exam 개인학습

Lec 2.1

AI summary Try on this page

Type something...

01) Multi-Layer Fully Connected Neural Networks

- 이번 과제에서는 임의의 깊이인 hidden layer를 가지는 fully-connected network를 만들 것이다.
- FullyConnected Layer parameter 초기화와 forward/backward 구현 부분을 보겠다.
- 2-layer net의 확장판이지만 `batchnorm` 과 `dropout` , `layernorm` 등을 같이 넣어야 되어서 쉽지 않았다.

01_ __init__

```
        tmp_dim=input_dim
        for i in range(self.num_layers):
          if i==self.num_layers-1: # 마지막 layer만 구분.
            self.params['W'+str(i+1)]=np.random.randn(tmp_dim,num_classes)*weight_scale
            self.params['b'+str(i+1)]=np.zeros(num_classes)

          else:
            if normalization=='batchnorm':
              self.params['W'+str(i+1)]=np.random.randn(tmp_dim,hidden_dims[i])*weight_scale
              self.params['b'+str(i+1)]=np.zeros(hidden_dims[i])
              self.params['gamma'+str(i+1)]=np.ones(hidden_dims[i])
              self.params['beta'+str(i+1)]=np.zeros(hidden_dims[i])
              tmp_dim=hidden_dims[i]

            elif normalization=='layernorm':
              self.params['W'+str(i+1)]=np.random.randn(tmp_dim,hidden_dims[i])*weight_scale
              self.params['b'+str(i+1)]=np.zeros(hidden_dims[i])
              self.params['gamma'+str(i+1)]=np.ones(hidden_dims[i])
              self.params['beta'+str(i+1)]=np.zeros(hidden_dims[i])
              tmp_dim=hidden_dims[i]

            else:
              self.params['W'+str(i+1)]=np.random.randn(tmp_dim,hidden_dims[i])*weight_scale
              self.params['b'+str(i+1)]=np.zeros(hidden_dims[i])
              tmp_dim=hidden_dims[i]
```

- 마지막 layer는 `affine` 만 구현해준다.
- 나머지는 batchnorm, layernorm, 일반 3가지 경우로 구분해서 구현해준다.
- `weight_scale` 을 곱해줘서 weight를 초기화 시켜주었다.
- `batchnorm` 의 경우에는 추가적으로 `gamma` , `beta` 도 각각 1과 0으로 초기화 시켜주었다.

02_loss(self, X, y = None)

```
cache={}
do_cache={}
out=X
for i in range(self.num_layers):
  W=self.params['W'+str(i+1)]
  b=self.params['b'+str(i+1)]

  if i == self.num_layers-1: # 마지막 layer는 linear만
    out, cache[i] = affine_forward(out,W,b)

  else:
    if self.normalization == 'batchnorm':
      gamma=self.params['gamma'+str(i+1)]
      beta=self.params['beta'+str(i+1)]
      bn=self.bn_params[i]
      out,cache[i]=affine_batchnorm_forward(out,W,b,gamma,beta,bn)

    elif self.normalization=="layernorm":
      gamma=self.params['gamma'+str(i+1)]
      beta=self.params['beta'+str(i+1)]
      bn=self.bn_params[i]
      out,cache[i]=affine_layernorm_forward(out,W,b,gamma,beta,bn)

    else:
      out, cache[i] = affine_relu_forward(out,W,b)

    if self.use_dropout:
      out, do_cache[i] = dropout_forward(out, self.dropout_param)
```

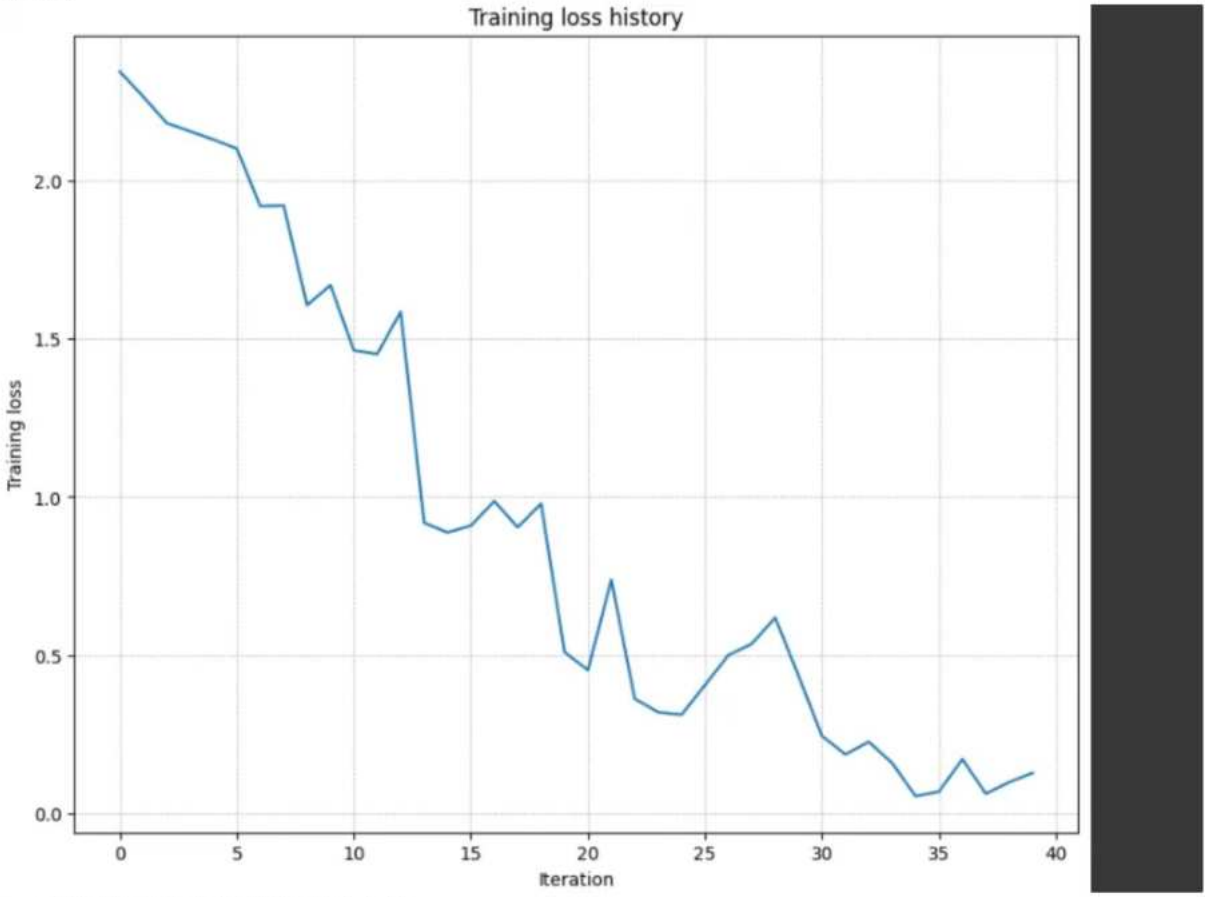
- `dropout_cache` = `do_cache`, 나머지 데이터 = `cache` 에 배열로 저장해준다.
- 마지막 layer만 구분하였고, `affine_batchnorm_forward` 와 `affine_relu_forward` 를 사용해주었다.

```
weight_sum_squared = 0
for i in range(self.num_layers):
  weight_sum_squared += np.sum(np.square(self.params['W' + str(i + 1)])) # weight regularization

loss, dx = softmax_loss(scores, y)
loss += self.reg * 0.5 * weight_sum_squared

for i in reversed(range(self.num_layers)): # backpropagation 진행
  if i == self.num_layers - 1: #마지막 layer이면
    dx, dW, db = affine_backward(dx, cache[i])
    grads['W' + str(i + 1)] = dW + self.reg * self.params['W' + str(i + 1)] # 2 * reg * W
    grads['b' + str(i + 1)] = db
  else:
    if self.use_dropout:
      dx = dropout_backward(dx, do_cache[i])

    if self.normalization == 'batchnorm':
      dx, dW, db, dgamma, dbeta = affine_batchnorm_backward(dx, cache[i])
      grads['W' + str(i + 1)] = dW + self.reg * self.params['W' + str(i + 1)] # 2 * reg * W
      grads['b' + str(i + 1)] = db
      grads['gamma' + str(i + 1)] = dgamma
      grads['beta' + str(i + 1)] = dbeta
```

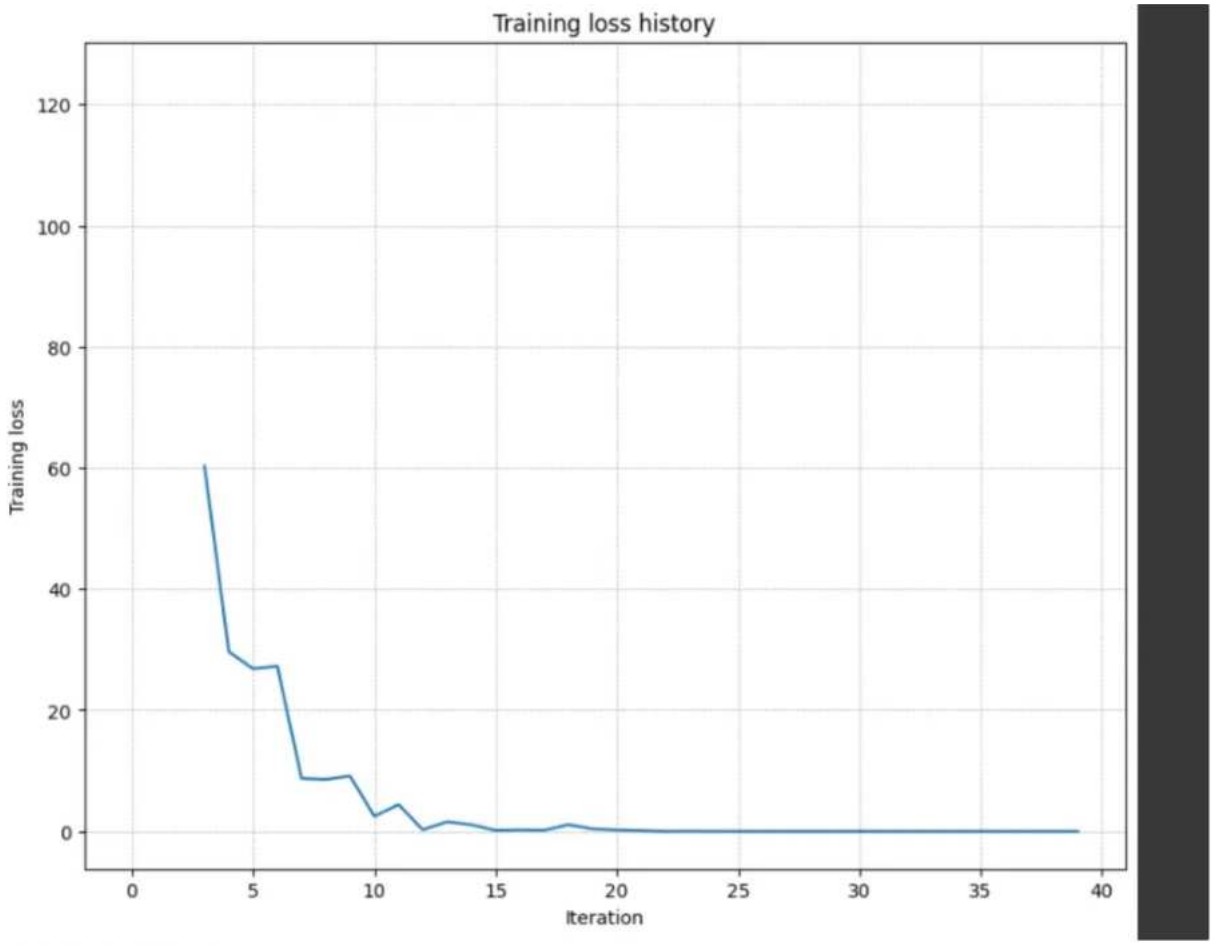
- backpropagation을 진행하기 위해서 `reversed` 를 사용해주었다.
- forward 반대로 backward를 사용해서 grads에 저장해주었다.

```
elif self.normalization == 'layernorm':
  dx, dW, db, dgamma, dbeta = affine_layernorm_backward(dx, cache[i])
  grads['W' + str(i + 1)] = dW + self.reg * self.params['W' + str(i + 1)] # 2 * reg * W
  grads['b' + str(i + 1)] = db
  grads['gamma' + str(i + 1)] = dgamma
  grads['beta' + str(i + 1)] = dbeta

else:
  dx, dW, db = affine_relu_backward(dx, cache[i])
  grads['W' + str(i + 1)] = dW + self.reg * self.params['W' + str(i + 1)] # 2 * reg * W
  grads['b' + str(i + 1)] = db
```

- 밑에 부분도 마찬가지다.



- 3-Layer net를 overfit 시켜보았다. 코드가 잘 작동하는 것을 확인할 수 있다.



- 4-Layer net도 잘 작동하는 것을 확인할 수 있다.

1. Inline Question 1

Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

A: five-layer network가 initialization scale에 더 민감할 것 같다. 더 깊은 네트워크이기 때문이다.

- 지금까지는 vanila SGD만 사용을 했었다.
- 더 정교한 업데이트를 위해서 다른 경사하강법 방식도 구현해보자.

03_sgd_momentum(w, dw, config = None)

$$v_{t+1} = \rho v_t - \alpha \partial f(x_t)$$

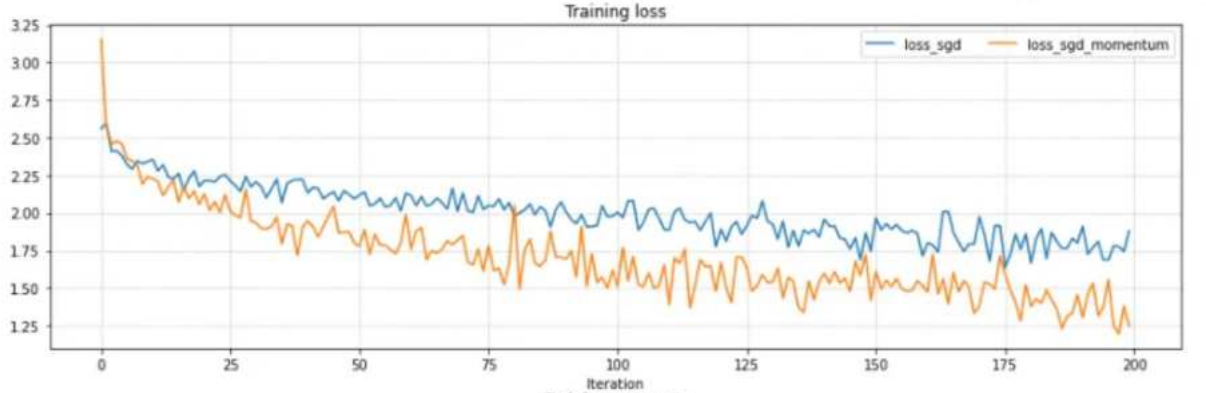
$$x_{t+1} = x_t + v_{t+1}$$

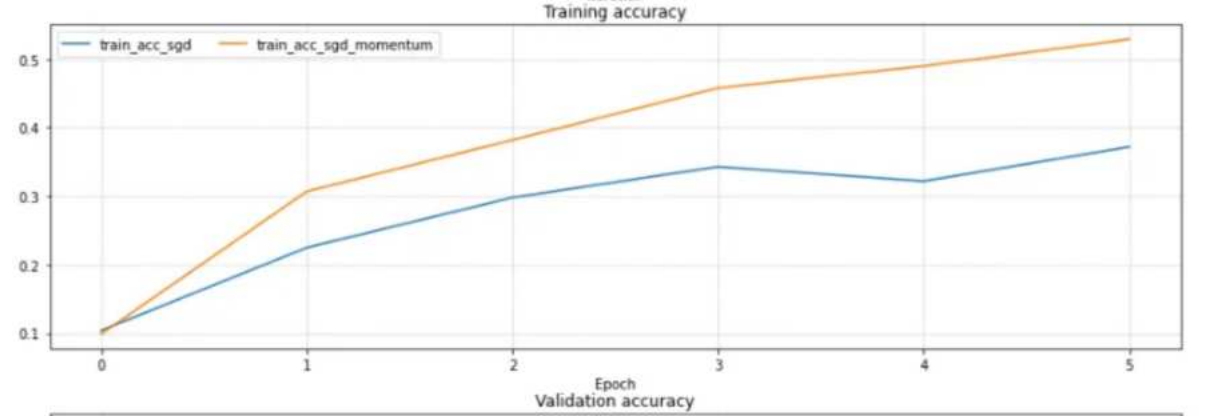
- 2가지 수식이 있는데 위의 수식을 사용하였다.
- 코드는 다음과 같다.

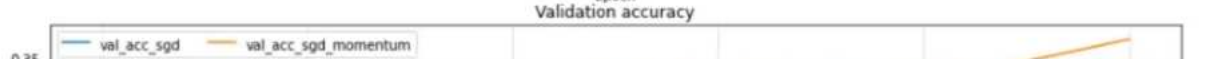
```python
def sgd_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with momentum.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - velocity: A numpy array of the same shape as w and dw used to store a
      moving average of the gradients.
    """
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("momentum", 0.9)
    v = config.get("velocity", np.zeros_like(w))

    next_w = None
    ###########################################################################
    # TODO: Implement the momentum update formula. Store the updated value in #
    # the next_w variable. You should also use and update the velocity v.     #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    ''' update w -> next_w, v -> v'''
    # rho: momentum
    # alpha: learning rate
    learning_rate = config['learning_rate']
    momentum = config['momentum']
    v = momentum * v - learning_rate * dw
    next_w = w + v
```

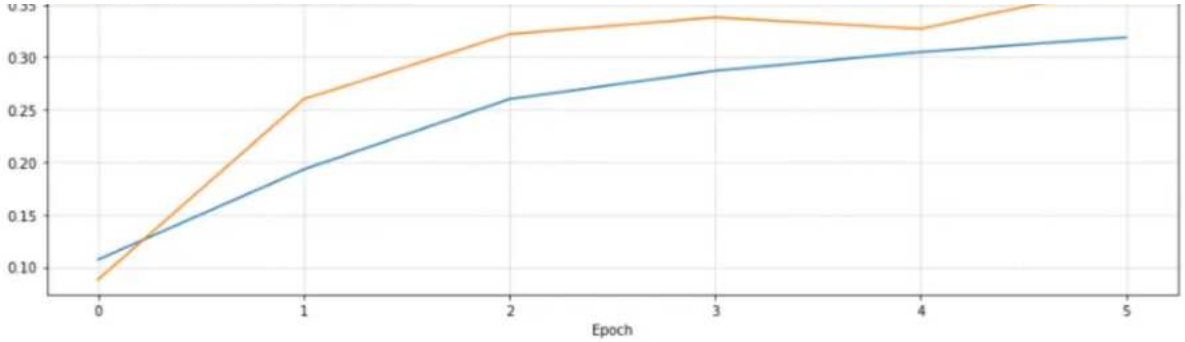
- momentum + SGD를 사용하니 일반 SGD 보다 수렴 속도가 더 빠른 것을 확인할 수 있었다.

```
(Epoch 5 / 5) train acc: 0.529000; val_acc: 0.369000
```









- 또한 정확도도 높이는데 도움을 주었다.

04_rmsprop(w, dw, config = None):

RMSprop Algorithm

$$s_{dw} = \beta_2 * s_{dw} + (1 - \beta_2) * dw^2$$
$$s_{db} = \beta_2 * s_{db} + (1 - \beta_2) * db^2$$

$$w := w - \alpha * \frac{dw}{(\sqrt{s_{dw}} + \varepsilon)}$$
$$b := b - \alpha * \frac{dw}{(\sqrt{s_{db}} + \varepsilon)}$$

- `decay_rate` , `eplison` , `learning_rate` 를 이용해서 gradient를 업데이트 한다.
- 최근 곡면의 변화량을 반영해주기 위하여 루트를 씌우고 나눠준다.
- momentum보다 개선된 점은 최근 step의 값을 많이 반영해준다는 점이다.
- `config[cache]` 에 이전 `grad_squared` 를 저장해줘서 업데이트를 진행한다.
- 코드로 작성하면 다음과 같다.

```
def rmsprop(w, dw, config=None):
    """
    Uses the RMSProp update rule, which uses a moving average of squared
    gradient values to set adaptive per-parameter learning rates.

    config format:
    - learning_rate: Scalar learning rate.
    - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
      gradient cache.
    - epsilon: Small scalar used for smoothing to avoid dividing by zero.
    - cache: Moving average of second moments of gradients.
    """
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("decay_rate", 0.99)
    config.setdefault("epsilon", 1e-8)
    config.setdefault("cache", np.zeros_like(w))

    next_w = None
    ###########################################################################
    # TODO: Implement the RMSprop update formula, storing the next value of w #
    # in the next_w variable. Don't forget to update cache value stored in    #
    # config['cache'].                                                        #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    learning_rate = config['learning_rate']
    decay_rate = config['decay_rate']
    eplison = config['epsilon']
    grad_squared = config['cache']
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * (dw ** 2)
```

```
    next_w = w - learning_rate * dw / (np.sqrt(grad_squared) + epsilon)
```

05_adam(W, dw, config = None)

- Adam은 momentum 과 rmsprop의 장점을 결합한 방식이다.

수식과 함께 Adam에 대해 자세히 알아보겠습니다.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_{t-1})$$

$$g_t = \beta_2 g_{t-1} + (1-\beta_2)(\nabla f(x_{t-1}))^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{g}_t = \frac{g_t}{1-\beta_2^t}$$

$$x_t = x_{t-1} - \frac{\eta}{\sqrt{\hat{g}_t + \epsilon}} \cdot \hat{m}_t$$

- β_1: Momentum의 지수이동평균 ≈ 0.9
- β_2: RMSProp의 지수이동평균 ≈ 0.999
- $\hat{m}, \hat{g}$: 학습 초기 시 m_t, g_t가 0이 되는 것을 방지하기 위한 보정 값
- ϵ: 분모가 0이 되는 것을 방지하기 위한 작은 값 $\approx 10^{-8}$
- η: 학습률 ≈ 0.001

- 코드로 구현하면 다음과 같다.
- gradient descent는 추후의 논문과 함께 다시 정리해보는 시간을 가지도록 하겠다.

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSPro

Bias correction for the fact that
first and second moment
estimates start at zero

- bias correction은 지수 가중 평균을 이용한 추정은 초기 구간에 오차가 있기 때문에 이를 보정하기 위해 사용된다.

- t값은 +1씩 해서 증가시킨다.
- 나머지는 값을 저장하면서 업데이트를 진행해주었다.

2. Inline Question 2

Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

Answer:

[FILL THIS IN]

- 나눠지는 값이 더 커지게 되어 update가 0으로 수렴할 것이다.

06_Train a Good Model

- 지금까지 배웠던 모든 hyperparamter들을 활용하여 최고의 모델을 구하는 과정이다.
- 50% 이상의 정확도를 내는 것을 목표로 코드를 구현해보자.
- 앞선 코드들과 유사하지만 hyperparameter들이 많아지면서 for문의 깊이가 조금 더 깊어졌다.

```
best_acc = -1

learning_rate = [1e-3, 1e-4]
weight_scale = [5e-2]
batch_size = [50, 100]
for lr in learning_rate:
  for w in weight_scale:
    for bs in batch_size:
      for drop_p in dropout_keep_ratio:
          model = FullyConnectedNet(hidden_dims = [100, 100, 100, 100, 100],
                                    weight_scale = w)
          solver = Solver(model, data, num_epochs = 10, batch_size = bs, update_rule = 'adam', optim_config = {'learning_rate': lr},
                          verbose = False)
          solver.train()
          acc = solver.check_accuracy(data['X_val'], data['y_val'], batch_size = 100)
          print('learning_rates:',lr,'weight_scale:',w, 'batch_size:', bs, 'dropout_ratio:', drop_p, 'batch_norm:', bn, 'acc:',acc)

          if acc > best_acc:
            best_acc = acc
            best_model = model

# ****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
learning_rates: 0.001 weight_scale: 0.05 batch_size: 50 dropout_ratio: 1 batch_norm: batchnorm acc: 0.508
learning_rates: 0.001 weight_scale: 0.05 batch_size: 50 dropout_ratio: 0.5 batch_norm: batchnorm acc: 0.51
learning_rates: 0.001 weight_scale: 0.05 batch_size: 100 dropout_ratio: 1 batch_norm: batchnorm acc: 0.526
learning_rates: 0.001 weight_scale: 0.05 batch_size: 100 dropout_ratio: 0.5 batch_norm: batchnorm acc: 0.517
learning_rates: 0.0001 weight_scale: 0.05 batch_size: 50 dropout_ratio: 1 batch_norm: batchnorm acc: 0.512
learning_rates: 0.0001 weight_scale: 0.05 batch_size: 50 dropout_ratio: 0.5 batch_norm: batchnorm acc: 0.51
learning_rates: 0.0001 weight_scale: 0.05 batch_size: 100 dropout_ratio: 1 batch_norm: batchnorm acc: 0.496
learning_rates: 0.0001 weight_scale: 0.05 batch_size: 100 dropout_ratio: 0.5 batch_norm: batchnorm acc: 0.502
```

- batch normalization 과 dropout을 구현해서 추후에 성능을 더 높여보자.