CS 189 Introduction to Machine Learning
Fall 2017 HW7

This homework is due **Saturday, October 14 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

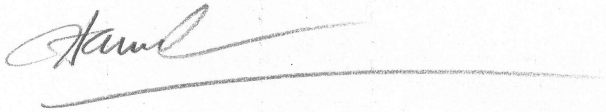
(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

None. I work alone
Comments The skeleton code is terrible! It is still buggy and not updated until last minutes. MAKE SURE IT WORKS ON YOUR COMPUTER BEFORE RELEASING IT.

(b) Please copy the following statement and sign next to it:

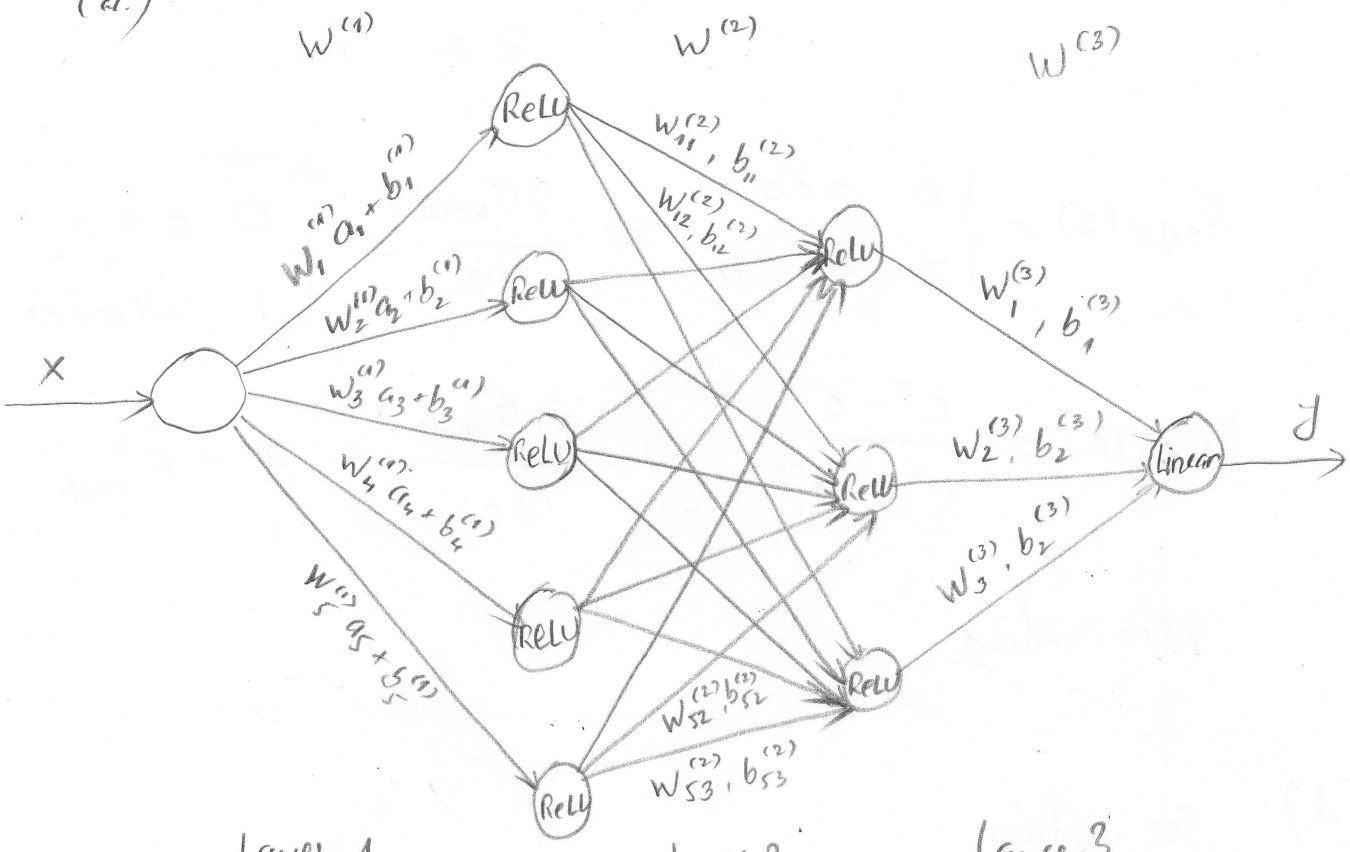
I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

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.



Problem #2

(a.)



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

$$\Rightarrow \frac{\partial MSE}{\partial \vec{\hat{y}}} = \vec{\hat{y}} - \vec{y}$$

See coding

(c) $\sigma_{linear}(z) = z \Rightarrow \frac{\partial \sigma_{linear}}{\partial z} = 1$

$$\sigma_{ReLU}(z) = \begin{cases} 0 & z<0 \\ z & \text{otherwise} \end{cases} \Rightarrow \frac{\partial \sigma_{ReLU}}{\partial z} = \begin{cases} 0 & z<0 \\ 1 & \text{otherwise} \end{cases}$$

$$\sigma_{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \Rightarrow \frac{\partial \sigma_{tanh}}{\partial z} = 1 - \sigma^2_{tanh}$$

See coding

(d) See coding

(e) $$\frac{\partial MSE}{\partial \vec{a}_i} = \frac{\partial MSE}{\partial \vec{a}_{i+1}} \underbrace{\frac{\partial \vec{a}_{i+1}}{\partial \vec{a}_i}}$$

$$\vec{z}_i = W_i \vec{a}_i + b_i$$

$$\underbrace{\frac{\partial \sigma(\vec{z}_i)}{\partial \vec{z}_i} \frac{\partial \vec{z}_i}{\partial a_i}}$$

$$\Rightarrow \frac{\partial MSE}{\partial \vec{a}_i} = \frac{\partial MSE}{\partial \vec{a}_{i+1}} \frac{\partial \sigma(\vec{z}_i)}{\partial \vec{z}_i} W_i$$

See coding

f.) $$\frac{\partial MSE}{\partial w_i} = \frac{\partial MSE}{\partial \vec{a}_{i+1}} \underbrace{\frac{\partial \vec{a}_{i+1}}{\partial w_i}}$$

$$\vec{z}_i = w_i \vec{a}_i + \vec{b}_i$$

$$\underbrace{\frac{\partial \sigma(\vec{z}_i)}{\partial \vec{z}_i} \underbrace{\frac{\partial \vec{z}_i}{\partial w_i}}_{\vec{a}_i}}$$

$$\frac{\partial MSE}{\partial w_i} = \frac{\partial MSE}{\partial \vec{a}_{i+1}} \frac{\partial \sigma(\vec{z}_i)}{\partial \vec{z}_i} \vec{a}_i$$

Similarly

$$\frac{\partial MSE}{\partial \vec{b}_i} = \frac{\partial MSE}{\partial \vec{a}_{i+1}} \frac{\partial \sigma(\vec{z}_i)}{\partial \vec{z}_i} \frac{\partial \vec{z}_i}{\partial \vec{b}_i} \rightarrow I$$

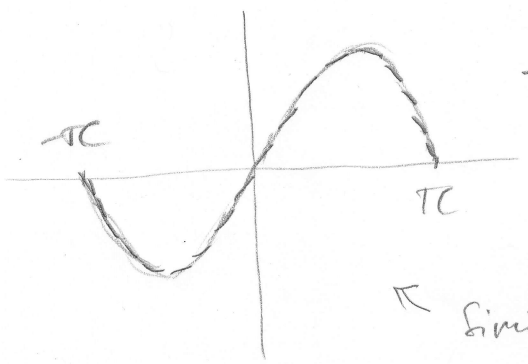
g.) Output

ReLU MSE : 0.00074328

Linear MSE : 0.12079632

tanh MSE : 0.00173546

Plot :



See coding

↖ Similar to the output

h.) not enough time to run

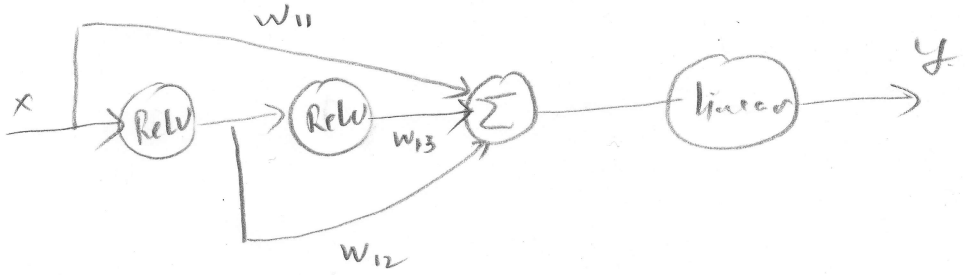
i.) not enough time to run

j.) skip

k.) skip

Problem #3

Consider the net-work:



1.) Write the out-put for each layers (from $x \to y$)

solution:

$$a_1 = w_{11} x + b_{11}$$

$$a_2 = w_{12} \sigma_{ReLU}(x) + b_{12}$$

$$a_3 = w_{13} \sigma_{ReLU}(\sigma_{ReLU}(x)) + b_{13}$$

$$a_4 = w_4 (a_1 + a_2 + a_3) + b_{14}$$

$$y = \sigma_{linear}(a_4) = a_4$$

where σ's are noise

2.) Define error & take the derivative of error w.r.t x given a_1, a_2, a_3 & a_4

Solution:

$$MSE(\vec{y_p}) = \frac{1}{2} \| \vec{y_p} - y \|_2^2$$

$$\frac{\partial MSE}{\partial \vec{x}} = \frac{\partial MSE}{\partial a_4} \frac{\partial a_4}{\partial a_1} \frac{\partial a_1}{\partial x} + \frac{\partial MSE}{\partial a_4} \frac{\partial a_4}{\partial a_2} \frac{\partial a_2}{\partial x} + \frac{\partial MSE}{\partial a_4} \frac{\partial a_4}{\partial a_3} \frac{\partial a_3}{\partial x}$$

w_{11} $\quad$ $w_{12} \frac{\partial \sigma_{ReLU}}{\partial x}$ $\quad$ $w_{13} \frac{\partial ReLU}{\partial a_2} \frac{\partial a_2}{\partial x}$

$w_{13} \frac{\partial \sigma_{ReLU}}{\partial x}$

```
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

```

```
49          # Derivative of the cost function with respect to yp
50          @staticmethod
51          def dx(y,yp):
52              # TODO
                ##########################################################################
                ###
53              # Implement me
54              #
                ##########################################################################
                ########
55              return y - yp
56
57      # Sigmoid function fully implemented as an example
58      class SigmoidActivation(object):
59          @staticmethod
60          def fx(z):
61              return 1 / (1 + np.exp(-z))
62
63          @staticmethod
64          def dx(z):
65              return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))
66
67      # Hyperbolic tangent function
68      class TanhActivation(object):
69
70          # Compute tanh for each element in the input z
71          @staticmethod
72          def fx(z):
73              # TODO
                ##########################################################################
                ###
74              # Implement me
75              #
                ##########################################################################
                ########
76              return np.tanh(z)
77
78          # Compute the derivative of the tanh function with respect to z
79          @staticmethod
80          def dx(z):
81              # TODO
                ##########################################################################
                ###
82              # Implement me
83              #
                ##########################################################################
                ########
84              return 1 - np.tanh(z) ** 2
85
86      # Rectified linear unit
87      class ReLUActivation(object):
88          @staticmethod
89          def fx(z):
90              # TODO
                ##########################################################################
                ###
91              # Implement me
```

```
92              #
                ##############################################################################
                ########
93              return np.where(z < 0, 0, z)
94
95          @staticmethod
96          def dx(z):
97              # TODO
                ##############################################################################
                ###
98              # Implement me
99              #
                ##############################################################################
                ########
100             return np.where(z <= 0, 0, 1)
101
102 # Linear activation
103 class LinearActivation(object):
104         @staticmethod
105         def fx(z):
106             # TODO
                ##############################################################################
                ###
107             # Implement me
108             #
                ##############################################################################
                ########
109             return z
110
111         @staticmethod
112         def dx(z):
113             # TODO
                ##############################################################################
                ###
114             # Implement me
115             #
                ##############################################################################
                ########
116             return np.ones(z.shape)
117
118 # This class represents a single hidden or output layer in the neural network
119 class DenseLayer(object):
120
121         # numNodes: number of hidden units in the layer
122         # activation: the activation function to use in this layer
123         def __init__(self, numNodes, activation):
124             self.numNodes = numNodes
125             self.activation = activation
126
127         def getNumNodes(self):
128             return self.numNodes
129
130         # Initialize the weight matrix of this layer based on the size of the matrix W
131         def initialize(self, fanIn, scale=1.0):
132             s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
133             self.W = np.random.normal(0, s,
134                                       (self.numNodes,fanIn))
```

```
135             self.b = np.random.uniform(-1,1,(self.numNodes,1))
136
137         # Apply the activation function of the layer on the input z
138         def a(self, z):
139             return self.activation.fx(z)
140
141         # Compute the linear part of the layer
142         # The input a is an n x k matrix where n is the number of samples
143         # and k is the dimension of the previous layer (or the input to the network)
144         def z(self, a):
145             return self.W.dot(a) + self.b # Note, this is implemented where we assume a
                is k x n
146
147         # Compute the derivative of the layer's activation function with respect to z
148         # where z is the output of the above function.
149         # This derivative does not contain the derivative of the matrix multiplication
150         # in the layer.  That part is computed below in the model class.
151         def dx(self, z):
152             return self.activation.dx(z)
153
154         # Update the weights of the layer by adding dW to the weights
155         def updateWeights(self, dW):
156             self.W = self.W + dW
157
158         # Update the bias of the layer by adding db to the bias
159         def updateBias(self, db):
160             self.b = self.b + db
161
162     # This class handles stacking layers together to form the completed neural network
163     class Model(object):
164
165         # inputSize: the dimension of the inputs that go into the network
166         def __init__(self, inputSize):
167             self.layers = []
168             self.inputSize = inputSize
169
170         # Add a layer to the end of the network
171         def addLayer(self, layer):
172             self.layers.append(layer)
173
174         # Get the output size of the layer at the given index
175         def getLayerSize(self, index):
176             if index >= len(self.layers):
177                 return self.layers[-1].getNumNodes()
178             elif index < 0:
179                 return self.inputSize
180             else:
181                 return self.layers[index].getNumNodes()
182
183         # Initialize the weights of all of the layers in the network and set the cost
184         # function to use for optimization
185         def initialize(self, cost, initializeLayers=True):
186             self.cost = cost
187             if initializeLayers:
188                 for i in range(0,len(self.layers)):
189                     if i == len(self.layers) - 1:
190                         self.layers[i].initialize(self.getLayerSize(i-1))
```

```
191                 else:
192                     self.layers[i].initialize(self.getLayerSize(i-1))
193 
194     # Compute the output of the network given some input a
195     # The matrix a has shape n x k where n is the number of samples and
196     # k is the dimension
197     # This function returns
198     # yp - the output of the network
199     # a - a list of inputs for each layer of the newtork where
200     #     a[i] is the input to layer i
201     # z - a list of values for each layer after evaluating layer.z(a) but
202     #     before evaluating the nonlinear function for the layer
203     def evaluate(self, x):
204         curA = x.T
205         a = [curA]
206         z = []
207         for layer in self.layers:
208             # TODO ↵
                ############################################################################ ↵
                #######
209             # Store the input to each layer in the list a
210             # Store the result of each layer before applying the nonlinear function ↵
                in z
211             # Set yp equal to the output of the network
212             # ↵
                ############################################################################ ↵
                ############
213             curZ = layer.z(curA)
214             z.append(curZ)
215             curA = layer.a(curZ)
216             a.append(curA)
217 
218         yp = a.pop(-1)
219 
220         return yp, a, z
221 
222     # Compute the output of the network given some input a
223     # The matrix a has shape n x k where n is the number of samples and
224     # k is the dimension
225     def predict(self, a):
226         a,_,_ = self.evaluate(a)
227         return a.T
228 
229     # Train the network given the inputs x and the corresponding observations y
230     # The network should be trained for numEpochs iterations using the supplied
231     # optimizer
232     def train(self, x, y, numEpochs, optimizer):
233 
234         # Initialize some stuff
235         n = x.shape[0]
236 
237         x = x.copy()
238         y = y.copy()
239         hist = []
240         optimizer.initialize(self.layers)
241 
242         # Run for the specified number of epochs
```

```
243             for epoch in range(0,numEpochs):
244
245                 # Feed forward
246                 # Save the output of each layer in the list a
247                 # After the network has been evaluated, a should contain the
248                 # input x and the output of each layer except for the last layer
249                 yp, a, z = self.evaluate(x)
250
251                 # Compute the error
252                 # C = self.cost.fx(yp,y.T)
253                 # d = self.cost.dx(yp,y.T)
254                 C = self.cost.fx(y.T,yp)
255                 d = self.cost.dx(y.T,yp)
256                 grad = []
257
258                 # Backpropogate the error
259                 idx = len(self.layers)
260                 for layer, curZ in zip(reversed(self.layers),reversed(z)):
261                     # TODO ↵
                    ############################################################################ ↵
                    ###########
262                     # Compute the gradient of the output of each layer with respect to ↵
                    the error
263                     # grad[i] should correspond with the gradient of the output of layer i
264                     # ↵
                    ############################################################################ ↵
                    ################
265                     # if len(grad) == 0:
266                     #   grad_i = d * layer.activation.dx(curZ)
267                     #   grad.append(grad_i)
268                     # else:
269                     #   grad_i = ( grad[-1].T.dot(last_layer_W) ).T * ↵
                    layer.activation.dx(curZ)
270                     #   grad.append(grad_i)
271                     # last_layer_W = layer.W
272                     if len(grad) == 0:
273                         grad_i = layer.W.T.dot( d * layer.activation.dx(curZ) )
274                         grad.append(grad_i)
275                     else:
276                         grad_i = layer.W.T.dot( grad[-1] * layer.activation.dx(curZ) )
277                         grad.append(grad_i)
278                 grad.pop(-1)
279                 grad.reverse()
280                 grad.append(d)
281
282                 # Update the errors
283                 optimizer.update(self.layers, grad, a)
284
285                 # Compute the error at the end of the epoch
286                 yh = self.predict(x)
287                 C = self.cost.fx(yh,y)
288                 C = np.mean(C)
289                 hist.append(C)
290         return hist
291
292 if __name__ == '__main__':
293
```

```
294         # Generate the training set
295         np.random.seed(9001)
296         x=np.random.uniform(-np.pi,np.pi,(1000,1))
297         y=np.sin(x)
298 
299         activations = dict(ReLU=ReLUActivation,
300                            tanh=TanhActivation,
301                            linear=LinearActivation)
302         lr = dict(ReLU=0.02,tanh=0.02,linear=0.005)
303 
304         for key in activations:
305 
306             # Build the model
307             activation = activations[key]
308             model = Model(x.shape[1])
309             model.addLayer(DenseLayer(100,activation()))
310             model.addLayer(DenseLayer(100,activation()))
311             model.addLayer(DenseLayer(1,LinearActivation()))
312             model.initialize(QuadraticCost())
313 
314             # Train the model and display the results
315             hist = model.train(x,y,500,GDOptimizer(eta=lr[key]))
316             yHat = model.predict(x)
317             error = np.mean(np.square(yHat - y))/2
318             print(key+' MSE: '+str(error))
319             plt.plot(hist)
320             plt.title(key+' Learning curve')
321             plt.show()
322 
323             # TODO ↵
                ############################################################################## ↵
                ###
324             # Plot the approximation of the sin function from all of the models
325             # ↵
                ############################################################################## ↵
                ########
326 
```