

Deep_learning

May 26, 2019

1 Chapter 1 - Basics of deep learning and neural networks

1.1 In this chapter, you'll become familiar with the fundamental concepts and terminology used in deep learning, and understand why deep learning techniques are so powerful today. You'll build simple neural networks yourself and generate predictions with them.

1.1.1 Coding the forward propagation algorithm

In this exercise, you'll write code to do forward propagation (prediction) for your first neural network:

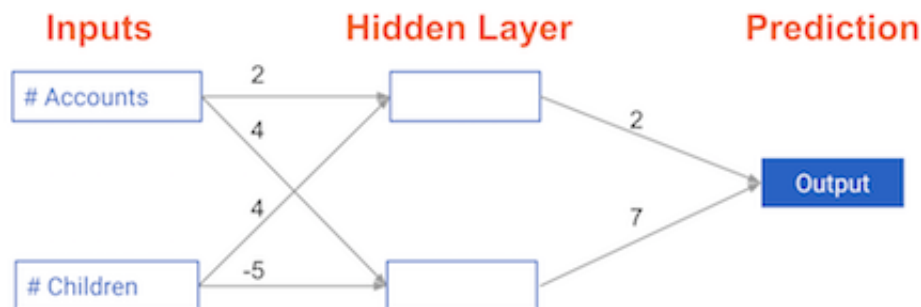
Each data point is a customer. The first input is how many accounts they have, and the second input is how many children they have. The model will predict how many transactions the user makes in the next year. You will use this data throughout the first 2 chapters of this course.

The input data has been pre-loaded as `input_data`, and the weights are available in a dictionary called `weights`. The array of weights for the first node in the hidden layer are in `weights['node_0']`, and the array of weights for the second node in the hidden layer are in `weights['node_1']`.

The weights feeding into the output node are available in `weights['output']`.

```
In [0]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
In [0]: input_data = np.array([3, 5])
```



title

```
In [0]: weights = {'node_0': np.array([2, 4]), 'node_1': np.array([ 4, -5]), 'output': np.array([ 4, -5])}
```

Instructions - Calculate the value in node 0 by multiplying input_data by its weights weights['node_0'] and computing their sum. This is the 1st node in the hidden layer. - Calculate the value in node 1 using input_data and weights['node_1']. This is the 2nd node in the hidden layer. - Put the hidden layer values into an array. This has been done for you. - Generate the prediction by multiplying hidden_layer_outputs by weights['output'] and computing their sum.

```
In [4]: # Calculate node 0 value: node_0_value
node_0_value = (input_data * weights['node_0']).sum()

# Calculate node 1 value: node_1_value
node_1_value = (input_data * weights['node_1']).sum()

# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_value, node_1_value])

# Calculate output: output
output = (hidden_layer_outputs * weights['output']).sum()

# Print output
output
```

```
Out [4]: -39
```

Wonderful work! It looks like the network generated a prediction of -39.

1.1.2 The Rectified Linear Activation Function

An “activation function” is a function applied at each node. It converts the node’s input into some output.

The rectified linear activation function (called ReLU) has been shown to lead to very high-performance networks. This function takes a single number as an input, returning 0 if the input is negative, and the input if the input is positive.

Here are some examples:

relu(3) = 3

relu(-3) = 0

```
In [5]: def relu(input):
        '''Define your relu activation function here'''
        # Calculate the value for the output of the relu function: output
        output = max(0, input)

        # Return the value just calculated
        return(output)

# Calculate node 0 value: node_0_output
node_0_input = (input_data * weights['node_0']).sum()
node_0_output = relu(node_0_input)
```

```

# Calculate node 1 value: node_1_output
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = relu(node_1_input)

# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_output, node_1_output])

# Calculate model output (do not apply relu)
model_output = (hidden_layer_outputs * weights['output']).sum()

# Print model output
model_output

```

Out [5]: 52

Great work! You predicted 52 transactions. Without this activation function, you would have predicted a negative number! The real power of activation functions will come soon when you start tuning model weights.

1.1.3 Applying the network to many observations/rows of data

You'll now define a function called `predict_with_network()` which will generate predictions for multiple data observations, which are pre-loaded as `input_data`. As before, weights are also pre-loaded. In addition, the `relu()` function you defined in the previous exercise has been pre-loaded.

```
In [0]: input_data = [np.array([3, 5]), np.array([ 1, -1]), np.array([0, 0]), np.array([8, 4])]
```

Instructions - Define a function called `predict_with_network()` that accepts two arguments - `input_data_row` and `weights` - and returns a prediction from the network as the output. - Calculate the input and output values for each node, storing them as: `node_0_input`, `node_0_output`, `node_1_input`, and `node_1_output`. - To calculate the input value of a node, multiply the relevant arrays together and compute their sum. - To calculate the output value of a node, apply the `relu()` function to the input value of the node. - Calculate the model output by calculating `input_to_final_layer` and `model_output` in the same way you calculated the input and output values for the nodes. - Use a for loop to iterate over `input_data`: - Use your `predict_with_network()` to generate predictions for each row of the `input_data` - `input_data_row`. Append each prediction to `results`.

```

In [7]: # Define predict_with_network()
def predict_with_network(input_data_row, weights):

    # Calculate node 0 value
    node_0_input = (input_data_row * weights['node_0']).sum()
    node_0_output = relu(node_0_input)

    # Calculate node 1 value
    node_1_input = (input_data_row * weights['node_1']).sum()

```

```

node_1_output = relu(node_1_input)

# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_output, node_1_output])

# Calculate model output
input_to_final_layer = (hidden_layer_outputs * weights['output']).sum()
model_output = relu(input_to_final_layer)

# Return model output
return(model_output)

# Create empty list to store prediction results
results = []
for input_data_row in input_data:
    # Append prediction to results
    results.append(predict_with_network(input_data_row, weights))

# Print results
results

```

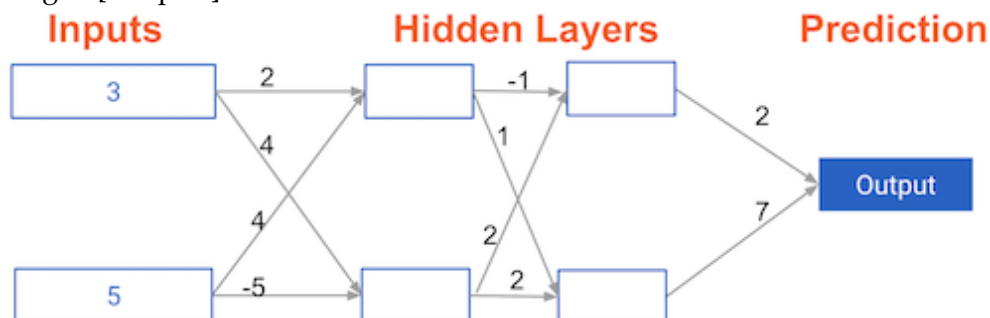
Out [7]: [52, 63, 0, 148]

1.1.4 Multi-layer neural networks

In this exercise, you'll write code to do forward propagation for a neural network with 2 hidden layers. Each hidden layer has two nodes. The input data has been preloaded as `input_data`. The nodes in the first hidden layer are called `node_0_0` and `node_0_1`. Their weights are pre-loaded as `weights['node_0_0']` and `weights['node_0_1']` respectively.

The nodes in the second hidden layer are called `node_1_0` and `node_1_1`. Their weights are pre-loaded as `weights['node_1_0']` and `weights['node_1_1']` respectively.

We then create a model output from the hidden nodes using weights pre-loaded as `weights['output']`.



```

In [0]: input_data = np.array([3, 5])
weights = {'node_0_0': np.array([2, 4]), 'node_0_1': np.array([4, -5]),
          'node_1_0': np.array([-1, 2]), 'node_1_1': np.array([1, 2]), 'output': np.array([2, 7])}

```

Instructions - Calculate node_0_0_input using its weights weights['node_0_0'] and the given input_data. Then apply the relu() function to get node_0_0_output. - Do the same as above for node_0_1_input to get node_0_1_output. - Calculate node_1_0_input using its weights weights['node_1_0'] and the outputs from the first hidden layer - hidden_0_outputs. Then apply the relu() function to get node_1_0_output. - Do the same as above for node_1_1_input to get node_1_1_output. - Calculate model_output using its weights weights['output'] and the outputs from the second hidden layer hidden_1_outputs array. Do not apply the relu() function to this output.

```
In [9]: def predict_with_network_1(input_data):
        # Calculate node 0 in the first hidden layer
        node_0_0_input = (input_data * weights['node_0_0']).sum()
        node_0_0_output = relu(node_0_0_input)

        # Calculate node 1 in the first hidden layer
        node_0_1_input = (input_data * weights['node_0_1']).sum()
        node_0_1_output = relu(node_0_1_input)

        # Put node values into array: hidden_0_outputs
        hidden_0_outputs = np.array([node_0_0_output, node_0_1_output])

        # Calculate node 0 in the second hidden layer
        node_1_0_input = (hidden_0_outputs * weights['node_1_0']).sum()
        node_1_0_output = relu(node_1_0_input)

        # Calculate node 1 in the second hidden layer
        node_1_1_input = (hidden_0_outputs * weights['node_1_1']).sum()
        node_1_1_output = relu(node_1_1_input)

        # Put node values into array: hidden_1_outputs
        hidden_1_outputs = np.array([node_1_0_output, node_1_1_output])

        # Calculate model output: model_output
        model_output = (hidden_1_outputs * weights['output']).sum()

        # Return model_output
        return(model_output)

output = predict_with_network_1(input_data)
output
```

Out [9]: 182

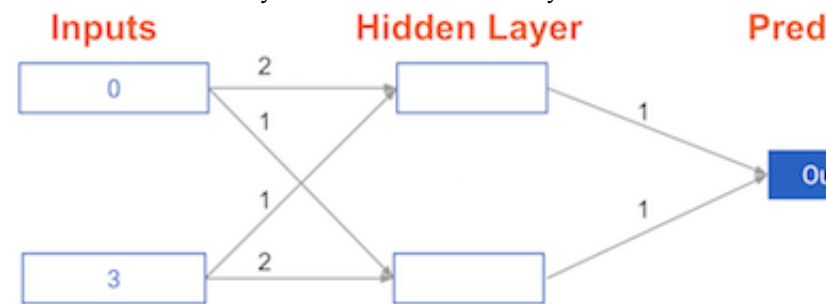
Wonderful work! The network generated a prediction of 182.

2 Chapter 2 - Optimizing a neural network with backward propagation

2.1 Here, you'll learn how to optimize the predictions generated by your neural networks. You'll do this using a method called backward propagation, which is one of the most important techniques in deep learning. Understanding how it works will give you a strong foundation to build from in the second half of the course.

2.1.1 Coding how weight changes affect accuracy

Now you'll get to change weights in a real network and see how they affect model accuracy!



Have a look at the following neural network:

Its weights have been pre-loaded as `weights_0`. Your task in this exercise is to update a single weight in `weights_0` to create `weights_1`, which gives a perfect prediction (in which the predicted value is equal to `target_actual`: 3).

Use a pen and paper if necessary to experiment with different combinations. You'll use the `predict_with_network()` function, which takes an array of data as the first argument, and weights as the second argument.

```
In [10]: # The data point you will make a prediction for
input_data = np.array([0, 3])

# Sample weights
weights_0 = {'node_0': [2, 1],
             'node_1': [1, 2],
             'output': [1, 1]
            }

# The actual target value, used to calculate the error
target_actual = 3

# Make prediction using original weights
model_output_0 = predict_with_network(input_data, weights_0)

# Calculate error: error_0
error_0 = model_output_0 - target_actual

# Create weights that cause the network to make perfect prediction (3): weights_1
weights_1 = {'node_0': [2, 1],
             'node_1': [1, 0],
             'output': [1, 1]
            }
```

```

# Make prediction using new weights: model_output_1
model_output_1 = predict_with_network(input_data, weights_1)

# Calculate error: error_1
error_1 = model_output_1 - target_actual

# Print error_0 and error_1
print(error_0)
print(error_1)

```

6
0

Fantastic! The network now generates a perfect prediction with an error of 0.

2.1.2 Scaling up to multiple data points

You've seen how different weights will have different accuracies on a single prediction. But usually, you'll want to measure model accuracy on many points. You'll now write code to compare model accuracies for two different sets of weights, which have been stored as `weights_0` and `weights_1`.

`input_data` is a list of arrays. Each item in that list contains the data to make a single prediction. `target_actuals` is a list of numbers. Each item in that list is the actual value we are trying to predict.

In this exercise, you'll use the `mean_squared_error()` function from `sklearn.metrics`. It takes the true values and the predicted values as arguments.

You'll also use the preloaded `predict_with_network()` function, which takes an array of data as the first argument, and weights as the second argument.

```

In [0]: weights_0 = {'node_0': np.array([2, 1]), 'node_1': np.array([1, 2]), 'output': np.array([1, 2])}
weights_1 = {'node_0': np.array([2, 1]), 'node_1': np.array([1, 1.5]), 'output': np.array([1, 1.5])}
input_data = [np.array([0, 3]), np.array([1, 2]), np.array([-1, -2]), np.array([4, 0])]
target_actuals = [1, 3, 5, 7]

```

Instructions - Import `mean_squared_error` from `sklearn.metrics`. - Using a for loop to iterate over each row of `input_data`: - Make predictions for each row with `weights_0` using the `predict_with_network()` function and append it to `model_output_0`. - Do the same for `weights_1`, appending the predictions to `model_output_1`. - Calculate the mean squared error of `model_output_0` and then `model_output_1` using the `mean_squared_error()` function. The first argument should be the actual values (`target_actuals`), and the second argument should be the predicted values (`model_output_0` or `model_output_1`).

```

In [12]: from sklearn.metrics import mean_squared_error

```

```

# Create model_output_0
model_output_0 = []
# Create model_output_1
model_output_1 = []

```

```

# Loop over input_data
for row in input_data:
    # Append prediction to model_output_0
    model_output_0.append(predict_with_network(row, weights_0))

    # Append prediction to model_output_1
    model_output_1.append(predict_with_network(row, weights_1))

# Calculate the mean squared error for model_output_0: mse_0
mse_0 = mean_squared_error(target_actuals, model_output_0)

# Calculate the mean squared error for model_output_1: mse_1
mse_1 = mean_squared_error(target_actuals, model_output_1)

# Print mse_0 and mse_1
print("Mean squared error with weights_0: %f" %mse_0)
print("Mean squared error with weights_1: %f" %mse_1)

```

Mean squared error with weights_0: 37.500000

Mean squared error with weights_1: 49.890625

Excellent work! It looks like model_output_1 has a higher mean squared error.

2.1.3 Calculating slopes

You're now going to practice calculating slopes. When plotting the mean-squared error loss function against predictions, the slope is $2 * x * (y - xb)$, or $2 * \text{input_data} * \text{error}$. Note that x and b may have multiple numbers (x is a vector for each data point, and b is a vector). In this case, the output will also be a vector, which is exactly what you want.

You're ready to write the code to calculate this slope while using a single data point. You'll use pre-defined weights called `weights` as well as data for a single point called `input_data`. The actual value of the target you want to predict is stored in `target`

```

In [0]: input_data = np.array([1, 2, 3])
        weights = np.array([0, 2, 1])
        target = 0

```

Instructions - Calculate the predictions, `preds`, by multiplying `weights` by the `input_data` and computing their sum. - Calculate the error, which is `target` minus `preds`. Notice that this error corresponds to $y - xb$ in the gradient expression. - Calculate the slope of the loss function with respect to the prediction. To do this, you need to take the product of `input_data` and error and multiply that by 2.

```

In [14]: # Calculate the predictions: preds
        preds = (weights * input_data).sum()

        # Calculate the error: error

```



```

error = target - preds

# Calculate the slope: slope
slope = 2 * input_data * error

# Print the slope
slope

```

```
Out[14]: array([-14, -28, -42])
```

Well done! You can now use this slope to improve the weights of the model!

2.1.4 Improving model weights

Hurray! You've just calculated the slopes you need. Now it's time to use those slopes to improve your model. If you add the slopes to your weights, you will move in the right direction. However, it's possible to move too far in that direction. So you will want to take a small step in that direction first, using a lower learning rate, and verify that the model is improving.

Instructions - Set the learning rate to be 0.01 and calculate the error from the original predictions. This has been done for you. - Calculate the updated weights by subtracting the product of learning_rate and slope from weights. - Calculate the updated predictions by multiplying weights_updated with input_data and computing their sum. - Calculate the error for the new predictions. Store the result as error_updated.

```

In [15]: # Set the learning rate: learning_rate
learning_rate = 0.01

# Calculate the predictions: preds
preds = (weights * input_data).sum()

# Calculate the error: error
error = preds - target

# Calculate the slope: slope
slope = 2 * input_data * error

# Update the weights: weights_updated
weights_updated = weights - learning_rate * slope

# Get updated predictions: preds_updated
preds_updated = (weights_updated * input_data).sum()

# Calculate updated error: error_updated
error_updated = preds_updated - target

# Print the original error
print(error)

```

```
# Print the updated error
print(error_updated)
```

```
7
5.04
```

Fantastic! Updating the model weights did indeed decrease the error!

2.1.5 Making multiple updates to weights

You're now going to make multiple updates so you can dramatically improve your model weights, and see how the predictions improve with each update.

To keep your code clean, there is a pre-loaded `get_slope()` function that takes `input_data`, `target`, and `weights` as arguments. There is also a `get_mse()` function that takes the same arguments. The `input_data`, `target`, and `weights` have been pre-loaded.

This network does not have any hidden layers, and it goes directly from the input (with 3 nodes) to an output node. Note that `weights` is a single array.

```
In [0]: def get_error(input_data, target, weights):
        preds = (weights * input_data).sum()
        error = preds - target
        return(error)

In [0]: def get_slope(input_data, target, weights):
        error = get_error(input_data, target, weights)
        slope = 2 * input_data * error
        return(slope)

In [0]: def get_mse(input_data, target, weights):
        errors = get_error(input_data, target, weights)
        mse = np.mean(errors**2)
        return(mse)
```

Instructions - Using a for loop to iteratively update weights: - Calculate the slope using the `get_slope()` function. - Update the weights using a learning rate of 0.01. - Calculate the mean squared error (mse) with the updated weights using the `get_mse()` function. - Append mse to `mse_hist`.

```
In [19]: n_updates = 20
        mse_hist = []

        # Iterate over the number of updates
        for i in range(n_updates):
            # Calculate the slope: slope
            slope = get_slope(input_data, target, weights)

            # Update the weights: weights
            weights = weights - 0.01 * slope
```

```

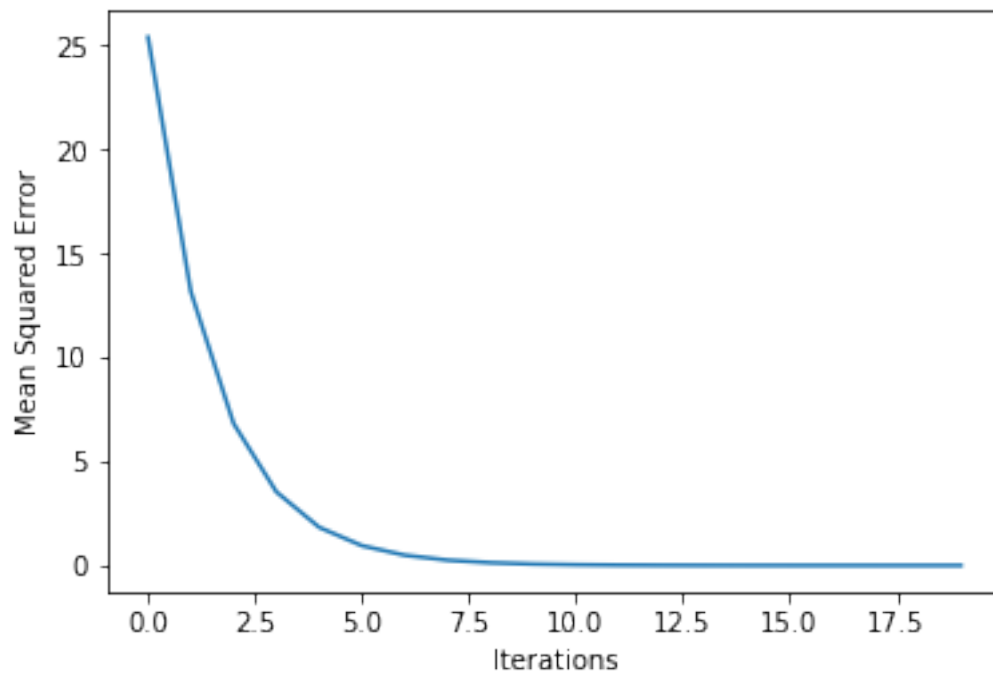
# Calculate mse with new weights: mse
mse = get_mse(input_data, target, weights)

# Append the mse to mse_hist
mse_hist.append(mse)

# Plot the mse history
plt.plot(mse_hist)
plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error')

```

Out[19]: Text(0, 0.5, 'Mean Squared Error')



Great work! As you can see, the mean squared error decreases as the number of iterations go up.

2.2 Backpropagation

You've use gradient descent to optimize weights in a simple model. Now we'll add a technique called "back propagation" to calculate the slopes you need to optimize more complex deep learning models.

Just as forward propagation sends input data through the hidden layers and into the output layer , back propagation takes the error from the output layer and propagates it backward through the hidden layers, towards the input layer.

It calculates the necessary slopes sequentially from the weights closest to the prediction, through

the hidden layers, eventually back to the weights coming from the input. We then use these slopes to update our weights as you've seen.

2.3 Backpropagation process

- We are trying to estimate the slope of the loss function w.r.t each weight in our network
- We always do forward propagation to make a prediction and calculate an error before we do back propagation
- We go back one layer at a time and each time we go back a layer, we'll use a formula for slopes that we saw earlier
- Every weight feed from some input node into some output node

The three things we multiply to get the slope for the weights are

Gradients for weight is product of: 1. Node value feeding into the weight (Either it's an input layer, in which case we have it from the data, or that node is in a hidden layer, in which case we calculated its value when we did forward propagation) 2. Slope of loss function w.r.t node it feeds into (We do backward propagation from the right side of our diagram to the left. So we already calculated that slope by the time we need to plug it into the current calculation) 3. Slope of the activation function at the node it feeds into (for ReLU Activation Function, the slope is 0 if the input to the node is negative, if the input to the node is positive, the output is the same as the input so the slope would be 1)

We also keep track of the slopes of the loss function with respect to node values because we use those slopes in our calculation of slopes at weights

The slope of the loss function w.r.t. any node value is the sum of the slopes for every weight coming into that node.

Each time you generate predictions using forward propagation, you update the weights using backward propagation.

3 Chapter 3 - Building deep learning models with keras

3.1 In this chapter, you'll use the keras library to build deep learning models for both regression as well as classification! You'll learn about the Specify-Compile-Fit workflow that you can use to make predictions and by the end of this chapter, you'll have all the tools necessary to build deep neural networks!

3.1.1 Specifying a model

Now you'll get to work with your first model in Keras, and will immediately be able to run more complex neural network models on larger datasets compared to the first two chapters.

To start, you'll take the skeleton of a neural network and add a hidden layer and an output layer. You'll then fit that model and see Keras do the optimization so your model continually gets better.

As a start, you'll predict workers wages based on characteristics like their industry, education and level of experience. You can find the dataset in a pandas dataframe called `df`. For convenience, everything in `df` except for the target has been converted to a NumPy matrix called `predictors`. The target, `wage_per_hour`, is available as a NumPy matrix called `target`.

```
In [20]: # Load the Drive helper and mount
        from google.colab import drive

        # This will prompt for authorization
        drive.mount('/content/drive')
```

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-

Enter your authorization code:

ûûûûûûûûûûûû

Mounted at /content/drive

```
In [0]: df = pd.read_csv('drive/My Drive/Colab Notebooks/Datasets/hourly_wages.csv')
```

```
In [22]: df.head()
```

```
Out[22]:
```

	wage_per_hour	union	education_yrs	...	south	manufacturing	construction
0	5.10	0	8	...	0	1	0
1	4.95	0	9	...	0	1	0
2	6.67	0	12	...	0	1	0
3	4.00	0	12	...	0	0	0
4	7.50	0	12	...	0	0	0

[5 rows x 10 columns]

```
In [0]: predictors = df.iloc[:, 1:].values
```

```
In [24]: predictors
```

```
Out[24]: array([[ 0,  8, 21, ...,  0,  1,  0],
                [ 0,  9, 42, ...,  0,  1,  0],
                [ 0, 12,  1, ...,  0,  1,  0],
                ...,
                [ 1, 17, 25, ...,  0,  0,  0],
                [ 1, 12, 13, ...,  1,  0,  0],
                [ 0, 16, 33, ...,  0,  1,  0]])
```

```
In [0]: target = df['wage_per_hour'].values
```

Instructions - Store the number of columns in the predictors data to `n_cols`. - Start by creating a Sequential model called `model`. - Use the `.add()` method on `model` to add a Dense layer. - Add 50 units, specify `activation='relu'`, and the `input_shape` parameter to be the tuple `(n_cols,)` which means it has `n_cols` items in each row of data, and any number of rows of data are acceptable as inputs. - Add another Dense layer. This should have 32 units and a `'relu'` activation. - Finally, add an output layer, which is a Dense layer with a single node. Don't use any activation function here.

```
In [26]: predictors.shape[1]
```

```
Out[26]: 9
```

```

In [0]: import warnings
        warnings.filterwarnings('ignore')

In [0]: # Import necessary modules
        import keras
        from keras.layers import Dense
        from keras.models import Sequential

        # Save the number of columns in predictors: n_cols
        n_cols = predictors.shape[1]

        # Set up the model: model
        model = Sequential()

        # Add the first layer
        model.add(Dense(50, activation='relu', input_shape=(n_cols,)))

        # Add the second layer
        model.add(Dense(32, activation='relu'))

        # Add the output layer
        model.add(Dense(1))

```

Well done! Now that you've specified the model, the next step is to compile it.

3.1.2 Compiling the model

You're now going to compile the model you specified earlier. To compile the model, you need to specify the optimizer and loss function to use. Adam optimizer is an excellent choice. You can read more about it as well as other keras optimizers [here](#), and if you are really curious to learn more, you can read the [original paper](#) that introduced the Adam optimizer.

In this exercise, you'll use the Adam optimizer and the mean squared error loss function. Go for it!

```

In [30]: # Compile the model
        model.compile(optimizer = 'adam', loss = 'mean_squared_error')

        # Verify that model contains information from compiling
        print("Loss function: " + model.loss)

```

Loss function: mean_squared_error

Fantastic work - all that's left now is to fit the model!

3.1.3 Fitting the model

You're at the most fun part. You'll now fit the model. Recall that the data to be used as predictive features is loaded in a NumPy matrix called predictors and the data to be predicted is stored in

a NumPy matrix called target. Your model is pre-written and it has been compiled with the code from the previous exercise.

```
In [32]: # Fit the model
```

```
model.fit(predictors, target)
```

```
Epoch 1/1
```

```
534/534 [=====] - 0s 54us/step - loss: 27.2594
```

```
Out[32]: <keras.callbacks.History at 0x7f91457437b8>
```

Superb! You now know how to specify, compile, and fit a deep learning model using keras!

3.1.4 Last steps in classification models

You'll now create a classification model using the titanic dataset, which has been pre-loaded into a DataFrame called df. You'll take information about the passengers and predict which ones survived.

The predictive variables are stored in a NumPy array predictors. The target to predict is in df.survived, though you'll have to manipulate it for keras. The number of predictive features is stored in n_cols.

Here, you'll use the 'sgd' optimizer, which stands for [Stochastic Gradient Descent](#). You'll learn more about this in the next chapter!

```
In [0]: df = pd.read_csv('drive/My Drive/Colab Notebooks/Datasets/titanic_all_numeric.csv')
```

```
In [34]: df.head()
```

```
Out[34]:
```

	survived	pclass	...	embarked_from_queenstown	embarked_from_southampton
0	0	3	...	0	1
1	1	1	...	0	0
2	1	3	...	0	1
3	1	1	...	0	1
4	0	3	...	0	1

```
[5 rows x 11 columns]
```

```
In [0]: df['age_was_missing'] = df['age_was_missing'].map({False:0, True:1})
```

```
In [0]: predictors = df.iloc[:, 1:].values
```

Instructions - Convert df.survived to a categorical variable using the to_categorical() function. - Specify a Sequential model called model. - Add a Dense layer with 32 nodes. Use 'relu' as the activation and (n_cols,) as the input_shape. - Add the Dense output layer. Because there are two outcomes, it should have 2 units, and because it is a classification model, the activation should be 'softmax'. - Compile the model, using 'sgd' as the optimizer, 'categorical_crossentropy' as the loss function, and metrics=['accuracy'] to see the accuracy (what fraction of predictions were correct) at the end of each epoch. - Fit the model using the predictors and the target.

```

In [37]: # Import necessary modules
         from keras.utils import to_categorical

         n_cols = predictors.shape[1]
         # Convert the target to categorical: target
         target = to_categorical(df.survived)

         # Set up the model
         model = Sequential()

         # Add the first layer
         model.add(Dense(32, activation = 'relu', input_shape = (n_cols,)))

         # Add the output layer
         model.add(Dense(2, activation = 'softmax'))

         # Compile the model
         model.compile(optimizer = 'sgd', loss = 'categorical_crossentropy', metrics = ['accuracy'])

         # Fit the model
         model.fit(predictors, target)

```

Epoch 1/1

891/891 [=====] - 0s 412us/step - loss: 1.6463 - acc: 0.6207

Out[37]: <keras.callbacks.History at 0x7f912d7e5c18>

3.1.5 Making predictions

The trained network from your previous coding exercise is now stored as model. New data to make predictions is stored in a NumPy array as pred_data. Use model to make predictions on your new data.

In this exercise, your predictions will be probabilities, which is the most common way for data scientists to communicate their predictions to colleagues.

```

In [0]: pred_data = np.array([[2, 34.0, 0, 0, 13.0, 1, False, 0, 0, 1], [2, 31.0, 1, 1, 26.25,

```

```

In [39]: # Calculate predictions: predictions
         predictions = model.predict(pred_data)

         # Calculate predicted probability of survival: predicted_prob_true
         predicted_prob_true = predictions[:, 1]

         # print predicted_prob_true
         print(predicted_prob_true)

```

```

[0.82276833 0.84816533 0.9991417  0.59330416 0.7419749  0.75518143
 0.5199844  0.74291867 0.8309923  0.9273992  0.7652387  0.8987095]

```



```

0.78583384 0.67362523 0.76239204 0.5283056 0.77008075 0.84092385
0.6871641 0.61284035 0.98672444 0.78126043 0.5199046 0.7776995
0.8355301 0.7316217 0.9491697 0.8808788 0.7513136 0.97892535
0.7493344 0.7006066 0.7378051 0.7604266 0.7359487 0.9768962
0.7542848 0.7726129 0.94305193 0.8728581 0.7448284 0.7337308
0.8707813 0.69730216 0.7339536 0.7221223 0.8377984 0.7372722
0.83840036 0.9914608 0.6810503 0.68095756 0.6079991 0.86846834
0.902855 0.71262884 0.999864 0.9155568 0.7395357 0.7378051
0.82252276 0.7312378 0.9135646 0.8446171 0.7868675 0.8352166
0.7794833 0.92842597 0.8060939 0.535645 0.7654304 0.9380063
0.5954908 0.7475986 0.84077406 0.7230199 0.7570002 0.73918307
0.76878947 0.9809574 0.83323884 0.7550383 0.7784348 0.7840982
0.74690694 0.7944439 0.79749215 0.8361225 0.8355053 0.83047163
0.7351059 ]

```

4 Chapter 4 - Fine-tuning keras models

4.1 Here, you'll learn how to optimize your deep learning models in keras. You'll learn how to validate your models, understand the concept of model capacity, and experiment with wider and deeper networks. Enjoy!

4.1.1 Changing optimization parameters

It's time to get your hands dirty with optimization. You'll now try optimizing a model at a very low learning rate, a very high learning rate, and a "just right" learning rate. You'll want to look at the results after running this exercise, remembering that a low value for the loss function is good.

For these exercises, we've pre-loaded the predictors and target values from your previous classification models (predicting who would survive on the Titanic). You'll want the optimization to start from scratch every time you change the learning rate, to give a fair comparison of how each learning rate did in your results. So we have created a function `get_new_model()` that creates an unoptimized model to optimize.

```
In [0]: input_shape = (predictors.shape[1],)
```

```
In [41]: input_shape
```

```
Out[41]: (10,)
```

```
In [0]: def get_new_model(input_shape = input_shape):
        model = Sequential()
        model.add(Dense(100, activation='relu', input_shape = input_shape))
        model.add(Dense(100, activation='relu'))
        model.add(Dense(2, activation='softmax'))
        return(model)
```

```
In [43]: # Import the SGD optimizer
        from keras.optimizers import SGD
```

```

# Create list of learning rates: lr_to_test
lr_to_test = [.000001, 0.01, 1]

# Loop over learning rates
for lr in lr_to_test:
    print('\n\nTesting model with learning rate: %f\n'%lr )

    # Build new model to test, unaffected by previous models
    model = get_new_model()

    # Create SGD optimizer with specified learning rate: my_optimizer
    my_optimizer = SGD(lr=lr)

    # Compile the model
    model.compile(optimizer = my_optimizer, loss = 'categorical_crossentropy')

    # Fit the model
    model.fit(predictors, target)

```

Testing model with learning rate: 0.000001

Epoch 1/1

891/891 [=====] - 0s 256us/step - loss: 5.2325

Testing model with learning rate: 0.010000

Epoch 1/1

891/891 [=====] - 0s 292us/step - loss: 2.1502

Testing model with learning rate: 1.000000

Epoch 1/1

891/891 [=====] - 0s 336us/step - loss: 6.0119

4.1.2 Evaluating model accuracy on validation dataset

Now it's your turn to monitor model accuracy with a validation data set. A model definition has been provided as model. Your job is to add the code to compile it and then fit it. You'll check the validation score in each epoch.

```

In [44]: # Save the number of columns in predictors: n_cols
         n_cols = predictors.shape[1]

```

```

input_shape = (n_cols,)

# Specify the model
model = Sequential()
model.add(Dense(100, activation='relu', input_shape = input_shape))
model.add(Dense(100, activation='relu'))
model.add(Dense(2, activation='softmax'))

# Compile the model
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Fit the model
hist = model.fit(predictors, target, validation_split = 0.3)

```

Train on 623 samples, validate on 268 samples

Epoch 1/1

623/623 [=====] - 1s 833us/step - loss: 0.7136 - acc: 0.6148 - val_loss: 0.7136

4.1.3 Early stopping: Optimizing the optimization

Now that you know how to monitor your model performance throughout optimization, you can use early stopping to stop optimization when it isn't helping any more. Since the optimization stops automatically when it isn't helping, you can also set a high value for epochs in your call to `.fit()`

The model you'll optimize has been specified as `model`. As before, the data is pre-loaded as `predictors` and `target`

```

In [45]: # Import EarlyStopping
from keras.callbacks import EarlyStopping

# Save the number of columns in predictors: n_cols
n_cols = predictors.shape[1]
input_shape = (n_cols,)

# Specify the model
model = Sequential()
model.add(Dense(100, activation='relu', input_shape = input_shape))
model.add(Dense(100, activation='relu'))
model.add(Dense(2, activation='softmax'))

# Compile the model
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])

# Define early_stopping_monitor
early_stopping_monitor = EarlyStopping(patience = 2)

# Fit the model
model.fit(predictors, target, epochs = 30, validation_split = 0.3, callbacks = [early_stopping_monitor])

```

Train on 623 samples, validate on 268 samples

Epoch 1/30

623/623 [=====] - 1s 926us/step - loss: 0.7834 - acc: 0.5795 - val_loss:

Epoch 2/30

623/623 [=====] - 0s 66us/step - loss: 0.6664 - acc: 0.6581 - val_loss:

Epoch 3/30

623/623 [=====] - 0s 66us/step - loss: 0.6239 - acc: 0.6565 - val_loss:

Epoch 4/30

623/623 [=====] - 0s 60us/step - loss: 0.6643 - acc: 0.6806 - val_loss:

Epoch 5/30

623/623 [=====] - 0s 64us/step - loss: 0.6159 - acc: 0.7030 - val_loss:

Epoch 6/30

623/623 [=====] - 0s 62us/step - loss: 0.6533 - acc: 0.6549 - val_loss:

Epoch 7/30

623/623 [=====] - 0s 59us/step - loss: 0.6344 - acc: 0.7014 - val_loss:

Out [45]: <keras.callbacks.History at 0x7f912c9f4ac8>

Wonderful work! Because optimization will automatically stop when it is no longer helpful, it is okay to specify the maximum number of epochs as 30 rather than using the default of 10 that you've used so far. Here, it seems like the optimization stopped after 7 epochs.

4.1.4 Experimenting with wider networks

Now you know everything you need to begin experimenting with different models!

A model called `model_1` has been pre-loaded. You can see a summary of this model printed in the IPython Shell. This is a relatively small network, with only 10 units in each hidden layer.

In this exercise you'll create a new model called `model_2` which is similar to `model_1`, except it has 100 units in each hidden layer.

After you create `model_2`, both models will be fitted, and a graph showing both models loss score at each epoch will be shown. We added the argument `verbose=False` in the fitting commands to print out fewer updates, since you will look at these graphically instead of as text.

Because you are fitting two models, it will take a moment to see the outputs after you hit run, so be patient

```
In [0]: model_1 = Sequential()
        model_1.add(Dense(10, activation='relu', input_shape = input_shape))
        model_1.add(Dense(10, activation='relu'))
        model_1.add(Dense(2, activation='softmax'))
        model_1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
In [63]: # Define early_stopping_monitor
        early_stopping_monitor = EarlyStopping(patience=2)
```

```
        # Create the new model: model_2
        model_2 = Sequential()
```

```
        # Add the first and second layers
```

```

model_2.add(Dense(100, activation='relu', input_shape=input_shape))
model_2.add(Dense(100, activation='relu'))

# Add the output layer
model_2.add(Dense(2, activation='softmax'))

# Compile model_2
model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

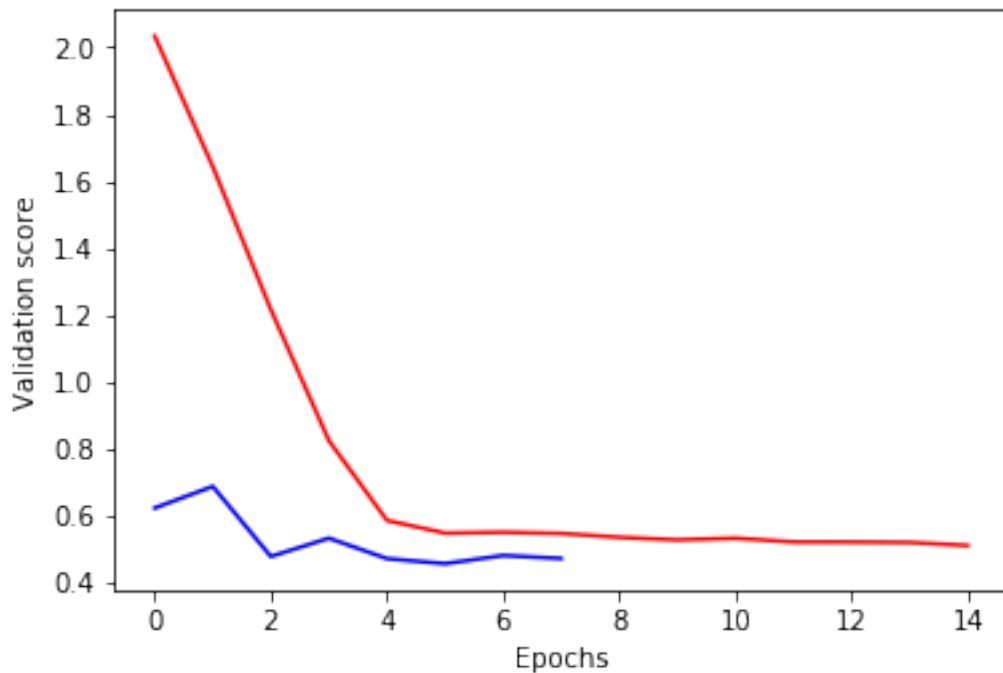
# Fit model_1
model_1_training = model_1.fit(predictors, target, epochs=15, validation_split=0.2, callbacks=[early_stopping])

# Fit model_2
model_2_training = model_2.fit(predictors, target, epochs=15, validation_split=0.2, callbacks=[early_stopping])

# Create the plot
plt.plot(model_1_training.history['val_loss'], 'r', model_2_training.history['val_loss'], 'b')
plt.xlabel('Epochs')
plt.ylabel('Validation score')

```

Out[63]: Text(0, 0.5, 'Validation score')



The blue model is the one you made, the red is the original model. Your model had a lower loss value, so it is the better model. Nice job!

4.1.5 Adding layers to a network

You've seen how to experiment with wider networks. In this exercise, you'll try a deeper network (more hidden layers).

Once again, you have a baseline model called `model_1` as a starting point. It has 1 hidden layer, with 50 units. You can see a summary of that model's structure printed out. You will create a similar network with 3 hidden layers (still keeping 50 units in each layer).

This will again take a moment to fit both models, so you'll need to wait a few seconds to see the results after you run your code.

```
In [0]: model_1 = Sequential()
        model_1.add(Dense(50, activation='relu', input_shape=input_shape))
        model_1.add(Dense(2, activation='softmax'))
        model_1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

In [65]: # The input shape to use in the first hidden layer
        input_shape = (n_cols,)

        # Create the new model: model_2
        model_2 = Sequential()

        # Add the first, second, and third hidden layers
        model_2.add(Dense(50, activation='relu', input_shape=input_shape))
        model_2.add(Dense(50, activation='relu'))
        model_2.add(Dense(50, activation='relu'))

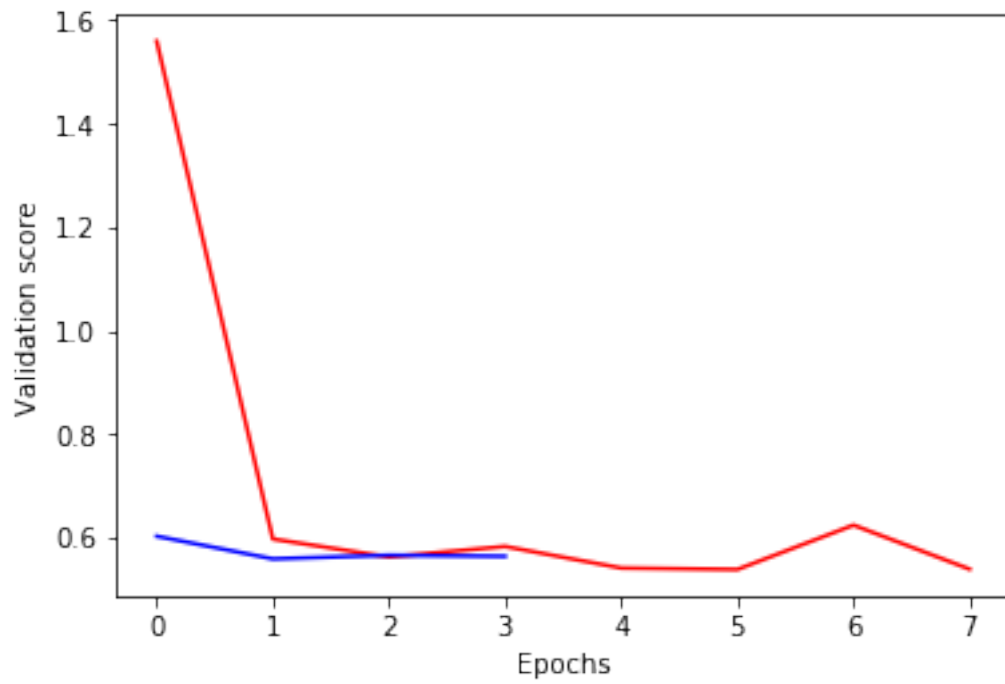
        # Add the output layer
        model_2.add(Dense(2, activation='softmax'))

        # Compile model_2
        model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
        # Fit model 1
        model_1_training = model_1.fit(predictors, target, epochs=20, validation_split=0.4, callbacks=[model_1_callback])

        # Fit model 2
        model_2_training = model_2.fit(predictors, target, epochs=20, validation_split=0.4, callbacks=[model_2_callback])

        # Create the plot
        plt.plot(model_1_training.history['val_loss'], 'r', model_2_training.history['val_loss'])
        plt.xlabel('Epochs')
        plt.ylabel('Validation score')

Out[65]: Text(0, 0.5, 'Validation score')
```



Great work! The blue model is the one you made and the red is the original model. The model with the lower loss value is the better model.