UNIVERSITY OF MINNESOTA, TWIN CITIES
COLLEGE OF SCIENCE AND ENGINEERING

Assignment #5

Student name: *Hardik Gupta*

Course: *Computer Vision (CSCI 5561)* – Professor: *Dr. Volkan Isler*
Due date: *December 4th, 2023*

Multi Layer Perceptron

```
def get_mini_batch(im_train, label_train, batch_size)
    ...
    return mini_batch_x, mini_batch_y
```

Input: im_train $\in \mathbb{R}^{196\times N}$ and label_train $\in R_1 \times N$ are a set of vectorized images and corresponding labels, and batch_size is the size of the mini-batch for stochastic gradient descent.
Output: mini_batch_x and mini_batch_y are python lists that contain a set of batches (images and labels, respectively). Each batch of images is a matrix with size 196$\times$ batch_size, and each batch of labels is a matrix with size 10$\times$ batch_size (one-hot encoding). Note that the number of images in the last batch may be smaller than batch_size.

```
def fc(x, w, b)
    ...
    return y
```

Input: x $\in \mathbb{R}^{m\times 1}$ is the input to the fully connected layer, and w $\in \mathbb{R}^{n\times m}$ and b $\in \mathbb{R}^{n\times 1}$ are the weights and bias.
Output: y $\in \mathbb{R}^{n\times 1}$ is the output of the linear transform (fully connected layer).

```
def fc_backward(dl_dy, x, w, b)
  ...
return dl_dx, dl_dw, dl_db
```

Input: dl_dy $\in \mathbb{R}^{n\times 1}$ is the loss derivative with respect to the output y of fully connected layer.
Output: dl_dx $\in \mathbb{R}^{m\times 1}$ is the loss derivative with respect the input x, dl_dw $\in \mathbb{R}^{n\times m}$ is the loss derivative with respect to the weights, and dl_db $\in \mathbb{R}^{n\times 1}$ is the loss derivative with respect to the bias.

```
def relu(x)
    ...
    return y
```

Input: x is a general tensor, matrix, or vector.
Output: y is the output of the Rectified Linear Unit (ReLu) with the same shape as input.

```
def relu_backward(dl_dy, x)
    ...
    return dl_dx
```

Input: dl_dy is the loss derivative with respect to the output y of ReLu layer. It has the same shape as y (it can be a tensor, matrix, or vector).
Output: dl_dx is the loss derivative with respect to the input x. It has the same shape as dl_dy.

```
def loss_cross_entropy_softmax(x, y)
    ...
    return l, dl_dx
```

Input: x $\in \mathbb{R}^{m\times 1}$ is the input to the softmax, and y $\in 0, 1^{m\times 1}$ is the ground truth label.
Output: l $\in$ R is the loss, and dl_dx$\in \mathbb{R}^{m\times 1}$ is the loss derivative with respect to x.

Solution. **get_mini_batch** iterates through the batches, creating mini-batches of input features (mini_batch_x) and their corresponding one-hot encoded labels (mini_batch_y). The one-hot encoding is applied to the labels, converting them into a binary matrix representation. Overall, this function sets up the data in a randomized and batched format, ready for training.

fc linear transform of x, i.e., $y = wx + b$

fc_backward provides mathematical calculation for each:

1. $\frac{dl}{dx} = \frac{dl}{dy} \cdot \frac{dy}{dx}$; where $\frac{dy}{dx} = w$

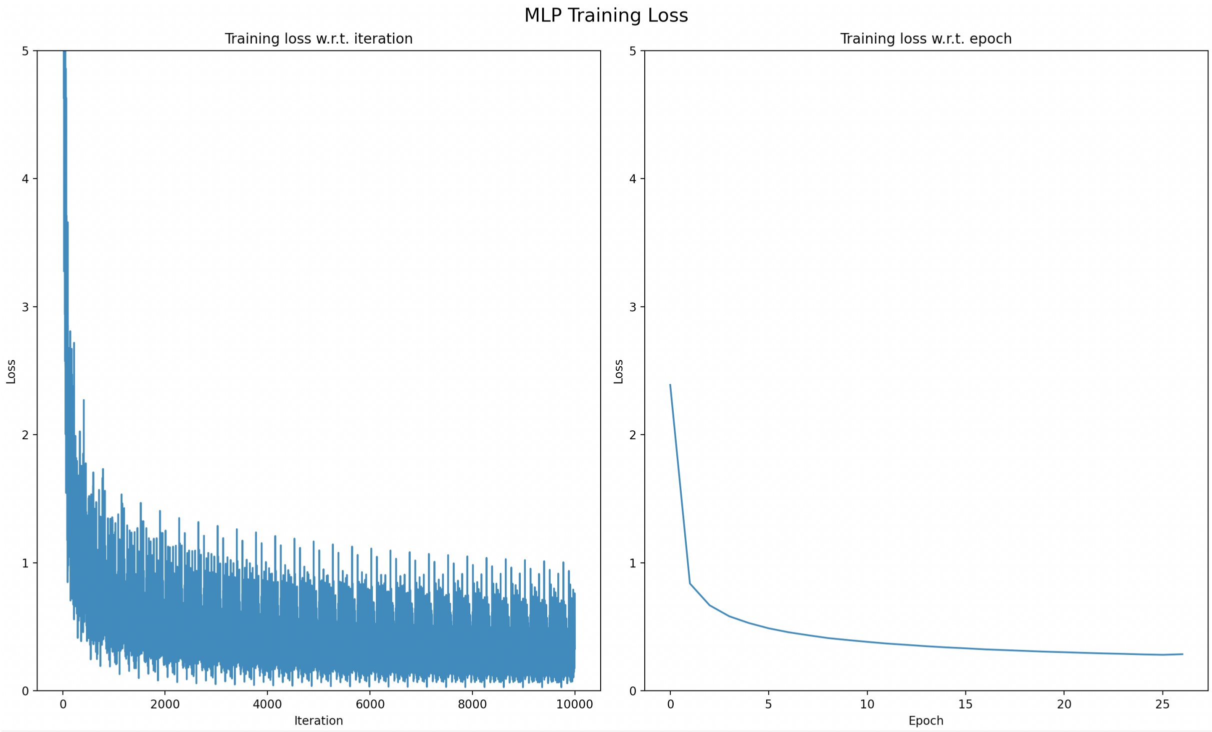
2. $\frac{dl}{dw} = \frac{dl}{dy} \cdot \frac{dy}{dw}$; where $\frac{dy}{dw} = x$

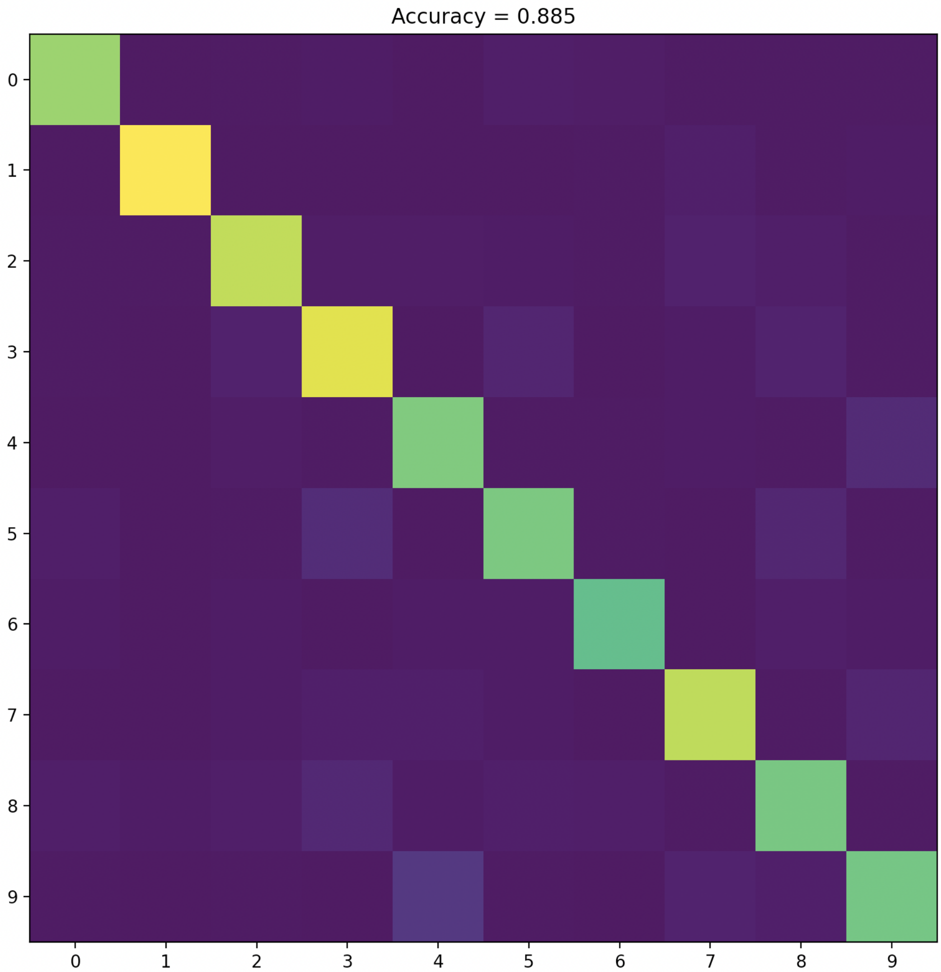
3. $\frac{dl}{db} = \frac{dl}{dy} \cdot \frac{dy}{db}$; where $\frac{dy}{db} = 1$

relu provides the maximum of (0, x)

relu_backward provides the $\frac{dy}{dx}$; where $y = ReLU$. $\frac{dl}{dx} = \frac{dl}{dy}\frac{dy}{dx}$;
$\frac{dy}{dx} = 1$ where $x > 0$; else 0

loss_cross_entropy is calculated using the ground truth label y and the softmax output y_hat. It measures the dissimilarity between the predicted distribution and the true distribution. The *softmax* function is applied element-wise to the input vector x. It exponentiates each element, subtracts the maximum value to improve numerical stability, and then normalizes the result to obtain a probability distribution. The derivative of the loss with respect to the input x is computed as the difference between the softmax output y_hat and the ground truth label y.


MLP Training Loss
Training loss w.r.t. iteration
Training loss w.r.t. epoch
Loss
Iteration
Epoch
0
1
2
3
4
5
2000
4000
6000
8000
10000
10
15
20
25


Accuracy = 0.885
0
1
2
3
4
5
6
7
8
9

Convolutional Neural Network

```
def conv(x, w_conv, b_conv)
    ...
    return y
```

Input: x $\in \mathbb{R}^{H\times W\times C_1}$ is an input to the convolutional operation, w_conv $\in \mathbb{R}^{h\times w\times C_1\times C_2}$ and b_conv $\in \mathbb{R}^{C_2\times 1}$ are weights and bias of the convolutional operation.
Output: y $\in \mathbb{R}^{H\times W\times C_2}$ is the output of the convolutional operation. Note that to get the same size with the input, you may pad zero at the boundary of the input image.

```
def conv_backward(dl_dy, x, w_conv, b_conv)
    ...
    return dl_dw, dl_db
```

Input: dl_dy $\in \mathbb{R}^{H\times W\times C_2}$ is the loss derivative with respect to the output of the convolutional layer. The rest inputs are the same as defined in function *conv*.
Output: dl_dw $\in \mathbb{R}^{h\times w\times C_1\times C_2}$ and dl_db $\in \mathbb{R}^{C2\times 1}$ are the loss derivatives with respect to convolutional weights w_conv and bias b_conv, respectively.

```
def pool2x2(x)
    ...
    return y
```

Input: x $\in \mathbb{R}^{H\times W\times C}$ is a general tensor or matrix.
Output: y $\in \mathbb{R}^{\frac{H}{2}\times\frac{W}{2}\times C}$ is the output of the 2×2 max-pooling operation with stride 2.

```
def pool2x2_backward(dl_dy, x)
    ...
    return dl_dx
```

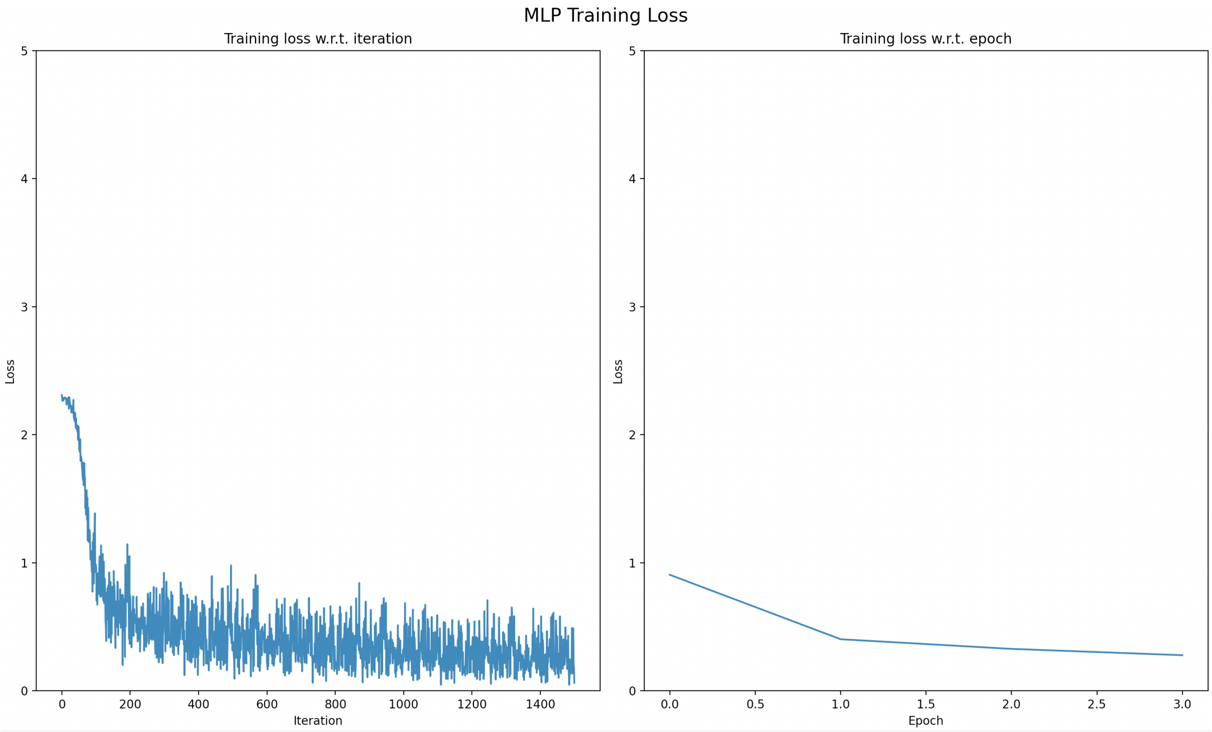
Input: dl_dy is the loss derivative with respect to the output y. x $\in \mathbb{R}^{H\times W\times C}$ is the input to the pooling layer.
Output: dl_dx $\in \mathbb{R}^{H\times W\times C}$ is the loss derivative with respect to the input x.

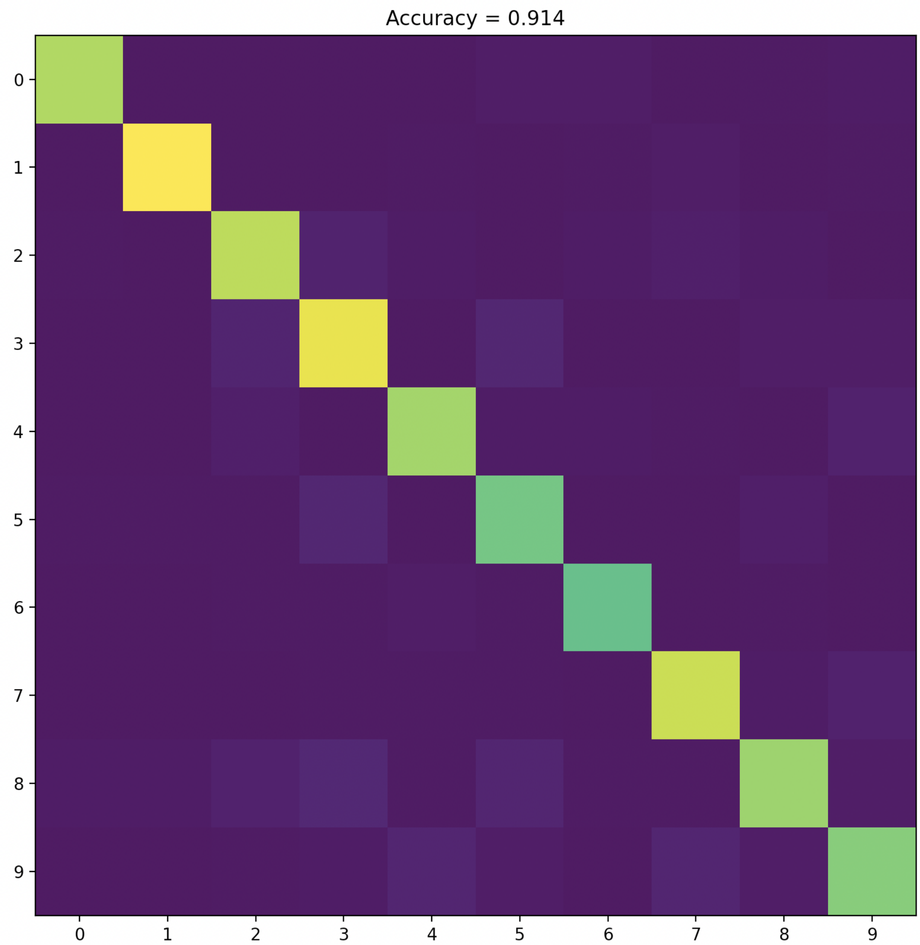
Solution. **conv** extracts the dimensions of the input tensor and the convolutional filter. It then calculates the padding required to maintain spatial dimensions during convolution. The input tensor is padded using NumPy's *np.pad* function to handle border effects. For each spatial position, the function extracts the corresponding region from the padded input tensor, performs element-wise multiplication with the convolutional filter, and sums the results along all dimensions except the channel dimension. The bias term is added to the sum. The final output tensor y represents the result of applying the convolution operation to the input tensor x with the specified convolutional filter *w_conv* and bias term *b_conv*.

conv_backward function performs the backpropagation step for a 2D convolution operation. It calculates and accumulates the gradients with respect to the filter weights and biases, allowing for the subsequent optimization of these parameters during the training of a neural network.

The **pool2x2** function implements a 2×2 max pooling operation on a given input tensor x. The function iterates over the input tensor and, for each 2×2 patch in each channel, extracts the maximum value. The result is stored in an output tensor y, effectively reducing the spatial dimensions by half in both height and width. The function employs a stride of 2 and is designed for 2D convolutional neural networks (CNNs) to down-sample feature maps, retaining essential information while reducing computational complexity.

pool2x2_backward function calculates the gradient of the loss with respect to the input tensor in the context of a 2×2 max pooling operation. It iterates through the output gradient tensor (dl_dy), identifies the corresponding 2×2 block in the input tensor x, and assigns the gradient values to the position of the maximum element. This facilitates the backpropagation process, crucial for updating model parameters during training in convolutional neural networks.


MLP Training Loss
Training loss w.r.t. iteration
Loss
Iteration
Training loss w.r.t. epoch
Loss
Epoch


Accuracy = 0.914