exam 개인학습

Lec 2.4

AI summary Try on this page

Type something...

01) Convolutional Neural Networks

- 지금까지는 fully connected networks로 학습을 해왔었다.
- 지금부터는 CNN을 사용하여 CIFAR-10 데이터셋 학습을 진행할 예정이다.
- 주의할 점은 항상 데이터와 filter의 channel개수가 같아야 한다.

01_conv_forward_navie(x, w, b, conv_param)

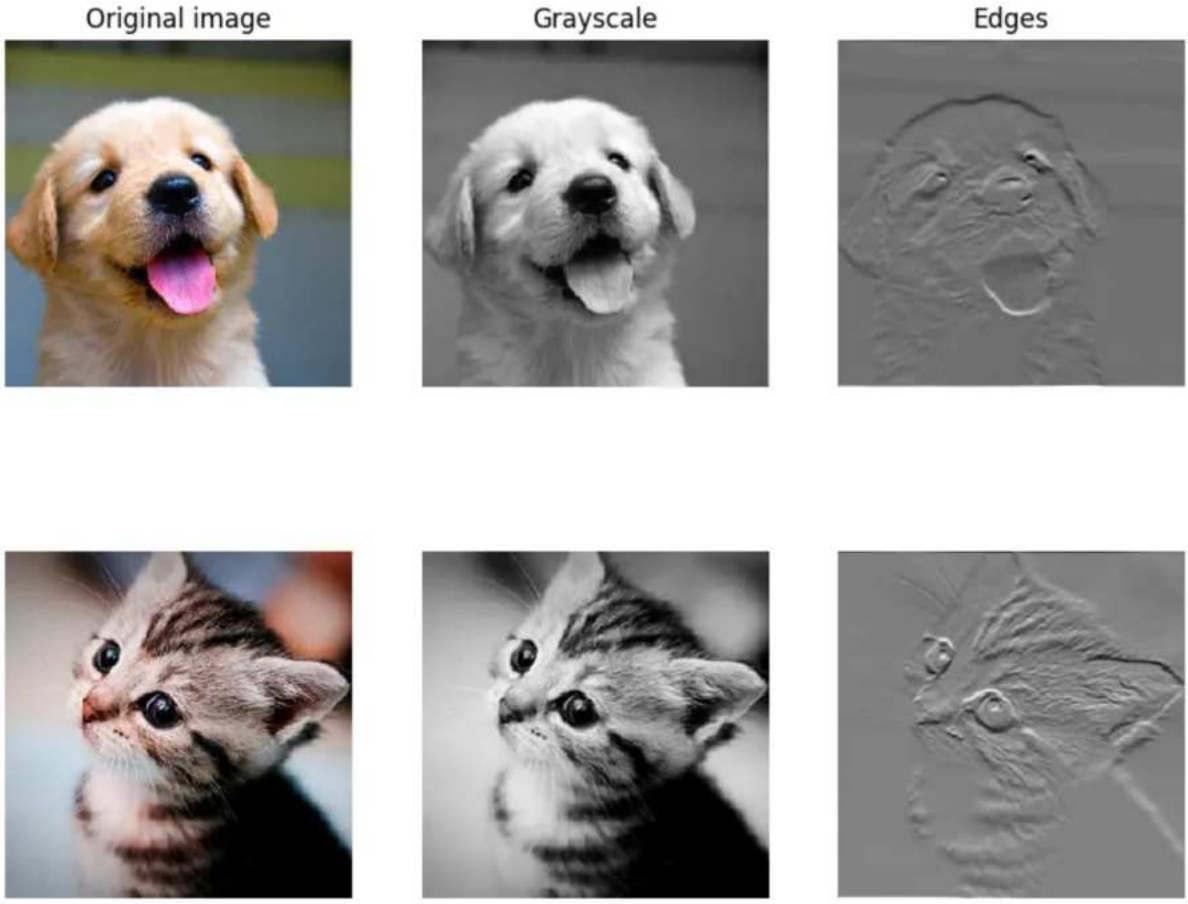
- (C, H, W). 세트의 출력은 C개 세트에서 나온 C개 세트의 결과를 다 더해주므로 output의 차원은 2차원이다 (H,W).

```
stride = conv_param['stride']
pad = conv_param['pad']
N = x.shape[0]
H = x.shape[2]
W = x.shape[3]

F = w.shape[0]
HH = w.shape[2]
WW = w.shape[3]

H_hat = int(1 + (H + 2 * pad - HH) / stride) # float -> int
W_hat = int(1 + (W + 2 * pad - WW) / stride)
out_shape = (N, F, H_hat, W_hat)
out = np.zeros(out_shape)

x_pad = np.pad(x, pad_width = [(0, 0), (0, 0), (pad, pad), (pad, pad)]) # 높이와 너비에만 padding
for n in range(N):
  for f in range(F):
    for i in range(H_hat):
      for j in range(W_hat):
        h_start = i * stride
        w_start = j * stride
        out[n][f][i][j] = np.sum(x_pad[n,:,h_start:h_start+HH, w_start:w_start+WW] * w[f,:,:,:]) + b[f]
```

- 먼저 out의 `shape`를 구한 후 `array`를 하나 만들어준다.
- 그 후 N → F 순서대로 반복문을 진행한다.
- filter 수만큼 2차원 행렬들의 곱을 out에다가 반영해준다.
- (C, H, W) * (C, HH, WW)를 계속 반복해주는 형태이다.
- `padding`은 `height`와 `width`에다만 반영해준다.

```
# padding all input data
x_pad = np.pad(x, ((0,0), (0,0),(pad,pad),(pad,pad)), 'constant')
H_pad, W_pad = x_pad.shape[2], x_pad.shape[3]

# create w_row matrix
w_row = w.reshape(F, C*FH*FW)                          #[F x C*FH*FW]

# create x_col matrix with values that each neuron is connected to
x_col = np.zeros((C*FH*FW, outH*outW))                 #[C*FH*FW x H'*W']
for index in range(N):
    neuron = 0
    for i in range(0, H_pad-FH+1, stride):
        for j in range(0, W_pad-FW+1,stride):
            x_col[:,neuron] = x_pad[index,:,i:i+FH,j:j+FW].reshape(C*FH*FW)
            neuron += 1
    out[index] = (w_row.dot(x_col) + b.reshape(F,1)).reshape(F, outH, outW)
```

- w들 row, x들 col로 `reshape` 해서 dot하면 convolution과 똑같은 결과들 만들 수 있다.



- convolution을 통해서 edge를 시각화할 수 있게 되었다.

02_conv_backward_naive(dout, cache)

```
x_pad = np.pad(x, pad_width = [(0, 0), (0, 0), (pad, pad), (pad, pad)]) # 높이와 너비에만 padding
dw = np.zeros_like(w)
dx = np.zeros_like(x_pad)
db = np.zeros_like(b)


for n in range(N):
  for f in range(F):
    for i in range(H_hat):
      for j in range(W_hat):
        h_start = i * stride
        w_start = j * stride
        db[f] += dout[n][f][i][j]
        dw[f,:,:,:] += dout[n][f][i][j] * x_pad[n,:,h_start:h_start+HH, w_start:w_start+WW]
        dx[n,:,h_start:h_start+HH, w_start:w_start+WW] += dout[n][f][i][j] * w[f,:,:,:]
        # dx : x_pad 모양으로 만들고 pad 빼주기

#print(dx.shape) # 4 x 3 x 7 x 7
dx = dx[:, :, pad:-pad, pad:-pad]
```

- dx를 x_pad모양으로 만들어서 `backpropagation`을 진행한다.
- **마지막에 padding 부분을 슬라이싱 해서 빼준다.**

03_Max_pool_forward_naive(x, pool_param)

```
pool_height = pool_param['pool_height']
pool_width = pool_param['pool_width']
stride = pool_param['stride']
N = x.shape[0]
C = x.shape[1]
H = x.shape[2]
W = x.shape[3]
H_hat = int(1 + (H - pool_height) / stride)
W_hat = int(1 + (W - pool_width) / stride)

out_shape = (N, C, H_hat, W_hat)
out = np.zeros(out_shape)

for n in range(N):
  for c in range(C):
    for i in range(H_hat):
      for j in range(W_hat):
        h_start = i * stride
        w_start = j * stride
        max_value = np.max(x[n, c, h_start:h_start+pool_height, w_start:w_start + pool_width])
        out[n, c, i, j] = max_value
```

- conv과 마찬가지로 슬라이싱을 이용하여 만들었다.
- 특정 구역의 max값을 반환해서 그 값을 out에 저장해주었다.

04_Max_pool_backward_naive(dout, cache)

- max 값에 해당하는 인덱스를 구한 후 dout을 업데이트 해준다.
- 나머지 부분은 0이므로 그냥 냅둔다.

```
H_hat = int(1 + (H - pool_height) / stride)
W_hat = int(1 + (W - pool_width) / stride)

dx = np.zeros_like(x)

for n in range(N):
  for c in range(C):
    for i in range(H_hat):
      for j in range(W_hat):
        h_start = i * stride
        w_start = j * stride
        max_idx = np.argmax(x[n, c, h_start:h_start+pool_height, w_start:w_start + pool_width])
        h_idx = max_idx // pool_height
        w_idx = max_idx % pool_width
        dx[n, c, h_start + h_idx, w_start + w_idx] += dout[n, c, i, j]
```

- 몫과 나머지를 이용해서 실제 위치를 구해주었다.

- `fast_layers` 에서 최적화로 구현해놓은 함수들이 있다.

```
Testing conv_forward_fast:
Naive: 6.382780s
Fast: 0.020139s
Speedup: 316.929112x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 9.284262s
Fast: 0.022117s
Speedup: 419.781362x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  3.481354613192702e-14
```

- 속도가 400배 정도 차이나는 것을 확인할 수 있다.

05_ThreeLayerConvNet

- `Conv - relu - 2x2 max pool - affine - relu - affine - softmax` 로 구성되어 있는 3 Layer Convoluion network를 구축할 것이다.

```
# conv_param = {"stride": 1, "pad": (filter_size - 1) // 2}
# pool_param = {"pool_height": 2, "pool_width": 2, "stride": 2}
self.params['W1'] = np.random.randn(num_filters, input_dim[0], filter_size, filter_size) * weight_scale
self.params['b1'] = np.zeros([num_filters, ])

conv_H = 1 + (input_dim[1] + 2 * (filter_size - 1) // 2 - filter_size) // 1
pool_H = 1 + (conv_H - 2) // 2
pool_W = pool_H

self.params['W2'] = np.random.randn(num_filters * pool_H * pool_W, hidden_dim) * weight_scale
self.parmas['b2'] = np.zeros([hidden_dim, ])

self.params['W3'] = np.random.randn(hidden_dim, num_classes) * weight_scale
self.parmas['b3'] = np.zeros([num_classes, ])
```

- 주어진 stride와 padding을 가지고 초기화를 진행하였다.
- `weight_scale` 을 까먹지 말자.

```
# forward part
out1, cache1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
out2, cache2 = affine_relu_forward(out1, W2, b2)
scores, cache3 = affine_forward(out2, W3, b3)
```

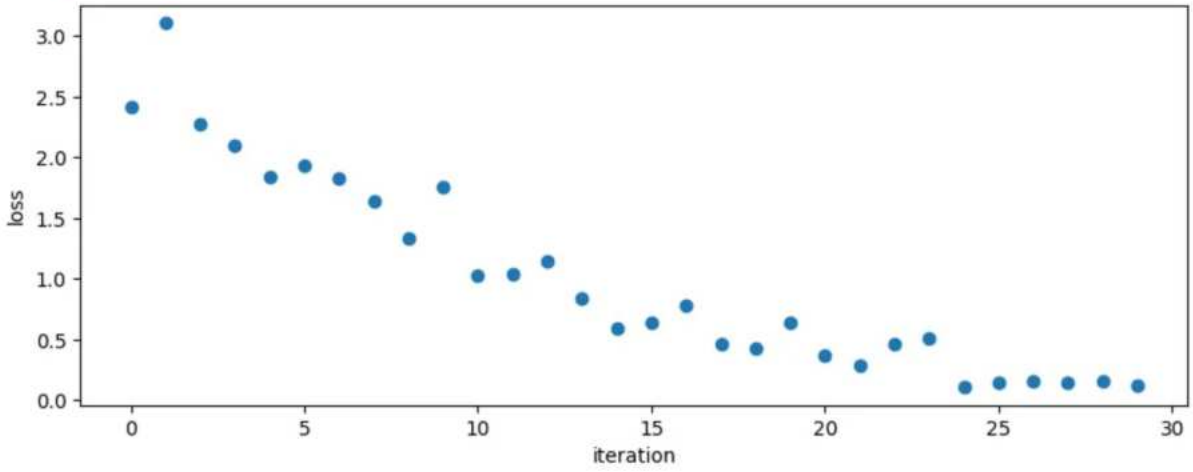
- forward 부분은 이전과 동일하다.
- 순서만 헷갈리지 말자.

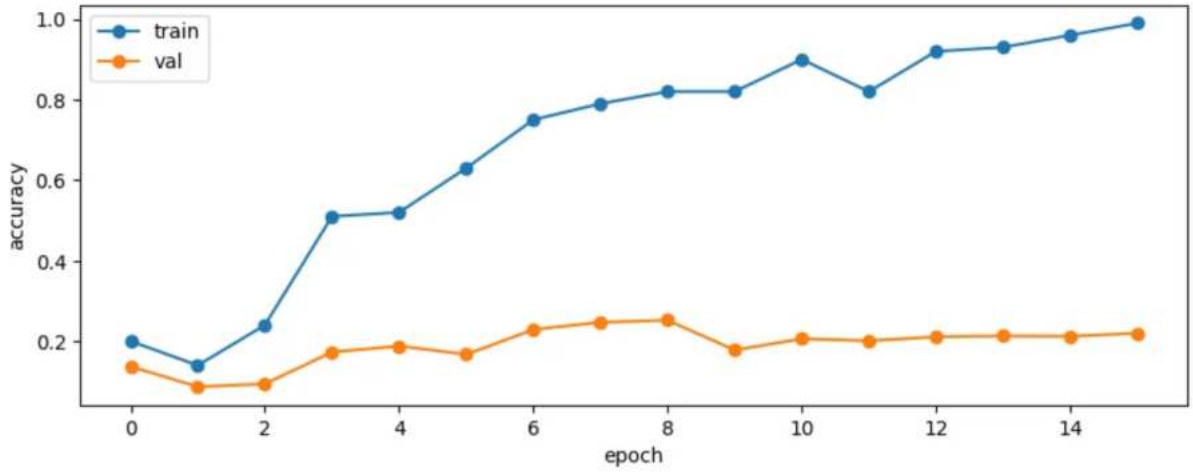
```
loss, dx = softmax_loss(scores, y)

dx3, dw3, db3 = affine_backward(dx, cache3)
dw3 += self.reg * W3
grads['W3'] = dw3
grads['b3'] = db3

dx2, dw2, db2 = affine_relu_backward(dx3, cache2)
dw2 += self.reg * W2
grads['W2'] = dw2
grads['b2'] = db2

dx1, dw1, db1 = conv_relu_pool_backward(dx2, cache1)
dw1 += self.reg * W1
grads['W1'] = dw1
grads['b1'] = db1

loss += self.reg * 0.5 * np.sum(np.square(W1))
loss += self.reg * 0.5 * np.sum(np.square(W2))
loss += self.reg * 0.5 * np.sum(np.square(W3))
```

- backward도 이전과 동일하게 작성하였다.
- Regularization 부분만 까먹지 말자.

1. Overfit small Data

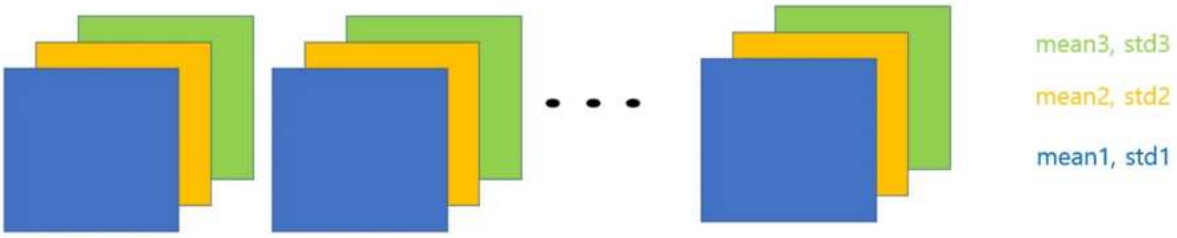
- 모델을 잘 만들었는지 확인하기 위해서는 작은 데이터를 `overfit` 시키는 것이다.
- 그 결과 높은 `training accuracy` 와 비교적 낮은 `validation accuracy` 을 얻을 수 있다.





- One epoch만 진행해서 40% 이상 정확도를 가지는 모델을 만들어보자.

06_spatial_batchnorm_forward(x, gamma, beta, bn_param)



- channel 별로 통계를 내는 이유는 같은 channel이 같은 weight(filter)의 영향을 받았기 때문이다.
- 기존 bn은 `(N, D)` 2차원에서 다뤘지만, convolution layer에서 사용하기 위해서는 `(N, C, H, W)` 4차원에서 bn을 사용해야 한다.
- train과 test로 나누어서 진행한다. 차원을 잘 고려해서 코드를 작성하자.
- `(N, C, H, W)` → transpose → `(C, N, H, W)` → `(N * H * W, C)` 순서로 변환하자.
- feature들로 bn을 구하기 때문에 C를 마지막에 둔다.

```
C = x.shape[1] # channel (N, C, H, W)

out, cache = None, None
x_T = np.reshape(x,[-1, C])
out, cache = batchnorm_forward(x_T, gamma, beta, bn_param)
out = np.reshape(out, x.shape)
```

- Channel 이용해서 shape를 바꿔준다.
- bn은 2차원에서 진행되기 때문에 나온 out값을 다시 4차원을 바꿔준다.

07_spatial_batchnorm_backward(dout, cache)

```
C = dout.shape[1]
d_out = np.zeros_like(dout)
d_out = np.reshape(dout, [-1, C])
dx, dgamma, dbeta = batchnorm_backward(d_out, cache)
dx = np.reshape(dx, dout.shape)
```

- forward와 마찬가지로 모양만 잘 맞춰주면 된다.

08_spatial_groupnorm_forward(x, gamma, beta, G, gn_param)

- batch normalization 의 대체로 Layer normalization을 소개했었다.
- 하지만 LN도 convolutional layer에서는 잘 작동하지 않았다.
- groupnorm은 을 단위로 하는데 G의 개수만큼 쪼개서 진행한다.
- GN은 하나의 N에 대하여 채널을 그룹 단위로 normalization을 한다. 만약에 채널이 6개있고, G=2이면 한 그룹당 3개의 채널이 존재한다.
- LN과 매우 유사하다.
- `[N, C, H, W]` → `[N, G, C // G, H, W]`
- ex) (2, 6, 4, 5) → (2, 2, 3, 4, 5) (G = 2)

```
N, C, H, W = x.shape
x = np.reshape(x, [N, G, C // G, H, W]) #(2, 2, 3, 4, 5)

mean = np.mean(x, axis = (2, 3, 4), keepdims = True) # (2, 2, 1, 1, 1)
# G를 기점으로 normalize를 해준다.
var = np.var(x , axis = (2, 3, 4), keepdims = True)# (2, 2, 1, 1, 1)
x_normal = (x - mean) / np.sqrt(var + eps)
x_normal = np.reshape(x_normal, [N, C, H, W])
out = gamma * x_normal + beta
cache = (x, x_normal, mean, var, gamma, G)
```

- G를 기점으로 normalize를 해주기 때문에 `axis = (2, 3, 4)` 를 대상으로 평균과 분산을 구해주었다.

In **Batch Norm** [26], the set $\mathcal{S}_i$ is defined as:

$$\mathcal{S}_i = \{k \mid k_C = i_C\}, \tag{3}$$

where i_C (and k_C) denotes the sub-index of i (and k) along the C axis. This means that the pixels sharing the same channel index are normalized together, *i.e.*, for each channel, BN computes μ and σ along the (N, H, W) axes. In **Layer Norm** [3], the set is:

$$\mathcal{S}_i = \{k \mid k_N = i_N\}, \tag{4}$$

meaning that LN computes μ and σ along the (C, H, W) axes for each sample. In **Instance Norm** [61], the set is:

$$\mathcal{S}_i = \{k \mid k_N = i_N, k_C = i_C\}. \tag{5}$$

meaning that IN computes μ and σ along the (H, W) axes for each sample and each channel. The relations among BN, LN, and IN are in Figure 2.

Group Norm. Formally, a Group Norm layer computes μ and σ in a set $\mathcal{S}_i$ defined as:

$$\mathcal{S}_i = \{k \mid k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor\}. \tag{7}$$

Here G is the number of groups, which is a pre-defined hyper-parameter ($G = 32$ by default). C/G is the number of channels per group. $\lfloor \cdot \rfloor$ is the floor operation, and "$\lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor$" means that the indexes i and k are in the same group of channels, assuming each group of channels are stored in a sequential order along the C axis. GN computes μ and σ along the (H, W) axes and along a group of $\frac{C}{G}$ channels. The computation of GN is illustrated in Figure 2 (rightmost), which is a simple case of 2 groups ($G = 2$) each having 3 channels.

Given $\mathcal{S}_i$ in Eqn.(7), a GN layer is defined by Eqn.(1), (2), and (6). Specifically, the pixels in the same group are normalized together by the same μ and σ. GN also learns the per-channel γ and β.

09_spatial_groupnorm_backward(dout, cache)

- 평균과 분산은 (H,W) axis와 C/G axis에 대해서 구한다.

```
eps = 1e-5
x, x_normal, feature_mean, feature_var, gamma, G = cache
N, C, H, W = dout.shape
M = x.shape[2] * x.shape[3] * x.shape[4]

x_normal = (x - feature_mean) / np.sqrt(feature_var + eps)
dgamma = np.sum(dout * np.reshape(x_normal, dout.shape), axis = (0, 2, 3), keepdims = True ).reshape(1, C, 1, 1) # channel 제외 나머
dbeta = np.sum(dout, axis = (0, 2, 3), keepdims = True).reshape(1, C, 1, 1)
dx_normal = dout * gamma

dlvar = np.sum(np.reshape(dx_normal, x.shape) * (x - feature_mean)
* -0.5 * (feature_var + eps)**-1.5, axis = (2, 3, 4), keepdims = True)

dlmean = np.sum(np.reshape(dx_normal, x.shape) * -1 / np.sqrt(feature_var + eps) , axis = (2, 3, 4), keepdims = True)
+ dlvar * np.sum(-2 * (x - feature_mean), axis = (2, 3, 4), keepdims = True) / M

dx = np.reshape(dx_normal, x.shape) * 1 / np.sqrt(feature_var + eps) + dlvar * 2 * (x - feature_mean) / M + dlmean / M
dx = np.reshape(dx, dout.shape)
```

- `dgamma` , `dbeta` 모두 channel를 기준으로 만들기 때문에 나머지 축은 합하는데 사용되었다.
- `dx_normal` 이 4차원이기 때문에 x와 shape을 맞춰주고 연산을 한다.
- 앞에서와 마찬가지로 `axis = (2, 3, 4)` 로 합을 한다.

- 논문을 읽고 따라가야 하다보니 상당한 난이도를 요구하였다.
- 나중에 다시 천천히 또 정리해도 배울 게 있을거라는 생각이 들었다.