Homework 4

4190.408 001, Artificial Intelligence

November 22, 2023

Submission

- In this homework we will learn and practice the basics of deep learning by building a deep learning model and training the model by ourselves.
- Assignment due: **Dec 5, 11:59 pm**
- Individual Assignment
- Submission through ETL. Please read the instructions in the PDF, and **fill in the TODOs in the neural_network.py file and submit**. The file should be compressed to a zip file, and file name should be in the format of "{student_ID}_{first name}_{last name}.zip", e.g. "2023-12345_Jeonghyeon_Na.zip". Please write your name in English.
- Collaborations on solving the homework is allowed. Discussions are encouraged but you should think about the problems on your own.
- If you do collaborate with someone or use a book or website, you are expected to write up your solution independently. That is, close the book and all of your notes before starting to write up your solution.
- Make sure you cite your work/collaborators at the end of the homework.
- **Using deep-learning libraries such as PyTorch, TensorFlow or Scikit-Learn is prohibited. You may and are recommended to use Numpy package.**
- **Honor Code**: This document is exclusively for Fall 2023, 4190.408 students with Professor Hanbyul Joo at Seoul National University. Any student who references this document outside of that course during that semester (including any student who retakes the course in a different semester), or who shares this document with another student who is not in that course during that semester, or who in any way makes copies of this document (digital or physical) without consent of Professor Hanbyul Joo is guilty of cheating, and therefore subject to penalty according to the Seoul National University Honor Code.

1 Neural Network Implementation

In this project, your objective is to classify images of handwritten alphabets by creating a Neural Network from the ground up. You'll need to develop all the necessary components for initializing, training, assessing, and making predictions using this network.

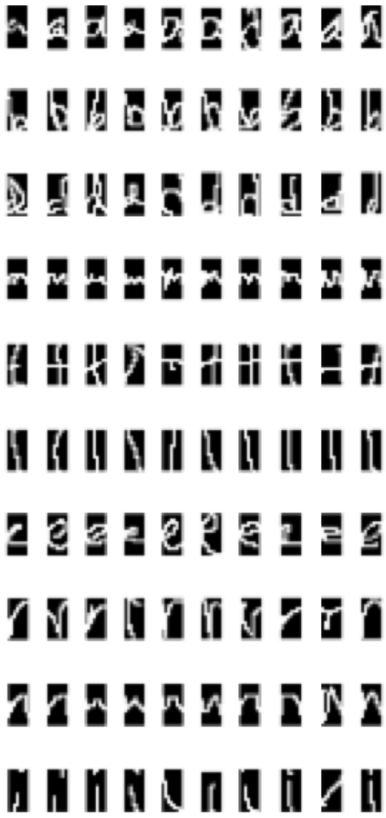

Figure 1: 10 Random Images of 10 Letters in OCR

1.1 The Task and Datasets

Datasets We will employ a section of an Optical Character Recognition (OCR) dataset, which includes images of the entire alphabet written by hand. For our purpose, we'll focus on a subset comprising the letters "a," "e," "g," "i," "l," "n," "o," "r," "t," and "u." Included in your materials are three separate datasets extracted from this collection: a small set with 60 instances per character (50 for training, 10 for testing), a medium set with 600 instances per character (500 for training, 100 for testing), and a large set with 1000 instances per character (900 for training, 100 for testing). Refer to Figure 1.1 for a selection of 10 representative images from various letters in the dataset.

File Format Each dataset size category (small, medium, large) is accompanied by two CSV files, one for training and the other for testing. Every row in these files is composed of 129 comma-separated columns. The first column is the label, and columns 2 through 129 represent the pixel values of a 16×8 image, arranged in row-major order. Labels are numerically assigned from 0 to 9 corresponding to the letters "a" through "u" respectively. The pixel values are binary (0 or 1), as the original images are in black and white, not grayscale. It's advisable to design your code to handle any pixel values within the [0,1] range. The images shown in Figure 1 are generated by converting these pixel values to .png format for easier visualization. Note that the datasets are raw, without any prior feature engineering. Your task is to build a neural network capable of learning the necessary features for effectively recognizing these characters.

Data Loading Function `load_data_small, load_data_medium, load_data_large` is already implemented. You can use these functions to load data and train your own implementations. Any extra helper function to test your implementation using the provided data is fine, as long as the required functions are implemented correctly.

1.2 Model Definition

In this project, you are tasked with developing a neural network that has a single hidden layer. This network should use a sigmoid activation function in the hidden layer and a softmax function at the output layer. Consider input vectors $\mathbf{x}$ with a dimensionality of M, the hidden layer $\mathbf{z}$ containing D units, and the output layer $\hat{\mathbf{y}}$ representing a probability distribution over K categories. Thus, each component y_k of the output vector indicates the probability of $\mathbf{x}$ being classified into class k.

Input (length)	Layer/Activation	Output (length)
x of length M	Linear (hidden layer)	a of length D
a of length D	Sigmoid Activation	z of length D
z of length D	Linear (output layer)	b of length K
b of length K	Softmax	y of length K

Table 1: Model Architecture

Enhance the model by introducing bias elements into the layers' inputs. Define $x_0 = 1$ as the input bias and $z_0 = 1$ as the bias in the hidden layer. The model includes two parameter matrices: $\boldsymbol{\alpha} \in \mathbb{R}^{D\times(M+1)}$ and $\boldsymbol{\beta} \in \mathbb{R}^{K\times(D+1)}$. The first column in each matrix ($\boldsymbol{\alpha}_{\cdot,0}$ and $\boldsymbol{\beta}_{\cdot,0}$) contains the bias parameters. It's essential to adjust the dimensions of your inputs and matrices to include these bias columns, resulting in dimensions $D+1$ instead of D.

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

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

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

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

The objective function for this model is the average cross-entropy, calculated 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:

- Focus on "vectorizing" your code. In Python, leveraging NumPy to perform matrix operations like multiplication, transpose, and subtraction over entire arrays at once is preferable to using for-loops. This method can enhance computation speed significantly, potentially by up to 200 times compared to standard Python lists.
- You'll want to pay close attention to the dimensions that you pass into and return from your functions.

1.3 [25pts] Q1 implementation: Feed Forward

Implement the forward functions for each of the layers:
`linearForward, sigmoidForward, softmaxForward, corssEntropy`

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

Algorithm 1 Forward Computation

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

1.4 [35pts] 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.

$$\alpha = \begin{bmatrix} \alpha_{10} & \alpha_{11} & \dots & \alpha_{1M} \\ \alpha_{20} & \alpha_{21} & \dots & \alpha_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{D0} & \alpha_{D1} & \dots & \alpha_{DM} \end{bmatrix} \qquad g_\alpha = \frac{\partial J}{\partial \alpha} = \begin{bmatrix} \frac{\partial e}{\partial \alpha_{10}} & \frac{\partial e}{\partial \alpha_{11}} & \dots & \frac{\partial e}{\partial \alpha_{1M}} \\ \frac{\partial e}{\partial \alpha_{20}} & \frac{\partial e}{\partial \alpha_{21}} & \dots & \frac{\partial e}{\partial \alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e}{\partial \alpha_{D0}} & \frac{\partial e}{\partial \alpha_{D1}} & \dots & \frac{\partial e}{\partial \alpha_{DM}} \end{bmatrix} \tag{1.1}$$

$$\beta = \begin{bmatrix} \beta_{10} & \beta_{11} & \dots & \beta_{1D} \\ \beta_{20} & \beta_{21} & \dots & \beta_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{K0} & \beta_{K1} & \dots & \beta_{KD} \end{bmatrix} \qquad g_\beta = \frac{\partial J}{\partial \beta} = \begin{bmatrix} \frac{\partial e}{\partial \beta_{10}} & \frac{\partial e}{\partial \beta_{11}} & \dots & \frac{\partial e}{\partial \beta_{1D}} \\ \frac{\partial e}{\partial \beta_{20}} & \frac{\partial e}{\partial \beta_{21}} & \dots & \frac{\partial e}{\partial \beta_{2D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e}{\partial \beta_{K0}} & \frac{\partial e}{\partial \beta_{K1}} & \dots & \frac{\partial e}{\partial \beta_{KD}} \end{bmatrix} \tag{1.2}$$

Reminder once again that α and g_α are $D \times (M+1)$ matrices, while β and g_β are $K \times (D+1)$ matrices. The $+1$ comes from the extra columns $\alpha_{.,0}$ and $\beta_{.,0}$ 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 (x, y), Parameters α, β, Intermediates $z, \hat{y}$)
  Place intermediate quantities $z, \hat{y}$ in scope
  $g_b = \texttt{SOFTMAXBACKWARD}^*(y, \hat{y})$
  $g_\beta, g_z = \texttt{LINEARBACKWARD}(z, \beta, g_b)$
  $g_a = \texttt{SIGMOIDBACKWARD}(z, g_z)$
  $g_\alpha, g_x = \texttt{LINEARBACKWARD}(x, \alpha, g_a)$ ▷ We discard g_x
  return parameter gradients $g_\alpha, g_\beta, g_b, g_z, g_a$
end procedure=0

1.5 [20pts] 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, and easier for grading 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 D, Validation data D', other relevant parameters)
  Initialize parameters α, β ▷ Use either RANDOM or ZERO from Section 1.5.1
  for $e \in \{1, 2, \ldots, E\}$ **do** ▷ For each epoch
    for $(x, y) \in D$ **do** ▷ For each training example (No shuffling)
      Compute neural network layers:
      $x, a, b, z, \hat{y}, J = \text{NNFORWARD}(x, y, \alpha, \beta)$
      Compute gradients via backprop:
      $g_\alpha = \frac{\partial J}{\partial \alpha}, g_\beta = \frac{\partial J}{\partial \beta} = \text{NNBACKWARD}(x, y, \alpha, \beta, z, \hat{y})$
      Update parameters:
      $\alpha \leftarrow \alpha - \gamma g_\alpha$
      $\beta \leftarrow \beta - \gamma g_\beta$
      Store training mean cross-entropy $J(\alpha, \beta)$ ▷ from Eq. 1.4
      Store validation mean cross-entropy $J(\alpha, \beta)$ ▷ from Eq. 1.4
    end for
  end for
  return α, β, 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:

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

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

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

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

This question depends on the correctness to your previous parts.

1.6 [10pts] 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 D, Validation data D', Parameters α, β)
  for $(x, y) \in D$ **do**
    Compute neural network prediction $\hat{y}$ from NNFORWARD(x, y, α, β)
    Predict the label with highest probability $l = \arg\max_k \hat{y}_k$
    Check for error $l \neq y$
  end for
  for $(x, y) \in D'$ **do**
    Compute neural network prediction $\hat{y}$ from NNFORWARD(x, y, α, β)
    Predict the label with highest probability $l = \arg\max_k \hat{y}_k$
    Check for error $l \neq y$
  end for
  return train_error, valid_error, train_predictions, valid_predictions

This question depends on the correctness to your previous parts.

1.7 [10pts] 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 depends on the correctness to your previous parts.