Introduction to Deep Learning in Python

Deeper machine learning technique

Black Raven (James Ng)
18 Feb 2021 · 36 min read

This is a memo to share what I have learnt in Introduction to Deep Learning in Python, capturing the learning objectives as well as my personal notes. The course is taught by Dan Becker from DataCamp, and it includes 4 chapters:

Chapter 1: Basics of deep learning and neural networks
Chapter 2: Optimizing a neural network with backward propagation
Chapter 3: Building deep learning models with keras
Chapter 4: Fine-tuning keras models

Deep learning is the machine learning technique behind the most exciting capabilities in diverse areas like robotics, natural language processing, image recognition, and artificial intelligence, including the famous AlphaGo. In this course, you'll gain hands-on, practical knowledge of how to use deep learning with Keras 2.0, the latest version of a cutting-edge library for deep learning in Python.

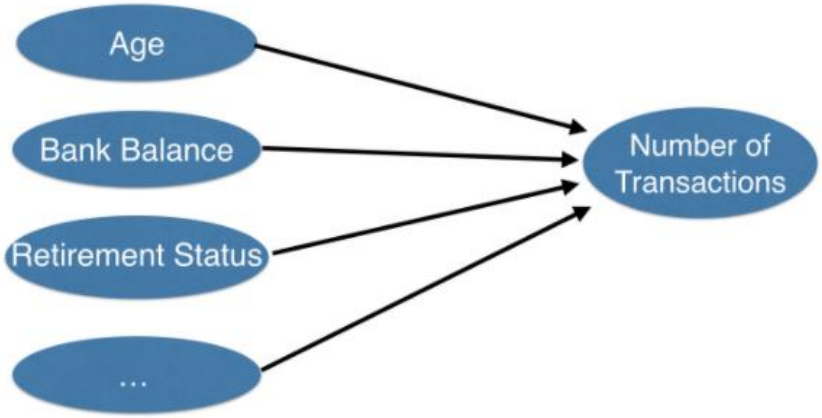
Chapter 1. Basics of deep learning and neural networks

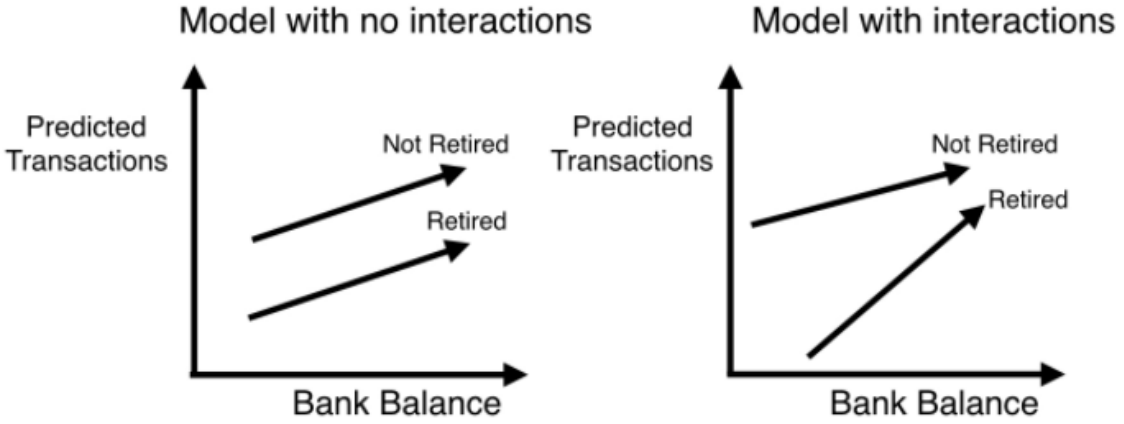
In this chapter, you'll become familiar with the fundamental concepts and terminology used in deep learning, and understand why deep learning techniques are so powerful today. You'll build simple neural networks and generate predictions with them.

Introduction to deep learning

To predict how many transactions each customer will make next year

Example using linear regression model using scikit-learn:

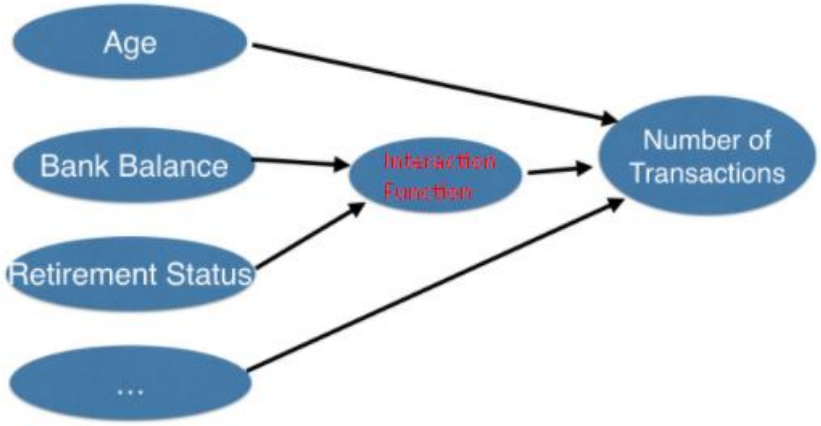


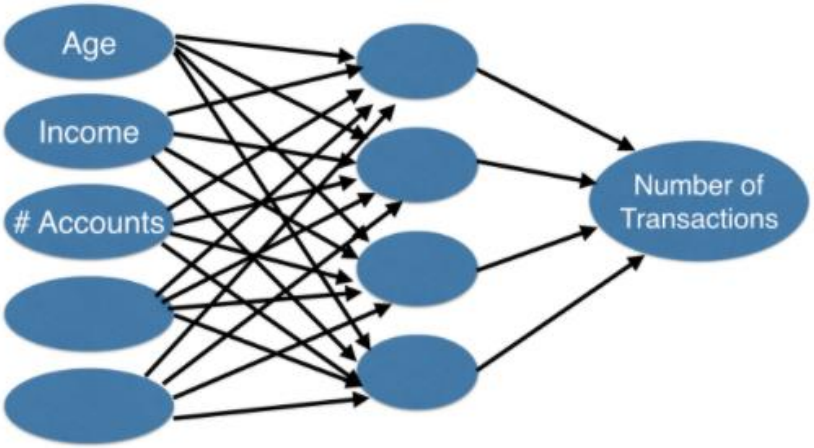


Neural networks are powerful modelling approach that accounts for interactions really well. Deep learning models use neural networks to perform predictions, with the ability to capture extremely complex interactions, for text, images, videos, audio, source code problems.

To focus on conceptual knowledge

- Build, debug and tune deep learning models (using keras) on conventional prediction problems
- Lay the foundation for progressing towards model applications

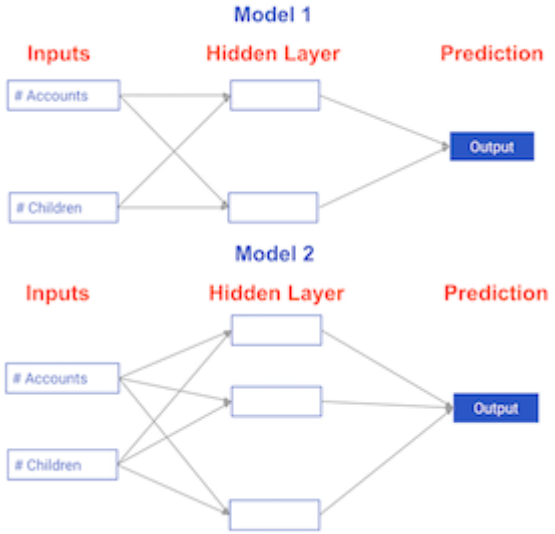




Input layer on far left, ie, predictive features like age or income (explicit data/observation)
Output layer on far right, ie, predicted number of transactions (explicit data/observation)
Hidden layers in between, ie, nodes (no explicit data/observation), an aggregation of info from input data, add to the model's ability to capture interactions

Comparing neural network models to classical regression models

Which of the models in the diagrams has greater ability to account for interactions?



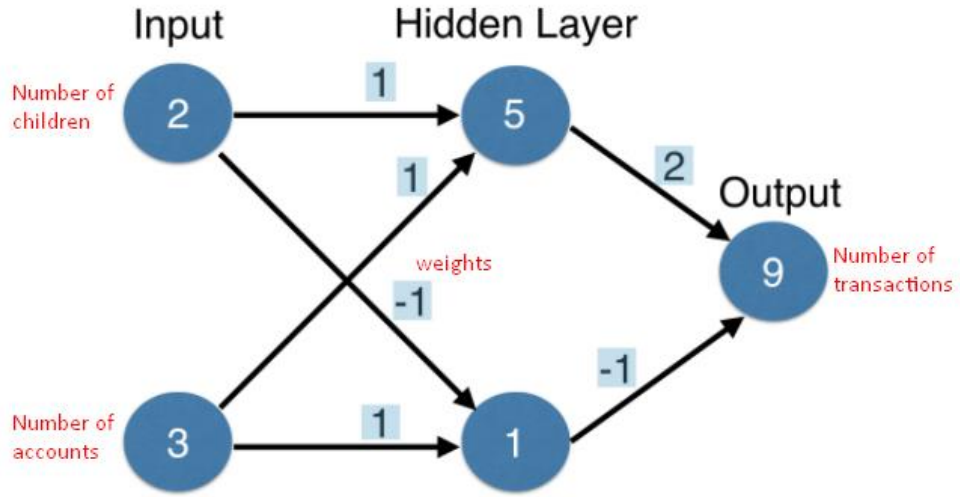
Possible Answers

- ○ Model 1.
- ◉ Model 2.
- ○ They are both the same.

Model 2 has more nodes in the hidden layer, and therefore, greater ability to capture interactions.

Forward propagation

Forward propagation = how neural network uses data to make predictions



Weights indicate how strongly each input effects the hidden node, to calculate the output using vector algebra dot product.

```
import numpy as np
input_data = np.array([2, 3])
weights = {'node_0': np.array([1, 1]),
               'node_1': np.array([-1, 1]),
               'output': np.array([2, -1])}
node_0_value = (input_data * weights['node_0']).sum()
node_1_value = (input_data * weights['node_1']).sum()
```

```
hidden_layer_values = np.array([node_0_value, node_1_value])
print(hidden_layer_values)
```

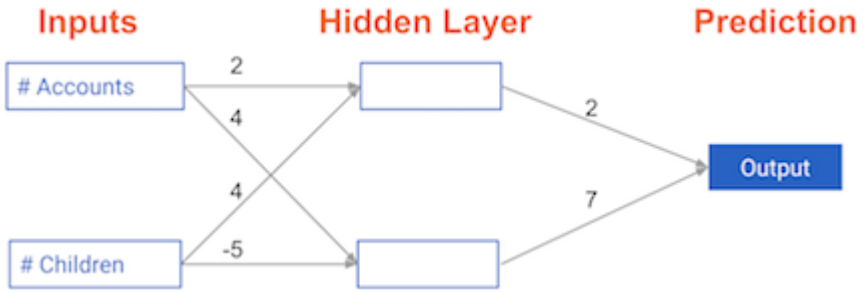
```
[5, 1]
```

```
output = (hidden_layer_values * weights['output']).sum()
print(output)
```

```
9
```

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']`. NumPy will be pre-imported for you as `np` in all exercises.

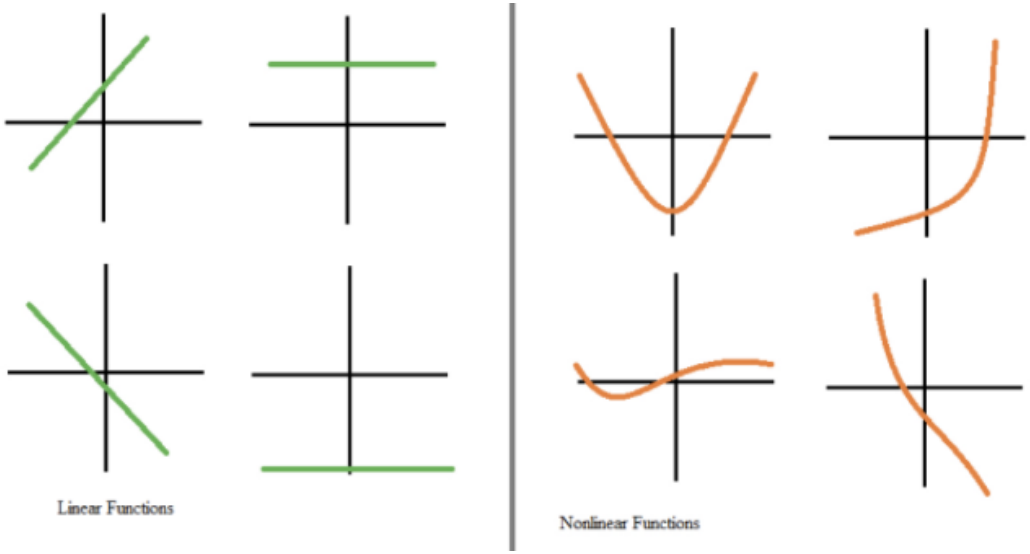
```
# Calculate node 0 value: node_0_value
node_0_value = (input_data * weights['node_0']).sum()

# 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
print(output)
```

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

```
<script.py> output:
    -39
```

Activation functions

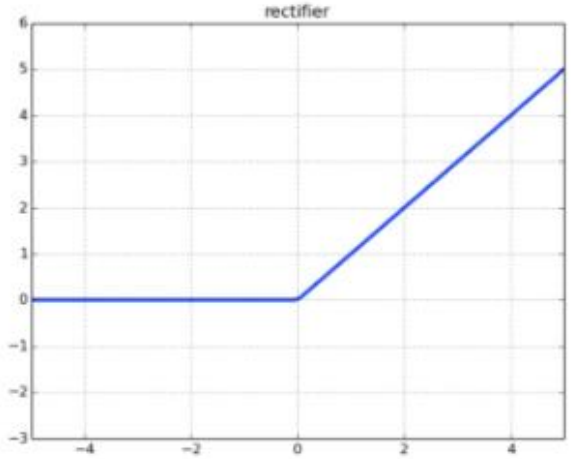
For neural networks to achieve their maximum predictive power, we must apply an activation function in the hidden layers. An activation function allows the model to capture linearities and **non-linearities**.



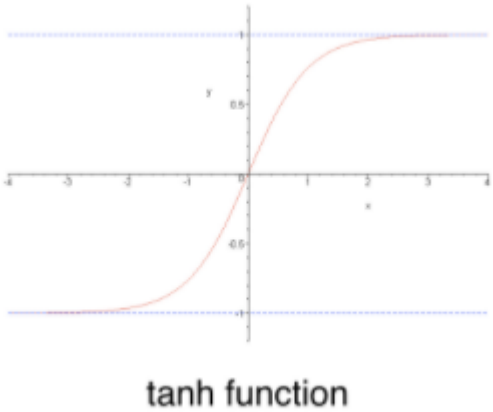
Non-linearities capture patterns like how going from no children to 1 child may impact your banking transactions differently than going from 3 children to 4.

An activation function is something applied to the value coming into a node, which then transforms it into the value stored in that node or the node output.

ReLU (Rectified Linear Activation) function



$$RELU(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$



tanh function

ReLU is surprisingly powerful when composed together through multiple successive hidden layers.

```
import numpy as np
input_data = np.array([-1, 2])
weights = {'node_0': np.array([3, 3]),
               'node_1': np.array([1, 5]),
               'output': np.array([2, -1])}
node_0_input = (input_data * weights['node_0']).sum()
node_0_output = np.tanh(node_0_input)
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = np.tanh(node_1_input)
hidden_layer_outputs = np.array([node_0_output, node_1_output])
output = (hidden_layer_output * weights['output']).sum()
```

```
print(output)
```

```
1.2382242525694254
```

The Rectified Linear Activation Function (ReLU)

As Dan explained to you in the video, 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

```
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
print(model_output)
```

```
<script.py> output:
    52
```

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.

Applying the network to many observations/rows of data

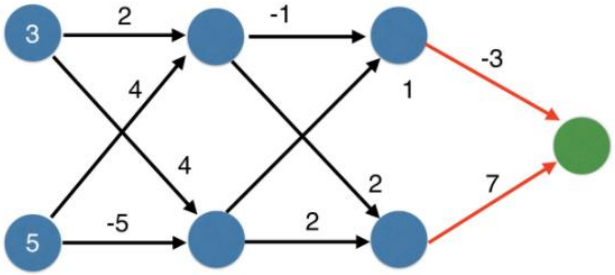
You'll now define a function called `predict_with_network()` which will generate predictions for multiple data observations, which are pre-loaded as `input_data`. As before, `weights` are also pre-loaded. In addition, the `relu()` function you defined in the previous exercise has been pre-loaded.

```
# 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
```

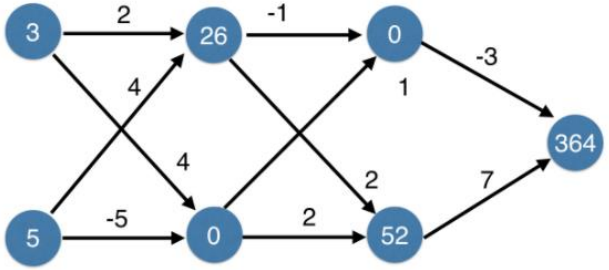
```
    node_1_input = (input_data_row * weights['node_1']).sum()
    node_1_output = relu(node_1_input)

    # 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
print(results)
```

Deeper networks

Multiple hidden layers

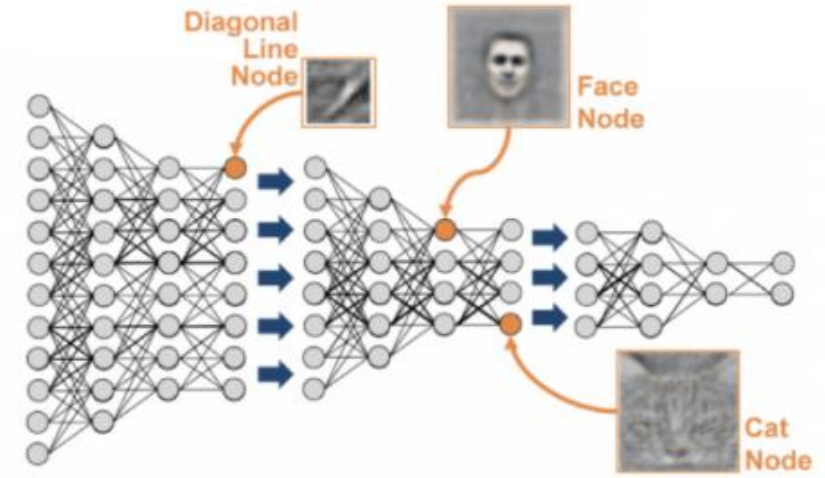


Calculate with ReLU Activation Function



Calculate with ReLU Activation Function

Deep networks internally build representations of patterns in the data.
This partially replace the need for feature engineering.
Representation learning: subsequent layers build increasingly sophisticated representations of the raw data, until we get to a stage where we can make predictions.



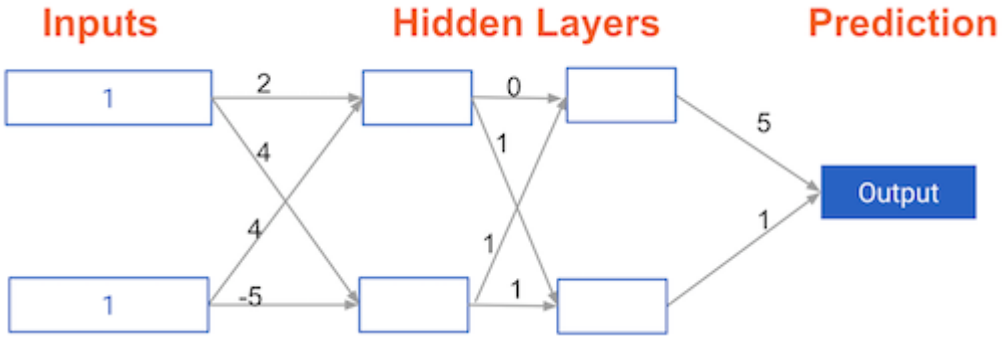
When a neural network tries to classify an image, the first hidden layers build up patterns or interactions that are conceptually simple. A simple interaction would look at groups of nearby pixels and find patterns like **diagonal lines**, **horizontal lines**, **vertical lines**, **blurry areas**, etc. Once the network has identified where there are these patterns, subsequent layers combine that information to find larger patterns, like big **squares**. A later layer might put together the location of squares and other geometric shapes to identify a **checkerboard** pattern, a **face**, a **car**, or whatever is in the image.

Modeler does not need to specify the patterns or interactions.
When you train the model, the network gets weights that find the relevant patterns or interactions to make better predictions.

Forward propagation in a deeper network

You now have a model with 2 hidden layers. The values for an input data point are shown inside the input nodes. The weights are shown on the edges/lines. What prediction would this model make on this data point?

Assume the activation function at each node is the *identity function*. That is, each node's output will be the same as its input. So the value of the bottom node in the first hidden layer is -1, and not 0, as it would be if the ReLU activation function was used.



Possible Answers

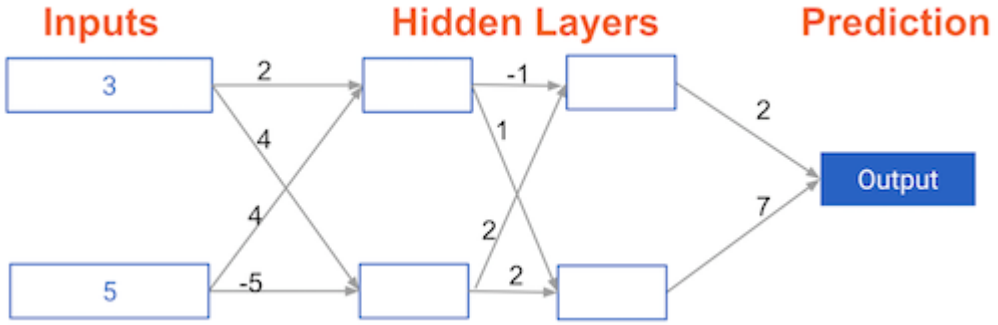
- 0.
- 7.
- 9.

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']`.



```
def predict_with_network(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 output here: model_output
    model_output = (hidden_1_outputs * weights['output']).sum()

    # Return model_output
    return(model_output)

output = predict_with_network(input_data)
print(output)
```

```
<script.py> output:
    182
```

The network generated a prediction of 182.

Representations are learned

How are the weights that determine the features/interactions in Neural Networks created?

Possible Answers

- ○ A user chooses them when creating the model.
- ◉ The model training process sets them to optimize predictive accuracy.
- ○ The weights are random numbers.

You will learn more about how Neural Networks optimize their weights in the next chapter!

Levels of representation

Which layers of a model capture more complex or "higher level" interactions?

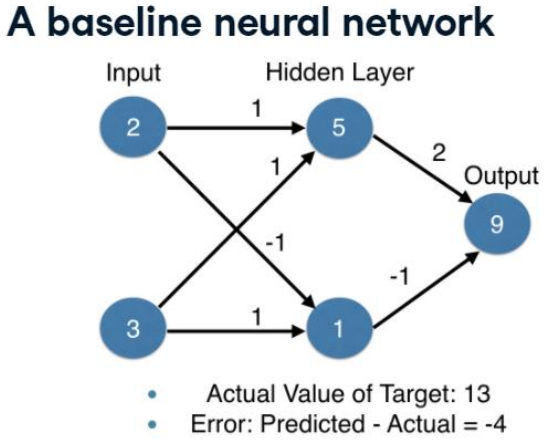
Possible Answers

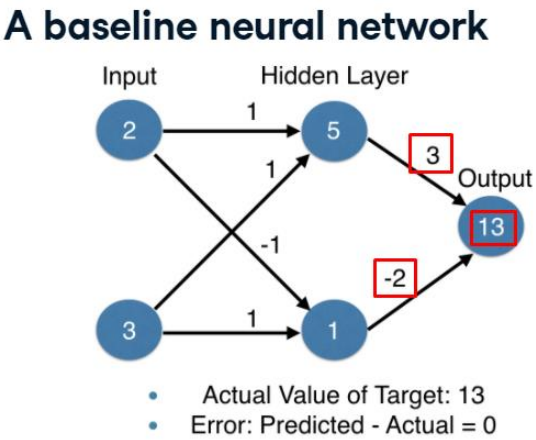
- ○ The first layers capture the most complex interactions.
- ◉ The last layers capture the most complex interactions.
- ○ All layers capture interactions of similar complexity.

Chapter 2. Optimizing a neural network with backward propagation

Learn how to optimize the predictions generated by your neural networks. You'll use 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 on in the second half of the course.

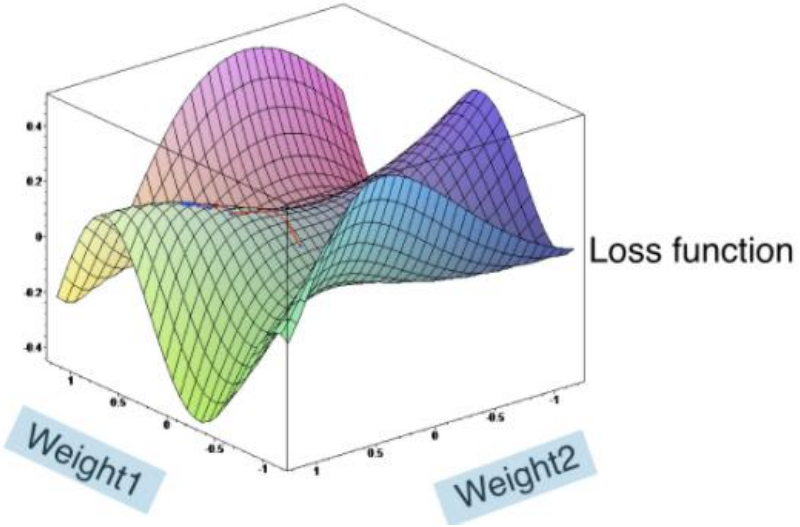
The need for optimization



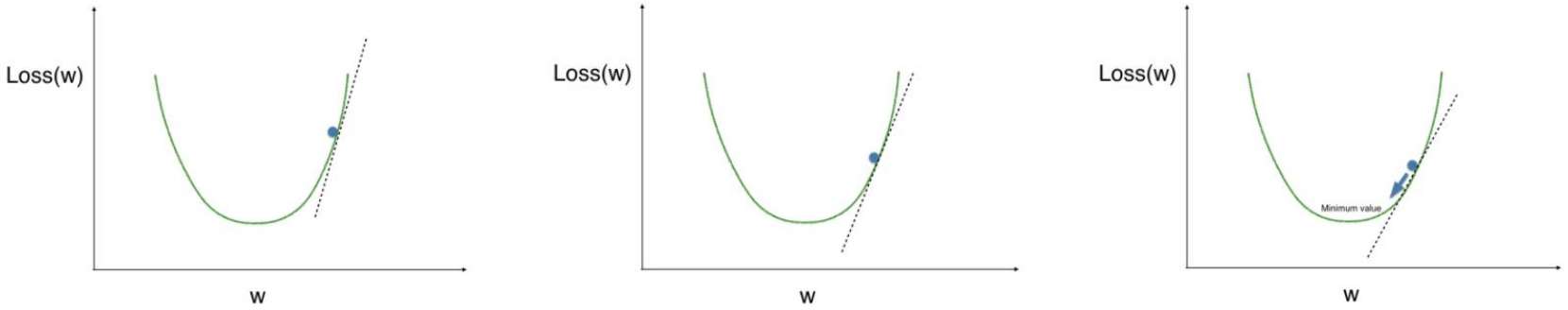


This change in weights improved the model for this data point.

Loss Function:
to aggregate all the errors into a single score,
measure of the model's predictive performance
example: mean-squared error for regression tasks



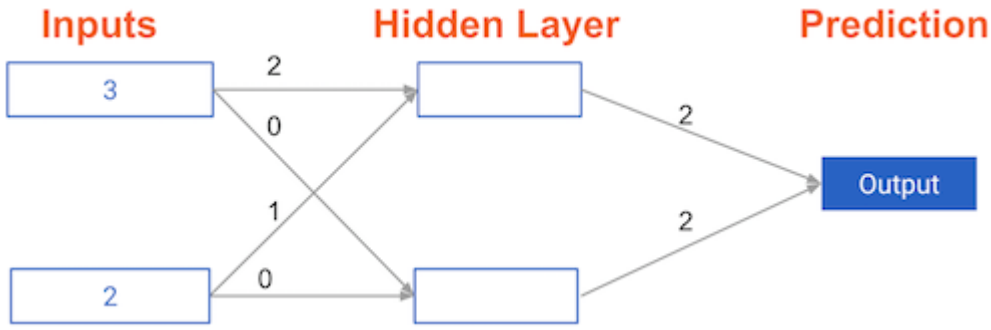
Goal: find the weights that will give the lowest value (better model) for the loss function
Approach: use an algorithm called **gradient descent**



Calculating model errors

For the exercises in this chapter, you'll continue working with the network to predict transactions for a bank.

What is the **error** (predicted - actual) for the following network using the ReLU activation function when the input data is [3, 2] and the actual value of the target (what you are trying to predict) is 5? It may be helpful to get out a pen and piece of paper to calculate these values.



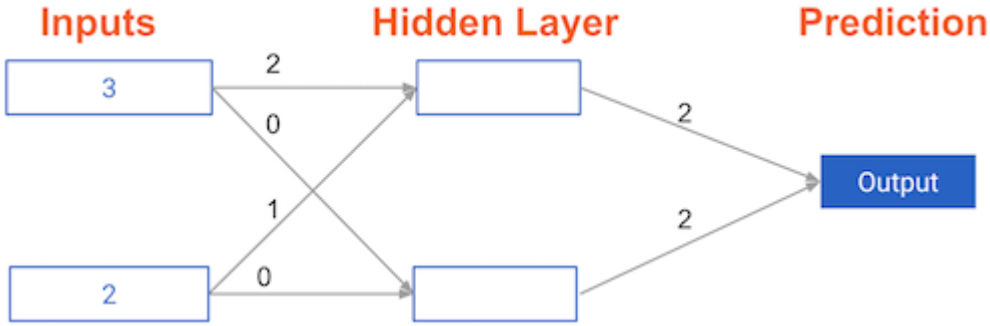
Possible Answers

- 5.
- 6.
- 11.
- 16.

The network generates a prediction of 16 (actual value is 5), which results in an error of 11.

Understanding how weights change model accuracy

Imagine you have to make a prediction for a single data point. The actual value of the target is 7. The weight going from `node_0` to the output is 2, as shown below. If you increased it slightly, changing it to 2.01, would the predictions become more accurate, less accurate, or stay the same?



Possible Answers

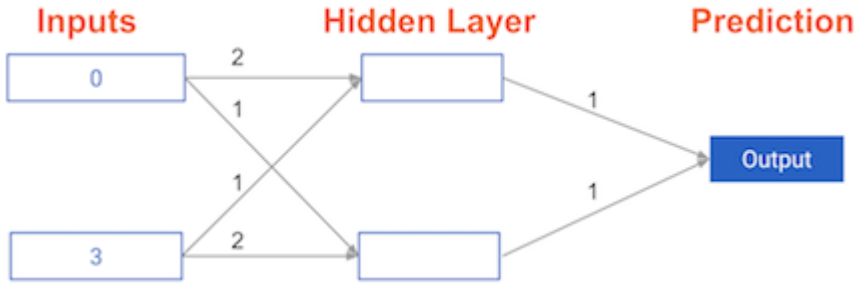
- More accurate.
- Less accurate.
- Stay the same.

Increasing the weight to 2.01 would increase the resulting error from 9 to 9.08, making the predictions less accurate.

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.

```
# 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, 2],
             'output': [1, 0]
            }

# 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)
```

```
<script.py> output:
    6
    0
```

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

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.

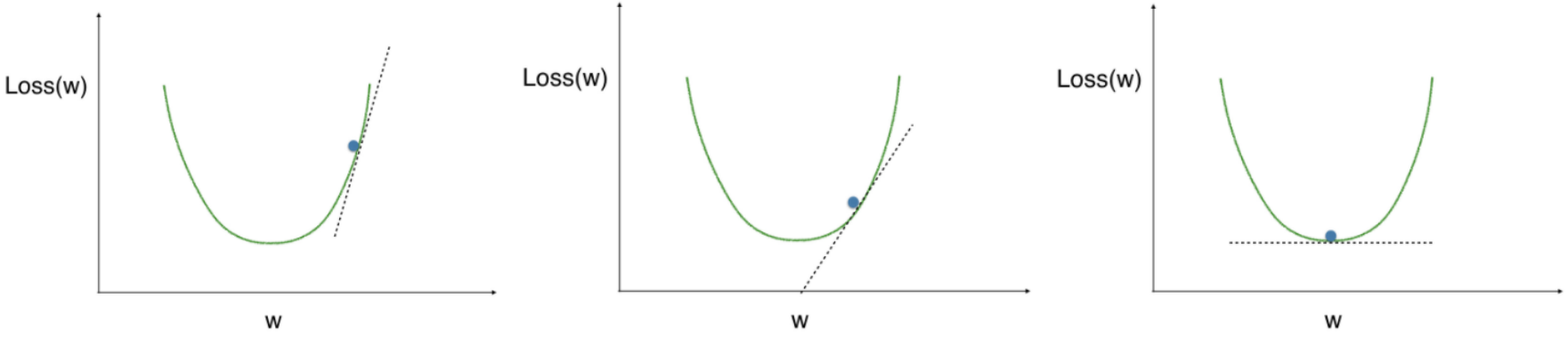
```
from sklearn.metrics import mean_squared_error

# 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)
```

```
<script.py> output:
    Mean squared error with weights_0: 37.500000
    Mean squared error with weights_1: 49.890625
```

It looks like model_output_1 has a higher mean squared error.

Gradient Descent



With gradient descent, you repeatedly find a slope capturing how your loss function changes as a weight changes. Then you make a small change to the weight to get to a lower point, and you repeat this until you could not go downhill anymore.

Learning Rate: Because too big a step might lead us astray, use learning rate to update each weight by subtracting `learning rate * slope`. This ensure we take small steps, so we reliably move towards the optimal weights.

```
import numpy as np
weights = np.array([1, 2])
input_data = np.array([3, 4])
target = 6
learning_rate = 0.01
preds = (weights * input_data).sum()
error = preds - target
print(error)
```

```
5
```

```
gradient = 2 * input_data * error
gradient
```

```
array([30, 40])
```

```
weights_updated = weights - learning_rate * gradient
preds_updated = (weights_updated * input_data).sum()
error_updated = preds_updated - target
print(error_updated)
```

```
2.5
```

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 * (xb-y)`, or `2 * input_data * 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`.

```
# 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

# Print the slope
print(slope)
```

```
<script.py> output:
    [14 28 42]
```

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

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.

The weights have been pre-loaded as `weights`, the actual value of the target as `target`, and the input data as `input_data`. The predictions from the initial weights are stored as `preds`.

```
# Set the learning rate: learning_rate
learning_rate = 0.01
```

```python
# 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)
```

```
<script.py> output:
    7
    5.04
```

Updating the model weights did indeed decrease the error!

Making multiple updates to weights

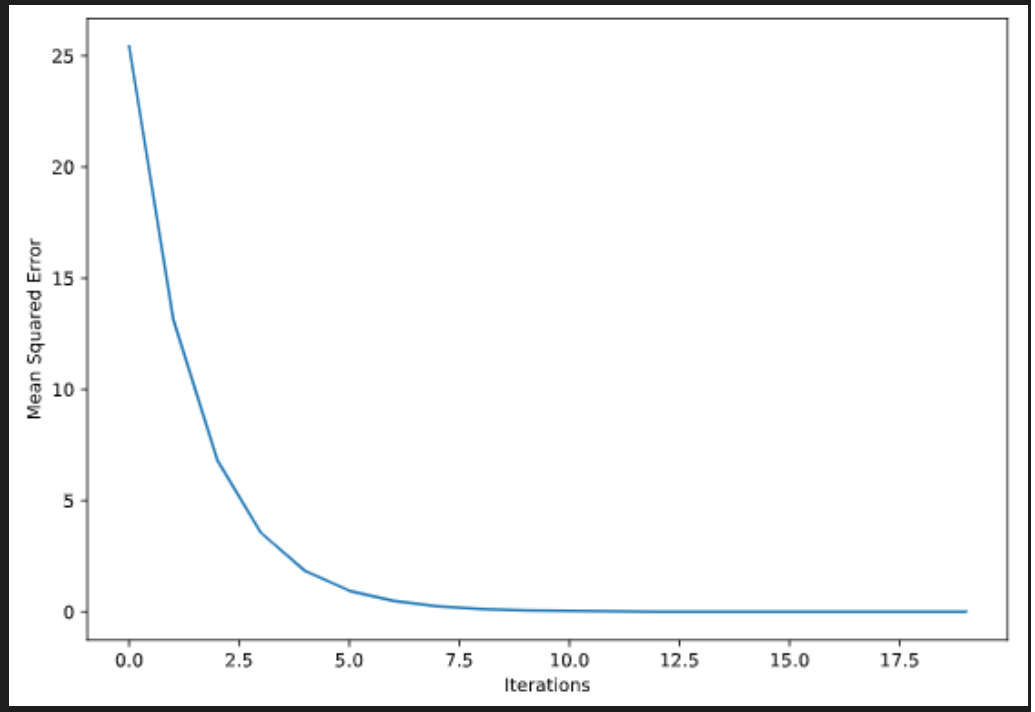
You're now going to make multiple updates so you can dramatically improve your model weights, and see how the predictions improve with each update.

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.

We have also pre-loaded `matplotlib.pyplot`, and the error history will be plotted after you have done your gradient descent steps.

```
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')
plt.show()
```

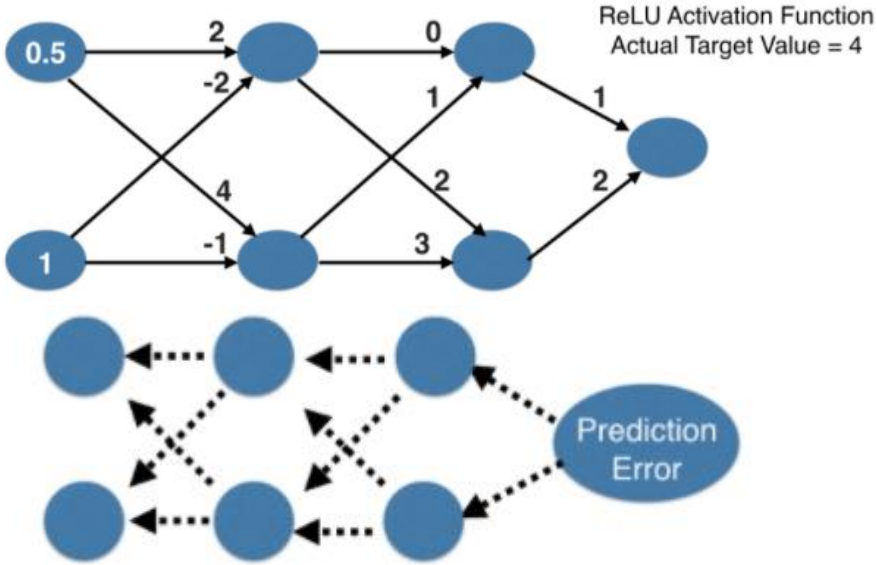


As you can see, the mean squared error decreases as the number of iterations go up.

Backpropagation

Allows gradient descent to update all weights in neural network (by getting gradients for all weights).
Comes from chainrule of calculus, to estimate the slope of the loss function w.r.t. each weight.
Important to understand the structure/process, and then use a library that implements this.

Do forward propagation to calculate predictions/errors before we do back propagation.



Gradients for weight is the product of:

1. the node value at the weights input
2. the slope from plotting the loss function against that weight's output node
3. the slope of the activation function at the weight's output

Also need to keep track of the slopes of the loss function w.r.t. node values, because we use those slopes in our calculations of slopes at weights.
Slope of node values are the sum of the slopes for all weights that come out of them.

The relationship between forward and backward propagation

If you have gone through 4 iterations of calculating slopes (using backward propagation) and then updated weights, how many times must you have done forward propagation?

Possible Answers

- 0.
- 1.
- 4.
- 8.

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

Thinking about backward propagation

If your predictions were all exactly right, and your errors were all exactly 0, the slope of the loss function with respect to your predictions would also be 0. In that circumstance, which of the following statements would be correct?

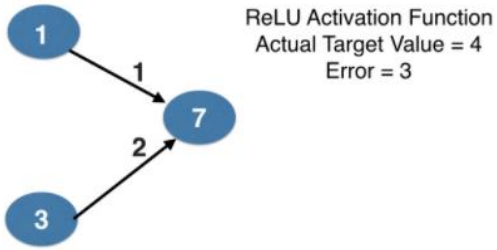
Possible Answers

- (●) The updates to all weights in the network would also be 0.
- ( ) The updates to all weights in the network would be dependent on the activation functions.
- ( ) The updates to all weights in the network would be proportional to values from the input data.

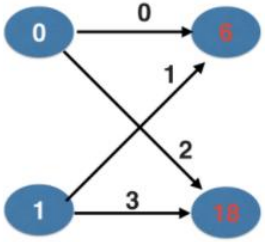
In this situation, the updates to all weights in the network would indeed also be `0`.

Backpropagation in practice

white = node values
black = weight values
red = calculated slope values of the loss function w.r.t that node



ReLU Activation Function
Actual Target Value = 4
Error = 3



Current Weight Value	Gradient
0	0
1	6
2	0
3	18

Steps:

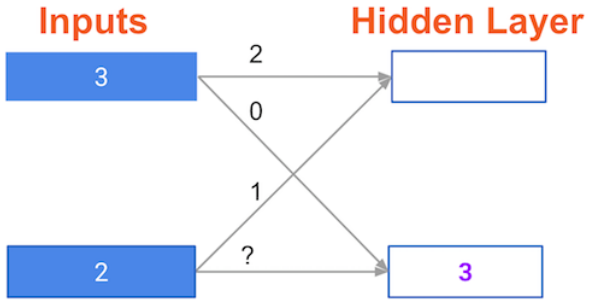
1. start at some random set of weights
2. use forward propagation to make a prediction
3. use backward propagation to calculate the slope of the loss function w.r.t. each weight
4. multiply that slope by the learning rate, and subtract from the current weights
5. keep going with that cycle until we get to a flat part

Stochastic Gradient Descent

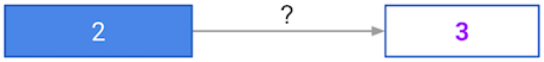
To calculate slopes on only a subset of the data (a batch), one batch at a time (i.s.o. all data)
Use a different batch of data to calculate the next update
Start over from the beginning once all data is used
Each time through the training data is called an epoch

A round of backpropagation

In the network shown below, we have done forward propagation, and node values calculated as part of forward propagation are shown in white. The weights are shown in black. Layers after the question mark show the slopes calculated as part of back-prop, rather than the forward-prop values. Those slope values are shown in purple.
This network again uses the ReLU activation function, so the slope of the activation function is 1 for any node receiving a positive value as input. Assume the node being examined had a positive value (so the activation function's slope is 1).



What is the slope needed to update the weight with the question mark?



Possible Answers

- 0.
- 2.
- 6.
- Not enough information.

The slope needed to update this weight is indeed `6`.
You're now ready to start building deep learning models with keras!

Chapter 3. Building deep learning models with keras

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

Creating a keras model

To create and optimise these networks using the Keras interface to the TensorFlow deep learning library.

Model building steps:

- specify **architecture**: number of layers, number of nodes in each layer, which activation function
- **compile** the model: specify loss function, optimisation details
- **fit** the model: cycle of back-propagation and optimisation of model weights using data
- **predict** using model

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential

predictors = np.loadtxt('predictors_data.csv', delimiter=',')
n_cols = predictors.shape[1]

model = Sequential()
model.add(Dense(100, activation='relu', input_shape = (n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
```

n_cols = number of nodes in the input layer
first hidden layer: specify the number of nodes, activation function, input shape (with any number of rows)
output layer has only 1 node, ie, prediction.

Understanding your data

You will soon start building models in Keras to predict wages based on various professional and demographic factors. Before you start building a model, it's good to understand your data by performing some exploratory analysis.

The data is pre-loaded into a pandas DataFrame called `df`. Use the `.head()` and `.describe()` methods in the IPython Shell for a quick overview of the DataFrame.

The target variable you'll be predicting is `wage_per_hour`. Some of the predictor variables are binary indicators, where a value of 1 represents `True`, and 0 represents `False`.

Of the 9 predictor variables in the DataFrame, how many are binary indicators? The min and max values as shown by `.describe()` will be informative here. How many binary indicator predictors are there?

Possible Answers

- 0.
- 5.
- 6.

There are 6 binary indicators.

```
In [2]: df.describe()
Out[2]:
       wage_per_hour       union  education_yrs  experience_yrs         age  \
count     534.000000  534.000000     534.000000      534.000000  534.000000
mean        9.024064    0.179775      13.018727       17.822097   36.833333
std         5.139097    0.384360       2.615373       12.379710   11.726573
min         1.000000    0.000000       2.000000        0.000000   18.000000
25%         5.250000    0.000000      12.000000        8.000000   28.000000
50%         7.780000    0.000000      12.000000       15.000000   35.000000
75%        11.250000    0.000000      15.000000       26.000000   44.000000
max        44.500000    1.000000      18.000000       55.000000   64.000000

            female        marr       south  manufacturing  construction
count   534.000000  534.000000  534.000000     534.000000    534.000000
mean      0.458801    0.655431    0.292135       0.185393      0.044944
std       0.498767    0.475673    0.455170       0.388981      0.207375
min       0.000000    0.000000    0.000000       0.000000      0.000000
25%       0.000000    0.000000    0.000000       0.000000      0.000000
50%       0.000000    1.000000    0.000000       0.000000      0.000000
75%       1.000000    1.000000    1.000000       0.000000      0.000000
max       1.000000    1.000000    1.000000       1.000000      1.000000
```

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`.

For all exercises in this chapter, we've imported the `Sequential` model constructor, the `Dense` layer constructor, and pandas.

```
# 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))
```

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

Compiling and fitting a model

Specify the **optimiser**, which controls the learning rate

- there are a few algorithms that automatically tune the learning rate, mathematically complex

- to choose a versatile algorithm and use that for most problems, eg. "`Adam`"
- Adam adjusts the learning rate during gradient descent, to ensure reasonable values throughout the weight optimisation process

Specify the **loss function**

- common choice for regression problems: `mean_squared_error`
- for classification, there is a new default metric

Fit the model

- applying backpropagation and gradient descent with your data to update the weights
- scaling data to similar sized values before fitting can ease optimisation, eg. subtract each feature by that feature mean and divide it by its standard deviation

```
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(predictors, target)
```

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. In the video, Dan mentioned that the 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!

```
# Import necessary modules
import keras
```

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

# Specify the model
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(50, activation='relu', input_shape = (n_cols,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

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

```
<script.py> output:
    Loss function: mean_squared_error
```

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.

```
# Import necessary modules
import keras
from keras.layers import Dense
from keras.models import Sequential

# Specify the model
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(50, activation='relu', input_shape = (n_cols,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# Compile the model
```

```
model.compile(optimizer='adam', loss='mean_squared_error')

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

```
<script.py> output:
    Epoch 1/10
     32/534 [>.............................] - ETA: 0s - loss: 146.0927000
    Epoch 2/10
     32/534 [>.............................] - ETA: 0s - loss: 85.6932000
    Epoch 3/10
     32/534 [>.............................] - ETA: 0s - loss: 21.0367000
    Epoch 4/10
     32/534 [>.............................] - ETA: 0s - loss: 16.9063000
    Epoch 5/10
     32/534 [>.............................] - ETA: 0s - loss: 23.2321000
    Epoch 6/10
     32/534 [>.............................] - ETA: 0s - loss: 13.3775000
    Epoch 7/10
     32/534 [>.............................] - ETA: 0s - loss: 28.1790000
    Epoch 8/10
     32/534 [>.............................] - ETA: 0s - loss: 11.5179000
    Epoch 9/10
     32/534 [>.............................] - ETA: 0s - loss: 21.9117000
    Epoch 10/10
     32/534 [>.............................] - ETA: 0s - loss: 5.4945000
```

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

Classification models

Specify the loss function

- most common: `categorical_crossentropy` (instead of mean_squared_error)
- similar to log loss: lower is better

Add `metrics=['accuracy']` to compile step for easy-to-understand diagnostics

Output layer to have separate node for each possible outcome, and uses `softmax` activation.

Softmax activation function ensures the predictions sum to 1, so they can be interpreted like probabilities.

shot_clock	dribbles	touch_time	shot_dis	close_def_dis	shot_result
10.8	2	1.9	7.7	1.3	1
3.4	0	0.8	28.2	6.1	0
0	3	2.7	10.1	0.9	0
10.3	2	1.9	17.2	3.4	0

→

Outcome 0	Outcome 1
0	1
1	0
1	0
1	0

(one-hot encoding)

```
from keras.utils.np_utils import to_categorical

data = pd.read_csv('basketball_shot_log.csv')
predictors = data.drop(['shot_result'], axis=1).as_matrix()
target = to_categorical(data.shot_result)

model = Sequential()
model.add(Dense(100, activation='relu', input_shape = (n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(2, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(predictors, target)
```

Understanding your classification data

Now you will start modeling with a new dataset for a classification problem. This data includes information about passengers on the Titanic. You will use predictors such as `age`, `fare` and where each passenger embarked from to predict who will survive. This data is from **a tutorial on data science competitions**. Look **here** for descriptions of the features.

The data is pre-loaded in a pandas DataFrame called `df`. It's smart to review the maximum and minimum values of each variable to ensure the data isn't misformatted or corrupted. What was the maximum age of passengers on the Titanic? Use the `.describe()` method in the IPython Shell to answer this question.

Possible Answers

- 29.699.
- 80.
- 891.
- It is not listed.

```
In [1]: df.describe()
Out[1]:
       survived      pclass         age       sibsp       parch        fare        male
count  891.000000  891.000000  891.000000  891.000000  891.000000  891.000000  891.000000
mean     0.383838    2.308642   29.699118    0.523008    0.381594   32.204208    0.647587
std      0.486592    0.836071   13.002015    1.102743    0.806057   49.693429    0.477990
min      0.000000    1.000000    0.420000    0.000000    0.000000    0.000000    0.000000
25%      0.000000    2.000000   22.000000    0.000000    0.000000    7.910400    0.000000
50%      0.000000    3.000000   29.699118    0.000000    0.000000   14.454200    1.000000
75%      1.000000    3.000000   35.000000    1.000000    0.000000   31.000000    1.000000
max      1.000000    3.000000   80.000000    8.000000    6.000000  512.329200    1.000000
```

The maximum age in the data is 80.

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!

```
# Import necessary modules
import keras
from keras.layers import Dense
from keras.models import Sequential
from keras.utils import to_categorical

# 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)
```

```
<script.py> output:
    Epoch 1/10
     32/891 [>.............................] - ETA: 0s - loss: 7.6250 - acc: 0.2188
    Epoch 2/10
     32/891 [>.............................] - ETA: 0s - loss: 1.1922 - acc: 0.3125
    Epoch 3/10
     32/891 [>.............................] - ETA: 0s - loss: 2.2434 - acc: 0.5000
    Epoch 4/10
     32/891 [>.............................] - ETA: 0s - loss: 0.7250 - acc: 0.5938
    Epoch 5/10
     32/891 [>.............................] - ETA: 0s - loss: 0.6109 - acc: 0.5938
    Epoch 6/10
     32/891 [>.............................] - ETA: 0s - loss: 0.4744 - acc: 0.7500
    Epoch 7/10
     32/891 [>.............................] - ETA: 0s - loss: 0.6627 - acc: 0.5625
    Epoch 8/10
     32/891 [>.............................] - ETA: 0s - loss: 0.5095 - acc: 0.7500
    Epoch 9/10
     32/891 [>.............................] - ETA: 0s - loss: 0.6721 - acc: 0.5938
    Epoch 10/10
     32/891 [>.............................] - ETA: 0s - loss: 0.4573 - acc: 0.7500
```

This simple model is generating an accuracy of 68!

Using models

Save the model after you have trained it, saved in a format called hdf5, extension “h5”
Reload that model: `load_model()`
Verify model structure: `my_model.summary()`
Make predictions with the model: `my_model.predict()`

```
from keras.models import load_model
model.save('model_file.h5')
my_model = load_model('my_model.h5')
predictions = my_model.predict(data_to_predict_with)
probability_true = predictions[:,1]
```

```
my_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               1100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 202
=================================================================
Total params: 21,502
Trainable params: 21,502
Non-trainable params: 0
```

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.

```
# Specify, compile, and fit the model
model = Sequential()
model.add(Dense(32, activation='relu', input_shape = (n_cols,)))
model.add(Dense(2, activation='softmax'))
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(predictors, target)

# 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)
```

You're now ready to begin learning how to fine-tune your models.

Chapter 4. Fine-tuning keras models

Learn how to optimize your deep learning models in Keras. Start by learning how to validate your models, then understand the concept of model capacity, and finally, experiment with wider and deeper networks.

Understanding model optimization

In practice, optimisation is hard

- simultaneously optimising 1000s of parameters with complex relationships
- weights updates may not improve model meaningfully
- updates too small (if learning rate is low) or too large (if learning is high)

The easiest way to see the effect of different learning rates is to use the simplest optimizer, **Stochastic Gradient Descent**, sometimes abbreviated to **SGD**. This optimizer uses a fixed learning rate, around 0.01.

```
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)

lr_to_test = [.000001, 0.01, 1]

# loop over learning rates
for lr in lr_to_test:
   model = get_new_model()
   my_optimizer = SGD(lr=lr)
   model.compile(optimizer = my_optimizer, loss = 'categorical_crossentropy')
   model.fit(predictors, target)
```

To compare the results of training models trained with low, medium, high learning rates.
If model is not training better, try changing the activation function.

Diagnosing optimization problems

Which of the following could prevent a model from showing an improved loss in its first few epochs?

Possible Answers

- ○ Learning rate too low.
- ○ Learning rate too high.
- ○ Poor choice of activation function.
- ◉ All of the above.

All the options listed could prevent a model from showing an improved loss in its first few epochs.

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.

```
# 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)
```

```
<script.py> output:


    Testing model with learning rate: 0.000001

    Epoch 1/10
     32/891 [>.............................] - ETA: 0s - loss: 3.605300
    Epoch 2/10
     32/891 [>.............................] - ETA: 0s - loss: 3.575100
    Epoch 3/10
     32/891 [>.............................] - ETA: 0s - loss: 2.669200
```

(poor performance)

```
    Testing model with learning rate: 0.010000

    Epoch 1/10
     32/891 [>.............................] - ETA: 1s - loss: 1.091000
    Epoch 2/10
     32/891 [>.............................] - ETA: 0s - loss: 2.050800
    Epoch 3/10
     32/891 [>.............................] - ETA: 0s - loss: 0.570400
    Epoch 4/10
     32/891 [>.............................] - ETA: 0s - loss: 0.604800
```

(best performing)

```
    Testing model with learning rate: 1.000000

    Epoch 1/10
     32/891 [>.............................] - ETA: 1s - loss: 1.027300
    Epoch 2/10
     32/891 [>.............................] - ETA: 0s - loss: 4.533200
    Epoch 3/10
     32/891 [>.............................] - ETA: 0s - loss: 7.051700
    Epoch 4/10
     32/891 [>.............................] - ETA: 0s - loss: 6.044300
```

(poor performance)

Model validation
Model's performance on the training data is not a good indication of how it will perform on new data.
Validation data is data that is explicitly held out from training, and used only to test model performance.

Commonly use validation split rather than cross-validation.
Deep learning widely used on large datasets.
Single validation score is based on large amount of data, and is reliable.

Keras makes it easy to use some of your data as validation data by specifying `validation _split=0.3` for 30%

```
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics=['accuracy'])
model.fit(predictors, target, validation_split=0.3)
```

```
Epoch 1/10
89648/89648 [=====] - 3s - loss: 0.7552 - acc: 0.5775 - val_loss: 0.6969 - val_acc: 0.5561
Epoch 2/10
89648/89648 [=====] - 4s - loss: 0.6670 - acc: 0.6004 - val_loss: 0.6580 - val_acc: 0.6102
...
Epoch 8/10
89648/89648 [=====] - 5s - loss: 0.6578 - acc: 0.6125 - val_loss: 0.6594 - val_acc: 0.6037
Epoch 9/10
89648/89648 [=====] - 5s - loss: 0.6564 - acc: 0.6147 - val_loss: 0.6568 - val_acc: 0.6110
Epoch 10/10
89648/89648 [=====] - 5s - loss: 0.6555 - acc: 0.6158 - val_loss: 0.6557 - val_acc: 0.6126
```

Our goal is to have the best validation score possible, so we should keep training while validation score is improving, and then stop training when the validation score isn't improving. We do this with something called "early stopping."

```
from keras.callbacks import EarlyStopping

early_stopping_monitor = EarlyStopping(patience=2)

model.fit(predictors, target, validation_split=0.3, nb_epoch=20,
          callbacks = [early_stopping_monitor])
```

```
Train on 89648 samples, validate on 38421 samples
Epoch 1/20
89648/89648 [====] - 5s - loss: 0.6550 - acc: 0.6151 - val_loss: 0.6548 - val_acc: 0.6151
Epoch 2/20
89648/89648 [====] - 6s - loss: 0.6541 - acc: 0.6165 - val_loss: 0.6537 - val_acc: 0.6154
...
Epoch 8/20
89648/89648 [====] - 6s - loss: 0.6527 - acc: 0.6181 - val_loss: 0.6531 - val_acc: 0.6160
Epoch 9/20
89648/89648 [====] - 7s - loss: 0.6524 - acc: 0.6176 - val_loss: 0.6513 - val_acc: 0.6172
Epoch 10/20
89648/89648 [====] - 6s - loss: 0.6527 - acc: 0.6176 - val_loss: 0.6549 - val_acc: 0.6134
Epoch 11/20
89648/89648 [====] - 6s - loss: 0.6522 - acc: 0.6178 - val_loss: 0.6517 - val_acc: 0.6169
```

In epoch 9 we had a validation loss score of 0.6513. We didn't beat that score in the next 2 epochs, so we stopped training.

Creating a great model requires some experimentation with different architectures:

- more/fewer layers
- layers with more/fewer nodes
- different activation functions

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.

```
# 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)
```

```
<script.py> output:
    Train on 623 samples, validate on 268 samples
    Epoch 1/10
     32/623 [>.............................] - ETA: 1s - loss: 3.3028 - acc: 0.4062
    Epoch 2/10
     32/623 [>.............................] - ETA: 0s - loss: 0.6112 - acc: 0.6875
    Epoch 3/10
     32/623 [>.............................] - ETA: 0s - loss: 0.6294 - acc: 0.6250
    Epoch 4/10
     32/623 [>.............................] - ETA: 0s - loss: 0.5277 - acc: 0.7812
    Epoch 5/10
     32/623 [>.............................] - ETA: 0s - loss: 0.6128 - acc: 0.6562
    Epoch 6/10
     32/623 [>.............................] - ETA: 0s - loss: 0.6458 - acc: 0.6562
    Epoch 7/10
     32/623 [>.............................] - ETA: 0s - loss: 0.5174 - acc: 0.7500
    Epoch 8/10
     32/623 [>.............................] - ETA: 0s - loss: 0.6238 - acc: 0.7500
    Epoch 9/10
     32/623 [>.............................] - ETA: 0s - loss: 0.5806 - acc: 0.7188
    Epoch 10/10
     32/623 [>.............................] - ETA: 0s - loss: 0.4975 - acc: 0.7500
```

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()`, as Dan showed in the video.

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

```
# Import EarlyStopping
```

```python
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])
```

```
<script.py> output:
    Train on 623 samples, validate on 268 samples
    Epoch 1/30
     32/623 [>.............................] - ETA: 1s - loss: 5.6563 - acc: 0.4688
    Epoch 2/30
     32/623 [>.............................] - ETA: 0s - loss: 1.8354 - acc: 0.4688
    Epoch 3/30
     32/623 [>.............................] - ETA: 0s - loss: 0.8245 - acc: 0.6562
    Epoch 4/30
     32/623 [>.............................] - ETA: 0s - loss: 1.3859 - acc: 0.5312
    Epoch 5/30
     32/623 [>.............................] - ETA: 0s - loss: 0.5649 - acc: 0.7188
    Epoch 6/30
     32/623 [>.............................] - ETA: 0s - loss: 0.4340 - acc: 0.8125
    Epoch 7/30
     32/623 [>.............................] - ETA: 0s - loss: 0.6571 - acc: 0.6875
```

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.

Experimenting with wider networks

Now you know everything you need to begin experimenting with a different **model with more nodes**!

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                110
_________________________________________________________________
dense_2 (Dense)              (None, 10)                110
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 22
=================================================================
Total params: 242.0
Trainable params: 242
Non-trainable params: 0.0
_________________________________________________________________
```

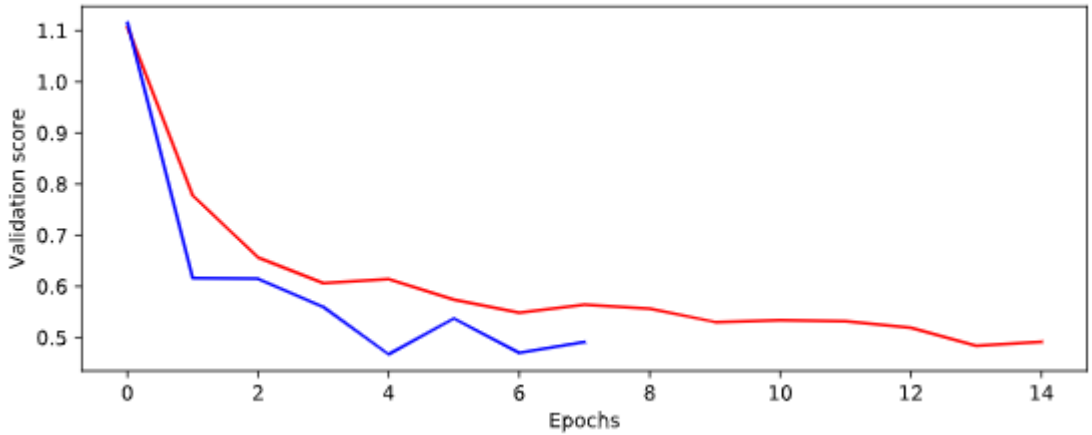
In this exercise you'll create a new model called `model_2` which is similar to `model_1`, except it has 100 nodes in each hidden layer.

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.

```
# 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_monitor], verbose=False)

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

# 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')
plt.show()
```



Your 100-nodes model in blue had a lower loss value (better model) than the original 10-nodes model in red.

Adding layers to a network

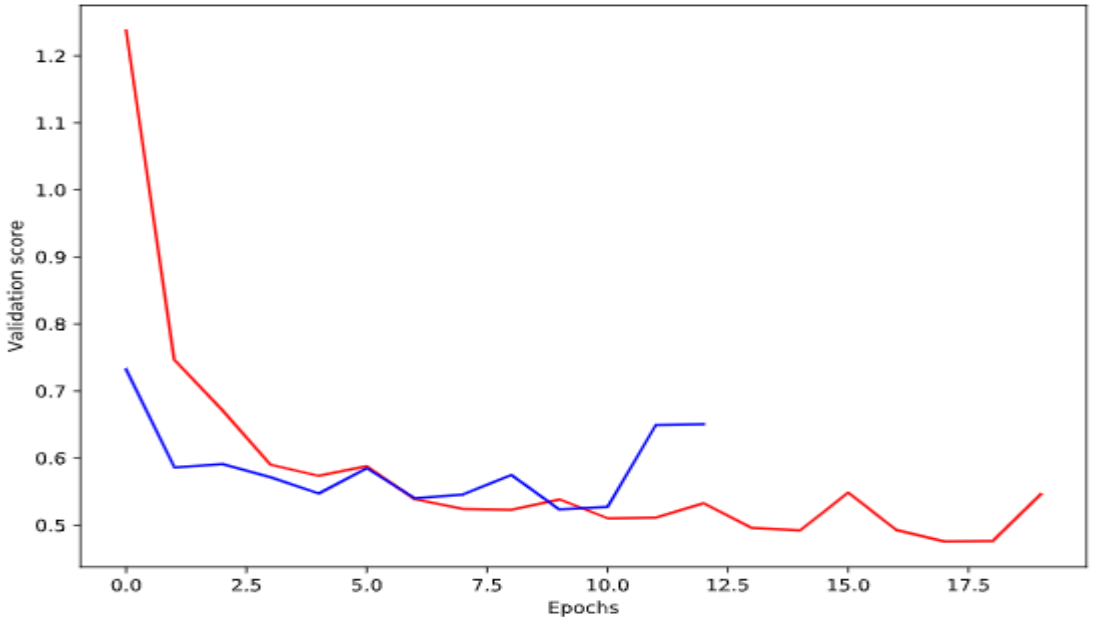
You've seen how to experiment with wider networks. In this exercise, you'll try a **deeper** network (**more hidden layers**).

Once again, you have a baseline model called `model_1` as a starting point. It has 1 hidden layer, with `50` nodes. 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 nodes in each layer).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 50)                550
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 102
=================================================================
Total params: 652.0
Trainable params: 652
Non-trainable params: 0.0
_________________________________________________________________
```

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.

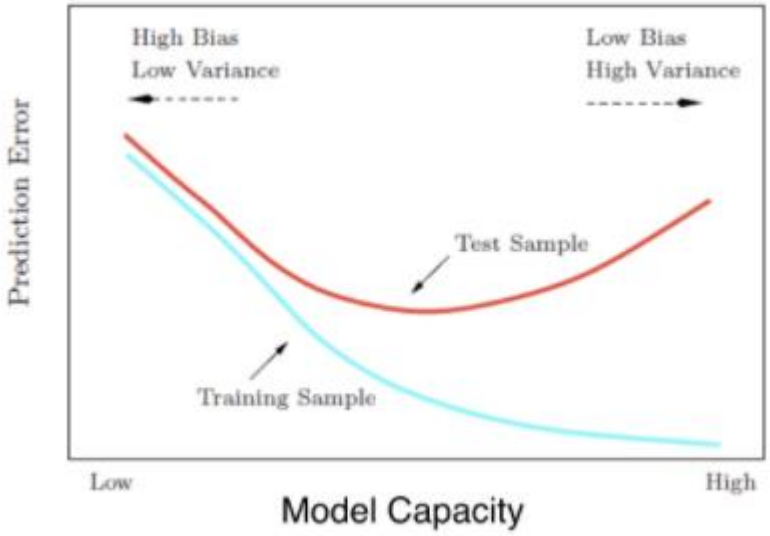
```
# The input shape to use in the first hidden layer
input_shape = (n_cols,)

# 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=[early_stopping_monito
r], verbose=False)

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

# 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')
plt.show()
```



The blue model (3 hidden layers) is the one you made and the red is the original model (1 hidden layer).
The model with the lower loss value is the better model.

Thinking about model capacity
Model capacity = network capacity = model's ability to capture predictive patterns in your data.

Overfitting is the ability of a model to fit oddities in your training data that are purely due to happenstance, and that would not apply in a new dataset.
Your model will make accurate predictions on training data, but it will make inaccurate predictions on validation data and new datasets.



Underfitting is when your model fails to find important predictive patterns in the training data. So it is accurate in neither in the training data nor validation data.

Validation score is the ultimate measure of a model's predictive ability.

How to increase model capacity:

- increase number of nodes in hidden layers
- add hidden layers

Workflow for optimising model capacity

- start with a small network
- gradually increase capacity as long as the score keeps improving
- keep increasing capacity until validation score is no longer improving

Hidden Layers	Nodes Per Layer	Mean Squared Error	Next Step
1	100	5.4	Increase Capacity
1	250	4.8	Increase Capacity
2	250	4.4	Increase Capacity
3	250	4.5	Decrease Capacity
3	200	4.3	Done

Experimenting with model structures

You've just run an experiment where you compared two networks that were identical except that the 2nd network had an extra hidden layer. You see that this 2nd network (the deeper network) had better performance. Given that, which of the following would be a good experiment to run next for even better performance?

Possible Answers

- ○ Try a new network with fewer layers than anything you have tried yet.
- ◉ Use more units in each hidden layer.
- ○ Use fewer units in each hidden layer.

Increasing the number of units in each hidden layer would be a good next step to try achieving even better performance.

Stepping up to images

MNIST dataset = images of handwritten digits
Each image is composed of a 28-pixel by 28-pixel grid
The image is represented by showing how dark each pixel is: 0 is light, 255 is dark
Flatten the 28 x 28 grid into 784 x 1 array for each image

To create a deep learning model taking in those 784 features as inputs, and to predict digits among 10 possible values (0 to 9) for the output.

Building your own digit recognition model

You've reached the final exercise of the course - you now know everything you need to build an accurate model to recognize handwritten digits!

We've already done the basic manipulation of the MNIST dataset shown in the video, so you have `X` and `y` loaded and ready to model with. `Sequential` and `Dense` from keras are also pre-imported.

To add an extra challenge, we've loaded only 2500 images, rather than 60000 which you will see in some published results. Deep learning models perform better with more data, however, they also take longer to train, especially when they start becoming more complex.

If you have a computer with a CUDA compatible GPU, you can take advantage of it to improve computation time. If you don't have a GPU, no problem! You can set up a deep learning environment in the cloud that can run your models on a GPU. Here is a **blog post** by Dan that explains how to do this - check it out after completing this exercise! It is a great next step as you continue your deep learning journey.

Ready to take your deep learning to the next level? Check out **Advanced Deep Learning with Keras in Python** to see how the Keras functional API lets you build domain knowledge to solve new types of problems. Once you know how to use the functional API, take a look at **"Convolutional Neural Networks for Image Processing"** to learn image-specific applications of Keras.

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

# Add the first hidden layer
model.add(Dense(50, activation='relu', input_shape=(784,)))

# Add the second hidden layer
model.add(Dense(50, activation='relu'))

# Add the output layer
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
```

```
              metrics=['accuracy'])

# Fit the model
model.fit(X, y, validation_split=0.3)
```

```
<script.py> output:
    Train on 1750 samples, validate on 750 samples
    Epoch 1/10
      32/1750 [..............................] - ETA: 2s - loss: 2.1979 - acc: 0.2188
    Epoch 2/10
      32/1750 [..............................] - ETA: 0s - loss: 0.9344 - acc: 0.7812
    Epoch 3/10
      32/1750 [..............................] - ETA: 0s - loss: 0.3727 - acc: 0.9688
    Epoch 4/10
      32/1750 [..............................] - ETA: 0s - loss: 0.1842 - acc: 0.9062
    Epoch 5/10
      32/1750 [..............................] - ETA: 0s - loss: 0.1728 - acc: 0.9688
    Epoch 6/10
      32/1750 [..............................] - ETA: 0s - loss: 0.0792 - acc: 1.0000
    Epoch 7/10
      32/1750 [..............................] - ETA: 0s - loss: 0.1679 - acc: 0.9375
    Epoch 8/10
      32/1750 [..............................] - ETA: 0s - loss: 0.0910 - acc: 1.0000
    Epoch 9/10
      32/1750 [..............................] - ETA: 0s - loss: 0.0593 - acc: 1.0000
    Epoch 10/10
      32/1750 [..............................] - ETA: 0s - loss: 0.1587 - acc: 1.0000
```

You've done something pretty amazing. You should see better than 90% accuracy recognizing handwritten digits, even while using a small training set of only 1750 images!

Course completed!

Recap topics covered:

- Feature selection is not needed, as deep learning models can account for patterns of interactions between features
- Forward and backward propagation helps to adjust weights to improve model performance
- The ReLU activation function is commonly used
- Gradient Descent is used to calculate the weights that will give the lowest value for the loss function
- Stochastic Gradient Descent is to calculate slopes using only a subset of the data (a batch), ie, one batch at a time instead of all data at one go
- Keras model building steps: specify architecture, compile model, fit model, make predictions
- Specifying the optimiser which controls the learning rate
- Use softmax activation function in output layer for classification models
- Keras makes it easy to use some of your data as validation data to evaluate model accuracy
- Early stopping when model is optimised
- Experiment with varying model capacity, by having fewer/more nodes and fewer/more hidden layers

Next steps:

- Start with standard prediction problems on tables of numbers
- Images (with convolutional neural networks) are common next steps
- Kaggle is a great place to find datasets to work with
- Check out the Wikipedia page titled “List of datasets for machine learning research”
- Check out keras.io for excellent documentation
- Keras and Tensorflow repositories on GitHub
- Explore graphical processing unit (GPU) for speedups in model training time
- Need a CUDA compatibility GPU
- For training on using GPU in the cloud: http://bit.ly/2mYQXQb

---

Happy learning!