|  |  |
| --- | --- |
| Name Of The Student | Aman Rai |
| Internship Project Topic | Automate Detection of different emotions from textual comments and feedback |
| Name of the Organization | TCS iON |
| Name of the Industry Mentor | Mr. Debashis Roy |
| Name of the Institute | Institute of Engineering & Management Kolkata |

|  |  |  |
| --- | --- | --- |
| Date | Day # | Hours Spent |
| 29 August 2020 | 10 | 6+ |
| Activities done during the day: On this last activity I am try combine all the code necessary for the Model.  ##Code  **import** **numpy** **as** **np**  **import** **pandas** **as** **pd**  **import** **matplotlib.pyplot** **as** **plt**  **import** **csv**  **import** **torch**  **import** **torch.nn** **as** **nn**  **import** **torch.nn.functional** **as** **F**  **import** **torch.optim** **as** **optim**  *# HELPER FUNCTIONS*  **def** read\_glove\_vecs(glove\_file):  **with** open(glove\_file, 'r') **as** f:  words = set()  word\_to\_vec\_map = {}  **for** line **in** f:  line = line.strip().split()  curr\_word = line[0]  words.add(curr\_word)  word\_to\_vec\_map[curr\_word] = np.array(line[1:], dtype=np.float64)    i = 1  words\_to\_index = {}  index\_to\_words = {}  **for** w **in** sorted(words):  words\_to\_index[w] = i  index\_to\_words[i] = w  i = i + 1  **return** words\_to\_index, index\_to\_words, word\_to\_vec\_map  **def** convert\_to\_one\_hot(Y, C):  Y = np.eye(C)[Y.reshape(-1)]  **return** Y  **def** read\_csv(filename):  phrase = []  emoji = []  **with** open (filename) **as** csvDataFile:  csvReader = csv.reader(csvDataFile)  **for** row **in** csvReader:  phrase.append(row[0])  emoji.append(row[1])  X = np.asarray(phrase)  Y = np.asarray(emoji, dtype=int)  **return** X, Y  X\_train, Y\_train = read\_csv('train.csv')  X\_test, Y\_test = read\_csv('test.csv')  Y\_oh\_train = convert\_to\_one\_hot(Y\_train, C = 5)  Y\_oh\_test = convert\_to\_one\_hot(Y\_test, C = 5)  word\_to\_index, index\_to\_word, word\_to\_vec\_map = read\_glove\_vecs('glove.6B.50d.txt')  **def** sentences\_to\_indices(X, word\_to\_index, max\_len):  *"""*  *Converts an array of sentences (strings) into an array of indices corresponding to words in the sentences.*  *"""*    m = X.shape[0] *# number of training examples*    *# Initialize X\_indices as a numpy matrix of zeros and the correct shape*  X\_indices = np.zeros((m,max\_len))    **for** i **in** range(m): *# loop over training examples*    *# Convert the ith sentence in lower case and split into a list of words*  sentence\_words = X[i].lower().split()    *# Initialize j to 0*  j = 0    *# Loop over the words of sentence\_words*  **for** w **in** sentence\_words:  *# Set the (i,j)th entry of X\_indices to the index of the correct word.*  X\_indices[i, j] = word\_to\_index[w]  *# Increment j to j + 1*  j = j + 1    **return** X\_indices  X1 = np.array(["lol", "I love you", "this is very yummy"])  X1\_indices = sentences\_to\_indices(X1,word\_to\_index, max\_len = 5)  print("X1 =", X1)  print("X1\_indices =", X1\_indices)   ## Defining the Network using Pretrained Embedding Layer using GloVe Word Embeddings **class** **NN**(nn.Module):  **def** \_\_init\_\_(self, embedding, embedding\_dim, hidden\_dim, vocab\_size, output\_dim, batch\_size):  super(NN, self).\_\_init\_\_()  self.batch\_size = batch\_size  self.hidden\_dim = hidden\_dim  self.word\_embeddings = embedding  *# The LSTM takes word embeddings as inputs, and outputs hidden states*  *# with dimensionality hidden\_dim.*  self.lstm = nn.LSTM(embedding\_dim,  hidden\_dim,  num\_layers=2,  dropout = 0.5,  batch\_first = **True**)  *# The linear layer that maps from hidden state space to output space*  self.fc = nn.Linear(hidden\_dim, output\_dim)  **def** forward(self, sentence):    *#sentence = sentence.type(torch.LongTensor)*  *#print ('Shape of sentence is:', sentence.shape)*  sentence = sentence.to(device)  embeds = self.word\_embeddings(sentence)  *#print ('Embedding layer output shape', embeds.shape)*  *# initializing the hidden state to 0*  *#hidden=None*    h0 = torch.zeros(2, sentence.size(0), hidden\_dim).requires\_grad\_().to(device)  c0 = torch.zeros(2, sentence.size(0), hidden\_dim).requires\_grad\_().to(device)    lstm\_out, h = self.lstm(embeds, (h0, c0))  *# get info from last timestep only*  lstm\_out = lstm\_out[:, -1, :]  *#print ('LSTM layer output shape', lstm\_out.shape)*  *#print ('LSTM layer output ', lstm\_out)*  *# Dropout*  lstm\_out = F.dropout(lstm\_out, 0.5)  fc\_out = self.fc(lstm\_out)  *#print ('FC layer output shape', fc\_out.shape)*  *#print ('FC layer output ', fc\_out)*    out = fc\_out  out = F.softmax(out, dim=1)  *#print ('Output layer output shape', out.shape)*  *#print ('Output layer output ', out)*  **return** out ## Creating the Glove Embedding Layer def pretrained\_embedding\_layer(word\_to\_vec\_map, word\_to\_index, non\_trainable=True):  num\_embeddings = len(word\_to\_index) + 1  embedding\_dim = word\_to\_vec\_map["cucumber"].shape[0] *# dimensionality of GloVe word vectors (= 50)*  *# Initialize the embedding matrix as a numpy array of zeros of shape (num\_embeddings, embedding\_dim)*  weights\_matrix = np.zeros((num\_embeddings, embedding\_dim))  *# Set each row "index" of the embedding matrix to be the word vector representation of the "index"th word of the vocabulary*  for word, index in word\_to\_index.items():  weights\_matrix[index, :] = word\_to\_vec\_map[word]  embed = nn.Embedding.from\_pretrained(torch.from\_numpy(weights\_matrix).type(torch.FloatTensor), freeze=non\_trainable)  return embed, num\_embeddings, embedding\_dim  **##Training the mode**  device = torch.device("cuda:0" **if** torch.cuda.is\_available() **else** "cpu")  **def** train(model, trainloader, criterion, optimizer, epochs=10):    model.to(device)  running\_loss = 0    train\_losses, test\_losses, accuracies = [], [], []  **for** e **in** range(epochs):  running\_loss = 0    model.train()    **for** sentences, labels **in** trainloader:  sentences, labels = sentences.to(device), labels.to(device)  *# 1) erase previous gradients (if they exist)*  optimizer.zero\_grad()  *# 2) make a prediction*  pred = model.forward(sentences)  *# 3) calculate how much we missed*  loss = criterion(pred, labels)  *# 4) figure out which weights caused us to miss*  loss.backward()  *# 5) change those weights*  optimizer.step()  *# 6) log our progress*  running\_loss += loss.item()      **else**:  model.eval()  test\_loss = 0  accuracy = 0    *# Turn off gradients for validation, saves memory and computations*  **with** torch.no\_grad():  **for** sentences, labels **in** test\_loader:  sentences, labels = sentences.to(device), labels.to(device)  log\_ps = model(sentences)  test\_loss += criterion(log\_ps, labels)    ps = torch.exp(log\_ps)  top\_p, top\_class = ps.topk(1, dim=1)  equals = top\_class == labels.view(\*top\_class.shape)  accuracy += torch.mean(equals.type(torch.FloatTensor))    train\_losses.append(running\_loss/len(train\_loader))  test\_losses.append(test\_loss/len(test\_loader))  accuracies.append(accuracy / len(test\_loader) \* 100)  print("Epoch: **{}**/**{}**.. ".format(e+1, epochs),  "Training Loss: **{:.3f}**.. ".format(running\_loss/len(train\_loader)),  "Test Loss: **{:.3f}**.. ".format(test\_loss/len(test\_loader)),  "Test Accuracy: **{:.3f}**".format(accuracy/len(test\_loader)))    *# Plot*  plt.figure(figsize=(20, 5))  plt.plot(train\_losses, c='b', label='Training loss')  plt.plot(test\_losses, c='r', label='Testing loss')  plt.xticks(np.arange(0, epochs))  plt.title('Losses')  plt.legend(loc='upper right')  plt.show()  plt.figure(figsize=(20, 5))  plt.plot(accuracies)  plt.xticks(np.arange(0, epochs))  plt.title('Accuracy')  plt.show()  **import** **torch.utils.data**  maxLen = len(max(X\_train, key=len).split())  X\_train\_indices = sentences\_to\_indices(X\_train, word\_to\_index, maxLen)  Y\_train\_oh = convert\_to\_one\_hot(Y\_train, C = 5)  X\_test\_indices = sentences\_to\_indices(X\_test, word\_to\_index, maxLen)  Y\_test\_oh = convert\_to\_one\_hot(Y\_test, C = 5)  embedding, vocab\_size, embedding\_dim = pretrained\_embedding\_layer(word\_to\_vec\_map, word\_to\_index, non\_trainable=**True**)  hidden\_dim=128  output\_size=5  batch\_size = 32  *#print ('Embedding layer is ', embedding)*  *#print ('Embedding layer weights ', embedding.weight.shape)*  model = NN(embedding, embedding\_dim, hidden\_dim, vocab\_size, output\_size, batch\_size)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=0.002)  epochs = 50  train\_dataset = torch.utils.data.TensorDataset(torch.tensor(X\_train\_indices).type(torch.LongTensor), torch.tensor(Y\_train).type(torch.LongTensor))  train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=batch\_size)  test\_dataset = torch.utils.data.TensorDataset(torch.tensor(X\_test\_indices).type(torch.LongTensor), torch.tensor(Y\_test).type(torch.LongTensor))  test\_loader = torch.utils.data.DataLoader(test\_dataset, batch\_size=batch\_size)  train(model, train\_loader, criterion, optimizer, epochs)  Epoch: 1/50.. Training Loss: 1.602.. Test Loss: 1.586.. Test Accuracy: 0.333  Epoch: 2/50.. Training Loss: 1.585.. Test Loss: 1.540.. Test Accuracy: 0.323  Epoch: 3/50.. Training Loss: 1.571.. Test Loss: 1.543.. Test Accuracy: 0.281  Epoch: 4/50.. Training Loss: 1.558.. Test Loss: 1.537.. Test Accuracy: 0.323  Epoch: 5/50.. Training Loss: 1.570.. Test Loss: 1.515.. Test Accuracy: 0.406  Epoch: 6/50.. Training Loss: 1.520.. Test Loss: 1.502.. Test Accuracy: 0.375  Epoch: 7/50.. Training Loss: 1.490.. Test Loss: 1.495.. Test Accuracy: 0.411  Epoch: 8/50.. Training Loss: 1.458.. Test Loss: 1.460.. Test Accuracy: 0.448  Epoch: 9/50.. Training Loss: 1.449.. Test Loss: 1.502.. Test Accuracy: 0.422  Epoch: 10/50.. Training Loss: 1.481.. Test Loss: 1.489.. Test Accuracy: 0.391  Epoch: 11/50.. Training Loss: 1.421.. Test Loss: 1.454.. Test Accuracy: 0.391  Epoch: 12/50.. Training Loss: 1.392.. Test Loss: 1.436.. Test Accuracy: 0.427  Epoch: 13/50.. Training Loss: 1.384.. Test Loss: 1.457.. Test Accuracy: 0.406  Epoch: 14/50.. Training Loss: 1.357.. Test Loss: 1.383.. Test Accuracy: 0.516  Epoch: 15/50.. Training Loss: 1.339.. Test Loss: 1.419.. Test Accuracy: 0.500  Epoch: 16/50.. Training Loss: 1.329.. Test Loss: 1.404.. Test Accuracy: 0.490  Epoch: 17/50.. Training Loss: 1.298.. Test Loss: 1.385.. Test Accuracy: 0.536  Epoch: 18/50.. Training Loss: 1.290.. Test Loss: 1.380.. Test Accuracy: 0.531  Epoch: 19/50.. Training Loss: 1.268.. Test Loss: 1.354.. Test Accuracy: 0.536  Epoch: 20/50.. Training Loss: 1.294.. Test Loss: 1.385.. Test Accuracy: 0.464  Epoch: 21/50.. Training Loss: 1.274.. Test Loss: 1.371.. Test Accuracy: 0.490  Epoch: 22/50.. Training Loss: 1.259.. Test Loss: 1.321.. Test Accuracy: 0.594  Epoch: 23/50.. Training Loss: 1.340.. Test Loss: 1.389.. Test Accuracy: 0.505  Epoch: 24/50.. Training Loss: 1.336.. Test Loss: 1.323.. Test Accuracy: 0.641  Epoch: 25/50.. Training Loss: 1.206.. Test Loss: 1.397.. Test Accuracy: 0.510  Epoch: 26/50.. Training Loss: 1.287.. Test Loss: 1.390.. Test Accuracy: 0.516  Epoch: 27/50.. Training Loss: 1.239.. Test Loss: 1.417.. Test Accuracy: 0.484  Epoch: 28/50.. Training Loss: 1.247.. Test Loss: 1.348.. Test Accuracy: 0.557  Epoch: 29/50.. Training Loss: 1.153.. Test Loss: 1.287.. Test Accuracy: 0.641  Epoch: 30/50.. Training Loss: 1.159.. Test Loss: 1.328.. Test Accuracy: 0.583  Epoch: 31/50.. Training Loss: 1.131.. Test Loss: 1.274.. Test Accuracy: 0.651  Epoch: 32/50.. Training Loss: 1.172.. Test Loss: 1.259.. Test Accuracy: 0.646  Epoch: 33/50.. Training Loss: 1.152.. Test Loss: 1.235.. Test Accuracy: 0.667  Epoch: 34/50.. Training Loss: 1.101.. Test Loss: 1.214.. Test Accuracy: 0.682  Epoch: 35/50.. Training Loss: 1.045.. Test Loss: 1.165.. Test Accuracy: 0.750  Epoch: 36/50.. Training Loss: 1.055.. Test Loss: 1.197.. Test Accuracy: 0.708  Epoch: 37/50.. Training Loss: 1.069.. Test Loss: 1.185.. Test Accuracy: 0.708  Epoch: 38/50.. Training Loss: 1.025.. Test Loss: 1.205.. Test Accuracy: 0.693  Epoch: 39/50.. Training Loss: 1.044.. Test Loss: 1.179.. Test Accuracy: 0.734  Epoch: 40/50.. Training Loss: 1.049.. Test Loss: 1.164.. Test Accuracy: 0.740  Epoch: 41/50.. Training Loss: 1.023.. Test Loss: 1.150.. Test Accuracy: 0.755  Epoch: 42/50.. Training Loss: 1.012.. Test Loss: 1.085.. Test Accuracy: 0.807  Epoch: 43/50.. Training Loss: 1.041.. Test Loss: 1.094.. Test Accuracy: 0.802  Epoch: 44/50.. Training Loss: 1.018.. Test Loss: 1.119.. Test Accuracy: 0.786  Epoch: 45/50.. Training Loss: 1.030.. Test Loss: 1.079.. Test Accuracy: 0.828  Epoch: 46/50.. Training Loss: 0.999.. Test Loss: 1.158.. Test Accuracy: 0.734  Epoch: 47/50.. Training Loss: 1.104.. Test Loss: 1.149.. Test Accuracy: 0.750  Epoch: 48/50.. Training Loss: 1.097.. Test Loss: 1.133.. Test Accuracy: 0.771  Epoch: 49/50.. Training Loss: 1.026.. Test Loss: 1.063.. Test Accuracy: 0.844  Epoch: 50/50.. Training Loss: 1.010.. Test Loss: 1.060.. Test Accuracy: 0.844     ##Testing the Model Accuracy test\_loss = 0  accuracy = 0  model.eval()  **with** torch.no\_grad():  **for** sentences, labels **in** test\_loader:  sentences, labels = sentences.to(device), labels.to(device)  ps = model(sentences)  test\_loss += criterion(ps, labels).item()  *# Accuracy*  top\_p, top\_class = ps.topk(1, dim=1)  equals = top\_class == labels.view(\*top\_class.shape)  accuracy += torch.mean(equals.type(torch.FloatTensor))  model.train()  print("Test Loss: **{:.3f}**.. ".format(test\_loss/len(test\_loader)),  "Test Accuracy: **{:.3f}**".format(accuracy/len(test\_loader)))  running\_loss = 0  Test Loss: 1.064.. Test Accuracy: 0.844 ##Testing the model with any sentence **def** predict(input\_text, print\_sentence=**True**):  labels\_dict = {  0 : "❤️ Loving",  1 : "⚽️ Playful",  2 : "😄 Happy",  3 : "😞 Annoyed",  4 : "🍽 Foodie",  }  *# Convert the input to the model*  x\_test = np.array([input\_text])  X\_test\_indices = sentences\_to\_indices(x\_test, word\_to\_index, maxLen)  sentences = torch.tensor(X\_test\_indices).type(torch.LongTensor)  *# Get the class label*  ps = model(sentences)  top\_p, top\_class = ps.topk(1, dim=1)  label = int(top\_class[0][0])  **if** print\_sentence:  print("**\n**Input Text: **\t**"+ input\_text +'**\n**Emotion: **\t**'+ labels\_dict[label])  **return** label  *# Change the sentence below to see your prediction. Make sure all the words are in the Glove embeddings.*  print("------------------------------------")  predict("I hate you")  predict("I want a pizza")  predict("Lets see the game")  predict("I love you Lisa")  predict("This is the best day of my life")  print("**\n**------------------------------------")    So it’s the last my activity on this project …    *References:*  <https://medium.com/@sabber/classifying-yelp-review-comments-using-cnn-lstm-and-visualize-word-embeddings-part-2-ca137a42a97d>  <https://medium.com/@sabber/classifying-yelp-review-comments-using-lstm-and-word-embeddings-part-1-eb2275e4066b#:~:text=Build%20a%20neural%20network%20with%20LSTM,-In%20the%20following&text=The%20network%20starts%20with%20an,word%20in%20a%20meaningful%20way.>  <https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>  <https://towardsdatascience.com/light-on-math-ml-intuitive-guide-to-understanding-glove-embeddings-b13b4f19c010>  <https://github.com/krishnaik06/Natural-Language-Processing/blob/master/Toeknization.py> | | |