```python
import numpy as np
import matplotlib.pyplot as plt

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear activation
    # a is a list of of the activations of each node in the previous layer going
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO ###################################################################################
            # Compute the gradients for layer.W and layer.b using the gradient for the output of the
            # layer curA and the gradient of the output curGrad
            # Use the gradients to update the weight and the bias for the layer
            # ########################################################################################
            # grad_W = curGrad.dot(curA.T)
            # layer.W -= self.eta * grad_W
            # grad_b = curGrad.dot( np.ones( (curGrad.shape[1], 1) ) )
            # layer.b -= self.eta * grad_b
            curZ = layer.z(curA)
            grad_W = (curGrad * layer.activation.dx(curZ)).dot(curA.T)
            layer.W -= self.eta * grad_W / m
            grad_b = curGrad * layer.activation.dx(curZ)
            layer.b = layer.b - self.eta * grad_b / m


# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO ###################################################################################
        # Implement me
        # ########################################################################################
        return (y - yp) * (y - yp) / 2

```

```python
    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return y - yp

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def dx(z):
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))

# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return np.tanh(z)

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return 1 - np.tanh(z) ** 2

# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
```

```
        #
        ###############################################################################
        ########
        return np.where(z < 0, 0, z)

    @staticmethod
    def dx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return np.where(z <= 0, 0, 1)

# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return z

    @staticmethod
    def dx(z):
        # TODO
        ###############################################################################
        ###
        # Implement me
        #
        ###############################################################################
        ########
        return np.ones(z.shape)

# This class represents a single hidden or output layer in the neural network
class DenseLayer(object):

    # numNodes: number of hidden units in the layer
    # activation: the activation function to use in this layer
    def __init__(self, numNodes, activation):
        self.numNodes = numNodes
        self.activation = activation

    def getNumNodes(self):
        return self.numNodes

    # Initialize the weight matrix of this layer based on the size of the matrix W
    def initialize(self, fanIn, scale=1.0):
        s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
        self.W = np.random.normal(0, s,
                                  (self.numNodes,fanIn))
```

```
            self.b = np.random.uniform(-1,1,(self.numNodes,1))

    # Apply the activation function of the layer on the input z
    def a(self, z):
        return self.activation.fx(z)

    # Compute the linear part of the layer
    # The input a is an n x k matrix where n is the number of samples
    # and k is the dimension of the previous layer (or the input to the network)
    def z(self, a):
        return self.W.dot(a) + self.b # Note, this is implemented where we assume a is k x n

    # Compute the derivative of the layer's activation function with respect to z
    # where z is the output of the above function.
    # This derivative does not contain the derivative of the matrix multiplication
    # in the layer.  That part is computed below in the model class.
    def dx(self, z):
        return self.activation.dx(z)

    # Update the weights of the layer by adding dW to the weights
    def updateWeights(self, dW):
        self.W = self.W + dW

    # Update the bias of the layer by adding db to the bias
    def updateBias(self, db):
        self.b = self.b + db

# This class handles stacking layers together to form the completed neural network
class Model(object):

    # inputSize: the dimension of the inputs that go into the network
    def __init__(self, inputSize):
        self.layers = []
        self.inputSize = inputSize

    # Add a layer to the end of the network
    def addLayer(self, layer):
        self.layers.append(layer)

    # Get the output size of the layer at the given index
    def getLayerSize(self, index):
        if index >= len(self.layers):
            return self.layers[-1].getNumNodes()
        elif index < 0:
            return self.inputSize
        else:
            return self.layers[index].getNumNodes()

    # Initialize the weights of all of the layers in the network and set the cost
    # function to use for optimization
    def initialize(self, cost, initializeLayers=True):
        self.cost = cost
        if initializeLayers:
            for i in range(0,len(self.layers)):
                if i == len(self.layers) - 1:
                    self.layers[i].initialize(self.getLayerSize(i-1))
```

```
                else:
                    self.layers[i].initialize(self.getLayerSize(i-1))

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
    # This function returns
    # yp - the output of the network
    # a - a list of inputs for each layer of the newtork where
    #     a[i] is the input to layer i
    # z - a list of values for each layer after evaluating layer.z(a) but
    #     before evaluating the nonlinear function for the layer
    def evaluate(self, x):
        curA = x.T
        a = [curA]
        z = []
        for layer in self.layers:
            # TODO ##################################################################################
            # Store the input to each layer in the list a
            # Store the result of each layer before applying the nonlinear function in z
            # Set yp equal to the output of the network
            # ########################################################################################
            curZ = layer.z(curA)
            z.append(curZ)
            curA = layer.a(curZ)
            a.append(curA)

        yp = a.pop(-1)

        return yp, a, z

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
    def predict(self, a):
        a,_,_ = self.evaluate(a)
        return a.T

    # Train the network given the inputs x and the corresponding observations y
    # The network should be trained for numEpochs iterations using the supplied
    # optimizer
    def train(self, x, y, numEpochs, optimizer):

        # Initialize some stuff
        n = x.shape[0]

        x = x.copy()
        y = y.copy()
        hist = []
        optimizer.initialize(self.layers)

        # Run for the specified number of epochs
```

```
        for epoch in range(0,numEpochs):

            # Feed forward
            # Save the output of each layer in the list a
            # After the network has been evaluated, a should contain the
            # input x and the output of each layer except for the last layer
            yp, a, z = self.evaluate(x)

            # Compute the error
            # C = self.cost.fx(yp,y.T)
            # d = self.cost.dx(yp,y.T)
            C = self.cost.fx(y.T,yp)
            d = self.cost.dx(y.T,yp)
            grad = []

            # Backpropogate the error
            idx = len(self.layers)
            for layer, curZ in zip(reversed(self.layers),reversed(z)):
                # TODO ###############################################################################
                # Compute the gradient of the output of each layer with respect to the error
                # grad[i] should correspond with the gradient of the output of layer i
                # ####################################################################################
                # if len(grad) == 0:
                #   grad_i = d * layer.activation.dx(curZ)
                #   grad.append(grad_i)
                # else:
                #   grad_i = ( grad[-1].T.dot(last_layer_W) ).T * layer.activation.dx(curZ)
                #   grad.append(grad_i)
                # last_layer_W = layer.W
                if len(grad) == 0:
                    grad_i = layer.W.T.dot( d * layer.activation.dx(curZ) )
                    grad.append(grad_i)
                else:
                    grad_i = layer.W.T.dot( grad[-1] * layer.activation.dx(curZ) )
                    grad.append(grad_i)
            grad.pop(-1)
            grad.reverse()
            grad.append(d)

            # Update the errors
            optimizer.update(self.layers, grad, a)

            # Compute the error at the end of the epoch
            yh = self.predict(x)
            C = self.cost.fx(yh,y)
            C = np.mean(C)
            hist.append(C)
        return hist

if __name__ == '__main__':

```

```
    # Generate the training set
    np.random.seed(9001)
    x=np.random.uniform(-np.pi,np.pi,(1000,1))
    y=np.sin(x)

    activations = dict(ReLU=ReLUActivation,
                       tanh=TanhActivation,
                       linear=LinearActivation)
    lr = dict(ReLU=0.02,tanh=0.02,linear=0.005)

    for key in activations:

        # Build the model
        activation = activations[key]
        model = Model(x.shape[1])
        model.addLayer(DenseLayer(100,activation()))
        model.addLayer(DenseLayer(100,activation()))
        model.addLayer(DenseLayer(1,LinearActivation()))
        model.initialize(QuadraticCost())

        # Train the model and display the results
        hist = model.train(x,y,500,GDOptimizer(eta=lr[key]))
        yHat = model.predict(x)
        error = np.mean(np.square(yHat - y))/2
        print(key+' MSE: '+str(error))
        plt.plot(hist)
        plt.title(key+' Learning curve')
        plt.show()

        # TODO #################################################################################
        # Plot the approximation of the sin function from all of the models
        # ######################################################################################

```