22417.202210 Homework 3 - Logistic Regression and Neural Networks

Pankaj Kumar Jatav

TOTAL POINTS

84 / 89

QUESTION 1

Q1 3 pts

1.1 Q1 - 1 **2 / 2**

✓ **- 0 pts** *Correct*

- 1 pts Only options 1 and 2 are correct. Missing one correct answer.

- 2 pts Only options 1 and 2 are correct. Missing both correct answers.

- 0.5 pts Only options 1 and 2 are correct. There's one wrongly selected answer.

- 1 pts Only options 1 and 2 are correct. There's two wrongly selected answers.

- 1.5 pts Only options 1 and 2 are correct. There's three wrongly selected answer.

1.2 Q1 - 2 **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Correct answer is the 3rd option.

QUESTION 2

Q2 10 pts

2.1 Q2 (a) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Wrong gradient (missing a minus)

- 0.1 pts No need for log. Simplify the expression.

- 0.1 pts No need for log

- 1 pts Click here to replace this description.

2.2 Q2 (b) - (a) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts blank

- 1 pts incorrect

- 0.1 pts Small notation error

2.3 Q2 (b) - (b) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

- 0.5 pts Incorrect (sign missing)

2.4 Q2 (b) - (c) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

- 1 pts What is $$\sigma(x)$$

2.5 Q2 (b) - (d) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

2.6 Q2 (b) - (e) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

2.7 Q2 (b) - (f) - (a) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

2.8 Q2 (b) - (f) - (b) **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

2.9 Q2 (b) - (f) - (c) **2 / 2**

✓ **- 0 pts** *Correct*

- 2 pts Blank/Incorrect

- 1 pts Partly incorrect

QUESTION 3

Neural Network Implementation 30 pts

3.1 (a) Avg. Train & Validation Cross-Entropy Loss **20 / 20**

✓ **- 0 pts** *Looks good*

- 0 pts I would have preferred a line-plot of a scatter plot instead of a bar plot, but this looks ok.

- 10 pts These shouldn't really be going down.

3.2 (b) Comments on effect of changing # of hidden units **5 / 10**

- 0 pts Correct

✓ **- 5 pts** *I'm not sure I follow the rationale here*

QUESTION 4

Learning rate 46 pts

4.1 (a) LR = 0.1 **10 / 10**

✓ **- 0 pts** *Correct*

- 5 pts This doesn't match what I and most classmates got.

- 10 pts Not following rubric

4.2 (a) LR = 0.01 **10 / 10**

✓ **- 0 pts** *Correct*

- 5 pts Does't match what I and most classmates got.

- 10 pts Blank or not following rubric.

4.3 (a) LR = 0.001 **10 / 10**

✓ **- 0 pts** *Correct*

- 10 pts Wrong or blank

- 5 pts Doesn't match what I or your other classmates got.

4.4 (b) Comments on effect of adjusting learning rate **10 / 10**

✓ **- 0 pts** *Correct*

4.5 (c) Highest validation accuracy **6 / 6**

✓ **- 0 pts** *Close enough (I got 0.898)*

- 1 pts Most people got high scores around 0.88 (I got 0.898)

QUESTION 5

5 Collaboration Questions **0 / 0**

✓ **- 0 pts** *Correct*

- 0 pts Unanswered

HOMEWORK 3
LOGISTIC REGRESSION AND NEURAL NETWORKS[1]

CS 688 MACHINE LEARNING (SPRING 2022)
https://nlp.cs.gmu.edu/course/cs688-spring22/
Note: this assignment was created by Matt Gormley

OUT: March 3, 2022
DUE: March 14, 2022

Your name: Pankaj Kumar Jatav

Your GID: G01338769

[1]Compiled on Monday 14th March, 2022 at 03:58

1 Written Questions [90 pts]

1.1 Logistic Regression and Stochastic Gradient Descent

1. (2 points) Which of the following are true about logistic regression?

   Select all that apply:

   ☒ Our formulation of binary logistic regression will work with both continuous and binary features.

   ☒ Binary Logistic Regression will form a linear decision boundary in our feature space.

   ☐ The function $\sigma(x) = \frac{1}{1+e^{-x}}$ is convex.

   ☐ The negative log likelihood function for logistic regression. $-\frac{1}{N}\sum_{i=1}^{N}\log(\sigma(\mathbf{x}^{(i)}))$ is non-convex so gradient descent may get stuck in a sub-optimal local minimum.

   ☐ None of above.

2. (1 point) The negative log-likelihood $J(\boldsymbol{\theta})$ for binary logistic regression can be expressed as

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} -y^{(i)}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)}\right) + \log\left(1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})\right)$$

   where $\mathbf{x}^{(i)} \in \mathbb{R}^{M+1}$ is the column vector of the feature values of ith data point, $y^{(i)} \in \{0,1\}$ is the i-th class label, $\boldsymbol{\theta} \in \mathbb{R}^{M+1}$ is the weight vector. When we want to perform logistic ridge regression (i.e. with ℓ_2 regularization), we modify our objective function to be

$$f(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\frac{1}{2}\sum_{j=0}^{M}\theta_j^2$$

   where λ is the regularization weight, θ_j is the jth element in the weight vector $\boldsymbol{\theta}$. Suppose we are updating θ_k with learning rate α, which of the following is the correct expression for the update?

   Select one:

   ○ $\theta_k \leftarrow \theta_k + \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(y^{(i)} - \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

   ○ $\theta_k \leftarrow \theta_k + \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} + \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) - \lambda\theta_k$

   ● $\theta_k \leftarrow \theta_k - \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} + \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

   ○ $\theta_k \leftarrow \theta_k - \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} - \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

2 Backpropagation

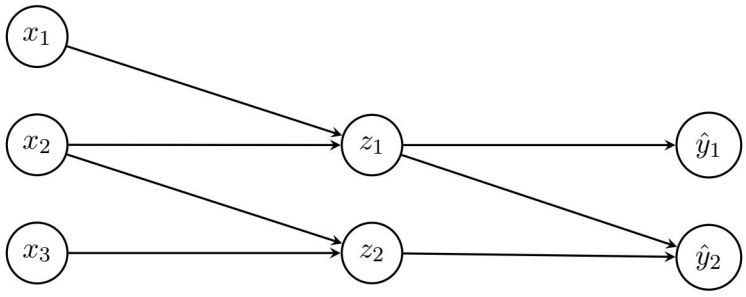


Figure 2.1: A Directed Acyclic Graph (DAG)

Consider this above Directed Acyclic Graph (DAG). Notice that it looks different than the fully-connected Neural Networks that we have seen before. Recall from lecture that you can perform back propagation on any DAG. We will work through back propagation on this graph.

Let $(\mathbf{x}, \mathbf{y})$ be the training example that is considered, where $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$ and $\mathbf{y} = \begin{bmatrix} y_1 & y_2 \end{bmatrix}^T$. All the other nodes in the graph are defined as:

$$
\begin{aligned}
z_1 &= \text{ReLU}(w_{1,1}x_1 + w_{2,1}x_2) \\
z_2 &= w_{2,2}x_2^2 + w_{3,2}x_3 + b_2 \\
\hat{y}_1 &= \sigma(m_{1,1}z_1^3 + c_1) \\
\hat{y}_2 &= m_{1,2}\sin(z_1) + m_{2,2}\cos(z_2)
\end{aligned}
$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, and $\text{ReLU}(x) = \max(0, x)$. Let $\boldsymbol{\theta}$ be the set of all parameters to be learned in this graph. We have that

$$\boldsymbol{\theta} = \{w_{1,1}, w_{2,1}, w_{2,2}, w_{3,2}, m_{1,1}, m_{1,2}, m_{2,2}, b_2, c_1\}$$

For every set of input $\mathbf{x} = (x_1, x_2, x_3)$, we will define objective of the problem as minimizing the loss function

$$J(\boldsymbol{\theta}) = \log\left((y_1 - \hat{y}_1)^2\right) + \log\left((y_2 - \hat{y}_2)^2\right)$$

Assume that you have already gone through the forward pass with inputs $\mathbf{x} = (x_1, x_2, x_3)$ and stored all the relevant values. In the following questions, you will derive the backpropagation algorithm applied to the above DAG.

(a) (1 point) First, we will derive the gradients with repect to the outputs. What are the expressions for $\frac{\partial J}{\partial \hat{y}_1}$? Write your solution in terms of $\hat{y}_1$.

Answer

$\frac{-2}{y_1 - \hat{y}_1}$

(b) Now, we will derive the gradients associated with the last layer, ie nodes y_1, y_2. Note that for the full backpropagation algorithm, you would need to calculate the gradients of the loss function with respect to every parameter $(m_{1,1}, m_{1,2}, m_{2,2}, c_1)$ as well as every input of the layer (z_1, z_2), but we are not asking for all of them in this part. For all of the questions in this part, you should use Chain Rule, and write your solution in terms of values from the forward pass, or **gradients with respect to the outputs of this layer,** $\frac{\partial \mathbf{J}}{\partial \hat{\mathbf{y}}_\mathbf{1}}$, $\frac{\partial \mathbf{J}}{\partial \hat{\mathbf{y}}_\mathbf{2}}$, because you have already calculated these values. In addition, use the sigmoid function $\sigma(x)$ in your answer instead of its explicit form.

(a) (1 point) What is the expression for $\frac{\partial J}{\partial z_1}$?

Answer

$-\frac{2}{y_1-\hat{y}_1}\hat{y}_1(1-\hat{y}_1)3m_{1,1}z_1^2 - \frac{2}{y_2-\hat{y}_2}m_{1,2}\cos(z_1)$

(b) (1 point) What is the expression for $\frac{\partial J}{\partial z_2}$?

Answer

$\frac{2}{y_2-\hat{y}_2}m_{2,2}\sin(z_2)$

(c) (1 point) What is the expression for $\frac{\partial J}{\partial m_{1,1}}$?

Answer

$-\frac{2}{y_1-\hat{y}_1}\hat{y}_1(1-\hat{y}_1)z_1^3$

(d) (1 point) What is the expression for $\frac{\partial J}{\partial m_{1,2}}$?

Answer

$-\frac{2}{y_2-\hat{y}_2}\sin(z_1)$

(e) (1 point) What is the expression for $\frac{\partial J}{\partial c_1}$?

Answer

$-\frac{2}{y_1-\hat{y}_1}\hat{y}_1(1-\hat{y}_1)$

(f) (1 point) Lastly, we will derive the gradients associated with the second layer, ie nodes z_1, z_2. Note that for the full backpropagation algorithm, you need to calculate the gradients of the loss function with respect to every parameter (every $w_{i,j}, b_2$). However, we do not need to calculate the gradients with respect to the inputs of this layer (x_1, x_2, x_3), because they are fixed inputs of the model. For all of the questions in this part, you should use Chain Rule, and write your solution in terms of values from the forward pass or **gradients with respect to the outputs of this layer,** $\frac{\partial \mathbf{J}}{\partial \mathbf{z_1}}$, $\frac{\partial \mathbf{J}}{\partial \mathbf{z_2}}$, because you have already calculated these values.

(a) (1 point) What is the expression for $\frac{\partial J}{\partial w_{2,2}}$?

Answer

$\frac{\partial J}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial z_2}\frac{\partial z_2}{\partial w_{2,2}} = \frac{2}{y_2-\hat{y}_2} m_{2,2}\sin(z_2)x_2^2$

(b) (1 point) What is the expression for $\frac{\partial J}{\partial b_2}$?

Answer

$\frac{\partial J}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial z_2}\frac{\partial z_2}{\partial b_2} = \frac{2}{y_2-\hat{y}_2} m_{2,2}\sin(z_2)$

(c) (2 points) Recall that $\text{ReLU}(x) = \max(x, 0)$. The ReLU function is not differentiable at $x = 0$, but for backpropagation, we define its derivative as

$$\text{ReLU}'(x) = \begin{cases} 0, x < 0 \\ 1, \text{ otherwise} \end{cases}$$

Now, what is the expression for $\frac{\partial J}{\partial w_{1,1}}$? Explicitly write out the cases.

Answer

$$\left(\frac{\partial J}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial z_1} + \frac{\partial J}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial z_1}\right)\frac{\partial z_1}{\partial w_{1,1}}$$
$$= \begin{cases} 0, w_{1,1}x_1 + w_{2,1}x_2 < 0 \\ -\left[\frac{2}{y_1-\hat{y}_1}\hat{y}_1(1-\hat{y}_1)3m_{1,1}z_1^2 + \frac{2}{y_2-\hat{y}_2}m_{1,2}\cos(z_1)\right]x_1, w_{1,1}x_1 + w_{2,1}x_2 \geq 0 \end{cases}$$

3 Implementing a Neural Network

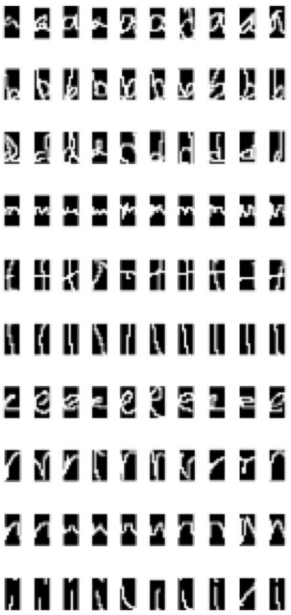


Figure 3.1: 10 random images of each of the 10 letters in the OCR dataset.

3.1 The Task

Your goal in this assignment is to implement a neural network to classify images using a single hidden layer neural network. We will use PyTorch[2], one of the currently most popular toolkits.[3]

3.2 Setup

Follow the instructions here: https://pytorch.org/get-started/locally/ in order to install the latest stable version of PyTorch (1.10.2) based on your system.

3.3 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 includes a small dataset with 60 samples *per class* (50 for training and 10 for validation). Figure 3.1 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 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 3.1 were produced by converting these pixel values into .png files for visualization. Observe

[2] https://pytorch.org/

[3] If you are familiar with a different toolkit, you're free to implement everything there, but (a) we won't be able to help you in office hours, and (b) all starter code and examples below are in PyTorch.

that no feature engineering has been done here; instead the neural network you build will *learn* features appropriate for the task of character recognition.

3.4 Model Definition

In this assignment, you will implement a single-hidden-layer neural network with a sigmoid activation function for the hidden layer, and a softmax on the output layer. Let the input vectors $\mathbf{x}$ be of length M, and the hidden layer $\mathbf{z}$ consist of D hidden units. In addition, let the output layer $\hat{\mathbf{y}}$ be a probability distribution over K classes. That is, each element $\hat{y}_k$ of the output vector represents the probability of $\mathbf{x}$ belonging to the class k.

We can compactly express this model by assuming that $x_0 = 1$ is a bias feature on the input. In this way, we have two parameter matrices $\boldsymbol{\alpha} \in \mathbb{R}^{D \times (M+1)}$ and $\boldsymbol{\beta} \in \mathbb{R}^{K \times (D)}$ (for simplicity, we will not use a bias in the output layer $\boldsymbol{\beta}$. The extra 0th column of the $\boldsymbol{\alpha}$ matrix (i.e. $\boldsymbol{\alpha}_{\cdot,0}$) holds the bias parameters.

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

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

$$
J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{3.1}
$$

In Equation 3.1, 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 $\mathbf{x}^{(i)}$. $\hat{y}_k^{(i)}$ and $y_k^{(i)}$ are the kth components of $\hat{\mathbf{y}}^{(i)}$ and $\mathbf{y}^{(i)}$ respectively.

To train, you should optimize this objective function using stochastic gradient descent (SGD), where the gradient of the parameters for each training example is computed via backpropagation. You should shuffle the training points when performing SGD using the provided `shuffle` function, passing in the epoch number as a random seed. Note that SGD has a slight impact on the objective function, where we are "summing" over the current point, i:

$$
J_{SGD}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{3.2}
$$

3.5 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 -1 to 1. The bias parameters are initialized to zero.

ONES All weights are initialized to 1.

You must support both of these initialization schemes. The provided code provides an input flag to control the initialization.

3.6 Implementation

Start with the provided `train_nn.py` and fill in the missing parts in order to implement 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, as well as the accuracy on the validation data at the end of each epoch.

Your implementation must satisfy the following requirements (see details below):

- 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 3.5, 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 on the data shuffled with the provided function. For each epoch, you must reshuffle the **original** file data, not the data from the previous epoch.
- 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.

3.7 Walkthrough Started Code

Importing useful packages and the relevant PyTorch modules.

```
import csv
import numpy as np
import argparse
import logging
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Defining the arguments to our script. All but the last one are positional.

```
parser = argparse.ArgumentParser()
parser.add_argument('train_input', type=str,
                    help='path to training input .csv file')
parser.add_argument('validation_input', type=str,
                    help='path to validation input .csv file')
parser.add_argument('train_out', type=str,
                    help='path to store prediction on training data')
parser.add_argument('validation_out', type=str,
                    help='path to store prediction on validation data')
parser.add_argument('metrics_out', type=str,
                    help='path to store training and testing metrics')
parser.add_argument('num_epoch', type=int,
                    help='number of training epochs')
parser.add_argument('hidden_units', type=int,
                    help='number of hidden units')
parser.add_argument('init_flag', type=int, choices=[1, 2],
                    help='weight initialization functions, 1: random, 2: all ones')
parser.add_argument('learning_rate', type=float,
                    help='learning rate')
parser.add_argument('--debug', type=bool, default=False,
                    help='set to True to show logging')
```

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

```
python train_nn.py tinyTrain.csv tinyValidation.csv _ _ tinyMetrics_out.txt 2 4 2
    0.1
```

(Already implemented) function that parses the input arguments, reads the datasets, and adds the bias features to the input features.

```
def args2data(parser):
    ...
```

(Already implemented) function that shuffles the data with a fixed seed (the `epoch`). **DO NOT MODIFY**

```
def shuffle(X, y, epoch):
    ...
```

(Already implemented) function that computes accuracy. **DO NOT MODIFY**

```
def accuracy(pred, gold):
    ...
```

Definition of our single-layer Feedforward Neural Network class. The `__init__` function defines the set of parameters. We will have:

- `fc1`: the linear layer that projects the input into the hidden dimension
- `sigmoid`: the non-linearity
- `fc2`: the linear layer that projects the intermediate output to the output space (producing 10 *logits*)

The class also defines the `forward` function. This is where we define the architecture (connectivity) of our network, by creating the computation graph. In our case, this is simple: we take the input `x`, pass it through the first linear layer, then the non-linearity, and then the second linear layer.

```
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, init_flag):
        # TODO: Add a flag that controls initialization
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        # We don't need PyTorch to add a bias (we have already folded it in our
    inputs)
        self.fc1 = nn.Linear(input_dim, hidden_dim, bias=False)

        # Non-linearity
        self.sigmoid = nn.Sigmoid()

        # Linear function (output)
        self.fc2 = nn.Linear(hidden_dim, output_dim, bias=False)

        # TODO: Implement the initialization. Random init is the default,
        # so you only need to implement the constant init.

    def forward(self, x):
        # Linear function  # LINEAR
        out = self.fc1(x)

        # Non-linearity  # NON-LINEAR
        out = self.sigmoid(out)

        # Linear function (readout)  # LINEAR
        out = self.fc2(out)
        return out
```

Now we're on to the main bulk of our experiments. First, let's parse the command-line arguments

```
# Parse the arguments
args = parser.parse_args()
X_tr, y_tr, X_val, y_val, out_tr, out_te, out_metrics, n_epochs, n_hid, init_flag,
    lr = args2data(args)
```

Now we will call our NN class to init our model. You'll need to call the `FeedforwardNeuralNetModel()` with the correct arguments based on the input.

```
# TODO: Init the model
net = FeedforwardNeuralNetModel(... , output_dim=10, ...)
```

We can print the details of the network or e.g. the shape of its parameters as follows.

```
# Print its details
print(net)

# The learnable parameters of a model are returned by net.parameters()
params = list(net.parameters())
print(len(params))
print(params[0].size())
print(params[1].size())
```

Now we will initialize the optimizer that we'll use, telling it that we'll use SGD, to be applied over our net's parameters (first argument) and using a learning rate equal to the one passed from the command-line.

```
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=lr)
```

We will also define the loss function that we'll use (cross-entropy). Look at the PyTorch documentation to see what inputs it expects.

```
# Choose your loss function
criterion = nn.CrossEntropyLoss()
```

Now, in the training loop (which will run for `n_epochs` epochs):

```
# Training loop:
for epoch in range(n_epochs):
```

... shuffle the data (do not change this) and print some outputs for debugging purposes (keep it there for now)

```
    # Shuffle the data
    X_tr_n, y_tr_n = shuffle(X_tr, y_tr, epoch)
    if args.debug:
        print(f"FF2 Weights term before training: {net.fc2.weight.data}")
```

... implement the training loop that goes through the (shuffled) training data and computes the forward and the backward pass

```
    # Iterate through through the training data and train
    # TODO: Implement the training loop
```

For each training example `[X,y]` you should:

```
    # 1. Clear gradients w.r.t. parameters
    optimizer.zero_grad()
    # 2. Prepare the input
    input = torch.tensor(X).float()
    input = input[None, :]
    # 3. Prepare the target label
    target = torch.tensor([y]).long()

    # 4. Forward pass to get output/logits
    output = net(input)

    # 5. Calculate Loss
    loss = criterion(output, target)

    # 6. Get gradients w.r.t. parameters
    loss.backward()

    # 7. Perform the optimization step and update the parameters
    optimizer.step()
```

Also keep this in the loop to print debug messages

```
        if args.debug:
            print(f"FF2 Weights after one update: {net.fc2.weight.data}")
```

When a training epoch is complete, you'll have to evaluate on the validation set.

```
# TODO: Evaluate on the validation set
```

You should do the following:

- For each example `[X,y]` in the validation set, prepare it in a similar manner as above.

- Run the forward pass as above to obtain the output
- Find which logit is the largest, because this will be the label predicted for this example. You can use the following function to do that (store its output!)

```
        torch.argmax(output)
```

- Compute the loss (as above)
- *Note: DO NOT call* `backward()` *or* `optimizer.step()`*. We do NOT want to train on the validation set!*
- Keep all predictions in a list (e.g. name it `predictions`) in order to compute accuracy by running

```
    validation_accuracy = accuracy(predictions, y_val)
```

. *Note: do NOT shuffle the validation data!*

Now that we have these, go back to the training and the validation loops and implement a tracker that will allow you to compute the total training loss and the total validation loss. Name these `total_train_loss` and `total_val_loss`. You can get the actual value of the loss by calling

```
float(loss.data)
```

These two will be needed for the logging functions (already implemented, see below), which will print the results for this epoch:

```
    if args.debug:
        print(f"Total validation loss for epoch {epoch}: {total_val_loss}")
        print(f"Validation predictions for epoch {epoch}: {predictions}")
        print(f"Validation accuracy: {validation_accuracy}")
    logs.append((epoch, total_train_loss/len(X_tr), total_val_loss/len(X_val),
    validation_accuracy))
```

When training is done (outside of the big training loop) the last part dumps the details of the training (epoch, average training loss, average validation loss, validation accuracy) in the log file (provided by the command-line arguments), so you can use it for visualizations in the following questions.

```
with open(out_metrics, 'w') as op:
    for e, tl, vl, acc in logs:
        op.write(f"{e},{tl},{vl},{acc}\n")
```

3.8 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` as well as `tinyMetrics.out` 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 the output that you should get if you have correctly implemented the above. We advise you to use this set to help you debug in case your implementation doesn't produce the same results as in the written part.

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

```
py train_nn.py tinyTrain.csv tinyValidation.csv _ _ tinyMetrics.out 1 5 2 0.01 --
    debug True
```

The specific output file names are not important, but be sure to keep the other arguments exactly as they are shown above.

Since it uses the constant initialization, and we use a fixed random seed for shuffling, you should get **exactly the same outputs**.

4 Empirical Questions

The following questions should be completed after you work through the programming portion of this assignment. **For any plotting questions, you must title your graph, label your axes and provide units, and provide a legend in order to receive full credit.**

For these questions, **use the large dataset**. Use the following values for the hyperparameters unless otherwise specified:

Parameter	Value
Number of Hidden Units	50
Weight Initialization	RANDOM
Learning Rate	0.01

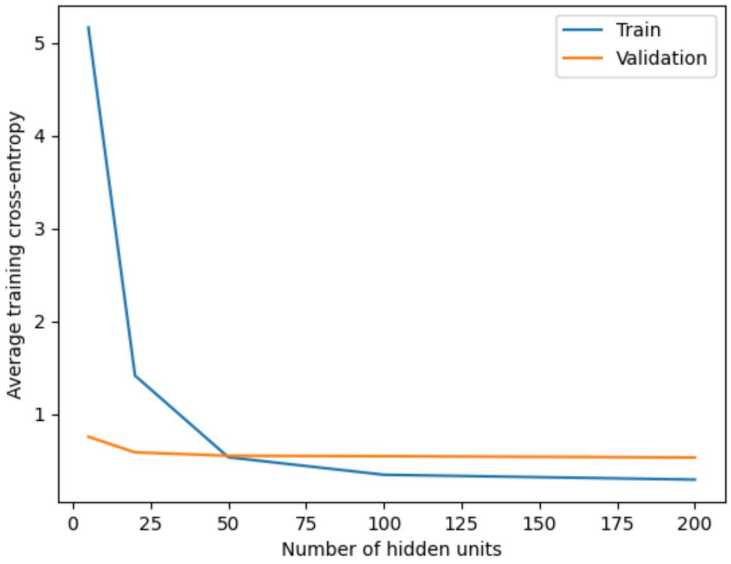
Please submit computer-generated plots for (a)i and (b)i. To get full credit, your plots must have a label for both axes with the value plotted on that axis, provide a legend labeling every line, and have a title. Note: we expect it to take about **5 minutes** to train each of these networks.

3. Hidden Units

   (a) (20 points) Train a single hidden layer neural network using the hyperparameters mentioned in the table above, except for the number of hidden units which should vary among 5, 20, 50, 100, and 200. Run the optimization for 100 epochs each time.

   Plot the average training cross-entropy (sum of the cross-entropy terms over the training dataset divided by the total number of training examples) on the y-axis vs number of hidden units on the x-axis. In the **same figure**, plot the average validation cross-entropy.

Avg. Train and Validation Cross-Entropy Loss



(b) (10 points) Examine and comment on the the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?

Answer

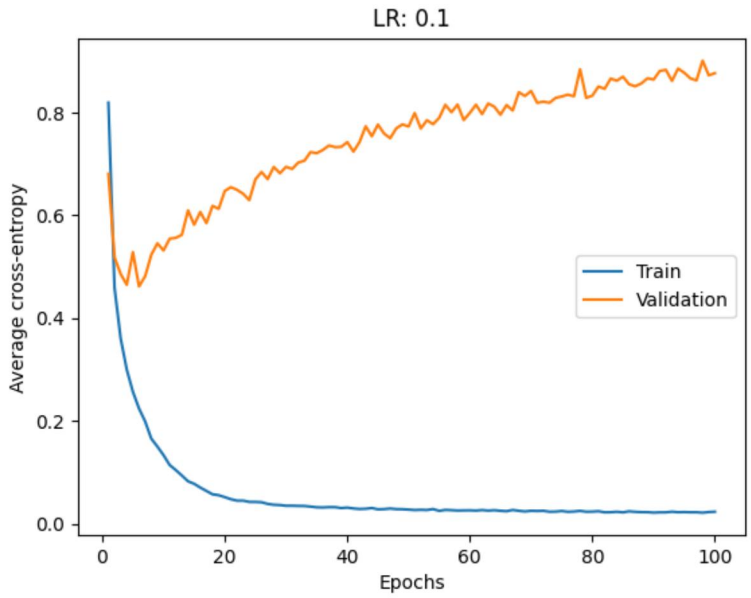
As we noticed in the train and validation average training cross-entropy meet when hidden units were 50. Before that the training loss was more which mean the model as too simple but duo to random weight initialization the mode perform gave less average training cross-entropy for validation data. But after the 50 layer model the training average cross-entropy went down, but the validation average cross-entropy remain same which menas the model over-fit with training data.

4. Learning Rate

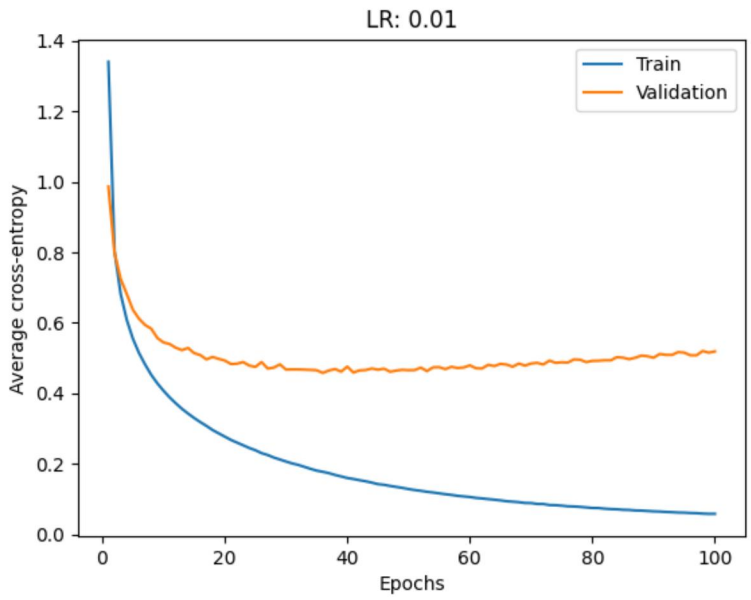
(a) (30 points) Train a single hidden layer neural network using the hyperparameters mentioned in the table above, except for the learning rate which should vary among 0.1, 0.01, and 0.001. Run the optimization for 100 epochs each time.

Plot the average training cross-entropy on the y-axis vs the number of epochs on the x-axis for the mentioned learning rates. In the **same figure**, plot the average validation cross-entropy loss. Make a separate figure for each learning rate.

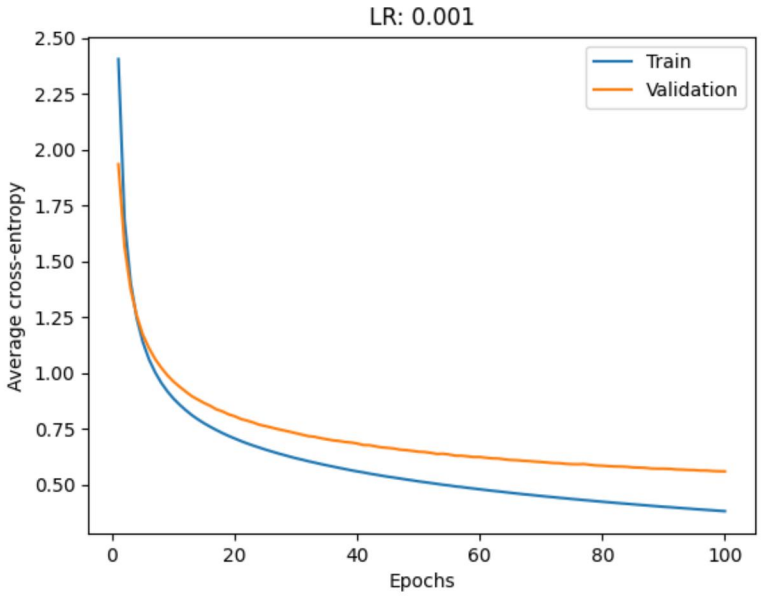
Plot LR 0.1



Plot LR 0.01



Plot LR 0.001



(b) (10 points) Examine and comment on the plots of training and validation cross-entropy. How does adjusting the learning rate affect the convergence of cross-entropy on the datasets?

Answer

When the learning rate is 0.1(too high) the training and validation, the model reach to log average training cross-entropy quickly but soon it. But as the learning rate was high instead the model did not reach to local minima and moving backward and which lead to high average validation cross-entropy.
When learning rate was 0.01 the model reach to local minima slowly as compare to 0.1 but later on the model was almost gave same average validation cross entropy.
When learning rate was 0.001 the model never reach to local minima because it the learning rate was to low.
To conclude if learning rate it too high the model may not reach to local minima or just bounce around the local minima. If the learning rate to too low the model may not reach to local minima due to very small step toward local minima.

(c) (6 points) What is the highest validation accuracy that any of your runs across all subsections achieve?

Answer

0.874

5 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found in the syllabus.

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details.
3. Did you find or come across code that implements any part of this assignment? If so, include full details.

Your Answer

No
No
No