Homework 3

KNNs, SVMs and Neural Networks

CMU 10-701: Machine Learning (Spring 2022)

piazza.com/cmu/spring2022/10701/home
OUT: Thursday, March 03, 2022

DUE: Wednesday, March 23, 2022, 11:59pm

START HERE: Instructions

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 2.1"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section in our course syllabus for more information: https://piazza.com/class_profile/get_resource/ksetdrgdkob78/ksqc9bxxjt56ic
- **Late Submission Policy:** See the late submission policy here: https://piazza.com/class_profile/get_resource/ksetdrgdkob78/ksqc9bxxjt56ic
- **Submitting your work:**
  - **Gradescope:** There will be two submission slots for this homework on Gradescope: Written and Programming.
    For the written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using the written submission slot. Please use the provided template. The best way to format your homework is by using the Latex template released in the handout and writing your solutions in Latex. However submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Each derivation/proof should be completed in the boxes provided below the question, **you should not move or change the sizes of these boxes** as Gradescope is expecting your solved homework PDF to match the template on Gradescope. If you find you need more space than the box provides you should consider cutting your solution down to its relevant parts, if you see no way to do this, please add an additional page at the end of the homework and guide us there with a 'See page xx for the rest of the solution'.
    You are also required to upload your code, which you wrote to solve the final question of this homework, to the Programming submission slot. Your code may be run by TAs so please make sure it is in a workable state.

Regrade requests can be made after the homework grades are released, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.

For multiple choice or select all that apply questions, shade in the box or circle in the template document corresponding to the correct answer(s) for each of the questions. For LaTeXusers, use ■ and ●for shaded boxes and circles, and don't change anything else. If an answer box is included for showing work, **you must show your work**!

Problem 1: K-Nearest Neighbors - Black Box [10 Points]

1. [**6 pts**] In a KNN classification problem, assume that the distance measure is not explicitly specified to you. Instead, you are given a "black box" where you input a set of instances $P_1, P_2, \ldots P_n$ and a new example Q, and the black box outputs the nearest neighbor of Q, say P_i and its corresponding class label C_i. Is it possible to construct a KNN classification algorithm (w.r.t the unknown distance metrics) based on this black box alone? If so, how and if not, why not?

2. [**4 pts**] If the black box returns the j nearest neighbors (and their corresponding class labels) instead of the single most nearest neighbor (assume $j \neq k$), is it possible to construct a KNN classification algorithm based on the black box? If so how, and if not why not?

Problem 2: Soft Margin Hyperplanes [9 points]

The soft-margin primal SVM problem is:

$$\min \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{N}\xi^{(i)} \tag{1}$$

subject to feasibility constraints that for all $i = 1, ..., N$:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi^{(i)}$$
$$\xi^{(i)} \geq 0$$

Suppose instead the optimization objective was changed to $\frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{N}\xi^{(i)2}$ while the feasibility constraints are kept the same.

Hint: Before attempting this problem, we highly recommend first trying to derive the dual formulations for the hard-margin and soft-margin SVMs presented in class by working through the steps detailed below.

1. **[2pts] Short Answer:** How is the penalty changed (relative to the original formulation 1) for points within the margin? How about misclassified points?

2. **[1 pt]** Incorporate the feasibility constraints into the objective function using the method of Lagrange multipliers by writing out the Lagrangian and the associated constraints.

3. **[3 pts]** Next, take the partial derivative of the Lagrangian with respect to all of the optimization variables. Be sure to show all of you work.

4. **[3 pts]** Finally, by setting the partial derivatives from the previous part equal to 0, derive the dual formulation of SVMs in this general case. Again, be sure to show all of you work.

Problem 3: Neural Nets: Written Questions [30 pts]

Note: We strongly encourage you to do the written part of this homework before the programming, as it will help you gain familiarity with the calculations you will have to code up in the programming section. We suggest that for each of these problems, you write out the equation required to calculate each value in terms of the variables we created (a_j, z_j, b_k, etc.) before you calculate the numerical value.

Note: For all questions which require numerical answers, round up your final answers to four decimal places. For integers, you may drop trailing zeros. We will use column vectors for the data and neural network layers, be consistent with the question in your answers involving vectors. If your answer involves application of an operator element-wise to a vector or a matrix, state it explicitly.

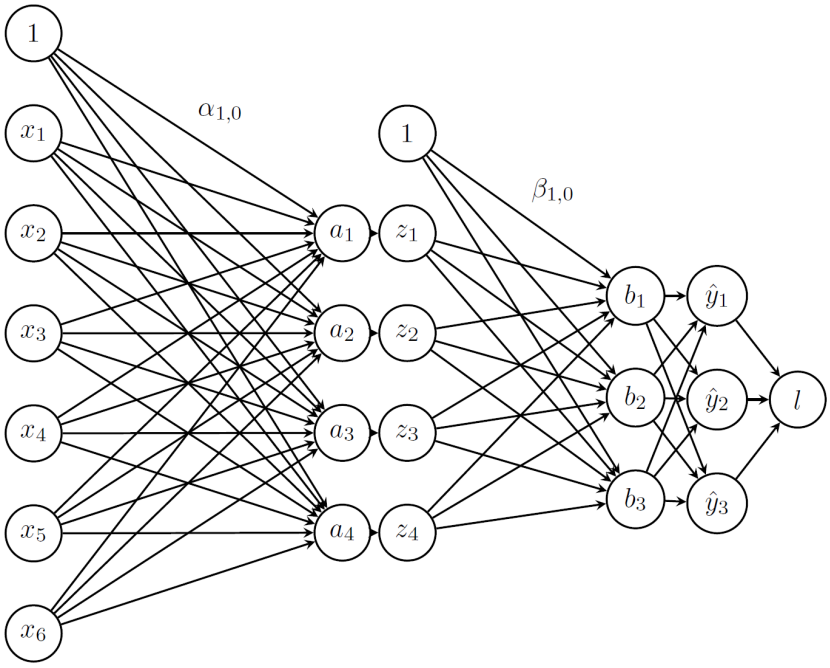

Figure 1: A One Hidden Layer Neural Network

Network Overview

Consider the neural network with one hidden layer shown in Figure 1. The input layer consists of 6 features $\mathbf{x} = [x_1, ..., x_6]^T$, the hidden layer has 4 nodes $\mathbf{z} = [z_1, ..., z_4]^T$, and the output layer is $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \hat{y}_3]^T$ that sums to one over 3 classes. We also add a bias to the input, $x_0 = 1$ and the hidden layer $z_0 = 1$, both of which are fixed to 1.

We adopt the following notation:

1. Let $\boldsymbol{\alpha}$ be the matrix of weights from the inputs to the hidden layer.
2. Let $\boldsymbol{\beta}$ be the matrix of weights from the hidden layer to the output layer.

3. Let $\alpha_{j,i}$ represent the weight going *to* the node z_j in the hidden layer *from* the node x_i in the input layer (e.g. $\alpha_{1,2}$ is the weight from x_2 to z_1)
4. Let $\boldsymbol{\beta}_{k,j}$ represent the weight going *to* the node y_k in the output layer *from* the node z_j in the hidden layer.
5. We will use a *sigmoid activation function* (σ) for the hidden layer and a *softmax* for the output layer.

Network Details

Equivalently, we define each of the following.

The input:

$$\mathbf{x} = [x_1, x_2, x_3, x_4, x_5, x_6]^T \tag{2}$$

Linear combination at first (hidden) layer:

$$a_j = \alpha_{j,0} + \sum_{i=1}^{6} \alpha_{j,i} x_i, \quad j \in \{1, \ldots, 4\} \tag{3}$$

Activation at first (hidden) layer:

$$z_j = \sigma(a_j) = \frac{1}{1 + \exp(-a_j)}, \quad j \in \{1, \ldots, 4\} \tag{4}$$

Linear combination at second (output) layer:

$$b_k = \beta_{k,0} + \sum_{j=1}^{4} \beta_{k,j} z_j, \quad k \in \{1, \ldots, 3\} \tag{5}$$

Activation at second (output) layer:

$$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=1}^{3} \exp(b_l)}, \quad k \in \{1, \ldots, 3\} \tag{6}$$

Note that the linear combination equations can be written equivalently as the product of the weight matrix with the input vector. We can even fold in the bias term α_0 by thinking of $x_0 = 1$, and fold in β_0 by thinking of $z_0 = 1$.

Loss

We will use cross entropy loss, $\ell(\hat{\mathbf{y}}, \mathbf{y})$. If $\mathbf{y}$ represents our target (true) output, which will be a one-hot vector representing the correct class, and $\hat{\mathbf{y}}$ represents the output of the network, the loss is calculated as (note that the log terms are in base e):

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{3} y_k \log(\hat{y}_k) \tag{7}$$

1. **[6 pts]** In the following questions you will derive the matrix and vector forms of the previous equations which define our neural network. These are what you should hope to program in order to avoid excessive loops and large run times.
   When working these out it is important to keep a note of the vector and matrix dimensions in order for you to easily identify what is and isn't a valid multiplication.
   Suppose you are given a training example: $\mathbf{x}^{(1)} = [x_1, x_2, x_3, x_4, x_5, x_6]^T$ with **label class 2**, so $\mathbf{y}^{(1)} = [0, 1, 0]^T$. We initialize the network weights as:

$$\boldsymbol{\alpha}^* = \begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \end{bmatrix}$$

$$\boldsymbol{\beta}^* = \begin{bmatrix} \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} \\ \beta_{2,1} & \beta_{2,2} & \beta_{2,3} & \beta_{2,4} \\ \beta_{3,1} & \beta_{3,2} & \beta_{3,3} & \beta_{3,4} \end{bmatrix}$$

   We want to also consider the bias term and the weights on the bias terms ($\alpha_{j,0}$ and $\beta_{k,0}$). To account for these we can add a new column to the beginning of our initial weight matrices.

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,0} & \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,0} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \end{bmatrix}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} & \beta_{1,4} \\ \beta_{2,0} & \beta_{2,1} & \beta_{2,2} & \beta_{2,3} & \beta_{2,4} \\ \beta_{3,0} & \beta_{3,1} & \beta_{3,2} & \beta_{3,3} & \beta_{3,4} \end{bmatrix}$$

   And we can set our first value of our input vectors to always be 1 ($x_0^{(i)} = 1$), so our input becomes:

$$\mathbf{x}^{(1)} = [1, x_1, x_2, x_3, x_4, x_5, x_6]^T$$

(a) **[1 pt]** By examining the shapes of the initial weight matrices, how many neurons (or nodes) do we have in the first hidden layer of the neural network? (Not including the bias neuron)

(b) **[1 pt]** How many output neurons will our neural network have?

(c) **[1 pt]** What is the vector $\mathbf{a}$ whose elements are made up of the entries a_j in equation (4). Write your answer in terms of $\boldsymbol{\alpha}$ and $\mathbf{x}^{(1)}$.

(d) **[1 pt]** What is the vector $\mathbf{z}$ whose elements are made up of the entries z_j in equation (5)? Write your answer in terms of $\mathbf{a}$.

(e) **[1 pt] Select one:** We cannot take the matrix multiplication of our weights $\boldsymbol{\beta}$ and our vector $\mathbf{z}$ since they are not compatible shapes. Which of the following would allow us to take the matrix multiplication of $\boldsymbol{\beta}$ and $\mathbf{z}$ such that the entries of the vector $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$ are equivalent to the values of b_k in equation (5)?

- ◯ Remove the last column of $\boldsymbol{\beta}$
- ◯ Remove the first row of $\mathbf{z}$
- ◯ Append a value of 1 to be the first entry of $\mathbf{z}$
- ◯ Append an additional column of 1's to be the first column of $\boldsymbol{\beta}$
- ◯ Append a row of 1's to be the first row of $\boldsymbol{\beta}$
- ◯ Take the transpose of $\boldsymbol{\beta}$

(f) **[1 pt]** What are the entries of the output vector $\hat{\mathbf{y}}$? Your answer should be written in terms of b_1, b_2, b_3.

2. **[7 pts]** We will now derive the matrix and vector forms for the backpropagation algorithm.

$$\frac{d\ell}{d\boldsymbol{\alpha}} = \begin{bmatrix} \frac{dJ}{d\alpha_{10}} & \frac{dJ}{d\alpha_{11}} & \cdots & \frac{dJ}{d\alpha_{1M}} \\ \frac{dJ}{d\alpha_{20}} & \frac{dJ}{d\alpha_{21}} & \cdots & \frac{dJ}{d\alpha_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dJ}{d\alpha_{D0}} & \frac{dJ}{d\alpha_{D1}} & \cdots & \frac{dJ}{d\alpha_{DM}} \end{bmatrix}$$

The mathematics which you have to derive in this section jump significantly in difficultly, you should always be examining the shape of the matrices and vectors and making sure that you are comparing your matrix elements with calculations of individual derivatives to make sure they match (e.g. the element of the matrix $(\frac{d\ell}{d\alpha})_{2,1}$ should be equal to $\frac{d\ell}{d\alpha_{2,1}}$). Recall that ℓ is our loss function defined in equation (1.8)

(a) **[1 pt]** The derivative of the softmax function with respect to b_k is as follows:

$$\frac{d\hat{y}_l}{db_k} = \hat{y}_l(\mathbb{I}[k = l] - \hat{y}_k)$$

where $\mathbb{I}[k = l]$ is an indicator function such that if $k = l$ then it it returns value 1 and 0 otherwise. Using this, write the derivative $\frac{d\ell}{db_k}$ in a smart way such that you do not need this indicator function. Write your solutions in terms of $\hat{y}_k, y_k$.

(b) **[1 pt]** What are the elements of the vector $\frac{d\ell}{db}$? (Recall that $\mathbf{y}^{(1)} = [0, 1, 0]^T$)

(c) **[1 pt]** What is the derivative $\frac{d\ell}{d\beta}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{b}}$ and $\mathbf{z}$.

(d) **[1 pt]** Explain in one short sentance why must we go back to using the matrix $\boldsymbol{\beta}^*$ (The matrix $\boldsymbol{\beta}$ without the first column of ones) when calculating the matrix $\frac{d\ell}{d\alpha}$?

(e) **[1 pt]** What is the derivative $\frac{d\ell}{d\mathbf{z}}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{b}}$ and $\boldsymbol{\beta}^*$

(f) **[1 pt]** What is the derivative $\frac{d\ell}{d\mathbf{a}}$ in terms of $\frac{d\ell}{d\mathbf{z}}$ and $\mathbf{z}$?

(g) **[1 pt]** What is the matrix $\frac{d\ell}{d\alpha}$? Your answer should be in terms of $\frac{d\ell}{d\mathbf{a}}$ and $x^{(1)}$.

Prediction

When doing prediction, we will predict the argmax of the output layer. For example, if $\hat{\mathbf{y}}$ is such that $\hat{y}_1 = 0.3,\ \hat{y}_2 = 0.2,\ \hat{y}_3 = 0.5$ we would predict class 3 for the input $\mathbf{x}$. If the true class from the training data $\mathbf{x}$ was 2 we would have a one-hot vector $\mathbf{y}$ with values $y_1 = 0,\ \ y_2 = 1,\ \ y_3 = 0$.

3. **[8 pts]** We initialize the weights as:

$$\boldsymbol{\alpha}^* = \begin{bmatrix} 2 & 1 & -1 & -1 & 0 & -2 \\ 0 & 1 & 0 & -1 & 1 & 3 \\ -1 & 2 & 1 & 3 & 1 & -1 \\ 1 & 3 & 4 & 2 & -1 & 2 \end{bmatrix}$$

$$\boldsymbol{\beta}^* = \begin{bmatrix} 2 & -2 & 2 & 1 \\ 3 & -1 & 1 & 2 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

And weights on the bias terms ($\alpha_{j,0}$ and $\beta_{j,0}$) are initialized to 1.

You are given a training example $\mathbf{x}^{(1)} = [1, 0, 1, 0, 1, 0]^T$ with label class 2, so $\mathbf{y}^{(1)} = [0, 1, 0]^T$. Using the initial weights, run the feed forward of the network over this training example (without rounding during the calculation) and then answer the following questions.

(a) **[1 pt]** What is the value of a_1?

(b) **[1 pt]** What is the value of z_1?

(c) **[1 pt]** What is the value of a_3?

(d) **[1 pt]** What is the value of z_3?

(e) **[1 pt]** What is the value of b_2?

(f) **[1 pt]** What is the value of $\hat{y}_2$?

(g) **[1 pt]** Which class value we would predict on this training example?

(h) **[1 pt]** What is the value of the total loss on this training example?

4. **[4 pts]** Now use the results of the previous question to run backpropagation over the network and update the weights. Use the learning rate $\eta = 1$.

   Do your backpropagation calculations without any rounding then answer the following questions: (in your final responses round to four decimal places)

   (a) **[1 pt]** What is the updated value of $\beta_{2,1}$?

   (b) **[1 pt]** What is the updated weight of the hidden layer bias term applied to y_1 (i.e. $\beta_{1,0}$)?

   (c) **[1 pt]** What is the updated value of $\alpha_{3,4}$?

   (d) **[1 pt]** If we ran backpropagation on this example for a large number of iterations and then ran feed forward over the same example again, which class would we predict?

5. **[5 pts]** Let us now regularize the weights in our neural network. For this question, we will incorporate L2 regularization into our loss function $\ell(\hat{\mathbf{y}}, \mathbf{y})$, with the parameter λ controlling the weight given to the regularization term.

   (a) **[1 pt]** Write the expression for the regularized loss function of our network after adding L2 regularization (**Hint:** Remember that bias terms should not be regularized!)

   (b) **[1 pts]** Compute the regularized loss for training example $\mathbf{x}^{(1)}$ (assume $\lambda = 0.01$ and use the weights before backpropagation)

   (c) **[1 pts]** For a network which uses the regularized loss function, write the gradient update equation for $\alpha_{j,i}$. You may use $\frac{\partial \ell(\hat{\mathbf{y}}, \mathbf{y})}{\partial \alpha_{j,i}}$ to denote the gradient update w.r.t non-regularized loss and η to denote the learning rate.

   (d) **[2 pts]** Based on your observations from previous questions, **select all statements which are true**:

   ☐ The non-regularized loss is always higher than the regularized loss

   ☐ As weights become larger, the regularized loss increases faster than non-regularized loss

   ☐ On adding regularization to the loss function, gradient updates for the network become larger

   ☐ When using large initial weights, weight values decrease more rapidly for a network which uses regularized loss

   ☐ None of the above

Problem 4: Neural Network Implementation [51 pts]

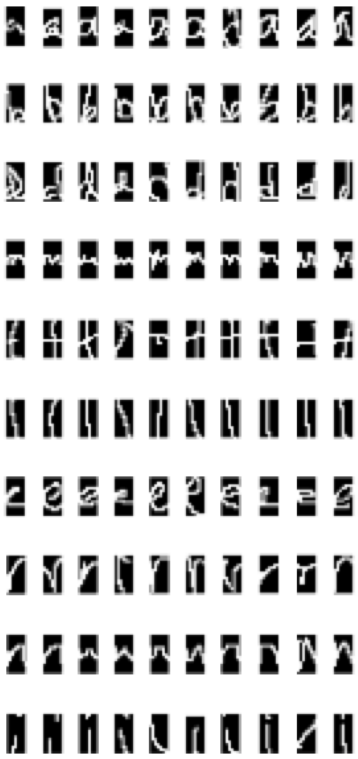

Figure 2: 10 Random Images of Each of 10 Letters in OCR

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.

0.1 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), a medium dataset with 600 samples per class (500 for training and 100 for), and a large dataset with 1000 samples per class (900 for training and 100 for). 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 . 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.

0.2 Model Definition

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

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

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

$$
\begin{aligned}
a^j &= \sum_{m=0}^{M} \alpha^{jm} x^m \\
z^j &= \frac{1}{1 + \exp(-a^j)} \\
b^k &= \sum_{j=0}^{D} \beta^{kj} z^j \\
\hat{y}^k &= \frac{\exp(b^k)}{\sum_{l=1}^{K} \exp(b^l)}
\end{aligned}
$$

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

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

Some points to mention:

- Do *not* use any machine learning libraries. You may and please do use NumPy.
- Try to "vectorize" your code as much as possible. In Python, you want to avoid for-loops and instead rely on numpy calls to perform operations such as matrix multiplication, transpose, subtraction, etc. over an entire numpy array at once. This is much faster; using NumPy over list can speed up your computation by 200x!
- You'll want to pay close attention to the dimensions that you pass into and return from your functions.
- Run your implementation locally first against the auto grader, you may need to run it a few times locally to achieve a full score, because of the stochastic nature of the assignment. Once you are able to get a full score locally, submit it to Gradescope. Again note that you may have to submit to Gradescope a few times to get full score, because of the stochastic nature of this assignment.

0.3 Feed Propagation Implementation [8 pts]

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

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

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

$$J_{SGD}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\sum_{k=1}^{K} y^{(i)^k} \log(\hat{y}^{(i)^k}) \tag{8}$$

J is a function of the model parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ because $\hat{y}^{(i)k}$ 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, $\hat{y}^{(i)^k}$ and $y^{(i)^k}$ are the kth components of $\hat{\mathbf{y}}^{(i)}$ and $\mathbf{y}^{(i)}$ respectively.

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

$$J(\boldsymbol{\alpha}, \boldsymbol{\beta}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y^{k(i)} \log(\hat{y}^{(i)k}) \tag{9}$$

0.3.2 Complete Forward Pass

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

Algorithm 1 Forward Computation

1: **procedure** NNFORWARD(Training example ($\mathbf{x}$, $\mathbf{y}$), Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
2: $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \boldsymbol{\alpha})$
3: $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$
4: $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \boldsymbol{\beta})$
5: $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$
6: $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$
7: **return** intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, \mathbf{J}$

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

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

0.4 Backward Propagation Implementation [12 pts]

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

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

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

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

Reminder once again that $\boldsymbol{\alpha}$ and $\mathbf{g}_{\boldsymbol{\alpha}}$ are $D \times (M+1)$ matrices, while $\boldsymbol{\beta}$ and $\mathbf{g}_{\boldsymbol{\beta}}$ are $K \times (D+1)$ matrices. The $+1$ comes from the extra columns $\boldsymbol{\alpha}^{\cdot,0}$ and $\boldsymbol{\beta}^{\cdot,0}$ which are the bias parameters for the first and second layer respectively. We will always assume $x^0 = 1$ and $z^0 = 1$.

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

Algorithm 2 Backpropagation

1: **procedure** NNBackward(Training example ($\mathbf{x}$, $\mathbf{y}$), Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $z, \hat{y}$)
2: Place intermediate quantities $\mathbf{z}, \hat{\mathbf{y}}$ in scope
3: $\mathbf{g}_{\mathbf{b}} = \text{softmaxBackward}^*(\mathbf{y}, \hat{\mathbf{y}})$
4: $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LinearBackward}(\mathbf{z}, \boldsymbol{\beta}, \mathbf{g}_{\mathbf{b}})$
5: $\mathbf{g}_{\mathbf{a}} = \text{SigmoidBackward}(\mathbf{z}, \mathbf{g}_{\mathbf{z}})$
6: $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LinearBackward}(\mathbf{x}, \boldsymbol{\alpha}, \mathbf{g}_{\mathbf{a}})$ ▷ We discard $\mathbf{g}_{\mathbf{x}}$
7: **return** parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{b}}, \mathbf{g}_{\mathbf{z}}, \mathbf{g}_{\mathbf{a}}$

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

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

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

0.5 Training with SGD [10 pts]

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

For testing on Gradescope, we make a few simplifications: (1) you should *not* shuffle your data and (2) you will use a fixed learning rate. In the real world, you would *not* make these simplifications.

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

Algorithm 3 Stochastic Gradient Descent (SGD) without Shuffle

1: **procedure** SGD(Training data $\mathcal{D}$, Validation data $\mathcal{D}'$, other relevant parameters)
2: Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ ▷ Use either RANDOM or ZERO from Section 0.5.1 **for** $e \in \{1, 2, \ldots, E\}$ **do**
end
For each epoch **for** $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
end
For each training example (No shuffling)
3: Compute neural network layers:
4: $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
5: Compute gradients via backprop:
6: $\mathbf{g}_{\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} \mathbf{g}_{\boldsymbol{\beta}} = \frac{\partial J}{\partial \boldsymbol{\beta}} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{z}, \hat{y})$
7: Update parameters:
9: $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
10: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
11:
12: Store training mean cross-entropy $J(\boldsymbol{\alpha}, \boldsymbol{\beta})$ ▷ from Eq. 9
13: Store validation mean cross-entropy $J(\boldsymbol{\alpha}, \boldsymbol{\beta})$ ▷ from Eq. 9
14:
15: **return** $\boldsymbol{\alpha}, \boldsymbol{\beta}$, cross_entropy_train_list, cross_entropy_valid_list

Note that training and validation losses (lines 13, 14) should be calculated **after** each epoch with the **updated** α and β (which includes running through the training set *again*).

0.5.1 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.

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

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

0.6 Label Prediction [5 pts]

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.

The error is given by:

$$\text{error} = \frac{\#\text{ incorrect}}{\#\text{ total}} = 1 - \text{accuracy}$$

Implement the `prediction` function as follows.

Algorithm 4 Prediction

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

return train_error, valid_error, train_predictions, valid_predictions
=0

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

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

0.7 Main `train_and_valid` function [4 pts]

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 0.5.1, 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, we ask that you **DO NOT** shuffle trials in this assignment.

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

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

0.8 Analysis [12 pts]

The following questions should be completed after you work through the programming portion of this assignment. The programming portion will be worth 30 points.

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 question 1 under sections 2.8.1 and 2.8.2. Include any code required to produce these results in `additional_code.py` when submitting the programming component. Note: we expect it to take about **5 minutes** to train each of these networks.

0.8.1 Hidden Units [7 pts]

1. [**5 pts**] 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.

2. [**2 pts**] Examine and comment on the the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?

0.8.2 Learning Rate [5 pts]

1. **[3 pts]** 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.

2. **[2 pts]** Examine and comment on the the plots of training and validation cross-entropy. How does adjusting the learning rate affect the convergence of cross-entropy of each dataset?

0.9 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.

1 Collaboration Questions

1. (a) Did you receive any help whatsoever from anyone in solving this assignment?

   (b) If you answered 'yes', give full details (e.g. "Jane Doe explained to me what is asked in Question 3.4")

2. (a) Did you give any help whatsoever to anyone in solving this assignment?

   (b) If you answered 'yes', give full details (e.g. "I pointed Joe Smith to section 2.3 since he didn't know how to proceed with Question 2")

3. (a) Did you find or come across code that implements any part of this assignment?

   (b) If you answered 'yes', give full details (book & page, URL & location within the page, etc.).