HOMEWORK 6 (PROGRAMMING): NEURAL NETWORKS

10-315 Introduction to Machine Learning (Spring 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 https://www.cs.cmu.edu/~10315/assignments/hw6/programming/hw6_programming.zip. This contains the skeleton code, data, and autograder for this assignment.

This assignment includes an autograder for you to grade your some aspects of your code on your machine. 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` Data files 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 Grade-

scope. 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.

1 Task: Neural Network Implementation

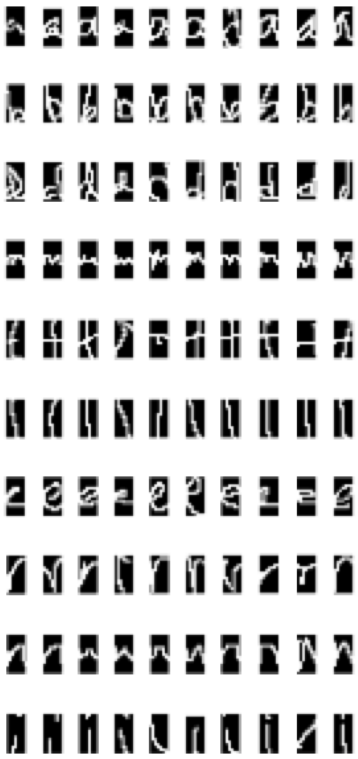

Figure 1.1: 10 Random Images of Each of 10 Letters in OCR

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.

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

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 $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$:

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{1.1}$$

In Equation 1.2, 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. Note that SGD has

a slight impact on the objective function, where we are "summing" over just the current point, i:

$$J_{SGD}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{1.2}$$

1.2.1 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.

1.3 Implementation

Implement the `train_and_test()` function in `neural_network.py` to train and validate your neural network implementation. See the docstring in the code for more details. You may implement any helper code you like outside of `train_and_test()` within `neural_network.py`.

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 will be determined by the `num_hidden` argument to the `train_and_test` function.
- Support two different **initialization strategies**, as described in Section 1.2.1, selecting between them based on the `init_rand` argument to the `train_and_test` function.
- Use stochastic gradient descent (SGD) to optimize the parameters for one hidden layer neural network. The number of **epochs** will be determined by the `num_epoch` argument to the `train_and_test` function.
- Use the **learning rate** specified by the `learning_rate` argument to the `train_and_test` function.
- 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 NumPy.

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

- Try to "vectorize" your code as much as possible. 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.
- 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.

1.4 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 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 1.3 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) 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 1.2.1
  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:
      $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
      $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
    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.

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

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

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 $\frac{d\ell}{d\alpha_{ij}}$ and $\frac{d\ell}{d\beta_{kj}}$.

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 $\frac{d\ell}{d\alpha_{ij}}$ 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.

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+

[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, you should convert all the deeply nested for-loops into efficient "vectorized" math via `numpy`. Doing so will ensure efficient code.

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 / Octave. 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]

[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

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 $\mathbf{g_a} = \frac{\partial J}{\partial \mathbf{a}} = [\frac{d\ell}{da_1}, \ldots, \frac{d\ell}{da_A}]$ given the gradient of output $\mathbf{g_b} = \frac{\partial J}{\partial \mathbf{b}} = [\frac{d\ell}{db_1}, \ldots, \frac{d\ell}{db_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{d\ell}{da_i} = \sum_{j=1}^{J} \frac{d\ell}{db_j} \frac{db_j}{da_i}$ 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.

procedure LINEARFORWARD($\mathbf{a}$, $\boldsymbol{\omega}$)
    Compute $\mathbf{b}$
    return $\mathbf{b}$
procedure LINEARBACKWARD($\mathbf{a}$, α, $\mathbf{b}$, $\mathbf{g_b}$)
    Compute $\mathbf{g}_{\boldsymbol{\alpha}}$
    Compute $\mathbf{g_a}$
    return $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g_a}$

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. (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 and NNBACKWARD as follows.

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 $\mathbf{o}$

form of implementation is different from the dynamic method we recommend here.

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
  $g_J = \frac{dJ}{dJ} = 1$ ▷ Base case
  $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
  $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
  $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
  $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
  $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$ ▷ 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:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}_i))}{2\epsilon} \tag{A.3}$$

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).