AI6104 Mathematics for AI Assignment

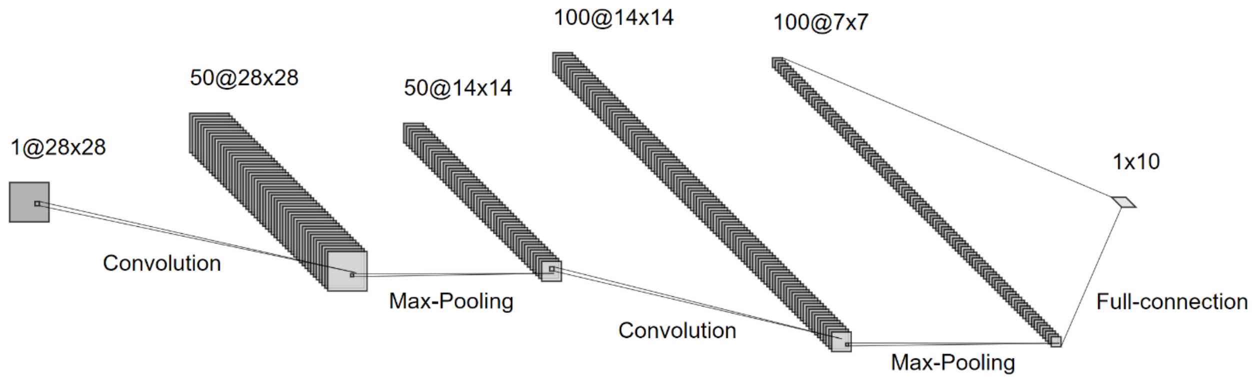
Chen Yongquan (G2002341D)

Nanyang Technological University

1.

a) Network Model

We will be using a convolutional neural network to classify MNIST handwritten digits, given in 28x28 single channel images.



The first layer of the network is a 50x3x3 convolution layer with stride 1 and padding 1 to retain the spatial dimension. Kernel has a dimension of 1x50x3x3. Using an input of size 1x28x28, this gives an output size of 50x28x28. For the equations that follow, w^1 refers to the weights in the first convolutional layer, w^2 refers to the weights in the second convolutional layer and w^3 refers to the weights in the fully-connected layer.

$$\boldsymbol{U} = \boldsymbol{w}^1 \star \boldsymbol{X} + \boldsymbol{b}^1$$

$$U_{i,j,k} = \left(\sum_{l=0}^{2} \sum_{m=0}^{2} w^1_{i,l,m} X_{sj+l,sk+m} \right) + b^1_i$$

$$\begin{aligned}
i &= \text{Filter/Output Feature Map index} \\
j &= \text{Output Feature Map y-coordinate} \\
k &= \text{Output Feature Map x-coordinate} \\
s &= \text{Stride} \\
\dim(\boldsymbol{U}) &= 50 \times 28 \times 28 \\
\dim(\boldsymbol{X}) &= 1 \times (28 + 2 \times 1) \times (28 + 2 \times 1) \text{ [padded dimension]} \\
\dim(\boldsymbol{w}^1) &= 1 \times 50 \times 3 \times 3 \\
\dim(\boldsymbol{b}^1) &= 50
\end{aligned}$$

$$\boldsymbol{U} = \begin{bmatrix} \begin{bmatrix} [(w^1_{0,0,0} \times X_{0,0}) + \cdots + (w^1_{0,2,2} \times X_{2,2})] + b^1_0 & \cdots & [(w^1_{0,0,0} \times X_{0,27}) + \cdots + (w^1_{0,2,2} \times X_{2,29})] + b^1_0 \\ \vdots & \ddots & \vdots \\ [(w^1_{0,0,0} \times X_{27,0}) + \cdots + (w^1_{0,2,2} \times X_{27,2})] + b^1_0 & \cdots & [(w^1_{0,0,0} \times X_{27,27}) + \cdots + (w^1_{0,2,2} \times X_{29,29})] + b^1_0 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} [(w^1_{49,0,0} \times X_{0,0}) + \cdots + (w^1_{49,2,2} \times X_{2,2})] + b^1_{49} & \cdots & [(w^1_{49,0,0} \times X_{0,27}) + \cdots + (w^1_{49,2,2} \times X_{2,29})] + b^1_{49} \\ \vdots & \ddots & \vdots \\ [(w^1_{49,0,0} \times X_{27,0}) + \cdots + (w^1_{49,2,2} \times X_{27,2})] + b^1_{49} & \cdots & [(w^1_{49,0,0} \times X_{27,27}) + \cdots + (w^1_{49,2,2} \times X_{29,29})] + b^1_{49} \end{bmatrix} \end{bmatrix}$$

This is followed by the ReLU activation layer and a max-pooling layer of filter size 2 and stride 2, giving an output of size 50x14x14. For simplicity reasons we use the same output label remain for both the two layers.

$$\begin{aligned}
U'_{i,j,k} &= \text{ReLU}(U_{i,j,k}) = \max(0, U_{i,j,k}) \\
U'_{i,j,k} &= \text{Max-Pooling}(\boldsymbol{U}') = \max(U'_{i,sj,sk}, U'_{i,sj,sk+1}, U'_{i,sj+1,sk}, U'_{i,sj+1,sk+1})
\end{aligned}$$

Next, there is a second convolution layer of dimension 100x3x3 with the same stride and padding values, followed by similar activation and max-pooling layers. Each point in the output feature map is the summation of the convolution result of all the input feature maps at that point. Each input feature map has a separate 3x3 kernel, and each output channel has a separate group of these 3x3 kernel, giving the final kernel dimension 50x100x3x3. Output size after convolution is 100x14x14 and after max-pooling is 100x7x7.

$$V_{i,j,k} = \left(\sum_{l=0}^{49} \sum_{m=0}^{2} \sum_{n=0}^{2} w^2_{i,l,m,n} U'_{l,sj+m,sk+n} \right) + b^2_i$$

$$\dim(\boldsymbol{V}) = 100 \times 14 \times 14$$
$$\dim(\boldsymbol{U'}) = 50 \times (14 + 2 \times 1) \times (14 + 2 \times 1) \text{ [padded dimension]}$$
$$\dim(\boldsymbol{w^2}) = 50 \times 100 \times 3 \times 3$$
$$\dim(\boldsymbol{b^2}) = 100$$

$$V'_{i,j,k} = \text{ReLU}(V_{i,j,k}) = \max(0, V_{i,j,k})$$
$$V'_{i,j,k} = \text{Max-Pooling}(\boldsymbol{V'}) = \max\left(V'_{i,sj,sk}, V'_{i,sj,sk+1}, V'_{i,sj+1,sk}, V'_{i,sj+1,sk+1}\right)$$

Finally, the feature maps are flattened into a single vector of size 1x4900 and passed into a fully connected layer of 10 nodes for each digit class.

$$Z_i = \left(\sum_{j=0}^{100} \sum_{k=0}^{7} \sum_{l=0}^{7} w^3_{i,j,k,l} V'_{j,k,l} \right) + b^3_i \ for\ i = 0, \dots, 9$$

$$\dim(\boldsymbol{Z}) = 10$$
$$\dim(\boldsymbol{V'}) = 100 \times 7 \times 7 \text{ [padded dimension]}$$
$$\dim(\boldsymbol{w^3}) = 4900 \times 10$$
$$\dim(\boldsymbol{b^3}) = 10$$

c) Objective function

The loss function is cross-entropy loss which will be a combination of the softmax function and negative log likelihood function. Cross-entropy loss is used for multiclass classification problems and the MNIST dataset has 10 classes (0-9). The softmax function is defined as follows:

$$\hat{Y}_i = \frac{e^{\hat{Z}_i}}{\sum_{j=0}^{9}\left(e^{\hat{Z}_j}\right)}$$

For a single input image x with target class c, the full cross-entropy loss function can be defined as:

$$L = \ell(Y, \hat{Y}) = -\log\left(\frac{e^{\hat{Z}_i}}{\sum_{j=0}^{9}\left(e^{\hat{Z}_j}\right)} \right) = -\log\left(e^{\hat{Z}_c}\right) + \log\left(\sum_{j=0}^{9} \left(e^{\hat{Z}_j}\right) \right)$$
$$= -\hat{z}_c + \log\left(\sum_{j=0}^{9} \left(e^{\hat{Z}_j}\right) \right)$$

The logarithm function here refers to the natural logarithm.

2. Gradients

Softmax Layer

$$\frac{\partial L}{\partial Z_i} = \frac{\partial}{\partial Z_i}\left(-\hat{Z}_c + \log\left(\sum_{i=0}^{9}\left(e^{\hat{Z}_i}\right)\right)\right) = \begin{cases} \frac{e^{\hat{Z}_i}}{\sum_{i=0}^{9}\left(e^{\hat{Z}_i}\right)} - 0 = \hat{Y}_i - 0 & if\ i \neq c \\ \frac{e^{\hat{Z}_i}}{\sum_{i=0}^{9}\left(e^{\hat{Z}_i}\right)} - 1 = \hat{Y}_i - 1 & if\ i = c \end{cases}$$

The derivative vector would be the vector containing predicted probabilities for all 10 digits minus a one-hot encoded vector of the target labels.

Fully Connected Layer

$$\frac{\partial L}{\partial \boldsymbol{w}^3} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{w}^3} \qquad \frac{\partial L}{\partial \boldsymbol{b}^3} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{b}^3}$$

$$\frac{\partial L}{\partial \boldsymbol{Z}} = \begin{bmatrix} \frac{\partial L}{\partial Z_0} & \cdots & \frac{\partial L}{\partial Z_9} \end{bmatrix} \quad Calculated\ above$$

$$\frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{w}^3} = \begin{bmatrix} \frac{\partial Z_0}{\partial w^3_{0,0,0,0}} & \cdots & \frac{\partial Z_0}{\partial w^3_{49,99,6,6}} \\ \vdots & \ddots & \vdots \\ \frac{\partial Z_9}{\partial w^3_{0,0,0,0}} & \cdots & \frac{\partial Z_9}{\partial w^3_{49,99,6,6}} \end{bmatrix} = \begin{bmatrix} V'_{0,0,0} & \cdots & V'_{99,6,6} \\ \vdots & \ddots & \vdots \\ V'_{0,0,0} & \cdots & V'_{99,6,6} \end{bmatrix} Calculated\ during\ forward\ pass$$

$$\frac{\partial L}{\partial \boldsymbol{w}^3} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{w}^3} = \begin{bmatrix} \frac{\partial L}{\partial w^3_{0,0,0,0}} & \cdots & \frac{\partial L}{\partial w^3_{49,99,6,6}} \end{bmatrix}$$

$$= \left[\left(\frac{\partial L}{\partial Z_0} \cdot \frac{\partial Z_0}{\partial w^3_{0,0,0,0}} + \cdots + \frac{\partial L}{\partial Z_9} \cdot \frac{\partial Z_9}{\partial w^3_{0,0,0,0}}\right) \quad \cdots \quad \left(\frac{\partial L}{\partial Z_0} \cdot \frac{\partial Z_0}{\partial w^3_{49,99,6,6}} + \cdots + \frac{\partial L}{\partial Z_9} \cdot \frac{\partial Z_9}{\partial w^3_{49,99,6,6}}\right)\right]$$

$$= \left[\left(\left(\hat{Y}_0 - \mathbb{I}(0 = c)\right) V'_{0,0,0} + \cdots + \left(\hat{Y}_9 - \mathbb{I}(9 = c)\right) V'_{0,0,0}\right) \quad \cdots \quad \left(\left(\hat{Y}_0 - \mathbb{I}(0 = c)\right)\left(V'_{99,6,6}\right) + \cdots + \left(\hat{Y}_9 - \mathbb{I}(9 = c)\right)\left(V'_{99,6,6}\right)\right)\right]$$

Matrices dimensions may vary depending on the data structure of the filter gradients e.g. order of nesting of inplanes and outplanes. To simplify the matrices shown here, we will squeeze/flatten the dimensions of the filters for all layers in our calculations and arrange all weights in a single row in the Jacobian matrices, but dimensions here will differ from the actual coded dimensions for the network.

$$\frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{b}^3} = \begin{bmatrix} \frac{\partial Z_0}{\partial b^3_0} & \cdots & \frac{\partial Z_0}{\partial b^3_9} \\ \vdots & \ddots & \vdots \\ \frac{\partial Z_9}{\partial b^3_0} & \cdots & \frac{\partial Z_9}{\partial b^3_9} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

Second Convolutional Layer

$$\frac{\partial L}{\partial \boldsymbol{w}^2} = \frac{\partial L}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w}^2} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w}^2} \qquad \frac{\partial L}{\partial \boldsymbol{b}^2} = \frac{\partial L}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{b}^2} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{b}^2}$$

$$\frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{V}} = \begin{bmatrix} \frac{\partial Z_0}{\partial V_{0,0,0}} & \cdots & \frac{\partial Z_0}{\partial V_{99,13,13}} \\ \vdots & \ddots & \vdots \\ \frac{\partial Z_9}{\partial V_{0,0,0}} & \cdots & \frac{\partial Z_9}{\partial V_{99,13,13}} \end{bmatrix} = \begin{bmatrix} w^3_{0,0,0,0} & \cdots & w^3_{0,99,6,6} \\ \vdots & \ddots & \vdots \\ w^3_{9,0,0,0} & \cdots & w^3_{9,99,6,6} \end{bmatrix}$$

Actual matrix will have sparsity after backpropagation through max-pooling and ReLU layers.

$$\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{w}^2} = \begin{bmatrix} \frac{\partial V_{0,0,0}}{\partial w^2_{0,0,0,0}} & \cdots & \frac{\partial V_{0,0,0}}{\partial w^2_{49,0,2,2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial V_{99,13,13}}{\partial w^2_{0,99,0,0}} & \cdots & \frac{\partial V_{99,13,13}}{\partial w^2_{49,99,2,2}} \end{bmatrix} = \begin{bmatrix} U'_{0,0,0} & \cdots & U'_{49,2,2} \\ \vdots & \ddots & \vdots \\ U'_{0,0,0} & \cdots & U'_{49,2,2} \end{bmatrix}$$

Because the operation from $\boldsymbol{U}$ to $\boldsymbol{V}$ is a convolution operation, the inner elements of the above final Jacobian will have to be mapped back according to the convolutional operation instead of performing a simple dot product. So, for instance:

$$\begin{aligned} \frac{\partial L}{\partial w^2_{0,0,0,0}} &= \frac{\partial L}{\partial V_{0,0,0}} \cdot \frac{\partial V_{0,0,0}}{\partial w^2_{0,0,0,0}} + \cdots + \frac{\partial L}{\partial V_{0,13,13}} \cdot \frac{\partial V_{0,13,13}}{\partial w^2_{0,49,0,0}} \\ &= \frac{\partial L}{\partial V_{0,0,0}} \cdot U'_{0,0,0} + \cdots + \frac{\partial L}{\partial V_{0,0,13}} \cdot U'_{0,13,13} + \frac{\partial L}{\partial V_{0,1,0}} \cdot U'_{0,1,0} + \cdots + \frac{\partial L}{\partial V_{0,13,13}} \cdot U'_{49,13,13} \end{aligned}$$

Essentially, the final matrix of $\frac{\partial L}{\partial \boldsymbol{w}^2}$ can be viewed as the result of convolving $\boldsymbol{U}'$ with $\frac{\partial L}{\partial \boldsymbol{V}}$.

$$\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{b}^2} = \begin{bmatrix} \frac{\partial V_{0,0,0}}{\partial b^2_0} & \cdots & \frac{\partial V_{0,0,0}}{\partial b^2_{99}} \\ \vdots & \ddots & \vdots \\ \frac{\partial V_{99,13,13}}{\partial b^2_0} & \cdots & \frac{\partial V_{99,13,13}}{\partial b^2_{99}} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

First Convolutional Layer

$$\frac{\partial L}{\partial \boldsymbol{w}^1} = \frac{\partial L}{\partial \boldsymbol{U}} \cdot \frac{\partial \boldsymbol{U}}{\partial \boldsymbol{w}^1} = \frac{\partial L}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{U}} \cdot \frac{\partial \boldsymbol{U}}{\partial \boldsymbol{w}^1} \qquad \frac{\partial L}{\partial \boldsymbol{b}^1} = \frac{\partial L}{\partial \boldsymbol{U}} \cdot \frac{\partial \boldsymbol{U}}{\partial \boldsymbol{b}^1} = \frac{\partial L}{\partial \boldsymbol{V}} \cdot \frac{\partial \boldsymbol{V}}{\partial \boldsymbol{U}} \cdot \frac{\partial \boldsymbol{U}}{\partial \boldsymbol{b}^1}$$

$$\frac{\partial L}{\partial \boldsymbol{V}} = \frac{\partial L}{\partial \boldsymbol{Z}} \cdot \frac{\partial \boldsymbol{Z}}{\partial \boldsymbol{V}} \quad \mathit{Calculated\ in\ previous\ layer}$$

$$\frac{\partial \boldsymbol{V}}{\partial \boldsymbol{U}} = \begin{bmatrix} \frac{\partial V_{0,0,0}}{\partial U_{0,0,0}} & \cdots & \frac{\partial V_{0,0,0}}{\partial U_{49,27,27}} \\ \vdots & \ddots & \vdots \\ \frac{\partial V_{99,13,13}}{\partial U_{0,0,0}} & \cdots & \frac{\partial V_{99,13,13}}{\partial U_{49,27,27}} \end{bmatrix} = \begin{bmatrix} w^2_{0,0,0,0} & \cdots & w^2_{0,49,2,2} \\ \vdots & \ddots & \vdots \\ w^2_{99,0,0,0} & \cdots & w^2_{99,49,2,2} \end{bmatrix}$$

$$\frac{\partial \boldsymbol{U}}{\partial \boldsymbol{w}^1} = \begin{bmatrix} \frac{\partial U_{0,0,0}}{\partial w^1_{0,0,0}} & \cdots & \frac{\partial U_{0,0,0}}{\partial w^1_{0,2,2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial U_{49,27,27}}{\partial w^1_{49,0,0}} & \cdots & \frac{\partial U_{49,27,27}}{\partial w^1_{49,2,2}} \end{bmatrix} = \begin{bmatrix} X_{0,0} & \cdots & X_{29,29} \\ \vdots & \ddots & \vdots \\ X_{0,0} & \cdots & X_{29,29} \end{bmatrix}$$

$\frac{\partial L}{\partial \boldsymbol{w}^1}$ will similarly have to be mapped back through the convolution operation here instead of a dot product as per the backpropagation through the fully connected layer.

$\frac{\partial L}{\partial \boldsymbol{U}}$ can be obtained in the previous layer by convolving $\frac{\partial L}{\partial \boldsymbol{V}}$ with the weight matrix of convolutional layer 2.

Note that the convolution here is an actual convolution with an inverted weight matrix instead of the usual cross-correlation operation.

$$\frac{\partial \boldsymbol{U}}{\partial \boldsymbol{b}^1} = \begin{bmatrix} \frac{\partial U_{0,0,0}}{\partial b_0^1} & \cdots & \frac{\partial U_{0,0,0}}{\partial b_{49}^1} \\ \vdots & \ddots & \vdots \\ \frac{\partial U_{49,27,27}}{\partial b_0^1} & \cdots & \frac{\partial U_{49,27,27}}{\partial b_{49}^1} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

When backpropagating through the max-pooling layer to obtain a larger sized matrix, only the maximum cells during the forward pass are filled with backpropagated gradients while all other cells are set to 0.

When backpropagating through the ReLU layer, the derivative of activated neurons in the forward pass is 1 while others are 0, and backpropagated gradients are multiplied as such.

3. Training by Gradient Descent

To train the network by gradient descent, we subtract each weight by the derivative of the loss with respect to the weight, multiplied by a learning rate coefficient:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - lr \cdot \frac{\partial L}{\partial \boldsymbol{w}}$$

So,

$$\begin{aligned} \boldsymbol{w}_{t+1}^1 &= \boldsymbol{w}_t^1 - lr \cdot \frac{\partial L}{\partial \boldsymbol{w}^1} & \boldsymbol{b}_{t+1}^1 &= \boldsymbol{b}_t^1 - lr \cdot \frac{\partial L}{\partial \boldsymbol{b}^1} \\ \boldsymbol{w}_{t+1}^2 &= \boldsymbol{w}_t^2 - lr \cdot \frac{\partial L}{\partial \boldsymbol{w}^2} & \boldsymbol{b}_{t+1}^2 &= \boldsymbol{b}_t^2 - lr \cdot \frac{\partial L}{\partial \boldsymbol{b}^2} \\ \boldsymbol{w}_{t+1}^3 &= \boldsymbol{w}_t^3 - lr \cdot \frac{\partial L}{\partial \boldsymbol{w}^3} & \boldsymbol{b}_{t+1}^3 &= \boldsymbol{b}_t^3 - lr \cdot \frac{\partial L}{\partial \boldsymbol{b}^3} \end{aligned}$$

The derivatives have been calculated as shown in the previous section. Because we will be performing stochastic gradient descent, the gradient will be the average gradient across N mini-batches.

4. Key Code Snippets

Key snippets of the code are attached below for reference.

Forward Pass

```
def conv2d(x, w, b, stride, padding):
  h_f, w_f, c_in, c_out = w.shape
  n, h_in, w_in, c_in = x.shape
  out_size = int((h_in - h_f + 2 * padding)/stride + 1)
  x_pad = np.pad(x, ((0,0), (1,1), (1,1), (0,0)))
  output = np.zeros((n, out_size,out_size, c_out))

  for i in range(out_size):
    for j in range(out_size):
      h_start = i * stride
      h_end = h_start + h_f
      w_start = j * stride
      w_end = w_start + w_f

      output[:, i, j, :] = np.sum(x_pad[:, h_start:h_end, w_start:w_end, :, np.newaxis] *
w[np.newaxis, :, :, :], axis=(1, 2, 3))
  return x, output + b

def relu(x):
  out = np.maximum(0, x)
  return out, out

def pool2d(x, pool_size, stride):
  n, h_in, w_in, c_in = x.shape
  out_size = int((h_in - pool_size)/stride + 1)
  out = np.zeros((n, out_size, out_size, c_in))
  masks = {}

  for i in range(out_size):
    for j in range(out_size):
      h_start = i * stride
      h_end = h_start + pool_size
      w_start = j * stride
      w_end = w_start + pool_size
      x_slice = x[:, h_start:h_end, w_start:w_end, :]

      mask = np.zeros_like(x_slice)
      idx = np.argmax(x_slice.reshape(n, pool_size * pool_size, c_in), axis = 1)
      n_idx, c_idx = np.indices((n, c_in))
      mask.reshape(n, pool_size * pool_size, c_in)[n_idx, idx, c_idx] = 1
      masks[(i,j)] = mask

      out[:, i, j, :] = np.max(x_slice, axis = (1, 2))
  return masks, out

def linear(x, w, b):
  return x, np.dot(x, w.T) + b

def softmax(x):
  e = np.exp(x - x.max(axis = 1, keepdims = True)) # Subtract maximum to prevent
overflow/underflow
  return e / np.sum(e, axis = 1, keepdims = True)

def crossentropyloss(y_hat, y):
  return - np.sum(y * np.log(np.clip(y_hat, 1e-20, 1.))) / y.shape[0]

def forward_pass(X_batch):
  global conv1_x, relu1_mask, pool1_mask, conv2_x, relu2_mask, pool2_mask, ll_x
  conv1_x, x = conv2d(X_batch, conv1_w, conv1_b, conv_stride, conv_padding)
  relu1_mask, x = relu(x)
  pool1_mask, x = pool2d(x, pool_size, pool_stride)
  conv2_x, x = conv2d(x, conv2_w, conv2_b, conv_stride, conv_padding)
  relu2_mask, x = relu(x)
  pool2_mask, x = pool2d(x, pool_size, pool_stride)
  x = np.ravel(x).reshape(x.shape[0], -1)
  ll_x, x = linear(x, ll_w, ll_b)
  x = softmax(x)
  return x
```

Backward Pass

```
def conv2d_backward(diff, x, w, b, stride, padding):
  h_f, w_f, c_in, c_out = w.shape
  n, h_in, w_in, c_in = x.shape
  x_pad = np.pad(x, ((0,0), (1,1), (1,1), (0,0)))
  diff_new = np.zeros(x_pad.shape)
  dw = np.zeros_like(w)
  db = diff.sum(axis = (0, 1, 2)) / diff.shape[0]

  for i in range(diff.shape[1]):
    for j in range(diff.shape[2]):
      h_start = i * stride
      h_end = h_start + h_f
      w_start = j * stride
      w_end = w_start + w_f
      diff_new[:, h_start:h_end, w_start:w_end, :] += np.sum(
        w[np.newaxis, :, :, :, :] *
        diff[:, i:i+1, j:j+1, np.newaxis, :],
        axis=4
      )
      dw += np.sum(
        x_pad[:, h_start:h_end, w_start:w_end, :, np.newaxis] *
        diff[:, i:i+1, j:j+1, np.newaxis, :],
        axis=0
      )

  dw /= diff.shape[0]
  w = w - lr * dw
  b = b - lr * db
  return diff_new[:, padding:padding+h_in, padding:padding+w_in, :], w, b

def relu_backward(diff, mask):
  diff[mask <= 0] = 0
  return diff

def pool2d_backward(diff, mask, pool_size, stride, out_size):
  diff_next = np.zeros(out_size)

  for i in range(diff.shape[1]):
    for j in range(diff.shape[2]):
      h_start = i * stride
      h_end = h_start + pool_size
      w_start = j * stride
      w_end = w_start + pool_size
      diff_next[:, h_start:h_end, w_start:w_end, :] += diff[:, i:i+1, j:j+1, :] * mask[(i,
j)]
    return diff_next

def linear_backward(diff, x, w, b):
  diff_next = np.dot(diff, w)
  n = x.shape[0]
  dw = np.dot(diff.T, x) / n
  db = np.sum(diff, axis = 0, keepdims = True) / n
  w = w - lr * dw
  b = b - lr * db
  return diff_next, w, b

def backward_pass(grad, n):
  global conv1_w, conv1_b, conv2_w, conv2_b, ll_w, ll_b
  grad, ll_w, ll_b = linear_backward(grad, ll_x, ll_w, ll_b)
  grad = grad.reshape(n, 7, 7, 100)
  grad = pool2d_backward(grad, pool2_mask, pool_size, pool_stride, (n, 14, 14, 100))
  grad = relu_backward(grad, relu2_mask)
  grad, conv2_w, conv2_b = conv2d_backward(grad, conv2_x, conv2_w, conv2_b, conv_stride,
conv_padding)
  grad = pool2d_backward(grad, pool1_mask, pool_size, pool_stride, (n, 28, 28, 50))
  grad = relu_backward(grad, relu1_mask)
  grad, conv1_w, conv1_b = conv2d_backward(grad, conv1_x, conv1_w, conv1_b, conv_stride,
conv_padding)
```

5. Training Report

To flatten the problem landscape and reduce training epochs, we normalize inputs by the mean and standard deviation of the entire training set. For testing, normalization is done using the same mean and standard deviation of the training set. To improve generalization and add some regularization to the model, inputs are sampled randomly from the entire training set without replacement for training.

MNIST training and testing data are converted from 8-bit intensity values in the range of $[0,255]$, into floating point values in the range of $[0.0,1.0]$.

We perform training for 10 epochs with a batch size of 128, resulting in 479 training iterations per epoch across 60000 training samples. For testing, 79 iterations are required for 10000 test samples.

Figure 1 shows the training loss across all 4790 iterations. Because the training loss per iteration is calculated across the mini-batch, it will be less stable than the ones per epoch, which is averaged across all iterations each epoch for the entire dataset. The x-axis is truncated and starts from the 10th iteration to focus on a smaller y-axis range.

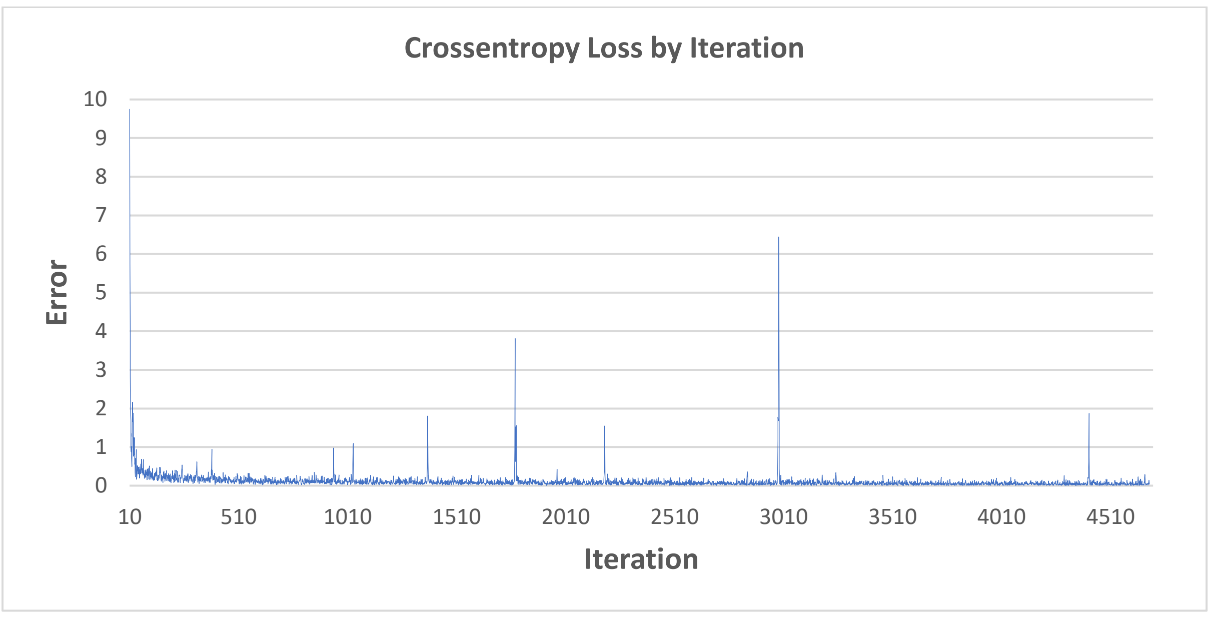
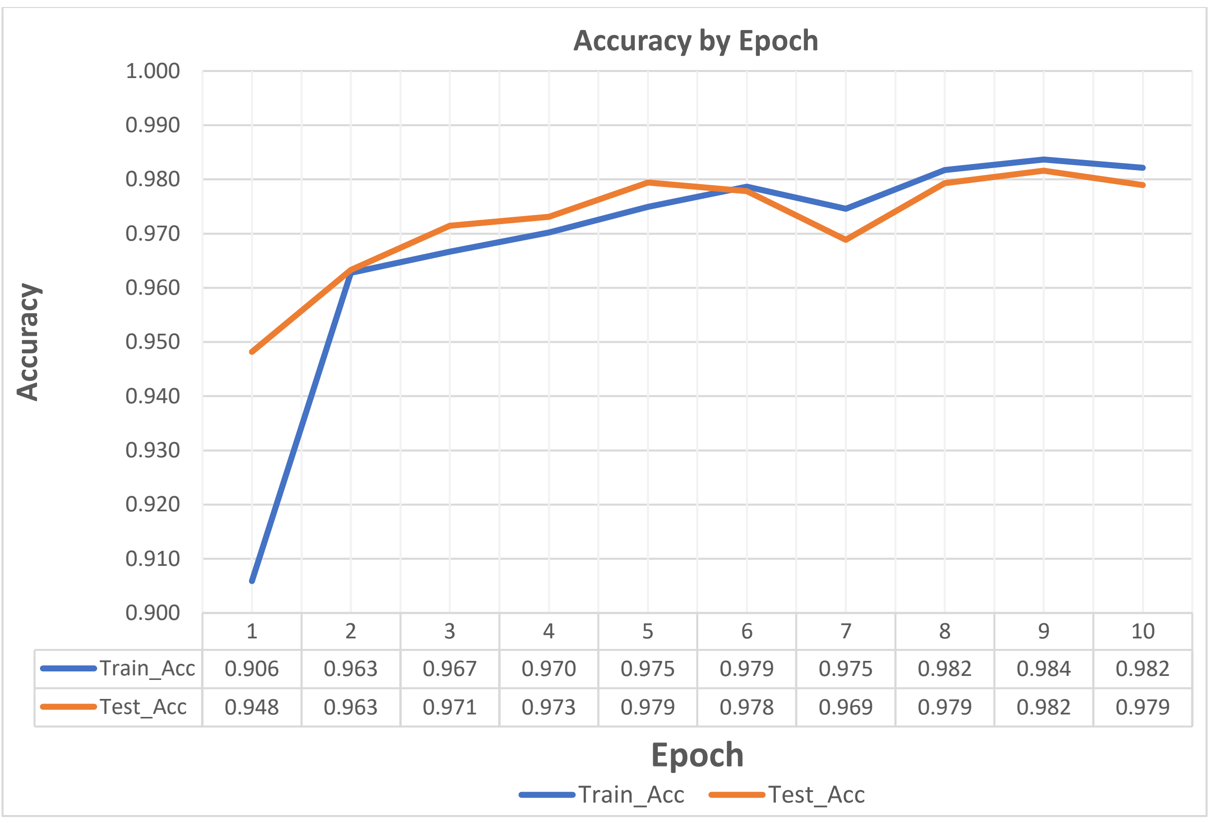
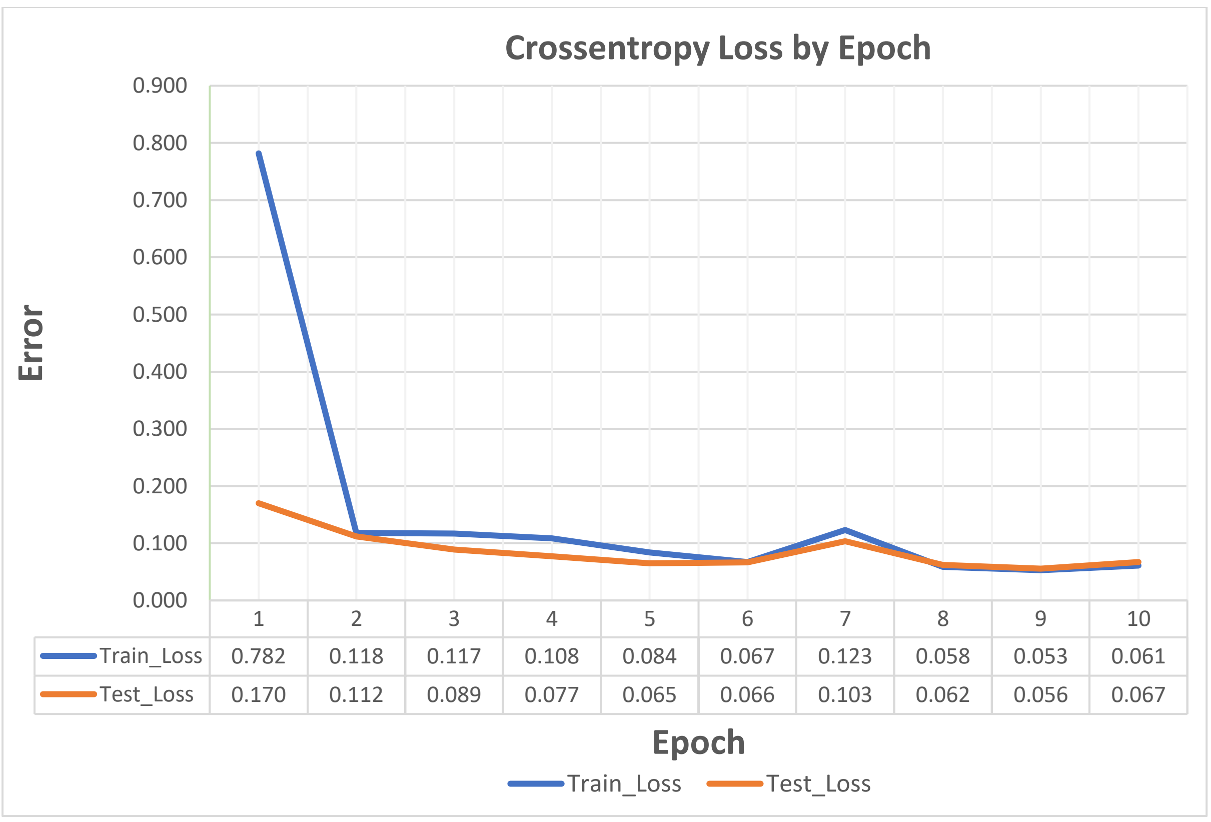


Figure 1



	1	2	3	4	5	6	7	8	9	10
Train_Acc	0.906	0.963	0.967	0.970	0.975	0.979	0.975	0.982	0.984	0.982
Test_Acc	0.948	0.963	0.971	0.973	0.979	0.978	0.969	0.979	0.982	0.979

Figure 2



	1	2	3	4	5	6	7	8	9	10
Train_Loss	0.782	0.118	0.117	0.108	0.084	0.067	0.123	0.058	0.053	0.061
Test_Loss	0.170	0.112	0.089	0.077	0.065	0.066	0.103	0.062	0.056	0.067

Figure 3