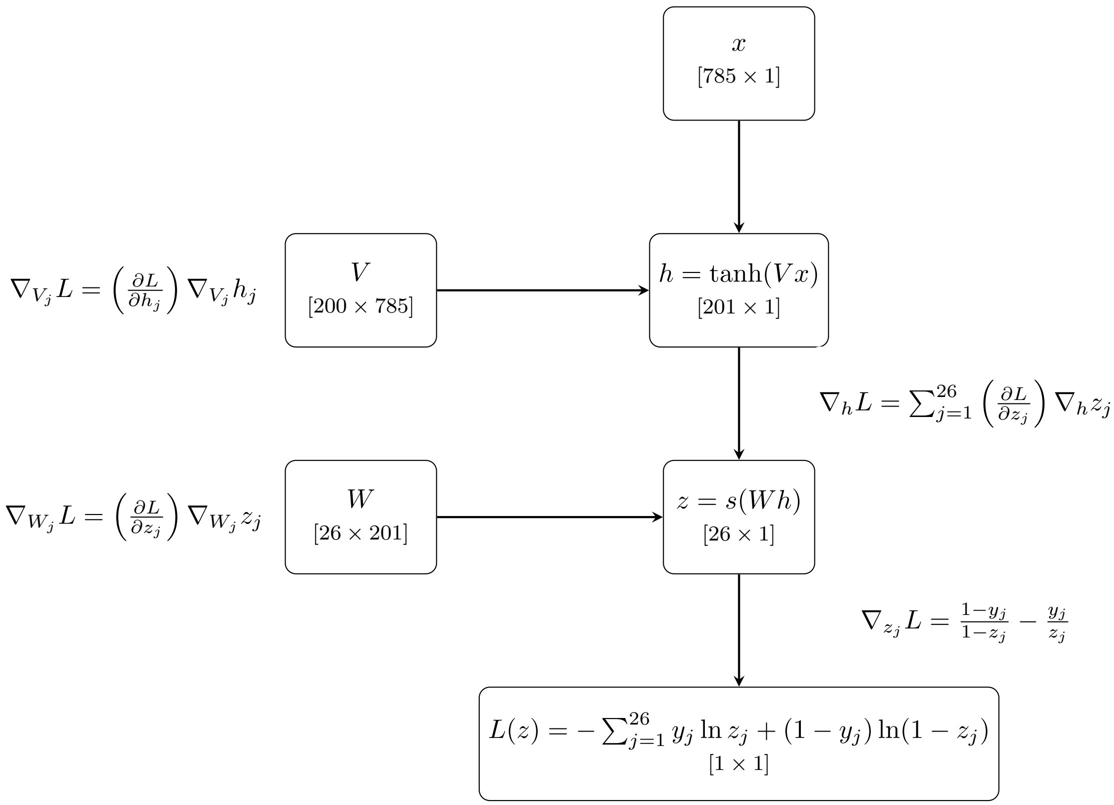
COMPSCI 289A Homework 6 **Mitch Negus**
4/14/2017 **3032146443**

Problem 1

The constructed neural net follows the diagram below. $x \in \mathbb{R}^{785}$. For clarity, shapes of the solutions are given in brackets. Subscript j denotes a row index. A_j^T indicates the transpose of A_j.



We can substitute the following expressions:

$\nabla_{W_j} z_j = z_j(1-z_j)h^T$ [1 × 201]

$\nabla_h z_j = z_j(1-z_j)W_j^T$ [201 × 1]

$\nabla_{V_j} h_j = \operatorname{sech}^2(V_j x)x^T$ [1 × 785]

We can also use backpropagation to show:

$\nabla_{W_j} L = \left(\frac{1-y_j}{1-z_j} - \frac{y_j}{z_j}\right)\left(z_j(1-z_j)h^T\right)$ [1 × 201]

$\nabla_h L = \sum_{j=1}^{26} \left(\frac{1-y_j}{1-z_j} - \frac{y_j}{z_j}\right)\left(z_j(1-z_j)W_j^T\right)$ [201 × 1]

$\nabla_{V_j} L = (\nabla_h L)_j \operatorname{sech}^2(V_j x)x^T$ [1 × 785]

To enhance calculation efficiency, we can reduce some of these equations into matrices. First, let $\mathcal{Q}$ be the matrix

$$\mathcal{Q} = \begin{bmatrix} z_1(1-z_1)\left(\frac{1-y_1}{1-z_1} - \frac{y_1}{z_1}\right) \\ z_2(1-z_2)\left(\frac{1-y_2}{1-z_2} - \frac{y_2}{z_2}\right) \\ \vdots \\ z_{26}(1-z_{26})\left(\frac{1-y_{26}}{1-z_{26}} - \frac{y_{26}}{z_{26}}\right) \end{bmatrix} = \begin{bmatrix} z_1 - y_1 \\ z_2 - y_2 \\ \vdots \\ z_{26} - y_{26} \end{bmatrix}$$

With this, we can reexpress the above equations.
(1)
$\nabla_{W_j} L = \mathcal{Q}_j h^T \qquad [1 \times 201]$
or

$$\nabla_W L = \begin{bmatrix} \mathcal{Q}_1 h^T \\ \mathcal{Q}_2 h^T \\ \vdots \\ \mathcal{Q}_{26} h^T \end{bmatrix} = \mathcal{Q} h^T = \mathcal{Q} \otimes h \qquad [26 \times 201]$$

(2)
$\nabla_h L = \sum_{j=1}^{26} \mathcal{Q}_j W_j^T \qquad [201 \times 1]$
or

$$\nabla_h L = \left[\mathcal{Q}_1 W_1^T + \mathcal{Q}_2 W_2^T + ... + \mathcal{Q}_{26} W_{26}^T\right]$$
$$\nabla_h L = \left[W_1^T \mathcal{Q}_1 + W_2^T \mathcal{Q}_2 + ... + W_{26}^T \mathcal{Q}_{26}\right]$$
$$\nabla_h L = \begin{bmatrix} W_1^T & W_2^T & \dots & W_{26}^T \end{bmatrix} \begin{bmatrix} \mathcal{Q}_1 \\ \mathcal{Q}_2 \\ \vdots \\ \mathcal{Q}_{26} \end{bmatrix} = W^T \mathcal{Q} \qquad [201 \times 1]$$

(3)
Additionally, let

$$\mathcal{S} = \begin{bmatrix} \operatorname{sech}^2(V_1 x) \\ \operatorname{sech}^2(V_2 x) \\ \vdots \\ \operatorname{sech}^2(V_{200} x) \end{bmatrix} = \operatorname{sech}^2(Vx)$$

Then, $\nabla_{V_j} L = (W^T \mathcal{Q})_j \operatorname{sech}^2(V_j x) x^T \qquad [1 \times 785]$
or

$$\nabla_V L = \begin{bmatrix} (W^T \mathcal{Q})_1 \operatorname{sech}^2(V_1 x) x^T \\ (W^T \mathcal{Q})_2 \operatorname{sech}^2(V_2 x) x^T \\ \vdots \\ (W^T \mathcal{Q})_{200} \operatorname{sech}^2(V_{200} x) x^T \end{bmatrix} = \begin{bmatrix} (W^T \mathcal{Q})_1 \operatorname{sech}^2(V_1 x) \\ (W^T \mathcal{Q})_2 \operatorname{sech}^2(V_2 x) \\ \vdots \\ (W^T \mathcal{Q})_{200} \operatorname{sech}^2(V_{200} x) \end{bmatrix} x^T$$

$$\nabla_V L = \left(\begin{bmatrix} (W^T \mathcal{Q})_1 \\ (W^T \mathcal{Q})_2 \\ \vdots \\ (W^T \mathcal{Q})_{200} \end{bmatrix} \circ \begin{bmatrix} \operatorname{sech}^2(V_1 x) \\ \operatorname{sech}^2(V_2 x) \\ \vdots \\ \operatorname{sech}^2(V_{200} x) \end{bmatrix} \right) x^T = (W^T \mathcal{Q}) \circ \mathcal{S} x^T$$

Since we are updating our matrices V and W using stochastic gradient descent, we repeat the following process:

$V, W \leftarrow$ weight matrices initialized randomly from normal distribution with mean $\mu = 0$ and $\sigma^2 = (...)$
while (continue = True or $L(z) > 0$)
    Forward calculation [$h = \tanh(Vx) \quad \rightarrow \quad z = s(Wh) \quad \rightarrow \quad L(z)$]
    Backward calculation to return $\nabla_V L$ and $\nabla_W L$
    $V \leftarrow V - \epsilon \nabla_V L)$
    $W \leftarrow W - \epsilon \nabla_W L$
return V, W

where, using our derived equations from above, the update rules are more specifically

$$V \leftarrow V - \epsilon(W^T \mathcal{Q}) \circ \mathcal{S} x^T$$
$$W \leftarrow W - \epsilon(\mathcal{Q} \otimes h).$$

Problem 2

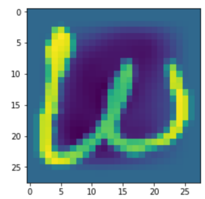
Implemenation & Results:

1) **Hyperparameters**
   Manually tested values of learning rate ϵ between 1 and and 1×10^{-5}. Settled on 0.1 as an optimal value.
2) **Training Accuracy**
   88.94% training accuracy was achieved after 20 epochs of stochastic gradient descent training.
3) **Validation Accuracy**
   86.63% validation accuracy was achieved on the 20% of data that was initially withheld.
4) **Loss vs. Iterations**
   (...)
5) **Kaggle**
   Display Name: mitch   Score: 0.87106
6) **Code**
   See appendix for full implementation in NeuralNet class

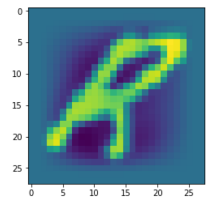
Problem 3

Visualization

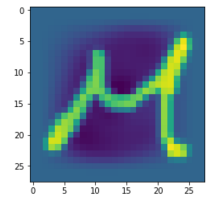
Correctly classified images:



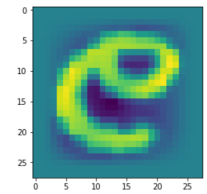
Guess: w
Actually: w



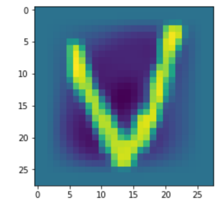
Guess: r
Actually: r



Guess: m
Actually: m

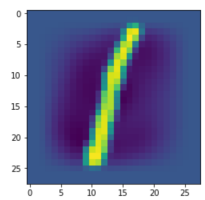


Guess: e
Actually: e

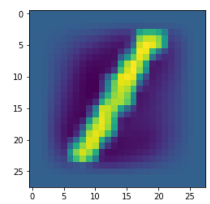


Guess: v
Actually: v

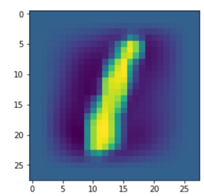
Incorrectly classified images:



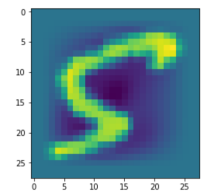
Guess: l
Actually: i



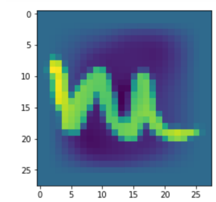
Guess: l
Actually: i



Guess: i
Actually: l



Guess: r
Actually: s



Guess: u
Actually: m

Problem 4

No extra bells and whistles were tested, though the neural network was designed to be flexible. The code currently operates using stochastic gradient descent, though it is designed to be able to take in multiple points (or the complete dataset) for easy conversion to batch/mini-batch gradient descent.

Furthermore, the modular structure allows the neural network to be adapted for any number of layers and any number of hidden units per layer. To do this, the code takes in the following sequences as arguments: layers, number of hidden units per layer, and activation functions per layer. Two modules are necessary to run this code, one containing activation functions to be included in the input sequence, and one containing a "gradient calculator" class which computes appropriate gradients for the desired neural network configuration.

Appendix

```python
"""
neuralnet.py
===============================================
Train a neural network and predict on test data
===============================================

This python module contains the neural network class, which can be trained on
labeled data (input as numpy arrays: for data, an Nxd matrix with N rows
corresponding to N sample points and d columns corresponding to d features; for
labels, an N vector with labels corresponding to each of the N sample points)

**NOTE: Use of this class requires the activationfns.py and gradients.py modules
"""

import numpy as np


class NeuralNet:
    """
    Train and store a neural network, based on supplied training data.
    Use this network to predict classifications.
    """

    def __init__(self,nlayers=3,unitsperlayer=None,actfns=[af.sigmoid,af.sigmoid],Gradients=None,verbose=False):
        """
        Initialize the neural network
        - nlayers:       the number of layers in the neural network (includes input and output layers)
        - unitsperlayer: a list specifying (in order) the number of units in all sequential layers except input
        - actfns:        a list specifying (in order) the activation function used by all sequential layers except input
        - Gradient:      a class providing optimized gradient calculations for the given sequence of
                         activation functions
        - verbose:       a boolean for descriptive output
        """
        if unitsperlayer == None:
            unitsperlayer = 3*np.ones(nlayers)
        elif nlayers == len(unitsperlayer)+1:
            self.nlayers = nlayers-1
            self.unitsperlayer = unitsperlayer
        elif nlayers > len(unitsperlayer)+1:
            print('ERROR: The number of units per layer were not given for at least one layer.')
        elif nlayers < len(unitsperlayer)+1:
            print('ERROR: More layers were given units than were specified by input "nlayers".')
        if nlayers == len(actfns)+1:
            self.actfns = actfns
        elif nlayers > len(actfns)+1:
            print('ERROR: The activation function was not given for at least one layer.')
        elif nlayers < len(actfns)+1:
            print('ERROR: More activation functions were provided than specified by input "nlayers".')
        if Gradients == None:
            print('ERROR: A gradient generator class must be included.')
        self.gradients = Gradients
        self.weight_matrices = []


    def initialize_weights(self,shape,mu=0,var=1):
        """
        Initialize weight matrix from normal distribution.
        - shape: tuple specifying desired shape of weight matrix
        - mu:    mean value of normal distribution
        - var:   variance of normal distribution
        """
        weight_matrix = np.random.normal(loc=mu,scale=np.sqrt(var),size=shape)

        return weight_matrix


    def weight_matrix_shape(self,layer_n,nfeatures):
        """
        Create weight matrix with the proper number of rows and columns for this layer
        - n:         the layer which will employ an activation function on the
                     product of the weight matrix and values
        - nfeatures: an integer specifying the number of features in the dataset
        """
        if layer_n != 0 and layer_n != range(self.nlayers)[-1]:
            WM_nrows = self.unitsperlayer[layer_n]-1
            WM_ncols = self.unitsperlayer[layer_n-1]
        elif layer_n == 0:
            WM_nrows = self.unitsperlayer[layer_n]-1
            WM_ncols = nfeatures
        else:
            WM_nrows = self.unitsperlayer[layer_n]
```

```python
            WM_ncols = self.unitsperlayer[layer_n-1]
        return WM_nrows,WM_ncols


    def forward(self,data):
        """
        Perform forward pass through neural network by multiplying data by weights
        and enforcing a nonlinear activation function for each layer.
        - data:           Nxd numpy array with N sample points and d features
        - weightmatrices: ordered list of sequential weight matrices corresponding to layers
        - actfns:         ordered list of sequential activation functions corresponding to layers
                         (functions are defined in activationfuncs.py)
        Returns layeroutputs, a list of the outputs from each layer. The last entry
        is an CxN numpy array with hypotheses for each sample N_i being in class C_j.
        """
        H = data.T
        layeroutputs = []
        for i in range(self.nlayers):
            W = self.weight_matrices[i]
            actfn = self.actfns[i]
            H = actfn(np.dot(W,H))
            # If the layer is not the output layer, add a fictitious unit for bias terms
            if i != self.nlayers-1:
                fictu = np.array([np.ones_like(H[0])])
                H = np.concatenate((H,fictu),axis=0)
            layeroutputs.append(H)
        return layeroutputs


    def backward(self,layeroutputs,labelrange,gradients=None):
        """
        Perform backward pass through neural network by computing gradients of
        input weight matrices with respect to the loss function comparing hypotheses
        to true values. Classes for gradients are provided in gradients.py module
        (a unique gradient class is required for neural networks with different
        numbers of layers and/or different activation functions)
        """
        if gradients == None:
            Gradients = self.gradients
        gradients = Gradients.calculate(self.weight_matrices,layeroutputs,labelrange)

        return gradients


    def classify_outputs(self,finaloutputs):
        """
        Convert final outputs into classifications
        -finaloutputs: a CxN numpy array with hypotheses for each sample N_i being in
                       class C_j.
        Returns a 1D, length-N array with values corresponding to point classifications
        """
        if len(finaloutputs) == 1:
            classifications = np.around(finaloutputs[0]).astype(int)
        if len(finaloutputs) > 1:
            # Add one for 1-indexing in classification labels
            classifications = (np.argmax(finaloutputs,axis=0)+np.ones(len(finaloutputs[0]))).astype(int)
        return classifications


    def train(self,data,labels,epsilon=0.1):
        """
        Train the neural network on input data
        - data:   Nxd numppy array with N sample points and d features
        - labels: 1D, length-N numpy array with labels for the N sample points
        """
        # Ensure labels are integers and that data and labels are the same length
        labels = labels.astype(int)
        if len(data) != len(labels):
            print('ERROR: Data and labels must be the same length.')

        # Add fictitious unit for bias terms
        fictu = np.array([np.ones(len(data))]).T
        data = np.concatenate((data,fictu),axis=1)

        # Initialize Weights
        nfeatures = len(data[0])
        for layer_n in range(self.nlayers):
            WM_nrows,WM_ncols = self.weight_matrix_shape(layer_n,nfeatures)
            # Variance of weight matrix determined by fan-in (eta), the number of units in the previous layer
            # (or the number of data features when initializing the first weight matrix)
            eta = WM_ncols
            weight_matrix = self.initialize_weights((WM_nrows,WM_ncols),mu=0,var=(1/eta))
            self.weight_matrices.append(weight_matrix)
```

```
        # Begin loop
        epochcounter = 0
        while epochcounter < 20:
            # Stochastic gradient descent: Loop over points randomly, one at a time
            # (Execute gradient class overhead before beginning)
            self.gradients.prepare(data,labels,self.unitsperlayer[-1])

            for datapoint_i in range(len(data)):
                X_i = np.array([data[datapoint_i]])
                layeroutput_i = self.forward(X_i)

                gradients = self.backward(layeroutput_i,[datapoint_i,datapoint_i+1])

                for n in range(self.nlayers):
                    self.weight_matrices[n]=self.weight_matrices[n]-epsilon*gradients[n]


            epochcounter+=1
            DL = np.concatenate((data,labels),axis=1)
            np.random.shuffle(DL)
            data = DL[:,:-1]
            labels = np.array([DL[:,-1]]).T
            epsilon *=0.75


    def predict(self,testdata):
        """
        Predict classfications for unlabeled data points using the previously
        trained neural network.
        - testdata: Nxd numpy array with N sample points and d features
                    *Note, dimension d must match that used for the data array in NeuralNet.train*
        Returns a 1D, length-N numpy array of predictions (one prediction per point)
        """

        # Add fictitious unit to input to match dimensions
        fictu = np.array([np.ones(len(testdata))]).T
        testdata = np.concatenate((testdata,fictu),axis=1)

        npoints = len(testdata)
        predictions = np.empty(npoints)
        layeroutputs = self.forward(testdata)
        predictions = self.classify_outputs(layeroutputs[-1])


        return predictions.astype(int)
```

```
"""
gradients.py
=========================================
Python module containing Gradient classes
=========================================

This module contains gradient classes which calculate gradients for neural
net backpropagation quickly. A new, unique gradient class must be constructed
for neural networks with different numbers of layers, and/or different orderings
of activation functions.
(New gradient classes do not need to be created for neural networks that only differ
in the number of units per layer.)
"""

import numpy as np
import activationfns as af


class tanhsig2layer:
    """
    Gradient class for a two layer neural network. The first layer employs a tanh
    activation function, the second layer employs a sigmoid activation function, and the
    neural network uses a cross-entropy loss function.
    """

    def __init__(self,verbose=False):
        self.V = None
        self.W = None
        self.h = None
        self.z = None
        self.X = None
        self.y = None
        self.grad_WL = None
        self.grad_VL = None
        self.verbose = verbose


    def prepare(self,data,labels,noutunits):
        self.X = data
        self.y = np.zeros((len(labels),noutunits))
        for l in range(len(labels)):
            self.y[l,int(labels[l,0])-1] += 1


    def calculate(self,weight_matrices,layeroutputs,labelrange):
        c = 0
        for listset in [weight_matrices,layeroutputs]:
            if len(listset) != 2:
                if len(listset) > 2:
                    estring = 'More'
                elif len(listset) < 2:
                    estring = 'Less'
                if not c:
                    lstring = 'weight␣matrices'
                else:
                    lstring = 'layer␣outputs'
                print('ERROR:␣This␣gradient␣is␣for␣a␣two-layer␣neural␣network.␣%s␣than␣two␣%s␣were␣provided.' %(estring,lstri
                return
            c+=1

        self.V = weight_matrices[0]
        self.W = weight_matrices[1]
        self.h = layeroutputs[0]
        self.z = layeroutputs[1]

        rangemin,rangemax = labelrange[0],labelrange[1]
        X = self.X[rangemin:rangemax]
        y = self.y[rangemin:rangemax]
        Q = self.z - y.T
        n = len(y)

        self.grad_VL = np.zeros_like(self.V)
        self.grad_WL = np.zeros_like(self.W)
        for i in range(n):
            S = np.array([1/np.square(np.cosh(np.dot(self.V,X[i])))]).T
            X_iTranspose = np.array([X[i]])
            self.grad_VL += np.dot((np.dot(self.W.T,Q)[:-1]*S),X_iTranspose)
            self.grad_WL += np.outer(Q,self.h[:,i])

        # Optional output of intermediate steps
        if self.verbose:
            print('V\n',self.V)
            print('W\n',self.W)
            print('h\n',self.h)
```

```
            print('z\n',self.z)
            print('Q\n',Q)
            print('S\n',S)
            print('X\n',X)
            print('Grad_VL\n',self.grad_VL)
            print('Grad_WL\n',self.grad_WL)

        return self.grad_VL,self.grad_WL
```

```
"""
activationfns.py
=============================================
Python module containing activation functions
=============================================

This module contains a variety of activation functions which can be imported
into a neural network using the NeuralNet class contained in neuralnet.py.
Each function takes at least one numpy array as an argument and returns a
numpy array.
"""

import numpy as np
import scipy.special as spsp

def sigmoid(x):
    """sigmoid function: s = 1/[1+e^(-x)]"""
    s = spsp.expit(x)

    return s


def tanh(x):
    """tanh function: s = [e^(x)-e^(-x)]/[e^(x)+e^(-x)]"""
    y = np.tanh(x)

    return(y)
```