Sentiment Analysis for Yelp Reviews

Group Report

Anuradha Tidke
Nitin Godi
Rakshith Eleti

Natural Language Processing
Department of Data Science
George Washington University

Table of Content

1. Introduction

The goal of this project is to estimate the Yelp ratings polarity (0 or 1) of a local business review based on the review text. This was done using NLP models with an emphasis on sentiment analysis. Yelp is a review-based website where people can discover and exchange information about local businesses. When consumers conduct a rapid search for businesses, they are more inclined to assess quality only on the star rating without reading the review language. As a result, our group is looking for probable connections between review wording and the corresponding rating.

In this project we will be exploring NLP algorithms like Recurrent Neural Networks (LSTM) and comparing the results with pre-trained models like DistilBERT and ELECTRA.

We will be using PyTorch packages such as Torch, Variable, get_tokenizer, nn, etc. to implement the model. PyTorch is an open source framework and is capable of using GPUs effectively and efficiently. It is very similar to Numpy hence it is easier to comprehend.

2. Dataset Description

We have taken our dataset from Kaggle. The Yelp reviews polarity dataset is constructed using the original dataset from the Yelp Dataset Challenge 2015. The ratings from this dataset are modified by considering stars 1 and 2 negative (label: 0), and 3 and 4 positive (label: 1). Thus this is a binary classification data. It contains over 560,000 training sample points and 38,000

testing sample points. The train and test sample points are distributed equally for both labels. The feature column is that of the review text and the target column is the label.

3. Data Preprocessing

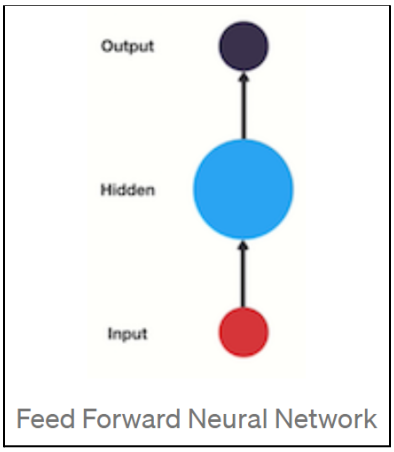
As the dataset is downloaded from Kaggle, we used Kaggle API to do the same. The instructions to download the dataset from Kaggle are outlined on the Readme file. Once the dataset was downloaded, we performed the data cleaning steps to remove the unwanted words and simplify the text. Firstly, we removed the stopwords, followed by punctuations and lemmatizing the data. This helped us reduce the number of unique tokens in the dataset, thus reducing the computational load.

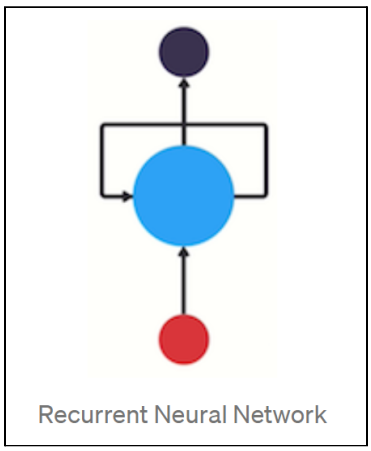
```
nltk.download('stopwords')
nltk.download('wordnet')
stopwords = nltk.corpus.stopwords.words('english')

def cleanData(list_of_strings):
    cleaned_list = []
    for string in list_of_strings:
        tokens = nltk.word_tokenize(string)
        """ remove stopwords """
        tokens = [token.lower() for token in tokens if token.lower() not in stopwords]

        """ remove punctuation"""
        tokens = [token for token in tokens if token.isalnum()]

        """ lemmatize """
        wnl = nltk.WordNetLemmatizer()
        tokens = [wnl.lemmatize(token) for token in tokens]

        cleanstring = ' '.join(tokens)
        cleaned_list.append(cleanstring)
    return cleaned_list
```

Cleaning the data resulted in some sequences (reviews) reducing to zero length. Running the LSTM code with padding enabled was giving errors for zero length sequences. As the percentage of such sequences was minimal, we could drop them. The train dataset was split into train and validation dataset with a split ratio of 0.9. Once the cleaned data was saved in different csv files, we used it to run our LSTM models.

4. Architecture of Recurrent Neural Networks

Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. Let's look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer. We get a recurrent neural network if we add a loop in the neural network that can pass prior information forward. An RNN has a looping mechanism that acts as a connection to allow information to flow from one step to the next. This information is the hidden state, which is a representation of previous inputs.





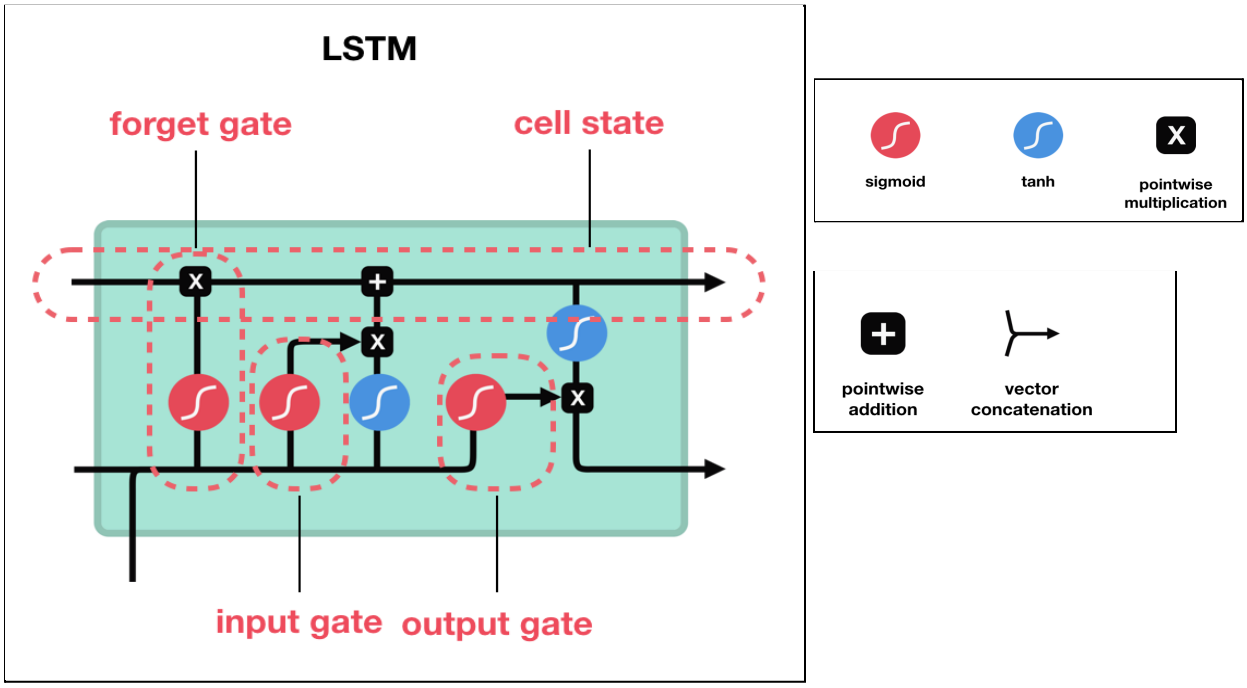
Training a neural network has three major steps. First, it does a forward pass and makes a prediction. Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.

As the RNN processes more steps, it has trouble retaining information from previous steps. This happens because of the vanishing gradient problem. Gradients are values used to update a neural network's weights. The bigger the gradient, the bigger the adjustments and vice

versa. When doing back propagation, each node in a layer calculates its gradient with respect to the effects of the gradient in the layer before it. So if the adjustments to the layers before it are small, adjustments to the current layer will be even smaller. That causes gradients to exponentially shrink as it back-propagates. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that's the vanishing gradient problem.

LSTM 's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

5. Working of Long Short Term Memory (LSTM)



There are three gates which regulate the flow of information from past cells to the next. The forget gate decides what is relevant to keep from prior steps. The input gate decides what

information is relevant to add from the current step. The output gate determines what the next hidden state should be.

6. LSTM Model Code Structure

Step 1: Define LSTM class

Our first step was to define the LSTM class. The embedding and all the model layers were initialized. In order to pad the sequences, we created an array of lengths of all the sequences. Maximum sequence length was used as an argument to pad zeroes after all the sequences.

Step 2: Data preparation

The cleaned data was imported into pandas dataframes. The feature column from train, validation and test dataset were converted to lists, the label columns were converted to tensors and moved to the GPU.

```
""" Using cleaned data """
train_data = pd.read_csv('train_cleaned.csv')
val_data = pd.read_csv('val_cleaned.csv')
test_data = pd.read_csv('test_cleaned.csv')

x_train_raw, y_train = train_data["cleaned_text"].values, torch.LongTensor(train_data["Label"].values).to(device)

x_val_raw, y_val = val_data["cleaned_text"].values, torch.LongTensor(val_data["Label"].values).to(device)

x_dev_raw, y_dev = test_data["cleaned_text"].values, torch.LongTensor(test_data["Label"].values).to(device)
```

The vocab dictionary including unique tokens from all the above datasets was created and maximum sequence length was calculated using the following function:

```
def extract_vocab_dict_and_msl(sentences_train, sentences_dev, sentences_val):
    """ Tokenizes all the sentences and gets a dictionary of unique tokens and also the maximum sequence length """
    tokens, ms_len = [], 0
    for sentence in list(sentences_train) + list(sentences_dev) + list(sentences_val):
        tokens_in_sentence = nltk.word_tokenize(sentence)
        if ms_len < len(tokens_in_sentence):
            ms_len = len(tokens_in_sentence)
        tokens += tokens_in_sentence
    token_vocab = {key: i for key, i in zip(set(tokens), range(1, len(set(tokens))+1))}
    return token_vocab, ms_len
```

Embedding dictionary was created from the vocab dictionary using the glove embeddings:

```
def get_glove_embeddings(vocab_dict):
    with open("glove.6B.50d.txt", "r") as s:
        glove = s.read()
    embeddings_dict = {}
    for line in glove.split("\n")[:-1]:
        text = line.split()
        if text[0] in vocab_dict:
            embeddings_dict[vocab_dict[text[0]]] = torch.from_numpy(np.array(text[1:], dtype="float32"))
    return embeddings_dict
```

Feature variable lists from the train, validation and test datasets were converted to a matrix of token ids using the following function. The number of columns of the matrix was taken as the maximum sequence length we obtained before.

```
def convert_to_ids(raw_sentences, vocab_dict, pad_to):
    """ Takes an NumPy array of raw text sentences and converts to a sequence of token ids """
    x = np.empty((len(raw_sentences), pad_to))
    for idx, sentence in enumerate(raw_sentences):
        word_ids = []
        for token in nltk.word_tokenize(sentence):
            try:
                word_ids.append(vocab_dict[token])
            except:
                word_ids.append(vocab_dict[token])
        if pad_to < len(word_ids):
            x[idx] = word_ids[:pad_to]
        else:
            x[idx] = word_ids + [0] * (pad_to - len(word_ids))
    return x
```

Following the above step, the features lists were converted to tensors and moved to the GPU.

Step 3: Training the model

A lookup table was created using the vocab dictionary and the embedding dictionary. It contained the embedding vectors of all the tokens in the vocab dictionary arranged row wise. The table was fed to our LSTM model as the embedding weights. We used ADAM optimizer and cross entropy loss as the loss criteria for our model.

```
def get_glove_table(vocab_dict, glove_dict):
    lookup_table = torch.empty((len(vocab_dict)+2, 50))
    for token_id in sorted(vocab_dict.values()):
        if token_id in glove_dict:
            lookup_table[token_id] = glove_dict[token_id]
        else:
            lookup_table[token_id] = torch.ones((1, 50))
    lookup_table[0] = torch.zeros((1, 50))
    return lookup_table
```

We ran 10 epochs to train this model. The learning rate was taken as 1e-3 and the batch size was 16. For each batch, the optimizer was initialized to zero to avoid gradient explosion. The predictions and loss were calculated for the data points of that batch. The loss was back propagated to calculate the gradient of previous batches. The ADAM optimizer used the gradients to update the weights. At the end of each epoch, the model was applied to the validation dataset and saved if the validation accuracy increased.

```
for epoch in range(args.n_epochs):

    loss_train, train_steps = 0, 0
    model.train()
    total = len(x_train) // args.batch_size #(total number of batches - 1)
    with tqdm(total=total, desc="Epoch {}".format(epoch)) as pbar:
        for batch in range(len(x_train)//args.batch_size + 1):
            inds = slice(batch*args.batch_size, (batch+1)*args.batch_size)
            optimizer.zero_grad()
            logits = model(x_train[inds])
            loss = criterion(logits, y_train[inds])
            loss.backward()
            optimizer.step()
            loss_train += loss.item()
            train_steps += 1
            pbar.update(1)
            pbar.set_postfix_str("Training Loss: {:.5f}".format(loss_train / train_steps))
```

.

Step 4: LSTM results

Version 1

Full train dataset was used to train the model

- train_data length: 503,924

Results:

- **Best val_acc: 92.22 %**
- Time req for all epochs = 2 hrs 45 min
- val_acc decreased after epoch 1 which indicates that the model was overfitting the train data
- **test_acc using the above best model: 93.63 %**

```
Starting training loop...
Epoch 0: 15748it [16:30, 15.90it/s, Training Loss: 0.24869]
Epoch 0 | Train Loss 0.24869, Train Acc 93.18, Val Acc 92.02
The model has been saved!
Epoch 1: 15748it [16:27, 15.95it/s, Training Loss: 0.18541]
Epoch 1 | Train Loss 0.18541, Train Acc 94.56, Val Acc 92.22
The model has been saved!
Epoch 2: 15748it [16:27, 15.95it/s, Training Loss: 0.15792]
Epoch 2 | Train Loss 0.15792, Train Acc 95.30, Val Acc 92.02
Epoch 3: 15748it [16:26, 15.96it/s, Training Loss: 0.13661]
Epoch 3 | Train Loss 0.13661, Train Acc 95.28, Val Acc 91.36
Epoch 4: 15748it [16:28, 15.94it/s, Training Loss: 0.11903]
Epoch 4 | Train Loss 0.11903, Train Acc 96.37, Val Acc 91.93
Epoch 5: 15748it [16:26, 15.96it/s, Training Loss: 0.10402]
Epoch 5 | Train Loss 0.10402, Train Acc 97.04, Val Acc 91.87
Epoch 6: 15748it [16:32, 15.87it/s, Training Loss: 0.09042]
Epoch 6 | Train Loss 0.09042, Train Acc 97.10, Val Acc 91.57
Epoch 7: 15748it [16:31, 15.88it/s, Training Loss: 0.07957]
Epoch 7 | Train Loss 0.07957, Train Acc 97.33, Val Acc 91.13
Epoch 8: 15748it [16:30, 15.90it/s, Training Loss: 0.06997]
Epoch 8 | Train Loss 0.06997, Train Acc 97.27, Val Acc 90.81
Epoch 9: 15748it [16:27, 15.95it/s, Training Loss: 0.06335]
Epoch 9 | Train Loss 0.06335, Train Acc 97.82, Val Acc 90.91
The accuracy on the test set is 93.63
The confusion matrix is
[[17720  1278]
 [ 1141 17858]]
```

Version 2

The train dataset was shrunk to 25% of what it was before and used to train the model.

- train_data length: 125,982

Results:

- **Best val_acc: 90.67 %**
- Time req for all epochs > 40 min
- Solution is to reduce learning rate after epoch 1
- **test_acc using the above best model: 91.98%**

```
Starting training loop...
Epoch 0: 3937it [03:56, 16.65it/s, Training Loss: 0.31155]
Epoch 0 | Train Loss 0.31155, Train Acc 91.65, Val Acc 90.19
The model has been saved!
Epoch 1: 3937it [03:54, 16.80it/s, Training Loss: 0.21484]
Epoch 1 | Train Loss 0.21484, Train Acc 93.98, Val Acc 90.67
The model has been saved!
Epoch 2: 3937it [03:54, 16.81it/s, Training Loss: 0.17009]
Epoch 2 | Train Loss 0.17009, Train Acc 95.23, Val Acc 90.29
Epoch 3: 3937it [03:55, 16.74it/s, Training Loss: 0.13860]
Epoch 3 | Train Loss 0.13860, Train Acc 96.05, Val Acc 89.97
Epoch 4: 3937it [03:54, 16.80it/s, Training Loss: 0.11315]
Epoch 4 | Train Loss 0.11315, Train Acc 96.87, Val Acc 89.61
Epoch 5: 3937it [03:53, 16.83it/s, Training Loss: 0.09169]
Epoch 5 | Train Loss 0.09169, Train Acc 97.54, Val Acc 89.56
Epoch 6: 3937it [03:53, 16.83it/s, Training Loss: 0.07487]
Epoch 6 | Train Loss 0.07487, Train Acc 97.53, Val Acc 88.93
Epoch 7: 3937it [03:54, 16.81it/s, Training Loss: 0.06261]
Epoch 7 | Train Loss 0.06261, Train Acc 97.90, Val Acc 88.87
Epoch 8: 3937it [03:54, 16.81it/s, Training Loss: 0.05364]
Epoch 8 | Train Loss 0.05364, Train Acc 98.45, Val Acc 89.06
Epoch 9: 3937it [03:54, 16.81it/s, Training Loss: 0.04750]
Epoch 9 | Train Loss 0.04750, Train Acc 97.87, Val Acc 88.48
The accuracy on the test set is 91.98
The confusion matrix is
[[17466  1532]
 [ 1514 17485]]
```

- Fortunately, shrinking the training data didn't affect our test accuracy by a lot (decreased by 1.7%). So we decided to use this size for the train data for our pretrained models

7. Transformers

In natural language processing, the Transformer is a unique design that seeks to handle sequence-to-sequence problems while also resolving long-range relationships. In the paper Attention Is All You Need, the Transformer was proposed. It uses the attention-mechanismTransformer, like LSTM, it is an architecture for changing one sequence into another using two components (Encoder and Decoder).

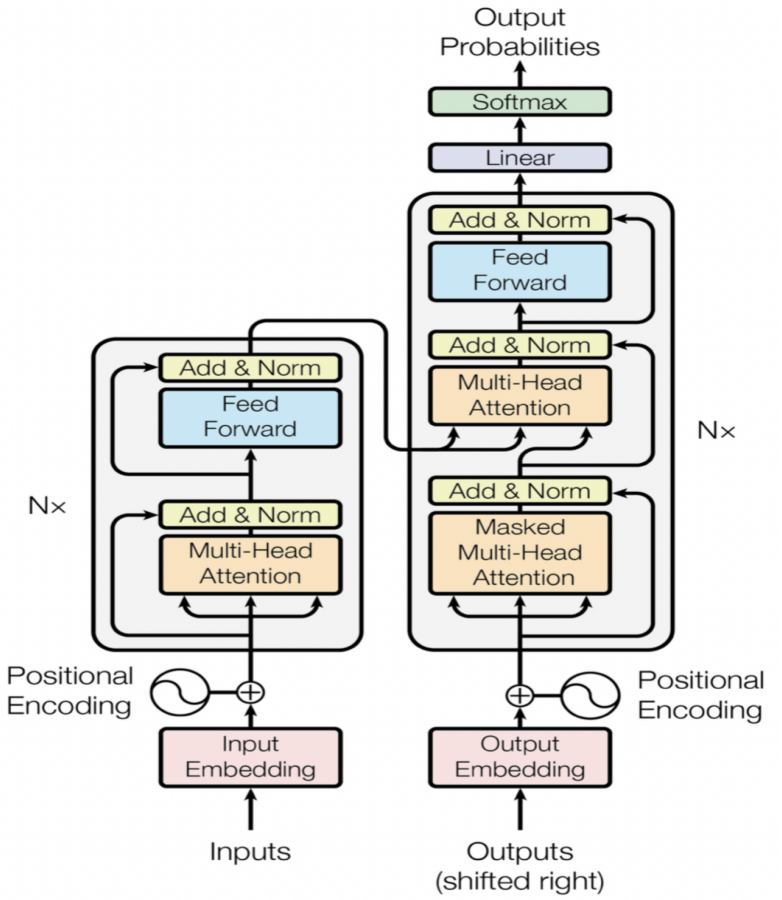
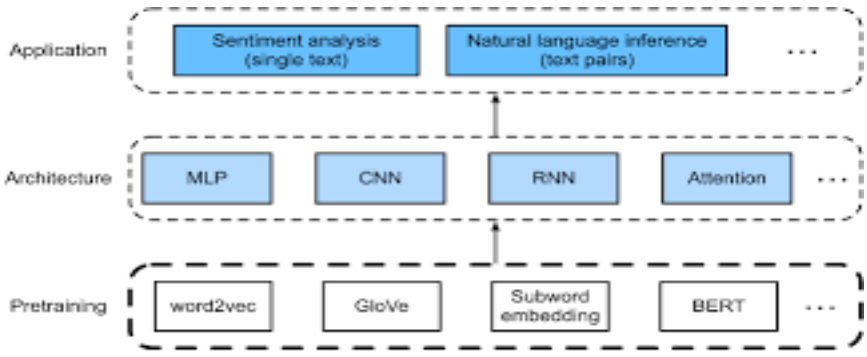


Figure 1: The Transformer - model architecture.

Pre-Trained Models

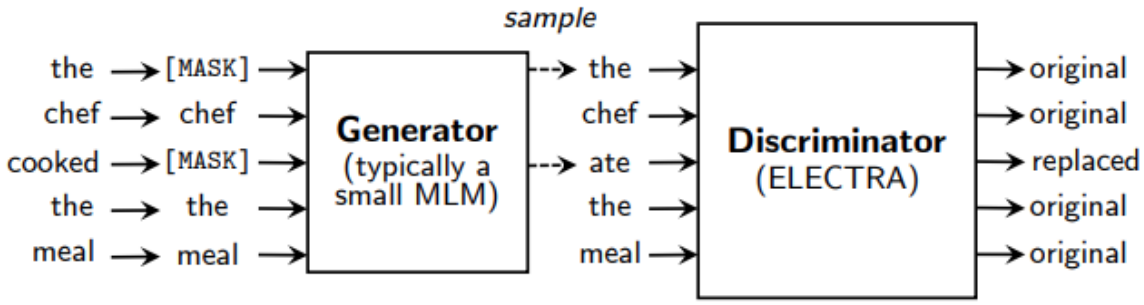
Deep learning models (such as transformers) that have been trained on a large dataset to accomplish certain NLP tasks are known as pre-trained models (PTMs) for NLP. When PTMs are trained on a large corpus, they can acquire universal language representations, which can help with downstream NLP tasks and prevent having to train a new model from the start. Pre-trained models may thus be referred to as reusable NLP models, which NLP developers can employ to quickly construct an NLP application. NLP assignments are accessible for free and do not require any prior understanding of NLP. The pioneering generation To learn excellent word embeddings, pre-trained models were used. The most recent or second generation PTM, on the other hand, is programmed to learn contextual word embeddings. Pre-trained models may be quickly loaded into NLP libraries like PyTorch, Tensorflow, and others, and utilized to execute NLP tasks with little effort on the part of NLP developers. Pre-trained models are increasingly being employed on NLP jobs since they are simpler to install, have higher accuracy, and need less training time than custom-built models. Some real-world examples where pre-trained models are used:

1. **Named Entity Recognition (NER)**
2. **Sentiment Analysis**
3. **Machine Translation**



ELECTRA

ELECTRA is a project that aims to train text encoders as discriminators rather than generators (Efficiently Learning an Encoder that Classifies Token Replacements Accurately). ELECTRA is a novel pre-training approach that entails the training of two transformer models: the generator and the discriminator. Because its function is to substitute tokens in a sequence, the generator is trained as a masked language model. The discriminator, which is the model we're interested in, tries to figure out which tokens were substituted by the generator in the sequence. In the ELECTRA checkpoints saved using Google Research's implementation, both the generator and discriminator are present. In the conversion script, the user must provide the model to export into the appropriate architecture. This means that the discriminator may be loaded in the ElectraForMaskedLM model, while the generator can be loaded in the ElectraForPreTraining model (the classification head in the generator will be set randomly since it doesn't exist in the discriminator).



The tokens are filled in a sequence using the generator, and the filled tokens are THE and COOKED, whereas the Discriminator informs us which tokens are filled represented by replaced, or it gives the not replaced tokens as original, as shown in the diagram.

DistilBERT

Operating huge models in on-the-edge and/or under tight computational training or inference budgets remains hard as Transfer Learning from large-scale pre-trained models becomes increasingly prominent in Natural Language Processing (NLP). DistilBERT, a method for pre-training a smaller general-purpose language representation model that can subsequently be fine-tuned to perform well on a variety of tasks, like it's bigger equivalents. While most previous research focused on using knowledge distillation to build task-specific models, by using knowledge distillation during the pre-training phase, we can reduce the size of a BERT model by 40% while retaining 97% of its language understanding capabilities and being 60% faster. Also, offers a triple loss that combines language modeling, distillation, and cosine-distance losses to utilize the inductive biases learnt by bigger models during pre-training.

8. Transformers Code Structure

The code structure for the two transformer models is the same with just the changes to the names of the functions.

Step 1: Data Preprocessing

Preprocessing the training data is the first stage. 140,000 data points were utilized as training data for the transformers. The training data was divided into two sets, one for training and one for validation, at a 4:1 ratio. Depending on the model employed for the experiment, both of these sets were then sent via DistilBertTokenizerFast or ElectraTokenizerFast. Then, for both

the training and validation sets, data loaders were developed. The code for the data preparation phase is shown in the figure below.

```
train_df = pd.read_csv('train.csv')[:140000]
train_df.columns = ['label','text']
train_df['label'] = train_df['label'] - 1
train_texts = train_df['text'].values.tolist()
train_labels = train_df['label'].values.tolist()

train_texts, val_texts, train_labels, val_labels = train_test_split(train_texts, train_labels, test_size=.2)

# ================Tokenizing and creating data loaders====================
tokenizer = DistilBertTokenizerFast.from_pretrained(checkpoint2)
train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=max_len)
val_encodings = tokenizer(val_texts, truncation=True, padding=True, max_length=max_len)

class YelpDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = YelpDataset(train_encodings, train_labels)
val_dataset = YelpDataset(val_encodings, val_labels)

train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=6)
eval_dataloader = DataLoader(val_dataset, batch_size=5)
```

Step 2: Initializing

Next step is to initialize the model being used for the experiment. The image below shows the lines of code for initializing the model.

```
model = DistilBertForSequenceClassification.from_pretrained(checkpoint1)
```

Step 3: Training

In this step AdamW was set as the optimizer along with learning rate scheduler. Next, using the training data loader, each batch of data was fed to the model and the outputs were used

to calculate loss and update the weights and biases of the model. Finally, the performance of the model was calculated on the validation set and the model was saved. The following image shows the script for training the transformers with respect to our dataset.

```
# ======================Model Training=====================
model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        labels = batch['labels'].to(device)
        if (model_name == model1) | (model_name == model2):
            outputs = model(input_ids, labels=labels)
            loss = outputs[0]
        else:
            outputs = model(input_ids)
            loss = criterian(outputs, labels)
        loss.backward()
        optim.step()
        lr_scheduler.step()
        progress_bar.update(1)
```

Ensemble Code for DistilBERT and ELECTRA

- The Training dataset was individually fed to the DistilBert and Electra models.
- These two models' outputs were input into a linear layer that works as a classifier.
- The ensemble's structure is shown in the code excerpt below.

```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.m1 = DistilBertForSequenceClassification.from_pretrained(checkpoint1)
        self.m2 = ElectraForSequenceClassification.from_pretrained(checkpoint2)
        self.dropout = nn.Dropout(0.3)
        self.out3 = nn.Linear(4,2)
    def forward(self, ids):
        output_1 = self.m1(ids, return_dict=False)
        output_2 = self.dropout(output_1[0])
        output_3 = self.m2(ids, return_dict=False)
        output_4 = self.dropout(output_3[0])
        output_5 = torch.cat((output_2, output_4), dim=1)
        output = self.out3(output_5)
        return output
```

- The training data length was 112,000 , the validation data length was 28,000, and the testing data was 37,997.
- In terms of parameters, the Learning rate is 5e-5, the Batch size is 8, and the Epoch size is one.
- The best validation accuracy was 94.03 percent, while the best testing data correctness was 94.63 percent.

9. Observations and Conclusion

The pretrained models have given higher accuracy on the test dataset than the LSTM model, even by using lesser training data, which was expected. It is worth noticing that we have used the raw data without any cleaning for our pretrained models. This is because they utilise the structure of the sentence from both directions to connect every output element to every input element. So, removing stopwords and punctuations might confuse the model and reduce the accuracy.

Model	Number of train data points	Accuracy
LSTM	125,982	91.98%
DistilBERT	112,000	96.35%
ELECTRA	112,000	96.55%
Ensemble*	112,000	94.63%

* The ensemble model was trained using only 1 epoch.

10. References

1. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
2. https://huggingface.co/docs/transformers/index
3. https://www.kdnuggets.com/2019/09/bert-roberta-distilbert-xlnet-one-use.html
4. http://ankit-ai.blogspot.com/2021/02/understanding-state-of-art-language.html
5. https://github.com/amir-jafari/Deep-Learning/blob/master/Pytorch/RNN/2_TextClassification/example_LSTM_sentiment_analysis.py