Introduction to Machine Learning (Spring 2019)
Homework #5 (50 Pts, June 5)

Student ID 2014310279

Name 김 승현

Instruction: We provide all codes and datasets in Python. Please write your code to complete Convolutional Neural Network Classifier. Compress 'Answer.py' & your report ONLY and submit with the filename 'HW5_STUDENT_ID.zip'.

(1) [30 pts] Implement CNN Classifier in 'Answer.py' with the loss function as follows:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i,$$

$$L_i = -\sum_{j=1}^{C} y_j log p_j,$$

where N is the number of (batch) data, C is the number of classes.

(a) [Convolution 2D] Implement convolution function in 'Answer.py' ('convolution2d').

(b) [ReLU] Implement ReLU activation in 'Answer.py' ('ReLU').

(c) [Convolution Layer] Implement a convolution layer in 'Answer.py' ('ConvolutionLayer').

(d) [Max-Pooling Layer] Implement a max-pooling layer in 'Answer.py' ('MaxPoolingLayer').

(e) [FC Layer & Softmax] Implement a FC, softmax layer in 'Answer.py' ('FCLayer', 'SoftmaxLayer').

Answer: Fill your code here. You also have to submit your code to i-campus.

NOTE 1: **You should write your codes in 'EDIT HERE' signs.** It is not recommended to edit other parts. Once you complete your implementation, run the check codes ('Checker.py') to check if it is done correctly.
NOTE 2: **Read the instructions in template codes VERY CAREFULLY.** Funcionality and input, output shape of any function must be the same as what is written.

 # Convolution 2D

```
    # ============================== EDIT HERE ==============================
    conv_height = int(1 + ((height - kernel_size) / stride))
    conv_width = int(1 + ((width - kernel_size) / stride))
    conv_out = np.zeros((conv_height, conv_width))

    for  i in range(conv_height):
        for j in range(conv_width):
```

```
            tmp_x = x[i*stride:i*stride+kernel_size,
j*stride:j*stride+kernel_size]
            conv_out[i, j] = np.sum(tmp_x * kernel)
    # ================================================================
```

 # ReLU

```
    def forward(self, z):
        out = None
        # ================================ EDIT HERE ===============
        self.zero_mask = (z < 0)
        out = z.copy()
        out[self.zero_mask] = 0
        # ==========================================================
        return out

    def backward(self, d_prev):
        dz = None
        # ================================ EDIT HERE ================
        d_prev[self.zero_mask] = 0
        dz = d_prev
        # ==========================================================
        return dz
```

 # Convolution Layer

```
def convolution(self, x, kernel, bias=None, stride=1, pad=0):

        batch_size, in_channel, _, _ = x.shape
        if pad > 0:
            x = self.zero_pad(x, pad)
        _, _, height, width = x.shape
        out_channel, _, kernel_size, _ = kernel.shape
        assert x.shape[1] == kernel.shape[1]

        conv = None
        # ================================ EDIT HERE ====================
        conv_height = int( 1+ ((height - kernel_size) / stride))
        conv_width = int( 1 + ((width - kernel_size) / stride))
        conv = np.zeros((batch_size, out_channel, conv_height, conv_width))
        for n in range(batch_size):
```

```
            for f in range(out_channel):
                for k in range(in_channel):
                    conv[n, f] += convolution2d(x[n, k], kernel[f, k], stride)
                if bias is not None:
                    conv[n, f] += bias[f]

        # ================================================================
        return conv

    def backward(self, d_prev):
        batch_size, in_channel, height, width = self.x.shape
        out_channel, _, kernel_size, _ = self.W.shape

        if len(d_prev.shape) < 3:
            d_prev = d_prev.reshape(*self.output_shape)

        self.dW = np.zeros_like(self.W, dtype=np.float64)
        self.db = np.zeros_like(self.b, dtype=np.float64)
        dx = np.zeros_like(self.x, dtype=np.float64)
        # ============================== EDIT HERE ==========================
        std = self.stride
        d_prev_height, d_prev_width = d_prev.shape[2], d_prev.shape[3]
        # dW
        tmp_x = np.transpose(self.x, [1, 0, 2, 3])
        tmp_d = np.transpose(d_prev, [1, 0, 2, 3])
        tmp_dw = self.convolution(tmp_x, tmp_d, None)
        self.dW = np.transpose(tmp_dw, [1, 0, 2, 3])

        # db
        for f in range(out_channel):
            self.db[f] = np.sum(d_prev[:, f])

        # dx
        d_pad = self.zero_pad(d_prev, kernel_size-self.pad-1)
        tmp_w = np.zeros_like(self.W, dtype=np.float64)
        for f in range(out_channel):
            for k in range(in_channel):
                tmp_w[f, k] = np.rot90(self.W[f, k], 2)
```

```
        tmp_w = np.transpose(tmp_w, [1,0,2,3])
        dx = self.convolution(d_pad, tmp_w , None, std)

        # ================================================
        return dx

    def zero_pad(self, x, pad):
        padded_x = None
        batch_size, in_channel, height, width = x.shape
        # ============================== EDIT HERE ======================
        padded_x = np.pad(x,((0,), (0,), (pad,), (pad,)), mode='constant')
        # ================================================================
        return padded_x
```

 # MaxPoolingLayer

```
def forward(self, x):
        max_pool = None
        batch_size, channel, height, width = x.shape
        # Where it came from x. (1 if it is pooled, 0 otherwise.)
        # Might be useful when backward
        self.mask = np.zeros_like(x)
        # ============================== EDIT HERE ======================
        std = self.stride
        pool_height = int((height - self.kernel_size) / std + 1 )
        pool_width = int((width - self.kernel_size) / std + 1 )
        max_pool = np.zeros((batch_size, channel, pool_height, pool_width))

        for n in range(batch_size):
            for c in range(channel):
                for h in range(pool_height):
                    for w in range(pool_width):
                        tmp_mask = self.mask[n, c, h*std : h*std +
self.kernel_size, w*std : w*std + self.kernel_size]
                        tmp_x = x[n, c, h*std : h*std + self.kernel_size, w*std :
w*std + self.kernel_size]
                        max_pool[n, c, h, w] = np.max(tmp_x)
                        tmp_mask = np.where(tmp_x == np.max(tmp_x), 1, 0)
                        self.mask[n, c, h*std : h*std + self.kernel_size, w*std :
w*std + self.kernel_size] = tmp_mask
```

```
        # ============================================================
        self.output_shape = max_pool.shape
        return max_pool


    def backward(self, d_prev=1):
        d_max = None
        if len(d_prev.shape) < 3:
            d_prev = d_prev.reshape(*self.output_shape)
        batch, channel, height, width = d_prev.shape
        # ================================ EDIT HERE ======================
        std = self.stride
        tmp_h, tmp_w = self.mask.shape[2], self.mask.shape[3]
        d_max = np.zeros_like(self.mask)
        d_prev_pp = np.zeros_like(self.mask)

        for n in range(batch):
            for c in range(channel):
                for h in range(height):
                    for w in range(width):
                        tmp_k = d_prev[n, c, h, w]
                        for i in range(self.kernel_size):
                            for j in range(self.kernel_size):
                                d_prev_pp[n, c, h*std+i, w*std+j] = tmp_k

        for n in range(batch):
            for c in range(channel):
                for h in range(tmp_h):
                    for w in range(tmp_w):
                        if self.mask[n, c, h, w] == 1:
                            d_max[n, c, h, w] = d_prev_pp[n, c, h, w]
        # ================================================================|
        return d_max
```

 # FCLayer & Softmax

```
class FCLayer:
    def __init__(self, input_dim, output_dim):
        # Weight Initialization
```

```
        self.W = np.random.randn(input_dim, output_dim) / np.sqrt(input_dim / 2)
        self.b = np.zeros(output_dim)

    def forward(self, x):
        if len(x.shape) > 2:
            batch_size = x.shape[0]
            x = x.reshape(batch_size, -1)
        self.x = x
        # ================================ EDIT HERE ================================
        self.out = np.dot(x, self.W) + self.b
        # ===========================================================================
        return self.out

    def backward(self, d_prev):
        self.dW = np.zeros_like(self.W, dtype=np.float64)    # Gradient w.r.t.
weight (self.W)
        self.db = np.zeros_like(self.b, dtype=np.float64)    # Gradient w.r.t.
bias (self.b)
        dx = np.zeros_like(self.x, dtype=np.float64)         # Gradient w.r.t.
input x
        # ================================ EDIT HERE ================================
        self.dW = np.dot((self.x).T, d_prev)
        self.db = np.sum(d_prev, axis=0)
        dx = np.dot(d_prev, (self.W).T)
        # ===========================================================================
        return dx
```

```
class SoftmaxLayer:
    def forward(self, x):
        y_hat = None
        # ================================ EDIT HERE ================================
        y_hat = softmax(x)
        self.y_hat = y_hat
        # ===========================================================================
        return self.y_hat

    def backward(self, d_prev=1):
        batch_size = self.y.shape[0]
        dx = None
```

```
        # ============================== EDIT HERE ==============================
        dProb = (self.y_hat).copy()
        dProb[np.arange(batch_size), np.argmax(self.y,axis=1)] -= 1
        dProb /= batch_size
        dx = dProb * d_prev
        # ================================================================
        return dx

    def ce_loss(self, y_hat, y):
        self.loss = None
        eps = 1e-10
        self.y_hat = y_hat
        self.y = y
        # ============================== EDIT HERE ======================
        batch_size = self.y.shape[0]
        log_probs = -self.y * np.log(self.y_hat + eps)
        ce_loss = np.sum(log_probs) / batch_size
        self.loss = ce_loss
        # ================================================================]
        return self.loss
```

(2) [20 Pts] Experiment results

(a) you are given a small MNIST dataset with 5 labels (0, 1, 2, 3, 4), which originally has 10 labels. Given CNN architecture and hyperparameters as below, build the classifier and adjust hyperparameters to achieve best test accuracy. (Your best accuracy should be at least 0.8 if the model is trained correctly.)

Answer: Fill the blank in the table. Show the plot of training & test accuracy with a brief explanation.

[CNN Architecture]

Layer name	Configuration
Conv - 1	Out Channel = 8, Kernel size = 3 Stride = 1, Pad = 1
ReLU - 1	-
Conv – 2	Out Channel = 8, Kernel size = 3 Stride = 1, Pad = 1
ReLU - 2	-
Max-pool - 1	Kernel size = 2, stride = 2
FC – 1	Input dim = 1568, Output dim = 500

FC - 2	Input dim = 500, Output dim = 5
Softmax Layer	-

[Results]

Epochs	Learning rate	Best Acc.	Best Epoch.
10	0.01	0.97	7

Plot Sample (Values are not correct. Delete when you submit).