Programming Assignment 2

Neural Networks (100 points)

Instructions

- Deadline for the assignment is July 25 2021 at 23:59:59 PDT.
- Do not import other libraries. You are only allowed to use Math, Numpy packages which are already imported in the file. DO NOT use scipy functions.
- Please use Python 3.5 or 3.6 (for full support of typing annotations). You have to make the functions' return values match the required type.
- In this programming assignment you will implement **Neural Networks**. We have provided the bootstrap code and you are expected to complete the **classes** and **functions**.
- Download all files of PA2 from Vocareum and save in the same folder.
- Only modifications in files (`neural_networks.py`) will be accepted and graded. Submit (`neural_networks.py`) on Vocareum once you are finished. Please **delete** unnecessary files before you submit your work on Vocareum.
- DO NOT CHANGE THE OUTPUT FORMAT. DO NOT MODIFY THE CODE UNLESS WE INSTRUCT YOU TO DO SO. A homework solution that mismatches the provided setup, such as format, name initializations, etc., will not be graded. It is your responsibility to make sure that your code runs well on Vocareum.

Office Hours:

```
Tuesday 5:00 PM to 7:00 PM [Chiransh] | Meeting Link: https://usc.zoom.us/j/92904525689
Wednesday 3:00 PM to 5:00 PM [Ge] | Meeting Link: https://usc.zoom.us/j/99290427209
Thursday 5:00 AM to 7:00 PM [Iris] | Meeting Link: https://usc.zoom.us/j/99957886019
```

Before You start

On Vocareum, when you submit your homework, it takes around 5-6 minutes to run the grading scripts and evaluate your code. So, please be patient regarding the same.

Problem 1 Neural Networks(40 Points)

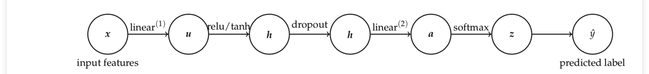

Figure 1: A diagram of a multi-layer perceptron (MLP). *The edges mean mathematical operations (modules), and the circles mean variables*. The term *relu* stands for rectified linear units.

For this Assignment, you are asked to implement neural networks. We will be using this neural network to classify MNIST database of handwritten digits (0-9). The architecture of the multi-layer perceptron (MLP, just another term for fully connected feedforward networks we discussed in the lecture) you will be implementing is shown in figure 1. Following MLP is designed for a K-class classification problem.

Let $(x \in \mathbb{R}^D, y \in \{1, 2, \cdots, K\})$ be a labeled instance, such an MLP performs the following computations.

$$
\begin{aligned}
\textbf{input features}: \quad & x \in \mathbb{R}^D \\
\textbf{linear}^{(1)}: \quad & u = W^{(1)}x + b^{(1)} \qquad , W^{(1)} \in \mathbb{R}^{M \times D} \text{ and } b^{(1)} \in \mathbb{R}^M \\
\textbf{tanh}: \quad & h = \frac{2}{1 + e^{-2u}} - 1 \\
\textbf{relu}: \quad & h = max\{0, u\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix} \\
\textbf{linear}^{(2)}: \quad & a = W^{(2)}h + b^{(2)} \qquad , W^{(2)} \in \mathbb{R}^{K \times M} \text{ and } b^{(2)} \in \mathbb{R}^K \\
\textbf{softmax}: \quad & z = \begin{bmatrix} \frac{e^{a1}}{\sum_k e^{ak}} \\ \vdots \\ \frac{e^{aK}}{\sum_k e^{ak}} \end{bmatrix} \\
\textbf{predicted label}: \quad & \hat{y} = argmax_k z_k .
\end{aligned}
$$

For a K-class classification problem, one popular loss function for training (i.e., to learn $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$) is the cross-entropy loss.Specifically we denote the cross-entropy loss with respect to the training example (x, y) by l:

$$l = -\log(z_y) = \log\left(1 + \sum_{k \neq y} e^{a_k - a_y}\right)$$

Note that one should look at l as a function of the parameters of the network, that is, $W^{(1)}, b^{(1)}, W^{(2)}$ and $b^{(2)}$. For ease of notation, let us define the one-hot (i.e., 1-of-K) encoding of a class y as

$$y \in \mathbb{R}^K \text{ and } y_k = \begin{cases} 1, \text{ if } y = k, \\ 0, \text{ otherwise.} \end{cases}$$

so that

$$l = -\sum_k y_k \log z_k = -y^T \begin{bmatrix} \log z_1 \\ \vdots \\ \log z_K \end{bmatrix} = -y^T \log z.$$

We can then perform error-backpropagation, a way to compute partial derivatives (or gradients) w.r.t the parameters of a neural network, and use gradient-based optimization to learn the parameters.

Submission: All you need to submit is neural_networks.py.

Q1.1 Mini batch Gradient Descent

First, you need to implement mini-batch gradient descent which is a gradient-based optimization to learn the parameters of the neural network. You need to realize two alternatives for SGD, one without momentum and one with momentum. We will pass a variable α to indicate which option. When $\alpha \leq 0$, the parameters are updated just by gradient. When $\alpha > 0$, the parameters are updated with momentum.

$$v = \alpha v - \eta \delta_i$$

$$w_j = w_{j-1} + v$$

You can use the formula above to update the weights using momentum.

Here, α is the discount factor such that $\alpha \in (0, 1)$

- v is the velocity update
- η is the learning rate
- δ_i is the gradient

You need to handle with as well without momentum scenario in the `miniBatchGradientDescent` function.

- `TODO 1` You need to complete `def miniBatchGradientDescent(model, momentum, _lambda, _alpha, _learning_rate)` in `neural_networks.py`

Q1.2 Linear Layer (10 Points)

Second, You need to implement the linear layer of MLP. In this part, you need to implement 3 python functions in `class linear_layer`. In `def __init__(self, input_D, output_D)` function, you need to initialize W with random values using np.random.normal such that the mean is 0 and standard deviation is 0.1. You also need to initialize gradients to zeroes in the same function

$$\textbf{forward pass:} \qquad u = \text{linear}^{(1)}.\text{forward}(x) = W^{(1)}x + b^{(1)}, \\ \text{where } W^{(1)} \text{ and } b^{(1)} \text{ are its parameters.}$$

$$\textbf{backward pass:} \qquad [\frac{\partial l}{\partial x}, \frac{\partial l}{\partial W^{(1)}}, \frac{\partial l}{\partial b^{(1)}}] = \text{linear}^{(1)}.\text{backward}(x, \frac{\partial l}{\partial u}).$$

You can use the above formula as a reference to implement the `def forward(self, X)` forward pass and `def backward(self, X, grad)` backward pass in `class linear_layer`. In backward pass, you only need to return the backward_output. You also need to compute gradients of W and b in backward pass

- `TODO 2` You need to complete `def __init__(self, input_D, output_D)` in `class linear_layer` of `neural_networks.py`
- `TODO 3` You need to complete `def forward(self, X)` in `class linear_layer` of `neural_networks.py`
- `TODO 4` You need to complete `def backward(self, X, grad)` in `class linear_layer` of `neural_networks.py`

Activation Function-tanh(10 Points)

Now, you need to implement the activation function tanh. In this part, you need to implement 2 python functions in `class tanh`. In `def forward(self, X)` you need to implement the forward pass and you need to compute the derivative and accordingly implement `def backward(self, X, grad)`. i.e. the backward pass.

$$\textbf{tanh}: \qquad h = \frac{2}{1 + e^{-2u}} - 1$$

You can use the above formula for tanh as a reference.

- `TODO 5` You need to complete `def forward(self, X)` in `class tanh` of `neural_networks.py`
- `TODO 6` You need to complete `def backward(self, X, grad)` in `class tanh` of `neural_networks.py`

Activation Function-relu(10 Points)

You need to implement another activation function called relu. In this part, you need to implement 2 python functions in `class relu`. In `def forward(self, X)` you need to implement the forward pass and you need to compute the derivative and accordingly implement `def backward(self, X, grad)`. i.e. the backward pass.

$$\textbf{relu}: \qquad h = max\{0, u\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix}$$

You can use the above formula for relu as a reference.

- `TODO 7` You need to complete `def forward(self, X)` in `class relu` of `neural_networks.py`
- `TODO 8` You need to complete `def backward(self, X, grad)` in `class relu` of `neural_networks.py`

Q1.5 Dropout(10 Points)

To prevent overfitting, we usually add regularization. Dropout is another way of handling overfitting. In this part, you will initially read and understand `def forward(self, X, is_train)` i.e. the forward pass of `class dropout` and derive partial derivatives accordingly to implement `def backward(self, X, grad)` i.e. the backward pass of `class dropout`. We define the forward and the backward passes as follows.

$$\text{forward pass:} \qquad s = \text{dropout.forward}(q \in \mathbb{R}^J) = \frac{1}{1 - r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times q_1 \\ \vdots \\ \mathbf{1}[p_J >= r] \times q_J \end{bmatrix},$$

$$\text{where } p_j \text{ is sampled uniformly from } [0, 1), \forall j \in \{1, \cdots, J\}, \\ \text{and } r \in [0, 1) \text{ is a pre-defined scalar named dropout rate.}$$

$$\text{backward pass:} \qquad \frac{\partial l}{\partial q} = \text{dropout.backward}(q, \frac{\partial l}{\partial s}) = \frac{1}{1 - r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times \frac{\partial l}{\partial s_1} \\ \vdots \\ \mathbf{1}[p_J >= r] \times \frac{\partial l}{\partial s_J} \end{bmatrix}.$$

Note that $p_j, j \in \{1, \cdots, J\}$ and r are not be learned so we do not need to compute the derivatives w.r.t. to them. Moreover, $p_j, j \in \{1, \cdots, J\}$ are re-sampled every forward pass, and are kept for the following backward pass. The dropout rate r is set to 0 during testing.

- `TODO 9` You need to complete `def backward(self, X, grad)` in `class dropout` of `neural_networks.py`

Q1.6 Connecting the Dots

In this part, you will combine the modules written from question Q1.1 to Q1.5 by implementing TODO snippets in the `def main(main_params, optimization_type="minibatch_sgd")` i.e. main function. After implementing forward and backward passes of MLP layers in Q1.1 to Q1.5,now in the main function you will call the forward methods and backward methods of every layer in the model in an appropriate order based on the architecture.

- `TODO 10` You need to complete `main(main_params, optimization_type="minibatch_sgd")` in `neural_networks.py`

Grading

Your code will be graded on Vocareum with autograding script. For your reference, the solution code takes around 5 minutes to execute. As long as your code can finish grading on Vocareum, you should be good. When you finish all TODO parts, please click submit button on Vocareum. Sometimes you may need to come back to check grading report later.

Your code will be tested on the correctness of modules you have implemented as well as certain custom testcases. 40 points are assigned for Question 1 while 60 points are assigned to custom testcases.

Grading Guideline for Neural Networks (100 points)

Question 1

1. Linear Layer: 10 points
2. Activation Function-tanh: 10 points
3. Activation Function-relu: 10 points
4. Dropout:10 points

Question 2

1. Custom testcases: 60 points
2. Make sure to complete the miniBatchGradientDescent and main functions for the custom cases to run. You will not receive any points for the custom test cases if these functions are not complete.

In [ ]: