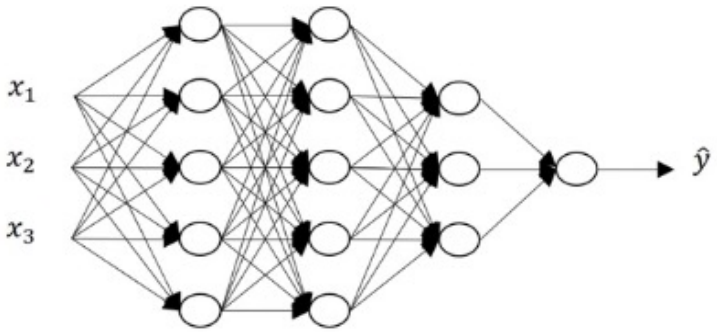
Parrot Deep Learning

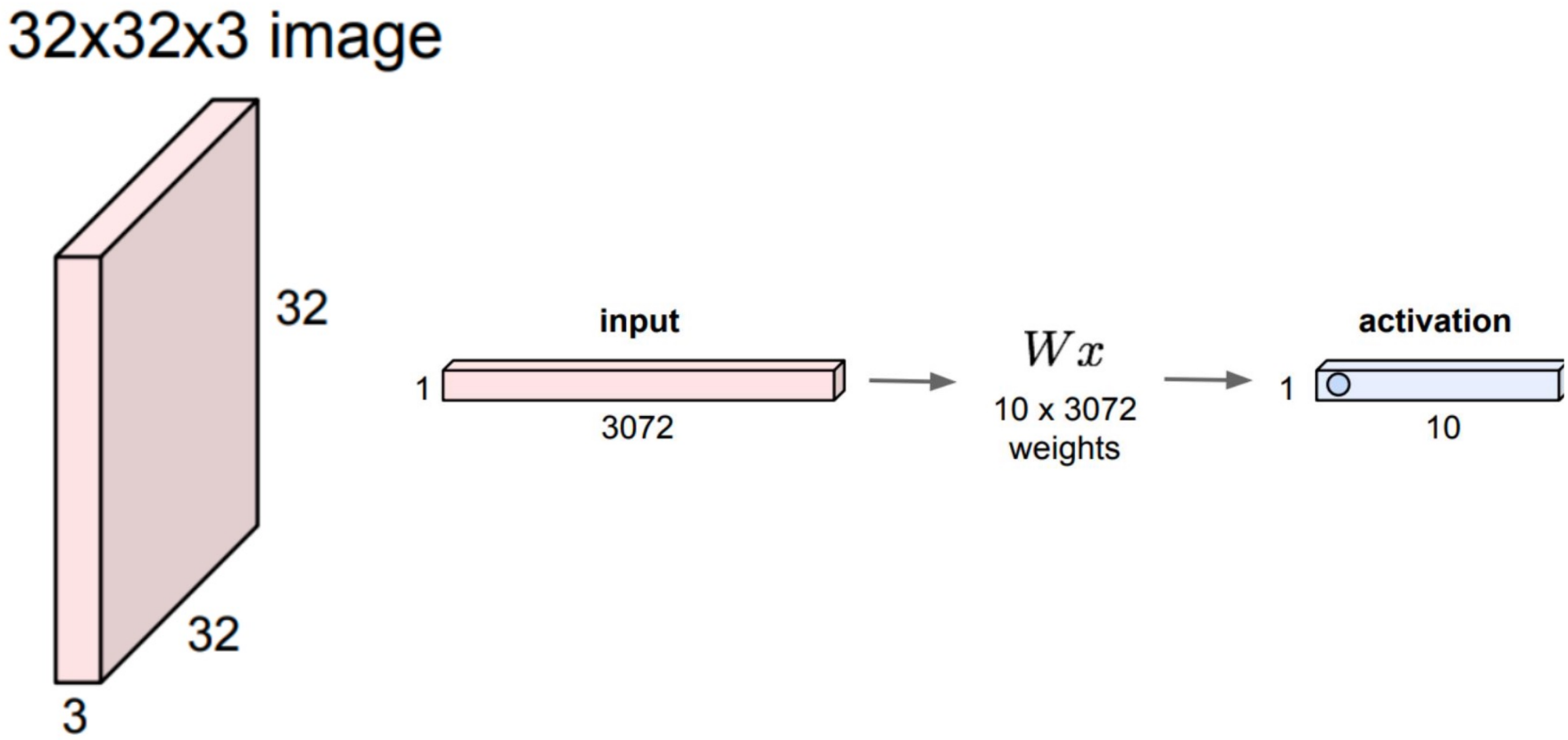
Session 05.
CNN, optimizer

*Why CNN ? – fc(fully connected)*의 문제



- 가중치가 너무 많다
- 공간구조를 활용하지 않는다.

*fc(fully connected)*의 문제 – 가중치가 너무 많다



*fc(fully connected)*의 문제 – 공간구조를 활용하지 않는다

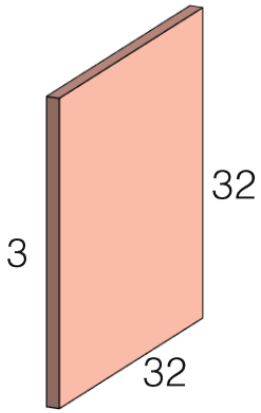
강아지를 인식할 때 어떻게 인식할까요?

이미지는 비슷한 위치를 가지고 있는
픽셀들은 공통된 특성을 가지고 있습니다.

*Why CNN ? – CNN*의 기본 아이디어

- **공간 정보를 활용한다 : 비슷한 위치를 가진 애들은 비슷한 가중치를 주고 싶다.**

CNN - filter



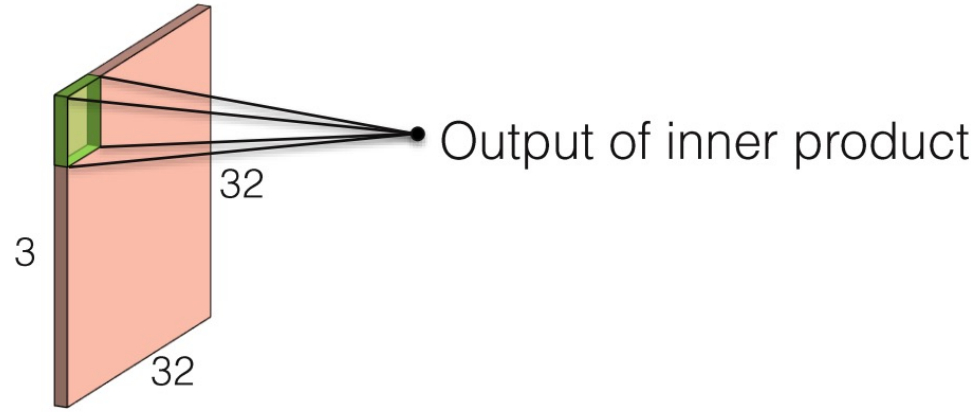
Input Image

weights 5x5x3

Filter

Width = 5
Height = 5
Channel = 3
Input의 channel == filter의 채널

CNN – convolution



Input Image

filter와 이미지의 내적을 구합니다.
쉽게 말하자면

CNN - convolution

Input

1	2	3	-2	5
5	6	7	3	2
4	3	2	5	1
0	2	3	4	3
1	-1	-2	3	2

filter

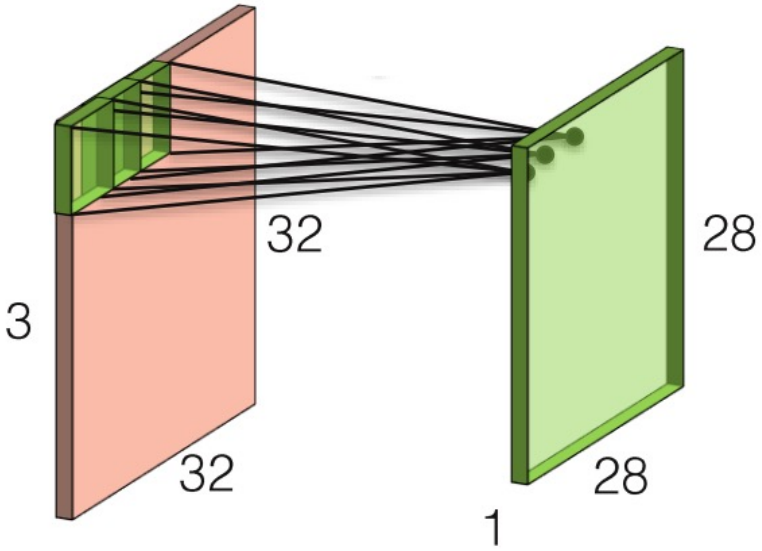
1	2	1
0	1	2
-1	2	3

1*(-1) + 2*2 + 3*1 +
5 * 0 + 6*1 + 7*2 +
4*(-1) + 3*2 + 2*3 =

output

36		

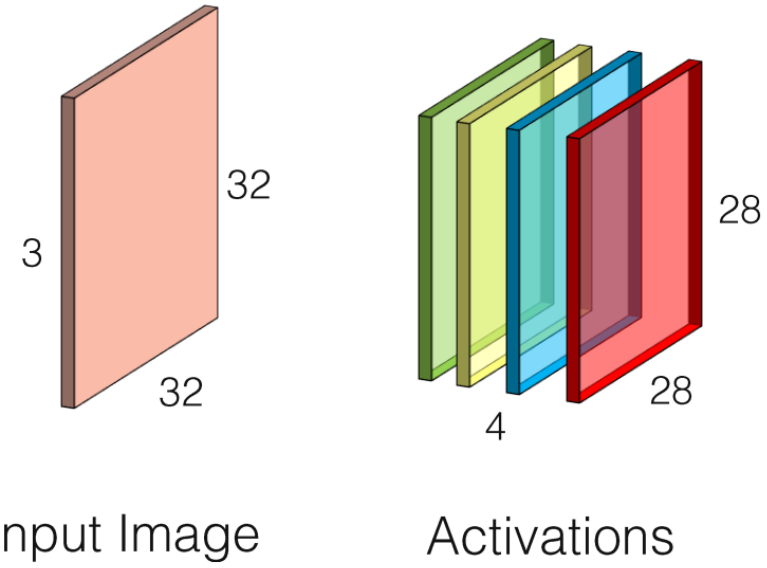
CNN – sliding window



이걸 쭉 돌아가면서 찍어줍니다.
그러면 output size는
28*28*1이 됩니다

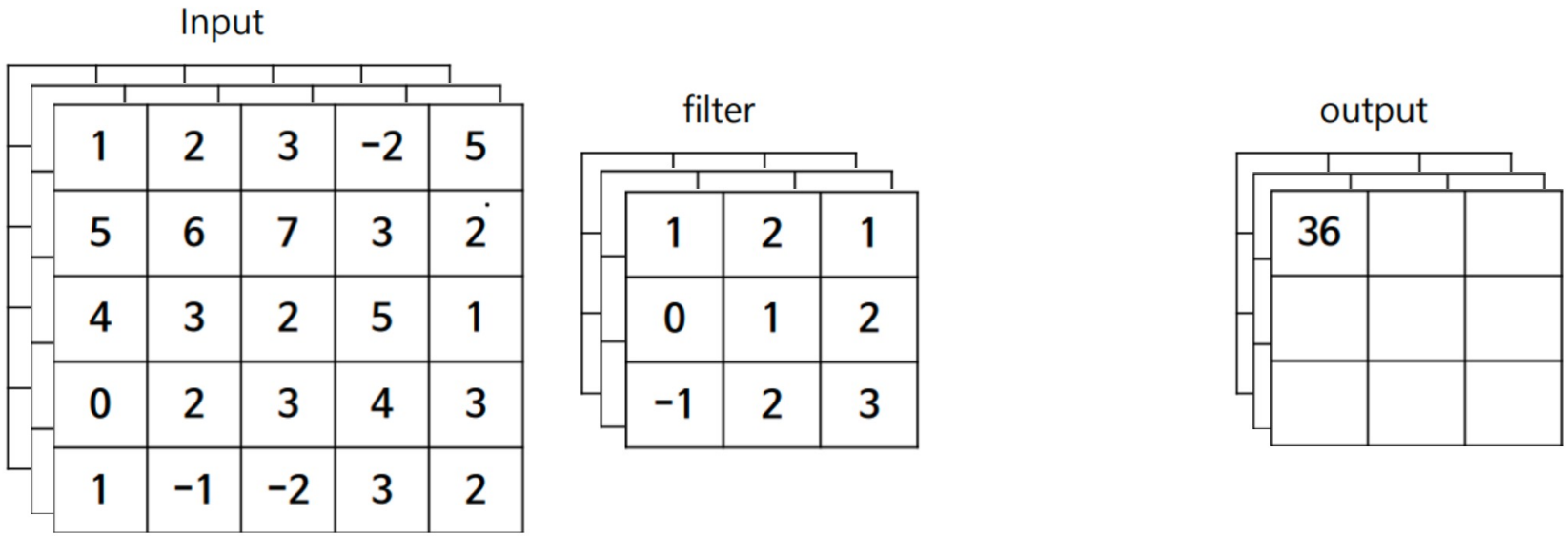
이 한장이 activation map입니다.
Activation map의 사이즈는
(N-F) + 1입니다.
N : input의 가로 세로
F : filte의 가로 세로

CNN – multi filters



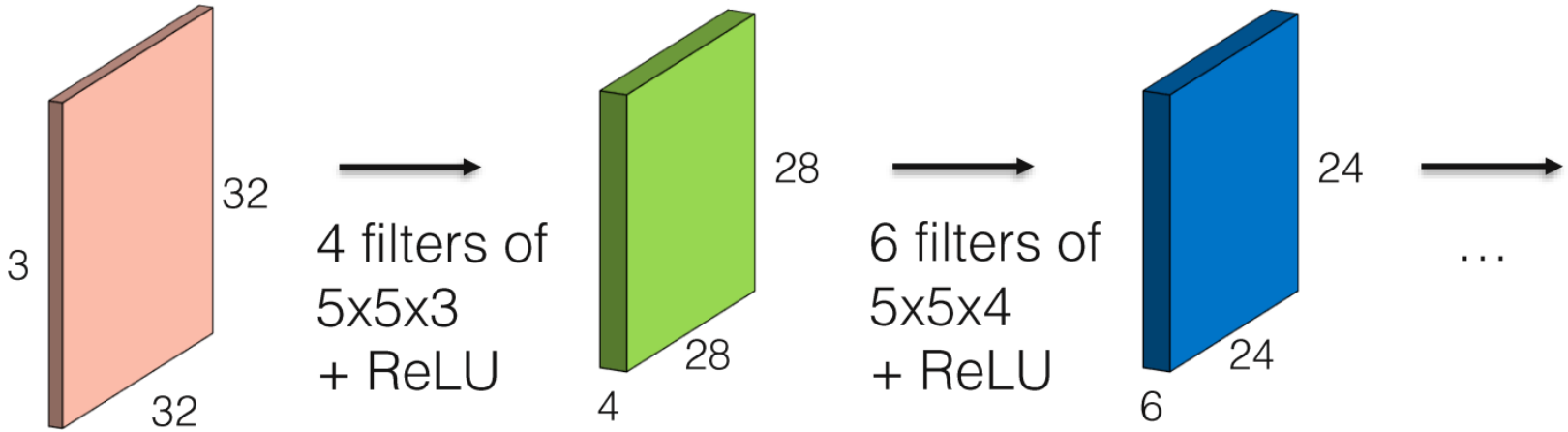
Activation map이 여러개라는건
Filter가 여러개라는 뜻입니다.

CNN – multi filters



Filter 하나 당 activation map 한 장입니다

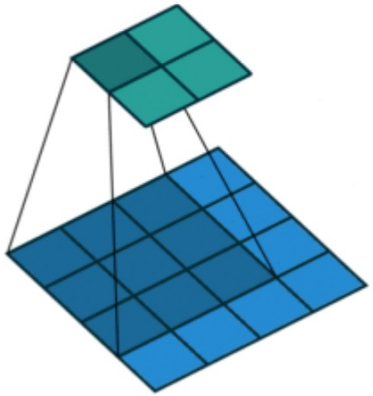
CNN – multi filters



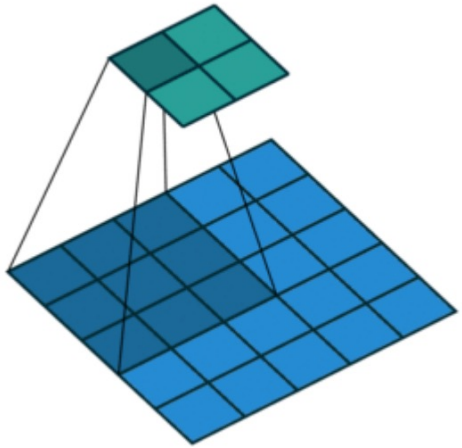
filter의 채널과 filter의 개수는 별개입니다.
Parameter의 개수?
(5*5*3+1)*4 = 304
(5*5*4+1)*6 = 606
bias

CNN – strides

stride



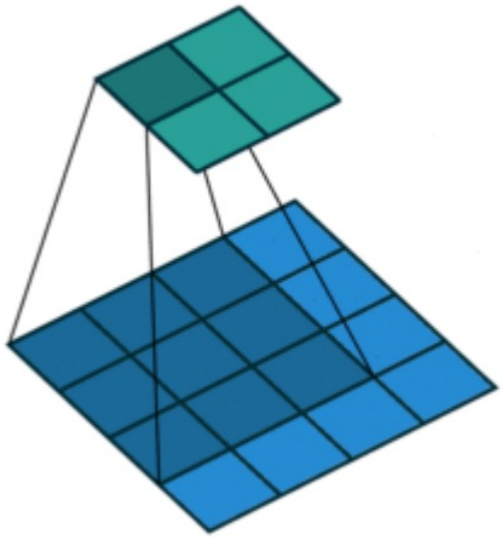
stride = 1



stride = 2

Stride란 움직일 때 몇칸 움직이냐입니다
output = (N – F)/S + 1 입니다

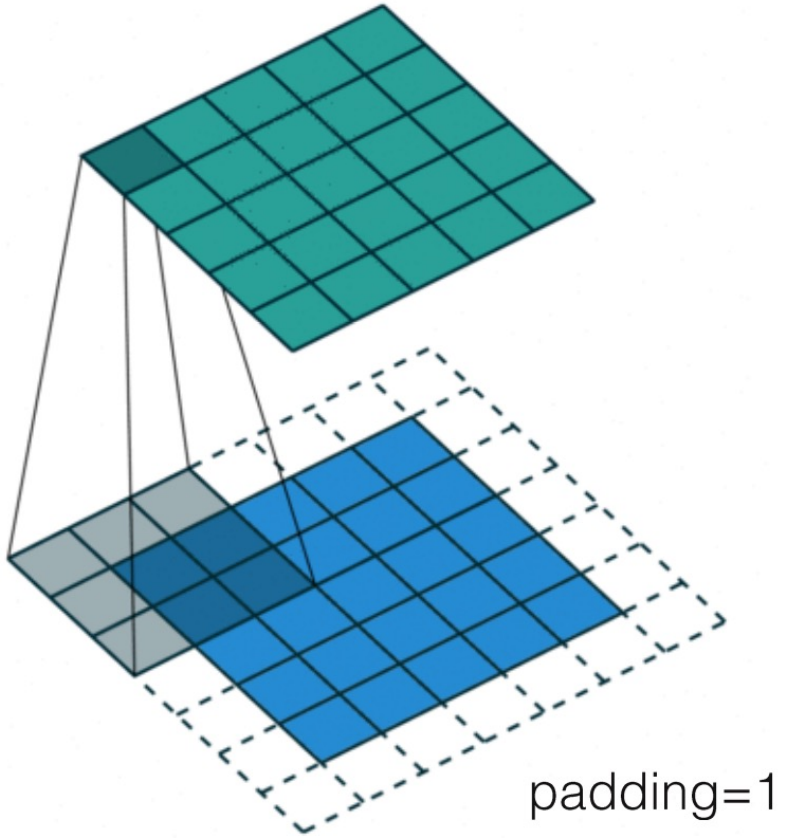
CNN – padding



이렇게 찍어낼 때의 문제점?

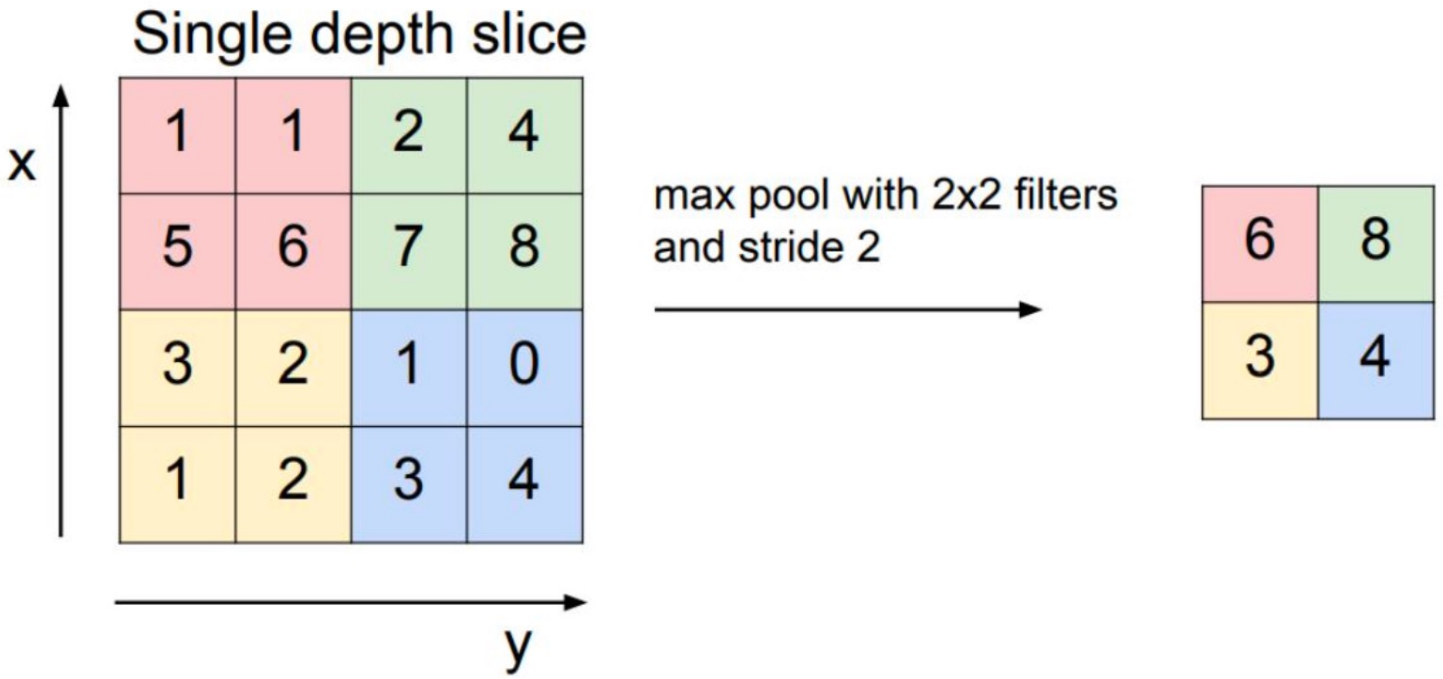
1. **가장 자리는 한번만 참조된다**
2. **이미지의 사이즈가 너무 금방 줄어든다**

CNN – padding



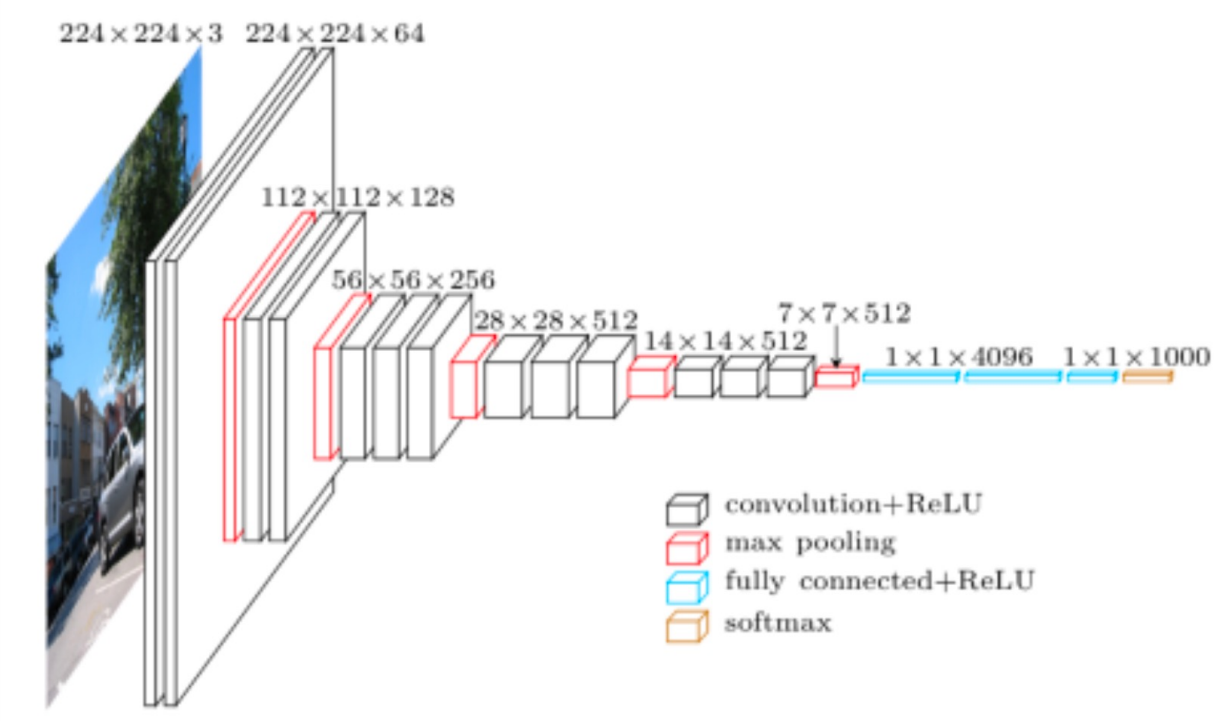
가장 자리를 0으로 채운다 (zero padding)
output size = (N – F + 2*P)/S + 1

CNN – pooling layer



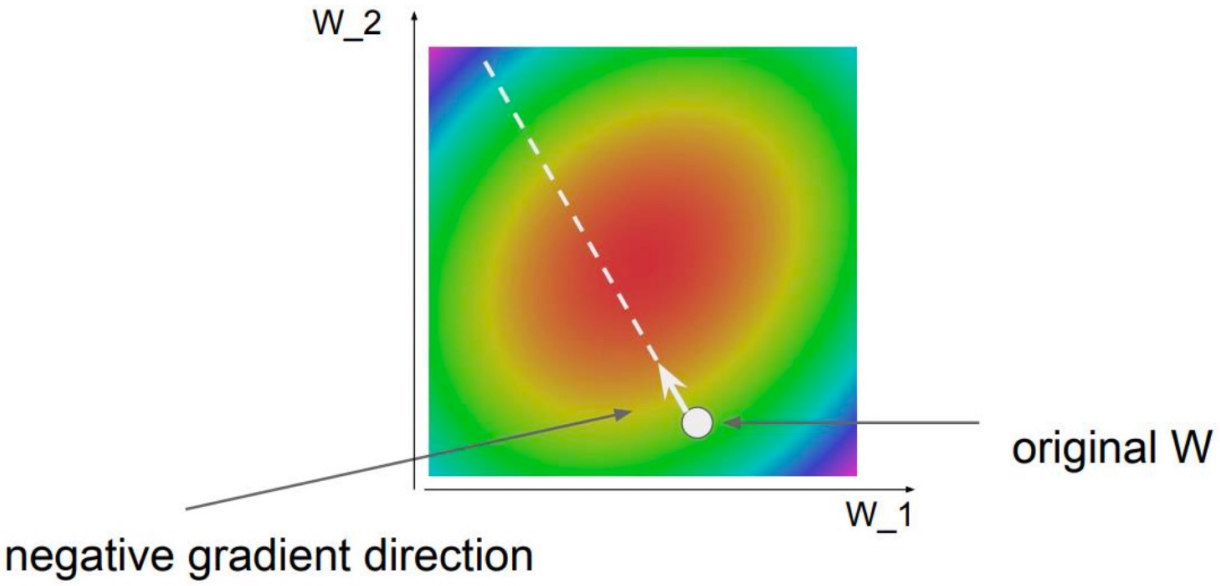
pooling layer의 파라미터 수는 0입니다
max pooling, average pooling 등이 있습니다

CNN – Overview



결국 마지막에는 fc layer를 추가해야합니다

Optimizer – Gradient Descent



Optimizer – SGD

가장 간단하고 직관적입니다.
Batch 단위로 학습합니다.
(MSGD)

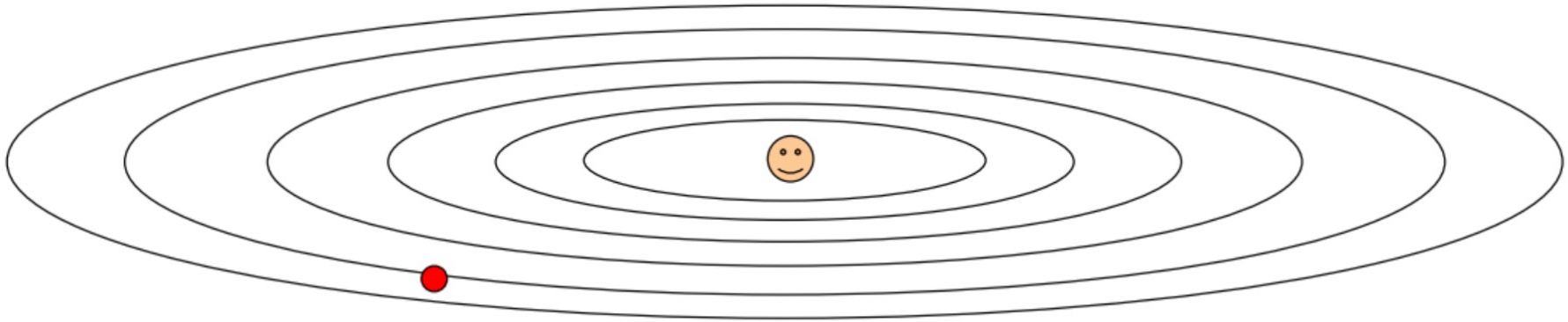
문제점
1. **학습이 비효율적이다**
2. **Saddle point, global minima에 취약**

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

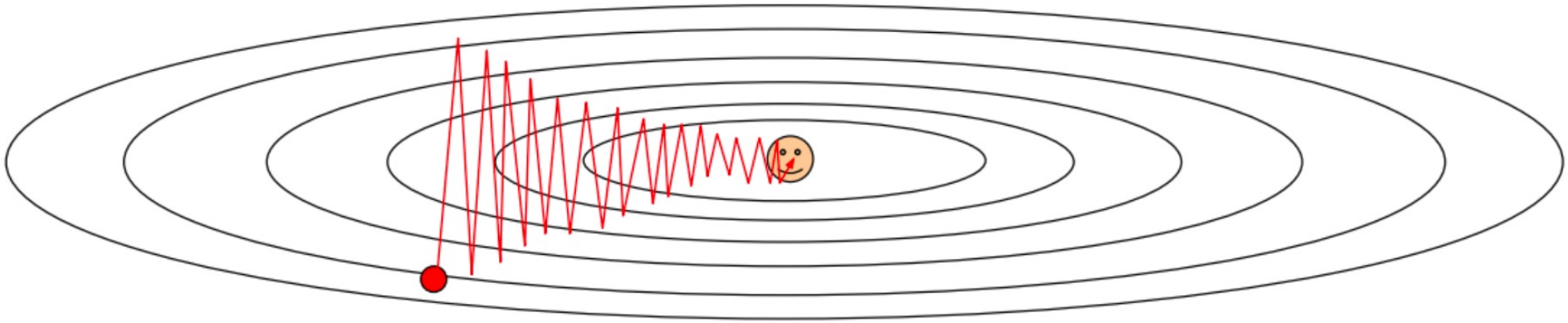
Optimizer – SGD



등고선이라고 생각해볼게요
웃는 지점으로 가는게 목표입니다.

가로축과 세로축이 가중치라고 했을 때,
세로축의 gradient는 높고, 가로축의 gradient는 낮아요

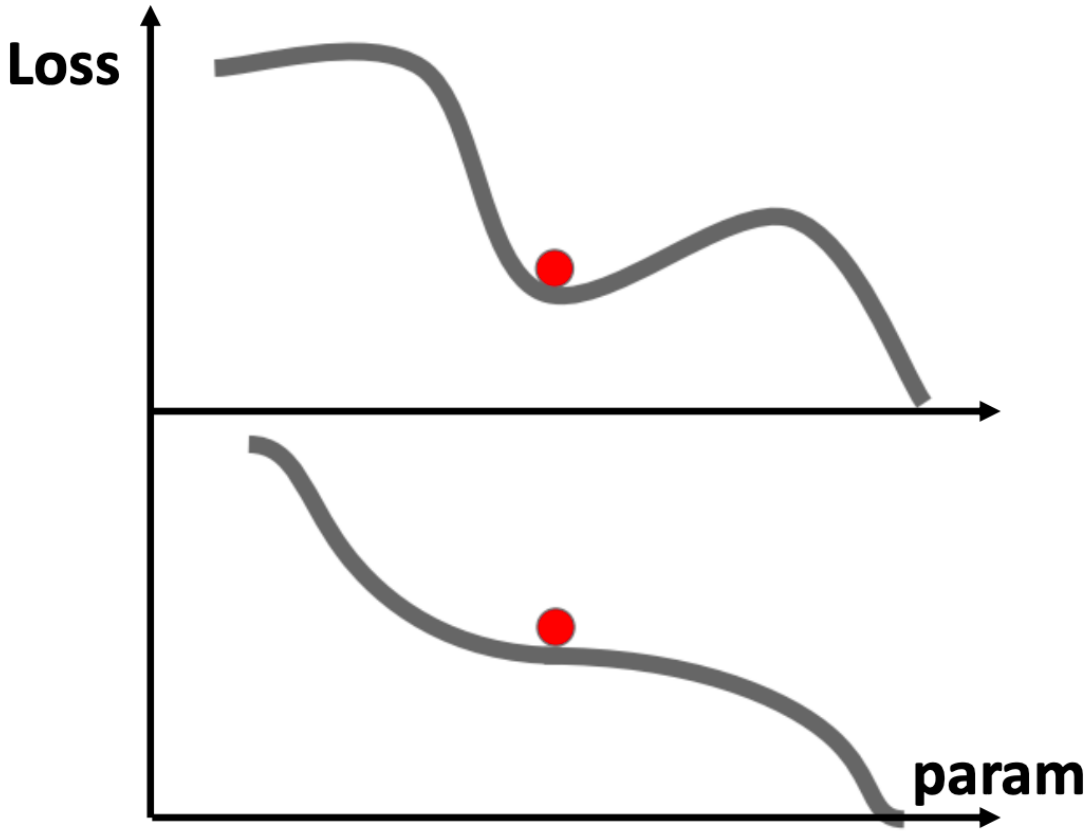
Optimizer – SGD



SGD – Saddle point, local minima

Local minima의 grad는 0입니다
학습이 멈추겠죠

Saddle point의 주변에서
학습이 느려지고 멈춥니다.



Optimizer – SGD with Momentum

velocity : 관성
기존의 gradient를 축적하는 변수

보통 rho : 0.9 ~ 0.99
-> velocity가 과도하게 커지는 것을 방지

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

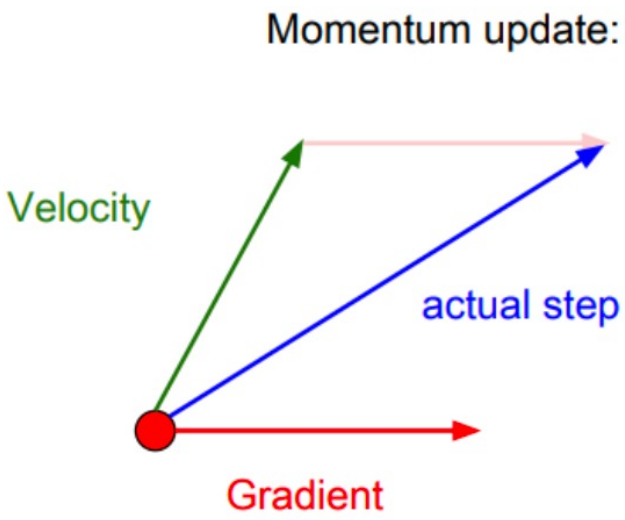
$$x_{t+1} = x_t - \alpha v_{t+1}$$

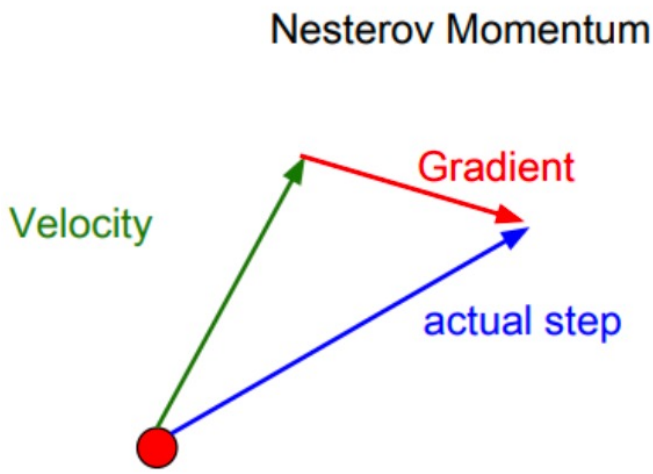
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

Optimizer – Nesterov Momentum

Momentum에 따라 움직이고
거기서 다시 grad를 구해서 움직입니다
-> momentum보다 수렴이 빠릅니다

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





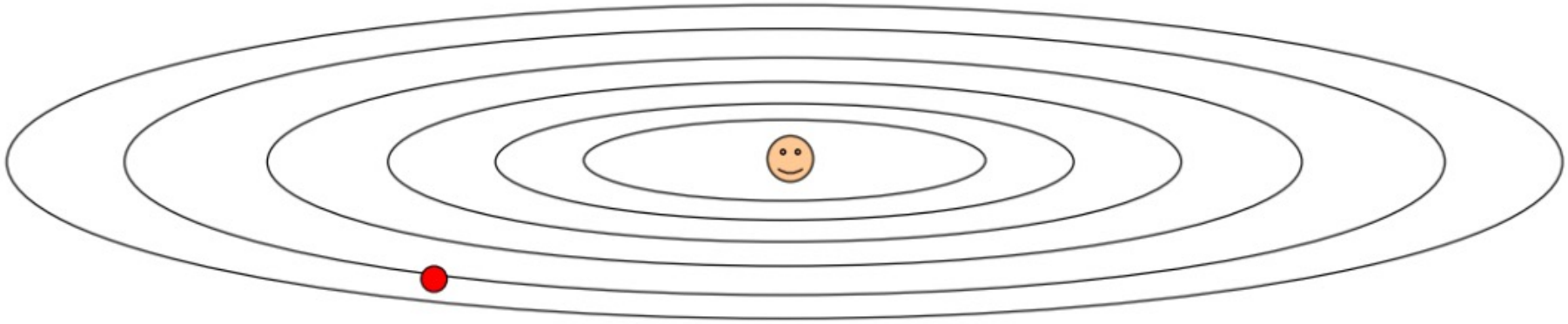
Optimizer – Adagrad

$$G_t = G_{t-1} + \nabla_{\theta_t} J(\theta_t) \odot \nabla_{\theta_t} J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{(G_t + \epsilon)}} \odot \nabla_{\theta_t} J(\theta_t)$$

Gradient의 squared sum을 저장
이들을 sqrt값으로 gradient를 나눠준 뒤 update 합니다

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Optimizer – Adagrad : what happens?



High gradient -> low update
Low gradient -> high update
update 할수록 squared sum이 커지는 문제 발생->stuck

Optimizer- RMSprop

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Squared를 decay하며 축적 : 모멘텀과 유사 (exponentially weighted average)

RMSProp

Decay rate = 0.9~0.99

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Optimizer – Momentum – Squared Grad : Adam

Momentum & Squared Grad 둘다 쓰면되지

하지만 이때 문제가 발생함
첫 번째 step을 생각하면 second moment = 0
이때 초기에 second moment가 0에 가까운 값
-> 초기에 학습폭이 너무 크다

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\theta_t} J(\theta_t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_{\theta_t} J(\theta_t) \odot \nabla_{\theta_t} J(\theta_t)$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

beta1 = 0.9
beta2 = 0.999

Momentum

AdaGrad / RMSProp

Optimizer – Adam + Bias Correction

Beta = 0.9, t가 1일 때? ->second_unbiased = second_moment*10
t가 커지면(학습이 진행되면) -> second_unbiased = second_moment*1
결국 second moment를 조정해주는 역할

```
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 / RMSProp

Optimizer

지금까지의 내용은 정말 단순한 abstraction 입니다.
딥러닝은 모두 행렬 혹은 벡터 단위의 연산입니다.
따라서 실제 optimizer를 제대로 이해하려면 Jacobian matrix를 이해해야합니다

관심있으신 분들은 찾아보셔도 좋을 것 같습니다.

QnA

- 질문 있으신가요?