HOMEWORK 5: NEURAL NETWORKS

10-301/10-601 Introduction to Machine Learning (Fall 2020)
https://www.cs.cmu.edu/~10601/
DUE: Monday, Oct 26, 2020 11:59 PM

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

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: https://www.cs.cmu.edu/~10601
- **Late Submission Policy:** See the late submission policy here: https://www.cs.cmu.edu/~10601
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using Gradescope (https://gradescope.com/). Please use the provided template. Submissions must be written in LaTeX. Regrade requests can be made, however this gives the staff the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed in the boxes provided. For short answer questions you **should not** include your work in your solution. If you include your work in your solutions, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.6.9, OpenJDK 11.0.5, g++ 7.4.0) and versions of permitted libraries (e.g. `numpy` 1.17.0 and `scipy` 1.4.1) match those used on Gradescope. You have unlimited Gradescope programming submissions. However, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before submitting you code to Gradescope.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on Piazza.

Programming (56 points)

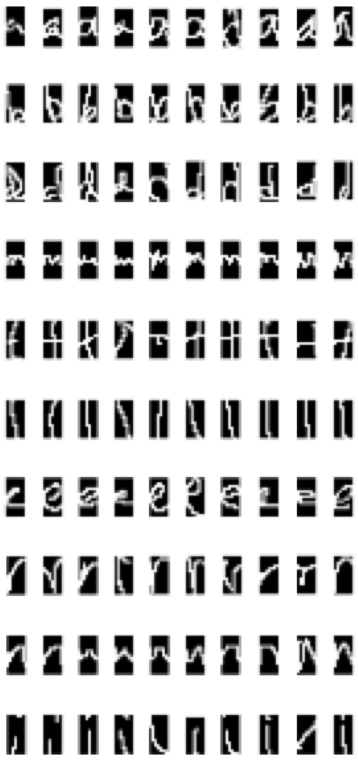

Figure 2: 10 Random Images of Each of 10 Letters in OCR

1 The Task

Your goal in this assignment is to label images of handwritten letters by implementing a Neural Network from scratch. You will implement all of the functions needed to initialize, train, evaluate, and make predictions with the network.

The programs you write will be automatically graded using the Gradescope system. You may write your programs in **Python, Java, or C++**. However, you should use the same language for all parts below.

2 The Datasets

Datasets We will be using a subset of an Optical Character Recognition (OCR) dataset. This data includes images of all 26 handwritten letters; our subset will include only the letters "a," "e," "g," "i," "l," "n," "o," "r," "t," and "u." The handout contains three datasets drawn from this data: a small dataset with 60 samples *per class* (50 for training and 10 for validation), a medium dataset with 600 samples per class (500 for training and 100 for validation), and a large dataset with 1000 samples per class (900 for training and 100 for validation). Figure 2 shows a random sample of 10 images of few letters from the dataset.

File Format Each dataset (small, medium, and large) consists of two csv files—train and validation. Each row contains 129 columns separated by commas. The first column contains the label and columns 2 to 129 represent the pixel values of a 16×8 image in a row major format. Label 0 corresponds to "a," 1 to "e," 2 to "g," 3 to "i," 4 to "l," 5 to "n," 6 to "o," 7 to "r," 8 to "t," and 9 to "u." Because the original images are black-and-white (not grayscale), the pixel values are either 0 or 1. However, you should write your code to

accept arbitrary pixel values in the range [0, 1]. The images in Figure 2 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.

3 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 $\hat{y}_k$ 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. Note that SGD has a slight impact on the objective function, where we are "summing" over the current point, i:

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

3.1 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.

4 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 3.1, 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. $\{0, 1, \ldots, 9\}$). Other than this, do not hard-code any aspect of the datasets into your code. We will autograde your programs on multiple (hidden) data sets that include different examples.
- Do *not* use any machine learning libraries. You may use supported linear algebra packages. See Section 4.1 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.

4.1 Command Line Arguments

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

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

For Java:
```
$ javac -cp "./lib/ejml-v0.38-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 2)
2. `<validation_input>`: path to the validation input `.csv` file (see Section 2)
3. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 4.2)
4. `<validation_out>`: path to output `.labels` file to which the prediction on the *validation* data should be written (see Section 4.2)
5. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and validation error should be written (see Section 4.3)
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 3.1 and Section 4)—that is, if `init_flag`==1 initialize your weights randomly from a uniform distribution over the range [-0.1, 0.1] (i.e. RANDOM), if `init_flag`==2 initialize all weights to zero (i.e. ZERO). For both settings, **always initialize bias terms to zero**.

9. `<learning_rate>`: float value specifying the learning rate for SGD.

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

```
$ python neuralnet.py smallTrain.csv smallValidation.csv \
smallTrain_out.labels smallValidation_out.labels smallMetrics_out.txt \
2 4 2 0.1
```

4.2 Output: Labels Files

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

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

Note: You should output your predicted labels using the same *integer* identifiers as the original training data. You should also insert an empty line (again using '\n') at the end of each sequence (as is done in the input data files). The first few lines of the predicted labels for the validation dataset is given below

```
6

4

8

8
```

4.3 Output Metrics

Generate a file where you report the following metrics:

cross entropy After each Stochastic Gradient Descent (SGD) epoch, report mean cross entropy on the training data `crossentropy(train)` and validation data `crossentropy(validation)` (See Equation 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): 2.18506276114
epoch=1 crossentropy(validation): 2.18827302588
```

```
epoch=2 crossentropy(train): 1.90103257727
epoch=2 crossentropy(validation): 1.91363803461
error(train): 0.77
error(validation): 0.78
```

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

4.4 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 described in Section 2.3.3.

There is more information in the `README.md` file. Do read through the README file if you plan to use it for debugging. For your reference, `tinyOutput.txt` is generated from the following command line specifications:

```
$ python 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.

5 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.