HOMEWORK 5: NEURAL NETWORKS

10-301/10-601 Introduction to Machine Learning (Fall 2022)
https://www.cs.cmu.edu/~mgormley/courses/10601/
OUT: Thursday, October 13
DUE: Thursday, October 27 at 11:59 PM
TAs: Alex, Lavanya, Neural, Qiuyi, Tara, Yuchen

Summary In this assignment, you will build an image recognition system using a neural network. In the Written component, you will walk through an on-paper example of how to implement a neural network. Then, in the Programming component, you will implement an end-to-end system that learns to perform image classification.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.9.12) and versions of permitted libraries (e.g. `numpy` 1.23.0) match those used on Gradescope. You have 10 free Gradescope programming submissions. After 10 submissions, you will begin to lose points from your total programming score. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.
- **Materials:** The data and reference output that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Instructions for Specific Problem Types

For "Select One" questions, please fill in the appropriate bubble completely:

Select One: Who taught this course?

- ● Matt Gormley
- ○ Marie Curie
- ○ Noam Chomsky

If you need to change your answer, you may cross out the previous answer and bubble in the new answer:

Select One: Who taught this course?

- ● Henry Chai
- ○ Marie Curie
- ● ~~Noam Chomsky~~

For "Select all that apply" questions, please fill in all appropriate squares completely:

Select all that apply: Which are scientists?

- ■ Stephen Hawking
- ■ Albert Einstein
- ■ Isaac Newton
- □ I don't know

Again, if you need to change your answer, you may cross out the previous answer(s) and bubble in the new answer(s):

Select all that apply: Which are scientists?

- ■ Stephen Hawking
- ■ Albert Einstein
- ■ Isaac Newton
- ■ ~~I don't know~~

For questions where you must fill in a blank, please make sure your final answer is fully included in the given space. You may cross out answers or parts of answers, but the final answer must still be within the given space.

Fill in the blank: What is the course number?

10-601	10-~~6~~301

Written Questions (29 points)

1 Example Feed Forward and Backpropagation (15 points)

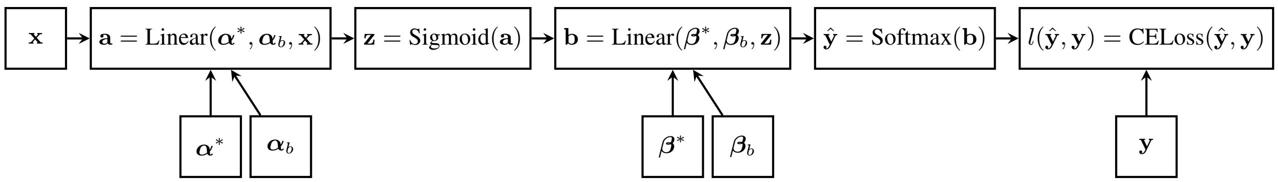


Figure 1: Computational Graph for a One Hidden Layer Neural Network

Network Overview Consider the neural network with one hidden layer shown in Figure 1. The input layer consists of 6 features $\mathbf{x} = [x_1, ..., x_6]^T$, 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 (**1-indexed** such that y_i is the probability of label i).

$\boldsymbol{\alpha}^*$ is the matrix of weights from the inputs to the hidden layer and $\boldsymbol{\beta}^*$ 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 $\boldsymbol{\beta}^*$ is defined similarly. We will use a sigmoid activation function for the hidden layer and a softmax for the output layer.

The bias vectors $\boldsymbol{\alpha}_b, \boldsymbol{\beta}_b$ are defined such that the jth value of $\boldsymbol{\alpha}_b$ (which we denote $\alpha_{j,b}$) is the bias value for a_j and the kth value of $\boldsymbol{\beta}_b$ is the bias value for b_k.

Network Details Equivalently, we define each of the following.

The input:

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

Linear combination at the first (hidden) layer:

$$a_j = \alpha_{j,b} + \sum_{i=1}^{6} \alpha^*_{j,i} \cdot x_i, \ \forall j \in \{1, \dots, 4\} \tag{2}$$

Activation at the first (hidden) layer:

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

Equivalently, we can write this as vector operation where the sigmoid activation is applied individually to each element of the vector $\mathbf{a}$:

$$\mathbf{z} = \sigma(\mathbf{a}) \tag{4}$$

Linear combination at the second (output) layer:

$$b_k = \beta_{k,b} + \sum_{j=1}^{4} \beta^*_{k,j} \cdot z_j, \ \forall k \in \{1, \dots, 3\} \tag{5}$$

Activation at the second (output) layer:

$$\hat{y}_k = \frac{\exp(b_k)}{\sum\limits_{l=1}^{3} \exp(b_l)}, \ \forall k \in \{1, \ldots, 3\} \tag{6}$$

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:

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{3} y_i \log(\hat{y}_i) \tag{7}$$

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 $y_1 = 0$, $y_2 = 1$, $y_3 = 0$.

1. 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 keep your program under the Gradescope time-out.
   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 an 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 ($\alpha_{j,b}$ and $\beta_{k,b}$). To account for these we can add them as a new column to the beginning of our initial weight matrices to represent biases, (e.g. $\alpha_{1,0} = \alpha_{1,b}$).

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

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

We then add a corresponding new first dimension to our input vectors, always set to 1 ($x_0^{(i)} = 1$), so our input becomes:

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

(a) (1 point) By examining the shapes of the initial weight matrices, how many neurons do we have in the first hidden layer of the neural network? Do not include the bias in your count.

Answer
4

(b) (1 point) How many output neurons will our neural network have?

Answer
3

(c) (1 point) What is the vector $\mathbf{a}$ whose elements are made up of the entries a_j in Equation 2 (using $x_i^{(1)}$ in place of x_i). Write your answer in terms of $\boldsymbol{\alpha}$ and $\mathbf{x}^{(1)}$.

Answer
$\boldsymbol{\alpha}\mathbf{x}^{(1)}$

(d) (1 point) **Select one:** We cannot take the matrix multiplication of our weights $\boldsymbol{\beta}$ and the vector $\mathbf{z} = [z_1, z_2, z_3, z_4]^T$ since they are not compatible shapes. Which of the following would allow us to take the matrix multiplication of $\boldsymbol{\beta}$ and $\mathbf{z}$ such that the entries of the vector $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$ are equivalent to the values of b_k in Equation 5?

- ○ A) Remove the first row of $\mathbf{z}$
- ● B) Append a value of 1 to be the first entry of $\mathbf{z}$.
- ○ C) Append an additional column of 1's to be the first column of $\boldsymbol{\beta}$
- ○ D) Append a row of 1's to be the first row of $\boldsymbol{\beta}$

(e) (1 point) What are the entries of the output vector $\hat{\mathbf{y}}$? Your answer should be written in terms of b_1, b_2, b_3.

ŷ

$$\begin{bmatrix} \frac{\exp(b_1)}{\exp(b_1)+\exp(b_2)+\exp(b_3)} \\ \frac{\exp(b_2)}{\exp(b_1)+\exp(b_2)+\exp(b_3)} \\ \frac{\exp(b_3)}{\exp(b_1)+\exp(b_2)+\exp(b_3)} \end{bmatrix}$$

2. We will now derive the matrix and vector forms for the backpropagation algorithm, for example

$$\frac{\partial \ell}{\partial \boldsymbol{\alpha}} = \begin{bmatrix} \frac{\partial \ell}{\partial \alpha_{10}} & \frac{\partial \ell}{\partial \alpha_{11}} & \cdots & \frac{\partial \ell}{\partial \alpha_{16}} \\ \frac{\partial \ell}{\partial \alpha_{20}} & \frac{\partial \ell}{\partial \alpha_{21}} & \cdots & \frac{\partial \ell}{\partial \alpha_{26}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \alpha_{40}} & \frac{\partial \ell}{\partial \alpha_{41}} & \cdots & \frac{\partial \ell}{\partial \alpha_{46}} \end{bmatrix}$$

The mathematics which you have to derive in this section jump significantly in difficulty, 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 $(\frac{\partial \ell}{\partial \boldsymbol{\alpha}})_{2,1}$ should be equal to $\frac{\partial \ell}{\partial \alpha_{2,1}}$). Recall that ℓ is our loss function defined in Equation 7:

Note: all vectors are column vectors (i.e. an n dimensional vector $v \in \mathbb{R}^{n\times 1}$). **Assume that all input vectors to linear layers have a bias term folded in, unless otherwise specified**. All partial derivatives should be written in denominator layout notation. An example of denominator notation is that $\frac{\partial \ell}{\partial \beta} \in \mathbb{R}^{3\times 5}$ because $\beta \in \mathbb{R}^{3\times 5}$.

(a) (1 point) What is the derivative $\frac{\partial \ell}{\partial \hat{y}_i}$, where $1 \leq i \leq 3$? Your answer should be in terms of y_i and $\hat{y}_i$. Recall that we define the loss $\ell(\hat{y}, y)$ as follows (*note: log is a natural log*):

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{3} y_i \log(\hat{y}_i) \tag{7}$$

$\partial \ell / \partial \hat{y}_i$

$-\frac{y_i}{\hat{y}_i}$

(b) (3 points) The derivative of the softmax function with respect to b_k is as follows:

$$\frac{\partial \hat{y}_l}{\partial b_k} = \hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k) \tag{8}$$

where $\mathbb{I}[k = l]$ is an indicator function such that if $k = l$ then it returns value 1 and 0 otherwise.

Using this and your result from (a), write the derivative $\frac{\partial \ell}{\partial b_k}$ in a smart way such that you do not need the indicator function in Equation 8. Write your solutions in terms of $\hat{y}_k, y_k$. Show your work below.

Hint 1: Recall that $\frac{\partial \ell}{\partial b_k} = \sum_l \frac{\partial \ell}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial b_k}$.

Hint 2: After substituting in your expressions for $\frac{\partial \ell}{\partial \hat{y}_l}$ and $\frac{\partial \hat{y}_l}{\partial b_k}$, try to rearrange terms so that you encounter the expression $\hat{y}_k \sum_l y_l$. What is the value of $\sum_l y_l$?

$\partial \ell / \partial b_k$

$$\frac{\partial l}{\partial \hat{y}_l} = -\frac{y_l}{\hat{y}_l} \tag{9}$$

$$\frac{\partial \hat{y}_l}{\partial b_k} = \hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k) \tag{10}$$

$$\frac{\partial \ell}{\partial b_k} = \sum_l \frac{\partial \ell}{\partial \hat{y}_l} \frac{\partial \hat{y}_l}{\partial b_k} \tag{11}$$

$$= \sum_l -y_l(\mathbb{I}[k = l] - \hat{y}_k) \tag{12}$$

$$= \sum_l -y_l \mathbb{I}[k = l] + \sum_l y_l \hat{y}_k \tag{13}$$

$$= -y_k + \hat{y}_k \tag{14}$$

(c) (2 points) What is the derivative $\frac{\partial \ell}{\partial \boldsymbol{\beta}}$? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{b}}$ and $\mathbf{z}$.

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

$\partial \ell / \partial \boldsymbol{\beta}$

$$\frac{\partial \ell}{\partial \beta_{kj}} = \frac{\partial \ell}{\partial b_k} \cdot \frac{\partial b_k}{\partial \beta_{kj}} \tag{15}$$

$$= \frac{\partial \ell}{\partial b_k} \cdot z_j \tag{16}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\beta}} = \begin{bmatrix} \frac{\partial \ell}{\partial b_1} \cdot z_0 & \frac{\partial \ell}{\partial b_1} \cdot z_1 & \frac{\partial \ell}{\partial b_1} \cdot z_2 & \frac{\partial \ell}{\partial b_1} \cdot z_3 & \frac{\partial \ell}{\partial b_1} \cdot z_4 \\ \frac{\partial \ell}{\partial b_2} \cdot z_0 & \frac{\partial \ell}{\partial b_2} \cdot z_1 & \frac{\partial \ell}{\partial b_2} \cdot z_2 & \frac{\partial \ell}{\partial b_2} \cdot z_3 & \frac{\partial \ell}{\partial b_2} \cdot z_4 \\ \frac{\partial \ell}{\partial b_3} \cdot z_0 & \frac{\partial \ell}{\partial b_3} \cdot z_1 & \frac{\partial \ell}{\partial b_3} \cdot z_2 & \frac{\partial \ell}{\partial b_3} \cdot z_3 & \frac{\partial \ell}{\partial b_3} \cdot z_4 \end{bmatrix} = \frac{\partial \ell}{\partial \mathbf{b}} \mathbf{z}^T \tag{17}$$

(d) (1 point) **Select one:** Why do we use the matrix $\boldsymbol{\beta}^*$ (the matrix $\boldsymbol{\beta}$ without the first column of bias values) instead of $\boldsymbol{\beta}$ when calculating the derivative matrix $\frac{\partial \ell}{\partial \alpha}$? *(Hint: try drawing a computation graph with the bias unfolded).*

- ○ A) The bias terms do not update, so there is no need to include them in backpropagation.
- ○ B) It is the computationally cheapest column to remove to ensure that the dimensions match.
- ● C) The elements $\boldsymbol{\beta}_{k,0}$ are not determined by the values of α
- ○ D) The derivative of loss with respect to the bias terms is always zero.

(e) (1 point) What is the derivative $\frac{\partial \ell}{\partial \mathbf{z}}$ (**not including the bias term**)? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{b}}$ and $\boldsymbol{\beta}^*$.

$\partial \ell / \partial \mathbf{z}$

$$\frac{\partial \ell}{\partial \mathbf{z}} = \frac{\partial \ell}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{z}} \tag{18}$$

$$= \frac{\partial \ell}{\partial \mathbf{b}} \frac{\partial (\boldsymbol{\beta}^* \mathbf{z} + \boldsymbol{\beta}_b)}{\partial \mathbf{z}} \tag{19}$$

$$= (\boldsymbol{\beta}^*)^T \frac{\partial \ell}{\partial \mathbf{b}} \tag{20}$$

(f) (1 point) What is the derivative $\frac{\partial \ell}{\partial a_j}$ in terms of $\frac{\partial \ell}{\partial z_j}$ and z_j?

$\partial \ell / \partial a_j$

$\frac{\partial \ell}{\partial a_j} = \frac{\partial \ell}{\partial z_j} \frac{\partial z_j}{\partial a_j}$
$= \frac{\partial \ell}{\partial z_j} z_j (1 - z_j)$

(g) (1 point) What is the matrix $\frac{\partial \ell}{\partial \boldsymbol{\alpha}}$? Your answer should be in terms of $\frac{\partial \ell}{\partial \mathbf{a}}$ and $\mathbf{x}^{(1)}$.

$\partial \ell / \partial \boldsymbol{\alpha}$

$\frac{\partial \ell}{\partial \alpha_{ji}} = \frac{\partial \ell}{\partial a_j} \frac{\partial a_j}{\partial \alpha_{ji}}$
$= \frac{\partial \ell}{\partial a_j} (x_i)^{(1)}$

$$\frac{\partial \ell}{\partial \boldsymbol{\alpha}} = \begin{bmatrix} \frac{\partial \ell}{\partial a_1}(x_0)^{(1)} & \frac{\partial \ell}{\partial a_1}(x_1)^{(1)} & ... & \frac{\partial \ell}{\partial a_1}(x_6)^{(1)} \\ \frac{\partial \ell}{\partial a_2}(x_0)^{(1)} & \frac{\partial \ell}{\partial a_2}(x_1)^{(1)} & ... & \frac{\partial \ell}{\partial a_2}(x_6)^{(1)} \\ ... & ... & ... & \\ \frac{\partial \ell}{\partial a_4}(x_0)^{(1)} & \frac{\partial \ell}{\partial a_4}(x_1)^{(1)} & ... & \frac{\partial \ell}{\partial a_4}(x_6)^{(1)} \end{bmatrix}$$

$= \frac{\partial \ell}{\partial \mathbf{a}} \mathbf{x}^{(1)^T}$

2 Empirical Questions (14 points)

The following questions should be completed after you work through the programming portion of this assignment. **For any plotting questions, you must using curves/line graph, title your graph, label your axes and provide units (if applicable), and provide a legend in order to receive full credit.**

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

Parameter	Value
Number of Hidden Units	50
Weight Initialization	RANDOM
Learning Rate	0.001

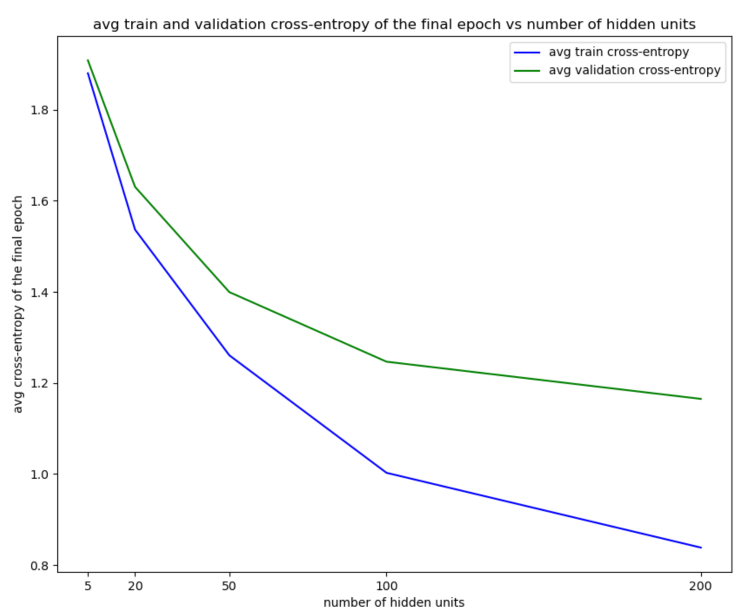
Please submit computer-generated plots for all parts. To get full credit, your plots must be line graphs with labels for both axes with the value plotted on that axis and legends labeling every line.

1. Hidden Units

   (a) (2 points) 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) of the final epoch on the y-axis vs number of hidden units on the x-axis. In the **same figure**, plot the average validation cross-entropy. The x-axis should be the number of hidden units, the y-axis should be average cross-entropy loss, and there should be one curve for validation loss and one curve for train loss.

Avg. Train and Validation Cross-Entropy Loss



(b) (2 points) Examine and comment on the the plots of training and validation cross-entropy. What problem arises with too few hidden units, and why does it happen?

Answer

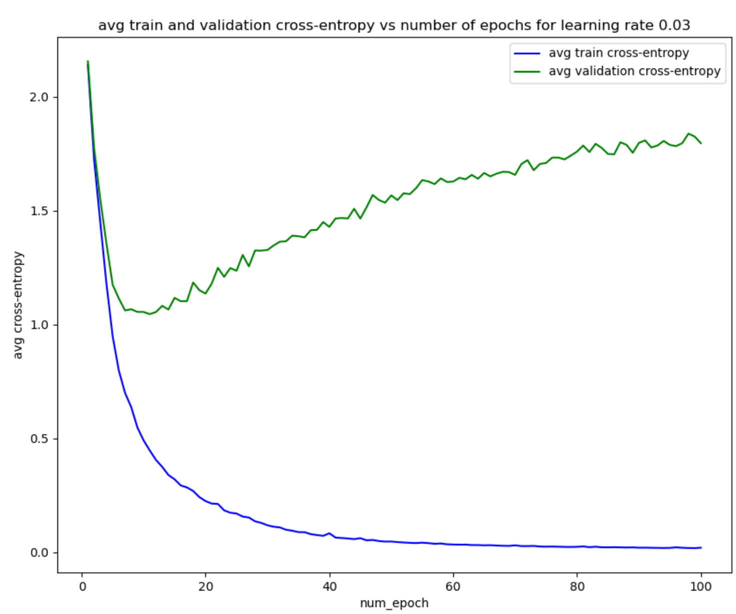
The validation cross-entropy is higher than training cross-entropy, which is expected, since validation error is generally higher than train error. As the hidden size increases, both training and validation cross-entropy decreases. If there are too few hidden units, the average cross-entropy will still be large at the final train epoch. It is possibly because the network with too few hidden units does not have the enough complexity to fit the data and cannot learn the features of dataset very well. E.g., The data feature dimension is 228, but the hidden layer has 5 units, which is likely not enough to learn the features.

2. Learning Rate

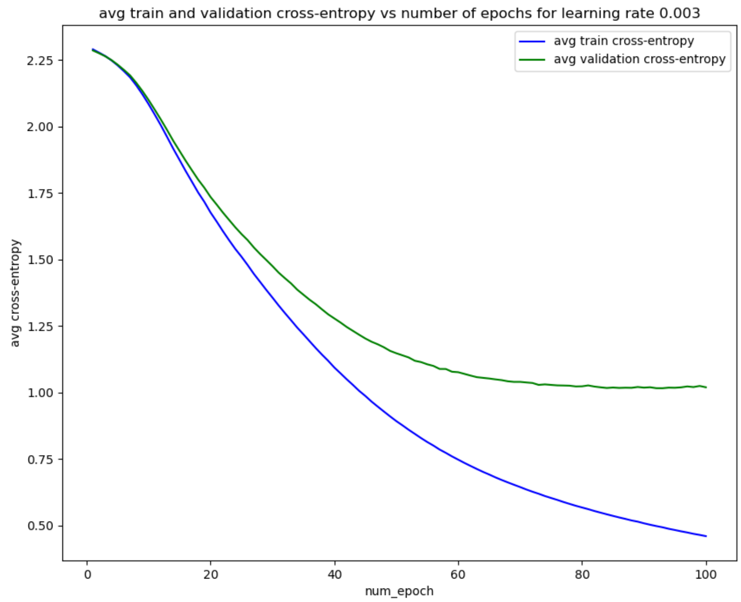
(a) (6 points) Train a single hidden layer neural network using the hyperparameters mentioned in the table above, except for the learning rate which should vary among **0.03, 0.003, and 0.0003**. Run the optimization for 100 epochs each time.

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. The x-axis should be epoch number, the y-axis should be average cross-entropy loss, and there should be one curve for training loss and one curve for validation loss.

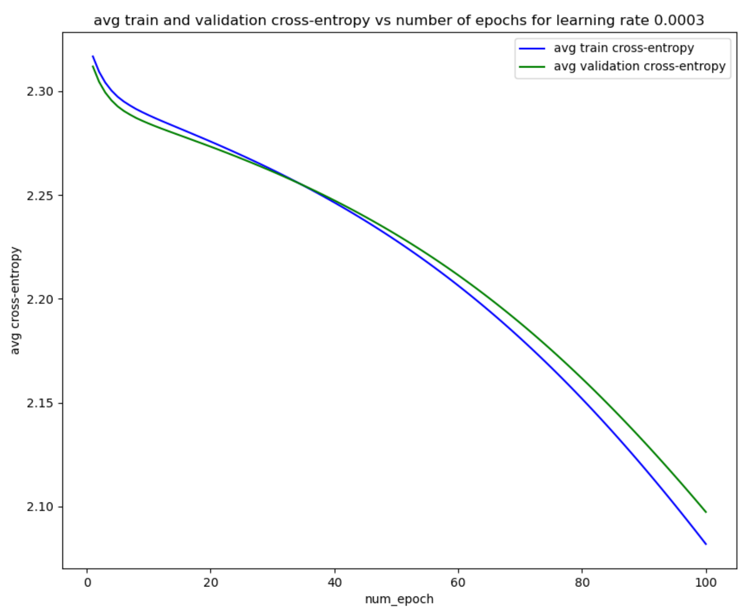
Plot LR 0.03



Plot LR 0.003



Plot LR 0.0003



(b) (2 points) Examine and comment on the plots of training and validation cross-entropy. Are there any learning rates for which convergence is not achieved? Are there any learning rates that exhibit other problems? If so, describe these issues and list the learning rates that cause them.

Answer

The smallest learning rate (0.0003) did not achieve convergence, since this learning rate is too small and therefore the learning process is very slow.
The largest learning rate (0.03) also exhibit problems. Although train cross-entropy keeps decreasing, converges very fast, and goes to nearly zero at convergence, the validation cross-entropy exhibits a pattern that it only decreases at the first few epochs and then increases. This causes the overfitting problem. It is likely because the learning rate is too large, the gradient updates get overshoots and the model is likely to converge too quickly to a suboptimal solution. The learning rate (0.003) has the best performance in the three choices.Both train and val cross-entropy decreases gradually as epoch increases.

3. Weight Initialization

(a) (2 points) For this exercise, you can work on any data set. Initialize $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ to zero and print them out after the first few updates. For example, you may use the following command to begin:

```
$ python neuralnet.py small_train.csv small_validation.csv \
small_train_out.labels small_validation_out.labels \
small_metrics_out.txt 1 4 2 0.1
```

Compare the values across rows and columns in $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Describe the observed behavior and how this may affect model capacity and convergence.

Answer

The values on the same columns for $\boldsymbol{\alpha}$ are the same; The values on the same rows for $\boldsymbol{\beta}$ are the same. This is because that with zero-initilization, the weight gradients at each row/column will remain the same, the weights will be the same for each row/column in future iterations. This will cause the the problem that the network fails to break symmetry, i.e., neurons to learn the same features in each iteration. Therefore, the model will have low capacity to fit the dataset features. It will also cause slower convergence because neurons to learn the same features in each iteration. Also, it also has the problem that the hidden layer units can get zero signal and learn nothing, which can also decreases the convergence rate. In addition, randomly initialization with different values may reduce the probability to be stuck at local minimum, but zero inilialized weights are more likely to have this problem.

3 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found here.

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.

Your Answer

1. No 2. No 3. No

Programming (94 points)

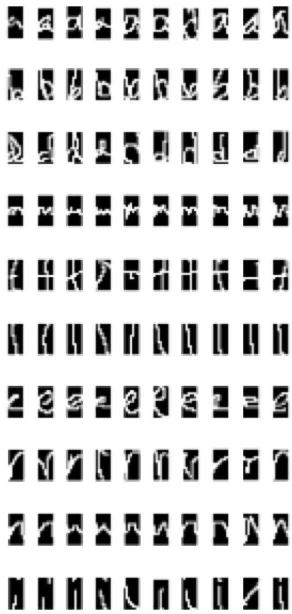


Figure 2: 10 random images of each of the 10 letters in the OCR dataset.

4 The Task

Your goal in this assignment is to implement a neural network to classify images using a single hidden layer neural network.

5 The Datasets

Datasets We will be using a **subset** of an Optical Character Recognition (OCR) dataset. 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 a small dataset with 60 samples *per class* (50 for training and 10 for validation). We will also evaluate your code on a medium dataset with 600 samples per class (500 for training and 100 for validation). Figure 2 shows a random sample of 10 images of a few letters from the dataset (not the same ones we're classifying in this assignment).

File Format Each dataset (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.

6 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, and the hidden layer $\mathbf{z}$ consist of D hidden units. In addition, let the output layer $\hat{\mathbf{y}}$ be a probability distribution over K classes. That is, each element $\hat{y}_k$ of the output vector represents the probability of $\mathbf{x}$ belonging to the class k.

We can 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. 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. $\boldsymbol{\alpha}_{\cdot,0}$ and $\boldsymbol{\beta}_{\cdot,0}$) hold the bias parameters.

$$
\begin{aligned}
a_j &= \sum_{m=0}^{M} \alpha_{j,m} x_m \\
z_j &= \sigma(a_j) = \frac{1}{1+\exp(-a_j)} \\
b_k &= \sum_{j=0}^{D} \beta_{k,j} z_j \\
\hat{y}_k &= \text{Softmax}(\mathbf{b}) = \frac{\exp(b_k)}{\sum_{l=1}^{K}\exp(b_l)}
\end{aligned}
$$

The objective function we will use for training the neural network is the average cross entropy over the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$:

$$
J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{21}
$$

In Equation 21, J is a function of the model parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ because $\hat{y}_k^{(i)}$ is the output of the neural network applied to $\mathbf{x}^{(i)}$ and is therefore implicitly a function of $\mathbf{x}^{(i)}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$. $\hat{y}_k^{(i)}$ and $y_k^{(i)}$ are the kth components of $\hat{\mathbf{y}}^{(i)}$ and $\mathbf{y}^{(i)}$ respectively.

To train, you should optimize this objective function using stochastic gradient descent (SGD), where the gradient of the parameters for each training example is computed via backpropagation. You should shuffle the training points when performing SGD using the provided `shuffle` function, passing in the epoch number as a random seed. Note that SGD has a slight impact on the objective function as we are "summing" over just the current point, i, and not the entire dataset:

$$
J_{SGD}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{22}
$$

You will use the (hopefully at this point) familiar SGD update rule to update the parameters of your model:

$$\boldsymbol{\alpha}_{t+1} \leftarrow \boldsymbol{\alpha}_t - \gamma \frac{\partial J_{SGD}(\boldsymbol{\alpha}_t, \boldsymbol{\beta}_t)}{\partial \boldsymbol{\alpha}_t} \tag{23}$$

$$\boldsymbol{\beta}_{t+1} \leftarrow \boldsymbol{\beta}_t - \gamma \frac{\partial J_{SGD}(\boldsymbol{\alpha}_t, \boldsymbol{\beta}_t)}{\partial \boldsymbol{\beta}_t} \tag{24}$$

where γ is the learning rate, and $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ are the values of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ at step t (similarly for $\boldsymbol{\alpha}_{t+1}$ and $\boldsymbol{\beta}_{t+1}$).

6.1 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 initializations:

RANDOM The weights are initialized randomly from a uniform distribution from -0.1 to 0.1. The bias parameters are initialized to zero.

ZERO All weights are initialized to 0.

You must support both of these initialization schemes.

7 Implementation

Write a program `neuralnet.py` that implements an optical character recognizer using a one hidden layer neural network with sigmoid activations. Your program should learn the parameters of the model on the training data, report the cross-entropy at the end of each epoch on both train and validation data, and at the end of training write out its predictions and error rates on both datasets.

Your implementation must satisfy the following requirements:

- Use a **sigmoid** activation function on the hidden layer and **softmax** on the output layer to ensure it forms a proper probability distribution.
- Number of **hidden units** for the hidden layer should be determined by a command line flag. (More details on command line flags provided below.)
- Support two different **initialization strategies**, as described in Section 6.1, selecting between them via a command line flag.
- Use stochastic gradient descent (SGD) to optimize the parameters for one hidden layer neural network. The number of **epochs** will be specified as a command line flag.
- Set the **learning rate** via a command line flag.
- Perform stochastic gradient descent updates on the training data on the data shuffled with the provided function. For each epoch, you must reshuffle the **original** file data, not the data from the previous epoch.
- You may assume that the input data will always have the same output label space (i.e. $\{0, 1, \ldots, 9\}$). Other than this, do not hard-code any aspect of the datasets into your code. We will autograde your programs on multiple data sets that include different examples.
- In case there is a tie in the output layer $\hat{\mathbf{y}}$, predict the smallest index to be the label. (Hint: you shouldn't need to write extra code for tie-breaking if you are using the appropriate NumPy function.)
- Do *not* use any machine learning libraries. You may use NumPy.

Implementing a neural network can be tricky: the parameters are not just a simple vector, but a collection of many parameters; computational efficiency of the model itself becomes essential; the initialization strategy dramatically impacts overall learning quality; other aspects which we will *not* change (e.g. activation function, optimization method) also have a large effect. These *tips* should help you along the way:

- 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 those calls can be much faster! Those operations are actually implemented in fast C code, which won't get bogged down the way a high-level scripting language like Python will.
- Implement a finite difference test to check whether your implementation of backpropagation 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.

7.1 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python3 neuralnet.py [args...]
```

Where above `[args...]` is a placeholder for nine command-line arguments: `<train_input>` `validation_input>` `<train_out>` `<validation_out>` `<metrics_out>` `<num_epoch>` `<hidden_units>` `<init_flag>` `<learning_rate>`. These arguments are described in detail below:

1. `<train_input>`: path to the training input `.csv` file (see Section 5)
2. `<validation_input>`: path to the validation input `.csv` file (see Section 5)
3. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 7.2)
4. `<validation_out>`: path to output `.labels` file to which the prediction on the *validation* data should be written (see Section 7.2)
5. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and validation error should be written (see Section 7.3)
6. `<num_epoch>`: integer specifying the number of times backpropagation loops through all of the training data (e.g., if `<num_epoch>` equals 5, then each training example will be used in backpropagation 5 times).
7. `<hidden_units>`: positive integer specifying the number of hidden units.
8. `<init_flag>`: integer taking value 1 or 2 that specifies whether to use RANDOM or ZERO initialization (see Section 6.1 and Section 6)—that is, if `init_flag`==1 initialize your weights randomly from a uniform distribution over the range [-0.1, 0.1] (i.e. RANDOM), if `init_flag`==2 initialize all weights to zero (i.e. ZERO). For both settings, **always initialize bias terms to zero**.
9. `<learning_rate>`: float value specifying the learning rate for SGD.
10. `<--debug>`: (optional argument) set the logging level, set to DEBUG to show logging.

As an example, if you implemented your program in Python, the following command line would run your program with 4 hidden units on the small data provided in the handout for 2 epochs using zero initialization and a learning rate of 0.1.

```
python3 neuralnet.py small_train.csv small_validation.csv \
small_train_out.labels small_validation_out.labels \
small_metrics_out.txt 2 4 2 0.1
```

7.2 Output: Labels Files

Your program should write two output `.labels` files containing the predictions of your model on training data (<train_out>) and validation data (<validation_out>). Each should contain the predicted labels for each example printed on a new line. Use `\n` to create a new line.

Your labels should exactly match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution.

Note: You should output your predicted labels using the same *integer* identifiers as the original training data. You should also insert an empty line (using '\n') at the end of each sequence (as is done in the input data files).

7.3 Output Metrics

Generate a file where you report the following metrics:

cross entropy After each epoch, report mean cross entropy on the training data `crossentropy(train)` and validation data `crossentropy(validation)` (See Equation 21). These two cross-entropy values should be reported at the end of each epoch and prefixed by the epoch number. For example, after the second pass through the training examples, these should be prefixed by `epoch=2`. The total number of train losses you print out should equal `num_epoch`—likewise for the total number of validation losses.

error After the final epoch (i.e. when training has completed fully), report the final training error `error(train)` and validation error `error(validation)`.

A sample output is given below. It contains the train and validation losses for the first 2 epochs and the final error rate when using the command given above.

```
epoch=1 crossentropy(train): 2.1415670910950144
epoch=1 crossentropy(validation): 2.1502231738985618
epoch=2 crossentropy(train): 1.8642629963917074
epoch=2 crossentropy(validation): 1.8780601379038728
error(train): 0.73
error(validation): 0.72
```

Take care that your output has the exact same format as shown above. There is an equal sign = between the word `epoch` and the epoch number, but no spaces. There should be a single space after the epoch number (e.g. a space after `epoch=1`), and a single space after the colon preceding the metric value (e.g. a space after `epoch=1 crossentropy(train):`). Each line should be terminated by a Unix line ending `\n`.

```
python3 neuralnet.py small_train.csv small_validation.csv \
small_train_out.labels small_validation_out.labels \
```

```
small_metrics_out.txt 1 4 2 0.1
```

The specific output file names are not important, but be sure to keep the other arguments exactly as they are shown above.

8 Gradescope Submission

You should submit your `neuralnet.py` to Gradescope. **Any other files will be deleted.** Please do not use any other file name for your implementation. This will cause problems for the autograder to correctly detect and run your code.

Make sure to read the autograder output carefully. The autograder for Gradescope prints out some additional information about the tests that it ran. For this programming assignment we've specially designed some buggy implementations that you might implement and will try our best to detect those and give you some more useful feedback in Gradescope's autograder. Make wise use of autograder's output for debugging your code.

Note: For this assignment, you may make up to **10** submissions to Gradescope before the deadline, but only your last submission will be graded.

9 Implementation Details

9.1 Module-based Method of Implementation

Module-based automatic differentiation (AD) is a technique that has long been used to develop libraries for deep learning, and is the method of implementation that you are encouraged to follow in this assignment. Dynamic neural network packages are those that allow a specification of the computation graph dynamically at runtime, such as Torch[1], PyTorch[2], and DyNet[3]—these all employ module-based AD in the sense that we will describe here.[4]

The key idea behind module-based AD is to componentize the computation of the neural-network into layers. Each layer can be thought of as consolidating numerous nodes in the computation graph (a subset of them) into one *vector-valued* node. Such a vector-valued node should be capable of the following and we call each one a **module** (corresponding to a class in Python):

1. Forward computation of output $\mathbf{b} = [b_1, \ldots, b_B]$ given input $\mathbf{a} = [a_1, \ldots, a_A]$ via some differentiable function f. That is, $\mathbf{b} = f(\mathbf{a})$.
2. Backward computation of the gradient of the input $\mathbf{g_a} = \frac{\partial J}{\partial \mathbf{a}} = [\frac{dJ}{da_1}, \ldots, \frac{dJ}{da_A}]$ given the gradient of output $\mathbf{g_b} = \frac{\partial J}{\partial \mathbf{b}} = [\frac{dJ}{db_1}, \ldots, \frac{dJ}{db_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{dJ}{da_i} = \sum_{j=1}^{B} \frac{dJ}{db_j} \frac{\partial b_j}{\partial a_i}$ for all $i \in \{1, \ldots, A\}$.

9.1.1 Module Definitions

In our implementation, the modules we will define for our neural network correspond to a Linear layer and a Sigmoid layer. While it is possible to additionally define modules for Softmax and Cross-Entropy, we keep them as functions for simplicity (though you are welcome to turn them into modules as well if you wish). Each module defines a forward method $\mathbf{b} = *.\textsc{Forward}(\mathbf{a})$, and a backward method $\mathbf{g_a} = *.\textsc{Backward}(\mathbf{g_b})$. In other words, the forward method yields the output, $\mathbf{b}$, given the input, $\mathbf{a}$; meanwhile, the backward method yields the gradient with respect to the input, $\mathbf{g_a}$, given the gradient with respect to the output, $\mathbf{g_b}$. Each module may also store certain values as appropriate (for instance, the Linear layers store the weight matrices $\boldsymbol{\alpha}, \boldsymbol{\beta}$).

Note that for linear modules in particular, while the gradients with respect to the inputs and outputs are passed in and out of the modules, the gradients with respect to the weight matrices, $\mathbf{g}_{\boldsymbol{\alpha}}$ and $\mathbf{g}_{\boldsymbol{\beta}}$ are **not**. This follows the object-oriented design principle of encapsulation – $\mathbf{g}_{\boldsymbol{\alpha}}$ and $\mathbf{g}_{\boldsymbol{\beta}}$ are only required by their respective linear layers, so we only store them within the linear module itself. Later on, they will be used for a SGD update, which will be performed by an additional STEP method. (Alternatively, since the SGD update for this assignment is always applied per example, you may directly perform the SGD update within BACKWARD, though you should be extra careful about the order of your operations.)

Further, if you've completed Written Question 2, you might notice that though we only pass $\mathbf{g_b}$, the gradient with respect to the module output, into $*.\textsc{Backward}(\mathbf{g_b})$, we may need more than that to calculate some of the layer's gradients. Specifically, if you inspect your expressions for the gradient, you may notice that they use certain values from the forward pass. Hence, in your forward methods, you will want to **cache** certain values to be used later on in the backward pass. In the starter code, we do so via a `cache` dictionary

[1] http://torch.ch/
[2] http://pytorch.org/
[3] https://dynet.readthedocs.io
[4] Static neural network packages are those that require a static specification of a computation graph which is subsequently compiled into code. Examples include Theano, Tensorflow, and CNTK. These libraries are also module-based but the particular form of implementation is different from the dynamic method we recommend here.

as a class attribute, wherein you can store parameter names as keys that map to their cached values.

Finally, you'll want to pay close attention to the dimensions that you pass into and return from your modules. The dimensions A and B are specific to the module such that we have input $\mathbf{a} \in \mathbb{R}^A$, output $\mathbf{b} \in \mathbb{R}^B$, gradient of output $\mathbf{g_a} \triangleq \nabla_{\mathbf{a}} J \in \mathbb{R}^A$, and gradient of input $\mathbf{g_b} \triangleq \nabla_{\mathbf{b}} J \in \mathbb{R}^B$.

We have provided you the pseudocode for the Linear Module as an example.

Linear Module

---

procedure FORWARD($\mathbf{a}$)
    Compute $\mathbf{b}$ using this layer's weight matrix
    Cache intermediate value(s) for the backward pass ▷ See Written Question 1.2(c)
    return $\mathbf{b}$
procedure BACKWARD($\mathbf{g_b}$)
    Bring the necessary cached values into scope
    Compute $\mathbf{g}_{\boldsymbol{\alpha}}$
    Compute $\mathbf{g_a}$
    Store $\mathbf{g}_{\boldsymbol{\alpha}}$ for subsequent SGD update
    return $\mathbf{g_a}$
procedure STEP( )
    Apply SGD update to weights α using stored gradient $\mathbf{g}_{\boldsymbol{\alpha}}$

9.1.2 Module-based AD for Neural Network

Given that our one hidden layer neural network is a composition of modules, we can define functions for forward and backward propagation using these modules as follows:

Algorithm 1 Forward Computation

procedure NNFORWARD(Training example ($\mathbf{x}$, $\mathbf{y}$))
    $\mathbf{a} = \text{LINEAR1.FORWARD}(\mathbf{x})$ ▷ First linear layer module
    $\mathbf{z} = \text{SIGMOID.FORWARD}(\mathbf{a})$ ▷ Sigmoid activation module
    $\mathbf{b} = \text{LINEAR2.FORWARD}(\mathbf{z})$ ▷ Second linear layer module
    $\hat{\mathbf{y}} = \text{SOFTMAX}(\mathbf{b})$ ▷ Softmax *function*
    $J = \text{CROSSENTROPY}(\mathbf{y}, \hat{\mathbf{y}})$ ▷ CrossEntropy *function*
    return $J, \hat{\mathbf{y}}$

Algorithm 2 Backpropagation

procedure NNBACKWARD($\mathbf{y}$, $\hat{\mathbf{y}}$)
    $g_J = \frac{\partial J}{\partial J} = 1$ ▷ Base case
    $\mathbf{g_b} = \text{DSOFTMAXCROSSENTROPY}(\mathbf{y}, \hat{\mathbf{y}}, g_J)$ ▷ See Written Question 1.2(b)
    $\mathbf{g_z} = \text{LINEAR2.BACKWARD}(\mathbf{g_b})$
    $\mathbf{g_a} = \text{SIGMOID.BACKWARD}(\mathbf{g_z})$
    $\mathbf{g_x} = \text{LINEAR1.BACKWARD}(\mathbf{g_a})$ ▷ We discard $\mathbf{g_x}$

Here's the big takeaway: we can actually view these two functions as themselves defining another module! It is a 1-hidden layer neural network module. That is, the cross-entropy of the neural network for a single training example is *itself* a differentiable function and we know how to compute the gradients of its inputs, given the gradients of its outputs.

9.2 Training Procedure

Consider the neural network described in Section 6 applied to the ith 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 $h_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ 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 $(\mathbf{x}, \mathbf{y})$ as well as 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 stochastic gradient descent (SGD). Because we want the behavior of your program to be approximately deterministic for testing on Gradescope, we will require that (1) you should use *our provided* shuffle function to shuffle your data at the start of each epoch and (2) you will use a fixed learning rate.

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

Algorithm 3 Training with Stochastic Gradient Descent (SGD)

procedure SGD(Training data $\mathcal{D}_{train}$, test data $\mDt$)
    Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ ▷ Use either RANDOM or ZERO from Section 6.1
    for $e \in \{1, 2, \ldots, E\}$ **do** ▷ For each epoch
        $\mathcal{D} = \text{SHUFFLE}(\mathcal{D}_{train}, e)$
        for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do** ▷ For each training example
            Compute neural network forward prop:
            $J, \hat{y} = \text{NN.FORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
            Compute gradients via backprop:
$$\left.\begin{aligned} \mathbf{g}_{\boldsymbol{\alpha}} &= \frac{\partial J}{\partial \boldsymbol{\alpha}} \\ \mathbf{g}_{\boldsymbol{\beta}} &= \frac{\partial J}{\partial \boldsymbol{\beta}} \end{aligned}\right\} \text{given by NN.BACKWARD}(\mathbf{y}, \hat{\mathbf{y}})$$
            Update parameters with SGD updates $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$:
            $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
        Evaluate training mean cross-entropy $J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
        Evaluate test mean cross-entropy $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
    return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$

9.3 Test Time Procedure

At test time, we output the most likely prediction for each example:

Algorithm 4 Prediction at Test Time

procedure PREDICT(Unlabeled train or test dataset $\mathcal{D}'$)
    for $\mathbf{x} \in \mathcal{D}'$ **do**
        Compute neural network prediction $\hat{\mathbf{y}} = h(\mathbf{x})$
        Predict the label with highest probability $l = \text{argmax}_k\, \hat{y}_k$