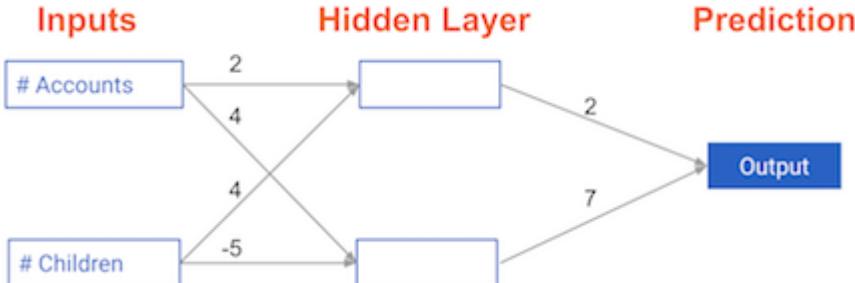


# Chapter 1)

## 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']`. 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()

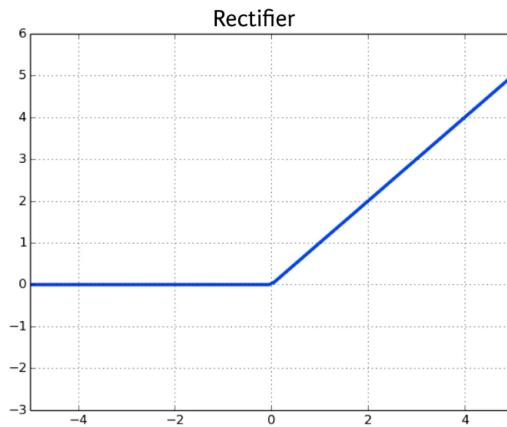
# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_value, node_1_value])

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

# Print output
print(output)
```

## Note)

# ReLU (Rectified Linear Activation)



$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$



2)

## The Rectified Linear Activation Function

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.

```
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)  
52
```

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.

```
# 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))
```

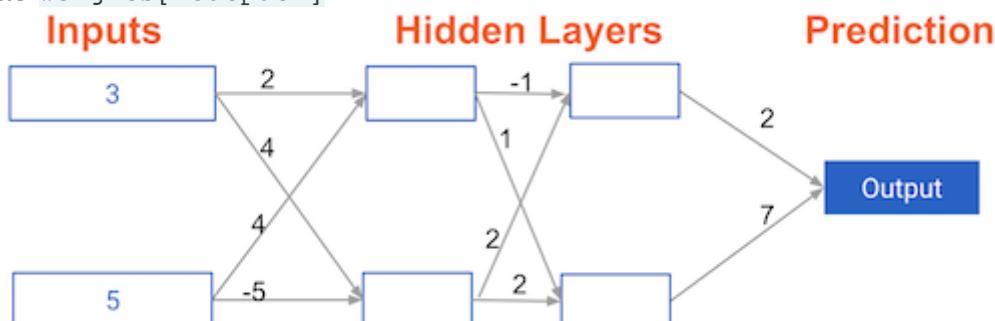
```
# Print results
print(results)
[52, 63, 0, 148]
```

**Solution trên có vấn đề, tại sao lại dùng relu cho output??**

## 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']`.



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

# Return model_output
return(model_output)

output = predict_with_network(input_data)
print(output)

```

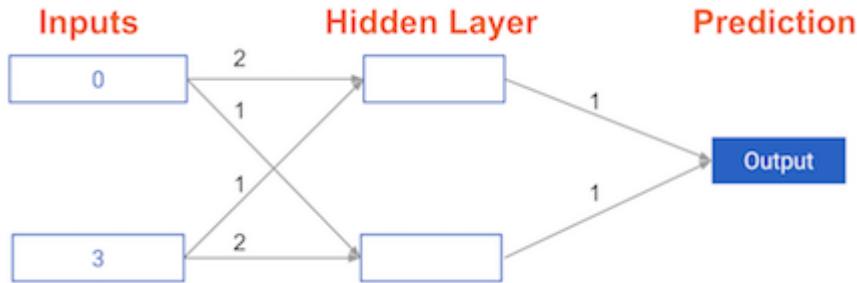
182

## 5)

### 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, 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)
6
print(error_1)
0

```

## 6)

### 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_0
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)

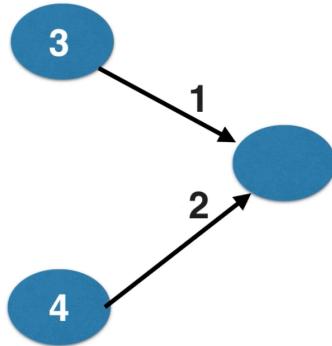
# Print mse_0 and mse_1
print("Mean squared error with weights_0: %f" %mse_0)
Mean squared error with weights_0: 294.000000

print("Mean squared error with weights_1: %f" %mse_1)
```

Mean squared error with weights\_1: 395.062500

Note)

## Network with two inputs affecting prediction



## Code to calculate slopes and update weights

```
In [1]: import numpy as np  
In [2]: weights = np.array([1, 2])  
In [3]: input_data = np.array([3, 4])  
In [4]: target = 6  
In [5]: learning_rate = 0.01  
In [6]: preds = (weights * input_data).sum()  
In [7]: error = preds - target  
In [8]: print(error)  
5
```



## Code to calculate slopes and update weights

```
In [9]: gradient = 2 * input_data * error  
In [10]: gradient  
Out[10]: array([30, 40])  
In [11]: weights_updated = weights - learning_rate * gradient  
In [12]: preds_updated = (weights_updated * input_data).sum()  
In [13]: error_updated = preds_updated - target  
In [14]: print(error_updated)  
-2.5
```



7)

## 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 * 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 = (input_data*weights).sum()
```

```
# Calculate the error: error  
error = preds - target
```

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

```
# Print the slope  
print(slope)
```

[14 28 42]

## 8)

## 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
```

```
# 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)
7

# Print the updated error
print(error_updated)
5.04

```

## 9)

### 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)

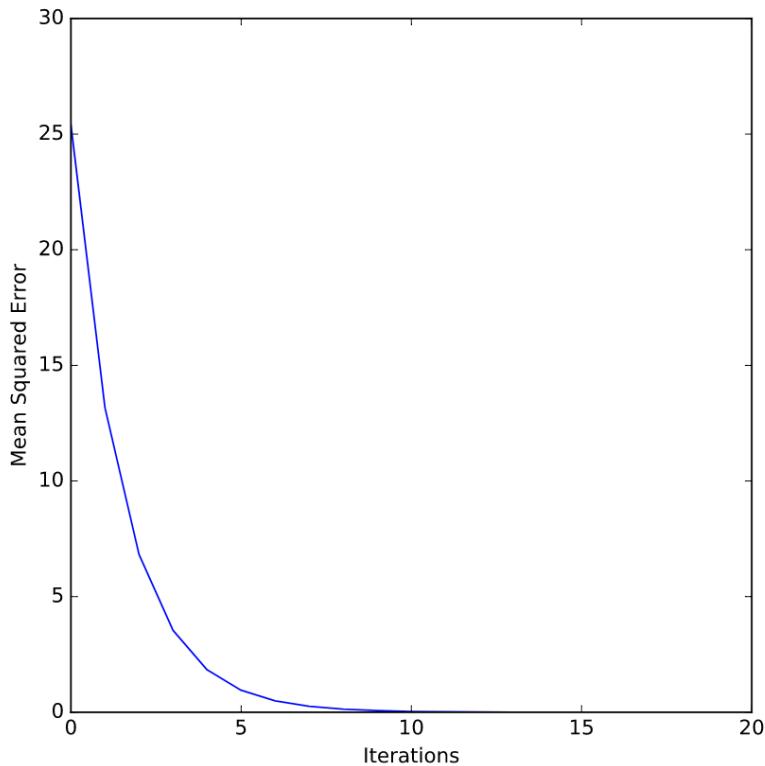
    # Update the weights: weights
    weights = weights - slope * 0.01

    # 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()
```



**Note)**

## Stochastic gradient descent

- It is common to calculate slopes on only a subset of the data ('batch')
- 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
- When slopes are calculated on one batch at a time: stochastic gradient descent

# Chapter 3)

# Model specification

```
In [1]: import numpy as np

In [2]: from keras.layers import Dense

In [3]: from keras.models import Sequential

In [4]: predictors = np.loadtxt('predictors_data.csv', delimiter=',')

In [5]: n_cols = predictors.shape[1]

In [6]: model = Sequential()

In [7]: model.add(Dense(100, activation='relu', input_shape = (n_cols,)))

In [8]: model.add(Dense(100, activation='relu'))

In [9]: model.add(Dense(1))
```

## 1) Specify 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))

```

## Note)

## Why you need to compile your model

- Specify the optimizer
  - Controls the learning rate
  - Many options and mathematically complex
  - “Adam” is usually a good choice
- Loss function
  - “mean\_squared\_error” common for regression

## Fitting a model

```

In [1]: n_cols = predictors.shape[1]

In [2]: model = Sequential()

In [3]: model.add(Dense(100, activation='relu', input_shape=(n_cols,)))

In [4]: model.add(Dense(100, activation='relu'))

In [5]: model.add(Dense(1))

In [6]: model.compile(optimizer='adam', loss='mean_squared_error')

In [7]: model.fit(predictors, target)

```

## 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. 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)
Loss function: mean_squared_error
```

## 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.

```
# 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)

Epoch 1/10
32/534 [>.....] - ETA: 0s - loss: 64.6144
534/534 [=====] - 0s - loss: 35.7872
Epoch 2/10

32/534 [>.....] - ETA: 0s - loss: 31.2072
534/534 [=====] - 0s - loss: 23.2191
Epoch 3/10

32/534 [>.....] - ETA: 0s - loss: 15.3669
534/534 [=====] - 0s - loss: 21.4135
Epoch 4/10

32/534 [>.....] - ETA: 0s - loss: 23.4556
534/534 [=====] - 0s - loss: 21.2680

Epoch 5/10
32/534 [>.....] - ETA: 0s - loss: 12.5044
534/534 [=====] - 0s - loss: 21.1607
Epoch 6/10

32/534 [>.....] - ETA: 0s - loss: 15.2592
534/534 [=====] - 0s - loss: 21.8015

Epoch 7/10

32/534 [>.....] - ETA: 0s - loss: 23.4040
534/534 [=====] - 0s - loss: 22.9461
```

Epoch 8/10

```
32/534 [>.....] - ETA: 0s - loss: 14.8813  
512/534 [=====>..] - ETA: 0s - loss: 21.1836  
534/534 [=====] - 0s - loss: 21.6939
```

Epoch 9/10

```
32/534 [>.....] - ETA: 0s - loss: 13.5766  
534/534 [=====] - 0s - loss: 20.7297
```

Epoch 10/10

```
32/534 [>.....] - ETA: 0s - loss: 13.6335  
534/534 [=====] - 0s - loss: 20.5297
```

**Note)**

## Classification

```
In[1]: from keras.utils import to_categorical  
  
In[2]: data = pd.read_csv('basketball_shot_log.csv')  
  
In[3]: predictors = data.drop(['shot_result'], axis=1).as_matrix()  
  
In[4]: target = to_categorical(data.shot_result)  
  
In[5]: model = Sequential()  
  
In[6]: model.add(Dense(100, activation='relu', input_shape = (n_cols,)))  
  
In[7]: model.add(Dense(100, activation='relu'))  
  
In[8]: model.add(Dense(100, activation='relu'))  
  
In[9]: model.add(Dense(2, activation='softmax'))  
  
In[10]: model.compile(optimizer='adam', loss='categorical_crossentropy',  
...:           metrics=['accuracy'])  
  
In[11]: model.fit(predictors, target)
```

**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!

```
# 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)
Epoch 9/10

32/891 [>.....] - ETA: 0s - loss: 0.4042 - acc: 0.8125
704/891 [=====>.....] - ETA: 0s - loss: 0.5924 - acc: 0.7145
891/891 [=====] - 0s - loss: 0.6024 - acc: 0.7037
Epoch 10/10

32/891 [>.....] - ETA: 0s - loss: 0.5144 - acc: 0.7500
704/891 [=====>.....] - ETA: 0s - loss: 0.5947 - acc: 0.6918
891/891 [=====] - 0s - loss: 0.5875 - acc: 0.6981
```

## Note)

# Saving, reloading and using your model

```
In [1]: from keras.models import load_model  
  
In [2]: model.save('model_file.h5')  
  
In [3]: my_model = load_model('my_model.h5')  
  
In [4]: predictions = my_model.predict(data_to_predict_with)  
  
In [5]: probability_true = predictions[:,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.

```
# 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)
```

```
[ 1.      1.      1.      0.99605548 1.      1.      1.
  0.99999917 1.      1.      1.      1.      1.
  1.      0.99999988 1.      1.      1.      1.      1.
  1.      1.      1.      1.      1.      1.      1.
  1.      1.      1.      0.99999726 1.      1.      0.9999994
  1.      0.99999988 1.      1.      1.      0.99999976
  0.9999994 1.      1.      0.99999905 1.      1.      1.
  1.      1.      1.      1.      0.99999952 1.      1.
  0.99999964 1.      1.      1.      1.      1.      1.
  1.      1.      1.      1.      1.      1.      1.
  0.99990213 1.      1.      1.      1.      1.      1.
  0.99999523 0.99999988 0.99999952 1.      1.      1.      1.
  1.      1.      1.      1.      1.      1.      1.
  1.      1.      ]
```

## Chapter 4)

Note)

## Stochastic gradient descent

```
In [1]: def get_new_model(input_shape = input_shape):

In [2]:     model = Sequential()

In [3]:     model.add(Dense(100, activation='relu', input_shape = input_shape))

In [4]:     model.add(Dense(100, activation='relu'))

In [5]:     model.add(Dense(2, activation='softmax'))

In [6]:     return(model)

In [7]: lr_to_test = [.000001, 0.01, 1]

In [8]: for lr in lr_to_test:

In [9]:     model = get_new_model()

In[10]:    my_optimizer = SGD(lr=lr)

In[11]:    model.compile(optimizer=my_optimizer, loss='categorical_crossentropy')

In[12]:    model.fit(predictors, target)
```

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.

```
# Import the SGD optimizer
from keras.optimizers import SGD

# Create list of learning rates: lr_to_test
lr_to_test = [0.000001,0.01,1]

# 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)

    Epoch 9/10

    32/891 [>.....] - ETA: 0s - loss: 5.5406
    448/891 [======>.....] - ETA: 0s - loss: 6.2961
    864/891 [======>.] - ETA: 0s - loss: 6.1562
    891/891 [======] - 0s - loss: 6.1867

    Epoch 10/10

    32/891 [>.....] - ETA: 0s - loss: 5.5406
    480/891 [======>.....] - ETA: 0s - loss: 6.5480
    891/891 [======] - 0s - loss: 6.1867
```

## Note)

# Early Stopping

```
In [3]: from keras.callbacks import EarlyStopping  
  
In [4]: early_stopping_monitor = EarlyStopping(patience=2)  
  
In [5]: model.fit(predictors, target, validation_split=0.3, epochs=20,  
...:           callbacks = [early_stopping_monitor])
```

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.

```
# 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)  
Epoch 10/10  
  
32/623 [>.....] - ETA: 0s - loss: 0.4892 - acc: 0.7812  
384/623 [======>.....] - ETA: 0s - loss: 0.6249 - acc: 0.7135  
623/623 [======] - 0s - loss: 0.6182 - acc: 0.6870 - val_loss:  
0.5319 - val_acc: 0.7388
```

### 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()`, 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` from `keras.callbacks`.
- Compile the model, once again using '`adam`' as the optimizer, '`categorical_crossentropy`' as the loss function, and `metrics=['accuracy']` to see the accuracy at each epoch.
- Create an `EarlyStopping` object called `early_stopping_monitor`. Stop optimization when the validation loss hasn't improved for 2 epochs by specifying the `patience` parameter of `EarlyStopping()` to be 2.
- Fit the model using the `predictors` and `target`. Specify the number of `epochs` to be 30 and use a validation split of 0.3. In addition, pass `[early_stopping_monitor]` to the `callbacks` parameter.

```
# 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,validation_split=0.3,callbacks=[early_stopping_monitor],epochs=30)
```

Epoch 5/30

```
32/623 [>.....] - ETA: 0s - loss: 0.5691 - acc: 0.7188  
544/623 [=====>....] - ETA: 0s - loss: 0.6443 - acc: 0.6636  
623/623 [=====] - 0s - loss: 0.6428 - acc: 0.6645 - val_loss:  
0.5871 - val_acc: 0.6903
```

Epoch 6/30

```
32/623 [>.....] - ETA: 0s - loss: 0.4315 - acc: 0.8750  
512/623 [=====>....] - ETA: 0s - loss: 0.5790 - acc: 0.7051  
623/623 [=====] - 0s - loss: 0.5944 - acc: 0.6950 - val_loss:  
0.5783 - val_acc: 0.6866
```

Epoch 7/30

```
32/623 [>.....] - ETA: 0s - loss: 0.6202 - acc: 0.6875  
480/623 [=====>....] - ETA: 0s - loss: 0.6076 - acc: 0.7042  
623/623 [=====] - 0s - loss: 0.6390 - acc: 0.7047 - val_loss:  
0.7021 - val_acc: 0.6493
```

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)

## 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.

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

---

None

```
# 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'))

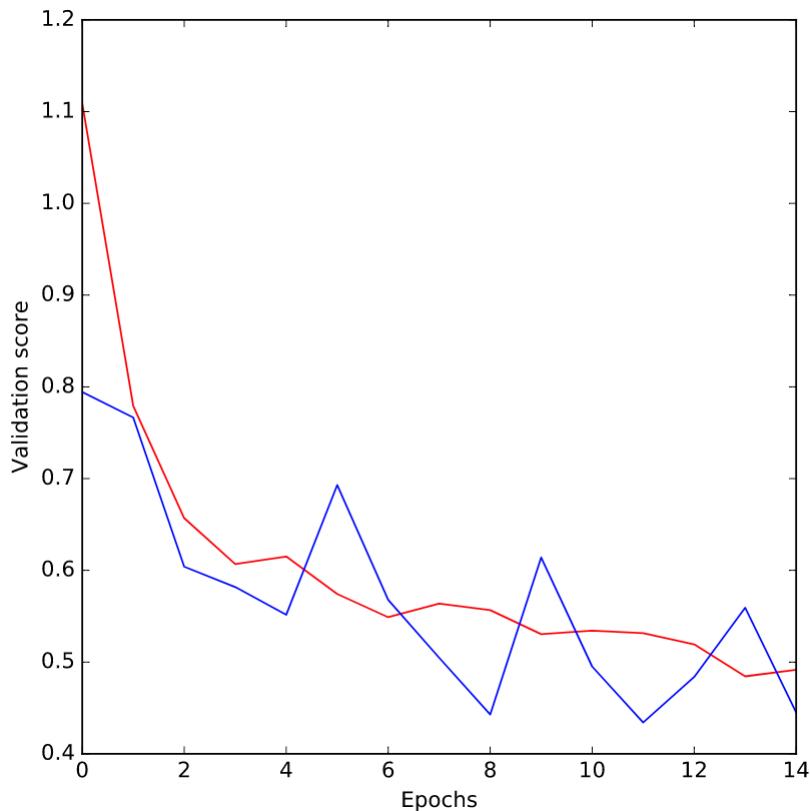
# Compile model_2
model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# 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()
```



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

## 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.

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		

---

None

```
# 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'))

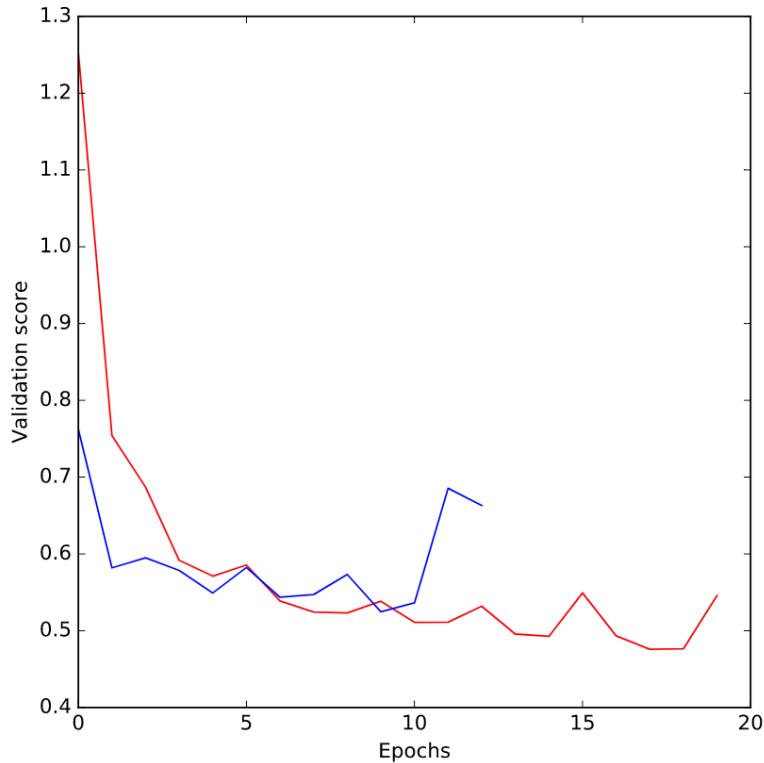
# Compile model_2
model_2.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])

# Fit model 1
model_1_training = model_1.fit(predictors, target, epochs=20,
validation_split=0.4, callbacks=[early_stopping_monitor], verbose=False)

# Fit model 2
model_2_training = model_2.fit(predictors, target, epochs=20,
validation_split=0.4, 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()
```



Note)

## Sequential experiments

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

## 6)

# 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.

```
# 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)
```

Epoch 10/10

```
32/1750 [.....] - ETA: 0s - loss: 0.0777 - acc: 0.9688
448/1750 [=====>.....] - ETA: 0s - loss: 0.0910 - acc: 0.9799
864/1750 [=====>.....] - ETA: 0s - loss: 0.0948 - acc: 0.9792
1312/1750 [=====>.....] - ETA: 0s - loss: 0.1237 - acc: 0.9703
1728/1750 [=====>.....] - ETA: 0s - loss: 0.1234 - acc: 0.9699
1750/1750 [=====] - 0s - loss: 0.1225 - acc: 0.9703 - val_loss:
0.3779 - val_acc: 0.8800
```

You should see better than 90% accuracy recognizing handwritten digits, even while using a small training set of only 1750 images!

- For training on using GPUs in the cloud look here:  
<http://bit.ly/2mYQXQb>