CS240-Week-7: Neural networks

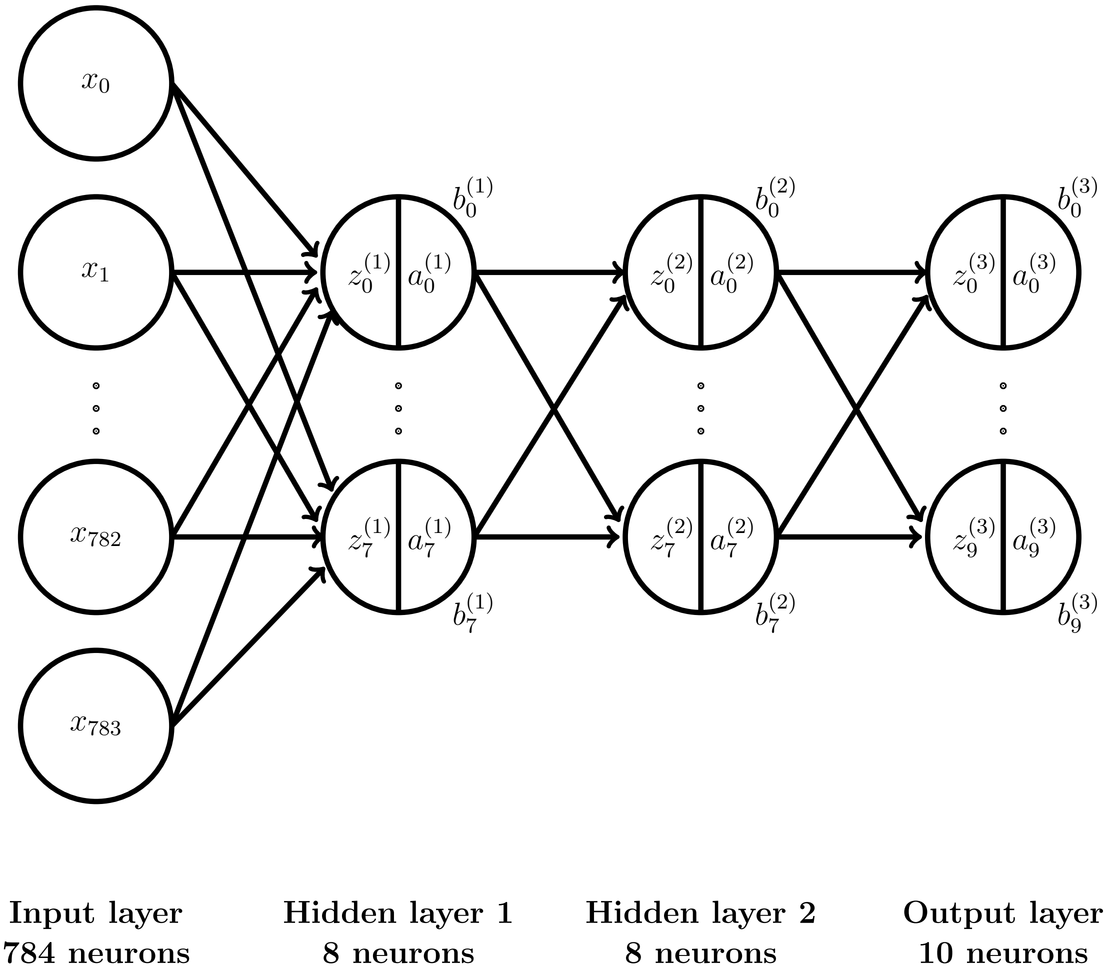
Instructions

- This lab will be **graded**. The weightage of each question is provided in this PDF.
- Please read the problem statement and the submission guidelines carefully.
- All code fragments need to be written within the `TODO` blocks in the given Python files. Do not change any other part of the code.
- **Do not** add any **additional** *print* statements to the final submission since the submission will be evaluated automatically.
- For any doubts or questions, please contact either the TA assigned to your lab group or one of the 2 TAs involved in making the lab.
- The deadline for this lab is **Monday, 4 March, 5 PM**.
- The submissions will be checked for plagiarism, and any form of cheating will be appropriately penalized.

The submissions will be on Gradescope. You need to upload the following Python files: `q1.py`. In Gradescope, you can directly submit these Python files using the upload option (you can either drag and drop or upload using browse). No need to create a tar or zip file.

1 Question 1: Neural networks for Classification [100 marks]

You are given a classification problem, where a given input image is to be classified into one of the 10 $\{0, 1, 2, \ldots, 9\}$ possible classes. Train a neural network from scratch using training dataset. The network has the following structure.



x_0
x_1
x_{782}
x_{783}
$z_0^{(1)}$ $a_0^{(1)}$ $b_0^{(1)}$
$z_7^{(1)}$ $a_7^{(1)}$ $b_7^{(1)}$
$z_0^{(2)}$ $a_0^{(2)}$ $b_0^{(2)}$
$z_7^{(2)}$ $a_7^{(2)}$ $b_7^{(2)}$
$z_0^{(3)}$ $a_0^{(3)}$ $b_0^{(3)}$
$z_9^{(3)}$ $a_9^{(3)}$ $b_9^{(3)}$
Input layer 784 neurons
Hidden layer 1 8 neurons
Hidden layer 2 8 neurons
Output layer 10 neurons

Let's denote:

- x as the input vector to the hidden layer 1.
- $W^{(1)}, W^{(2)}, W^{(3)}$ as the weight matrix of the hidden layer 1, hidden layer 2, and the output layer respectively, where each row corresponds to the weights associated with one neuron, and each column corresponds to the weights connecting to one input neuron.
- $b^{(1)}, b^{(2)}, b^{(3)}$ as the bias vectors of hidden layer 1, hidden layer 2, and the output layer, where each element represents the bias term associated with one neuron in the layer.
- $z^{(1)}, z^{(2)}, z^{(3)}$ as the output vector of hidden layer 1, hidden layer 2, and the output layer, and $a^{(1)}, a^{(2)}, a^{(3)}$ as the corresponding activations (activation function applied on the output vector z) of the respective layers. Note, the active function used is sigmoid.

The **forward pass** involves calculating output vectors and activations for all the nodes.

Forward Pass:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$a^{(1)} = \frac{1}{1 + \exp^{-z^{(1)}}}$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$

$$a^{(2)} = \frac{1}{1 + \exp^{-z^{(2)}}}$$

$$z^{(3)} = W^{(3)}a^{(2)} + b^{(3)}$$

$$a^{(3)} = \frac{1}{1 + \exp^{-z^{(3)}}}$$

- In the output layer, the node with the maximum value is chosen as the predicted class.
- The cost function that needs to be minimized is the sum of squared differences i.e., $C = \frac{1}{2}\sum_{j=0}^{j=9}(a_j^{(3)} - y_j)^2$, where $a_j^{(3)}$ are the activations at the output layer.
- The derivative of the costs w.r.t $a^{(3)}$ is $(a^{(3)} - y)$ in vector form.
- Use mini batch gradient descent to update the weights and biases.

Backward pass involves calculating layer by layer gradients of the cost function w.r.t all weights and biases. This will be used to update weights and biases in the gradient descent update step. **Note, for implementation do the below calculations using numpy arrays in vector form itself.**

Backward pass:

- Gradients output layer:

$$\frac{dC}{dW_{ij}^{(3)}} = \frac{dC}{da_i^{(3)}}.\frac{da_i^{(3)}}{dz_i^{(3)}}.\frac{dz_i^{(3)}}{dW_{ij}^{(3)}}$$

$$\frac{dC}{db_i^{(3)}} = \frac{dC}{da_i^{(3)}}.\frac{da_i^{(3)}}{dz_i^{(3)}}.\frac{dz_i^{(3)}}{db_i^{(3)}}$$

where,

$$\frac{dC}{da_i^{(3)}} = a_i^{(3)} - y_i \quad \text{(implement and use cost\_derivative function for this)}$$

$$\frac{da_i^{(3)}}{dz_i^{(3)}} = \text{sigmoid}(z_i^{(3)}) * (1 - \text{sigmoid}(z_i^{(3)})) \quad \text{(implement and use sigmoid\_derivative function for this)}$$

$$\frac{dz_i^{(3)}}{dW_{ij}^{(3)}} = a_j^{(2)}$$

$$\frac{dz_i^{(3)}}{db_i^{(3)}} = 1$$

- Hidden layer 2:

$$\frac{dC}{dW_{ij}^{(2)}} = \frac{dC}{da_i^{(2)}}.\frac{da_i^{(2)}}{dz_i^{(2)}}.\frac{dz_i^{(2)}}{dW_{ij}^{(2)}}$$

$$\frac{dC}{db_i^{(2)}} = \frac{dC}{da_i^{(2)}}.\frac{da_i^{(2)}}{dz_i^{(2)}}.\frac{dz_i^{(2)}}{db_i^{(2)}}$$

where,

$$\frac{dC}{da_i^{(2)}} = \sum_{k=0}^{k=9}\frac{dC}{da_k^{(3)}}.\frac{da_k^{(3)}}{dz_k^{(3)}}.\frac{dz_k^{(3)}}{da_i^{(2)}}, \qquad \text{note, } \frac{dz_k^{(3)}}{da_i^{(2)}} = W_{ki}^{(3)}$$

$$\frac{da_i^{(2)}}{dz_i^{(2)}} = \text{sigmoid}(z_i^{(2)}) * (1 - \text{sigmoid}(z_i^{(2)})) \quad \text{(implement and use sigmoid\_derivative function)}$$

$$\frac{dz_i^{(2)}}{dW_{ij}^{(2)}} = a_j^{(1)}$$

$$\frac{dz_i^{(2)}}{db_i^{(2)}} = 1$$

- The gradient for the other hidden layer can be calculated similarly.

Note that the above backward pass is for a single data point. We'll need to calculate this for all examples in a mini batch and then sum the gradients over this batch to do the gradient update step for all weights and biases.

1.1 Code

Your task is to train the neural network using mini-batch gradient descent. Implement the following functions. **Note, the return values of the function should be in the same format as mentioned in the code/doc strings. Additionally, keep your implementations generic such that your code works even when the structure (number of layers, number of neurons per layer) of the neural network changes.**

- Function **sigmoid**: Implement the activation fucntion. Use this function in forward pass and backward pass. [**5 marks**]
- Function **sigmoid_derivative**: Calculate the derivative of the sigmoid function. Use this function in backward pass. [**5 marks**]
- Function **cost**: Calculate cost (sum of squared differences) over training data for the current weights and biases. [**5 marks**]
- Function **cost_derivative**: Derivative of the cost function w.r.t activations at the output layer. Use this function in backward pass. [**5 marks**]
- Function **forward_pass**: Perform forward pass and return the output of the NN $a^{(3)}$. [**10 marks**]
- Function **mini_batch_GD**: Train the neural network using mini batch gradient descent. [**10 marks**]
  For every epoch the following needs to be done.
  (1) Shuffle the training data set, and create mini-batches of size *mini_batch_size*. Note, this has been done for you.
  (2) For every mini-batch:
  - Compute gradients for every data point in the mini batch using your implemented "back_propagation" function. Note that the above is for a single data point. We'll need to calculate this for all examples in a mini batch and then sum the gradients over this batch to do the gradient update step for all weights and biases.
  - Update biases and weights using the computed gradients

  $$W_{ij} = W_{ij} - (\eta/\text{mini\_batch\_size}) \sum_{b \in B} \frac{dC}{dW_{ij}}$$

  B : minibatch of examples, $\quad b$: an example in B

  After every epoch, the accuracy on test data and the cost for the current weights and biases will be displayed. Note, for testing this function, we are keep track of the cost history. The cost history returned by the function will be matched which will account for weight/bias updates in the gradient descent update step.

- Function **back_propagation**: Finish back propagation for one data point to compute gradients of the cost function w.r.t all weights and biases. [**25 (gradient biases) + 35 (gradient weights) = 60 marks**]
  The following steps can be followed.
  (1) forward pass - calculate layer by layer z's and activations which will be used to calculate gradients
  (2) backward pass - calculate gradients starting with the output layer and then the hidden layers
  (3) Return the gradients in lists dC_db, dC_dw.
  - Input: (x – (784, 1) numpy array, y – (10, 1) numpy array)
  - Output: dC_dw – list, dC_db – list
    These lists contain layer-by-layer gradients as numpy arrays. For example, if we have an input layer with 784 neurons, two hidden layer with 8 neurons, and an output layer with 10 neurons:

$$dC\_db = [\text{numpy array } (8,1),\ \text{numpy array } (8,1),\ \text{numpy array } (10,1)]$$

$$dC\_dw = [\text{numpy array } (784,8),\ \text{numpy array } (8,8),\ \text{numpy array } (8,10)]$$

Additional information on input/output for functions can be found in code/doc-strings.

1.2 Hyperparameter Tuning ***(Ungraded)***

You were tasked with building a neural network model using the provided *NeuralNetwork* class. Your goal now is to achieve the **highest possible accuracy** on a held-out test set through hyperparameter tuning.

Given the *mini_batch_GD* function and the *NeuralNetwork class* provided, you have the flexibility to adjust several hyperparameters to optimize the performance of your model.
The key hyperparameters available for tuning are:

- **Learning rate (Eta)**: The rate at which the model adjusts its weights during training.
- **Mini-batch size**: The number of training examples utilized in each iteration of gradient descent.
- **Number of layers in the neural network (num_layers)**: The depth of the network architecture.
- **Number of neurons per layer (num_neurons_per_layer_list)**: The width of each layer in the network.

Your task is to experiment with different values for these hyperparameters and find the combination that maximizes the accuracy of your model on the test data.

Performance Targets:

- Accuracy around 82%: Acceptable
- Accuracy greater than 87%: Good
- Accuracy greater than 90%: Very Good
- Accuracy greater than 92%: Excellent