

INSTRUCTIONS

- **Due:** March 2, 2022 at 11:59 PM EDT.
- **Format:** Complete this pdf with your work and answers. Whether you edit the latex source, use a pdf annotator, or hand write / scan, **make sure that your answers (tex'ed, typed, or handwritten) are within the dedicated regions for each question/part**. If you do not follow this format, we may deduct points. **Do not remove / add any extra pages or modify the sizes of the answer boxes.**
- **How to submit:** Submit a pdf with your answers on Gradescope. Log in and click on our class 10-315, click on the appropriate *Written* assignment, and upload your pdf containing your answers. Don't forget to submit the associated *Programming* component on Gradescope if there is any programming required.
- **Policy:** See the course website for homework policies and Academic Integrity.

Name	
Andrew ID	
Hours to complete (both written and programming)?	

Q1. [38 pts] Example Feed Forward and Backpropagation

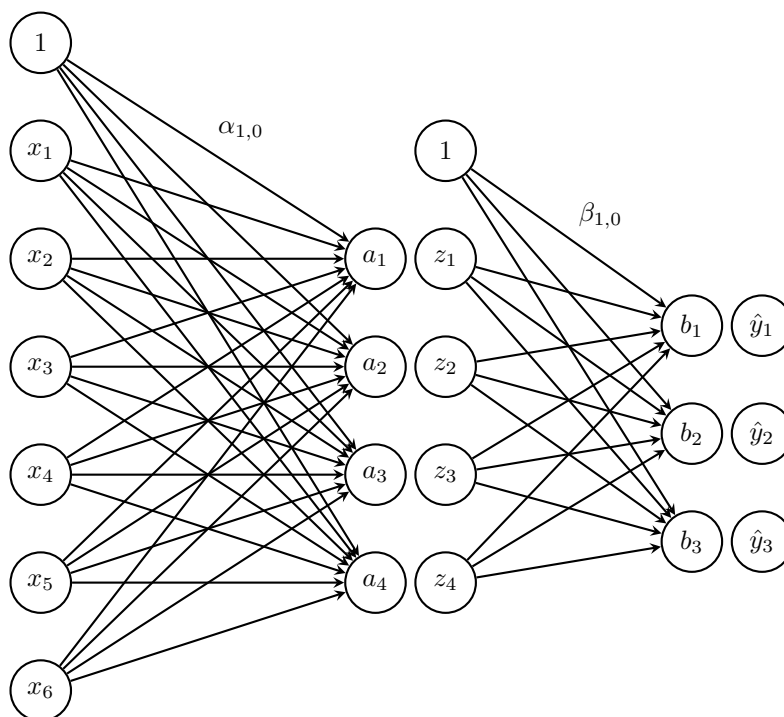


Figure 1: A One Hidden Layer Neural Network

Network Overview Consider the neural network with one hidden layer shown in Figure 1. The input layer consists of 6 features $\mathbf{x} = [x_1, \dots, x_6]^T$, the hidden layer has 4 nodes $\mathbf{z} = [z_1, \dots, z_4]^T$, 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.

α is the matrix of weights from the inputs to the hidden layer and β is the matrix of weights from the hidden layer to the output layer. $\alpha_{j,i}$ represents 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), and β is defined similarly. We will use a sigmoid activation function for the hidden layer and a softmax for the output layer.

Network Details Equivalently, we define each of the following.

The input:

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

Linear combination at the first (hidden) layer:

$$a_j = \alpha_{j,0} + \sum_{i=1}^6 \alpha_{j,i} * x_i, \quad \forall j \in \{1, \dots, 4\} \quad (2)$$

Sigmoid activation at the first (hidden) layer:

$$z_j = \sigma(a_j) = \frac{1}{1 + \exp(-a_j)}, \quad \forall j \in \{1, \dots, 4\} \quad (3)$$

Linear combination at the second (output) layer:

$$b_k = \beta_{k,0} + \sum_{j=1}^4 \beta_{k,j} * z_j, \quad \forall k \in \{1, \dots, 3\} \quad (4)$$

Softmax activation at the second (output) layer:

$$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^3 \exp(b_l)}, \quad \forall k \in \{1, \dots, 3\} \quad (5)$$

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 $\beta_{j,0}$ 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 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 by:

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = - \sum_{i=1}^3 y_i \log(\hat{y}_i) \quad (6)$$

For the below questions use natural log in the equation.

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

- (a) 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 the latter part of this assignment.

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 a training example: $\mathbf{x}^{(1)} = [x_1, x_2, x_3, x_4, x_5, x_6]^T$ with **label class 2**, so $\mathbf{y}^{(1)} = [0, 1, 0]^T$. We initialize the network weights as:

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

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

We want to also consider the bias term and the weights on the bias terms ($\alpha_{j,0}$ and $\beta_{k,0}$). To account for these we can add a new column to the beginning of our initial weight matrices.

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

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

And we can set our first value of our input vectors to always be 1 ($x_0^{(i)} = 1$), so our input becomes:

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

- (i) [1 pt] What is the vector \mathbf{a} whose elements are made up of the entries a_j in equation (2). Write your answer in terms of α and $\mathbf{x}^{(1)}$.

- (ii) [1 pt] What is the **vector** \mathbf{z} whose elements are made up of the entries z_j in equation (3)?

- (iii) [1 pt] **Select one:** We cannot take the matrix multiplication of our weights β and our vector \mathbf{z} since they are not compatible shapes. Which of the following would allow us to take the matrix multiplication of β and \mathbf{z} such that the entries of the vector $\mathbf{b} = \beta \mathbf{z}$ are equivalent to the values of b_k in equation (4)?
- ☐ A) Remove the last column of β
 - ☐ B) Remove the first row of \mathbf{z}
 - ☐ C) Append a value of 1 to be the first entry of \mathbf{z}
 - ☐ D) Append an additional column of 1's to be the first column of β
 - ☐ E) Append a row of 1's to be the first row of β

(b) We will now derive the matrix and vector forms for the backpropagation algorithm.

We start by recalling the following matrix derivatives. Given $L \in \mathbb{R}$, $\alpha \in \mathbb{R}^{D \times M}$, $\mathbf{y} \in \mathbb{R}^M$, and $\mathbf{x} \in \mathbb{R}^D$:

$$\frac{\partial L}{\partial \alpha} = \begin{bmatrix} \frac{\partial \ell}{\partial \alpha_{1,0}} & \frac{\partial \ell}{\partial \alpha_{1,1}} & \cdots & \frac{\partial \ell}{\partial \alpha_{1,M}} \\ \frac{\partial \ell}{\partial \alpha_{2,0}} & \frac{\partial \ell}{\partial \alpha_{2,1}} & \cdots & \frac{\partial \ell}{\partial \alpha_{2,M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \alpha_{D,0}} & \frac{\partial \ell}{\partial \alpha_{D,1}} & \cdots & \frac{\partial \ell}{\partial \alpha_{D,M}} \end{bmatrix} \quad \text{and} \quad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_M}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_D} & \frac{\partial y_2}{\partial x_D} & \cdots & \frac{\partial y_M}{\partial x_D} \end{bmatrix}$$

We also note below the multivariate chain rule, which will be very helpful in solving these problems.

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial L}{\partial \mathbf{y}} \quad \text{or} \quad \frac{\partial L}{\partial x_i} = \sum_{j=1}^M \frac{\partial y_j}{\partial x_i} \frac{\partial L}{\partial y_j}$$

Recall that we are using the **denominator layout** for this course. If you are using numerator layout, the order of chain rule will be switched, but the key idea is otherwise the same. (See Recitation 4 for more detailed explanations on matrix calculus.)

Tip: You should always be examining the shape of the matrices and vectors and making sure that you are comparing your matrix elements with calculations of individual derivatives to make sure they match (e.g. the element of the matrix $(\frac{\partial \ell}{\partial \alpha})_{2,1}$ should be equal to $\frac{\partial \ell}{\partial \alpha_{2,1}}$).

(i) [3 pts] The derivative of the softmax function with respect to b_j is as follows:

$$\frac{\partial \hat{y}_i}{\partial b_j} = \hat{y}_i (\mathbb{I}[i = j] - \hat{y}_j)$$

where $\mathbb{I}[i = j]$ is an indicator function such that if $i = j$ then it returns value 1 and 0 otherwise. Here, \hat{y}_i is the i th component in the softmax output described in Eq. 5, while b_j is the j th component in the output of the linear layer described in Eq. 4.

Using this, express the derivative $\frac{\partial \ell}{\partial b_j}$ in a way such that you do not need this indicator function. Write your solution in terms of \hat{y}_j and y_j only. (Here, ℓ is a scalar representing the loss as defined in Eq. 6.)

$\frac{\partial \ell}{\partial b_j}$:

- (ii) [3 pts] What is the derivative $\frac{\partial \ell}{\partial \beta}$? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{b}}$ and $\mathbf{z}^{(1)}$ where $\mathbf{z}^{(1)} = [1, z_1, z_2, z_3, z_4]^T$.

(Hint: You should first consider a single entry in this matrix: $\frac{\partial \ell}{\partial \beta_{kj}}$.)

$\frac{\partial \ell}{\partial \beta}:$

- (iii) [1 pt] Explain in one short sentence why must we go back to using the matrix β^* (The matrix β without the first column of β) when calculating the matrix $\frac{\partial \ell}{\partial \alpha}$?

- (iv) [3 pts] What is the derivative $\frac{\partial \ell}{\partial \mathbf{z}}$? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{b}}$ and β^*

$\frac{\partial \ell}{\partial \mathbf{z}}:$

- (v) [1 pt] What is the derivative $\frac{\partial \ell}{\partial a_j}$ in terms of $\frac{\partial \ell}{\partial z_j}$ and z_j

$\frac{\partial \ell}{\partial a_j}:$

- (vi) [3 pts] What is the matrix $\frac{\partial \ell}{\partial \alpha}$? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{a}}$ and $\mathbf{x}^{(1)}$.

$\frac{\partial \ell}{\partial \alpha}:$

- (c) Now you will put these equations to use in an example with numerical values. **You should use the answers you get here to debug your code.**

You are given a training example $\mathbf{x}^{(1)} = [1, 1, 0, 0, 1, 1]^T$ with **label class 2**, so $\mathbf{y}^{(1)} = [0, 1, 0]^T$. We initialize the network weights as:

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

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

We want to also consider the bias term and the weights on the bias terms ($\alpha_{j,0}$ and $\beta_{j,0}$). Lets say they are all initialized to 1. To account for this we can add a column of 1's to the beginning of our initial weight matrices.

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

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

And we can set our first value of our input vectors to always be 1 ($x_0^{(i)} = 1$), so our input becomes:

$$\mathbf{x}^{(1)} = [1, 1, 1, 0, 0, 1, 1]^T$$

Using the initial weights, run the feed forward of the network over this example (rounding to 4 decimal places during the calculation) and then answer the following questions.

Showing your work in these questions is optional, but it is recommended to help us understand where any misconceptions may occur.

- (i) [1 pt] What is a_1 ?

a_1 :

Work:

- (ii) [1 pt] What is a_2 ?

a_2 :

Work:

(iii) [1 pt] What is z_1 ?

z_1 :

Work:

(iv) [1 pt] What is z_3 ?

z_3 :

Work:

(v) [2 pts] What is b_1 ?

b_1 :

Work:

(vi) [2 pts] What is b_2 ?

b_2 :

Work:

(vii) [1 pt] What is \hat{y}_2 ?

\hat{y}_2 :

Work:

(viii) [2 pts] Which class would we predict on this example? Your answer should just be an integer $\in \{1, 2, 3\}$.

Class:

Work:

(ix) [1 pt] What is the total loss on this example?

Loss:

Work:

- (d) Now use the results of the previous question to run backpropagation over the network and update the weights. Use learning rate $\eta = 1$.

Do your backpropagation calculations rounding to 4 decimal places then answer the following questions. Showing your work in these questions is optional, but it is recommended to help us understand where any misconceptions may occur.

- (i) [2 pts] What is the value of $\frac{\partial \ell}{\partial \beta_{1,0}}$?

$\frac{\partial \ell}{\partial \beta_{1,0}}:$

Work:

- (ii) [1 pt] What is the updated value of the weight $\beta_{1,0}$?

$\beta_{1,0}:$

Work:

- (iii) [2 pts] What is the updated value of the weight $\alpha_{3,4}$?

$\alpha_{3,4}:$

Work:

- (iv) [4 pts] What is the updated weight of the input layer bias term applied to z_2 (i.e. $\alpha_{2,0}$)?

$\alpha_{2,0}$:

Work:

Q2. [7 pts] Convolutional Neural Networks (CNNs)

In this problem, consider the convolutional layer of a standard implementation of a CNN as described in the lecture.

(a) We are given image X and filter F below:

$$X = \begin{bmatrix} 1 & 0 & -2 & 3 & 4 & 1 \\ 2 & 9 & 5 & 6 & 0 & -1 \\ 0 & -3 & 1 & 3 & 4 & 4 \\ 6 & 5 & 2 & 0 & 6 & 8 \\ -5 & 4 & -3 & 1 & 3 & -2 \\ 4 & 1 & 2 & 8 & 9 & 7 \end{bmatrix} \quad F = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad Y = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

- (i) [1 pt] Let X be convolved with F using no padding and a stride of 1 to produce output Y . What is the value of j in the output Y ?

- (ii) [1 pt] Suppose you had an input feature map of size (height \times width) 6x4 and filter size 2x2, using no padding and a stride of 2, what would be the resulting output size?

Write your answer in the format height \times width

- (b) Parameter sharing is a very important concept for CNNs because it drastically reduces the complexity of the learning problem. For the following questions, assume that there is no bias term in our convolutional layer.

- (i) [1 pt] Which of the following are parameters of a convolutional layer? **Select all that apply:**

- ☐ Stride size
- ☐ Padding size
- ☐ Image size
- ☐ Filter size
- ☐ Weights in the filter
- ☐ None of the above

- (ii) [1 pt] Which of the following are hyperparameters of a convolutional layer? **Select all that apply:**

- ☐ Stride size
- ☐ Padding size
- ☐ Image size
- ☐ Filter size
- ☐ Weights in the filter
- ☐ None of the above

- (iii) [1 pt] Suppose for the convolutional layer, we are given grayscale images of size 22×22 . Using one single 4×4 filter with a stride of 2 and no padding, what is the number of parameters you are learning in this layer?

- (iv) [1 pt] Suppose instead of sharing the same weights for the entire image, you learn a new set of weights each time you apply a filter to the image. Using 4×4 filters with a stride of 2 and no padding, and considering the same grayscale 22×22 image in (iii), what is the number of parameters you are learning in this layer?

- (v) [1 pt] Now suppose you are given a 40×40 colored image, which consists of 3 channels (so your input is a $40 \times 40 \times 3$ tensor), each representing the intensity of one primary color. Suppose you learn a new set of weights each time you apply a filter to the image. Using 4×4 filters with a stride of 2 and no padding, what is the number of parameters you are learning in this layer?

Q3. [45 pts] Neural Network Implementation

Summary In this section, you will build a handwriting recognition system using a neural network. Use the written component of this assignment (Q1) as a guide to lead you through an example of how to implement a neural network. Then, you will implement an end-to-end system that learns to perform handwritten letter classification. You will implement all of the functions needed to initialize, train, evaluate, and make predictions with the network.

Begin by downloading and unzipping the HW2 release from course webpage. This contains the skeleton code and data for this assignment. It also includes an autograder for you to grade your code on your machine. This can be run with the command:

```
python3 autograder.py
```

The code for this assignment consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

Files you will edit

- `neural_network.py`: Your code to implement, train, and execute your neural network.
- `additional_code.py`: Add additional code that you will need to write to answer various questions will go here. This code should be runnable by calling `python3.6 additional_code.py`, but there are no requirements on the format and it will not be executed by the autograder.

Files you might want to look at

- `test_cases/Q*/*.py`: These are the unit tests that the autograder runs. Ideally, you would be writing these unit tests yourself, but we are saving you a bit of time and allowing the autograder to check these things. You should definitely be looking at these to see what is and is not being tested. The autograder on Gradescope may run a different version of these unit tests.
- `test_utils.py`: Utility file used by the test case code.
- `Reference_Outputs`: Expected outputs used by the test case code.

Files you can safely ignore

- `autograder.py`: Autograder infrastructure code.

Files to Edit and Submit:

You will fill in portions of `neural_network.py` and `additional_code.py` during the assignment. You should submit these two files containing your code and comments to the Programming component on Gradescope. Please do not change the other files in this distribution or submit any of our original files other than these files. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Data

We will be using a subset of an Optical Character Recognition (OCR) data set. This data includes images of all 26 handwritten letters; our subset will include only the letters “a,” “e,” “g,” “i,” “l,” “n,” “o,” “r,” “t,” and “u.” The handout contains three data sets drawn from this data: a small data set with 60 samples per class (50 for training and 10 for validation), a medium data set with 600 samples per class (500 for training and 100 for validation), and a large data set with 1000 samples per class (900 for training and 100 for validation). Figure 2 shows a random sample of 10 images of few letters from the data set. You do not need to worry about reading these images; we have already processed them into vectors for you (more details below).

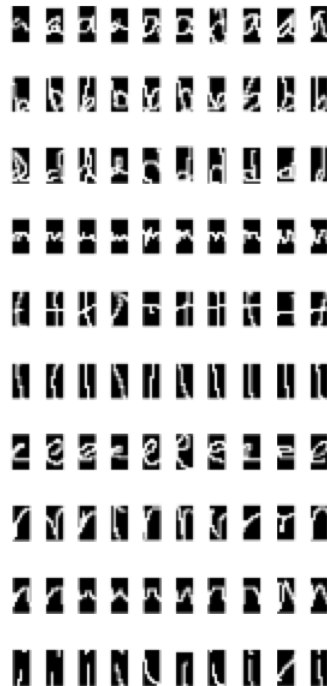


Figure 2: 10 Random Images of Each of 10 Letters in OCR

File Format

Each data set (small, medium, and large) consists of two csv files—train and validation. Each row contains 129 columns separated by commas. The first column contains the label and columns 2 to 129 represent the pixel values of a 16×8 image in a row major format. Label 0 corresponds to “a,” 1 to “e,” 2 to “g,” 3 to “i,” 4 to “l,” 5 to “n,” 6 to “o,” 7 to “r,” 8 to “t,” and 9 to “u.”

Because the original images are black-and-white (not grayscale), the pixel values are either 0 or 1. However, you should write your code to accept arbitrary pixel values in the range $[0,1]$. The images in Figure 2 were produced by converting these pixel values into .png files for visualization. Observe that no feature engineering has been done here; instead the neural network you build will *learn* features appropriate for the task of character recognition.

Model Definition

In this assignment, you will implement a single-hidden-layer neural network with a sigmoid activation function for the hidden layer, and a softmax on the output layer. Let the input vectors \mathbf{x} be of length M , the hidden layer \mathbf{z} consist of D hidden units, and the output layer $\hat{\mathbf{y}}$ be a probability distribution over K classes. That is, each element y_k of the output vector represents the probability of \mathbf{x} belonging to the class k .

Model Architecture		
Input (length)	Layer/Activation	Output (length)
\mathbf{x} of length M	Linear (hidden layer)	\mathbf{a} of length D
\mathbf{a} of length D	Sigmoid Activation	\mathbf{z} of length D
\mathbf{z} of length D	Linear (output layer)	\mathbf{b} of length K
\mathbf{b} of length K	Softmax	\mathbf{y} of length K

We can further express this model by adding bias features to the inputs of layers; assume $x_0 = 1$ is a bias feature on the input and that $z_0 = 1$ is also fixed. In this way, we have two parameter matrices $\boldsymbol{\alpha} \in \mathbb{R}^{D \times (M+1)}$ and $\boldsymbol{\beta} \in \mathbb{R}^{K \times (D+1)}$. The extra 0th column of each matrix (i.e. $\boldsymbol{\alpha}_{:,0}$ and $\boldsymbol{\beta}_{:,0}$) hold the bias parameters. Remember to add the appropriate 0th columns to your inputs/matrices and update the dimensions accordingly (i.e. length $D+1$ instead of D).

To help you recall, we list down the operations being performed in a forward pass of the neural network below:

$$\begin{aligned}
 a_j &= \sum_{m=0}^M \alpha_{jm} x_m \\
 z_j &= \frac{1}{1 + \exp(-a_j)} \\
 b_k &= \sum_{j=0}^D \beta_{kj} z_j \\
 \hat{y}_k &= \frac{\exp(b_k)}{\sum_{l=1}^K \exp(b_l)}
 \end{aligned}$$

The objective function we're using is the average cross entropy over the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$:

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

Some points to mention:

- Do *not* use any machine learning libraries. You may and please do use NumPy.
- Try to “vectorize” your code as much as possible. 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. This is much faster; using NumPy over list can speed up your computation by 200x!
- You'll want to pay close attention to the dimensions that you pass into and return from your functions.

(a) [12 pts] Feed Forward

Implement the forward functions for the layers `linearForward`, `sigmoidForward`, `softmaxForward`, `crossEntropy`.
Next, implement the `NNForward` function that calls a complete forward pass on the neural network.

Algorithm 1 Forward Computation

```

1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ )
2:    $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \alpha)$ 
3:    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$ 
4:    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$ 
5:    $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$ 
6:    $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$ 
7:   return intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ 
8: end procedure

```

This question will be autograded. You may run the following command to run some tests on Q1:

```
python3 autograder.py -q Q1
```

Tip: Check on your dimensions, and make sure you account for the bias features.

(b) [16 pts] Backward Propagation

Implement the backward functions for the layers `softmaxBackward`, `sigmoidBackward`, `linearBackward`.

The gradients we need are the matrices of partial derivatives. Let M be the number of input features, D the number of hidden units, and K the number of outputs.

$$\alpha = \begin{bmatrix} \alpha_{10} & \alpha_{11} & \dots & \alpha_{1M} \\ \alpha_{20} & \alpha_{21} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{D0} & \alpha_{D1} & \dots & \alpha_{DM} \end{bmatrix} \quad \mathbf{g}_\alpha = \frac{\partial J}{\partial \alpha} = \begin{bmatrix} \frac{\partial \ell}{\partial \alpha_{10}} & \frac{\partial \ell}{\partial \alpha_{11}} & \dots & \frac{\partial \ell}{\partial \alpha_{1M}} \\ \frac{\partial \ell}{\partial \alpha_{20}} & \frac{\partial \ell}{\partial \alpha_{21}} & \dots & \frac{\partial \ell}{\partial \alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \alpha_{D0}} & \frac{\partial \ell}{\partial \alpha_{D1}} & \dots & \frac{\partial \ell}{\partial \alpha_{DM}} \end{bmatrix} \quad (7)$$

$$\beta = \begin{bmatrix} \beta_{10} & \beta_{11} & \dots & \beta_{1D} \\ \beta_{20} & \beta_{21} & \dots & \beta_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{K0} & \beta_{K1} & \dots & \beta_{KD} \end{bmatrix} \quad \mathbf{g}_\beta = \frac{\partial J}{\partial \beta} = \begin{bmatrix} \frac{\partial \ell}{\partial \beta_{10}} & \frac{\partial \ell}{\partial \beta_{11}} & \dots & \frac{\partial \ell}{\partial \beta_{1D}} \\ \frac{\partial \ell}{\partial \beta_{20}} & \frac{\partial \ell}{\partial \beta_{21}} & \dots & \frac{\partial \ell}{\partial \beta_{2D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \beta_{K0}} & \frac{\partial \ell}{\partial \beta_{K1}} & \dots & \frac{\partial \ell}{\partial \beta_{KD}} \end{bmatrix} \quad (8)$$

Reminder once again that α and \mathbf{g}_α are $D \times (M+1)$ matrices, while β and \mathbf{g}_β are $K \times (D+1)$ matrices. The +1 comes from the extra columns $\alpha_{\cdot,0}$ and $\beta_{\cdot,0}$ which are the bias parameters for the first and second layer respectively. We will always assume $x_0 = 1$ and $z_0 = 1$.

Next, implement the `NNBackward` function that calls a complete backward pass on the neural network.

Algorithm 2 Backpropagation

```

1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ , Intermediates  $\mathbf{z}, \hat{\mathbf{y}}$ )
2:    $\mathbf{g}_b = \text{SOFTMAXBACKWARD}^*(\mathbf{y}, \hat{\mathbf{y}})$ 
3:    $\mathbf{g}_\beta, \mathbf{g}_z = \text{LINEARBACKWARD}(\mathbf{z}, \beta, \mathbf{g}_b)$ 
4:    $\mathbf{g}_a = \text{SIGMOIDBACKWARD}(\mathbf{z}, \mathbf{g}_z)$ 
5:    $\mathbf{g}_\alpha, \mathbf{g}_x = \text{LINEARBACKWARD}(\mathbf{x}, \alpha, \mathbf{g}_a)$  ▷ We discard  $\mathbf{g}_x$ 
6:   return parameter gradients  $\mathbf{g}_\alpha, \mathbf{g}_\beta, \mathbf{g}_b, \mathbf{g}_z, \mathbf{g}_a$ 
7: end procedure

```

*It is common to combine the Cross-Entropy and Softmax backpropagation into one, due to the simpler calculation.

This question will be autograded. You may run the following command to run some tests on Q2:

```
python3 autograder.py -q Q2
```


(c) [9 pts] Training with Stochastic Gradient Descent

Implement the **SGD** function, where you apply stochastic gradient descent to your training.

Because we want the behavior of your program to be deterministic for testing on Gradescope, 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.

SGD proceeds as follows, where γ is the learning rate and E is the number of epochs. One epoch is one full pass through the entire dataset.

Algorithm 3 Stochastic Gradient Descent (SGD) without Shuffle

```

procedure SGD(Training data  $\mathcal{D}$ , Validation data  $\mathcal{D}'$ , other relevant parameters)
  Initialize parameters  $\alpha, \beta$  ▷ Use either RANDOM or ZERO, depending on the init_rand
  Initialize empty lists losses_train and losses_val
  for  $e \in \{1, 2, \dots, E\}$  do ▷ For each epoch
    for  $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$  do ▷ For each training example (No shuffling)
      Compute neural network layers:
       $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \alpha, \beta)$ 
      Compute gradients via backprop:
       $\left. \begin{array}{l} \mathbf{g}_\alpha = \frac{\partial J}{\partial \alpha} \\ \mathbf{g}_\beta = \frac{\partial J}{\partial \beta} \end{array} \right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \alpha, \beta, \mathbf{z}, \hat{\mathbf{y}})$ 
      Update parameters:
       $\alpha \leftarrow \alpha - \gamma \mathbf{g}_\alpha$ 
       $\beta \leftarrow \beta - \gamma \mathbf{g}_\beta$ 
    end for
    Compute training mean cross-entropy  $J(\alpha, \beta)$  and store in losses_train ▷ from Eq. 10
    Compute validation mean cross-entropy  $J(\alpha, \beta)$  and store in losses_val ▷ from Eq. 10
  end for
  return  $\alpha, \beta$ , losses_train, losses_val
end procedure

```

Initialization

In order to use 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, we will be using two possible initialization methods:

RANDOM – Weights are initialized randomly from Uniform $[-0.1, 0.1]$. Biases are initialized to zero.

ZERO – All weights are initialized to 0.

You must support both of these initialization schemes. To keep things consistent with the autograder, we recommend using `numpy.random.uniform` for the random initialization. (Tip: Think carefully about the dimensions of α and β . If you're confused, you can revisit your answers in the written part.)

Cross-Entropy $J_{SGD}(\alpha, \beta)$

Cross-entropy $J_{SGD}(\alpha, \beta)$ for a single example i is defined as follows:

$$J_{SGD}(\alpha, \beta) = - \sum_{k=1}^K y_k^{(i)} \log(\hat{y}_k^{(i)}) \quad (9)$$

J is a function of model parameters α and β because $\hat{y}_k^{(i)}$ is implicitly a function of $\mathbf{x}^{(i)}$, α , and β since it is the output of the neural network applied to $\mathbf{x}^{(i)}$. Of course, $\hat{y}_k^{(i)}$ and $y_k^{(i)}$ are the k th components of $\hat{\mathbf{y}}^{(i)}$ and $\mathbf{y}^{(i)}$ respectively. The objective function you then use to calculate the average cross entropy over, say the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, is:

$$J(\alpha, \beta) = - \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_k^{(i)} \log(\hat{y}_k^{(i)}) \quad (10)$$

This question is autograded and depends on the correctness to your previous parts. You may run the following command to run some tests on Q3:

```
python3 autograder.py -q Q3
```

(d) [4 pts] Label Prediction

Recall that for an input x , your network outputs a probability distribution over K classes, \hat{y} . After training your network and obtaining weight parameters α and β , you now want to predict the labels given the data. We also want to find the train and validation error, which is equivalent to 1 minus the accuracy.

Implement the `prediction` function as follows:

Algorithm 4 Prediction

```

1: procedure PREDICTION(Training data  $\mathcal{D}$ , Validation data  $\mathcal{D}'$ , Parameters  $\alpha, \beta$ )
2:   for  $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$  do
3:     Compute neural network prediction  $\hat{\mathbf{y}}$  from NNFORWARD( $\mathbf{x}, \mathbf{y}, \alpha, \beta$ )
4:     Predict the label with highest probability  $l = \operatorname{argmax}_k \hat{y}_k$ 
5:     Check for error  $l \neq y$ 
6:   end for
7:   for  $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}'$  do
8:     Compute neural network prediction  $\hat{\mathbf{y}}$  from NNFORWARD( $\mathbf{x}, \mathbf{y}, \alpha, \beta$ )
9:     Predict the label with highest probability  $l = \operatorname{argmax}_k \hat{y}_k$ 
10:    Check for error  $l \neq y$ 
11:  end for
12:  return train_error, valid_error, train_predictions, valid_predictions
13: end procedure

```

This question is autograded and depends on the correctness to your previous parts. You may run the following command to run some tests on Q4:

```
python3 autograder.py -q Q4
```

(e) [4 pts] Main train_and_valid function

Finally, implement the `train_and_valid()` function to train and validate your neural network implementation. Your program should learn the parameters of the model on the training data, and report the following: (1) cross-entropy on both train and validation data for each epoch, (2) predictions for both train and validation data, and (3) error rates on both train and validation data. See the docstring in the code for more details. You may implement any helper code or functions you'd like within `neural_network.py`.

Your implementation must satisfy the following requirements:

- Number of **hidden units** for the hidden layer will be determined by the `num_hidden` argument to the `train_and_valid` function.
- SGD must support two different **initialization strategies** (namely RANDOM and ZERO), selecting between them based on the `init_rand` argument to the `train_and_valid` function.
- The number of **epochs** will be determined by the `num_epochs` argument to the `train_and_valid` function.
- The **learning rate** for SGD is specified by the `learning_rate` argument to the `train_and_valid` function.
- Perform SGD updates on the training data in the order that the data is given in the input file. Although you would typically shuffle training examples when using stochastic gradient descent, in order to autograde the assignment, we ask that you **DO NOT** shuffle trials in this assignment.

This question is autograded and depends on the correctness to your previous parts. You may run the following command to run some tests on Q5:

```
python3 autograder.py -q Q5
```

Submission

Upload `neural_network.py` and `additional_code.py` to Gradescope. Your submission should finish running within 20 minutes, after which it will time out on Gradescope.

You may submit to Gradescope as many times as you like. You may also run the autograder on your own machine to speed up the development process. Just note that the autograder on Gradescope will be slightly different than the local autograder. The autograder can be invoked on your own machine using the command:

```
python3 autograder.py
```

Note that running the autograder locally will not register your grades with us. Remember to submit your code when you want to register your grades for this assignment.

The autograder on Gradescope might take a while, but don't worry; so long as you submit before the deadline, it's not late.

Q4. [10 pts] Programming (continued)

The following questions should be completed after you work through the programming portion of this assignment.

For these questions, **use the large dataset**. Use the following values for the hyperparameters unless otherwise specified:

Parameter	Value
Number of Hidden Units	50
Weight Initialization	RANDOM
Learning Rate	0.01

Please submit computer-generated plots for (a(i)) and (b(i)). Include any code required to produce these results in `additional_code.py` when submitting the programming component. Note: we expect it to take about **5 minutes** to train each of these networks.

(a) Hidden Units

- (i) [3 pts] Train a single hidden layer neural network using the hyperparameters mentioned in the table above, except for the number of hidden units which should vary among 5, 20, 50, 100, and 200. Run the optimization for 100 epochs each time.

Plot the average training cross-entropy (sum of the cross-entropy terms over the training dataset divided by the total number of training examples) on the y-axis vs number of hidden units on the x-axis. In the **same figure**, plot the average validation cross-entropy.

Plot:



- (ii) [2 pts]

Examine and comment on the the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?

Answer:



(b) Learning Rate

- (i) [3 pts] Train a single hidden layer neural network using the hyperparameters mentioned in the table above, except for the learning rate which should vary among 0.1, 0.01, and 0.001. Run the optimization for 100 epochs each time.

Plot the average training cross-entropy on the y-axis vs the number of epochs on the x-axis for the mentioned learning rates. In the **same figure**, plot the average validation cross-entropy loss. Make a separate figure for each learning rate.

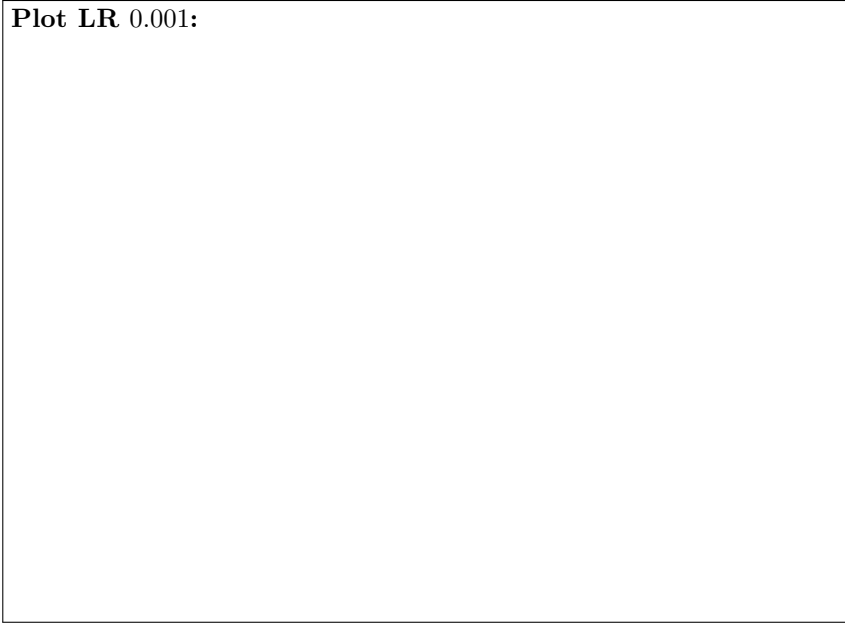
Plot LR 0.1:



Plot LR 0.01:

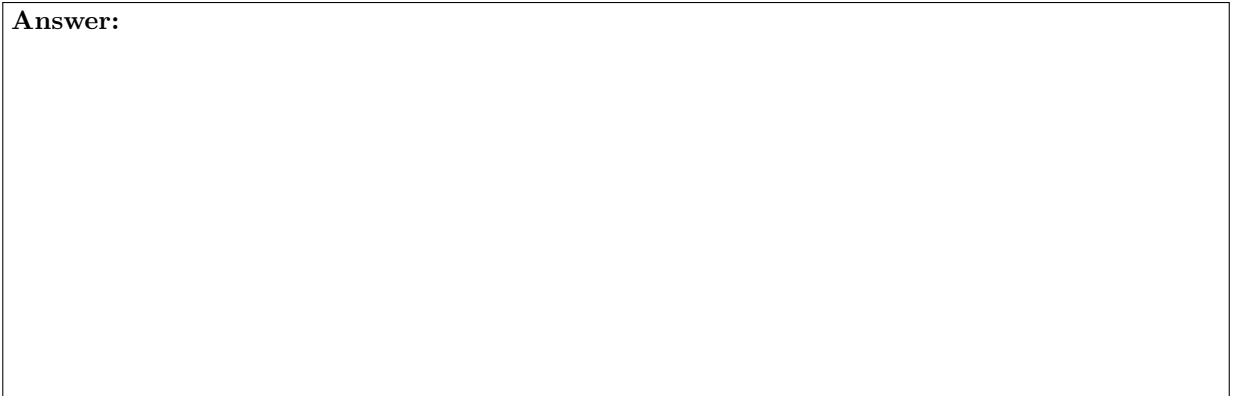


Plot LR 0.001:



- (ii) [2 pts] Examine and comment on the the plots of training and validation cross-entropy. How does adjusting the learning rate affect the convergence of cross-entropy of each dataset?

Answer:



Collaboration Questions

After you have completed all other components of this assignment, report your answers to the collaboration policy questions detailed in the Academic Integrity Policies found on the course site.

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details.

2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details?

3. Did you find or come across code that implements any part of this assignment ? If so, include full details.