HOMEWORK 3: NEURAL NETWORKS

10-701 Introduction to Machine Learning
(PhD) (Spring 2021)
Carnegie Mellon University
OUT: March 17, 2021[*]
DUE: March 31, 2021 11:59 PM

START HERE: Instructions

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 2.1"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information:
  `https://www.cs.cmu.edu/~aarti/Class/10701_Spring21/index.html`
- **Extension Policy:** See the homework extension policy here:
  `https://www.cs.cmu.edu/~aarti/Class/10701_Spring21/index.html`
- **Submitting your work:**
  - All portions of the assignments should be submitted to Gradescope (`https://gradescope.com/`).
  - **Programming:** We will autograde your Python code in Gradescope. After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). You have a **total of 30 Gradescope submissions.** Use them wisely. In order to not waste Gradescope submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly before any submission. Our autograder requires that you write your code using Python 3.6.9 and Numpy 1.17.0.
  - **Written questions:** You must type the answers in the provided .tex file. Handwritten solutions will not be accepted. Make sure to answer each question in the provided box. DO NOT change the size of the boxes, because it may mess up the autograder. Upon submission, make sure to label each question using the template provided by Gradescope. Please make sure to assign ALL pages corresponding to each question.

[*]Compiled on Saturday 3rd April, 2021 at 01:02

1 Neural Nets: Written Questions [15 points]

Note: We strongly encourage you to do the written part of this homework first, as it will help you gain familiarity with the calculations you will have to code up in the programming section. We suggest that for each of these problems, you write out the equation required to calculate each value in terms of the variables we created (a_j, z_j, b_k, etc.) before you calculate the numerical value.

Note: For all questions which require numerical answers, round up your final answers to four decimal places. Example: $0.12345 \rightarrow 0.1235$. For integers, you may drop trailing zeros. Example: $1.00001 \rightarrow 1$

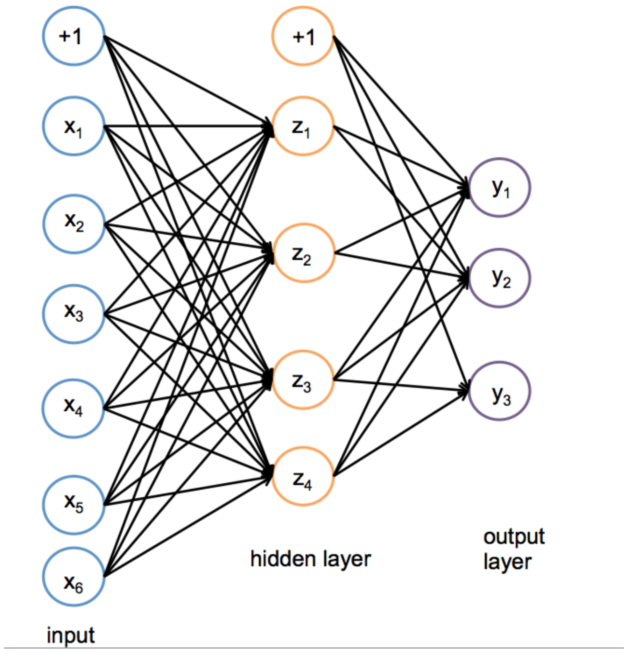


Figure 1.1: A One Hidden Layer Neural Network

Network Overview

Consider the neural network with one hidden layer shown in Figure 1.1. The input layer consists of 6 features $\mathbf{x} = [x_1, ..., x_6]^T$, the hidden layer has 4 nodes $\mathbf{z} = [z_1, ..., z_4]^T$, and the output layer is a probability distribution $\mathbf{y} = [y_1, y_2, y_3]^T$ over 3 classes. We also add a bias to the input, $x_0 = 1$ and the hidden layer $z_0 = 1$, both of which are fixed to 1.

We adopt the following notation:

1. Let $\boldsymbol{\alpha}$ be the matrix of weights from the inputs to the hidden layer.
2. Let $\boldsymbol{\beta}$ be the matrix of weights from the hidden layer to the output layer.
3. Let $\alpha_{j,i}$ represent the weight going *to* the node z_j in the hidden layer *from* the node x_i in the input layer (e.g. $\alpha_{1,2}$ is the weight from x_2 to z_1)
4. Let $\beta_{k,j}$ be the weight going *to* the node y_k in the output layer *from* the node z_j in the hidden layer.
5. We will use a *sigmoid activation function* (σ) for the hidden layer and a *softmax* for the output layer.

Network Details

Equivalently, we define each of the following.

The input:

$$\mathbf{x} = [x_1, x_2, x_3, x_4, x_5, x_6]^T \tag{1.1}$$

Linear combination at first (hidden) layer:

$$a_j = \alpha_0 + \sum_{i=1}^{6} \alpha_{j,i} x_i, \quad j \in \{1, \dots, 4\} \tag{1.2}$$

Activation at first (hidden) layer:

$$z_j = \sigma(a_j) = \frac{1}{1 + \exp(-a_j)}, \quad j \in \{1, \dots, 4\} \tag{1.3}$$

Linear combination at second (output) layer:

$$b_k = \beta_0 + \sum_{j=1}^{4} \beta_{k,j} z_j, \quad k \in \{1, \dots, 3\} \tag{1.4}$$

Activation at second (output) layer:

$$\hat{y}_k = \frac{\exp(b_k)}{\sum\limits_{l=1}^{3} \exp(b_l)}, \quad k \in \{1, \dots, 3\} \tag{1.5}$$

Note that the linear combination equations can be written equivalently as the product of the weight matrix with the input vector. We can even fold in the bias term α_0 by thinking of $x_0 = 1$, and fold in β_0 by thinking of $z_0 = 1$.

Loss

We will use cross entropy loss, $\ell(\hat{\mathbf{y}}, \mathbf{y})$. If $\mathbf{y}$ represents our target (true) output, which will be a one-hot vector representing the correct class, and $\hat{\mathbf{y}}$ represents the output of the network, the loss is calculated as:

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{3} y_k \log(\hat{y}_k) \tag{1.6}$$

1. **[1.5 points]** In the following questions you will derive the matrix and vector forms of the previous equations which define our neural network. These are what you should hope to program in order to avoid excessive loops and large run times. When working these out it is important to keep a note of the vector and matrix dimensions in order for you to easily identify what is and isn't a valid multiplication.

   Suppose you are given the training example $\mathbf{x}^{(1)} = [x_1, x_2, x_3, x_4, x_5, x_6]^T$ with label **class 3**, so $\mathbf{y}^{(1)} = [0, 0, 1]^T$. We initialize the network weights as:

$$\boldsymbol{\alpha}^* = \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \end{bmatrix}$$

$$\boldsymbol{\beta}^* = \begin{bmatrix} \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} \\ \beta_{2,1} & \beta_{2,2} & \beta_{2,3} & \beta_{2,4} \\ \beta_{3,1} & \beta_{3,2} & \beta_{3,3} & \beta_{3,4} \end{bmatrix}$$

To account for the weights on the bias term in our weight matrices, we can add a new column to the beginning of $\boldsymbol{\alpha}^*$ and $\boldsymbol{\beta}^*$. So, we define

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,0} & \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \end{bmatrix}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} \\ \beta_{2,0} & \beta_{2,1} & \beta_{2,2} & \beta_{2,3} & \beta_{2,4} \\ \beta_{3,0} & \beta_{3,1} & \beta_{3,2} & \beta_{3,3} & \beta_{3,4} \end{bmatrix}$$

Accordingly, we can also set the first element of our input vectors to be always 1 so that our input becomes:

$$\mathbf{x}^{(1)} = [1, x_1, x_2, x_3, x_4, x_5, x_6]^T$$

(a) **[0.5 points]** What is the vector $\mathbf{a}$ defined as per equation (1.2) in terms of α and $\mathbf{x}^{(1)}$?

$a = \alpha \cdot \mathbf{x}^{(1)}$

(b) **[0.5 points]** What is the vector $\mathbf{z}$ defined as per equation (1.3) in terms of $\mathbf{a}$?

$z = \sigma(a) = \frac{1}{1+\exp(-a)}$

(c) **[0.5 points]** Notice that $\boldsymbol{\beta}$ and $\mathbf{z}$ are of incompatible dimensions to take a matrix multiplication for defining $\mathbf{b}$ as per equation (1.4). Which of the following can solve this problem to allow us to define $\mathbf{b} = \boldsymbol{\beta} * \mathbf{z}$?

- ○ Take the transpose of $\boldsymbol{\beta}$
- ○ Take the transpose of $\mathbf{z}$
- ○ Append a row of 1's to be the first row of $\boldsymbol{\beta}$
- [✓ Append a value of 1 to be the first element of $\mathbf{z}$]
- ○ Append a column of 1's to be the first column of $\boldsymbol{\beta}$

2. **[4.5 points]** We will now derive the matrix and vector forms for the backpropagation algorithm. In doing these computations, you should always be examining the shapes of the matrices and vectors and comparing your matrix elements with calculations of individual derivatives to make sure they match. Recall that ℓ is our loss function defined in equation (1.6).

   (a) **[0.5 points]** Compute the derivative $\frac{d\ell}{db_k}$ and express it in the most simplified form in terms of $y_k, \hat{y}_k$ only.

   $\frac{d\ell}{db_k} = \hat{y}_k - y_k$

   (b) **[0.5 points]** What is the derivative $\frac{dl}{d\boldsymbol{\beta}}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{b}}$ and $\mathbf{z}$.

   $\frac{dl}{d\boldsymbol{\beta}} = \frac{d\ell}{d\mathbf{b}} \cdot z^T$

   (c) **[0.5 points]** What are the dimensions of $\frac{dl}{d\boldsymbol{\beta}}$?

   3 * 5

   (d) **[1 points]** Explain in 1 short sentence why the computation of $\frac{d\ell}{d\boldsymbol{\alpha}}$ should involve $\boldsymbol{\beta}^*$ instead of $\boldsymbol{\beta}$.

   This is because we append the bias term to z ourselves, which means the value of the bias term does not depend on α, so the weight of the bias term won't be able to propagate.

   (e) **[0.5 points]** What is the derivative $\frac{d\ell}{d\mathbf{z}}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{b}}$ and $\boldsymbol{\beta}^*$.

   $\frac{d\ell}{d\mathbf{z}} = (\boldsymbol{\beta}^*)^T \cdot \frac{d\ell}{d\mathbf{b}}.$

   (f) **[0.5 points]** What is the derivative $\frac{d\ell}{d\mathbf{a}}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{z}}$ and $\mathbf{z}$.

   $\frac{d\ell}{d\mathbf{a}} = z * (1 - z) * \frac{d\ell}{d\mathbf{z}}$
   ($*$ means element-wise multiplication)

   (g) **[0.5 points]** What is the derivative $\frac{d\ell}{d\boldsymbol{\alpha}}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{a}}$ and $\mathbf{x}^{(1)}$.

   $\frac{d\ell}{d\boldsymbol{\alpha}} = \frac{d\ell}{d\mathbf{a}} \cdot (\mathbf{x}^{(1)})^T$

   (h) **[0.5 points]** What are the dimensions of $\frac{d\ell}{d\boldsymbol{\alpha}}$?

   4 * 7

Prediction

When doing prediction, we will predict the argmax of the output layer. For example, if $\hat{\mathbf{y}}$ is such that $\hat{y}_1 = 0.3,\ \hat{y}_2 = 0.2,\ \hat{y}_3 = 0.5$ we would predict class 3 for the input $\mathbf{x}$. If the true class from the training data $\mathbf{x}$ was 2 we would have a one-hot vector $\mathbf{y}$ with values $y_1 = 0,\ y_2 = 1,\ y_3 = 0$.

3. **[3 points]** Note that unless otherwise mentioned, we are defining our weight matrices to include the bias weights in the first column. When we say training example, it has not been augmented to append 1 as the first element to deal with the bias term. Ensure that you check that the dimensions match as per your expectations. If needed, you may write your own code to compute the answers to the following questions but you do not have to submit it.

   We initialize the weights as:

$$\boldsymbol{\alpha} = \begin{bmatrix} 1 & 1 & 2 & -3 & 0 & 1 & -3 \\ 1 & 2 & 1 & 1 & 1 & 0 & 2 \\ 1 & 3 & 2 & 2 & 2 & 2 & 1 \\ 1 & 2 & 0 & 3 & 1 & -2 & 2 \end{bmatrix}$$

$$\boldsymbol{\beta} = \begin{bmatrix} 1 & 1 & 2 & -2 & 3 \\ 1 & 2 & -1 & 3 & 1 \\ 1 & 3 & 1 & -1 & 1 \end{bmatrix}$$

   You are given a training example $\mathbf{x}^{(1)} = [1, 0, 1, 0, 1, 1]^T$ with label class 3, so $\mathbf{y}^{(1)} = [0, 0, 1]^T$. Using the initial weights, run the feed forward of the network over this training example (without rounding during the calculation) and then answer the following questions.

   (a) **[0.5 points]** What is the value of a_1?

   -3

   (b) **[0.5 points]** What is the value of z_1? (after rounding properly as instructed)

   0.0474

   (c) **[0.5 points]** What is the value of b_2? (after rounding properly as instructed)

   4.0945

   (d) **[0.5 points]** What is the value of $\hat{y}_2$? (after rounding properly as instructed)

   0.4799

   (e) **[0.5 points]** Which class value (1 or 2 or 3) we would predict on this training example?

   2

(f) **[0.5 points]** What is the value of the total loss (1.6) on this training example? (after rounding properly as instructed)

$\boxed{2.6912}$

4. **[2 points]** Now use the results of the previous question to run backpropagation over the network and update the weights. Use the learning rate $\eta = 1$.

   Do your backpropagation calculations first without any rounding then answer the following questions after rounding to four decimal places:

   (a) **[0.5 points]** What is the updated value of $\beta_{2,1}$? (after rounding properly as instructed)

   1.9772

   (b) **[0.5 points]** What is the updated weight of the hidden layer bias term applied to y_1 (i.e. $\beta_{1,0}$)? (after rounding properly as instructed)

   0.5477

   (c) **[0.5 points]** What is the updated value of $\alpha_{3,4}$? (after rounding properly as instructed)

   2

   (d) **[0.5 points]** If we ran backpropagation on this example for a large number of iterations and then ran feed forward over the same example again, which class (1 or 2 or 3) would we predict?

   3

5. **[4 points]** Let us now introduce regularization into this neural network. For this question, we will incorporate L2 regularization into our loss function $\ell(\hat{\mathbf{y}}, \mathbf{y})$, with the parameter λ controlling the weight given to the regularization term.

   (a) **[1 points]** Write the most simplified expression for the regularized loss function of our network after adding L2 regularization (**Hint:** Remember that bias terms should not be regularized!). Please use $\ell(\hat{\mathbf{y}}, \mathbf{y})$ to indicate the non-regularized loss function.

   $\ell(\hat{\mathbf{y}}, \mathbf{y}) + \lambda(||\alpha^*||_2^2 + ||\beta^*||_2^2)$
   $= \ell(\hat{\mathbf{y}}, \mathbf{y}) + \lambda(\sum_{j,i}(\alpha^*_{j,i})^2 + \sum_{l,k}(\beta^*_{l,k})^2)$

   (b) **[0.5 points]** Compute the regularized loss for training example $\mathbf{x}^{(1)}$ (assume λ = 0.01 and use the weights before backpropagation). Write your answer after rounding properly as instructed.

   3.9712

   (c) **[0.5 points]** For a network which uses the regularized loss function, write the gradient update equation for $\alpha_{j,i}$ as $\alpha_{j,i} = \alpha_{j,i} - \eta \times G$. What is the expression for G? You should use $\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \alpha_{j,i}}$ to denote the gradient update w.r.t non-regularized loss and η to denote the learning rate.

   $\alpha_{j,i} = \alpha_{j,i} - \eta \times (\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \alpha_{j,i}} + 2\lambda\alpha_{j,i})$

(d) **[2 points]** Based on your observations from previous questions, **select all statements which are true**:

☐ The non-regularized loss is always higher than the regularized loss

■ As weights become larger, the regularized loss increases faster than non-regularized loss

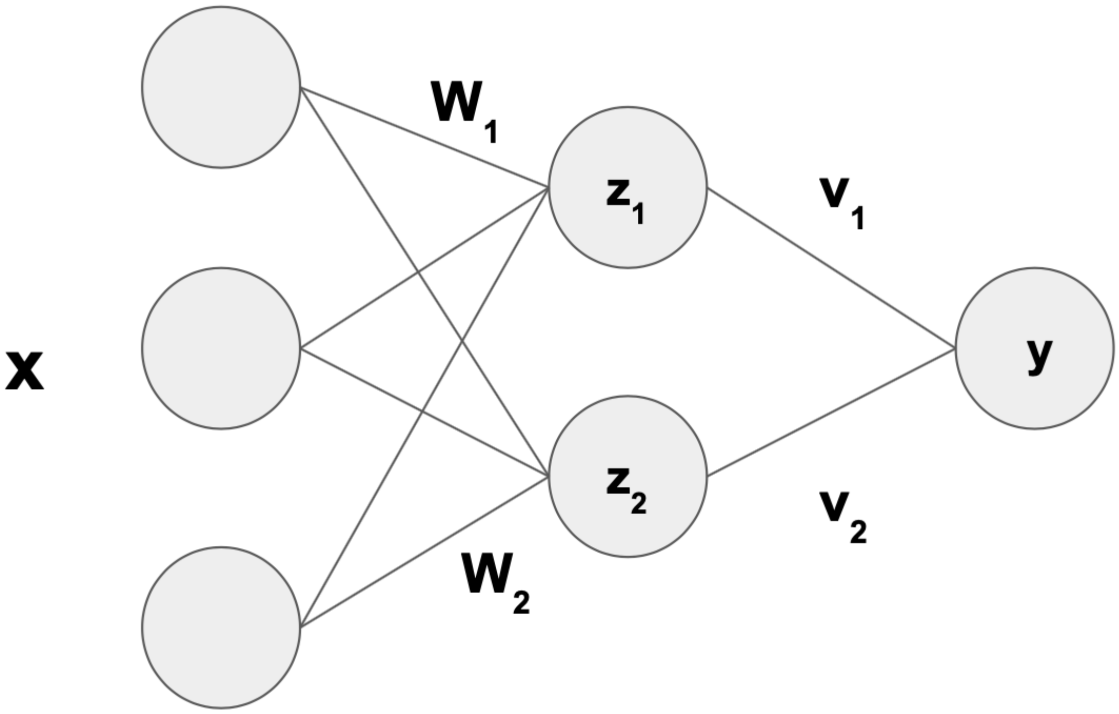
☐ On adding regularization to the loss function, gradient updates for the network always become larger

■ When using large initial weights, weight values decrease more rapidly for a network which uses regularized loss

☐ None of the above

2 Effect of initialization [5 points]

1. **[5 points]** Let us now study the effect of initialization of weights on the learning ability of neural networks. For this problem, consider a 2 layer neural network as shown below.



$\mathbf{x}$ is a 3×1 input vector, $\mathbf{w}_1$ and $\mathbf{w}_2$ are 3×1 weight vectors such that $z_1 = \mathbf{x}^T\mathbf{w}_1$ and $z_2 = \mathbf{x}^T\mathbf{w}_2$. The output $y = v_1\sigma(z_1) + v_2\sigma(z_2)$. Suppose we have a loss function ℓ and $\frac{d\ell}{dy}$ is well defined. Wherever needed, assume $\frac{d\ell}{dy} \neq 0$.

(a) We initialize $\mathbf{w}_1 = \mathbf{0}$, $\mathbf{w}_2 = \mathbf{0}$, $v_1 = 0$ and $v_2 = 0$. We want to run backpropagation over the network for training. Assume $\eta = 1$.

After 1 iteration of backpropagation, what are the values of

i. **[0.5 points]** $\mathbf{w}_1 = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$

a:

0

b:

0

c:

0

ii. **[0.5 points]** v_2 (Your answer can be in terms of $\frac{d\ell}{dy}$)

$-\frac{1}{2}\frac{d\ell}{dy}$

After n iterations of backpropagation,

i. **[1 points]** how are $\mathbf{w}_1$ and $\mathbf{w}_2$ related?

○ $\mathbf{w}_1 > \mathbf{w}_2 > \mathbf{0}$

○ $\mathbf{w}_1 = \mathbf{w}_2 = \mathbf{0}$

[✓ $\mathbf{w}_1 = \mathbf{w}_2 \neq \mathbf{0}$]

○ $\mathbf{w}_1 < \mathbf{w}_2 < \mathbf{0}$

○ None of the above

ii. **[1 points]** how are v_1 and v_2 related?

○ $v_1 > v_2$

[✓ $v_1 = v_2$]

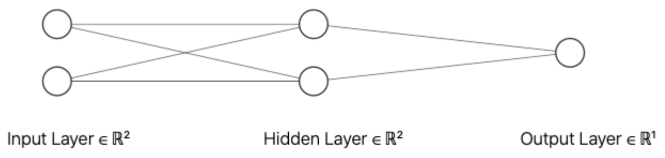
○ $v_1 < v_2$

[2 points]Is this an effective initialization strategy? Explain why or why not in not more than 2 sentences.

This is not an effective initialization strategy.
If we initialize all the weights with the same value, then they will always be equal and the model cannot learn a function where these weights are different.

3 Activation Functions and Learning Rates [10 pts]

Consider the below neural network architecture. The weight and bias terms of the input and hidden layer are denoted by $\mathbf{w}^{(1)}, b^{(1)}$, and $\mathbf{w}^{(2)}, b^{(2)}$ respectively, and the hidden layer has a ReLU activation.



Suppose this network is trained on the data set below according using a mean square error loss function

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

.

X_1	X_2	Y
-1	0.4	2
3	8	-4
-3	0.3	3.2
4	10	-8.3
-0.5	5	-2.3

Suppose we train this network for k iterations of full gradient descent, and on the kth iteration, the weight values for $\mathbf{w}^{(1)}, b^{(1)}$, $\mathbf{w}^{(2)}, b^{(2)}$ are as follows.

$$\mathbf{w}^{(1)} = \begin{pmatrix} 0.2 & 3.8 \\ -1.1 & 2.3 \end{pmatrix}$$
$$b^{(1)} = [-1.3, 0.86]$$
$$\mathbf{w}^{(2)} = [-1.2, 0.7]$$
$$b^{(2)} = 0$$

1. **[2 pt]**What is the loss for the kth iteration of the gradient descent?. Please round your answer to **4 numbers after the decimal point**

   209.4557

2. **[4 pts] Select one:** Suppose we want to try different activation functions to see if it yields better accuracy. Which of the following activation will minimize the loss of the kth iteration of gradient descent?:

   ○ ReLU

   [✓ ● tanh]

   ○ Sigmoid $\left(\sigma(x) = \frac{1}{1+e^{-x}}\right)$

○ Linear

3. **[4 pts]** Suppose we now have the ability to tune the learning rate to minimize the loss. Which of the following learning rates $\eta = \{0.1, 0.001, 0.0001, 0.9\}$ would minimize the loss function after 2 additional iterations of gradient descent? Assume ReLU activation is used.

$\boxed{\eta = 0.001}$

4 Miscellaneous [10 pts]

1. **[2 pt] True or False:** For any neural network, the training error must always decrease monotonically when using SGD, provided the step size is sufficiently small (assume the step size is constant).

   True

   [✓ False]

2. **[1 pt] True or False:** Convolutional Neural Networks (CNNs) can only be used with two-dimensional inputs such as images.

   True

   [✓ False]

3. **[1 pt]** If you answered False above, give an example of an application of CNNs to non 2D inputs. If you answered True, simply put N/A.

   CNN can take 3d data, such as RGB images, as input.

4. **[2 pt]** Consider the image X and filter F given below. Let X be convolved with F, padded with 0 such that the output Y is the same size of X, a stride of 1 to produce an output Y. What is value of J in the output Y?

X =

-1	0	-2	3	4	1
2	9	5	6	0	-1
0	-3	1	3	4	4
6	5	2	0	6	8
-5	4	-3	1	3	-2
4	1	2	8	9	7

F =

-2	-2	-2
-2	5	-2
-2	-2	-2

Y =

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z	Aa	Ab	Ac	Ad
Ae	Af	Ag	Ah	Ai	Aj

-6

5. **[2 pt]** Young wants to train a seq2seq model and first uses a vanilla RNN architecture with sigmoid activation in all the layers. However, after training he finds his model is suffering from the vanishing gradient problem. Explain what the vanishing gradient problem is and how it arises.

   Vanishing gradient problem is when the gradient of the loss function become extremely small, so we can hardly update the weights.
   The problem occurs because if the input values for sigmoid function are very large or very small, then the derivatives of sigmoid function will be close to 0.

6. **[2 pt]** Propose a change to Young's architecture to help with performance, while still letting Young use sigmoid activation in all the layers.

   Use LSTM instead of the vanilla RNN.

5 Neural Net Programming + Writing[60 points]

Figure 5.1: Random Images of Each of 10 digits in MNIST

Your goal in this assignment is to label images of handwritten digits (0 to 9) by implementing a neural network from scratch. You will implement all of the functions needed to initialize, train, evaluate, and make predictions with the network.

The MNIST dataset is comprised of 70,000 handwritten numerical digit images and their respective labels. There are 60,000 training images and 10,000 test images, all of which are 28 pixels by 28 pixels.

In this particular case, you will work with a smaller subset of MNIST that consist of 4000 images where the pixel values in each of them ranges from 0 to 1.

Note: Please look at the **tips** in the appendix (last page) about vectorization, gradient checking, and overflow/underflow concerns to make a successful implementation.

The programs you write will be automatically graded using Gradescope Autograder. You need to write your program using **Python 3** and **NumPy**. We recommend download Anaconda platform from Anaconda and choose your appropriate operating system. Gradescope will grade submissions using **Python 3.6** and **Numpy 1.17**.

Optional: for local development, any reasonably up-to-date versions of Python and Numpy should work. But you may also create your own Python environment. Follow the next steps in your **Terminal-Console** to create a Python 3.6 environment with Numpy 1.17:

1. Create the environment:
```
conda create -n ml10701s20_env -c conda-forge python=3.6.9 numpy=1.17.0
```
2. Activate your environment:
```
source activate ml10701s20_env
```

The commands above will create a Python environment called `ml10701s20_env` with the necessary packages you will need for this problem. When you are done working in your environment, you can deactivate it as follows:

- To end a session in the current environment:
  `source deactivate`

5.1 The Task and Datasets

What is included in the handout

- `hw3_sol.py`, `hw3_lib.py` are the files you are going to modify. These files are generated from the reference solution with some key parts removed, and you will need to fill in those parts. **Do not modify other files**.
- `training_data_student/` dataset. See the next paragraph.
- `run_tests.py` and `tests/` allows you to run part of the grading process to make sure your code will run on Gradescope. See Section 5.4.2.

Dataset format For the dataset stored under `training_data_student`, you are provided with 2 files `train.csv`, `test.csv` that contain the digits and their corresponding labels (0 through 9). Each row contains $784 + 1$ columns separated by commas. Columns 1 to 784 represent the pixel values and column 785 contains the corresponding label (0 to 9). Two files contains 3000 and 1000 instances respectively.

5.2 Model Definition

5.2.1 Preliminaries

In this section, you will implement a modular neural network framework with a choice of sigmoid or ReLU activation functions for the hidden layer, and a softmax on the output layer. For this particular problem, the input vectors $\mathbf{x}$ are of length $M = 28 \times 28$ (flattened to 1×784), the hidden layer $\mathbf{z}$ consist of D hidden units, and the output layer $\hat{\mathbf{y}}$ represents a probability distribution over the $K = 10$ classes. In other words, each element y_k of the output vector $\hat{\mathbf{y}}$ represents the probability of $\mathbf{x}$ belonging to the class k.

Following the notation from Section 1 we have:

- For the output layer:

$$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)}, \quad k \in \{1, \ldots, K\}, \quad \text{(softmax activation)}$$

$$b_k = \beta_{k,0} + \sum_{j=1}^{D} \beta_{kj} z_j, \quad k \in \{1, \ldots, K\}, \quad \text{(pre-activation)}$$

- To prevent overflow in exponential calculations, you can subtract

- For the hidden layer:

$$z_j = \frac{1}{1+\exp(-a_j)}, \quad j \in \{1,\dots,D\}, \quad (\sigma - \text{activation})$$

$$z_j = \begin{cases} a_j, a_j > 0 \\ 0, a_j \le 0 \end{cases}, \quad j \in \{1,\dots,D\}, \quad (\text{ReLU activation})$$

(You would only use one of the two activation functions for each network)

$$a_j = \alpha_{j,0} + \sum_{i=1}^{M} \alpha_{ji} x_i, \quad j \in \{1,\dots,D\}, \quad (\text{pre-activation})$$

Although it is possible to compactly express this model by assuming that $x_0 = 1$ is a bias feature on the input and that $z_0 = 1$ is also fixed, for performance reason, you should still implement the equations above with two separate variables holding the weights and biases.

Note: Since ReLU is not differentiable at $a_j = 0$, we will use its subderivative at $a_j = 0$ instead. For convenience, we take its subderivative at $a_j = 0$ as 0.

5.2.2 Objective function

Since the output corresponds to a probabilistic distribution over the K classes, the objective function (loss function) we will use for training our neural network is the average cross entropy loss,

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{B} \sum_{n=1}^{B} \sum_{k=1}^{K} y_k^{(n)} \log(\hat{y}_k^{(n)}) \tag{5.1}$$

over each batch of inputs,

$$\mathcal{D} = \{(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})\}, \quad \text{for } n \in \{1, \dots B\}$$

In Equation (5.1), J is a function of the model parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ because $\hat{y}_k^{(n)}$ is implicitly a function of $\mathbf{x}^{(n)}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ since it is the output of the neural network applied to $\mathbf{x}^{(n)}$. As before, $\hat{y}_k^{(n)}$ and $y_k^{(n)}$ present the k-th component of $\hat{\mathbf{y}}^{(n)}$ and $\mathbf{y}^{(n)}$ respectively.

To train the network, you should optimize this objective function using batch gradient descent, where the gradient of the parameters for each batch of training examples is computed via backpropagation.

5.2.3 Initialization

In order to train a deep network, we must first initialize the weights and biases in the network. This is typically done with a random initialization, or initializing the weights from some other training procedure. For this assignment, You implementation need to use the provided `init_params` function to initialize your weights and biases.

5.3 Implementation

Modify the provided `hw3_sol.py` and `hw3_lib.py` to implement the neural-network-based digits recognizer.

WHAT PARTS YOU SHOULD MODIFY Only modify parts with `### TYPE HERE AND REMOVE `pass` below ###`, **leave all other parts untouched** unless you really know what you are doing.

To proceed, you are provided with the following guide. This guide is to help you implement functions and classes that should be invoked in the four functions to be graded as described in Section 5.4. You can also ignore this guide and directly implement those four functions.

5.3.1 `Module`

The provided `hw_lib.py` implements neural network layers as different `Module`s with methods `forward` and `backward` to implement the forward computation process and the backpropagation gradient computation process. The structure of `Module` is inspired by the popular deep learning framework PyTorch. You are **not required** to install PyTorch.

By convention (which closely follows that of PyTorch), for each module, you should implement the following methods. **Check `module_tutorial.py` for some examples**.

1. `Module.__init__` initializes this module. The input arguments should be some hyperparameters (number of classes, number of hidden units, initialization schemes of trainable parameters, etc.) and (optionally) the initial values of trainable parameters (weight, bias, etc.).
2. `Module.forward` takes input NumPy `np.ndarray`(s) and generates the output `np.ndarray`(s) according the feedforward computation to be performed by this module. For example, for a linear layer module, if the input is a vector, then the output should also be a vector computed from the weight and bias parameters of this module. For a loss layer, the output should be a single number for the loss.
3. `Module.backward` updates gradients across the module for the back-propagation process.

   For a non-loss layer, `Module.backward` takes in the gradient of the loss w.r.t. the output in the **last** call of `Module.forward`; once the gradient w.r.t. the **last** `Module.forward` is there, the module should perform two things.

   - the gradients of the loss w.r.t. parameters are computed and reflected in `Module.d_weights` and `Module.d_bias`.For example:

     `Module.d_weights` = $\frac{\partial L}{\partial W}$

     for all trainable parameters W after calling `Module.backward`.
   - the gradients of the loss w.r.t. input `np.ndarray`(s) in the **last** call of `Module.forward` should be returned.

   As an example, for a linear layer that computes $\vec{y} = W\vec{x} + \vec{b}$, if we know $\frac{\partial L}{\partial \vec{y}}$ as input, after calling `Module.backward`, we should have

   - `Module.d_weights` = $\frac{\partial L}{\partial W}$.
   - `Module.d_bias` = $\frac{\partial L}{\partial \vec{b}}$.
   - $\frac{\partial L}{\partial \vec{x}}$ as return value.

   For a loss layer, which typically has no trainable parameters, `Module.backward` takes no input, and it returns the gradient of the loss computed in the **last** call of `Module.forward` w.r.t. input `np.ndarray`(s).

4. You can see that the behavior of `Module.backward` is dependent on both its own input and the last call of `Module.forward`. This means that you may need to store some information about the forward computation every time you call `Module.forward`. The variables are already defined for you.

5.3.2 Defining layers

First, we will implement four types of layers in `hw3_lib.py`: linear, sigmoid, ReLu and softmax cross entropy loss layers. Check comments in `hw3_lib.py` for additional guidance. Then we will implement a Gradient Descent Optimizer to apply the gradients

Sigmoid layer

1. Implement the provided function `sigmoid(x).`
2. Implement the provided module `Sigmoid`.

ReLU layer

1. Implement the provided module `ReLU`.

Linear layer

1. Implement the provided module `Linear`.

Softmax cross entropy loss layer

1. implement the provided module `CrossEntropyLoss`, which implements softmax followed by average cross entropy computation; the fusion of softmax and average cross entropy computation in one layer is common in popular deep learning frameworks.

GradientDescentOptimizer

1. Implement the `step` function to update weights using the calculated gradients

5.3.3 Defining your one layer network

After implementing all layers, now it's time to implement a one-hidden-layer neural network. Please modify the code of `define_network` in `hw3_sol.py`. The `MultiLayerPerceptron` class takes a list of layers.

5.3.4 Defining your network trainer

After implementing the network itself, now it's time to implement a trainer for it. Please modify the code around `train_network` in `hw3_sol.py`. The function `train_network` implements Batch Gradient Descent algorithm as specified in Appendix A plus some additional logging capability.

This function takes 6 inputs.

- `model` a `MultiLayerPerceptron` instance

- `dataset` a data set, in the form of Python dictionary. It has four keys `train_x`, `train_y`, `test_x`, `test_y`.
- `num_epoch` total number of epochs for training.
- `learning_rate` learning rate γ.
- `batch_size` size of each batch.
- `seed` controls the seed.

The main loop of Batch Gradient Descent is already implemented. You only need to compute some statistics (losses, errors, prediction labels) using the model obtained at the end of each epoch.

5.4 Autograding

Gradescope will grade the following functions in `hw3_sol.py` and `hw3_lib.py`. For input output specifications of these functions, see comments in the respective Python files.

- **[5 points]** Implement Linear layer.
- **[5 points]** Implement GradientDescentOptimizer.
- **[5 points]** Implement Sigmoid layer.
- **[5 points]** Implement ReLU layer.
- **[10 points]** Implement CrossEntropyLoss layer.
- **[20 points]** Implement `train_network` to train a neural network.

5.4.1 Tips for `numpy`

Optional arguments of most aggregation functions of `numpy`, such as `mean`, `argmax`, `sum`, etc:

- `axis=1` controls the 0-indexed axis on which the aggregation is performed.
- `keepdims=True` retains the aggregated dimension as a 1-element dimension. You can use the resulting `ndarray` to do targeted broadcast operation on a that specific axis. For example, A `(n, n) - (n, 1)` operation will subtract every element in the first row of the first array by the first number in the second array, second row by the second number and so on.

`np.argmax(a, axis)` or `a.argmax(axis)` returns the indices of the maximum values along an axis.

`np.where(condition, x, y)` return elements chosen from x or y depending on condition.

For complete documentations, please read the official numpy documentations.

5.4.2 TESTING YOUR CODE BEFORE SUBMISSION

THIS IS REALLY IMPORTANT. You must test your code by running `python run_tests.py` included in the handout before submitting to Gradescope. Note that the local tests only has a subset of the full tests on Gradescope, so passing local tests does not guarantee passing the autograder.

5.5 Programming Submission

You must submit a .zip file containing `hw3_sol.py` and `hw3_lib.py` (only these two files will be considered by the autograder. Make sure they are not in any sub-directories).

5.6 Written Questions [10pts]

1. For the following questions, train a network, observe the training accuracy and test accuracy after each epoch, as returned by the `train_network` function. You'll report the final statistics here. All numbers should be rounded to 4 decimal places without using scientific notation (e.g. 0.1234). **Keep the seed in the `define_network` function as the default value (10701)** and use the function to define the networks specified below.

   (a) Train a neural network with a single hidden layer of size 256 and Sigmoid activation, using learning rate of 0.01 and batch size of 50 for 50 epochs.

   Final (after the last epoch) training accuracy **[1 points]**

   0.8307

   Final (after the last epoch) test accuracy **[1 points]**

   0.82

   (b) Train a neural network with a single hidden layer of size 256 and Sigmoid activation, using learning rate of 0.001 and batch size of 50 for 50 epochs.

   Final training accuracy **[0.5 points]**

   0.7873

   Final test accuracy **[0.5 points]**

   0.782

   Comparing to (a), what is the impact of learning rate to the results? **[1 points]**

   For (b), the learning rate is too small, and it leads to lower training and testing accuracy.

   (c) Train a neural network with a single hidden layer of size 256 and ReLU activation, using learning rate of 0.01 and batch size of 50 for 50 epochs.

   Final training accuracy **[0.5 points]**

   0.933

   Final test accuracy **[0.5 points]**

   0.877

   Comparing to (a), what is the impact of a different activation function to the results? **[1 points]**

For this problem, ReLU leads to better training and testing accuracy.

(d) Train a neural network with a single hidden layer of size 256 and ReLU activation, using learning rate of 0.01 and batch size of 200 for 50 epochs.

Final training accuracy **[0.5 points]**

0.9083

Final test accuracy **[0.5 points]**

0.876

Comparing to (c), what is the impact of batch size to the results and why? (in a single sentence) **[1 points]**

A larger batch size leads to a smaller training accuracy, and a slightly smaller test accuracy, because for a large batch size we will consider more samples before we make an update.

(e) Train a neural network with two hidden layer of size 384 and 256 respectively and ReLU activations, using learning rate of 0.01 and batch size of 50 for 50 epochs.

Final training accuracy **[0.5 points]**

0.9067

Final test accuracy **[0.5 points]**

0.863

Comparing to (c), did a deeper network improve test accuracy? Explain the most probably reason in a single sentence. (Hint: we are using a small subset of MNIST data for training) **[1 points]**

No, the test accuracy did not improve for the deeper network.
The deeper network has more parameters for the data to fit, so small dataset might lead to bad performance.

A Implementation Details for Neural Networks

This section provides a variety of suggestions for how to efficiently and succinctly implement a neural network and backpropagation.

A.1 Batch Gradient Descent for Neural Networks

Consider the neural network described in Section 5.3 applied to the $n-$th training example $(\mathbf{x}, \mathbf{y})$ where $\mathbf{y}$ is a one-hot encoding of the true label. Our neural network outputs $\hat{\mathbf{y}} = h_{\boldsymbol{\alpha},\boldsymbol{\beta}}(\mathbf{x})$, where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are the parameters of the first and second layers respectively and $h_{\boldsymbol{\alpha},\boldsymbol{\beta}}(\cdot)$ is a one-hidden layer neural network with a sigmoid activation and softmax output. The loss function is negative cross-entropy $J = \ell(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y}^T \log(\hat{\mathbf{y}})$. $J = J_{\mathbf{x},\mathbf{y}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is actually a function of our training example $(\mathbf{x}, \mathbf{y})$, and our model parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ though we write just J for brevity.

In order to train our neural network, we are going to apply batch gradient descent. Because we want the behavior of your program to be deterministic for testing on Autolab, we make a few simplifications: (1) you should *not* shuffle your data and (2) you will use a fixed learning rate. In the real world, you would *not* make these simplifications.

GD proceeds as follows, where E is the number of epochs and γ is the learning rate.

Algorithm 1 Batch Gradient Descent

```
1: procedure GD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
2:     Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$          ▷ Use either RANDOM from Section 5.2.3
3:     for $e \in \{1, 2, \ldots, E\}$ do          ▷ For each epoch
4:         for $B \in \mathcal{D}$ do          ▷ For each batch of training examples
5:             $\mathbf{g}_{\boldsymbol{\alpha}} = 0$
6:             $\mathbf{g}_{\boldsymbol{\beta}} = 0$
7:             for $(\mathbf{x}, \mathbf{y}) \in B$ do          ▷ For each training example
8:                 Compute neural network layers:
9:                 $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
10:                Compute gradients via backprop:
11:                $\left.\begin{array}{l} \mathbf{g}_{\boldsymbol{\alpha}} += \nabla_{\boldsymbol{\alpha}} J \\ \mathbf{g}_{\boldsymbol{\beta}} += \nabla_{\boldsymbol{\beta}} J \end{array}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$
12:            Update parameters:
13:            $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}} \div \text{size}(B)$
14:            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}} \div \text{size}(B)$
15:        Evaluate training mean cross-entropy $J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
16:        Evaluate test mean cross-entropy $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
17:    return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$
```

A.2 General Implementation Tips

- To make your code well-structured and easy to debug, we encourage you NOT to call *compute_XX* functions from their NN class, and in general, to move as much functionality as possible into *hw4_lib.py*
- Nested loop in Python is slow. Try to "vectorize" your code as much as possible—this is particularly important for Python. For example, in Python, you want to avoid for-loops and instead rely on Numpy

calls to perform operations such as matrix multiplication, transpose, subtraction, etc. over an entire Numpy array at once. Why? Because these operations are actually implemented in fast C code, which won't get bogged down the way a high-level scripting language like Python will. More information is at `https://towardsdatascience.com/python-vectorization-5b882eeef658`

- You might want to implement a finite difference test to check whether your implementation of back-propagation is correctly computing gradients. If you choose to do this, comment out this functionality once your backward pass starts giving correct results and before submitting to Gradescope—since it will otherwise slow down your code.

Finite difference checking The gradients can be approximate as

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\Delta) - f(x-\Delta)}{2\Delta}$$

as $\Delta \to 0$. In practical, we can use small Δ to check the correctness of computed gradients. A nice explanation is here: http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/

- With Python 3.6 and Numpy 1.17, you should take advantage of their new features, such as type hints and matrix multiplication operator `@` to make your code clearer and less error-prone.

- Be aware of computing issues. $\log(x)$ is problematic when $x \to 0$. Similarly $exp(x)$ may overflow when it is huge. Think of using $\log$ to avoid some exponential calculations and dividing both numerator and denominator by a large value to avoid overflowing:

$$\frac{e^{x_i}}{\sum e^{x_j}} = \frac{e^{x_i - b}}{\sum e^{x_j - b}}$$

Collaboration Questions Please answer the following:

1. Did you receive any help whatsoever from anyone in solving this assignment?
   Yes / **No**.
   - If you answered 'yes', give full details: ____________________
   - (e.g. "Jane Doe explained to me what is asked in Question 3.4")

2. Did you give any help whatsoever to anyone in solving this assignment?
   Yes / **No**.
   - If you answered 'yes', give full details: ____________________
   - (e.g. "I pointed Joe Smith to section 2.3 since he didn't know how to proceed with Question 2")

3. Did you find or come across code that implements any part of this assignment ?
   Yes / **No**. (See below policy on "found code")
   - If you answered 'yes', give full details: ____________________
   - (book & page, URL & location within the page, etc.).