# COMP0090 2020/21 Assignment 2: “Half-bag”

**Sahil Shah**
20194624
sahil.shah.20@ucl.ac.uk

**Akshaya Natarajan**
20069959
akshaya.natarajan.20@ucl.ac.uk

**Kamiylah Charles**
20092484
kamiylah.charles.20@.ucl.ac.uk

**Akshay Parmar**
20153279
akshay.parmar.20@ucl.ac.uk

**Chanel Sadrettin-Brown**
16050121
chanel.sadrettin-brown.20@ucl.ac.uk

## 1 Contributions

Each individual in the group did a question to make it easier working remotely. Chanel did 1, Kamiylah did 2, Akshay did 3, Sahil and Akshaya worked on 4 and 5 together. From time to time the group came together to help each other on questions and check over each others work. All members of the group have used PyTorch for this assignment.

## 2 Project Code

All project code on Google Colab can be found here:

https://colab.research.google.com/drive/1vpqkAapPZXkZw-xH4hzW6GnjeDNMDhHx?usp=sharing

## Contents

## 3 Tasks

### 3.1 A multi-class, multi-layer perceptron

**Problem set-up:** the Fashion-MNIST data set consists of $60,000$ training images and labels, and $10,000$ test images and labels. It contains images of clothing items and consists of 10 unique classes to classify these items, given in table 1.

Table 1: Item per class

| Number | Item |
|---|---|
| 0 | 'T-shirt/top' |
| 1 | 'Trouser' |
| 2 | 'Pullover' |
| 3 | 'Dress' |
| 4 | 'Coat' |
| 5 | 'Sandal' |
| 6 | 'Shirt' |
| 7 | 'Sneaker' |
| 8 | 'Bag' |
| 9 | 'Ankle boot' |

**Code to load and split the data:**

As aforementioned the data is given as a training and test set, the first step of this question was to split the training set into two to produce a validation set of $10,000$ images and labels.

```
## importing packages ##

import torch
import numpy as np
import torchvision
import torch.nn as nn
import torch.optim as optim
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix,
    classification_report
import seaborn as sns


def load_data(batch_size = 64):
    ## loading and transforming the data for use in pytorch ##


    transform = transforms.ToTensor()
    trainset = datasets.FashionMNIST('Fashion_MNIST', download
        = True, train = True, transform = transform)
    testset = datasets.FashionMNIST('Fashion_MNIST', download =
         True, train = False, transform = transform)

    # create an array of all indices in training set (60,000
        images) and shuffle them to make split unbiased
    indices = np.arange(60000)
    np.random.shuffle(indices)
    # load the datasets using batch size defined (50,000:10,000
        for train:validation), only shuffle the test data
    train_load = torch.utils.data.DataLoader(trainset,
       batch_size=batch_size, shuffle=False, sampler=torch.
       utils.data.SubsetRandomSampler(indices[:50000]))
    validation_load = torch.utils.data.DataLoader(trainset,
       batch_size=batch_size, shuffle=False, sampler=torch.
       utils.data.SubsetRandomSampler(indices[50000:]))
    test_load = torch.utils.data.DataLoader(testset, batch_size
        = batch_size, shuffle = True)

    return trainset, testset, train_load, validation_load,
       test_load
```

**Code to view images:**

In order to establish that the data loading step was successful, the shapes of the tensors and an example set of images were printed. This produced figure 1.

```
# checking data loaded correctly

trainset, testset, train_load, validation_load, test_load =
    load_data()

# check that the data has been loaded correctly by testing the
    shape of the train/test sets
train_shape = train_load.dataset.data.shape
print(train_shape)
test_shape = test_load.dataset.data.shape
print(test_shape)

# check that split and batches are working
print(len(train_load))
print(len(validation_load))
print(len(test_load))

data = iter(train_load)
images, labels = data.next()

item_classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
    'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# display some random images in training set

for i in range(10):
    plt.subplot(2,5,i+1)
    plt.imshow(images[i][0], cmap='gray')
    plt.title(item_classes[labels[i].item()])
plt.show()

# checking that the image and label have the right dimension,
    given the batch size

print(images.shape)
print(labels.shape)
```

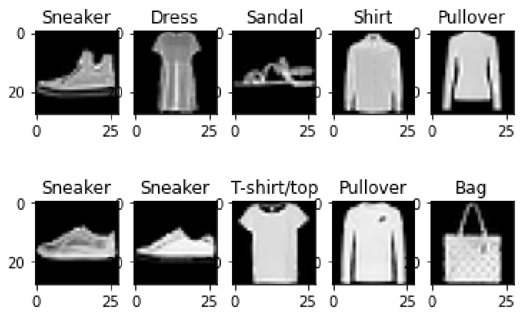


Figure 1: Fashion-MNIST random sample images

### 3.1.1 Implement a multi-class, multi-layer perceptron with cross-entropy loss for the Fashion- MNIST data

PyTorch was used to produce automatic forward and backward propagation and parameter updates, based on specified network architecture and hyperparameters such as layer size, number of layers, batch size and learning rate. In order to train the model and evaluate it on the validation set, the function trainvalidate() was created (shown in the code below).

The loss function as specified in the question was cross-entropy loss, using this in PyTorch means that it automatically gives the final layer of the MLP a softmax activation function. ReLU was chosen for the hidden layer activation functions as it has a reduced likelihood of the vanishing gradient problem when compared to other activation functions such as logistic sigmoid; it is also more computationally efficient.

The optimiser chosen was Adam, as from prior experience of using it on neural network tasks it has performed better than stochastic gradient descent (SGD). Adam combines AdaGrad and RMSProp, two extensions of SGD. The algorithm not only makes use of the average first moment (mean) as in RMSProp when adapting parameter learning rates, but it makes use of the average second moment (uncentred variance) as in AdaGrad.*Adam Optimiser*(ada, 2017).

The initial hyper parameters used for creating the model were as follows: number of layers = 2, depth of layers = 200, learning rate = 0.001, batch size = 64, number of epochs = 10. These choices were based on judgement and had not been optimised, they were test that the model worked. The results from using these parameters are given in figure 2, with: training loss = 0.2620 , training accuracy = 90.40%, validation loss = 0.3078, validation accuracy = 88.43%. This model was later improved by tuning the hyperparameters by a grid search method.

```
## initialise parameters ##

in_size = 784 # input size is flattened vector, 28x28
h_size = 200 # hidden layer size
L = 1 # number of layers
classes = 10 # number of classes

## create MLP and define forward propagation ##

class MLP(nn.Module):
    def __init__(self, in_size, h_size, classes, L):
        super(MLP, self).__init__()
        self.hidden_layers = []
        self.L = L
        self.l1 = nn.Linear(in_size, h_size)
        for i in range(self.L):
            self.hidden_layers.append(nn.Linear(h_size, h_size)
                )
        self.l2 = nn.Linear(h_size, classes)

    def forward(self, x):
        x = torch.relu(self.l1(x))
        for i in range(self.L):
            x = torch.relu(self.hidden_layers[i](x))
        x = self.l2(x)
        return x

MLP_model = MLP(in_size, h_size, classes, L)

## loss function and optimizer ##

#using cross entropy loss as specified in question, this will
    automatically compute softmax at the output layer
# to find the probability of image belonging to each of the 10
    classes
loss_func = nn.CrossEntropyLoss()

def optim(learning_rate = 0.001):
    # use the ADAM optimiser
    optimizer = torch.optim.Adam(MLP_model.parameters(), lr=
    learning_rate)
    return optimizer


## training the model and testing it on the validation set ##

# iterating over epochs and images, forward pass, calculate
   loss, backward pass, optimizer step for updates.
# with each epoch the validation set will be run through the
   model

def train_validate(epochs = 10):

   # calculing relevant functions

    trainset, testset, train_load, validation_load, test_load =
         load_data()
    optimizer = optim()



    loss_train = []
    acc_train = []
    loss_val = []
    acc_val = []

    for epoch in range(epochs):

        # initialising empty lists to help print and plot
        progress per epoch

        epoch_loss_train= []
        epoch_acc_train = []
        epoch_loss_val = []
        epoch_acc_val = []
```

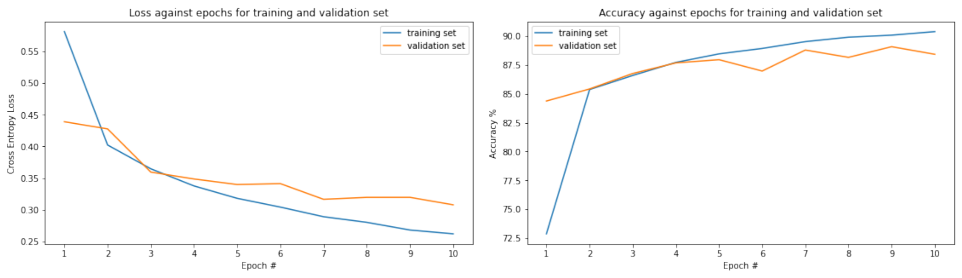
```
correct_t = 0
total_t = 0
correct_v = 0
total_v = 0

# training the model

for i, (images, labels) in enumerate(train_load):

    MLP_model.train()

    images = images.reshape(-1, 784) # flatten image
       tensor to be a vector of 28x28 features, with
       batch_size examples (i.e (batch_size, 784))

    # Forward pass
    outputs = MLP_model(images)
    loss_t = loss_func(outputs, labels)

    # Backward and optimize
    optimizer.zero_grad()
    loss_t.backward()
    optimizer.step()

    epoch_loss_train.append(loss_t.item()) # appending
       loss to list

    _, predict = torch.max(outputs.data, 1) # convert
       output to probability
    total_t += labels.size(0) # total number of
       classifications made
    correct_t += (predict == labels).sum().item() #
       number of correct classifications

    epoch_acc_train.append(100.0 * correct_t / total_t)
        # calculate accuracy for each epoch


# evaluate trained model on validation set per epoch

for j, (images, labels) in enumerate(validation_load):

    MLP_model.eval()

    images = images.reshape(-1, 784)

    outputs = MLP_model(images)
    loss_v = loss_func(outputs, labels)
    epoch_loss_val.append(loss_v.item())

    _, predict = torch.max(outputs.data, 1)
    total_v += labels.size(0) # total number of samples
    correct_v += (predict == labels).sum().item() #
       total number of samples classified correctly
    epoch_acc_val.append(100.0 * correct_v / total_v)


loss_train.append(sum(epoch_loss_train)/len(
   epoch_loss_train)) # loss per epoch train set
```

```
    acc_train.append(sum(epoch_acc_train)/len(
        epoch_acc_train)) # accuracy per epoch train set
    loss_val.append(sum(epoch_loss_val)/len(epoch_loss_val)
        ) # loss per epoch validation set
    acc_val.append(sum(epoch_acc_val)/len(epoch_acc_val)) #
         loss per epoch validation set

    # printing the loss and accuracy for training and
        validation per epoch

    print(f'Epoch: {epoch +1}/{epochs}, Training Loss: {
        loss_train[epoch]:.4f}, Accuracy:{acc_train[epoch
        ]:.2f}% and Validation Loss:{loss_val[epoch]:.4f},
        Accuracy: {acc_val[epoch]:.2f}%')


# plotting loss per epoch for the training and validation
    sets

plt.figure(figsize=(9,4.8))
plt.plot(np.linspace(1, epochs, epochs).astype(int),
    loss_train, label = 'training set')
plt.plot(np.linspace(1, epochs, epochs).astype(int),
    loss_val, label ='validation set')
plt.legend()
plt.title('Loss against epochs for training and validation
    set')
plt.xticks(np.arange(1, epochs+1))
plt.xlabel('Epoch #')
plt.ylabel('Cross Entropy Loss')
plt.show()

# plotting accuracy per epoch for the training and
    validation sets

plt.figure(figsize=(9,4.8))
plt.plot(np.linspace(1, epochs, epochs).astype(int),
    acc_train, label = 'training set')
plt.plot(np.linspace(1, epochs, epochs).astype(int),
    acc_val, label ='validation set')
plt.legend()
plt.title('Accuracy against epochs for training and
    validation set')
plt.xticks(np.arange(1, epochs + 1))
plt.xlabel('Epoch #')
plt.ylabel('Accuracy %')
plt.show()

# printing final loss and accuracy on training and
    validation sets

print(f'Final accuracy on training set is {acc_train[-1]:.2
    f}% and final loss is {loss_train[-1]:.4f}')
print(f'Final accuracy on validation set is {acc_val[-1]:.2
    f}% and final loss is {loss_val[-1]:.4f}')

return acc_val
```

**Output for the initial model:**

The results for training the initial model are below. Figure 2 shows the loss and accuracy plots for the training and validation sets.

```
Output:
Epoch: 10/10, Training Loss: 0.2620, Accuracy:90.40% and
    Validation Loss:0.3078, Accuracy: 88.43%
```



(a) Training and validation loss (b) Training and validation accuracy

Figure 2: Initial model training and validation

### 3.1.2 Explore model variants with different layer sizes, depths, etc. to find what works well on the validation set and describe the process through which you arrive at your final model

In order to find the optimal model the combination of hyperparameters that perform best on the validation set, the following method was used:

- The initial model was run for a range of different number of epochs: 10, 30 and 50, it was established that 30 epochs was a reasonable for the model to converge
- The initial model was run for a range of different batch sizes: 32, 64, and 128, it was established that a batch size of 64 performed best
- A grid search was created that loops over a range of different layer depth: 200, 500 and 1000, number of layers in network: 2, 3 and 4 and learning rates: 0.0001, 0.001 and 0.01 using 30 epochs and a batch size of 64
- The range of values selected for the grid search were picked on the basis of previous experience of training similar models
- The grid search returned the combination of hyperparameters that had the highest accuracy on the validation set

The highest validation accuracy achieved through grid search was 90.27%, this occurred with batch size = 64, number of epochs = 30, number of layers = 3, layer depth = 500 and learning rate = 0.0001.

**Code for grid search:**

```
#Altering: # layers, width of layers and learning_rate

#use two lists one to store final validation accuracy per
    combination and the other to store that combination of
    hyper parameters
#loop over the lists of parameters given below
#then use argmax on the final validation accuracy per
    combination, find out the overall highest val accuracy
    index
# use the index find the corresponding combination in the other
     list
```

```
#epochs = 30
#batchsize = 64
layersize = [200, 500, 100]
learningrate = [0.0001, 0.001, 0.01]
numlayers = [2, 3, 4]


final_validation = []
validation_accuracy_v = []

for h in numlayers:
    #for i in batchsize:
  for j in layersize:
    for k in learningrate:
    #for l in epoch_list:

      print(f'number of layers = {h}, layer size = {j},
         learning rate = {k}')

      load_data(batch_size = 64) # load data with specified
         batch size
      MLP_model = MLP(in_size, j, classes,h) # set up model and
          forward prop with specified layer size
      optim(learning_rate = k) # set up optimiser with
         specified learning rate
      acc_val = train_validate(epochs = 30) # train model with
         specified epochs

      print(f'Final validation acuracy with current parameters:
          {acc_val[-1]:2f}%')
      final_validation.append(acc_val[-1]) # store final
         validation accuracy for current combination

      validation_accuracy_v.append([h,j,k]) # store current
         combination


highest_val_index = np.argmax(final_validation)
highest_val_accuracy = final_validation[highest_val_index]
optimal_params = validation_accuracy_v[highest_val_index]

print(f'The highest validation accuracy {highest_val_accuracy}
   was achieved with: batch_size = 64, number_epochs = 30,
   number_layers = {optimal_params[0]} , layer_size = {
   optimal_params[1]}, learning_rate = {optimal_params[2]}')

output: The highest validation accuracy 90.2669866264288 was
   achieved with: batch_size = 64, number_epochs = 30,
   number_layers = 3 , layer_size = 500, learning_rate =
   0.0001
```

#### 3.1.3 Provide a plot of the loss on the training set and validation set for each epoch of training

The model was trained using the optimal parameters and evaluated on the validation set, figure 3 shows a plot of the training and validation loss per epoch and figure 4 shows a plot of the training and validation accuracy per epoch.

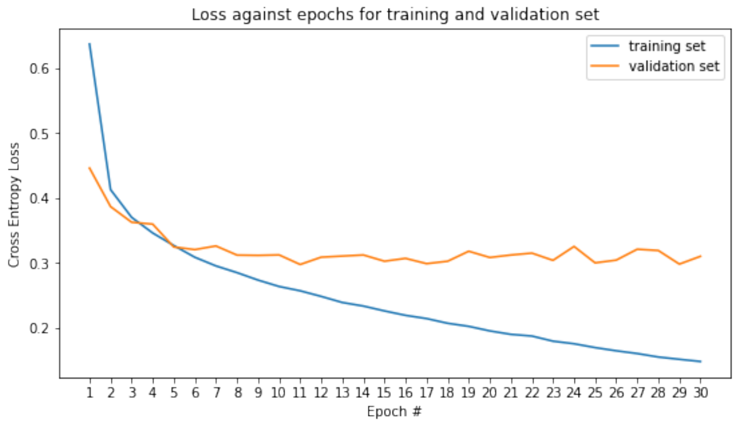


Figure 3: Training and validation loss - final model

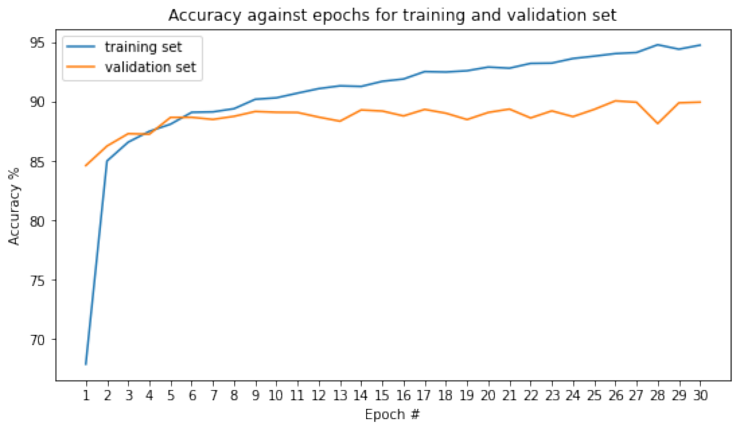


Figure 4: Training and validation accuracy - final model

**Code for the training, validation and testing of the final model:**

The code and output for training, validation and testing is below. The code for the test set is similar to that for evaluating the validation set, it was altered in order to produce the confusion matrix and remove the loss calculation.

```
## using the optimal parameter model on test set to classify
   unseen data and plot the confusion matrix ##

def test():
    trainset, testset, train_load, validation_load, test_load =
        load_data()

    with torch.no_grad():
        MLP_model.eval()
        correct = 0
        total = 0
        prediction = []
        label_all = []

        for images, labels in test_load:
            images = images.reshape(-1, 784)
            labels = labels
            outputs = MLP_model(images)
            loss = loss_func(outputs, labels)

            # max returns (value ,index)
            _, predict = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predict == labels).sum().item()
            prediction += predict
            label_all += labels
            #print(predict,labels)

        confusion = confusion_matrix(label_all,  prediction) #
            produce confusion matrix

        plt.subplots(figsize = (8,6)) # plot heatmap of
            confusion matrix
        ax = sns.heatmap(confusion, annot=True, fmt = 'g', cmap
             = 'Blues')
        ax.set_xlabel('Predicted')
        ax.set_ylabel('Actual')


        #print(classification_report(label_all, prediction)) #
            produce classification report


        acc = 100.0 * correct / total
        print(f'Accuracy on test set: {acc:.2f} %')

        print(f'Labels: {item_classes}')


## training the model on the optimal parameters, applying model
     to the test set and producing confusion matrix ##

#optimal parameters: batch size = 64, number of epochs = 30,
    number of layers = 3, layer size = 500 and learning rate =
    0.0001

load_data(batch_size = 64) # load data with specified batch
    size
MLP_model = MLP(in_size, 500, classes, 3) # set up model and
    forward prop with specified layer size
optim(learning_rate = 0.0001) # set up optimiser with specified
     learning rate
train_validate(epochs = 30) # train model with specified epochs
test() # use model on the test set

output:
Final accuracy on training set is 94.78% and final loss is
    0.1483
Final accuracy on validation set is 89.97% and final loss is
    0.3104
Accuracy on test set: 88.97 %
```

### 3.1.4 Provide the final accuracy on the training, validation, and test set

- Final accuracy on training set = 94.78%

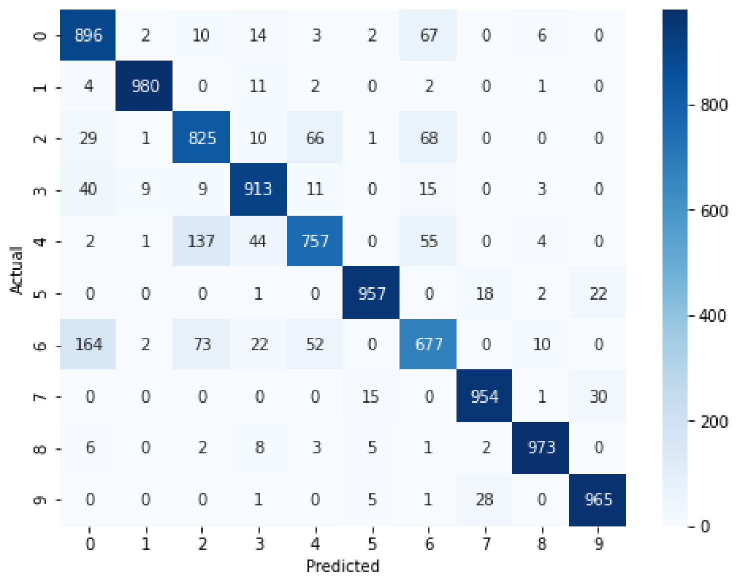
- Final accuracy on validation set = 89.97%
- Final accuracy on test set = 88.97 %

#### 3.1.5 Analyse the errors of your models by constructing a confusion matrix. Which classes are easily “confused” by the model? Hypothesise why

Figure 5 shows the confusion matrix of the classifications of the test set. Table 1 shows the class index labels. The entries on the diagonal show where the predicted matched the label, all other entries show misclassifications.

The model is relatively accurate at classifying the images correctly, however certain classes are more easily confused than others. The hypothesis is that this occurs when images are of visually similar items. Classes 0 and 6 are the ’most confused’ out of any other pair, 0 is a ’T-shirt/Top’ and 6 is a ’Shirt’, both are similar looking items so will have a similar pixel structure and as such it may be difficult for the model to notice differences between them. Classes 2 (’Pullover’) and 4 (’Coat’) are quite ’confused’, this is again because they are similar looking items. The model seems to get confused between classes 2 and 6, and classes 4 and 6 for the same reasoning, the visual similarity between these items is that they all are types of top. It also occasionally gets classes 7 (’Sneaker’) and 9 (’Ankle boot’) confused, most likely because they are both types of shoe.

Labels: [0 = ’T-shirt/top’, 1 = ’Trouser’, 2 = ’Pullover’, 3 = ’Dress’, 4 = ’Coat’, 5 = ’Sandal’, 6 = ’Shirt’, 7 = ’Sneaker’, 8 = ’Bag’, 9 = ’Ankle boot’]



| Actual \ Predicted | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 896 | 2 | 10 | 14 | 3 | 2 | 67 | 0 | 6 | 0 |
| 1 | 4 | 980 | 0 | 11 | 2 | 0 | 2 | 0 | 1 | 0 |
| 2 | 29 | 1 | 825 | 10 | 66 | 1 | 68 | 0 | 0 | 0 |
| 3 | 40 | 9 | 9 | 913 | 11 | 0 | 15 | 0 | 3 | 0 |
| 4 | 2 | 1 | 137 | 44 | 757 | 0 | 55 | 0 | 4 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 957 | 0 | 18 | 2 | 22 |
| 6 | 164 | 2 | 73 | 22 | 52 | 0 | 677 | 0 | 10 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 954 | 1 | 30 |
| 8 | 6 | 0 | 2 | 8 | 3 | 5 | 1 | 2 | 973 | 0 |
| 9 | 0 | 0 | 0 | 1 | 0 | 5 | 1 | 28 | 0 | 965 |

Figure 5: Confusion matrix

## 3.2 Denoising autoencoder

**Problem Set Up**: As with task 3.1, the data being used to answer this question is the Fashion-MNIST data set. Due to the nature of this problem, only the data concerning images were used. This included 60,000 training images, which were split into a training and validation set, consisting of 50,000 and 10,000 images respectively, and 10,000 test images used as the test set.

First a package (idx2numpy) was used to load in the data saved as arrays, and the rest of the required packages were also loaded in.

```
!pip install idx2numpy
import idx2numpy

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# importing the training images and splitting it into a
    training and validation set
train_data = idx2numpy.convert_from_file('train-images-idx3-
    ubyte')
x_train = train_data[:50000,:]
x_train = x_train.copy()

x_val = train_data[50000:,::]
x_val = x_val.copy()

x_test = idx2numpy.convert_from_file('t10k-images-idx3-ubyte')
x_test = x_test.copy()
```

A noisy version of each data split was created using the noisy function given.

```
# making the images noisy
t = [0,1]
def noisy(x):
  # remove one or two quadrants of the image
  for _ in t:
    xoffset = np.random.choice([0,14])
    yoffset = np.random.choice([0,14])
    interval1 = range(xoffset, xoffset+14)
    interval2 = range(yoffset, yoffset+14)
    for i in interval1:
      for j in interval2:
        x[i, j] = 0

  return x

x_train_shaped = x_train.reshape(50000,784)
x_train_noisy = np.zeros((x_train_shaped.shape))
for i in range(x_train_shaped.shape[0]):
  noisied = noisy(x_train_shaped[i,:].T.reshape(28,28))
  x_train_noisy[i, :] = noisied.reshape(784,)

x_val_shaped = x_val.reshape(10000,784)
x_val_noisy = np.zeros((x_val_shaped.shape))
```

```
for i in range(x_val_shaped.shape[0]):
  noisied = noisy(x_val_shaped[i,:].T.reshape(28,28))
  x_val_noisy[i, :] = noisied.reshape(784,)

x_test_shaped = x_test.reshape(10000,784)
x_test_noisy = np.zeros((x_test_shaped.shape))
for i in range(x_test_shaped.shape[0]):
  noisied = noisy(x_test_shaped[i,:].T.reshape(28,28))
  x_test_noisy[i, :] = noisied.reshape(784,)

# reloading the data so we get the clean images back
train_data = idx2numpy.convert_from_file('train-images-idx3-
    ubyte')
x_train = train_data[:50000,::]
x_val = train_data[50000:,::]
x_test = idx2numpy.convert_from_file('t10k-images-idx3-ubyte')
```

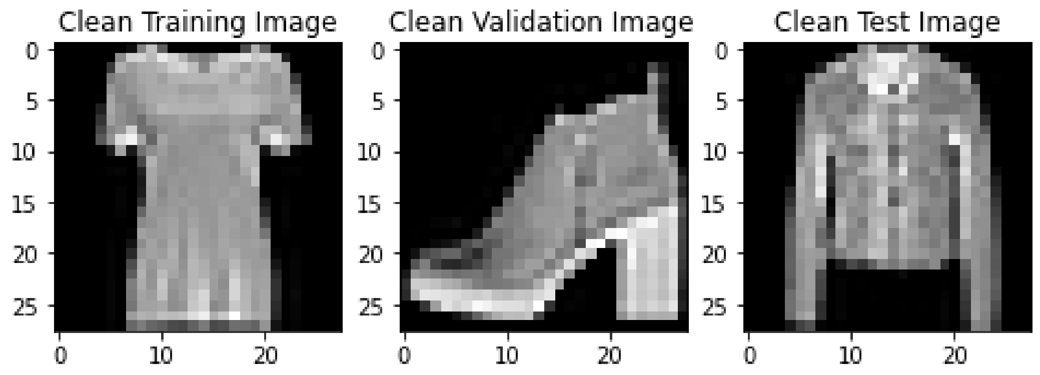


Figure 6: Example of clean training, validation and set test image respectively.

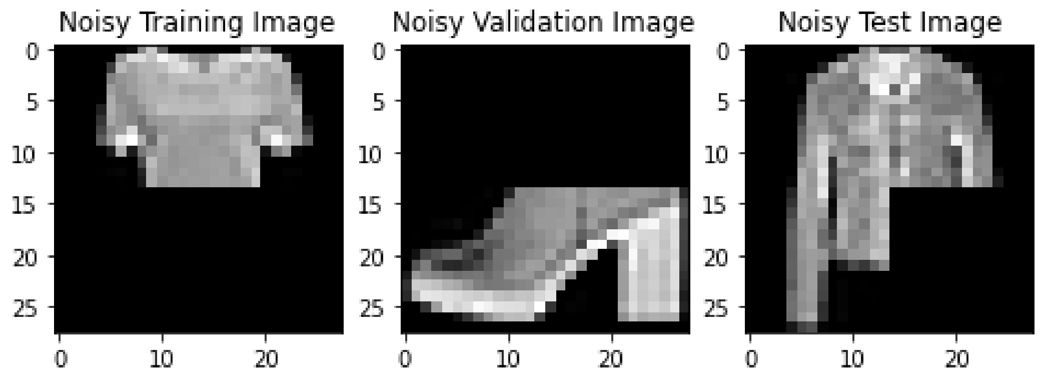


Figure 7: Example of noisy training, validation and set test image respectively.

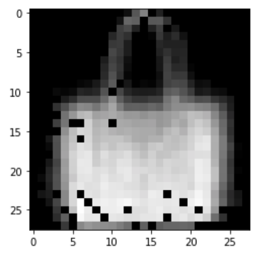
All the data were then converted from arrays to tensors.

```
# converting the data into tensors
x_train_tensor = torch.tensor(x_train.reshape(50000,28*28),
    dtype=torch.float)
x_val_tensor = torch.tensor(x_val.reshape(10000,28*28), dtype=
    torch.float)
x_test_tensor = torch.tensor(x_test.reshape(10000,28*28), dtype
    =torch.float)
```

```
x_train_noisy_tensor = torch.tensor(x_train_noisy.reshape
    (50000,28*28), dtype=torch.float)
x_val_noisy_tensor = torch.tensor(x_val_noisy.reshape
    (10000,28*28), dtype=torch.float)
x_test_noisy_tensor = torch.tensor(x_test_noisy.reshape
    (10000,28*28), dtype=torch.float)
```

#### 3.2.1 Implement a denoising autoencoder with mean squared error loss for the Fashion-MNIST data.

```
# putting the noisy and clean training images into one data set
batch_size = 64
trainset = torch.utils.data.TensorDataset(x_train_noisy_tensor,
     x_train_tensor)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=
    batch_size, shuffle=True, num_workers=2)

# checking implementation
class AutoEncoder(nn.Module):
  def __init__(self):
    super().__init__()
    self.en1 = nn.Linear(28*28, 256)
    self.en2 = nn.Linear(256,128)
    self.en3 = nn.Linear(128,64)

    self.dec1 = nn.Linear(64,128)
    self.dec2 = nn.Linear(128,256)
    self.dec3 = nn.Linear(256,28*28)

  def forward(self, x):
    x = torch.relu(self.en1(x))
    x = torch.relu(self.en2(x))
    x = torch.relu(self.en3(x))

    x = torch.relu(self.dec1(x))
    x = torch.relu(self.dec2(x))
    x = torch.relu(self.dec3(x))

    return x

denoised_auto_encoder = AutoEncoder()

# setting up
learning_rate = 0.01
epochs = 50

# picking cost function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(denoised_auto_encoder.parameters(), lr=
    learning_rate)

# training the model:
losses = []
for epoch in range(epochs):
  loss = 0
  for X_prime, X in trainloader:
    # get inputs
    X_prime = X_prime.view(-1, 28*28)
    X_prime = X_prime/255 # normalising the data
    X = X.view(-1,28*28)
    X = X/255

    # set param grads to 0
    optimizer.zero_grad()

    #forward + backward + optimize
    output = denoised_auto_encoder(X_prime)
    train_loss = criterion(output, X)

    train_loss.backward()
    optimizer.step()


    loss += train_loss.item()

  loss = loss/(len(trainloader))
  losses.append(loss)
  if epoch%10 == 0:
    print('Epoch: %d/%d,  loss: %.6f'%(epoch + 1,epochs, loss))
```


(a) 'Denoised' output.

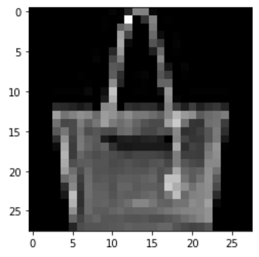

(b) Corresponding clean image.

Figure 8: Example of 'denoised' output of initial denoising autoencoder implementation, compared with its corresponding clean image.

### 3.2.2 Explore model variants with different layer sizes, depths, etc. to find what works well on the validation set and describe the process through which you arrive at your final model.

It was decided that the parameters that were to be optimised would be the number of layers, depth of the network (number of nodes on each layer) and the learning rate. Through trial and error it was found that the validation loss was lowest when fewer layers, more nodes and a lower learning rate were used. As a result, it was decided that the best combination of parameters would be 1 layer, 256 nodes and a learning rate of 0.001.

Ideally, to conclude which would have been the best possible combination, a grid search would have been performed. Unfortunately, due to the lack of computational power this was not feasible, however below is the code which would have been used to perform this grid search.

```
class AutoEncoderNodesLayers(nn.Module):
  def __init__(self,N,L):
    super().__init__()
    self.N = N
    self.num_layers = L

    self.en1 = nn.Linear(28*28, self.N)
    self.encoder = []
    for i in range(self.num_layers):
      self.encoder.append(nn.Linear(self.N, self.N))

    self.enlast = nn.Linear(self.N, 64)


    self.dec1 = nn.Linear(64, self.N)
    self.decoder = []
    for i in range(self.num_layers):
      self.decoder.append(nn.Linear(self.N, self.N))

    self.declast = nn.Linear(self.N,28*28)

  def forward(self, x):
    x = torch.relu(self.en1(x))
    for i in range(self.num_layers):
      x = torch.relu(self.encoder[i](x))
    x = torch.relu(self.enlast(x))

    x = torch.relu(self.dec1(x))
    for i in range(self.num_layers):
      x = torch.relu(self.decoder[i](x))
    x = torch.relu(self.declast(x))

    return x



def training_nodes_lays(nodes, layers, learning):
  model = AutoEncoderNodesLayers(nodes, layers)

  learning_rate = learning
  epochs = 50

  # picking cost function and optimizer
  criterion = nn.MSELoss()
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)


  train_losses = []
  val_losses = []
  for epoch in range(epochs):
    loss = 0
    for X_prime, X in trainloader:
      # get inputs
      X_prime = X_prime.view(-1, 28*28)
      X_prime = X_prime/255
      X = X.view(-1,28*28)
      X = X/255

      # set param grads to 0
      optimizer.zero_grad()

      #forward + backward + optimize
      output = model(X_prime)
      train_loss = criterion(output, X)

      train_loss.backward()
      optimizer.step()


      loss += train_loss.item()

    train_loss = loss/(len(trainloader_val))
    train_losses.append(train_loss)

    val_output = model(x_val_noisy_tensor/255)
    val_loss = criterion(val_output, x_val_tensor/255)
    val_losses.append(val_loss

    if epoch%10 == 0:
      print('Epoch: %d/%d,  Train Loss: %.6f, Validation Loss:
          %6f'%(epoch + 1,epochs, train_loss, val_loss))

  final_val_loss = val_losses[-1]
  print("Final Validation Loss: ", final_val_loss)
  return final_val_loss

nodes_range = [64, 128, 256]
layer_range = [1, 3, 5]
learning_range = [0.01, 0.05, 0.001]

cv = np.zeros((len(nodes_range), len(layer_range), len(
    learning_range)))

for i in range(len(nodes_range)):
  for j in range(len(layer_range)):
    for k in range(len(learning_range)):
      nodes = nodes_range[i]
      layers = layer_range[j]
      learning = learning_range[k]

      print('\n CROSS VALIDATION for number of nodes = %d,
          number of layers = %d, learning rate = %f'%(nodes,
          layers, learning))
      final_val_loss = training_nodes_lays(nodes, layers,
          learning)
      cv[i,j,k] = final_val_loss


# grid search
min_ind = cv.argmin()
node_ind, layer_ind, learn_ind = np.unravel_index(min_ind, cv.
    shape)
best_num_nodes = nodes_range[node_ind]
best_num_layers = layer_range[layer_ind]
best_learning = learning_range[learn_ind]

print('Lowest validation loss given by number of nodes = %d,
    number of layers = %d, learning rate = %f'%(best_num_nodes,
     best_num_layers, best_learning))
```

### 3.2.3 Train your final model to convergence on the training set using an optimisation algorithm of your choice.

Using the parameters found in the previous section, the model was trained to convergence. Adam was used as the optimisation algorithm as it was found during initial implementation that SGD produced indecipherable "denoised" output images, as well as for similar reasons as in question 3.1.

Note: Training on the training set for more than 50 epochs caused Colab to crash due to RAM limitations, therefore the number of training epochs have been set to 50.

```
class AutoEncoder(nn.Module):
  def __init__(self):
    super().__init__()
    self.en1 = nn.Linear(28*28, 256)
    self.en2 = nn.Linear(256,256)
    self.en3 = nn.Linear(256,64)

    self.dec1 = nn.Linear(64,256)
    self.dec2 = nn.Linear(256,256)
    self.dec3 = nn.Linear(256,28*28)

  def forward(self, x):
    x = torch.relu(self.en1(x))
    x = torch.relu(self.en2(x))
    x = torch.relu(self.en3(x))

    x = torch.relu(self.dec1(x))
    x = torch.relu(self.dec2(x))
    x = torch.relu(self.dec3(x))

    return x

denoised_auto_encoder = AutoEncoder()

# setting up
learning_rate = 0.001
epochs = 50

# picking cost function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(denoised_auto_encoder.parameters(), lr=
    learning_rate)

# putting the noisy and clean training images into one data set

batch_size = 64
trainset = torch.utils.data.TensorDataset(x_train_noisy_tensor,
     x_train_tensor)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=
    batch_size, shuffle=True, num_workers=2)

# training the model:
training_loss = []
validation_loss = []

for epoch in range(epochs):
  loss = 0
  for X_prime, X in trainloader:
    # get inputs
    X_prime = X_prime.view(-1, 28*28)
```

```
      X_prime = X_prime/255
      X = X.view(-1,28*28)
      X = X/255

      # set param grads to 0
      optimizer.zero_grad()

      #forward + backward + optimize
      output = denoised_auto_encoder(X_prime)
      train_loss = criterion(output, X)

      train_loss.backward()
      optimizer.step()


      loss += train_loss.item()

    loss = loss/(len(trainloader))
    training_loss.append(loss)

    val_output = denoised_auto_encoder(x_val_noisy_tensor)
    val_loss = criterion(val_output, x_val_tensor)
    validation_loss.append(val_loss)

    if epoch%10 == 0:
      print('Epoch: %d/%d,  loss: %.6f, val loss: %.6f'%(epoch +
          1,epochs, loss, val_loss))

print('Final Loss: %f \nFinal Validation Loss: %f'%(
    training_loss[-1],validation_loss[-1]))
```

### 3.2.4 Provide a plot of the loss on the training set and validation set for each epoch of training.

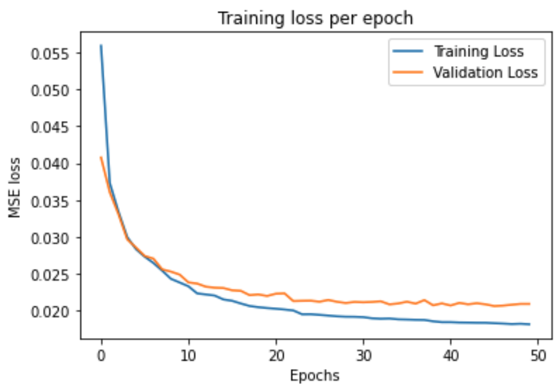


Figure 9: Training and validation loss for each epoch of training.

The final loss on the training and validation set were 0.0182 and 0.0209 respectively.

### 3.2.5 Provide a selection of 32 images in the original, "noisy", and "denoised" form from the test set and comment on what the model has been able to learn and what it appears to find challenging.

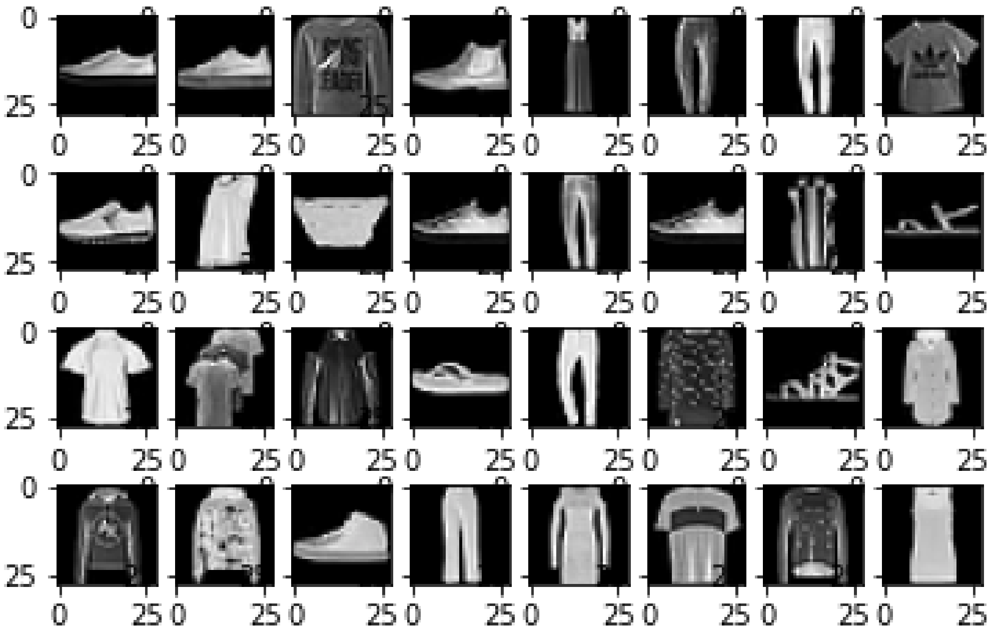


Figure 10: 32 randomly selected test images in their original form.

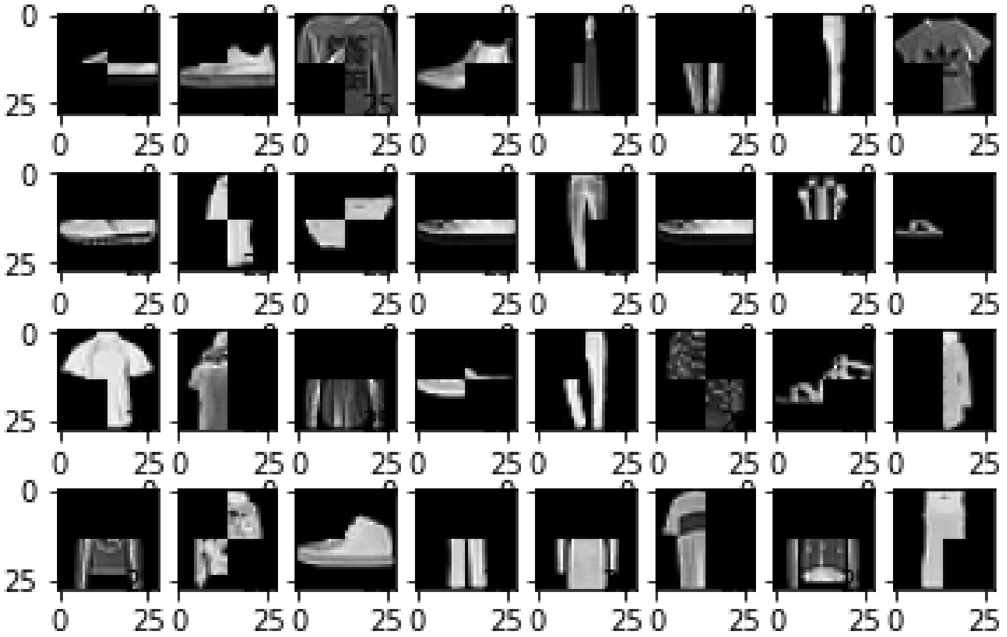


Figure 11: Corresponding 32 test images in their noisy form.

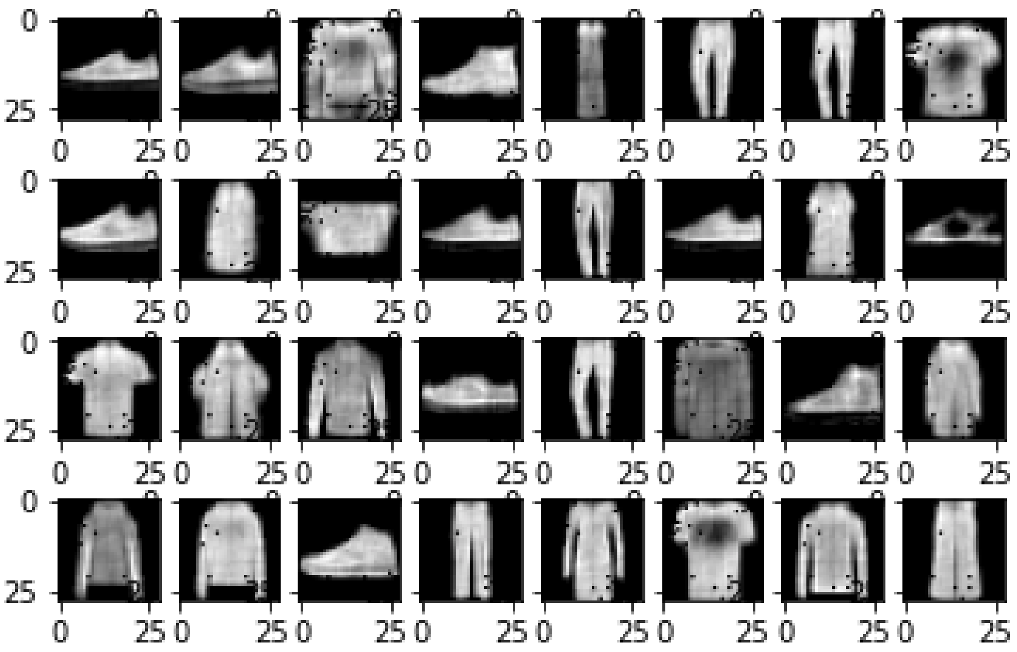


Figure 12: Corresponding 32 test images in their ’denoised’ form.

The model does well to produce the correct article of clothing, that is to say, if given a noisy image of a t-shirt for example, it correctly reproduces an image of a t-shirt.

On the other hand, the model seems to struggle in instances in which the item of clothing has details and patterns. The model struggles to recreate these logos and patterns, resulting in splotches of a darker or lighter colour in the place where they would be, although the model does get the placement correct.

The model also seems to find reproducing the correct colour/hue difficult in general.

Note: The places where the model fails or finds difficulty may be due to the limit on the number of epochs the model could be trained for as a result of Colab’s GPU capacity.

### 3.3 Sequence prediction

**Problem set-up:** the Stanford sentiment treebank data consists of sentences. The data was split into sentences where each was given a certain sentiment. The training set consisted of 8544 sentences, the validation set consisted of 1101 sentences and the testing set consisted off 2210 sentences. The aim of this task was to predict the next character given a observed sentence or a set sequence length.

#### 3.3.1 Implement a character-level recurrent neural network model with a cross-entropy loss for the textual portion of the Stanford Sentiment Treebank data, where the model is to predict the next character based on the character previously observed for a given sentence.

#### 3.3.2 Explore model variants with different recurrent units (“vanilla”, LSTM, and GRU), number of layers, different depths and layer sizes for the multi-layer perceptron, etc. to find what works well on the validation set and describe the process through which you arrive at your final model.

The code below contains the data processing steps and the construction of the "vanilla" RNN model. It also demonstrates explored variants of different recurrent units such as LSTM and GRU. The loss function used was cross entropy loss as specified in the question. Adam was chosen as the optimiser for the same reasons as in question 3.1.

**Code for data processing and construction of "vanilla" RNN:**

```
import numpy as np
import matplotlib.pyplot as plt
import random
import torch
from torch import nn
from torch.utils.tensorboard import SummaryWriter
# torch.cuda.is_available() - returns true if avaible or false
   if not.
is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'
   ll use this device variable later in our code.
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available to use")
else:
    device = torch.device("cpu")
    print("GPU not available to use, CPU used")

!curl -fsS https://nlp.stanford.edu/sentiment/
   trainDevTestTrees_PTB.zip -o ./trainDevTestTrees_PTB.zip #
   command to download the zip folder
!unzip -q -o -d ./ ./trainDevTestTrees_PTB.zip
!rm -f ./trainDevTestTrees_PTB.zip

def loadsst(path):
    xs = []
    ys = []
    with open(path, 'r') as file:
        # Quick, dirty, and improper S-expression parsing.
        for line in file.readlines():
            soup = line.split()  # Split the line into
               individual words
            # converts first array then l strip it and then
               convert to integer and then add it to the list.
            ys.append(int(soup[0].lstrip('(')))
            tokens = []
            for chunk in soup[2:]:
                if not chunk.endswith(")"):
                    continue
                tokens.append(chunk.rstrip(')'))
            xs.append(tokens)
    return xs, ys

ssttrainxs, ssttrainys = loadsst("./trees/train.txt")  #
    Training data
sstvalidxs, sstvalidys = loadsst("./trees/dev.txt")  #
    Validation data
ssttestxs, ssttestys = loadsst("./trees/test.txt")  # Testing
    data

def scrubData(xs):
    overall = []  # Big list
    for i in range(len(xs)):
        # Making all the sentances in one list where sentance
            becomes a element broken up by comma's ,
        overall.append(" ".join(xs[i]))
    return overall

# sentances we want the model to output when fed with words/
    characters
#  added valid and test set
sstraintext = scrubData(ssttrainxs)
ssvalidtext = scrubData(sstvalidxs)
sstesttext = scrubData(ssttestxs)

# extracts characters from our training data
# join all the words in the sentances together and extract the
    unique chracters from the combined sentances
#  this is potentially slow, since the runtime of the join
    function is proportional to the length of sstraintext
charact = set(''.join(sstraintext))


# maps integers to the characters as a dictionary
#  enumerate provides an interface of index and item
integer_to_char = dict(enumerate(charact))
# print(int_to_characters) - maps integers to numbers
# Dictionairy to map characters to integers
# also because of .items calling items from the dictionary.

#  enumerate provides an interface of index and item, so you
    get c-the character as key, and ind- the index as index
char_to_integer = {c: ind for ind, c in integer_to_char.items()
    }
# We have 94 characters all in all where 94 characters
    extracted from the training data and 94rd position not from
     the training data to account for other possible characters

char_to_integer['others'] = max(char_to_integer.values()) + 1


# Chat_to_integers maps characters to integers.
```

```
dictionary_size = len(char_to_integer) # 94 characters possible
     from characters from training and +1 for other characters
    mapped to integers



# Sequence length. Can be changed
maxlength = 80

def paddingAndClipping(wholeDataset):
    for i in range(len(wholeDataset)):
        if len(wholeDataset[i])<maxlength:
            # padding
            # this method of padding is slow. also, this is
               right padding to the end, which is bad?
            #while len(overall[i]) < maxlength:
            #    overall[i] += '*'  # padds with a star, but
               can pad with a space.
            # could just directly pad the string. string
               manipulation here, |||||| character * (number
               of character needed to pad) + original sentence
                = padded sentence e.g. 'a'*10 + 'bcdef' = '
               aaaaaaaaaabcdef'
            wholeDataset[i] = '*' * (maxlength-len(wholeDataset
               [i])) + wholeDataset[i]
        else:
            # clipping
            wholeDataset[i] = wholeDataset[i][:maxlength]
    return wholeDataset


def splittingXY(wholeDataset):
    # Creating lists that will hold our input and target
       sequences # Many to many process
    input_sequence = []
    target_sequence = []

    for i in range(len(wholeDataset)):
        # removes the last character of the input sequence so
           it is prepared to predict the last
        input_sequence.append(wholeDataset[i][:-1])
        # all input sequences gets last character removed
        # Removes the first character of the sequence to
           predict the last
        target_sequence.append(wholeDataset[i][1:])
        # all outputsequenceces gets first character removed.
        # My name is bob and I like ice crea -> input sequence.
            We do not feed the last character into the model.
        # y name is bob and I like ice cream -> out put
           sequence. So this will be the correct answer for
           the model at each time step corresponding to the
           input sequence
        # So the target sequence will always be one time step
           ahead of the input sequence
    return input_sequence, target_sequence


#  changed to function
# Converting target sequences of integer instead of characters
   by mapping them using dictionaires to allow one-hotencoding
def convertXYtoInt(xs, ys):
    for i in range(len(xs)):
        # converting characters in the sequences to sequences
           of integers by mapping them using dictionaries
        xs[i] = [char_to_integer[character] if character in
           char_to_integer else (dictionary_size-1)
                            for character in xs[i]]
        # We are using our dictionaries .
        ys[i] = [char_to_integer[character] if character in
           char_to_integer else (dictionary_size-1)
                            for character in ys[i]]
    return xs, torch.Tensor(ys).to(device)

# Defining initial hyperparameters for model construction
n_layers = 1 # number of layers
hidden_dim = 150 # hidden state size
n_epochs = 200 #epochs to consider number of times model goes
   through the entire training dataset
lr = 0.001 # Small learning rate can be adjusted . Rate at
   which our model updates the weights in the cell each time
   back propagation is done.
sequence_length = maxlength - 1 # Standardising the length of
   all the sequences to be equal to the sentances specified -1
    as the last character is removed because it will be
   pedicted
batch_size = 128 # number of all the sentances in the training
   is the batch size. This can change to what we put in our
   model.

# one hot encoding
#  the way of setting batch maybe incorrect, removed batch_size
def one_hot_encoding(incoming_sequence, dictionary_size,
   sequence_length): # input a sequence, it has numbers mapped
    in dictionary, squeezed to a sequence length and of the
   whole training batch size
    # where n_vector represents the features. 3 diensional
       vector
    n_vector = np.zeros((len(incoming_sequence),
       sequence_length, dictionary_size), dtype =np.float32) #
        creates multi array with a desired outpput shape

    # Replacing the 0 at the relevant character index with a 1
       to represent that character.  # going through all the
       sentances

    for i in range(len(incoming_sequence)):
        for j in range(sequence_length):
            n_vector[i, j, incoming_sequence[i][j]] = 1

    return torch.from_numpy(n_vector).to(device)

# do the preprocessing of all train,test, valid here
print(len(sstraintext))
print(sstraintext[0])
sstraintext = paddingAndClipping(sstraintext)
ssvalidtext = paddingAndClipping(ssvalidtext)
sstesttext  = paddingAndClipping(sstesttext)
```

```
print(len(sstraintext))
print(sstraintext[0])

trainxs, trainys = splittingXY(sstraintext)
validxs, validys = splittingXY(ssvalidtext)
testxs, testys   = splittingXY(sstesttext)

trainxs, trainys = convertXYtoInt(trainxs, trainys)
validxs, validys = convertXYtoInt(validxs, validys)
testxs, testys   = convertXYtoInt(testxs, testys)

# Input shape --> (Batch Size, Sequence Length, One-Hot
   Encoding Size)
# Number of characters is the size of the one hot endcoded
   vector.
# input sequence -> (through all sentances, sequence_length,
   one-hot encoded size)
# Replaces the corresponding character index with 1.
trainxs = one_hot_encoding(trainxs, dictionary_size,
   sequence_length)
validxs = one_hot_encoding(validxs, dictionary_size,
   sequence_length)
testxs = one_hot_encoding(testxs, dictionary_size,
   sequence_length)

print(trainxs.shape)
print(validxs.shape)
print(testxs.shape)
print(len(trainys))
print(len(testys))
print(len(validys))

# Use Dataset from pytorch to help batchify training.
class TorchDataset(torch.utils.data.Dataset):
    def __init__(self, xs, ys):
        self.x = xs
        self.y = ys
    def __getitem__(self, index):
        return (self.x[index], self.y[index])
    def __len__(self):
        return len(self.x)

trainDataset = TorchDataset(trainxs, trainys)
validDataset = TorchDataset(validxs, validys)
testDataset  = TorchDataset(testxs, testys)

# Use Dataloader from pytorch to help batchify training.
trainDataLoader = torch.utils.data.DataLoader(trainDataset,
   batch_size=batch_size, shuffle=True)
validDataLoader = torch.utils.data.DataLoader(validDataset,
   batch_size=len(validDataset), shuffle=True)
testDataLoader = torch.utils.data.DataLoader(testDataset,
   batch_size=len(testDataset), shuffle=True)

# Buiding a neural network model we define classes that
   inherits PyTorchs base class nn.module
class Model(nn.Module): # defining the class of the model we
   are consdiering .
```

```
    def __init__(self, input_size, output_size, hidden_dim,
       n_layers): # all classes in python use innit which is
       excuted when class is intiated. Function assigns values
        to object properties are other operations nessary to
       do when the object is being created
        super(Model, self).__init__() # constructor passing
           over the varaibles in the classes
        # innit assignes values to the paramters. self paramter
            is a reference to current instance of the class
           and is used to access the varaibles that belongs to
            the class
        # Defining parameters. self used to access the
           paramters in the class.
        self.hidden_dim = hidden_dim # refers to hidden size
        self.n_layers = n_layers # number of layers in the RNN.
            These parsamters of number of layers and
           hidden_dim will be passed into the RNN.

        # RNN model used but can consider LSTM, GRU here.
        self.rnn = nn.RNN(input_size, hidden_dim, n_layers,
           batch_first = True) # since batch_first = true then
            the output tensor is (batch,seq,feature)
        self.fc = nn.Linear(hidden_dim, output_size) # Weight
           matrix . No bias is used
        # using tanh on default setting for non-linearity

    def forward(self, x):

        batch_size = x.size(0) # size of the batch
        # Intialize hidden state for the first call in the RNN.
        # removed .cuda, we only need to do that within the
           function
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model
           and obtaining outputs
        out, hidden = self.rnn(x, hidden) # passing intialized
           terms in the hidden state
        out = out.contiguous().view(-1, self.hidden_dim) # this
            does reshaping of the outputs so it can be fit
           into a fully connected layer
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # We'll send the tensor holding the hidden state to the
            device we specified earlier as well
        #  changed to to(device) for generalization.
        hidden = torch.zeros(self.n_layers, batch_size, self.
           hidden_dim).to(device) # used for first hidden
           state ofzeros that will be used in the forward pass
        return hidden

class GRUModel(nn.Module): # defining the class of the model we
    are consdiering .
    def __init__(self, input_size, output_size, hidden_dim,
       n_layers): # all classes in python use innit which is
       excuted when class is intiated. Function assigns values
```

```
         to object properties are other operations nessary to
        do when the object is being created
        super(GRUModel, self).__init__() # constructor passing
            over the varaibles in the classes
        # innit assignes values to the paramters. self paramter
             is a reference to current instance of the class
            and is used to access the varaibles that belongs to
             the class
        # Defining parameters. self used to access the
            paramters in the class.
        self.hidden_dim = hidden_dim # refers to hidden size
        self.n_layers = n_layers # number of layers in the RNN.
             These parsamters of number of layers and
            hidden_dim will be passed into the RNN.


        self.gru = nn.GRU(input_size, hidden_dim, n_layers,
            batch_first = True) # since batch_first = true then
             the output tensor is (batch,seq,feature)
        self.fc = nn.Linear(hidden_dim, output_size) # Weight
            matrix . No bias is used
        # using tanh on default setting for non-linearity

    def forward(self, x):

        batch_size = x.size(0) # size of the batch
        # Intialize hidden state for the first call in the RNN.
        #  removed .cuda, we only need to do that within the
            function
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model
            and obtaining outputs
        out, hidden = self.gru(x, hidden) # passing intialized
            terms in the hidden state
        out = out.contiguous().view(-1, self.hidden_dim) # this
             does reshaping of the outputs so it can be fit
            into a fully connected layer
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # We'll send the tensor holding the hidden state to the
             device we specified earlier as well
        #  changed to to(device) for generalization.
        hidden = torch.zeros(self.n_layers, batch_size, self.
            hidden_dim).to(device) # used for first hidden
            state ofzeros that will be used in the forward pass
        return hidden

class LSTMModel(nn.Module): # defining the class of the model
    we are consdiering .
    def __init__(self, input_size, output_size, hidden_dim,
        n_layers): # all classes in python use innit which is
        excuted when class is intiated. Function assigns values
         to object properties are other operations nessary to
        do when the object is being created
        super(LSTMModel, self).__init__() # constructor passing
             over the varaibles in the classes
        # innit assignes values to the paramters. self paramter
             is a reference to current instance of the class
           and is used to access the varaibles that belongs to
             the class
        # Defining parameters. self used to access the
           paramters in the class.
        self.hidden_dim = hidden_dim # refers to hidden size
        self.n_layers = n_layers # number of layers in the RNN.
             These parsamters of number of layers and
           hidden_dim will be passed into the RNN.


        self.lstm = nn.LSTM(input_size, hidden_dim, n_layers,
           batch_first = True) # since batch_first = true then
             the output tensor is (batch,seq,feature)
        self.fc = nn.Linear(hidden_dim, output_size) # Weight
           matrix . No bias is used
        # using tanh on default setting for non-linearity

    def forward(self, x):

        batch_size = x.size(0) # size of the batch
        # Intialize hidden state for the first call in the RNN.

        hidden = self.init_hidden(batch_size)
        c = self.init_hidden(batch_size)

        # Passing in the input and hidden + c state into the
           model and obtaining outputs
        out, (hidden, c) = self.lstm(x, (hidden, c)) # passing
           intialized terms in the hidden state
        out = out.contiguous().view(-1, self.hidden_dim) # this
             does reshaping of the outputs so it can be fit
           into a fully connected layer
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # We'll send the tensor holding the hidden state to the
             device we specified earlier as well

        hidden = torch.zeros(self.n_layers, batch_size, self.
           hidden_dim).to(device) # used for first hidden
           state ofzeros that will be used in the forward pass
        return hidden

criterion = nn.CrossEntropyLoss() # cross entropy loss is what
   we want. Measures the peformance of the model

def optim(lr = 0.001):
  optimizer = torch.optim.Adam(model.parameters(), lr= lr) #
     adam is stocastic optimization with also adaptive
     learning rate. Where as in SGD the learning rate has an
     equivalent effect for all weights/ paramters
  return optimizer
```

In order to train the model on the training set and evaluate its performance on the validation set the train_valid() function was created. The number of epochs used to train the model was selected to be 200, through trial and error it was seen that this was a sufficient number for the model to reach convergence. A grid search was performed on each model variant, looping over the number of layers of the respective models (RNN, LSTM and GRU), the hidden dimension size of each layer and the learning rate used to train the model. Due to the GPU intensive nature of the problem it was deemed appropriate to grid search over three hyperparamters as opposed to more.

The sequence length was chosen to be 79, as through trial and error it performed better than other sequence lengths. The batch size was chosen to be 128, after a similar process was followed. The learning rate is an important hyperparameter to tune as if it is too large the output of model will oscillate over training epochs, due to it overshooting the minima, and if it is to small the model may learn too slowly. The network architecture of the RNN was tuned by altering the number of layers and hidden dimension size hyperparamters. This is important as they have a large impact on the model outputs. Each grid search finds the highest validation accuracy achieved and returns the corresponding hyperparameters.

The results of the grid search for each model variant are listed below:

- "vanilla": validation accuracy = $44.05\%$, number of layers = 2 , layer size = 200, learning rate = 0.005
- GRU: validation accuracy = $71.75\%$, number of layers = 2 , layer size = 100, learning rate = 0.01
- LSTM: validation accuracy = $72.12\%$, number of layers = 2 , layer size = 150, learning rate = 0.001

The LSTM model variant achieved the highest validation accuracy at $72.12\%$. This suggested that it was the best candidate for the final model.

In order to train the final model the hyperparameters that achieved this accuracy were used.

**Code for the training function, grid searches, test function and final model training, validation and testing:**

```
def train_valid(n_epochs = 200):

  optimizer = optim() # calling ADAM optimizer

  #initialize empty lists to store values for plots
  epoch_num = []
  train_l = []
  train_acc = []
  valid_l = []
  valid_acc = []

  # Training Run. hidden - just hidden states into the next
     cell.

  for epoch in range(1, n_epochs + 1):

      # training mode
      model.train()

      train_loss = 0
      train_accuracy = 0

      for batch, (xs, ys) in enumerate(trainDataLoader):
          optimizer.zero_grad() # clears the gradient from the
             preavious epoch.

          output, hidden = model(xs) # so coming in of length
             maxlength -1 inputs in
          loss = criterion(output, ys.view(-1).long())
          loss.backward() # Does backpropagation and calculates
              gradients to help update the weights (and biases
              if set)
          optimizer.step() # updating the weights in steps

          # log data
          #aggregate batch performance
          train_loss = train_loss + loss.item()

          #batch accuracy calculated with only the last element
          train_accuracy = train_accuracy + (torch.max(output.
             view(-1,sequence_length,dictionary_size)
             [:,-1,:],1).indices == ys.view(-1,sequence_length
             )[:,-1]).sum()/batch_size # batch accuracy

      # update initialized lists
      train_l.append(train_loss/len(trainDataLoader))
      train_acc.append(train_accuracy/len(trainDataLoader)*100)

  # Not using full batch but mini batch in training . Full
     batch is slower in training lower. When hidden paramters
     are to big it could be equivalent to a full batch

      #evaluation mode

      model.eval()

      valid_loss = 0
      valid_accuracy = 0

      for batch, (xs, ys) in enumerate(validDataLoader):

          output, hidden = model(xs) # so coming in of length
             maxlength -1 inputs in
          loss = criterion(output, ys.view(-1).long())

          #log data
          #aggregate batch performance

          valid_loss = valid_loss + loss.item()

          #batch accuracy calculated with only the last element

          valid_accuracy = valid_accuracy + (torch.max(output.
             view(-1,sequence_length,dictionary_size)
             [:,-1,:],1).indices == ys.view(-1,sequence_length
             )[:,-1]).sum()/len(xs) # we use full batch in
             validation set

      # update initialized lists
      epoch_num.append(epoch)
      valid_l.append(valid_loss/len(validDataLoader))
      valid_acc.append(valid_accuracy/len(validDataLoader)*100)
```

```
      # print summary for each epoch
      print('Epoch: {}/{}.............'.format(epoch, n_epochs)
          , end=' ')
      print("Loss: {:.3f}".format(train_loss/len(
          trainDataLoader)), end=' ')
      print("Accuracy: {:.3f}".format(train_accuracy/len(
          trainDataLoader)), end=' ')

      print("ValLoss: {:.3f}".format(valid_loss/len(
          validDataLoader)), end=' ')
      print("ValAccuracy: {:.3f}".format(valid_accuracy/len(
          validDataLoader)))

  return valid_accuracy, train_l, train_acc, valid_l, valid_acc
      , epoch_num


# grid search for RNN model

hiddendim = [100, 150, 200]
learningrate = [0.001, 0.005, 0.01]
numlayers = [1, 2, 3]


final_validation = []
validation_accuracy_v = []

for l in numlayers:
  for h in hiddendim:
    for k in learningrate:

      print(f'number of layers = {l}, hidden dimension = {h},
          learning rate = {k}')

      model = Model(input_size = dictionary_size, output_size =
           dictionary_size, hidden_dim= h, n_layers=l).to(
          device) # set up model and forward prop with
          specified layer size
      optim(lr = k) # set up optimiser with specified learning
          rate
      valid_accuracy, train_l, train_acc, valid_l, valid_acc,
          epoch_num = train_valid() # train model with
          specified epochs

      print(f'Final validation acuracy with current parameters:
           {valid_accuracy:2f}') # acc_valid[-1] list of
          validation accuracies from
      final_validation.append(valid_accuracy) # store final
          validation accuracy for current combination #
          acc_valid[-1]

      validation_accuracy_v.append([l,h,k]) # store current
          combination


highest_val_index = np.argmax(final_validation)
highest_val_accuracy = final_validation[highest_val_index]
optimal_params = validation_accuracy_v[highest_val_index]
```

```
print(f'The highest validation accuracy {highest_val_accuracy}
   on the RNN was achieved with: number_epochs = 200,
   number_layers = {optimal_params[0]} , hidden_dim = {
   optimal_params[1]}, learning_rate = {optimal_params[2]}')

output:
The highest validation accuracy 0.440508633852005 on the RNN
   was achieved with: number_epochs = 200, number_layers = 2 ,
    hidden_dim = 200, learning_rate = 0.005



# grid search for GRU

hiddendim = [100, 150, 200]
learningrate = [0.001, 0.005, 0.01]
numlayers = [1, 2, 3]


final_validation = []
validation_accuracy_v = []

for l in numlayers:
  for h in hiddendim:
    for k in learningrate:

      print(f'number of layers = {l}, hidden dimension = {h},
         learning rate = {k}')

      model = GRUModel(input_size = dictionary_size,
         output_size = dictionary_size, hidden_dim= h,
         n_layers=l).to(device) # set up model and forward
         prop with specified layer size
      optim(lr = k) # set up optimiser with specified learning
         rate
      valid_accuracy, train_l, train_acc, valid_l, valid_acc,
         epoch_num = train_valid() # train model with
         specified epochs

      print(f'Final validation acuracy with current parameters:
          {valid_accuracy:2f}') # acc_valid[-1] list of
         validation accuracies from
      final_validation.append(valid_accuracy) # store final
         validation accuracy for current combination #
         acc_valid[-1]

      validation_accuracy_v.append([l,h,k]) # store current
         combination


highest_val_index = np.argmax(final_validation)
highest_val_accuracy = final_validation[highest_val_index]
optimal_params = validation_accuracy_v[highest_val_index]

print(f'The highest validation accuracy {highest_val_accuracy}
   on the GRU was achieved with: number_epochs = 200,
   number_layers = {optimal_params[0]} , hidden_dim = {
   optimal_params[1]}, learning_rate = {optimal_params[2]}')
```

```
output:
The highest validation accuracy 0.7175295352935791 on the GRU
   was achieved with: number_epochs = 200, number_layers = 2 ,
    hidden_dim = 100, learning_rate = 0.01
```

```
# grid search lstm

hiddendim = [100, 150, 200]
learningrate = [0.001, 0.005, 0.01]
numlayers = [1, 2, 3]


final_validation = []
validation_accuracy_v = []

for l in numlayers:
  for h in hiddendim:
    for k in learningrate:

      print(f'number of layers = {l}, hidden dimension = {h},
         learning rate = {k}')

      model = LSTMModel(input_size = dictionary_size,
         output_size = dictionary_size, hidden_dim= h,
         n_layers=l).to(device) # set up model and forward
         prop with specified layer size
      optim(lr = k) # set up optimiser with specified learning
         rate
      valid_accuracy, train_l, train_acc, valid_l, valid_acc,
         epoch_num = train_valid() # train model with
         specified epochs

      print(f'Final validation acuracy with current parameters:
          {valid_accuracy:2f}') # acc_valid[-1] list of
         validation accuracies from
      final_validation.append(valid_accuracy) # store final
         validation accuracy for current combination #
         acc_valid[-1]

      validation_accuracy_v.append([l,h,k]) # store current
         combination


highest_val_index = np.argmax(final_validation)
highest_val_accuracy = final_validation[highest_val_index]
optimal_params = validation_accuracy_v[highest_val_index]

print(f'The highest validation accuracy {highest_val_accuracy}
   on the LSTM was achieved with: number_epochs = 200,
   number_layers = {optimal_params[0]} , hidden_dim = {
   optimal_params[1]}, learning_rate = {optimal_params[2]}')

output:
The highest validation accuracy 0.7211626172065735 on the LSTM
   was achieved with: number_epochs = 200, number_layers = 2 ,
    hidden_dim = 150, learning_rate = 0.001
```

```
def test():

  test_loss = 0
  test_accuracy = 0

  for batch, (xs, ys) in enumerate(testDataLoader):

    output, hidden = model(xs) # so coming in of length
       maxlength -1 inputs in
    loss = criterion(output, ys.view(-1).long())

    #aggregate batch performance

    test_loss = test_loss + loss.item()
    test_accuracy = test_accuracy + (torch.max(output.view(-1,
       sequence_length,dictionary_size)[:,-1,:],1).indices ==
       ys.view(-1,sequence_length)[:,-1]).sum()/len(xs) # we
       use full batch in validation set

    #batch accuracy calculated with only the last element


    print("Test_Loss: {:.3f}".format(test_loss/len(
       testDataLoader)), end=' ')
    print("Test_Accuracy: {:.3f}".format(test_accuracy/len(
       testDataLoader)), end=' ')


## Final Model ##

# From grid search findings
#The highest validation accuracy 0.440508633852005 on the RNN
    was achieved with: number_epochs = 200, number_layers = 2 ,
     hidden_dim = 200, learning_rate = 0.005
#The highest validation accuracy 0.7175295352935791 on the GRU
    was achieved with: number_epochs = 200, number_layers = 2 ,
     hidden_dim = 100, learning_rate = 0.01
#The highest validation accuracy 0.7211626172065735 on the LSTM
     was achieved with: number_epochs = 200, number_layers = 2
    , hidden_dim = 150, learning_rate = 0.001
# Clearly LSTM is the best model out of Vanilla, GRU andl LSTM
    as it produces the highest accuracy 72.1% > 71.8% > 44.1% (
     LSTM>GRU >> RNN)
# Paramters we consider for the final model are:
    sequence_length = 79, epochs = 200 , bacth_size = 128,
    learning_rate = 0.001, hidden_dim = 150 , num_layers = 2 .

model = LSTMModel(input_size = dictionary_size, output_size =
    dictionary_size, hidden_dim= 150, n_layers=2).to(device) #
    set up model and forward prop with specified layer size
optim(lr = 0.001) # set up optimiser with specified learning
    rate

# train the model

print('Training and validation.......')
```

```
valid_accuracy, train_l, train_acc, valid_l, valid_acc,
    epoch_num = train_valid(n_epochs=200) # train model with
    specified epochs

# plotting loss per epoch for the training and validation sets

plt.plot(epoch_num, train_l, label = 'Training')
plt.plot(epoch_num, valid_l, label = 'Validation')
plt.title('Loss against epochs for training and validation set'
    )
plt.xlabel('Epoch #')
plt.ylabel('Loss')
plt.legend()
plt.show()

# plotting accuracy per epoch for the training and validation
    sets

plt.plot(epoch_num, train_acc, Label = 'Training')
plt.plot(epoch_num, valid_acc, label = 'Validation')
plt.title('Accuracy against epochs for training and validation
    set')
plt.xlabel('Epoch #')
plt.ylabel('Accuracy %')
plt.legend()
plt.show()

# test the model

print('Testing.......')
test()

output:
Epoch: 200/200 Loss: 1.046 Accuracy: 0.741 ValLoss: 1.253
    ValAccuracy: 0.727
Test_Loss: 1.247 Test_Accuracy: 0.701
```

#### 3.3.3 Train your final model to convergence on the training set using an optimisation algorithm of your choice.

The optimisation algorithm chosen was Adam, as discussed in subsection 3.3.1. The model was trained to convergence using the train_valid function defined in subsection 3.3.2. The loss and accuracy on the training and validation set is provided below.

- training set: loss = 1.046, accuracy = 74.1%
- validation set: loss = 1.253, accuracy = 72.7%

#### 3.3.4 Provide a plot of the loss on the training set and validation set for each epoch of training.

The loss and accuracy plots for the training and validation sets are provided in Figures 13 and 14.

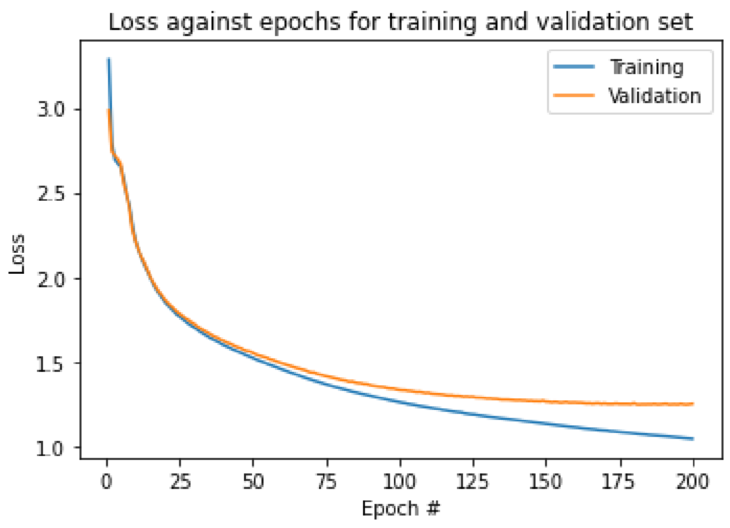


Figure 13: Loss for the training and validation set on the final model

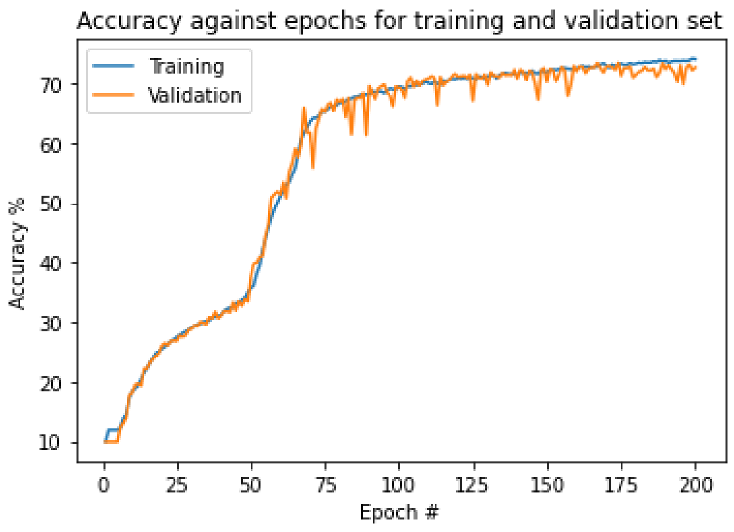


Figure 14: Accuracy for the training and validation set on the final model

#### 3.3.5 Provide the final accuracy on the training, validation, and test set.

The final accuracy is as follows:

- training set accuracy = 74.1%
- validation set accuracy = 72.7%
- test set accuracy = 70.1%

#### 3.3.6 Provide 5 sentences completed by your final model, after you have provided a first few words. For these generated sentences, please mark which part was provided by you and which part was generated by the model.

The test_sequence list contains a few words in the list below. Test_sequence examples were clipped and padded according to a character length of 79. Characters were then produced which are to be predicted in the future in order to complete the sentences from the test sequence examples. The model output shows the sentence predicted below.

**Code for sentence completion:**

```
# We only care about the last output
# We pad or clip the test sequence, then let it complete the
    sentence
# sequence length is 282,
# we supply example sentence
# "I am an idiot, what are you going "
# we want to complete such sentence
# pad it to 79 or clip it
# then predict, shift and predict, until the max number of time
     is done.
# We set limit to maybe 150 operations and see the results.



test_sequence = ["Absolutely rubbish and","Please give me a
    100.", "This movie is really good, have you", "I love
    action, comedy ", "The meaning of this movie is"]

test_sequencexs = paddingAndClipping(test_sequence) # pad up to
     80 characters

test_sequencexs = [item[1:] for item in test_sequencexs]
# converting characters in the sequences to sequences of
    integers by mapping them using dictionaries
for i in range(len(test_sequencexs)):
  test_sequencexs[i] = [char_to_integer[character] if character
       in char_to_integer else (dictionary_size-1)
     for character in test_sequencexs[i]]
test_sequencexs = one_hot_encoding(test_sequencexs,
    dictionary_size, sequence_length)


n_predictions = 150 # 150 characters in the future.

for prediction in range(n_predictions):
# evaluation mode
  model.eval()
  output, hidden = model(test_sequencexs.view(5 ,
      sequence_length, dictionary_size)) # so coming in of
      length maxlength -1 inputs in

  # get last item from output, which is our next character
      prediction
  # the predicted class is a index
  predicted_character = torch.max(output.view(-1,
      sequence_length,dictionary_size)[:,-1,:],1).indices


  test_sequencexs = torch.cat((test_sequencexs[:,1:,:],
      one_hot_encoding(torch.tensor(predicted_character.view
      (-1,1), dtype=torch.int32).tolist(), dictionary_size, 1))
      , 1)

#convert one hot encoded test sequence back to normal sequence
# use argmax or max to find.
```

```
output_sequences = []
for sequence in test_sequencexs:
  characterIntegerList = torch.max(sequence,1).indices
  tempString = ""
  for integer in characterIntegerList:
    for key, value in char_to_integer.items(): # for name, age
       in dictionary.iteritems(): (for Python 2.x)
      if value == integer:
        tempString+=key
  output_sequences.append(tempString)

for i in range(len(test_sequence)):
    print("Input sentence: ", test_sequence[i])
    print("Model prediction: ", output_sequences[i])
```

```
Input sentence:
   *****************************************************
   Absolutely rubbish and
Model prediction:  ar of a movie ... a movie that scenes ...
   and the film is n't a terrible ... bu

Input sentence:
   ******************************************************
   Please give me a 100.
Model prediction:  gh that is an intelligent , and it is an
   exceptional , and the film ... it 's a

Input sentence:  ******************************************
   This movie is really good, have you
Model prediction:  , and it 's ... an advantage ... an action
   film ... it 's a star ... a compelli

Input sentence:
   *****************************************************I
    love action, comedy
Model prediction:  .. a movie ... it 's a standard and stale
   ... an action ... is that it 's a fai

Input sentence:
   ***********************************************The
   meaning of this movie is
Model prediction:  , but it 's a star of a story that should
   be probably wonder ... and the film
```

The results show the sentences predicted. A limitation of the model is that there are full stops randomly distributed in the predicted sentences. This could be due to the model predicting the last character of the input sentence, as full stops are commonly used as the last character in a sentence. Hence using full stops as a more common type of character to predict in the model itself. The model itself is producing some words in the sentences that make sense such as " that is an intelligent , and it is an exceptional". However, there are also other parts of sentences that do not make sense. This type of model demonstrates that feeding in a few characters in from words into the model can generate sentences.

## 3.4 Bag of vectors

### 3.4.1 Implement a bag of vectors model with a cross-entropy loss for the Stanford Sentiment Treebank data, based on the pre-trained word2vec embeddings provided in the Assignment Colaboratory notebook.

```
!wget -P /root/input/ -c "https://s3.amazonaws.com/dl4j-
    distribution/GoogleNews-vectors-negative300.bin.gz"

import gensim
import numpy as np
from sklearn.preprocessing import OneHotEncoder
all_stopwords = gensim.parsing.preprocessing.STOPWORDS

from gensim.models import KeyedVectors
EMBEDDING_FILE = '/root/input/GoogleNews-vectors-negative300.
    bin.gz' # from above
word2vec = KeyedVectors.load_word2vec_format(EMBEDDING_FILE,
    binary=True, limit = 100000)

def loadsst(path):
    xs = []
    ys = []
    file1 = open(path, 'r')
    Lines = file1.readlines()
    for line in Lines:
        soup = line.split()
        ys.append(int(soup[0].lstrip('(')))
        tokens = []
        for chunk in soup[2:]:
            if not chunk.endswith(")"):
                continue
            tokens.append(chunk.rstrip(')'))
        xs.append(tokens)
    return xs, ys

x_train, y_train = loadsst("/content/train.txt")
x_val, y_val = loadsst("/content/dev.txt")
x_test, y_test = loadsst("/content/test.txt")
len(x_train), len(y_train), len(x_val), len(y_val), len(x_test)
    , len(y_test)

X_train = x_train
X_val = x_val
X_test = x_test

vocab = []
for sentence in X_train:
    vocab += sentence
vocab = set(vocab)

word2vec.wv.vocab.get("UNK").index

# build vocab to integer mapping
from collections import Counter
counts = Counter(vocab)

X_train_ints = []
for sentence in X_train:
    X_train_ints.append([word2vec.wv.vocab.get(word).index if
        word in word2vec.wv.vocab else 98307 for word in
        sentence])

X_val_ints = []
for sentence in X_val:
    X_val_ints.append([word2vec.wv.vocab.get(word).index if
        word in word2vec.wv.vocab else 98307 for word in
        sentence])

X_test_ints = []
for sentence in X_test:
    X_test_ints.append([word2vec.wv.vocab.get(word).index if
        word in word2vec.wv.vocab else 98307 for word in
        sentence])

# Compute maximum sentence length
sentence_lens = Counter([len(x) for x in X_train_ints])

# build list of unique words in training and validation sets

train_sentence_lengths = []
for sentence in X_train:
    train_sentence_lengths.append(len(sentence))

val_sentence_lengths = []
for sentence in X_val:
    val_sentence_lengths.append(len(sentence))

test_sentence_lengths = []
for sentence in X_test:
    test_sentence_lengths.append(len(sentence))

def pad_features(X_ints, seq_length):
    features = np.zeros((len(X_ints), seq_length), dtype=int)

    for i, row in enumerate(X_ints):
        features[i, -len(row):] = np.array(row)[:seq_length]
    return features

seq_length = max(sentence_lens)

features_train = pad_features(X_train_ints, seq_length=
    seq_length)
features_val = pad_features(X_val_ints, seq_length=seq_length)
features_test = pad_features(X_test_ints, seq_length=seq_length
    )

print(features_train.shape)
print(features_val.shape)
print(features_test.shape)

import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

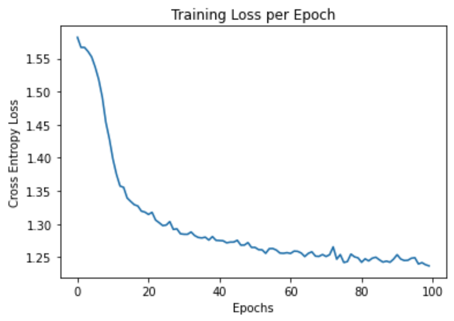
```
if torch.cuda.is_available():
    dev = "cuda:0"
else:
    dev = "cpu"
device = torch.device(dev)

weights = torch.FloatTensor(word2vec.wv.vectors)

training_features_tensor = torch.tensor(features_train, dtype=
   torch.float)
training_sentence_lengths = torch.tensor(train_sentence_lengths
   , dtype=torch.float)
training_labels_tensor = torch.tensor(y_train, dtype=torch.
   float)
val_features_tensor = torch.tensor(features_val, dtype=torch.
   float)
val_sentence_lengths = torch.tensor(val_sentence_lengths, dtype
   =torch.float)
val_labels_tensor = torch.tensor(y_val, dtype=torch.float)
test_features_tensor = torch.tensor(features_test, dtype=torch.
   float)
test_sentence_lengths = torch.tensor(test_sentence_lengths,
   dtype=torch.float)
test_labels_tensor = torch.tensor(y_test, dtype=torch.float)
```

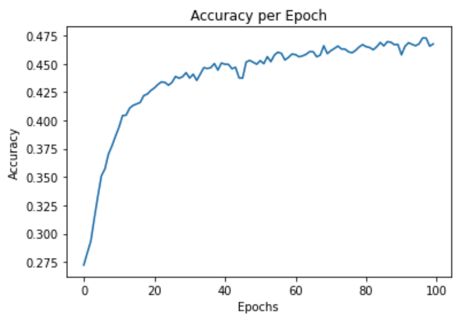
Using the word2vec pretrained embeddings, a bag of vectors model with a cross-entropy loss for the Stanford Sentiment Treebank data was implemented. The code for which is shown below:

```
LEARNING_RATE = 0.001
EPOCHS = 100
BATCH_SIZE = 1024
hidden_layer_1 = 128
hidden_layer_2 = 64
output_size = 5
emb_dim = 300

trainset = torch.utils.data.TensorDataset(
   training_features_tensor, training_labels_tensor,
   training_sentence_lengths)
valset = torch.utils.data.TensorDataset(val_features_tensor,
   val_labels_tensor, val_sentence_lengths)
testset = torch.utils.data.TensorDataset(test_features_tensor,
   test_labels_tensor, test_sentence_lengths)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=
   BATCH_SIZE, shuffle=True, num_workers=2)
valloader = torch.utils.data.DataLoader(valset, batch_size=
   val_features_tensor.shape[0], shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=
   test_features_tensor.shape[0], shuffle=True, num_workers=2)

class Net(nn.Module):
    def __init__(self, pre_trained_embeddings, sentence_length)
       :
      super(Net, self).__init__()
      self.sentence_length = sentence_length
      self.embeddings = nn.Embedding.from_pretrained(
         pre_trained_embeddings, freeze=True)
      self.fc1 = nn.Linear(emb_dim, hidden_layer_1)
```

```
          self.drop_layer = nn.Dropout(p= 0.2)
          self.fc2 = nn.Linear(hidden_layer_1, hidden_layer_2)
          self.drop_layer_2 = nn.Dropout(p= 0.2)
          self.fc3 = nn.Linear(hidden_layer_2, output_size)

      def forward(self, x, sentence_length):
          input = x.type(torch.LongTensor)
          sentence_length = sentence_length.to(dtype=torch.long)
          embeds = self.embeddings(input) # [batch_size, max_length
              , emb]
          final = torch.mean(embeds, dim = 1)

          x = self.fc1(final)
          x = self.drop_layer(x)
          x = torch.tanh(x)
          x = self.fc2(x)
          x = self.drop_layer_2(x)
          x = torch.tanh(x)
          x = self.fc3(x)
          return x

net = Net(weights, train_sentence_lengths)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=LEARNING_RATE
    )

train_costs = []
train_accuracies = []

for epoch in range(EPOCHS):
    running_loss = 0.0
    num_of_batches = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels, lengths = data
        optimizer.zero_grad()
        logits = net(inputs, lengths)
        targets = labels.type(torch.LongTensor)
        loss = criterion(logits, targets)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        num_of_batches +=1


    train_logits = net(training_features_tensor,
       training_sentence_lengths)
    _, train_predictions = torch.max(train_logits.data, 1)
    train_correct_predictions = (train_predictions.int() ==
       training_labels_tensor.int()).sum().numpy()
    train_length = training_labels_tensor.size()[0]
    train_accuracy = train_correct_predictions/train_length

    train_costs.append(running_loss/num_of_batches)
    train_accuracies.append(train_accuracy)

    if epoch%10 == 0:
```

```
print("Epoch: %d/%d, Loss: %.6f, Accuracy: %.6f"%(epoch +
     1,EPOCHS, running_loss/num_of_batches,
    train_accuracy))
```





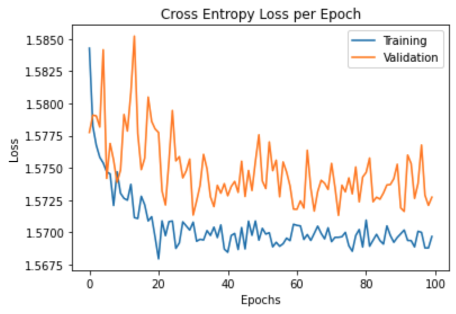
(a) Training loss (b) Training accuracy

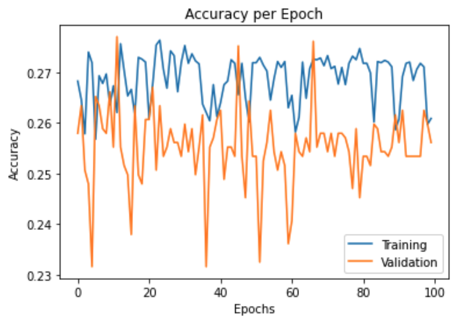
Figure 15: Initial model training loss and accuracy plot per epoch.

### 3.4.2 Explore model variants with different pooling operations, keeping the word embeddings fixed during training or updating them, different depths and layer sizes for the multi-layer perceptron, etc. to find what works well on the validation set and describe the process through which you arrive at your final model.

The model was improved by changing certain features of the model. For example, the pooling operation used to find the embedding representation for the entire sentence was varied between mean, maximum and minimum, where the mean operation gave higher accuracies than the other two operations.

```
_, final = torch.max(embeds, dim=1)
final = final.type(torch.float)
```

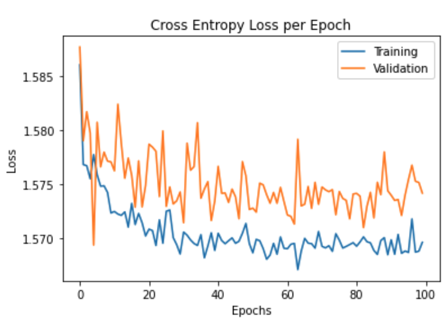




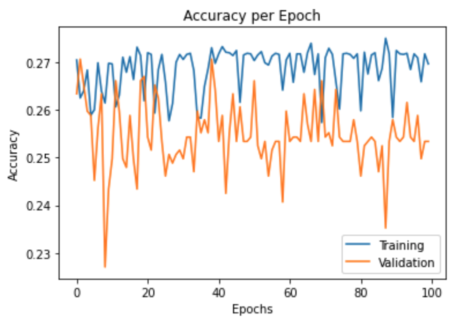
(a) Loss plot (b) Accuracy plot

Figure 16: Loss and accuracy plots per epoch for model using max pooling.

```
_, final = torch.min(embeds, dim=1)
final = final.type(torch.float)
```



(a) Loss plot



(b) Accuracy plot

Figure 17: Loss and accuracy plots per epoch for model using min pooling.

It can be seen in Figures 16 and 17 that the maximum and minimum pooling operations lead to large fluctuations in the loss and accuracy plots per epoch and also lead to lower accuracies than the mean pooling operation.

The model was then tested to see whether fixing the embeddings, or updating them during the iterations would give the better accuracies. It was found that through fixing the embeddings, better validation accuracies were found than for the updated embeddings. This was to be expected, as after many epochs, the updated embeddings would end up being far from the initial embeddings which represented the words, hence representing a loss of information during training. This loss of information resulted in a lower validation accuracy.

Finally, other parameters, such as batch size, number of layer, layer size, dropout ratio and learning rate were varied independently. Initially it was found that 3 layers performed better than lower numbers of layers. Following from this, it was found that a batch size of 256 yielded the best accuracy along with a number of nodes of 256 in the each layer, before going to a 5 node output. Lastly, it was found that a dropout percentage of 20% and learning rate of 0.0001 performed best compared to other values. It was also found that the tanh activation function gave the best accuracies, when compared to the sigmoid and ReLU activation functions.

```
# the focus will be on optimising parameters for the mean
    pooling model

class Optimisation_Net(nn.Module):
    def __init__(self, pre_trained_embeddings, sentence_length,
        num_nodes):
      super(Optimisation_Net, self).__init__()
      self.sentence_length = sentence_length
      self.num_nodes = num_nodes

      self.embeddings = nn.Embedding.from_pretrained(
          pre_trained_embeddings, freeze=True)
      self.fc1 = nn.Linear(emb_dim, self.num_nodes)
      self.drop_layer = nn.Dropout(p= 0.2)
      self.fc2 = nn.Linear(num_nodes, num_nodes)
      self.drop_layer_2 = nn.Dropout(p= 0.2)
      self.fc3 = nn.Linear(num_nodes, output_size)

    def forward(self, x, sentence_length):
      input = x.type(torch.LongTensor)
      sentence_length = sentence_length.to(dtype=torch.long)
      embeds = self.embeddings(input) # [batch_size, max_length
          , emb]
      final = torch.mean(embeds, dim = 1)

      x = self.fc1(final)
```

```
        x = self.drop_layer(x)
        x = torch.tanh(x)
        x = self.fc2(x)
        x = self.drop_layer_2(x)
        x = torch.tanh(x)
        x = self.fc3(x)
        return x

batch_sizes = [256, 512, 1024]
nodes = [64, 128, 256]

for batch in batch_sizes:
    for num_nodes in nodes:

        trainloader = torch.utils.data.DataLoader(trainset,
           batch_size=batch, shuffle=True, num_workers=2)

        net = Optimisation_Net(weights, train_sentence_lengths,
            num_nodes)

        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(net.parameters(), lr=
           LEARNING_RATE)

        train_costs = []
        train_accuracies = []
        val_costs = []
        val_accuracies = []

        for epoch in range(EPOCHS):
            running_loss = 0.0
            num_of_batches = 0
            for i, data in enumerate(trainloader, 0):
                inputs, labels, lengths = data
                optimizer.zero_grad()
                logits = net(inputs, lengths)
                targets = labels.type(torch.LongTensor)
                loss = criterion(logits, targets)
                loss.backward()
                optimizer.step()

                running_loss += loss.item()
                num_of_batches +=1
                # break

            train_logits = net(training_features_tensor,
               training_sentence_lengths)
            _, train_predictions = torch.max(train_logits.data,
                1)
            train_correct_predictions = (train_predictions.int
               () == training_labels_tensor.int()).sum().numpy
               ()
            train_length = training_labels_tensor.size()[0]
            train_accuracy = train_correct_predictions/
               train_length

            val_logits = net(val_features_tensor,
               val_sentence_lengths)
            val_loss = criterion(val_logits, val_labels_tensor.
               type(torch.LongTensor))
            _, val_predictions = torch.max(val_logits.data, 1)
            val_correct_predictions = (val_predictions.int() ==
                 val_labels_tensor.int()).sum().numpy()
            val_length = val_labels_tensor.size()[0]
            val_accuracy = val_correct_predictions/val_length


            train_costs.append(running_loss/num_of_batches)
            train_accuracies.append(train_accuracy)
            val_costs.append(val_loss.item())
            val_accuracies.append(val_accuracy)

        print("Model: Batch Size: ", batch, " No. Nodes: ",
            num_nodes , " Best train accuracy: ", max(
            train_accuracies), 'at epoch: ', np.argmax(
            train_accuracies), "Best val accuracy: ", max(
            val_accuracies), 'at epoch: ', np.argmax(
            val_accuracies))
```

### 3.4.3 Train your final model to convergence on the training set using an optimisation algorithm of your choice.

The best model was trained to convergence on the training set, using the Adam optimisation algorithm for the same reasons as in question 3.1.

```
LEARNING_RATE = 0.001
EPOCHS = 30
BATCH_SIZE = 256
hidden_layer_1 = 256
hidden_layer_2 = 256
output_size = 5
emb_dim = 300

trainset = torch.utils.data.TensorDataset(
   training_features_tensor, training_labels_tensor,
   training_sentence_lengths)
valset = torch.utils.data.TensorDataset(val_features_tensor,
   val_labels_tensor, val_sentence_lengths)
testset = torch.utils.data.TensorDataset(test_features_tensor,
   test_labels_tensor, test_sentence_lengths)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=
   BATCH_SIZE, shuffle=True, num_workers=2)
valloader = torch.utils.data.DataLoader(valset, batch_size=
   val_features_tensor.shape[0], shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=
   test_features_tensor.shape[0], shuffle=True, num_workers=2)

class Net(nn.Module):
    def __init__(self, pre_trained_embeddings, sentence_length)
        :
      super(Net, self).__init__()
      self.sentence_length = sentence_length
      self.embeddings = nn.Embedding.from_pretrained(
         pre_trained_embeddings, freeze=True)
      self.fc1 = nn.Linear(emb_dim, hidden_layer_1)
      self.drop_layer = nn.Dropout(p= 0.2)
```

```
        self.fc2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        self.drop_layer_2 = nn.Dropout(p= 0.2)
        self.fc3 = nn.Linear(hidden_layer_2, output_size)

    def forward(self, x, sentence_length):
        input = x.type(torch.LongTensor)
        sentence_length = sentence_length.to(dtype=torch.long)
        embeds = self.embeddings(input) # [batch_size, max_length
            , emb]
        final = torch.mean(embeds, dim = 1)

        x = self.fc1(final)
        x = self.drop_layer(x)
        x = torch.tanh(x)
        x = self.fc2(x)
        x = self.drop_layer_2(x)
        x = torch.tanh(x)
        x = self.fc3(x)
        return x

net = Net(weights, train_sentence_lengths)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=LEARNING_RATE
    )

train_costs = []
train_accuracies = []
val_costs = []
val_accuracies = []
test_costs = []
test_accuracies = []

for epoch in range(EPOCHS):
    running_loss = 0.0
    num_of_batches = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels, lengths = data
        optimizer.zero_grad()
        logits = net(inputs, lengths)
        targets = labels.type(torch.LongTensor)
        loss = criterion(logits, targets)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        num_of_batches +=1


    train_logits = net(training_features_tensor,
        training_sentence_lengths)
    _, train_predictions = torch.max(train_logits.data, 1)
    train_correct_predictions = (train_predictions.int() ==
        training_labels_tensor.int()).sum().numpy()
    train_length = training_labels_tensor.size()[0]
    train_accuracy = train_correct_predictions/train_length

    val_logits = net(val_features_tensor, val_sentence_lengths)
    val_loss = criterion(val_logits, val_labels_tensor.type(
       torch.LongTensor))
    _, val_predictions = torch.max(val_logits.data, 1)
    val_correct_predictions = (val_predictions.int() ==
       val_labels_tensor.int()).sum().numpy()
    val_length = val_labels_tensor.size()[0]
    val_accuracy = val_correct_predictions/val_length

    test_logits = net(test_features_tensor,
       test_sentence_lengths)
    test_loss = criterion(test_logits, test_labels_tensor.type(
       torch.LongTensor))
    _, test_predictions = torch.max(test_logits.data, 1)
    test_correct_predictions = (test_predictions.int() ==
       test_labels_tensor.int()).sum().numpy()
    test_length = test_labels_tensor.size()[0]
    test_accuracy = test_correct_predictions/test_length
    if epoch%5 == 0:
        print('Epoch: %d/%d, train loss: %.3f, train acc: %.3f,
            val loss: %.3f, val acc: %.3f, test loss: %.3f,
           test acc: %.3f'%(epoch + 1,EPOCHS, running_loss /
           num_of_batches, train_accuracy, val_loss,
           val_accuracy, test_loss, test_accuracy))

    train_costs.append(running_loss/num_of_batches)
    train_accuracies.append(train_accuracy)
    val_costs.append(val_loss)
    val_accuracies.append(val_accuracy)
    test_costs.append(test_loss)
    test_accuracies.append(test_accuracy)

print("Best train accuracy: ", max(train_accuracies), 'at epoch
   : ', np.argmax(train_accuracies), "Best val accuracy: ",
   max(val_accuracies), 'at epoch: ', np.argmax(val_accuracies
   ), "Best test accuracy: ", max(test_accuracies), 'at epoch:
    ', np.argmax(test_accuracies))
```

#### 3.4.4 Provide a plot of the loss on the training set and validation set for each epoch of training.

The trained model was then projected on the training and validation set to give the following plot of loss for each epoch in training.

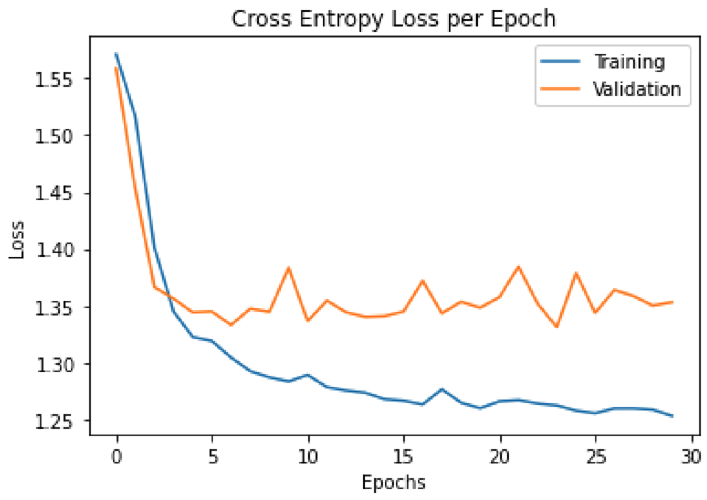


Figure 18: Plot of loss on the training and validation set for each epoch of training.

### 3.4.5 Provide the final accuracy on the training, validation, and test set.

The final accuracy on the training, validation and test set were 46.3%, 43.3% and 42.4% respectively.

### 3.4.6 Provide a selection of the classification decisions of the final model for 5 opinionated sentences from movies reviews that you find online – perhaps for a movie you liked, or did not like? Remember to provide a links to the reviews from which you selected the sentences.

The model can be used to classify movie reviews into how well they were received by critics, as shown in table 3. Movie reviews have been sourced from Rotten Tomatoes for the films *Murder Mystery* (Tomatoes, 2019a), *Toy Story 4* (Tomatoes, 2019b), *The Spongebob Squarepants Movie* (Tomatoes, 2004), *The Human Centipede* (B, 2010) and CINEMABLEND for *The Judge* (Brent McKnight, 2014). The films were chosen to demonstrate the models ability to identify different sentiments correctly.

Table 2: Class labels

| Class | Sentiment |
|---|---|
| 0 | 'Extremely Negative' |
| 1 | 'Negative' |
| 2 | 'Neutral' |
| 3 | 'Positive' |
| 4 | 'Extremely Positive' |

**Results:**

```
output:
Review:  Heartwarming , funny , and  beautifullyanimated , Toy
     Story 4 manages  the  unlikely  feat of extending  , and
    perhaps  concluding  , a practically  perfectanimated  saga
    .
Sentiment:   Extremely Positive
Class:  4
-----------------------------------------------------------------
Review:  Murder  Mystery  reunites  Jennifer  Aniston  and
    Adam Sandler  for a lightweight  comedy   t h a t s  contentto
      settle  for  merely  mediocre.
Sentiment:   Neutral
```

```
Class:  2
-------------------------------------------------------------
Review:  While  watching  these  two  actors  share  the
   screen  asan  estranged  father  and son is  compelling ,
    t h a t s  really  all The  Judge  has  going  for it.
   Outsideof  Duvall  and RDJ , the  script  is super
   formulaic.
Sentiment:   Positive
Class:  3
-------------------------------------------------------------
Review:  Sometimes  you see  failed  films  and think , well ,
   they  tried  for  this or that  but  couldn  t  pull itoff
   . After  seeing  Ben  Stiller  s  version  of thevenerable
     James  Thurber  story , I  d o n t  know  whatthe  film
   was  trying  for.
Sentiment:   Negative
Class:  1
-------------------------------------------------------------
Review:  A film  for kids , students , stoners , anyone
   whoenjoys a break  from  reality.
Sentiment:   Negative
Class:  1
-------------------------------------------------------------
```

It can be seen that for the above movie reviews, the model accurately predicts the sentiment of three of the five reviews. The first, second and fourth reviews are all correct, however the third should be negative and the fifth should be positive. This performed slightly better than expected as the best test accuracy obtained was 42.4%, so we expected only two of the reviews to be correctly classified.

## 3.5 Sequence encoding

### 3.5.1 Implement a recurrent neural network model which encodes a sequence of words into inputs to a multi-layer perceptron with a cross-entropy loss for the Stanford Sentiment Treebank data.

A model similar to the bag of vectors model was created, but instead of using the mean pooling operation to create a representation for the sentence, a recurrent neural network was used, which was fed into the multi-layer perceptron. In this model, the cross-entropy loss was calculated and the Stanford Sentiment Treebank data was used.

**Pre-processing steps:** Tokenized data was used to find unique words in the training data and form the vocabulary (around 18000 words). Then, an embedding matrix was built on the vocabulary using the pre-trained word2vec embeddings (from Google News dataset each embedding with dim = 300). The pre-trained embedding was then used to train the different model variants which are further explained in following sections.

**Code for data pre-processing:**

```
!wget -P /root/input/ -c "https://s3.amazonaws.com/dl4j-
    distribution/GoogleNews-vectors-negative300.bin.gz"

import gensim
import numpy as np
from sklearn.preprocessing import OneHotEncoder
all_stopwords = gensim.parsing.preprocessing.STOPWORDS

from gensim.models import KeyedVectors
EMBEDDING_FILE = '/root/input/GoogleNews-vectors-negative300.
    bin.gz' # from above
word2vec = KeyedVectors.load_word2vec_format(EMBEDDING_FILE,
    binary=True, limit = 100000)

def loadsst(path):
    xs = []
    ys = []
    file1 = open(path, 'r')
    Lines = file1.readlines()
    for line in Lines:
        soup = line.split()
        ys.append(int(soup[0].lstrip('(')))
        tokens = []
        for chunk in soup[2:]:
            if not chunk.endswith(")"):
                continue
            tokens.append(chunk.rstrip(')'))
        xs.append(tokens)
    return xs, ys

X_train, y_train = loadsst("/content/train.txt")
X_val, y_val = loadsst("/content/dev.txt")
X_test, y_test = loadsst("/content/test.txt")
len(X_train), len(y_train), len(X_val), len(y_val), len(X_test)
    , len(y_test)

# build vocab
vocab = []
for sentence in X_train:
    vocab += sentence
vocab = list(set(vocab))
vocab.insert(0, "UNK")
print("Number of unique vocabs: ", len(vocab))

# # build vocab to integer mapping
from collections import Counter
counts = Counter(vocab)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 0)}


def convert_words_to_idx(X):
    X_ints = []
    X_lengths = []
    for sentence in X:
        X_ints.append([vocab_to_int["UNK"] if word not in
            vocab_to_int else vocab_to_int[word] for word in
            sentence])
        X_lengths.append(len(sentence))
    return X_ints, X_lengths


X_train_ints, train_sentence_lengths = convert_words_to_idx(
   X_train)
X_val_ints, val_sentence_lengths = convert_words_to_idx(X_val)
X_test_ints, test_sentence_lengths = convert_words_to_idx(
   X_test)

def vocab_embeddings(vocab):
    k = 0
    embedding = np.zeros((len(vocab),300))
    for word in vocab:
        if word in word2vec.vocab:
            embedding[k] = word2vec[word]
        else:
            embedding[k] = word2vec['UNK']
        k = k + 1
    return embedding

embedding_matrix = vocab_embeddings(vocab)
embedding_matrix.shape

for i in range(len(embedding_matrix)):
    if vocab[i] in word2vec.vocab:
        if (word2vec[vocab[i]] != embedding_matrix[i]).all():
            print(vocab[i], "wrong")
    else:
        if (word2vec["UNK"] != embedding_matrix[i]).all():
            print(vocab[i], "wrong")

del word2vec

def left_pad(X_ints, seq_length):

    features = np.zeros((len(X_ints), seq_length), dtype=int)
    for i, row in enumerate(X_ints):
        features[i, -len(row):] = np.array(row)
    return features
```

```
def right_pad(X_ints, seq_length):

    features = np.zeros((len(X_ints), seq_length), dtype=int)
    for i, row in enumerate(X_ints):
        features[i, :len(row)] = np.array(row)
    return features



seq_length = max(train_sentence_lengths)

features_train = left_pad(X_train_ints, seq_length=seq_length)
features_val = left_pad(X_val_ints, seq_length=seq_length)

test_seq_length = max(test_sentence_lengths)
features_test = left_pad(X_test_ints, seq_length=
   test_seq_length)

print(features_train.shape)
print(features_val.shape)
print(features_test.shape)

assert len(features_train)==len(X_train),
assert len(features_train[0])==seq_length,
assert len(features_val)==len(X_val),
assert len(features_val[0])==seq_length,
assert len(features_test)==len(X_test),
assert len(features_test[0])==test_seq_length),

# converting the embedding matrix into a tensor
pretrained_embeddings = torch.FloatTensor(embedding_matrix)
```

**Main RNN Model:**

```
import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt

if torch.cuda.is_available():
    device = torch.device("cuda")
    print('GPU IS AVAILABLE :D')
else:
    device = torch.device("cpu")
    print('GPU not available')

training_features_tensor = torch.tensor(features_train, dtype=
   torch.float).to(device)
training_sentence_lengths = torch.tensor(train_sentence_lengths
   , dtype=torch.float).to(device)
training_labels_tensor = torch.tensor(y_train, dtype=torch.
   float).to(device)
val_features_tensor = torch.tensor(features_val, dtype=torch.
   float).to(device)
```

```
val_sentence_lengths = torch.tensor(val_sentence_lengths, dtype
   =torch.float).to(device)
val_labels_tensor = torch.tensor(y_val, dtype=torch.float).to(
   device)
test_features_tensor = torch.tensor(features_test, dtype=torch.
   float).to(device)
test_sentence_lengths = torch.tensor(test_sentence_lengths,
   dtype=torch.float).to(device)
test_labels_tensor = torch.tensor(y_test, dtype=torch.float).to
   (device)


# Final Hyperparameters used for all the model variants
LEARNING_RATE = 0.0001
EPOCHS = 20
BATCH_SIZE = 128
hidden_layer_1 = 128
emb_dim = 300
hidden_dim = 400
output_size = 5

trainset = torch.utils.data.TensorDataset(
   training_features_tensor, training_labels_tensor,
   training_sentence_lengths)
valset = torch.utils.data.TensorDataset(val_features_tensor,
   val_labels_tensor, val_sentence_lengths)
testset = torch.utils.data.TensorDataset(test_features_tensor,
   test_labels_tensor, test_sentence_lengths)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=
   BATCH_SIZE, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=
   BATCH_SIZE, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=
   BATCH_SIZE, shuffle=True)

class RNN(nn.Module):
    def __init__(self, pre_trained_embeddings, emb_dim,
       hidden_dim):
        super(RNN, self).__init__()
        self.emb_dim = emb_dim
        self.hidden_dim = hidden_dim
        self.embeddings = nn.Embedding.from_pretrained(
           pre_trained_embeddings, freeze=True)
        self.rnn = nn.RNN(emb_dim, hidden_dim, 1, batch_first=
           True)
        self.fc1 = nn.Linear(hidden_dim, hidden_layer_1)
        self.drop_layer = nn.Dropout(p= 0.3)
        # self.fc2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        # self.drop_layer_2 = nn.Dropout(p= 0.2)
        self.fc3 = nn.Linear(hidden_layer_1, output_size)

    def init_hidden(self, batch_size):
        hidden = torch.zeros(1, batch_size, self.hidden_dim).to
           (device)
        # hidden = torch.zeros(1, batch_size, self.hidden_dim)
        return hidden

    def forward(self, x, batch_size, x_lens):
        input = x.long().type(torch.LongTensor)
        lengths = x_lens.long().type(torch.LongTensor)
        hidden = self.init_hidden(batch_size)
        cell = self.init_hidden(batch_size)
        embeds = self.embeddings(input) # (b_z, m_l, emb_dim)
        output, hidden = self.rnn(embeds)
        final = output[:,-1,:]

        x = torch.tanh(self.fc1(final))
        x = self.drop_layer(x) # if relu, apply dropout before
           activation
        # x = torch.tanh(self.fc2(x))
        # x = self.drop_layer_2(x)
        x = self.fc3(x)
        return x


## training and validation function ##

def train_val():

# using the cross entropy loss
  criterion = nn.CrossEntropyLoss()
  # using Adam optimizer
  optimizer = torch.optim.Adam(net.parameters(), lr=
     LEARNING_RATE)

  train_costs = []
  train_accuracies = []
  val_costs = []
  val_accuracies = []
  num_epochs = []

  patience = 10
  steps = 0

  for epoch in range(EPOCHS):

      num_of_batches = 0
      val_batch = 0
      running_train_loss = 0.0
      running_train_acc = 0.0
      running_val_loss = 0.0
      running_val_acc = 0.0

      for i, (inputs, labels, lengths) in enumerate(trainloader
         , 0):
          optimizer.zero_grad()
          logits = net(inputs, BATCH_SIZE, lengths).to(device)
          targets = labels.type(torch.LongTensor).to(device)
          loss = criterion(logits, targets)
          loss.backward()
          optimizer.step()

          num_of_batches += 1
          running_train_loss += loss.item()

          _, train_predictions = torch.max(logits.data, 1)
```

```
            train_correct_predictions = (train_predictions.int()
                == labels.int()).sum()
            train_length = labels.size()[0]
            running_train_acc += train_correct_predictions/
                train_length

        for i, (inputs, labels, lengths) in enumerate(valloader,
            0):
            val_logits = net(inputs, BATCH_SIZE, lengths).to(
                device)
            val_loss = criterion(val_logits, labels.type(torch.
                LongTensor).to(device))
            running_val_loss += val_loss.item()

            _, val_predictions = torch.max(val_logits.data, 1)
            val_correct_predictions = (val_predictions.int() ==
                labels.int()).sum()
            val_length = labels.size()[0]
            running_val_acc += val_correct_predictions/val_length
            val_batch += 1


        train_costs.append(running_train_loss / num_of_batches)
        train_accuracies.append(running_train_acc /
            num_of_batches)
        val_costs.append(running_val_loss / val_batch)
        val_accuracies.append(running_val_acc / val_batch)
        num_epochs.append(epoch)

        if epoch > 0 and val_costs[epoch] > val_costs[epoch-1]:
            steps += 1
        else:
            steps = 0

        if steps == patience:
            break

        if epoch % 1 == 0:
            print('Epoch: %d/%d, train loss: %.3f, train acc: %.3
                f val loss: %.3f, val acc: %.3f, step:  %.3f'%(
                epoch + 1,EPOCHS, running_train_loss /
                num_of_batches, running_train_acc /
                num_of_batches, running_val_loss / val_batch,
                running_val_acc / val_batch, int(steps)))

    print('Final Train loss: %.3f -- Validation Loss: %.3f'%(
        train_costs[-1],val_costs[-1]))
    print("Final Train accuracy: %.3f -- Validation Accuracy: %.3
        f"%(train_accuracies[-1].item(), val_accuracies[-1].item
        ()))

    return train_costs, train_accuracies, val_costs,
        val_accuracies, num_epochs
```

### 3.5.2 Explore model variants with different recurrent units (“vanilla”, LSTM, and GRU), number of layers, keeping the word embeddings fixed during training or updating them, different depths and layer sizes for the multi-layer perceptron, etc. to find what works well on the validation set and describe the process through which you arrive at your final model.

In this section, the class definitions of the LSTM and GRU model variants are given. The Vanilla RNN was defined in the previous section.

**LSTM Model:**

```
class LSTM(nn.Module):
    def __init__(self, pre_trained_embeddings, emb_dim,
        hidden_dim):
        super(LSTM, self).__init__()
        self.emb_dim = emb_dim
        self.hidden_dim = hidden_dim
        self.embeddings = nn.Embedding.from_pretrained(
            pre_trained_embeddings, freeze=True)
        self.lstm = nn.LSTM(emb_dim, hidden_dim, 1, batch_first
            =True)
        self.fc1 = nn.Linear(hidden_dim, hidden_layer_1)
        self.drop_layer = nn.Dropout(p= 0.3)
        # self.fc2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        # self.drop_layer_2 = nn.Dropout(p= 0.2)
        self.fc3 = nn.Linear(hidden_layer_1, output_size)

    # def init_hidden(self, batch_size):
        # hidden = torch.zeros(1, batch_size, self.hidden_dim).
            to(device)
        # hidden = torch.zeros(1, batch_size, self.hidden_dim)
        # return hidden

    def forward(self, x, batch_size, x_lens):
        input = x.long().type(torch.LongTensor)
        lengths = x_lens.long().type(torch.LongTensor)
        # hidden = self.init_hidden(batch_size)
        # cell = self.init_hidden(batch_size)
        embeds = self.embeddings(input)
        output, hidden = self.lstm(embeds)
        # taking the final node of RNN as the output
        final = output[:,-1,:]
        #applying tanh activation
        x = torch.tanh(self.fc1(final))
        # using dropout for regularization
        x = self.drop_layer(x) # if relu, apply dropout before
            activation
        # x = torch.tanh(self.fc2(x))
        # x = self.drop_layer_2(x)
        # linear output layer
        x = self.fc3(x)
        return x
```

**GRU Model:**

```
class GRU(nn.Module):
    def __init__(self, pre_trained_embeddings, emb_dim,
        hidden_dim):
        super(GRU, self).__init__()
        self.emb_dim = emb_dim
        self.hidden_dim = hidden_dim
        self.embeddings = nn.Embedding.from_pretrained(
            pre_trained_embeddings, freeze=True)
        self.gru = nn.GRU(emb_dim, hidden_dim, 1, batch_first=
            True)
        self.fc1 = nn.Linear(hidden_dim, hidden_layer_1)
        self.drop_layer = nn.Dropout(p= 0.3)
        # self.fc2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        # self.drop_layer_2 = nn.Dropout(p= 0.2)
        self.fc3 = nn.Linear(hidden_layer_1, output_size)

    def init_hidden(self, batch_size):
        hidden = torch.zeros(1, batch_size, self.hidden_dim).to
            (device)
        # hidden = torch.zeros(1, batch_size, self.hidden_dim)
        return hidden

    def forward(self, x, batch_size, x_lens):
        input = x.long().type(torch.LongTensor)
        lengths = x_lens.long().type(torch.LongTensor)
        hidden = self.init_hidden(batch_size)
        cell = self.init_hidden(batch_size)
        embeds = self.embeddings(input) # (b_z, m_l, emb_dim)
        output, hidden = self.gru(embeds)
        final = output[:,-1,:]

        x = torch.tanh(self.fc1(final))
        x = self.drop_layer(x) # if relu, apply dropout before
            activation
        # x = torch.tanh(self.fc2(x))
        # x = self.drop_layer_2(x)
        x = self.fc3(x)
        return x
```

The training and validation code remains the same as the one posted in section 3.5.1.

Another variant experimented with was updating embeddings while training. The classes defined above are for the case when the embeddings were "fixed" during training. Setting freeze = False ensures that the embeddings were updated during training.

```
self.embeddings = nn.Embedding.from_pretrained(
    pre_trained_embeddings, freeze=False)
```

The following results were observed for updated embeddings:

**1) RNN:**

```
Epoch: 1/20, train loss: 1.539, train acc: 0.313 val loss:
    1.517, val acc: 0.336, step:  0.000
Epoch: 2/20, train loss: 1.269, train acc: 0.460 val loss:
    1.398, val acc: 0.399, step:  0.000
Epoch: 3/20, train loss: 1.008, train acc: 0.560 val loss:
    1.501, val acc: 0.354, step:  1.000
Epoch: 4/20, train loss: 0.829, train acc: 0.665 val loss:
    1.624, val acc: 0.383, step:  2.000
Epoch: 5/20, train loss: 0.653, train acc: 0.780 val loss:
    1.782, val acc: 0.380, step:  3.000
Epoch: 6/20, train loss: 0.531, train acc: 0.841 val loss:
    1.979, val acc: 0.376, step:  4.000
Best val accuracy:  tensor(0.3989, device='cuda:0') at epoch:
    1
```

**2) LSTM:**

```
Epoch: 1/20, train loss: 1.572, train acc: 0.292 val loss:
    1.563, val acc: 0.295, step:  0.000
Epoch: 2/20, train loss: 1.448, train acc: 0.397 val loss:
    1.413, val acc: 0.382, step:  0.000
Epoch: 3/20, train loss: 1.033, train acc: 0.563 val loss:
    1.481, val acc: 0.386, step:  1.000
Epoch: 4/20, train loss: 0.879, train acc: 0.650 val loss:
    1.601, val acc: 0.374, step:  2.000
Epoch: 5/20, train loss: 0.631, train acc: 0.790 val loss:
    1.787, val acc: 0.387, step:  3.000
Epoch: 6/20, train loss: 0.502, train acc: 0.849 val loss:
    2.099, val acc: 0.360, step:  4.000
```

**3) GRU:**

```
Epoch: 1/20, train loss: 1.560, train acc: 0.300 val loss:
    1.550, val acc: 0.297, step:  0.000
Epoch: 2/20, train loss: 1.323, train acc: 0.465 val loss:
    1.598, val acc: 0.305, step:  1.000
Epoch: 3/20, train loss: 1.077, train acc: 0.552 val loss:
    1.438, val acc: 0.394, step:  0.000
Epoch: 4/20, train loss: 0.784, train acc: 0.702 val loss:
    1.613, val acc: 0.393, step:  1.000
Epoch: 5/20, train loss: 0.590, train acc: 0.823 val loss:
    1.887, val acc: 0.382, step:  2.000
Epoch: 6/20, train loss: 0.455, train acc: 0.869 val loss:
    2.178, val acc: 0.369, step:  3.000
Epoch: 7/20, train loss: 0.376, train acc: 0.889 val loss:
    2.291, val acc: 0.381, step:  4.000
```

It can be observed that the final validation accuracy on each variant is coming out lower than on its fixed embedded counterpart. Patience was set to 5 for the early stopping algorithm as the model was overfitting for the earlier epochs. It seemed that at this point, the model started to memorize the training data. From this it was concluded that it is better to fix the embeddings while training.

- "vanilla" RNN: training accuracy = 84.1%, training loss = 0.531, validation accuracy = 37.6% and validation loss = 1.979
- LSTM RNN: training accuracy = 84.9%, training loss = 0.502, validation accuracy = 36.0% and validation loss = 2.099
- GRU RNN: training accuracy = 88.9%, training loss = 0.376, validation accuracy = 38.1% and validation loss = 2.291

### 3.5.3 Train your final model to convergence on the training set using an optimisation algorithm of your choice.

The hyperparameters that were explored were chosen on the basis of prior experience in training MLP models, ideally a grid search method would have been implemented to arrive at the optimal hyperparameter combination, but this was too GPU intensive for the problem. The optimal hyperparamters for the MLP were: number of layers = 1, hidden nodes = 128, learning rate = 0.0001, batch size = 128, activation function = tanh, word embedding = fixed. Early stopping was implemented in the model to stop it overfitting, the patience was set to 10 and the model was trained for around 20 - 25 epochs, as the RAM allocated was over-used. It was found that the MLP used in this model performed better with fewer layers than required in the MLP architecture in 3.4. The Adam optimiser was chosen using the same reasoning as in 3.1.

**Process to arrive at the final model:**

- Build a "vanilla" RNN that feeds into an MLP with any reasonable hyperparameters (to be tuned)
- On the MLP, test out different hyperparameter values for the number of layers, hidden nodes, learning rate, batch size, MLP activation function and changing between updating the word embeddings and keeping them fixed during training
- Test the following hyperparameter values: number of layers = [1, 2], hidden nodes = [32, 64, 128], learning rate = [0.01, 0.001, 0.0001], batch size = [64, 128, 256, 1024], activation function = [tanh, ReLU] and word embeddings = [fixed, updating]
- The hyperparameter combinations that perform best on the validation set are to be used in the MLP for LSTM and GRU RNN model variants
- Store the highest validation accuracy's on each model variant
- Use the model variant with the highest overall validation accuracy as the final model to use on the test set

The LSTM and GRU outperformed the "vanilla" RNN, this was expected due to their capacity to retain information in a memory from previous iterations in the RNN. It was also found that fixing the word embeddings led to higher validation accuracies for the training and validation sets. The LSTM had the highest validation accuracy ($42.5\%$) out of the three variants, and as such was the best candidate for the final model.

Figures 19, 20 and 21 show the loss and accuracy plots for the "vanilla", LSTM and GRU model variants on the RNN, respectively. The loss and accuracy achieved for each model variant is presented below:

**"vanilla" RNN**

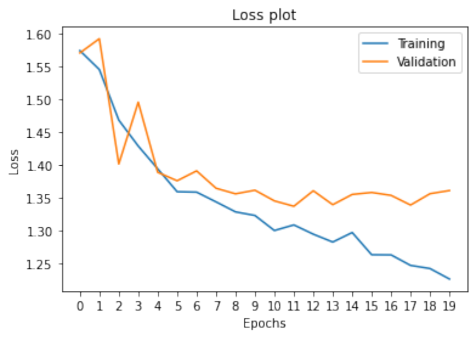
- Training: accuracy = $47.5\%$, loss = 1.227
- Validation: accuracy = $40.3\%$, loss = 1.361

**LSTM RNN**

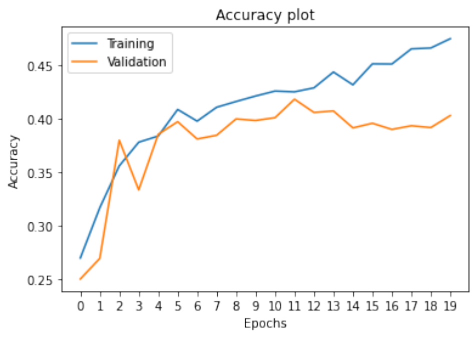
- Training: accuracy = $44.3\%$, loss = 1.267
- Validation: accuracy = $42.5\%$, loss = 1.343

**GRU RNN**

- Training: accuracy = $45.5\%$, loss = 1.246
- Validation: accuracy = $41.0\%$, loss = 1.322

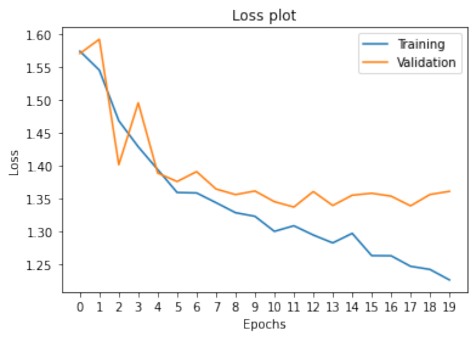


(a) Training and validation loss

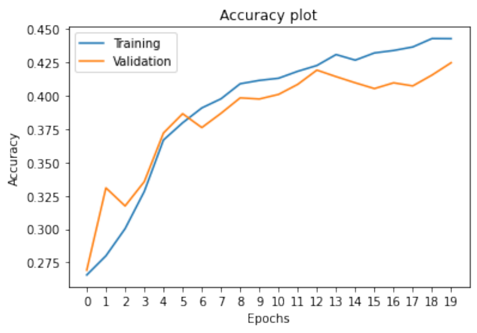


(b) Training and validation accuracy

Figure 19: "vanilla" variant training and validation

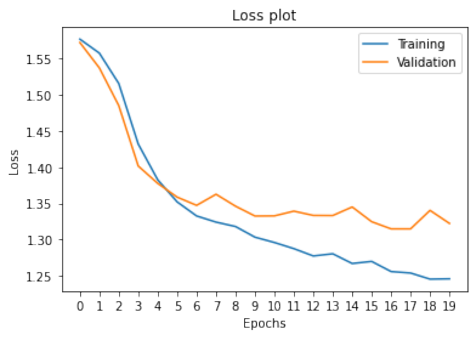


(a) Training and validation loss

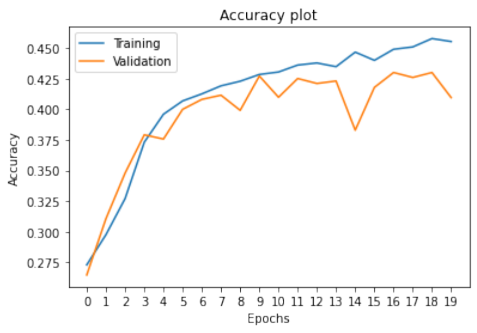


(b) Training and validation accuracy

Figure 20: LSTM variant training and validation



(a) Training and validation loss



(b) Training and validation accuracy

Figure 21: GRU variant training and validation

**Model variant and final model outputs:**

```
## "vanilla" RNN variant of the model ##

net = RNN(pretrained_embeddings, emb_dim, hidden_dim) # specify
    model variant
train_costs, train_accuracies, val_costs, val_accuracies,
   num_epochs = train_val() # train the model and evaluate on
   the validation set

output:
Final Train loss: 1.227 -- Validation Loss: 1.361
Final Train accuracy: 0.475 -- Validation Accuracy: 0.403


## LSTM RNN variant of the model ##

net = LSTM(pretrained_embeddings, emb_dim, hidden_dim) #
   specify model variant
train_costs, train_accuracies, val_costs, val_accuracies,
   num_epochs = train_val() # train the model and evaluate on
   the validation set

output:
Final Train loss: 1.267 -- Validation Loss: 1.343
Final Train accuracy: 0.443 -- Validation Accuracy: 0.425
```

```
## GRU RNN variant of the model ##

net = GRU(pretrained_embeddings, emb_dim, hidden_dim) # specify
     model variant
train_costs, train_accuracies, val_costs, val_accuracies,
    num_epochs = train_val() # train the model and evaluate on
    the validation set

output:
Final Train loss: 1.246 -- Validation Loss: 1.322
Final Train accuracy: 0.455 -- Validation Accuracy: 0.410
```

**Final Model:**
In the model variant evaluation above LSTM performed the best with a validation accuracy of 42.5%, this model variant will be used as the final model

```
## final model ##
# specify model variant
net = LSTM(pretrained_embeddings, emb_dim, hidden_dim)

print('Training...')
# train the model and evaluate on the validation set
train_costs, train_accuracies, val_costs, val_accuracies,
    num_epochs = train_val()
print('Testing...')

## test function ##
def test():
  test_batch = 0
  running_test_acc = 0
  running_test_loss = 0

  for i, (inputs, labels, lengths) in enumerate(testloader, 0):
          test_logits = net(inputs, BATCH_SIZE, lengths).to(
              device)
          test_loss = criterion(test_logits, labels.type(torch.
              LongTensor).to(device))
          running_test_loss += test_loss.item()

          _, test_predictions = torch.max(test_logits.data, 1)
          test_correct_predictions = (test_predictions.int() ==
               labels.int()).sum()
          test_length = labels.size()[0]
          running_test_acc += test_correct_predictions/
              test_length
          test_batch += 1

  print('Final Accuracy on the test set: ', running_test_acc/
      test_batch)


#plot loss for train and validation set

plt.plot(num_epochs, train_costs, label = 'Training')
plt.plot(num_epochs, val_costs, label = 'Validation')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.xticks(num_epochs)
plt.legend()
plt.title('Loss plot')
plt.show()

# plot accuracy for train and validation set

plt.plot(num_epochs, train_accuracies, label = 'Training')
plt.plot(num_epochs, val_accuracies, label = 'Validation')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.xticks(num_epochs)
plt.title('Accuracy plot')
plt.show()

criterion = nn.CrossEntropyLoss()
test()
```

**Output:**

```
Final Train loss: 1.252 -- Validation Loss: 1.318
Final Train accuracy: 0.454 -- Validation Accuracy: 0.432
Testing...
Final Accuracy on the test set:  tensor(0.4263, device='cuda:0'
    )
```

#### 3.5.4 Provide a plot of the loss on the training set and validation set for each epoch of training.

The plots for the final model training and validation can be seen in figures 22 and 23. The code for plotting the accuracy and loss plots is below. Note that on the x-axis Epochs start from 0 instead of 1.

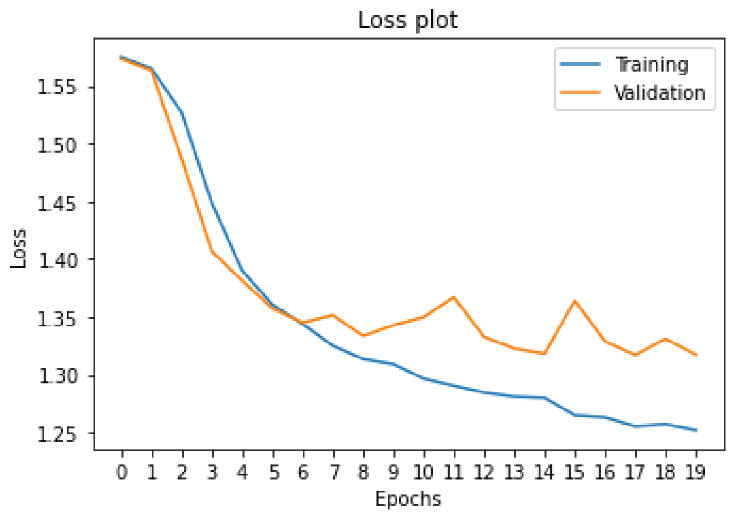


Figure 22: Loss for the training and validation set on the final model (LSTM variant)

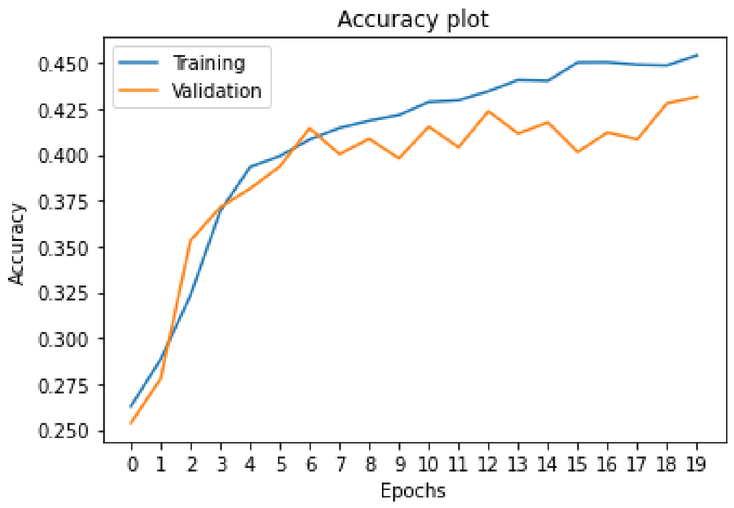


Figure 23: Accuracy for the training and validation set on the final model (LSTM variant)

#### 3.5.5 Provide the final accuracy on the training, validation, and test set.

The final model variant chosen was the LSTM, the final accuracy on the training, validation and test set is:

- Training Accuracy = $45.4\%$
- Validation Accuracy = $43.2\%$
- Test Accuracy = $42.6\%$

### 3.5.6 Provide a selection of the classification decisions of the final model for 5 opinionated sentences from movies reviews that you find online – perhaps for a movie you liked, or did not like? Remember to provide a links to the reviews from which you selected the sentences.

The model can be used to classify movie reviews into how well they were received by critics, as shown in table 3. Movie reviews have been sourced from Rotten Tomatoes for the films *Murder Mystery* (Tomatoes, 2019a), *Toy Story 4* (Tomatoes, 2019b), *The Spongebob Squarepants Movie* (Tomatoes, 2004), *The Human Centipede* (B, 2010) and CINEMABLEND for *The Judge* (Brent McKnight, 2014). The films were chosen to demonstrate the models ability to identify different sentiments correctly.

Table 3: Class labels

| Class | Sentiment |
|---|---|
| 0 | 'Extremely Negative' |
| 1 | 'Negative' |
| 2 | 'Neutral' |
| 3 | 'Positive' |
| 4 | 'Extremely Positive' |

The results displayed below show that the model correctly classifies the five movie reviews that were tested. It seems to perform poorly when classifying what humans may perceive as a "neutral" review, this may be due to the fact that it looks for key words with positive connotations such as "good", "great", "amazing" ect... to classify the review as "positive", with the same logic applied to "negative" connotations. If a review contains even one such word from either connotation it could confuse the model into thinking it is "positive" or "negative", when in actual fact it is more of a "neutral" opinion. In order to establish that the model is able of classifying a review as "neutral", a made up sentence "the movie is alright" was tested (note that this is not shown in the final output), it correctly identified this fake review as "neutral". This is most likely due to the absence of both positive and negative connotations, humans tend to be quite emotive in speech so it is hard to real examples of reviews with a similar language structure to the fake example.

**Code for classifying movie reviews:**

```
!pip install nltk
nltk.download('punkt')
import nltk
movie_reviews = [['Heartwarming, funny, and beautifully
    animated, Toy Story 4 manages the unlikely feat of
    extending , and perhaps concluding , a practically perfect
    animated saga.'],
           ["Murder Mystery reunites Jennifer Aniston and Adam
              Sandler for a lightweight comedy that's content
              to settle for merely mediocre."],
           ["While watching these two actors share the screen as
               an estranged father and son is compelling, that'
              s really all The Judge has going for it. Outside
              of Duvall and RDJ, the script is super formulaic.
              "],
           ["A film for kids, students, stoners, anyone who
              enjoys a break from reality."],
           ["It's artless and repulsive. I've seen many other
              films that are grotesque or disturbing and even
              plain disgusting but when the whole film is this
              way just for that sole purpose it becomes
              worthless."]]
```

```
X_test = []
for sentence in movie_reviews:
    X_test.append(nltk.word_tokenize(sentence[0]))

X_test_ints = []
test_sentence_lengths = []
for sentence in X_test:
    X_test_ints.append([vocab_to_int["UNK"] if word not in
        vocab_to_int else vocab_to_int[word] for word in
        sentence])
    test_sentence_lengths.append(len(sentence))

test_seq_length = max(test_sentence_lengths)
features_test = left_pad(X_test_ints, seq_length=
   test_seq_length)
assert len(features_test[0]==test_seq_length)

test_features_tensor = torch.tensor(features_test, dtype=torch.
   float).to(device)


test_batch = 0
running_test_acc = 0
running_test_loss = 0

net.eval()

idx_to_sent = {
    0: 'Extremely Negative',
    1: 'Negative',
    2: 'Neutral',
    3: 'Positive',
    4: 'Extremely Positive'}

test_logits = net(test_features_tensor, 5, torch.Tensor(
   test_sentence_lengths)).to(device)
_, test_predictions = torch.max(test_logits.data, 1)
for i in range(len(movie_reviews)):
    print('Review: ', movie_reviews[i][0])
    print('Sentiment: ',idx_to_sent[test_predictions[i].item()
        ])
    print('Class:', test_predictions[i].item())
    print('--------------------------------------------------')



output:
Review:  Heartwarming, funny, and beautifully animated, Toy
   Story 4 manages the unlikely feat of extending , and
   perhaps concluding , a practically perfect animated saga.
Sentiment:  Extremely Positive
Class: 4
--------------------------------------------------
Review:  Murder Mystery reunites Jennifer Aniston and Adam
   Sandler for a lightweight comedy that's content to settle
   for merely mediocre.
Sentiment:  Negative
Class: 1
--------------------------------------------------
```

```
Review:  While watching these two actors share the screen as an
    estranged father and son is compelling, that's really all
   The Judge has going for it. Outside of Duvall and RDJ, the
   script is super formulaic.
Sentiment:  Negative
Class: 1
--------------------------------------------------
Review:  A film for kids, students, stoners, anyone who enjoys
   a break from reality.
Sentiment:  Positive
Class: 3
--------------------------------------------------
Review:  It's artless and repulsive. I've seen many other films
    that are grotesque or disturbing and even plain disgusting
    but when the whole film is this way just for that sole
   purpose it becomes worthless.
Sentiment:  Extremely Negative
Class: 0
--------------------------------------------------
```

## References

Gentle introduction to the adam optimisation algorithm for deep learning, 2017. URL https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/#:~:text=Adam%20is%20an%20optimization%20algorithm,iterative%20based%20in%20training%20data.&text=The%20algorithm%20is%20called%20Adam,derived%20from%20adaptive%20moment%20estimation.

Rotten Toamtoes Chris B. The human centipede, 2010. URL https://www.rottentomatoes.com/m/human_centipede.

CINEMABLEND Brent McKnight. The 15 most remarkably average movies of all time: The judge, 2014. URL https://www.cinemablend.com/new/15-Most-Remarkably-Average-Movies-All-Time-113387.html.

Rotten Tomatoes. The spongebob squarepants movie, 2004. URL https://www.rottentomatoes.com/m/spongebob_squarepants_movie.

Rotten Tomatoes. Murder mystery, 2019a. URL https://www.rottentomatoes.com/m/murder_mystery.

Rotten Tomatoes. Toy story 4, 2019b. URL https://www.rottentomatoes.com/m/toy_story_4.