HOMEWORK 5: NEURAL NETWORKS

10-601 Introduction to Machine Learning (Spring 2018)
Carnegie Mellon University
https://piazza.com/cmu/spring2018/10601
OUT: Feb 27, 2018[*]
DUE: March 9, 2018 11:59 PM
TAs: Abhijeet, Eti, Sarah, Shawn

Summary In this assignment, you will build a handwriting recognition system using a neural network. As a warmup, Section 1 will lead you through an on-paper example of how to implement a neural network. Then, in Section 2, you will implement an end-to-end system that learns to perform handwritten letter classification.

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 3.4"). 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 collaboration policy on the website for more information: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions, and Autolab to submit your code. Please follow instructions at the end of this PDF to correctly submit all your code to Autolab.
  - **Gradescope:** For written problems such as derivations, proofs, or plots we will be using Gradescope (https://gradescope.com/). Please use ther 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. Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed on a separate page.
  - **Autolab:** You will submit your code for programming questions on the homework to Autolab (https://autolab.andrew.cmu.edu/). After uploading your code, our grading

[*]Compiled on Sunday 4th March, 2018 at 02:24

scripts will autograde your assignment by running your program on a virtual machine (VM). The software installed on the VM is identical to that on `linux.andrew.cmu.edu`, so you should check that your code runs correctly there. If developing locally, check that the version number of the programming language environment (e.g. Python 2.7, Octave 3.8.2, OpenJDK 1.8.0, g++ 4.8.5) and versions of permitted libraries (e.g. `numpy` 1.7.1) match those on `linux.andrew.cmu.edu`. (Octave users: Please make sure you do not use any Matlab-specific libraries in your code that might make it fail against our tests.) You have a **total of 10 Autolab submissions**. Use them wisely. In order to not waste Autolab submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before any Autolab submission.

- **Materials:** Download from Autolab the tar file ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment.

1 Written Questions [20 points]

Answer the following questions in the HW5 solutions template provided. Then upload your solutions to Gradescope. You may use LaTeX or print the template and hand-write your answers then scan it in. Failure to use the template may result in a penalty.

1.1 Example Feed Forward and Backpropagation [10 points]

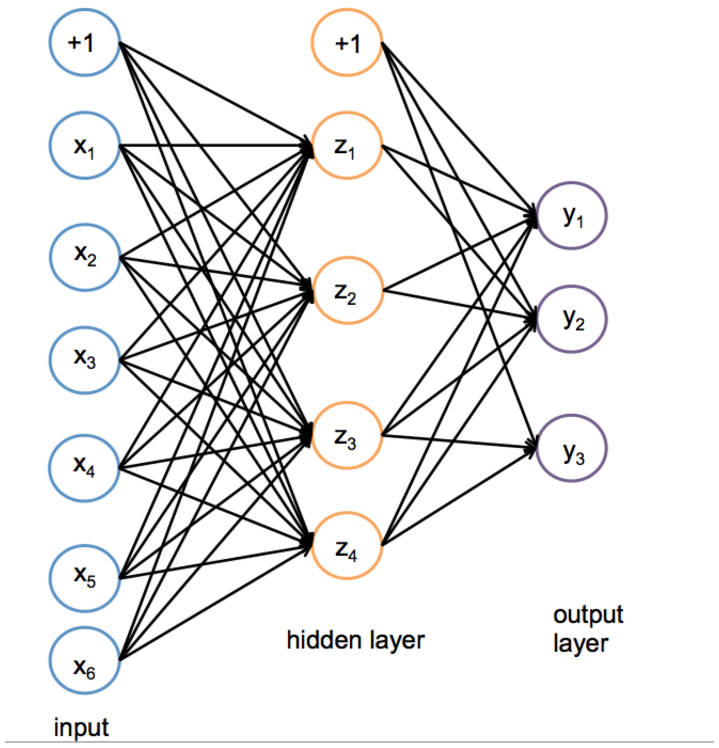

Figure 1.1: A One Hidden Layer Neural Network

Consider the neural network with one hidden layer shown in Figure 1.1. The inputs have 6 features $(x_1, ..., x_6)$, the hidden layer has 4 nodes $(z_1, ..., z_4)$, and the output is a probability distribution (y_1, y_2, y_3) over 3 classes . We also add a bias to the input, x_0, as well as to the hidden layer, z_0 and set them to 1. $\boldsymbol{\alpha}$ 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.

Equivalently, we define each of the following. The input:

$$\mathbf{x} = (x_1, x_2, x_3, x_4, x_5, x_6) \tag{1.1}$$

Linear combination at first (hidden) layer:

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

Activation at first (hidden) layer:

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

Linear combination at second (output) layer:

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

Activation at second (output) layer:

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

Note that the linear combination equations can be written equivalently as the product of the transpose 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 $z_0 = 1$.

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:

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{3} y_i \log(\hat{y}_i) \tag{1.6}$$

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. **[4 points]** We initialize the weights as:

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

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

And weights on the bias terms ($\alpha_{j,0}$ and $\beta_{j,0}$) are initialized to 1.

You are given a training example $x^{(1)} = (1,1,0,0,1,1)$ with label class 2, so $y^{(1)} = (0,1,0)$. Using the initial weights, run the feed forward of the network over this example (without rounding) and then answer the following questions. In your responses, round to four decimal places (if the answer is an integer you need not include trailing zeros). (Note: the superscript (1) simply indicates that a value corresponds to using training example $x^{(1)}$)

(a) What is $a_1^{(1)}$?

(b) What is $z_1^{(1)}$?

(c) What is $a_3^{(1)}$?

(d) What is $z_3^{(1)}$?

(e) What is $b_2^{(1)}$?

(f) What is $\hat{y}_2^{(1)}$?

(g) Which class would we predict on this example?

(h) What is the total loss on this example?

2. **[5 points]** 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 without rounding then answer the following questions, then in your responses, round to four decimal places

   (a) What is the updated value of $\beta_{2,1}$?

   (b) What is the updated weight of the hidden layer bias term applied to y_1 (e.g. $\beta_{1,0}$)?

   (c) What is the updated value of $\alpha_{3,4}$?

   (d) What is the updated weight of the input layer bias term applied to z_2 (e.g. $\alpha_{2,0}$)?

   (e) Once we've updated all of our weights if we ran feed forward over the same example again, which class would we predict?

3. **[1 points]** Suppose you are now given a collection of training examples $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$. Explain in English why the cross-entropy loss averaged over these examples is exactly the same quantity as the negative average conditional log-likelihood of the data.

1.2 Empirical Questions [10 points]

The following questions should be completed after you work through the programming portion of this assignment (Section 2).

For these questions, use the large dataset.

Use the following values for the hyperparameters unless otherwise specified:

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

Table 1.1: Default values of hyperparameters for experiments in Section 1.2.

For the following questions, submit your solutions to Gradescope. Do **not** include any visualization-related code when submitting to Autolab! Note: we expect it to take about **5 minutes** to train each of these networks.

4. **[4 points]** Train a single hidden layer neural network using the hyperparameters mentioned in Table 1.1, 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. On the same figure, plot the average validation cross-entropy.

5. **[1 points]** Examine and comment on the the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?

6. **[4 points]** Train a single hidden layer neural network using the hyperparameters mentioned in Table 1.1, 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. On the same figure, plot the average validation cross-entropy loss. You may make a separate figure for each learning rate.

7. **[1 points]** 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?

2 Programming [80 points]

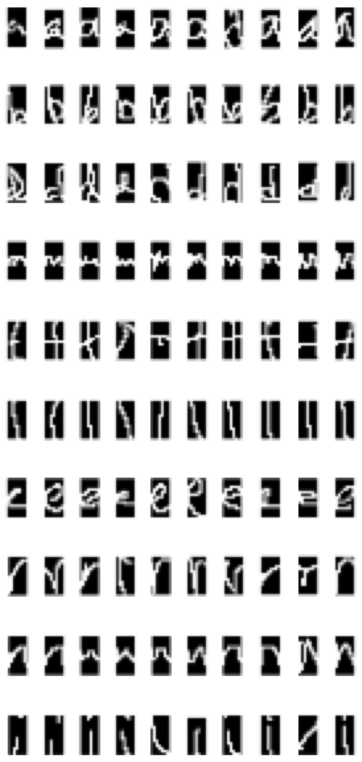

Figure 2.1: 10 Random Images of Each of 10 Letters in OCR

Your goal in this assignment is to label images of handwritten letters 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 programs you write will be automatically graded using the Autolab system. You may write your programs in **Octave, Python, Java, or C++**. However, you should use the same language for all parts below.

2.1 The Task and Datasets

Materials Download the tar file from Autolab ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment.

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 three datasets drawn from this data: a small dataset with 60 samples per class (50 for training and 10 for validation), a medium dataset with 600 samples per class (500 for training and 100 for validation), and a large dataset with 1000 samples per class (900 for training and 100 for validation). Figure 2.1 shows a random sample of 10 images of few letters from the dataset.

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 2.1 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.

2.2 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.

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

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.

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

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

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{2.1}$$

In Equation 2.1, J is a function of the model parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ because $\hat{y}_k$ is implicitly a function of $\mathbf{x}^{(i)}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ since it is the output of the neural network applied to $\mathbf{x}^{(i)}$. Of course, $\hat{y}_k$ and $y_k^{(i)}$ are the kth components of $\hat{\mathbf{y}}$ 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.

2.2.1 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 initialization:

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.

2.3 Implementation

Write a program `neuralnet.{py|java|cpp|m}` 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.
- Support two different **initialization strategies**, as described in Section 2.2.1, 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 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.
- You may assume that the input data will always have the same *number* of features (i.e. number of columns) and the same output label space (i.e. $\{0, 1, \ldots, 9\}$). Other than these assumptions, do not hard-code any aspects of the data sets into your code. We will autograde your programs on multiple (hidden) data sets that include different examples.
- Do *not* use any machine learning libraries. You may use supported linear algebra packages. See Section 2.3.1 for more details.

Implementing a neural network can be tricky: the parameters are not just a simple vector, but a collection of many paramters; computational efficiency of the model itself becomes essential; the initialization strategy dramatically impacts overall learning quality; other aspects 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 and Octave. 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.
- For low level languages such as Java/C++, the use of primitive arrays and for-loops would not pose any computational efficiency problems—however, it is still helpful to make use of a linear algebra library to cut down on the number of lines of code you will write.
- 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 Autolab—since it will otherwise slow down your code.

2.3.1 Command Line Arguments

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

For Python:
```
$ python neuralnet.py [args...]
```

For Java:
```
$ javac -cp "./lib/ejml-v0.33-libs/*:./" neuralnet.java
$ java -cp "./lib/ejml-v0.33-libs/*:./" neuralnet [args...]
```

For C++:
```
$ g++ -g -std=c++11 -I./lib neuralnet.cpp; ./a.out [args...]
```

For Octave:
```
$ octave -qH neuralnet.m [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 2.1)
2. `<validation_input>`: path to the validation input `.csv` file (see Section 2.1)
3. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 2.3.2)
4. `<validation_out>`: path to output `.labels` file to which the prediction on the *validation* data should be written (see Section 2.3.2)
5. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and validation error should be written (see Section 2.3.3)
6. `<num_epoch>`: integer specifying the number of times backpropogation loops through all of the training data (e.g., if `<num_epoch>` equals 5, then each training example will be used in backpropogation 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 2.2.1 and Section 2.3)—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.

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.

```
$ python neuralnet.py smalltrain.csv smallvalidation.csv \
model1train_out.labels model1val_out.labels model1metrics_out.txt \
2 4 2 0.1
```

Linear Algebra Libraries When implementing a neural network, it is often more convenient to have a linear algebra library at your disposal. In this assignment, Java users may use EJML[a] and C++ users Eigen[b]. Details below. (As usual, Python users have numpy; Octave users have built-in matrix support.)

Java EJML is a pure Java linear algebra package with three interfaces. We strongly recommend using the SimpleMatrix interface. Autolab will use EJML version 3.3. The command line arguments above demonstrate how we will call you code. The classpath inclusion `-cp "./lib/ejml-v0.33-libs/*:./"` will ensure that all the EJML jars are on the classpath as well as your code.

C++ Eigen is a header-only library, so there is no linking to worry about—just `#include` whatever components you need. Autolab will use Eigen version 3.3.4. The command line arguments above demonstrate how we will call you code. The argument `-I./lib` will include the `lib/Eigen` subdirectory, which contains all the headers.

We have included the correct versions of EJML/Eigen in the handout.tar for your convenience. Do **not** include EJML or Eigen in your Autolab submission tar; the autograder will ensure that they are in place.

[a] `https://ejml.org`
[b] `http://eigen.tuxfamily.org/`

2.3.2 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 (again using \n) at the end of each sequence (as is done in the input data files). The first few lines of the predicted labels for the validation dataset is given below

```
6
4
8
8
```

2.3.3 Output Metrics

Generate a file where you report the following metrics:

cross entropy After each Stochastic Gradient Descent (SGD) epoch, report mean cross entropy on the training data `crossentropy(train)` and validation data `crossentropy(validation)` (See Equation 2.1). 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): 2.18506276114
epoch=1 crossentropy(validation): 2.18827302588
epoch=2 crossentropy(train): 1.90103257727
epoch=2 crossentropy(validation): 1.91363803461
error(train): 0.77
error(validation): 0.78
```

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 likelihood(train):`). Each line should be terminated by a Unix line ending `\n`.

2.4 Autolab Submission

You must submit a .tar file named `neuralnet.tar` containing `neuralnet.{py|m|java|cpp}`. You can create that file by running:

```
tar -cvf neuralnet.tar neuralnet.{py|m|java|cpp}
```

from the directory containing your code.

Some additional tips: **DO NOT** compress your files; you are just creating a tarball. Do not use tar `-czvf`. **DO NOT** put the above files in a folder and then tar the folder. Autolab is case sensitive, so observe that all your files should be named in **lowercase**. You must submit this file to the corresponding homework link on Autolab. The autograder for Autolab prints out some additional information about the tests that it ran. You can view this output by selecting ”Handin History” from the menu and then clicking one of the scores you received for a submission. For example on this assignment, among other things, the autograder will print out which language it detects (e.g. Python, Octave, C++, Java).

Python3 Users: Please include a blank file called python3.txt (case-sensitive) in your tar submission and we will execute your submitted program using Python 3 instead of Python 2.7.

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

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 SGD for Neural Networks

Consider the neural network described in Section 2.3 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 $h_{\boldsymbol{\alpha},\boldsymbol{\beta}}(\cdot)$ 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 stochastic 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.

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

Algorithm 1 Stochastic Gradient Descent (SGD)

```
 1: procedure SGD(Training data $\mathcal{D}$, Validation data $\mathcal{D}_v$)
 2:     Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$          ▷ Use either RANDOM or ZERO from Section 2.2.1
 3:     for $e \in \{1, 2, \dots, E\}$ do                                   ▷ For each epoch
 4:         for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ do                     ▷ For each training example
 5:             Compute neural network layers:
 6:             $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
 7:             Compute gradients via backprop:
 8:             $\left.\begin{aligned} \mathbf{g}_{\boldsymbol{\alpha}} &= \nabla_{\boldsymbol{\alpha}} J \\ \mathbf{g}_{\boldsymbol{\beta}} &= \nabla_{\boldsymbol{\beta}} J \end{aligned}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$
 9:             Update parameters:                      update for each sample
10:             $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
11:             $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
12:         Evaluate training mean cross-entropy $J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
13:         Evaluate validation mean cross-entropy $J_{\mathcal{D}_v}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
14:     return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$
```

The functions NNFORWARD and NNBACKWARD are described in Algorithms 3 and 4 respectively. At test time, we output the most likely prediction for each example:

Algorithm 2 Prediction at Test Time

```
1: procedure PREDICT(Unlabeled train or validation dataset $\mathcal{D}'$, Parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$)
2:     for $\mathbf{x} \in \mathcal{D}'$ do
3:         Compute neural network prediction $\hat{\mathbf{y}} = h(\mathbf{x})$
4:         Predict the label with highest probability $l = \text{argmax}_k \, \hat{y}_k$
```

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

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{10} & \alpha_{11} & \dots & \alpha_{1M} \\ \alpha_{20} & \alpha_{21} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{D0} & \alpha_{D1} & \dots & \alpha_{DM} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\alpha}} = \nabla_{\boldsymbol{\alpha}} J = \begin{bmatrix} \frac{dJ}{d\alpha_{10}} & \frac{dJ}{d\alpha_{11}} & \dots & \frac{dJ}{d\alpha_{1M}} \\ \frac{dJ}{d\alpha_{20}} & \frac{dJ}{d\alpha_{21}} & \dots & \frac{dJ}{d\alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dJ}{d\alpha_{D0}} & \frac{dJ}{d\alpha_{D1}} & \dots & \frac{dJ}{d\alpha_{DM}} \end{bmatrix} \tag{A.1}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{10} & \beta_{11} & \dots & \beta_{1D} \\ \beta_{20} & \beta_{21} & \dots & \beta_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{K0} & \beta_{K1} & \dots & \beta_{KD} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J = \begin{bmatrix} \frac{dJ}{d\beta_{10}} & \frac{dJ}{d\beta_{11}} & \dots & \frac{dJ}{d\beta_{1D}} \\ \frac{dJ}{d\beta_{20}} & \frac{dJ}{d\beta_{21}} & \dots & \frac{dJ}{d\beta_{2D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dJ}{d\beta_{K0}} & \frac{dJ}{d\beta_{K1}} & \dots & \frac{dJ}{d\beta_{KD}} \end{bmatrix} \tag{A.2}$$

Observe that we have (in a rather *tricky* fashion) defined the matrices such that both $\boldsymbol{\alpha}$ and $\mathbf{g}_{\boldsymbol{\alpha}}$ are $D \times (M+1)$ matrices. Likewise, $\boldsymbol{\beta}$ and $\mathbf{g}_{\boldsymbol{\beta}}$ are $K \times (D+1)$ matrices. The $+1$ comes from the extra columns $\boldsymbol{\alpha}_{\cdot,0}$ and $\boldsymbol{\beta}_{\cdot,0}$ which are the bias parameters for the first and second layer respectively. We will always assume $x_0 = 1$ and $z_0 = 1$. This should greatly simplify your implementation as you will see in Section A.3.

A.2 Recursive Derivation of Backpropagation

In class, we described a very general approach to differentiating arbitrary functions: backpropagation. One way to understand *how* we go about deriving the backpropagation algorithm is to consider the natural consequence of recursive application of the chain rule.

In practice, the partial derivatives that we need for learning are $\frac{dJ}{d\alpha_{ij}}$ and $\frac{dJ}{d\beta_{kj}}$.

A.2.1 Symbolic Differentiation

Note In this section, we motivate backpropagation via a strawman: that is, we will work through the *wrong* approach first (i.e. symbolic differentiation) in order to see why we want a more efficient method (i.e. backpropagation). Do **not** use this symbolic differentiation in your code.

Suppose we wanted to find $\frac{dJ}{d\alpha_{ij}}$ using the method we know from high school calculus. That is, we will analytically solve for an equation representing that quantity.

1. Considering the computational graph for the neural network, we observe that α_{ij} has exactly one child $a_j = \sum_{m=0}^{M} \alpha_{jm} x_m$. That is a_j is the *first and only* intermediate quantity that uses α_{ji}. Applying the chain rule, we obtain
$$\frac{dJ}{d\alpha_{ij}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ij}} = \frac{dJ}{da_j} x_i$$

2. So far so good, now we just need to compute $\frac{dJ}{da_j}$. Not a problem! We can just apply the chain rule again. a_j just has exactly one child as well, namely $z_j = \sigma(a_j)$. The chain rule gives us that

$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j} = \frac{dJ}{dz_j} z_j(1-z_j)$. Substituting back into the equation above we find that

$$\frac{dJ}{d\alpha_{ij}} = \frac{dJ}{dz_j}(z_j(1-z_j))x_i$$

3. How do we get $\frac{dJ}{dz_j}$? You guessed it: apply the chain rule yet again. This time, however, there are *multiple* children of z_j in the computation graph; they are $b_1, b_2, \dots b_K$. Applying the chain rule gives us that $\frac{dJ}{dz_j} = \sum_{k=1}^{K} \frac{dJ}{db_k}\frac{db_k}{dz_j} = \sum_{k=1}^{K} \frac{dJ}{db_k}\beta_{kj}$. Substituting back into the equation above gives:

$$\frac{dJ}{d\alpha_{ij}} = \sum_{k=1}^{K} \frac{dJ}{db_k}\beta_{kj}(z_j(1-z_j))x_i$$

4. Next we need $\frac{dJ}{db_k}$, which we again obtain via the chain rule: $\frac{dJ}{db_k} = \sum_{l=1}^{K} \frac{dJ}{d\hat{y}_l}\frac{d\hat{y}_l}{db_k} = \sum_{l=1}^{K} \frac{dJ}{d\hat{y}_l}\hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k)$. Substituting back in above gives:

$$\frac{dJ}{d\alpha_{ij}} = \sum_{k=1}^{K}\sum_{l=1}^{K} \frac{dJ}{d\hat{y}_l}\hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k)\beta_{kj}(z_j(1-z_j))x_i$$

5. Finally, we know that $\frac{dJ}{d\hat{y}_l} = -\frac{y_l}{\hat{y}_l}$ which we can again substitute back in to obtain our final result:

$$\frac{dJ}{d\alpha_{ij}} = \sum_{k=1}^{K}\sum_{l=1}^{K} -\frac{y_l}{\hat{y}_l}\hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k)\beta_{kj}(z_j(1-z_j))x_i$$

   Although we have successfully derived the partial derivative w.r.t. α_{ij}, the result is far from satisfying. It is overly complicated and requires deeply nested for-loops to compute.

The above is an example of **symbolic differentiation**. That is, at the end we get an equation representing the partial derivative w.r.t. α_{ij}. At this point, you should be saying to yourself: What a mess! Isn't there a better way? Indeed there is and its called backpropagation. The algorithm works just like the above symbolic differentiation except that we *never* subsitute the partial derivative from the previous step back in. Instead, we work "backwards" through the steps above computing partial derivatives in a top-down fashion.

A.2.2 Backpropagation

The backpropagation algorithm for the neural network used in this assignment is shown below. We proceed first through steps 1 through 5 to compute the cross-entropy of a given training examples $(\mathbf{x}, \mathbf{y})$ using parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. This is the *forward* computation. Next we work through steps 6 through 10 in order to compute the partial derivatives of the parameters $\frac{dJ}{d\alpha_{ij}}$ and $\frac{dJ}{d\beta_{kj}}$.

Forward	Backward	
5. $J = -\sum_{k=1}^{K} y_k \log \hat{y}_k$	6. $\frac{dJ}{d\hat{y}_k} = -\frac{y_k}{\hat{y}_k}$	
4. $\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)}$	7. $\frac{dJ}{db_k} = \sum_{l=1}^{K} \frac{dJ}{d\hat{y}_l} \frac{d\hat{y}_l}{db_k},$	$\frac{d\hat{y}_l}{db_k} = \hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k)$
3. $b_k = \sum_{j=0}^{D} \beta_{kj} z_j$	8. $\frac{dJ}{d\beta_{kj}} = \frac{dJ}{db_k} \frac{db_k}{d\beta_{kj}},$	$\frac{db_k}{d\beta_{kj}} = z_j$
	$\frac{dJ}{dz_j} = \sum_{k=1}^{K} \frac{dJ}{db_k} \frac{db_k}{dz_j},$	$\frac{db_k}{dz_j} = \beta_{kj}$
2. $z_j = \frac{1}{1 + \exp(-a_j)}$	9. $\frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j},$	$\frac{dz_j}{da_j} = z_j(1 - z_j)$
1. $a_j = \sum_{m=0}^{M} \alpha_{jm} x_m$	10. $\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}},$	$\frac{da_j}{d\alpha_{ji}} = x_i$

Figure A.1: Backpropagation for 1-hidden layer neural network

Notice in step 8 above, that we compute partial derivatives for both $\frac{dJ}{d\beta_{kj}}$ and $\frac{dJ}{dz_j}$. This is because there are two types of inputs needed to compute b_k in the forward pass. By contrast, in step 10, we only compute $\frac{dJ}{d\alpha_{ji}}$ because $\frac{dJ}{dx_m}$ is not needed.

Below, we rewrite the same computation, but with two simplifications:

1. We substitute the quantities from the third column above into the second column above. This yields a single "Backward" column below.
2. Below, we denote each of the intermediate partial derivatives that we need by a named variable. Specifically, for any node v in the computation graph we let $g_v \equiv \frac{dJ}{dv}$.

Note that Figure A.1 and Figure A.2 are identical except for these changes. This yields the following version of the backpropagation algorithm.

Forward

5. $J = -\sum_{k=1}^{K} y_k \log \hat{y}_k$

4. $\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)}$

3. $b_k = \sum_{j=0}^{D} \beta_{kj} z_j$

2. $z_j = \frac{1}{1 + \exp(-a_j)}$

1. $a_j = \sum_{m=0}^{M} \alpha_{jm} x_m$

Backward

6. $g_{\hat{y}_k} = -\frac{y_k}{\hat{y}_k}$

7. $g_{b_k} = \sum_{l=1}^{K} g_{\hat{y}_l} \hat{y}_l (\mathbb{I}[k = l] - \hat{y}_k)$

8. $g_{\beta_{kj}} = g_{b_k} z_j$

$g_{z_j} = \sum_{k=1}^{K} g_{b_k} \beta_{kj}$

9. $g_{a_j} = g_{z_j} z_j (1 - z_j)$

10. $g_{\alpha_{ji}} = g_{a_j} x_i$

Figure A.2: Backpropagation for 1-hidden layer neural network (with intermediate variables)

A.3 Matrix / Vector Operations for Neural Networks

Some programming languages are fast and some are slow. Below is a simple benchmark to show this concretely. The task is to compute a dot-product $\mathbf{a}^T\mathbf{b}$ between two vectors $\mathbf{a} \in \mathbb{R}^{500}$ and $\mathbf{b} \in \mathbb{R}^{500}$ one thousand times. Table A.1 shows the time taken for several combinations of programming language and data structure.

language	data structure	time (ms)
Python	list	200.99
Python	`numpy` array	1.01
Java	`float[]`	4.00
C++	`vector<float>`	0.81

Table A.1: Computation time required for dot-product in various languages.

Notice that Java[1] and C++ with standard data structures are quite efficient. By contrast, Python differs dramatically depending on which data structure you use: with a standard list object (e.g. `a = [float(i) for x in range(500)]`) the computation time is an appallingly slow 200+ milliseconds. Simply by switching to a numpy array (e.g. `a = np.arange(500, dtype=float)`) we obtain a 200x speedup. This is because a numpy array is actually carrying out the dot-product computation in pure C, which is just as fast as our C++ benchmark, modulo some Python overhead.

Thus, for this assignment, Java and C++ programmers could easily implement the entire neural network using standard data structures and some for-loops. However, Python or Octave programmers would find that

[1]Java would approach the speed of C++ if we had given the just-in-time (JIT) compiler inside the JVM time to "warm-up".

their code is simply too slow if they tried to do the same. As such, particularly for Python and Octave users, one must convert all the deeply nested for-loops into efficient "vectorized" math via `numpy`. Doing so will ensure efficient code. Java and C++ programmers can also benefit from linear algebra packages since it can cut down on the total number of lines of code you need to write.

A.4 Procedural Method of Implementation

Perhaps the simplest way to implement a 1-hidden-layer neural network is procedurally. Note that this approach has some drawbacks that we'll discuss below (Section A.4.2).

The procedural method is simply the one we derived in Section A.2: one function computes the outputs of the neural network and all intermediate quantities $\mathbf{o} = \text{NNF}{\scriptsize\text{ORWARD}}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J)$, where the object is just some struct (i.e. steps 1 - 5 below). Then another function computes the gradients of our parameters $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}} = \text{NNB}{\scriptsize\text{ACKWARD}}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$, where $\mathbf{o}$ is a data structure that stores all the forward computation (i.e. steps 6 - 11 below). Here we describe the same computation shown in Figures A.1 and A.2, but this time we have shown how it could be carried out using matrix/vector operations.

Forward	Backward
5. $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$	6. $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$ kx1. 1xk
4. $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$	7. $\mathbf{g}_{\mathbf{b}} = \overset{1xk}{\mathbf{g}_{\hat{\mathbf{y}}}^T} \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$ kxk
3. $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$	8. $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}\mathbf{z}^T$
	$\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}$
2. $\mathbf{z} = \sigma(\mathbf{a})$	10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
1. $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$	11. $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}}\mathbf{x}^T$

Figure A.3: Backpropagation for 1-hidden layer neural network (with matrix/vector operations)

Above $\odot$ denotes element-wise multiplication and $\div$ denotes element-wise division between two vectors. For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that $\text{diag}(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero. $\sigma(\mathbf{a})$ denotes element-wise application of the sigmoid function and $\text{softmax}(\mathbf{b})$ applies the softmax function.

One must be careful to ensure that the sigmoid and softmax functions are *also* vectorized. For example, the sigmoid function can be efficiently computed as

$$\sigma(\mathbf{a}) = \mathbf{1} \div (\mathbf{1} + \exp(-\mathbf{a})) \tag{A.3}$$

where $\mathbf{1}$ is a column vector of all ones, and $\exp$ is (efficiently) applied element-wise to the negated vector $\mathbf{a}$. All of these operations should avoid for-loops when working in a high-level language like Python / Octave. We can compute the softmax function in a similar vectorized manner as,

$$\text{softmax}(\mathbf{b}) = \exp(\mathbf{b}) \div \text{sum}(\exp(\mathbf{b})) \tag{A.4}$$

where the function $\text{sum}(\cdot)$ returns the sum of the entires in the vector.

A.4.1 Algorithm Definitions

We can write the above computation as two functions, NNFORWARD() and NNBACKWARD(). These two functions complete the learning algorithm presented in Algorithm 1.

Algorithm 3 Forward Computation

```
1: procedure NNFORWARD(Training example ($\mathbf{x}$, $\mathbf{y}$), Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
2:     $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$
3:     $\mathbf{z} = \sigma(\mathbf{a})$
4:     $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$
5:     $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
6:     $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
7:     $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
8:     return intermediate quantities $\mathbf{o}$
```

Algorithm 4 Backpropagation

```
1: procedure NNBACKWARD(Training example ($\mathbf{x}$, $\mathbf{y}$), Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
2:     Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
3:     $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
4:     $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
5:     $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}\mathbf{z}^T$
6:     $\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}$
7:     $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
8:     $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}}\mathbf{x}^T$
9:     return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$
```

A.4.2 Drawbacks to Procedural Method

As noted in Section A.6, it is possible to use a finite difference method to check that the backpropagation algorithm is correctly computing the gradient of its corresponding forward computation. We *strongly* encourage you to do this.

There is a big problem however: what if your finite difference check informs you that the gradient is *not* being computed correctly. How will you know *which* part of your NNFORWARD() or NNBACKWARD() functions has a bug? There are two possible solutions here:

1. As usual, you can (and should) work through a tiny example dataset on paper. Compute each intermediate quantity and each gradient. Check that your code reproduces each number. The one that does not should indicate where to find the bug.

2. Replace your procedural implementation with a module-based one (as described in Section A.5) and then run a finite-difference check on *each* layer of the model individually. The finite-difference check that fails should indicate where to find the bug.

Of course, rather than waiting until you have a bug in your procedural implementation, you could jump straight to the module-based version—though it increases the complexity slightly (i.e. more lines of code) it *might* save you some time in the long run.

A.5 Module-based Method of Implementation

Module-based automatic differentiation (AD) is a technique that has long been used to develop libraries for deep learning. Dynamic neural network packages are those that allow a specification of the computation graph dynamically at runtime, such as Torch[2], PyTorch[3], and DyNet[4]—these all employ module-based AD in the sense that we will describe here.[5]

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**:

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

A.5.1 Module Definitions

The modules we would define for our neural network would correspond to a Linear layer, a Sigmoid layer, a Softmax layer, and a Cross-Entropy layer. Each module defines a forward function $\mathbf{b} = $ *FORWARD($\mathbf{a}$), and a backward function $\mathbf{g_a} = $ *BACKWARD($\mathbf{a}, \mathbf{b}, \mathbf{g_b}$) method. These methods accept parameters if appropriate. 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 $\mathbf{g_b} \triangleq \nabla_{\mathbf{b}} J \in \mathbb{R}^B$.

Sigmoid Module The sigmoid layer has only one input vector $\mathbf{a}$. Below σ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \dots, u_M v_M]$.

```
1: procedure SIGMOIDFORWARD($\mathbf{a}$, $\boldsymbol{\alpha}$)
2:     $\mathbf{b} = \sigma(\mathbf{a})$
3:     return $\mathbf{b}$
4: procedure SIGMOIDBACKWARD($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
5:     $\mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
6:     return $\mathbf{g_a}$
```

Softmax Module The softmax layer has only one input vector $\mathbf{a}$. For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that diag($\mathbf{v}$) returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \dots, v_D$ and whose non-diagonal entries are zero.

```
1: procedure SOFTMAXFORWARD($\mathbf{a}$, $\boldsymbol{\alpha}$)
2:     $\mathbf{b} = \text{softmax}(\mathbf{a})$
3:     return $\mathbf{b}$
4: procedure SOFTMAXBACKWARD($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
5:     $\mathbf{g_a} = \mathbf{g_b}^T \left(\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
```

[2] http://torch.ch/
[3] http://pytorch.org/
[4] https://dynet.readthedocs.io
[5] 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.

```
6:     return $\mathbf{g_a}$
```

Linear Module The linear layer has two inputs: a vector $\mathbf{a}$ and parameters $\omega \in \mathbb{R}^{B \times A}$. The output $\mathbf{b}$ is not used by LINEARBACKWARD, but we pass it in for consistency of form.

```
1: procedure LINEARFORWARD($\mathbf{a}$, $\boldsymbol{\alpha}$)
2:     $\mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
3:     return $\mathbf{b}$
4: procedure LINEARBACKWARD($\mathbf{a}$, $\omega$, $\mathbf{b}$, $\mathbf{g_b}$)
5:     $\mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
6:     $\mathbf{g_a} = \boldsymbol{\omega}^T \mathbf{g_b}$
7:     return $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$
```

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. It's output $b \in \mathbb{R}$ is a scalar. Below $\div$ is element-wise division. The output b is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

```
1: procedure CROSSENTROPYFORWARD($\mathbf{a}$, $\hat{\mathbf{a}}$)
2:     $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
3:     return $\mathbf{b}$
4: procedure CROSSENTROPYBACKWARD($\mathbf{a}$, $\hat{\mathbf{a}}$, $b$, $g_b$)
5:     $\mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
6:     return $\mathbf{g_a}$
```

It's also quite common to combine the Cross-Entropy and Softmax layers into one. The reason for this is the cancelation of numerous terms that result from the zeros in the cross-entropy backward calculation. You can read more about this in a detailed writeup from your TAs on Piazza.[6] (Said trick is *not* required to obtain a sufficiently fast implementation for Autolab.)

A.5.2 Module-based AD for Neural Network

Using these modules, we can re-define our functions NNFORWARD (Algorithm 3) and NNBACKWARD (Algorithm 4) as follows.

Algorithm 5 Forward Computation

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

[6] https://piazza.com/class/jar95m88puk3v6?cid=1092

Algorithm 6 Backpropagation

```
1: procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
2:     Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
3:     $g_J = \frac{dJ}{dJ} = 1$                                   ▷ Base case
4:     $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
5:     $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
6:     $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
7:     $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
8:     $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$        ▷ We discard $\mathbf{g}_{\mathbf{x}}$
9:     return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$
```

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.

A.6 Testing Backprop with Numerical Differentiation

Numerical differentiation provides a convenient method for testing gradients computed by backpropagation. The *centered* finite difference approximation is:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}_i))}{2\epsilon} \tag{A.5}$$

where $\boldsymbol{d}_i$ is a 1-hot vector consisting of all zeros except for the ith entry of $\boldsymbol{d}_i$, which has value 1. Unfortunately, in practice, it suffers from issues of floating point precision. Therefore, it is typically only appropriate to use this on small examples with an appropriately chosen ϵ.

In order to apply this technique to test the gradients of your backpropagation implementation, you will need to ensure that your code is appropriately factored. Any of the modules including NNFORWARD (Algorithm 3 or Algorithm 5) and NNBACKWARD (Algorithm 4 or Algorithm 6) could be tested in this way.

For example, you could use two functions: `forward(x,y,theta)` computes the cross-entropy for a training example. `backprop(x,y,theta)` computes the gradient of the cross-entropy for a training example via backpropagation. Finally, `finite_diff` as defined below approximates the gradient by the centered finited difference method. The following pseudocode provides an overview of the entire procedure.

```
def finite_diff(x, y, theta):
    epsilon = 1e-5
    grad = zero_vector(theta.length)
    for m in [1, ..., theta.length]:
        d = zero_vector(theta.length)
        d[m] = 1
        v = forward(x, y, theta + epsilon * d)
        v += forward(x, y, theta - epsilon * d)
        v /= 2*epsilon

# Compute the gradient by backpropagation
grad_bp = backprop(x, y, theta)
```

```
# Approximate the gradient by the centered finite difference method
grad_fd = finite_diff(x, y, theta)

# Check that the gradients are (nearly) the same
diff = grad_bp - grad_fd  # element-wise difference of two vectors
print l2_norm(diff) # this value should be small (e.g. < 1e-7)
```

A.6.1 Limitations

This does *not* catch all bugs—the only thing it tells you is whether your backpropagation implementation is correctly computing the gradient for the forward computation. Suppose your *forward* computation is incorrect, e.g. you are always computing the cross-entropy of the wrong label. If your *backpropagation* is also using the same wrong label, then the check above will not expose the bug. Thus, you always want to *separately* test that your forward implementation is correct.

A.6.2 Finite Difference Checking of Modules

Note that the above would test the gradient for the entire end-to-end computation carried output by the neural network. However, if you implement a module-based automatic differentiation method (as in Section A.5), then you can test each individual component for correctness. The only difference is that you need to run the finite-difference check for each of the output values (i.e. a double for-loop).