Multi Class Text Classification with LSTM using TensorFlow 2.0

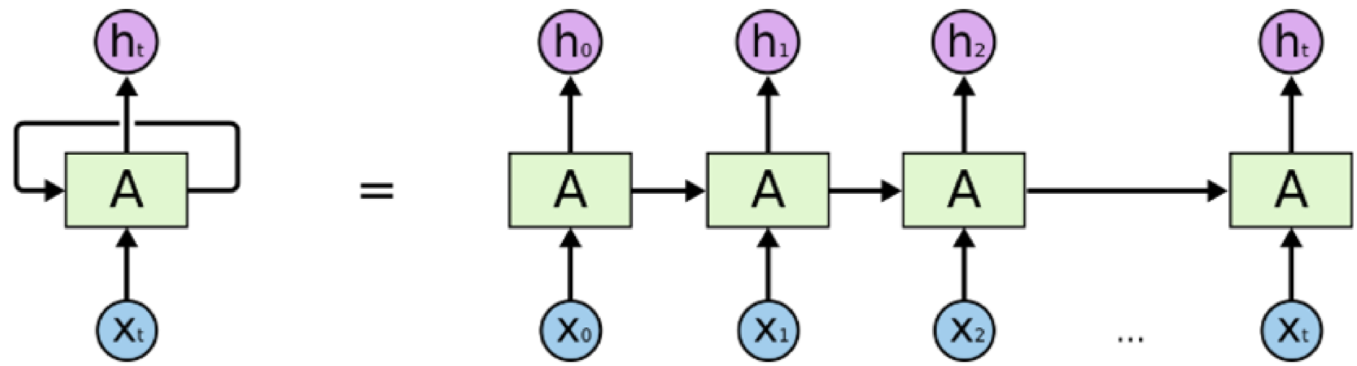
Recurrent Neural Networks, Long Short Term Memory

Susan Li Follow

Dec 8, 2019 · 7 min read

A lot of innovations on NLP have been how to add context into word vectors. One of the common ways of doing it is using Recurrent Neural Networks. The following are the concepts of Recurrent Neural Networks:

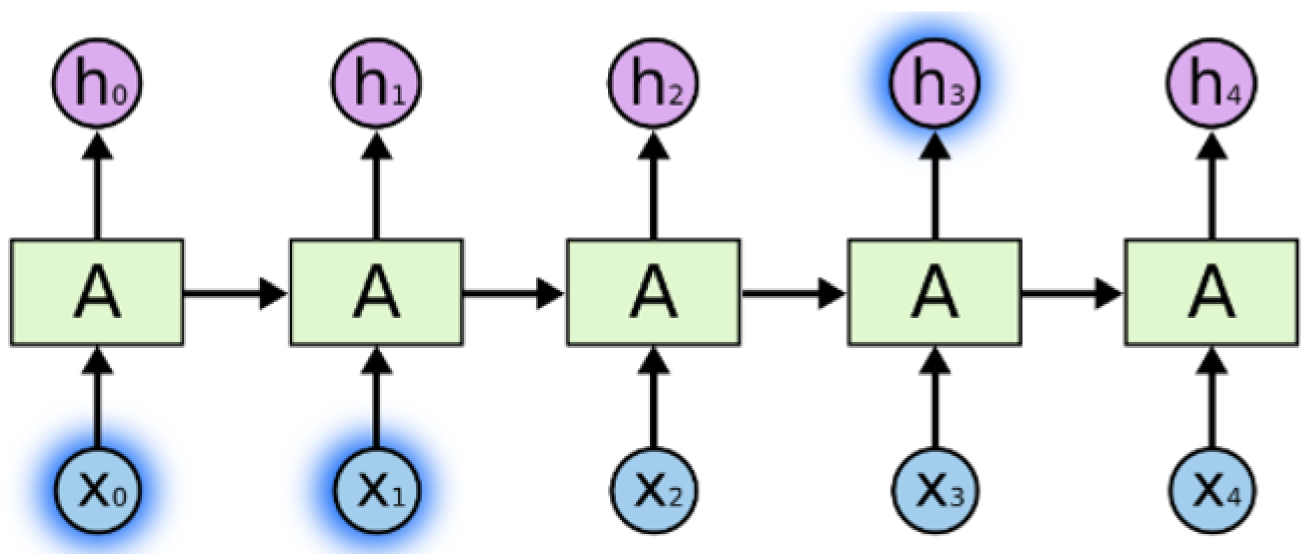
- They make use of sequential information.
- They have a memory that captures what have been calculated so far, i.e. what I spoke last will impact what I will speak next.
- RNNs are ideal for text and speech analysis.
- The most commonly used RNNs are LSTMs.



Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

The above is the architecture of Recurrent Neural Networks.

- "A" is one layer of feed-forward neural network.
- If we only look at the right side, it does recurrently to pass through the element of each sequence.
- If we unwrap the left, it will exactly look like the right.

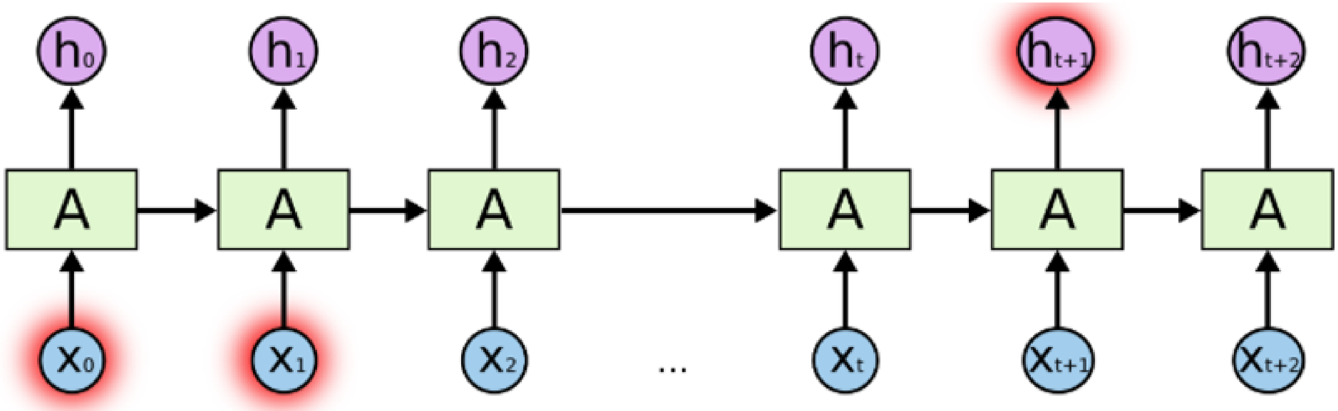


Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs

Assuming we are solving document classification problem for a news article data set.

- We input each word, words relate to each other in some ways.
- We make predictions at the end of the article when we see all the words in that article.

- RNNs, by passing input from last output, are able to retain information, and able to leverage all information at the end to make predictions.



https://colah.github.io/posts/2015-08-Understanding-LSTMs

- This works well for short sentences, when we deal with a long article, there will be a long term dependency problem.

Therefore, we generally do not use vanilla RNNs, and we use Long Short Term Memory instead. LSTM is a type of RNNs that can solve this long term dependency problem.

In our document classification for news article example, we have this many-to- one relationship. The input are sequences of words, output is one single class or label.

Now we are going to solve a BBC news document classification problem with LSTM using TensorFlow 2.0 & Keras. The data set can be found here.

- First, we import the libraries and make sure our TensorFlow is the right version.

```
import csv
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from nltk.corpus import stopwords
STOPWORDS = set(stopwords.words('english'))

print(tf.__version__)
```

import.py hosted with ❤ by **GitHub** **view raw**

```
2.0.0
```

- Put the hyperparameters at the top like this to make it easier to change and edit.
- We will explain how each hyperparameter works when we get there.

```
vocab_size = 5000
embedding_dim = 64
max_length = 200
trunc_type = 'post'
padding_type = 'post'
oov_tok = '<OOV>'
training_portion = .8
```

hyperparameter.py hosted with ❤ by **GitHub** **view raw**

hyperparameter.py

- Define two lists containing articles and labels. In the meantime, we remove stopwords.

```
articles = []
labels = []

with open("bbc-text.csv", 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    next(reader)
```

```
    for row in reader:
        labels.append(row[0])
        article = row[1]
        for word in STOPWORDS:
            token = ' ' + word + ' '
            article = article.replace(token, ' ')
            article = article.replace(' ', ' ')
        articles.append(article)
print(len(labels))
print(len(articles))
```

articles_labels.py hosted with ❤ by **GitHub** **view raw**

articles_labels.py

```
2225
2225
```

There are 2,225 news articles in the data, we split them into training set and validation set, according to the parameter we set earlier, 80% for training, 20% for validation.

```
train_size = int(len(articles) * training_portion)

train_articles = articles[0: train_size]
train_labels = labels[0: train_size]

validation_articles = articles[train_size:]
validation_labels = labels[train_size:]

print(train_size)
print(len(train_articles))
print(len(train_labels))
print(len(validation_articles))
print(len(validation_labels))
```

train_valid.py hosted with ❤ by **GitHub** **view raw**

train_valid.py

```
1780
1780
1780
```

```
445
445
```

Tokenizer does all the heavy lifting for us. In our articles that it was tokenizing, it will take 5,000 most common words. `oov_token` is to put a special value in when an unseen word is encountered. This means we want `<OOV>` to be used for words that are not in the `word_index`. `fit_on_text` will go through all the text and create dictionary like this:

```python
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_articles)
word_index = tokenizer.word_index
dict(list(word_index.items())[0:10])
```

tokenize.py hosted with ❤ by **GitHub** **view raw**

tokenize.py

```
{'<OOV>': 1,
 'said': 2,
 'mr': 3,
 'would': 4,
 'year': 5,
 'also': 6,
 'people': 7,
 'new': 8,
 'us': 9,
 'one': 10}
```

We can see that “<OOV>” is the most common token in our corpus, followed by “said”, followed by “mr” and so on.

After tokenization, the next step is to turn those tokens into lists of sequence. The following is the 11th article in the training data that has been turned into sequences.

```python
train_sequences = tokenizer.texts_to_sequences(train_articles)
print(train_sequences[10])
```

```
[2432, 1, 225, 4995, 22, 642, 587, 225, 4995, 1, 1, 1662, 1, 1, 2432, 22, 565, 1, 1, 140, 278, 1, 140, 27
8, 796, 822, 662, 2308, 1, 1144, 1693, 1, 1720, 4996, 1, 1, 1, 1, 1, 4737, 1, 1, 122, 4513, 1, 2, 2875, 15
06, 352, 4738, 1, 52, 341, 1, 352, 2173, 3961, 41, 22, 3794, 1, 1, 1, 1, 543, 1, 1, 1, 835, 631, 2367, 34
7, 4739, 1, 365, 22, 1, 787, 2368, 1, 4301, 138, 10, 1, 3665, 682, 3531, 1, 22, 1, 414, 822, 662, 1, 90, 1
3, 633, 1, 225, 4995, 1, 599, 1, 1693, 1021, 1, 4997, 807, 1863, 117, 1, 1, 1, 2975, 22, 1, 99, 278, 1, 16
08, 4998, 543, 492, 1, 1446, 4740, 778, 1320, 1, 1860, 10, 33, 642, 319, 1, 62, 478, 565, 301, 1507, 22, 4
79, 1, 1, 1665, 1, 797, 1, 3067, 1, 1365, 6, 1, 2432, 565, 22, 2972, 4734, 1, 1, 1, 1, 1, 850, 39, 1824, 6
75, 297, 26, 979, 1, 882, 22, 361, 22, 13, 301, 1507, 1343, 374, 20, 63, 883, 1096, 4302, 247]
```

Figure 1

When we train neural networks for NLP, we need sequences to be in the same size, that's why we use padding. If you look up, our `max_length` is 200, so we use `pad_sequences` to make all of our articles the same length which is 200. As a result, you will see that the 1st article was 426 in length, it becomes 200, the 2nd article was 192 in length, it becomes 200, and so on.

```
train_padded = pad_sequences(train_sequences, maxlen=max_length,
padding=padding_type, truncating=trunc_type)

print(len(train_sequences[0]))
print(len(train_padded[0]))

print(len(train_sequences[1]))
print(len(train_padded[1]))

print(len(train_sequences[10]))
print(len(train_padded[10]))
```

```
425
200
192
200
186
200
```

In addition, there is `padding_type` and `truncating_type` , there are all `post` , means for example, for the 11th article, it was 186 in length, we padded to 200, and we padded at the end, that is adding 14 zeros.

```
print(train_padded[10])
```

```
[2432    1  225 4995   22  642  587  225 4995    1    1 1662    1    1
 2432   22  565    1    1  140  278    1  140  278  796  822  662 2308
    1 1144 1693    1 1720 4996    1    1    1    1    1 4737    1    1
  122 4513    1    2 2875 1506  352 4738    1   52  341    1  352 2173
 3961   41   22 3794    1    1    1    1  543    1    1    1  835  631
 2367  347 4739    1  365   22    1  787 2368    1 4301  138   10    1
 3665  682 3531    1   22    1  414  822  662    1   90   13  633    1
  225 4995    1  599    1 1693 1021    1 4997  807 1863  117    1    1
    1 2975   22    1   99  278    1 1608 4998  543  492    1 1446 4740
  778 1320    1 1860   10   33  642  319    1   62  478  565  301 1507
   22  479    1    1 1665    1  797    1 3067    1 1365    6    1 2432
  565   22 2972 4734    1    1    1    1    1  850   39 1824  675  297
   26  979    1  882   22  361   22   13  301 1507 1343  374   20   63
  883 1096 4302  247    0    0    0    0    0    0    0    0    0    0
    0    0    0    0]
```

Figure 2

And for the 1st article, it was 426 in length, we truncated to 200, and we truncated at the end as well.

Then we do the same for the validation sequences.

```
validation_sequences = tokenizer.texts_to_sequences(validation_articles)
validation_padded = pad_sequences(validation_sequences, maxlen=max_length, padding=paddin

print(len(validation_sequences))
print(validation_padded.shape)
```

val_tok.py hosted with ❤ by **GitHub** **view raw**

val_tok.py

```
445
(445, 200)
```

Now we are going to look at the labels. Because our labels are text, so we will tokenize them, when training, labels are expected to be numpy arrays. So we will turn list of labels into numpy arrays like so:

```
label_tokenizer = Tokenizer()
label_tokenizer.fit_on_texts(labels)

training_label_seq =
np.array(label_tokenizer.texts_to_sequences(train_labels))
validation_label_seq =
np.array(label_tokenizer.texts_to_sequences(validation_labels))

print(training_label_seq[0])
print(training_label_seq[1])
print(training_label_seq[2])
print(training_label_seq.shape)

print(validation_label_seq[0])
print(validation_label_seq[1])
print(validation_label_seq[2])
print(validation_label_seq.shape)
```

```
[4]
[2]
[1]
(1780, 1)
[5]
[4]
[3]
(445, 1)
```

Before training deep neural network, we should explore what our original article and article after padding look like. Running the following code, we explore the 11th article, we can see that some words become “<OOV>”, because they did not make to the top 5,000.

```
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])

def decode_article(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
print(decode_article(train_padded[10]))
print('---')
print(train_articles[10])
```

Figure 3

Now its the time to implement LSTM.

- We build a `tf.keras.Sequential` model and start with an embedding layer. An embedding layer stores one vector per word. When called, it converts the sequences of word indices into sequences of vectors. After training, words with similar meanings often have the similar vectors.

- The Bidirectional wrapper is used with a LSTM layer, this propagates the input forwards and backwards through the LSTM layer and then concatenates the outputs. This helps LSTM to learn long term dependencies. We then fit it to a dense neural network to do classification.

- We use `relu` in place of `tahn` function since they are very good alternatives of each other.

- We add a Dense layer with 6 units and `softmax` activation. When we have multiple outputs, `softmax` converts outputs layers into a probability distribution.

```
model = tf.keras.Sequential([
    # Add an Embedding layer expecting input vocab of size 5000, and output embedding di
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
#    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32))
```

```
#    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    # use ReLU in place of tanh function since they are very good alternatives of each o
    tf.keras.layers.Dense(embedding_dim, activation='relu'),
    # Add a Dense layer with 6 units and softmax activation.
    # When we have multiple outputs, softmax convert outputs layers into a probability d
    tf.keras.layers.Dense(6, activation='softmax')
])
model.summary()
```

lstm_model.py hosted with ❤ by **GitHub** **view raw**

lstm_model.py

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          320000
_________________________________________________________________
bidirectional (Bidirectional (None, 128)               66048
_________________________________________________________________
dense (Dense)                (None, 64)                8256
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 390
=================================================================
Total params: 394,694
Trainable params: 394,694
Non-trainable params: 0
_________________________________________________________________
```

Figure 4

In our model summary, we have our embeddings, our Bidirectional contains LSTM, followed by two dense layers. The output from Bidirectional is 128, because it doubled what we put in LSTM. We can also stack LSTM layer but I found the results worse.

```
print(set(labels))
```

```
{'tech', 'politics', 'sport', 'business', 'entertainment'}
```

We have 5 labels in total, but because we did not one-hot encode labels, we have to use `sparse_categorical_crossentropy` as loss function, it seems to think 0 is a possible label as well, while the tokenizer object which tokenizes starting with integer 1, instead of integer 0. As a result, the last Dense layer needs outputs for labels 0, 1, 2, 3, 4, 5 although 0 has never been used.

If you want the last Dense layer to be 5, you will need to subtract 1 from the training and validation labels. I decided to leave it as it is.

I decided to train 10 epochs, and it is plenty of epochs as you will see.

```
model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])

num_epochs = 10
history = model.fit(train_padded, training_label_seq,
epochs=num_epochs, validation_data=(validation_padded,
validation_label_seq), verbose=2)
```

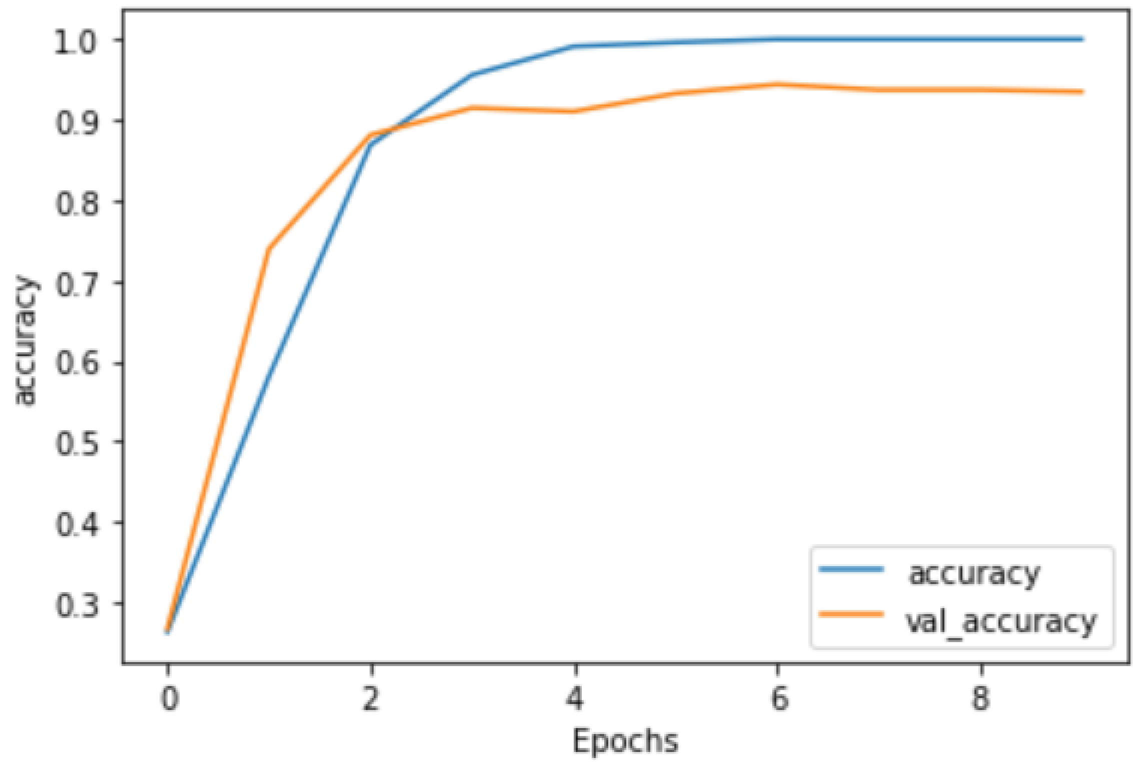
```
Train on 1780 samples, validate on 445 samples
Epoch 1/10
1780/1780 - 10s - loss: 1.6322 - accuracy: 0.2635 - val_loss: 1.4729 - val_accuracy: 0.2674
Epoch 2/10
1780/1780 - 5s - loss: 1.0612 - accuracy: 0.5809 - val_loss: 0.7554 - val_accuracy: 0.7393
Epoch 3/10
1780/1780 - 5s - loss: 0.3791 - accuracy: 0.8685 - val_loss: 0.3497 - val_accuracy: 0.8809
Epoch 4/10
1780/1780 - 5s - loss: 0.1476 - accuracy: 0.9556 - val_loss: 0.2603 - val_accuracy: 0.9146
Epoch 5/10
1780/1780 - 5s - loss: 0.0444 - accuracy: 0.9910 - val_loss: 0.3338 - val_accuracy: 0.9101
Epoch 6/10
1780/1780 - 5s - loss: 0.0185 - accuracy: 0.9961 - val_loss: 0.2438 - val_accuracy: 0.9326
Epoch 7/10
1780/1780 - 5s - loss: 0.0042 - accuracy: 1.0000 - val_loss: 0.2118 - val_accuracy: 0.9438
Epoch 8/10
1780/1780 - 5s - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.2476 - val_accuracy: 0.9371
Epoch 9/10
1780/1780 - 5s - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.2752 - val_accuracy: 0.9371
Epoch 10/10
1780/1780 - 5s - loss: 7.9578e-04 - accuracy: 1.0000 - val_loss: 0.2882 - val_accuracy: 0.9348
```

Figure 5

```
def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
```

```
    plt.xlabel("Epochs")
    plt.ylabel(string)
    plt.legend([string, 'val_'+string])
    plt.show()

plot_graphs(history, "accuracy")
plot_graphs(history, "loss")
```



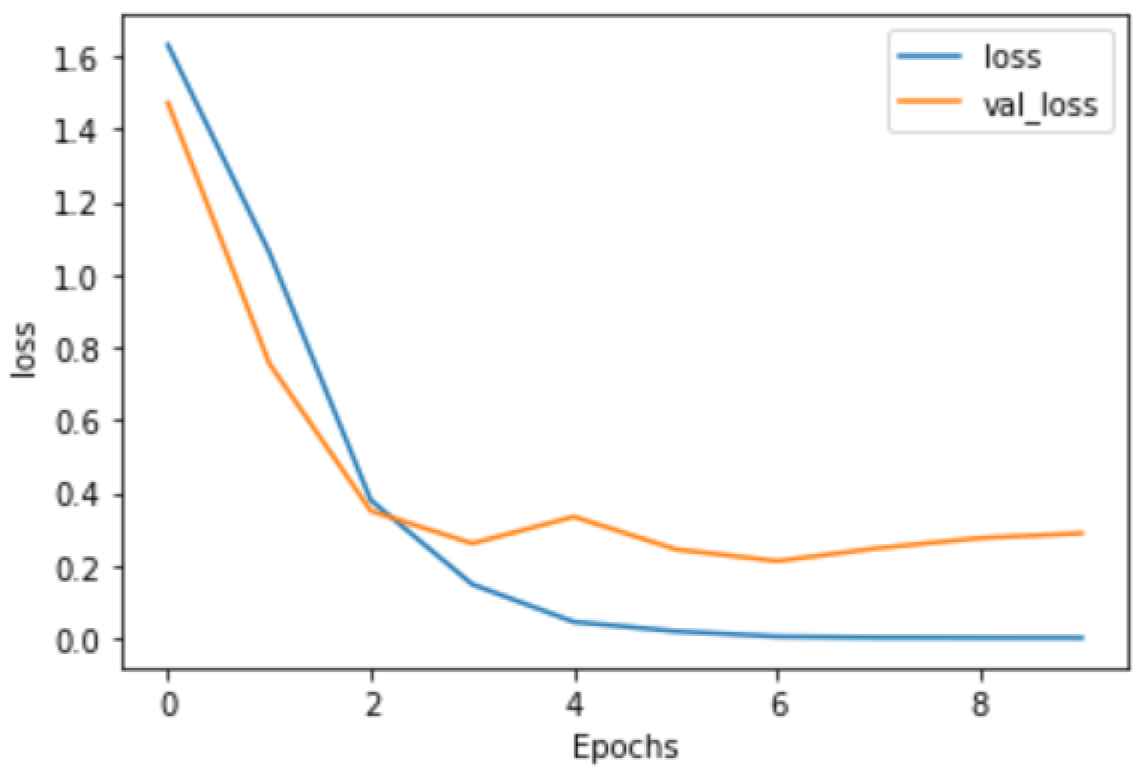


Figure 6

We probably only need 3 or 4 epochs. At the end of the training, we can see that there is a little bit overfitting.

In the future posts, we will work on improving the model.

Jupyter notebook can be found on Github. Enjoy the rest of the weekend!

References:

Coursera | Online Courses From Top Universities. Join for Free

1000+ courses from schools like Stanford and Yale - no application required. Build career skills in data science…

www.coursera.org

O'Reilly Strata Data Conference 2019 - New York, New York

Selection from O'Reilly Strata Data Conference 2019 - New York, New York [Video]

learning.oreilly.co

Machine Learning NLP Tensorflow2 Tensorflow Tutorial Lstm