Thie notebook explores using CNN for binary text classification using the pytorch library.

In [ ]:

```
from collections import Counter
import nltk
import torch
import torch.nn as nn
import numpy as np
import random
```

In [ ]:

```
def get_batches(x, y, batch_size=12):
    batches_x=[]
    batches_y=[]
    for i in range(0, len(x), batch_size):
        xbatch=x[i:i+batch_size]
        ybatch=y[i:i+batch_size]

        maxlen=max([len(sent) for sent in xbatch])

        # pad sequence with 0's to maximum sequence length within that batch
        for j in range(len(xbatch)):
            xbatch[j].extend([0] * (maxlen-len(xbatch[j])))

        batches_x.append(torch.LongTensor(xbatch))
        batches_y.append(torch.LongTensor(ybatch))

    return batches_x, batches_y
```

In [ ]:

```
PAD_INDEX = 0               # reserved for padding words
UNKNOWN_INDEX = 1           # reserved for unknown words

data_lens = []

def read_embeddings(filename, vocab_size=100000):
    """
  Utility function, loads in the `vocab_size` most common embeddings from `fi

  Arguments:
  - filename:     path to file
                  automatically infers correct embedding dimension from filen
  - vocab_size:   maximum number of embeddings to load

  Returns
  - embeddings:   torch.FloatTensor matrix of size (vocab_size x word_embeddi
  - vocab:        dictionary mapping word (str) to index (int) in embedding m
  """

  # get the embedding size from the first embedding
    with open(filename, encoding="utf-8") as file:
        word_embedding_dim = len(file.readline().split(" ")) - 1

    vocab = {}

    embeddings = np.zeros((vocab_size, word_embedding_dim))
    with open(filename, encoding="utf-8") as file:
        for idx, line in enumerate(file):

            if idx + 1 >= vocab_size:
                break

            cols = line.rstrip().split(" ")
            val = np.array(cols[1:])
            word = cols[0]
            embeddings[idx + 1] = val
            vocab[word] = idx + 1

    return torch.FloatTensor(embeddings), vocab
```

In [ ]:

```
embeddings, vocab=read_embeddings("../data/glove.6B.100d.100K.txt")
```

In [ ]:

```
def read_labels(filename):
    labels={}
    with open(filename) as file:
        for line in file:
            cols = line.split("\t")
            label = cols[0]
            if label not in labels:
                labels[label]=len(labels)
    return labels
```

In [ ]:
```
def read_data(filename, vocab, labels, max_data_points=1000):
    """
    :param filename: the name of the file
    :return: list of tuple ([word index list], label)
    as input for the forward and backward function
    """
    data = []
    data_labels = []
    with open(filename) as file:
        for line in file:
            cols = line.split("\t")
            label = cols[0]
            text = cols[1]
            w_int = []

            if label == 'MWS':

                for w in nltk.word_tokenize(text.lower()):
                    if w in vocab:
                        w_int.append(vocab[w])
                    else:
                        w_int.append(UNKNOWN_INDEX)

                data.append((w_int))
                data_labels.append(labels[label])

            if label == 'HPL':

                label = cols[0]
                text = cols[1]
                w_int = []
                for w in nltk.word_tokenize(text.lower()):
                    if w in vocab:
                        w_int.append(vocab[w])
                    else:
                        w_int.append(UNKNOWN_INDEX)

                data.append((w_int))
                data_labels.append(labels[label])


    # shuffle the data
    tmp = list(zip(data, data_labels))
    random.shuffle(tmp)
    data, data_labels = zip(*tmp)

    if max_data_points is None:
        return data, data_labels

    return data[:max_data_points], data_labels[:max_data_points]
```

In [ ]:
```
# Change this to the directory with your data (from the CheckData_TODO.ipynb
# The directory should contain train.tsv, dev.tsv and test.tsv
directory="../data/spooky"
```

In [ ]:
```
labels=read_labels("%s/train.tsv" % directory)
```

We'll limit the training and dev data to 10,000 data points for this exercise.

In [ ]:
```
trainX, trainY=read_data("%s/train.tsv" % directory, vocab, labels, max_data_
```

In [ ]:
```
devX, devY=read_data("%s/dev.tsv" % directory, vocab, labels, max_data_points
```

In [ ]:
```
testX, testY=read_data("%s/test.tsv" % directory, vocab, labels, max_data_poi
```

In [ ]:
```
batch_trainX, batch_trainY=get_batches(trainX, trainY)
batch_devX, batch_devY=get_batches(devX, devY)
batch_testX, batch_testY=get_batches(testX, testY)
```

In [ ]:

```
class CNNClassifier_bigram(nn.Module):

    """
    CNN with a window size of 2 (i.e., 2grams) and 96 filters

    """
    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96

        self.num_labels = 2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters, self.num_labels)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        # convolution
        x2 = self.conv_2(x0)
        # non-linearity
        x2 = torch.tanh(x2)
        # global max-pooling over the entire sequence
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out
```

In [ ]:

```
class CNNClassifier_unigram_bigram(nn.Module):

    """
    CNN over window sizes of 1 (unigrams) and 2 (bigrams) each 96 filters, whe
    is representated as the concatentation of the 96 ungram filters + 96 bigr

    """
```

```
    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96

        self.num_labels = 2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 1 word
        self.conv_1 = nn.Conv1d(embedding_dim, self.num_filters, 1, 1)

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters*2, self.num_labels)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        # convolution
        x1 = self.conv_1(x0)
        # non-linearity
        x1 = torch.tanh(x1)
        # global max-pooling over the entire sequence
        x1=torch.max(x1, 2)[0]

        x2 = self.conv_2(x0)
        x2 = torch.tanh(x2)
        x2=torch.max(x2, 2)[0]

        combined=torch.cat([x1, x2], dim=1)

        out = self.fc(combined)

        return out
```

In [ ]:
```
def evaluate(model, x, y):
    model.eval()
    corr = 0.
    total = 0.
    with torch.no_grad():
        for x, y in zip(x, y):
            y_preds=model.forward(x)
            for idx, y_pred in enumerate(y_preds):
                prediction=torch.argmax(y_pred)
                if prediction == y[idx]:
                    corr += 1.
                total+=1
    return corr/total
```

In [ ]:
```
def predict(model, x):
    model.eval()
    preds=[]

    with torch.no_grad():
        for batch_x in x:
            y_preds=model.forward(batch_x).numpy()
            for y_pred in y_preds:
                prediction=np.argmax(y_pred)
                preds.append(prediction)

    return preds
```

In [ ]:
```
def train(model, model_filename, train_batches_x, train_batches_y, dev_batche
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1
    losses = []
    cross_entropy=nn.CrossEntropyLoss()

    best_dev_acc=0.

    for epoch in range(5):
        model.train()

        for x, y in zip(train_batches_x, train_batches_y):
            y_pred=model.forward(x)
            loss = cross_entropy(y_pred.view(-1, 2), y.view(-1))
            losses.append(loss)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        dev_accuracy=evaluate(model, dev_batches_x, dev_batches_y)

        # we're going to save the model that performs the best on *dev* data
        if dev_accuracy > best_dev_acc:
            torch.save(model.state_dict(), model_filename)
            print("%.3f is better than %.3f, saving model ..." % (dev_accurac
            best_dev_acc = dev_accuracy
        if epoch % 1 == 0:
            print("Epoch %s, dev accuracy: %.3f" % (epoch, dev_accuracy))

    model.load_state_dict(torch.load(model_filename))
    print("\nBest Performing Model achieves dev accuracy of : %.3f" % (best_d
```

First, let's examine the performance of a CNN that only has access to bigram features (from a CNN window size of 2)

In [ ]:
```
cnn_model = CNNClassifier_bigram(pretrained_embeddings=embeddings)
train(cnn_model, "cnn.bigram.model", batch_trainX, batch_trainY, batch_devX,
```

```
0.834 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.834
0.844 is better than 0.834, saving model ...
Epoch 1, dev accuracy: 0.844
0.849 is better than 0.844, saving model ...
Epoch 2, dev accuracy: 0.849
0.857 is better than 0.849, saving model ...
Epoch 3, dev accuracy: 0.857
Epoch 4, dev accuracy: 0.856

Best Performing Model achieves dev accuracy of : 0.857
```

Now let's add unigram features to the bigram features.

In [ ]:

```
cnn_model = CNNClassifier_unigram_bigram(pretrained_embeddings=embeddings)
train(cnn_model, "cnn.unigram_bigram.model", batch_trainX, batch_trainY, batc
```

```
0.846 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.846
0.855 is better than 0.846, saving model ...
Epoch 1, dev accuracy: 0.855
0.864 is better than 0.855, saving model ...
Epoch 2, dev accuracy: 0.864
0.866 is better than 0.864, saving model ...
Epoch 3, dev accuracy: 0.866
0.867 is better than 0.866, saving model ...
Epoch 4, dev accuracy: 0.867

Best Performing Model achieves dev accuracy of : 0.867
```

Q1: Experiment with the network structure that works best for your binary classification dataset. Explore the following choices: a.) the order of ngrams (window size); b.) the number of filters; c.) the activation functions; d.) the use of dropout. Which architecture performs best on the **development data**? (Remember, never optimize this choice on your test data!) Create 5 different models and execute them below.

In [ ]:

```
class CNN_model1(nn.Module):

    """
    Use this bigram CNN as a starting point for developing your own custom mo
    
    """

    # 96*2 filters, 3-2 biagrams, tanh, dropout 0.25

    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96*2

        self.num_labels = 2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 3, 2)

        self.fc = nn.Linear(self.num_filters, self.num_labels)

        # Define proportion or neurons to dropout
        self.dropout = nn.Dropout(0.25)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        x2 = self.conv_2(x0)
        x2 = torch.tanh(x2)
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out

cnn_model1 = CNN_model1(pretrained_embeddings=embeddings)
train(cnn_model1, "cnn.1.model", batch_trainX, batch_trainY, batch_devX, batc
```

```
0.822 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.822
0.828 is better than 0.822, saving model ...
Epoch 1, dev accuracy: 0.828
Epoch 2, dev accuracy: 0.828
Epoch 3, dev accuracy: 0.815
Epoch 4, dev accuracy: 0.822

Best Performing Model achieves dev accuracy of : 0.828
```

In [ ]:

```
class CNN_model2(nn.Module):

    """
    Use this bigram CNN as a starting point for developing your own custom mo

    """

    # 96 filters, 2-1 biagrams, relu, dropout 0.25

    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96

        self.num_labels = 2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters, self.num_labels)

        # Define proportion or neurons to dropout
        self.dropout = nn.Dropout(0.25)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        x2 = self.conv_2(x0)
        x2 = torch.relu(x2)
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out

cnn_model2 = CNN_model2(pretrained_embeddings=embeddings)
train(cnn_model2, "cnn.2.model", batch_trainX, batch_trainY, batch_devX, batc
```

```
0.833 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.833
0.850 is better than 0.833, saving model ...
Epoch 1, dev accuracy: 0.850
0.860 is better than 0.850, saving model ...
Epoch 2, dev accuracy: 0.860
0.869 is better than 0.860, saving model ...
Epoch 3, dev accuracy: 0.869
Epoch 4, dev accuracy: 0.868

Best Performing Model achieves dev accuracy of : 0.869
```

In [ ]:
```
class CNN_model3(nn.Module):

    """
    Use this bigram CNN as a starting point for developing your own custom mo

    """

    # 96 filters, 2-1 biagrams, tanh, dropout 0.15

    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96

        self.num_labels = 2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters, self.num_labels)

        # Define proportion or neurons to dropout
        self.dropout = nn.Dropout(0.15)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        x2 = self.conv_2(x0)
        x2 = torch.tanh(x2)
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out

cnn_model3 = CNN_model3(pretrained_embeddings=embeddings)
train(cnn_model3, "cnn.3.model", batch_trainX, batch_trainY, batch_devX, batc
```

```
0.830 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.830
0.844 is better than 0.830, saving model ...
Epoch 1, dev accuracy: 0.844
0.847 is better than 0.844, saving model ...
Epoch 2, dev accuracy: 0.847
0.852 is better than 0.847, saving model ...
Epoch 3, dev accuracy: 0.852
Epoch 4, dev accuracy: 0.852

Best Performing Model achieves dev accuracy of : 0.852
```

In [ ]:
```
class CNN_model4(nn.Module):

    """
    Use this bigram CNN as a starting point for developing your own custom mo

    """

    # 96 filters, 2-1 biagrams, tanh, dropout 0.45

    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=96

        self.num_labels=2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters, self.num_labels)
        self.dropout = nn.Dropout(0.45)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        x2 = self.conv_2(x0)
        x2 = torch.relu(x2)
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out

cnn_model4 = CNN_model4(pretrained_embeddings=embeddings)
train(cnn_model4, "cnn.4.model", batch_trainX, batch_trainY, batch_devX, batc
```

```
0.832 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.832
0.857 is better than 0.832, saving model ...
Epoch 1, dev accuracy: 0.857
0.863 is better than 0.857, saving model ...
Epoch 2, dev accuracy: 0.863
Epoch 3, dev accuracy: 0.859
Epoch 4, dev accuracy: 0.863

Best Performing Model achieves dev accuracy of : 0.863
```

In [ ]:

```
class CNN_model5(nn.Module):

    """
    Use this bigram CNN as a starting point for developing your own custom mo
    """

    # 96 filters, 2-1 biagrams, tanh, dropout 0.45

    def __init__(self, pretrained_embeddings):
        super().__init__()

        self.num_filters=80

        self.num_labels=2

        _, embedding_dim=pretrained_embeddings.shape

        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings,

        # convolution over 2 words
        self.conv_2 = nn.Conv1d(embedding_dim, self.num_filters, 2, 1)

        self.fc = nn.Linear(self.num_filters, self.num_labels)
        self.dropout = nn.Dropout(0.45)


    def forward(self, input):

        # batch_size x max_seq_length x embeddings_size
        x0 = self.embeddings(input)

        # batch_size x embeddings_size x max_seq_length
        # (the input order expected by nn.Conv1d)
        x0 = x0.permute(0, 2, 1)

        x2 = self.conv_2(x0)
        x2 = torch.relu(x2)
        x2=torch.max(x2, 2)[0]

        out = self.fc(x2)

        return out

cnn_model5 = CNN_model5(pretrained_embeddings=embeddings)
train(cnn_model5, "cnn.5.model", batch_trainX, batch_trainY, batch_devX, batc
```

```
0.831 is better than 0.000, saving model ...
Epoch 0, dev accuracy: 0.831
0.854 is better than 0.831, saving model ...
Epoch 1, dev accuracy: 0.854
0.856 is better than 0.854, saving model ...
Epoch 2, dev accuracy: 0.856
0.857 is better than 0.856, saving model ...
Epoch 3, dev accuracy: 0.857
Epoch 4, dev accuracy: 0.856

Best Performing Model achieves dev accuracy of : 0.857
```

We can generate predictions for a given test set with the `predict` function:

In [ ]:

```
# gold data for test
gold=[]
for batchY in batch_testY:
    gold.extend(batchY)

# prediction data for test
model=cnn_model4
predictions=predict(model, batch_testX)
```

Q2: For the single model that performed best on the dev data (that you identified in Q1 above), calculate its 95% confidence intervals for accuracy on the **test data**.

In [ ]:

```
import pandas as pd
def accuracy(truth, predictions):
    correct=0.
    for idx in range(len(truth)):
        g=truth[idx]
        p=predictions[idx]
        if g == p:
            correct+=1
    return correct/len(truth)

def bootstrap(gold, predictions, metric, B=10000, confidence_level=0.95):
    #inspired by https://machinelearningmastery.com/calculate-bootstrap-confi
    bs_statistics = []
    df = pd.DataFrame([predictions, gold], index=['predictions', 'truth']).T
    for i in range(0, B):
        sample = df.sample(frac=1, replace=True)
        stat = metric(np.array(sample['truth']), np.array(sample['predictions
        bs_statistics.append(stat)
    bs_ordered = sorted(bs_statistics)
    lower = np.percentile(np.array(bs_ordered), (1-confidence_level)/2)
    median = np.percentile(np.array(bs_ordered), 0.5)
    upper = np.percentile(np.array(bs_ordered), confidence_level+((1-confiden

    return lower, median, upper
```

In [ ]:

```
bootstrap(gold, predictions, accuracy)
```

Out[ ]:

```
(0.8261096723044398, 0.8361522198731501, 0.8393234672304439)
```