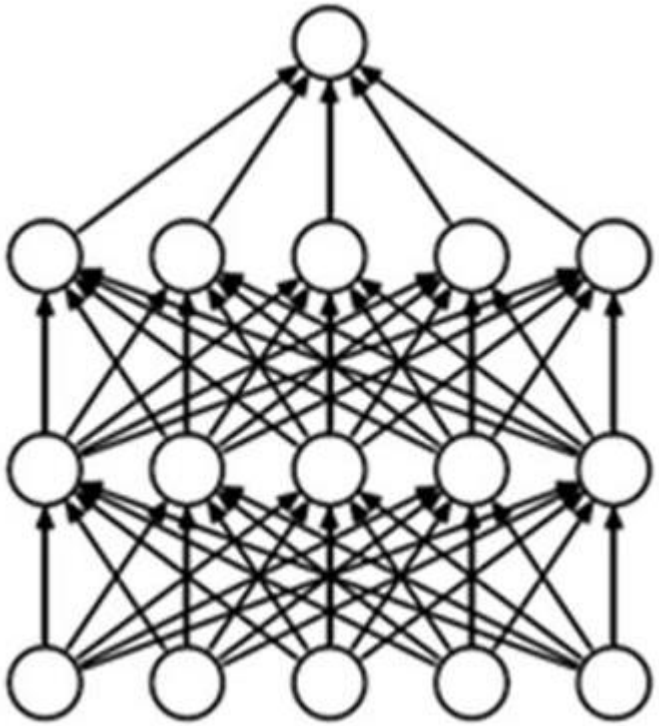
Dropout
&
Batch-Normalization

Contents

- 1. Dropout
- 2. Batch-Normalization

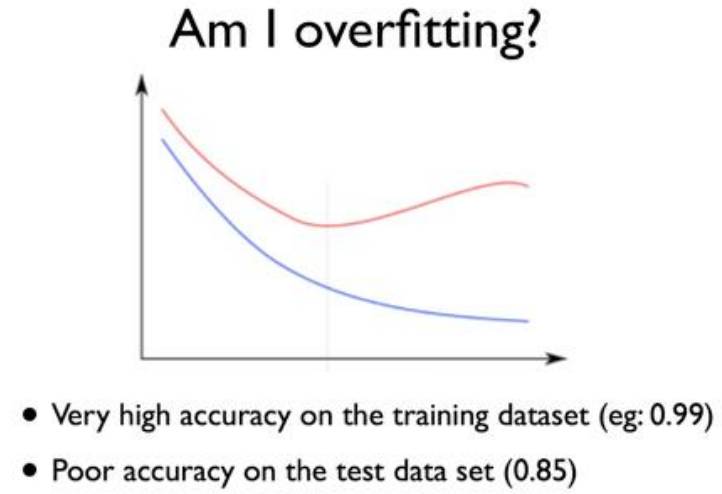
Dropout

Why?

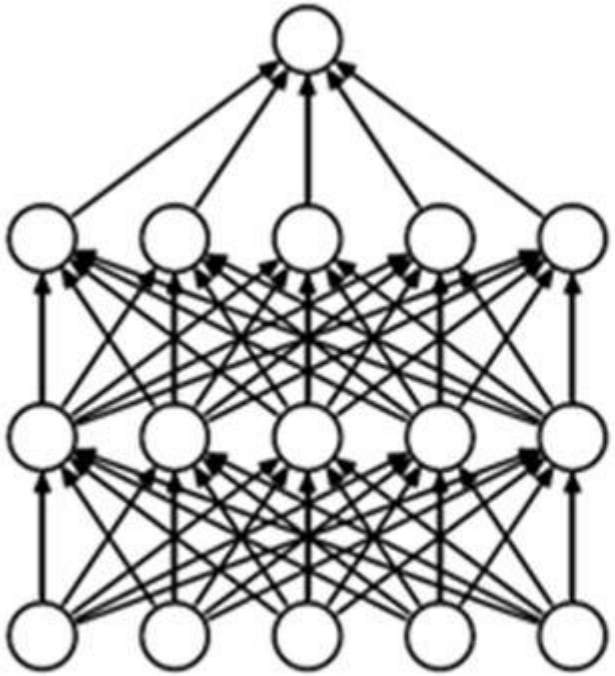


DNN
-> Training data가 많이 필요, 학습 시간이 길어짐, overfitting을 피하기 힘듦.

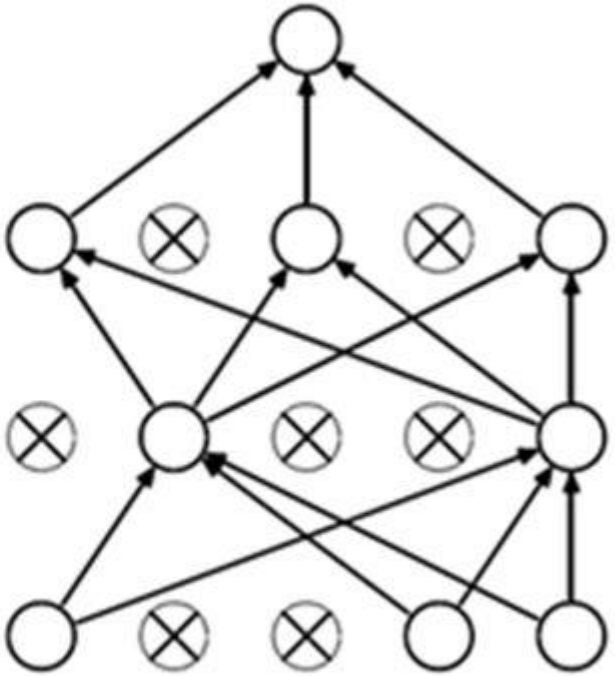
NN의 고질적인 문제인 over-fitting 문제를 완화시킬 수 있는 방법 중 하나.



Dropout



(a) Standard Neural Net



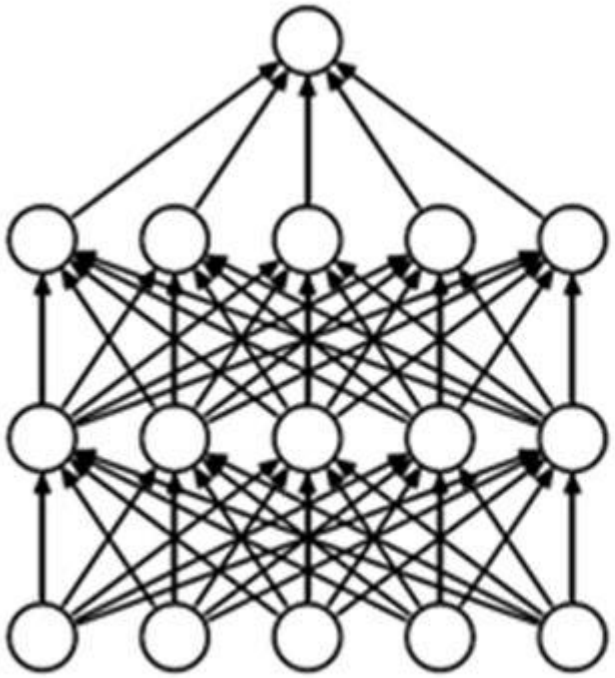
(b) After applying dropout.

layer에 포함된 weight 중에서 일부만 참여시킴.

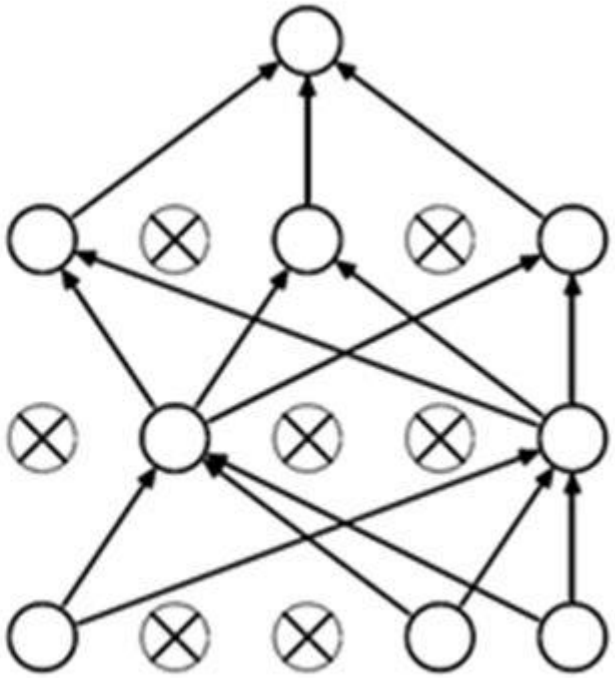
난수를 사용해서 일부 뉴런을 0으로 만드는 방법.

일정한 mini-batch 구간 동안 dropout된 망에 대한 학습을 끝내면, 다시 무작위로 다른 뉴런들을 dropout하면서 반복적으로 학습.

Dropout



(a) Standard Neural Net



(b) After applying dropout.

1. Voting 효과 – mini batch 구간 동안 dropout된 각자의 망에 피팅이 되고 평균 효과를 얻을 수 있음.

2. Co-adaptation 효과 – 특정 뉴런의 바이어스나 가중치가 큰 값을 갖게 되면 그 영향이 커지면서 다른 뉴런들의 학습 속도가 느려 지거나 학습이 제대로 되지 않음. 결과적으로 이런 뉴런의 영향을 받을 확률이 적음.

Dropout

```
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Dropout

일정한 mini-batch 구간 동안 생략된 망에 대한 학습을 끝내면, 다시 무작위로 다른 뉴런들을 생략(dropout) 하면서 반복적으로 학습함.

```
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.initialize_all_variables())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
        print "step %d, training accuracy %g" % (i, train_accuracy)
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print "test accuracy %g" % accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
```

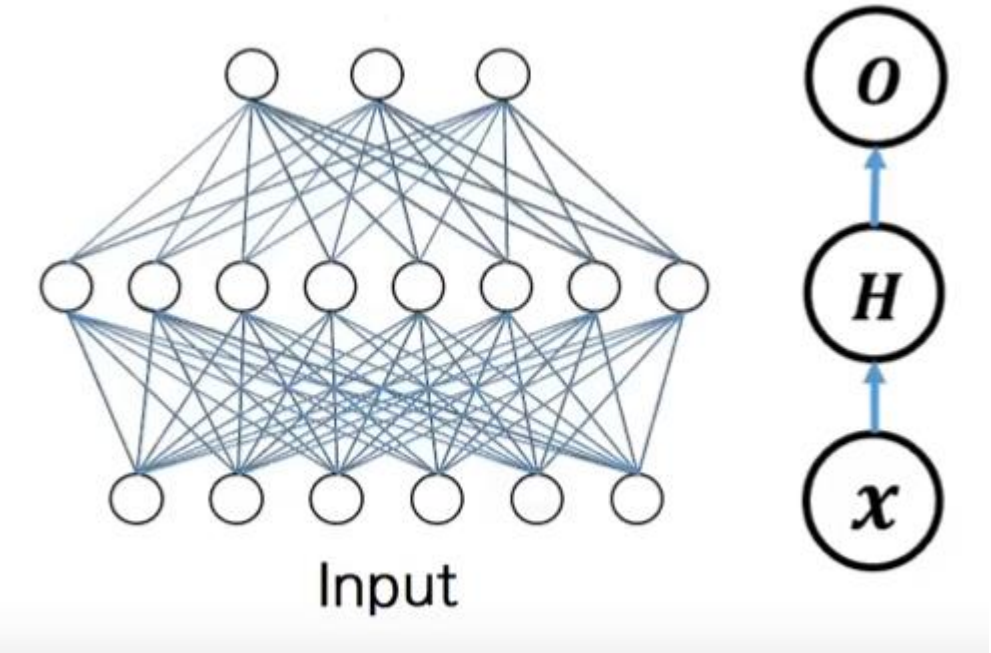
Dropout

Test 시에는 모든 layer의 노드 활성화

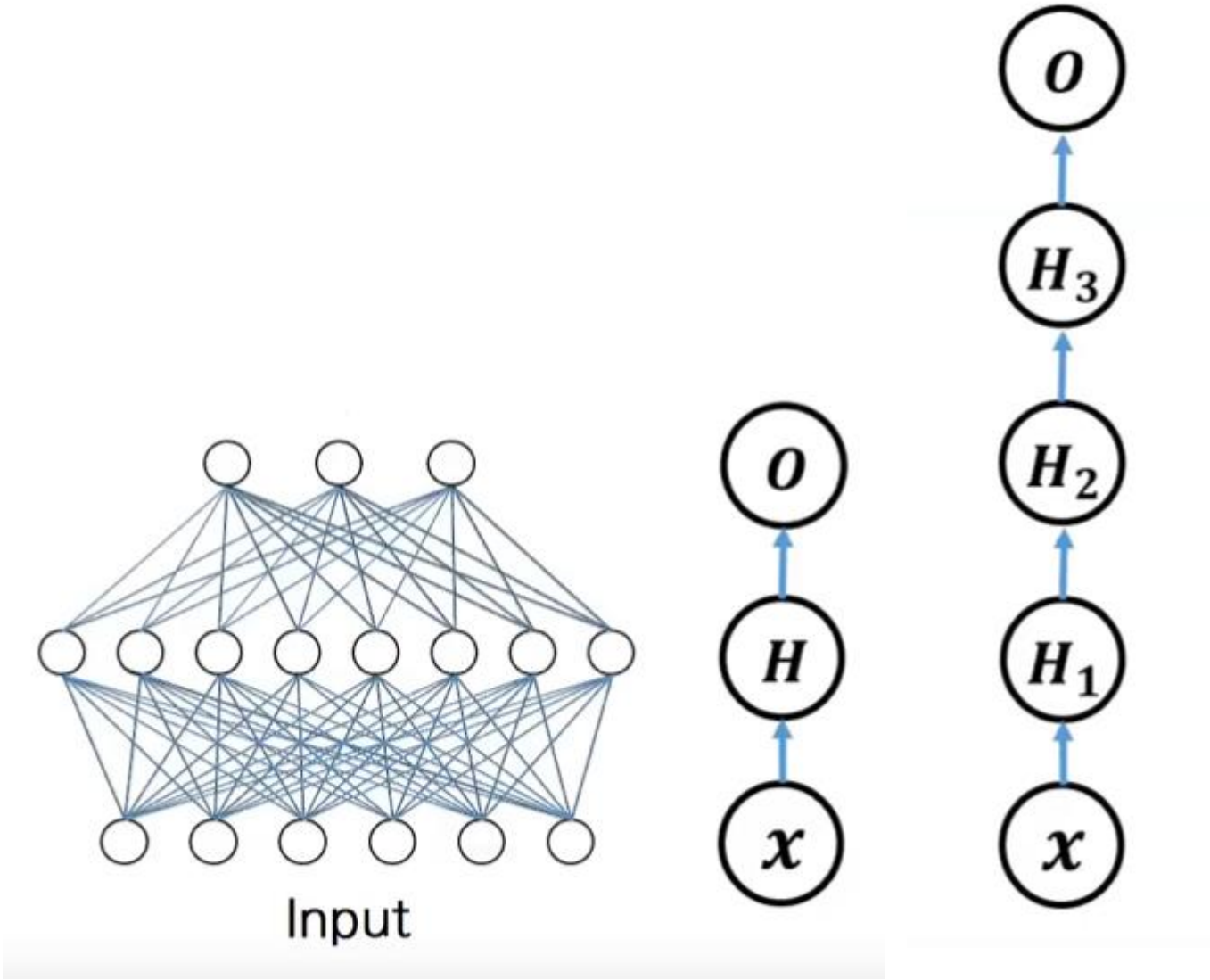
```
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.initialize_all_variables())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
        print "step %d, training accuracy %g" % (i, train_accuracy)
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print "test accuracy %g" % accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
```

Batch-Normalization

Why?



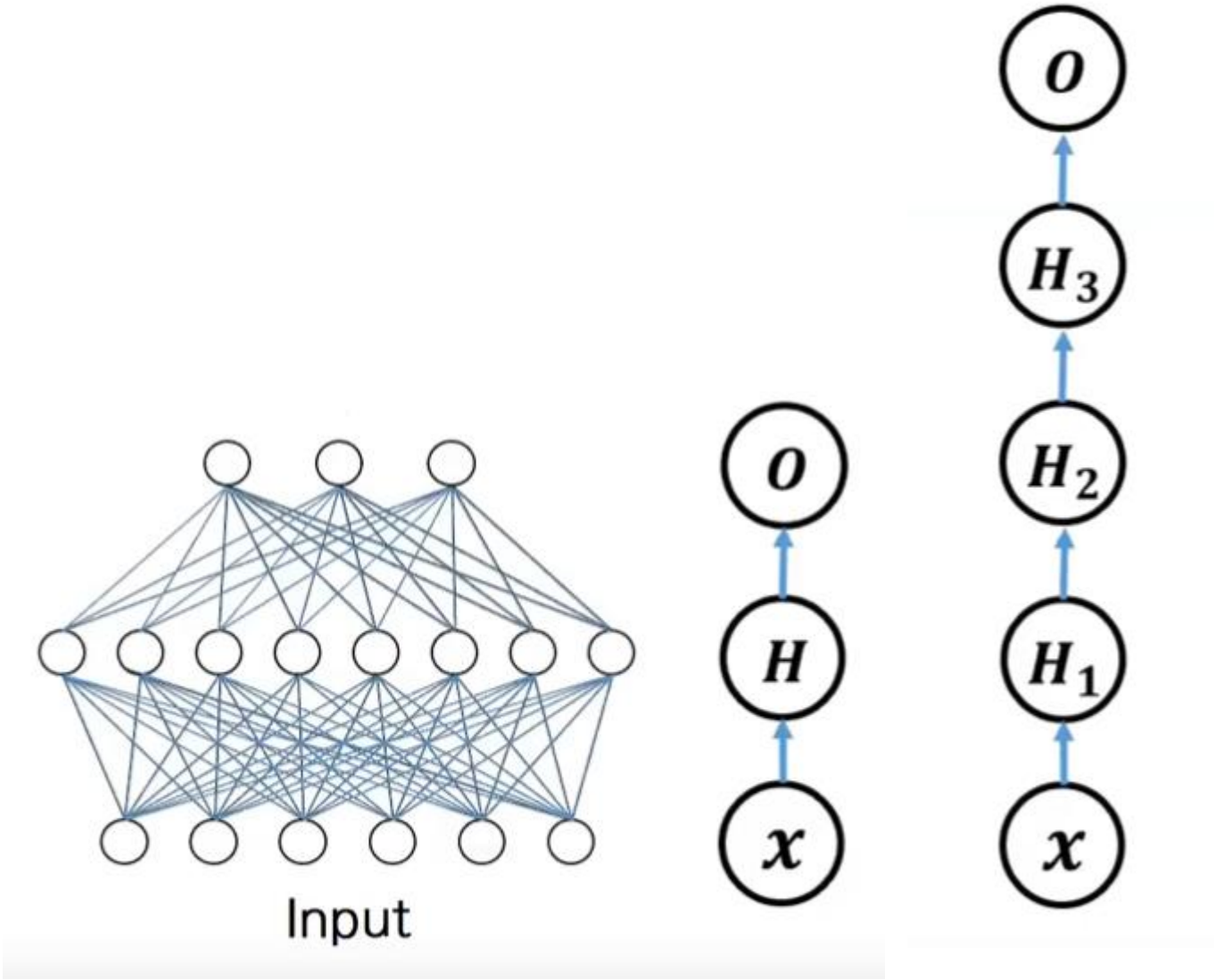
Batch-Normalization



학습시켜야 되는 Parameter가 많음.

Activation function, Initialization method의 한계.

Batch-Normalization

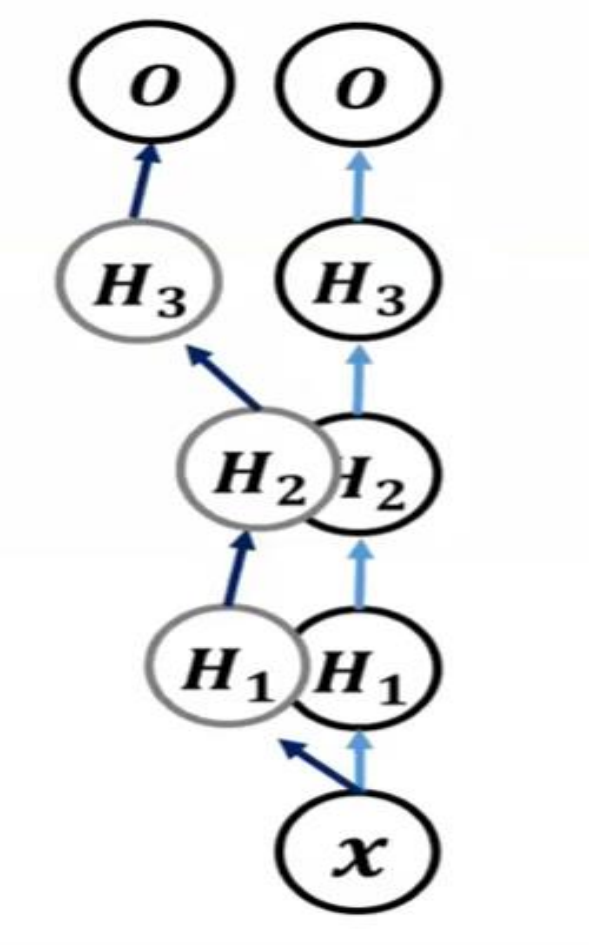


학습시켜야 되는 Parameter가 많음.

Activation function, Initialization method의 한계.

Weight의 조그만 변화가 가중되어 쌓이면 Hidden layer를 올라갈수록 값의 변화가 큼

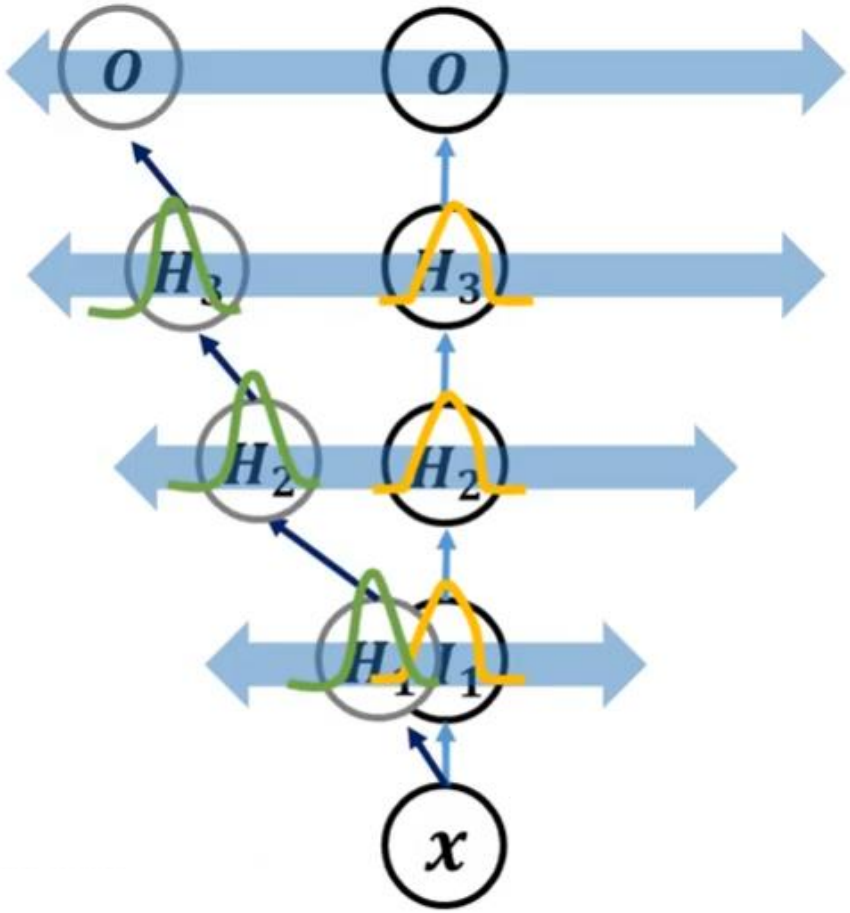
Batch-Normalization



Internal Covariate Shift(**내부 공변량 변화)**
-> 이전에 학습 했던 노드의 분포와 다음에 학습하는 분포가 다르게 되는 것

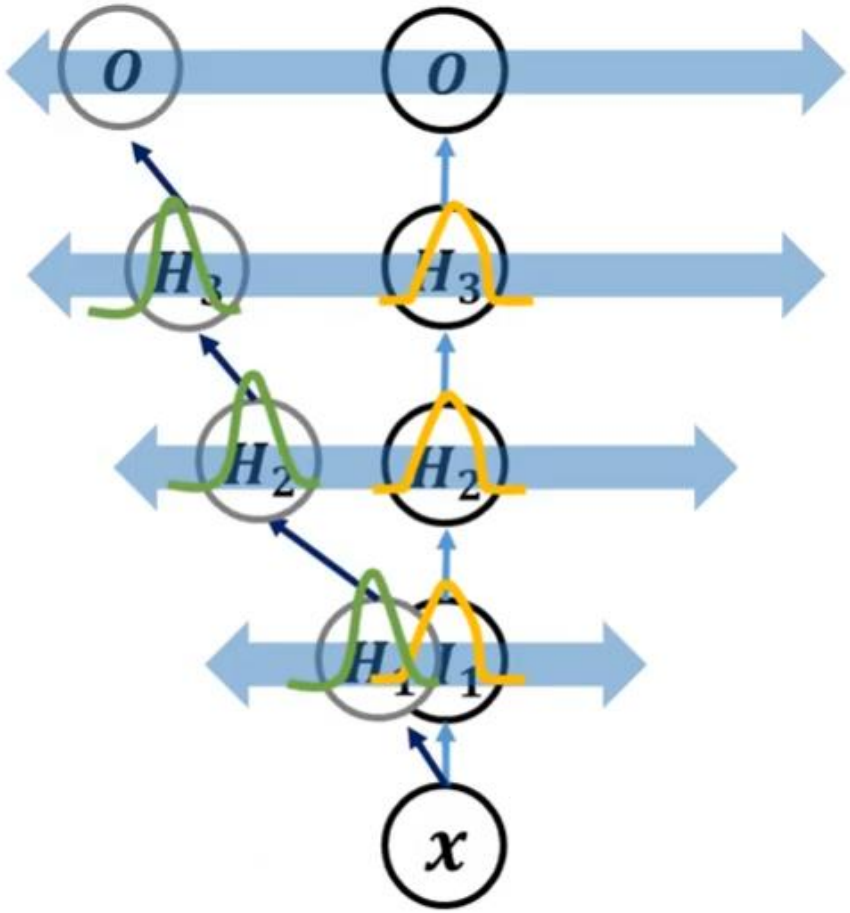
결과적으로 학습이 힘듦.

Batch-Normalization



Layer 상단으로 갈수록 노드의 분포 변화가 더 심해짐.

Batch-Normalization



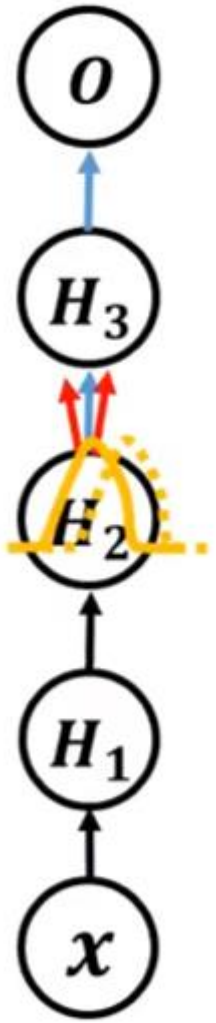
해결법

Weight Initialization을 잘한다.

Learning rate를 줄인다.

->어렵고 학습 속도가 느림.

Batch-Normalization

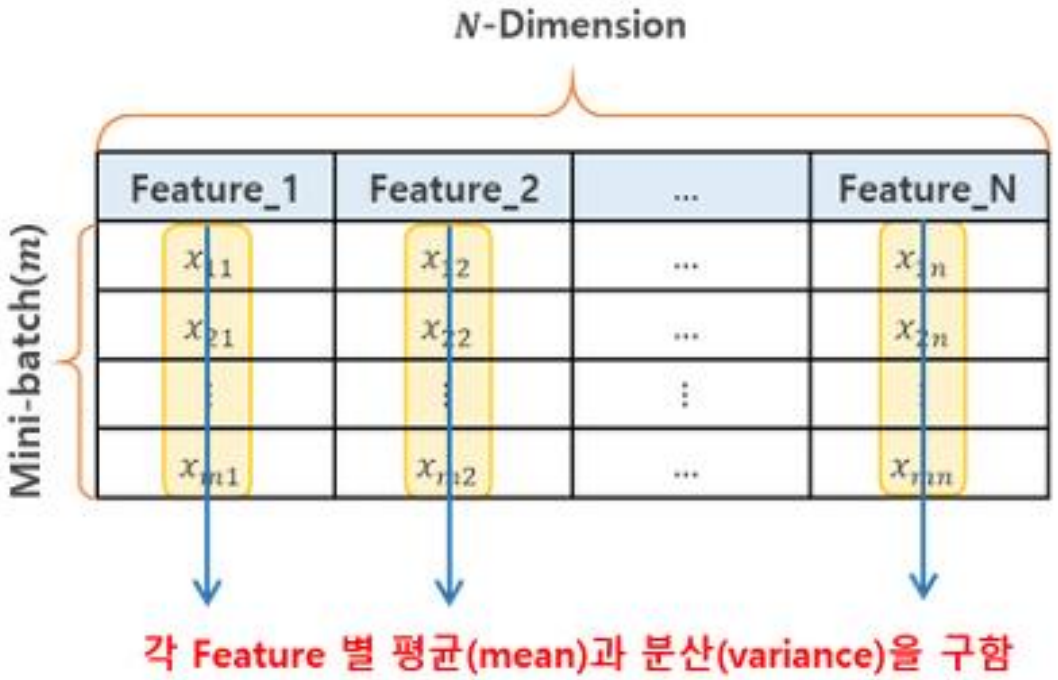


Weight값이 가중되어서 히든 layer의 노드 값이 변화하는 범위가 적으면 학습이 잘 될 것이다라는 가정을 함.

BN을 이용해서 해보겠다~

간단히 activation function에 들어가기 전 input의 range를 restrict하는 것.

Batch-Normalization



Normalize

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

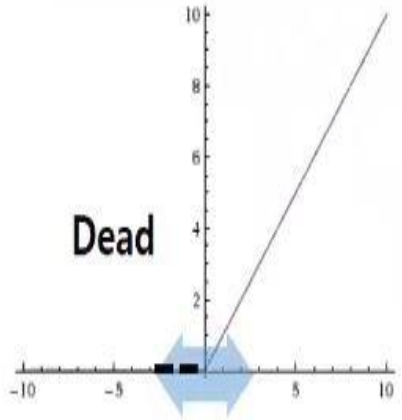
$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \qquad \text{// mini-batch variance}$$

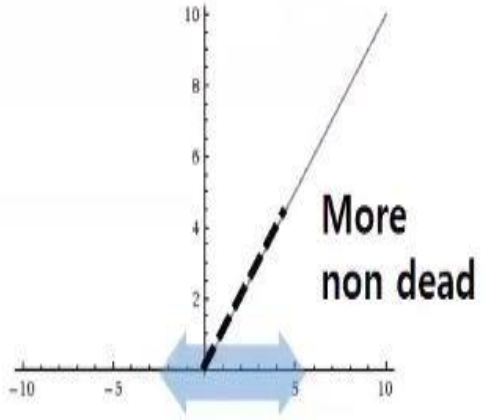
$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad \text{// normalize}$$

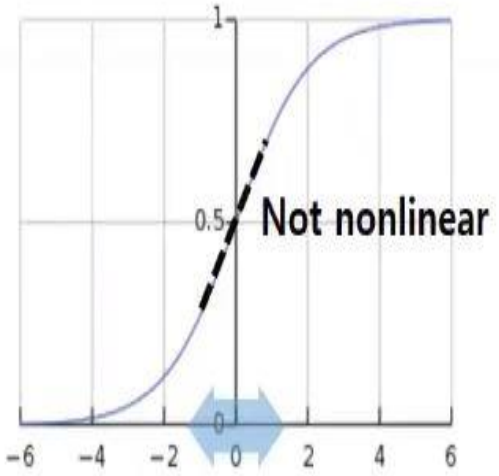
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

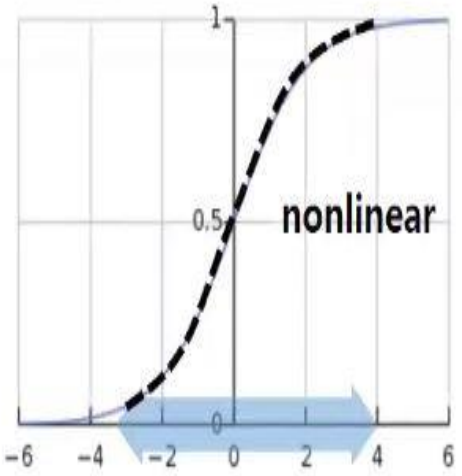
Batch-Normalization

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

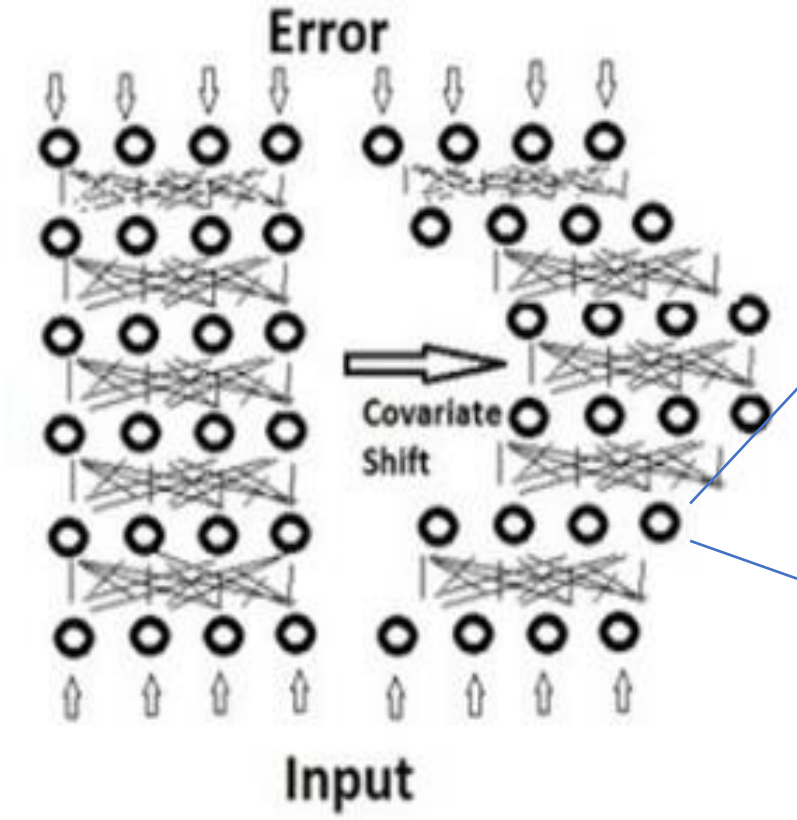


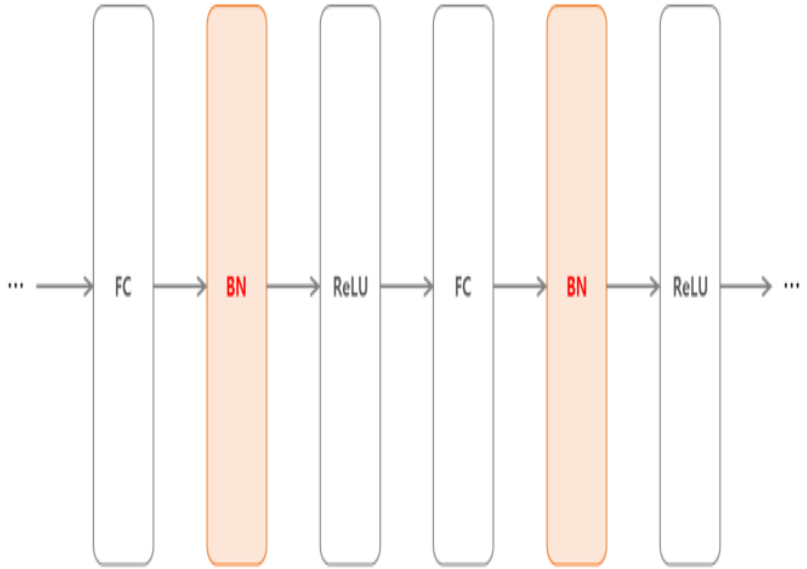






Batch-Normalization





[BN을 사용한 신경망 예]

Batch-Normalization

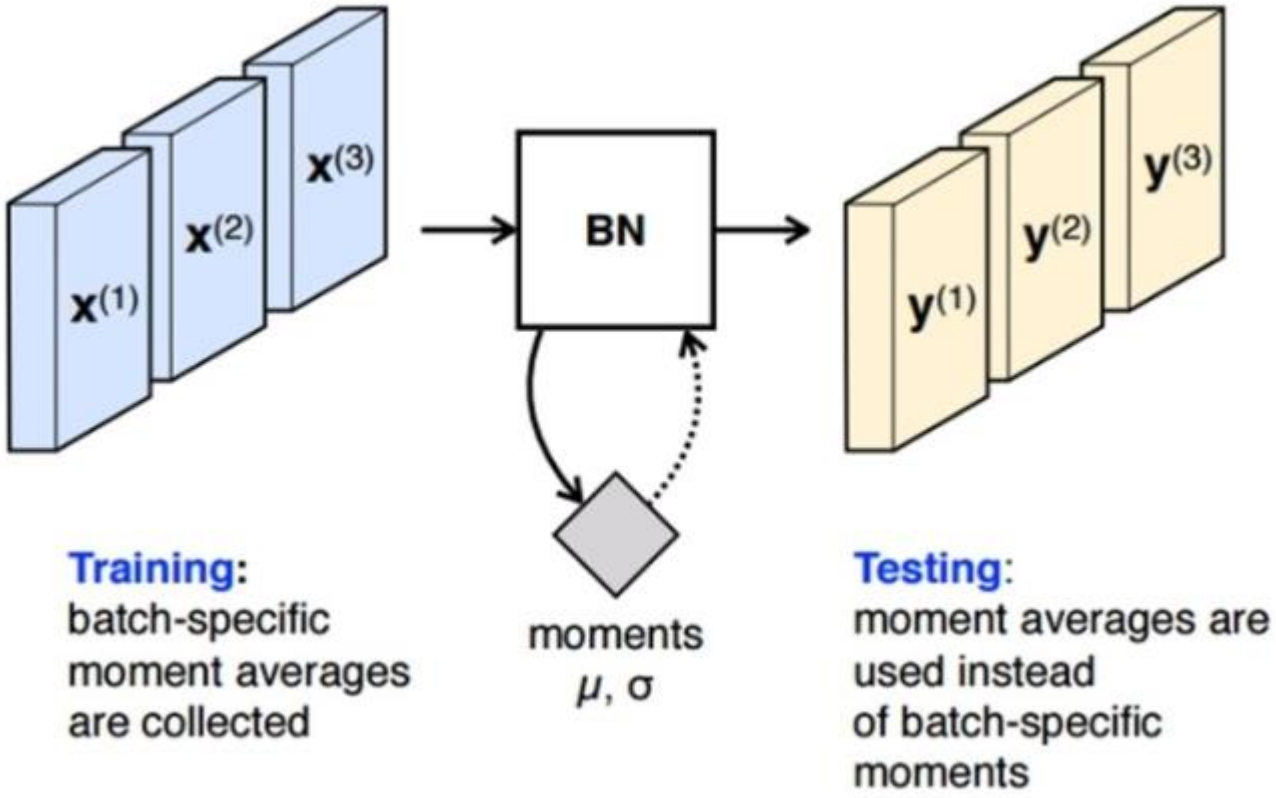
```
with tf.name_scope('dnn'):
    # batch normalization layer using partial
    batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training=training,
            momentum=batch_norm_momentum)

    # 1st - hidden
    hidden1 = tf.layers.dense(inputs, n_hidden1, name="hidden1")
    # batch norm
    bn1 = batch_norm_layer(hidden1)
    # activation function
    bn1_act = tf.nn.elu(bn1)

    # 2nd - hidden
    hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
    bn2 = batch_norm_layer(hidden2)
    bn2_act = tf.nn.elu(bn2)

    # outputs
    logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
    logits = batch_norm_layer(logits_before_bn)
```

Tensorflow에서는 tf.nn.batch_normalization()과 tf.layers.batch_normalization()을 통해 BN 기능을 제공

전자의 경우 Mean, Variance값(학습시, 테스트시) 각각 계산해 인자로 넘겨주어야 하고 Scaling, Shift factor의 경우도 따로 만들어 줘야한다.

후자의 경우 다 해줌~

Batch-Normalization



- Test 단계
- 데이터가 하나씩 들어간다면, mean, variance를 계산할 mini-batch가 없기 때문에 전체 Training Set의 평균과 표준편차를 사용.
- 계산량이 많은 경우 Normalization에 필요한 mean, variance를, 그동안 계산했던 mean, variance의 평균으로 놓고 output을 구한다.

Batch-Normalization

- Tanh, Sigmoid같은 활성화 함수에 대해 vanishing gradient 문제가 감소한다.
- 가중치 초기화에 덜 민감.
- Learning rate를 크게 잡아도 GD가 잘 수렴.
- Overfitting 억제, drop out 필요성 감소

끝