CS 189: Introduction to Machine Learning

Fall 2017

•

Homework 7

•

Solutions by

Jinhong Du

3033483677

Question 1

(a)

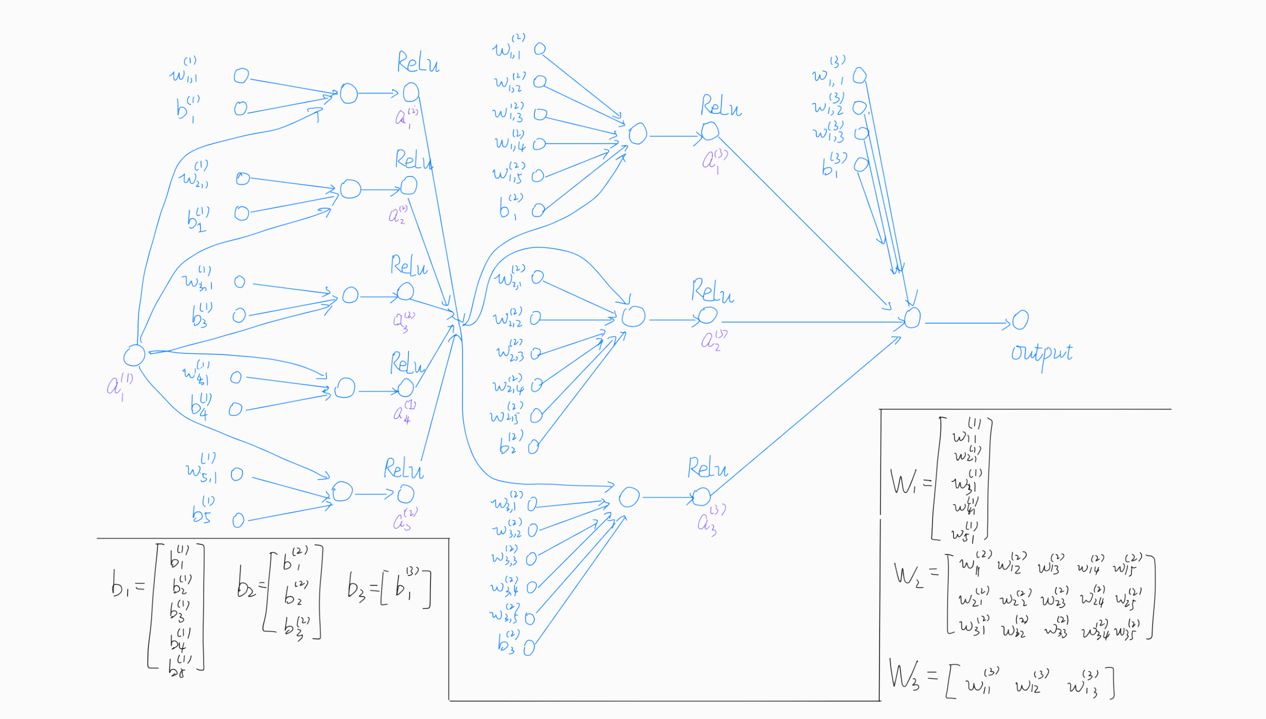
Jinhong Du
jaydu@berkeley.edu

(b)

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up. Jinhong Du

Question 2

(a)



(b)

$\because$

$$MSE(\hat{\vec{y}}) = \frac{1}{2}\sum_{i=1}^{n} \|y_i - \hat{y}_i\|_2^2$$

$\therefore$

$$\begin{aligned}\nabla_{\hat{y}_i} MSE &= \frac{1}{2}\sum_{j=1}^{n} \nabla_{\hat{y}_i}(y_j^2 + \hat{y}_j^2 - 2y_j\hat{y}_j) \\ &= \frac{1}{2}\nabla_{\hat{y}_i}(y_i^2 + \hat{y}_i^2 - 2y_i\hat{y}_i) \\ &= \hat{y}_i - y_i\end{aligned}$$

$\therefore$

$$\nabla_{\hat{\vec{y}}} MSE = \begin{pmatrix} \hat{y}_1 - y_1 \\ \hat{y}_2 - y_2 \\ \vdots \\ \hat{y}_n - y_n \end{pmatrix}$$

```
class QuadraticCost(object):
@staticmethod
def fx(y,yp):
    return np.square(y - yp)/2
```

Solution (cont.)

```

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

(c)

$$\sigma_{linear}(z) = z$$
$$\nabla_z \sigma_{linear}(z) = 1$$
$$\sigma_{ReLU}(z) = \begin{cases} 0 & z < 0 \\ z & otherwise \end{cases}$$
$$\nabla_z \sigma_{ReLU}(z) = \begin{cases} 0 & z < 0 \\ 1 & otherwise \end{cases}$$
$$\sigma_{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\nabla_z \sigma_{tanh}(z) = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$
$$= 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2$$

```
# 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):
        return np.tanh(z)

```

Solution (cont.)

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

# Rectified linear unit
class ReLUActivation(object):
        @staticmethod
        def fx(z):
            return np.maximum(z,0)

        @staticmethod
        def dx(z):
            x =np.zeros_like(z)
            x[z>=0] =1
            return x

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

        @staticmethod
        def dx(z):
            return np.ones_like(z)
```

(d)

```
def evaluate(self, x):
    curA = x.T
    a = [curA]
    z = []
    for layer in self.layers:
        z.append(layer.z(a[-1]))
        a.append(layer.a(z[-1]))
    yp = a[-1]
    a = a[:-1]
    return yp, a, z
```

(e)

$$\begin{aligned}
\vec{a}_{i+1} &= \sigma_i(\vec{z}_i) \\
\vec{z}_i &= W_i\vec{a}_i + \vec{b}_i \\
\frac{\partial \vec{a}_{i+1}}{\partial \vec{z}_i} &= \sigma_i'(\vec{z}_i) \\
\frac{\partial \vec{z}_i}{\partial \vec{a}_i} &= W_i \\
\frac{\partial MSE}{\partial \vec{a}_i} &= \frac{\partial MSE}{\partial \vec{a}_{i+1}} .* \frac{\partial \vec{a}_{i+1}}{\partial \vec{z}_i} \cdot \frac{\partial \vec{z}_{i+1}}{\partial \vec{a}_i} \\
&= \frac{\partial MSE}{\partial \vec{a}_{i+1}} .* \sigma_i'(\vec{z}_i) \cdot W_i
\end{aligned}$$

where $.*$ is element-wise multiplication and $\cdot$ is matrix multiplication.

Here,

$$\frac{\partial MSE}{\partial \vec{a}_3} \in \mathbb{R}^{1000\times 1}$$

$$\frac{\partial \vec{a}_3}{\partial \vec{z}_3} = \begin{bmatrix} \sigma'(\vec{z}_3(1))) \\ \sigma'(\vec{z}_3(2))) \\ \vdots \\ \sigma'(\vec{z}_3(1000))) \end{bmatrix} \in \mathbb{R}^{1000\times 1}$$

$$\frac{\partial \vec{z}_3}{\partial \vec{a}_2} = W_3 \in \mathbb{R}^{100\times 1}$$

$$\frac{\partial MSE}{\partial \vec{a}_2} = \frac{\partial MSE}{\partial \vec{a}_3} .* \frac{\partial \vec{a}_3}{\partial \vec{z}_3} \cdot \frac{\partial \vec{z}_3}{\partial \vec{a}_2}$$

$$\vdots$$

```python
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):

        yp, a, z = self.evaluate(x)

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

Solution (cont.)

```python
        w = np.ones((len(d),len(d)))
        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers),reversed(z)):
            if len(grad)==0:
                grad.insert(0,(layer.dx(curZ)*d).T)
            else:
                grad.insert(0,d.dot(w)*layer.dx(curZ.T))
            w = layer.W
            d = grad[0]

        # 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
```

(f)

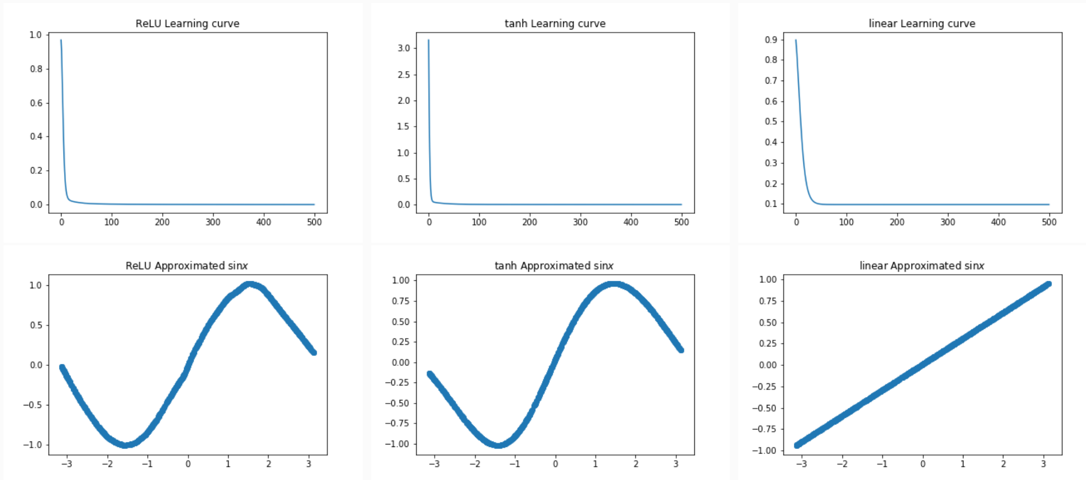
$$\frac{\partial MSE}{\partial \vec{W}_i} = \left(\frac{\partial MSE}{\partial \vec{z}_i}\right)^T \cdot \left(\frac{\partial \vec{z}_i}{\partial \vec{a}_{i-1}}\right)^T$$

```python
def update(self, layers, g, a):
  for layer, curGrad, curA in zip(layers, g, a):
    n = len(curA.T)
    layer.updateWeights(self.eta *
       curGrad.T.dot(curA.T) / n)
    layer.updateBias(self.eta* np.mean(
      curGrad.T,axis=1,keepdims=True))
```

(g)

ReLU MSE: 0.000408666485428
tanh MSE: 0.00126499054514
linear MSE: 0.0967346963549

Solution (cont.)




```python
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):
        yp, a, z = self.evaluate(x)

        # Compute the error
        C = self.cost.fx(yp,y.T)
        d = self.cost.dx(yp,y.T)
        d = np.mean(d,keepdims=True)
        grad = []
        w = np.ones((1,1))
        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers),reversed(z)):
            if len(grad)==0:
                grad.insert(0,(d*w*((np.mean(layer.dx(curZ),
                    axis=1,keepdims=True)))))
            else:
                grad.insert(0,((w.T).dot(d)*((np.mean(
                    layer.dx(curZ),axis=1,keepdims=True)))))
            w = layer.W
            d = grad[0]

```

Solution (cont.)

```python
            # 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
```

(h)

```
MSE for ReLU with 1 hidden layer(s) and 5 hidden nodes per layer: 0.0234909362319
MSE for ReLU with 2 hidden layer(s) and 5 hidden nodes per layer: 0.157080596055
MSE for ReLU with 3 hidden layer(s) and 5 hidden nodes per layer: 0.00620014687172
MSE for ReLU with 1 hidden layer(s) and 10 hidden nodes per layer: 0.0439981734314
MSE for ReLU with 2 hidden layer(s) and 10 hidden nodes per layer: 0.0375509396886
MSE for ReLU with 3 hidden layer(s) and 10 hidden nodes per layer: 0.0195351366292
MSE for ReLU with 1 hidden layer(s) and 25 hidden nodes per layer: 0.0101003292316
MSE for ReLU with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00919474762221
MSE for ReLU with 3 hidden layer(s) and 25 hidden nodes per layer: 0.00179816199785
MSE for ReLU with 1 hidden layer(s) and 50 hidden nodes per layer: 0.0164250943699
MSE for ReLU with 2 hidden layer(s) and 50 hidden nodes per layer: 0.00423117129644
MSE for ReLU with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00135987758048
MSE for tanh with 1 hidden layer(s) and 5 hidden nodes per layer: 0.0516499405974
MSE for tanh with 2 hidden layer(s) and 5 hidden nodes per layer: 0.0309338075028
MSE for tanh with 3 hidden layer(s) and 5 hidden nodes per layer: 0.0387015231705
MSE for tanh with 1 hidden layer(s) and 10 hidden nodes per layer: 0.0261680724503
MSE for tanh with 2 hidden layer(s) and 10 hidden nodes per layer: 0.0120426949507
MSE for tanh with 3 hidden layer(s) and 10 hidden nodes per layer: 0.0207633717414
MSE for tanh with 1 hidden layer(s) and 25 hidden nodes per layer: 0.0160555156441
MSE for tanh with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00674050797273
MSE for tanh with 3 hidden layer(s) and 25 hidden nodes per layer: 0.0040228468676
MSE for tanh with 1 hidden layer(s) and 50 hidden nodes per layer: 0.0138829301916
MSE for tanh with 2 hidden layer(s) and 50 hidden nodes per layer: 0.00244803141724
MSE for tanh with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00044184469178
```

(i)

We can see that:

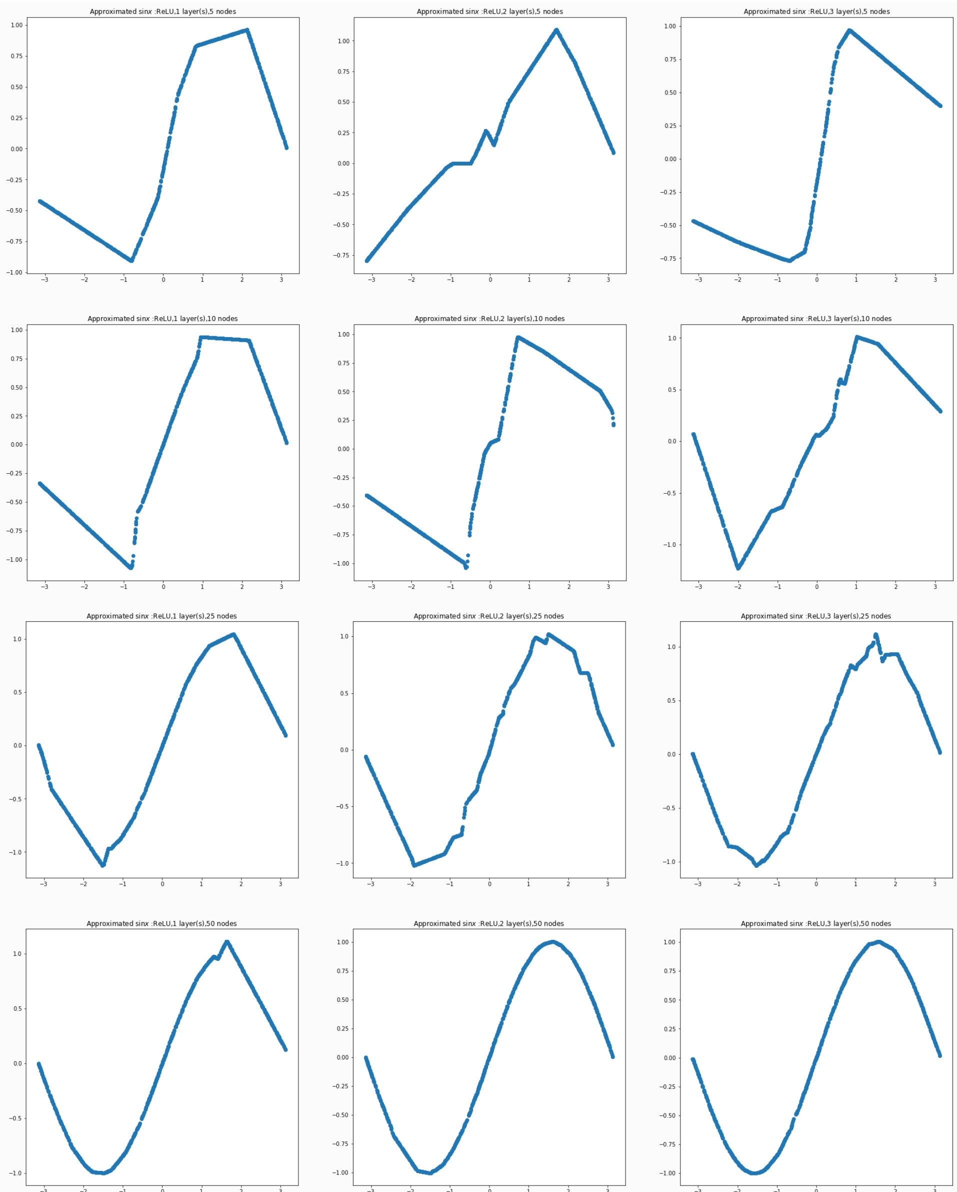
Increasing the number of nodes of each hidden layers will decrease the MSE.

tanh performs better than ReLu.

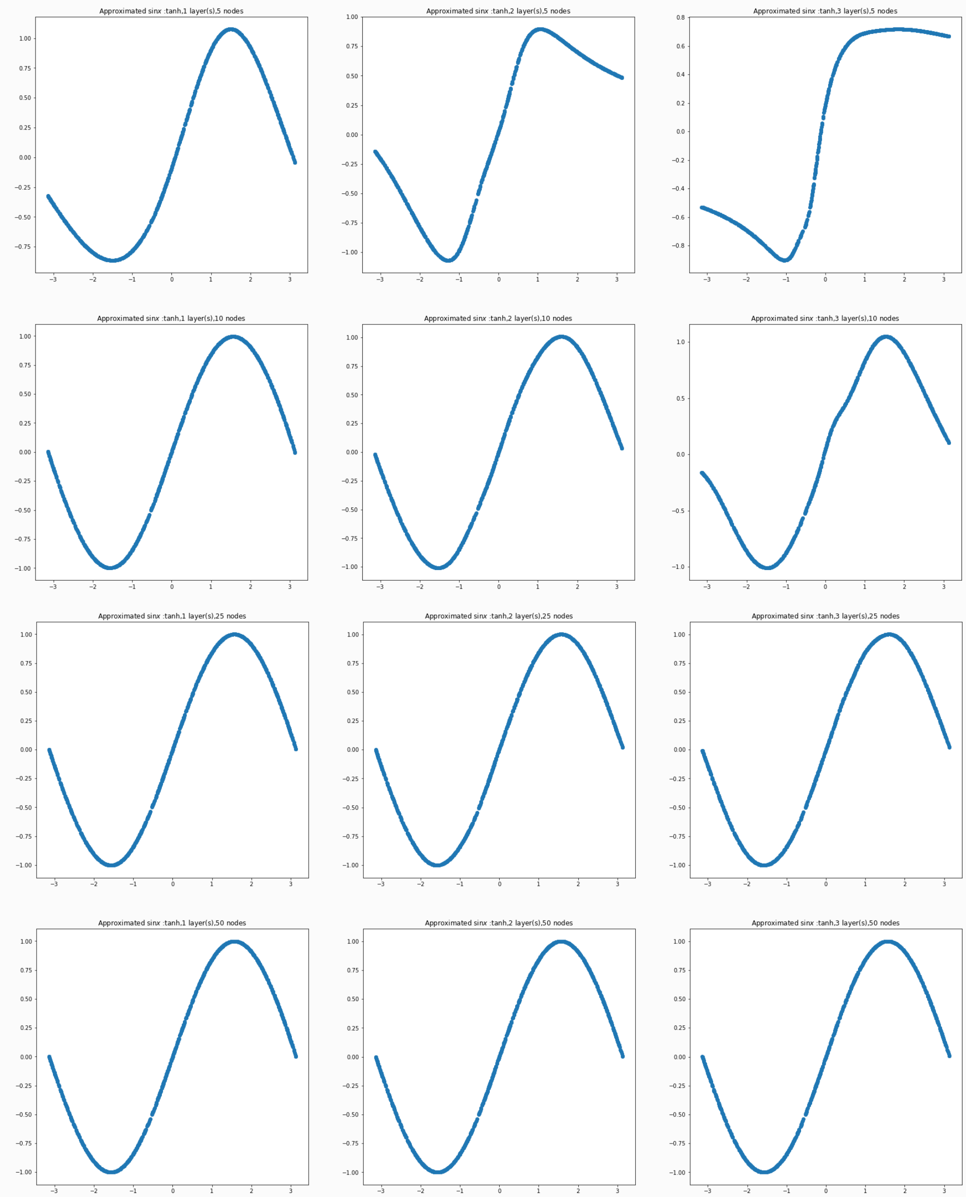
For ReLu, increasing the number of hidden layers will decrease MSE; For tanh, increasing the number of hidden layers will increase MSE.

Solution (cont.)

```
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 5 hidden nodes per layer: 11.6666057636
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 5 hidden nodes per layer: 85.5131245524
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 5 hidden nodes per layer: 25.4156105084
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 10 hidden nodes per layer: 6.61275585732
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 10 hidden nodes per layer: 16.4723774847
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 10 hidden nodes per layer: 8.75137056198
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 25 hidden nodes per layer: 0.48518834232
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 25 hidden nodes per layer: 0.469035013708
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 25 hidden nodes per layer: 0.456402450694
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 50 hidden nodes per layer: 0.431777427145
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 50 hidden nodes per layer: 0.010011733539
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00557636785786
Shortcut Method MSE for tanh with 1 hidden layer(s) and 5 hidden nodes per layer: 4.37525255244
Shortcut Method MSE for tanh with 2 hidden layer(s) and 5 hidden nodes per layer: 9.32830455191
Shortcut Method MSE for tanh with 3 hidden layer(s) and 5 hidden nodes per layer: 24.6123101195
Shortcut Method MSE for tanh with 1 hidden layer(s) and 10 hidden nodes per layer: 0.00401742362083
Shortcut Method MSE for tanh with 2 hidden layer(s) and 10 hidden nodes per layer: 0.0343273343733
Shortcut Method MSE for tanh with 3 hidden layer(s) and 10 hidden nodes per layer: 0.974263874259
Shortcut Method MSE for tanh with 1 hidden layer(s) and 25 hidden nodes per layer: 0.00111533676201
Shortcut Method MSE for tanh with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00302590366175
Shortcut Method MSE for tanh with 3 hidden layer(s) and 25 hidden nodes per layer: 0.00577514226946
Shortcut Method MSE for tanh with 1 hidden layer(s) and 50 hidden nodes per layer: 0.00085890681071
Shortcut Method MSE for tanh with 2 hidden layer(s) and 50 hidden nodes per layer: 0.000150263123278
Shortcut Method MSE for tanh with 3 hidden layer(s) and 50 hidden nodes per layer: 8.22888677309e-05
```



Solution (cont.)




```
def regreession_multi(X,y,lamb):
    X_train = X.T
    y_train = y
    n = len(X_train.T)
    A = np.linalg.solve(X_train.T.dot(X_train)
      +lamb*np.identity(n),X_train.T.dot(y_train))
    yhat = X_train.dot(A)

    Rmean = np.sum(np.square(yhat-y_train))/2

    return {'A':A,'yhat':yhat,'error':Rmean}

```

Solution (cont.)

```
    def shortcut_train(self, x, y):
        # Initialize some stuff
        n = x.shape[0]
        x = x.copy()
        y = y.copy()

        yp, a, z = self.evaluate(x)

        r = regreession_multi(a[-1],y,0.001)

        # Compute the error at the end of the epoch
        yh = r['yhat']
        error = r['error']

        return yh, error
```

(j)

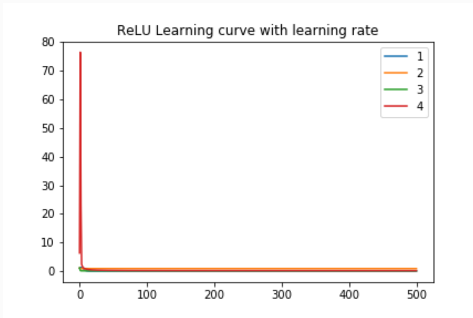
Smaller batch size may decrease the MSE. However, if the batch size is too small, the MSE will increase.

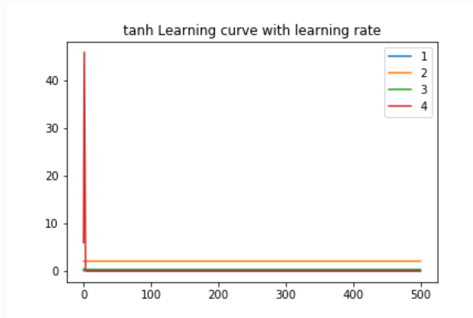
```
ReLU MSE with batch size 50 : 0.0379531806247
ReLU MSE with batch size 200 : 0.0243604434209
ReLU MSE with batch size 500 : 0.00544977526828
ReLU MSE with batch size 750 : 0.00350383781535
ReLU MSE with batch size 1000 : 0.00505661595353
tanh MSE with batch size 50 : 0.00719265709315
tanh MSE with batch size 200 : 0.00559489895698
tanh MSE with batch size 500 : 0.00188544761721
tanh MSE with batch size 750 : 0.00145286979291
tanh MSE with batch size 1000 : 0.00114572700474
linear MSE with batch size 50 : 0.0969320216089
linear MSE with batch size 200 : 0.0967507480698
linear MSE with batch size 500 : 0.0967698026411
linear MSE with batch size 750 : 0.0968384993999
linear MSE with batch size 1000 : 0.0967346963549
```

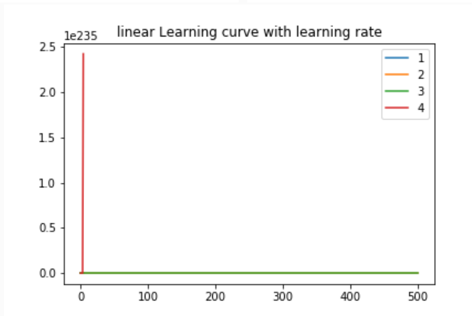
(k)

I choose the learning rate 0.02, 0.01, 0.005 and $\frac{0.1}{i+1}$. We can see that not every choice of learning rate works for all situations.

Solution (cont.)








```
ReLU MSE with learning rate 1 : 0.120725255269
ReLU MSE with learning rate 2 : 1.00873795376
ReLU MSE with learning rate 3 : 0.243439908327
ReLU MSE with learning rate 4 : 0.299413032558
```

```
tanh MSE with learning rate 1 : 0.138984140006
tanh MSE with learning rate 2 : 2.13982396204
tanh MSE with learning rate 3 : 0.414022252071
tanh MSE with learning rate 4 : 0.0522339096461
```

```
linear MSE with learning rate 1 : nan
linear MSE with learning rate 2 : 0.897948664528
linear MSE with learning rate 3 : 0.86985136178
linear MSE with learning rate 4 : nan
```

Question 3

Question What are the common variants of the artificial neural network? And what are the problems using gradient descend in neural network?

Solution

Variants

(1) Convolutional neural networks (CNN)

CNNs are suitable for processing visual and other two-dimensional data. They have shown superior results in both image and speech applications. They can be trained with standard backpropagation. CNNs are easier to train than other regular, deep, feed-forward neural networks and have many fewer parameters to estimate. Examples of applications in computer vision include DeepDream.

(2) Long short-term memory (LSTM)

Long short-term memory (LSTM) networks are RNNs that avoid the vanishing gradient problem. LSTM is normally augmented by recurrent gates called forget gates. LSTM networks prevent backpropagated errors from vanishing or exploding. Instead errors can flow backwards through unlimited numbers of virtual layers in space-unfolded LSTM. That is, LSTM can learn "very deep learning" tasks that require memories of events that happened thousands or even millions of discrete time steps ago. Problem-specific LSTM-like topologies can be evolved. LSTM can handle long delays and signals that have a mix of low and high frequency components.

(3) Deep belief networks (DBN)

A deep belief network (DBN) is a probabilistic and generative model made up of multiple layers of hidden units. It can be considered a composition of simple learning modules that make up each layer. A DBN can be used to generatively pre-train a DNN by using the learned DBN weights as the initial DNN weights. Backpropagation or other discriminative algorithms can then tune these weights. This is particularly helpful when training data are limited, because poorly initialized weights can significantly hinder model performance. These pre-trained weights are in a region of the weight space that is closer to the optimal weights than were they randomly chosen. This allows for both improved modeling and faster convergence of the fine-tuning phase.

Problems

(1) Rely on initialization of hyperparameter.

(2) Gradient vanishing or blowing up.

(3) Hardware issue.

HW7

October 18, 2017

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [3]: def regreession_multi(X,y,lamb):
            X_train = X.T
            y_train = y
            n = len(X_train.T)
            A = np.linalg.solve(X_train.T.dot(X_train)+lamb*np.identity(n),X_train.T.dot(y_train
            yhat = X_train.dot(A)

            Rmean = np.sum(np.square(yhat-y_train))/2

            return {'A':A,'yhat':yhat,'error':Rmean}
```

```
In [88]: # 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):

                 for layer, curGrad, curA in zip(layers, g, a):
                     # TODO ################################################################
                     # Compute the gradients for layer.W and layer.b using the gradient for the
                     # layer curA and the gradient of the output curGrad
                     # Use the gradients to update the weight and the bias for the layer
                     # ######################################################################
                     n = len(curA.T)
                     layer.updateWeights(self.eta * curGrad.T.dot(curA.T) / n)
                     layer.updateBias(self.eta* np.mean(curGrad.T,axis=1,keepdims=True))
```

```python
# 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 np.square(y - yp)/2

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

# 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.maximum(z,0)

    @staticmethod
    def dx(z):
        # TODO ######################################################################
        # Implement me
        # ##########################################################################
        x =np.zeros_like(z)
        x[z>=0] =1
        return x

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

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

# 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

    # 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
            # Set yp equal to the output of the network
            # ########################################################################
            z.append(layer.z(a[-1]))
            a.append(layer.a(z[-1]))
        yp = a[-1]
        a = a[:-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
```

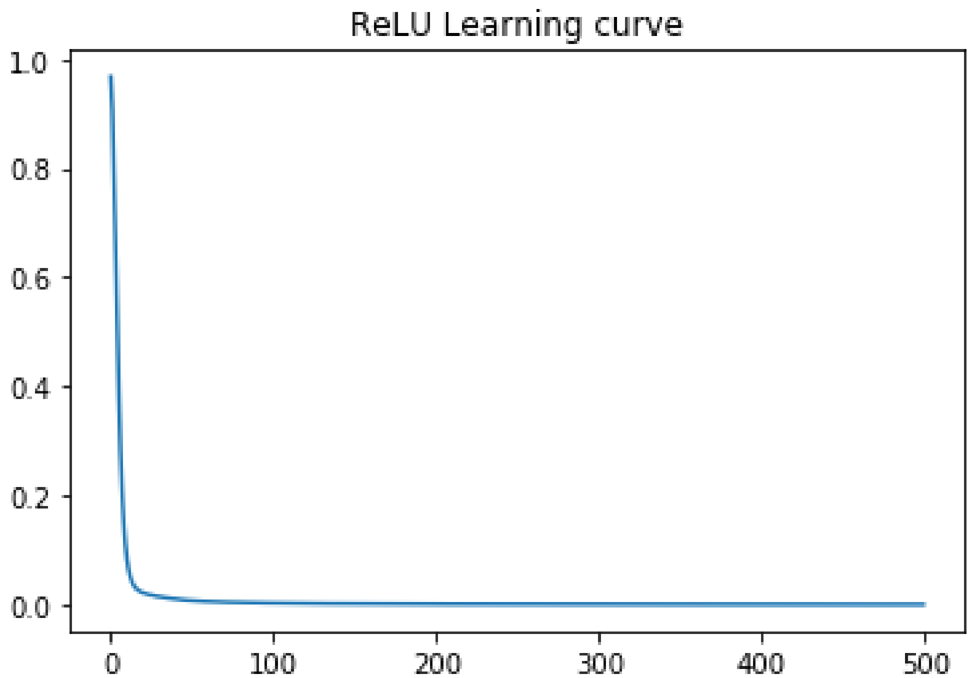
```python
# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
def train(self, x, y, numEpochs, optimizer):

    # 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)
        grad = []
        w = np.ones((len(d),len(d)))
        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers),reversed(z)):
            # TODO ################################################################
            # Compute the gradient of the output of each layer with respect to the
            # grad[i] should correspond with the gradient of the output of layer i
            # ######################################################################
            # dE/dsigma3*dsigma3/dz3
            # dE/dsigma3*dsigma3/dz3*dz3/dW2
            if len(grad)==0:
                grad.insert(0,(layer.dx(curZ)*d).T)
            else:
                grad.insert(0,d.dot(w)*layer.dx(curZ.T))
            #print('-------')
            w = layer.W
            d = grad[0]

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

        # Compute the error at the end of the epoch
        yh = self.predict(x)
```

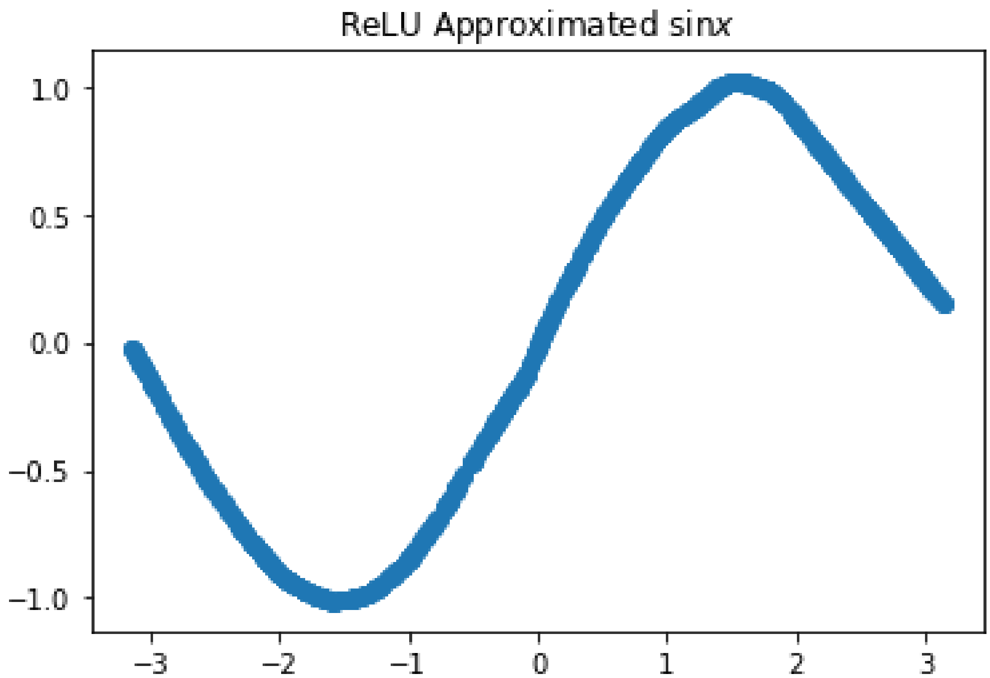
```python
            C = self.cost.fx(yh,y)
            C = np.mean(C)
            hist.append(C)

        return hist

    def shortcut_train(self, x, y):

        # Initialize some stuff
        n = x.shape[0]
        x = x.copy()
        y = y.copy()

        yp, a, z = self.evaluate(x)

        r = regreession_multi(a[-1],y,0.001)

        # Compute the error at the end of the epoch
        yh = r['yhat']
        error = r['error']

        return yh, error
```

In [89]:
```python
# 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))
```

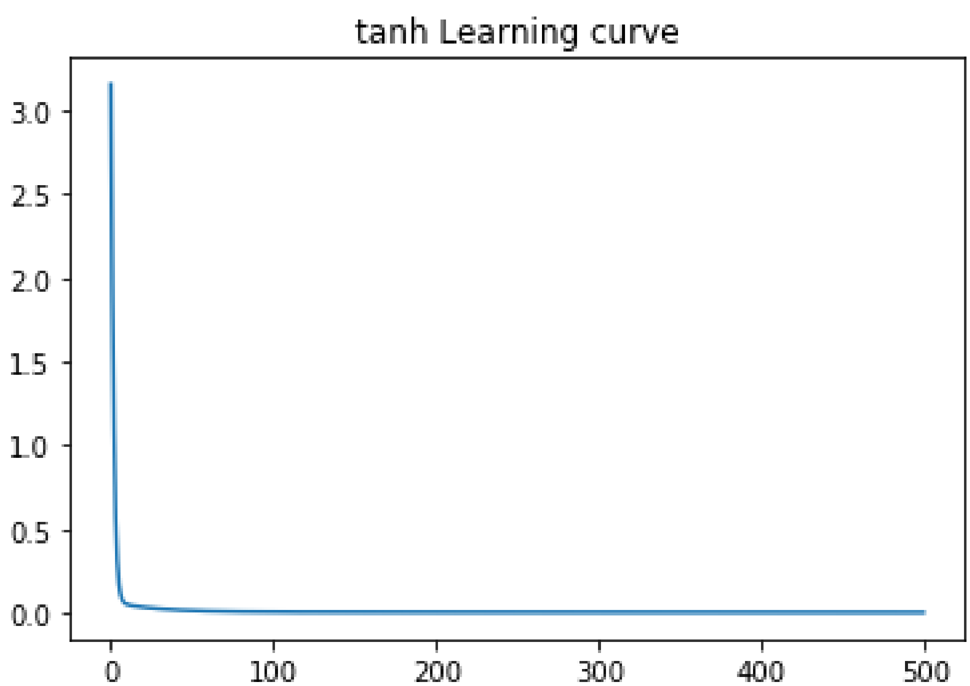
```python
    plt.plot(hist)
    plt.title(key+' Learning curve')
    plt.savefig(key+'l')
    plt.show()

    # TODO ################################################################################
    # Plot the approximation of the sin function from all of the models
    # ################################################################################
    plt.scatter(x,yHat)
    plt.title(key+' Approximated $\sin x$')
    plt.savefig(key+'a')
    plt.show()
```

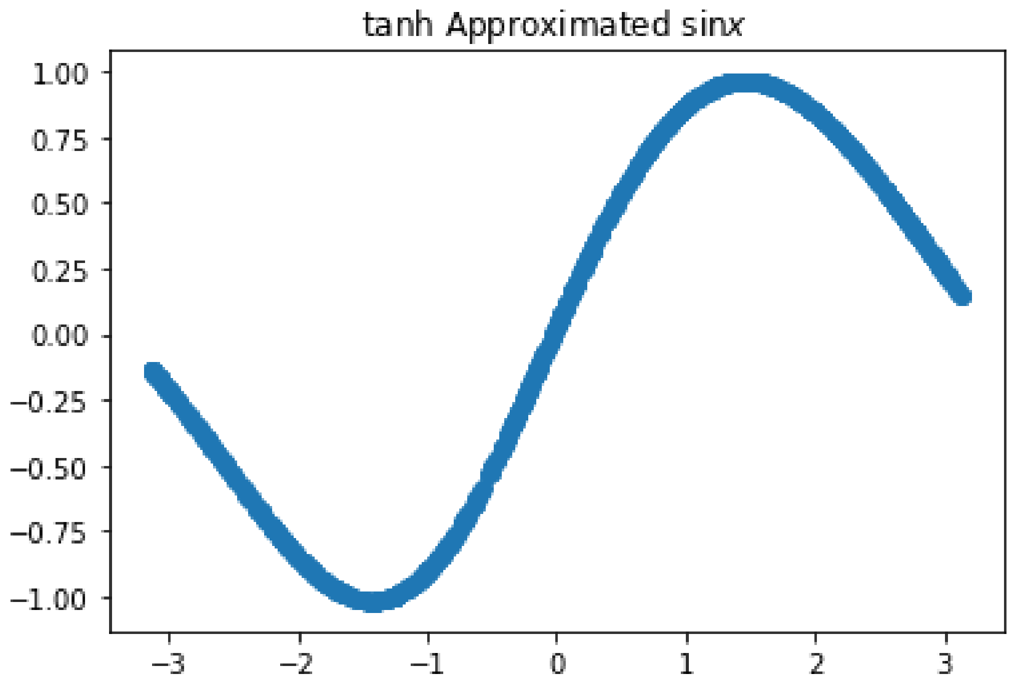
```
ReLU MSE: 0.000408666485428
```



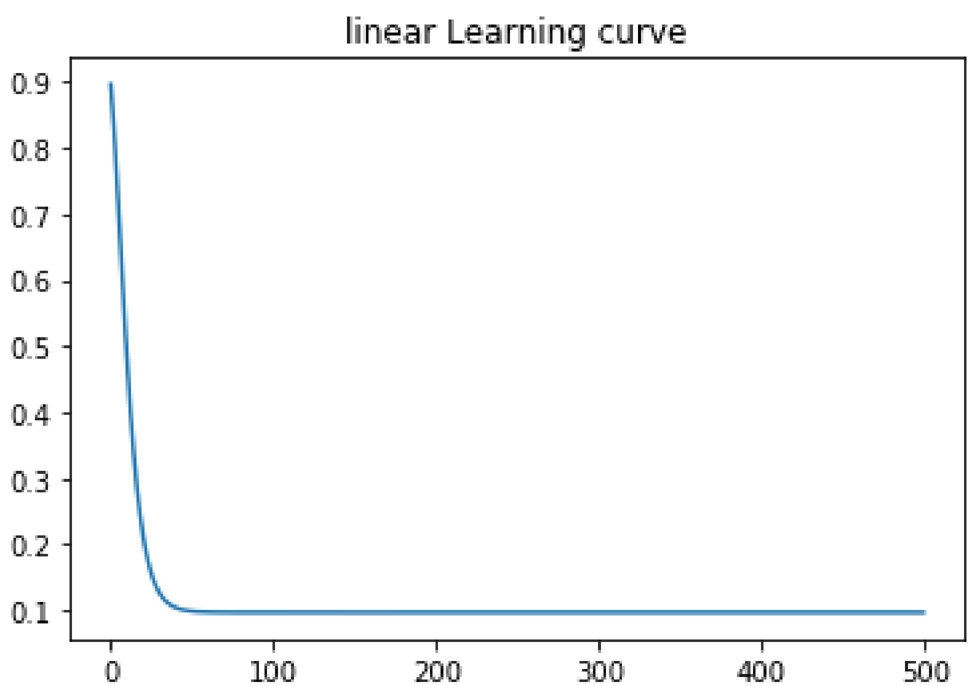


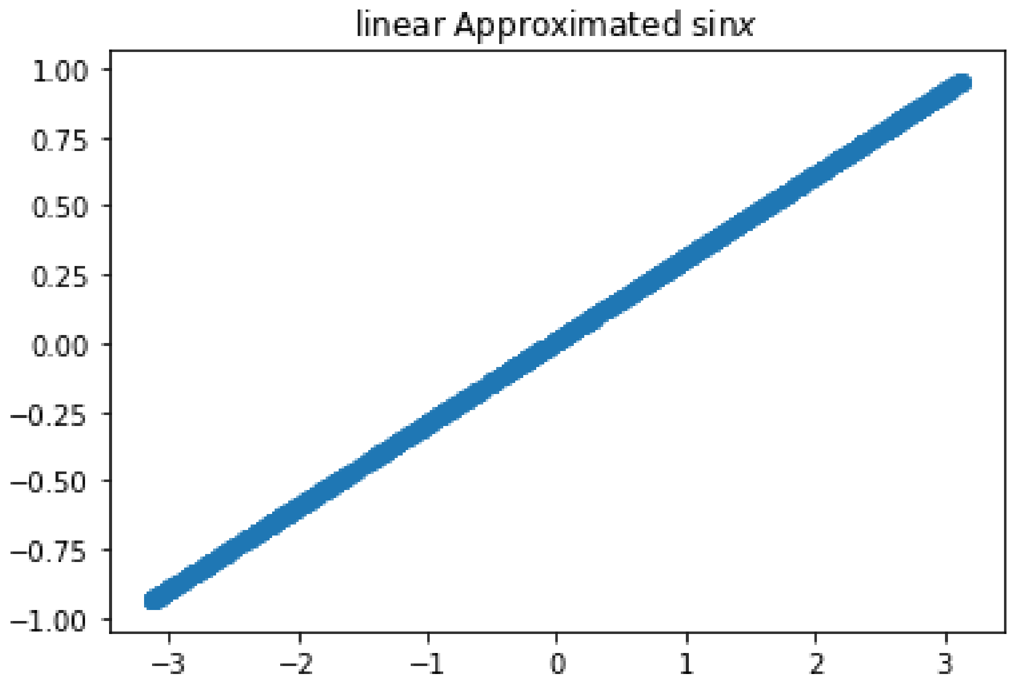
`tanh MSE: 0.00126499054514`






```
linear MSE: 0.0967346963549
```






```
In [66]: # 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)
         lr = dict(ReLU=0.01,tanh=0.01)
         num_hidden_nodes = [5,10,25,50]
         num_hidden_layers = [1,2,3]

         result = []
         for key in activations:
             # Build the model
             activation = activations[key]
             MSE = np.zeros((len(num_hidden_nodes),len(num_hidden_layers)))
             for i in range(len(num_hidden_nodes)):
                 for j in range(len(num_hidden_layers)):
                     model = Model(x.shape[1])
                     for _ in range(num_hidden_layers[j]):
                         model.addLayer(DenseLayer(num_hidden_nodes[i],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
                MSE[i,j] = error
                print('MSE for '+key+' with '+str(num_hidden_layers[j])+' hidden layer(s) a
                      +str(num_hidden_nodes[i])+' hidden nodes per layer: '+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
            # ###############################################################################
            #plt.scatter(x,yHat)
            #plt.title(key+' Approximated $\sin x$')
            #plt.show()
        result.append(MSE)
```

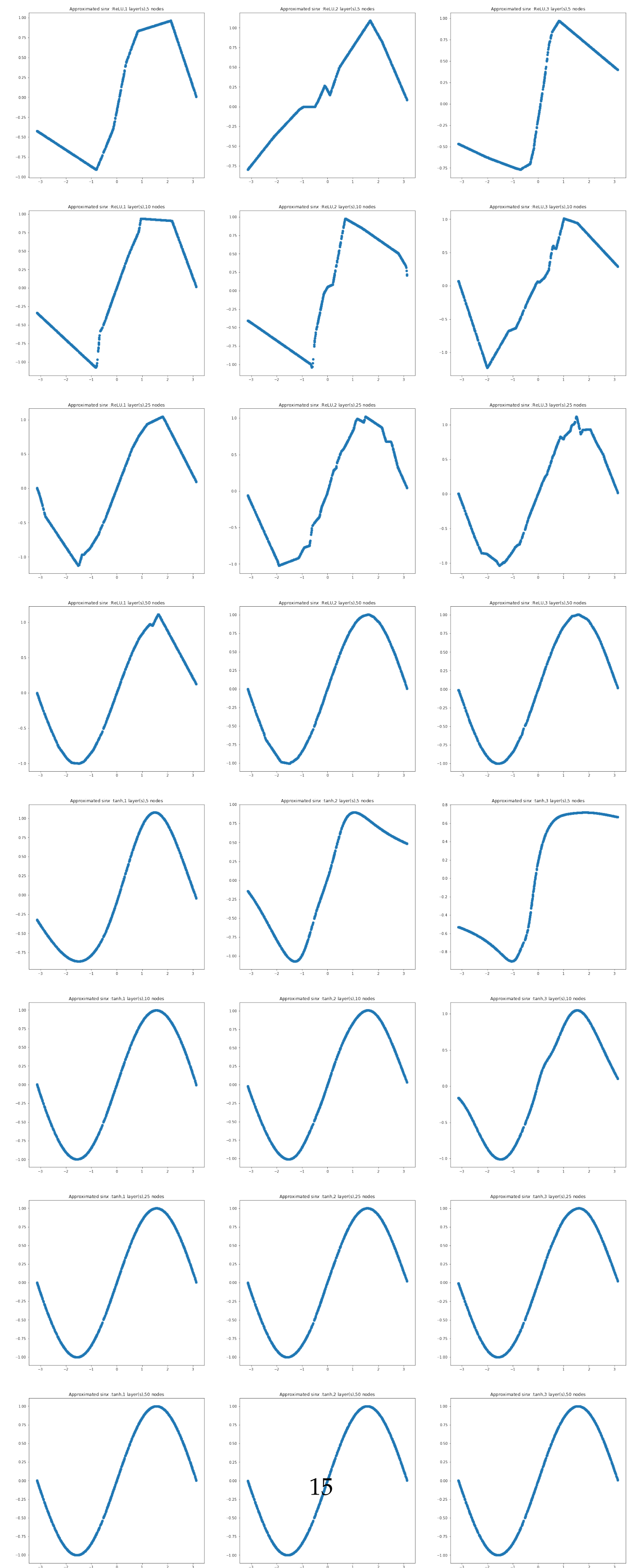
```
MSE for ReLU with 1 hidden layer(s) and 5 hidden nodes per layer: 0.0234909362319
MSE for ReLU with 2 hidden layer(s) and 5 hidden nodes per layer: 0.157080596055
MSE for ReLU with 3 hidden layer(s) and 5 hidden nodes per layer: 0.00620014687172
MSE for ReLU with 1 hidden layer(s) and 10 hidden nodes per layer: 0.0439981734314
MSE for ReLU with 2 hidden layer(s) and 10 hidden nodes per layer: 0.0375509396886
MSE for ReLU with 3 hidden layer(s) and 10 hidden nodes per layer: 0.0195351366292
MSE for ReLU with 1 hidden layer(s) and 25 hidden nodes per layer: 0.0101003292316
MSE for ReLU with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00919474762221
MSE for ReLU with 3 hidden layer(s) and 25 hidden nodes per layer: 0.00179816199785
MSE for ReLU with 1 hidden layer(s) and 50 hidden nodes per layer: 0.0164250943699
MSE for ReLU with 2 hidden layer(s) and 50 hidden nodes per layer: 0.00423117129644
MSE for ReLU with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00135987758048
MSE for tanh with 1 hidden layer(s) and 5 hidden nodes per layer: 0.0516499405974
MSE for tanh with 2 hidden layer(s) and 5 hidden nodes per layer: 0.0309338075028
MSE for tanh with 3 hidden layer(s) and 5 hidden nodes per layer: 0.0387015231705
MSE for tanh with 1 hidden layer(s) and 10 hidden nodes per layer: 0.0261680724503
MSE for tanh with 2 hidden layer(s) and 10 hidden nodes per layer: 0.0120426949507
MSE for tanh with 3 hidden layer(s) and 10 hidden nodes per layer: 0.0207633717414
MSE for tanh with 1 hidden layer(s) and 25 hidden nodes per layer: 0.0160555156441
MSE for tanh with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00674050797273
MSE for tanh with 3 hidden layer(s) and 25 hidden nodes per layer: 0.0040228468676
MSE for tanh with 1 hidden layer(s) and 50 hidden nodes per layer: 0.0138829301916
MSE for tanh with 2 hidden layer(s) and 50 hidden nodes per layer: 0.00244803141724
MSE for tanh with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00044184469178
```

0.1 (i)

```
In [67]: # 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)
         lr = dict(ReLU=0.01,tanh=0.01)
         num_hidden_nodes = [5,10,25,50]
         num_hidden_layers = [1,2,3]

         result2 = []
         ip = 1
         plt.figure(figsize=(30,80))
         for key in activations:
             # Build the model
             activation = activations[key]
             MSE2 = np.zeros((len(num_hidden_nodes),len(num_hidden_layers)))
             for i in range(len(num_hidden_nodes)):
                 for j in range(len(num_hidden_layers)):
                     model = Model(x.shape[1])
                     for _ in range(num_hidden_layers[j]):
                         model.addLayer(DenseLayer(num_hidden_nodes[i],activation()))

                     model.addLayer(DenseLayer(1,LinearActivation()))
                     model.initialize(QuadraticCost())

                     # Train the model and display the results
                     yHat, error = model.shortcut_train(x,y)
                     MSE2[i,j] = error
                     print('Shortcut Method MSE for '+key+' with '+str(num_hidden_layers[j])+' h
                           +str(num_hidden_nodes[i])+' hidden nodes per layer: '+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
                 # ##########################################################################
                     plt.subplot(8,3,ip)
                     ip += 1
                     plt.scatter(x,yHat)
                     plt.title('Approximated $\sin x$ :'+key+','+str(num_hidden_layers[j])+' lay
                           +str(num_hidden_nodes[i])+' nodes')

             result2.append(MSE2)
```

```
    plt.savefig('i.png')
    plt.show()
```

```
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 5 hidden nodes per layer: 11.6666057636
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 5 hidden nodes per layer: 85.5131245524
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 5 hidden nodes per layer: 25.4156105084
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 10 hidden nodes per layer: 6.61275585732
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 10 hidden nodes per layer: 16.4723774847
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 10 hidden nodes per layer: 8.75137056198
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 25 hidden nodes per layer: 0.48518834232
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 25 hidden nodes per layer: 0.46903501370
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 25 hidden nodes per layer: 0.45640245069
Shortcut Method MSE for ReLU with 1 hidden layer(s) and 50 hidden nodes per layer: 0.43177742714
Shortcut Method MSE for ReLU with 2 hidden layer(s) and 50 hidden nodes per layer: 0.01001173353
Shortcut Method MSE for ReLU with 3 hidden layer(s) and 50 hidden nodes per layer: 0.00557636785
Shortcut Method MSE for tanh with 1 hidden layer(s) and 5 hidden nodes per layer: 4.37525255244
Shortcut Method MSE for tanh with 2 hidden layer(s) and 5 hidden nodes per layer: 9.32830455191
Shortcut Method MSE for tanh with 3 hidden layer(s) and 5 hidden nodes per layer: 24.6123101195
Shortcut Method MSE for tanh with 1 hidden layer(s) and 10 hidden nodes per layer: 0.00401742362
Shortcut Method MSE for tanh with 2 hidden layer(s) and 10 hidden nodes per layer: 0.03432733437
Shortcut Method MSE for tanh with 3 hidden layer(s) and 10 hidden nodes per layer: 0.97426387425
Shortcut Method MSE for tanh with 1 hidden layer(s) and 25 hidden nodes per layer: 0.00111533676
Shortcut Method MSE for tanh with 2 hidden layer(s) and 25 hidden nodes per layer: 0.00302590366
Shortcut Method MSE for tanh with 3 hidden layer(s) and 25 hidden nodes per layer: 0.00577514226
Shortcut Method MSE for tanh with 1 hidden layer(s) and 50 hidden nodes per layer: 0.00085890681
Shortcut Method MSE for tanh with 2 hidden layer(s) and 50 hidden nodes per layer: 0.00015026312
Shortcut Method MSE for tanh with 3 hidden layer(s) and 50 hidden nodes per layer: 8.22888677309
```



Approximated sinx :ReLU,1 layer(s),5 nodes
Approximated sinx :ReLU,2 layer(s),5 nodes
Approximated sinx :ReLU,3 layer(s),5 nodes
Approximated sinx :ReLU,1 layer(s),10 nodes
Approximated sinx :ReLU,2 layer(s),10 nodes
Approximated sinx :ReLU,3 layer(s),10 nodes
Approximated sinx :ReLU,1 layer(s),25 nodes
Approximated sinx :ReLU,2 layer(s),25 nodes
Approximated sinx :ReLU,3 layer(s),25 nodes
Approximated sinx :ReLU,1 layer(s),50 nodes
Approximated sinx :ReLU,2 layer(s),50 nodes
Approximated sinx :ReLU,3 layer(s),50 nodes
Approximated sinx :tanh,1 layer(s),5 nodes
Approximated sinx :tanh,2 layer(s),5 nodes
Approximated sinx :tanh,3 layer(s),5 nodes
Approximated sinx :tanh,1 layer(s),10 nodes
Approximated sinx :tanh,2 layer(s),10 nodes
Approximated sinx :tanh,3 layer(s),10 nodes
Approximated sinx :tanh,1 layer(s),25 nodes
Approximated sinx :tanh,2 layer(s),25 nodes
Approximated sinx :tanh,3 layer(s),25 nodes
Approximated sinx :tanh,1 layer(s),50 nodes
Approximated sinx :tanh,2 layer(s),50 nodes
Approximated sinx :tanh,3 layer(s),50 nodes

0.2 (j)

```
In [70]: import random
```

```
In [86]: # 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, index):
                 for layer, curGrad, curA in zip(layers, g, a):
                     # TODO #####################################################################
                     # Compute the gradients for layer.W and layer.b using the gradient for the
                     # layer curA and the gradient of the output curGrad
                     # Use the gradients to update the weight and the bias for the layer
                     # ##########################################################################
                     n = len(curA.T)
                     layer.updateWeights(self.eta * curGrad.T.dot(curA[:,index].T) / n)
                     layer.updateBias(self.eta* np.mean(curGrad.T,axis=1,keepdims=True))

         # 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 np.square(y - yp)/2

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

```
        # Implement me
        # ###########################################################################
        return yp - y

# 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
        # ###########################################################################
        x = z
        x[x<0] = 0
        return x

    @staticmethod
    def dx(z):
        # TODO ######################################################################
        # Implement me
```

```
        # ###########################################################################
        x = np.ones_like(z)
        x[z<0] = 0
        return x

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

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

# 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

    # 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):
        #print('W',np.shape(self.W),np.shape(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
            # Set yp equal to the output of the network
            # ######################################################################
            z.append(layer.z(a[-1]))
            a.append(layer.a(z[-1]))
        yp = a[-1]
        a = a[:-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, batchsize):

        # 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)
            index = random.sample(range(len(yp.T)),batchsize)
            # Compute the error
            C = self.cost.fx(yp,y.T)
            d = self.cost.dx(yp[:,index],y[index].T)
            grad = []
            w = np.ones((len(d),len(d)))

            # Backpropogate the error
            for layer, curZ in zip(reversed(self.layers),reversed(z)):
                # TODO ###############################################################
                # Compute the gradient of the output of each layer with respect to the
                # grad[i] should correspond with the gradient of the output of layer i
                # #####################################################################
                if len(grad)==0:
                    grad.insert(0,(layer.dx(curZ[:,index])*d).T)
                else:
                    grad.insert(0,d.dot(w)*layer.dx(curZ[:,index].T))
                w = layer.W
                d = grad[0]

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

            # 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
```

In [87]:
```
# 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)

batchsize = [50,200,500,750, 1000]
```

```python
    for key in activations:
        for b in batchsize:
            # 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]), b)
            yHat = model.predict(x)
            error = np.mean(np.square(yHat - y))/2
            print(key+' MSE with batch size '+str(b)+' : ' +str(error))
            #plt.plot(hist)
            #plt.title(key+' Learning curve')
            #plt.savefig(key+'l')
            #plt.show()

            # TODO ###########################################################################
            # Plot the approximation of the sin function from all of the models
            # ################################################################################
            #plt.scatter(x,yHat)
            #plt.title(key+' Approximated $\sin x$')
            #plt.savefig(key+'a')
            #plt.show()
```

```
ReLU MSE with batch size 50 : 0.0379531806247
ReLU MSE with batch size 200 : 0.0243604434209
ReLU MSE with batch size 500 : 0.00544977526828
ReLU MSE with batch size 750 : 0.00350383781535
ReLU MSE with batch size 1000 : 0.00505661595353
tanh MSE with batch size 50 : 0.00719265709315
tanh MSE with batch size 200 : 0.00559489895698
tanh MSE with batch size 500 : 0.00188544761721
tanh MSE with batch size 750 : 0.00145286979291
tanh MSE with batch size 1000 : 0.00114572700474
linear MSE with batch size 50 : 0.0969320216089
linear MSE with batch size 200 : 0.0967507480698
linear MSE with batch size 500 : 0.0967698026411
linear MSE with batch size 750 : 0.0968384993999
linear MSE with batch size 1000 : 0.0967346963549
```

0.3 (k)

In [82]:
```
# 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, index, i):
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO ##################################################################
            # Compute the gradients for layer.W and layer.b using the gradient for the
            # layer curA and the gradient of the output curGrad
            # Use the gradients to update the weight and the bias for the layer
            # ######################################################################
            n = len(curA.T)
            layer.updateWeights(self.eta(i) * curGrad.T.dot(curA[:,index].T) / n)
            layer.updateBias(self.eta(i) * np.mean(curGrad.T,axis=1,keepdims=True))

# 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 np.square(y - yp)/2

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

# 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
        # ##########################################################################
        x = z
        x[x<0] = 0
        return x

    @staticmethod
    def dx(z):
        # TODO ######################################################################
        # Implement me
        # ##########################################################################
        x = np.ones_like(z)
        x[z<0] = 0
        return x
```

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

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

# 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

    # 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
```

```python
    # 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):
        #print('W',np.shape(self.W),np.shape(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
```

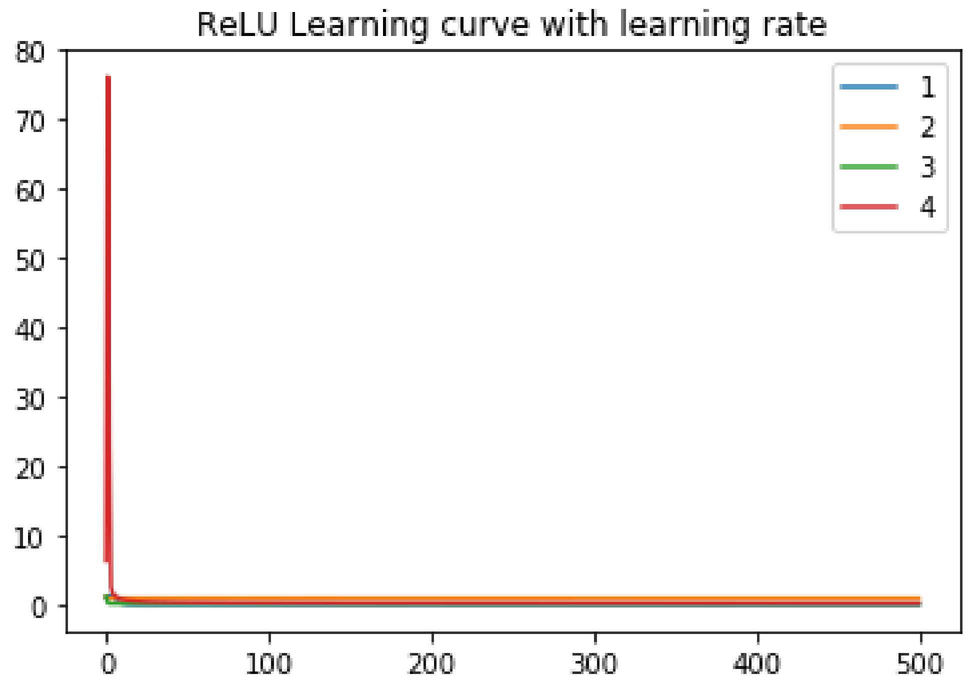
```
    # This function returns
    # yp - the output of the network
    # a - a list of inputs for each layer of the newtork where
    #     a[i] is the input to layer i
    # z - a list of values for each layer after evaluating layer.z(a) but
    #     before evaluating the nonlinear function for the 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
            # Set yp equal to the output of the network
            # ###########################################################################
            z.append(layer.z(a[-1]))
            a.append(layer.a(z[-1]))
        yp = a[-1]
        a = a[:-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, batchsize):

        # 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)
                index = random.sample(range(len(yp.T)),batchsize)
                # Compute the error
                C = self.cost.fx(yp,y.T)
                d = self.cost.dx(yp[:,index],y[index].T)
                d = np.mean(d,keepdims=True)
                grad = []
                w = np.ones((1,1))
                # Backpropogate the error
                for layer, curZ in zip(reversed(self.layers),reversed(z)):
                    # TODO ################################################################
                    # Compute the gradient of the output of each layer with respect to the
                    # grad[i] should correspond with the gradient of the output of layer i
                    # ######################################################################
                    if len(grad)==0:
                        grad.insert(0,(layer.dx(curZ[:,index])*d).T)
                    else:
                        grad.insert(0,d.dot(w)*layer.dx(curZ[:,index].T))
                    w = layer.W
                    d = grad[0]

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

                # 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
```

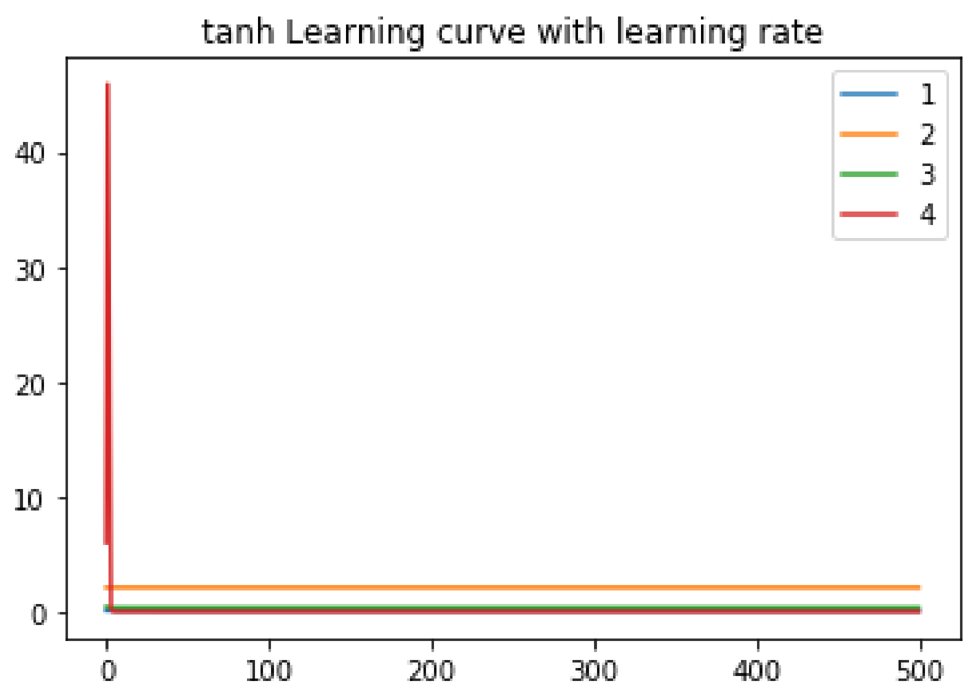
In [85]:
```
# Generate the training set
np.random.seed(9001)
x=np.random.uniform(-np.pi,np.pi,(1000,1))
y=np.sin(x)

activations = dict(ReLU=ReLUActivation,
                   tanh=TanhActivation,
                   linear=LinearActivation)
lr = [lambda i:0.02,lambda i:0.01,lambda i:0.005,lambda i:0.1/(i+1)]

for key in activations:
    i = 1
    for l in lr:
        # 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=l), 1000)
            yHat = model.predict(x)
            error = np.mean(np.square(yHat - y))/2
            print(key+' MSE with learning rate '+str(i)+' : ' +str(error))
            plt.plot(hist,label=str(i))
            i+=1
            plt.title(key+' Learning curve with learning rate')
            #plt.savefig(key+'l')

            # TODO ##################################################################
            # Plot the approximation of the sin function from all of the models
            # #######################################################################
            #plt.scatter(x,yHat)
            #plt.title(key+' Approximated $\sin x$')
            #plt.savefig(key+'a')
            #plt.show()
        plt.legend()
        plt.savefig('k'+key)
        plt.show()
```

```
ReLU MSE with learning rate 1 : 0.120725255269
ReLU MSE with learning rate 2 : 1.00873795376
ReLU MSE with learning rate 3 : 0.243439908327
ReLU MSE with learning rate 4 : 0.299413032558
```

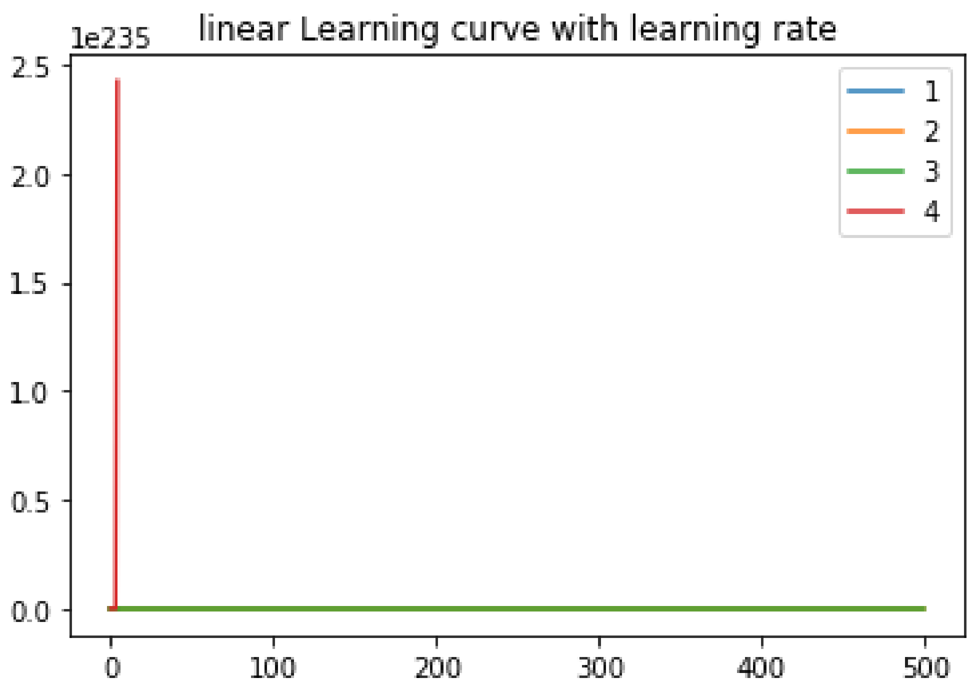



```
tanh MSE with learning rate 1 : 0.138984140006
tanh MSE with learning rate 2 : 2.13982396204
tanh MSE with learning rate 3 : 0.414022252071
tanh MSE with learning rate 4 : 0.0522339096461
```



```
/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:36: RuntimeWarning: overflow encount
```

```
linear MSE with learning rate 1 : nan
linear MSE with learning rate 2 : 0.897948664528
linear MSE with learning rate 3 : 0.86985136178
linear MSE with learning rate 4 : nan
```




```
In [ ]:
```