```
!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 4.1MB/s
Collecting tokenizers==0.9.4
  Downloading https://files.pythonhosted.org/packages/0f/1c/e789a8b12e28be5bc1ce2156c
     |████████████████████████████████| 2.9MB 9.0MB/s
Collecting sacremoses
  Downloading https://files.pythonhosted.org/packages/7d/34/09d19aff26edcc8eb2a01bed8
     |████████████████████████████████| 890kB 34.5MB/s
Requirement already satisfied: dataclasses; python_version < "3.7" in /usr/local/lib/
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: packaging in /usr/local/lib/python3.6/dist-packages (f
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: numpy in /usr/local/lib/python3.6/dist-packages (from
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: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-pa
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 03:52:59 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 P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   48C    P0    29W / 250W |      0MiB / 16280MiB |      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']
```

```
len(train.target_names)
```

```
20
```

```
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: Pas
  FutureWarning
```

BERT with 140 features

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True
```

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

Downloading: 100% 232k/232k [00:00<00:00, 308kB/s]

```
# Get last three tokens and truncate head
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []
docoder_sent=[]
before_trunc=[]
maxlen=140
labels = torch.tensor(train.target)
# 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=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    before_trunc.append(encoded_dict['input_ids'])

    ids = encoded_dict['input_ids']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
```

```
        if len(ids)>=maxlen:
          ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
        else:
          ids = ids + ([0] * (maxlen-len(ids)))
        encoded_dict['attention_mask']=torch.tensor([ids])

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

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

        # Get the decoded sentence
        docoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()))

    # Convert the lists into tensors.
    input_ids = torch.cat(input_ids, dim=0)
    #print(input_ids)
    attention_masks = torch.cat(attention_masks, dim=0)
```

```
Token indices sequence length is longer than the specified maximum sequence length fc
```

```
    # Get last three tokens and truncate head
    # Tokenize all of the sentences and map the tokens to thier word IDs.
    test_input_ids = []
    test_attention_masks = []
    test_decoder_sent=[]
    test_before_trunc=[]
    maxlen=140
    test_labels = torch.tensor(test.target)
    # 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=False,
                            padding=False,
                            #max_length = maxlen,           # Pad & truncate all sent
                            return_attention_mask = True,   # Construct attn. masks.
                            #return_tensors = 'pt',     # Return pytorch tensors.
                       )

        test_before_trunc.append(encoded_dict['input_ids'])

        ids = encoded_dict['input_ids']
        if len(ids)>=maxlen:
```

```
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['attention_mask']=torch.tensor([ids])

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

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

    # Get the decoded sentence
    test_decoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()

# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
#print(input_ids)
test_attention_masks = torch.cat(test_attention_masks, dim=0)


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.
        )
```

```
from transformers import  BertModel

bert_model = BertModel.from_pretrained('bert-base-uncased')
```

Downloading: 100% 433/433 [00:00<00:00, 15.0kB/s]

Downloading: 100% 440M/440M [00:08<00:00, 53.8MB/s]

```
bert_model
```

```
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (10): BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (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): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
```

```
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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)


    # 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): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
```

```
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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])
```

```python
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: 2.7737    Valid Loss: 2.3332, Duration: 0:00:57.140807
   Epoch 2/10, Train Loss: 2.5742    Valid Loss: 2.1514, Duration: 0:00:56.940840
   Epoch 3/10, Train Loss: 2.5062    Valid Loss: 2.0596, Duration: 0:00:56.956009
   Epoch 4/10, Train Loss: 2.4841    Valid Loss: 2.0286, Duration: 0:00:56.963300
   Epoch 5/10, Train Loss: 2.4514    Valid Loss: 1.9505, Duration: 0:00:56.987166
   Epoch 6/10, Train Loss: 2.4478    Valid Loss: 1.9623, Duration: 0:00:56.957383
   Epoch 7/10, Train Loss: 2.4323    Valid Loss: 1.9529, Duration: 0:00:56.968736
   Epoch 8/10, Train Loss: 2.4437    Valid Loss: 1.8624, Duration: 0:00:56.948046
   Epoch 9/10, Train Loss: 2.4210    Valid Loss: 1.8885, Duration: 0:00:56.981147
   Epoch 10/10, Train Loss: 2.4209    Valid Loss: 1.8612, Duration: 0:00:56.944154
```

```
# Accuracy- write a function to get accuracy
# use this function to get accuracy and print 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: 0.5127,       Valid acc: 0.4602,      Test acc: 0.4336
```

```
# 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)
```

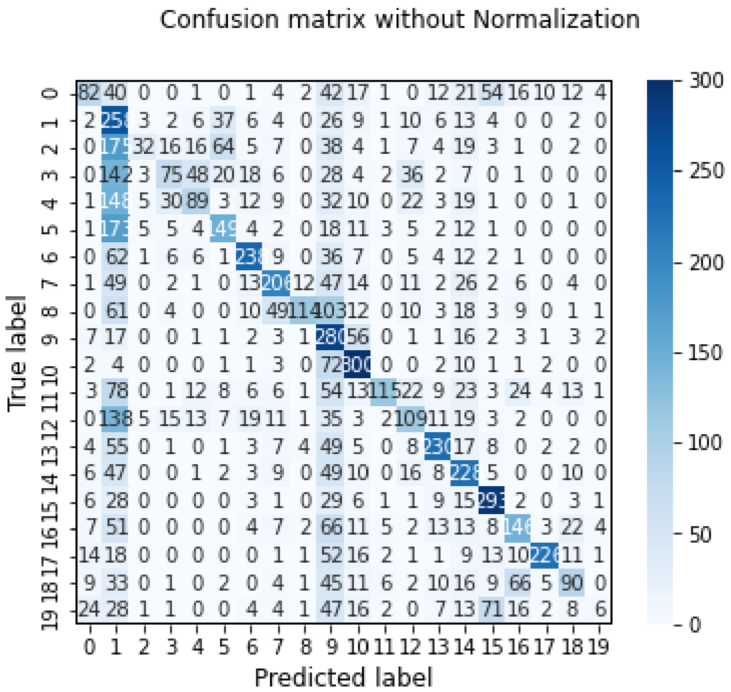
```
# Confusion Matrix
cm=confusion_matrix(y_test,predictions)
cm
```

```
array([[ 82,  40,   0,   0,   1,   0,   1,   4,   2,  42,  17,   1,   0,
         12,  21,  54,  16,  10,  12,   4],
       [  2, 258,   3,   2,   6,  37,   6,   4,   0,  26,   9,   1,  10,
          6,  13,   4,   0,   0,   2,   0],
       [  0, 175,  32,  16,  16,  64,   5,   7,   0,  38,   4,   1,   7,
```

```
           4,  19,   3,   1,   0,   2,   0],
       [  0, 142,   3,  75,  48,  20,  18,   6,   0,  28,   4,   2,  36,
           2,   7,   0,   1,   0,   0,   0],
       [  1, 148,   5,  30,  89,   3,  12,   9,   0,  32,  10,   0,  22,
           3,  19,   1,   0,   0,   1,   0],
       [  1, 173,   5,   5,   4, 149,   4,   2,   0,  18,  11,   3,   5,
           2,  12,   1,   0,   0,   0,   0],
       [  0,  62,   1,   6,   6,   1, 238,   9,   0,  36,   7,   0,   5,
           4,  12,   2,   1,   0,   0,   0],
       [  1,  49,   0,   2,   1,   0,  13, 206,  12,  47,  14,   0,  11,
           2,  26,   2,   6,   0,   4,   0],
       [  0,  61,   0,   4,   0,   0,  10,  49, 114, 103,  12,   0,  10,
           3,  18,   3,   9,   0,   1,   1],
       [  7,  17,   0,   0,   1,   1,   2,   3,   1, 280,  56,   0,   1,
           1,  16,   2,   3,   1,   3,   2],
       [  2,   4,   0,   0,   0,   1,   1,   3,   0,  72, 300,   0,   0,
           2,  10,   1,   1,   2,   0,   0],
       [  3,  78,   0,   1,  12,   8,   6,   6,   1,  54,  13, 115,  22,
           9,  23,   3,  24,   4,  13,   1],
       [  0, 138,   5,  15,  13,   7,  19,  11,   1,  35,   3,   2, 109,
          11,  19,   3,   2,   0,   0,   0],
       [  4,  55,   0,   1,   0,   1,   3,   7,   4,  49,   5,   0,   8,
         230,  17,   8,   0,   2,   2,   0],
       [  6,  47,   0,   0,   1,   2,   3,   9,   0,  49,  10,   0,  16,
           8, 228,   5,   0,   0,  10,   0],
       [  6,  28,   0,   0,   0,   0,   3,   1,   0,  29,   6,   1,   1,
           9,  15, 293,   2,   0,   3,   1],
       [  7,  51,   0,   0,   0,   0,   4,   7,   2,  66,  11,   5,   2,
          13,  13,   8, 146,   3,  22,   4],
       [ 14,  18,   0,   0,   0,   0,   0,   1,   1,  52,  16,   2,   1,
           1,   9,  13,  10, 226,  11,   1],
       [  9,  33,   0,   1,   0,   2,   0,   4,   1,  45,  11,   6,   2,
          10,  16,   9,  66,   5,  90,   0],
       [ 24,  28,   1,   1,   0,   0,   4,   4,   1,  47,  16,   2,   0,
           7,  13,  71,  16,   2,   8,   6]])
```

```
# 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)
```



BERT with 128

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True

     Loading BERT tokenizer...


# Get last three tokens and truncate head
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []
docoder_sent=[]
before_trunc=[]
maxlen=128
labels = torch.tensor(train.target)
# 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=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    before_trunc.append(encoded_dict['input_ids'])

    ids = encoded_dict['input_ids']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['attention_mask']=torch.tensor([ids])

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

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

    # Get the decoded sentence
    docoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()))

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
#print(input_ids)
attention_masks = torch.cat(attention_masks, dim=0)
```

```
Token indices sequence length is longer than the specified maximum sequence length fo
```

```
# Get last three tokens and truncate head
# Tokenize all of the sentences and map the tokens to thier word IDs.
test_input_ids = []
test_attention_masks = []
test_decoder_sent=[]
test_before_trunc=[]
maxlen=128
```

```
maxlen=128
test_labels = torch.tensor(test.target)
# 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=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    test_before_trunc.append(encoded_dict['input_ids'])

    ids = encoded_dict['input_ids']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['attention_mask']=torch.tensor([ids])

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

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

    # Get the decoded sentence
    test_decoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()

# Convert the lists into tensors.
test_input_ids = torch.cat(test_input_ids, dim=0)
#print(input_ids)
test_attention_masks = torch.cat(test_attention_masks, dim=0)


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.
        )
```

```
from transformers import  BertModel

bert_model = BertModel.from_pretrained('bert-base-uncased')
```

```
bert_model
```

```
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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)


    # 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): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (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): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
```

```
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
```

```
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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: 2.7475    Valid Loss: 2.2492, Duration: 0:00:53.070521
Epoch 2/10, Train Loss: 2.5362    Valid Loss: 2.0960, Duration: 0:00:53.050337
Epoch 3/10, Train Loss: 2.4558    Valid Loss: 1.9681, Duration: 0:00:53.024704
Epoch 4/10, Train Loss: 2.4217    Valid Loss: 1.9456, Duration: 0:00:53.033459
Epoch 5/10, Train Loss: 2.4028    Valid Loss: 1.8900, Duration: 0:00:52.978229
Epoch 6/10, Train Loss: 2.3793    Valid Loss: 1.8418, Duration: 0:00:53.005764
Epoch 7/10, Train Loss: 2.3721    Valid Loss: 1.8189, Duration: 0:00:53.001333
Epoch 8/10, Train Loss: 2.3657    Valid Loss: 1.7925, Duration: 0:00:53.069971
Epoch 9/10, Train Loss: 2.3626    Valid Loss: 1.8083, Duration: 0:00:53.006311
Epoch 10/10, Train Loss: 2.3318    Valid Loss: 1.7937, Duration: 0:00:53.013311
```

```
# 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: 0.5056,       Valid acc: 0.4929,      Test acc: 0.4420
```

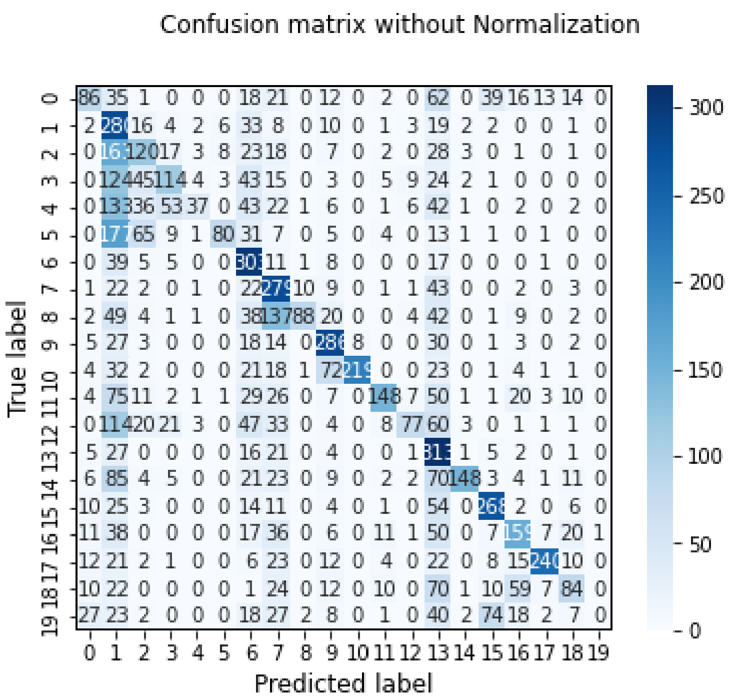
```
# Write a function to get predictions

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([[ 86,  35,   1,   0,   0,   0,  18,  21,   0,  12,   0,   2,   0,
         62,   0,  39,  16,  13,  14,   0],
       [  2, 280,  16,   4,   2,   6,  33,   8,   0,  10,   0,   1,   3,
         19,   2,   2,   0,   0,   1,   0],
       [  0, 163, 120,  17,   3,   8,  23,  18,   0,   7,   0,   2,   0,
         28,   3,   0,   1,   0,   1,   0],
       [  0, 124,  45, 114,   4,   3,  43,  15,   0,   3,   0,   5,   9,
         24,   2,   1,   0,   0,   0,   0],
       [  0, 133,  36,  53,  37,   0,  43,  22,   1,   6,   0,   1,   6,
         42,   1,   0,   2,   0,   2,   0],
       [  0, 177,  65,   9,   1,  80,  31,   7,   0,   5,   0,   4,   0,
         13,   1,   1,   0,   1,   0,   0],
       [  0,  39,   5,   5,   0,   0, 303,  11,   1,   8,   0,   0,   0,
         17,   0,   0,   0,   1,   0,   0],
       [  1,  22,   2,   0,   1,   0,  22, 279,  10,   9,   0,   1,   1,
         43,   0,   0,   2,   0,   3,   0],
       [  2,  49,   4,   1,   1,   0,  38, 137,  88,  20,   0,   0,   4,
         42,   0,   1,   9,   0,   2,   0],
       [  5,  27,   3,   0,   0,   0,  18,  14,   0, 286,   8,   0,   0,
         30,   0,   1,   3,   0,   2,   0],
       [  4,  32,   2,   0,   0,   0,  21,  18,   1,  72, 219,   0,   0,
         23,   0,   1,   4,   1,   1,   0],
       [  4,  75,  11,   2,   1,   1,  29,  26,   0,   7,   0, 148,   7,
         50,   1,   1,  20,   3,  10,   0],
       [  0, 114,  20,  21,   3,   0,  47,  33,   0,   4,   0,   8,  77,
         60,   3,   0,   1,   1,   1,   0],
       [  5,  27,   0,   0,   0,   0,  16,  21,   0,   4,   0,   0,   1,
        313,   1,   5,   2,   0,   1,   0],
       [  6,  85,   4,   5,   0,   0,  21,  23,   0,   9,   0,   2,   2,
         70, 148,   3,   4,   1,  11,   0],
       [ 10,  25,   3,   0,   0,   0,  14,  11,   0,   4,   0,   1,   0,
         54,   0, 268,   2,   0,   6,   0],
       [ 11,  38,   0,   0,   0,   0,  17,  36,   0,   6,   0,  11,   1,
         50,   0,   7, 159,   7,  20,   1],
       [ 12,  21,   2,   1,   0,   0,   6,  23,   0,  12,   0,   4,   0,
         22,   0,   8,  15, 240,  10,   0],
       [ 10,  22,   0,   0,   0,   0,   1,  24,   0,  12,   0,  10,   0,
         70,   1,  10,  59,   7,  84,   0],
       [ 27,  23,   2,   0,   0,   0,  18,  27,   2,   8,   0,   1,   0,
         40,   2,  74,  18,   2,   7,   0]])
```

```
# 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)
```



BERT 512 Features

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
```

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True
```

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

```
# Get last three tokens and truncate head
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []
docoder_sent=[]
before_trunc=[]
maxlen=512
labels = torch.tensor(train.target)
# 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=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    before_trunc.append(encoded_dict['input_ids'])

    ids = encoded_dict['input_ids']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['attention_mask']=torch.tensor([ids])

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

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

    # Get the decoded sentence
```

```
    docoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()))

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
#print(input_ids)
attention_masks = torch.cat(attention_masks, dim=0)
```

```
Token indices sequence length is longer than the specified maximum sequence length fo
```

```
# Get last three tokens and truncate head
# Tokenize all of the sentences and map the tokens to thier word IDs.
test_input_ids = []
test_attention_masks = []
test_decoder_sent=[]
test_before_trunc=[]
maxlen=512
test_labels = torch.tensor(test.target)
# 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=False,
                        padding=False,
                        #max_length = maxlen,           # Pad & truncate all sent
                        return_attention_mask = True,   # Construct attn. masks.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    test_before_trunc.append(encoded_dict['input_ids'])

    ids = encoded_dict['input_ids']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[102]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['input_ids']=torch.tensor([ids])

    ids = encoded_dict['attention_mask']
    if len(ids)>=maxlen:
      ids = ids[0:int(maxlen/2)]+[1]+ids[-int(maxlen/2)+1:]
    else:
      ids = ids + ([0] * (maxlen-len(ids)))
    encoded_dict['attention_mask']=torch.tensor([ids])

    # Add the encoded sentence to the list.
```

```
        test_input_ids.append(encoded_dict['input_ids'])
        #print(input_ids)

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

        # Get the decoded sentence
        test_decoder_sent.append(tokenizer.decode(encoded_dict['input_ids'].squeeze()

    # Convert the lists into tensors.
    test_input_ids = torch.cat(test_input_ids, dim=0)
    #print(input_ids)
    test_attention_masks = torch.cat(test_attention_masks, dim=0)


    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.
        )
```

```
from transformers import  BertModel

bert_model = BertModel.from_pretrained('bert-base-uncased')
```

```
bert_model
```

```
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
```

```
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (pooler): BertPooler(
        (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)


    # 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): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (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): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (11): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (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): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (pooler): BertPooler(
        (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): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (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: 3.0453    Valid Loss: 2.7915, Duration: 0:03:40.208218
Epoch 2/10, Train Loss: 2.9336    Valid Loss: 2.7112, Duration: 0:03:40.178451
Epoch 3/10, Train Loss: 2.9005    Valid Loss: 2.6608, Duration: 0:03:40.188497
Epoch 4/10, Train Loss: 2.8691    Valid Loss: 2.6401, Duration: 0:03:40.153125
Epoch 5/10, Train Loss: 2.8512    Valid Loss: 2.5712, Duration: 0:03:40.138505
Epoch 6/10, Train Loss: 2.8469    Valid Loss: 2.5815, Duration: 0:03:40.169101
Epoch 7/10, Train Loss: 2.8382    Valid Loss: 2.5170, Duration: 0:03:40.138531
Epoch 8/10, Train Loss: 2.8223    Valid Loss: 2.5006, Duration: 0:03:40.147015
Epoch 9/10, Train Loss: 2.8239    Valid Loss: 2.5016, Duration: 0:03:40.169116
Epoch 10/10, Train Loss: 2.8186    Valid Loss: 2.4261, Duration: 0:03:40.133534
```

```
# 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)
```

```
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: 0.3437,      Valid acc: 0.3180,     Test acc: 0.2943
```

```
# 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
```

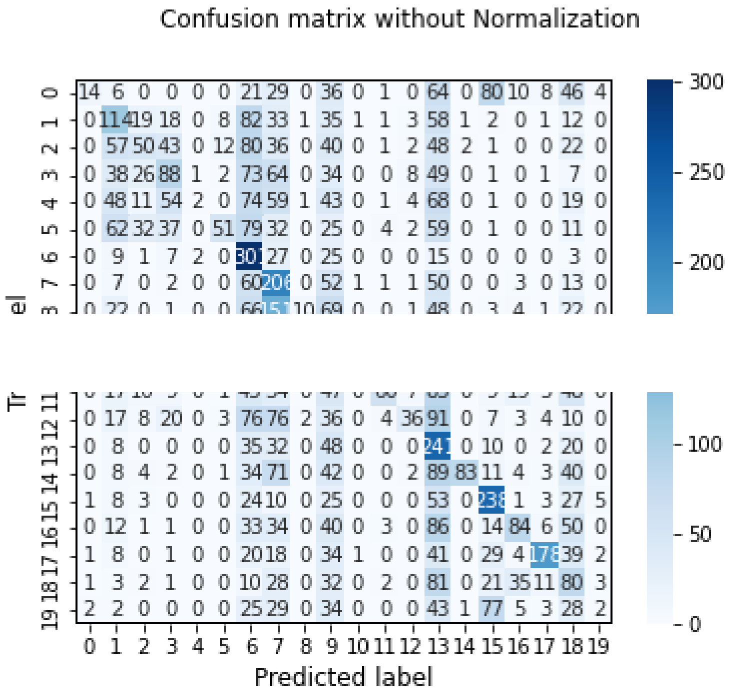
```
y_test, predictions=get_predictions(test_dataloader, model)
```

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

```
array([[ 14,   6,   0,   0,   0,   0,  21,  29,   0,  36,   0,   1,   0,
         64,   0,  80,  10,   8,  46,   4],
       [  0, 114,  19,  18,   0,   8,  82,  33,   1,  35,   1,   1,   3,
         58,   1,   2,   0,   1,  12,   0],
       [  0,  57,  50,  43,   0,  12,  80,  36,   0,  40,   0,   1,   2,
         48,   2,   1,   0,   0,  22,   0],
       [  0,  38,  26,  88,   1,   2,  73,  64,   0,  34,   0,   0,   8,
         49,   0,   1,   0,   1,   7,   0],
       [  0,  48,  11,  54,   2,   0,  74,  59,   1,  43,   0,   1,   4,
         68,   0,   1,   0,   0,  19,   0],
       [  0,  62,  32,  37,   0,  51,  79,  32,   0,  25,   0,   4,   2,
         59,   0,   1,   0,   0,  11,   0],
       [  0,   9,   1,   7,   2,   0, 301,  27,   0,  25,   0,   0,   0,
         15,   0,   0,   0,   0,   3,   0],
       [  0,   7,   0,   2,   0,   0,  60, 206,   0,  52,   1,   1,   1,
         50,   0,   0,   3,   0,  13,   0],
       [  0,  22,   0,   1,   0,   0,  66, 151,  10,  69,   0,   0,   1,
         48,   0,   3,   4,   1,  22,   0],
       [  0,  15,   1,   0,   0,   0,  41,  27,   0, 234,   9,   0,   0,
         42,   0,   4,   1,   5,  17,   1],
       [  0,   4,   0,   1,   0,   0,  30,  43,   0,  93, 145,   0,   1,
         53,   0,   3,   2,   4,  20,   0],
       [  0,  17,  10,   9,   0,   1,  45,  34,   0,  47,   0,  60,   7,
         85,   0,   9,  19,   5,  48,   0],
       [  0,  17,   8,  20,   0,   3,  76,  76,   2,  36,   0,   4,  36,
         91,   0,   7,   3,   4,  10,   0],
       [  0,   8,   0,   0,   0,   0,  35,  32,   0,  48,   0,   0,   0,
```

```
        241,   0,  10,   0,   2,  20,   0],
      [  0,   8,   4,   2,   0,   1,  34,  71,   0,  42,   0,   0,   2,
         89,  83,  11,   4,   3,  40,   0],
      [  1,   8,   3,   0,   0,   0,  24,  10,   0,  25,   0,   0,   0,
         53,   0, 238,   1,   3,  27,   5],
      [  0,  12,   1,   1,   0,   0,  33,  34,   0,  40,   0,   3,   0,
         86,   0,  14,  84,   6,  50,   0],
      [  1,   8,   0,   1,   0,   0,  20,  18,   0,  34,   1,   0,   0,
         41,   0,  29,   4, 178,  39,   2],
      [  1,   3,   2,   1,   0,   0,  10,  28,   0,  32,   0,   2,   0,
         81,   0,  21,  35,  11,  80,   3],
      [  2,   2,   0,   0,   0,   0,  25,  29,   0,  34,   0,   0,   0,
         43,   1,  77,   5,   3,  28,   2]])
```

```
# 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)
```


Confusion matrix without Normalization
Predicted label
True label