[2018년 여름]

Tensorflow를 이용한 Deep Learning의 기초

제4강: Convolutional Neural Network (CNN)

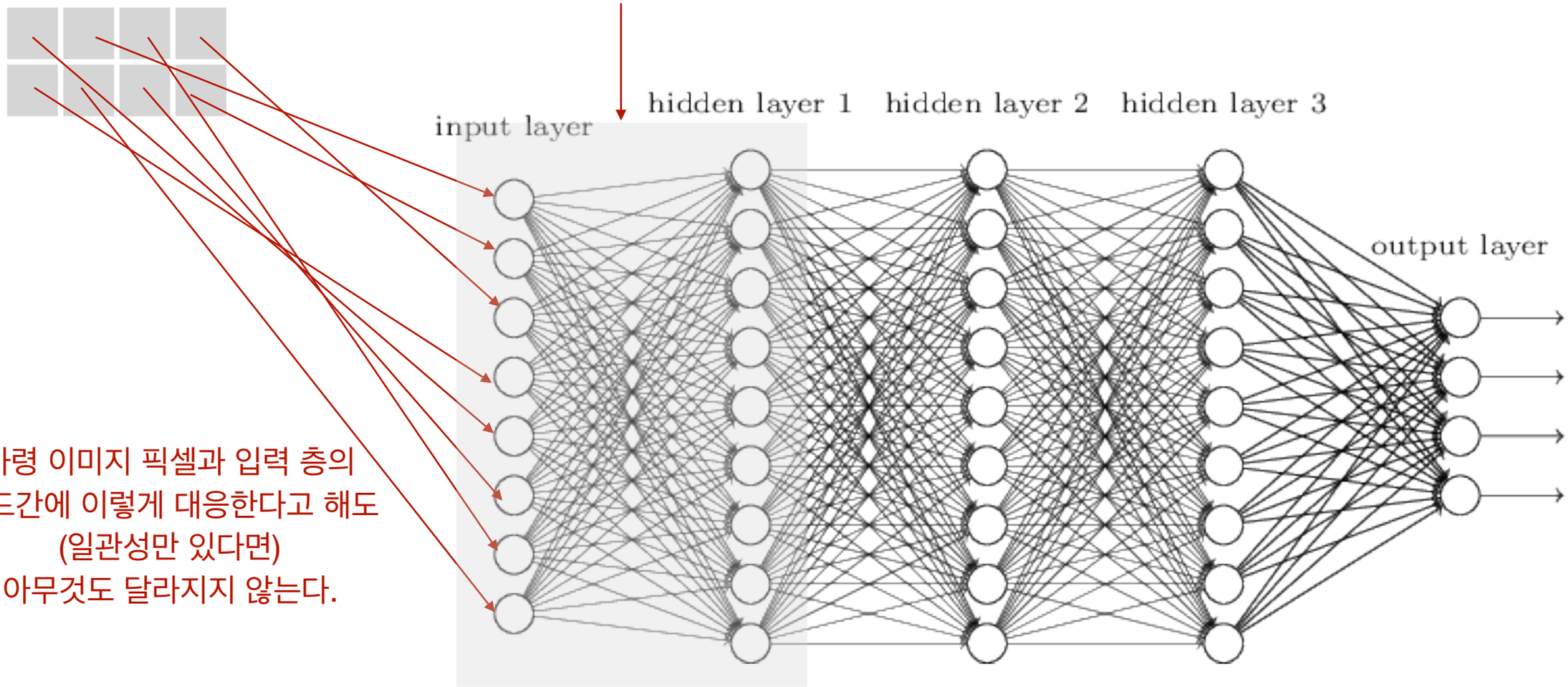
CNN: Motivation

- 이미지는 다른 유형의 데이터와 달리 데이터 내에서 각 구성요소의 상대적 위치가 중요한 의미를 가진다.
- Fully connected layer는 이미지가 가진 지역적 특성(local feature)를 효과적으로 추출하지 못한다.

Fully Connected Layers

왜냐하면 이 연결이 fully connected이기 때문에
입력 층의 노드들은 모두 **구조적으로 동일한 위치**이기 때문이다.

2×4 image

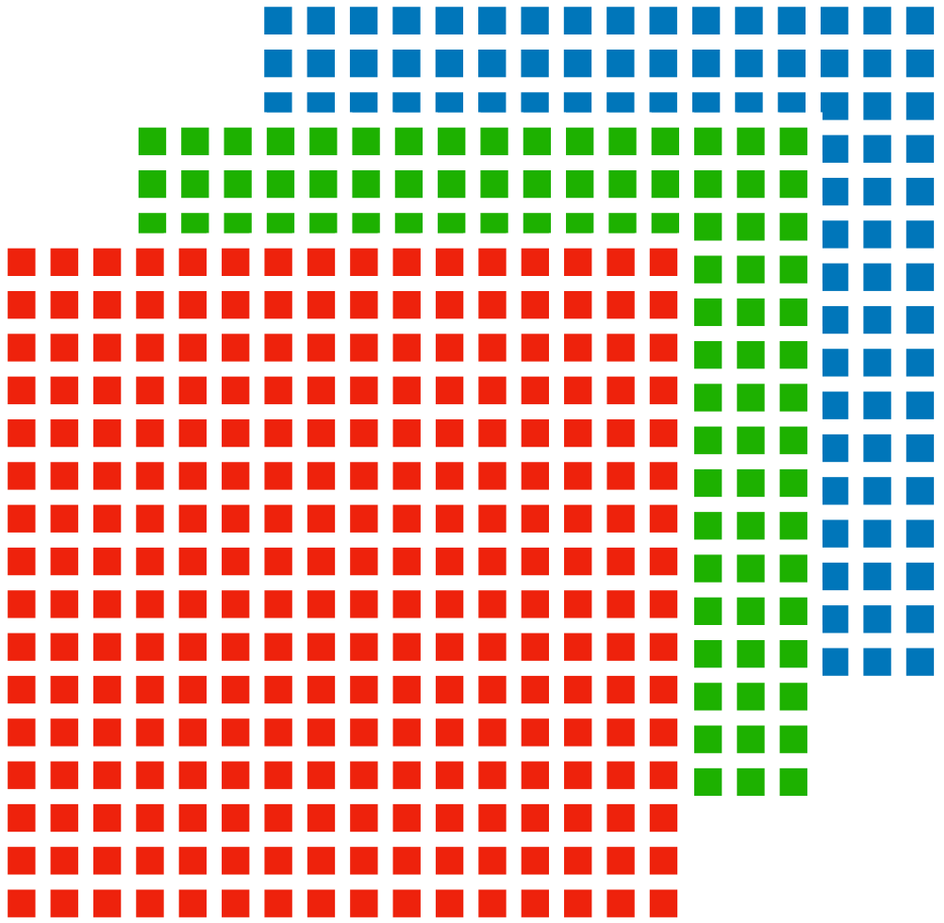


가령 이미지 픽셀과 입력 층의
노드간에 이렇게 대응한다고 해도
(일관성만 있다면)
아무것도 달라지지 않는다.

- **Fully connected 네트워크는 이미지가 가진 국부적(local) 특성을 효과적으로 추출하지 못한다.**

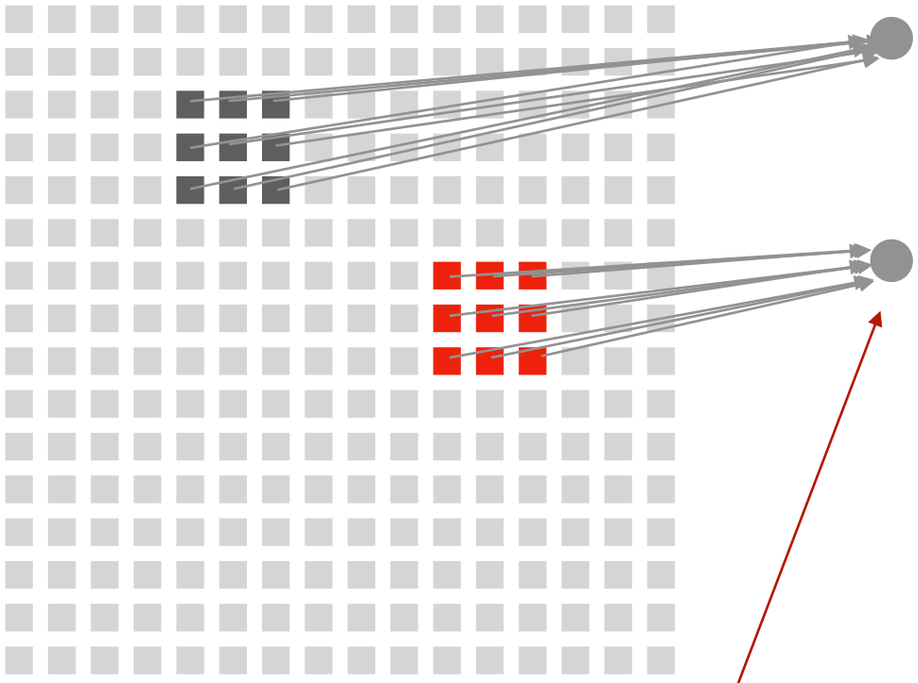
- 입력 이미지를 1차원 벡터로 변환하지 않고 2차원 (혹은 RGB이미지의 경우 3차원) 구조를 그대로 유지한다.

1 채널 이미지의 경우



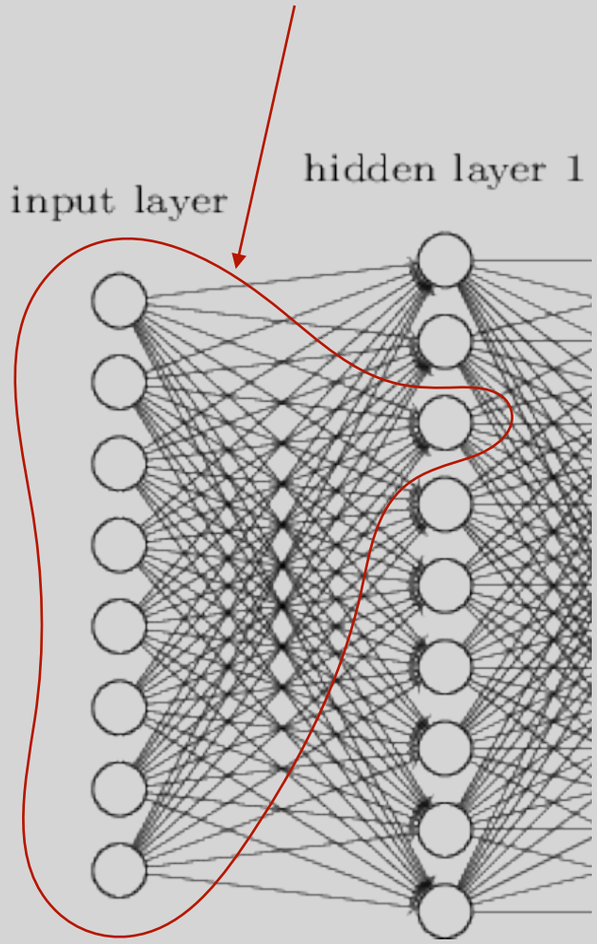
3 채널 이미지의 경우

- 모든 픽셀들을 다음 layer의 각 노드에 연결하는 대신 **“작은 지역적인 영역 (local receptive field)”**만을 다음 층의 한 노드에 연결한다.

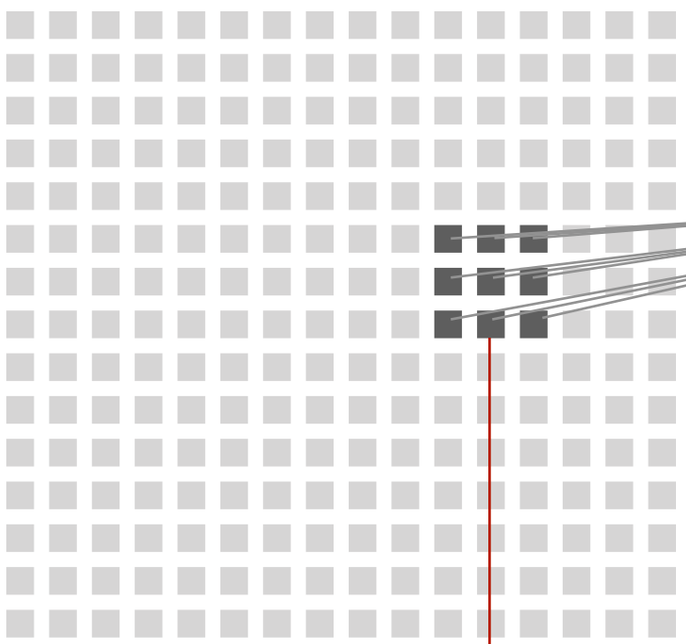


이 노드에는 빨간 9개의 픽셀만이 연결된다. 즉 이 노드는
9개의 픽셀로 구성된 국부적인 한 영역의 특징을 추출하는 역할을 한다.
이 예에서는 3×3 크기의 영역을 사용하였다.
보통 3×3 혹은 5×5 크기가 주로 사용된다.

Fully connected의 경우
모든 pixel들이 다음 층의
각각의 노드와 연결되었다.



- “작은 지역적인 영역(local receptive field)”의 특징을 추출하기 위해서 하나의 작은 가중치 행렬을 사용한다. 이 행렬을 보통 필터(filter) 혹은 커널(kernel)이라고 부른다.
- 이 행렬은 모든 receptive field에 대해서 공유(shared)된다.



5	9	1
2	3	6
15	8	10

픽셀 값

⊛

3×3 Filter

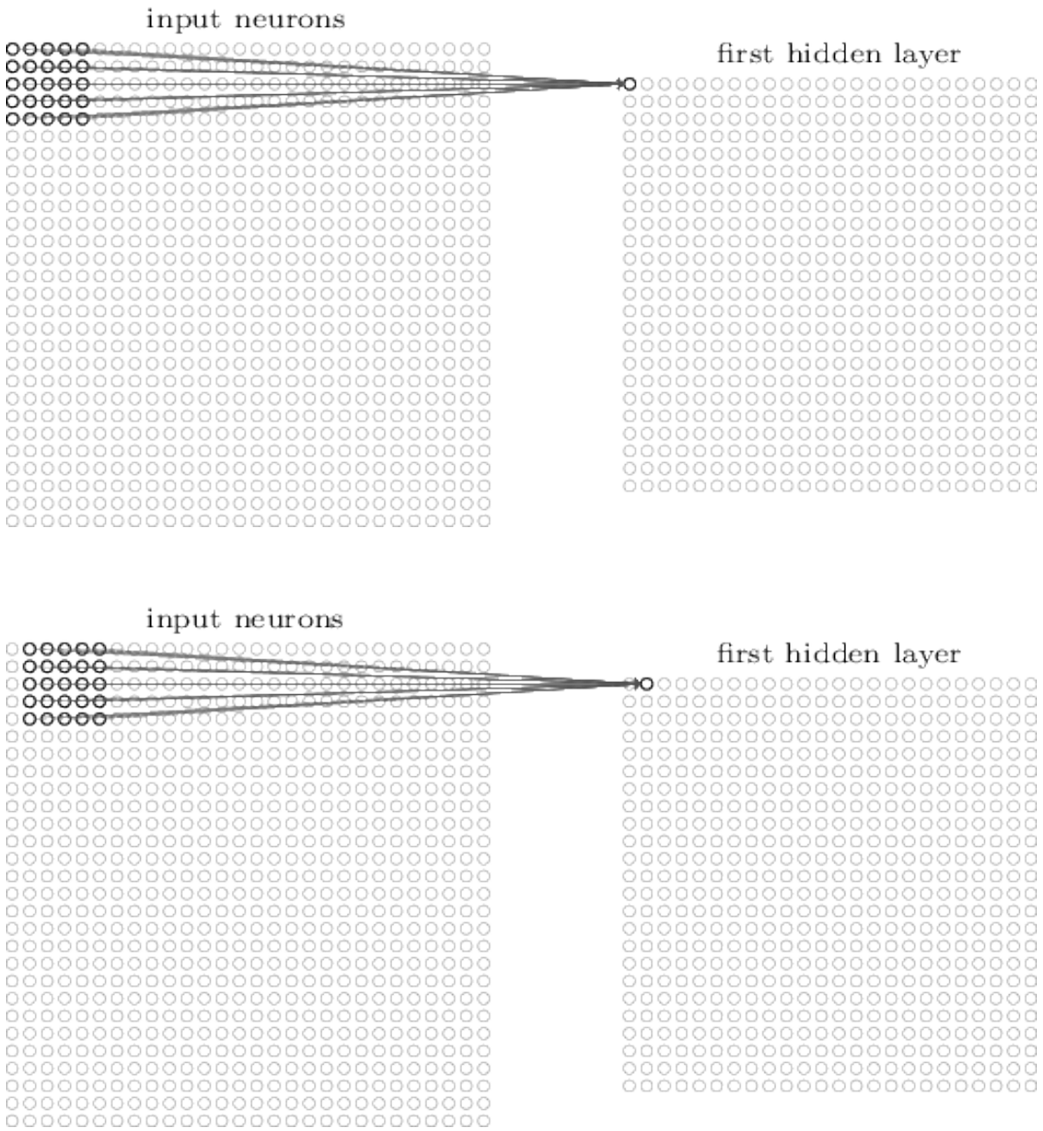
-1	1	-2
3	1	-4
-11	3	9

=

$$5 \cdot (-1) + 9 \cdot 1 + 1 \cdot (-2) + \cdots + 10 \cdot 9 + \text{bias}$$

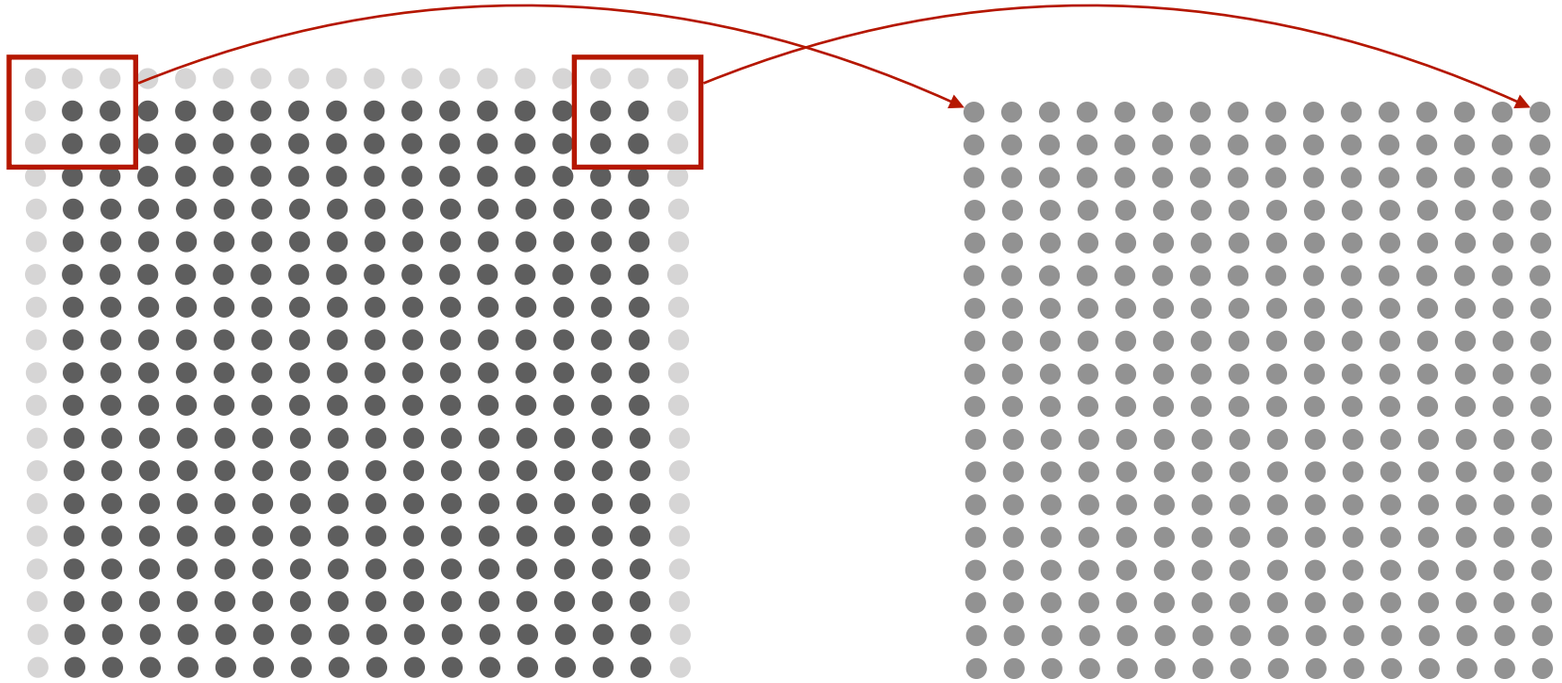
이 filter 행렬과 bias가 바로 최적화할 변수이다.

- **Receptive field를 슬라이딩 시키면서 공유된 filter를 사용해 계산한다.** 아래의 그림은 필터의 크기가 5×5인 경우이다.



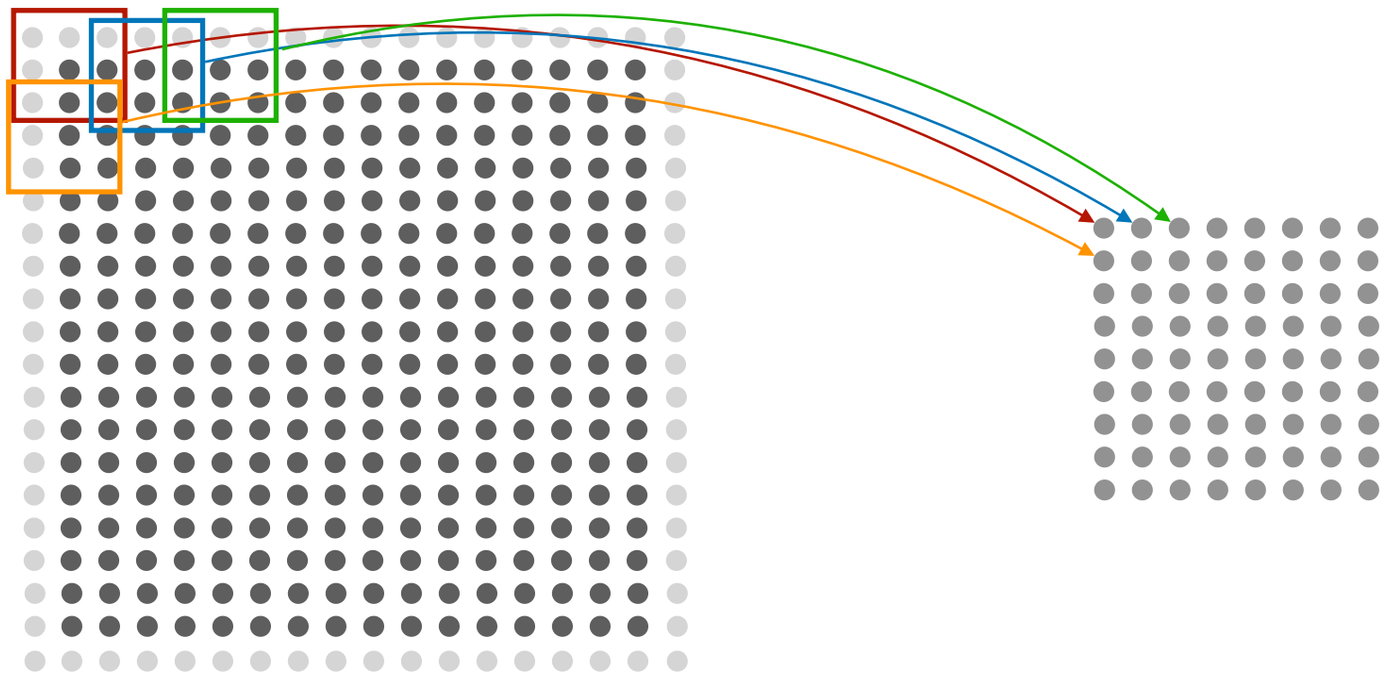
원래 이미지가 N×M 크기이면
(N-4)×(M-4) 크기로 줄어든다.
만약 필터가 3×3이면 (N-2)×(M-2)
로 줄어든다.
이것은 약간 성가시다.

- 이미지의 상하좌우에 가상적인 픽셀을 추가(padding)하여 크기가 줄어드는 것을 방지한다. 3×3 filter를 사용한다면 1줄씩, 5×5 filter를 사용한다면 2줄씩 추가한다. 추가된 행과 열의 픽셀값은 보통 0으로 가정하거나 혹은 인접한 실제 픽셀의 값을 사용하기도 한다.



이미지의 크기가 그대로 유지된다.

- 윈도우를 슬라이딩할 때 반드시 한 픽셀씩 슬라이딩 해야하는 것은 아니다. 가령 2칸씩 슬라이딩 할 수도 있다. 이 경우 이미지의 크기는 1/2로 줄어든다. 이렇게 슬라이딩하는 단위를 stride라고 부른다. 가로와 세로의 stride 값을 다르게 할 수도 있다.



가로/세로 stride 값이 2라면 이미지의 크기가 1/2로 줄어든다.

http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html

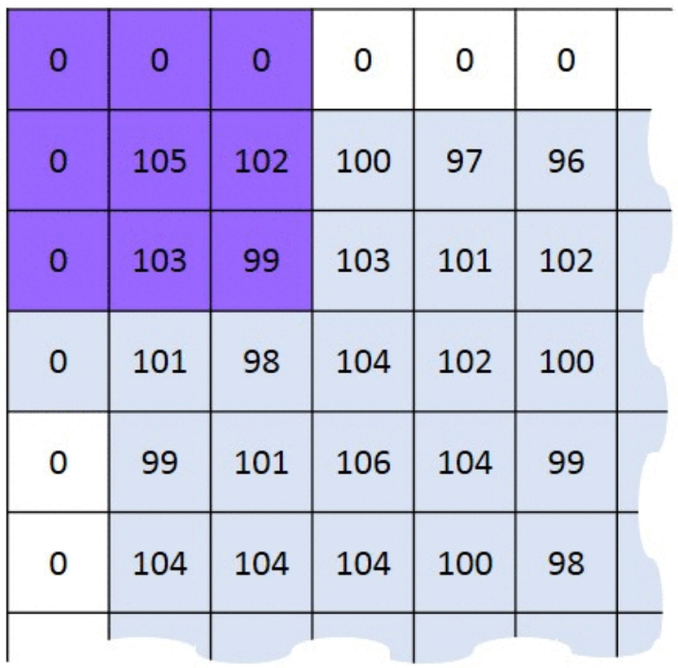
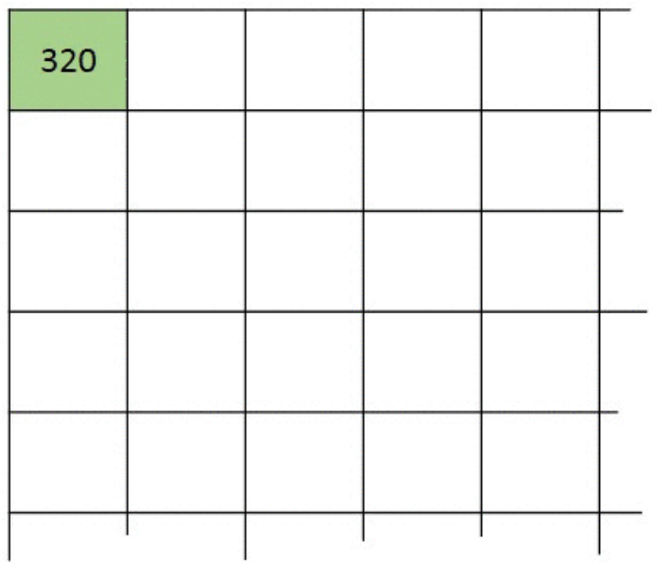


Image Matrix

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

$$0*0+0*-1+0*0$$
$$+0*-1+105*5+102*-1$$
$$+0*0+103*-1+99*0 = 320$$



Output Matrix

Convolution with horizontal and vertical strides = 1

Download and watch this GIF animation

1채널 이미지의 경우 (strides=2)

http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html

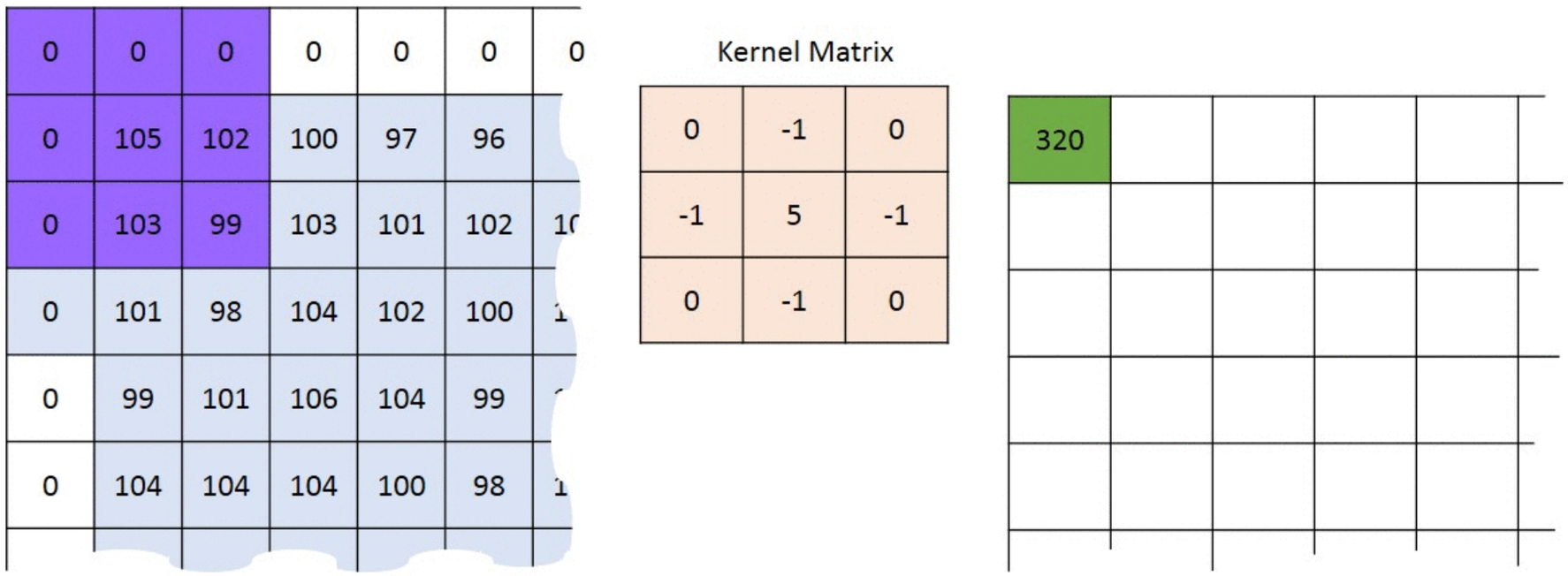


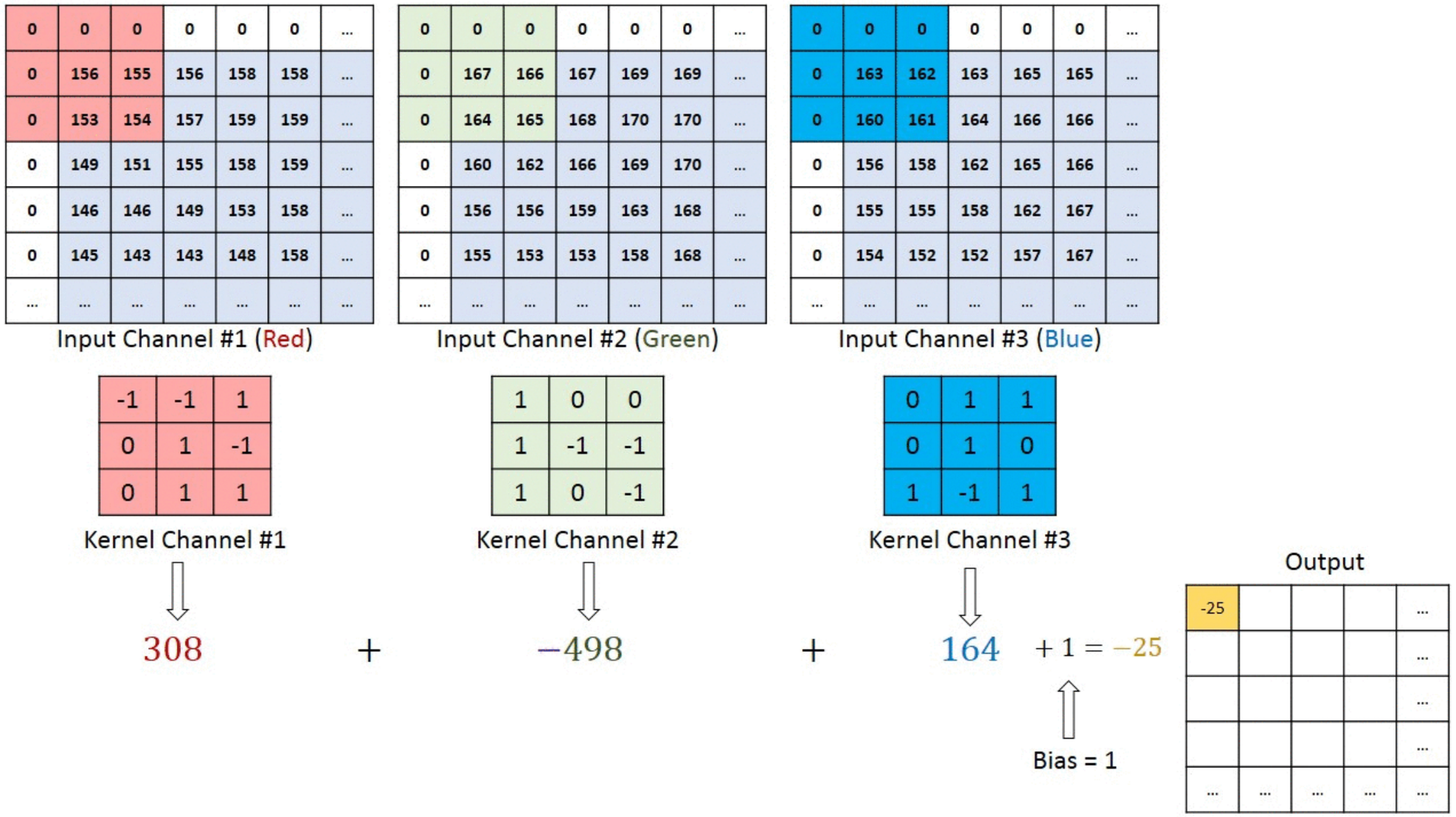
Image Matrix

$$0*0+0*-1+0*0$$
$$+0*-1+105*5+102*-1$$
$$+0*0+103*-1+99*0 \quad = 320$$

Output Matrix

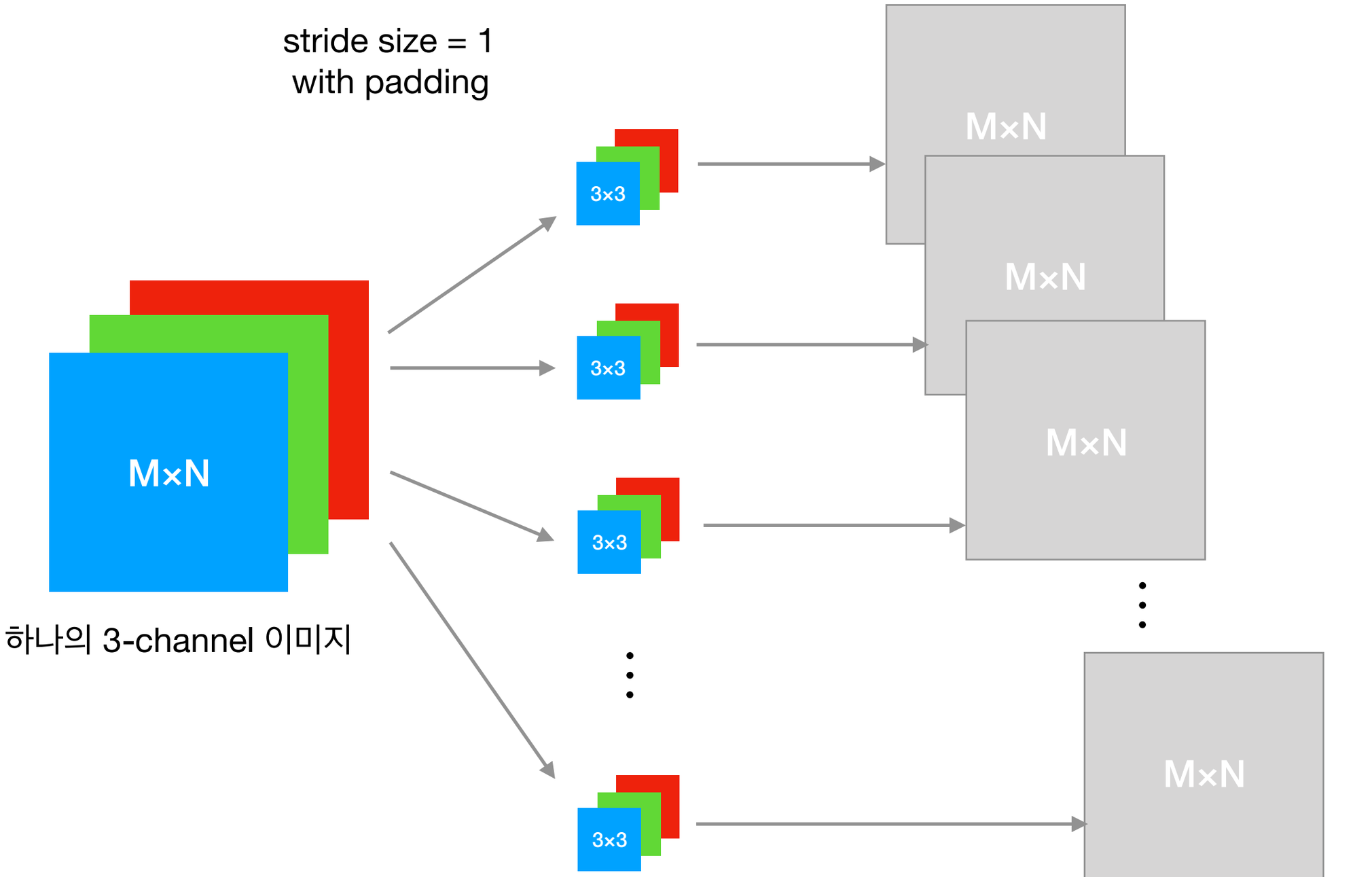
Convolution with horizontal and vertical strides = 2

Download and watch this GIF animation



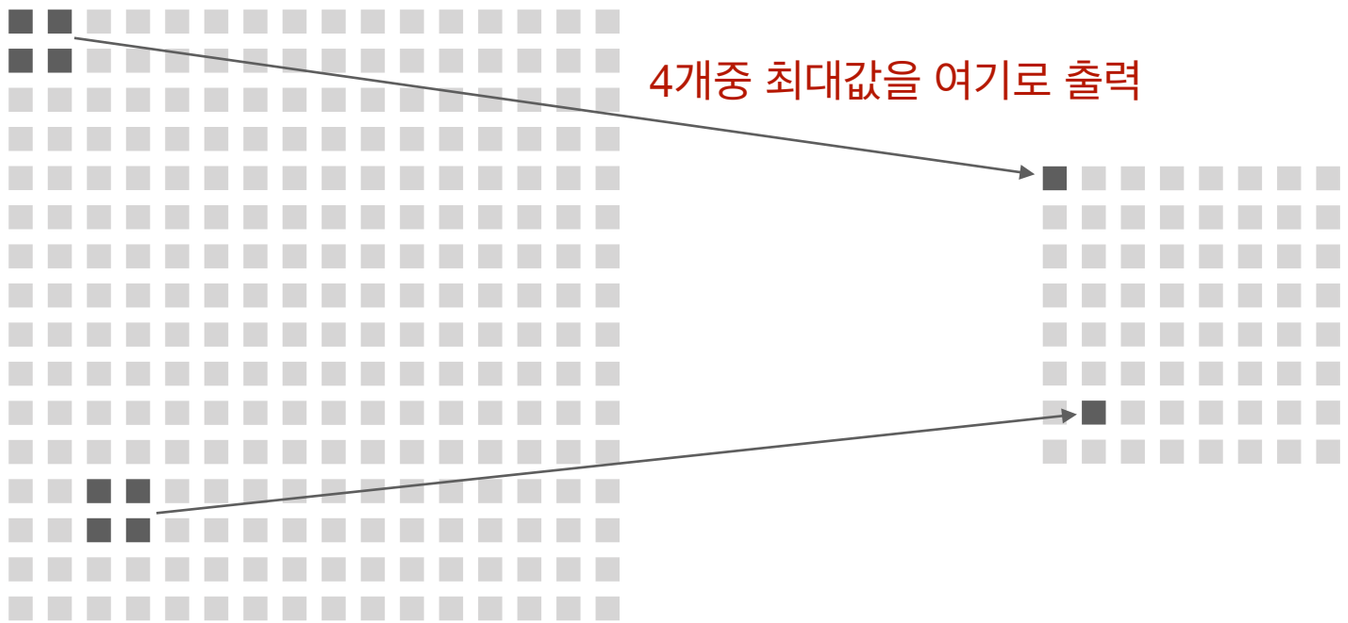
Download and watch this GIF animation

Single Convolution Layer



Max Pooling Layers

- 일반적으로 convolution layer에 뒤이어 pooling layer를 적용한다.
- Pooling layer의 역할은 convolution layer의 출력을 압축(혹은 단순화)하는 것이다.
- Pooling에는 max-pooling, L2-pooling 등이 있으며, max-pooling이 가장 흔하게 사용된다.
- 2×2 max pooling에서는 2×2 영역의 값 중에서 최대값을 선택한다.

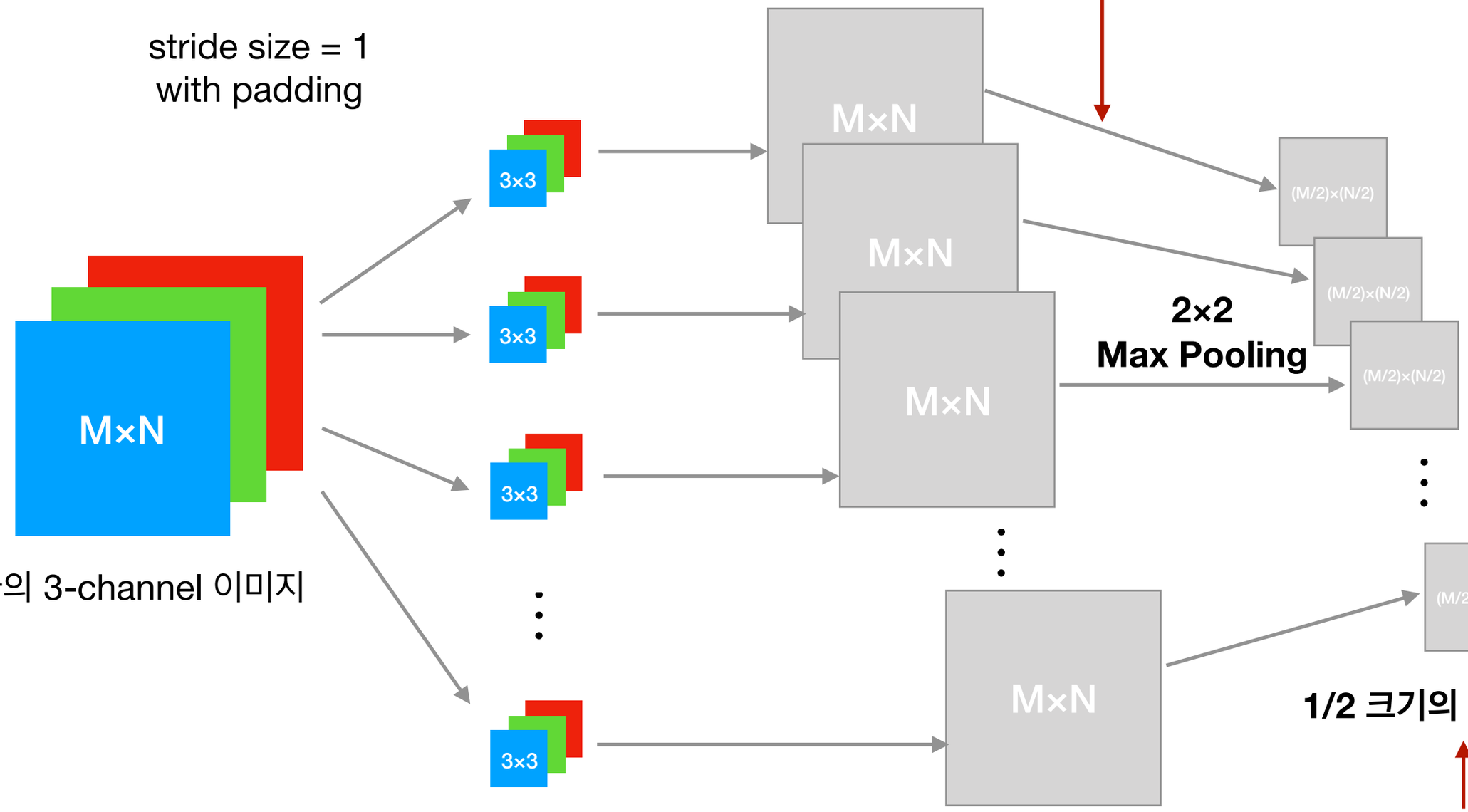


2×2 max pooling을 적용하면 이미지의 크기가 가로/세로 각각 반으로 줄어든다.

Convolution + Max Pooling Layer

그림에는 없지만 여기에
relu activation 함수도 적용한다.

stride size = 1
with padding



하나의 3-channel 이미지

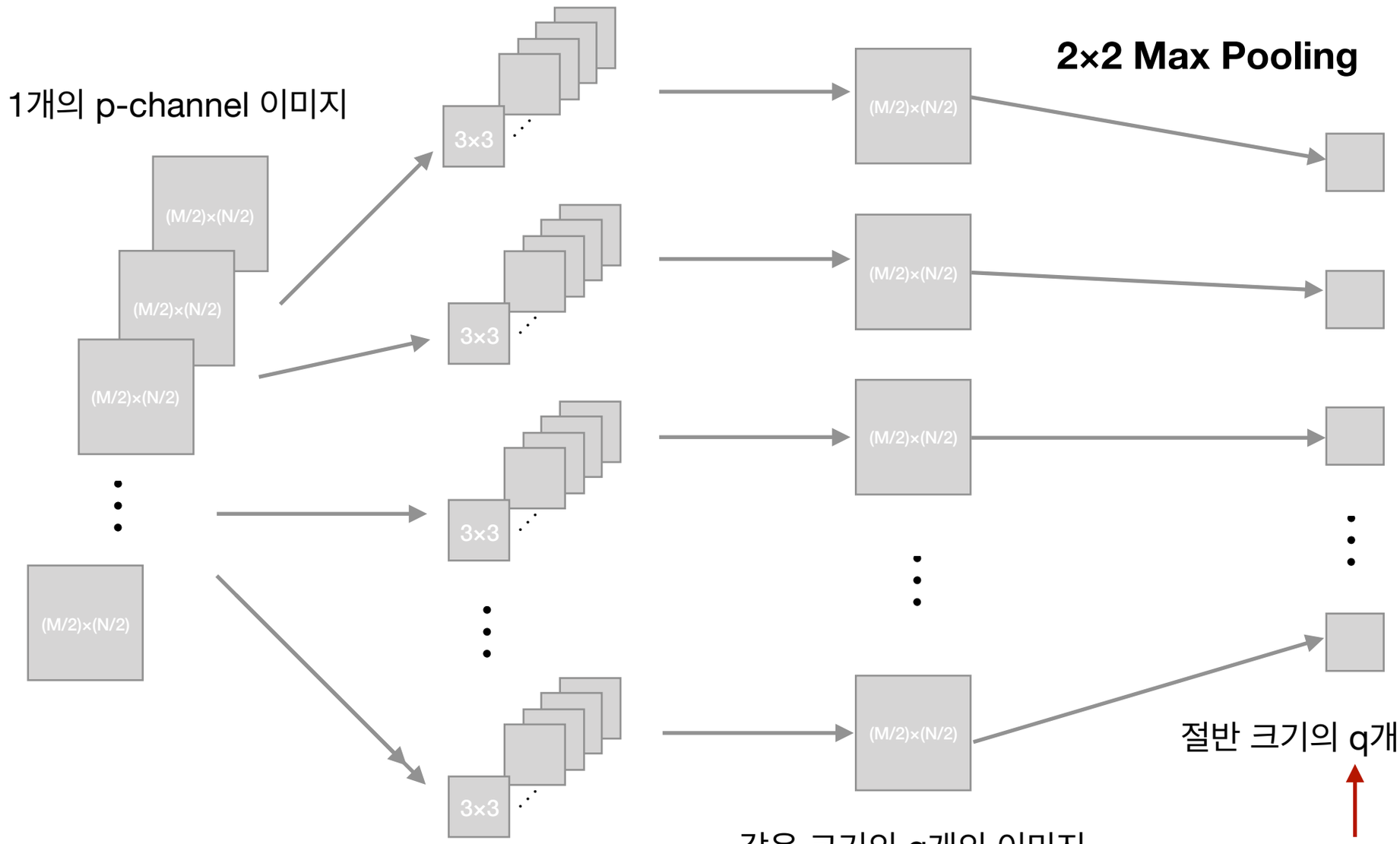
p개의 3-channel kernel

같은 크기의 p개의 이미지

1/2 크기의 p개의 이미지

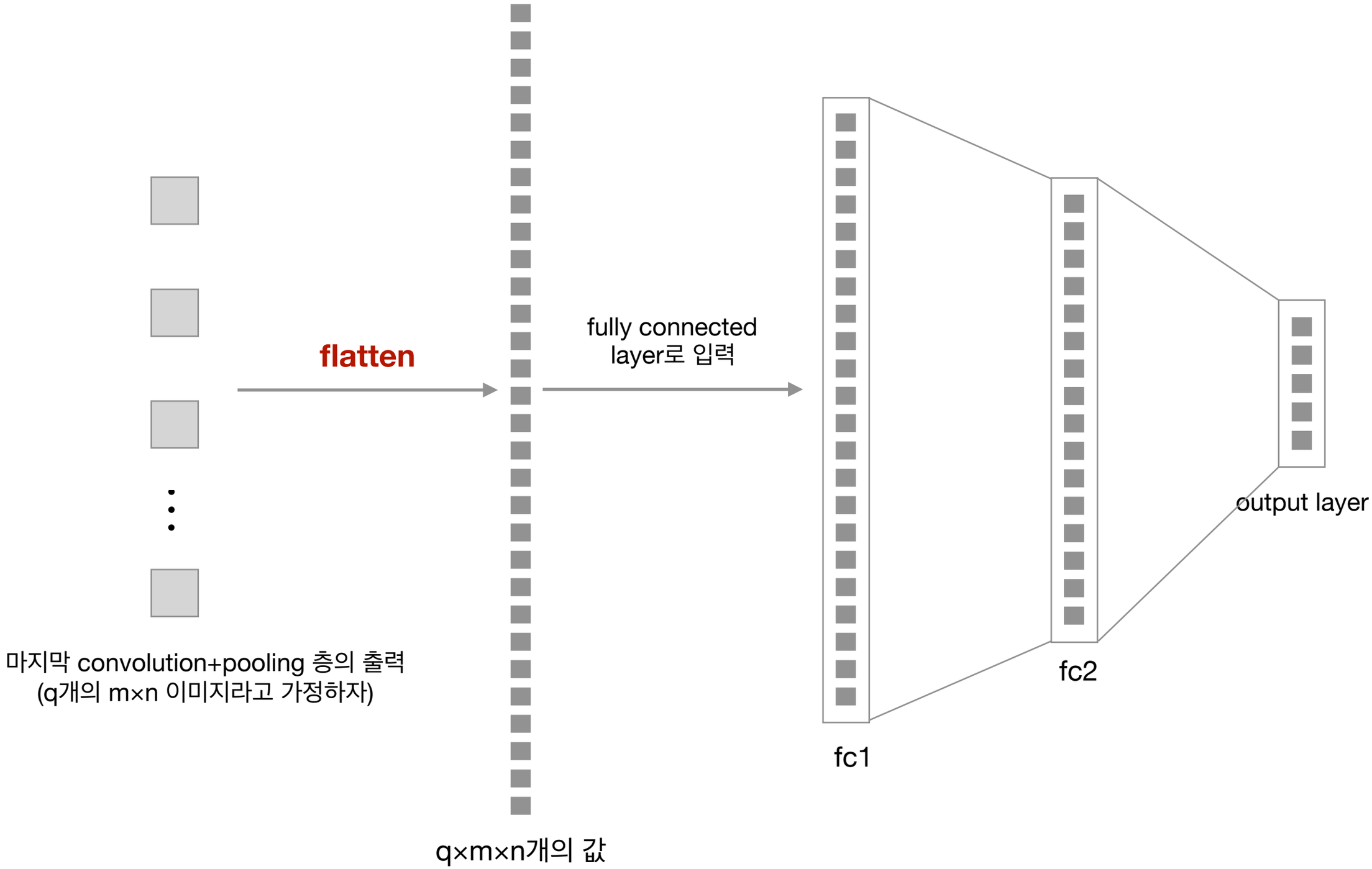
이것들이 다음 convolution 층에 마치 하나의 p-채널 이미지 처럼 제공된다.

Next (Convolution + Pooling) Layer

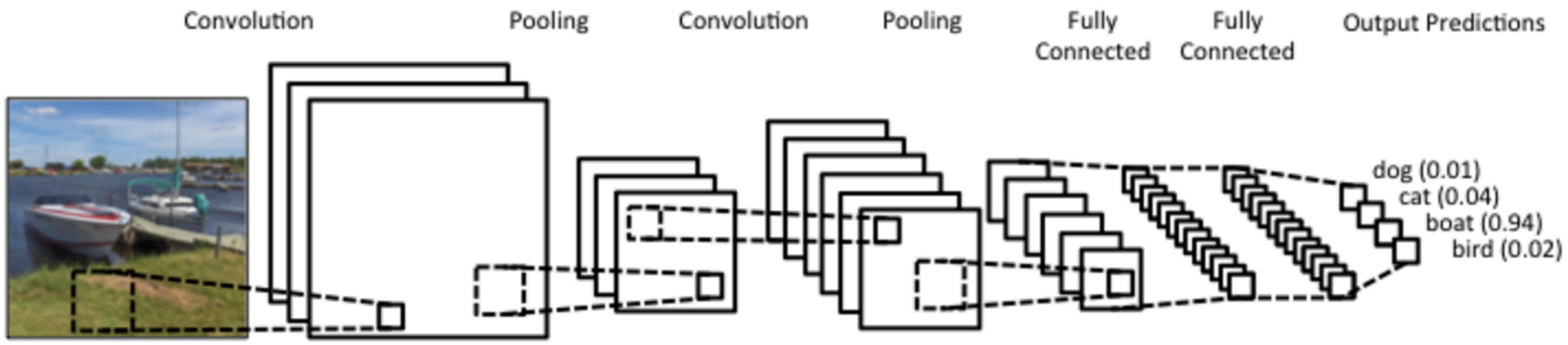


이것들이 다음 convolution 층에 마치 하나의 q-채널 이미지 처럼 제공된다.

- 일반적으로 convolution + pooling 층들 뒤에 몇 개의 fully connected 층을 배치한다.

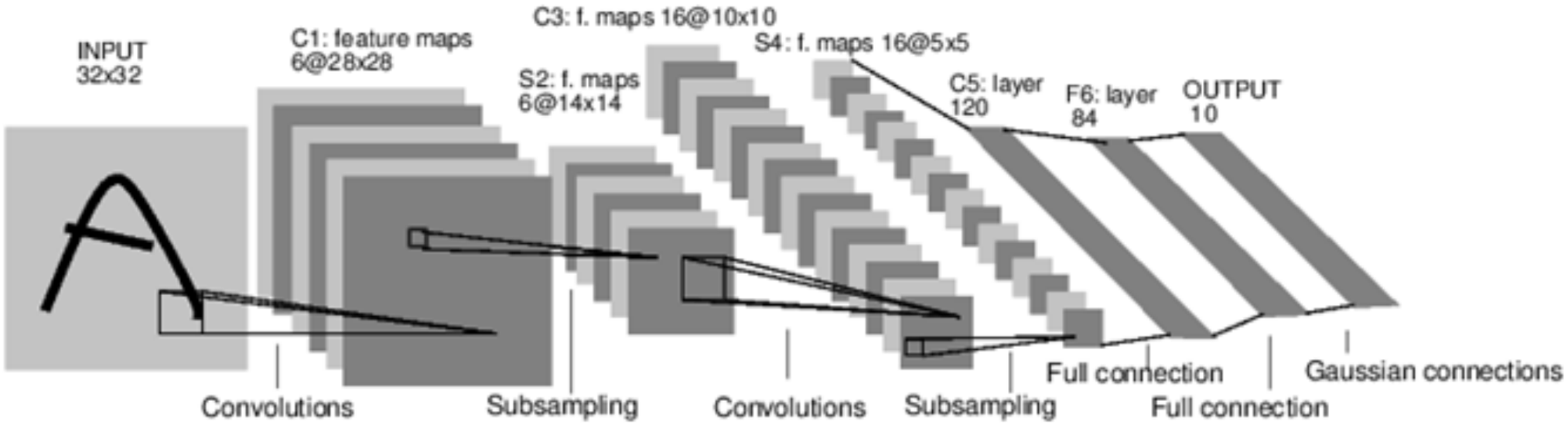


전형적인 CNN의 구조



http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

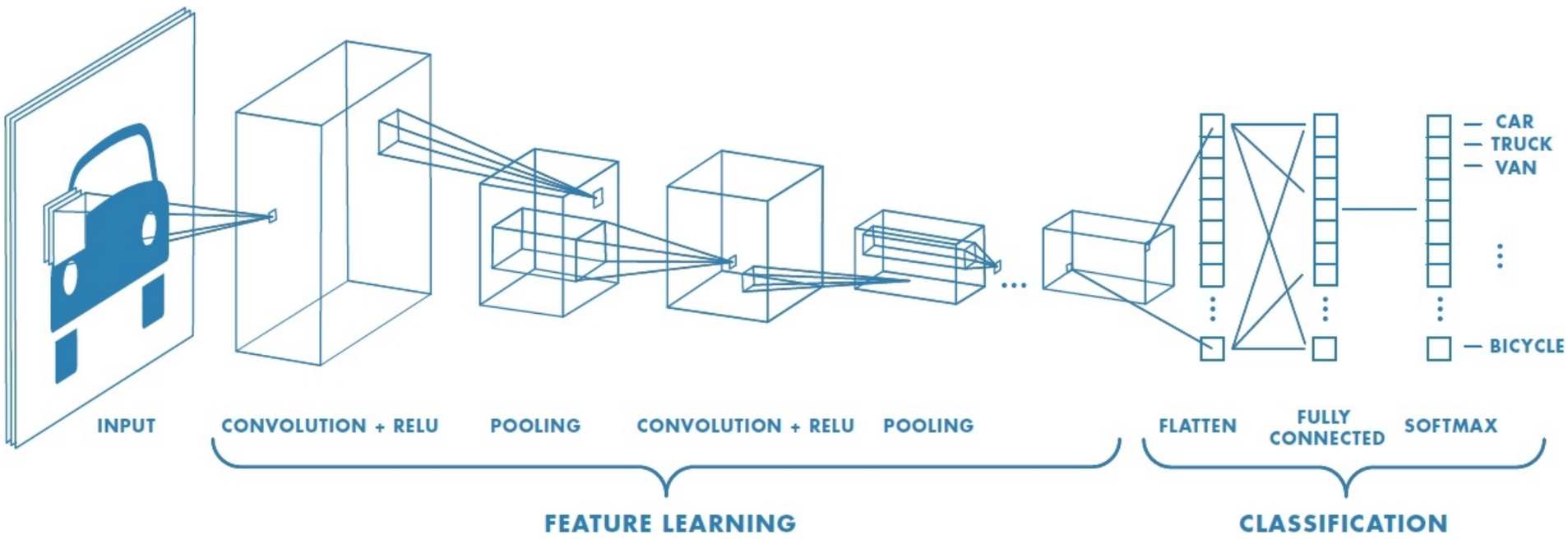
전형적인 CNN의 구조



A Full Convolutional Neural Network (LeNet)

https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

전형적인 CNN의 구조



https://www.mathworks.com/discovery/convolutional-neural-network.html

CNN 구현

- 2 convolution layers with 32 and 64 filters each
- one fully-connected hidden layer of 1024 nodes with dropout
- output layer with softmax

```
#
# 이 부분 이전의 코드는 지난 시간과 완전히 동일하다.
# 단, load_image함수에서 img.ravel() 함수는 호출하지 않는다.
#

import tensorflow as tf

images_batch = tf.placeholder(dtype=tf.float32,
                              shape=[None, IMG_HEIGHT, IMG_WIDTH, NUM_CHANNEL])
```

CNN은 3채널 이미지를 1차원으로 flatten하지 않고 그대로 입력받는다.

```
labels_batch = tf.placeholder(dtype=tf.int32, shape=[None, ])
```

하나의 convolution layer를 생성하는 함수를 만들어 두자.

입력의 채널 수

이 layer의 이름(문자열)

사용할 필터의 크기(예: 3, 5 등)

출력 채널수 = 사용할 필터의 개수

이 layer에 대한 입력 데이터

```
def con_layer(input_tensor, filter_size, in_channels, out_channels, layer_name):

    with tf.variable_scope(layer_name):

        filt = tf.get_variable(name="filter",
                               shape=[filter_size, filter_size, in_channels, out_channels],
                               dtype=tf.float32)

        bias = tf.get_variable(name='bias', shape=[out_channels], dtype=tf.float32)

        pre_act = tf.nn.conv2d(input_tensor, filt, [1, 1, 1, 1], padding='SAME') + bias

        activated = tf.nn.relu(pre_act)

        return activated
```

이 scope 내부에서 생성되는 변수는 전부
'layer_name/var_name'의 형태를 갖는다.
따라서 이름이 동일한 변수의 생성을 방지한다.

가운데 두 값이
각각 가로, 세로 방향의 stride이다.
만약 stride=2로 하려면
[1, 2, 2, 1]로 설정하면 된다.

maxpooling과 dense layer를 생성하는 함수

window size

stride size

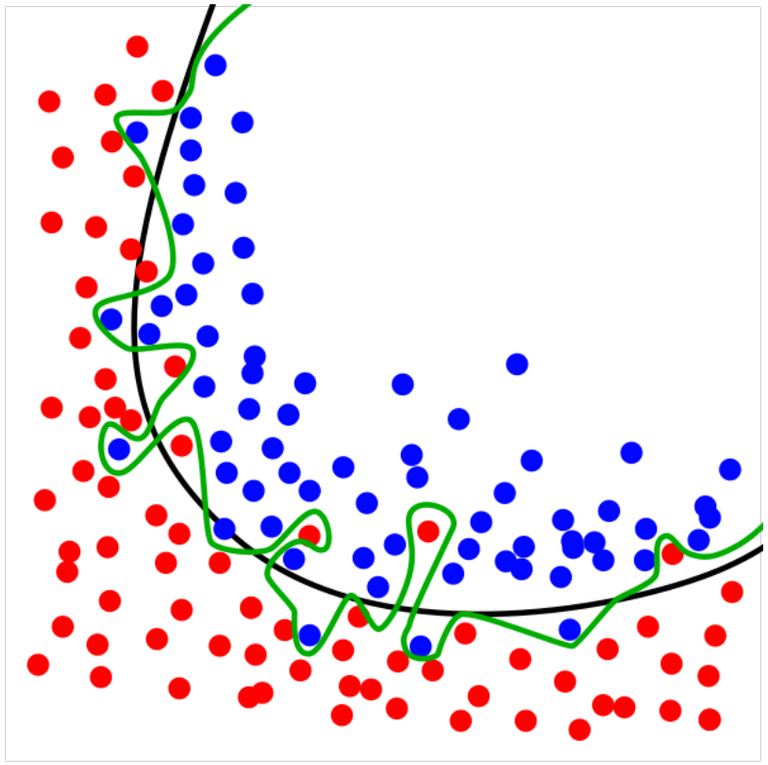
```
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

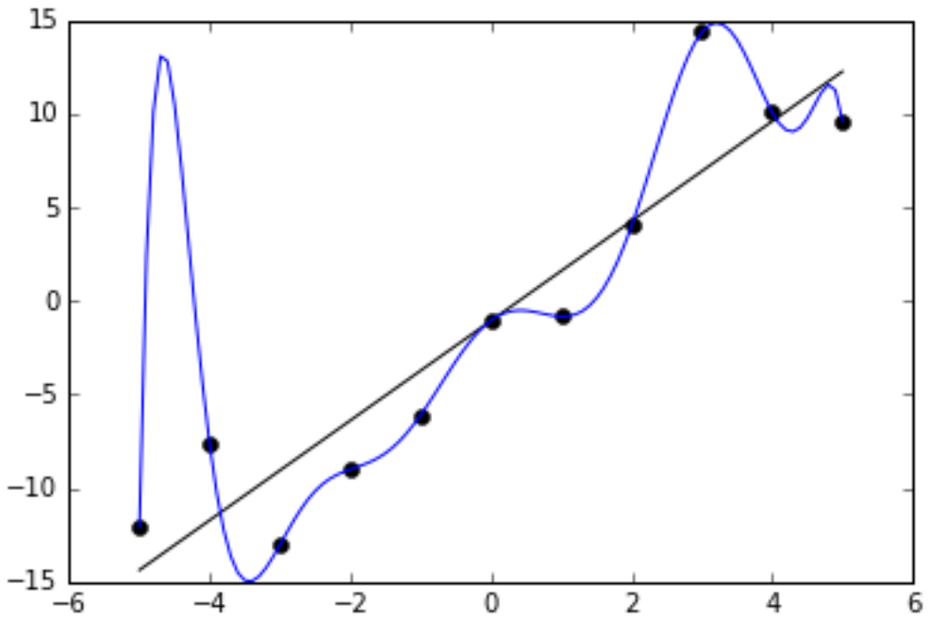
```
def dense_layer(input_tensor, input_dim, output_dim, layer_name, act=True):
    with tf.variable_scope(layer_name):
        weights = tf.get_variable(name="weight", shape=[input_dim, output_dim])
        biases = tf.get_variable(name='bias', shape=[output_dim])
        preact = tf.matmul(input_tensor, weights) + biases
        if act:
            return tf.nn.relu(preact)
        return preact
```

Build Layers

```
input_shape = images_batch.get_shape()[1:]

conv1 = con_layer(images_batch, 5, input_shape[2], 32, ‘con_layer1')
h_pool1 = max_pool_2x2(conv1)

conv2 = con_layer(h_pool1, 5, 32, 64, 'con_layer2')
h_pool2 = max_pool_2x2(conv2)

fc_size = input_shape[0]//(2**2) * input_shape[1]//(2**2) * 64

h_pool2_flat = tf.reshape(h_pool2, [-1, fc_size])

fc1 = dense_layer(h_pool2_flat, fc_size, 1024, 'dense1')
```

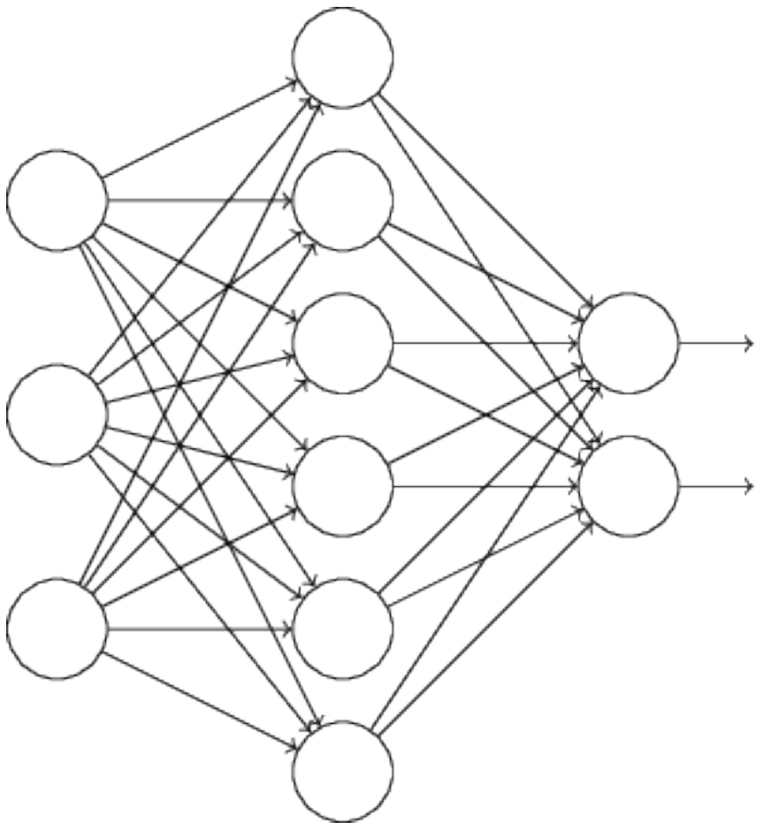
- Overfitting은 “특정한 데이터 집합에 지나치게 정확하게 맞춰짐으로써 오히려 새로운 추가 데이터에 잘 맞지 않게되고 따라서 예측력이 떨어지게 된 상태”를 의미한다.

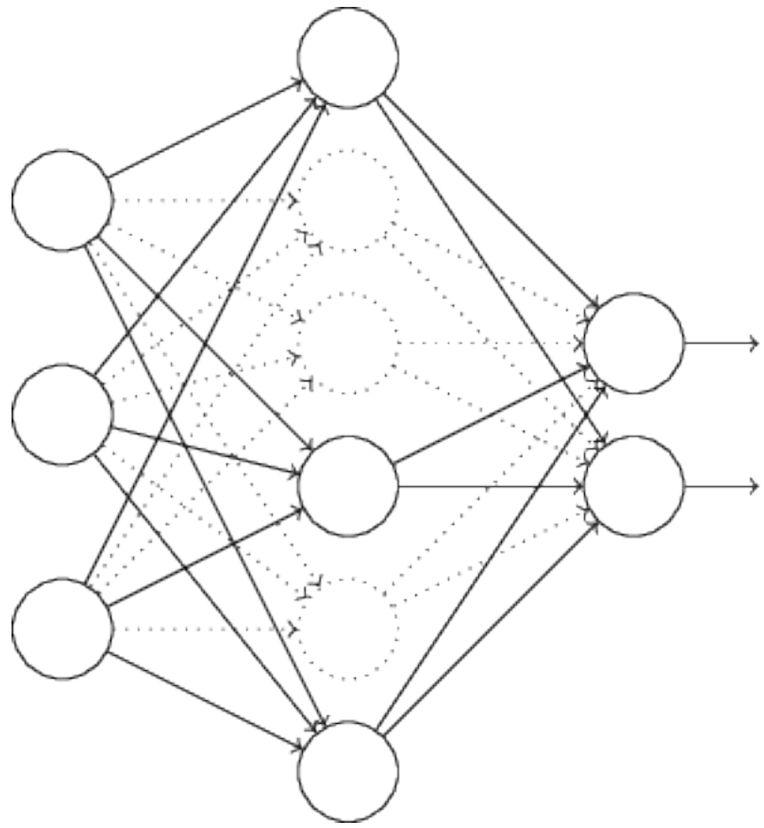




- **Dropout은 overfitting을 억제하기 위한 기법의 하나이다.**

training batch마다 랜덤하게 일정한 비율의
노드를 네트워크에서 제거하고 training 한다.
보통 fully connected layer에 적용한다.





마치 여러 개의 신경망을 동시에 training하고
나중에 weight들을 평균하는 것과 유사한 효과를 얻는다.

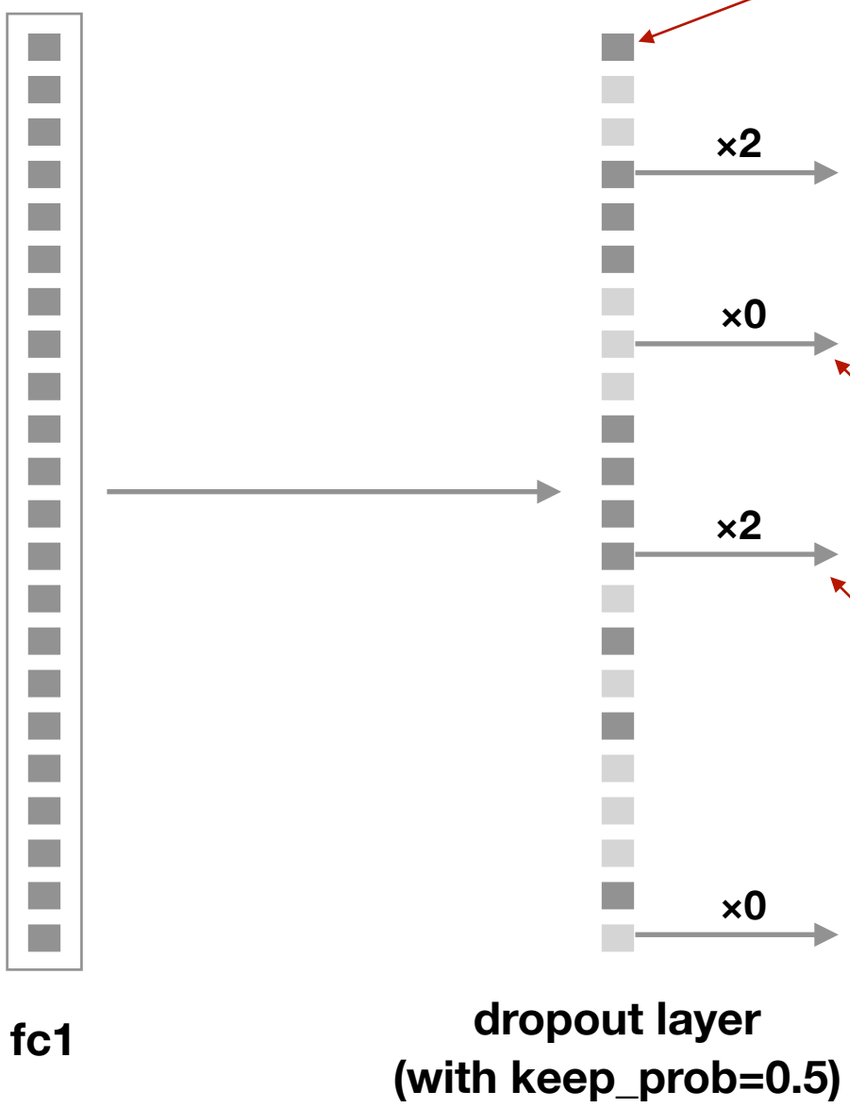
Dropout Layer

dropout 확률이다. 0.5라고 가정하자.

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(fc1, keep_prob)
```

- Dropout은 overfitting을 억제하기 위한 대표적인 기법들 중 하나이다.

training의 매 스텝마다 keep_prob의 확률로 살아 남을 노드들을 랜덤하게 선택한다.



선택되지 못한 노드의 출력에는 0을 곱한다.
즉 노드를 없애버리는 것과 같은 의미이다.

살아 남은 노드의 출력에는 1/keep_prob를 곱한다.

Prediction and Accuracy

```
y_pred = dense_layer(h_fc1_drop, 1024, NUM_CLASS, 'dense2', act=False)
class_prediction = tf.argmax(y_pred, 1, output_type=tf.int32)
correct_prediction = tf.equal(class_prediction, labels_batch)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Loss and Train_op

```
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y_pred,
                                                      labels=labels_batch)
loss_mean = tf.reduce_mean(loss)
train_op = tf.train.AdamOptimizer().minimize(loss_mean)
```

Visualize Graph

```
# To visualize graph in tensorboard
LOG_DIR = ‘tmp/logs'
writer = tf.summary.FileWriter(LOG_DIR)
writer.add_graph(tf.get_default_graph())
writer.flush()
```

- tmp/logs 디렉토리에 다음과 같은 이름의 이벤트 파일이 생성된다:
  `events.out.tfevents.{timestamp}.{hostname}`
- 터미널에서 다음의 명령으로 Tensorboard를 실행한다.(먼저 tfenv3.5를 activate한다.)
  `tensorboard --logdir=tmp/logs`
- 브라우저에서 `localhost:6006`으로 접속하면 그래프를 볼 수 있다.

Saver

```
saver = tf.train.Saver()
```

Training된 변수의 값을 파일로 저장하거나, 저장된 변수의 값을 읽어와서 학습을 지속할 수 있는 기능을 제공한다.

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

```
CONTINUE_TRAIN = False
if CONTINUE_TRAIN is True:
    states = tf.train.get_checkpoint_state(LOG_DIR)
    if states is not None:
        saver.restore(sess, LOG_DIR + os.sep + "model.ckpt")
```

저장된 변수값이 있는지 확인한다.

변수값이 model.ckpt라는 이름의 파일로 저장되어 있다고 가정한다.

```
iter_ = train_data_iterator()
for step in range(500):

    images_batch_val, labels_batch_val = next(iter_)

    accuracy_, _, loss_val = sess.run([accuracy, train_op, loss_mean],
                      feed_dict={
                          images_batch:images_batch_val,
                          labels_batch:labels_batch_val,
                          keep_prob: 0.5
                      })
    print('Iteration {}: ACC={}, LOSS={}'.format(step, accuracy_, loss_val))

print('Training Finished....')
```

training 동안에는 dropout rate를 0.5로 한다.

```
print('Test begins….')

TEST_BSIZE = 50
for i in range(int(len(test_features)/TEST_BSIZE)):

    images_batch_val = test_features[i*TEST_BSIZE:(i+1)*TEST_BSIZE]
labels_batch_val = test_labels[i*TEST_BSIZE:(i+1)*TEST_BSIZE]

loss_val, accuracy_ = sess.run([loss_mean, accuracy], feed_dict={
                      images_batch:images_batch_val,
                      labels_batch:labels_batch_val,
                      keep_prob: 1.0
                      })
print('ACC = {}, LOSS = {}'.format(accuracy_, loss_val))
```

Testing에서는 dropout하지 않는다.

학습된 변수값의 저장

학습된 변수의 값을 저장한다.

```
save_path = saver.save(sess, LOG_DIR + os.sep + ‘model.ckpt’)
print('Model saved in file: {}'.format(save_path))
```

- 학습된 변수값을 읽어와서 training을 지속한 후에 testing을 하면 갑자기 accuracy가 급상승한다.
- 진짜로 accuracy가 높아진 걸까? 아니다.
- 잘 생각해보면 이전 run에서 training에 사용되었던 데이터가 다음 run에서 testing에 사용되었기 때문임을 알수 있다.
- 이렇게 transferred learning을 할 때는 조심해야 한다. 즉, 처음부터 test data set을 따로 빼 두어야 한다.

과제

- 어떤 테스트 데이터가 잘못 분류되었는지, 그리고 무엇으로 잘못 분류되었는지 알고싶다. 또한 모든 클래스 쌍 (A, B)에 대해서 A에 속한 이미지가 B로 잘못 분류된 퍼센티지를 구하여 하나의 행렬로 출력 하려면 어떻게 해야 할까?
- Google은 할 수 없는 자신만의 작은 프로젝트를 시작해보면 어떨까? 예를 들면 가족의 얼굴을 구별한다든가...