Sample pseudo-code for coursework 1

H Batatia

Activation functions

```
#abstract activation class
# provide evaluate and derivative methods
class Activation:
        def evaluate(x):
                pass
        def derivate(x):
                pass
#sigmoid activation – sub class of activation
class Sigmoid (Activation):
        def evaluate(x):
                return 1 / (1 + math.exp(-x))
        def derivative(x):
                f = 1/(1+exp(-x))
                return f * (1 - f)
```

```
#tanh activation – sub class of activation
class tanh(Activation):
        …

#other activation – sub class of activation
…
```

Loss functions

```
#abstract Loss class
# provides evaluate and derivative methods
class Loss:
        def evaluate(x):
                pass
        def derivate(x):
                pass
#MSE Loss – subclass of Loss
class Mse (Loss):
        def evaluate(y,t)):
                return 2*(t-y)**2
        def derivative(y,t):
                return t-y
```

```
#Binary cross entropy  Loss – subclass of Loss
class Binary_cross_entropy (Loss):
        def evaluate(y,t)):
                y_pred = np.clip(y, 1e-7, 1 - 1e-7)
                term0 = (1-t) * np.log(1-y + 1e-7)
                term1 = t * np.log(y + 1e-7)
                return – (term0 + term1)
        def derivative(y,t):
                return t/y + (1-t) /(1-y)
#Hinge Loss – subclass of Loss
class Hinge (Loss):
        def evaluate(y,t)):
                return max(0, 1-t*y)
        def derivative(y,t):
                return …
```

Layer: forward and backpropagation

```
#Layer class providing forward and backpropagate methods
class Layer:
        def __init__(self, nodes, activation):
                #declare attributes: nb_nodes, X_in, W, B, activation
        def  forward(in):
                self.X_in = in
                out = activation.evaluate(W * in + B)
                return out
        def backpropagate(delta, rate): #delta is the error backpropagated from the next layer
                dz = activation.derivative(W*X_in) * delta
                dw = X_in * dz
                db = dz
                delta = W*dz
                #update the weights after calculating the error to backpropagate
                W -= rate * dw
                B -= rate * db
                return delta # return the error to be backpropagated
```

Network: forward and backpropagation

```
#Network class encapsulates the list of layers and provides forward and backpropagate methods
class Network:
        def __init__(self): #initialise the empty list of layers
                self.layers = []
        def append(layer): #to append a layer to the network
                self.layers.append(layer)
        def forward (data_in):
                out = data_in
                for layer in self.layers:
                        out = layer.forward(out)
                return out
        def packpropagate(delta, rate): #delta  initially holds the derivative of the loss
                for layer in self.layers.reverse():
                        delta = layer.backpropagate(delta, rate)
```

Network builder

#Create a network using the parameters provides by the user

```
Class ANNBuilder:
    def build(nb_layers, list_nb_nodes, list_functions):
        ann = Network()
        for i in range(nb_layers):
            layer = Layer(list_nb_nodes[i], list_function[i])
            ann.append(layer)
```

Gradient descent

```
#Base gradient descent that iterates on a batch of data and then backpropagate the error
def base_gd(ann, data, classes, rate, loss):
    for x in data: #considering data as a list
        y = ann. forward(x)
        t = getTrue(classes, x) #simply retrieve the class corresponding to sample x
        L += loss.evaluate(y, t) #cumulate the loss
        dL += loss.dervative(y, t) #cumulate the error
        accuracy += 1 if y==t else 0 #count the good classifications
    L /= len(data) # take the average loss
    dL /= len(data) # take the average error
    accuracy /= len(data) #calculate the percent accuracy
    ann.backpropagate(dL, rate) #backpropagate the error and update the weights
    # adapt the rate here if needed
    return L, accuracy
```

GD variants

```
def mini_batch(ann, data, classes, epochs, rate, loss, batch_size):
    loss, accu = gd(ann, data, classes, epochs, rate, loss, batch_size)
    return loss, accu

def dgd(ann, data, classes, epochs, rate, loss): # batch size = N
    loss, accu = gd(ann, data, classes, epochs, rate, loss, data.size)
    return loss, accu

def sgd(ann, data, classes, epochs, rate, loss): # batch size = 1
    loss, accu = gd(ann, data, classes, epochs, rate, loss, 1)
    return loss, accu
```

```
def gd(ann, data, classes, epochs, rate, loss, batch_size):
    L = 0
    accuracy=0
    #partition the dataset into batches
    batches = createBatches(data, classes, batch_size)
    #iterate on the epochs
    for epoch in range(epochs):
        #batch assumed to have data and classes attributes
        for batch in batches:
            lo, accu  = base_gd(ann, batch.data, batch.classes, rate, loss)
        #store loss L and accuracy in lists for later plotting
        L +=lo
        accuracy += accu
    return
```

Main

```
#read and prepare your data x, y
…

#read ANN params from user: layers, nodes, functions
ann = ANNBuilder.build(layers, nodes, functions)

# read hyper-parameters: epochs, rate, batch_size, loss
# run experiment
loss, accuracy = mini_batch(ann, data, classes, epochs, rate, loss, batch_size)
…
# plot, display results
```