[4차시 수업]

1. Advanced Optimizer than SGD
- SGD
- 액티베이션 함수(activation function) 소개

2. CNN(컨벌루션 뉴럴네트워크)
- 실습; Keras_CNN(Convolution_Neural_Network)_예제.ipynb
- 실습: CNN.ipynb

3. CNN(컨벌루션 뉴럴네트워크)
- 실습: CNN을 사용한 MNIST 이미지 인식
https://www.youtube.com/watch?v=7F3BYD5N8XE

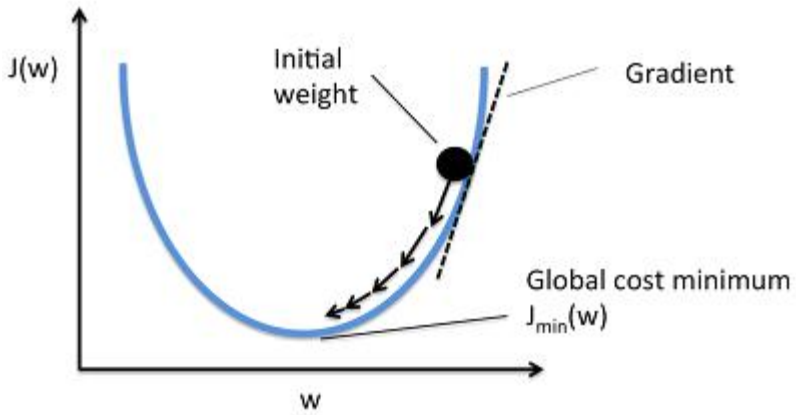
- 실습: 인공지능 웃음 판독기 - Python, Deep Learning
https://www.youtube.com/watch?v=GrN1tKjVBM8

- 영상 수업 관련 OT

[1교시]

1. Advanced Optimizer than SGD

1) Neural Network의 Weight를 조정하는 과정에 보통 Gradient Descent방법을 사용 (Gradient Descent란 네트워크에서 내놓는 결과 값과 실제 값 사이의 차이를 정의하는 Loss Function의 값을 최소화하기 위해 기울기를 이용하는 것)



문제는 최적 값을 찾아 나가기 위해서 한 칸 전진할 때마다 **모든 데이터 셋을 넣어주어야** 하기 때문에 **학습이 굉장히 오래 걸리는 문제가 발생**하게 된다는 것.
그래서 더 빠른 Optimizer로 나온 것 중 하나가 **Stochastic Gradient Descent**이다.

- Gradient Descent를 식으로 보면

$$\text{Error}_{(m,b)} = \frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b))^2$$

N : 데이터 샘플들 갯수
Yi : 데이터 샘플 중 i 번째의 y 값
mxi + b : 1차직선의방정식의 i 에서 y 값(기울기)
Error : 비용 (오차)
m,b: 미지수 m 과 b, 미지수를 조절해서 비용(Error) 값이 최소가 되도록 만들어 나간다.

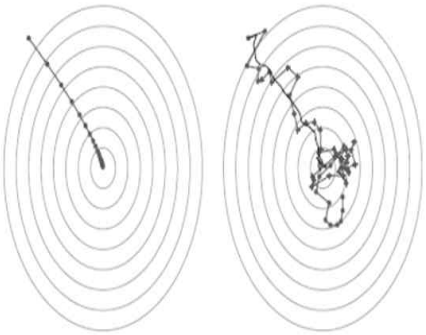
2) Loss Function을 계산할 때 전체 Train-Set을 사용하는 것을 **Batch Gradient Descent**라고 한다.
- Batch Gradient Descent 단점: step을 내딛을 때, 전체 데이터에 대해 Loss Function을 계산해야 하므로 너무 많은 계산 양을 필요로 한다.

그래서 Stochastic Gradient Desenct(SGD)방법이 나오게 됨

- Loss Function을 계산할 때, 전체 데이터(Batch) 대신 **일부 데이터의 모음(Mini-Batch)**를 사용하여 Loss Function을 계산한다.
- Batch Gradient Descent보다 다소 부정확할 수는 있지만, **계산 속도가 훨씬 빠르기** 때문에 같은 시간에 더 많은 step을 갈 수 있으며, 여러 번 반복할 경우 Batch 처리한

결과로 수렴한다.

- Batch Gradient Descent에서 빠질 **Local Minima에 빠지지 않고** 더 좋은 방향으로 수렴할 가능성도 높다.
- SGD를 변형시켜 성능을 향상 시킨 알고리즘들에 Momentum, NAG, Adagrad, AdaDelta, RMSprop 등이 있다.



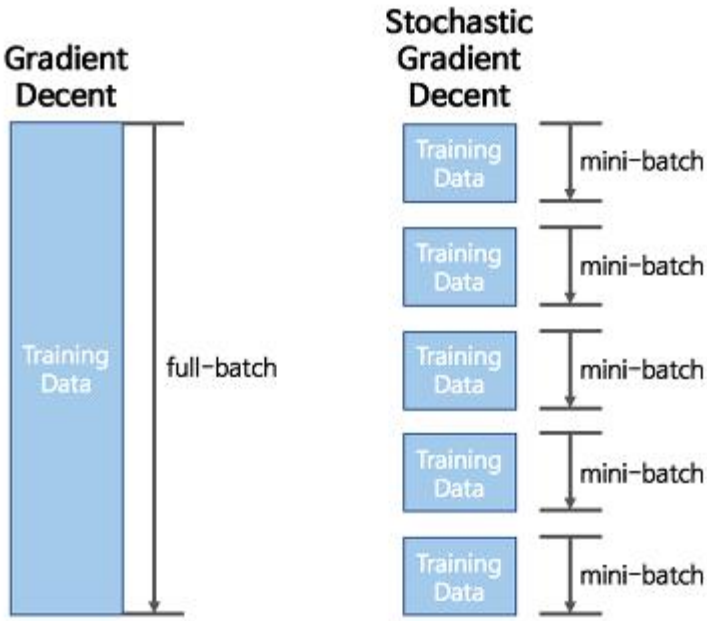
<BGD, SGD>

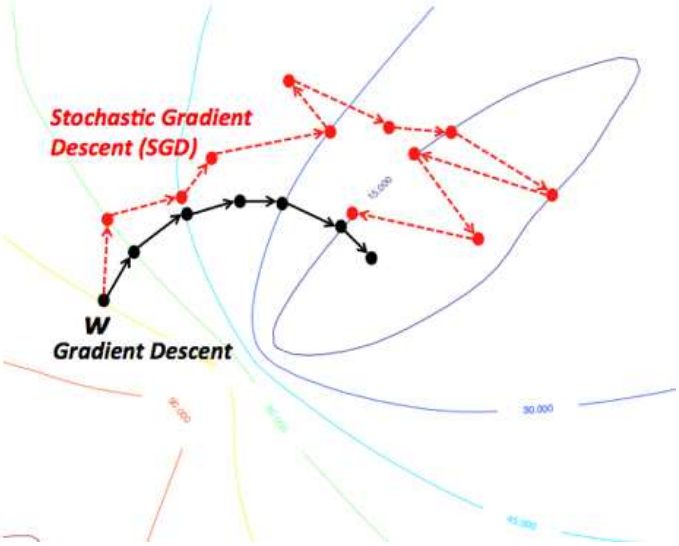
✓ Gradient Descent를 전체 데이터(Batch)가 아닌 일부 데이터의 모음(Mini-Batch)를 사용하는 방법

✓ BGD(Batch Gradient Descent)는 하나의 step을 위해 전체 데이터를 계산 하므로 계산량이 많음

✓ SGD(Stochastic Gradient Descent)는 Mini-Batch를 사용하여 다소 부정확할 수는 있지만 계산 속도가 훨씬 빠르기 때문에, 같은 시간에 더 많은 Step을 나아갈 수 있음

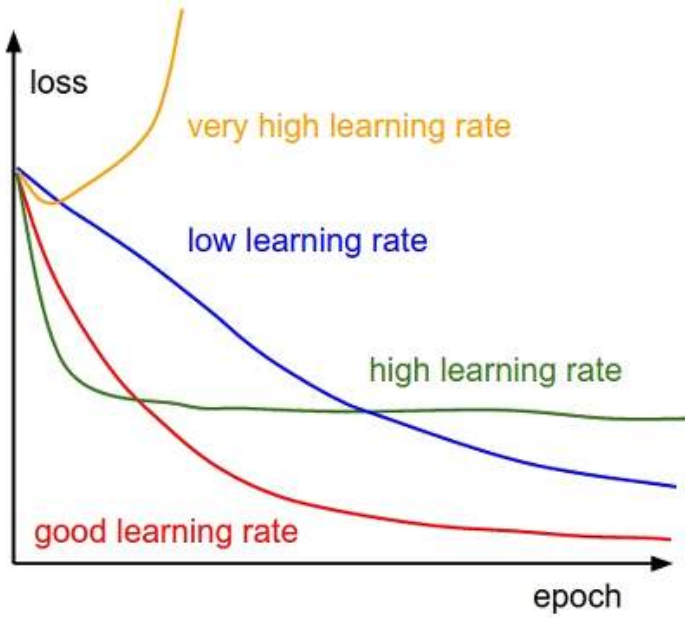
✓ Local Minima에 빠지지 않고 Global Minima에 수렴할 가능성이 더 높음





◈ GD
- 모든 데이터를 계산 한다 => 소요시간 1시간
- 최적의 한 스텝을 나아간다.
- 6 스텝 * 1시간 = 6시간
- 확실한데 너무 느리다.

◈ SGD
- 일부 데이터만 계산 한다 => 소요시간 5분
- 빠르게 전진한다.
- 10 스텝 * 5분 => 50분
- 조금 헤메지만 그래도 빠르게 간다!



- SGD 식으로(수식은 경사 하강법과 동일)

$$W(t+1) = W(t) - \alpha \frac{\partial}{\partial w} Cost(w)$$

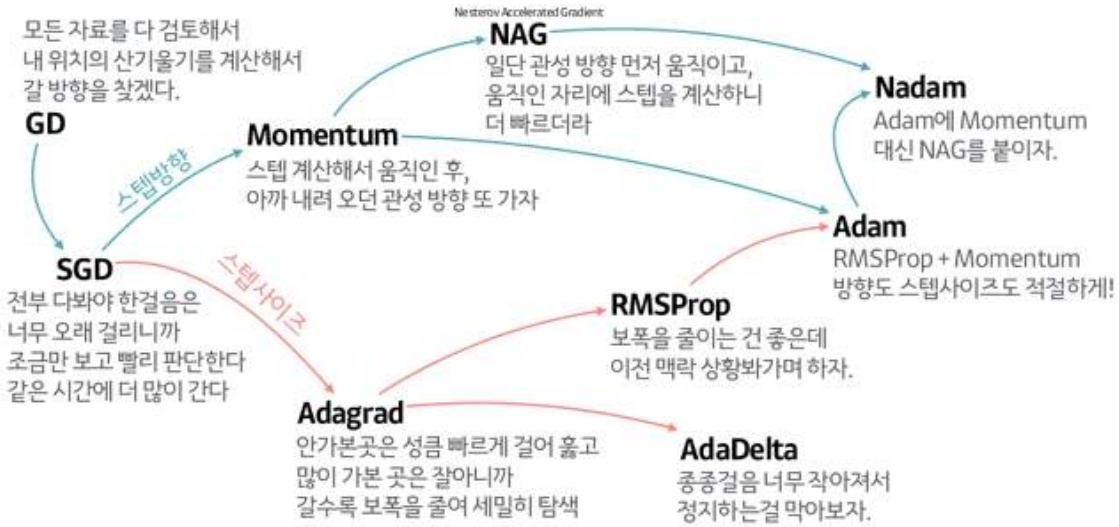
- Cost(w)에 사용되는 x: 입력 데이터의 수가 전체가 아닌 **확률적으로 선택된 부분**이 사용
- α(알파)는 Learning Rate, η(에타)와 동일한 개념

파이썬 소스 코드

```
1 weight[i] += - learning_rate * gradient
```

Keras 소스 코드

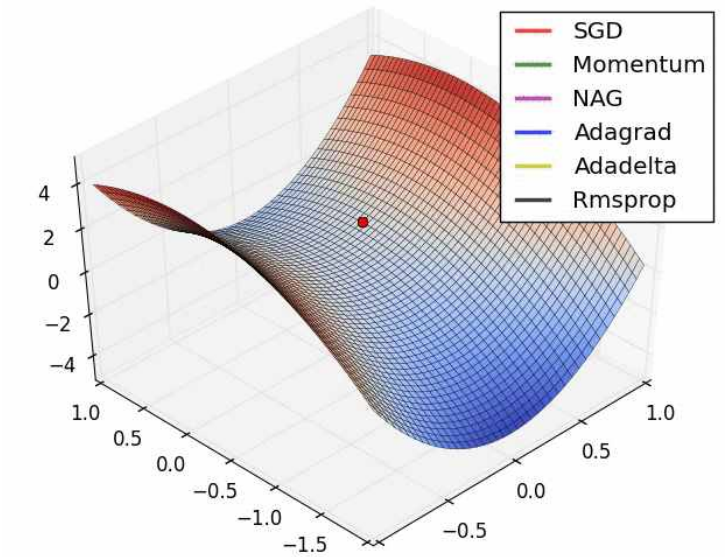
```
1 keras.optimizers.SGD(lr=0.1)
```



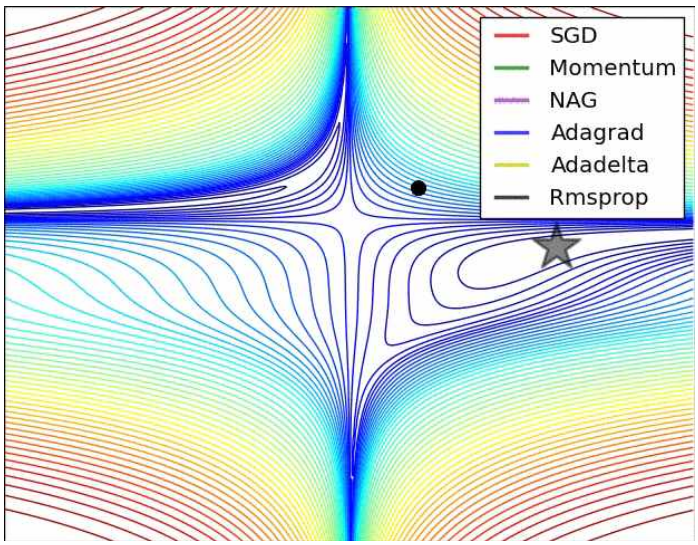
(Optimizer 계보)

Stochastic Gradient Descent를 통해서 빠르게 찾을 수 있도록 발전
그러나
- 최적의 값을 찾아가는 방향이 뒤죽 박죽이고,
- 한 스텝 나아가기 위한 사이즈를 정하기 어렵다.

방향과 스텝 사이즈를 고려하는 새로운 Optimizer들이 많이 나왔다. **방향성: Momentum, NAG** **스텝 사이즈: Adagrad, RMSPop, AdaDelta** **방향성 + 스텝사이즈: Adam, Nadam**



(Gradient Descent Optimization Algorithms at Long Valley)



(Gradient Descent Optimization Algorithms at Beale's Function)

- 위의 그림들은 각각 SGD 및 SGD의 변형 알고리즘들이 최적값을 찾는 과정을 시각화한 것
- 빨간색이 SGD 알고리즘
- Momentum, NAG, Adagrad, AdaDelta, RMSprop 등은 SGD의 변형
- **SGD는 다른 알고리즘들에 비해 성능이 월등하게 안 좋다.**

(다른 알고리즘들 보다 이동속도가 현저하게 느릴 뿐만 아니라, 방향을 제대로 잡지 못하고 이상한 곳에서 수렴하여 이동하지 못하는 모습도 관찰)

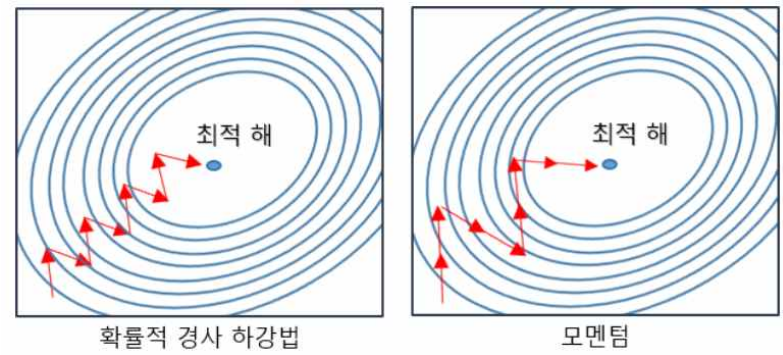
- **따라서 단순한 SGD를 이용하여 네트워크를 학습시킬 경우 네트워크가 상대적으로 좋은 결과를 얻지 못할 것이라고 예측**

※ SGD 코드 참조;

https://colab.research.google.com/drive/17iTEhaFUu0pXe-6_7ODipYBVxGgEmOAq#scrollTo=IlKkFm5U2EIj

2) Momentum

- Gradient Descent를 통해 이동하는 과정에 일종의 **관성을 더해** 주는 것이다.
- 경사 하강법과 마찬가지로 매번 기울기를 구하지만, 가중치를 수정하기 **전 이전 수정 방향(+,-)를 참고**하여 같은 방향으로 일정한 비율만 수정되게 하는 방법
- 수정이 양(+) 방향, 음(-) 방향 순차적으로 일어나는 **지그재그 현상이 줄어들고**, 이전 이동 값을 고려해서 일정 비율만큼 다음 값을 결정하므로 관성의 효과를 낼 수 있다.
- 현재 Gradient를 통해 이동하는 방향과는 별개로, **과거에 이동했던 방식을 기억**하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식이다.



- 수식으로 표현하면

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

vt; time step t에서의 이동 벡터

v; 속도(velocity)

ɣ(감마); momentum의 계수(0~1사이의 값), 무한이 커지지 않게 한다.

η(에타); learning Rate, step size(보통 0.01~0.001의 값)

J(θ); Loss function

η∇θJ(θ); 기울기(gradient)

θ; 파라미터들(w1,w2,...wn)

※ ∇: 델(Del), 나블라(nabla), 미분기호

파이썬 소스 코드

```
v = m * v - learning_rate * gradient
weight[i] += v
```

Tensorflow 소스 코드

```
optimize = tf.train.MomentumOptimizer(learning_rate=0.01,momentum=0.9).minimize(loss)
```

Keras 소스 코드

```
keras.optimizers.SGD(lr=0.1, momentum=0.9)
```

3) Nesterov Accelerated Gradient (NAG)

- Momentum 방식에서는 이동 벡터 vt를 계산할 때 현재 위치에서의 gradient와 momentum step을 독립적으로 계산하고 합친다.
- 반면, NAG에서는 **momentum step을 먼저 고려**하여, momentum step을 먼저 이동했다고 생각한 후 그 자리에서의 gradient를 구해서 gradient step을 이동한다.

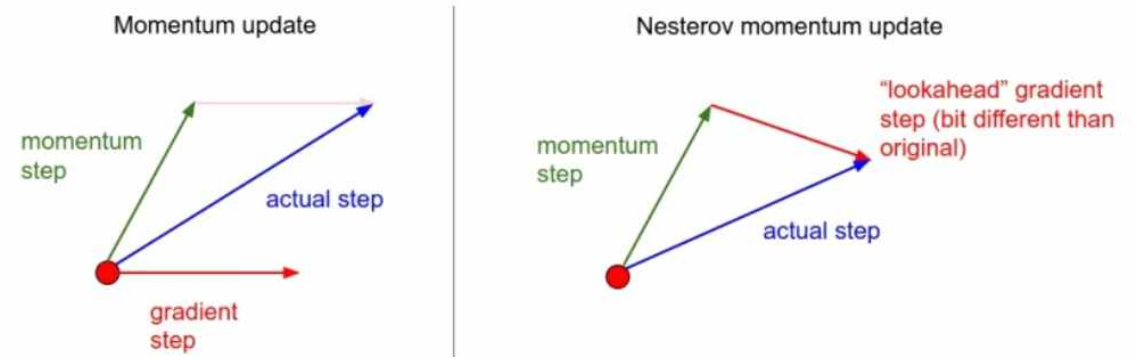
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

θ: 원래의 θ

γvt-1: 새로운 θ

(θ - γvt-1) : 원래의 θ에서 새로운 θ를 뺀다.



- Momentum 방식의 경우 멈춰야 할 시점에서도 관성에 의해 훨씬 멀리 갈수도 있다는 단점
- NAG 방식은 일단 모멘텀으로 이동을 반 정도 한 후 어떤 방식으로 이동해야할지를 결정한다. 따라서 **Momentum 방식의 빠른 이동에 대한 이점은 누리면서도, 멈춰야 할 적절한 시점에서 제동**을 거는 데에 훨씬 용이

- 파이썬 소스 코드

```
v = m * v - learning_rate * gradient(weight[i-1]+m*v)
weight[i] += v
```

- Tensorflow 소스 코드

```
optimize =
tf.train.MomentumOptimizer(learning_rate=0.01,momentum=0.9,use_nesterov=True).minimize(loss)
```

- Keras 소스 코드

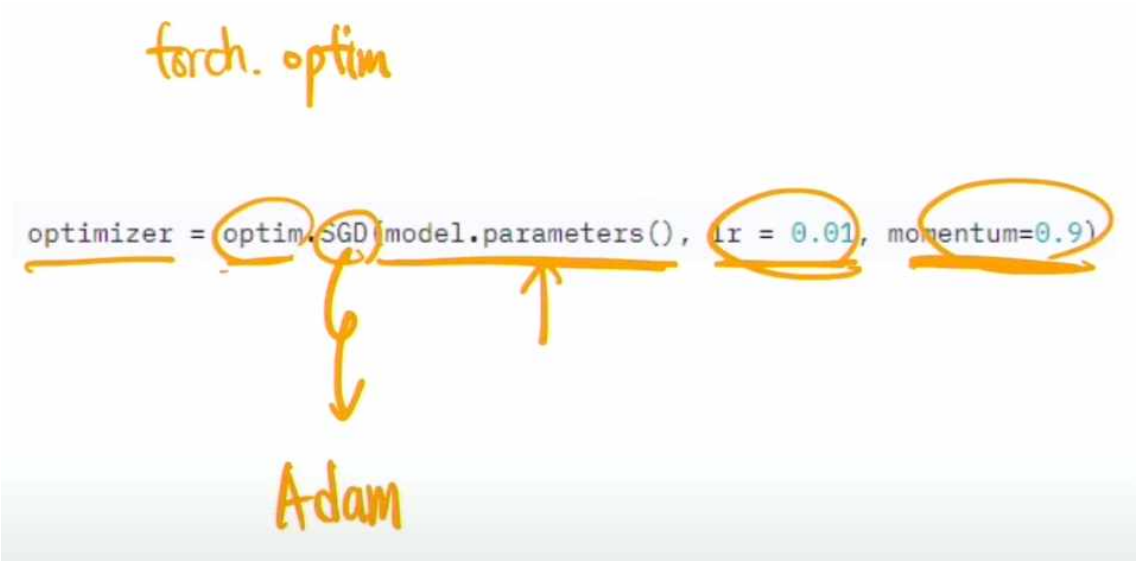
```
keras.optimizers.SGD(lr=0.1, momentum=0.9, nesterov=True)
```

4) 이외 Adam(Adaptive Moment Estimation, 아담), AdaDelta(Adaptive Delta,

아다델타), RMSProp, Summing up등 참조;
http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html

https://twinw.tistory.com/247

- 파이토치 코드



2. 활성화함수(Activation Function)

1) 활성화 함수가 필요한 이유:

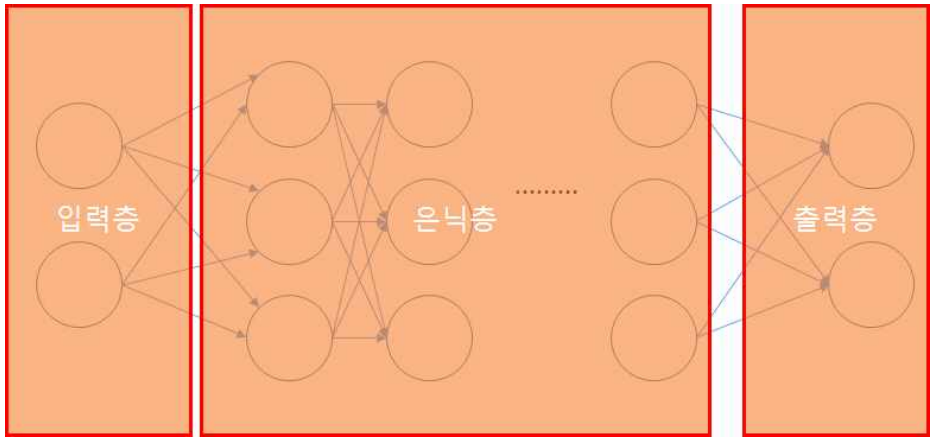
- 활성화함수(Activation Function)는 출력 값을 활성화를 일으키게 할 것이냐를 결정하고, 그 값을 부여하는 함수라 할 수 있다.

- 활성화 함수 사용의 이유는 Data를 비선형으로 바꾸기 위해서이다.

- 비선형 함수는 직선으로 표현할 수 없는 데이터사이의 관계의 표현이 가능하다.

일반적으로 선형 시스템을 사용해야 예측이 가능하고 장점도 많다고 알고 있다.
하지만 선형시스템을 망에 적용 시, 망이 깊어지지 않는다.
선형 시스템의 경우 망이 아무리 깊어지더라도, 1층의 은닉층으로 구현이 가능하다.
모든 a,b,x,y(a,b는 상수, x,y는 변수) 에 대하여 f(ax+by)=af(x)+bf(y)의 성질을 가졌기 때문에, 망이 아무리 깊어진들 1개의 은닉층으로 구현이 가능하다.

- 망이 깊어진다는 것은 은닉 층의 수가 많아진다는 것이다.



● 망이 깊어질 때의 장점

1. 매개 변수(파라미터)가 줄어든다.
2. 필요한 연산의 수가 줄어든다.

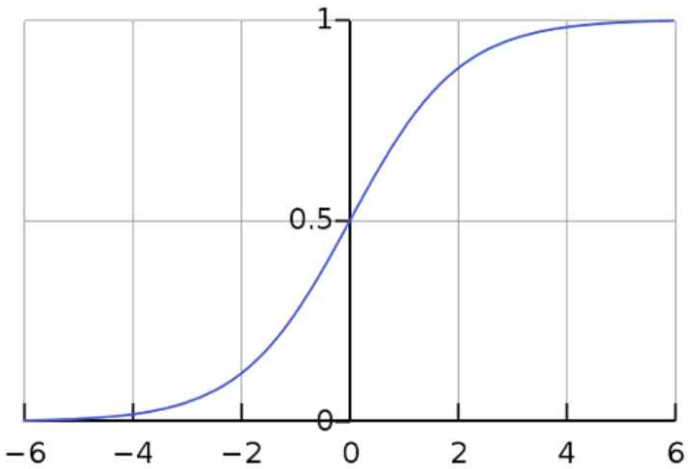
Filter를 Convolution할 때 필터의 크기를 줄이고, 망을 깊게 만들면 연산 횟수가 줄어들면서도 정확도를 유지하는 결과를 가져옴

2) Sigmoid Function(시그모이드 함수)

- 단일퍼셉트론에서 사용했던 활성화 함수이다. 입력을 (0,1) 사이의 값으로 normalize해준다.

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{\partial x} sigmoid(x) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right)$$

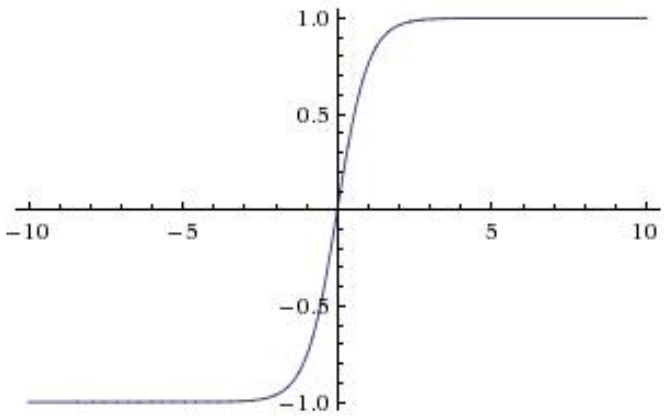


- 시그모이드는 (A′=A(1−A))로 미분이 되어 코드로 구현하기가 쉽다.
- 문제점은 Gradient Vanishing이다. x가 0일 때 기울기가 최대가 되는데, 그 미분값이 -4, 딥러닝을 하기 위해서는 Layer를 많이 쌓아야 하는데 이렇게 작은 미분 값은 에너지함수 최적화 과정에서 Layer을 거쳐갈 때마다 곱하기 연산을 거쳐 deep할 수록 기울기가 사라져 버리는 **Gradient Vanishing을 야기** 시킬 수 있다.
-1~2개의 Layer에서는 사용할 수 있겠지만 Deep한 학습법에서 사용하는데 안 좋음

3) tanh Function
- sigmoid fuction을 보완하고자 나온 함수이다.
- 입력신호를 (−1,1) 사이의 값으로 normalization(정규화)해준다.
- Sigmoid와 비교하여 tanh는 출력 범위가 더 넓고 경사면이 큰 범위가 더 크기 때문에 **더 빠르게 수렴하여 학습**
- 기울기가 최대인 x가 0인 지점의 미분 값은 1이 된다. 그럼에도 불구하고 역시 **Gradient Vanishing문제가 발생**하게 된다.

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\frac{d}{dx} tanh(x) = 1 - tanh(x)^{2}$$

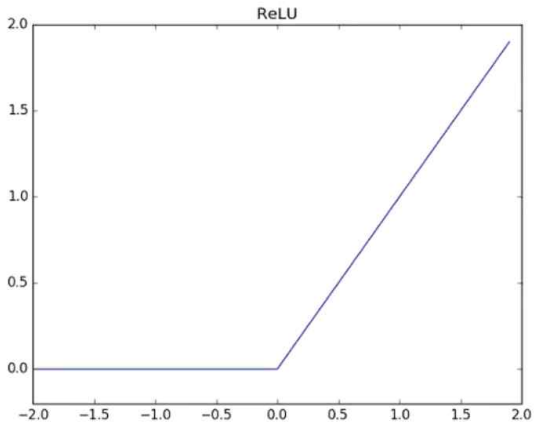


4) ReLU Function (Rectified Linear Unit)
- 현재 가장 인기 있는 활성화 함수
- ReLu는 양수에서 Linear Function과 같으며 음수는 0으로 버려버리는 함수(Relu(x)=max(0,x))이다.
- 기울기(미분값)가 0 또는 1의 값을 가지기 때문에 Sigmoid Function에서 나타나는 **Gradient Vanishing 문제가 발생하지 않는다.**

- **ReLU 함수는 입력이 0 이상이면 그 입력 값을 그대로 출력하고, 그 이하에서는 0을 출력하는 함수입니다.**

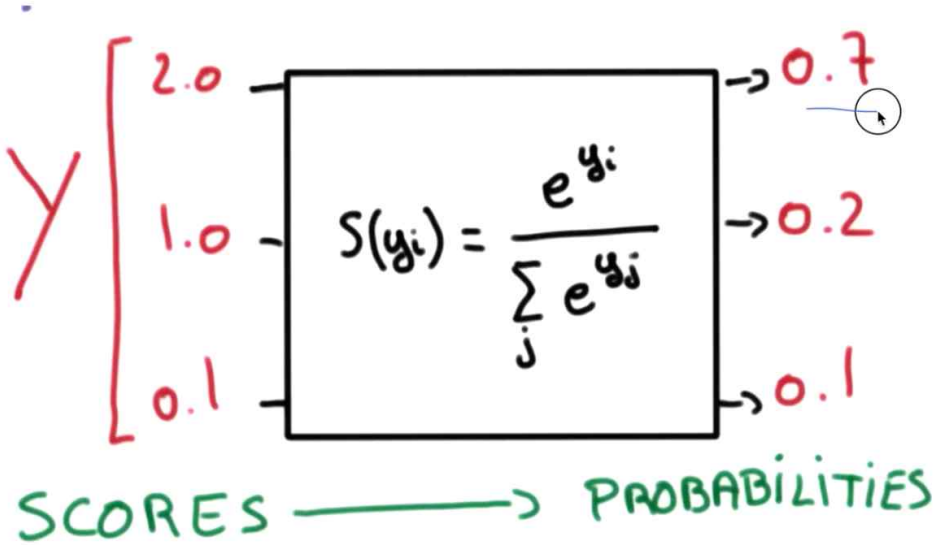
$$h(x) = \begin{cases} x(x > 0) \\ 0(x \leq 0) \end{cases}$$

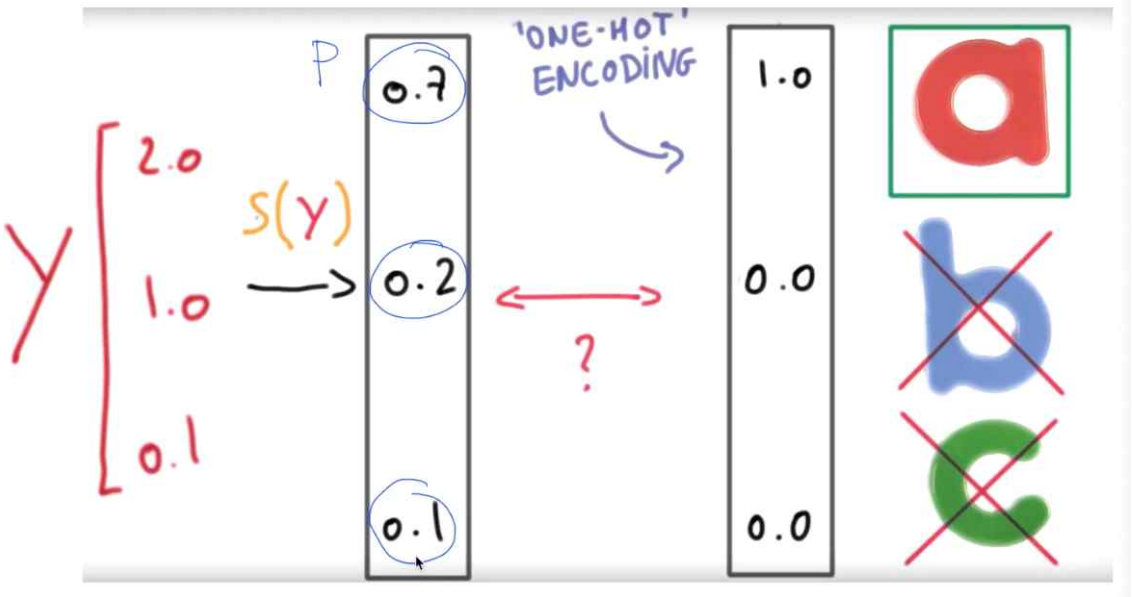
- sigmoid함수나 tanh함수보다 6배 정도 빠르게 학습이 된다.



5) Softmax function

- 소프트맥스는 인풋값을 여러개 갖는 일종의 함수로, 입력받은 값을 출력으로 **0~1사이의 값**으로 모두 정규화하며 출력 값들의 **총합은 항상 1**이 되는 특성을 가진 함수이다.
- 만약 인풋값이 하나밖에 없다면? 그런 함수는 시그모이드 함수라고 한다.
- 분류하고 싶은 클래수의 수 만큼 출력으로 구성하고, **가장 큰 출력 값을 부여받은 클래스가 확률이 가장 높은 것으로 이용**된다.
- softmax 함수는 K개의 값이 존재할 때 각각의 값의 편차를 확대시켜 큰 값은 상대적으로 더 크게, 작은 값은 상대적으로 더 작게 만든 다음에 normalization 시키는 함수다.





$$p_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

$$= \frac{e^{x_j}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}} \quad for\ j\ =\ 1,\ldots,K$$

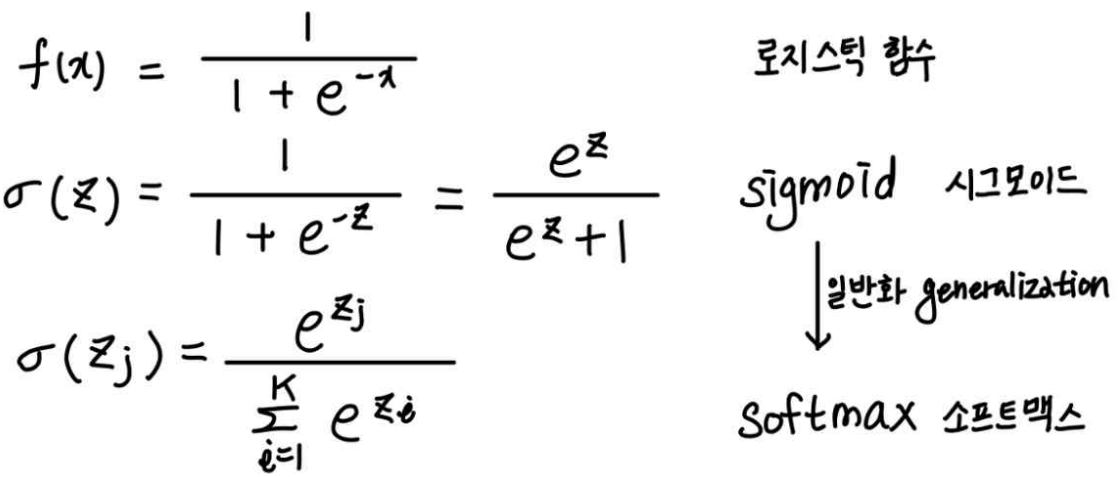
분자에 있는 지수함수가 각각의 값의 편차를 확대시키는 역할을 한다. 지수함수 그래프를 생각해보면 입력 값이 커질수록 기하급수적으로 출력 값이 커짐을 알 수 있다. 따라서 K개 값 중에서 큰 값이었던 것은 상대적으로 확실히 커지게 되고, 작은

값이었던 것은 상대적으로 작아지게 된다. 분모에 모든 K값의 지수함수 값을 모두 더했기 때문에 상대성을 갖게 되는 것이다.
그리고 또 하나 중요한 것은 softmax 함수를 거친 K개 값의 합은 1이 된다는 것이다.

$$p_1+p_2+\cdots+p_K = \frac{e^{x_1}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}}+\frac{e^{x_2}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}}+\cdots+\frac{e^{x_K}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}}$$
$$= \frac{e^{x_1}+e^{x_2}+\cdots+e^{x_K}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}}$$
$$= 1$$

- softmax를 통해서 K개 값들은 상대적인 중요도를 나타내는 K개 값으로 새로 태어나고, 그 값들의 총합은 1이 된다.
- 이러한 특성이 있기 때문에 **softmax는 딥러닝의 마지막 출력단에 사용**되는 것
- **fully-connected layer의 마지막 층의 출력 값에 softmax를 적용하면 그 값들은 클래스 확률로 변모한다.** 어떤 클래스에 속할 확률을 나타내는 것이다(강아지 클래스에 속하는지, 고양이 클래스에 속하는지의 확률을 가지게 된다)
- 소프트맥스 결과 값을 One hot encoder로 연결하면 가장 큰 값만 True값, 나머지는 False값이 나오게 됨

※ Softmax 계산식에 대한 자세한 설명:
https://tensorflow.blog/%ED%95%B4%EC%BB%A4%EC%97%90%EA%B2%8C-%EC%A0%84%ED%95%B4%EB%93%A4%EC%9D%80-%EB%A8%B8%EC%8B%A0%EB%9F%AC%EB%8B%9D-3/



- Softmax function 구글콜렙 체험:
1) softmax-regression-scratch.ipynb;
https://colab.research.google.com/github/d2l-ai/d2l-en-colab/blob/master/chapter_linear-networks/softmax-regression-scratch.ipynb#scrollTo=jAAyt_zlJZr5

[2교시]

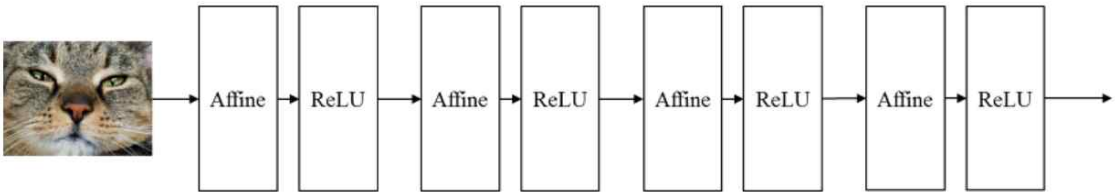
1. 합성곱 신경망 (Convolutional Neural Network, CNN)

1) CNN은 **필터링 기법**을 인공신경망에 적용함으로써 이미지를 더욱 효과적으로 처리하기 위해 (LeCun et al., 1989)에서 처음 소개되었으며, 이후에 (LeCun et al., 1998)에서 현재 딥 러닝에서 이용되고 있는 형태의 CNN이 제안되었다.

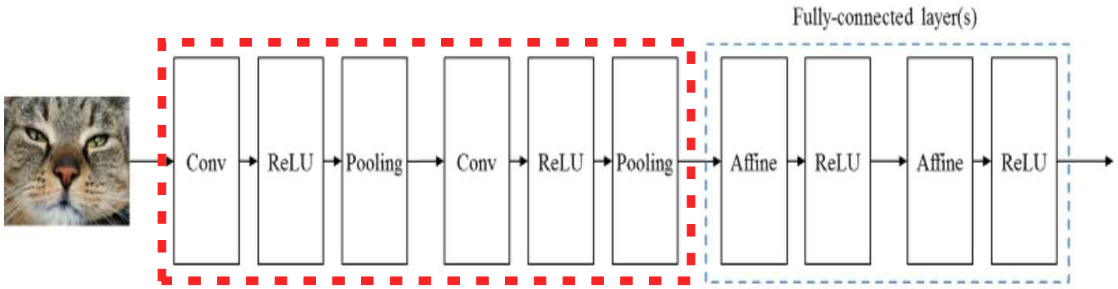
2) CNN의 기본 개념은 "행렬로 표현된 필터의 각 요소가 데이터 처리에 적합하도록 자동으로 학습되게 하자"는 것이다.

3) CNN의 구조

일반적인 인공신경망은 fully-connected 연산과 **ReLU**와 같은 비선형 활성 함수 (nonlinear activation function)의 합성으로 정의된 계층을 여러 층 쌓은 구조이다.



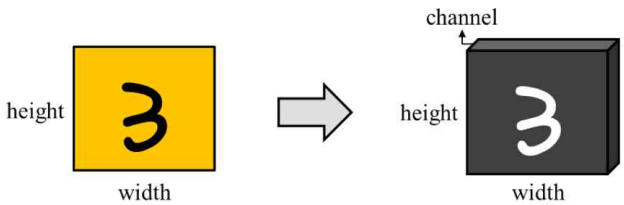
CNN은 **합성곱 계층 (convolutional layer)**과 **풀링 계층 (pooling layer)**이라고 하는 새로운 층을 fully-connected 계층 이전에 추가함으로써 원본 이미지에 필터링 기법을 적용한 뒤에 필터링된 이미지에 대해 분류 연산이 수행되도록 구성된다.



- 합성곱 계층은 이미지에 필터링 기법이 적용하고, 풀링 계층은 이미지의 국소적인 부분들을 하나의 대표적인 스칼라 값으로 변환함으로써 이미지의 크기를 줄이는 등의 다양한 기능들을 수행한다.

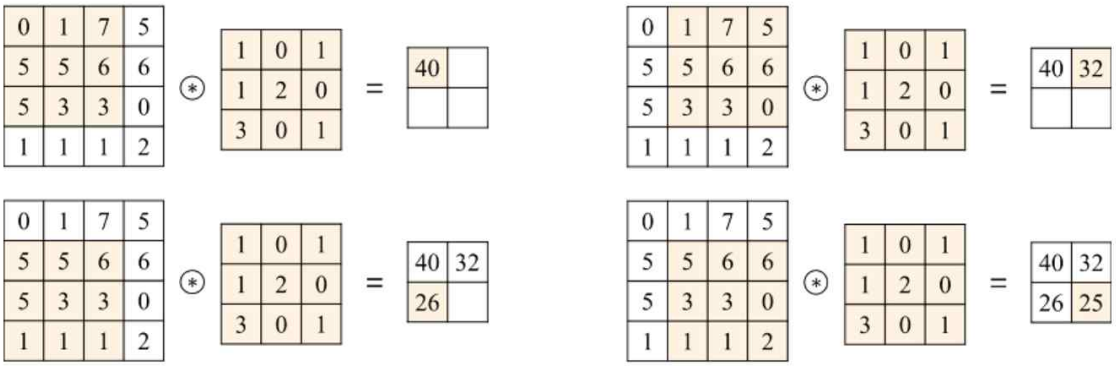
(1) 합성곱 계층 (Convolutional Layer)

- 이미지 데이터는 높이X너비X채널의 3차원 텐서 (tensor)로 표현될 수 있다.
- 이미지의 색상이 RGB 코드로 표현(채널의 크기는 3이 되며 각각의 채널에는 R, G, B 값이 저장)

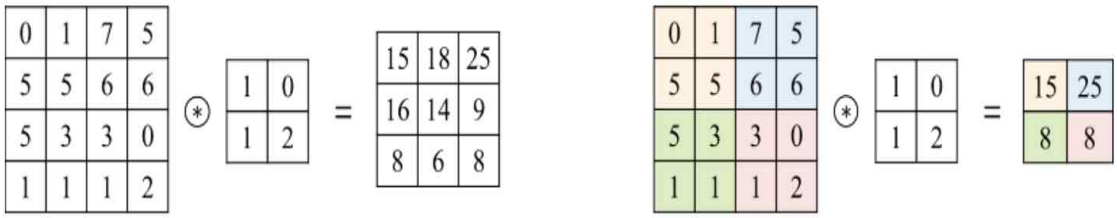


(컬러 이미지 데이터에 대한 텐서 표현)

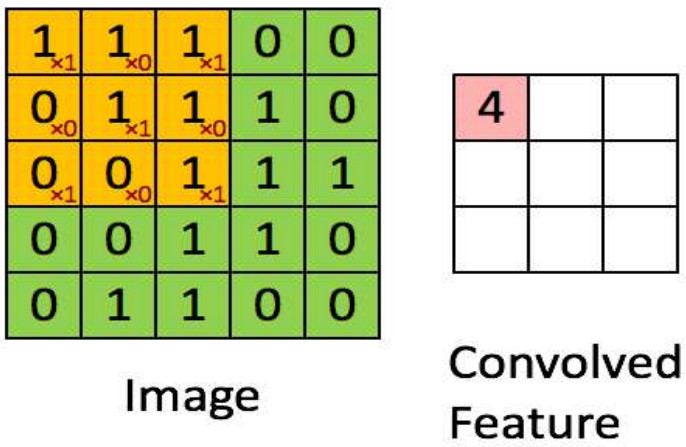
- 하나의 합성곱 계층에는 입력되는 이미지의 채널 개수만큼 필터가 존재하며, 각 채널에 할당된 필터를 적용함으로써 합성곱 계층의 출력 이미지가 생성된다. 높이X너비X채널이 4X4X1인 텐서 형태의 입력 이미지에 대해 **3X3 크기의 필터를 적용**하는 합성곱 계층에서는 이미지와 필터에 대한 **합성곱 연산**을 통해 **2X2X1 텐서 형태의 이미지가 생성**된다.

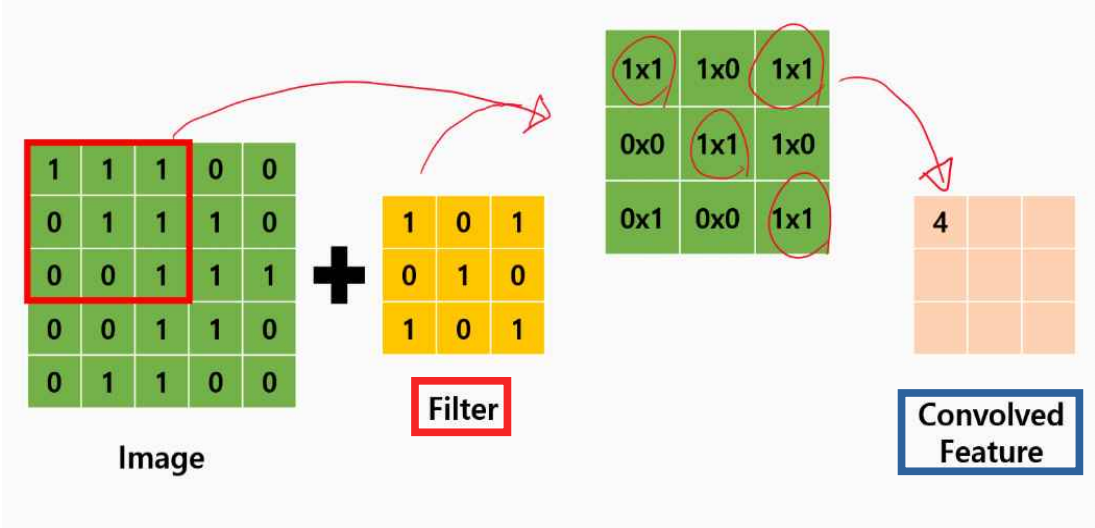


- 이미지에 대해 필터를 적용할 때는 필터의 이동량을 의미하는 스트라이드 (stride)를 설정한다.

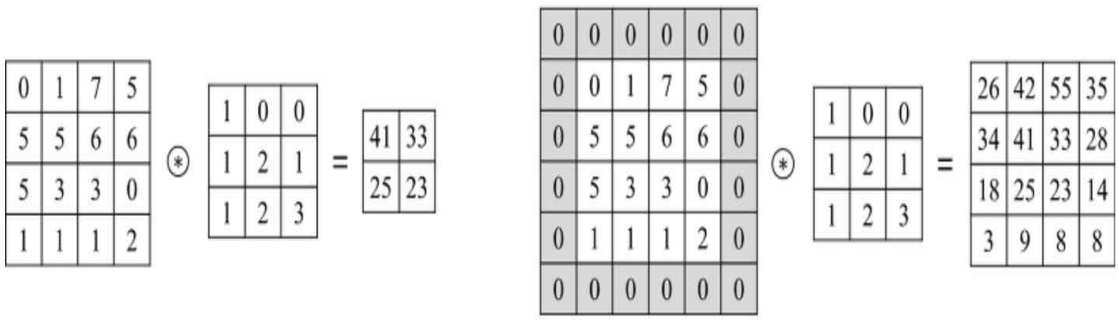


(스트라이드가 1로 설정된 경우 (좌)와 2로 설정된 경우 (우))





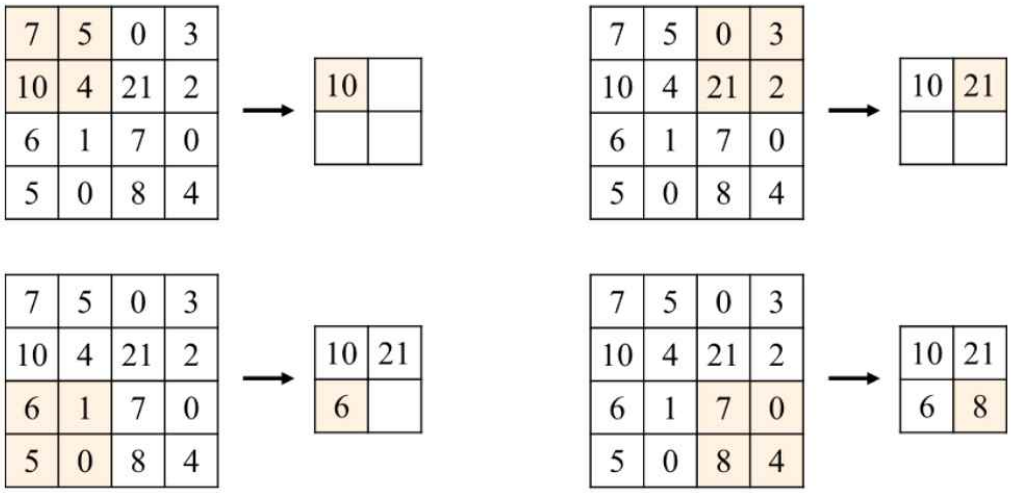
- 입력 이미지에 대해 합성곱을 수행하면, 출력 이미지의 크기는 입력 이미지의 크기보다 작아지게 된다. 그러므로 합성곱 계층을 거치면서 이미지의 크기는 점점 작아지게 되고, **이미지의 가장자리에 위치한 픽셀들의 정보는 점점 사라지게 된다.**
- 이러한 문제점을 해결하기 위해 이용되는 것이 **패딩 (padding)**이다. 패딩은 입력 이미지의 가장자리에 특정 값으로 설정된 픽셀들을 추가함으로써 입력 이미지와 출력 이미지의 크기를 같거나 비슷하게 만드는 역할을 수행한다. 이미지의 가장자리에 0의 값을 갖는 픽셀을 추가하는 것을 **zero-padding**이라고 하며, CNN에서는 주로 이러한 zero-padding이 이용된다.



(패딩이 적용되지 않은 합성곱 연산 (좌)과 zero-padding이 적용된 합성곱 연산 (우))

(2) 풀링 계층 (Pooling Layer)

- CNN에서 **합성곱 계층**과 ReLU와 같은 비선형 활성 함수를 거쳐서 생성된 이미지는 **풀링 계층에 입력된다.**
- 풀링 계층은 주로 **max-pooling**을 기반으로 구현된다.

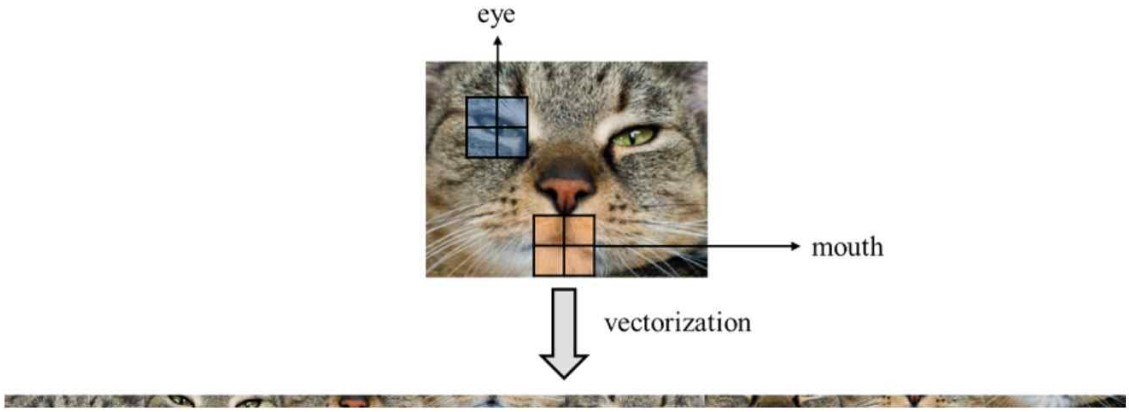


(Max-pooling 기반 풀링 계층의 동작)

- pooling의 장점: 픽셀들이 이동 및 회전 등에 의해 위치가 변경되더라도 출력 값을 동일하다. 따라서, 풀링 계층을 이용할 경우, **이미지를 구성하는 요소들의 이동 및 회전 등에 의해 CNN의 출력값이 영향을 받는 문제를 완화**할 수 있다.
- CNN이 처리해야하는 이미지의 크기가 크게 줄어들기 때문에 인공신경망의 model **parameter 크게 감소**한다. 따라서, 풀링 계층을 이용함으로써 **CNN의 학습 시간을 크게 절약**할 수 있으며, **오버피팅 (overfitting) 문제 또한 어느 정도 완화**할 수 있다.

4) Fully-connected neural networks (FNNs)의 문제점과 CNN

- 이미지를 처리할 때 발생하는 FNN의 첫 번째 문제점은 인접 픽셀 간의 상관관계가 무시된다는 것이다.
- FNN은 벡터 형태로 표현된 데이터를 입력 받기 때문에 이미지를 반드시 벡터화해야 한다.



(FNN을 이용하여 이미지를 처리하기 위한 이미지 벡터화)

- 이미지 데이터에서는 일반적으로 인접 픽셀간의 매우 높은 상관관계가 존재하기 때문에 이미지를 벡터화 (vectorization)하는 과정에서 막대한 정보 손실이 발생한다.
- **CNN은** 이미지의 형태를 보존하도록 행렬 형태의 데이터를 입력 받기 때문에 이미지를 **벡터화 하는 과정에서 발생하는 정보 손실을 방지**할 수 있다.

※ 파이토치 CNN 스터디 코드;

https://www.youtube.com/watch?v=7CMxgqvrDgg&list=PLSAJwo7mw8jn8iaXwT4MqLbZnS-LJwnBd&index=17

[3교시]

1. 실습: CNN을 사용한 MNIST 이미지 인식
https://www.youtube.com/watch?v=7F3BYD5N8XE

2. 실습: 인공지능 웃음 판독기 - Python, Deep Learning
https://www.youtube.com/watch?v=GrN1tKjVBM8

3. 영상 수업 관련 OT