Programming (61 points)

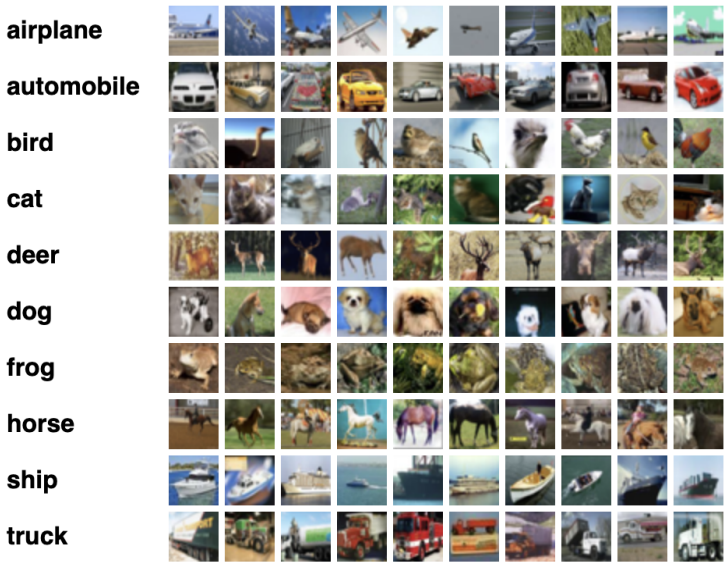

Figure 2: 10 random images from each of 10 image categories in CIFAR-10. You will be using a simplified version of this dataset.

5 The Task

Your goal in this assignment is to implement a neural network to classify images using a single hidden layer neural network. In addition, you will implement Adagrad, a variant of stochastic gradient descent.

6 The Datasets

Datasets We will be using a subset of a standard Computer Vision dataset, CIFAR-10. This data includes color images of various vehicles and animals; our subset will include black and white images of the 4 classes `automobile`, `bird`, `frog`, `ship`. The handout contains one dataset drawn from this data with 500 samples for training and 100 for validation.

File Format Each dataset (small, medium, and large) consists of two csv files—train and validation. Each row contains 1025 columns separated by commas. The first column contains the label and columns 2 to 1025 represent the pixel values of a 32×32 image in a row major format. Label 0 corresponds to `automobile`, 1 to `bird`, 2 to `frog`, and 3 to `ship`.

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 image recognition.

7 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.

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

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.

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

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

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{8}$$

In Equation 8, J is a function of the model parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ because $\hat{y}_k^{(i)}$ 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, $\hat{y}_k^{(i)}$ 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 **not** shuffle the training points when performing SGD. Note that SGD has a slight impact on the objective function, where we are "summing" over the current point, i:

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

Lastly, let's take a look at the Adagrad update that you will be performing. For parameters θ_t, you will first compute an intermediate value $\mathbf{s}_t$, and then use this to compute θ_{t+1}. $\mathbf{s}_t$ will contain the element-wise sums (denoted by $\odot$) of all the element-wise squared gradients. Therefore, $\mathbf{s}_t$ should have the same shape as $\frac{\partial J(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t}$. $\mathbf{s}_t$ should be initialized once, before the first epoch, to a zero vector. The update equations for $\mathbf{s}$ and θ are below.

$$\mathbf{s}_{t+1} = \mathbf{s}_t + \frac{\partial J(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t} \odot \frac{\partial J(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t}. \tag{10}$$

Then, we use $\mathbf{s}_t$ to scale the gradient for the update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbf{s}_{t+1} + \epsilon}} \odot \frac{\partial J(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t}. \tag{11}$$

Here, η is the learning rate, and $\epsilon = \texttt{1e-5}$.

7.1 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.

8 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 7.1, 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.

- In case there is a tie in the output layer $\hat{\mathbf{y}}$, predict the smallest index to be the label.
- You may assume that the input data will always have the same output label space (i.e. $\{0, 1, \ldots, 3\}$). 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.
- Do *not* use any machine learning libraries. You may use supported linear algebra packages. See Section 8.1 for more details.

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 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 Gradescope—since it will otherwise slow down your code.

8.1 Command Line Arguments

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

For Python:
```
$ python3 neuralnet.py [args...]
```

For Java:
```
$ javac -cp "./lib/ejml-v0.38-libs/*:./" neuralnet.java
$ java -cp "./lib/ejml-v0.38-libs/*:./" neuralnet [args...]
```

For C++:
```
$ g++ -g -std=c++11 -I./lib neuralnet.cpp; ./a.out [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 6)
2. `<validation_input>`: path to the validation input `.csv` file (see Section 6)
3. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 8.2)
4. `<validation_out>`: path to output `.labels` file to which the prediction on the *validation* data should be written (see Section 8.2)

5. <metrics_out>: path of the output .txt file to which metrics such as train and validation error should be written (see Section 8.3)
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 7.1 and Section 8)—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 base learning rate for SGD with Adagrad.

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 smallTrain.csv smallValidation.csv \
smallTrain_out.labels smallValidation_out.labels smallMetrics_out.txt \
2 4 2 0.1
```

8.2 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).

8.3 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 8). 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): 1.37990286268
epoch=1 crossentropy(validation): 1.40340064552
epoch=2 crossentropy(train): 1.37986222014
epoch=2 crossentropy(validation): 1.40252226818
error(train): 0.728
error(validation): 0.74
```

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`.

8.4 Tiny Data Set

To help you with this assignment, we have also included a tiny data set, `tinyTrain.csv` and `tinyValidation.csv`, and a reference output file `tinyOutput.txt` for you to use. The tiny dataset is in a format similar to the other datasets, but it only contains two samples with five features. The reference file contains outputs from each layer of one correctly implemented neural network, for both forward and back-propagation steps. We advise you to use this set to help you debug in case your implementation doesn't produce the same results as in the written part.

For your reference, `tinyOutput.txt` is generated from the following command line specifications:

```
$ python3 neuralnet.py tinyTrain.csv tinyValidation.csv \
tinyTrain_out.labels tinyValidation_out.labels tinyMetrics_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.

9 Gradescope Submission

You should submit your `neuralnet.{py|java|cpp}` to Gradescope. 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.

Some additional tips: 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.

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 8 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 Gradescope, we make a few simplifications: (1) you should *not* shuffle your data and (2) you will use a fixed learning rate. In the real world, you would *not* make these simplifications.

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

Algorithm 1 Stochastic Gradient Descent (SGD) without Shuffle

procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
  Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ ▷ Use either RANDOM or ZERO from Section 7.1
  for $e \in \{1, 2, \ldots, E\}$ **do** ▷ For each epoch
    for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do** ▷ For each training example (No shuffling)
      Compute neural network layers:
      $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
      Compute gradients via backprop:
      $$\left.\begin{aligned} \mathbf{g}_{\boldsymbol{\alpha}} &= \frac{\partial J}{\partial \boldsymbol{\alpha}} \\ \mathbf{g}_{\boldsymbol{\beta}} &= \frac{\partial J}{\partial \boldsymbol{\beta}} \end{aligned}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$$
      Update parameters with Adagrad 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 $\boldsymbol{\alpha}, \boldsymbol{\beta}$

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

Algorithm 2 Prediction at Test Time

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

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.

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{10} & \alpha_{11} & \dots & \alpha_{1M} \\ \alpha_{20} & \alpha_{21} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{D0} & \alpha_{D1} & \dots & \alpha_{DM} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} = \begin{bmatrix} \frac{dJ}{d\alpha_{10}} & \frac{dJ}{d\alpha_{11}} & \dots & \frac{dJ}{d\alpha_{1M}} \\ \frac{dJ}{d\alpha_{20}} & \frac{dJ}{d\alpha_{21}} & \dots & \frac{dJ}{d\alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dJ}{d\alpha_{D0}} & \frac{dJ}{d\alpha_{D1}} & \dots & \frac{dJ}{d\alpha_{DM}} \end{bmatrix} \tag{12}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{10} & \beta_{11} & \dots & \beta_{1D} \\ \beta_{20} & \beta_{21} & \dots & \beta_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{K0} & \beta_{K1} & \dots & \beta_{KD} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\beta}} = \frac{\partial J}{\partial \boldsymbol{\beta}} = \begin{bmatrix} \frac{dJ}{d\beta_{10}} & \frac{dJ}{d\beta_{11}} & \dots & \frac{dJ}{d\beta_{1D}} \\ \frac{dJ}{d\beta_{20}} & \frac{dJ}{d\beta_{21}} & \dots & \frac{dJ}{d\beta_{2D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dJ}{d\beta_{K0}} & \frac{dJ}{d\beta_{K1}} & \dots & \frac{dJ}{d\beta_{KD}} \end{bmatrix} \tag{13}$$

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 $a_i = \sum_{m=0}^{M} \alpha_{im} x_m$. That is a_i is the *first and only* intermediate quantity that uses α_{ij}. Applying the chain rule, we obtain
$$\frac{dJ}{d\alpha_{ij}} = \frac{dJ}{da_i}\frac{da_i}{d\alpha_{ij}} = \frac{dJ}{da_i}x_j$$

2. So far so good, now we just need to compute $\frac{dJ}{da_i}$. Not a problem! We can just apply the chain rule again. a_i just has exactly one child as well, namely $z_i = \sigma(a_i)$. The chain rule gives us that $\frac{dJ}{da_i} = \frac{dJ}{dz_i}\frac{dz_i}{da_i} = \frac{dJ}{dz_i}z_i(1-z_i)$. Substituting back into the equation above we find that
$$\frac{dJ}{d\alpha_{ij}} = \frac{dJ}{dz_i}(z_i(1-z_i))x_j$$

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

$$\frac{dJ}{d\alpha_{ij}} = \sum_{k=1}^{K} \frac{dJ}{db_k} \beta_{ki}(z_i(1 - z_i))x_j$$

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

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

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:

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

   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.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 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 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+

[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".

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 programmers would find that their code is simply too slow if they tried to do the same. As such, particularly for Python 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.4.1).

The procedural method: one function computes the outputs of the neural network and all intermediate quantities $\mathbf{o} = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J)$, where the object is just some struct. Then another function computes the gradients of our parameters $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$, where $\mathbf{o}$ is a data structure that stores all the forward computation.

One must be careful to ensure that functions are vectorized. For example, your Sigmoid function should be able to take a vector input and return a vector output with the Sigmoid function applied to all of its elements. All of these operations should avoid for-loops when working in a high-level language like Python. We can compute the softmax function in a similar vectorized manner.

A.4.1 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 $\mathbf{b} = [b_1, \ldots, b_B]$ 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 $\frac{dJ}{da_i} = \sum_{j=1}^{M} \frac{dJ}{db_j} \frac{\partial b_j}{\partial a_i}$ for all $i \in \{1, \ldots, A\}$.

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 $\mathbf{b} = *\text{FORWARD}(\mathbf{a})$, and a backward function $\mathbf{g_a} = *\text{BACKWARD}(\mathbf{a}, \mathbf{b}, \mathbf{g_b})$ method. These methods accept parameters if appropriate. You'll want to pay close attention to the dimensions that you pass into and return from your modules.

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(a, α)
2:     Compute b
3:     return b
4: procedure LINEARBACKWARD(a, α, b, g_b)
5:     Compute g_α
6:     Compute g_a
7:     return g_α, 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. (Said trick is *not* required to obtain a sufficiently fast implementation for Gradescope.)

A.5.2 Module-based AD for Neural Network

Using these modules, we can re-define our functions NNFORWARD and NNBACKWARD as follows.

[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.

Algorithm 3 Forward Computation

procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
  $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \boldsymbol{\alpha})$
  $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$
  $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \boldsymbol{\beta})$
  $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$
  $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$
  $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
  return intermediate quantities $\mathbf{o}$

Algorithm 4 Backpropagation

procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
  Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
  $g_J = \frac{\partial J}{\partial J} = 1$ ▷ Base case
  $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, g_J)$
  $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
  $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \boldsymbol{\beta}, \mathbf{g}_{\mathbf{b}})$
  $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
  $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \boldsymbol{\alpha}, \mathbf{g}_{\mathbf{a}})$ ▷ We discard $\mathbf{g}_{\mathbf{x}}$
  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:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}_i))}{2\epsilon} \tag{14}$$

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 and NNBACKWARD 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
        grad[m] = v

# 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).

A.7 Why AdaGrad?

So far, the loss functions we have discussed make it quite easy to find a global optimum. In these cases, a larger step size makes it quick and easy to reach convergence. This isn't always the case with neural networks. Nonconvex loss functions are much harder to optimize over. Here we want step sizes that will adapt to the domain in which they optimize. We want to take larger steps where possible, but smaller steps where we are in danger of overshooting the optima. Adagrad implicitly changes the step size based on the shape of the function inferred from the gradients. Interested? Read all about it here.