CMPSCI 390A Homework 5

YOUR NAME HERE

Assigned: Mar 4 2021; Due: Mar 16 2021 @ 11:00 am ET

Abstract

In this assignment we will finally create our very first neural network! To submit this assignment, upload a `.pdf` to Gradescope containing your responses to the written response questions below. You are required to use LaTeX for your write-up. When submitting your answers, use the template LaTeX code provided and put your answers below the question they are answering. Do not forget to put your name on the top of the .pdf. To submit the assignment's coding portion, upload a single python file called `my_hw5.py`, to the Gradescope programming assignment for Homework 5. An auto-grader will check your code for the correct output. As such, your program must meet the requirements specified below. We will also be using cheating detection software, so, as a reminder, you are allowed to discuss the homework with other students, but you must write the code on your own. Come to office

1 Install TQDM

For this assignment, we use the TQDM library that will automatically print out a progress bar for your loops. To add it to your conda environment run the following commands:

```
conda activate cs390a
conda install tqdm
```

To have your for loops automatically print out a progress bar you can use the `tqdm` function on any for loop. For example,

```
x = 0
for n in tqdm(range(num_iterations)):
    x += n
    print(x)
```

This code will output the print statements in for loop and show the progress bar like in Figure 1.

Figure 1: Example of `tqdm` output.

2 Network Initialization (10 points)

In previous assignments, we have initialized the weights to all be zero, but for our neural networks this will not be an effective initialization method. Spend some time to think about how initializing all the weights to zero will affect the gradients. There will be a question on it later. So instead, weights will be initialized by sampling from a Gaussian/normal distribution with zero mean and standard deviation corresponding to hyperparameter, `scale`. This initialization is described in more detail below, but you can look up *numpy.random.randn()* to see how to this can be implemented.

To create a "fully-connected", multilayer, feedforward network, each layer will need a 2D array to represent the weights of that layer. To keep track of all these weights, the weights in each layer will be labeled as W_0 to W_L for a network with L hidden layers (W_L are the weights in the output layer). An example can be seen in Figure 2. Each weight array will be stored in a dictionary, where the key for a weight array is the string of its label, e.g., the weights W_2 will have the label "W2". In

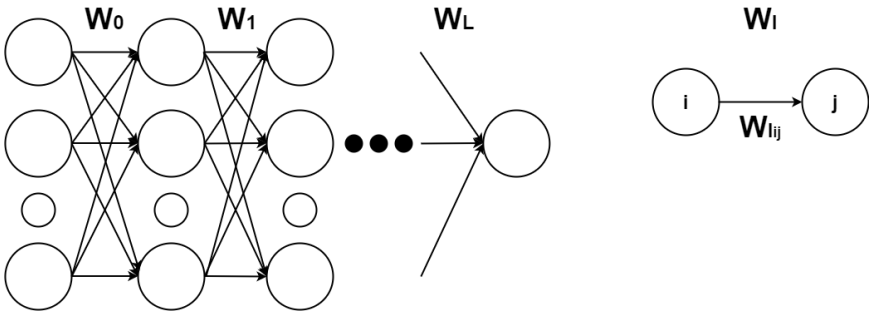

Figure 2: Example of a L layer neural network (left) and an example of the indexing of some particular weight in weight layer W_L (right) – this is lower-case l to denote an index between 0 and uppercase L (the number of layers). On the right the weight is labelled $W_{l_{ij}}$ because it is a weight in layer l that goes from the ith input to the jth output neuron

the code this is called the "init_params" and "param" dictionary (different names in different functions). This function will be checked by an autograder for correctness.

To initialize the weights of the network, you will complete the function `initialize_network()`, which takes two variables as input: `layer_sizes` and `scale` and returns a dictionary containing the weights of the network. The variable `layer_sizes` specifies the number of neurons in each layer and the first number in the list should correspond to the number of inputs to the network. The variable `scale` represents the scaling applied to randomly generated weights. For example, given layer_sizes = [100, 50, 1], you would need to create only two weight matrices – the first, W_0, would have the shape (100, 50). To set this in your dictionary using numpy, you might write:

```
init_params["W"+str(0)] = np.random.randn(100,50) * scale
```

3 Forward Pass (10 points)

Now that you have your parameters initialized, we can start to actually use your neural net (well, ""use""). Now, you'll need to complete the *forward()* method for your network. This method will take your dictionary of parameters and the 2D array, X, (that's the input) and will calculate a prediction. We **will not** be adding a constant term to our input at all for this homework.

Recall that the output of a layer of your network should act as the input to the next – that is, the output of the layer with weights W_0 will be the input of layer with weights W_1.

For our networks in this homework we will be using σ in every layer besides the first (the real input) and the last (the prediction).

We must also have our forward pass remember the intermittent values of our calculation so that we are able to perform backpropagation. Please save all your intermittent values (that is, in_j and a_j values – the weighted sums and outputs of each node) in a dictionary, *cache*. You must create keys for your dictionary of this form for autograding – "AL" where L is an int and where this key maps to the input to the L-th weight layer, and similarly "INL". That is, there should be two values in your cache per value in your param dictionary.

4 Loss and Backpropagation (20 points)

This is the bread and butter of your neural network code – you will be calculating the gradient of the loss using *backpropagation*. For this homework we will be using the Least Squares (LS) as the loss, which, as a reminder, is:

$$l(w) = \sum_{i=1}^{n} (y_i - f_w(x_i))^2.$$

To calculate the gradient of this loss with respect to the parameters, first calculate the partial derivative of the loss with respect to our final layer's weights, and then use the *chain rule* to propagate that loss backward and find the partial derivative of loss with respect to each of the layers' weights.

$$\nabla l(w) = \left[\frac{\partial l(w)}{\partial W_0}, \frac{\partial l(w)}{\partial W_1}, \ldots, \frac{\partial l(w)}{\partial W_L} \right].$$

Hint: you will use the values stored in your cache to calculate the gradient – each layer will have to use the output of the previous layer. Also here's an example of the chain rule to find the partial of W_l when you know the partial of IN_{l+1}:

$$\frac{\partial l(w)}{\partial W_l} = \frac{\partial l(w)}{\partial A_{l+1}} \cdot \frac{\partial A_{l+1}}{\partial \text{IN}_{l+1}} \cdot \frac{\partial \text{IN}_{l+1}}{\partial W_l}$$

$$= \frac{\partial l(w)}{\partial A_{l+1}} \cdot \frac{\partial \sigma(\text{IN}_{l+1})}{\partial \text{IN}_{l+1}} \cdot A_l$$

Implement the *backprop_and_loss()* method, which will take a prediction array, your network parameters, your *cache*, and your true labels Y, and will calculate the gradient and loss of your network.

Recall the following as well:

$$A_l = \sigma(\text{IN}_l)$$

and

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x))$$

and

$$\frac{\partial l(w)}{\partial A_l} = \frac{\partial l(w)}{\partial \text{IN}_{l+1}} \cdot W_l$$

Look in the course notes if you are having trouble with these calculations

5 Gradient Descent(10)

Now that all the parts are in order we can finally put them all together. Implement your *gradient_descent()* method which will perform gradient descent on you network. You may begin with your implementation from previous homeworks, but you will need to modify it to work on an arbitrary number of layers.

Once you have your gradient descent working you can run your code and you should get a graph that looks ***somewhat like*** figure 2. Because random initial weights are used the learning curves will be different on each run.

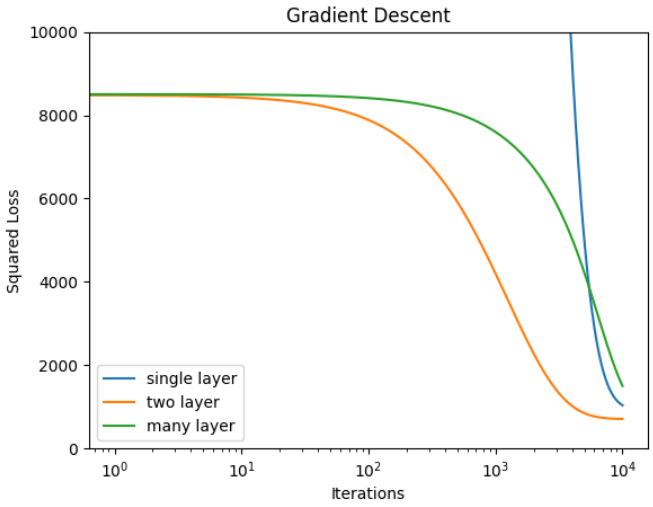

Figure 3: Example Learning curve plot with default hyperparameters.

6 Hyper Parameters and Best Model (10 points)

Now that we have *some* results, let's try to get *good* results. Try changing the scale, alpha, and number of iterations on each of the three provided models and see what performance you're able to get out of each. You should be able to get them all down to about 350 training loss.

You'll notice now that we are also finding the *testing* loss of each of your models – this is the measurement of how your models are performing on data that it has not trained on but which is from the same dataset (we call this the *testing* data). Take a look at which cases are doing well on that metric, particularly, is there a case when better training loss means worse testing loss?

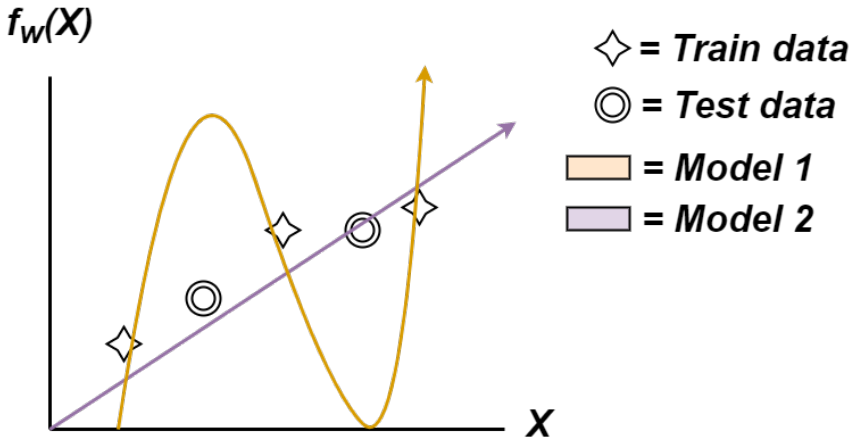

Figure 4: A model that is very well fit to training data (model 1) vs a model that is not as well fit to the training data but better fit to the training + test data (model 2)

To understand the need for a testing dataset take a look at Figure 4. Although our model may fit extremely well to the training data like model 1 – the second we try to use it on any data that we didn't have for training (like when it's deployed and using real world data) it fails *tremendously*. In this case we say that model 1 is **overfit** on the training data.

So now imagine we reserve some of the data from the training dataset such that our model can't be trained on it. Will an overfit model do well on our reserved data? No! So now we have a way to identify when our model is overfitting. Take note of the testing loss scores of your models while tuning the hyperparameters and see if you can see any overfitting.

Once you've worn yourself out playing with the hyperparameters, change the values in the *train_best_model()* method to values that you think would have very good performance on data that it isn't trained on (this new data would be the *testing* data). Be careful though! We are going to be testing your best model on a *different* test dataset than the one provided (but still taken from the same distribution of data). Your best model must average below a test loss of 700 for you to get the 10 points in this section.

7 Written Report (40 points)

1. (10 points) Include a plot of the losses of each of your 3 tuned models as well as your best model (you'll need to change code in main to plot this) – Describe your best model and list the values of its hyperparameters here.

2. (15 points)

   (a) Remove the forward and backward of your σ and make sure that your networks still run. Tune the hyperparameters until you get alright results (remember your best hyperparams with sigmoid though). How does the performance compare?

   (b) Now, try using a very low (close to zero) value for scale and a very high value (significantly larger than 1) for your multi-layer network. Print out the gradients of each layer during training (don't include this in the write up please). How do the gradients of the weights in different layers compare?

   (c) Write out the equation for the gradient without σ (it's simpler than with). Based on the gradient calculation, why is the behavior in part b expected?

   (d) What happens to the gradient with high and low values of scale, where there are sufficiently many layers? (5-10 layers should be enough here – what ever you can run)

   (e) Try adding back sigmoid and see if it still behaves the same way, do you see the same behavior?

3. (5 points) Propose some solution, using something you worked on in HW4 (and not using sigmoid), to the issue observed in question 2, explain your reasoning.

4. (5 points) Why is it important to use a different training and testing dataset? Think up a real world example where measuring your performance off of your training loss would cause serious issues (See section 6 if you're not sure).

5. (5 points) In choosing our best model, we could look at its performance on the given test dataset to tune the hyperparameters – Is this a good method to avoid the problem from question 4?

8 Extra Credit (15 points)

1. (5 points) Implement the method *extra_credit_fit()* which will create models and train them very similarly to our *main()* method, except it will do the normalization and basis expansions (on your input X) from the previous homework.

   Plot the training losses of each permutation of basis expansion, normalization and model size (one-layer, two-layer, many-layer) – there should be $2 \times 3 \times 3 = 18$ lines in total. Briefly describe any findings that you found interesting

2. (5 points) Using your *add_constant()* function from the previous homework, add a constant term to each layers' output (besides the final output) and change the size of the initialized weights such that they take an input one larger than normal. This will add a bias term to each of your layers. Plot the performance (and show it here) of your three models with and without the new bias terms, does the performance improve?

3. (5 points) Modify your network so that the activation function is $\sigma(x) = \max\{0, x\}$. This activation function is called a rectified linear unit (ReLU).

   Make sure that you work out how this new definition of σ impacts both the forwards and backwards passes. Note that the derivative of this activation function should be extremely simple, except at the point $x = 0$, where you should assume that $\partial\sigma(x)/\partial x = 0$.