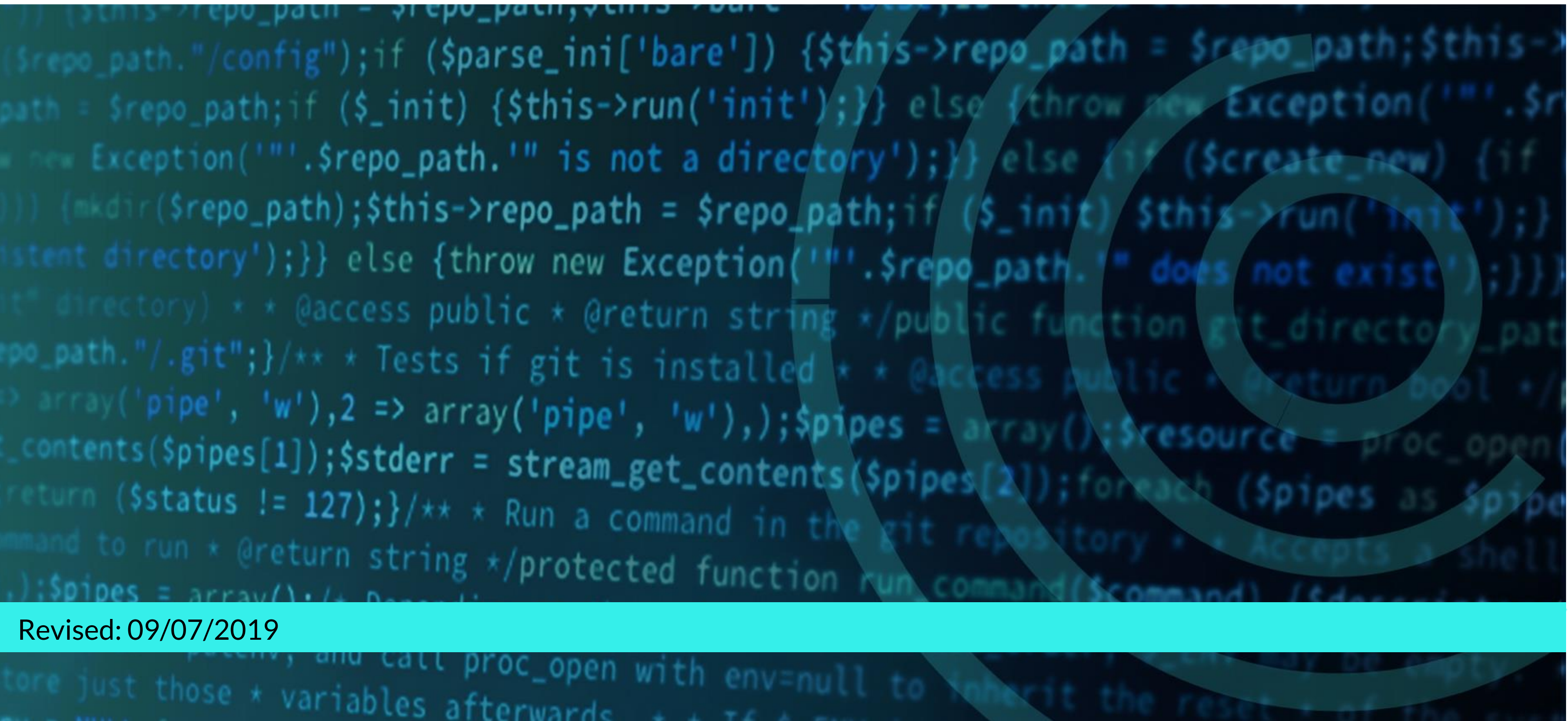
Compiled Content Module 5

MScFE 650

Machine Learning in Finance



Revised: 09/07/2019

Table of Contents

Module 5: Introduction to Deep Learning

Module 5 introduces deep learning, with a focus on the intuitions behind the algorithm, rather than a derivation of backpropagation. After completing this module, you will not only be able to describe, apply and analyze feedforward neural networks, but also understand the intuition behind the modeling of nonlinear patterns.

Unit 1: TensorFlow Basics

There is a vast amount of material freely available on deep learning in general, and TensorFlow in particular. Although we introduce several useful techniques in this notebook, to become very proficient in TensorFlow you will have to explore further on your own.

Useful resources to get you started, include:

1 A good place to start: www.tensorflow.org
2 Aurélien Géron's set of notebooks to complement his equally informative book, *Hands-On Machine Learning with Scikit-Learn & TensorFlow*.

Deep neural networks often involve a huge number of trainable parameters, easily of the order of tens of millions. This large number of parameters requires a vast amount of training data, and the computational resources to handle all of this. TensorFlow is designed for exactly this purpose, as described in the 2015 whitepaper.

One of the important design issues is that of scalability. TensorFlow should be able to take advantage of whatever computational power is available, whether it is the CPU on your cell phone or laptop, multiple GPU's, or distributed machines. To allow this, TensorFlow has a very different design than more traditional programming languages such as Python, Java and C++, among others. In NumPy, for example, the different methods (like SVD, EIG, etc.) reside outside Python where compiled code is run. If not, everything will be slow. In TensorFlow, everything is run outside Python (assuming you use the Python API as we will do in these tutorials).

To achieve this, and also to take advantage of the available computational power, TensorFlow computations proceed in two steps. First, a computational graph is constructed. The nodes represent abstract operations, such as matrix multiplication or function evaluation. Each node receives zero or more data-inputs, and zero or more outputs. The data, in the form of tensors, flow along the edges. During this first stage, the computational graph is constructed symbolically. It is important to keep in mind that, at this stage, no computations whatsoever are performed. In fact, at this point, the graph is empty – that is, it only exists symbolically. The second execution stage is when the graph is populated with values and when the operations are executed. This happens by creating a session.

It is perhaps best to illustrate this with a very simple example:

```
In [0]: # Import the necessary modules
        import IPython.display
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import tensorflow as tf

        plt.style.use('ggplot')
        plt.rcParams['figure.figsize'] = 10, 6
        plt.rcParams['axes.facecolor'] = "0.92"
```

Let's start by initializing two constants, called matrix1 and matrix2

Note that you cannot do anything with these constants. They need to be launched in a session first. You can't even print their values. Try, for example, to run matrix1.eval. TensorFlow constants are exactly what their name implies: they are assigned values and cannot be changed.

It is always a good idea to reset the default graph. If not, a second execution of the cell will add to the previously constructed, and still existing, graph.

```
In [0]: tf.reset_default_graph()

        matrix1 = tf.constant([[3., 3.]])
        matrix2 = tf.constant([[2.],[2.]])
```

Let's look at the constants. It is not quite what you would expect, because at this point they are simply nodes in the computational graph. We'll get hold of their values below.

```
In [0]: print('matrix1:', matrix1, 'matrix2', matrix2)
```

Create a matrix multiplication op

```
In [0]: product = tf.matmul(matrix1, matrix2)
```

Launch graph in a session

This constructs the computational graph so that you can do something with it. If you launch a session, resources such as the CPU or GPU are automatically allocated to the session. You can also do this manually if, for example, you have more than one GPU, or you want to do distributed computing.

```
In [0]: with tf.Session() as sess:
            result = sess.run(product)
```

After the session is closed, all TensorFlow tensors and ops cease to exist. What about the **result** computed above? Let's investigate.

```
In [0]: print(result)
        print(type(result))
```

result is a NumPy array and therefore persists after the session is closed.

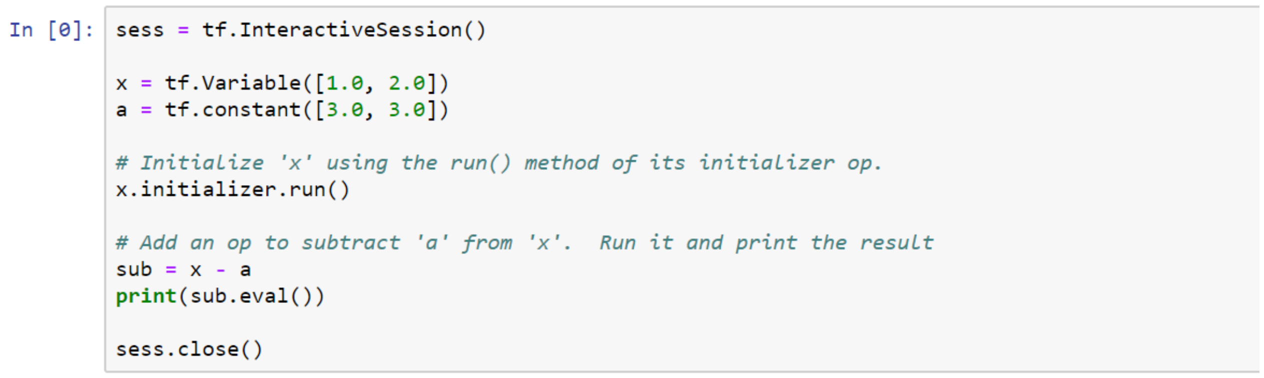
Note:

- All ops lower down in the graph – i.e. the dependencies of `product` in this case, the constant ops, `matrix1` and `matrix2`, are first executed.
- `matmul` is matrix multiplication – not, for example, pointwise multiplication.
- It is important to close the session when you are done in order to save resources. (We used the "`with function`" to do this.)
- It is better to use interactive sessions if you work in iPython or Jupyter.
- Close the session before you run the cell a second time.

Let's find the values of the constants we defined above.

```
In [0]: with tf.Session() as sess:
            print('matrix1:', matrix1.eval(),'matrix2', matrix2.eval())
```

Here is another way of starting a session. This is particularly useful for interactive sessions, as we use here.



```
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# Initialize 'x' using the run() method of its initializer op.
x.initializer.run()

# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = x - a
print(sub.eval())

sess.close()
```

Variables and placeholders

The next important data structures are the **variable** and **placeholder**. You can think of the variables as your trainable parameters. In fact, by declaring something a variable, TensorFlow's `tf.gradients( )` method will automatically calculate the gradient of a cost function with respect to all the specified variables.

All variables need to be explicitly initialized. Also, since variables are trainable parameters, they are assigned initial values during declaration.

Placeholders are, as the name implies, waiting for values to be assigned to them during execution. Note that `tf.gradients( )` ignore placeholders. Typically, you will feed your training or test-data into placeholders during execution, using a dictionary feed. Read more on the subject here.

Note:

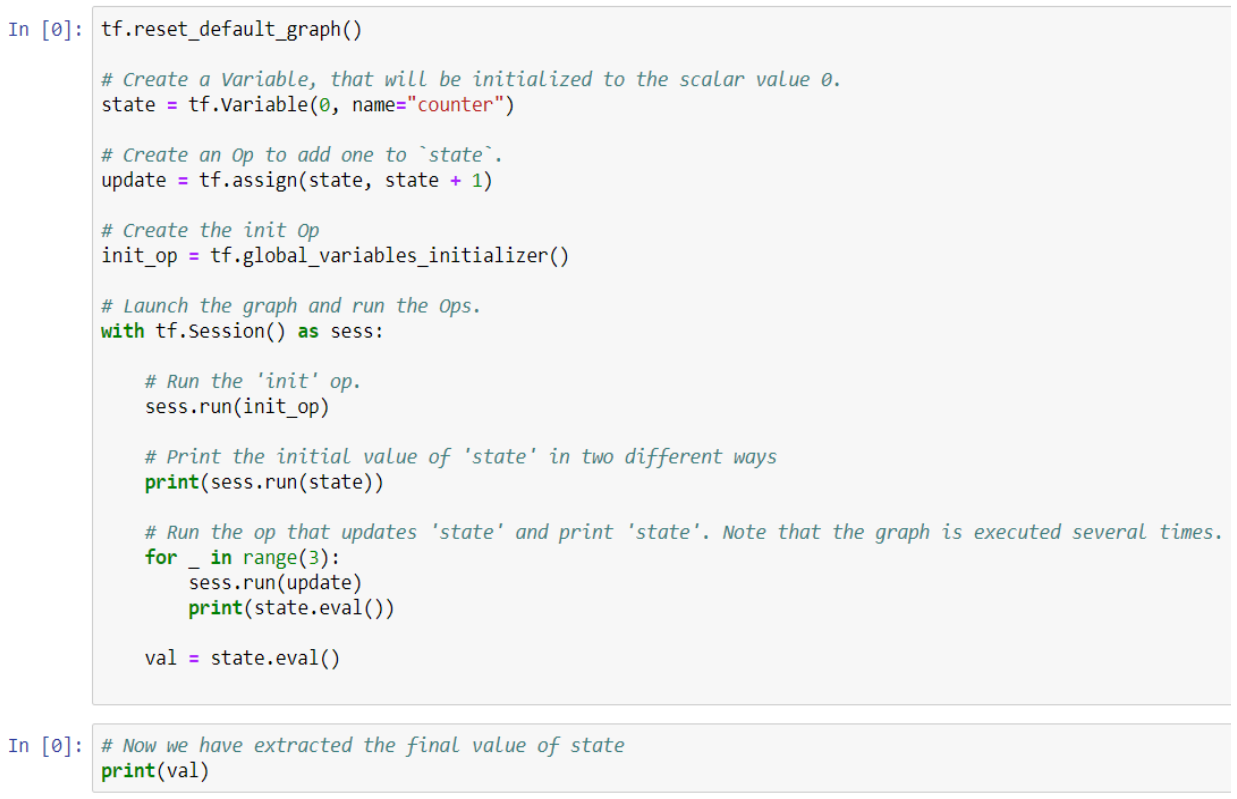
- There are multiple ways of launching sessions. We provide one important example:

```
In [0]: tf.reset_default_graph()

        # Create a Variable, that will be initialized to the scalar value 0.
        state = tf.Variable(0, name="counter")

        # Create an update Op.
        update = tf.assign(state, state + 1)

        # Variables must be initialized by running an `init` Op after having
        # launched the graph.  We first have to add the `init` Op to the graph.
        init_op = tf.global_variables_initializer()

        # Launch the graph and run the ops.
        with tf.Session() as sess:

          # Run the 'init' op.
          sess.run(init_op)

          # Print the initial value of 'state'
          print(state.eval())

          # Run the op that updates 'state' and print 'state'. Note that the graph is executed several times.
          for _ in range(3):
            sess.run(update)
            print(state.eval())
```

Note that all the tensors are cleared when the session is closed. So, how do we get hold of a variable outside the session? One has to assign it to a NumPy array.

```
In [0]: # Note that this will fail, as expected
        print(state.eval())
```

Let's assign it to a NumPy array.

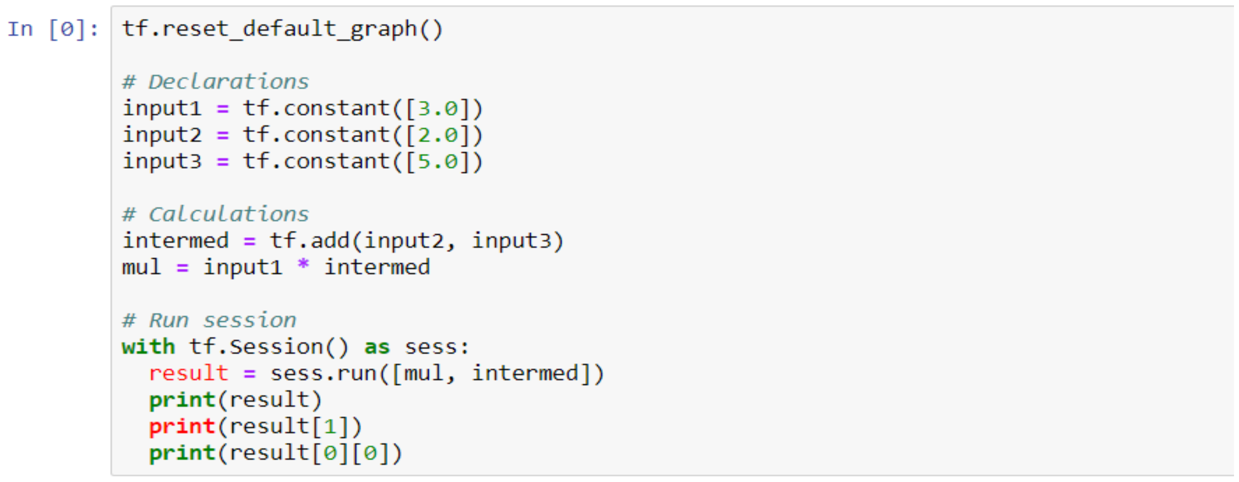


```
In [0]: tf.reset_default_graph()

        # Create a Variable, that will be initialized to the scalar value 0.
        state = tf.Variable(0, name="counter")

        # Create an Op to add one to `state`.
        update = tf.assign(state, state + 1)

        # Create the init Op
        init_op = tf.global_variables_initializer()

        # Launch the graph and run the Ops.
        with tf.Session() as sess:

            # Run the 'init' op.
            sess.run(init_op)

            # Print the initial value of 'state' in two different ways
            print(sess.run(state))

            # Run the op that updates 'state' and print 'state'. Note that the graph is executed several times.
            for _ in range(3):
                sess.run(update)
                print(state.eval())

            val = state.eval()
```

```
In [0]: # Now we have extracted the final value of state
        print(val)
```

Multiple outputs

Below is an example of how you can fetch multiple tensors at the same time. The values (NumPy arrays) of the different tensors are returned in a list. Note that the graph is executed only once.



```
In [0]: tf.reset_default_graph()

        # Declarations
        input1 = tf.constant([3.0])
        input2 = tf.constant([2.0])
        input3 = tf.constant([5.0])

        # Calculations
        intermed = tf.add(input2, input3)
        mul = input1 * intermed

        # Run session
        with tf.Session() as sess:
          result = sess.run([mul, intermed])
          print(result)
          print(result[1])
          print(result[0][0])
```

Placeholders and feed

When creating placeholders, you can feed them values when you execute the graph. The placeholders are fed through dictionaries. Note that you input arrays and that the outputs are also arrays.

```
In [0]: tf.reset_default_graph()

        # Declarations
        input1 = tf.placeholder(tf.float32)
        input2 = tf.placeholder(tf.float32)

        # Calculations
        output = input1 * input2

        # Run session
        with tf.Session() as sess:
            print(sess.run(output, feed_dict={input1: 7., input2: 2. }))
```

Linear regression example

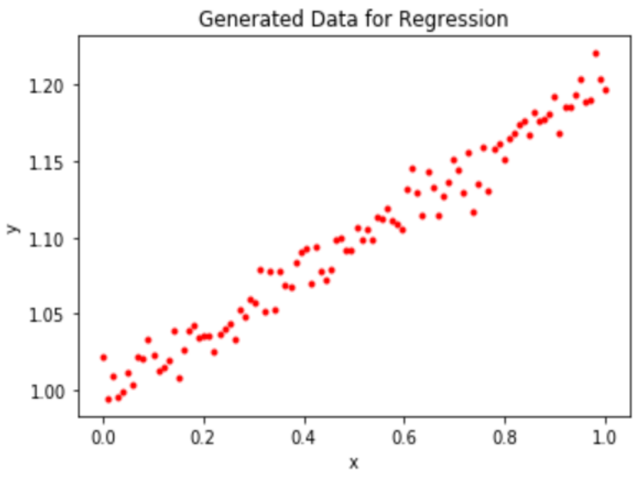
In this case the model is of the form,

$$y = wx + b,$$

and the idea is to estimate w and b given (x, y) pairs.

Although one can solve the system using the normal equations, we are going to use TensorFlow's built-in gradient calculator to solve the system using gradient descent. Let's first generate some data:

```
In [0]: # Generate data
        x_train = np.linspace(0, 1, 100)
        y_train = 0.2 * x_train + 1 + 0.01 * np.random.randn(x_train.shape[0])

        plt.plot(x_train, y_train, 'r.')
        plt.title('Generated Data for Regression')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()
```



```
In [0]: tf.reset_default_graph()

        # Set hyperparameter
        learning_rate = 0.001

        # Model parameters (variables). Initialised with values not close to the correct values.
        w = tf.Variable([-3.], dtype=tf.float32)
        b = tf.Variable([3.], dtype=tf.float32)

        # Model input (placeholders)
        x = tf.placeholder(tf.float32)
        y = tf.placeholder(tf.float32)

        # Linear model
        linear_model = w * x + b

        # Loss
        mse = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares

        #Gradient
        grad = tf.gradients(mse, [w, b])

        # Gradient descent update
        update_w = tf.assign(w, w - learning_rate * grad[0])
        update_b = tf.assign(b, b - learning_rate * grad[1])

        # Define init Op
        init_op = tf.global_variables_initializer()

        # Execution
        with tf.Session() as sess:
            sess.run(init_op)

            # Descend for 1000 steps
            for i in range(1000):
                # print loss at every 100th step
                if i%100 == 0:
                    print('mse = ',sess.run(mse, feed_dict = {x:x_train, y:y_train}))

                sess.run([update_w, update_b], {x:x_train, y:y_train})

            # evaluate training accuracy
            curr_W, curr_b, curr_mse = sess.run([w, b, mse], {x:x_train, y:y_train})
            print("w: %s b: %s loss: %s"%(curr_W, curr_b, curr_mse))
```

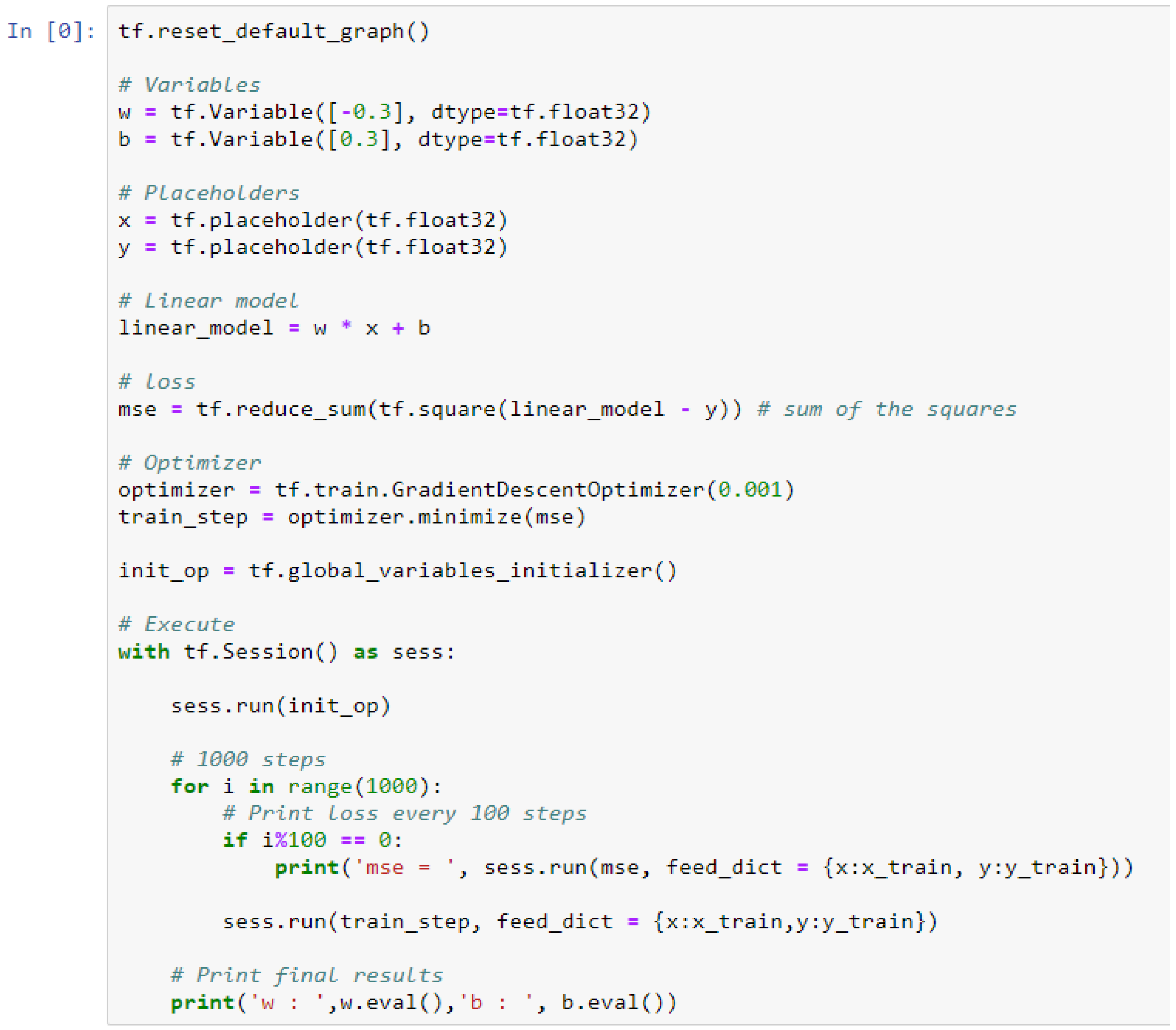
As you can see, the results above are very close to 0.2 W and 1 b that we set. The reason for the slight difference is the noise that we introduced:

`0.01*np.random.randn(x_train.shape[0]).`

Experiment with the learning rate and the number of training steps and see how that affects your accuracy.

TensorFlow's built-in optimizer

Again, we first reset the default graph.



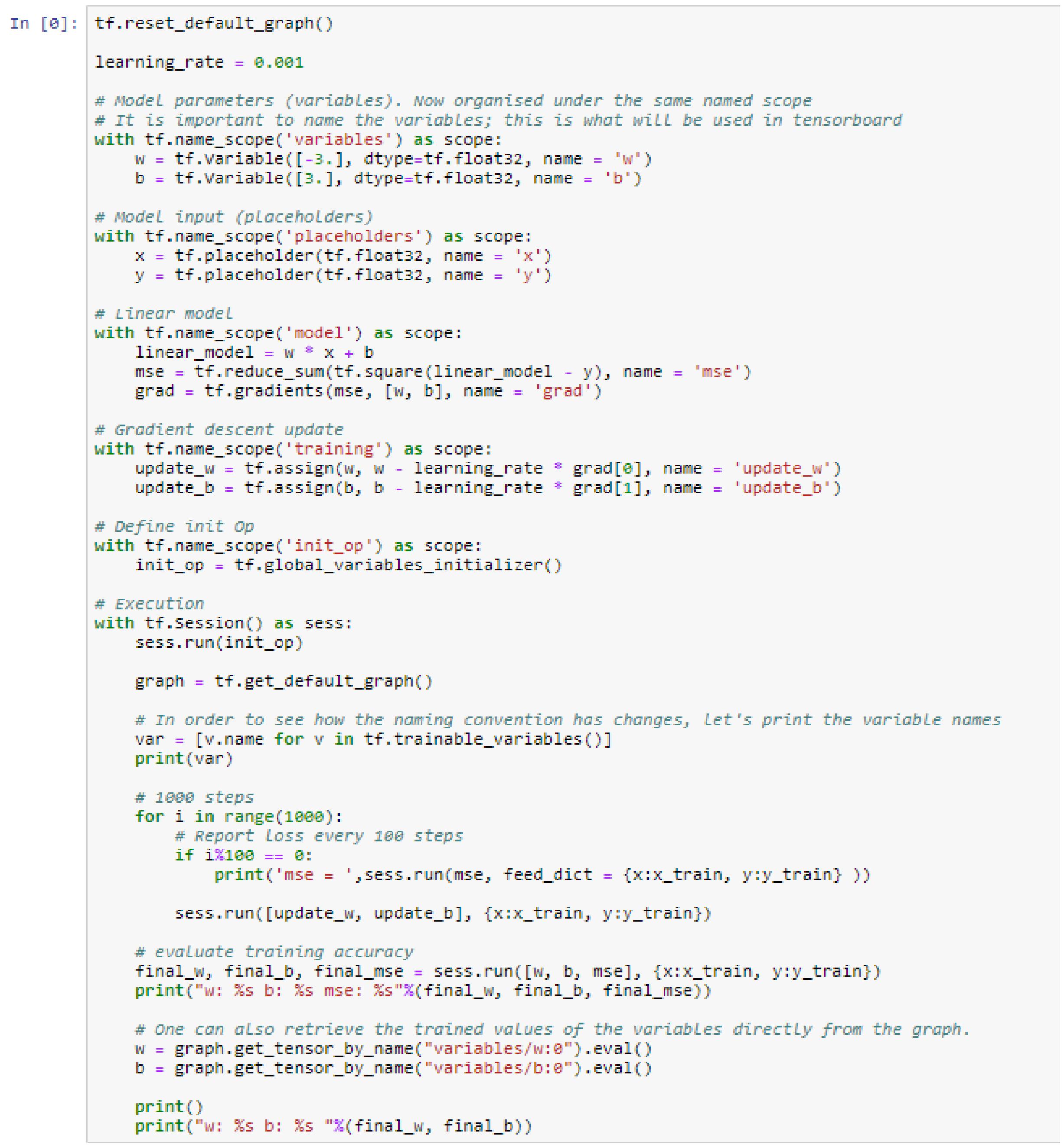
```
In [0]: tf.reset_default_graph()

        # Variables
        w = tf.Variable([-0.3], dtype=tf.float32)
        b = tf.Variable([0.3], dtype=tf.float32)

        # Placeholders
        x = tf.placeholder(tf.float32)
        y = tf.placeholder(tf.float32)

        # Linear model
        linear_model = w * x + b

        # loss
        mse = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares

        # Optimizer
        optimizer = tf.train.GradientDescentOptimizer(0.001)
        train_step = optimizer.minimize(mse)

        init_op = tf.global_variables_initializer()

        # Execute
        with tf.Session() as sess:

            sess.run(init_op)

            # 1000 steps
            for i in range(1000):
                # Print loss every 100 steps
                if i%100 == 0:
                    print('mse = ', sess.run(mse, feed_dict = {x:x_train, y:y_train}))

                sess.run(train_step, feed_dict = {x:x_train,y:y_train})

            # Print final results
            print('w : ',w.eval(),'b : ', b.eval())
```

Visualization using TensorBoard

TensorFlow provides a convenient utility of visualizing the computation graph, called TensorBoard. Computation graphs quickly become quite complicated, and this is reflected in the computation graphs. It is, therefore, important to effectively visualize the computation graph. One of the convenient mechanisms is the `tf.name_scope( )`. This allows one to group tensors that belong together under the same named scope. This is something you probably want to do anyway, to organize your code, regardless of whether you are going to use the visualizations.

Apart from the visualization, TensorBoard also allows one to visualize how quantities – such as the loss function – change during training.

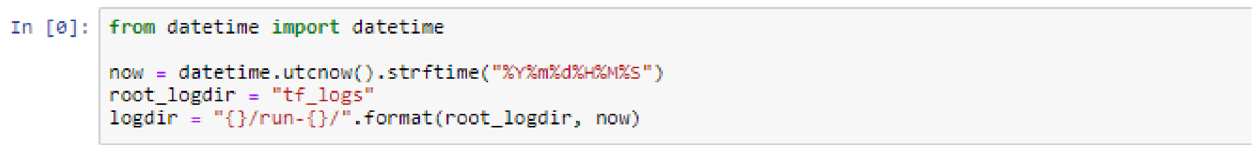
Let's do this systematically, and first just add the name scopes:



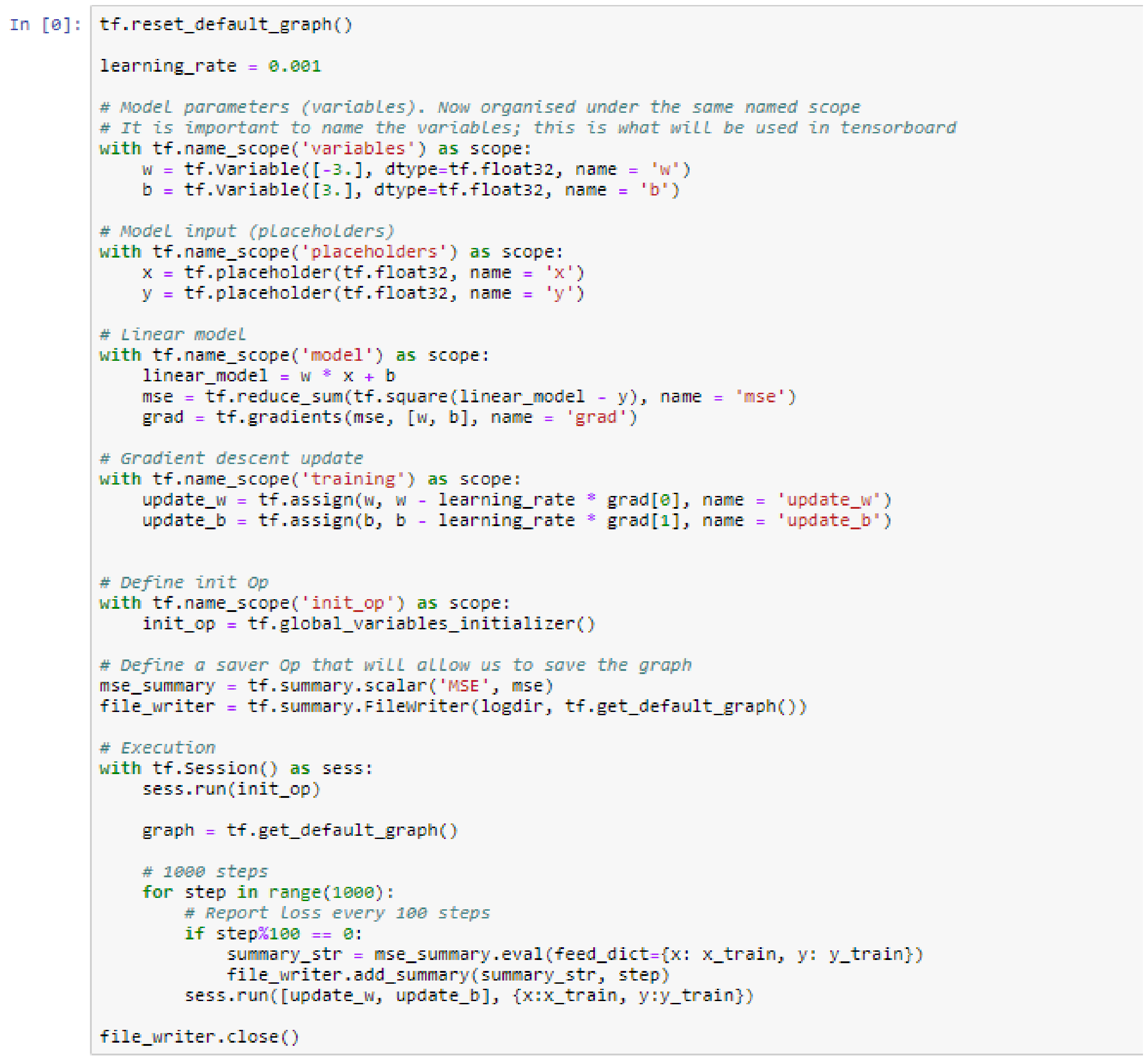
```
In [0]: tf.reset_default_graph()

        learning_rate = 0.001

        # Model parameters (variables). Now organised under the same named scope
        # It is important to name the variables; this is what will be used in tensorboard
        with tf.name_scope('variables') as scope:
            w = tf.Variable([-3.], dtype=tf.float32, name = 'w')
            b = tf.Variable([3.], dtype=tf.float32, name = 'b')

        # Model input (placeholders)
        with tf.name_scope('placeholders') as scope:
            x = tf.placeholder(tf.float32, name = 'x')
            y = tf.placeholder(tf.float32, name = 'y')

        # Linear model
        with tf.name_scope('model') as scope:
            linear_model = w * x + b
            mse = tf.reduce_sum(tf.square(linear_model - y), name = 'mse')
            grad = tf.gradients(mse, [w, b], name = 'grad')

        # Gradient descent update
        with tf.name_scope('training') as scope:
            update_w = tf.assign(w, w - learning_rate * grad[0], name = 'update_w')
            update_b = tf.assign(b, b - learning_rate * grad[1], name = 'update_b')

        # Define init Op
        with tf.name_scope('init_op') as scope:
            init_op = tf.global_variables_initializer()

        # Execution
        with tf.Session() as sess:
            sess.run(init_op)

            graph = tf.get_default_graph()

            # In order to see how the naming convention has changes, let's print the variable names
            var = [v.name for v in tf.trainable_variables()]
            print(var)

            # 1000 steps
            for i in range(1000):
                # Report loss every 100 steps
                if i%100 == 0:
                    print('mse = ',sess.run(mse, feed_dict = {x:x_train, y:y_train} ))

                sess.run([update_w, update_b], {x:x_train, y:y_train})

            # evaluate training accuracy
            final_w, final_b, final_mse = sess.run([w, b, mse], {x:x_train, y:y_train})
            print("w: %s b: %s mse: %s"%(final_w, final_b, final_mse))

            # One can also retrieve the trained values of the variables directly from the graph.
            w = graph.get_tensor_by_name("variables/w:0").eval()
            b = graph.get_tensor_by_name("variables/b:0").eval()

            print()
            print("w: %s b: %s "%(final_w, final_b))
```

Adding the ops needed for TensorBoard

First, we need to define a log directory. Since we probably want a new log for each run, we conveniently use the date-time of the run as the naming convention.



```
In [0]: from datetime import datetime

        now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
        root_logdir = "tf_logs"
        logdir = "{}/run-{}/".format(root_logdir, now)
```

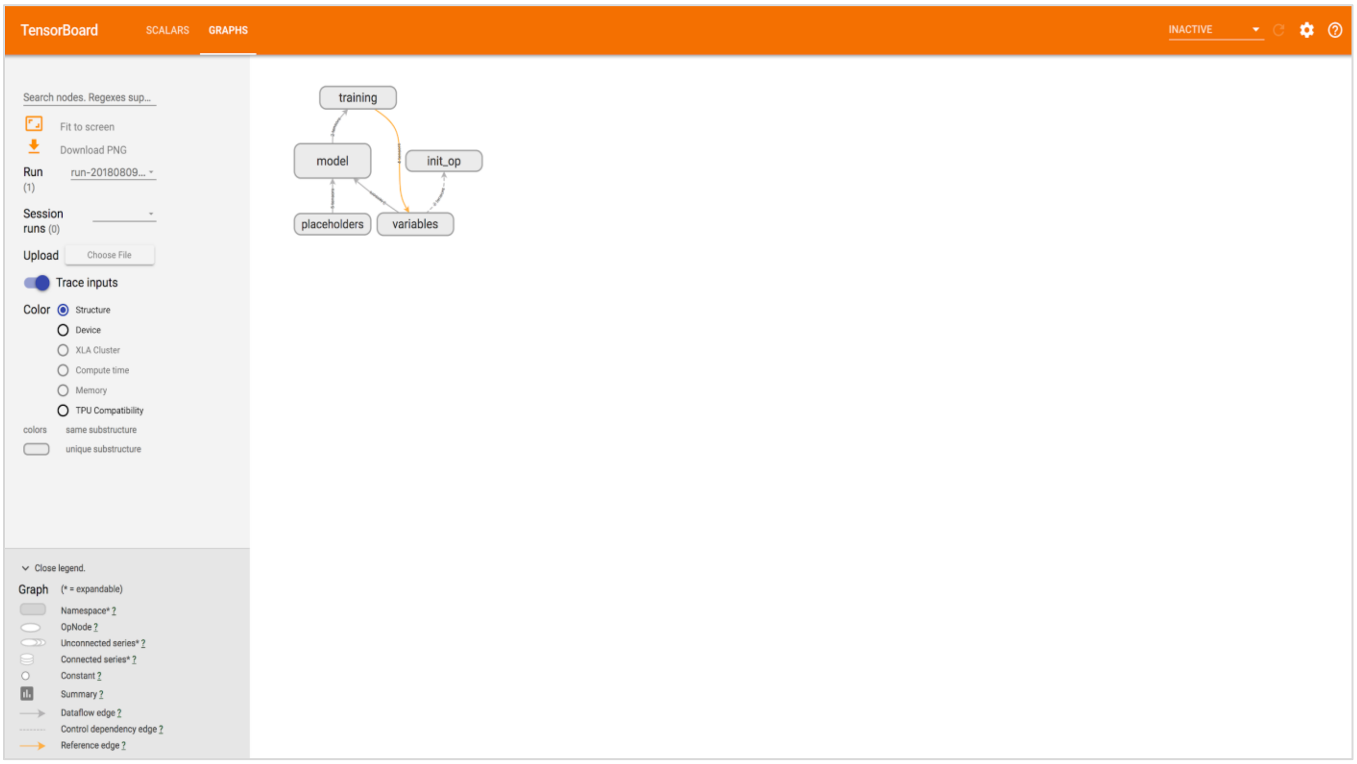


```
In [0]: tf.reset_default_graph()

        learning_rate = 0.001

        # Model parameters (variables). Now organised under the same named scope
        # It is important to name the variables; this is what will be used in tensorboard
        with tf.name_scope('variables') as scope:
            w = tf.Variable([-3.], dtype=tf.float32, name = 'w')
            b = tf.Variable([3.], dtype=tf.float32, name = 'b')

        # Model input (placeholders)
        with tf.name_scope('placeholders') as scope:
            x = tf.placeholder(tf.float32, name = 'x')
            y = tf.placeholder(tf.float32, name = 'y')

        # Linear model
        with tf.name_scope('model') as scope:
            linear_model = w * x + b
            mse = tf.reduce_sum(tf.square(linear_model - y), name = 'mse')
            grad = tf.gradients(mse, [w, b], name = 'grad')

        # Gradient descent update
        with tf.name_scope('training') as scope:
            update_w = tf.assign(w, w - learning_rate * grad[0], name = 'update_w')
            update_b = tf.assign(b, b - learning_rate * grad[1], name = 'update_b')


        # Define init Op
        with tf.name_scope('init_op') as scope:
            init_op = tf.global_variables_initializer()

        # Define a saver Op that will allow us to save the graph
        mse_summary = tf.summary.scalar('MSE', mse)
        file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

        # Execution
        with tf.Session() as sess:
            sess.run(init_op)

            graph = tf.get_default_graph()

            # 1000 steps
            for step in range(1000):
                # Report loss every 100 steps
                if step%100 == 0:
                    summary_str = mse_summary.eval(feed_dict={x: x_train, y: y_train})
                    file_writer.add_summary(summary_str, step)
                sess.run([update_w, update_b], {x:x_train, y:y_train})

        file_writer.close()
```

If you now go to `tf_logs`, the name of the directory you created, you should see one or more (depending how many times you ran the code) files of the form, run-`20180520092455`. You now want to use TensorBoard. TensorBoard is launched on your command line, `tensorboard --logdir=path/to/log-directory`. When is it running, it can be accessed on port 6006 in your browser – that is, at `localhost:6006` (use this link if you get stuck).

You should see something like this:

```
In [0]: IPython.display.Image("./tf_logs/tensorboard.png")
```



Access to the variables in a graph

If you need access to all the ops in a graph, you can do the following:

```
In [0]: for op in tf.get_default_graph().get_operations():
            print(op.name)
```

Even for a very simple graph like this one, there are many ops – too many to inspect by hand if the graph gets even a little more involved. To keep track of the ops, in particular those that you may want to access later, there is a better way to do it. This brings us to the next topic.

Saving and restoring a computational graph

You have trained your model and now you want to take it into production. The one thing you do not want to do is to retrain every time you need to query the model. This means that you need to be able to save and restore the model.

Note that not only do you want to save the trained variables, but also the computation graph itself so that you don't need to build it from scratch every time you want to use it. This also makes it easy to share your trained model, without needing to provide the code of how you built the model in the first place.

The key idea is to add everything necessary to a collection.

To illustrate these ideas, we now turn to a more realistic problem, namely digit classification using the MNIST database.

The MNIST database consists of the order of 70,000 handwritten digits, divided into training-, validation-, and test-sets of 55,000, 5,000, and 10,000 images, respectively.

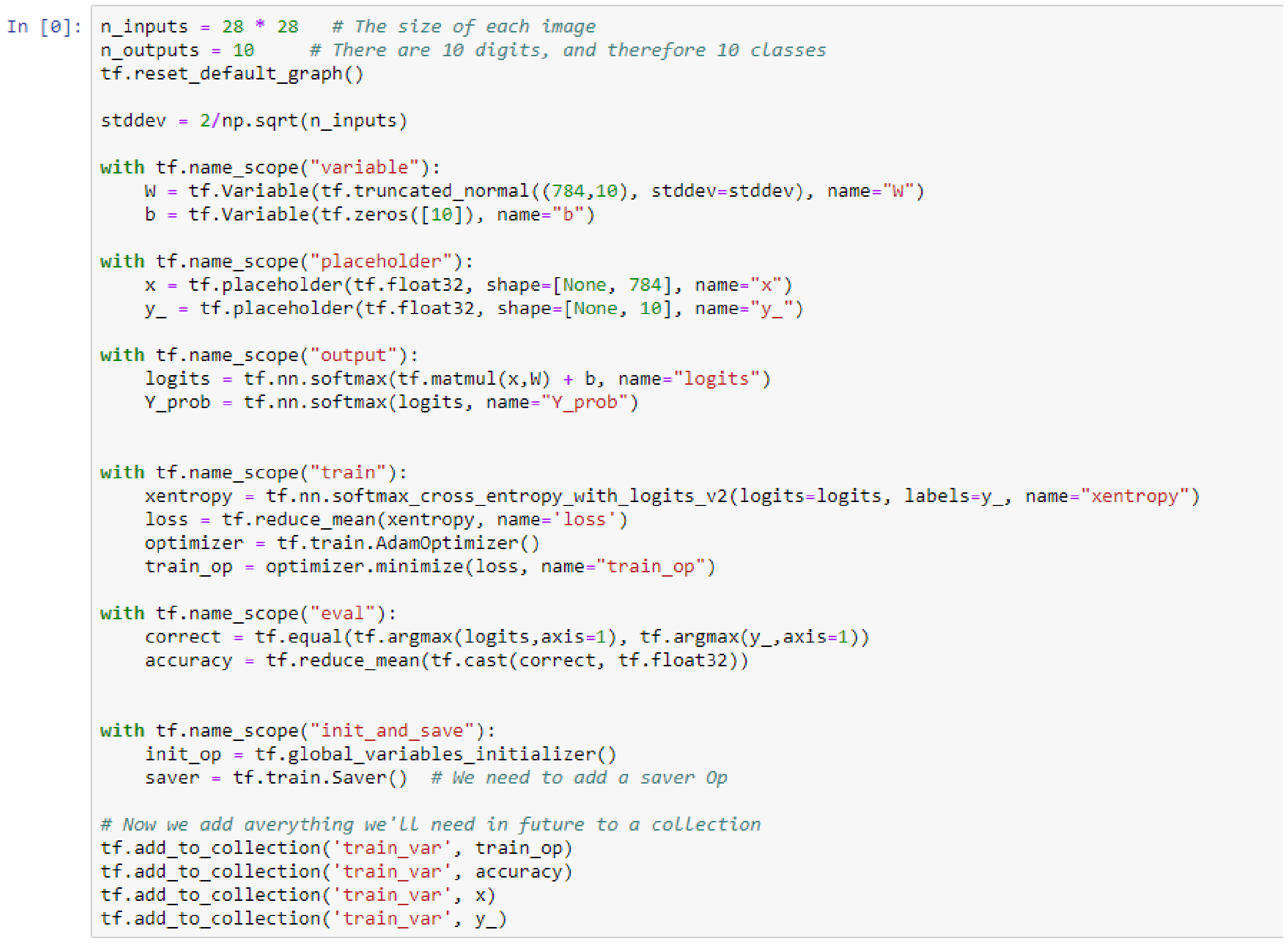
```
In [0]: from tensorflow.examples.tutorials.mnist import input_data
        mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```
In [0]: from ipywidgets import interact

        m,n = mnist.train.images.shape
        number_to_show = 100

        def show_digits(i=0):
            """
            Show some of the digits
            """
            im = np.reshape(mnist.train.images[i], (28,28))
            plt.imshow(im, cmap='viridis')
            plt.title('The digits')

            plt.show()

        w = interact(show_digits, i =(0, number_to_show))
```

Each image in the training set comes with a label. The idea is to build a classifier from the training data – that is, images and labels that can accurately predict the label of a new image.

To keep things simple, we build a straightforward SoftMax classifier. At this point, we are not interested in the accuracy of the system – the idea is to store the trained variables and graph for later use.

Much of the code below should already be familiar. Apart from using cross-entropy instead of MSE, a different optimizer, and a few other smaller details, the main novelty is the addition of the final statements where we add the ops that we plan to use later. Note that it is not necessary to add the trained variables to the collection since they are automatically added.



```
In [0]: n_inputs = 28 * 28   # The size of each image
        n_outputs = 10     # There are 10 digits, and therefore 10 classes
        tf.reset_default_graph()

        stddev = 2/np.sqrt(n_inputs)

        with tf.name_scope("variable"):
            W = tf.Variable(tf.truncated_normal((784,10), stddev=stddev), name="W")
            b = tf.Variable(tf.zeros([10]), name="b")

        with tf.name_scope("placeholder"):
            x = tf.placeholder(tf.float32, shape=[None, 784], name="x")
            y_ = tf.placeholder(tf.float32, shape=[None, 10], name="y_")

        with tf.name_scope("output"):
            logits = tf.nn.softmax(tf.matmul(x,W) + b, name="logits")
            Y_prob = tf.nn.softmax(logits, name="Y_prob")


        with tf.name_scope("train"):
            xentropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=y_, name="xentropy")
            loss = tf.reduce_mean(xentropy, name='loss')
            optimizer = tf.train.AdamOptimizer()
            train_op = optimizer.minimize(loss, name="train_op")

        with tf.name_scope("eval"):
            correct = tf.equal(tf.argmax(logits,axis=1), tf.argmax(y_,axis=1))
            accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))


        with tf.name_scope("init_and_save"):
            init_op = tf.global_variables_initializer()
            saver = tf.train.Saver()  # We need to add a saver Op

        # Now we add averything we'll need in future to a collection
        tf.add_to_collection('train_var', train_op)
        tf.add_to_collection('train_var', accuracy)
        tf.add_to_collection('train_var', x)
        tf.add_to_collection('train_var', y_)
```

```
In [0]: n_epoch = 100

        with tf.Session() as sess:
            sess.run(init_op)

            graph = tf.get_default_graph()
            for epoch in range(n_epoch):

                # One step of the training
                sess.run(train_op, feed_dict={x: mnist.train.images,
                                              y_: mnist.train.labels})

                # Evaluate accuracy on the training set
                acc_train = accuracy.eval(feed_dict={x: mnist.train.images,
                                                     y_: mnist.train.labels})

                # Evaluate the accuracy on the test set. This should be
                # smaller than the accuracy on the training set
                acc_test  = accuracy.eval(feed_dict={x: mnist.test.images,
                                                     y_: mnist.test.labels})

                print(epoch, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)

            # Print the variable names for later access.
            var = [v.name for v in tf.trainable_variables()]
            print(var)

            weights = graph.get_tensor_by_name("variable/W:0").eval()
            bias   = graph.get_tensor_by_name("variable/b:0").eval()

            # Save the current values of the variables, the meta graph, and the collection
            # variables to files.
            save_path = saver.save(sess,"./test/test.ckpt")
            saver.export_meta_graph(filename='./test/test.meta',
                                    collection_list=["train_var"])
        # print('weights:', weights)
        # print('bias:', bias)
```

It might be interesting to look at the weights as images. Perhaps you will agree that the weights largely seem to emphasize those regions of the image that best distinguish between the digits. However, this was a bit of a digression – let's continue.

```
In [0]: m,n = weights.shape

        def show_weights(i=0):
            im = weights.T[i].reshape([28,28])
            plt.imshow(im, cmap='viridis')
            plt.title('The weigths of filter '+str(i))
            plt.show()

        w = interact(show_weights, i =(0, n-1))
```

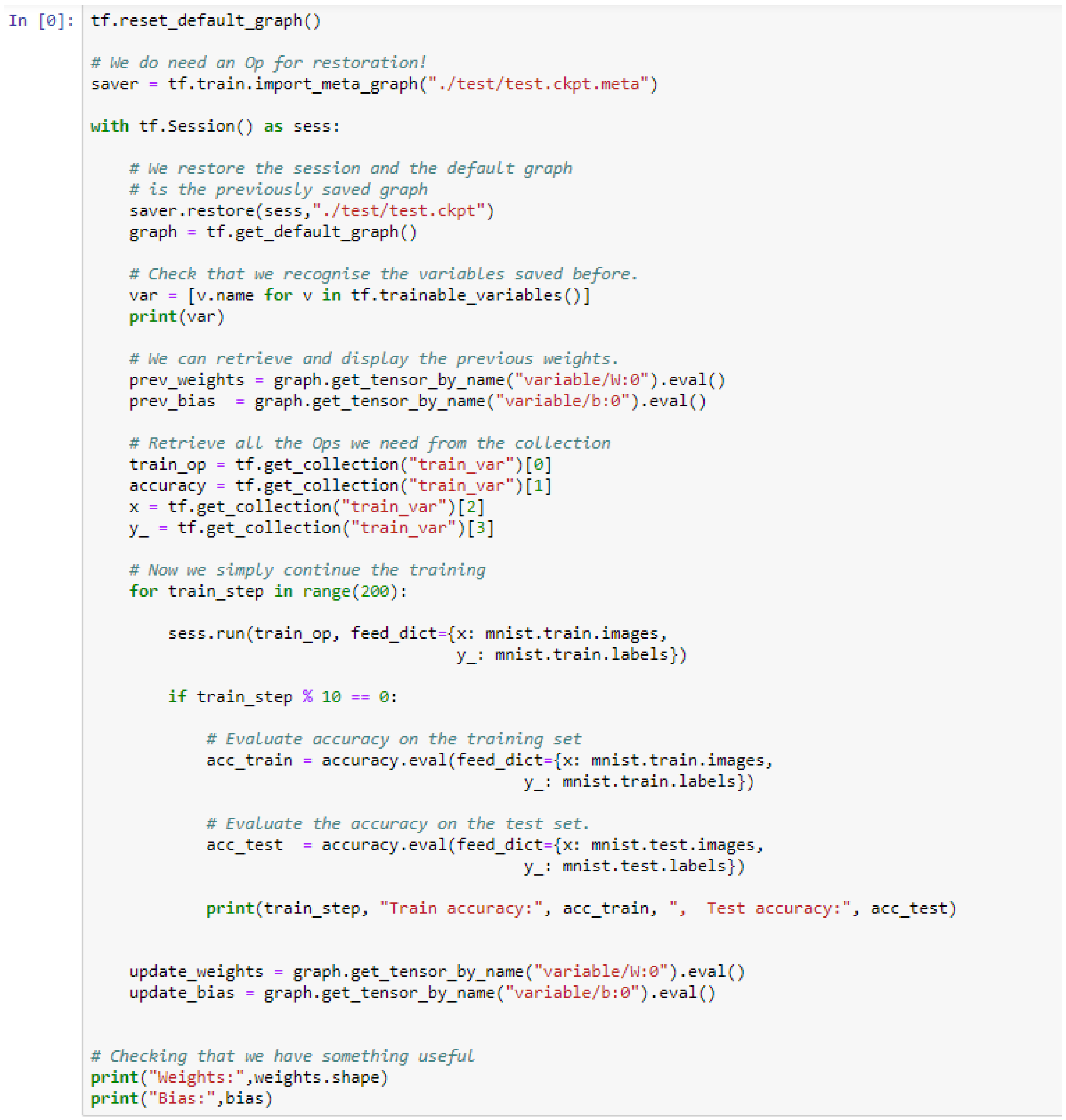
Restoring the variables and the graph

The previous training used only 100 training steps. This is far from the accuracy that can be achieved, even by the basic SoftMax classifier. The task we now set ourselves is to retrieve the graph, as well as the previously trained variables, and continue the training.

Crucially, we need to get access to the placeholders in order to feed the training data to the training process.

Note that we, as usual, clear the previous graph. We want to make sure we restore the previously saved graph.

Also note that we skip the construction phase in its entirety. Everything we need should be restored from what we saved before.



```
In [0]: tf.reset_default_graph()

        # We do need an Op for restoration!
        saver = tf.train.import_meta_graph("./test/test.ckpt.meta")

        with tf.Session() as sess:

            # We restore the session and the default graph
            # is the previously saved graph
            saver.restore(sess,"./test/test.ckpt")
            graph = tf.get_default_graph()

            # Check that we recognise the variables saved before.
            var = [v.name for v in tf.trainable_variables()]
            print(var)

            # We can retrieve and display the previous weights.
            prev_weights = graph.get_tensor_by_name("variable/W:0").eval()
            prev_bias  = graph.get_tensor_by_name("variable/b:0").eval()

            # Retrieve all the Ops we need from the collection
            train_op = tf.get_collection("train_var")[0]
            accuracy = tf.get_collection("train_var")[1]
            x = tf.get_collection("train_var")[2]
            y_ = tf.get_collection("train_var")[3]

            # Now we simply continue the training
            for train_step in range(200):

                sess.run(train_op, feed_dict={x: mnist.train.images,
                                              y_: mnist.train.labels})

                if train_step % 10 == 0:

                    # Evaluate accuracy on the training set
                    acc_train = accuracy.eval(feed_dict={x: mnist.train.images,
                                                         y_: mnist.train.labels})

                    # Evaluate the accuracy on the test set.
                    acc_test  = accuracy.eval(feed_dict={x: mnist.test.images,
                                                         y_: mnist.test.labels})

                    print(train_step, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)


            update_weights = graph.get_tensor_by_name("variable/W:0").eval()
            update_bias = graph.get_tensor_by_name("variable/b:0").eval()


        # Checking that we have something useful
        print("Weights:",weights.shape)
        print("Bias:",bias)
```

The accuracy has improved but is actually still poor, by modern standards. Deep neural networks have near-perfect solutions.

```
In [0]: m,n = update_weights.shape

        def show_update_weights(i=0):
            """
            Show the weights
            """
            im = update_weights.T[i].reshape([28,28])
            plt.imshow(im, cmap='viridis')
            plt.title('The weigths of filter '+str(i))

            plt.show()

        w = interact(show_update_weights, i =(0, n-1))
```

Unit 2: Introduction to Deep Learning

We are in the fortunate situation in that there is an abundance of excellent teaching material available on deep learning. In our discussion of this topic, we will not hesitate to ask you to read through some of the material yourself, or to consult some of the more highly recommended websites. In deep learning, perhaps even more than in other fields, one learns by doing things oneself. This course simply offers guidance.

Resources:

1. We'll make good use of Michael Nielson's online textbook.
2. Aurélien Géron's *Hands-On Machine Learning with Scikit-Learn and Tensorflow* is one of the best resources available.
3. An interesting read for clarity and depth of understanding is Christopher Olah's blog. Unfortunately, however, it does not cover the entire field.
4. The classic book on deep learning by some of the most important contributors in the field, is by Ian Goodfellow, Yoshua Bengio and Aaron Courville.
5. Along the way, we'll also ask you to read some of the seminal papers in the field.

General background

The birth of artificial neural networks can be traced at least as far back as the perceptron of Frank Rosenblatt in 1958. The excitement in neural networks reached a peak during the 1980's, after which interest started to wane as methods like support vector machines (SVMs) started to find favor among researchers and practitioners. All of that changed around 2010. The revival of artificial neural networks started inauspiciously enough when Fei-Fei Li conceived the idea of gathering a really large dataset of images, called ImageNet.

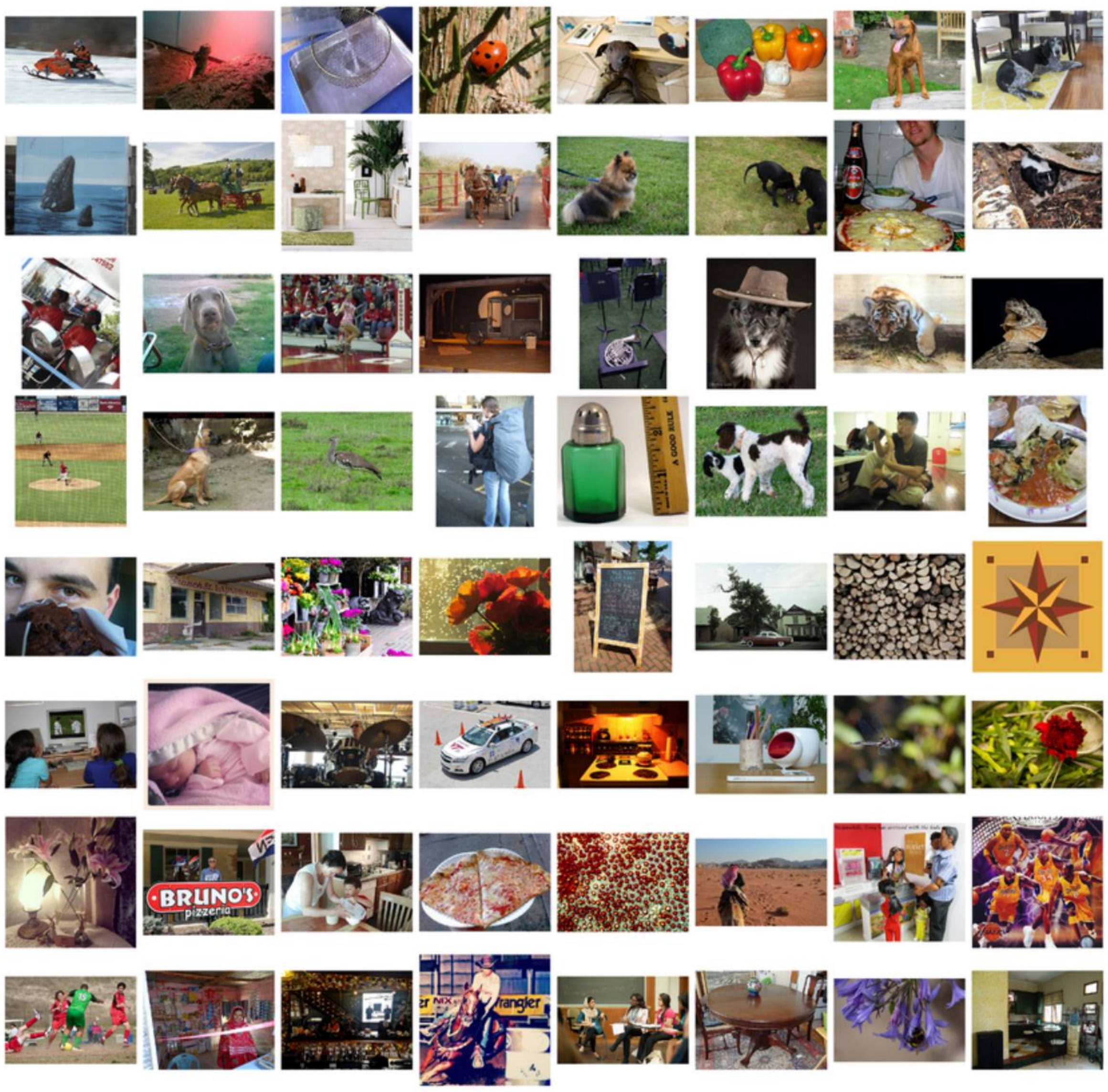


Figure 1: Images from ImageNet

Her idea was simple: better and more data will lead to better algorithms. Building ImageNet was not simple, however. The solution was Amazon's Mechanical Turk, an internet-based crowdsourcing endeavor. Even so, it took two-and-a-half years to build and label the 3.5 million images into 5,247 different object categories. Since 2010, the ImageNet **Large Scale Visual Recognition Challenge** (ILSVRC) has used a subset with 'only' 1,000 different categories. More precisely, each image contains an object, or even more than one object, belonging to one of the 1,000 object categories. The challenge is to assign the dominant object to the appropriate category.

The best results obtained in 2011 had an error rate of about 25%. The very next year that dropped to an astonishing 16% by the deep neural network of Krizhevsky, Sutskever and Hinton. Today, deep neural networks perform better than humans, on a problem long-considered to be a uniquely human endeavor. Figure 2 (below) shows the top five classification results from the Hinton paper, together with the correct classification.

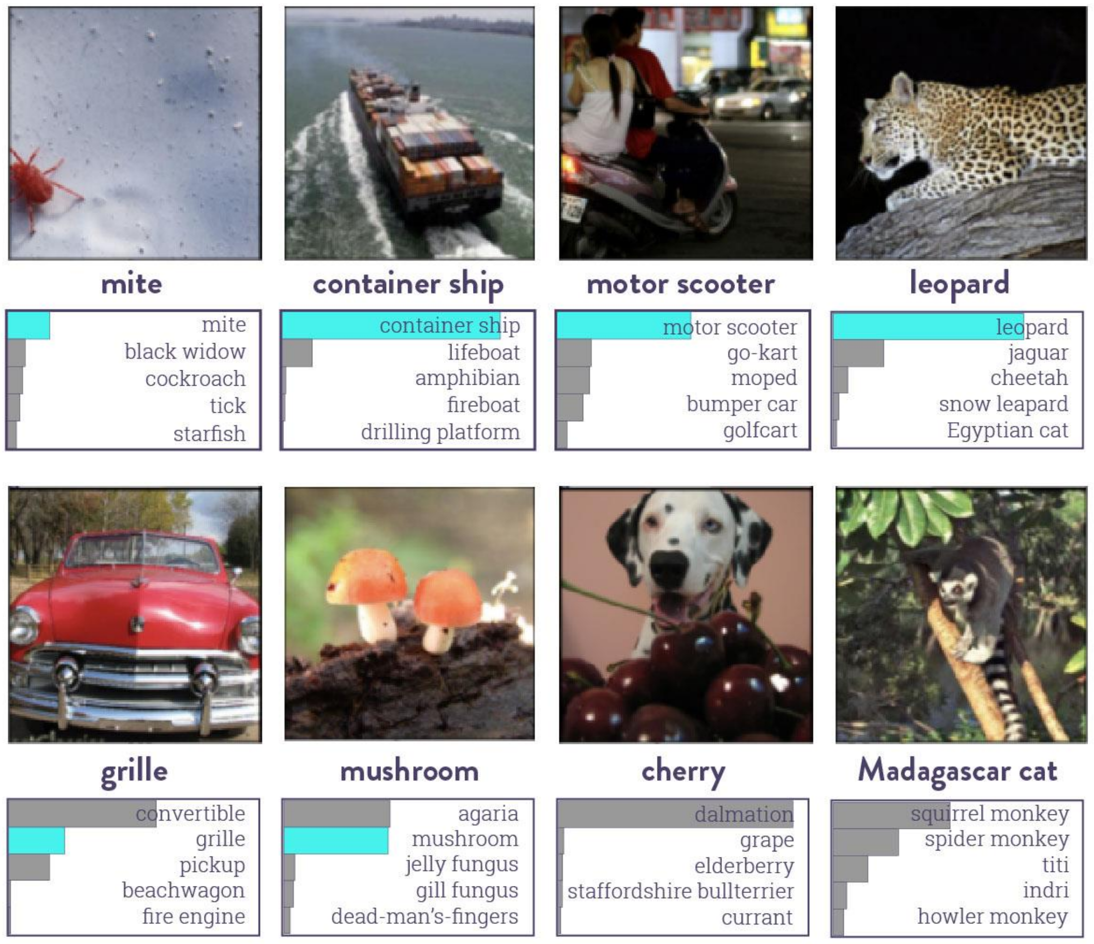


Figure 2: Classification results from ImageNet

What has brought about this remarkable progress? It is probably not possible to point to any single factor, but three things do stand out. The first is the availability of a massive amount of training data, as pointed out above. The second is the availability of sufficient computer power to process all this data, in the form of sufficiently powerful GPU's. These two advances made it possible to train deep networks. Of course – and this is the third factor – there were also significant advances in the algorithms themselves. Once it was clear that it is possible to train deep neural networks, innovative neural network architectures were developed.

One should also point out another significantly different approach to the problem. In more traditional approaches, researchers spent much effort on so-called feature engineering. The idea was to encode domain knowledge by transforming the data into something that best reflects human understanding of the problem. With the availability of massive amounts of data, there is little or no emphasis on feature engineering; although it is possible to use human engineered features, if that is desirable. The idea behind a deep neural network is that the network should extract the relevant features, moving from low to higher-level features as the data flows through the network.

Neural network basics

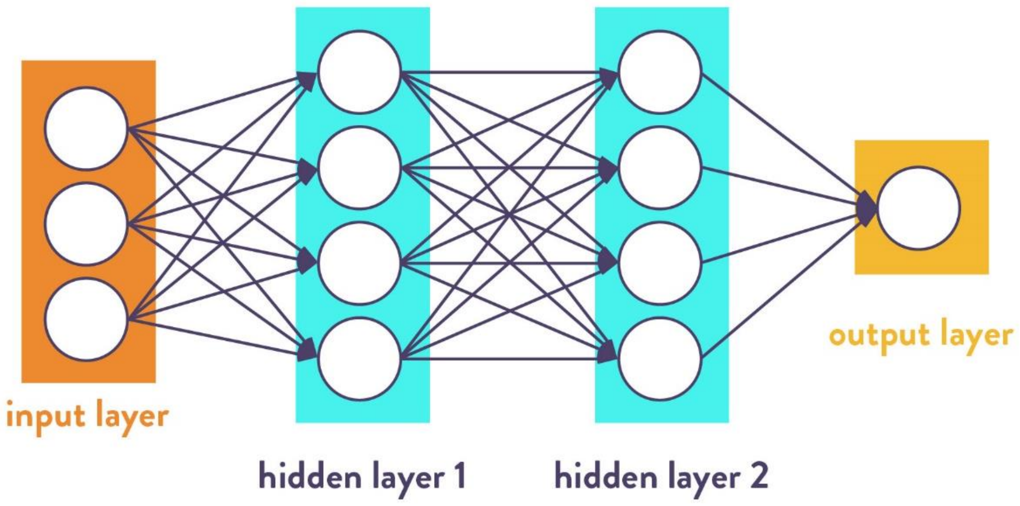


Figure 3: A neural network

(Adapted from CS231n Convolutional Neural Networks for Visual Recognition, 2018)

Figure 3 (above) shows a neural network with an input layer, two hidden layers, and an output layer. This is the basic structure of all **feedforward** neural networks. The main differences are in what exactly happens in the different layers.

The output layer returns the solution of the problem we are after. In neural networks, one can solve one of two different problems – namely, the **classification problem**, or the **regression problem**. The difference between the two lies in the nature of the output. For classification, the output is discrete; for regression, the output is continuous. You may think that this severely restricts the scope of the problems solvable by neural networks, but that is only partially true. Researchers are discovering more and more innovative ways of casting their problems as either a classification or regression problem.

SoftMax classification as a neural network

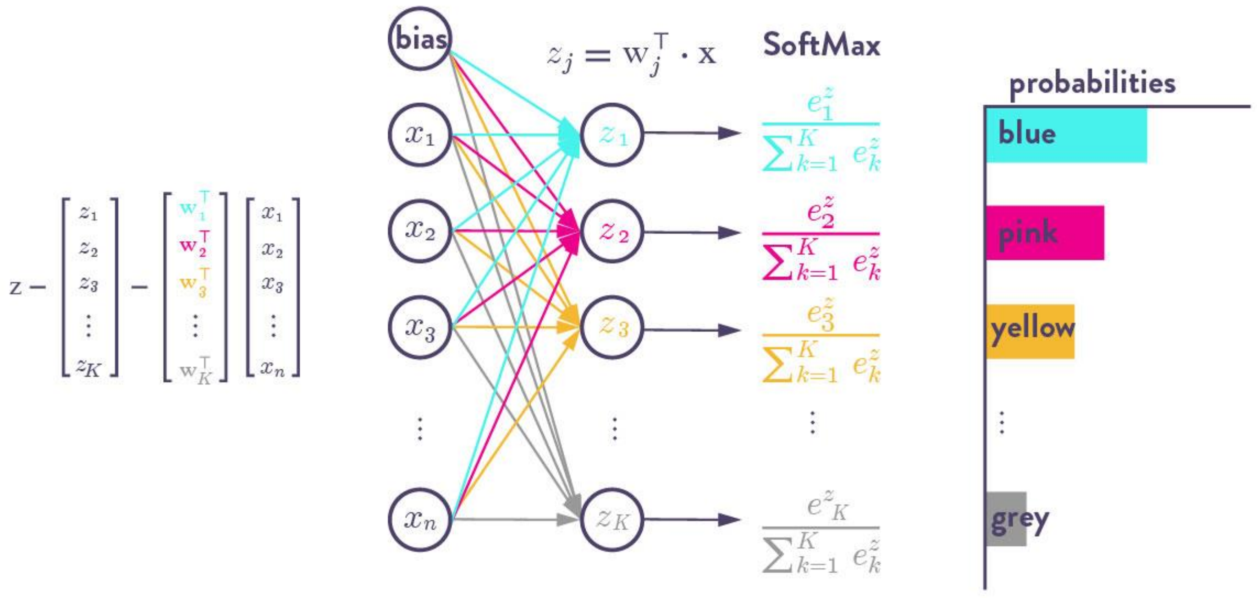


Figure 4: Multi-class classification with NN and SoftMax function

(Adapted from Parellada, 2017)

Figure 4 shows the by-now-familiar SoftMax classifier, interpreted as a single-layer neural network.

Let us briefly look at the ingredients again. The input is a d-dimensional vector $\mathbf{x} = [x_1, \cdots, x_d]^T$. The single layer is a fully connected linear layer with neurons,

$$z_j = \sum_{i=1}^{d} w_{ij}\, x_i + b_j \,,$$

or more compactly,

$$z_j = \mathbf{w}_j^T \mathbf{x} \,,$$

where we define $\mathbf{w}_j = [w_{1j}, \cdots, w_{dj}, b_j]^T$. The logits, z_j, are then passed through the SoftMax function as indicated in the figure.

The SoftMax function is a perfectly good classifier and, indeed, we'll continue to use it even as we extend our neural networks. The main difference will be that we will add more layers and inject nonlinearities into the layer. There are a few other things that will prove necessary, but we will cover these later.

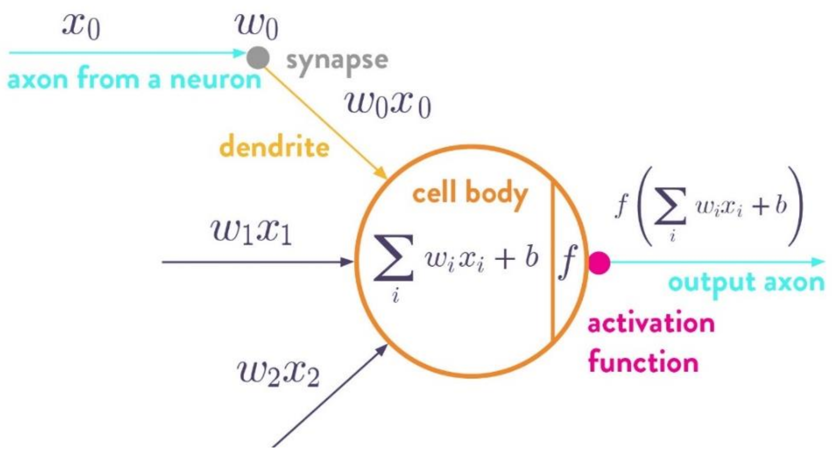


Figure 5: Neuron model

(Adapted from: CS231n Convolutional Neural Networks for Visual Recognition, 2018)

Figure 5 shows the basic structure of a single neuron. It takes a linear combination plus offset of the previous layer, as explained in the image above, followed by a nonlinear activation function. There are several activation functions used in practice, some of which we will engage with later. Their purpose is to inject nonlinearities into the system. This is, of fundamental importance. Without nonlinearities, there will be no point in stacking different layers. The end result can still be expressed as a single linear layer.

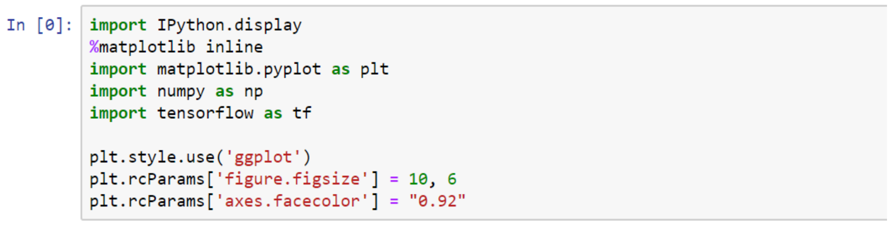
The sigmoid activation function

One of the most important – yet not always the best – activation functions is the **sigmoid activation function**,

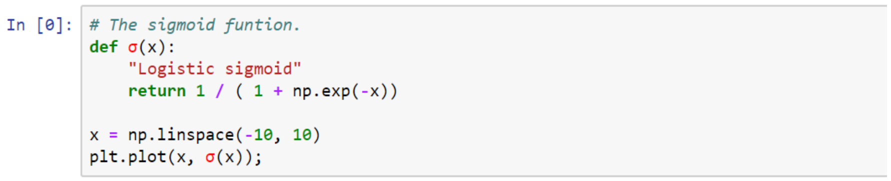
$$\sigma(x) = \frac{1}{1 + \exp(-x)} \,,$$

with shape shown in a code cell below (the s-like shape). Note that it maps any real number, $-\infty < x < \infty$ to a real number in $(0,1)$. Thus, it associates something we can think of as a probability to any real number x. Here, we are not as much interested in this fact as in another property: equipped with the sigmoid activation function, a neural network becomes a universal approximator. To learn more on the topic, refer to this article. This simply means that it is possible to approximate any (sufficiently smooth) function with the neural network. In the rest of this notebook we explore the intuition behind this idea.

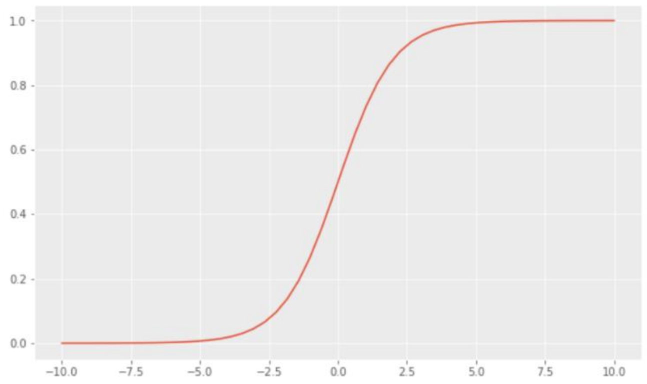
First, we import a few useful functions:



```
In [0]: import IPython.display
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import tensorflow as tf

        plt.style.use('ggplot')
        plt.rcParams['figure.figsize'] = 10, 6
        plt.rcParams['axes.facecolor'] = "0.92"
```



```
In [0]: # The sigmoid funtion.
        def σ(x):
            "Logistic sigmoid"
            return 1 / ( 1 + np.exp(-x))

        x = np.linspace(-10, 10)
        plt.plot(x, σ(x));
```



A neural network as universal approximator

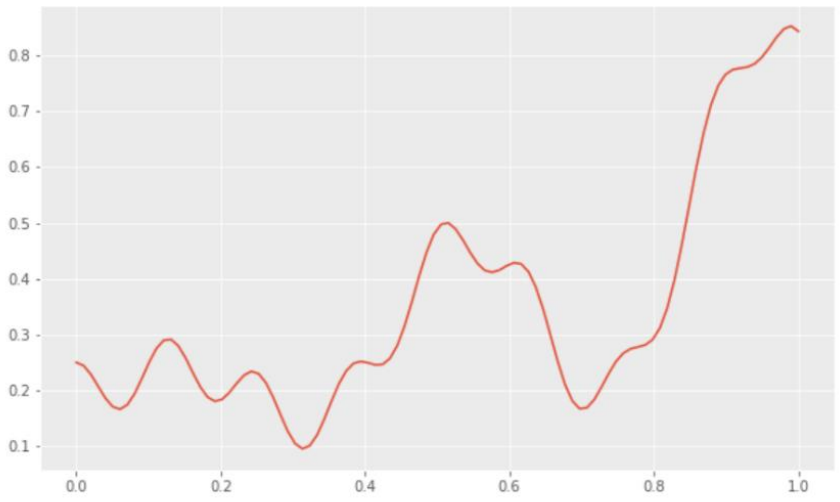
The function that we will try to approximate is given by,

$$f(x) = 0.2 + 0.4x^2 + 0.3x\, sin(15x) + 0.05\, cos(50x).$$

Example

Let us see how we can approximate the following function, using TensorFlow.

```
In [0]: def f(x):
            return 0.2 + 0.4 * x**2 + 0.3 * x * np.sin(15 * x) + 0.05 * np.cos(50 * x)

        resolution = 100
        x = np.linspace(0,1,resolution)
        y = f(x)

        # Plot
        plt.plot(x,y)
        plt.show()# The neural network.
```



The neural network

The neural network we will use consists of the following:

1 Single hidden layer with sigmoid activations.

2 Input: `x = np.linspace(0, 1 ,100).`

3 Neural network output: $\sigma(w\mathbf{x} + \mathbf{b})$, where $\mathbf{b}$ is a vector of offsets.

4 Target output: $y = f(x)$.

5 For simplicity, we use only a single, tied weight w which we fix at $w = 100$.

6 Each neuron in the single hidden layer is therefore given by $zj = \sigma(w * X + bj)$.

Let us see what kind of flexibility this network allows us.

Note: We are not trying to fit the function yet; we are just looking at the shape of the hidden neurons.

Set up the computational graph

Construction phase

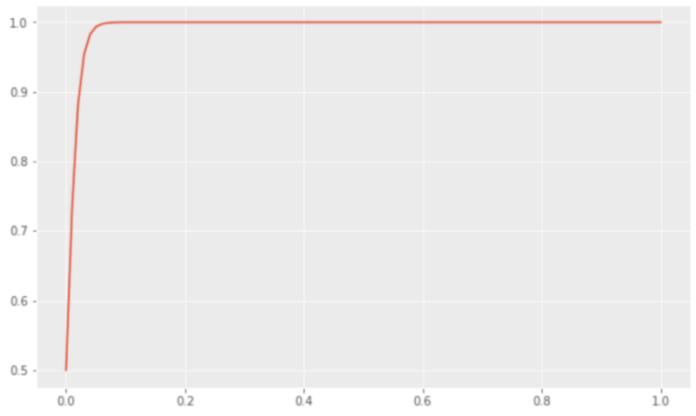
In [0]:
```
tf.reset_default_graph()
x = np.linspace(0.0, 1.0, resolution, dtype=np.float32)  # The input values.
x = x[:, np.newaxis]  # The additional axis is added so that the broadcasting below works correctly
n_neurons = 2 * 5      # The number of hidden neurons. For convenience we want it to be an even number.

w = tf.constant(100.)          # Set it at a reasonably large value.
b = -w * tf.constant(np.linspace(0.0, 1.0, n_neurons, dtype=np.float32))
# Note that the biases b shift the sigmoid function to the right.
# We'll illustrate this below.

x_ = tf.placeholder(tf.float32)
out = tf.nn.sigmoid(w*x_+b)
init_op = tf.global_variables_initializer()
```

Execution phase (run session)

In [0]:
```
with tf.Session() as sess:
    sess.run(init_op)
    z = sess.run(out,feed_dict={x_:x})
```

Display the neurons in the hidden layer

In [0]:
```
from ipywidgets import interact

def show_neurons(k=0):
    """
    Display the neurons in the hidden layer
    """
    plt.plot(x, z[:, k])
    plt.show()

w = interact(show_neurons, k =(0, n_neurons-1))
```

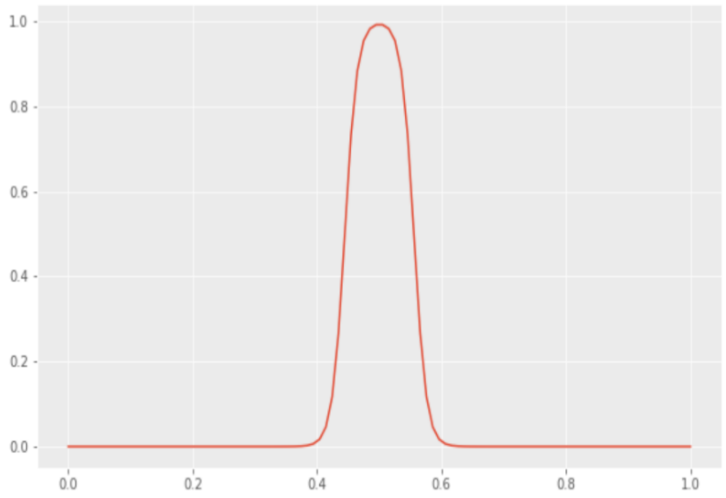


Conclusion: The different neurons are just translated sigmoid functions.

Combining the hidden neurons

The given function can be approximated by appropriately combining the different neurons:

```
In [0]: plt.plot(x, z[:, 4] - z[:, 5])
        plt.show()
```



Note: This looks a bit like a piecewise constant function.

Combine pairwise

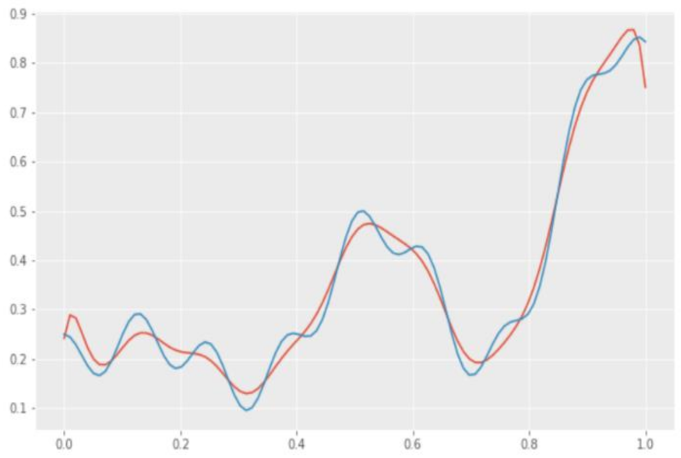
```
In [0]: def show_combined(k=0):
            """
            Display the neurons in the hidden layer
            """
            plt.plot(x, z[:, k] - z[:, k + 1])
            plt.show()

        w = interact(show_combined, k =(0, n_neurons - 2))
```

Train the network to approximate the function above

The next step is to train for the heights of the piecewise constant functions (above).

```
In [0]: tf.reset_default_graph()
        n_neurons = 2 * 50      # Specify an even number
        x = np.linspace(0.0, 1.0, resolution,dtype=np.float32)
        x = x[:,np.newaxis]

        w = tf.constant(100.0)  # This defines the `steepness' of the sigmoid.
                                # We want to have it steep in order to approximate a
                                # piecewise constant function.

        b = -w * tf.constant(np.linspace(0.0, 1.0 , n_neurons, dtype=np.float32))
        j = [i for i in range(n_neurons)]
        q = np.ones(n_neurons)
        q[1::2] = -q[1::2]
        q = np.array(q, dtype=np.float32)
        w1 = tf.Variable(q)

        x_ = tf.placeholder(tf.float32)
        y_ = tf.placeholder(tf.float32)

        out1 = tf.nn.sigmoid(w * x_ + b) * w1
        out = tf.reduce_sum(out1, axis=1)
        cost = tf.reduce_sum((out-y_)**2)

        train_step = tf.train.GradientDescentOptimizer(0.0001).minimize(cost)

        init_op = tf.global_variables_initializer()
        with tf.Session() as sess:
            sess.run(init_op)
            for i in range(1000):
                train_step.run(feed_dict={x_: x, y_: y})

            z = sess.run(out,feed_dict={x_:x})
            err = sess.run(cost,feed_dict={x_:x,y_:y})
            weight = sess.run(w1)
            offset = sess.run(b)

        # Plot and print cost
        print('Cost:',err)
        plt.plot(x,z,x,y)
        plt.show()
```

```
Cost: 0.078198
```



Conclusion: Although a little smoothed out, we indeed get a reasonable approximation of the original function.

Unit 3: Classification

TensorFlow example

The first problem we look at should be familiar, since it was in the notes for Module 3. Let's generate training data.

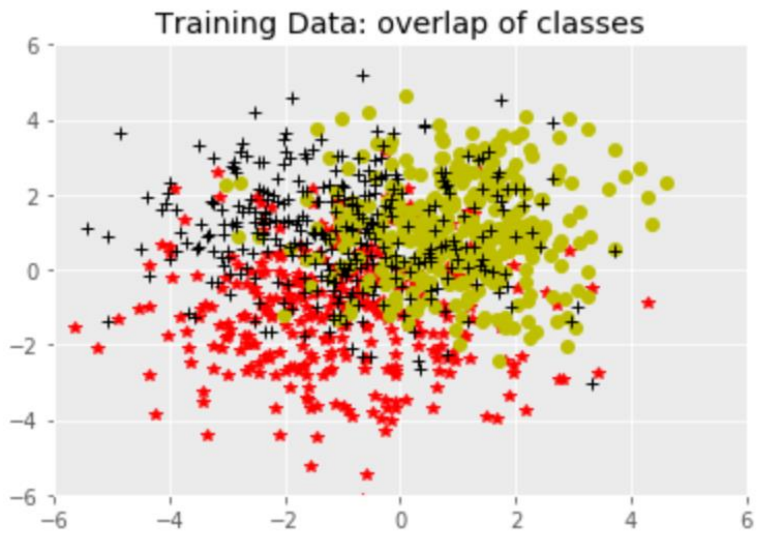
Import the necessary modules

```
In [1]: import IPython.display
        import numpy as np
        from numpy.random import randint
        from sklearn.metrics import confusion_matrix
        import tensorflow as tf
        from skimage import io
        from ipywidgets import interact

        from IPython.display import HTML
        from IPython.display import display
        from IPython.display import Image

        import matplotlib
        import matplotlib.pyplot as plt
        import matplotlib.cm as cm

        %matplotlib inline

        plt.style.use('ggplot')
        plt.rcParams['figure.figsize'] = 10, 6
        plt.rcParams['axes.facecolor'] = "0.92"
```

Training data

First, we generate training data, making sure that we have a good overlap of classes. To do this, we use two randomly generated Gaussian-distributed clouds of points in 2D-space.

```
In [2]: np.random.seed(0)

        # Number of points
        N = 1000
        # Labels for each cluster
        y = np.random.randint(low=0, high=3, size = N)
        # Mean of each cluster
        means = np.array([[-1, 1, -1], [-1, 1, 1],])
        # Covariance (in X and Y direction) of each cluster
        covariances = np.random.random_sample((2, 3)) + 1
        # Dimensions of each point
        X = np.vstack([np.random.randn(N)*covariances[0, y] + means[0, y],
                       np.random.randn(N)*covariances[1, y] + means[1, y]])
        X = X.T
        # Convert to targets, as floatX
        t = np.zeros((N,3))
        for i in range (N):
            t[i,y[i]]=1

        label = np.argmax(t,axis=1)

        # Plot the data
        col = ['r*','yo','k+', 'g*', 'b*']

        for cl in range(3):
            cl_labels = np.array([label==cl]).flatten()
            X_cl = X[cl_labels,:]
            plt.plot(X_cl[:,0],X_cl[:,1],col[cl])

        plt.axis([-6, 6, -6, 6])
        plt.title('Training Data: overlap of classes')
        plt.show()
```



Create the model in TensorFlow

We want to do the classification using TensorFlow. You will recall that this proceeds in two steps. First is the construction step:

Construction step

```
In [3]: tf.reset_default_graph()

        # Set placeholders
        x = tf.placeholder(tf.float32, [None, 2])  #2D features
        y_ = tf.placeholder(tf.float32, [None, 3]) #3 classes

        # Set Variables
        W = tf.Variable(tf.truncated_normal([2,3], stddev=0.1))
        b = tf.Variable(tf.truncated_normal([3], stddev=0.1))

        # Set Constants
        lr = tf.constant(0.1)

        # Softmax
        y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Cross entropy

We use the cross entropy as objective function.

```
In [4]: cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Calculate the gradients and updates

We use TensorFlow to calculate the gradients.

```
In [5]: grad_W = tf.gradients(cross_entropy,W)
        grad_b = tf.gradients(cross_entropy,b)

        W_update = tf.assign(W,W-lr*grad_W[0])
        b_update = tf.assign(b,b-lr*grad_b[0])

        init_op = tf.global_variables_initializer()  # Initialiser op
```

Run the session

- Up to this point, everything is symbolic.

- At runtime, the placeholders are populated by values and the nodes are executed.

Execution step

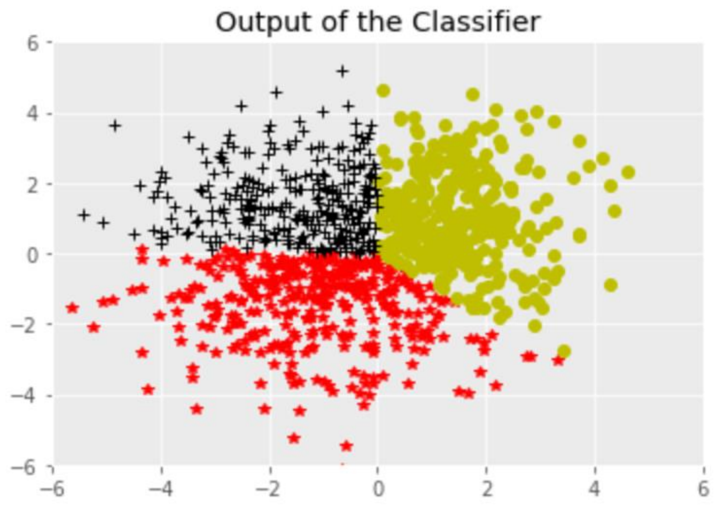
We are now ready to execute:

```
In [6]: with tf.Session() as sess:
          # Run the 'init' op.
          sess.run(init_op)

          for i in range(100):
              sess.run(W_update, feed_dict={y_:t,x:X})
              sess.run(b_update, feed_dict={y_:t,x:X})

          predict = sess.run(y, feed_dict={x:X})
          weights = W.eval()
          offsets = b.eval()

        print('Weights shape:', weights.shape)
        print('Offsets shape:', offsets.shape)
        print('Predictions shape:', predict.shape)
```

```
Weights shape: (2, 3)
Offsets shape: (3,)
Predictions shape: (1000, 3)
```

Visualize the results

```
In [7]: label = np.argmax(predict,axis=1)

        # Plot the data
        col = ['r*','yo','k+', 'g*', 'b*']

        for cl in range(3):
            cl_labels = np.array([label==cl]).flatten()
            X_cl = X[cl_labels,:]
            plt.plot(X_cl[:, 0], X_cl[:, 1], col[cl])

        plt.axis([-6, 6, -6, 6])
        plt.title('Output of the Classifier')
        plt.show()
```



Linear separation

This is clearly a linear separation of the classes. Since we use a linear separation, we cannot expect to do any better than this. In fact, this is a rather hard problem because of the considerable overlap between the classes. Let's now introduce nonlinearity. For that, we generate a new dataset that will illustrate the power of nonlinearity.

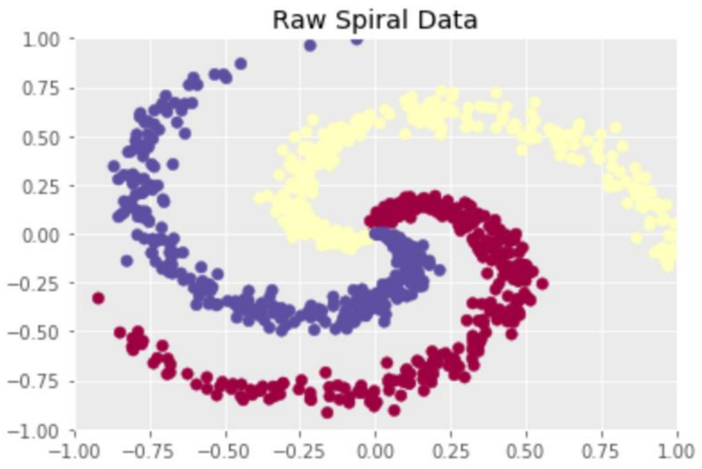
A nonlinear example

Next, let's generate data in that of a spiral formation:

```
In [8]: # Functions
        def generate_spiral_data(num_classes, dimensions, points_per_class):
            """
            Generate `num_classes` spirals with `points_per_class` points per spiral.
            """

            # Create an empty matrix to hold our X values
            X = np.zeros((points_per_class * num_classes, dimensions), dtype='float32')
            # Create an empty vector to hold our y values
            y = np.zeros(points_per_class * num_classes, dtype='uint8')

            # Generate data for each class
            for y_value in range(num_classes):
                # The indices in X and y where we will save this class of data
                ix = range(points_per_class * y_value, points_per_class * (y_value + 1))

                # Generate evenly spaced numbers in the interval 0 to 1
                radius = np.linspace(0.0, 1, points_per_class)
                theta = np.linspace(y_value * 4, (y_value + 1) * 4, points_per_class) + np.random.randn(points_per_class) * 0.2

                # Convert polar coordinates to standard Euclidian coordinates
                X[ix] = np.column_stack([radius * np.sin(theta), radius * np.cos(theta)])
                y[ix] = y_value

            return X, y


        def plot_data(X, y):
            """
            Use Matplotlib to plot X, y data on a figure.
            """

            fig = plt.figure()
            plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
            plt.xlim([-1, 1])
            plt.ylim([-1, 1])
            return fig
```

```
In [9]: np.random.seed(0)  # Setting a random seed allows us to get the exact same data each time we run the cell.
        num_classes = 3  # The number of classes (distinct groups) of data (these are our "y" values)
        dimensions = 2  # The number of dimensions of our input or "X" values
        points_per_class = 300  # number of X points to generate for each of the y values
```

```
In [10]: # Generate and plot data
         X, Y_ = generate_spiral_data(num_classes, dimensions, points_per_class)
         fig = plot_data(X, Y_)
         plt.title('Raw Spiral Data')
         plt.show()

         # Convert labels to one-hot encoding
         t = np.zeros((Y_.shape[0], 3), dtype=int)
         for i in range(Y_.shape[0]):
             t[i, Y_[i]] = 1
```



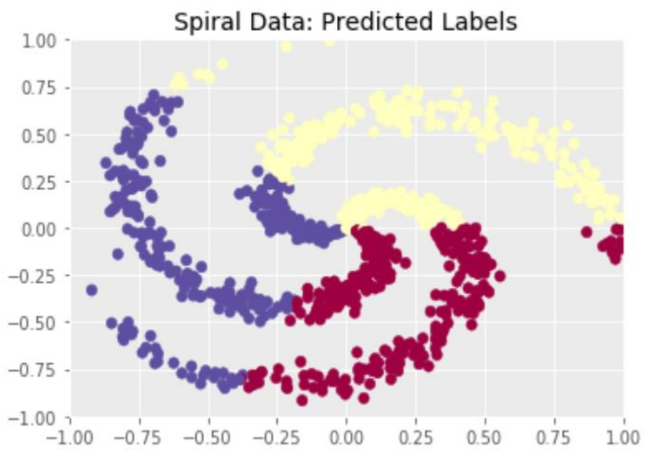
You can see that the spiral has three different classes: red, blue, and yellow.

Let's first fit a model that performs linear classification.

```
In [11]: # Construction
         # ============
         tf.reset_default_graph()

         # Set placeholders
         x = tf.placeholder(tf.float32, [None, 2])  #2D features
         y_ = tf.placeholder(tf.float32, [None, 3]) #3 classes

         # Set Variables
         W = tf.Variable(tf.truncated_normal([2,3], stddev=0.1))
         b = tf.Variable(tf.truncated_normal([3], stddev=0.1))

         # Set Constants
         lr = tf.constant(0.1)

         # Softmax
         y = tf.nn.softmax(tf.matmul(x, W) + b)

         # Cost Function
         cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

         # Set gradients and update rule
         grad_W = tf.gradients(cross_entropy,W)
         grad_b = tf.gradients(cross_entropy,b)
         W_update = tf.assign(W, W - lr * grad_W[0])
         b_update = tf.assign(b, b - lr * grad_b[0])

         # Initialiser op
         init_op = tf.global_variables_initializer()

         # Execution
         # =========
         with tf.Session() as sess:
           # Run the 'init' op.
           sess.run(init_op)

           for i in range(100):
              sess.run(W_update, feed_dict={y_:t, x:X})
              sess.run(b_update, feed_dict={y_:t, x:X})

           predict = sess.run(y, feed_dict={x:X})
           weights = W.eval()
           offsets = b.eval()

         print('Weights shape:', weights.shape)
         print('Offsets shape:', offsets.shape)
         print('Predictions shape:', predict.shape)
```

```
Weights shape: (2, 3)

Offsets shape: (3,)

Predictions shape: (900, 3)
```

```
In [12]: # Visualise the results of the linear model
         label = np.argmax(predict, axis=1)
         fig = plot_data(X, label)
         plt.title('Spiral Data: Predicted Labels')
         plt.show()
```



Conclusion: This is not a good fit. We can see the SoftMax classification fails to correctly identify the data points. It is unable to learn the nonlinear pattern of the data.

Adding a hidden layer

In order to do better, we need to introduce another nonlinear layer to our neural network.

Assuming D-dimensional input x, the first, hidden, layer is given by,

$$\mathbf{h}_1 = \mathrm{relu}(\mathbf{z}_1),$$

where

$$\mathbf{z}_1 = W_1^T\mathbf{x} + \mathbf{b}_1 \text{ and } \mathrm{relu}\,(x) = \max\,(0, x).$$

Note: The number of neurons in the hidden layer is a hyperparameter that must be specified by the user. That is, if the input dimension is D, as before, W_1 is a $D \times H_1$ dimensional tensor, where H_1 has to be specified by you, the user.

The $\mathbf{h_1}$ neurons are now simply passed on to the SoftMax layer, and therefore behaves exactly as our linear model above. From the output of the SoftMax function, we can again calculate the cross entropy, and all that remains is to calculate the gradient. The algebra becomes tedious, but the ideas are exactly the same as for the linear problem.

Backward pass: Deriving the gradients

Our model has now changed a little, but notice that W_2 (which maps from the hidden layer to logits) is now doing the same thing as W in our linear model, just on a transformed version of the inputs $\mathbf{h_1}$.

So, the good news is that the mechanics of computing the gradient with regard to W_2 will be similar to how we derived $\frac{\partial E}{\partial W}$ for the linear model above, but now we just need to replace "input activations" $\mathbf{x}$ with the "hidden activations" $\mathbf{h_1}$ (compare `derivative_loss_W2()` below with `derivative_loss_W()` in the linear model above).

So, all that is left is to compute $\frac{\partial E}{\partial W_1}$, the gradients on W_1 (the input-to-hidden layer weights; omitting the biases for now). For this, we will again use the chain rule to derive,

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial W_1}$$

Convince yourself that this is again just an application of the chain rule for derivatives, but over a longer chain ($E \to z_2 \to z_1 \to W$).

Note:

- The gradient on input weights W_1 is a product of three terms.
- We already know the first term $\frac{\partial E}{\partial z_2}$.

What is happening here?

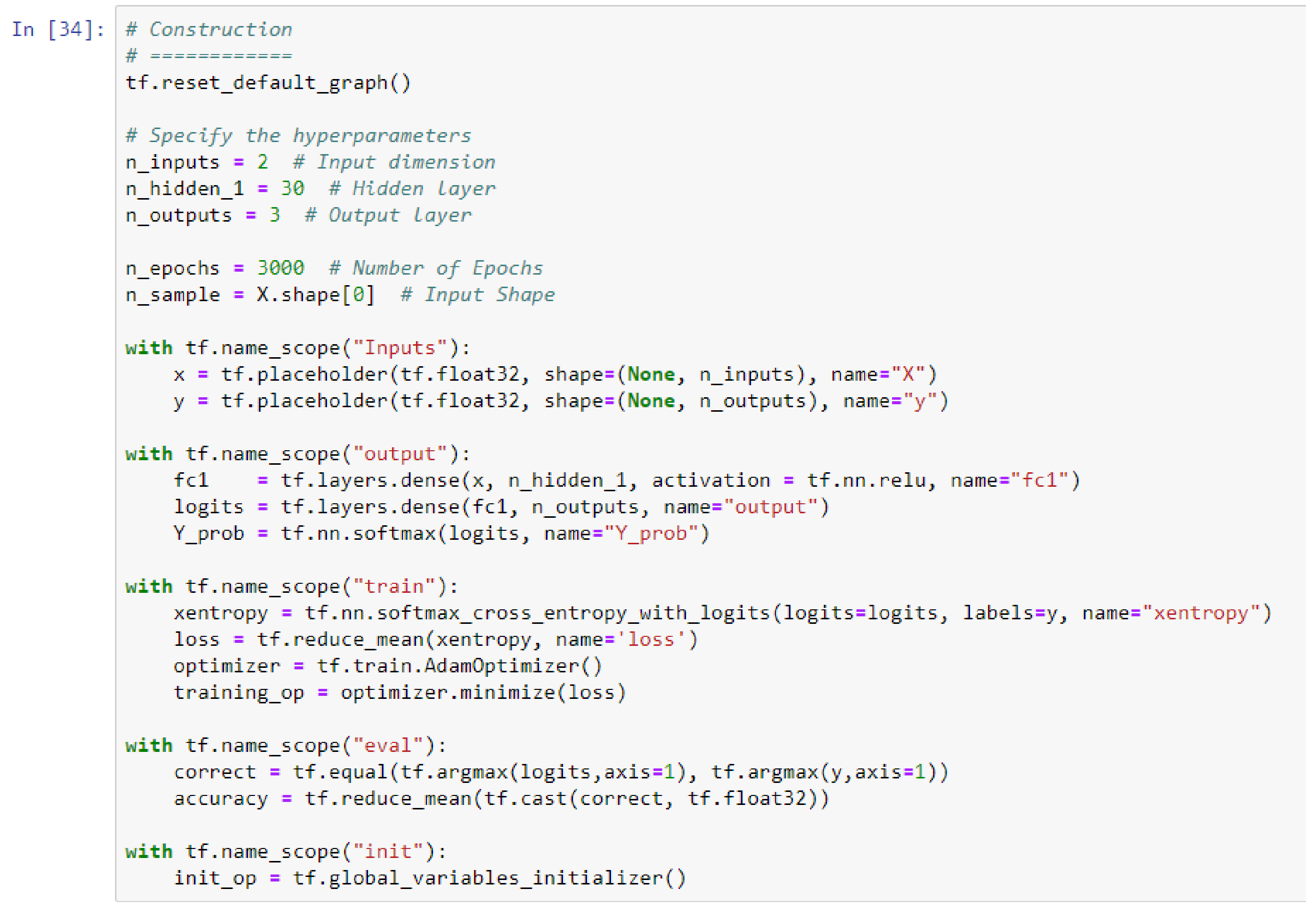
So far, we have been manually deriving the gradients of the loss with respect to all model parameters. Notice that a specific pattern is emerging:

- Propagate activations forward through the network ("make a prediction"),
- Compute an error delta ("see how far we're off"), and
- Propagate errors backwards to update the weights ("update the weights to do better next time").

Derivatives of the loss with respect to the inputs of a layer (e.g. $\frac{\partial E}{\partial z}$) are referred to as `(error) deltas`. For now, we just need the gradients calculated above, but we will use this insight in the next practical when we show how this all forms part of a more general algorithm for efficiently computing gradients in deep neural networks `(called (error) back-propagation).`

TensorFlow implementation

- Add a nonlinear layer with `n_hidden_1 = 30` hidden layer neurons.
- Use ReLU as activation function.
- Plot the results of your model.

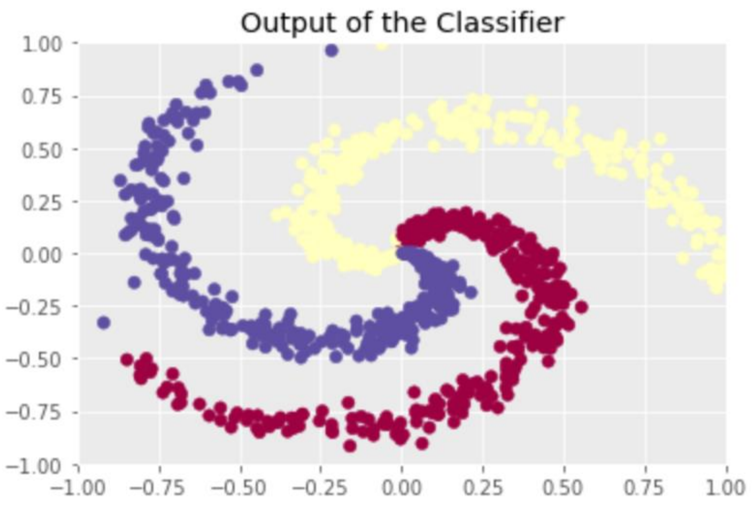


```
In [34]: # Construction
         # ============
         tf.reset_default_graph()

         # Specify the hyperparameters
         n_inputs = 2  # Input dimension
         n_hidden_1 = 30  # Hidden layer
         n_outputs = 3  # Output layer

         n_epochs = 3000  # Number of Epochs
         n_sample = X.shape[0]  # Input Shape

         with tf.name_scope("Inputs"):
             x = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
             y = tf.placeholder(tf.float32, shape=(None, n_outputs), name="y")

         with tf.name_scope("output"):
             fc1    = tf.layers.dense(x, n_hidden_1, activation = tf.nn.relu, name="fc1")
             logits = tf.layers.dense(fc1, n_outputs, name="output")
             Y_prob = tf.nn.softmax(logits, name="Y_prob")

         with tf.name_scope("train"):
             xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y, name="xentropy")
             loss = tf.reduce_mean(xentropy, name='loss')
             optimizer = tf.train.AdamOptimizer()
             training_op = optimizer.minimize(loss)

         with tf.name_scope("eval"):
             correct = tf.equal(tf.argmax(logits,axis=1), tf.argmax(y,axis=1))
             accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

         with tf.name_scope("init"):
             init_op = tf.global_variables_initializer()
```

```
In [14]: # Execution
         # =========
         with tf.Session() as sess:
             # Run the 'init' op
             sess.run(init_op)

             for epoch in range(n_epochs):
                 sess.run(training_op, feed_dict={x: X, y: t})

                 #Print the training accuracy
                 if epoch % 100==0:
                     acc_train = accuracy.eval(feed_dict={x: X, y: t})
                     print(epoch, "Train accuracy:", acc_train)

             output_probs = sess.run(Y_prob, feed_dict={x: X, y: t})
```

```
0 Train accuracy: 0.22555555
100 Train accuracy: 0.58666664
200 Train accuracy: 0.5777778
300 Train accuracy: 0.5788889
400 Train accuracy: 0.61
500 Train accuracy: 0.64555556
600 Train accuracy: 0.68777776
700 Train accuracy: 0.7177778
800 Train accuracy: 0.74
900 Train accuracy: 0.76222223
1000 Train accuracy: 0.79444444
1100 Train accuracy: 0.82222223
1200 Train accuracy: 0.83666664
1300 Train accuracy: 0.8511111
1400 Train accuracy: 0.8622222
1500 Train accuracy: 0.88222224
1600 Train accuracy: 0.8977778
1700 Train accuracy: 0.90555555
1800 Train accuracy: 0.9222222
1900 Train accuracy: 0.9355556
2000 Train accuracy: 0.9433333
2100 Train accuracy: 0.9533333
2200 Train accuracy: 0.9622222
2300 Train accuracy: 0.9688889
2400 Train accuracy: 0.9711111
2500 Train accuracy: 0.97333336
2600 Train accuracy: 0.97444445
2700 Train accuracy: 0.9766667
2800 Train accuracy: 0.9766667
2900 Train accuracy: 0.9766667
```

```
In [15]: # Visualise the results of the NN with hidden layer
         label = np.argmax(output_probs,axis=1)
         fig = plot_data(X, label)
         plt.title('Output of the Classifier')
         plt.show()
```



Conclusion: You can see from the plot above that, by adding a hidden layer, introducing nonlinearities, the model performance is dramatically improved.

Keras implementation

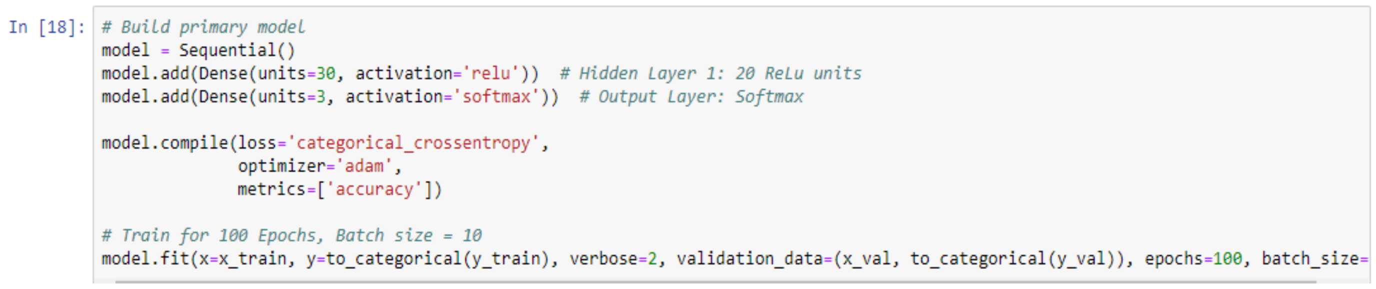
Next, we have a look at how to build the same neural network in Keras. Keras is a wrapper for TensorFlow that allows us to build deep networks with much less code.

Let's start by importing the needed packages. To read more about Keras, you can turn to the documentation.

```
In [16]: # import keras
         from keras.models import Sequential
         from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D
         from keras.utils import to_categorical
         from keras.layers.normalization import BatchNormalization
         from keras.preprocessing.image import ImageDataGenerator
         from keras.callbacks import ReduceLROnPlateau

         from sklearn.model_selection import train_test_split
```

Next, we split the data into train, validation, and test sets.

```
In [17]: # Split into Train, Validation, and Test sets
         x_train, x_test, y_train, y_test = train_test_split(X, Y_, test_size = 0.1, random_state=42)
         x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size = 0.2, random_state=42)
```

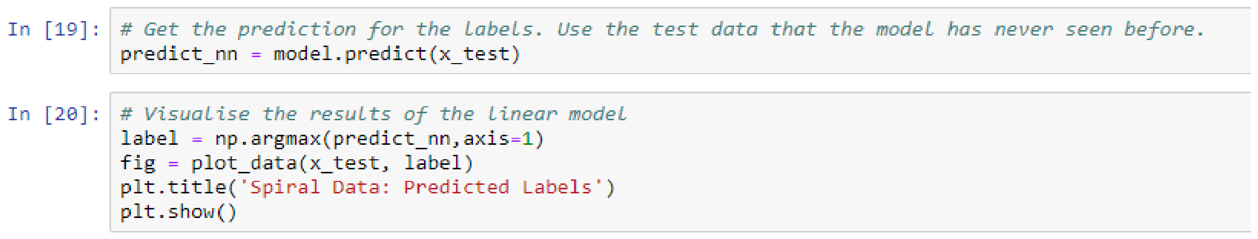


```
In [18]: # Build primary model
         model = Sequential()
         model.add(Dense(units=30, activation='relu'))  # Hidden Layer 1: 20 ReLu units
         model.add(Dense(units=3, activation='softmax'))  # Output Layer: Softmax

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

         # Train for 100 Epochs, Batch size = 10
         model.fit(x=x_train, y=to_categorical(y_train), verbose=2, validation_data=(x_val, to_categorical(y_val)), epochs=100, batch_size=
```

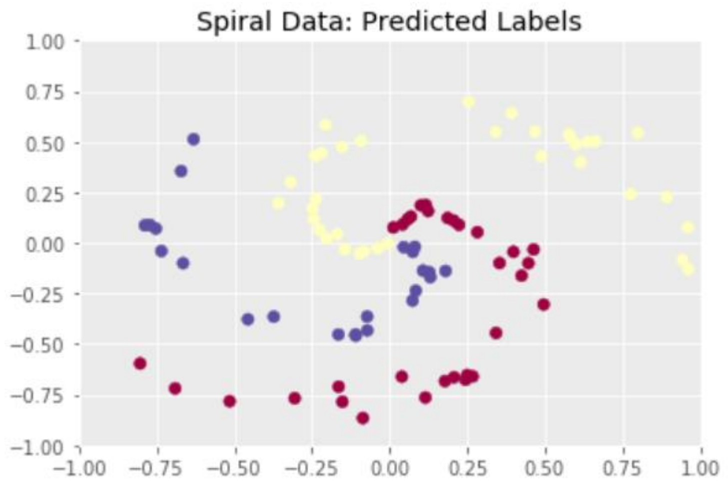
Here we have an accuracy of above 97%.

Plotting the predicted labels using the test data

This is data to which the model has never been exposed before. It is done to make sure that the model generalizes well out-of-sample.



```
In [19]: # Get the prediction for the labels. Use the test data that the model has never seen before.
         predict_nn = model.predict(x_test)

In [20]: # Visualise the results of the linear model
         label = np.argmax(predict_nn,axis=1)
         fig = plot_data(x_test, label)
         plt.title('Spiral Data: Predicted Labels')
         plt.show()
```

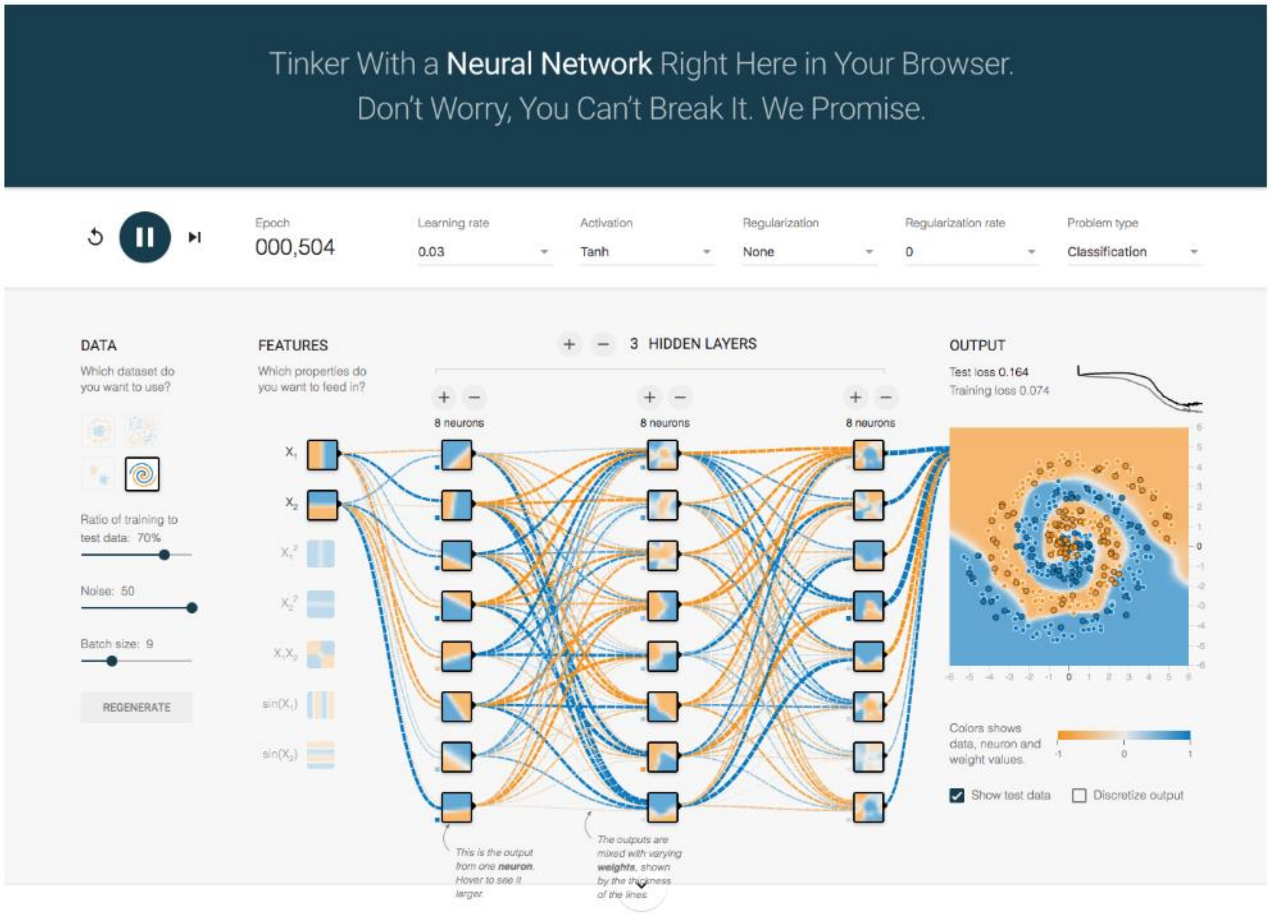


Conclusion: You can see from the plot above that by adding a hidden layer, introducing nonlinearities, the model performance is dramatically improved.

If you would like to get a much better intuition for what a neural network is doing, then we highly recommend that students refer to the TensorFlow Playground.

```
In [21]: Image("./images/tfplay.png")
```

Out[36]:



Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,504
Learning rate 0.03
Activation Tanh
Regularization None
Regularization rate 0
Problem type Classification

Ratio of training to test data: 70%
Noise: 50
Batch size: 9

Test loss 0.164
Training loss 0.074

This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Colors shows data, neuron and weight values.

Going deeper with the MNIST-dataset

We'll now switch to the MNIST dataset in order to illustrate the improvements caused by adding more layers.

Note: The MNIST dataset is conveniently already divided into a training set, a validation set, and a test set. The training set is used for training, and the validation set for determining the values of the hyper parameters, while the test set is used only at the end of your experimentation, to show the performance of your method.

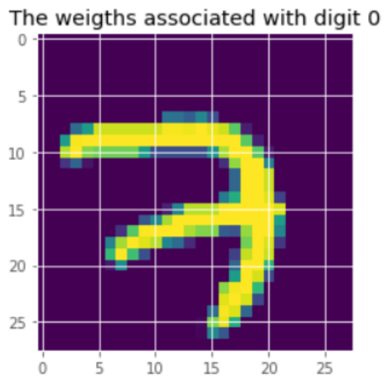
```
In [22]: # Functions
         def show_weights(k=0):
             """
             Show the weights for the 10 digits.
             """
             plt.imshow(images[k], cmap='viridis')
             plt.title('The weigths associated with digit ' + str(k))
             plt.show()
```

Import the data

The MNIST data is available in TensorFlow and provides a useful helper function for feeding it in mini-batches.

```
In [33]: from tensorflow.examples.tutorials.mnist import input_data
         mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

           Extracting MNIST_data/train-images-idx3-ubyte.gz
           Extracting MNIST_data/train-labels-idx1-ubyte.gz
           Extracting MNIST_data/t10k-images-idx3-ubyte.gz
           Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
In [39]: # Inspect the images
         images = mnist.train.images.reshape((-1, 28, 28))
         w = interact(show_weights, k =(0, 100))

         # Note that the images have been flattened and that a one-hot encoding
         # is used for the labels
         print('X shape:', mnist.train.images.shape)
         print('Y shape:', mnist.train.labels.shape)
```



```
X shape: (55000, 784)
Y shape: (55000, 10)
```

Build a basic linear classifier

In [25]:
```
# Construction
# ============
tf.reset_default_graph()

# Specify the hyperparameters
# The MNIST images
height = 28
width = 28
channels = 1
n_inputs = height * width  # Size of MNIST images

# Output layer
n_outputs = 10  # 10 different digits for classification

with tf.name_scope("Inputs"):
    X = tf.placeholder(tf.float32, shape=(None,n_inputs), name="X")
    y = tf.placeholder(tf.float32, shape=(None,n_outputs), name="y")

with tf.name_scope("output"):
    logits = tf.layers.dense(X, n_outputs, name="output")
    Y_prob = tf.nn.softmax(logits, name="Y_prob")

with tf.name_scope("train"):
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y, name="xentropy")
    loss = tf.reduce_mean(xentropy, name='loss')
    optimizer = tf.train.AdamOptimizer()
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.equal(tf.argmax(logits,axis=1), tf.argmax(y,axis=1))
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

with tf.name_scope("init"):
    init_op = tf.global_variables_initializer()
```

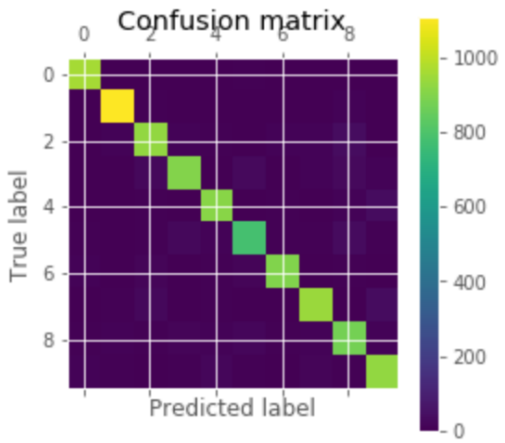
Execution phase

One feature that should be new is the use of mini-batch training. The MNIST data has a convenient feature that allows you to extract mini-batches from the data. Unfortunately, there doesn't seem to be a general function available that allows you to extract mini-batches from arbitrary data. Don't worry too much about that. It is not too difficult to write your own function if you need it (or visit Stack Overflow).

```
In [26]: n_epochs = 10
         batch_size = 50

         # Execution
         # =========
         with tf.Session() as sess:
             # Run the 'init' op.
             sess.run(init_op)

             for epoch in range(n_epochs):
                 for iteration in range(mnist.train.num_examples // batch_size):
                     X_batch, y_batch = mnist.train.next_batch(batch_size)
                     sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

                 acc_train = accuracy.eval(feed_dict={X: mnist.train.images, y: mnist.train.labels}
                 acc_test  = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})
                 print(epoch, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)

             labels = sess.run(Y_prob, feed_dict={X: mnist.test.images, y: mnist.test.labels})
```

```
0 Train accuracy: 0.9060909 ,  Val accuracy: 0.9148
1 Train accuracy: 0.9159273 ,  Val accuracy: 0.919
2 Train accuracy: 0.92218184 ,  Val accuracy: 0.9246
3 Train accuracy: 0.92185456 ,  Val accuracy: 0.925
4 Train accuracy: 0.9259091 ,  Val accuracy: 0.9278
5 Train accuracy: 0.9272 ,  Val accuracy: 0.9266
6 Train accuracy: 0.92898184 ,  Val accuracy: 0.927
7 Train accuracy: 0.92881817 ,  Val accuracy: 0.9292
8 Train accuracy: 0.931 ,  Val accuracy: 0.9296
9 Train accuracy: 0.93045455 ,  Val accuracy: 0.929
```

```
In [27]: # Get the true and predicted labels in the correct format
         t_true = np.argmax(mnist.test.labels, axis=1)
         t_predict = np.argmax(labels, axis=1)

         # Plot the confusion matrix.
         cm = confusion_matrix(t_true, t_predict)
         plt.matshow(cm)
         plt.title('Confusion matrix')
         plt.colorbar()
         plt.ylabel('True label')
         plt.xlabel('Predicted label')
         plt.show()
```



Conclusion: In this case, one cannot tell much from the graphics. Let us therefore look at the actual numbers of the confusion matrix.

```
In [28]: # Print the values of the confusion matrix
         print('Confusion Matrix:')
         print(cm)
```

```
Confusion Matrix:
[[ 961    0    2    1    0    6    4    3    2    1]
 [   0 1111    5    0    0    1    3    2   13    0]
 [   4    9  932   13    7    4   10   10   39    4]
 [   3    0   23  905    0   29    2   11   27   10]
 [   1    1    8    2  914    0    5    4    9   38]
 [   8    3    6   26    9  784    8    7   33    8]
 [  10    3   14    1    8   15  899    2    6    0]
 [   1    7   24    4    6    1    0  946    2   37]
 [   5    9    7   15    9   20    6   11  879   13]
 [  10    7    1    7   21    5    0   17    8  933]]
```

Add a nonlinear layer

Next, let's add a nonlinear layer and see how the model performs.

- Add a hidden layer with ReLU activation.
- Use `n_hidden_1 = 64`, hidden neurons.
- Display the confusion matrix as a graph and also print the actual numbers.
- You should see a marked improvement over the linear system.

```
In [29]: # Construction
         # ============
         tf.reset_default_graph()

         # Specify the hyperparameters
         # The MNIST images
         height = 28
         width = 28
         channels = 1
         n_inputs = height * width  # Size of MNIST images

         n_hidden_1 = 64  # The hidden, nonlinear layer
         n_outputs = 10  # Output layer

         with tf.name_scope("Inputs"):
             X = tf.placeholder(tf.float32, shape=(None,n_inputs), name="X")
             y = tf.placeholder(tf.float32, shape=(None,n_outputs), name="y")

         with tf.name_scope("Fc"):
             fc1 = tf.layers.dense(X, n_hidden_1, activation = tf.nn.relu, name="fc")

         with tf.name_scope("output"):
             logits = tf.layers.dense(fc1, n_outputs, name="output")
             Y_prob = tf.nn.softmax(logits, name="Y_prob")

         with tf.name_scope("train"):
             xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y, name="xentropy")
             loss = tf.reduce_mean(xentropy, name='loss')
             optimizer = tf.train.AdamOptimizer()
             training_op = optimizer.minimize(loss)

         with tf.name_scope("eval"):
             correct = tf.equal(tf.argmax(logits,axis=1), tf.argmax(y,axis=1))
             accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

         with tf.name_scope("init"):
             init_op = tf.global_variables_initializer()
```

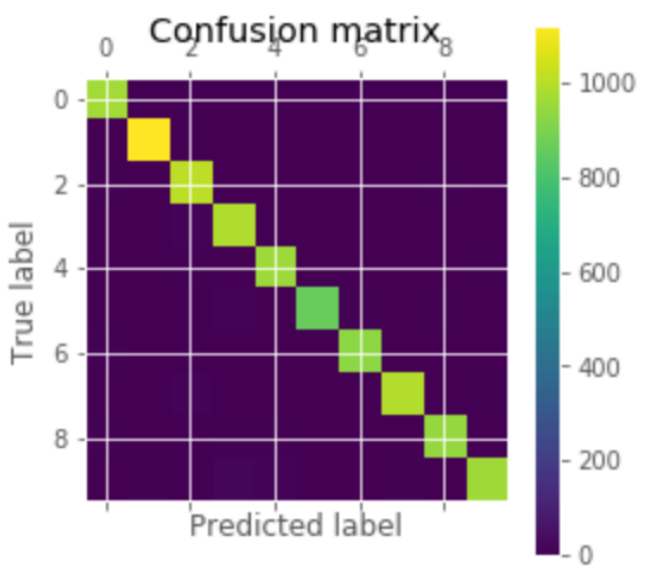
```
In [30]: n_epochs = 8
         batch_size = 50

         # Execution
         # ==========
         with tf.Session() as sess:
             # Run the 'init' op.
             sess.run(init_op)

             for epoch in range(n_epochs):
                 for iteration in range(mnist.train.num_examples // batch_size):
                     X_batch, y_batch = mnist.train.next_batch(batch_size)
                     sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

                 acc_train = accuracy.eval(feed_dict={X: mnist.train.images, y: mnist.train.labels})
                 acc_test  = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})
                 print(epoch, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)

             # Get the predictions on the test set
             labels = sess.run(Y_prob, feed_dict={X: mnist.test.images, y: mnist.test.labels})
```

```
0 Train accuracy: 0.9440182 ,  Val accuracy: 0.9466
1 Train accuracy: 0.9610909 ,  Val accuracy: 0.958
2 Train accuracy: 0.9712545 ,  Val accuracy: 0.9658
3 Train accuracy: 0.97810906 ,  Val accuracy: 0.9694
4 Train accuracy: 0.98023635 ,  Val accuracy: 0.9694
5 Train accuracy: 0.98309094 ,  Val accuracy: 0.9722
6 Train accuracy: 0.98745453 ,  Val accuracy: 0.9744
7 Train accuracy: 0.9895818 ,  Val accuracy: 0.9756
```

Conclusion: Depending on the duration of your training, you should get an accuracy on the test set exceeding 0.97. This is a significant improvement on the linear system which had only around 92% accuracy.

Below, we display the confusion matrix and print the actual numbers:

```
In [31]: # Get the true and predicted labels in the correct format
         t_true = np.argmax(mnist.test.labels, axis=1)
         t_predict = np.argmax(labels, axis=1)

         # Plot the confusion matrix.
         cm = confusion_matrix(t_true, t_predict)
         plt.matshow(cm)
         plt.title('Confusion matrix')
         plt.colorbar()
         plt.ylabel('True label')
         plt.xlabel('Predicted label')
         plt.show()
```



```
In [32]: # Print the values of the confusion matrix
         print('Confusion Matrix:')
         print(cm)
```

```
Confusion Matrix:
[[ 968    0    2    1    1    0    3    3    1    1]
 [   0 1124    4    0    0    1    2    0    4    0]
 [   6    0 1010    3    2    0    2    4    5    0]
 [   1    0   10  989    1    2    0    4    2    1]
 [   2    0    6    0  960    0    3    2    1    8]
 [   2    0    0   11    0  864    5    1    7    2]
 [   6    3    3    1    4    4  935    1    1    0]
 [   1    3   13    4    3    0    0  995    1    8]
 [   6    0    4    6    5    2    2    3  944    2]
 [   3    5    1   14   10    1    0    8    4  963]]
```

Unit 4: Overfitting and Dropout

Making things work

Up to this point, we have mainly worked with small problems. This is constructive but it does not illustrate the full power of (deep) neural networks, nor does it point out some of the problems that need to be overcome when solving largescale problems.

We have pointed out before that neural networks have been around for a long time, but it is only relatively recently that it became possible to solve largescale problems. We referred to the importance of the availability of large datasets in these developments. Now, we turn to some of the algorithmic advances that made this possible.

Interestingly enough, it is not as if someone suddenly had a fantastic new idea that changed the world. (One should, however, not underestimate the more recent, truly innovative ideas that have emerged.) It was more a question of learning how to make a number of small innovative steps work together. We have already pointed out the importance of regularization to prevent overfitting.

Here we wish to point out four more ideas:

1. Dropout
2. Activation functions
3. Initialization of the weights
4. Stochastic gradient descent and mini-batch training

Dropout

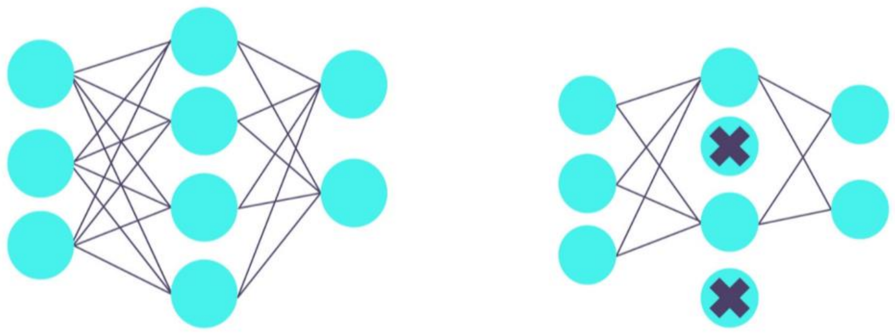


Figure 6: Dropout

Dropout was introduced in 2014 by Hinton and co-workers as a means to prevent overfitting. Again, they took their cue from biology: "A motivation for dropout comes from a theory of the role of sex in evolution (Livnat et al., 2010). Sexual reproduction involves taking half the genes of one parent and half of the other, adding a very small amount of random mutation, and combining them to produce an offspring. The asexual alternative is to create an offspring with a slightly mutated copy of the parent's genes. It seems plausible that asexual reproduction should be a better way to optimize individual fitness because a good set of genes that have come to work well together can be passed on directly to the offspring. On the other hand, sexual reproduction is likely to break up these co-adapted sets of genes, especially if these sets are large. Intuitively, this should decrease the fitness of organisms that have already evolved complicated coadaptations. However, sexual reproduction is the way most advanced organisms evolved." (Hinton, et al., 2014).

The two images (above) demonstrate how this idea was implemented in neural networks. The image on the left is a standard feedforward network where three neurons are fed into four, which in turn, are fed into two output neurons. The second image shows how a 50% dropout is implemented. Given the dropout rate for a specific layer – 50% in this case – that percentage of randomly chosen neurons is temporarily turned off, of course, together with their connecting weights. Since the active neurons are no longer able to rely on their neighbors, they must become as useful as possible themselves.

Note that this only happens during training. Once the neural network is fully trained, all the neurons become active. Since twice as many neurons are now in use as during training, the weights are divided by two.

Activation Functions

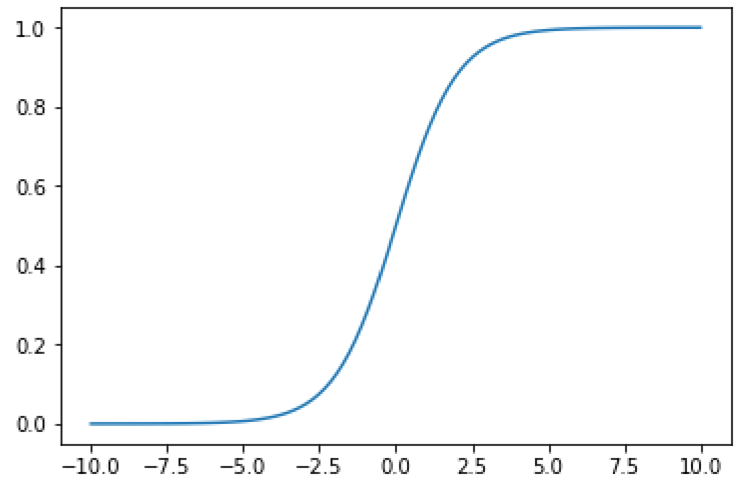


Figure 7: Sigmoid activation function

The sigmoid activation function was, for a long time, a favorite as an activation function. However, it turned out to be very hard, if not impossible to train for deep neural networks. The problem is that its gradient vanishes for input values away from zero, as should be clear from Figure 8. Since the magnitude of the gradient determines how much the weights are updated during gradient descent, vanishing gradients mean no update, hence no training. We will discuss this issue in much more detail in a later module when we look at the mechanics of backpropagation.

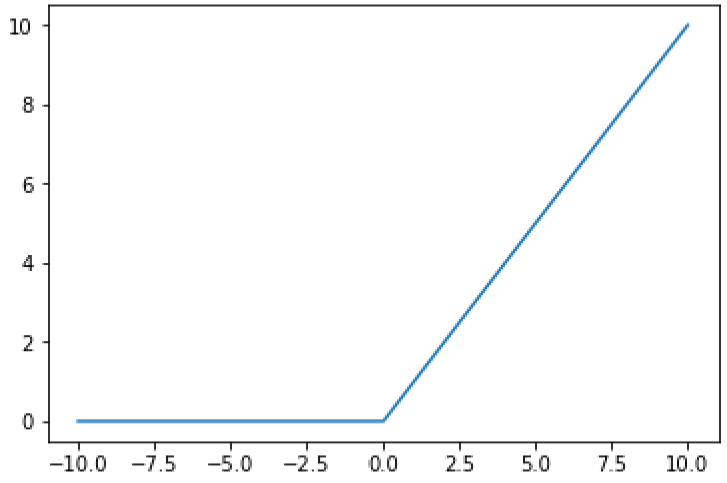


Figure 8: ReLU Activation Function

Nair and Hinton proposed the rectifying linear unit in Nair and Hinton, its efficacy subsequently demonstrated by Xu et al. It should be clear that it largely eliminates the vanishing gradient problem, most definitely for positive inputs. It remains a popular activation function.

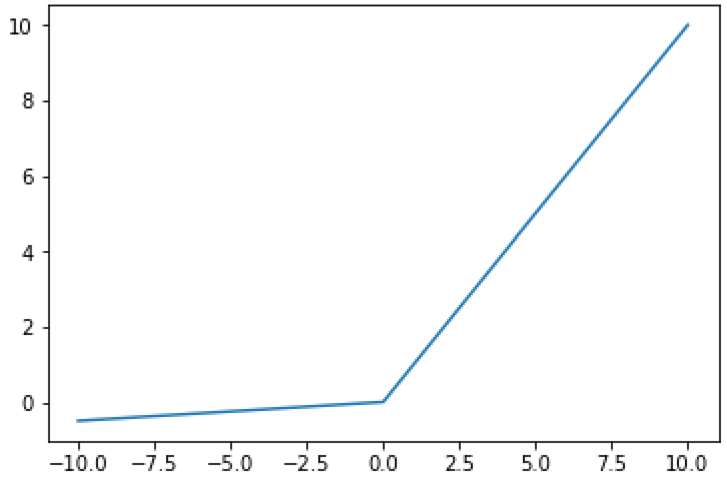


Figure 9: Leaky ReLU activation function

One weakness of ReLU is that it may suffer from "dying" neurons. If the weighted sum of the neuron's inputs is negative, its output becomes zero. Since the gradient is zero for zero inputs, it is possible that a "dead" neuron will not come back to life. Xu et al. showed that the so-called leaky ReLU outperforms the standard ReLU. Instead of using $max(0, z)$, as in ReLU, leaky ReLU uses $max(\alpha z, z)$, with α determining the amount of "leaking". The small gradient for negative inputs allows for the possibility that neurons can always come back to life.

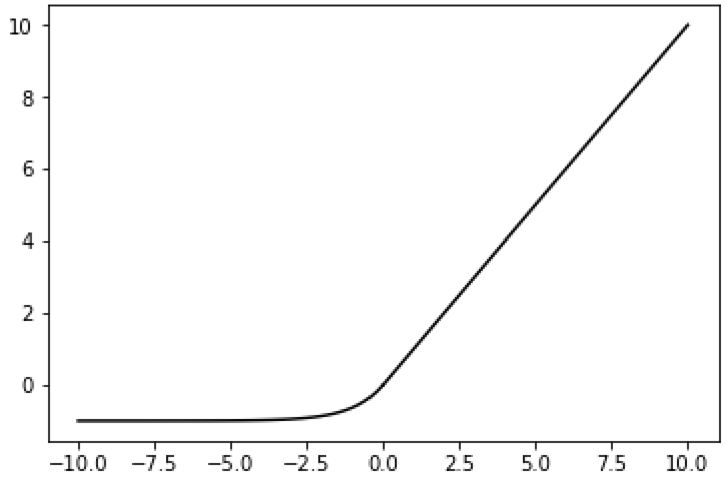


Figure 10: ELU Activation Function

The final activation function that should be mentioned is the Exponential Linear Unit (ELU) of Clevert et al., defined by,

$$\mathrm{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}.$$

Its main drawback is that it is more expensive to calculate than the activation function mentioned above.

Initialization of the weights

It remains something of a minor miracle that we are able to train very large neural networks at all. Not only are we working in very high dimensional spaces – that is, we may need to train millions of parameters – but the optimization space should be highly convoluted (for example, riddled with local optima). Using gradient descent, we select an initial set of weights and then slide downhill until we reach the nearest local minimum. How can we ensure that we often end up in a good local minimum to make the effort worthwhile? One key ingredient is surely the initialization.

Although there is no general theory, the seminal paper by Xavier Glorot and Josua Bengio (2010), showed the importance of initialization strategies. The authors argued that one should have the variance of the output neurons to be roughly equal to the variance of the input layers.

While the paper by Glorot and Bengio was mainly concerned with the sigmoid activation function, He et al. also investigated the ReLU activation. Both papers show that a random initialization, typically using a normal distribution, with the variance constrained by the number of input neurons, vastly improves training.

The following table gives the appropriate initialization standard deviations for roughly equal numbers of input and output neurons:

Activation	Standard deviation
Sigmoid	$\sigma = \frac{1}{\sqrt{n_{inputs}}}$
ReLU	$\sigma = \frac{2}{\sqrt{n_{inputs}}}$

Mini-batch training

Let's pause for a moment and reflect on how we have done our training thus far. For each optimization step, all the data is passed through the network to calculate the objective function. Then, we do a backward pass to calculate the gradients. Since it becomes quite technical, a more in-depth discussion of backpropagation is postponed to a later module. The crucial point is that it is expensive to pass all the data through the network for each update step. One easy way to remedy this is to use only part of the data for each update step. This is known as batch training. Once one has decided on an appropriate batch size (another hyperparameter), one can then take a random sample of the data of the appropriate size. Once one has passed through all the data, it is the end of an epoch. Note that for each batch, one update step is calculated. Thus, one epoch may consist of many update steps. Taken to the extreme, a batch consisting of only a single data value is known as **stochastic gradient descent**.

It should be noted that smaller batch sizes lead to more erratic training – that is, the value of the objective function will not decrease in a smooth manner. This might not be entirely undesirable as the stochasticity inherent in small batch sizes might be helpful to get out of local minima, or advance the training in regions where the optimization surface become flat. Although it is not necessary to extract your own random batches (TensorFlow does that) it may be of interest to reflect on how you will go about implementing it.

Finally, one should note the ideas of Ioffe and Szegedy regarding batch normalization. They have proposed adding an operation just before the activation function is applied to centre and normalize the inputs. This only happens during training, as we don't use mini-batches during testing. Instead, one uses the empirical mean and standard deviation of the whole training set. These are estimated during training and are stored together with the rest of the neural network parameters.

MNIST classification using one hidden layer

In this notebook, we add a hidden layer as well as dropout and regularization to the basic SoftMax classifier.

Note: With the parameter values used here, we don't get very different results. If, however, we get significantly worse results with dropout and regularization, this is likely because our model isn't overfitting the data, and when we add regularization or dropout, we are increasing the bias/underfitting the model. Hence, the worse performance.

Modification for dropout

The main modification required by dropout is to distinguish between training and predicting. Dropout is only employed during training and, since we want to use the same computational graph for training and testing, we need to let it know when we are training. This is done by defining a **training** placeholder that has the default value **False**. During the training cycle, it is set to **True** through a dictionary feed.

During the construction phase, the computational graph is constructed. In this case, we'll define different functions to help us achieve this. During the execution phase, data is passed to the computational graph and the appropriate ops are executed.

Import modules

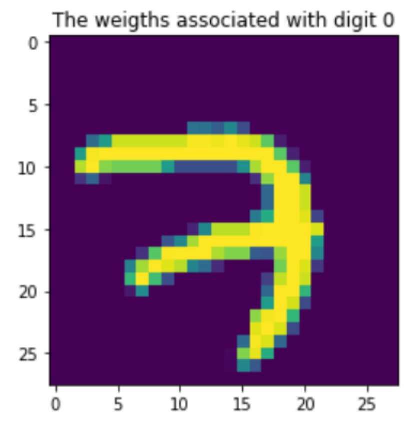
```
In [1]: # Imports
        import tensorflow as tf
        import numpy as np
        import IPython.display
        from sklearn.metrics import confusion_matrix

        from ipywidgets import interact
        from matplotlib import pylab as plt
        %matplotlib inline
```

Import the data

The MNIST data is available in TensorFlow and provides a useful helper function for feeding it in mini-batches.

```
In [2]: from tensorflow.examples.tutorials.mnist import input_data
        mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

        Extracting MNIST_data/train-images-idx3-ubyte.gz
        Extracting MNIST_data/train-labels-idx1-ubyte.gz
        Extracting MNIST_data/t10k-images-idx3-ubyte.gz
        Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
In [3]: images = mnist.train.images.reshape((-1, 28, 28))
        def show_weights(k=0):
            """
            Show the weights for the 10 digits.
            """
            plt.imshow(images[k], cmap='viridis')
            plt.title('The weigths associated with digit ' + str(k))
            plt.show()

        w = interact(show_weights, k =(0, 100))
```



Construction phase

Adding a dropout layer as well as regularization of the weights:

- Introduce L1 regularization of the weights.
- Use the same nonlinear layer as in the previous problem (see notes on neural net classification), but now add a dropout layer after it.
- Display your results as a confusion matrix, as before.

Helpful hint for regularization: `tf.layers.dense` takes an argument, `kernel_regularizer`, that specifies the regularizer you want to use. Report your results using the l1 regularizer, but do experiment with the l2 regularizer as well.

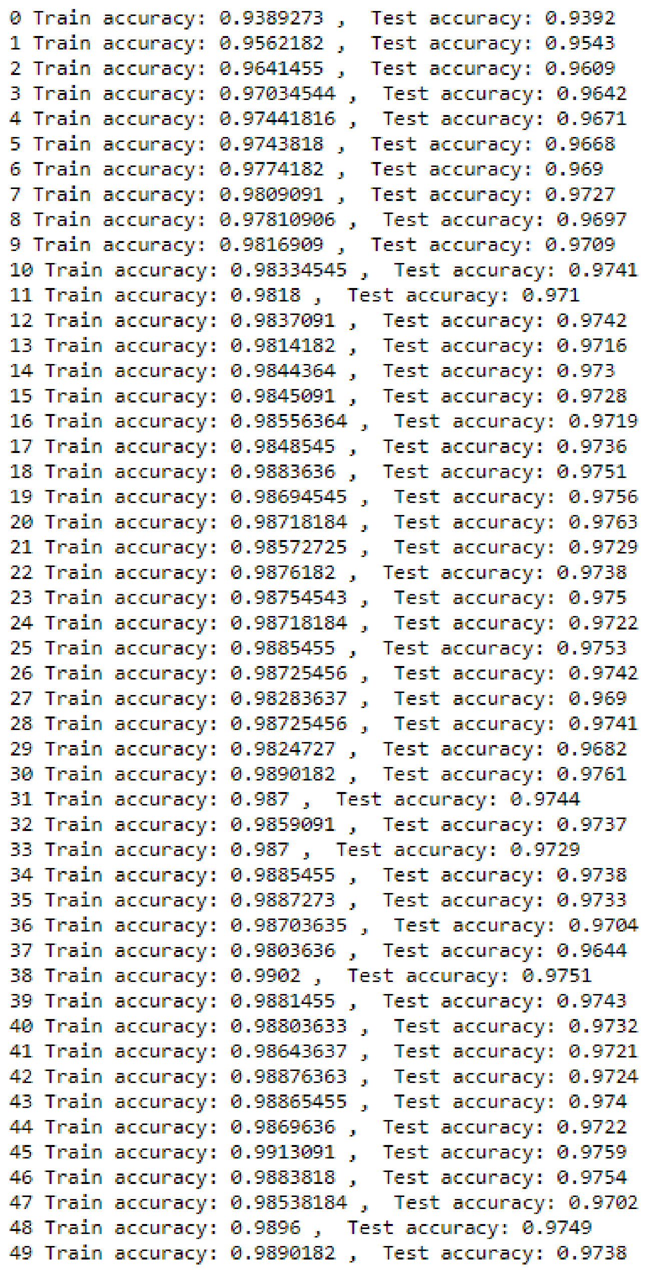
Helpful hints for dropout:

- Dropout is only used during training, not during testing when all the weights are restored.
- This means that you will need to tell TensorFlow when you train. There are two things you need to do:
  1. Add another placeholder called `training` during the construction phase, as follows: `training = tf.placeholder_with_default(False, shape=( ), name="training")`.
  2. You need to add a dropout layer after the activation was applied in the dense layer, as follows: `drop = tf.layers.dropout(hidden_layer_1, dropout_rate, training=training)`.
- You will then set training to be True during training. If not specified, it will take its default value of False.
- Note that you also need to set the dropout rate. Use 0.5.

```
In [8]: # Construction
        # ============
        tf.reset_default_graph()

        # Specify the hyperparameters
        # The MNIST images
        height = 28
        width = 28
        channels = 1
        n_inputs = height * width

        n_hidden_1 = 64  # The hidden, nonlinear layer
        dropout_rate = 0.5  # Dropout rate
        scale = 0.001  # The regularization
        n_outputs = 10  # Output layer

        with tf.name_scope("Inputs"):
            X = tf.placeholder(tf.float32, shape=(None,n_inputs), name="X")
            y = tf.placeholder(tf.float32, shape=(None,n_outputs), name="y")
            training = tf.placeholder_with_default(False, shape=(), name="training")

        # Also add l2 regularization to the dense layer
        with tf.name_scope("Fc"):
            fc_1 = tf.layers.dense(X, n_hidden_1, activation = tf.nn.relu,
                           kernel_regularizer=tf.contrib.layers.l2_regularizer(scale),
                           name="fc")

        with tf.name_scope("dropout"):
            drop_1 = tf.layers.dropout(fc_1, dropout_rate, training=training)

        with tf.name_scope("output"):
            logits = tf.layers.dense(drop_1, n_outputs, name="output")
            Y_prob = tf.nn.softmax(logits, name="Y_prob")

        with tf.name_scope("train"):
            xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y, name="xentropy")

            # Do not forget to add the reglarization loss to the standard loss
            loss = tf.reduce_mean(xentropy, name='loss') + tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
            optimizer = tf.train.AdamOptimizer()
            training_op = optimizer.minimize(loss)

        with tf.name_scope("eval"):
            correct = tf.equal(tf.argmax(logits,axis=1), tf.argmax(y,axis=1))
            accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

        with tf.name_scope("init"):
            init_op = tf.global_variables_initializer()
```

Execution phase

```
In [9]: n_epochs = 50
        batch_size = 50

        # Execution
        # =========
        with tf.Session() as sess:
            sess.run(init_op)
            for epoch in range(n_epochs):
                for iteration in range(mnist.train.num_examples // batch_size):
                    X_batch, y_batch = mnist.train.next_batch(batch_size)
                    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

                acc_train = accuracy.eval(feed_dict={X: mnist.train.images, y: mnist.train.labels})
                acc_test  = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})
                print(epoch, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)

            # Get the predictions on the test set
            labels = sess.run(Y_prob, feed_dict={X: mnist.test.images, y: mnist.test.labels})
```



```
0 Train accuracy: 0.9389273 ,  Test accuracy: 0.9392
1 Train accuracy: 0.9562182 ,  Test accuracy: 0.9543
2 Train accuracy: 0.9641455 ,  Test accuracy: 0.9609
3 Train accuracy: 0.97034544 ,  Test accuracy: 0.9642
4 Train accuracy: 0.97441816 ,  Test accuracy: 0.9671
5 Train accuracy: 0.9743818 ,  Test accuracy: 0.9668
6 Train accuracy: 0.9774182 ,  Test accuracy: 0.969
7 Train accuracy: 0.9809091 ,  Test accuracy: 0.9727
8 Train accuracy: 0.97810906 ,  Test accuracy: 0.9697
9 Train accuracy: 0.9816909 ,  Test accuracy: 0.9709
10 Train accuracy: 0.98334545 ,  Test accuracy: 0.9741
11 Train accuracy: 0.9818 ,  Test accuracy: 0.971
12 Train accuracy: 0.9837091 ,  Test accuracy: 0.9742
13 Train accuracy: 0.9814182 ,  Test accuracy: 0.9716
14 Train accuracy: 0.9844364 ,  Test accuracy: 0.973
15 Train accuracy: 0.9845091 ,  Test accuracy: 0.9728
16 Train accuracy: 0.98556364 ,  Test accuracy: 0.9719
17 Train accuracy: 0.9848545 ,  Test accuracy: 0.9736
18 Train accuracy: 0.9883636 ,  Test accuracy: 0.9751
19 Train accuracy: 0.98694545 ,  Test accuracy: 0.9756
20 Train accuracy: 0.98718184 ,  Test accuracy: 0.9763
21 Train accuracy: 0.98572725 ,  Test accuracy: 0.9729
22 Train accuracy: 0.9876182 ,  Test accuracy: 0.9738
23 Train accuracy: 0.98754543 ,  Test accuracy: 0.975
24 Train accuracy: 0.98718184 ,  Test accuracy: 0.9722
25 Train accuracy: 0.9885455 ,  Test accuracy: 0.9753
26 Train accuracy: 0.98725456 ,  Test accuracy: 0.9742
27 Train accuracy: 0.98283637 ,  Test accuracy: 0.969
28 Train accuracy: 0.98725456 ,  Test accuracy: 0.9741
29 Train accuracy: 0.9824727 ,  Test accuracy: 0.9682
30 Train accuracy: 0.9890182 ,  Test accuracy: 0.9761
31 Train accuracy: 0.987 ,  Test accuracy: 0.9744
32 Train accuracy: 0.9859091 ,  Test accuracy: 0.9737
33 Train accuracy: 0.987 ,  Test accuracy: 0.9729
34 Train accuracy: 0.9885455 ,  Test accuracy: 0.9738
35 Train accuracy: 0.9887273 ,  Test accuracy: 0.9733
36 Train accuracy: 0.98703635 ,  Test accuracy: 0.9704
37 Train accuracy: 0.9803636 ,  Test accuracy: 0.9644
38 Train accuracy: 0.9902 ,  Test accuracy: 0.9751
39 Train accuracy: 0.9881455 ,  Test accuracy: 0.9743
40 Train accuracy: 0.98803633 ,  Test accuracy: 0.9732
41 Train accuracy: 0.98643637 ,  Test accuracy: 0.9721
42 Train accuracy: 0.98876363 ,  Test accuracy: 0.9724
43 Train accuracy: 0.98865455 ,  Test accuracy: 0.974
44 Train accuracy: 0.9869636 ,  Test accuracy: 0.9722
45 Train accuracy: 0.9913091 ,  Test accuracy: 0.9759
46 Train accuracy: 0.9883818 ,  Test accuracy: 0.9754
47 Train accuracy: 0.98538184 ,  Test accuracy: 0.9702
48 Train accuracy: 0.9896 ,  Test accuracy: 0.9749
49 Train accuracy: 0.9890182 ,  Test accuracy: 0.9738
```

Show the values of the confusion matrix

```
In [6]: # Get the true and predicted labels in the correct format
        t_true = np.argmax(mnist.test.labels, axis=1)
        t_predict = np.argmax(labels, axis=1)

        # Get the confusion matrix.
        cm = confusion_matrix(t_true, t_predict)
```

```
In [7]: print('Confusion Matrix:')
        print(cm)
```

```
Confusion Matrix:
[[ 972    0    2    0    1    0    1    1    3    0]
 [   0 1123    0    3    1    1    1    0    5    1]
 [   6   11  995    1    6    0    2    5    6    0]
 [   2    2   17  961    1   10    0    7    8    2]
 [   1    0    3    0  960    0    3    0    0   15]
 [  12    2    5   12    3  833    9    1   12    3]
 [  14    3    1    0   23    5  908    0    4    0]
 [   5   10   20    5   10    0    0  961    4   13]
 [   9    3    4    1    2    3    4    1  945    2]
 [   3    3    0    0   13    8    1    4    4  973]]
```

Conclusion

Regularization has not made too much of a difference.

Unit 5: Academic Papers in Review

This week we are going to look at one paper, which is publicly available, entitled "Deep Neural Networks, Gradient-boosted Trees, Random Forests: Statistical Arbitrage on the S&P 500" (Krauss, Do & Huck, 2016). It gives a good outline on how to develop a trading strategy using deep neural networks. It explains an end-to-end solution but, on top of that, it explores other nonlinear techniques, and combines all three base models in an ensemble which, in turn, outperforms any single base model.

This paper covers much of the relevant literature – for example, you will find references to other works that outline the reason why classification works better than regression for these types of problems (alpha design).

Students will be able to get some great insights and ideas for their group project. Please note that we will be asking multiple-choice questions from this paper.

Another paper that isn't publicly available, but worth a read, is etitled "The Benefits of Tree-based Models for Stock Selection" (Zhu, Philpotts & Stevenson, 2012). It doesn't cover neural networks specifically but it very eloquently explains the benefits of using nonlinear models in finance. In particular, the two benefits are:

1. "If the true structural relationships are actually nonlinear, then it is reasonable to assume that unexploited profit opportunities are more likely to be identified in such a framework."
2. "Such techniques may offer a high degree of model diversification from more traditional approaches."

Bibliography

References

Clevert, D., Unterthiner, T. & Hochreiter, S. (2016). “Fast and Accurate Deep Network Learning by Exponential Linear Units (Elus).” Published as a conference paper at *International Conference on Learning Representations (ICLR).*

CS231n Convolutional Neural Networks for Visual Recognition. (2018). [Online]. Available at: http://cs231n.github.io/neural-networks-1/.

Géron, A. (2017) *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O’Reilly Media.

Glorot, X. & Bengio, Y. (2010). “Understanding the difficulty of training deep feedforward neural networks.” *PMLR* 9: pp. 249-256.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. MIT Press.

He, K., Zhang, X., Ren, S. & Sun, J. (2015). “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. *IEEE International Conference on Computer Vision (ICCV)*: pp. 1026-1034.

Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *Journal of Machine Learning Research*, 15: pp. 1929-1958.

Ioffe, S. & Szegedy, C. (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” [Online]. Available at: https://arxiv.org/pdf/1502.03167.pdf.

Krauss, C., Do, X.A. & Huck, N. (2016). “Deep Neural Networks, Gradient-boosted Trees, Random Forests: Statistical Arbitrage on the S&P 500”. *FAU Discussion Papers in Economics*, 3.

Neural Network SoftMax Activation. (2018). [Online]. Available at: https://stats.stackexchange.com/questions/273465/neural-network-SoftMax-activation

Nielson, M. (2015). *Neural Networks and Deep Learning*. Determination Press.

Parellada, A. (2017). *Neural Network SoftMax Activation* [Online]. Available at: https://stats.stackexchange.com/users/67822/antoni-parellada.

Zhu, M., Philpotts, D. & Stevenson, M.J. (2012). "The Benefits of Tree-based Models for Stock Selection". *Journal of Asset Management*, 13(6): pp. 437-448.