Goal
To explore the feasibility of an ANN-BPN in a stock trading scenario; to build a framework that allows for the quick testing of different scenarios and learning models.

Context
With the easy of availability of high frequency tick data for Stocks, commodities and cheap computing power, large scale simulations of algorithms, with and without learning, is becoming relatively trivial. One website, Quantopian.com tries to offer both, a rich web based IDE, that provides access to over ten years of market data, along with providing a rich API to simulate different trading scenarios.

Normalization
One of the important heuristics of making the neural network perform better relates to input normalization. Each input variable should be preprocessed so that its mean value, averaged over the entire training set, is close to zero, or else it is small compared to its standard deviation. The ranges of the indexes vary slightly, as their domain is totally different. In order to normalize each index range to [-1, 1], we are going to use the following simple formula: Index (x) = (Index (x) - Min(Index)) / (Max(Index) - Min(Index)). Thus each of the input variables will lie in the same range

ANN-BPN
One method of learning, presented below is an Artificial Neural Network with Back Propagation enabled. A simple implementation of an ANN-BPN is presented in the Appendix which can be invoked using regular python. A regression test asserts that this works for a simple XOR gate.

The simulation leverages this same ANN-BPN and feeds it the following:

- Stock A Opening Price
- Stock B Opening Price

The ANN then guesses the:

- Stock B Closing price, given past data (seeded initially only with ten days of data)

If the Closing price for Stock B is greater than the opening price by a certain threshold, i.e. the ANN thinks the stock is likely to go up, then a simulation trade is made in Stock B.

At end of day, the actual closing price for Stock B is recorded, and compared with the predicted closing price; this is then fed back into the ANN, which uses Back propagation to adjust its weights accordingly. And the simulation continues for the next day.

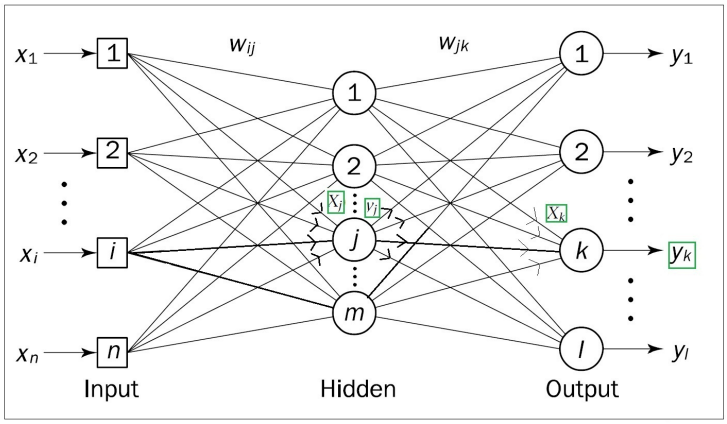
ANN BPN Diagram

Forward Stage, Input->Hidden->Output

- $X_j(p)$ is the net weighted input to neuron j $= sigmoid\left[\sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j\right]$
- $y_j(p)$ is the output of neuron j $= \frac{1}{1+e^{-X_j(p)}}$

- $X_k(p)$ is the net weighted input to neuron k $= sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k\right]$
- $y_k(p)$ is the output of neuron k at iteration p $= \frac{1}{1+\exp[-X_k(p)]}$.

- $y_{d,k}(p)$ is the desired output of neuron k at iteration p.



Backward Stage, Output->Hidden->Input

- $e_k(p) = y_{d,k}(p) - y_k(p)$

- $\Delta w_{jk}(p)$ is the weight correction in the output layer $= \alpha \times y_j(p) \times \delta_k(p)$
- $\delta_k(p)$ is the error gradient for neuron k in the output layer $= y_k(p) \times [1 - y_k(p)] \times e_k(p)$

- $e_j(p) = \sum_{k=1}^{l} \delta_k(p) w_{jk}(p)$

- $\Delta w_{ij}(p)$ is the weight correction for the hidden layer $= \alpha \times x_i(p) \times \delta_j(p)$
- $\delta_j(p)$ is the error gradient at neuron j in the hidden layer $= y_j(p) \times [1 - y_j(p)] \times e_j(p)$

Appendix

- $\delta_k(p)$ is the error gradient for neuron k in the output layer is determined as the derivative of the activation function multiplied by the error at the neuron output.

$$= \frac{\partial y_k(p)}{\partial X_k(p)} \times e_k(p)$$

$$= \frac{\partial\left\{\frac{1}{1+\exp[-X_k(p)]}\right\}}{\partial X_k(p)} \times e_k(p)$$

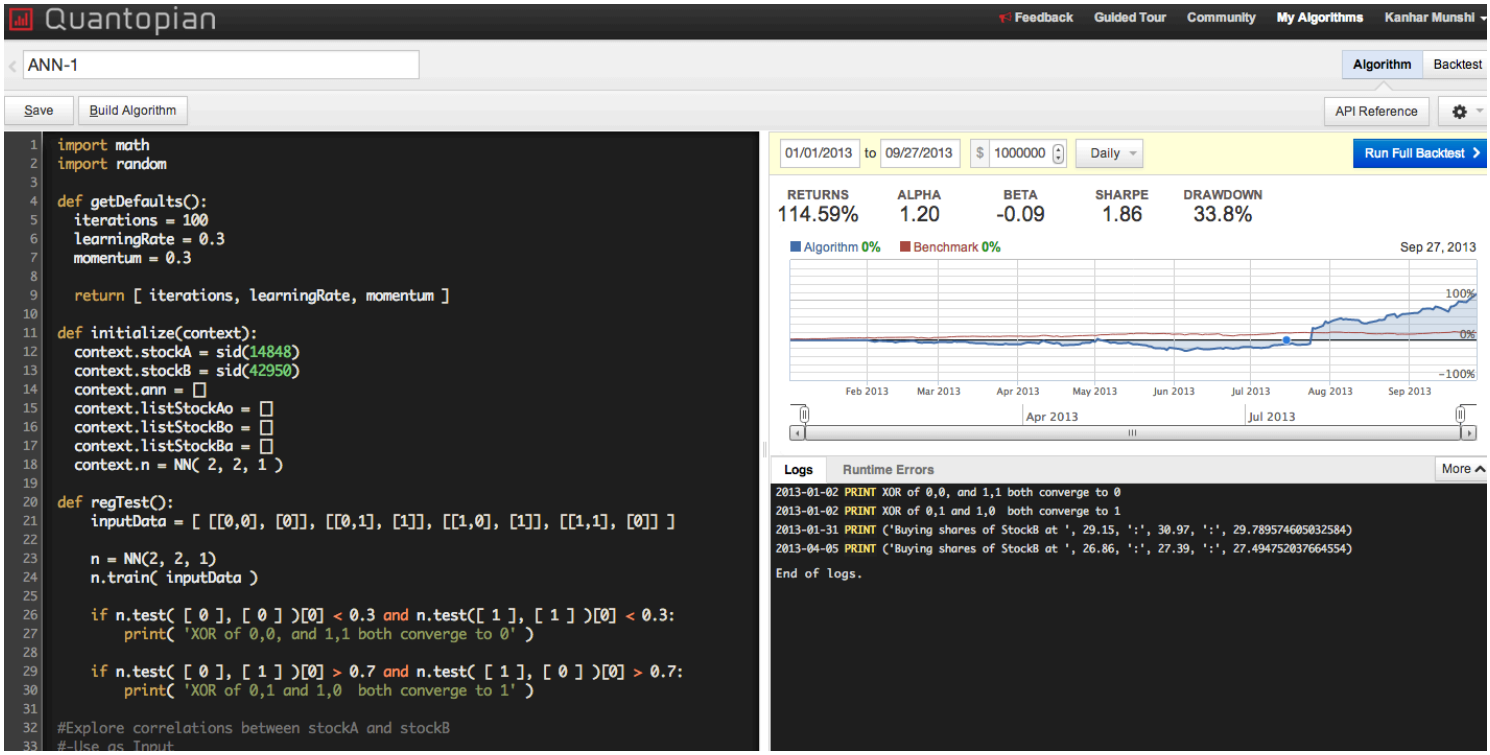
$$= \frac{\exp[-X_k(p)]}{\{1+\exp[-X_k(p)]\}^2} \times e_k(p) \qquad \left[1 + \exp[-X_k(p)] = \frac{1}{y_k(p)}\right]$$

$$= \frac{\left\{\frac{1}{y_k(p)} - 1\right\}}{\left\{\frac{1}{y_k(p)}\right\}^2} \times e_k(p)$$

$$= y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

Figure 1: ANN BPN Diagram

Result
Stock A: YHOO
Stock B: FB
Date Range: 1/1/2013 till 9/27/2013



Appendix: Code Listing

Invocation Code – Regression Test

```
import math
import random

def getDefaults():
  iterations = 100
  learningRate = 0.3
  momentum = 0.3

  return [ iterations, learningRate, momentum ]

def initialize(context):
  context.stockA = sid(14848)
  context.stockB = sid(42950)
  context.ann = []
  context.listStockAo = []
  context.listStockBo = []
  context.listStockBa = []
  context.n = NN( 2, 2, 1 )

def regTest():
```

```
    inputData = [ [[0,0], [0]], [[0,1], [1]], [[1,0], [1]], [[1,1], [0]] ]

    n = NN(2, 2, 1)
    n.train( inputData )

    if n.test( [ 0 ], [ 0 ] )[0] < 0.3 and n.test([ 1 ], [ 1 ] )[0] < 0.3:
        print( 'XOR of 0,0, and 1,1 both converge to 0' )

    if n.test( [ 0 ], [ 1 ] )[0] > 0.7 and n.test( [ 1 ], [ 0 ] )[0] > 0.7:
        print( 'XOR of 0,1 and 1,0  both converge to 1' )

#Explore correlations between stockA and stockB
#-Use as Input
#    StockA open Price
#    StockB open Price
#-Predict Stock C close price
#-If Price expected to go up, buy (and sell the next day)
#-Record error in prediction
#-Try again next day.
def handle_data(context, data):

  context.listStockAo.append( data[context.stockA].open_price )
  context.listStockBo.append( data[context.stockB].open_price  )
  context.listStockBa.append( data[context.stockB].close_price )

  Ao = context.listStockAo
  Bo = context.listStockBo
  Bc = context.listStockBa

  #Run regtest to check if ANN is working fine
  if( len( Ao ) == 1 ):
     regTest()

  #Start training
  if len( Ao ) > 3:
     context.n.train( merge( norm( Ao ) , norm( Bo ), norm( Bc ) ) )

  #Only make stock predictions if Greater than 10 Data points
  if len( Ao ) > 10:

     #Given past training, make a prediction, Normalize First, then Denormalize
     pBc = denorm( Bc, context.n.test( [norm( Ao )[-1] ], [norm( Bo )[-1]] ) )
     #print( in1[-1], ":", in2[-1], ":", out[-1], "->", out2[-1] )

     if pBc[-1] - Bo[-1] > 0.6 :
        order(context.stockB, 25000 )
        print( 'Buying shares of StockB at ', Bo[-1], ":", Bc[-1], ":", pBc[-1] )
```

ANN with BPN Class

```
class NN:
    def __init__(self, sizeInput, sizeHidden, sizeOutput):
        self.sizeInput  = sizeInput + 1 # +1 for bias node
        self.sizeHidden = sizeHidden
        self.sizeOutput = sizeOutput

        # Seed Nodes
        self.nodeInput  = [1.0]*self.sizeInput
        self.nodeHidden = [1.0]*self.sizeHidden
        self.nodeOutput = [1.0]*self.sizeOutput

        # Seed Weights
        self.weightInput  = makeMatrix(self.sizeInput, self.sizeHidden, Random=1)
        self.weightOutput = makeMatrix(self.sizeHidden, self.sizeOutput, Random=1)

        # Seed last change in weights for momentum
        self.changeInWeightInput  = makeMatrix(self.sizeInput, self.sizeHidden, Random=0)
        self.changeInWeightOutput = makeMatrix(self.sizeHidden, self.sizeOutput, Random=0)

    #Iterate through data X times
    #For each iteration
    #   Get output using current weights
    #   Compare output with expected output
    #   Get error and backpropagate, update weights
    #End For
    def train(self, inputData ):
        [ iterations, learningRate, momentum ] = getDefaults()
        for i in range(iterations):
            error = 0.0
            for p in inputData:
                inputs  = p[0]
                outputs = p[1]
                self.nodeOutput = self.getOutput(inputs)
                error = error + self.backPropagate(outputs, learningRate, momentum)
                #print('error %-.3f' % error)

    #Set Input  Nodes
    #Set Hidden Nodes
    #Set Output Nodes and Return
    def getOutput(self, inputs):

        # Copy inputs
        for k in range(self.sizeInput-1):
            self.nodeInput[k] = inputs[k]

        # Calculate hidden weights
        for j in range(self.sizeHidden):
            sum = 0.0
            for i in range(self.sizeInput):
                sum = sum + self.nodeInput[i] * self.weightInput[i][j]
            self.nodeHidden[j] = sigmoid(sum)
```

```
    # Calculate Output
    for k in range(self.sizeOutput):
        sum = 0.0
        for j in range(self.sizeHidden):
            sum = sum + self.nodeHidden[j] * self.weightOutput[j][k]
        self.nodeOutput[k] = sigmoid(sum)

    return self.nodeOutput


def backPropagate(self, outputs, N, M):
    if len(outputs) != self.sizeOutput:
        raise ValueError('wrong number of target values')

    # calculate error terms for output
    output_deltas = [0.0] * self.sizeOutput
    for k in range(self.sizeOutput):
        error = outputs[k]-self.nodeOutput[k]
        output_deltas[k] = dsigmoid(self.nodeOutput[k]) * error

    # calculate error terms for hidden
    hidden_deltas = [0.0] * self.sizeHidden
    for j in range(self.sizeHidden):
        error = 0.0
        for k in range(self.sizeOutput):
            error = error + output_deltas[k]*self.weightOutput[j][k]
        hidden_deltas[j] = dsigmoid(self.nodeHidden[j]) * error

    # update output weights
    for j in range(self.sizeHidden):
        for k in range(self.sizeOutput):
            change = output_deltas[k]*self.nodeHidden[j]
            self.weightOutput[j][k] = self.weightOutput[j][k] + N*change + M*self.changeInWeightOutput[j][k]
            self.changeInWeightOutput[j][k] = change

    # update input weights
    for i in range(self.sizeInput):
        for j in range(self.sizeHidden):
            change = hidden_deltas[j]*self.nodeInput[i]
            self.weightInput[i][j] = self.weightInput[i][j] + N*change + M*self.changeInWeightInput[i][j]
            self.changeInWeightInput[i][j] = change

    # calculate error
    error = 0.0
    for k in range(len(outputs)):
        error = error + 0.5*(outputs[k]-self.nodeOutput[k])**2
    return error


def test(self, in1, in2 ):
    res = []
    for a,b in zip( in1, in2 ):
```

```
            out = self.getOutput( [ a, b ] )
            res.append( out[0] )
        return res

 #if price < vwap * 0.995 and notional > context.min_notional:
  #  order(context.aapl,-100)
  #elif price > vwap * 1.005 and notional < context.max_notional:
  #  order(context.aapl,+100)

def trim( num ):
    return int(  num * 10 ) / 10.0

def norm( arr ):
    res = []
    minA = min( arr )
    maxA = max( arr )
    for i in arr:
        res.append( trim( (i - minA ) / ( maxA - minA ) ) )
    return res

def denorm( orig, A ):
    res = []
    minA = min( orig )
    maxA = max( orig )
    for i in A:
        res.append( ( maxA - minA ) * i  + minA )
    return res

def printArr( arr ):
    for a in arr:
        print( a )

def merge( in1, in2, out ):
    res = []
    for a,b,c in zip( in1 , in2 , out ) :
        res.append( [ [a,b ], [ c ] ] )
    return res

def unmerge( arr ):
    in1 = []
    in2 = []
    out = []

    for a in arr :
        in1.append( a[0][0] )
        in2.append( a[0][1] )
        out.append( a[1][0] )

    return [in1, in2, out]

# Get random number x where a <= x < b
def rand(a, b):
```

```
    return (b-a)*random.random() + a

# Get Matrix[I,J]=
def makeMatrix(I, J, Random=0):
    m = []
    fill = 0

    for i in range(I):
        m.append([fill]*J)

    if Random==1:
      for i in range(I):
        for j in range(J):
          m[i][j] = rand(-0.2, 0.2)

    return m

# Get sigmoid(x)
def sigmoid(x):
    return math.tanh(x)

# Get Derivative of sigmoid function, in terms of the output (i.e. y)
def dsigmoid(y):
    return 1.0 - y**2

#amount = context.portfolio['positions'][context.stockB].amount
#record( openingPrice=Open, RealClosingPrice=RealClose, predictedClosingPrice=PredictedClose)
```