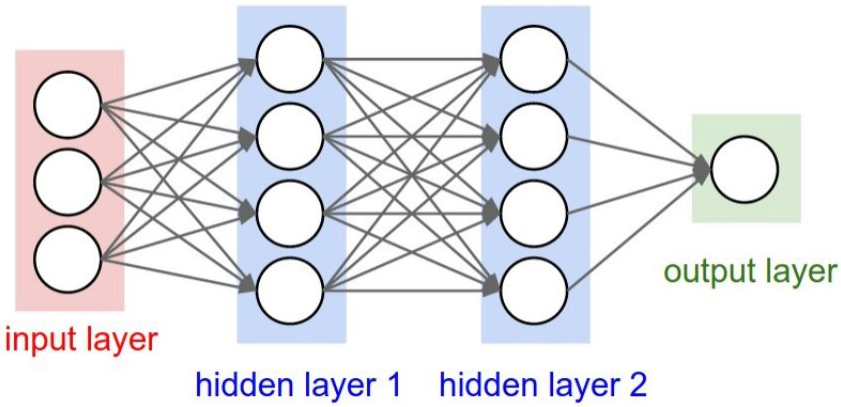
Eijaz Allibhai Follow
Machine Learning
Sep 17 · 9 min read

Building A Deep Learning Model using Keras



A Neural Network (image credit)

Deep learning is an increasingly popular subset of machine learning. Deep learning models are built using neural networks. A neural network takes in inputs, which are then processed in hidden layers using weights that are adjusted during training. Then the model spits out a prediction. The weights are adjusted to find patterns in order to make better predictions. The user does not need to specify what patterns to look for—the neural network learns on its own.

Keras is a user-friendly neural network library written in Python. In this tutorial, I will go over two deep learning models using Keras: one for regression and one for classification. We will build a regression model to predict an employee's wage per hour, and we will build a classification model to predict whether or not a patient has diabetes.

Note: The datasets we will be using are relatively clean, so we will not perform any data preprocessing in order to get our data ready for modeling. Datasets that you will use in future projects may not be so clean—for example, they may have missing values—so you may need to use data preprocessing techniques to alter your datasets to get more accurate results.

Reading in the training data

For our regression deep learning model, the first step is to read in the data we will use as input. For this example, we are using the 'hourly wages' dataset. To start, we will use Pandas to read in the data. I will not go into detail on Pandas, but it is a library you should become familiar with if you're looking to dive further into data science and machine learning.

'df' stands for dataframe. Pandas reads in the csv file as a dataframe. The 'head()' function will show the first 5 rows of the dataframe so you can check that the data has been read in properly and can take an initial look at how the data is structured.

```
Import pandas as pd
```

```
#read in data using pandas
train_df = pd.read_csv('data/hourly_wages_data.csv')
```

```
#check data has been read in properly
train_df.head()
```

	wage_per_hour	union	education_yrs	experience_yrs	age	female	marr	south	manufacturing	construction
0	5.10	0	8	21	35	1	1	0	1	0
1	4.95	0	9	42	57	1	1	0	1	0
2	6.67	0	12	1	19	0	0	0	1	0
3	4.00	0	12	4	22	0	0	0	0	0
4	7.50	0	12	17	35	0	1	0	0	0

Split up the dataset into inputs and targets

Next, we need to split up our dataset into inputs (train_X) and our target (train_y). Our input will be every column except 'wage_per_hour' because 'wage_per_hour' is what we will be attempting to predict. Therefore, 'wage_per_hour' will be our target.

We will use pandas 'drop' function to drop the column 'wage_per_hour' from our dataframe and store it in the variable 'train_X'. This will be our input.

```
#create a dataframe with all training data except the
target column
train_X = train_df.drop(columns=['wage_per_hour'])
```

```
#check that the target variable has been removed
train_X.head()
```

	union	education_yrs	experience_yrs	age	female	marr	south	manufacturing	construction
0	0	8	21	35	1	1	0	1	0
1	0	9	42	57	1	1	0	1	0
2	0	12	1	19	0	0	0	1	0
3	0	12	4	22	0	0	0	0	0
4	0	12	17	35	0	1	0	0	0

We will insert the column 'wage_per_hour' into our target variable (train_y).

```
#create a dataframe with only the target column
train_y = train_df[['wage_per_hour']]

#view dataframe
train_y.head()
```

	wage_per_hour
0	5.10
1	4.95
2	6.67
3	4.00
4	7.50

Building the model

Next, we have to build the model. Here is the code:

```
from keras.models import Sequential
from keras.layers import Dense
```

```
#create model
model = Sequential()

#get number of columns in training data
n_cols = train_X.shape[1]

#add model layers
model.add(Dense(10, activation='relu', input_shape=
(n_cols,)))
model.add(Dense(10, activation='relu'))
model.add(Dense(1))
```

The model type that we will be using is Sequential. Sequential is the easiest way to build a model in Keras. It allows you to build a model layer by layer. Each layer has weights that correspond to the layer the follows it.

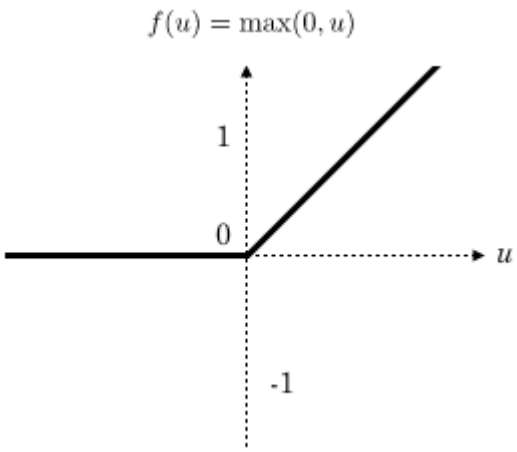
We use the 'add()' function to add layers to our model. We will add two layers and an output layer.

'Dense' is the layer type. Dense is a standard layer type that works for most cases. In a dense layer, all nodes in the previous layer connect to the nodes in the current layer.

We have 10 nodes in each of our input layers. This number can also be in the hundreds or thousands. Increasing the number of nodes in each layer increases model capacity. I will go into further detail about the effects of increasing model capacity shortly.

'Activation' is the activation function for the layer. An activation function allows models to take into account nonlinear relationships. For example, if you are predicting diabetes in patients, going from age 10 to 11 is different than going from age 60–61.

The activation function we will be using is ReLU or Rectified Linear Activation. Although it is two linear pieces, it has been proven to work well in neural networks.



ReLU Activation Function (image credit)

The first layer needs an input shape. The input shape specifies the number of rows and columns in the input. The number of columns in our input is stored in 'n_cols'. There is nothing after the comma which indicates that there can be any amount of rows.

The last layer is the output layer. It only has one node, which is for our prediction.

Compiling the model

Next, we need to compile our model. Compiling the model takes two parameters: optimizer and loss.

The optimizer controls the learning rate. We will be using 'adam' as our optmizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training.

The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

For our loss function, we will use 'mean_squared_error'. It is calculated by taking the average squared difference between the predicted and actual values. It is a popular loss function for regression problems. The closer to 0 this is, the better the model performed.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

where N is the number of data points,
f_i the value returned by the model and
y_i the actual value for data point i.

Mean Squared Error (image credit)

```
#compile model using mse as a measure of model
performance
model.compile(optimizer='adam',
loss='mean_squared_error')
```

Training the model

Now we will train our model. To train, we will use the 'fit()' function on our model with the following five parameters: training data (train_X), target data (train_y), validation split, the number of epochs and callbacks.

The validation split will randomly split the data into use for training and testing. During training, we will be able to see the validation loss, which give the mean squared error of our model on the validation set. We will set the validation split at 0.2, which means that 20% of the training data we provide in the model will be set aside for testing model performance.

The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that point, the model will stop improving during each epoch. In addition, the more epochs, the longer the model will take to run. To monitor this, we will use 'early stopping'.

Early stopping will stop the model from training before the number of epochs is reached if the model stops improving. We will set our early stopping monitor to 3. This means that after 3 epochs in a row in which the model doesn't improve, training will stop. Sometimes, the validation loss can stop improving then improve in the next epoch, but after 3 epochs in which the validation loss doesn't improve, it usually won't improve again.

```
from keras.callbacks import EarlyStopping
```

```
#set early stopping monitor so the model stops
training when it won't improve anymore
early_stopping_monitor = EarlyStopping(patience=3)
```

```
#train model
model.fit(train_X, train_y, validation_split=0.2,
epochs=30, callbacks=[early_stopping_monitor])
```

```
Train on 427 samples, validate on 107 samples
Epoch 1/30
427/427 [==============================] - 3s 7ms/step - loss: 200.7186 - val_loss: 243.7071
Epoch 2/30
427/427 [==============================] - 0s 89us/step - loss: 130.2738 - val_loss: 172.7188
Epoch 3/30
427/427 [==============================] - 0s 97us/step - loss: 86.9109 - val_loss: 123.9527
Epoch 4/30
427/427 [==============================] - 0s 97us/step - loss: 58.5707 - val_loss: 91.3879
Epoch 5/30
427/427 [==============================] - 0s 98us/step - loss: 41.0577 - val_loss: 68.7715
Epoch 6/30
427/427 [==============================] - 0s 97us/step - loss: 30.9307 - val_loss: 53.7192
Epoch 7/30
427/427 [==============================] - 0s 98us/step - loss: 25.0882 - val_loss: 44.5493
Epoch 8/30
427/427 [==============================] - 0s 97us/step - loss: 22.1689 - val_loss: 38.9246
Epoch 9/30
427/427 [==============================] - 0s 98us/step - loss: 20.8058 - val_loss: 35.9207
Epoch 10/30
427/427 [==============================] - 0s 99us/step - loss: 20.3621 - val_loss: 34.2123
Epoch 11/30
427/427 [==============================] - 0s 101us/step - loss: 20.1909 - val_loss: 33.4018
Epoch 12/30
427/427 [==============================] - 0s 94us/step - loss: 20.1781 - val_loss: 32.6290
Epoch 13/30
427/427 [==============================] - 0s 98us/step - loss: 20.1422 - val_loss: 32.8353
Epoch 14/30
427/427 [==============================] - 0s 91us/step - loss: 20.1181 - val_loss: 32.6664
Epoch 15/30
427/427 [==============================] - 0s 98us/step - loss: 20.0899 - val loss: 33.2220
```

Making predictions on new data

If you want to use this model to make predictions on new data, we would use the 'predict()' function, passing in our new data. The output would be 'wage_per_hour' predictions.

```
#example on how to use our newly trained model on how
to make predictions on unseen data (we will pretend
our new data is saved in a dataframe called 'test_X').
```

```
test_y_predictions = model.predict(test_X)
```

Congrats! You have built a deep learning model in Keras! It is not very accurate yet, but that can improve with using a larger amount of training data and 'model capacity'.

Model capacity

As you increase the number of nodes and layers in a model, the model capacity increases. Increasing model capacity can lead to a more accurate model, up to a certain point, at which the model will stop improving. Generally, the more training data you provide, the larger the model should be. We are only using a tiny amount of data, so our model is pretty small. The larger the model, the more computational capacity it requires and it will take longer to train.

Let's create a new model using the same training data as our previous model. This time, we will add a layer and increase the nodes in each layer to 200. We will train the model to see if increasing the model capacity will improve our validation score.

```
#training a new model on the same data to show the
effect of increasing model capacity

#create model
model_mc = Sequential()

#add model layers
model_mc.add(Dense(200, activation='relu',
input_shape=(n_cols,)))
model_mc.add(Dense(200, activation='relu'))
model_mc.add(Dense(200, activation='relu'))
model_mc.add(Dense(1))

#compile model using mse as a measure of model
performance
model_mc.compile(optimizer='adam',
loss='mean_squared_error')
```

```
#train model
model_mc.fit(train_X, train_y, validation_split=0.2,
epochs=30, callbacks=[early_stopping_monitor])
```

```
Train on 427 samples, validate on 107 samples
Epoch 1/30
427/427 [==============================] - 3s 7ms/step - loss: 39.2107 - val_loss: 40.3353
Epoch 2/30
427/427 [==============================] - 0s 226us/step - loss: 22.6596 - val_loss: 43.5961
Epoch 3/30
427/427 [==============================] - 0s 230us/step - loss: 21.1501 - val_loss: 35.9529
Epoch 4/30
427/427 [==============================] - 0s 222us/step - loss: 20.0189 - val_loss: 34.9528
Epoch 5/30
427/427 [==============================] - 0s 220us/step - loss: 19.5254 - val_loss: 33.4263
Epoch 6/30
427/427 [==============================] - 0s 221us/step - loss: 19.1911 - val_loss: 29.8092
Epoch 7/30
427/427 [==============================] - 0s 218us/step - loss: 18.8980 - val_loss: 28.0586
Epoch 8/30
427/427 [==============================] - 0s 210us/step - loss: 21.4672 - val_loss: 28.4561
Epoch 9/30
427/427 [==============================] - 0s 188us/step - loss: 22.2891 - val_loss: 28.5794
Epoch 10/30
427/427 [==============================] - 0s 189us/step - loss: 19.4059 - val_loss: 32.4649
```

We can see that by increasing our model capacity, we have improved our validation loss from 32.63 in our old model to 28.06 in our new model.

. . .

Classification model

Now let's move on to building our model for classification. Since many steps will be a repeat from the previous model, I will only go over new concepts.

For this next model, we are going to predict if patients have diabetes or not.

```
#read in training data
train_df_2 =
pd.read_csv('documents/data/diabetes_data.csv')

#view data structure
train_df_2.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age	diabetes
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
#create a dataframe with all training data except the
target column
train_X_2 = df_2.drop(columns=['diabetes'])

#check that the target variable has been removed
train_X_2.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

When separating the target column, we need to call the 'to_categorical()' function so that column will be 'one-hot encoded'. Currently, a patient with no diabetes is represented with a 0 in the diabetes column and a patient with diabetes is represented with a 1. With one-hot encoding, the integer will be removed and a binary variable is inputted for each category. In our case, we have two categories: no diabetes and diabetes. A patient with no diabetes will be represented by [1 0] and a patient with diabetes will be represented by [0 1].

```
from keras.utils import to_categorical
```

```
#one-hot encode target column
train_y_2 = to_categorical(df_2.diabetes)

#vcheck that target column has been converted
train_y_2[0:5]
```

```
array([[ 0.,  1.],
       [ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  0.],
       [ 0.,  1.]], dtype=float32)
```

```
#create model
model_2 = Sequential()

#get number of columns in training data
n_cols_2 = train_X_2.shape[1]

#add layers to model
model_2.add(Dense(250, activation='relu', input_shape=
```

```
(n_cols_2,)))
model_2.add(Dense(250, activation='relu'))
model_2.add(Dense(250, activation='relu'))
model_2.add(Dense(2, activation='softmax'))
```

The last layer of our model has 2 nodes—one for each option: the patient has diabetes or they don't.

The activation is 'softmax'. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities. The model will then make its prediction based on which option has a higher probability.

```
#compile model using accuracy to measure model
performance
model_2.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
```

We will use 'categorical_crossentropy' for our loss function. This is the most common choice for classification. A lower score indicates that the model is performing better.

To make things even easier to interpret, we will use the 'accuracy' metric to see the accuracy score on the validation set at the end of each epoch.

```
#train model
model_2.fit(X_2, target, epochs=30,
validation_split=0.2, callbacks=
[early_stopping_monitor])
```

```
Train on 614 samples, validate on 154 samples
Epoch 1/30
614/614 [==============================] - 0s 240us/step - loss: 0.6972 - acc: 0.6612 - val_loss: 0.7372 - val_acc:
0.6299
Epoch 2/30
614/614 [==============================] - 0s 265us/step - loss: 0.6634 - acc: 0.6547 - val_loss: 0.7887 - val_acc:
0.6688
Epoch 3/30
614/614 [==============================] - 0s 308us/step - loss: 0.6772 - acc: 0.6889 - val_loss: 0.6767 - val_acc:
0.6623
Epoch 4/30
614/614 [==============================] - 0s 267us/step - loss: 0.6745 - acc: 0.6938 - val_loss: 0.8132 - val_acc:
0.5455
Epoch 5/30
614/614 [==============================] - 0s 228us/step - loss: 0.6251 - acc: 0.6922 - val_loss: 0.6641 - val_acc:
0.6299
Epoch 6/30
614/614 [==============================] - 0s 235us/step - loss: 0.5660 - acc: 0.7085 - val_loss: 0.6837 - val_acc:
0.5779
Epoch 7/30
614/614 [==============================] - 0s 256us/step - loss: 0.6009 - acc: 0.6922 - val_loss: 0.5935 - val_acc:
0.7013
Epoch 8/30
614/614 [==============================] - 0s 238us/step - loss: 0.5647 - acc: 0.7166 - val_loss: 0.6208 - val_acc:
0.6753
Epoch 9/30
614/614 [==============================] - 0s 228us/step - loss: 0.5514 - acc: 0.7068 - val_loss: 0.6295 - val_acc:
0.6558
Epoch 10/30
614/614 [==============================] - 0s 229us/step - loss: 0.5920 - acc: 0.7068 - val_loss: 0.6410 - val_acc:
0.6558
```

Congrats! You are now well on your way to building amazing deep learning models in Keras!

Thanks for reading! The github repository for this tutorial can be found here.