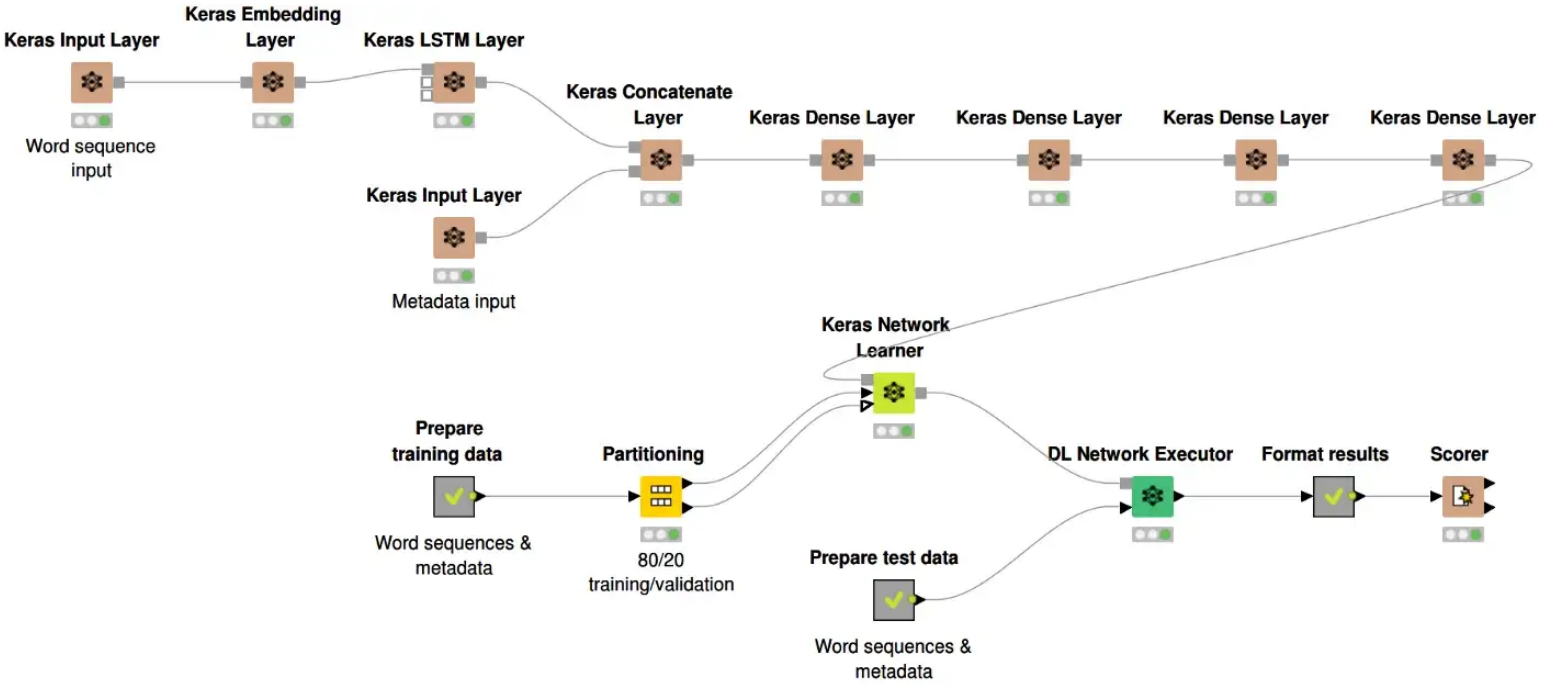
Kenny Hunt Follow

May 27, 2019 · 8 min read · ✨ · Listen

Predicting Customer Churn with Neural Networks in Keras



Why Predict Customer Churn?

This is a big one for organisations everywhere and one of the main areas in which we see high adoption rate of machine learning, this is probably down to the fact that we are **predicting customer behaviour.**

"Churn" is the term used to describe when a customer stops using a certain organisation's services. This is a really powerful KPI for subscription services, who make most of their income from repeating (usually monthly) payments.

Deep Learning Use Case — Netflix

Netflix is a great example of a subscription company, they're also an organisation that is close to the bleeding edge of tech. Netflix uses **Deep Learning techniques** to predict if a customer is going to leave before they actually do, this means they can take preventative action to ensure they stay.

How do they do it?

Without going too far down the rabbit hole, here goes...

Netflix collect a LOT of data on individuals, what you watch, when you watch it, everything you like and dislike etc etc. They can use this data in combination with **Deep Learning** classification techniques to work out when they think a customer will leave. The simple explanation could be something like "if a customer doesn't watch anything for N days then its realistic to expect they will churn soon".

Analysing all that harvested data using a Neural Network will enable the organisation to build profiles of their customers based on the data they have available. Once a bunch of users are categorised, Netflix can decide what action to take if a customer is highlighted a a suspected churn. e.g. they can understand which customers they want to keep — and offer these guys discounts and promotions — then also identify which customers are 'lost causes' and can be let go.

So, the experiment...

The IBM Telco Dataset has been doing the rounds on the internet for over a year now, so I guess now is as good time as any to have a crack at using it to predict customer churn.

I will need to borrow some code from elsewhere. I lifted the data prep code directly from this blog post. This allowed me to spend a bit more time tweaking the model and (attempting) to gain a bit more accuracy.

Importing the Necessary Libraries

This is pretty simple, its pretty much a straight copy/paste job into your own notebook. On scouring the internet I found a whole load of people distributing code that imported incorrect

libraries. The main thing to look out for here is the second block where I import the Keras modules.

Plenty of posts (I assume it all originated from the same post) have become confused between importing **Keras** libraries and **TensorFlow** libraries. A good rule of thumb here is to ensure that you are EITHER:

- Only importing Keras stuff: They will start with 'Keras.' e.g. from keras.models import…
- Only importing TensorFlow Keras libraries, these start with TensorFlow.Keras. e.g. from TensorFlow.Keras.models import…

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
import keras
from keras.models import Sequential
from keras.layers import InputLayer
from keras.layers import Dense
from keras.layers import Dropout
from keras.constraints import maxnorm
```

Read the test data in a pandas DataFrame (grab the CSV by clicking the link to the IBM dataset at the top).

```
data = pd.read_csv('churn.csv')
```

Data Preprocessing

Once we have the data in a Pandas DF we can use the stolen preprocessing code to get the data into a format that is optimised for the neural network. Essentially we are doing the following things:

- Converting categorical variables into numerical variables (e.g. Yes/No to 0/1).
- Getting the column formats right so they are all in numeric format
- Filling in any nulls.

```
data.SeniorCitizen.replace([0, 1], ["No", "Yes"], inplace= True)
data.TotalCharges.replace([" "], ["0"], inplace= True) data.TotalCharges =
data.TotalCharges.astype(float) data.drop("customerID", axis= 1, inplace=
True) data.Churn.replace(["Yes", "No"], [1, 0], inplace= True)
```

Once we have a clean dataset we can use pandas get_dummies functionality to replace all our categorical columns with 'dummy' numeric 'indicatior' columns. Similar to what the code above is doing, but this will strip out the whole data set.

```
data = pd.get_dummies(data)
```

Splitting the Data

Like any model, we should be splitting our data into a training and validation (or test set).

First we split our dataset down into X and y.

- X contains all the variables that we are using the make the predictions.
- y contains just the outcomes (whether or not the customer churned).

```
X = data.drop("Churn", axis= 1) y = data.Churn
```

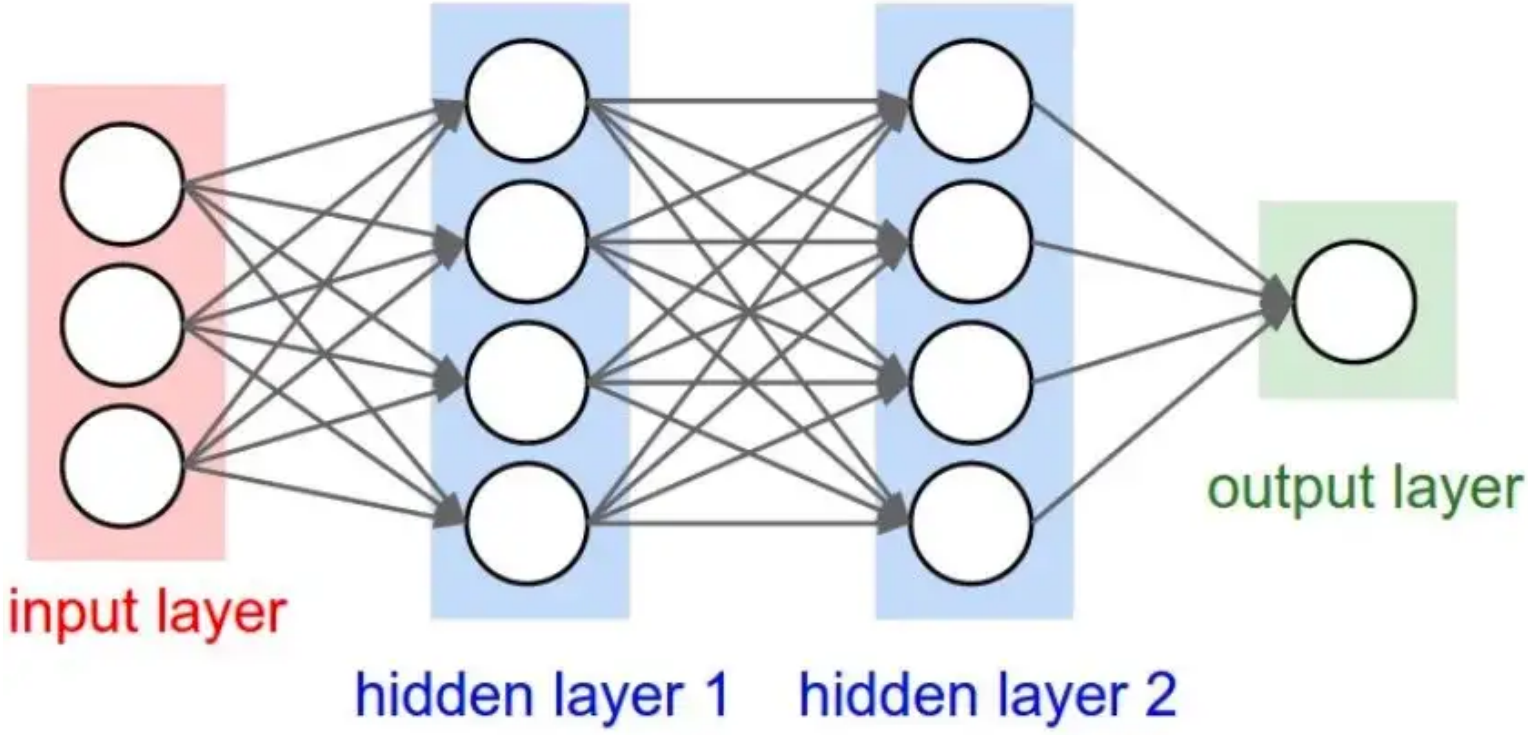
Next we use a standard train_test_split to split the data into training and testing (validation) sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2,
random_state= 1234)
```

Now we have our datasets at the ready, lets build a model.

Building the Model

One great thing about Keras is that we can very simply build a neural network based on layers. It looks a bit like this diagram.



First we tell Keras what type of model we want to employ. Mostly you'll be using sequential models.

```
model = Sequential()
```

Next lets first build the input layer. Its the red layer on the diagram. There are a few things to notice here.

We are using dense layers throughout this NN, I wont go into the differences between different layers and neural networks but most of the time you will be using either Dense or LSTM layers. More info here.

The first number — 16 is the number of nodes (or circles in the above diagram. We start with 16, we can change this later on to see how it impacts our accuracy.

Its important that you set the *input_dim* argument it must match the amount of columns in our X_train dataset. We can simply count the columns in the dataset and type it as I have done below — but realistically you will want to use X_train.shape[1] to automatically compute the value.

Activation Functions — you should really read up on these at some point — ill probably do another post specifically on these, but for the time being just know that our layers all use 'Relu'

except for the output layer (which we will get to later. As a general rule of thumb — *"if you are unsure, use relu"*

```
model.add(Dense(16, input_dim=46, activation='relu',
kernel_constraint=maxnorm(3)))
```

We add in a Dropout Layer. The dropout layer ensures that we remove a set percentage (in this case 0.2 or 20%) of the data each time we iterate through the neural network. This is optional but worth including in your code to stop it from overfitting. You can change the different drop out rates to experiment to try to get more accuracy.

```
model.add(Dropout(rate=0.2))
```

Now we add in what's called a hidden layer, we can have as many of these as we like, but its worth thinking about the additional computational power required to make NN's with a large number of hidden layers work.

Notice the 'kernal_constraint' argument? This deals with the scaling of the weights required to make the "dropout" layers function effectively. Again more info in the documentation, but I guess all you need to know is that by nature a NN will automatically trial and error all the different weightings of the variables in the dataset (this is why we use a neural net in the first place), the kernel_constraint argument adds control around this process.

```
model.add(Dense(8, activation='relu', kernel_constraint=maxnorm(3)))
```

Add in another Dropout layer to avoid overfitting.

```
model.add(Dropout(rate=0.2))
```

Lastly we add an output layer: this defines the final output from our neural network. There are a few things you need to keep in mind here:

- Our first argument is the number 1. This is because the out of our NN is one column containing an indicator which will specify whether our customer will churn or not.

- The activation function is different. For a single (yes/no) classification model we use ‘sigmoid’ there are many different functions available, for example if we are building a network that classifies multiple outcomes (for example grouping customers into specific groups) we may use a ‘softmax’ activation function.

```
model.add(Dense(1, activation='sigmoid'))
```

So now i’ve described each individual layer, ill show the whole lot squeezed together.

```
model = Sequential()
model.add(Dense(16, input_dim=46, activation='relu',
kernel_constraint=maxnorm(3)))
model.add(Dropout(rate=0.2))
model.add(Dense(8, activation='relu', kernel_constraint=maxnorm(3)))
model.add(Dropout(rate=0.2)) model.add(Dense(1, activation='sigmoid'))
```

Compiling the Model

Next we compile our model (stick it all together and tell it how it should work).

We use the compile method to achieve this. It takes three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (we use ‘Adam’), its possible to play around with these and see which is best, you can also split it out and manually adjust the learning rate to benefit from greater accuracy, but we will use the defaults. See: optimizers.
- A loss function. This is the objective that the model will try to minimize. As this is a single classification problem, we will use ‘binary_crossentropy’. See: losses.
- A list of metrics. For any classification problem we to set this to ‘accuracy’.

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

Fitting the Model

Quick one liner to fit the model, notice we fit BOTH the test and validation data sets into our model here, this way the Keras will tell us how we performed on both data sets at once.

There are a couple of arguments we need to look out for here.

- Epochs = this is the number of times we travel up and down the neural network — more epochs can mean more accuracy, but also more processing time, also too many epochs can lead to overfitting. You can adjust this anytime, so try some different values to see what happens to your results.
- batch_size = this is how many batches of records are put through the neural network at once. A smaller batch size is less accurate but quicker. Again, you can experiment by changing this number directly in the code.

```
history = model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=40, batch_size=10)
```

Model Scoring

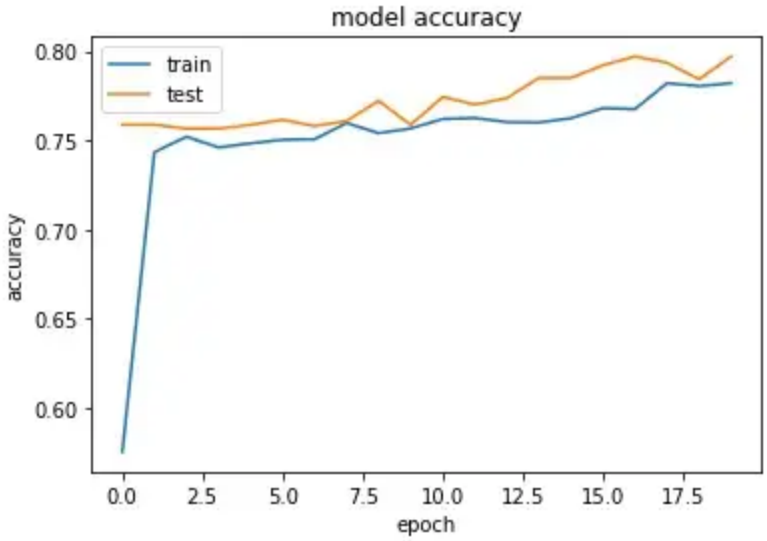
Once we run the above code, we will get an indication of how well the model performs — you will see it running through the epochs and providing accuracy scores for training (acc) and test (val_acc) sets.

```
5634/5634 [==============================] - 1s 217us/step - loss: 0.4575 - acc: 0.7902 - val_loss: 0.4516 - val_acc: 0.7970
Epoch 32/40
5634/5634 [==============================] - 1s 207us/step - loss: 0.4617 - acc: 0.7812 - val_loss: 0.4590 - val_acc: 0.7963
Epoch 33/40
5634/5634 [==============================] - 1s 210us/step - loss: 0.4671 - acc: 0.7781 - val_loss: 0.4687 - val_acc: 0.7935
Epoch 34/40
5634/5634 [==============================] - 2s 268us/step - loss: 0.4723 - acc: 0.7783 - val_loss: 0.4627 - val_acc: 0.7935
Epoch 35/40
5634/5634 [==============================] - 1s 248us/step - loss: 0.4578 - acc: 0.7875 - val_loss: 0.4654 - val_acc: 0.7963
Epoch 36/40
5634/5634 [==============================] - 1s 236us/step - loss: 0.4616 - acc: 0.7797 - val_loss: 0.4591 - val_acc: 0.7963
Epoch 37/40
5634/5634 [==============================] - 1s 239us/step - loss: 0.4580 - acc: 0.7827 - val_loss: 0.4783 - val_acc: 0.7970
Epoch 38/40
5634/5634 [==============================] - 1s 266us/step - loss: 0.4644 - acc: 0.7842 - val_loss: 0.4696 - val_acc: 0.7984
Epoch 39/40
5634/5634 [==============================] - 1s 245us/step - loss: 0.4697 - acc: 0.7835 - val_loss: 0.4710 - val_acc: 0.7906
Epoch 40/40
5634/5634 [==============================] - 2s 275us/step - loss: 0.4607 - acc: 0.7861 - val_loss: 0.4599 - val_acc: 0.7892
```

Once its done its thing, all that's left to do is visualise the results.

```
plt.plot(history.history['acc']) plt.plot(history.history['val_acc'])
plt.title('model accuracy') plt.ylabel('accuracy') plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper left') plt.show()
```



The last thing we need to do is save the model, this is so that we can deploy it to production later — AWS SageMaker / Azure / Google all have different ways to do this, but generally youll need both a JSON and a WEIGHTS file.

```
# serialize model to JSON
model_json = model.to_json() with open("model.json", "w") as json_file:
json_file.write(model_json)

# serialize weights to HDF5 model.save_weights("model.h5") print("Saved model
to disk")
```

So there we have it. One neural network that produced around 79% accuracy for customer churn. Of course we can spend more time playing around with learning rates, activation functions, number of nodes, number of epochs etc to try and make it more accurate, but hopefully this is a solid baseline to start your investigation.

Next ill look into how we can deploy this code to production — it gets tricky.

Originally published at http://drunkendatascience.com on May 27, 2019.

Machine Learning
