Published in Towards Data Science · Follow

Karsten Eckhardt · Follow

Nov 29, 2018 · 9 min read

Choosing the right Hyperparameters for a simple LSTM using Keras

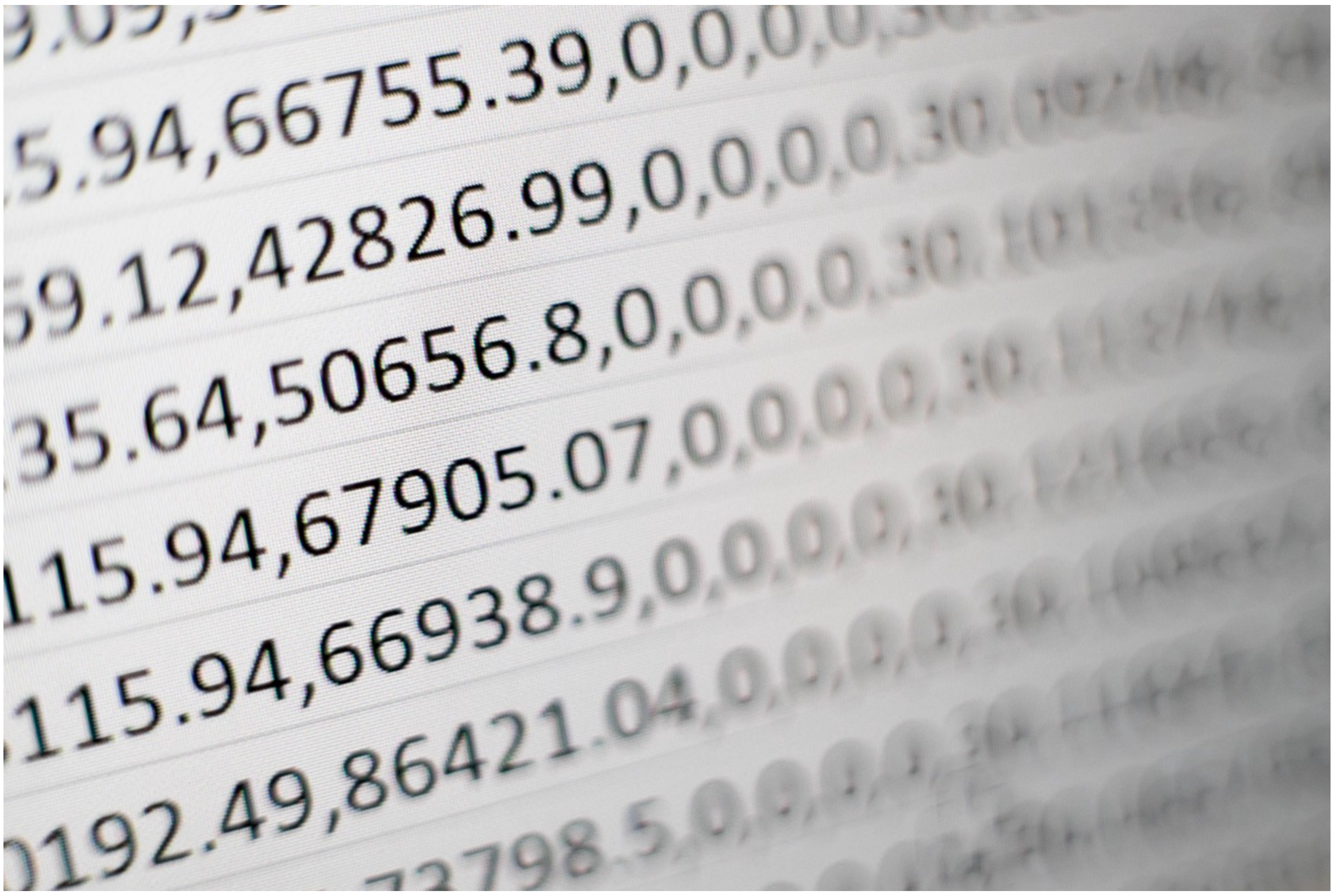


Photo by Mika Baumeister on Unsplash

Building Machine Learning models has never been easier and many articles out there give a great high-level overview on what Data Science is and the amazing things it can do, or go into depth about a really smaller implementation detail. This leaves aspiring Data Scientists, like me a while ago, often looking at Notebooks out there, thinking “It looks great and works, but why did the author choose this type of architecture/number of neurons or this activation function instead of another? In this article, I want to give concepts behind Recurring Neural Networks (RNN), e.g. Andrew Ng's deep learning specialization or here on Medium, I will not dig deeper into them and perceive this knowledge as given. Instead, we will only focus on the high-level implementation using Keras. The goal is to get a more practical understanding of decisions one has to make building a neural network like this, especially on how to chose some of the hyperparameters.

The full article with code and outputs can be found on Github as a Notebook.

On Keras: Latest since its TensorFlow Support in 2017, Keras has made a huge splash as an easy to use and intuitive interface into more complex machine learning libraries. As a result, building the actual neural network, as well as training the model is going to be the shortest part in our script.

The first step is to determine the type of network we want to use since that decision can impact our data preparation process. The order of characters in any name (or word) matters, meaning that, if we want to analyze a name using a neural network, RNN are the logical choice. Long Short-Term Memory Networks (LSTM) are a special form of RNNs are especially powerful when it comes to finding the right features when the chain of input-chunks becomes longer. In our case, the input is always a string (the name) and the output a 1x2 vector indicating if the name belongs to a male or a female person.

After making this decision, We will start with loading all the packages that we will need as well as the dataset — a file containing over 1.5 Mio German users with their name and gender, encoded as *f* for female and *m* for male.

Preprocessing the data

The next step in any natural language processing is to convert the input into a machine-readable vector format. In theory, neural networks in Keras are able to handle inputs with a variable shape. In praxis, working with a fixed input length in Keras can improve performance noticeably, especially during the training. The reason for this behavior is that this fixed input length allows for the creation of fixed-shaped tensors and therefore more stable weights.

First, we will convert every (first) name into a vector. The method we'll be using is the so-called One-Hot Encoding.
Here, every word is represented by a vector of n binary sub-vectors, where n is the number of different chars in the alphabet (26 using the English alphabet). The reason why we can not simply convert every character to its position in the alphabet, e.g. a — 1, b — 2 etc.) is that this would lead the network to assume that the characters are on an ordinal scale, instead of a categorical - the letter *Z* not is "worth more" than an *A*.

Example:
S becomes:
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

hello becomes:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

Now that we determined how the input has to look like, we have two decisions to make: How long shall the char vector be (how many different chars do we allow for) and how long shall the name vector be (how many chars we want to look at). We will only allow for the most common characters in the German alphabet (standard latin + öäü) and the hyphen, which is part of many older names.
For simplicity purposes, we will set the length of the name vector to be the length of the longest name in our dataset, but with 25 as an upper bound to make sure our input vector doesn't grow too large just because one person made a mistake during the name entering the process.

Scikit-learn already incorporates a One Hot Encoding algorithm in it’s preprocessing library. However, in this case, because of our special situation that we are not converting labels into vectors but split every string apart into its characters, the creation of a custom algorithm seemed to be quicker than the preprocessing otherwise needed.

It has been shown that Numpy arrays need around 4 times less memory compared to Python lists. For that reason, we use list comprehension as a more pythonic way of creating the input array but already convert every word vector into an array inside of the list. When working with Numpy arrays, we have to make sure that all lists and/or arrays that are getting combined have the same shape.

Now that we have our input ready, we can start building our neural network. We already decided on the model (LSTM). In Keras we can simply stack multiple layers on top of each other, for this we need to initialize the model as `Sequential()`.

Choosing the right amount of nodes and layers

There is no final, definite, rule of thumb on how many nodes (or hidden neurons) or how many layers one should choose, and very often a trial and error approach will give you the best results for your individual problem. The most common framework for this is most likely the k-fold cross-validation. However, even for a testing procedure, we need to choose some (k) numbers of nodes.
The following formula may give you a starting point:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

N_i is the number of input neurons, N_o the number of output neurons, N_s the number of samples in the training data, and **α** represents a scaling factor that is usually between 2 and 10. We can calculate 8 different numbers to feed into our validation procedure and find the optimal model, based on the resulting validation loss.

If the problem is simple and time an issue, there are various other rules of thumbs to determine the number of nodes, which are mostly simply based on the input and output neurons. We have to keep in mind that, while easy to use, they will rarely yield the optimal result. Here is just one example, which we will use for this basic model:

$$N_h = \frac{2}{3} * (N_i + N_o)$$

As mentioned, the same uncertainty about the amount also exists for the number of hidden layers to use. Again, the ideal number for any given use case will be different and is best to be decided by running different models against each other. Generally, 2 layers have shown to be enough to detect more complex features. More layers can be better but also harder to train. As a general rule of thumb — 1 hidden layer work with simple problems, like this, and two are enough to find reasonably complex features. In our case, adding a second layer only improves the accuracy by ~0.2% (0.9807 vs. 0.9819) after 10 epochs.

Choosing additional Hyper-Parameters

Every LSTM layer should be accompanied by a Dropout layer. This layer will help to prevent overfitting by ignoring randomly selected neurons during training, and hence reduces the sensitivity to the specific weights of individual neurons. 20% is often used as a good compromise between retaining model accuracy and preventing overfitting.

After our LSTM layer(s) did all the work to transform the input to make predictions towards the desired output possible, we have to reduce (or, in rare cases extend) the shape, to match our desired output. In our case, we have two output labels and therefore we need two-output units.

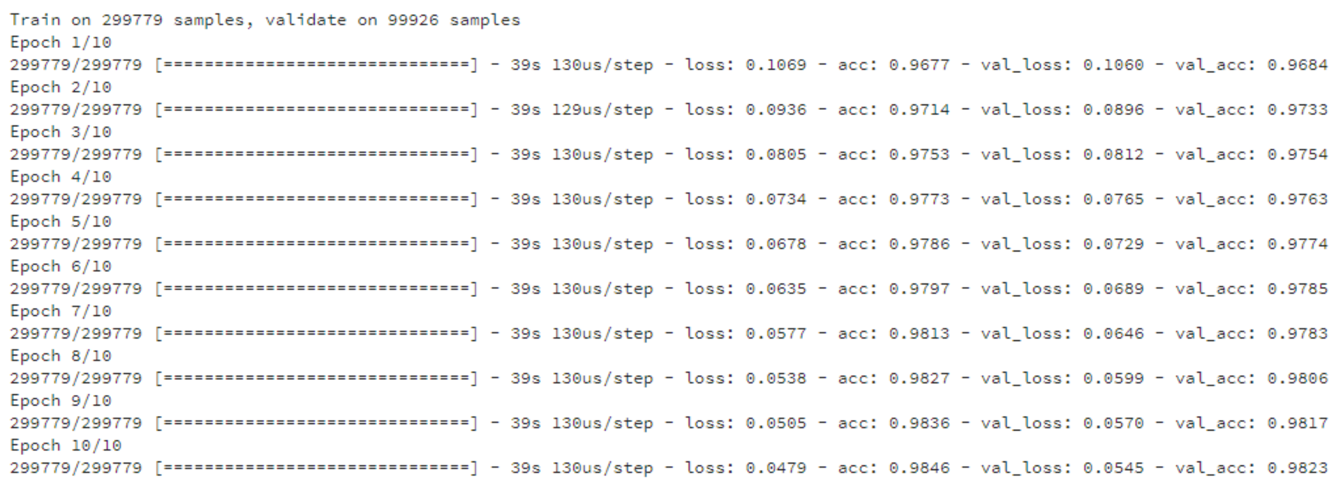
the density layer and the activation layer makes it possible to retrieve the reduced output of the density layer of the model. Which activation function to use is, again, depending on the application. For our problem at hand, we have multiple classes (male and female) but only one of the classes can be present at a time. For these types of problems, generally, the softmax activation function works best, because it allows us (and your model) to interpret the outputs as probabilities.

Loss function and activation function are often chosen together. Using the softmax activation function points us to cross-entropy as our preferred loss function or more precise the binary cross-entropy, since we are faced with a binary classification problem. Those two functions work well with each other because the cross-entropy function cancels out the plateaus at each end of the soft-max function and therefore speeds up the learning process.

For choosing the optimizer, adaptive moment estimation, short _Adam_, has been shown to work well in most practical applications and works well with only little changes in the hyperparameters. Last but not least we have to decide, after which metric we want to judge our model. Keras offered multiple accuracy functions. In many cases, judging the models' performance from an overall _accuracy_ point of view will be the option easiest to interpret as well as sufficient in resulting model performance.

Building, training and evaluating the model

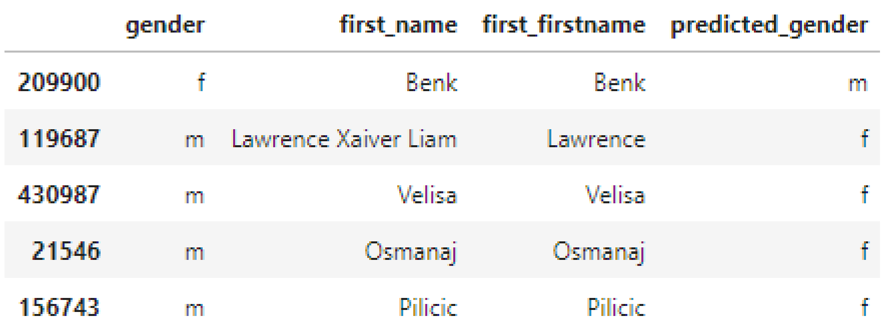
After getting some intuition about how to chose the most important parameters, let's put them all together and train our model:



```
Train on 299779 samples, validate on 99926 samples
Epoch 1/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.1069 - acc: 0.9677 - val_loss: 0.1060 - val_acc: 0.9684
Epoch 2/10
299779/299779 [==============================] - 39s 129us/step - loss: 0.0936 - acc: 0.9714 - val_loss: 0.0896 - val_acc: 0.9733
Epoch 3/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0805 - acc: 0.9753 - val_loss: 0.0812 - val_acc: 0.9754
Epoch 4/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0734 - acc: 0.9773 - val_loss: 0.0765 - val_acc: 0.9763
Epoch 5/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0678 - acc: 0.9786 - val_loss: 0.0729 - val_acc: 0.9774
Epoch 6/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0635 - acc: 0.9797 - val_loss: 0.0689 - val_acc: 0.9785
Epoch 7/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0577 - acc: 0.9813 - val_loss: 0.0646 - val_acc: 0.9783
Epoch 8/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0538 - acc: 0.9827 - val_loss: 0.0599 - val_acc: 0.9806
Epoch 9/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0505 - acc: 0.9836 - val_loss: 0.0570 - val_acc: 0.9817
Epoch 10/10
299779/299779 [==============================] - 39s 130us/step - loss: 0.0479 - acc: 0.9846 - val_loss: 0.0545 - val_acc: 0.9823
```

Training output

An accuracy of 98.2% is pretty impressive and will most likely result from the fact that most names in the validation set were already present in our test set. Using our validation set we can take a quick look at where our model comes to the wrong prediction:



	gender	first_name	first_firstname	predicted_gender
209900	f	Benk	Benk	m
119687	m	Lawrence Xaiver Liam	Lawrence	f
430987	m	Velisa	Velisa	f
21546	m	Osmanaj	Osmanaj	f
156743	m	Pilicic	Pilicic	f

Model validation output

Looking at the results, at least some of the false predictions seem to occur for people that typed in their family name into the first name field. Seeing that, a good next step seems to clean the original dataset from those cases. For now, the result looks pretty promising. With the accuracy we can achieve, this model could already be used in many real-world situations. Also, training the model for more epochs might increase its performance, important here is to looks out for the performance on the validation set to prevent a possible overfitting

Final thoughts

activation function, we can rely on rules of thumbs or can determine the right parameter based on our problem. However, in some other cases, the best result will come from testing various configurations and then evaluating the outcome.