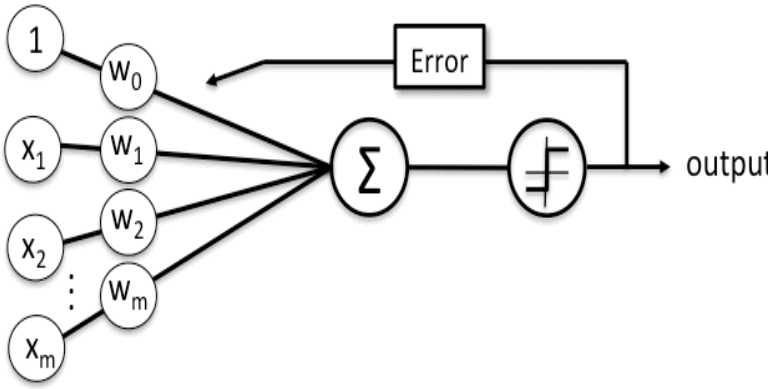
Neural Networks

Perceptrons

The perceptron is a binary linear classifier that is only capable of predicting classes of samples if those samples can be separated via a straight line. The perceptron algorithm was introduced by Frank Rosenblatt in 1957. It classifies samples using hand crafted features which represents information about the samples, weighs the features on how important they are to the final prediction and the resulting computation is compared against a threshold value.



In the image above, X represents the inputs to the model and W represents the weights (how important are individual features). A linear computation of the weighted sum of features is carried out during the formula below:

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m$$

The value of z is then passed through a step function to predict the class of the sample. A step function is an instant transformation of a value from 0 to 1. What this means is that if z is greater than or equal to 0, its predicts one class, else it predicts the other. The step function can be represented mathematically as:

$$f(x) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

At each iteration, the predicted class gets compared to the actual class and the weights gets updated if the prediction was wrong else it is left unchanged in the case of a correct prediction. Updates of weights continue until all samples are correctly predicted, at which point we can say that the perceptron classifier has found a linear decision boundary that perfectly separates all samples into two mutually exclusive classes.

During training the weights are updated by adding a small value to the original weights. The amount added is determined by the perceptron learning rule. The weight update process can be experienced mathematically as shown below.

$$w_j := w_j + \Delta w_j$$

The amount by which weights are updated is given by the perceptron learning rule below.

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

The first coefficient on the right hand side of the equation is called the learning rate and acts as a scaling factor to increase or decrease the extent of the update. The intuitive understanding of the above equation is that with each pass through the training set, the weights of misclassified examples are nudged in the correct direction so that the value of z can be such that the step function correctly classifies the sample. It should be noted that the

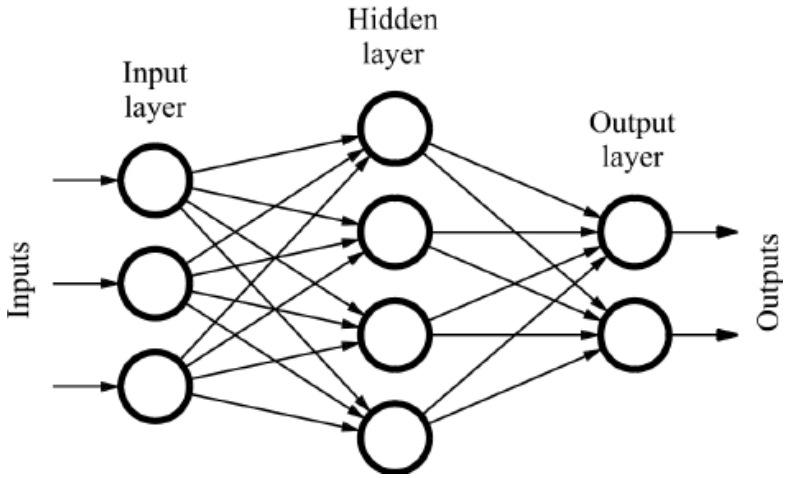
perceptron learning algorithm described is severely limited as it can only learn simple functions that have a clear linear boundary. The perceptron is almost never used in practice but served as an integral building block during the earlier development of artificial neural networks.

Modern iterations are known as multi-layer perceptrons. Multi-layer perceptrons are feed forward neural networks that have several nodes in the structure of a perceptron. However, there are important differences. A multilayer perceptron is made up of multiple layers of neurons stacked to form a network. The activation functions used are non-linear unlike the perceptron model that uses a step function. Nonlinear activations are capable of capturing more interesting representations of data and as such do not require input data to be linearly separable. The other important difference is that multi-layer perceptrons are trained using a different kind of algorithm called backpropagation which enables training across multiple layers.

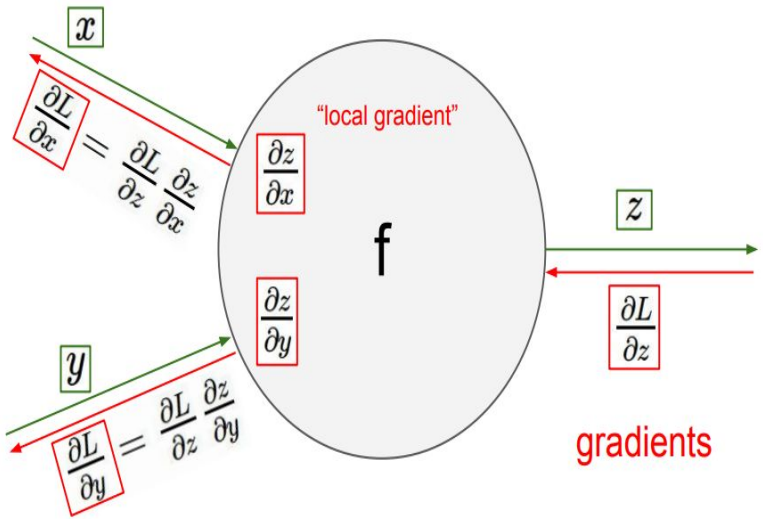
Backpropagation

Backpropagation is an algorithm technique that is used to solve the issue of credit assignment in artificial neural networks. What that means is that it is used to determine how much an input's features and weights contribute to the final output of the model. Unlike the perceptron learning rule, backpropagation is used to calculate the gradients, which tell us how much a change in the parameters of the model affects the final output. The gradients are used to train the model by using them as an error signal to indicate to the model how far off its predictions are from the ground truth. The backpropagation algorithm can be thought of as the chain rule of derivatives applied across layers.

Let us look at a full fledged illustration of a multi-layer perceptron to understand things further.



The network above is made up of three layers, the input layer which are the features fed into the network, the hidden layer which is so called because we cannot observe what goes on inside and the output layer, through which we get the prediction of the model. During training, in order to calculate by how each node contributes to the final prediction and adjust them accordingly to yield a higher accuracy across samples, we need to change the weights using the backpropagation algorithm. It is the weights that are learned during the training process hence they are sometimes referred to as the learnable parameters of the model. To visually understand what goes on during backpropagation, lets us look at the image of a single node below.



In the node above x and y are the input features while f is the nonlinear activation function. During training computations are calculated in a forward fashion from the inputs, across the hidden layers, all the way to the output. This is known as the forward pass denoted by green arrows in the image. The prediction of the model is then compared to the ground truth and the error is propagated backwards. This is known as the backward pass and assigns the amount by which every node is responsible for the computed error through the backpropagation algorithm. It is depicted with red arrows in the image above. This process continues until the model finds a set of weights that captures the underlying data representation and correctly predicts majority of samples.

How to run the Neural Network using TensorFlow

For our hands on example, we would do image classification using the MNIST handwritten digits database which contains pictures of handwritten digits ranging from 0 to 9 in black and white. The task is to train a neural network that given an input digit image, it can predict the class of the number contained therein.

How to get our data

TensorFlow includes several preloaded datasets which we can use to learn or test out our ideas during experimentation. The MNIST database is one of such cleaned up datasets that is simple and easy to understand. Each data point is a black and white image with only one color channel. Each pixel in the image denotes the brightness of that point with 0 indicating black and 255 white. The numbers range from 0 to 255 for 784 points in a 28 × 28 grid.

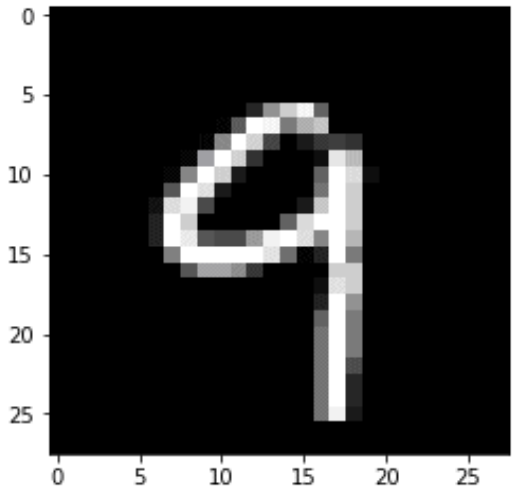
Let's go ahead and load the data from TensorFlow along with importing other relevant libraries.

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Let us use the matplotlib library to display an image to see what it looks like by running the following lines of code.

```
plt.imshow(np.reshape(mnist.train.images[8], [28, 28]), cmap='gray')
plt.show()
```



The displayed image is a handwritten digit of number 9.

How to train and test the data

In order to train an artificial neural network model on our data, we first need to define the parameters that describe the computation graph such as number of neurons in each hidden layer, number of hidden layers, input size, number of output classes etc. Each image in the dataset is 28 by 28 pixels therefore, the input shape is 784 which is 28 × 28.

```
# Parameters
learning_rate = 0.1
num_steps = 500
batch_size = 128
display_step = 100
```

```
# Network Parameters
n_hidden_1 = 10 # 1st layer number of neurons
n_hidden_2 = 10 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

We then declare weights and biases which are trainable parameters and initialise them randomly to very small values. The declarations are stored in a Python dictionary.

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

We are would then describe a 3-layer neural network with 10 units in the output for each of the class digits and define the model by creating a function which forward propagates the inputs through the layers. Note that we are still describing all these operations on the computation graph.

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 10 neurons
```

```
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 10 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

Next we call our function, define the loss objective, choose the optimizer that would be used to train the model and initialise all variables.

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

Finally, we create a session, supply images in batches to the model for training and print the loss and accuracy for each mini-batch.

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)
```

```
for step in range(1, num_steps+1):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    # Run optimization op (backprop)
    sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
    if step % display_step == 0 or step == 1:
        # Calculate batch loss and accuracy
        loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                             Y: batch_y})
        print("Step " + str(step) + ", Minibatch Loss= " + \
              "{:.4f}".format(loss) + ", Training Accuracy= " + \
              "{:.3f}".format(acc))

print("Optimization Finished!")

# Calculate accuracy for MNIST test images
print("Testing Accuracy:", \
    sess.run(accuracy, feed_dict={X: mnist.test.images,
                                  Y: mnist.test.labels}))
```

The session was created using **with**, so it automatically closes after executing. This is the recommended way of running a session as we would not need to manually close it. Below is the output

```
Step 1, Minibatch Loss= 159.5374, Training Accuracy= 0.156
Step 100, Minibatch Loss= 1.0810, Training Accuracy= 0.773
Step 200, Minibatch Loss= 1.0142, Training Accuracy= 0.797
Step 300, Minibatch Loss= 0.5115, Training Accuracy= 0.844
Step 400, Minibatch Loss= 0.4631, Training Accuracy= 0.891
Step 500, Minibatch Loss= 0.4863, Training Accuracy= 0.867
Optimization Finished!
Testing Accuracy: 0.85
```

The loss drops to 0.4863 after training for 500 steps and we achieve an accuracy of 85% on the test set.

Here is the code in full:

```
# Parameters
```

```
learning_rate = 0.1
num_steps = 500
batch_size = 128
display_step = 100

# Network Parameters
n_hidden_1 = 10 # 1st layer number of neurons
n_hidden_2 = 10 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}

# Create model
def neural_net(x):
    # Hidden fully connected layer with 10 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 10 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
```

```
    return out_layer

# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))
```

```
print("Optimization Finished!")

# Calculate accuracy for MNIST test images
print("Testing Accuracy:", \
    sess.run(accuracy, feed_dict={X: mnist.test.images,
                                  Y: mnist.test.labels}))
```