**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:
2. **[Convolution 2D]** Implement convolution function in ‘Answer.py’ (‘convolution2d’).
3. **[ReLU]** Implement ReLU activation in ‘Answer.py’ (‘ReLU’).
4. **[Convolution Layer]** Implement a convolution layer in ‘Answer.py’ (‘ConvolutionLayer’).
5. **[Max-Pooling Layer]** Implement a max-pooling layer in ‘Answer.py’ (‘MaxPoolingLayer’).
6. **[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

1. **[20 Pts]** Experiment results
2. 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).**

**A close up of a map

Description automatically generated**