating our DataLoaders)
- Learning rate: 2e-5
- Epochs: 4 (we'll see that this is probably too many...)

The epsilon parameter `eps = 1e-8` is "a very small number to prevent any division by zero in the implementation" (from here).

You can find the creation of the AdamW optimizer in `run_glue.py` here.

4.3. Training Loop

Define a helper function for calculating accuracy.

Helper function for formatting elapsed times as `hh:mm:ss`

We're ready to kick off the training!

```
# STEP 6: INSTANTIATE OPTIMIZER CLASS
epochs = 2
no_decay = ['bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
        {'params': [p for n, p in model.named_parameters()
          if not any(nd in n for nd in no_decay)],
         'weight_decay': 0.5},

        {'params': [p for n, p in model.named_parameters()
        if any(nd in n for nd in no_decay)],
         'weight_decay': 0.0}
        ]

optimizer = AdamW(optimizer_grouped_parameters,
                  lr = 5e-5,
                  eps = 1e-8
                )

no_decay = ['bias', 'LayerNorm.weight']
```

```
from transformers import get_linear_schedule_with_warmup

# Total number of training steps is [number of batches] x [number of epochs].
total_steps = len(train_dataloader) * epochs

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default value
                                            num_training_steps = total_steps)
```

```
import random
from datetime import datetime

seed = 123

random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

epochs = 2

# STEP 7: TRAIN THE MODEL

train_losses= np.zeros(epochs)
valid_losses= np.zeros(epochs)
```

```
for epoch in range(epochs):

  t0= datetime.now()
  train_loss=[]

  model.train()
  for batch in train_dataloader:
    b_input_ids = batch[0]
    b_input_mask = batch[1]
    b_labels = batch[2]
    # forward pass

    outputs = model(b_input_ids.to(device),
                        token_type_ids=None,
                        attention_mask=b_input_mask.to(device),
                        labels=b_labels.to(device))


    # set gradients to zero
    optimizer.zero_grad()
    # backward pass
    outputs.loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()
    scheduler.step()
    train_loss.append(outputs.loss.item())

  train_loss=np.mean(train_loss)

  valid_loss=[]
  model.eval()
  with torch.no_grad():
    for batch in validation_dataloader:

      # forward pass
      b_input_ids = batch[0].to(device)
      b_input_mask = batch[1].to(device)
      b_labels = batch[2].to(device)
    # forward pass

      outputs = model(b_input_ids,
                        token_type_ids=None,
                        attention_mask=b_input_mask,
                        labels=b_labels)

      valid_loss.append(outputs.loss.item())

    valid_loss=np.mean(valid_loss)

  # save Losses
  train_losses[epoch]= train_loss
  valid_losses[epoch]= valid_loss
  dt= datetime.now()-t0
  print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f},    Valid Loss: {
```

```
  print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}    Valid Loss: {
```

```
Epoch 1/2, Train Loss: 1.4071    Valid Loss: 1.1054, Duration: 0:02:04.004429
Epoch 2/2, Train Loss: 0.8135    Valid Loss: 0.9469, Duration: 0:02:05.435136
```

```
# Accuracy- write a function to get accuracy
# use this function to get accuracy and print accuracy
def get_accuracy(data_iter, model):
  model.eval()
  with torch.no_grad():
    correct =0
    total =0

    for batch in data_iter:

      b_input_ids = batch[0].to(device)
      b_input_mask = batch[1].to(device)
      b_labels = batch[2].to(device)
    # forward pass

      outputs = model(b_input_ids,
                      token_type_ids=None,
                      attention_mask=b_input_mask,
                      labels=b_labels)

      _,indices = torch.max(outputs.logits,dim=1)
      correct+= (b_labels==indices).sum().item()
      total += b_labels.shape[0]

    acc= correct/total

    return acc


train_acc = get_accuracy(train_dataloader, model)
valid_acc = get_accuracy(validation_dataloader, model)
test_acc = get_accuracy(test_dataloader, model)


print(f'Train acc: {train_acc:.4f},\t Valid acc: {valid_acc:.4f},\t Test acc: {te
```

```
Train acc: 0.8338,       Valid acc: 0.7191,      Test acc: 0.6899
```

```
# Write a function to get predictions
def get_predictions(data_iter, model):
  model.eval()
  with torch.no_grad():
    predictions= np.array([])
    y_test= np.array([])

    for batch in data_iter:

      b_input_ids = batch[0].to(device)
      b_input_mask = batch[1].to(device)
      b_labels = batch[2].to(device)
```

```
      # forward pass

        outputs = model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask,
                            labels=b_labels)

        _,indices = torch.max(outputs.logits,dim=1)
        predictions=np.concatenate((predictions,indices.cpu().numpy()))
        y_test = np.concatenate((y_test,b_labels.cpu().numpy()))

  return y_test, predictions
```

```
y_valid, predictions=get_predictions(validation_dataloader, model)
```

```
predictions.max()
```

```
19.0
```

```
# Confusion Matrix
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_valid,predictions)
cm
```

```
array([[26,  0,  0,  0,  0,  0,  0,  1,  1,  0,  1,  1,  0,  3,  1,  5,
         1,  3,  2,  2],
       [ 1, 41,  3,  4,  3,  5,  0,  0,  1,  0,  0,  0,  1,  0,  0,  0,
         0,  0,  0,  0],
       [ 0,  2, 50,  3,  1,  2,  0,  4,  0,  0,  1,  0,  1,  0,  0,  0,
         0,  0,  0,  0],
       [ 0,  3,  7, 60,  2,  0,  2,  1,  0,  0,  0,  0,  4,  0,  0,  0,
         0,  0,  0,  0],
       [ 0,  3,  0, 12, 38,  0,  5,  3,  0,  0,  0,  0,  2,  0,  0,  0,
         0,  0,  0,  0],
       [ 1,  4,  3,  0,  1, 49,  0,  0,  1,  0,  0,  1,  0,  0,  0,  0,
         0,  0,  0,  0],
       [ 0,  1,  1,  2,  0,  0, 41,  1,  2,  0,  0,  1,  2,  0,  1,  0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 36,  2,  1,  0,  0,  2,  0,  0,  0,
         1,  1,  0,  0],
       [ 0,  0,  1,  1,  0,  0,  4,  3, 46,  1,  0,  0,  1,  3,  0,  0,
         3,  1,  0,  0],
       [ 1,  0,  0,  0,  0,  0,  0,  4,  3, 48,  1,  0,  0,  0,  0,  0,
         0,  0,  1,  0],
       [ 0,  0,  0,  0,  0,  0,  1,  1,  2,  0, 55,  0,  0,  0,  0,  0,
         1,  1,  1,  0],
       [ 0,  2,  1,  1,  0,  1,  0,  1,  1,  0,  0, 44,  3,  0,  0,  0,
         2,  1,  4,  1],
       [ 0,  0,  0,  2,  2,  0,  1,  2,  0,  0,  0,  1, 48,  2,  2,  0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  1,  0,  1,  1,  0,  0, 41,  1,  0,
         0,  0,  2,  0],
       [ 0,  4,  0,  0,  0,  1,  1,  2,  2,  3,  0,  0,  3,  5, 47,  0,
         1,  1,  3,  0],
       [ 9,  0,  1,  0,  0,  0,  0,  1,  1,  0,  0,  0,  0,  1,  0, 44,
         0,  0,  1,  3],
```

```
       [ 1,  0,  0,  0,  0,  1,  0,  2,  2,  1,  2,  2,  1,  1,  0,  0,
        32,  2,  6,  0],
       [ 3,  0,  0,  0,  0,  0,  0,  1,  0,  1,  0,  1,  0,  0,  0,  0,
         0, 39,  3,  0],
       [ 2,  0,  0,  0,  0,  0,  0,  2,  1,  1,  1,  1,  1,  1,  0,  0,
         1,  2, 25,  0],
       [ 6,  0,  0,  0,  0,  1,  0,  1,  1,  0,  0,  0,  1,  0,  0, 15,
         4,  3,  3,  4]])
```

```python
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(y_true,y_pred,normalize=None):
  cm=confusion_matrix(y_true,y_pred,normalize=normalize)
  fig, ax = plt.subplots(figsize=(6,5))
  if normalize == None:
    fmt='d'
    fig.suptitle('Confusion matrix without Normalization', fontsize=12)

  else :
    fmt='0.2f'
    fig.suptitle('Normalized confusion matrix', fontsize=12)

  ax=sns.heatmap(cm,cmap=plt.cm.Blues,annot=True,fmt=fmt)
  ax.axhline(y=0, color='k',linewidth=1)
  ax.axhline(y=cm.shape[1], color='k',linewidth=2)
  ax.axvline(x=0, color='k',linewidth=1)
  ax.axvline(x=cm.shape[0], color='k',linewidth=2)

  ax.set_xlabel('Predicted label', fontsize=12)
  ax.set_ylabel('True label', fontsize=12)


plot_confusion_matrix(y_valid,predictions)
```