|  |  |
| --- | --- |
| 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 |
| 28 August 2020 | 9 | 6+ |
| Activities done during the day: On Day 9 I am trying to implement the code for the project.  The code made up of the following parts Emotion detection from text  * **Defining the Network using Pretrained Embedding Layer using GloVe Word Embeddings** * **Creating the Glove Embedding Layer** * **Training the model** * **Testing the Model Accuracy** * **Testing the model with any sentence**   For this project, we are going to implement an NLP task of creating a model to detect the emotion from text. We will develop this using the PyTorch library.  We will create an emotion detection for the following 5 emotions:  **Emotions Labels**  **Loving 0**  **Playful 1**  **Happy 2**  **Annoyed 3**  **Foodie 4**      **Dataset**  We will work with a dataset (X, Y) where we have:   * X contains 132 sentences * Y contains a label between [0, 4] corresponding to the five emotions.     **The Model**  We will build an LSTM model that takes as input word sequences that will take word ordering into account. We will use 50-dimensional [GloVe](https://nlp.stanford.edu/projects/glove/) pre-trained word embeddings to represent words. We will then feed those as an input into an LSTM that will predict the most appropiate emotion for the text.  **Pretrained Word Embeddings** are the **embeddings** learned in one task that are used for solving another similar task. These **embeddings** are **trained** on large datasets, saved, and then used for solving other tasks. That's why **pretrained word embeddings** are a form of Transfer Learning.  GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.  There are two popular word-level pretrained word embeddings:   * Google’s Word2Vec * Stanford’s GloVe     **Defining the Network using Pretrained Embedding Layer using GloVe Word Embeddings**  #Code  **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 model *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)* 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*  **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**------------------------------------")  **References:**  <https://www.youtube.com/results?search_query=LSTm>  <https://github.com/krishnaik06/Word-Embedding/blob/master/Untitled2.ipynb>  <https://www.analyticsvidhya.com/blog/2020/03/pretrained-word-embeddings-nlp/> | | |