Neural Network fundamentals with TensorFlow

# Course Outline

## Tensorflow Basics

* Creation, Initializing, Saving and Restoring TensorFlow variables
* Feeding, Reading and Preloading TensorFlow data
* How to use TensorFlow infrastructure to train models at scale
* Visualizing and Evaluating models with TensorBoard

## TensorFlow Mechanics

* Inputs and Placeholders
* Build the Graph
  + Inference
  + Loss
  + Training
* Train the model
  + The graph
  + The session
  + Train loop
* Evaluate the model.
  + Build the eval graph
  + Eval output

## The perceptron

* Activation functions
* The perceptron learning algorithm
* Binary classification with the perceptron
* Document classification with the perceptron
* Limitations of the perceptron

## Support Vector Machines

* Kernels and the kernel trick.
* Maximum margin classification and support vectors

## Artificial Neural Networks

* Nonlinear decision boundaries
* Feedforward and feedback artificial neural networks
* Multilayer perceptrons
* Minimizing the cost function
* Forward propagation
* Back propagation
* Improving the way neural networks learn

## Convolutional Neural Networks

* Goals
* Model architecture
* Principles
* Code organization
* Launching and training the model.
* Evaluating a model.

# TensorFlow Basics.

## What is TensorFlow?

TensorFlow is an open source library and application programming interface (API) for building programs that employ machine learning for a range of tasks. Components of TensorFlow include a symbolic math library as well as interfaces for building and training machine learning applications to detect and decipher patterns and correlations in data. This is similar to the way that biological entities learn and reason.

TensorFlow was created and made available by Google, Inc and was released under the Apache 2.0 Open Source License on 9 November, 2015.

TensorFlow was based on an earlier system, also built at Google called DistBelief. DistBelief gained increasing popularity among Google’s (Later Alphabet) different companies and divisions for use in both commercial and research applications. Google spent significant resources to simply and refactor the code base of DistBelief into a robust, performant applications-grade library which was then called TensorFlow. Version 1.0.0 of TensorFlow was released on 11 February 2017. TensorFlow can run on single or multiple CPU’s and GPU’s. TensorFlow is available on 64 bit Linux, MacOS, Windows and mobile platforms including iOS and Android.

TensorFlow provides a number of programmatic API’s for various computer languages, including Python, C++, Java, GoLang and Rust.

Additionally, third party API’s are available for C#, Julia, R and Scala.

TensorFlow is known for use in image processing and recognition, deep learning with AlphaGo and other cutting edge applications in the field of Machine Learning.

## How does TensorFlow work?

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that these neural networks and other machine learning algorithms perform on multidimensional data arrays, known as *tensors*.

Tensors are formally known as mutilinear maps from vector spaces to real numbers, so, for example,

* A scalar is a tensor
* A vector is a tensor
* A matrix is a tensor

Basically, tensors can be represented as a multidimensional array of numbers.

### TensorFlow vs. Numpy

TensorFlow and Numpy are similar. Both are N-d array libraries. While Numpy has Ndarray support, it doesn’t offer methods to create tensor functions and automatically compute derivatives. Additionally, numpy offers no support for using GPU’s, unlike TensorFlow.

Let’s take a look at a similar program using both standard Python with Numpy and a TensorFlow application.

Table 1. A simple numpy program.

|  |
| --- |
| #!/usr/bin/env python3  import numpy as np  # Create an array 'a' as a 2 by 2 dimensional array  # initialized to zeros.  a = np.zeros((2,2))  # Create an array 'b' as a 2 by 2 dimensional array initialized to  # ones.  b = np.ones((2,2))  # Add the two up. The axis parameter refers to columsn vs. rows.  # axis=0 refers aggregation along the row, axis=1 refers to  # aggregation along the columns.  print (np.sum (b,axis=1))  # The reshape method changes a to be a 1 X 4 array.  print (np.reshape(a,(1,4))) |

Now let’s take a look at the same program using TensorFlow.

Table 2*. A* simple TensorFlow example.

|  |
| --- |
| #!/usr/bin/env python3  import tensorflow as tf  # Running in interactive mode is good for debugging as it  # executes the tensorflow code straight away rather  # than having to create a session variable.  tf.InteractiveSession()  # Create our 'a' array as a 2 X 2 tensor initialized to zeros.  a = tf.zeros((2,2))  # Create our 'b' array as a 2 X 2 tensor initialized to ones.  b = tf.ones((2,2))  # Sum array a and b, reduction\_indices is similar to 'axis'.  # note that we have to invoke the eval() method to  # actually get the reduce\_sum method to run.  print (tf.reduce\_sum(b,reduction\_indices=1).eval())  # Print the shape of tensor 'a'.  print (a.get\_shape())  # Reshape 'a' as a 1 X 4 tensor and print it out.  print (tf.reshape(a,(1,4)).eval()) |

Following is a table of Numpy vs. TensorFlow statements for the previous examples.

Table 3. A summary of differences between Numpy and TensorFlow statements.

|  |  |
| --- | --- |
| **Numpy** | **TensorFlow** |
| a = np.zeros((2,2)); | a = tf.zeros((2,2)) |
| b = np.ones((2,2)) | b = tf.ones((2,2)) |
| np.sum(b, axis=1) | tf.reduce\_sum(a,reduction\_indices=[1]) |
| a.shape | a.get\_shape() |
| np.reshape(a, (1,4)) | tf.reshape(a, (1,4)) |
| np.dot(a,b) | tf.matmul(a, b) |
| b \* 5 + 1 | b \* 5 + 1 |
| a[0,0], a[:,0], a[0,:] | a[0,0], a[:,0], a[0,:] |

Note that TensorFlow requires explicit evaluation of operations. A big difference between the two is that the TensorFlow variables have no actual value until the TensorFlow operation is evaluated. What you will get when printing a tensorflow variable before execution is something like this:

Table 4. Output of a tensor variable before execution of the graph.

|  |
| --- |
| Tensor("zeros:0", shape=(2, 2), dtype=float32) |

This is what will happen in our simple TensorFlow example if you insert a print (a) in the code before you invoke the tf.reduce\_sum evaluation method.

### The TensorFlow session.

The TensorFlow documentation defines a session object as: “A Session object encapsulates the environment in which Tensor objects are evaluated.”

Let’s see how this would work.

Table 5. A simple example of using sessions in TensorFlow.

|  |
| --- |
| #!/usr/bin/env python3  import tensorflow as tf  # Here we declare two constants.  a = tf.constant (5.0)  b = tf.constant (6.0)  # Multiply the two tensors together to output a third tensor 'c'.  c = a \* b  # Nothing will run until we start a TensorFlow session.  # In the previous example, tf.InteractiveSession() was just  # syntactic sugar for the following.  # Note that we can use the Python 'with' to treat tf.Session  # as a context object, containing \_\_entry\_\_ and \_\_exit\_\_ methods.  with tf.Session() as sess:  # Note that c.eval() is, again, syntactic sugar for the sess.run(c)  # statment. sess.run() is an exxample of a Tensorflow fetch  # statement.  print (sess.run(c))  print (c.eval()) |

### The TensorFlow Computation Graph.

TensorFlow programs are structured as follows:

* A construction phase, which creates and assembles a graph.
* An execution phase, used to run operations inside the graph.

When training a TensorFlow model, we use variables to hold and update parameters. Variables are in-memory buffers that contain tensor objects.

Up until now, all the tensors we’ve seen have been constants, not variables. Let’s see how tensor variables work.

Table 6. The TensorFlow variable.

|  |
| --- |
| #!/usr/bin/env python3  import tensorflow as tf  W1 = tf.ones((2,2))  # Note that W2, unlike W1, is a node in the TensorFlow Graph.  # The name parameter is useful when we want to save or restore  # the values of the variable from a file.  W2 = tf.Variable(tf.zeros((2,2)),name='weights')  with tf.Session() as sess:  print ('\nW1')  print (sess.run(W1))  # Here we initialize all of the graph variaables.  sess.run(tf.global\_variables\_initializer())  # We have to run the graph in order to gain access to W2 to print it.  print ('\nW2')  print (sess.run(W2)) |

As noted previously, TensorFlow variables *must* be initialized via the global\_variables\_initializer() method before they will hold any values.

To change the value of a TensorFlow variable we can use the tf.assign() method as follows:

Table 7. Changing TensorFlow variable state.

|  |
| --- |
| state = tf.Variable(0,name-=’counter’)  new\_value = tf.add(state, tf.constant(1) # This is equivalent to new\_value = state + 1  update = tf.assign(state,newvalue) # Assign the value of new\_value back to state.  with tf.Session() as sess:  sess.run(tf.global\_initialize\_variables())  print (sess.run(state))  for \_ in range(3):  sess.run(update)  print (sess.run(state)) |

The output of this program should be the values, 0,1,2,3.

### Fetching Variable State.

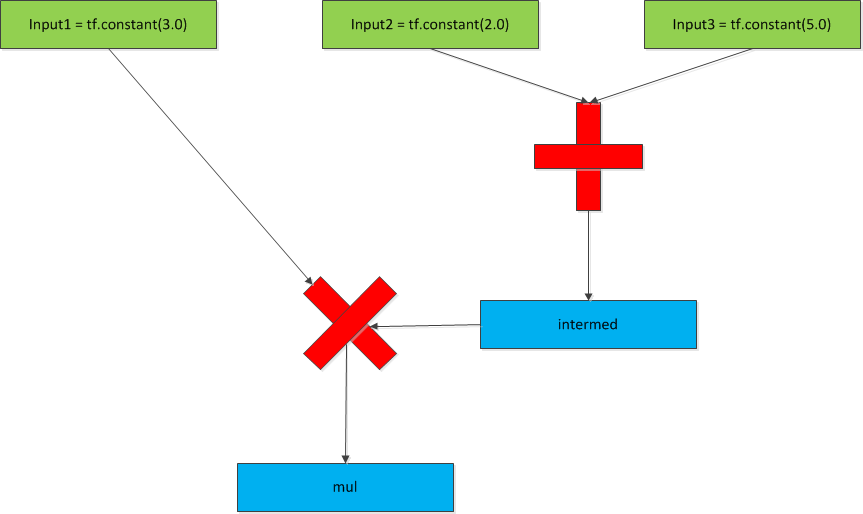
Calling the run(var) method on the TensorFlow session object and passing in a TensorFlow variable name will result in a *fetch* operation. Note that multiple variables can be passed to the run method. The state of each variable will be fetched and stored in a standard Python variable.

For example:

Table 8. Fetching TensorFlow variables.

|  |
| --- |
| input1 = tf.constant(3.0)  input2 = tf.constant(2.0)  input3 = tf.constant(5.0)  intermed = tf.add(input2, input3)  mul = tf.mul(input1, intermed)  with tf.Session() as sess:  # Fetch the mul and intermed variables  result = sess.run([mul, intermed]) print(result) |

The TensorFlow graph for this operation appears as follows:



### Inputting data into a TensorFlow graph.

Up to now, we’ve defined our tensors manually. However, in nearly all cases, most data will be loaded via external sources. We can do this in several ways.

* By using the numpy library to create data and then convert it to a TensorFlow tensor.
* By creating a tf.placeholder variable and then feeding data to it via a dictionary.

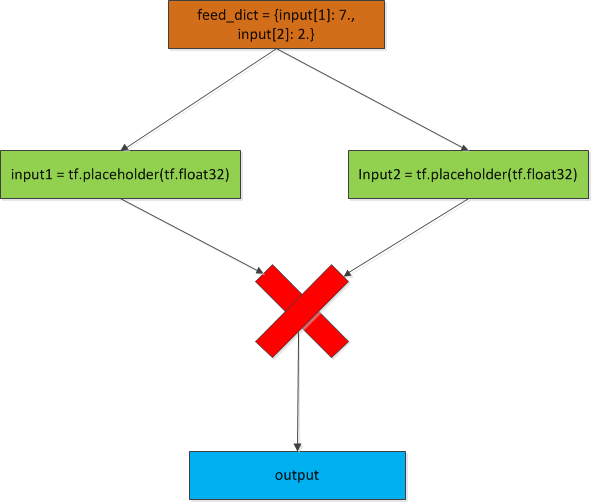
The second way is by far the most preferable method, since method one doesn’t scale well to large data sets.

Here is an example of both methods.

Table 9. Using TensorFlow placeholders.

|  |
| --- |
| import numpy as np  import tensorflow as tf  # Method 1. Converting a numpy array to a tensor.  a = np.zeros((3,3))  ta = tf.convert\_to\_tensor(a)  with tf.Session() as sess:  print (sess.run(ta))  # Method 2. Using a placeholder.  # Here we declare a placeholder as a 32 bit floating point value.  input1 = tf.placeholder(tf.float32)  input2 = tf.placeholder(tf.float32)  output = tf.mul(input1, input2)  with tf.Session as sess:  print (sess.run([output], feed\_dict = {input1:[7.], input2: [2.]})) |

The graph for the above program looks like this:



### TensorScope Variable Scoping.

In a complex system, a TensorFlow model can have hundreds of different variable. This means that the possibility for name clashes (i.e. two variables both called ‘foo’) increases. TensorFlow allows us to sidestep this problem by providing namespaces for variables. For example:

|  |
| --- |
| with tf.variable\_scope("foo"):  with tf.variable\_scope("bar"):  v = tf.get\_variable("v", [1])  assert v.name == "foo/bar/v:0" |

Here we see that we have two variables scopes, ‘foo’ and ‘bar. bar is a nested scope underneath foo. Therefore any variable declared in the ‘bar’ scope will have a fully qualified name of ‘foo/bar/v’

Note that an optional parameter to the tf.variable.scope() method is variable\_reuse = True. This allows for variables to be re-used as necessary. This can be required when implementing Recurral Neural Networks.

# Linear Regression with TensorFlow.

In machine learning and statistics, Linear Regression is the modeling of the relationship between a variable such as Y and at least one independent variable as X. In the linear regression, the linear relationships will be modeled by a predictor function which its parameters will be estimated by the data and is called a Linear Model. The main advantage of Linear Regression algorithm is it’s simplicity and ease of implementation. The biggest disadvantage of Linear Regression is the fact that it only works on data that is linearly correlated, that is, there is a linear relationship between one or more independent variables and the dependent variable.

Here we see an example of some linear data plotted along an x/y coordinate axis plane.

With linear regression, we try and find a ‘best fit’ line that has the least amount of error.

Note here that we have drawn a ‘best-fit’ line through the points so that if we know the line’s equation, we can now use it to predict a new housing price, say, for the value 2.5.

The linear regression model has the following components.

1. The linear equation model algorithm for our linear model.
2. The cost function. We can choose from several cost functions available to us. In our example, we’ll use the *least mean square error* cost function, this function is also known as the L2 function. We’ll use the following formula to evaluate this cost function

The cost function is there to try and determine which possible line is the best fit, that is the line that has the smallest errors.

Note that in every single machine learning example, we’re trying to find the best ‘weight’ and the best ‘bias’ to fit our training data.

1. Our optimizing function. The standard optimizing function is called *Gradient Descent*. The concept here is that if we plot our cost function on an x/y axis, it shows that the cost function is a convex function, that is, it looks like a ‘U’ shape in two dimensions and a bowl shape in three dimensions. Because of its shape, there is only one point on that graph that has the smallest possible cost. The optimizer function tries to find the line equation that matches to that smallest value. There are many other optimizing functions to choose from, however, we’ll just stick with the standard optimizing function. We pass a parameter to our optimizer which represents our *learning rate*. That is how big a jump does the algorithm perform when testing cost function values. If is too large, then the algorithm may not find the minimum value of the cost function. If is too small, then the algorithm may take a long time to find the minimum value of the function.
2. Our *coefficient of determination*. We’d like to know how well our model has done in predicting values, so we can calculate that by determing a value called . This value is also known as the coefficient of determination. Generally, has a value between 0 and 1 which corresponds to a percentage between 0 and 100 %. The higher the value, the better fit the model is. We can calculate this value by the following formula. where SSE is the error sum of the squares. It quantifies how much the values vary around the estimated regression line . SSTO is the total sum of the squares and quantifies how much the data points vary around their mean . For this example, the SSE defines our cost function. The SSTO is defined as. Calculating will give us our .

Once we’ve described our model, we can create our training and test sets and then train the model.

Let’s take a look at an example of a TensorFlow graph running a simple monovariate (single variable) linear regression.

|  |
| --- |
| #!/usr/bin/python3  ''' In this example, we're going to use linear regression in tensorflow to predict housing prices based  on the size of the lot as our features.  '''  import pandas as pd  import matplotlib.pyplot as plt  from matplotlib.mlab import PCA  from mpl\_toolkits.mplot3d import Axes3D  import numpy as np  import tensorflow as tf  import sys  from sklearn import model\_selection  from sklearn import preprocessing  np.set\_printoptions(precision=3,suppress=True)  def normalize\_features(dataset):  mu = np.mean(dataset,axis=0)  sigma = np.mean(dataset,axis=0)  return (dataset - mu)/sigma    rng = np.random  # learning\_rate is the alpha value that we pass to the gradient descent algorithm.  learning\_rate = 0.005  # How many cycles we're going to run to try and get our optimum fit.  training\_epochs = 1000  display\_step = 50  # We're going to pull in a the csv file and extract the X value (RM) and Y value (MEDV)  boston\_dataset = pd.read\_csv('data/housing.csv')  label = boston\_dataset['MEDV']  features = boston\_dataset['RM'].reshape(-1,1)  features = normalize\_features(features)  dataset = np.asarray(boston\_dataset['RM'])  dataset = np.column\_stack((np.asarray(boston\_dataset['RM']),np.asarray(boston\_dataset['MEDV'])))  pca(dataset)  # Here we take our total test data and split it into training and  # testing data. The usual ratio is about 2/3rds training data  # and 1/3 testing data.  train\_X, test\_X, train\_Y, test\_Y = model\_selection.train\_test\_split(features, label, test\_size = 0.33, random\_state = 5)  # Normalize and do feature scaling.  scaler = preprocessing.StandardScaler()  train\_X = scaler.fit\_transform(train\_X)  # This is the total number of data samples that we're going to run through.  n\_samples = train\_X.shape[0]  # Variable placeholders.  X = tf.placeholder('float')  Y = tf.placeholder('float')  W = tf.Variable(rng.randn(), name = 'weight')  b = tf.Variable(rng.randn(), name = 'bias')  R\_squared = tf.Variable(0,name='R\_squared')  # Here we describe our training model. It's a linear regression model using the standard y = mx + b  # point slope formula. We calculate the cost by using least mean squares.  # This is our prediction algorithm: y = mx + b  prediction = tf.add(tf.multiply(X,W),b)  # Let's now calculate the cost of the prediction algorithm using least mean squares  training\_cost = tf.reduce\_sum(tf.pow(prediction-Y,2))/(2 \* n\_samples)  unexplained\_cost = tf.reduce\_sum(tf.square(tf.subtract(Y,prediction)))  R\_squared = tf.subtract(1.0, tf.divide(training\_cost,unexplained\_cost))  # This is our gradient descent optimizer algorithm. We're passing in alpha, our learning rate  # and we want the minimum value of the training cost.  optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(training\_cost)  init = tf.global\_variables\_initializer()  # Now we'll run our training data through our model.  with tf.Session() as tf\_session:  # Initialize all of our tensorflow variables.  tf\_session.run(init)  # We'll run the data through for 1000 times (The value of training\_epochs).  for epoch in range(training\_epochs):  # For each training cycle, pass in the x and y values to our optimizer algorithm to calculate the cost.  for (x,y) in zip(train\_X,train\_Y):  tf\_session.run(optimizer,feed\_dict = {X: x, Y: y})  # For every fifty cycles, let's check and see how we're doing.  if (epoch + 1 ) % 50 == 0:  c = tf\_session.run(training\_cost,feed\_dict = {X: train\_X, Y: train\_Y})  print ('Epoch: ', '%04d' % (epoch+1),'cost=','{:.9f}'.format(c), \  'W = ',tf\_session.run(W), 'b = ',tf\_session.run(b))  print ('Optimization finished')  print ('Training cost = ',training\_cost,' W = ',tf\_session.run(W), ' b = ', tf\_session.run(b),'\n')  print ('R squared = ', tf\_session.run(R\_squared,feed\_dict={X: train\_X,Y: train\_Y}))  plt.plot(train\_X, train\_Y, 'ro',label='Original data')    plt.plot(train\_X,tf\_session.run(W) \* train\_X + tf\_session.run(b), label = 'Fitted line')  plt.legend()  plt.show()    # We're now going to run test data to see how well our trained model works.  print ('Testing...(mean square loss comparison)')  testing\_cost = tf\_session.run(tf.reduce\_sum(tf.pow(prediction - Y, 2)) / (2 \* test\_Y.shape[0]),  feed\_dict = {X: test\_X, Y: test\_Y})  print ('Testing cost = ',testing\_cost)  print ('Absolute mean square loss difference: ', abs(training\_cost - testing\_cost))  plt.plot(test\_X,test\_Y,'bo',label='Testing data')  plt.plot(test\_X,tf\_session.run(W) \* test\_X + tf\_session.run(b), label = 'Fitted line')  plt.legend()  plt.show() |

# Multivariate Linear Regression.

Multivariate linear regression is just linear regression using more than one independent variable. We can simply extend our linear regression formula to accommodate more than one X and one w parameter. The linear regression prediction formula now looks like this:

We can use linear algebra to express this equation as follows:

Where n is the number of features being examined, and is the transpose of the columnar w vector. The gradient descent algorithm then runs for each feature and weight and sums up the partial derivatives of each with respect to the cost function.

Following is an example of using multivariate linear regression with TensorFlow.

|  |
| --- |
| #!/usr/bin/python3  ''' In this example, we're going to use linear regression in tensorflow to predict housing prices based  on the size of the lot and the parent teacher ratio as our features.  '''  import pandas as pd  import matplotlib.pyplot as plt  from matplotlib.mlab import PCA  from mpl\_toolkits.mplot3d import Axes3D  import numpy as np  import tensorflow as tf  import sys  from sklearn import model\_selection  from sklearn import preprocessing  np.set\_printoptions(precision=3,suppress=True)  def normalize\_features(dataset):  mu = np.mean(dataset,axis=0)  sigma = np.mean(dataset,axis=0)  return (dataset - mu)/sigma  rng = np.random  # learning\_rate is the alpha value that we pass to the gradient descent algorithm.  learning\_rate = 0.0001  # How many cycles we're going to run to try and get our optimum fit.  training\_epochs = 1000  display\_step = 50  # We're going to pull in a the csv file and extract the X value (RM) and Y value (MEDV)  portland\_dataset\_features = pd.read\_csv('data/portland\_features.dat',delim\_whitespace=True)  portland\_dataset\_label = pd.read\_csv('data/portland\_label.dat')  portland\_dataset\_features.columns = ['Living Area','Num Bedrooms']  portland\_dataset\_label.columns = ['House Value']  #portland\_dataset\_features = normalize\_features(portland\_dataset\_features)  train\_X, test\_X, train\_Y, test\_Y = model\_selection.train\_test\_split(portland\_dataset\_features, portland\_dataset\_label, test\_size = 0.33, random\_state = 5)  scaler = preprocessing.StandardScaler()  train\_X = scaler.fit\_transform(train\_X)  train\_Y = scaler.fit\_transform(train\_Y)  print (train\_X)  print (train\_Y)  # This is the total number of data samples that we're going to run through.  n\_samples = train\_X.shape[0] # m  n\_features = 2 # n  # Variable placeholders.  X = tf.placeholder('float',[n\_samples,n\_features])  Y = tf.placeholder('float',[n\_samples,1])  W = tf.Variable(tf.zeros([n\_features,1]), name = 'weight')  b = tf.Variable(tf.zeros([1]), name = 'bias')  R\_squared = tf.Variable(0,name='R\_squared')  # Here we describe our training model. It's a linear regression model using the standard y = mx + b  # point slope formula. We calculate the cost by using least mean squares.  # This is our prediction algorithm: y = mx + b  prediction = tf.add(tf.matmul(X,W),b)  # Let's now calculate the cost of the prediction algorithm using least mean squares  training\_cost = tf.reduce\_mean(tf.pow(prediction-Y,2))/(2 \* n\_samples)  unexplained\_cost = tf.reduce\_mean(tf.square(tf.subtract(Y,prediction)))  R\_squared = tf.subtract(1.0, tf.divide(training\_cost, unexplained\_cost))  # This is our gradient descent optimizer algorithm. We're passing in alpha, our learning rate  # and we want the minimum value of the training cost.  optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(training\_cost)  init = tf.global\_variables\_initializer()  # Now we'll run our training data through our model.  with tf.Session() as tf\_session:  # Initialize all of our tensorflow variables.  tf\_session.run(init)  # We'll run the data through for 1000 times (The value of training\_epochs).  for epoch in range(training\_epochs):  # For each training cycle, pass in the x and y values to our optimizer algorithm to calculate the cost.  for (x,y) in zip(train\_X,train\_Y):  tf\_session.run(optimizer,feed\_dict = {X: train\_X, Y: train\_Y})  # For every fifty cycles, let's check and see how we're doing.  if (epoch + 1 ) % 50 == 0:  c = tf\_session.run(training\_cost,feed\_dict = {X: train\_X, Y: train\_Y})  print ('Epoch: ', '%04d' % (epoch+1),'cost=','{:.9f}'.format(c), \  'W = ',tf\_session.run(W), 'b = ',tf\_session.run(b))  print ('Optimization finished')  print ('Training cost = ',training\_cost,' W = ',tf\_session.run(W), ' b = ', tf\_session.run(b),'\n')  print ('R squared = ', tf\_session.run(R\_squared,feed\_dict = {X: train\_X, Y: train\_Y}))  print ('Train\_X shape = ',train\_X.shape)  print ('Train\_y shape = ',train\_Y.shape)  plt.scatter(train\_X[0], train\_Y, 'ro',label='Original data - 0')  plt.scatter(train\_X[1], train\_Y, 'bo',label='Original data - 1' )    line\_fit = train\_X.dot(tf\_session.run(W)) + tf\_session.run(b)  plt.plot(train\_X,line\_fit, label = 'Fitted line')  plt.legend()  plt.show()    # We're now going to run test data to see how well our trained model works.  print ('Testing...(mean square loss comparison)')  testing\_cost = tf\_session.run(tf.reduce\_sum(tf.pow(prediction - Y, 2)) / (2 \* test\_Y.shape[0]),  feed\_dict = {X: test\_X, Y: test\_Y})  print ('Testing cost = ',testing\_cost)  print ('Absolute mean square loss difference: ', abs(training\_cost - testing\_cost))  plt.plot(test\_X,test\_Y,'bo',label='Testing data')  plt.plot(test\_X,tf\_session.run(W) \* test\_X + tf\_session.run(b), label = 'Fitted line')  plt.legend()  plt.show() |

# Logistic Regression.

Logistic regression is a machine learning algorithm that can be applied to binary classification problems. A binary classification problem is one where a data point can be in one of two classes. For example, predicting based on some value whether the person’s gender is male or female, or based on some other data whether the person is an adult or a child (for filtering purposes).

Logistic regression has some similarities to linear regression, however, the cost function to calculate the best fit is quite different. This is because we can no longer estimate cost based on the magnitude of the error. In binary classification, we only have two possible error values ‘0’ or negative and ‘1’ or positive. Linear regression cost functions calculate the magnitude of the error, which isn’t meaningful here since there is effectively no difference in classification between a large error and a small error. There is only a positive or a negative value.

This algorithm is called *logistic* regression because it employs the natural logarithm *ln()* as a factor in determining our cost function. In fact, the standard cost function used for binary classifications is of the form: . This function gives *sigmoid* curved, so-called because the curve is shaped similarly to the letter ‘S’. Here is an example of a sigmoid graph.



As with linear regression, we combine weights and features (w and x) along with a bias (b) to determine a predicted value ‘y’. However, as we mentioned above, the value of y will vary from 0 to 1 as a probabilistic value rather than an absolute error value. Following is an example of using the logistic regression equation with y as the predicted output, ‘b’ as the bias or y-intercept term, and w as the coefficient for the feature value x.

Logistic regression models the probability of any given data point belong to the *default* class (in other words, the first class defined).

As an example, if we were predicting the gender of a person as a dependency on their height, then the default first class could be male and the logistic regression model is defined as the probability of a person being a male depending on their height.

More formally, we can define this as:

Note that probabilities are not in and of themselves defined as binary values (0 or 1), therefore, we’ll need to transform these values into binary values in order to actually make a probability prediction.

We can transform our probability function as follows:

This can be further transformed by taking the natural logarithm ln() to both sides.

Note that this equation is now linear again, and the input on the left is now the log of the probability of the default class.

The log term on the left side of the equation is the odds of the default class. Basically it’s the ratio of probability of the event happening divided by the probability of the event not happening. Because we transform the left hand side using the natural log, we call this the *probit*, or log-odds.

We can use this to now calculate our loss function for our logistic regression example. We use an algorithm called *cross entropy*.

Cross entropy is defined as: where y is what we’re predicting and y’ is what we know to be true. We take the mean of the cross entropy as our loss value. In TensorFlow, there are a number of supplied methods that will calculate this for you.

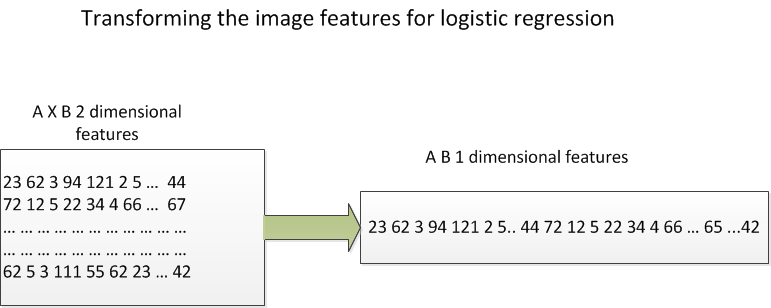
Here is a simple example of binary classification using logistic regression.

|  |
| --- |
| #!/usr/bin/env python3  import tensorflow as tf  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn import model\_selection  import sys  gender\_df = pd.read\_csv('data/binary\_data.csv')  # Shuffle our data  gender\_df = gender\_df.sample(frac=1).reset\_index(drop=True)  # We'll go ahead and split the data set into training and testing parts.  # 70 per cent will go to training, the rest to testing.  train\_x,test\_x, train\_y, test\_y = model\_selection.train\_test\_split(gender\_df['HEIGHT'],gender\_df['GENDER'],test\_size = 0.3)  n\_samples = train\_x.shape[0]  # These will be the placeholders for the testing and training data  x = tf.placeholder(tf.float32)  y = tf.placeholder(tf.float32)  # Variables for the weight and bias.  W = tf.Variable(0,dtype = tf.float32)  b = tf.Variable(0,dtype = tf.float32)  # This is our activation function to determine the probability  # of our gender based on height.  activation = tf.nn.sigmoid((W \* x) + b)  # Set our alpha value for the optimizer.  learning\_rate = 0.001  # cross\_entropy is our cost function.  cross\_entropy = tf.reduce\_mean(-(y\*tf.log(activation) + (1 - y) \* tf.log(1-activation)))  # We'll use a standard gradient descent optimizer.  train\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cross\_entropy)  with tf.Session() as sess:  sess.run(tf.global\_variables\_initializer())  # Now train our model.  for epoch in range(1000):  \_,l = sess.run([train\_step, cross\_entropy], feed\_dict = {x: train\_x, y:train\_y})  if epoch % 50 == 0:  print ('loss = %f' %(l))  # Now let's see how our model performed on the test data.  correct = tf.equal(tf.argmax(activation,1), tf.argmax(y,1))  accuracy = tf.reduce\_mean(tf.cast(correct,'float'))  print ('Accuracy: ', sess.run(accuracy,feed\_dict = {x: test\_x, y:test\_y})) |

# Multiclassification with Logistic Regression.

Whereas the last example only contained two classifications, male and female, many classifications divide data into more than two classes. A classic example of this is the MNIST image database. This is a well known data store of 55,000 greyscale images of 28 X 28 pixels that contain hand drawn digits from zero to nine. We can use logistic regression to predict what value is represented by an image.

Let’s see how this will work. First, we’ll convert each image from a 28 X 28 two dimensional image into a one dimensional image of 784 values ranging from 0 (White) to 255 (black). Other values represent various shades of grey.



This is known as *matrix unrolling*.

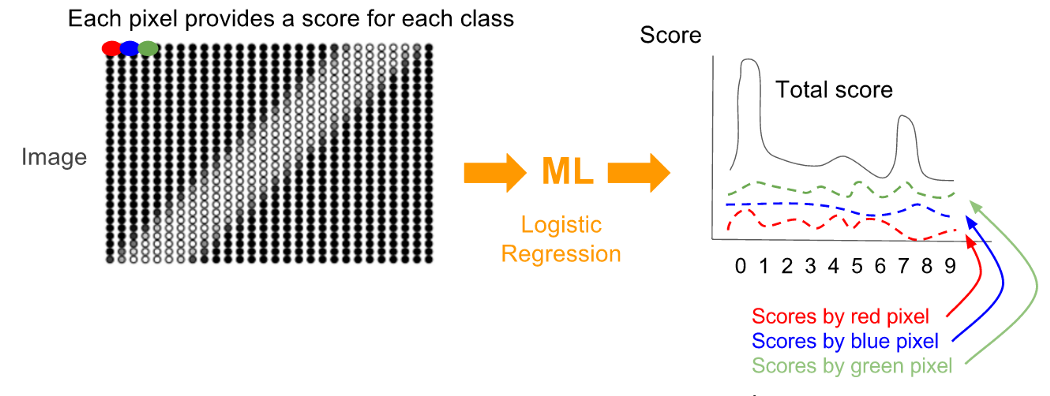
Our prediction needs to be in the values from zero to nine, which means that we have to convert the probabilities (say, for example 2.3 or 11) into something in the range of the classes [0 – 9]

To fix this, we have a prediction vector y, which is a single column vector. Each element in this vector represents the score of what the logistic regression model thinks is likely to be a particular class. In the exampke below class ‘1’ is the prediction since it has the highest score.

[ 1.3 33 2 1.2 3.2 0.5 3 9.2 2 1 1]

Where the score (probability) for class 0 is 1.3, the score for class 1 is 33 and so on.

How do we get this vector? Each pixel provides its own score for each class, i.e. pixel 0 will provide what it thinks is the most likely class that the image belongs to, pixel 1 will do the same, and so on up to pixel 768.



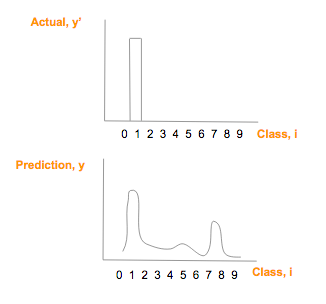
Now, for each score, we have to find out how badly or well the algorithm’s prediction was. However, we can’t use a standard linear regression loss function. Those compute *magnitudes* of error. In our case, it doesn’t matter what the magnitude is, error is error whether it’s a small or a large value. We’ll use a different type of loss function, called the *cross-entropy* function.

This function has multiple steps.

1. One-hot vectors

We’ve gone ahead and transformed our prediction (y) into a vector of scores, we should also transform the actual image class values into a vector as well so that we can compare each element (pixel) in the predicted with the actual value. Each element in the actual value vector will be ‘0’ except for the value that corresponds to the actual number drawn in the image. So, for example, an image of 2 will have a one-hot vector that looks like this:

[0 0 1 0 0 0 0 0 0]. Let’s see a plot of the comparison between the predicted value vector and the actual value vector.



1. Calculate the probability distribution using the softmax algorithm.

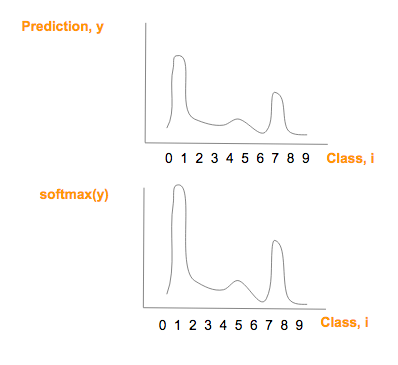
Using the cross entropy will allow us to compare the similarity of two graphs. To do this, however, we need to convert both the actual outcome vector (y’) and the predicted outcome vector (y) into a probability distribution. By this we mean

* The probability score of each class has to be between 0 and 1.
* The sum of all the probabilities/scores for each class has to be 1.

The outcome vector, being a one-hot vector already meets these constraints, however, we need to transform the prediction vector into a probability distribution using softmax. The softmax algorithm looks like this:

Where I is the class, in this case of 0, 1,2, …9. Every element of the prediction score is exp’d and then divided by the sum of the exp’d total.

Here’s a graph of the softmax values after transformation.



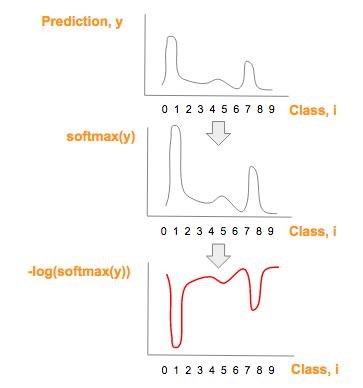
Note that the graphs are similar, except that the softmax values having a larger maximum value and a smaller minimum value.

Now we can apply the cross-entropy formula to the softmax transformed prediction vector.

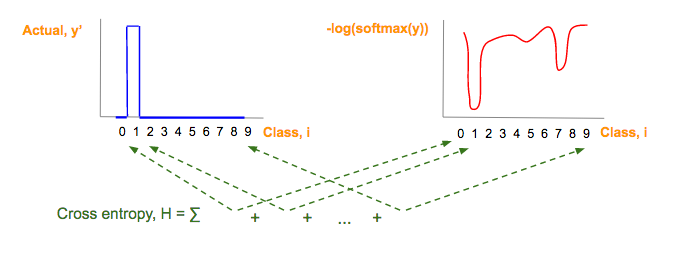
The cross entropy formula is as follows:

This formula will calculate our loss as we run our optimizing function.

Let’s now see what our transformed prediction vector looks like graphically.



Now, we can simply multiply our actual probabilities as referenced by our one hot vector vs. the predicted values. Note that since all values in the one-hot vector other than the actual value are zero, the multiplication gives us a 1 value in the right position (hopefully) for the predicted value vector.



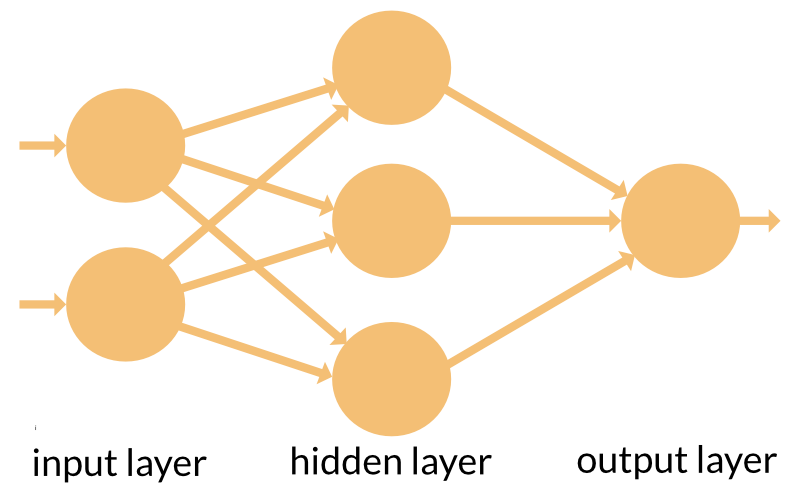
# Introduction to Neural Networks with the Perceptron

In this module, we introduction the concept of artificial neural networks (ANNs) and the perceptron, which is a type of node in an ANN. ANNs work very differently than most type of programming logic. Instead of applying a set of rules in order to tell a computer what to do, the ANN allows the computer to learn on its own in order to achieve an optimum result.

We use ANNs for many different purposes. For example:

* Data classification, for example image recognition. Is the picture of a dog or a cat?
* Anomaly detection. Is this seismic activity showing signs of an earthquake about to happen?
* Predictions and approximations. Will this storm turn into a hurricane?

There are many different types of ANN’s, but the general structure of an ANN looks something like this:



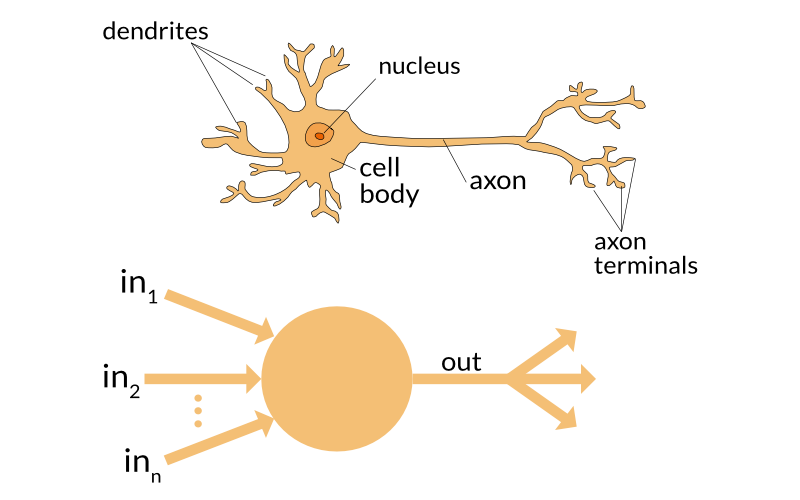
ANNs can have one or more nodes in the input layer, one or more hidden layers (A neural network with multiple hidden layers is known as a *deep* neural network), and an output layer.

Neural networks are not programmed or configured to perform a specific task, instead, they are *trained* to do so, in the same fashion that a human brain is trained to perform a specific skill.

There are three general types of training that a neural network can undergo.

1. Supervised learning. If we have a large enough set of test data with known results, then we can feed a dataset into the neural network to train it. We can then compare the trained output with the actual output. Adjust the network and repeat until a satisfactory result is achieved.
2. Unsupervised learning. If we don’t have any known test data, and if we can figure out some sort of cost function from our desired behavior, i.e. cost function of 0 means that the unsupervised neural network doesn’t drive the car off the road, for example, then the network can adjust its parameters on the fly while it works on the real production data.
3. Reinforced learning. Also known as the *carrot and stick* approach. This can be used if the ANN generates continuous action. I.e. Follow the carrot in front of your nose. If the ANN goes the wrong way, then penalize it. Over time, the ANN learns to prefer the right kind of action and avoid the wrong kind.

The basic ingredient of a neural network is the *neuron* or neural network node. This neuron works very similarly in concept to a neuron in the human brain. A neuron has one or more input channels, a processing stage and an output that can fan out to other neurons. Here is a simple conceptual diagram of a neuron in an ANN vs. a biological neuron.



Let’s take a closer look at the artificial neuron.

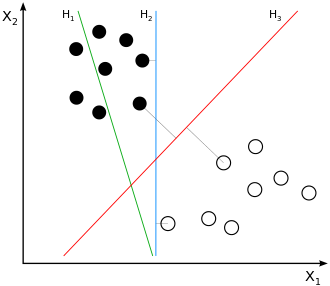
There are three basic steps to an artificial neuron.

1. Each input gets scaled up or down. When a signal comes in, it gets multiplied by a *weight* value that is assigned o that particular input. I.e. if a neuron has three inputs, then it has three different weights, one for each input. Each weight can be adjusted individually. The learning phase consists of the neural network adjusting each weight based on the error of the last test results.
2. All of the signals are summed up. All of the modified input signals are summed up to a single value, including a bias value which is added to the sum. The bias, as well as the weights, is adjusted during the learning phase. When an ANN is first defined, we set the weights and biases to random numbers. After each training iteration, the weights and biases are gradually changed so that the next result is close to the desired outcome. This is the “learning” portion of the ANN training.
3. Finally, the results of the summing up is turned into an output signal which is created by feeding this summed value into an activation function which outputs a results which can be sent to another node in the network.

# The Support Vector Machine.

The Support Vector Machine is a learning algorithm that is used to solve classification problems. SVM’s were invented in 1963 by two Russian mathematicians, Vladimir Vapnik and Alexey Chervonenkis. This algorithm, however, did not gain popularity in the field until the early 1990’s, when Vapnik emigrated to the United States. At the time the neural network model was falling out of favor with ML researchers due to the inability of computers at the time to be able to train the networks and handle the massive data sets required SVM’s performed better than the ANNs of the time and didn’t need nearly as much computing power or data storage.

The SVM uses the concept of a *hyperplane* to separate groups of data into their appropriate classes. Consider the following graph.



Here we see we have to classes of data points. Let’s look at each potential hyperplane to see which one best separates the two groups.

H1  clearly does not separate the points at all. This makes it completely unsuitable as a potential hyperplane for our SVM. H2 does separate the two groups, however it does not do an optimal job of separating them. H3 however, does the best job of the three lines in separating the points, allowing for other points in the group that may fall closer to the plane boundary.

SVM’s have many uses, including

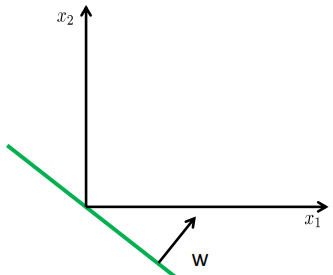
* Image classification
* Protein classification
* Document classifications such as spam filtering

The concept behind the SVM is to be able to pass a linearly separable hyperplane between two groups through a dataset in order to clearly identify which data points belong to which groups.

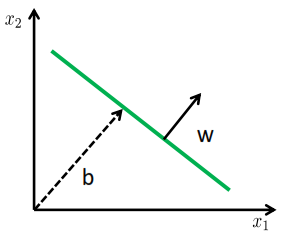
To do this, the SVM must calculate the slope of the parameters of the the line by finding the magnitude of the optimal parameter vector *w*. To do this, we perform calculations using the following formula, where the following holds true:

* X denotes the data points
* Y denotes the labels
* W denotes the weights vector
* b denotes the bias.

The main goal of the SVM is to find the optimal weight vector. The bias is used to allow shifting of the hyperplane in the coordinate space. Without a bias, or setting the bias equal to zero, the hyperplane would always pass through the (0,0) coordinate, like so:

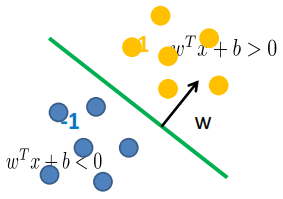


With the bias added, the hyperplane shifts in the coordinate plane.



The equation to determine this hyperplane is given as:

Where *w* is a vector of the weights. Now we can define a function for each data point. If the value of that function is less than zero, then it goes to the -1 class. If it is greater than or equal to zero, it goes to the +1 class.



Again, when determining the hyperplane we want the maximum margin between the groups. The ponts that lie on this margin are called the *support* *vectors*. It’s these support vectors that determine the margin. Adding new data points not on this margin does not affect the model’s accuracy.

As with any ML model, when training it, we have to calculate our loss function. Loss funtions only matter in SVM’s using soft margins as opposed to hard margins. Generally the loss function that is used for calculating soft margin SVM’s is called a *hinge loss function*. The hinge loss function looks like this:

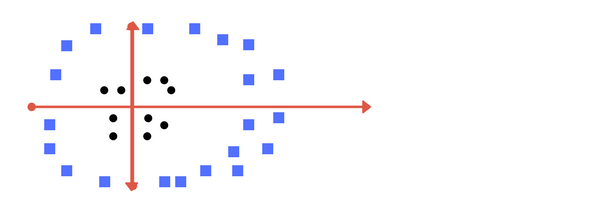
The + operator means that a is a if the value of a is > 0, otherwise, the value is 0.

Notice that the value of the function is 0 if a particular training element is on the correct side of the decision boundary, otherwise the loss is proportional to how far away the point is from the boundary. Since the hinge loss is a convex function, there is only one minima, unlike with neural networks.

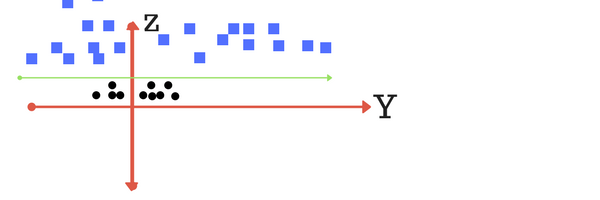
What happens if we have data points inside the margin? If we require that all data points be outside or on the margins than this Is called a *hard margin*. Hard margins are very sensitive to outlier data points. This means that this model may suffer from overfitting and not generalize well to test or production data. This means that we probably need a more flexible approach. SVM’s have the ability to handle *soft margins.* The trick is to find a balance between keeping the margin as side as possible and limiting the margin violations. We can control this by using a hyperparameter called *c*. The larger the c value, the smaller the margin is, and the more outliers can be tolerated. The smaller the c value the wider the margin is but the more outliers you may encounter. Usually setting the c value as small as possible tends to be preferred as it keeps the algorithm from overfitting the training data.

What happens if the data isn’t linearly separable?

Consider the following graph.



Here we have two sets of points, but there is no possible line that can separate the two groups. So, what do we do? Consider a transforming function this is applied to each data point in both sets. In this case,, we add a dimension to each point transforming it from a 2-d graph to a 3-d graph.



Note now that we’ve added a dimension that the points now become linearly separable using a 2-d hyperplane. The problem with applying this transformation function to each data point is that it becomes computationally expensive to do so, hence we can use a short called called the *kernel trick*.

The kernel trick can be defined as follows:

“If an algorithm is described solely in terms of inner products in input space then it can be lifted into feature space by replacing occurrences of those inner products by k(x,x’); this is sometimes called the kernel trick.”

Rasmussen and Williams, “Gaussian Processes for Machine Learning,” MIT press, 2006.

Why is this interesting? It turns out that we can write linear ML models and apply them to non-linear data without having to know anything writing non-linear learning models. Note that this kernel trick applies only to the data points (X), and not the labels (Y). Additionally, we don’t have to apply our transforming function to every data point in the graph. Note that the computing cost goes up exponentially, the greater the dimensions applied to each data point. With the kernel trick, we can calculate the inner dot products for each data point and then simply raise it to the correct power. This means that we can apply this to any set of n dimensions without having to increase our computing complexity.

Fortunately, ML libraries such as TensorFlow don’t need you to calculate these things manually. They supply different *kernels* that you can apply to your model to get your best results. There are a number of kernels available in the standard libraries, or, in the unlikely event you need to do so, you can create your own and apply it.

The most common non-linear kernel applied is called the *Gaussian* or *Radial Basis Function Kernel.* This is a good second choice if you are finding that the standard linear kernel doesn’t fit your data very well.

To sum up, SVM’s are a good choice as a classifier algorithm. At this time, however, TensorFlow doesn’t have very good support for this model. TensorFlow cannot (as of this writing) support kernels other than linear, which restricts its usefulness. It is suggested that if SVM is needed, another framework, such as sklearn be used rather than TensorFlow.

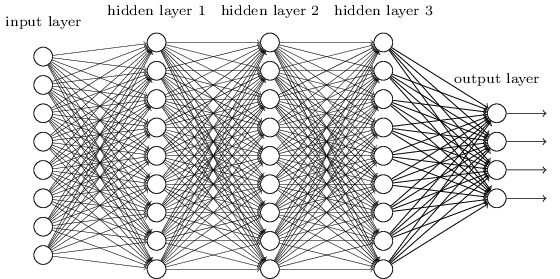
# Deep Learning with Neural Networks and TensorFlow

Building ANNs is where TensorFlow really shines. This is because TensorFlow makes it very easy to construct complex neural networks without having to write a lot of code. Additionally, TensorFlow can run on many different platforms, including mobile platforms, so it becomes easy to train a model and then export it to the mobile platform to run.

As we’ve seen from earlier chapters, Machine Learning allows the computer to ‘learn’ how to perform a task rather than use rules-based systems where all behavior is pre-defined. Neural Networks have the following general structure

* An input layer. This is where the inputs, weights and biases are fed into the DNN. When we start the training, the weights and biases are usually set to random numbers. (Setting them to zeros is a bad idea as the DNN will never actually learn anything).
* One or more hidden layers. These hidden layers sum up the inputs from the previous layer and run it through an activation func tion. If the threshold is met, the neuron outputs a ‘1’, otherwise, it outputs a ‘0’.
* The output layer. This layer takes all the input from the last hidden layer and runs it through an activation function to give us predicted output.

A graphical picture of a DNN might look like this:



Note that in a conventional neural network, every node in each layer is connected to every node in the adjacent layer.

As we’ve seen in a previous chapter, we introduced the concept of the *perceptron*. A perceptron is a very simple neural network consisting of a single node. This node takes some inputs, runs them through an activation function and produces some outputs. We then check to see if the predicted outputs match the expected outputs. If not, we repeat the process, adjusting the weights and bias until the model give us the outputs we expect. Unfortunately, perceptrons are very limited. They only work on linearly separable data (for example, we can model the AND, OR and NAND boolean functions, but not XOR).

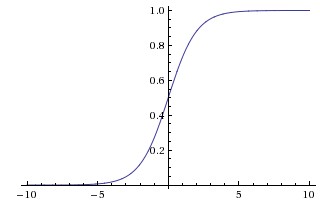
## Activation Functions for Neural Networks

Every activation function for an ANN takes a single number, and performs a certain fixed mathematical operation on it. There are a number of these functions that you may see in practice.

1. Sigmoid Function.
2. Tanh function
3. Relu function

### Sigmoid function

The sigmoid activation function as the form and looks like this:



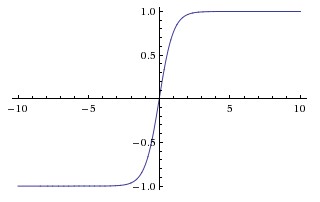
Note that the sigmoid function *squashes* real numbers to a range between 0 and 1. In particular, large negative numbers go to 0 and large positive numbers go to 1. This function has been historically used because it is a nice interpretation of how the firing rate of a biological neuron is modeled (i.e. not firing at all is a 0 and a fully saturated neuron firing at 1 with a maximum frequency). Sigmoid functions have two undesirable properties which is why they have fallen out of favor with researchers and scientists.

1. Sigmoids saturate and kill gradients. An undesirable property of the sigmoid neuron is that when the neurons activations saturates at either the tail of 0 or 1, the gradient becomes nearly zero. When doing backpropagation this local gradient is multiplied to the gradient of this neuron’s output for the whole scenario. A small local gradient basically means that the neuron will no longer ever effectively fire, which “kills” the neuron. This means that the architect must pay special attention when initializing the weights to prevent saturation. A set of weights that is too large means that the network will quickly become saturated and fail to learn.
2. Sigmoid outputs are not zero-centered. This becomes undesirable because neurons in later layers will be receiving data that isn’t centered around zero. This will affect the backpropagation algorithm given that the gradient can be all positive or all negative (i.e. never zero). This can produce some undesirable results. This is, however, more of an inconvenience rather than a fatal flaw as once the gradients are summed up across a batch of data, the final update for the weights can have positive or negative signs, which may mitigate the problem.

### Tanh function

The tanh function squashes a real-valued number into the range -1 to 1. Since the output is centered around zero, it is preferred to the sigmoid function. The function for the tanh is as follows:

Plotting this function displays the following:



### ReLU function

The ReLU function is now the standard activation function for use in neural networks. The function has the form: This means that this function is simply thresholded at zero. The function plot looks like this:



ReLU’s have a number of benefits, including:

1. It greatly accelerates the convergence of stochastic gradient descent compared to the earlier functions.
2. ReLU is easier to compute. It’s implemented by thresholding a matrix of activations at zero.

There are, however, some disadvantages to ReLU’s. Mainly, a ReLU can die if a large gradient flowing through the neuron causes the weights to update in such a way (i.e. the weight value approaches zero) so that the ReLU never again activates. This is especially true if you set the learning rate too high.

## Hidden Layers and Back Propagation

In 1986, a breakthrough came about in the field of neural network research. A paper published by three authors, Geoffrey Hinton, David Rumelhart and Ronald Williams entitled “*Learning representations by back propagating errors”*  was published. This paper introduced the following concepts:

1. Hidden Layers. Neuron nodes stacked in between the input and output layers to allow neural networks to learn more complex features.
2. Back Propagation. This is a procedure to repeatedly adjust the weights so as to minimize the difference between the actual output and the desired output

Let’s take a look at these two ideas.

## Hidden Layers.

“Hidden layers” are simply layers in the neural network that sit between the input and the output layers. The nodes in a hidden layer take their inputs, weights and biases either from the input layer, or from another hidden layer. Why do we need or want hidden layers? Often times, we need to model complex functions that take multiple simple functions and conjugate them together. Additionally, data that isn’t linearly separable requires a hidden layer in order to properly train the model. Neurons in a hidden layer take inputs and weights and a bias from all nodes that link to it and add them up, afterwards, it passes the total sum through an activation function, however, unlike the perceptron, this activation function is *sigmoid* in nature. That is, it approaches 1 as the function moves to the right and zero as the function moves to the left. It then gives out a value of ‘1’ or ‘0’, depending on what the threshold for the functions activation is.

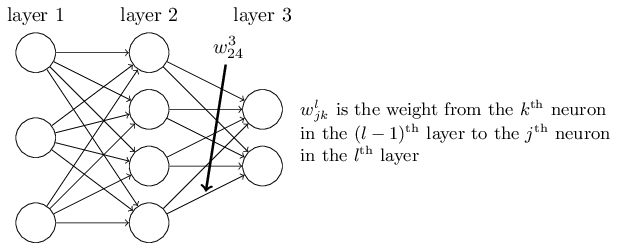
How many hidden layers are required? There is an entire area of machine learning centered on tuning the hyperparameters for your ML algorithm correctly, however, a good rule of thumb is that it is rarely necessary to have more than one or two hidden layers. As for the number of nodes, a good rule of thumb is to take the mean between the number of nodes in the input layer and the number of nodes in the output layer.

## Back propagation

As with other machine learning algorithms, we need to be able to compute a cost function in order to properly assess how much training the model needs in order to give us optimum results. This is the reason that we need a sigmoid activation function for our network nodes. How does this work with neural networks?

Let’s first define some notation. We’ll refer to the weights our ANN as follows:

We’ll use to denote the weight for a connection from the in the to the neuron in the layer. So, for example, the diagram below shows the weight on a connection rrom the fourth neuron in the second layer to the second neuron of the third layer of a network.



Using this notation, the activation function of the neuron in the layer is related to the activation functions of the nodes in the by the following equation

=

This equation describes the sum over all neurons k in the

We can rewrite this expression in a matrix form. We can define a *weight matrix* for each layer l. The entries of the weight matrix are just the weights connecting to the layer of neurons. Each entry in the jth row and kth column is . We also identify a bias vector where each component of the b ias vector is the value . Finally, we define an activation vector whose components are the activations of the function We can vectorize the function sigma so that we can re-write the above function like this:

=

The goal of backpropagation is to calculate the partial derivatives of and of the cost function C with respect to any weight w or bias b in the network and then apply them to the network nodes as appropriate. Writing backpropagation as an algorithm, you would have the following steps.

1. Input x: Set the corresponding activation for the input layer
2. Feedforward: For each l = 2,3,…L, compute
3. Output error:
4. Backpropagate: for each l = L -1, L-2,… 2 compute: =

Back propagation just means that we compute the error vectors : going backward from the output through to the first hidden layer. This backward movement is a consequence of the fact that the cost is a function of outputs from the network. We need to use the *chain rule* to walk backward through the layers to obtain usable expressions.

## Neural Network Architecture.

Finally, let’s examine how we can define our architecture for an ANN.

It’s important to note that a neural network with one hidden layer is a *universal approximator*. That is, it can approximate any continuous function , however, in practice, most people add at least one more hidden layer to get optimal results. It is rarely necessary to add more than three hidden layers as the performance does not seem to increase significantly. Additionally, too many hidden layers can lead to *overfitting* of data.

This limitation of NN nodes, however, is not usually considered the best way to allow generalization of data. Instead, using techniques such as L2 regularization seems to be the better approach. This is because smaller NN’s are harder to train with methods such as Gradient Descent, GD optimization with smaller neural networks can lead to converging on a local minima that isn’t the best fit. Larger neural networks have more local minima, but each of these minima have a better chance of approaching the global minima for the network.

In practice, what this means is that a small NN can display significant overfitting of data unless you get lucky and converge to a minima that closely approximates the global minima. Larger neural networks have a better chance of finding a good solution that doesn’t rely on luck of the random initialization.

The best way to control overfitting in your network is to regularize your parameters by modifying the *lambda* hyperparemeter.

# Convolution Neural Networks

Let us consider the following problem.

Let’s say we want to train a neural network to perform image recognition for us. Each image is 1000 x 1000 pixels in length with a color channel feature of 3. This means that the input vector is (1000 x 1000 x 3) in length, with each input neuron connected to each neuron in the first hidden layer and so on. This very quickly starts to become infeasible both from a storage and a computing standpoint. The end result is that DNN’s are somewhat limited in their ability to process images as it quickly becomes unscalable with larger and larger images. For example, in 2012 Geoff Hinton, et.al. published a paper on image recognition using neural networks that contained 60 million features. Standard image recognition problems can contains features on the order of 100 million pixels or more per image. Because of this, we need to come up with a different approach that still allows us to use the power of neural networks to learn.

Convolutional neural networks are very similar to standard DNN’s as discussed in the last module. They are made up of neurons that have tunable weights and biases. Each neuron receives some inputs, performs a dot product and may follow it with an activation function. The entire network will still express a single differentiable score function from the raw image pixels on one end to the class scores on the other. As with DNN’s, they have a loss function on the last fully connected layer and all of the tips and tricks for developing and using ANN’s still apply.

What changes? CNN’s are explicitly designed for image processing, so they assume that the inputs are images and not some other type of data, which means that they can be optimized for this specific type of input. These optimizations are encoded as properties of the CNN’s architecture. Because of this, the feed forward function becomes significantly more efficient and easier to implement and vastly reduces the amount of parameters in the network.

One of the big differences between DNN’s and CNN’s is that the CNN architecture is now three dimensional rather than the two dimensional architecture in a normal DNN. Here’s a picture showing the differences between the two.



The above picture shows a standard DNN.



The figure above shows an example of a CNN architecture

Note that the CNN arranges its neurons along three dimensions (width, height, depth). Every layer of a CNN transforms the 3d input volume to a 3d output volume of neuron activations. In this example the input layer (in red) holds the image, so its width and height would be the dimensions of the image and the depth would be 3. (Red, Green, Blue channels).

## CNN Layers

Unlike standard DNN’s, Layers in CNN’s have specific functions and perform different tasks. CNN’s contain three main types of layers

1. Convolutional Layers
2. Pooling Layers
3. Fully connected layers (just like DNN’s)

We can then stack these layers together to form a CNN. Here’s a graphical representation of what a CNN might look like:



This picture shows the activations of an example CNN architecture. The initial volume stores the raw image pixels (far left), and the last volume stores the class scores (far right). We’re laying out each 3-d slice in rows. Here we’re only visualizing the top 5 scores and print the labels of each one.

Let’s now describe each type of CNN layer.

## The Convolutional Layer

The convolutional layer is the foundational building block of a CNN. The convolutional layer does most of the heavy lifting in our CNN.

The CONV layer’s parameters consist of a set of learnable filters. Each filter is small spatially along width and height) but extend through the full depth of the input. For example, an individual filter on the first layer of a CNN might have a size of 5x5x3 (five pixels wide, five pixls high and 3 in depth because of the color channels red, green and blue).

During the forward pass of the network, we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height, we can product a 2-d activation map that gives the responses of that filter at every spatial position along the image. The idea is that the network will learn filters that activate when they see some sort of visual feature such as the edge of some orientation or some splash of color on the first layer. These shapes evolve in higher order layers to be more specific (such as wheel patterns).

Once this layer is finished, we now have an entire set of filters in each layer and each of them will produce a separate 2-d activation map. We then stack these maps along the depth dimension like so:



### Convolutional layer hyperparameters.

There are three hyperparameters which can be tuned at the CONV layer.

1. The depth of the output. This corresponds to the number of filters we would like to use. Each filter is learning to look for something different in the input. For example if the first CONV layer is taking as its input the raw image then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color.
2. The *stride* at which we slide the filter. A stride of 1 means that we’re moving the filter one pixel at a time. A larger size means that the stride jumps n pixels, depending on the value of n. The larger the stride, the smaller the output volumes will be spatially.
3. The zero-padding. It can become convenient to pad the input volume with zeros. The size of this zero-padding is a hyperparameter. We use this to control the size of the output volumes so that the spatial size of the input volume and the output volumes are the same.

## The Pooling layer.

As the filters in the convolution layer slide over the patch of image pixels, the pixel values get multiplied by weights in order to indicate whether or not a particular feature has been recognized. The pooling layer takes these recognized features and *pools* them together. For example, if the filter is sliding over an image and it recognizes a features, such as a line, or a curve, it will transmit this to the pooling layer. The pooling layer will then combine the features together, reducing the size of the image. We can then send this pooled image to another convolution layer, which will again apply a number of filters to that pooled image to see if it can recognize more complex shapes, such as an eye or a nose or a mouth (if the image being looked at is a face).

Here is what pooling looks like graphically.

Some researchers have proposed doing away with the pooling layer altogether and simply replacing it with another conv layer with a larger stride. It is very possible that in the next few years, pooling will disappear from the architecture of a CNN altogether.

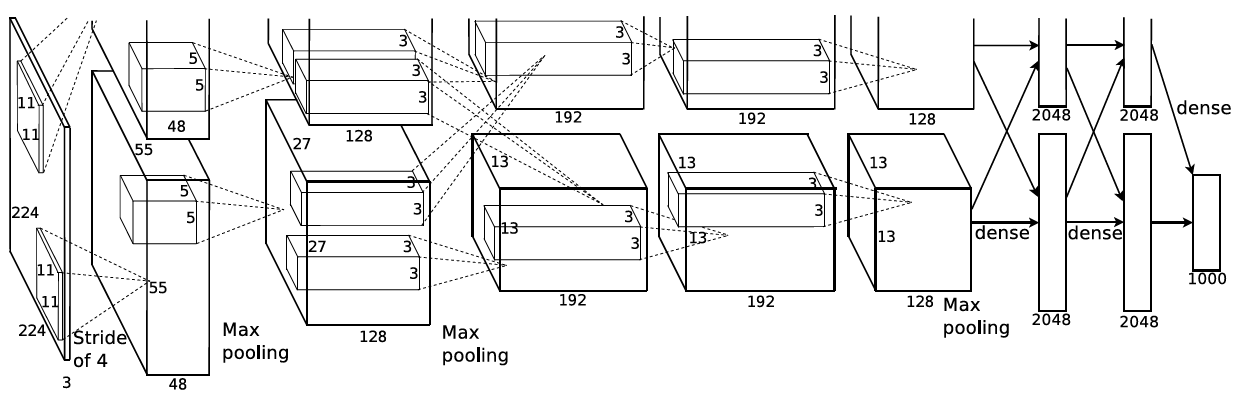
## The fully connected layer

The fully connected layer is just a normal NN layer that connects the pooling layer and sums up the weights and features with the biases and runs them through the activation function. The output of the FC layer is a score that indicates the likelihood of the image class.

Note that FC layers and CONV layers differ only in that the CONV layer only connects to local regions in the input whereas the FC layer connects to all neurons in the previous layer. This means that it is easy to convert an FC layer to a CONV layer.

## CNN architectures.

Here is a graph showing the ALEXnet CNN architecture.



The standard form of a CNN architecture is:

1. The input layer
2. One or more convolution layers + RELU
3. A pooling layer
4. Repeat steps 2 and 3 as required
5. One or more fully connected layers
6. An output layer.

A summary way to describe this architecture is as follows:

where the \* indicates repetition, and the POOL? indicates an optional pooling layer. Moreover, N >= 0 (and usually N <= 3), M >= 0, K >= 0 (and usually K < 3).

It’s usually a good idea to prefer a stack of small convolution filters rather than one larger filter. For example, stacking three 3X3 convolution layers on top of each other with RELU’s between them has an arrangement such that the first CONV layer has a 3X3 view of the input. A connected neuron in the second CONV layer has a 3X3 view of the first layer and hence, by extension, a 5X5 view of the entire input volume. A neuron on the third CONV layer has a 3x3 view of the second layer, and, by extension a 7X7 view of the entire input volume.

Having one large 7X7 conv layer means that a linear function is being computed over the input, whereas with three stacks of 3X3, you can now compute non-linearities which may make the features more expressive. If, for example, we have C channels, then the single 7X7 conv layer contains C X (7 X 7 X C) = parameters, whereas with the three layers, we would have 3 X (C X (3 X 3 X C)) = parameters. Intuitively, stacking smaller CONV layers with smaller filters lets us express more powerful features with a smaller number of parameters.

### Sizing layers.

Some rules of thumb for setting the hyperparameters of a CNN.

The input layer should be divisible by 2 many times. I.e. powers of 2. For example 32 (for the CIFAR-10 data set) 64, 96 (for the STL-10 dataset) or 224 (ImageNet CNN’s), 384 and 512

The conv layer should use small filters (3X3 or 5X5) using a stride of 1 and padding the input volume with zeros so that the conv layer does not alter the spatial dimensions of the input.

The pool layer should use a type of pooling called *max-pooling* with 2X2 fields and a stride of 2. Note that these settings discard 75 % of the activations in the input volume. Setting too large a filter size generally result in worse performance as the pooling is too lossy and leads to reduction of valuable features.

Here is an example, (a written description) of a CNN called *VGGNet.* VGGNet is a CNN architecture developed at the University of Oxford in Great Britain.

|  |
| --- |
| INPUT: [224x224x3] memory: 224\*224\*3=150K weights: 0  CONV3-64: [224x224x64] memory: 224\*224\*64=3.2M weights: (3\*3\*3)\*64 = 1,728  CONV3-64: [224x224x64] memory: 224\*224\*64=3.2M weights: (3\*3\*64)\*64 = 36,864  POOL2: [112x112x64] memory: 112\*112\*64=800K weights: 0  CONV3-128: [112x112x128] memory: 112\*112\*128=1.6M weights: (3\*3\*64)\*128 = 73,728  CONV3-128: [112x112x128] memory: 112\*112\*128=1.6M weights: (3\*3\*128)\*128 = 147,456  POOL2: [56x56x128] memory: 56\*56\*128=400K weights: 0  CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*128)\*256 = 294,912  CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*256)\*256 = 589,824  CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*256)\*256 = 589,824  POOL2: [28x28x256] memory: 28\*28\*256=200K weights: 0  CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*256)\*512 = 1,179,648  CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*512)\*512 = 2,359,296  CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*512)\*512 = 2,359,296  POOL2: [14x14x512] memory: 14\*14\*512=100K weights: 0  CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296  CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296  CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296  POOL2: [7x7x512] memory: 7\*7\*512=25K weights: 0  FC: [1x1x4096] memory: 4096 weights: 7\*7\*512\*4096 = 102,760,448  FC: [1x1x4096] memory: 4096 weights: 4096\*4096 = 16,777,216  FC: [1x1x1000] memory: 1000 weights: 4096\*1000 = 4,096,000  TOTAL memory: 24M \* 4 bytes ~= 93MB / image (only forward! ~\*2 for bwd)  TOTAL params: 138M parameters |

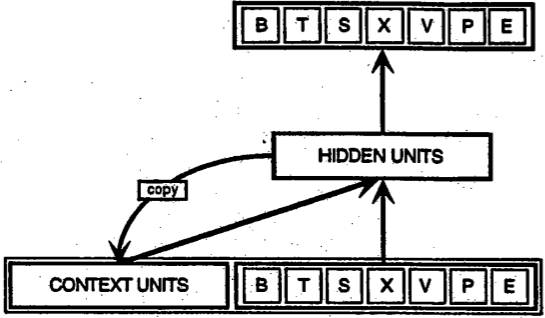
Another note on architectures. Training a CNN can be computationally very expensive, especially on large data sets. Computational expense can very quickly transform into financial expense. While a CNN may be technically feasible, hardware and software resources may make it unfeasible for an organization to train its own CNN’s. Consequently, many organizations, such as Google, have released *pre-trained* CNN’s that can be used for the purpose. Google, for example, offers the *Inception* network. Other companies and organizations have also released CNN’s for general use.

# Recurrent Neural Networks

## The simple recurrent network

The Simple Recurrent Network (SRN) was conceived and first used by Jeff Elman, and was first published in a paper entitled ”*Finding structure in time” (Elman, 1990).* The paper was ground-breaking for many cognitive scientists and psycholinguists. It explored the concept that specific linguistic units such as words and phonemes might be emergent consequences of a learning process operating over the latent structure in the speech stream.

The SRN could be graphically represented like this:



In the previous figure, each box represents a pool of units and each forward arrow represents a complete set of trainable connections from each sending unit to each receiving unit in the next pool. The backward arrow, from the hidden layer to the context layer denotes a copy operation

SRN’s are very simple. In fact, it is really just a three-layer, feed-forward back propagation network. The only proviso is that one of the two parts of the input to the network is the pattern of activation over the network’s own hidden units at the previous time step.

## The vanishing (and exploding) gradient problem.

Like most neural networks, recurrent nets are old. By the early 1990s, the vanishing gradient problem emerged as a major obstacle to recurrent net performance.

Just as a straight line expresses a change in x alongside a change in y, the gradient expresses the change in all weights with regard to the change in error. If we can’t know the gradient, we can’t adjust the weights in a direction that will decrease error, and our network ceases to learn.

Recurrent nets seeking to establish connections between a final output and events many time steps before were hobbled, because it is very difficult to know how much importance to accord to remote inputs. (Like great-great-\*-grandparents, they multiply quickly in number and their legacy is often obscure.)

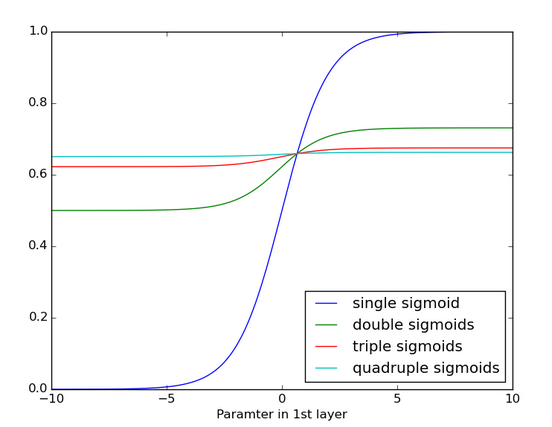
This is partially because the information flowing through neural nets passes through many stages of multiplication.

Everyone who has studied compound interest knows that any quantity multiplied frequently by an amount slightly greater than one can become immeasurably large (indeed, that simple mathematical truth underpins network effects and inevitable social inequalities). But its inverse, multiplying by a quantity less than one, is also true. Gamblers go bankrupt fast when they win just 97 cents on every dollar they put in the slots.

Because the layers and time steps of deep neural networks relate to each other through multiplication, derivatives are susceptible to vanishing or exploding.

Exploding gradients treat every weight as though it were the proverbial butterfly whose flapping wings cause a distant hurricane. Those weights’ gradients become saturated on the high end; i.e. they are presumed to be too powerful. But exploding gradients can be solved relatively easily, because they can be truncated or squashed. Vanishing gradients can become too small for computers to work with or for networks to learn – a harder problem to solve.

Below you see the effects of applying a sigmoid function over and over again. The data is flattened until, for large stretches, it has no detectable slope. This is analogous to a gradient vanishing as it passes through many layers.

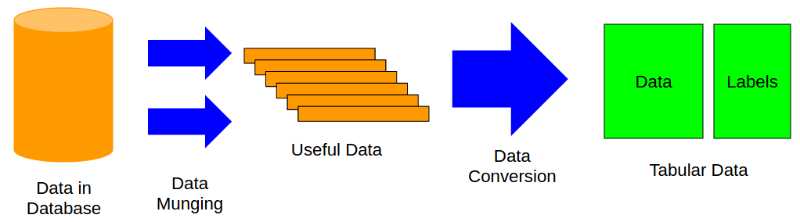


## Long Short Term Memory (LSTM) networks.

# Approaching problems in Machine Learning.

Approximately 60-70 per cent of the time spent by a data scientist is in cleaning and processing data. By this we mean munging data and converting it to a suitable data format so that a ML model can be applied to that data.

Let’s start by convering our input data source into a tabular form. This is usually the most time consuming, difficult, and frankly irritating part of the process. Below is a graphical representation of this.

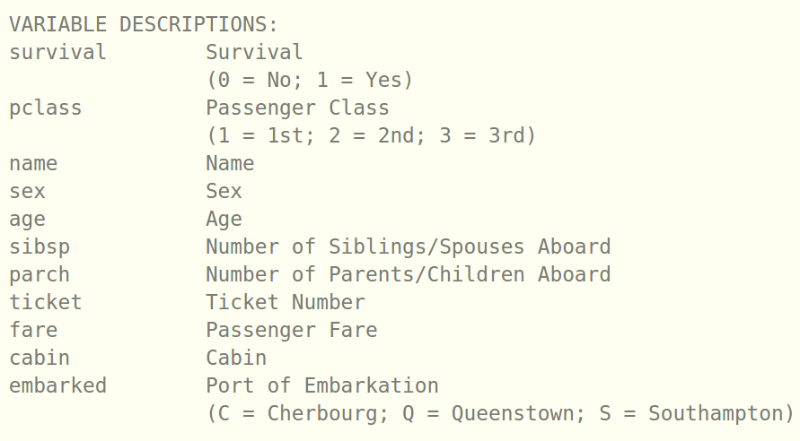


Note that our final format, (data, labels) is the format that ML models use for training and testing. The labels define the problem we have. Labels can be of many different types, including:

* Single column binary values (classification problem)
* Single column real values (regression problems, predictions using only one value)
* Multiple columns, binary values (classification problems but there are more than two classes)
* Multiple columns, real values ( multivariate regression problems)
* Multilabel (classification problems, but one sample can belong to multiple classes).

The key here is to identify the labels and the features. Care must be taken to identify and use only those features that have relevance to the model that you’re using. For example, if you want to predict the price of a house, and you have features such as lot size, location, size of yard, etc. and one of the features is the paint color of the house, then perhaps that last feature is not a useful feature to add to your training set. Also, it is important to keep a data log and identify the types of variables in your data set. Separate out the numeric variables as no extra text processing will need to be done on them.

For example, here we have the *Titanic*  data set. We can use this dataset to predict the chance of survivability of any given passenger after the sinking of the ship.



In this case, the label is obvious, ‘Survival’. Other variables have specific types, for example, Sex, and class are categorical variables (M/F, First/Second/Third Class). Name is also a variable, but as with the example of the house price and the paint color, it is unlikely that the value of the Name is predictive of survival.

Separate out the numerical variables first, we don’t need to do any processing on them and we can start applying techniques such as feature normalization on them.

With categorical variables we can do one of two things.

1. Convert them to labels.
2. Convert them to binary variables via one-hot encoding. Don’t forget to convert values to numbers first.

Take relevant text data and convert it into a count vector. I.e. a dictionary of mapping occurrences of the word to the word itself.

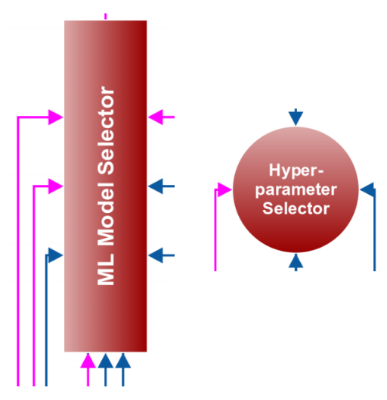
Pick the features that you want to use for the model. There are many techniques for doing this, including, for example the *chi-square* technique.

Stack the features into a single vector for use in the model as necessary.

Apply feature normalization to any numerical continuous variables as necessary. Note that feature normalization only work on dense features (i.e. features that have values for every training example) and not sparse features (i.e. features where most values are zero for the range of training examples).

A major problem with higher order dimensional data is that it isn’t always clear if the data is linearly correlated. Using techniques such as Principle Component Analysis (PCA) to decompose the data and plot it can tell you if a given set of features makes sense to put into an ML model.

Once we have what we think is a good data set, we can now try and optimize the hyperparameters for the model.



We also need to pick the ML algorithm that we want to try, we may try many different models, although research does seem to indicate that given multiple models of the same type, the differences often tend to be minimal.

Here are some examples of ML models

* Classification
  + Random Forest
  + Logistic Regression
  + Naïve Bayes
  + Support Vector Machines
  + K-Nearest Neighbors
  + Neural Networks (Convolution Neural Networks)
* Regression
  + Random Forest
  + Gradient Boosting
  + Linear Regression
  + Neural Networks

There are a number of tools and frameworks that will allow you to facilitate the processing of data and the ML models that you wish, including:

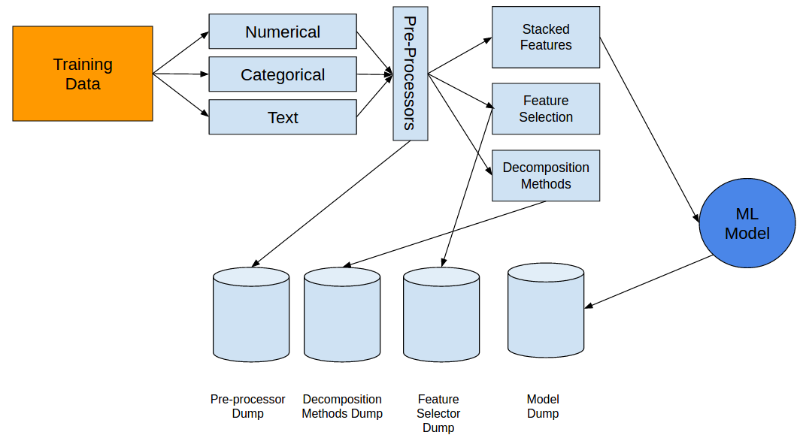
* Data operations: Pandas
* ML models: Scikit Learn
* Gradient boosting libraries: xgboost
* Neural Networks: TensorFlow, Keras
* Plotting of data: matplotlib
* Progress monitoring: tqdm
* Documentation: The Jupyter Notebook

Many (not all) of these libraries are available to you via the Anaconda python package. Be warned, however that it is over 400 MB in size!

For any kind of machine learning problem, we need to know how we’ll evaluate the results, or what the evaluation metric or objective is. For example, for linear regression we might use mean squared error to calculate our cost. For regression, we might use some sort of cross-entropy formula.

When creating data sets for training and testing, the proper way to do this with ML is to use a technique called *cross-validation*. The idea with cross-validation is that the ML model is trained on the training set, and then the results are tested on the cross-validation set first, before being run on the testing set. A common type of cross validation is *K-fold* cross validation, where the data is split up into chunks and the training and validation is done on each chunk in turn. This allows for proper training across the range of data and helps prevent overfitting.

To put everything we’ve talked about into a picture examine the following:



Then apply the outputs to our validation set.

