CPSC 470 – Artificial Intelligence
Problem Set #6 – Deep Neural Network
35 points
Due Wednesday April 10th, 11:59:59pm

Some reminders:

- **Grading contact:** Irene Li (irene.li@yale.edu) is the point of contact for initial questions about grading for this problem set.
- **Late assignments** are not accepted without a Dean's excuse.
- **Collaboration policy:** You are encouraged to discuss assignments with the course staff and with other students. However, you are required to implement and write any assignment on your own. This includes both pencil-and-paper and coding exercises. You are not permitted to copy, in whole or in part, any written assignment or program as part of this course. You are not to take code from any online repository or web source. You will not allow your own work to be copied. Homework assignments are your individual responsibility, and plagiarism will not be tolerated.
- **Students taking CPSC570:** There is no extra section for this assignment. Your assignment is the same as CPSC470.

In this exercise, you will implement part of a deep neural network and apply it to the task of hand-written digit recognition. This assignment is adapted from Andrew Ng's machine learning class on Coursera.

We encourage to type in your answers directly on this handout. Please only put answers in the designated areas, as we will ignore anything outside those designated areas.

Linear Algebra and Numpy

Neural networks rely upon linear algebra. For the purpose of this assignment, you don't need to know a lot about matrices to finish this assignment as most of the code has been implemented for you. However, if you find it challenging, here are some basics of linear algebra that may help:
https://minireference.com/static/tutorials/linear_algebra_in_4_pages.pdf
https://math.boisestate.edu/~wright/courses/m365/LAIntroSlides.pdf

To represent matrices in python, we will use the **numpy** library. Most of the code is already implemented, but you may find the documentation for numpy (http://www.numpy.org/) or these other references useful:
http://cs231n.github.io/python-numpy-tutorial/#numpy
https://www.numpy.org/devdocs/user/quickstart.html

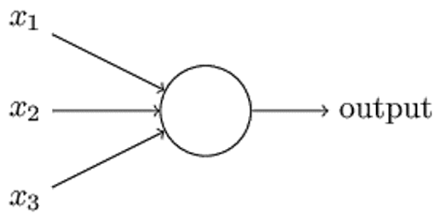
All of the libraries needed of this assignment has been installed on zoo. As before, we unfortunately won't be able to help with library installation problems on personal machines.

Dataset

The dataset you will use is taken and modified from the MNIST digit dataset (http://yann.lecun.com/exdb/mnist/). The dataset consists of 5000 handwritten digit images and the corresponding labels. Each image is 20 by 20 pixels. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The figure below shows some examples from the dataset:

Neural Networks

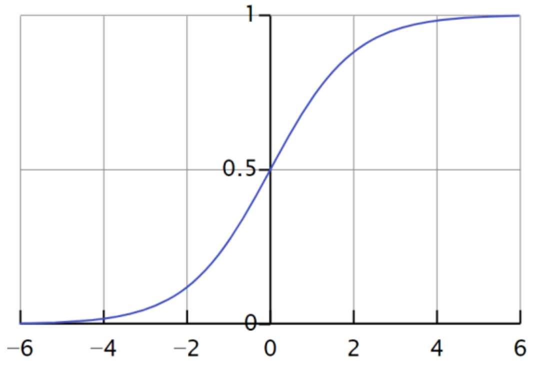
We will use a typical model of an individual neuron:



A neuron computes a weighted sum of its inputs: z = w1 * x1 + w2 * x2 + w3 * x3 where w1, w2, and w3 are the weights correspondingly. The output is a = g(z) where g is a non-linear activation function. In this assignment, we use the sigmoid function which is defined by:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

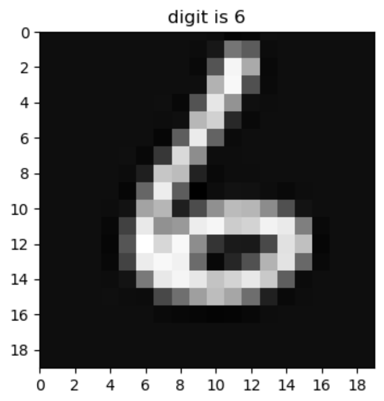
A sigmoid function looks like this:



In the basic sigmoid shown above, the transition point is at x=0. To shift the threshold to the left or to the right, we can add an additional term into the sum. (For example, if we add -4 into the sum, this is the same as having the sigmoid function transition at x=4.). This shift is called a **bias.** To simplify the computation of the bias term, we will add one additional neuron to each layer (except the output layer) and always assign an input value of 1 to that neuron. The weight on that constant input thus determines the bias and can be adjusted automatically by any algorithm that adjusts the network weights.

The neural network is composed of a connected set of individual neurons. There are many units in each layer, and there are multiple layers. In this assignment, we will first use a 3-layer neural net: an input layer, a hidden layer and an output layer. The input layer will have one input neuron for each pixel in the image, plus one additional neuron for the bias term, giving a total of 401 input layer neurons.

The training data will be loaded into the variables train_x and train_y by the function load_data(training_percentage). We will soon vary the training_percentage, but for now let's leave it as 1 (using the entire data set for training). The variable train_x contains 5000 vectorized input images, and the variable train_y stores the corresponding correct output labels (such as 6, 1, 2, etc.) To visualize a sample from the training examples, you can use the function display_digit_image(…). One example output from this display function is shown below:



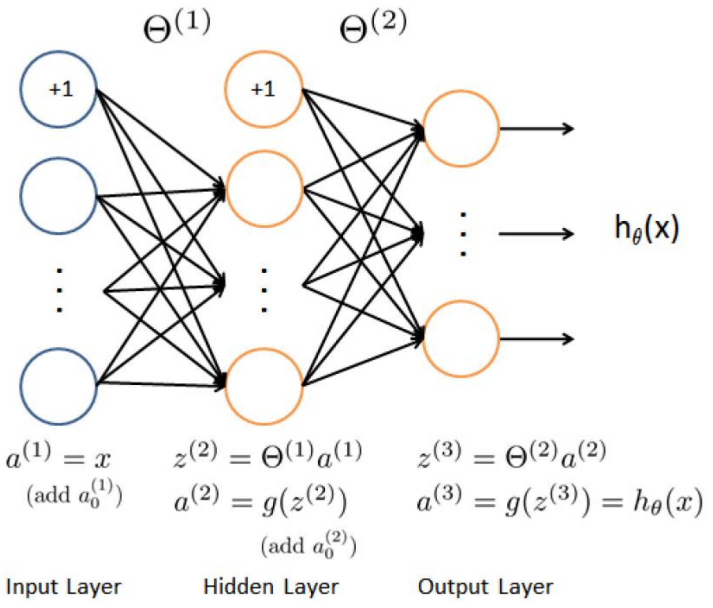
We will first use a hidden layer with 25 neurons, and an output layer of 10 neurons.

Please answer the following question:
Q1. Why there are 10 units in the output layer? Please choose one (1 point):

$\boxed{\text{B}}$

A. Because 10 can be divided by 400 * 25;
B. Because there are 10 labels;
C. The number 10 is generated randomly, and it can be any number here.
D. Because 9 is a perfect square and then we add one for the bias neuron.

The first version of our neural network will look like this:



The vectorized image will be provided to the input layer, with each pixel corresponding to a single input neuron. The output of the first layer is simply these pixel values (along with the fixed +1 bias neuron.). These values will be weighted and then passed as inputs to the hidden layer. The hidden layer will compute the weighted sum, pass the values through a sigmoid activation function, and then pass along its output to the third layer. These are again weighted, summed, and then passed through an activation function. The output layer neuron with the largest output value will be considered the winner and the label representing that node will be the image's label. (This is the **feed-forward** process which computes the output of the system, given an input image.)

Weights are initialized randomly, so our initial output will likely be very different from the true label. A **cost function** is used to evaluate how different it is between the true label and the predicted label. The **backpropogation** algorithm (as presented in class) is then used to iteratively update the weights in the network to (hopefully) bring the actual output of the network to be closer to the desired output.

Most of the code has already been implemented. Your task is to complete the deep_NN function by choosing the correct functions. You will implement about 4 lines of code.

Feedforward

The function *initialize_parameters* returns a dictionary *parameters*, with the keywords "W1", "b1", "W2" and "b2". The "W*" represents the weight matrix, and the "b" is the bias vector. "1" means the parameters from the input layer to hidden layer, and "2" represents the parameters from the hidden layer to the output layer. By examine the output of the *initialize_parameters* function, please fill in the dimensions of the parameters in the table below (you may need to write a few lines of code to call the *initialize_parameters*, however, you don't need to submit any code you wrote you this part. You can use the *np.shape*() function).

For example, for the array which has 2 rows and 3 columns:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}$$

It will be initialized as a = np.array([[1, 2, 3], [4, 5, 6]]) and a.shape is: (2, 3)

Then you will fill in the form as

a	2	3

Q2. Please fill in the dimension (2 points)

W1	25	400
b1	25	1
W2	10	25
b2	10	1

Now let's examine the dimensions of each layer. The feedforward function returns a dictionary of caches, with the keyword "a1, z2, a2, z3, a3". "z*" represents the weighted result at the corresponding layer. "a*" represents the value of sigmoid("z*") at the layer. "1" refers to input layer, "2" refers to the hidden layer, and "3" refers to the output layer. Please note that the overall output of the neural net is also represented as AL/al, which should share the same value as a3.

Please answer the following questions
Q3. Why there is no "z1"? Please choose one (1 point): A
A. Because 1 is the input layer, the output of layer 1 is the pixel values themselves;
B. Because the TA made a mistake, there should be a "z1";
C. "z1" should exist, but it is just not used in the calculation later, so this value is excluded.

Q4. Please fill in the dimensions below (4 points):

X (input, which is train_x)	400	5000
y (true label, which is train_y)	1	5000
Y (converted label, which is rehape_Y(train_y))	10	5000
a1	400	5000
z2	25	5000
a2	25	5000
z3	10	5000
a3	10	5000

Here you can see that the dimension of Y and y is different. This is because the raw data provided 1, 2, 3, 4 as labels, and the output of the neural net is a vector of length 10, with index 0 corresponds to the probability of digit 0, index 1 corresponds to the probability of digit 1, etc. (We will describe column vectors using "**.T**" to indicate the transpose.)

Please answer the following questions.
Q5. Given a column of y: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].T . From 0 to 9, which digit is this label corresponding to (1 point): $\boxed{2}$

Q6. Given a column of a3 from a trained neural network with accuracy 99.4%: [0.02, 0.06, 0.1, 0.0004, 0.9, 0.3, 0.1, 0.025, 0.062, 0.12].T . From 0 to 9, which digit does this image mostly likely represents (1 point): $\boxed{4}$

That's all for feedforward. You don't need to implement any part of the feedforward algorithm. We won't go into the details in this assignment, but if you are curious about how the difference between the prediction and the true label is calculated, here is the formula:

$$J(w,b) = \frac{1}{m} sum(-Y\ \log(1-a3) - (1-Y)\log(1-a3))$$
$$where\ m\ is\ the\ number\ of\ samples$$

Backpropogation

As we described in lecture, the backpropagation algorithm iteratively adjusts the weights in the network so that the output of the network is a closer match to the desired output. We will start at the output layer and move backward through the network to assign "blame" and change the weights to better match our data. We denote the differences at each layer dA*, where * is the layer, and the updated weights (denoted as dW*) and bias terms (denoted as db*).

At the output layer, dA3 is calculated as:

$$dA3 = a3 - Y$$

At the hidden layer, the dA2, dW2 and db2 are calculated as (* is matrix multiplication, and **.*** is element-wise multiplication):

$$dA2 = (W2.T * dA3)\ .*\ sigmoid\_gradient(z2)$$
$$dW2 = dA3 *\ a2.T\ /\ m$$
$$db2 = dA3 * vector\ of\ 1s\ with\ length\ of\ a2.shape[1]\ /\ m$$

At the output layer, dW1 and db1 is calculated as:

$$dW1 = dA2 * a1.T / m$$
$$db2 = dA2 * vector\ of\ 1s\ with\ the\ length\ of\ a1.shape[1]\ /\ m$$

Please answer the following question.
Q7. Why there is no dA1? Please choose one (1 point): $\boxed{\text{B}}$
A. dA1 should exist, but it is just not used in the calculation later, so this value is excluded;
B. If we are going to calculate dA1, that will be the difference between the desired value and the actual value. The actual value is the image, and we cannot modify the input, so there is no need to compute dA1.
C. Because the TA made a mistake, there should be dA1.

The formulas provided above are specific to a 3-layer neural network. Please think about how to generalize it to an n-layer neural network and answer the following questions.

Q8. For the output layer of an n-layer neural network, dAn is calculated the same way, which is (1 point): $\boxed{\text{C}}$

A. $\boldsymbol{dAn = Y + an}$
B. $\boldsymbol{dAn = Y - an}$
C. $\boldsymbol{dAn = an - Y}$

Q9. For a hidden layer *i* of an n-layer neural network, the dAi is calculated as (1 point): $\boxed{\text{A}}$

A. $\boldsymbol{dAi = (Wi.T * dA(i+1)) .* \ sigmoid\_gradient(zi)}$
B. $\boldsymbol{dAi = (W(i+1).T * dA(i+1)) .* \ sigmoid\_gradient(z(i+1))}$
C. $\boldsymbol{dAi = (Wi.T * dA(i)) .* \ sigmoid\_gradient(zi)}$

Q10. For any layer other than the output layer of an n-layer neural network, the dWi is calculated as (1 point): $\boxed{\text{B}}$

A. $\boldsymbol{dWi = dAi * \ a(i+1).T \ / \ m}$
B. $\boldsymbol{dWi = dA(i+1) * \ ai.T \ / \ m}$
C. $\boldsymbol{dWi = dA(i+1) * \ ai.T}$

Q11. For any layer other than the output layer of an n-layer neural network, the dbi is calculated as (1 point): $\boxed{\text{C}}$

A. $\boldsymbol{dbi = dAi * vector\ of\ 1s\ with\ the\ length\ of\ ai.shape[1] \ / \ m}$
B. $\boldsymbol{dbi = dA(i+2) * vector\ of\ 1s\ with\ the\ length\ of\ ai.shape[1] / \ m}$
C. $\boldsymbol{dbi = dA(i+1) * vector\ of\ 1s\ with\ the\ length\ of\ ai.shape[1] / \ m}$

Now that you understand how the weights are updated, please implement the corresponding part in the function *backpropogation*. You will write about 5 lines of code.

Now let's take a look at how the weights and biases are updated. The differences dWi and dbi, could be used to directly update these terms:

$$Wi = Wi - dWi$$
$$bi = bi - dbi$$

However, we would like to have some control of how the parameters are updated, so we add a scalar coefficient here called learning rate. As the name indicates, the learning rate describes how quickly the model learns or update itself. A higher learning rate will learn faster, but also be more sensitive to noise in the inputs and desired outputs. Our final update rules are:

$$Wi = Wi - learning\_rate\ * dWi$$
$$bi = bi - learning\_rate\ * dbi$$

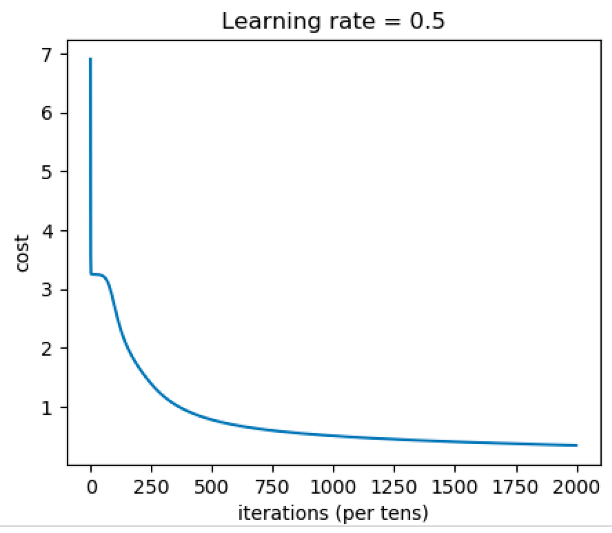
Using the formula above, please complete the *update_parameters* function. You will implement about 3 lines of code.

Train the neural network

Congratulations! Now you have a completed fully-connected neural network. Let's train the neural network by running the existing code in the main section under "section 1". Please do not modify any of the hyper-parameters here like *training_percentage, learning_rate, num_iterations*, etc. It will take a few minutes to run and you will see the cost of every iteration printed to screen. At the end, it will print the accuracy of this model on the training set and a figure with the cost of each iteration.

Q12. Please fill in the accuracy (4 points): 0.9572

Q13. Please copy and paste your figure below (2 points)



Training Set and Test Set

Currently all of the dataset is used to train the model. To evaluate a neural network, we usually separate the dataset into two parts, with the majority of data set allocated as the **training set** and the rest being the **test set**. This ensures the test sets shares some commonality with the training set (e.g., samples from similar situation, guidelines, etc.) while ensuring that the test samples have never been used in the training.

To separate the dataset to training set and test set, we will change the *training_percentage* to 0.8, meaning 80% of the dataset is randomly selected as the training set and the remaining 20% make up the test set. Please comment out section 1 and uncomment section 2 and run the code. You will see the training set accuracy is 0.959 and the testing set accuracy is 0.932. Please note that if you hard code the total sample size to be 5000 anywhere in the code, you might not get the correct result. Please go back and fix it.

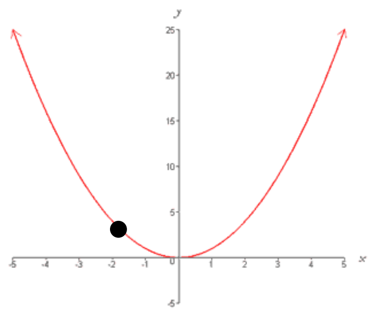
Q14. Why is the testing set accuracy lower than the training set accuracy? Is it by chance? (2 points)

The testing set accuracy is lower because the neural net hadn't seen the testing set during training. This input was novel—the net wasn't able to learn weights associated with this set. On the other hand, the net had developed its weights in accordance with the training set. It's not surprising that the net performed better at guessing at things it had already seen as opposed to things it had not yet seen.

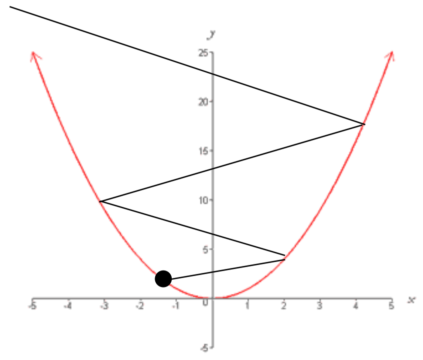
Learning Rate

Now let's play around with the learning rate and set it to a large number, 10. Theoretically, this neural net should learn very fast. To see whether this is true, please comment section 1 and 2, and uncomment section 3 and run the code. You will see that the cost stays at 3.250830, and the accuracy is 0.0934, which is at chance level. This is actually not surprising, and let's take a look at why.

In backpropagation, we attempt to find the weights that minimize the cost. Imagine for a moment that our cost function was a quadratic curve where the x-axis represents one of our weights and the y-axis represents the associated cost function for that particular weight:



Backpropagation uses a process called *gradient descent* to find the x-value that results in the smallest y value. Suppose the black dot is where we are, then gradient descent attempts to move down the slope (in the direction of the gradient) toward the nearest function minima. If our learning rate is small, we take small steps down toward the minima. If we slightly increase our steps, we will get to the minima faster. However, if the steps we take are too large, we might "overshoot" the minima and end up with a value that creates worse and worse cost values as the function "explodes":



This is what happens when we set the learning rate to the thoroughly unreasonable value of 10. However, different from the situation above, the cost we obtained stayed at 3.25 and didn't "explode" like the diagram shown above.

Q15. Why when we set the learning rate to 10 did the cost stay at 3.25 instead of increasing much more significantly? (2 points)

Because of the egregiously large learning rate, the network was locked into a local minimum where all outputs equal 0.1 but are in infinitesimal flux. Although it could have better minimized cost if the learning rate were reasonable, it found that it could not make a jump in any meaningful direction because the large disturbance would force cost to skyrocket, and after one such jump the gradient by this point had vanished significantly enough that its weights could not be made to escape the local minimum. In the convergence of all output values to 0.1, the neural net begins picking the same answer for every input, bringing about chance-level accuracy.

Number of Iterations and More layers

Now let's change the number of iterations to a large number, 30000, and train a neural network with 4 layers. Please comment out section 1, 2 and 3, and uncomment section 4 and run the code. It will take longer to complete compared to the previous few sections. Be patient! You will see that the training set accuracy has improved to 1.0. But the test set accuracy is worse than before (now 0.931). This is called overfitting.

Q16. Based on your reading, explain why overfitting happens. (2 points)

An overtrained neural net with too many parameters essentially builds a large lookup table for examples in the training set and learns how to perform the action of looking up these select examples very, very well. In doing so, however, it's also learning the characteristics associated with any given example e so well that it's able to more easily tell what is NOT e. Because many inputs will involve a good degree of noise—in this case, extra colored pixels—an overfitted net will therefore incorporate this noise into its weights. A loss of generalizability results, and this is undesirable because generalizability should be a property of any (good) learning schema (in general, for some example e you would want a highly similar example e' to be classified in the same way as e was classified).

Your own experiment with the hyperparameters

You may have more questions about how the hyperparameters influence the neural network, and now is the time for you to experiment with the hyperparameters on your own. Please comment out sections 1 to 4, and uncomment section 5. Please feel free to experiment with the *training_percentage, learning_rate, layers_dims* which defines the architecture of the neural network, and *num_iterations*. Please document your experiment below.

Q17. What is the purpose of your experiment, that is, what question are you trying to address? (2 points)

My experiment's purpose is to determine how the learning rate (LR) affects testing set accuracy. LR will be varied across three trials, and all other parameters will be held constant so as to isolate the effect of changing LR.

Q18. What are the values of the hyperparameters in your experiment? (2 points)

training_percentage	0.8	0.8	0.8
learning_rate	0.6	0.75	0.9
layers_dims	[400, 25, 10]	[400, 25, 10]	[400, 25, 10]
num_iterations	2000	2000	2000

Q19. Please describe the empirical results of your experiment. Only list empirical facts, not your conclusions or explanation… that comes next. (2 points)

In Trial 1, the training set and testing set accuracies were 0.96525 and 0.937, respectively. In Trial 2, these values were 0.973 and 0.94, respectively. In Trial 3, these values were 0.9795 and 0.936, respectively.

Q20. What conclusion do you draw from these results? Why do these results happen, and are there any limitations to your results? (2 point)

These results suggest a quasi-linear relationship between learning rate (LR) and training set accuracy and a parabolic relationship between LR and testing set accuracy. The first relationship makes sense because a higher LR should logically cause the neural net to learn the specified examples more efficiently. The second relationship also makes sense because, although a higher LR leads to more efficient learning, at a certain point it also leads to a loss of generality. This is because, at higher LR values, blame is propagated back almost too efficiently—the net becomes rigid in its classification and a similar problem to that described in Q16 results. These results absolutely do have limitations at the extrema of learning rate values. Obviously, negative learning rates would lead to bizarre results (perhaps an exploding gradient), and at some point between 0.9 and 10, accuracy bottoms out at chance.

Submission

We encourage to type in your answers directly on this handout. Please only put answers in the designated areas, as we will ignore anything outside those designated areas.

Also please do not alter the format of this file, otherwise we may not able to find your answer at the right place and you may lose points.

Please submit this file to Problem Set 6 on canvas.

And please submit your code to Problem Set 6 - Programming.