Part 1 coding

For this assignment, you are asked to implement neural networks. You will use this neural network to classify MNIST database of handwritten digits (0-9). The architecture of the neural network you will implement is based on the multi-layer perceptron (MLP, just another term for fully connected feedforward networks), which is shown as following. It is designed for a K-class classification problem.

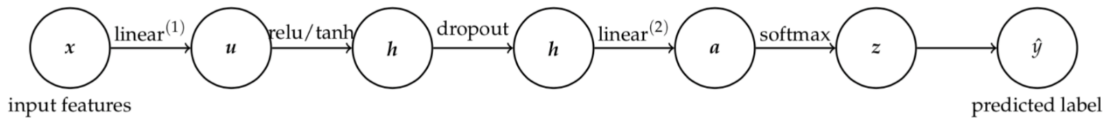

Figure 1: A diagram of a multi-layer perceptron (MLP). *The edges mean mathematical operations (modules), and the circles mean variables.* The term *relu* stands for rectified linear units.

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

$$
\begin{aligned}
\textbf{input features}: &\quad x \in \mathbb{R}^D \\
\textbf{linear}^{(1)}: &\quad u = W^{(1)}x + b^{(1)} \qquad , W^{(1)} \in \mathbb{R}^{M \times D} \text{ and } b^{(1)} \in \mathbb{R}^M \\
\textbf{tanh}: &\quad h = \frac{2}{1 + e^{-2u}} - 1 \\
\textbf{relu}: &\quad h = max\{0, u\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix} \\
\textbf{linear}^{(2)}: &\quad a = W^{(2)}h + b^{(2)} \qquad , W^{(2)} \in \mathbb{R}^{K \times M} \text{ and } b^{(2)} \in \mathbb{R}^K \\
\textbf{softmax}: &\quad z = \begin{bmatrix} \frac{e^{a_1}}{\sum_k e^{a_k}} \\ \vdots \\ \frac{e^{a_K}}{\sum_k e^{a_k}} \end{bmatrix} \\
\textbf{predicted label}: &\quad \hat{y} = \text{argmax}_k z_k.
\end{aligned}
$$

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.

1 Mini batch Stochastic Gradient Descent

First, you need to implement mini-batch stochastic 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 and α will also represents the discount factor as following:

$$
\begin{aligned}
v &= \alpha v - \eta \delta_t \\
w_t &= w_{t-1} + v
\end{aligned}
$$

You can use the formula above to update the weights.
Here, α is the discount factor such that $\alpha \in (0, 1)$. It is given by us and you do not need to adjust it.
η is the learning rate. It is also given by us.
v is the velocity update (A.K.A momentum update). δ_t is the gradient

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

2 Linear Layer

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 the function `def __init__(self, input_D, output_D)`, 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.

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

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

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`

3 Activation function – tanh

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.

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

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`

4 Activation function – relu

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.

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

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`

5 Dropout

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`. You will also derive partial derivatives accordingly to implement `def backward(self, X, grad)` i.e. the backward pass of `class dropout`.
Now we take an intermediate variable $q \in \mathbb{R}^J$ which is the output from one of the layers. Then we define the forward and the backward passes in dropout as follows.
The forward pass obtains the output after dropout.

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

$$\text{where } p_j \text{ is generated randomly from } [0, 1), \forall j \in \{1, \cdots, J\}, \\ \text{and } r \in [0, 1) \text{ is a pre-defined scalar named dropout rate which is given to you.}$$

The backward pass computes the partial derivative of loss with respect to q from the one with respect to the forward pass result, which is $\frac{\partial l}{\partial s}$.

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

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. You do not need to find the best r since we have picked it for you. Moreover, $p_j, j \in \{1, \cdots, J\}$ are re-sampled every forward pass, and are kept for the corresponding backward pass.

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

6 Main

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

Part 2

Suppose we have a Multi-Class Neural Networks defined below. An illustration is provided in Fig. 2. Please answer the following questions.

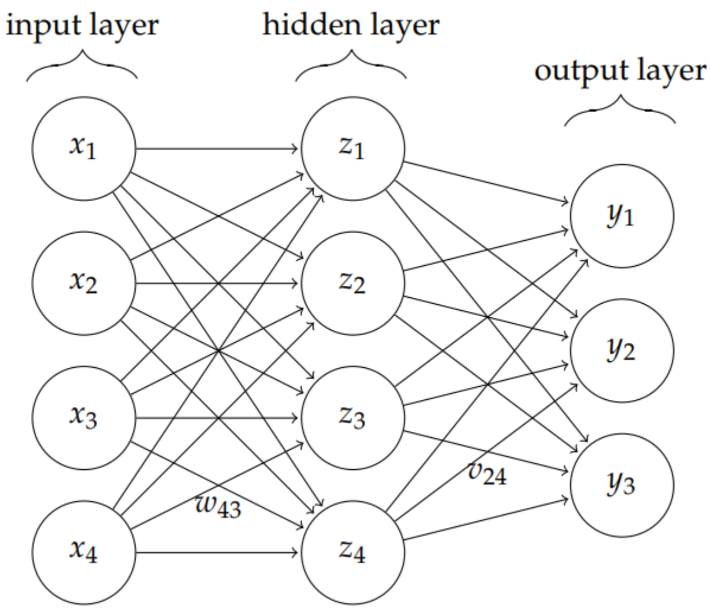

Figure 2: A neural network with one hidden layer.

Forward Propagation. For multi-class classification, we use softmax layer with cross-entropy loss as output. In the hidden layer, we use *tanh* activation function. The forward propagation can be expressed as:

input layer	x_i,
hidden layer	$z_k = tanh\left(\sum_{i=1}^{4} w_{ki}x_i\right), tanh(\alpha) = \frac{e^{\alpha} - e^{-\alpha}}{e^{\alpha} + e^{-\alpha}}$
output layer	$\hat{y}_j = softmax(o_j) = \frac{\exp(o_j)}{\sum_{i=1}^{3}\exp(o_i)}$, where $o_j = \sum_{k=1}^{4} v_{jk}z_k$
loss function	$L(y, \hat{y}) = -\sum_{j=1}^{3} y_j \log \hat{y}_j$, where $\hat{y}_j$ is prediction, y_j is ground truth

Backpropagation Please write down $\frac{\partial L}{\partial v_{jk}}$ and $\frac{\partial L}{\partial w_{ki}}$ in terms of only x_i, z_k, o_j, y_j, and/or $\hat{y}_j$ using backpropagation.