An easy tutorial about Sentiment Analysis with Deep Learning and Keras

Learn how to easily build, train and validate a Recurrent Neural Network

Sergio Virahonda Oct 8, 2020 · 14 min read

Photo by Alina Grubnyak on Unsplash

you as data scientist/machine learning engineer must know how to perform because at some point of your career you'll be required to do so. In the context of this article, I'll assume you have a basic understanding of what I'm going to talk in the next lines. I'll be stacking layers of concepts as I move forward, keeping a very low-level language — don't worry if you fell a little lost between lines, later I will probably clarify your doubts. The main idea is for you to understand what I'll be explaining. That being said, let's get hands-on (Btw, don't miss any detail and download the whole project from my repo.)

I'll start by defining the first unusual term in the title: **Sentiment Analysis** is a very frequent term within text classification and is essentially to use natural language processing (quite often referred simply as NLP) + machine learning to interpret and classify emotions in text information. Imagine the task of determining whether a product's review is positive or negative; you could do it yourself just by reading it, right? But what happens when the company you work for sells 2k products *every single day*? Are you pretending to read all the reviews and manually classify them? Let's be honest, your job would be the worst ever. There's where Sentiment Analysis comes in and makes your life and job easier.

Let's get into the matter

There are several ways to implement Sentiment Analysis and each data scientist has his/her own preferred method, I'll guide you through a very simple one so you can understand what it involves, but also suggest you some others that way you can research about them. Let's place first things first: If you are not familiar with Machine Learning, you must know all algorithms are only able to understand and process numeric data (particularly floating point data), thus you cannot feed them with text and wait for them to solve your problems; instead, you'll have to make several transformations to your data until it reaches a representative numeric shape. The common and most basic steps are:

- Remove URLs and email addresses from every single sample — because they won't add meaningful value.
- Remove punctuation signs — otherwise your model won't understand that "good!" and "good" are actually meaning the same thing.

understood differently than the “good” in another sample.

- Remove stop-words — because they only add noise and won’t make the data more meaningful. Btw, stop-words refer to the most common words in a language, such as “I”, “have”, “are” and so on. I hope you get the point because there’s not an official stop-words list out there.

- Stemming/Lemmatizing: This step is optional, but for most of data scientist considered as crucial. I’ll show you that it’s not THAT relevant to achieve good results. Stemming and lemmatizing are very similar tasks, both look forward to extract the root words from every word of a sentence of the corpus data. Lemmatizing generally returns valid words (that exist) while stemming techniques return (most of the times) shorten words, that’s why lemmatizing is used more in real world implementations. This is how lemmatizers vs. stemmers work: suppose you want to find the root word of ‘caring’: ‘Caring’ -> *Lemmatization* -> ‘Care’. In the other hand: ‘Caring’ -> *Stemming* -> ‘Car’; did you get the point? You can research about both and obviously implement any if the business requires it.

- Transform dataset (text) into numeric tensors — Usually referred as *vectorization*. If you recall some lines above, I explained that like all other neural networks, deep-learning models don’t take as input raw text: they only work with numeric tensors, that’s why this step is not negotiable. There are multiple ways to do so; for example, if you’re going to use a classic ML model (not DL) then you definitely should go with CountVectorizer, TFIDF Vectorizer or just the basic but not so good approach: Bag-Of-Words. It’s up to you. However, if you’re going to implement Deep Learning you might know that the best way is to turn your text data (that can be understood as sequences of word or sequences of characters) into low-dimensional floating-point vectors — don’t worry, I’ll explain this in a bit.

This is how a very basic text cleaning Python function would look like (this is a very simple way, you can implement the one that best works for your purpose — out there are very complete libraries such as Gensim or NLTK):

```
    #Removing URLs with a regular expression
    url_pattern = re.compile(r'https?://\S+|www\.\S+')
    data = url_pattern.sub(r'', data)

    # Remove Emails
    data = re.sub('\S*@\S*\s?', '', data)

    # Remove new line characters
    data = re.sub('\s+', ' ', data)

    # Remove distracting single quotes
    data = re.sub("\'", "", data)

    return data
```

Now a very simple way to remove repeating words from your dataset, following the idea that the shortest and longest ones are usually useless:

```
def sent_to_words(sentences):
    for sentence in sentences:
        yield(gensim.utils.simple_preprocess(str(sentence),
deacc=True))
```

Finally, a function to detokenize all sentences (this because I'll use word embeddings and not this old-fashioned tokenizing method):

```
def detokenize(text):
    return TreebankWordDetokenizer().detokenize(text)
```

To run everything in the right order, you just need to run this:

```
temp = []
#Splitting pd.Series to list
data_to_list = train['selected_text'].values.tolist()
for i in range(len(data_to_list)):
    temp.append(depure_data(data_to_list[i]))
data_words = list(sent_to_words(temp))
data = []
```

At this point, you will have transformed your noisy text dataset into a very flat and simple one. In this particular case, you'll have gone from this:

```
['I`d have responded, if I were going',
 'Sooo SAD',
 'bullying me',
 'leave me alone',
 'Sons of ****,']
```

To this:

```
['have responded if were going', 'sooo sad', 'bullying me', 'leave me
alone', 'sons of']
```

If you want to go even beyond, then take the stemming or lemmatizing paths, you'll get much better results. In this particular how-to, will keep it this way just you to get that it's totally possible to achieve great results skipping that step (if you're building your model in a business context then you'll be 100% in obligation to do so, don't skip it!)

From sentences to word embeddings

Alright, it's time to understand an extremely important step you'll have to deal with when working with text data. Once you have your text data completely clean of noise, it's time to transform it into floating-point tensors. In order to perform this task, we'll use *word-embeddings*.

Word embeddings (or sometimes called *word vectors*) are learned from data and essentially are low-dimensional floating-point vectors (dense vectors, as opposed to sparse vectors obtained from processes such as one-hot-encoding) that pack information in few dimensions. Why would you use this method and not any other different and more simple? Because deep learning models converge easier with dense vectors than with sparse ones. Again, it always depends on the dataset nature and the business need.

There are two ways to obtain word embeddings:

- Use a pretrained word embedding stack on top of your model, just as you would use a pretrained NN layer (or group of layers) — a very infrequent approach.
- Learn word embeddings from scratch. To achieve this, you'd start with random word vectors and progressively learn meaningful ones just as a NN would learn its weights. This is the option that we'll use and actually is reasonable to learn a new embedding space with every single new task. Fortunately, this step is very straightforward with TensorFlow or Keras, and you'd implement word embedding just like one more layer in your NN stack.

Before moving forward, we need to take a previous step. We need to transform our array of text into 2D numeric arrays:

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras import regularizers

max_words = 5000
max_len = 200

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(data)
sequences = tokenizer.texts_to_sequences(data)
tweets = pad_sequences(sequences, maxlen=max_len)
print(tweets)
```

The output you'll get would look like this:

```
[[   0    0    0 ...   68  146   41]
 [   0    0    0 ...    0  397   65]
 [   0    0    0 ...    0    0   11]
 ...
 [   0    0    0 ...  372   10    3]
 [   0    0    0 ...   24  542    4]
 [   0    0    0 ... 2424  199  657]]
```

> *This function transforms a list (of length* `num_samples`*) of sequences (lists of integers) into a 2D Numpy array of shape* `(num_samples, num_timesteps)`*.* `num_timesteps` *is either the* `maxlen` *argument if provided, or the length of the longest sequence in the list.*
>
> *Sequences that are shorter than* `num_timesteps` *are padded with* `value` *until they are* `num_timesteps` *long.*
>
> *Sequences longer than* `num_timesteps` *are truncated so that they fit the desired length*

The Embedding layer

It's extremely important you to keep in mind that no matter if you use TensorFlow or any other *Astraction API* such as Keras*,* you should get the same result at the end of your training. In this opportunity we'll use Keras for obvious reasons: It's extremely easy to implement. This is how you create an embedding layer:

```
from keras.layers import Embedding
embedding_layer = Embedding(1000, 64)
```

The above layer takes 2D integer tensors of shape (samples, sequence_length) and at least two arguments: the number of possible tokens and the dimensionality of the embeddings (here 1000 and 64, respectively). To be more figurative, just imagine the embedding layer is a dictionary that links integer indices to dense vectors. Finally, it returns a 3D floating-point tensor of shape (samples, sequence_length, embedding_dimensionality) that can now be processed by our neural network. Let's talk about that topic, particularly about Recurrent Neural Networks that are the best when processing text-related sequences is required.

Recurrent Neural Networks made easy

Usually other types of neural networks out there such as densely connected networks or convolutional networks have no memory, this means that every single input is processed independently with no relation with the other ones. This is the opposite of what you

meaning and this is exactly the same principle that RNNs adopt. They process sequences by iterating along the sequence elements and keeping information relative to what it has processed so far. The math that is under a RNN's hood is, to be honest, a topic that you should cover by yourself in order to understand its logic. I suggest you to give a read to "*Learning TensorFlow*" by Tom Hope (available here) which explains all the process in a very easy manner.

In this article, I'll implement three RNN types: a single LSTM (long short-term memory) model, a Bidirectional LSTM and a very infrequent used Conv1D model. As a bonus, I show how to implement a SimpleRNN model but to be honest, it's not deployed in production anywhere because it's extremely simple.

LSTM layers

Coming back to our example, this is how the code would look like when implementing a single LSTM layer model with its respective embedding layer:

```
from keras.models import Sequential
from keras import layers
from keras import regularizers
from keras import backend as K
from keras.callbacks import ModelCheckpoint

model1 = Sequential()
model1.add(layers.Embedding(max_words, 20)) #The embedding layer
model1.add(layers.LSTM(15,dropout=0.5)) #Our LSTM layer
model1.add(layers.Dense(3,activation='softmax'))


model1.compile(optimizer='rmsprop',loss='categorical_crossentropy',
metrics=['accuracy'])

checkpoint1 = ModelCheckpoint("best_model1.hdf5",
monitor='val_accuracy', verbose=1,save_best_only=True, mode='auto',
period=1,save_weights_only=False)

history = model1.fit(X_train, y_train, epochs=70,validation_data=
(X_test, y_test),callbacks=[checkpoint1])
```

layers) can take several arguments but the ones that I defined are *15* which is the number of hidden units within the layer (must be a positive integer and represents the dimensionality of the output space) and the dropout rate of the layer. **Dropout is one of the most effective and most commonly used regularization techniques for NNs** and consists of randomly turning off hidden units during training, that way the network does not rely 100% on all its neurons and instead, forces itself to find more meaningful patterns in the data in order to increase the metric you're trying to optimize. There are several other arguments to pass, you can find the complete documentation here, but for this particular example, these settings will achieve good results.

FYI, sometimes it's useful to **stack several recurrent layers** one after the other in order to increase the representational power of a network. If you want to do so then **you'll have to return full sequences of outputs**. This is an example:

```
model0 = Sequential()
model0.add(layers.Embedding(max_words, 15))
model0.add(layers.SimpleRNN(15,return_sequences=True))
model0.add(layers.SimpleRNN(15))
model0.add(layers.Dense(3,activation='softmax'))
```

In our LSTM example I'm stacking a Dense layer with three output units that would be the 3 possible classes of our dataset. In order to make probabilistic outputs, it's always good to use 'softmax' as activation function in the final layer. Use the next table when building a neural network and you feel confused:

Problem type	Last-layer function	Loss function
Binary classfication	**Sigmoid** ('sigmoid')	Binary Crossentropy ('binary_crossentropy')
Multiclass classfication (single label classification)	**Softmax** ('softmax')	Categorical Crossentropy ('categorical_crossentropy')
Multiclass, multilabel classfication	**Sigmoid** ('sigmoid')	Binary Crossentropy ('binary_crossentropy')

Regression (to values between 0 and 1)	**Sigmoid** ('sigmoid')	MSE or Binary Crossentropy

Basic parameters for Neural Networks' configuration and training — Image by author.

When compiling the model, I'm using RMSprop optimizer with its default learning rate but actually this is up to every developer. Some people love Adam, some others Adadelta, and so on. To be honest, RMSprop or Adam should be enough in most of the cases. If you don't know what an optimizer is, it's simply the mechanism that constantly computes the gradient of the loss and defines how to move against the loss function in order to find its global minima and therefore, find the best network's parameters (the model kernel and its bias' weights). As loss function, I use categorical_crossentropy (Check the table) that is typically used when you're dealing with multiclass classification tasks. In the other hand, you would use binary_crossentropy when binary classification is required.

Finally, I'm using checkpoints to save the best model achieved in the training process. This is very useful when you need to get the model that best satisfies the metric you're trying to optimize. Then the classic model.fit step and wait for it to complete the training iterations.

This is the validation score achieved by this NN architecture at its last epoch:

```
Epoch 70/70
645/645 [==============================] - ETA: 0s - loss: 0.3090 -
accuracy: 0.8881
Epoch 00070: val_accuracy did not improve from 0.84558
```

Let's compare this with a more complex network.

Bidirectional layers

This is how our example's BidRNN implementation looks like:

```
model2.add(layers.Dense(3,activation='softmax'))
model2.compile(optimizer='rmsprop',loss='categorical_crossentropy',
metrics=['accuracy'])
checkpoint2 = ModelCheckpoint("best_model2.hdf5",
monitor='val_accuracy', verbose=1,save_best_only=True, mode='auto',
period=1,save_weights_only=False)
history = model2.fit(X_train, y_train, epochs=70,validation_data=
(X_test, y_test),callbacks=[checkpoint2])
```

Let's understand better how a bidirectional layer works. It maximizes the order sensitivity of the RNNs: essentially it consists of two RNNs (LSTMs or GRUs) that process the input sequence in one different direction to finally merge representations. By doing this, they're able to catch more complex patterns than a single RNN layer would catch. In other words, one of the layers interprets the sequences in chronological order and the second one do so in anti-chronological order, that's why Bidirectional RNNs are widely used, because they offer greater performance than regular RNNs.

The way they are implemented is not complex, it is just one layer inside the other. If you read carefully, I'm using almost the same parameters but achieved about .3% more of validation accuracy on its overall training:

```
Epoch 70/70
644/645 [============================>.] - ETA: 0s - loss: 0.2876 -
accuracy: 0.8965
Epoch 00070: val_accuracy did not improve from 0.84849
```

It's a very good number even when it's a very simple model and I wasn't focused on hyperparameter tuning. I'm sure that if you dedicate yourself to adjust them then will get a very good result. Unfortunately, there's no magical formula to do so, it's all about adjusting its architecture and forcing it to learn every time more complex patterns and control its overfitting tendency with more regularization. One important thing to highlight is that if you see your model accuracy/loss stuck around certain value, it's likely because the learning rate is too small and therefore making your optimizer to get stuck around a local minima of the loss function; make the LR bigger or simply try another optimizer.

datasets. Let's get into it.

Going even further — 1D Convolutional neural networks

I hope you're still with me, because this is one of the fastest models out there when talking about convergence — it demands a cheaper computational cost. I know by prior experience that it tends to overfit extremely quick on small datasets. In this sense, just will implement it to show you how to do so in case it's of your interest and also give you an overview about how it works.

It uses the same principles as classic 2D ConvNets used for image classification. Convolutional layers extract patches from 1D/2D tensors (depending on the type of task and layer) and apply the same convolutional transformations to every one of them (getting as output several subsequences). I won't get deep in such explanation because that's out of the scope of this article, but if you want to fully understand how these layers work I would suggest to you check the book previously recommended. The most important fact of these layers is that they can recognize patters in a sequence — A pattern learned at a certain position in a sentence can later be identified in a different position or even in another sentence.

This is how 1D ConvNets are implemented:

```
model3.add(layers.Embedding(max_words, 40, input_length=max_len))
model3.add(layers.Conv1D(20, 6,
activation='relu',kernel_regularizer=regularizers.l1_l2(l1=2e-3,
l2=2e-3),bias_regularizer=regularizers.l2(2e-3)))
model3.add(layers.MaxPooling1D(5))
model3.add(layers.Conv1D(20, 6,
activation='relu',kernel_regularizer=regularizers.l1_l2(l1=2e-3,
l2=2e-3),bias_regularizer=regularizers.l2(2e-3)))
model3.add(layers.GlobalMaxPooling1D())
model3.add(layers.Dense(3,activation='softmax'))
model3.compile(optimizer='rmsprop',loss='categorical_crossentropy',me
trics=['acc'])
history = model3.fit(X_train, y_train, epochs=70,validation_data=
(X_test, y_test))
```

convolutional output. Once the convolution operation is performed, the MaxPooling window extracts the highest value within it and outputs patches of maximum values. It's important to highlight the importance of regularizers in this type of configuration, otherwise your network will learn meaningless patterns and overfit extremely fast — just FYI.

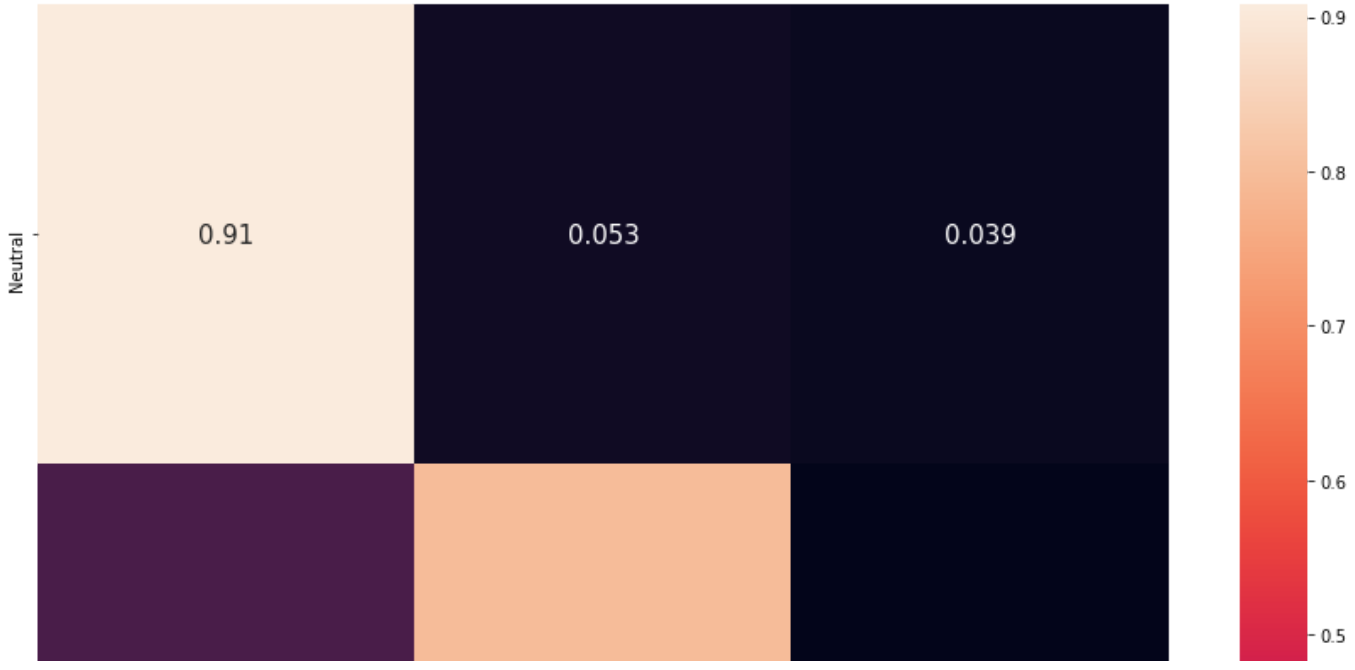
To contrast how the previous model performs, this is the metric achieved at the last epoch:

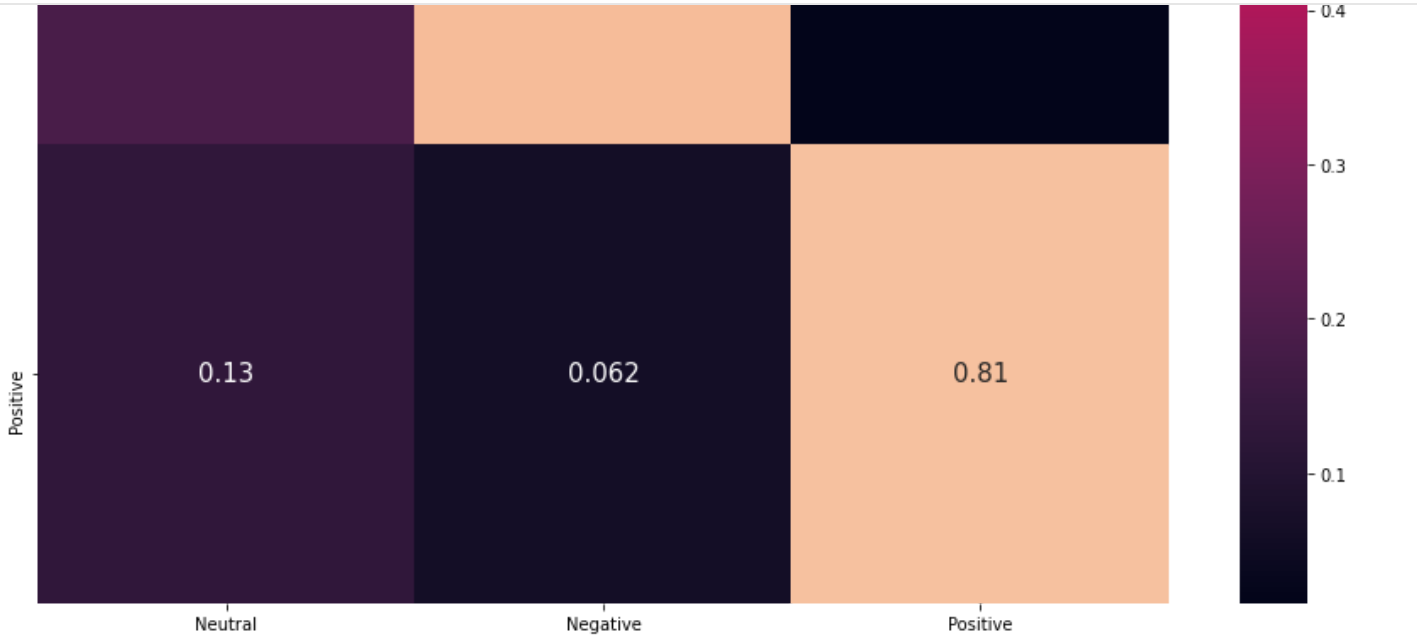
```
Epoch 70/70
645/645 [==============================] - 5s 7ms/step - loss: 0.3096
- acc: 0.9173 - val_loss: 0.5819 - val_acc: 0.8195
```

And its best validation accuracy obtained was around 82%. It overfits very fast even when I've implemented very drastic regularization.

Validation of our best model

At this point, the best model so far has been the Bidirectional RNN. Please keep in mind that these metrics were obtained with little-to-zero hyperparameter tuning. To understand in a better way how well its predictions are, let's look at its confusion matrix:





Best model's confusion matrix — Image by author.

From the above's image, we can deduce: 81% of positive ratings were classified as positive, 80% of negative ratings were classified as negative and 91% of neutral ratings were classified as neutral. These are not the best predictions, but are a good baseline to work on even better models. In a business scenario you would need to be close to 95% in the most easy cases.

If you want to test how it works on an input made by yourself, just compute the next lines:

```
sentiment = ['Neutral','Negative','Positive']

sequence = tokenizer.texts_to_sequences(['this data science article
is the best ever'])
test = pad_sequences(sequence, maxlen=max_len)
sentiment[np.around(best_model.predict(test),
decimals=0).argmax(axis=1)[0]]
```

And the output will be:

```
'Positive'
```

Final thoughts

Alright, we've reached the end of this post. I encourage you to implement all models by yourself and focus on hyperparameter tuning which is one of the tasks that takes longer. Once you've reached a good number, I'll see you back here to guide you through that model's deployment ☺.

There are several ways to do such a task. You can use the Google Cloud Platform, take the Azure path, even the cheaper way of Heroku, but let's be honest: most of the biggest companies are adopting AWS as their main public cloud provider and these guys have a fantastic platform to build, train and deploy ML models: **AWS SageMaker**; which has tons of documentation out there. I'll be posting another step-by-step well-explained tutorial about how to easily deploy models on it. I hope to see you there!