Implemented neural network to classify MNIST database of handwritten digits (0-9). The architecture of the neural network that I implemented is based on the multi-layer perceptron. It is designed for a K-class classification problem.

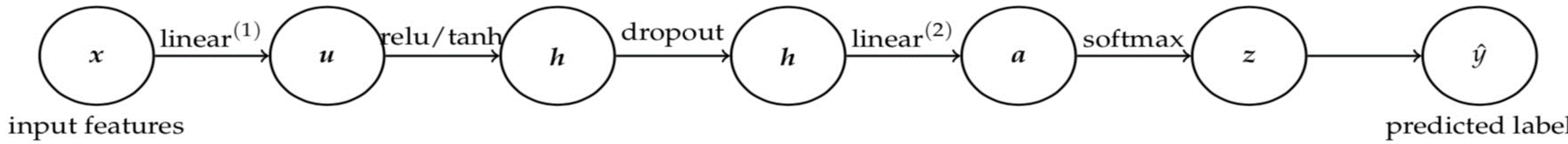

Figure 1: A diagram of a multi-layer perceptron (MLP). *The edges mean mathematical operations (modules), and the circles mean variables.* The term *relu* stands for rectified linear units.

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,

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

I have implemented mini-batch stochastic gradient descent which is a gradient-based optimization to learn the parameters of the neural network. I used two alternatives for SGD, one without momentum and one with momentum.

To prevent overfitting, we usually add regularization. Here, I have made use of Dropout which is another way of handling overfitting.

The forward pass obtains the output after dropout.

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

where p_j is generated randomly from $[0, 1), \forall j \in \{1, \cdots, J\}$,
and $r \in [0, 1)$ 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 $\frac{\partial l}{\partial s}$.

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

Some default values that I used are:

Learning rate: 0.01
Mini-batch size: 5
# of epochs: 10
Step-size: 10

Hidden layer size: 1000
Output layer size: 10