HOMEWORK 2 (PROGRAMMING): NEURAL NETWORKS

10-315 Introduction to Machine Learning (Fall 2020)
Carnegie Mellon University

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

Begin by downloading and unzipping the HW2 release from course webpage. This contains the skeleton code, data, and autograder for this assignment.

This assignment includes an autograder for you to grade your code on your machine. Remember to finish Q2 of written after completing this programming. This can be run with the command:

```
python3.6 autograder.py
```

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

Files you will edit

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

Files you might want to look at

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

Files you can safely ignore

- `autograder.py` Autograder infrastructure code.

Files to Edit and Submit:

You will fill in portions of `neural_network.py` and `additional_code.py` during the assignment. You should submit this file containing your code and comments to the Programming component on Gradescope. Please do not change the other files in this distribution or submit any of our original files other than

these files. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Report:

The Written component of this assignment contains questions that require additional programming but are not autograded. You will place the requested results in the appropriate locations within the PDF of the Written component of this assignment.

Evaluation:

Your assignment will be assessed based on your code, the output of the autograder, and the required contents of in the Written component.

Academic Dishonesty:

We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help:

You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, recitation, and Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these assignments to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

For staff use only

Q1	Q2	Q3	Q4	Q5	Total
/8	/10	/6	/4	/2	/30

1 Task: Neural Network Implementation

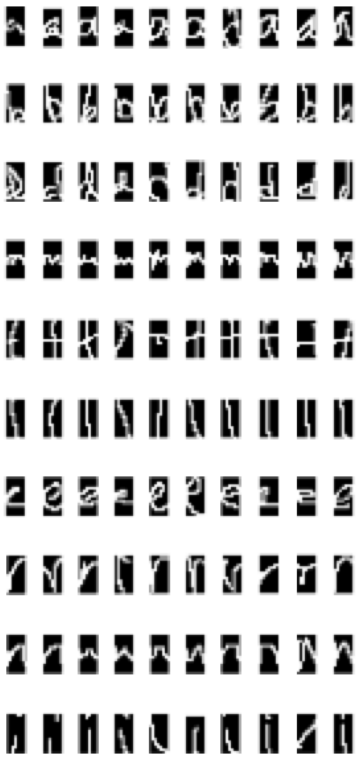

Figure 1.1: 10 Random Images of Each of 10 Letters in OCR

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.

1.1 The Task and Datasets

Datasets We will be using a subset of an Optical Character Recognition (OCR) dataset. This data includes images of all 26 handwritten letters; our subset will include only the letters "a," "e," "g," "i," "l," "n," "o," "r," "t," and "u." The handout contains three datasets drawn from this data: a small dataset with 60 samples per class (50 for training and 10 for test), a medium dataset with 600 samples per class (500 for training and 100 for test), and a large dataset with 1000 samples per class (900 for training and 100 for test). Figure 1.1 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 test. 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 1.1 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.

1.2 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.

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

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

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

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

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

Some points to mention:

- Do *not* use any machine learning libraries. You may and please do use NumPy.
- Try to "vectorize" your code as much as possible. In Python, you want to avoid for-loops and instead rely on `numpy` calls to perform operations such as matrix multiplication, transpose, subtraction, etc. over an entire `numpy` array at once. This is much faster; using NumPy over list can speed up your computation by 200x!

- You'll want to pay close attention to the dimensions that you pass into and return from your functions.

1.3 [8 pts] Q1 implementation: Feed Forward

Implement the forward functions for each of the layers:
- `linearForward`, `sigmoidForward`, `softmaxForward`, `crossEntropy`.

Next, implement the `NNForward` function that calls a complete forward pass on the neural network.

Algorithm 1 Forward Computation

1: **procedure** NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
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: **return** intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, \mathbf{J}$

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

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

1.4 [10 pts] Q2 implementation: Backward Propagation

Implement the backward functions for each of the layers: (note: softmax and cross-entropy backpropagation are combined to one due to easier calculation)
- `softmaxBackward`, `sigmoidBackward`, `linearBackward`.

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

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{10} & \alpha_{11} & \dots & \alpha_{1M} \\ \alpha_{20} & \alpha_{21} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{D0} & \alpha_{D1} & \dots & \alpha_{DM} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} = \begin{bmatrix} \frac{d\ell}{d\alpha_{10}} & \frac{d\ell}{d\alpha_{11}} & \dots & \frac{d\ell}{d\alpha_{1M}} \\ \frac{d\ell}{d\alpha_{20}} & \frac{d\ell}{d\alpha_{21}} & \dots & \frac{d\ell}{d\alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{d\ell}{d\alpha_{D0}} & \frac{d\ell}{d\alpha_{D1}} & \dots & \frac{d\ell}{d\alpha_{DM}} \end{bmatrix} \tag{1.1}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{10} & \beta_{11} & \dots & \beta_{1D} \\ \beta_{20} & \beta_{21} & \dots & \beta_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{K0} & \beta_{K1} & \dots & \beta_{KD} \end{bmatrix} \qquad \mathbf{g}_{\boldsymbol{\beta}} = \frac{\partial J}{\partial \boldsymbol{\beta}} = \begin{bmatrix} \frac{d\ell}{d\beta_{10}} & \frac{d\ell}{d\beta_{11}} & \dots & \frac{d\ell}{d\beta_{1D}} \\ \frac{d\ell}{d\beta_{20}} & \frac{d\ell}{d\beta_{21}} & \dots & \frac{d\ell}{d\beta_{2D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{d\ell}{d\beta_{K0}} & \frac{d\ell}{d\beta_{K1}} & \dots & \frac{d\ell}{d\beta_{KD}} \end{bmatrix} \tag{1.2}$$

Reminder once again that $\boldsymbol{\alpha}$ and $\mathbf{g}_{\boldsymbol{\alpha}}$ are $D \times (M+1)$ matrices, while $\boldsymbol{\beta}$ 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 $z_0 = 1$.

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

Algorithm 2 Backpropagation

procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $z, \hat{y}$)
  Place intermediate quantities $\mathbf{z}, \hat{\mathbf{y}}$ in scope
  $\mathbf{g_b} = \text{SOFTMAXBACKWARD}^*(\mathbf{y}, \hat{\mathbf{y}})$
  $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g_z} = \text{LINEARBACKWARD}(\mathbf{z}, \boldsymbol{\beta}, \mathbf{g_b})$
  $\mathbf{g_a} = \text{SIGMOIDBACKWARD}(\mathbf{z}, \mathbf{g_z})$
  $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g_x} = \text{LINEARBACKWARD}(\mathbf{x}, \boldsymbol{\alpha}, \mathbf{g_a})$ ▷ We discard $\mathbf{g_x}$
  return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g_b}, \mathbf{g_z}, \mathbf{g_a}$

*It is common to combine the Cross-Entropy and Softmax backpropagation into one, due to the simpler calculation (from cancellation of numerous terms).

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

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

1.5 [6 pts] Q3: Training with SGD

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

Because we want the behavior of your program to be deterministic for testing on Gradescope, we make a few simplifications: (1) you should *not* shuffle your data and (2) you will use a fixed learning rate. In the real world, you would *not* make these simplifications.

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

Algorithm 3 Stochastic Gradient Descent (SGD) without Shuffle

procedure SGD(Training data $\mathcal{D}$, Validation data $\mathcal{D}'$, other relevant parameters)
  Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ ▷ Use either RANDOM or ZERO from Section 1.5.1
  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:
      $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
      Compute gradients via backprop:

$$\left.\begin{aligned} \mathbf{g}_{\boldsymbol{\alpha}} &= \frac{\partial J}{\partial \boldsymbol{\alpha}} \\ \mathbf{g}_{\boldsymbol{\beta}} &= \frac{\partial J}{\partial \boldsymbol{\beta}} \end{aligned}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{z}, \hat{y})$$

      Update parameters:
      $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
      $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
    Store training mean cross-entropy $J(\boldsymbol{\alpha}, \boldsymbol{\beta})$ ▷ from Eq. 1.4
    Store validation mean cross-entropy $J(\boldsymbol{\alpha}, \boldsymbol{\beta})$ ▷ from Eq. 1.4
  return $\boldsymbol{\alpha}, \boldsymbol{\beta}$, cross_entropy_train_list, cross_entropy_valid_list

1.5.1 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.

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

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

$$J_{SGD}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{1.3}$$

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.

The objective function you then use to calculate the average cross entropy over, say the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, is:

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{1.4}$$

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

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

1.6 [4 pts] Q4: Label Prediction

Recall that for a single input x, your network outputs a probability distribution over K classes, $\hat{y}$. After you've trained your network and obtained the weight parameters α and β, you now want to predict the labels given the data.

Implement the `prediction` function as follows.

Algorithm 4 Prediction

procedure PREDICTION(Training data $\mathcal{D}$, Validation data $\mathcal{D}'$, Parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$)
  for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
    Compute neural network prediction $\hat{\mathbf{y}}$ from NNFORWARD$(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
    Predict the label with highest probability $l = \text{argmax}_k \, \hat{y}_k$
    Check for error $l \neq y$
  for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}'$ **do**
    Compute neural network prediction $\hat{\mathbf{y}}$ from NNFORWARD$(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
    Predict the label with highest probability $l = \text{argmax}_k \, \hat{y}_k$
    Check for error $l \neq y$
  return train_error, valid_error, train_predictions, valid_predictions

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

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

1.7 [2 pts] Q5: Main `train_and_valid` function

Finally, implement the `train_and_valid()` function to train and validate your neural network implementation.

Your program should learn the parameters of the model on the training data, and report the 1) cross-entropy on both train and validation data for each epoch. After training, it should write out its 2) predictions and 3) error rates on both train and validation data. See the docstring in the code for more details. You may implement any helper code or functions you'd like within `neural_network.py`.

Your implementation must satisfy the following requirements:

- Number of **hidden units** for the hidden layer will be determined by the `num_hidden` argument to the `train_and_valid` function.
- SGD must support two different **initialization strategies**, as described in Section 1.5.1, selecting between them based on the `init_rand` argument to the `train_and_valid` function.
- The number of **epochs** for SGD will be determined by the `num_epoch` argument to the `train_and_valid` function.
- The **learning rate** for SGD is specified by the `learning_rate` argument to the `train_and_valid` function.
- Perform SGD updates on the training data in the order that the data is given in the input file. Although you would typically shuffle training examples when using stochastic gradient descent, in order to autograde the assignment, we ask that you **DO NOT** shuffle trials in this assignment.

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

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

1.8 Submission

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

Don't forget to include any request results in the PDF of the Written component, which is to be submitted on Gradescope as well.

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

```
python3.6 autograder.py
```

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

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

A Debugging Backpropagation

This section has some approaches to verifying your backpropagation

A.1 Testing Backpop with Numerical Differentiation

What happens if your backprogation is not working?

- Use what you've done in the written portion. Check your backpropagation code with your derivations in Q1, and check your function outputs with the answers found in Q2.
- 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.
- An alternative is to 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.

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

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}_i))}{2\epsilon} \tag{A.1}$$

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.1.1 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.2 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 $\frac{d\ell}{d\alpha_{ij}}$ 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_{ij}. Applying the chain rule, we obtain
$$\frac{d\ell}{d\alpha_{ij}} = \frac{d\ell}{da_i}\frac{da_i}{d\alpha_{ij}} = \frac{d\ell}{da_i}x_j$$

2. So far so good, now we just need to compute $\frac{d\ell}{da_j}$. 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 $\frac{d\ell}{da_j} = \frac{d\ell}{dz_j}\frac{dz_j}{da_j} = \frac{d\ell}{dz_j} z_j(1-z_j)$. Substituting back into the equation above we find that
$$\frac{d\ell}{d\alpha_{ij}} = \frac{d\ell}{dz_j}(z_j(1-z_j))x_i$$

3. How do we get $\frac{d\ell}{dz_j}$? 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 $b_1, b_2, \ldots b_K$. Applying the chain rule gives us that $\frac{d\ell}{dz_j} = \sum_{k=1}^{K} \frac{d\ell}{db_k}\frac{db_k}{dz_j} = \sum_{k=1}^{K} \frac{d\ell}{db_k}\beta_{kj}$. Substituting back into the equation above gives:
$$\frac{d\ell}{d\alpha_{ij}} = \sum_{k=1}^{K} \frac{d\ell}{db_k}\beta_{kj}(z_j(1-z_j))x_i$$

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

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

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

   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.