Singularity Systems Project 1

Akhil Devarashetti

August 2, 2020

Abstract

This project deals with training a deep neural network to classify text into 20 categories. The dataset for this classification is given - Twenty Newsgroups. I tried various kinds of neural network architectures and studied how they perform.

1 System Description

I divided the task of classification into 3 modules. They are described as now. The final code consists of many classes and functions as opposed to Jupyter notebooks for code-reusability, however most of the experimentation is done in notebooks.

1.1 Preprocessing

The given dataset is in a directory-based labelling fashion. The first task is to convert these strings into numerical sequences of fixed length. This is what the first module does - preprocess the whole dataset and store it as sequences and one-hot vectors in 2 csv files, one for training and one for testing.

I decided to try many different models on this dataset and find the best model. I could create a tokenizer from scratch or use a predefined tokenizer. For pre-trained models like word2vec[3] and BERT[1], I will need to use their tokenizer, and for my own models, it makes sense to define my own tokenizer. So, I preprocessed using both tokenizers resulting in 4 files. 2 files predefined tokenizer and 2 files for custom tokenizer (2 files for training and testing). The sequence size for my custom tokenizer is 150 (decided after examining length of a few files) and the sequence size for BERT is 512. Any word after 150 words is truncated and for texts that are less than 150 words, a sequence of 0s is used as padding. The tokenizer indexes the words seen in the training set only. The same tokenizer is used to convert test sentences into sequences.

Each file is stored as a comma separated list of integers in a line within the output file. The labels are appened for each line in the CSV as a one-hot vector.

I used Keras' [2] text library to (1.) flow the texts from directories to labelled batches of texts and to (2.) filter, change to lower case and tokenize the words [4]. I also used BERT's tokenizer in PyTorch using the libraries "pytorch pretrained bert" "pytorch-nlp" [5].

1.2 Training

Since I created the csv file with data in the desired format, the next task is to develop a neural network which can take these sequences in batches, produce batches of probability distributions for the classes and train based on the cross-entropy error.

I experimented with quite a few types of models with layers of Embedding, Linear, LSTM, Convolutional and Pooling. Each of the experimentation is discussed in the next section. I used Google Colab to train the networks on a GPU. I uploaded the csv files to my google cloud so that it's easy to download on the colab environment. As I am familiar with both Keras and PyTorch, I used the best of both worlds to create these models - Keras for quick high-level experimentation of model architecture and preprocessing, PyTorch for easy experimentation on BERT like training only the last two layers.

I did not do much of hyper-parameter tuning due to the large number of models to test in the given time. The learning is taken care of ADAM and all the activations are ReLU except for the last layer.

1.3 Prediction

The last task is to unify the preprocessing and the trained model into a program that can predict the labels for any given texts on the fly.

2 Methods

I experimented with various kinds of model architectures, 11 of them are discussed below, others are just minor variations. The evaluation of these models is based on test loss and test accuracy.

Models	Global Avg	Base + deeper	Base + BERT tok	LSTM	Bidirectional LSTM	Conv1D + MaxPool	Conv * 3	BERT + last only	BERT
Test accuracy	77.3	66.14	75.16	73.12	50.64	75.79	70.57	14.48	82.69
test loss	1.2234	1.99	1.055	2.754	21.86	2.595	1.813	0.183	0.049
test f1 macro	74.76		75.45	74.71		74.19			0.35
train loss	0.0113	0.034	0.026	0.002	0.4291	0.028	0.0749	0.186	0.024
train accuracy	99.83	99.38	99.73	99.6		99.15	98.29	12.67	92.98

Figure 2.1: Performance of all models - Table.

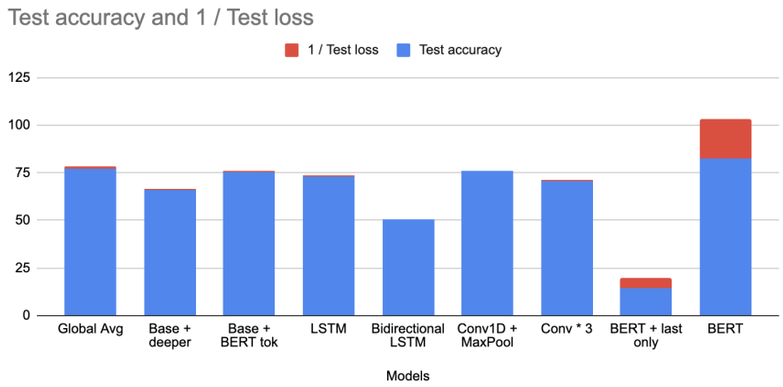


Figure 2.2: Performance of all models - Chart.

2.1 Linear Models

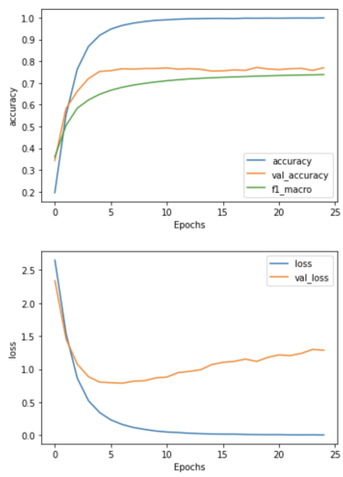
2.1.1 Embedding with Global Avg Pooling (base):

For text-based problems, I always use a model with embeddings global average pooling and a few linear layers as the base model. Luckily, for this task, the base model turned out to be the best model with an accuracy of 77.3%. The architecture of the model and the results are seen in the Figures 2.3a and 2.3b. The model seems to overfit slightly, so I added dropouts and saved the model by stopping early.

```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    GlobalAveragePooling1D(),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', m
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 150, 64)           640000
_________________________________________________________________
global_average_pooling1d (Gl (None, 64)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               8320
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 20)                2580
=================================================================
Total params: 650,900
Trainable params: 650,900
Non-trainable params: 0
```

(a) Base model architecture.



(b) Accuracy and loss of base model.

Figure 2.3: Base model.

2.1.2 Deeper model:

The next natural step for me was to add more layers. I replaced the Global Avg Pooling layer with a Flatten layer so that there's more learning opportunity. However, this model did not perform well on test data. The accuracy is 66.14%. It started overfitting too soon. You can see the architecture in Figure 2.4a and the performance in Figure 2.4b

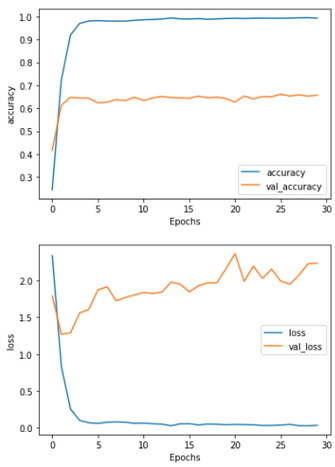
2.1.3 Base model with sequence length 512:

I wondered if the sequence length of 150 was too small. Then I preprocessed the whole dataset with sequence length of 512 and trained the base model on it. But the test accuracy obtained was just as high as the base model.

```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    #GlobalAveragePooling1D(),
    Flatten(),
    Dropout(0.3),
    Dense(512, activation='relu'),
    Dropout(0.3),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, 150, 128)          1280000
_________________________________________________________________
flatten_9 (Flatten)          (None, 19200)             0
_________________________________________________________________
dropout_29 (Dropout)         (None, 19200)             0
_________________________________________________________________
dense_26 (Dense)             (None, 512)               9830912
_________________________________________________________________
dropout_30 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_27 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_31 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_28 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_32 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_29 (Dense)             (None, 20)                2580
=================================================================
Total params: 11,277,716
Trainable params: 11,277,716
Non-trainable params: 0
_________________________________________________________________
```

(a) Deeper base model architecture.



(b) Accuracy and loss of deeper base model.

Figure 2.4: Deeper base model.

2.1.4 Base model with BERT's tokenizer:

Since I used 512 sequence length, I tried replacing the tokenizer with a predefined BERT tokenizer which is a WordPiece tokenizer. This gave results as good as the base model, but nothing better. The accuracy was 75.16%.

2.2 RNN Models

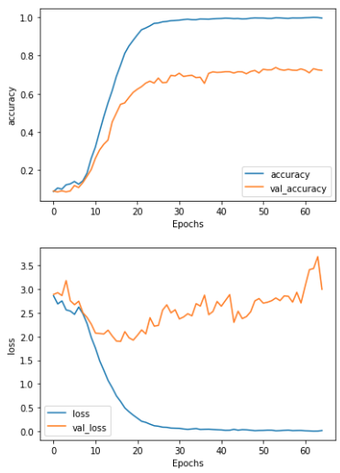
2.2.1 LSTM:

RNNs are generally the go-to models for sequence learning. So for my next model, I tried adding an LSTM and removed the Global Avg Pool layer. LSTM took extremely long to train and converge compared to the linear models. I tried training for 15 epochs and observed that the test loss was still going down. So I increased the epochs to 65. But sadly after 15 epochs or so, the test loss climbed up. The accuracy was 73.12%. You can see the architecture in Figure 2.5a and the performance in Figure 2.5b

```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    LSTM(128),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', me
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 150, 64)           640000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               98816
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 128)               16512
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 20)                2580
=================================================================
Total params: 757,908
Trainable params: 757,908
Non-trainable params: 0
_________________________________________________________________
```

(a) LSTM model architecture.



(b) Accuracy and loss of LSTM model.

Figure 2.5: LSTM model.

2.2.2 Bidirectional LSTM:

I tried adding a Bidirectional LSTM. It took much longer than a unidirectional LSTM. The accuracy seemed to flatten out at 50%, after very long 50 epochs, so I stopped the training. Moreover, the loss sometimes reaches as high as the order of 1×10^{14} before it lowers. The high spike in the Figure 2.6b shows this extreme loss. It turned out that the bidirectional LSTM is performing poorly. The accuracy was 50.64%. You can see the architecture in Figure 2.6a and the performance in Figure 2.6b

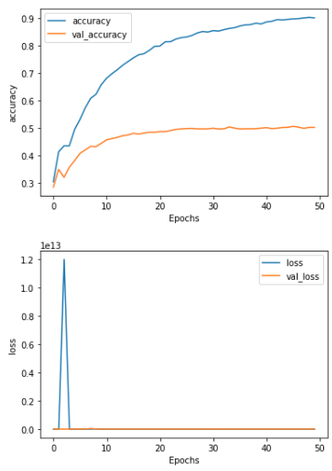
```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    #GlobalAveragePooling1D(),
    #LSTM(128, activation='relu'),
    Bidirectional(LSTM(64, activation='relu')),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', met
model.summary()
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doe
WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doe
WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doe
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 150, 64)	640000
bidirectional (Bidirectional	(None, 128)	66048
dropout (Dropout)	(None, 128)	0
dense (Dense)	(None, 128)	16512
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 20)	2580

```
Total params: 725,140
Trainable params: 725,140
Non-trainable params: 0
```

(a) Bidirectional LSTM model architecture.



(b) Accuracy and loss of Bidirectional LSTM model.

Figure 2.6: Bidirectional LSTM model.

2.2.3 2-Layer LSTM:

abcd I tried adding another LSTM on top of the LSTM in hopes that it might do better than the base model. This model was notoriously slow to train and progress. I halted the training as it wasn't getting any better.

2.3 Convolutional Models

My next attempt was to use convolutional layers to the model.

2.3.1 1-Layer Conv with Max Pooling:

I added 1 conv layer to test how it works. The test accuracy was on par with the base model at 75.79% although it overfits too much. You can see the architecture in Figure 2.7a and the performance in Figure 2.7b

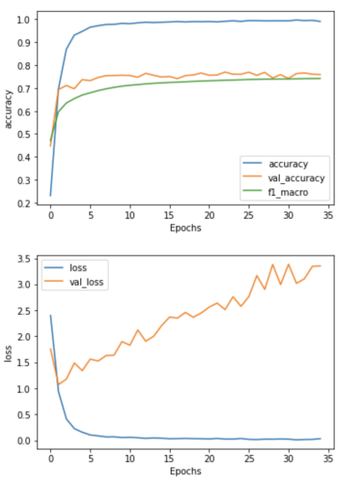
```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    #GlobalAveragePooling1D(),
    #LSTM(128, activation='relu'),
    #Bidirectional(LSTM(64, activation='relu')),
    Conv1D(128, 5, activation='relu'),
    MaxPooling1D(2),
    Flatten(),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', met
model.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 150, 64)	640000
conv1d_2 (Conv1D)	(None, 146, 128)	41088
max_pooling1d (MaxPooling1D)	(None, 73, 128)	0
flatten_1 (Flatten)	(None, 9344)	0
dropout_4 (Dropout)	(None, 9344)	0
dense_4 (Dense)	(None, 128)	1196160
dropout_5 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 20)	2580

```
Total params: 1,879,828
Trainable params: 1,879,828
Non-trainable params: 0
```

(a) Conv model architecture.



(b) Accuracy and loss of Conv model.

Figure 2.7: Conv model.

2.3.2 3-Layer Conv with Max Pooling:

Then I tried using more conv layers in a VGG-like network. This did not overfit as much as the single conv layer, but the accuracy on test data is not as good as the single conv layer at 70.57%. You can see the architecture in Figure 2.8a and the performance in Figure 2.8b

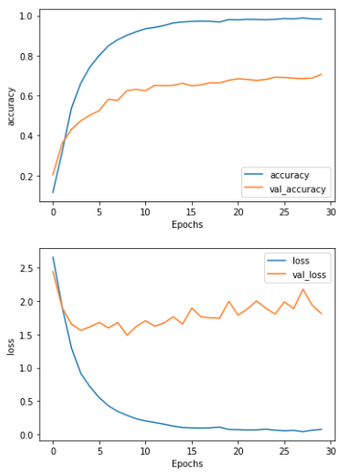
2.4 BERT Based Models

Transfer learning is one of the most useful techniques in building deep learning models. BERT [1] is one of the most famous transformer [6] models which is based on Attention Mechanism with no

```
model = tf.keras.Sequential([
    Embedding(vocab_size, embed_size, input_length=150),
    Conv1D(64, 5, activation='relu'),
    Dropout(0.3),
    MaxPooling1D(2),
    Conv1D(128, 5, activation='relu'),
    Dropout(0.3),
    MaxPooling1D(2),
    Conv1D(256, 5, activation='relu'),
    Dropout(0.3),
    MaxPooling1D(2),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(20, activation='softmax')
])
model.compile(loss='categorical_crossentropy', optimizer='adam', me
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_8 (Embedding)      (None, 150, 64)           640000
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 146, 64)           20544
_________________________________________________________________
dropout_17 (Dropout)         (None, 146, 64)           0
_________________________________________________________________
max_pooling1d_9 (MaxPooling1 (None, 73, 64)            0
_________________________________________________________________
conv1d_12 (Conv1D)           (None, 69, 128)           41088
_________________________________________________________________
dropout_18 (Dropout)         (None, 69, 128)           0
_________________________________________________________________
max_pooling1d_10 (MaxPooling (None, 34, 128)           0
_________________________________________________________________
conv1d_13 (Conv1D)           (None, 30, 256)           164096
_________________________________________________________________
dropout_19 (Dropout)         (None, 30, 256)           0
_________________________________________________________________
max_pooling1d_11 (MaxPooling (None, 15, 256)           0
_________________________________________________________________
flatten_7 (Flatten)          (None, 3840)              0
_________________________________________________________________
dense_16 (Dense)             (None, 128)               491648
_________________________________________________________________
dropout_20 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_17 (Dense)             (None, 20)                2580
=================================================================
Total params: 1,359,956
Trainable params: 1,359,956
Non-trainable params: 0
_________________________________________________________________
```

(a) 3-Layer Conv model architecture.



(b) Accuracy and loss of 3-Layer Conv model.

Figure 2.8: 3-Layer Conv model.

RNNs. BERT outputs a 786 dimensional vector for each input token. The first input token is '[cls]' and is always fixed. The output vector corresponding to this token is usually used to classify the given sequence. Each of these vectors is sort-of like a vector in the meaning space in the context of the input sentence.

2.4.1 Last layer only:

I initially thought that finetuning the entire bert model would be an intensive task. So I added a single output layer at the end of BERT and initialized the optimizer to train just parameters from the last layer. I saw no different in the training time of finetuning vs training the last layer only. But interestingly, I see absolutely no improvement in the accuracy of the model. So I halted the training.

2.4.2 Fine-tuning entire model:

Then I decided to fine-tune the entire BERT model. As expected, training time was very slow for each epoch. This model outperformed the base model at an accuracy of 82.7%. You can see the performance in Figure 2.9. We can see that the model does not overfit the data as much as the other models have.

3 Discussion

This was not an easy problem to solve. One reason might be that the dataset has some emails that are too short and not that indicative of what class it might be. Language based problems are notoriously difficult to train with shallow data. Although this dataset consists over 10,000 sequences, the meaning of words used in the emails may not be properly embedded into the network. So, it is easy to overfit the data. This might be one of the reasons why complicated models like LSTMs and Conv nets failed to perform better than a simple model.

BERT has seen a lot of data. It has seen words in many contexts and learned to embed words differently with different contexts. This gave the network a better idea of what the words mean in this dataset. This might be the reason why I saw little to no overfitting on the data.

4 Further Improvements

Since the revolution of Transformer networks [6], Attention Mechanism has become much more popular than before. I would love to implement an Attention layer in the model.

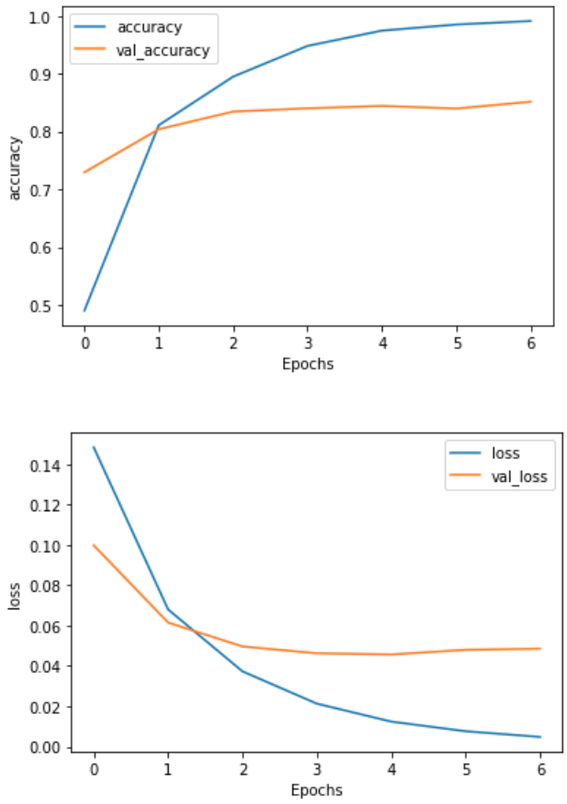


Figure 2.9: Accuracy and loss of BERT model.

Another interesting idea to explore would be to use ensemble prediction. A committee-of-experts, boosting or bagging kind-of system with redundancy and degeneracy would be a perfect system to solve this problem.

Another interesting idea I would love to explore is to self-supervise the language model like word2vec [3] by masking a word and predicting the surrounding words, or using the surrounding words to predict the masked word. This would yield in better word-embeddings.

References

[1] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. arXiv: 1810.04805 [cs.CL].

[2] Google. *tf.keras.preprocessing.text.Tokenizer*. 2020. URL: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer (visited on 08/01/2020).

[3] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[4] Laurence Moroney. *Tensorflow In Practice*. 2020. URL: https://github.com/lmoroney/dlaicourse/blob/master/TensorFlow%5C%20In%5C%20Practice/Course%5C%203%5C%20-%5C%20NLP/Course%5C%203%5C%20-%5C%20Week%5C%202%5C%20-%5C%20Lesson%5C%202.ipynb (visited on 08/01/2020).

[5] Dima Shulga. *BERT to the rescue!* 2020. URL: https://github.com/shudima/notebooks/blob/master/BERT_to_the_rescue.ipynb (visited on 08/01/2020).

[6] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].