

3일차 인공신경망 및 CNN 모델

강의자료: <https://github.com/nahyungsun/tutorial>

0. 인공 신경망

신경망

- 기계 학습 역사에서 가장 오래된 기계학습 모델, 현재 가장 다양한 형태를 가짐
- 1950년대 퍼셉트론 → 1980년대 다층 퍼셉트론
- 딥러닝의 기초가 됨

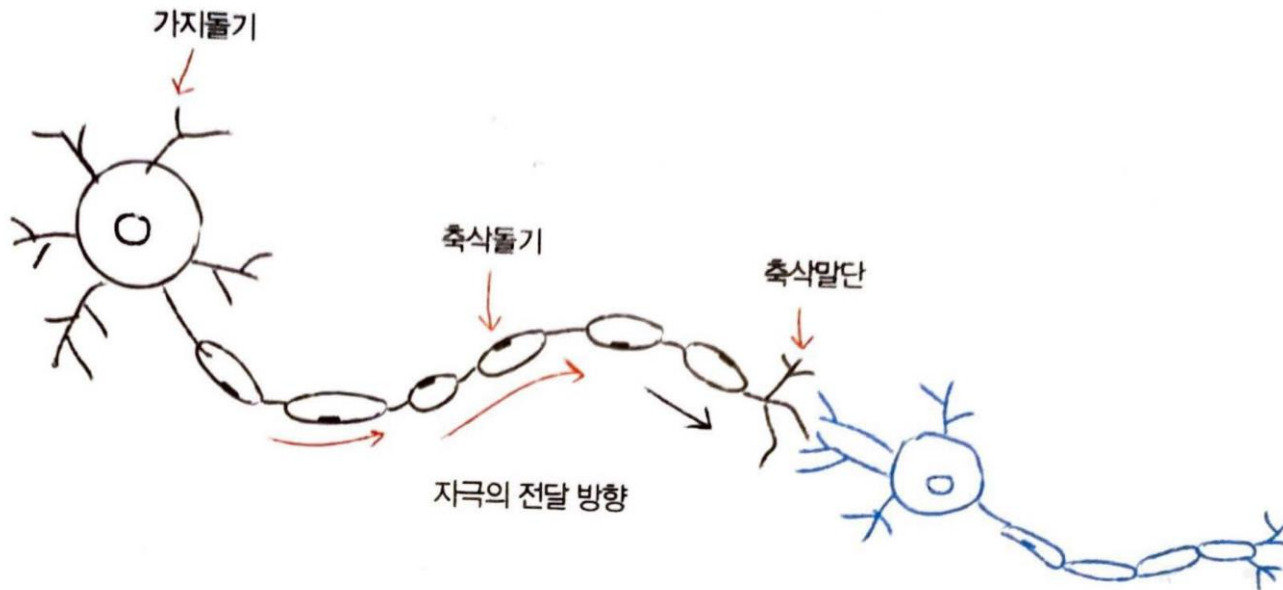
신경망의 발전

- 컴퓨터 과학과 의학의 시너지
- 컴퓨터 과학: 계산 능력 발전
- 의학: 두뇌의 정보 처리 방식 연구(뉴런의 동작 이해 등)

0. 인공 신경망

뉴런의 동작 원리

- 가지돌기에서 신호를 받아들임
- 신호가 축삭돌기를 지나 축삭말단으로 전달됨
- 축삭돌기를 지나는 동안 신호가 약해지거나, 강하게 전달되기도 함
- 축삭말단까지 전달된 신호는 다른 뉴런의 가지돌기로 전달



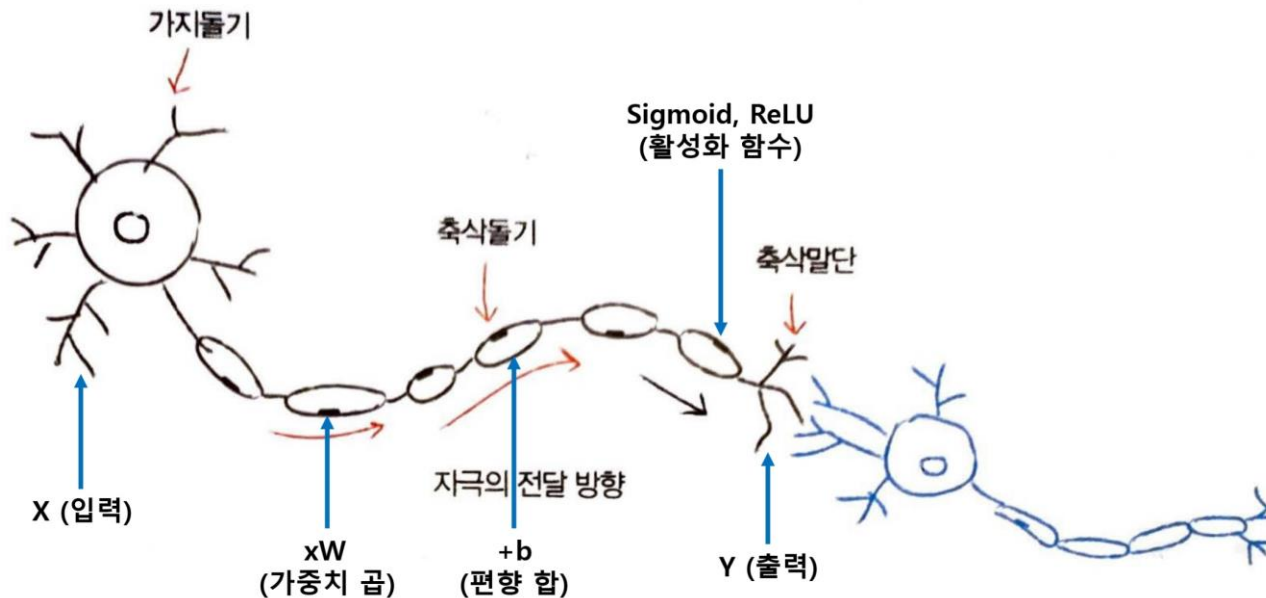
0. 인공 신경망

뉴런의 기본 동작

- 입력 신호, 즉 입력 값 x 에 가중치(W)를 곱하고 편향(b)을 더함
- 그 후 활성화 함수를 거쳐 결과 값 Y 를 만들어 냄

학습

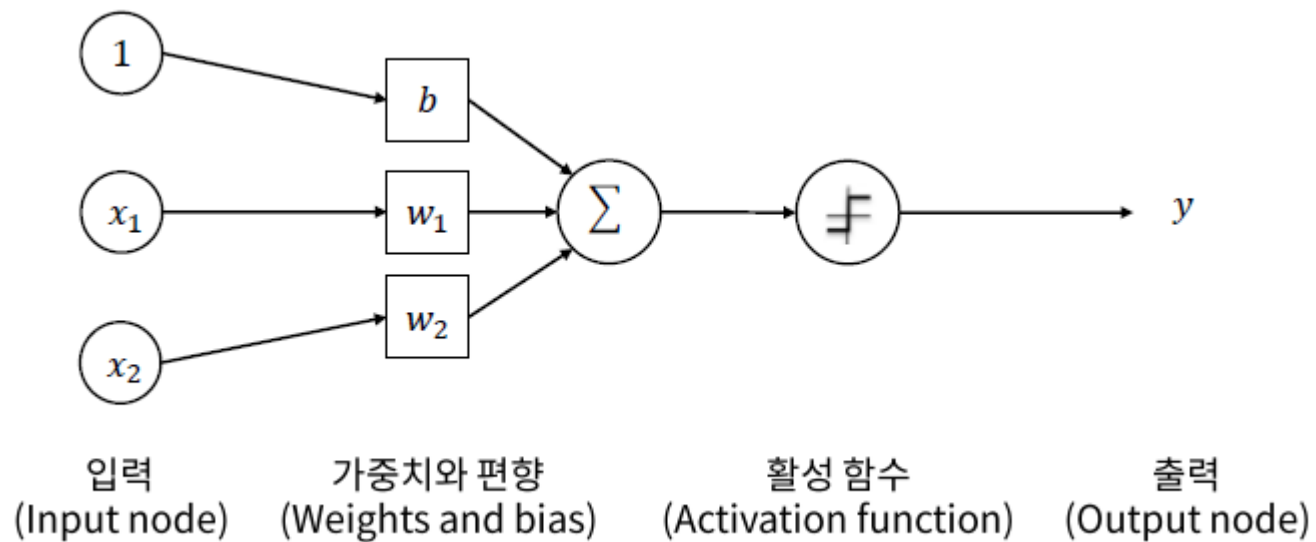
- 원하는 Y 값을 만들기 위해 가중치와 편향을 변경해가면서 적절한 값을 찾아내는 과정



0. 인공 신경망

인공신경망 구조

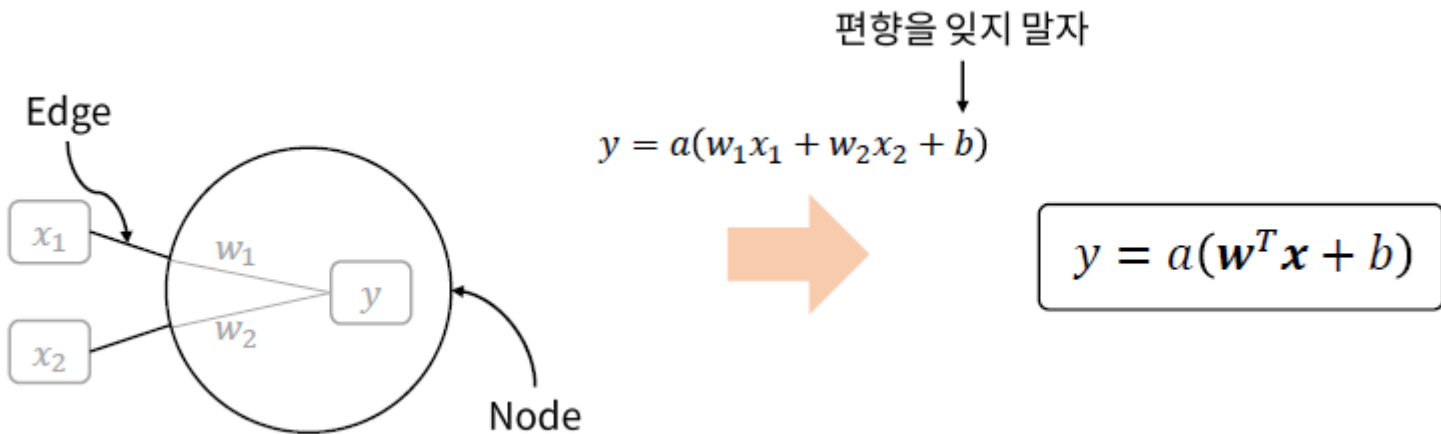
- 인공 뉴런인 퍼셉트론을 기본 단위로 하며, 이를 조합해 복잡한 구조를 이룸



0. 인공 신경망

인공신경망 표현

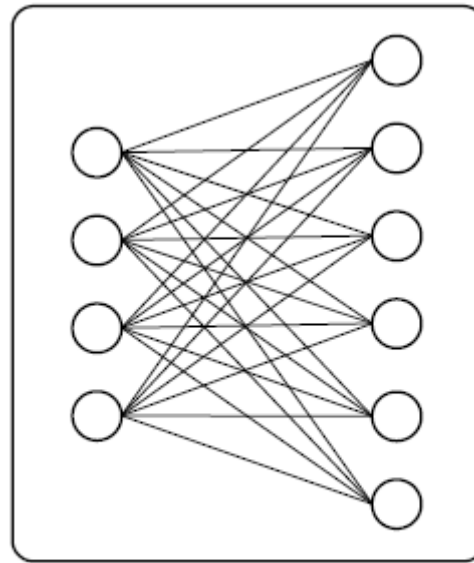
- 노드(Node)와 간선(Edge)을 이용해 표현
- 노드: 단일 뉴런 연산
- 간선: 뉴런의 연결성



0. 인공 신경망

전결합 계층

- Fully-Connected Layer
- 두 계층 간의 연결이 모두 되어 있는 계층으로 고전적인 신경 네트워크 아키텍처

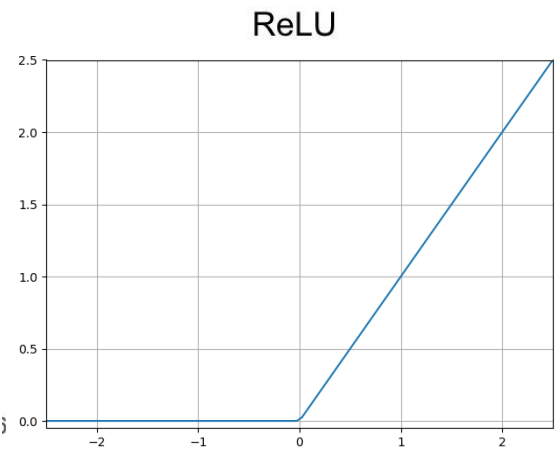
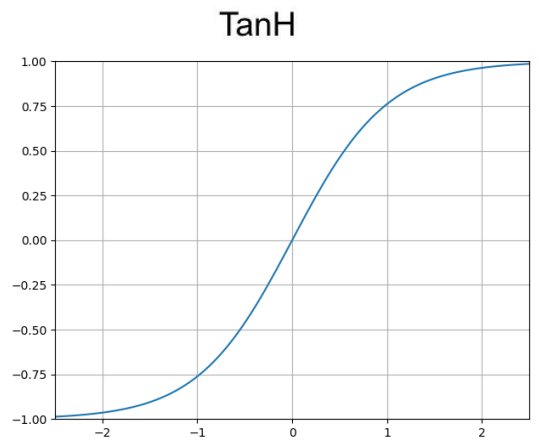
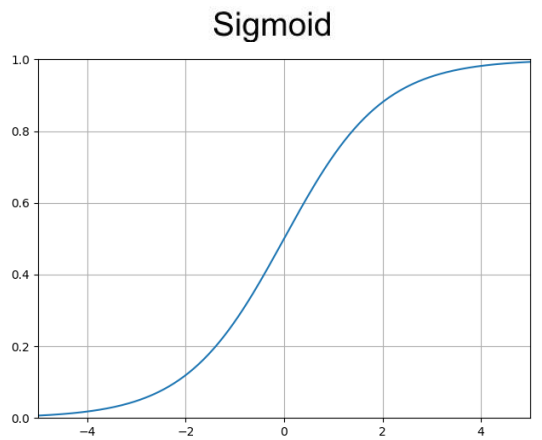


FC Layer의 그래프 표현

0. 인공 신경망

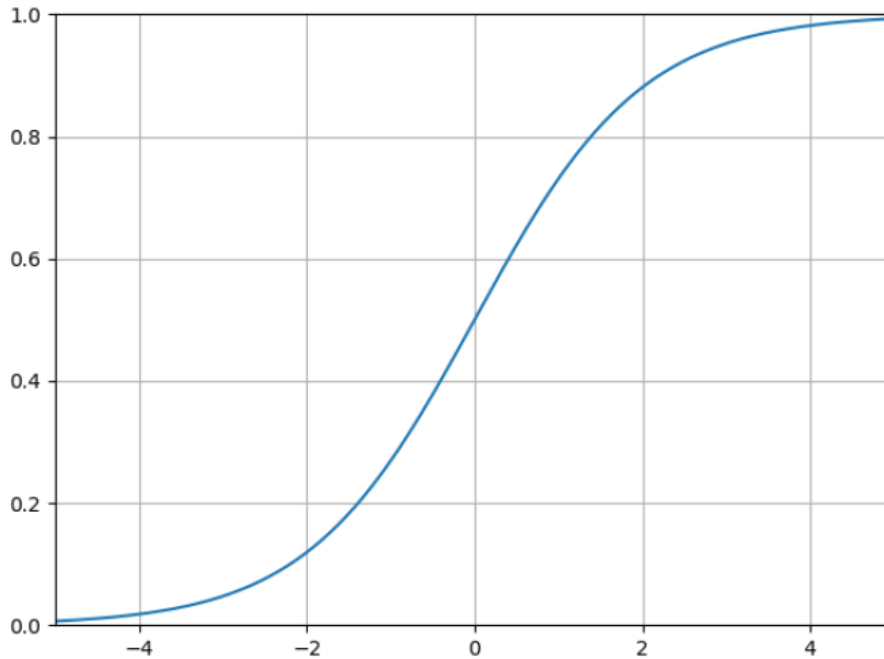
활성화 함수

- 인공신경망을 통과해온 값을 최종적으로 어떤 값으로 만들지 결정
- 일반적으로 비선형 함수



0. 인공 신경망

Sigmoid

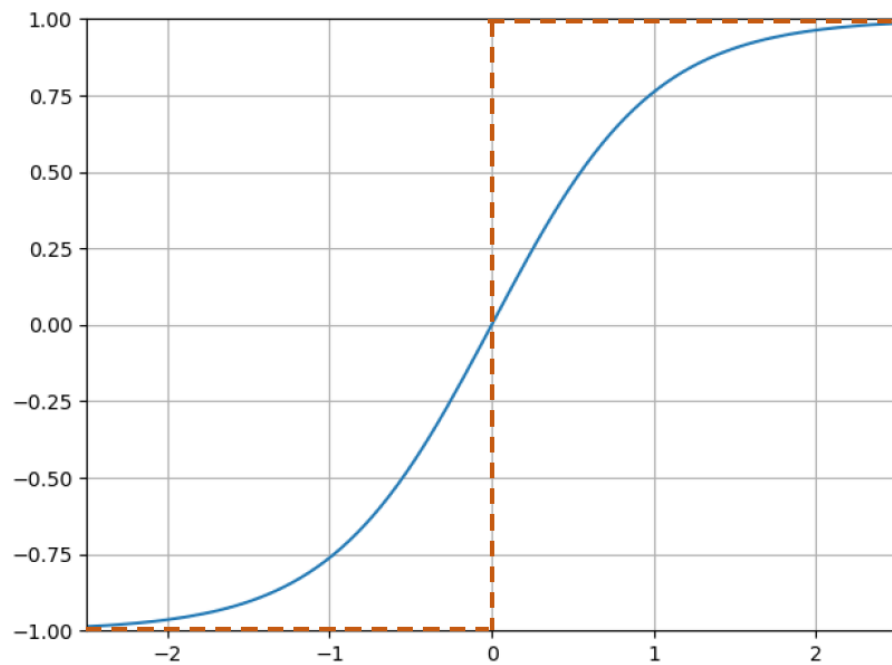


Sigmoid 함수

- 값이 작아질 수록 0, 커질 수록 1에 수렴
- 0~1사이의 실수 값으로 출력이 정의됨
- 확률을 표현할 수 있음
- 입력값이 0에 가까울 수록 출력이 크게 변함

0. 인공 신경망

Tanh

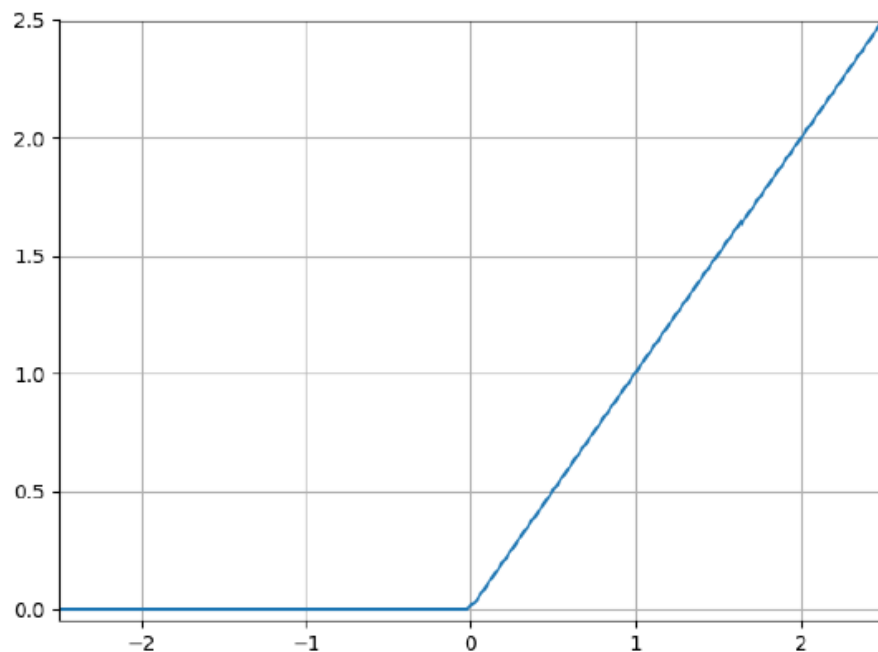


Tanh (Hyperbolic tangent) 함수

- 값이 작아질 수록 -1, 커질 수록 1에 수렴
- -1~1사이의 실수 값으로 출력이 정의됨
- 입력값이 0에 가까울 수록 출력이 크게 변함

0. 인공 신경망

ReLU

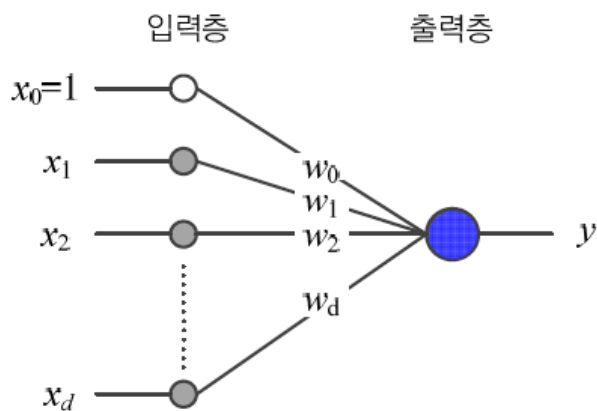


ReLU (Rectified linear unit) 함수

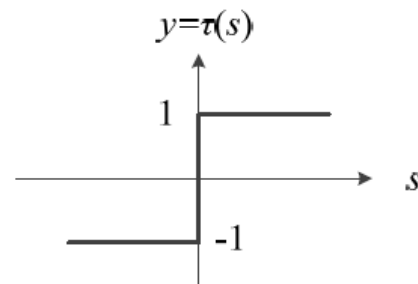
- 0보다 작은 값을 0으로 강제하는 함수
- 딥러닝에서 가장 많이 사용됨
- 미분 값이 일정해 학습에 용이함
- 구현이 단순해 빠른 연산이 가능

0. 인공 신경망

- 단층 퍼셉트론의 구조
 - 입력층과 출력층을 가짐
 - 입력층의 i 번째 노드는 입력 벡터 $x = (x_1, x_2, \dots, x_d)$ 의 요소 x_i 를 담당
 - i 번째 입력층 노드와 출력층을 연결하는 간선은 가중치 w_i 를 가짐
 - 출력층은 하나의 노드



(a) 퍼셉트론의 구조

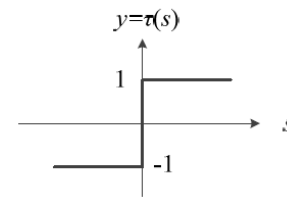
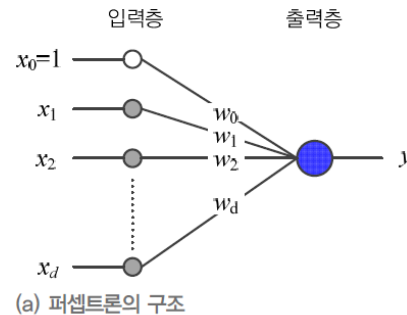


(b) 계단함수를 활성화함수 $\tau(s)$ 로 이용함

0. 인공 신경망

- 퍼셉트론의 동작
 - 해당하는 입력값과 가중치를 곱한 결과를 모두 더해 s 를 구하고, 활성화함수를 적용
 - 활성함수로 계단함수를 사용하므로 최종 출력 y 는 +1 또는 -1 즉, 선형 분류기

$$y = \tau(s)$$
$$\text{이때 } s = w_0 + \sum_{i=1}^d w_i x_i, \quad \tau(s) = \begin{cases} 1 & s \geq 0 \\ -1 & s < 0 \end{cases}$$

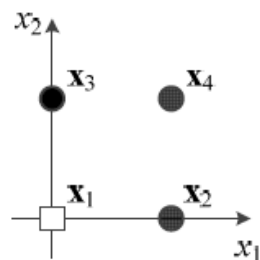


$$S = X_0 * W_0 + X_1 * W_1 + X_2 * W_2 + \dots + X_d * W_d$$
$$Y = \text{활성화함수}(S)$$

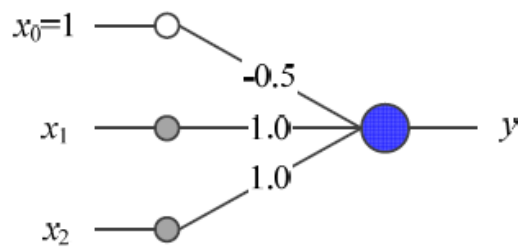
0. 인공 신경망

- 예제
 - OR 분류

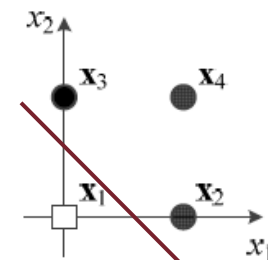
$$\mathbf{x}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, y_1 = -1, \quad \mathbf{x}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, y_2 = 1, \quad \mathbf{x}_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, y_3 = 1, \quad \mathbf{x}_4 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, y_4 = 1$$



(a) 훈련집합



(b) 퍼셉트론



(a) 훈련집합

$$\mathbf{x}_1: s = -0.5 + 0 * 1.0 + 0 * 1.0 = -0.5,$$

$$\mathbf{x}_2: s = -0.5 + 1 * 1.0 + 0 * 1.0 = 0.5,$$

$$\mathbf{x}_3: s = -0.5 + 0 * 1.0 + 1 * 1.0 = 0.5,$$

$$\mathbf{x}_4: s = -0.5 + 1 * 1.0 + 1 * 1.0 = 1.5,$$

$$\tau(-0.5) = -1$$

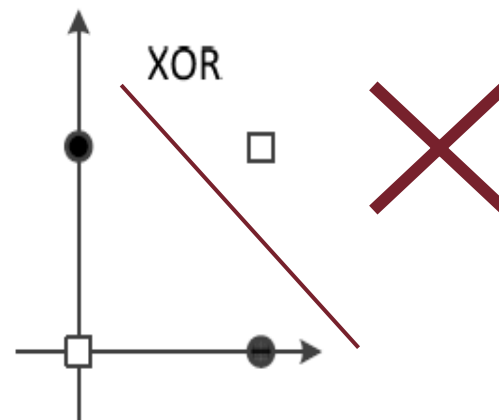
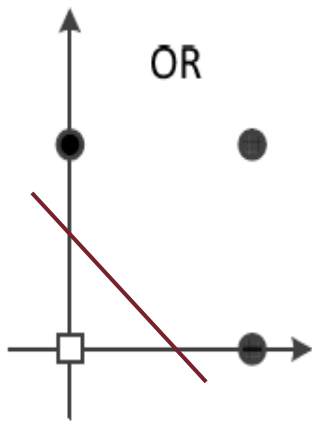
$$\tau(0.5) = 1$$

$$\tau(0.5) = 1$$

$$\tau(1.5) = 1$$

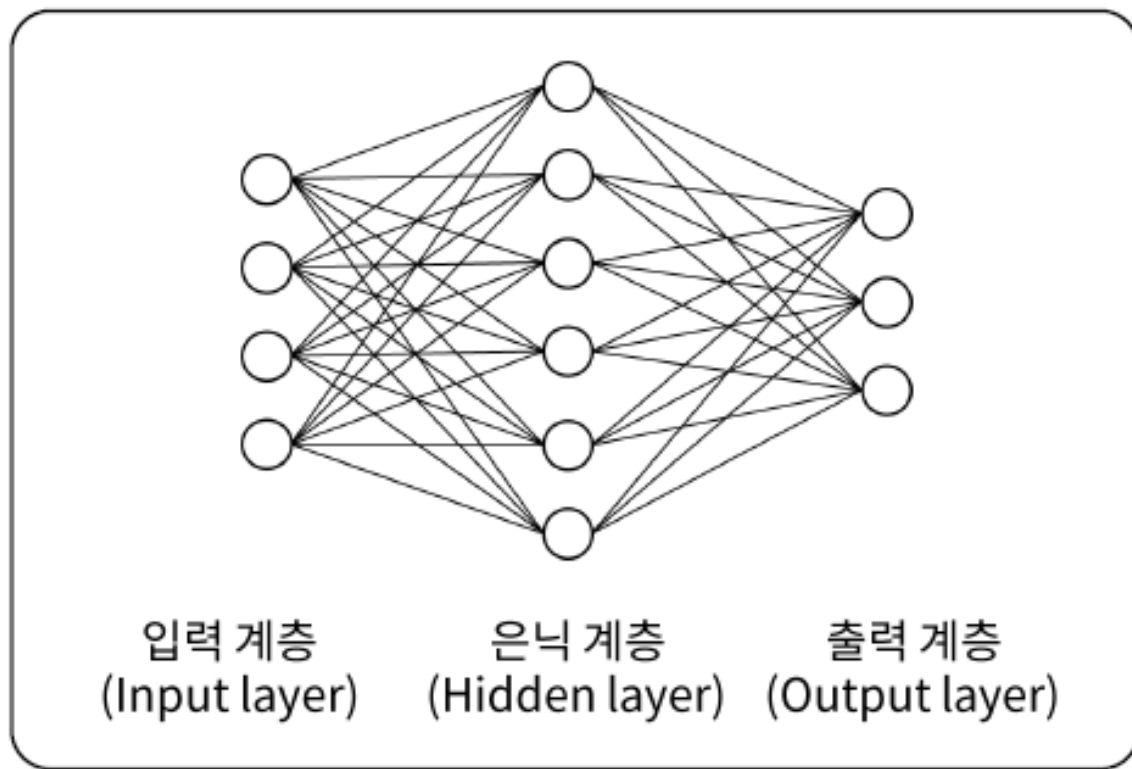
0. 인공 신경망

- 퍼셉트론의 한계
 - 선형 분류가 불가능한 상황에서는 해결 할 수 없음



0. 인공 신경망

- 다층 퍼셉트론
 - 입력 층(Input Layer), 은닉 층(Hidden Layer), 출력 층(Output layer)로 이루어진 신경망
 - 각 계층은 전결합층으로 이루어져 있음



0. 인공 신경망

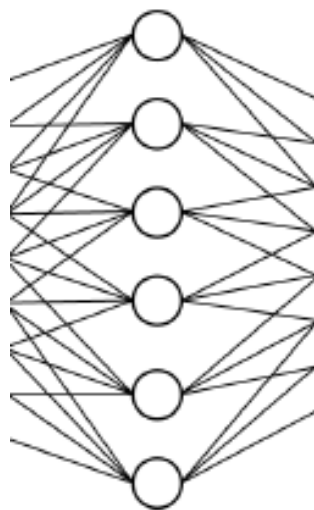
- 입력 층(Input Layer)
 - 아무런 연산이 일어나지 않고 신경망의 입력 값을 받아 다음 계층으로 전달하는 역할
 - 입력 층의 노드 수는 입력 벡터의 길이와 동일



입력 계층
(Input layer)

0. 인공 신경망

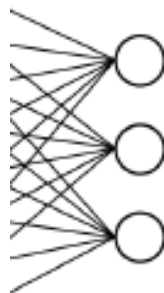
- 은닉 층(Hidden Layer)
 - 입력 층과 연결된 전결합 계층
 - 복잡한 문제를 해결할 수 있게 하는 핵심 계층



은닉 계층
(Hidden layer)

0. 인공 신경망

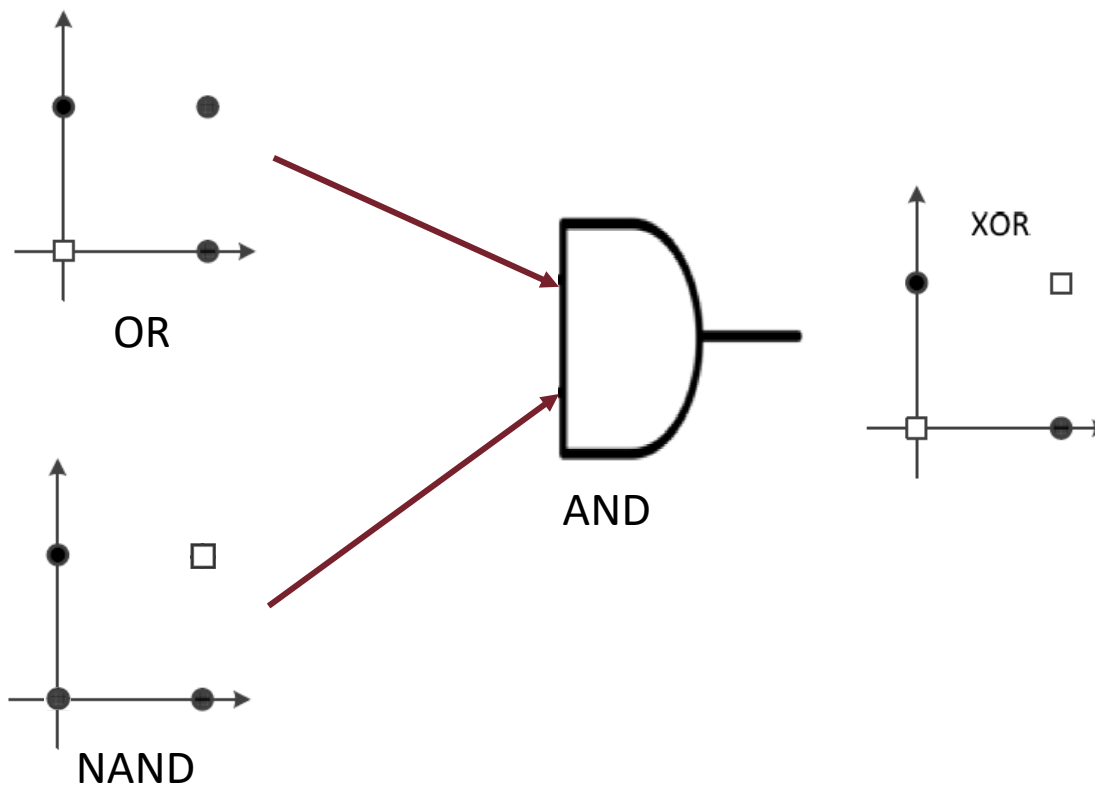
- 출력 층(Output Layer)
 - 은닉 층 다음에 오는 전결합 계층
 - 신경망 외부로 출력 값을 전달 함
 - 출력 층의 노드 수는 출력 벡터의 길이와 동일



출력 계층
(Output layer)

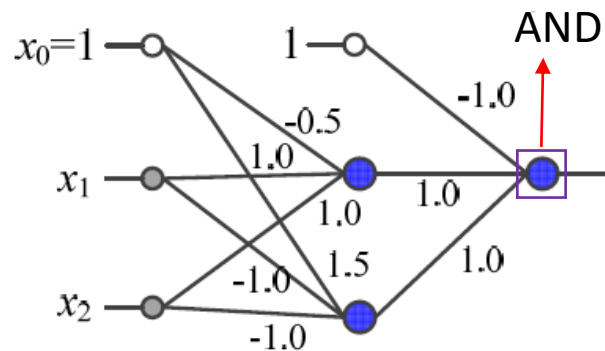
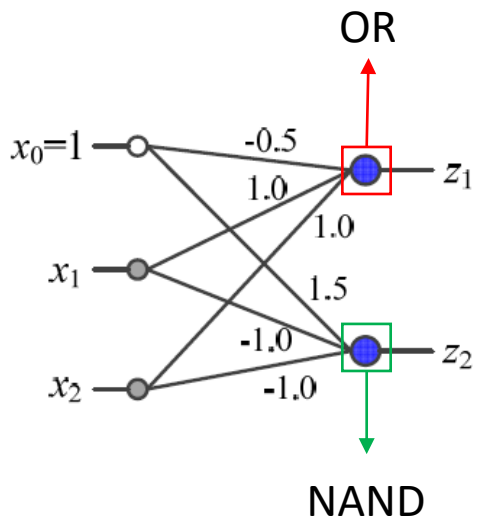
0. 인공 신경망

- 다층 퍼셉트론
 - XOR 분류



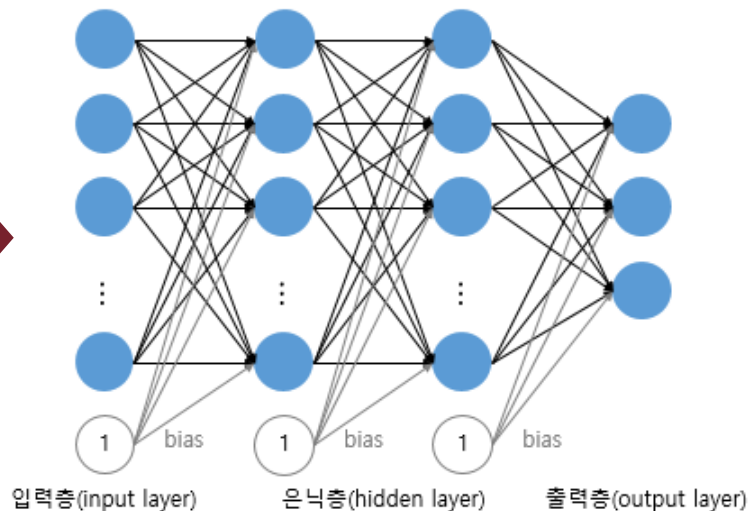
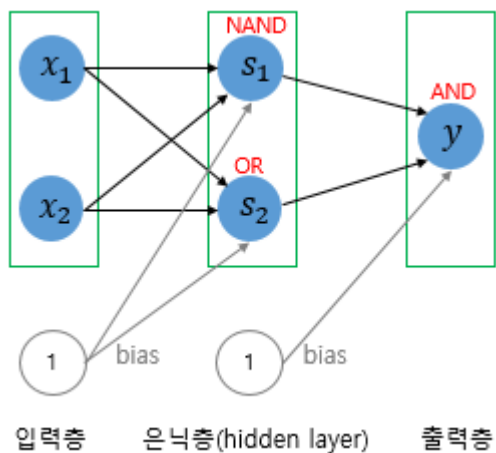
0. 인공 신경망

- 다층 퍼셉트론
 - XOR 분류
 - OR퍼셉트론과 NAND퍼셉트론을 병렬로 연결
 - 결과를 AND 퍼셉트론에 순차 연결



0. 인공 신경망

- 다층 퍼셉트론
 - 은닉층을 더 쌓아 심층 신경망(Deep Neural Network)을 만들어 더 복잡한 문제도 해결 가능



0. 인공 신경망

딥러닝

- 머신러닝의 선형적(Linear)인 성질 때문에 비선형적 문제를 해결할 수 없음
- 인간의 뇌 신경과 유사한 구조인 인공신경망을 이용하여 학습함
- 인공신경망이 다층으로 쌓인 것이 딥러닝(Deep-Learning)

	T	F
T	T	F
F	F	F

AND

	T	F
T	T	T
F	T	F

OR

	T	F
T	F	T
F	T	F

XOR

0. 인공 신경망

XOR 실습

- 필요한 라이브러리 импорт

```
1 import torch
2 device = 'cuda' if torch.cuda.is_available() else 'cpu'
3 torch.manual_seed(777)
4 if device == 'cuda':
5     torch.cuda.manual_seed_all(777)
```

- XOR 문제에 해당되는 입력과 출력 정의

```
1 X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
2 Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)
```

- 1개의 뉴런을 가지는 단층 퍼셉트론 구현

```
1 linear = nn.Linear(2, 1, bias=True)
2 sigmoid = nn.Sigmoid()
3 model = nn.Sequential(linear, sigmoid).to(device)
4 criterion = torch.nn.BCELoss().to(device)
5 optimizer = torch.optim.SGD(model.parameters(), lr=1)
```


0. 인공 신경망

XOR 실습

- 200번 부터 epoch 비용이 감소되지 않는 것을 확인할 수 있다.
- 단층 퍼셉트론은 XOR 문제를 풀 수 없기 때문!

```
1 #10,001번의 에포크 수행. 0번 에포크부터 2,000번 에포크까지.
2 for step in range(2000):
3     optimizer.zero_grad()
4     hypothesis = model(X)
5
6     # 비용 함수
7     cost = criterion(hypothesis, Y)
8     cost.backward()
9     optimizer.step()
10
11 if step % 100 == 0: # 100번째 에포크마다 비용 출력
12     print(step, cost.item())
```

```
0 0.7666423320770264
100 0.6931473612785339
200 0.6931471824645996
300 0.6931471824645996
400 0.6931471824645996
500 0.6931471824645996
600 0.6931471824645996
700 0.6931471824645996
800 0.6931471824645996
900 0.6931471824645996
```

```
1000 0.6931471824645996
1100 0.6931471824645996
1200 0.6931471824645996
1300 0.6931471824645996
1400 0.6931471824645996
1500 0.6931471824645996
1600 0.6931471824645996
1700 0.6931471824645996
1800 0.6931471824645996
1900 0.6931471824645996
```

0. 인공 신경망

XOR 실습

- 결과를 확인해 봐도 XOR 문제를 풀 수 없는 것을 확인할 수 있다.

```
1 with torch.no_grad():
2     hypothesis = model(X)
3     predicted = (hypothesis > 0.5).float()
4     accuracy = (predicted == Y).float().mean()
5     print('모델의 출력값(Hypothesis): ', hypothesis.detach().cpu().numpy())
6     print('모델의 예측값(Predicted): ', predicted.detach().cpu().numpy())
7     print('실제값(Y): ', Y.cpu().numpy())
8     print('정확도(Accuracy): ', accuracy.item())
```

모델의 출력값(Hypothesis): [[0.5]

[0.5]

[0.5]

[0.5]]

모델의 예측값(Predicted): [[0.]

[0.]

[0.]

[0.]]

실제값(Y): [[0.]

[1.]

[1.]

[0.]]

정확도(Accuracy): 0.5

0. 인공 신경망

XOR 실습

- XOR 문제를 해결하기 위해 다층 퍼셉트론을 시도
- 필요한 라이브러리 импорт
- XOR 문제에 해당되는 입력과 출력 정의

```
1 import torch
2 import torch.nn as nn
3 device = 'cuda' if torch.cuda.is_available() else 'cpu'
4
5 # for reproducibility
6 torch.manual_seed(777)
7 if device == 'cuda':
8     torch.cuda.manual_seed_all(777)
```

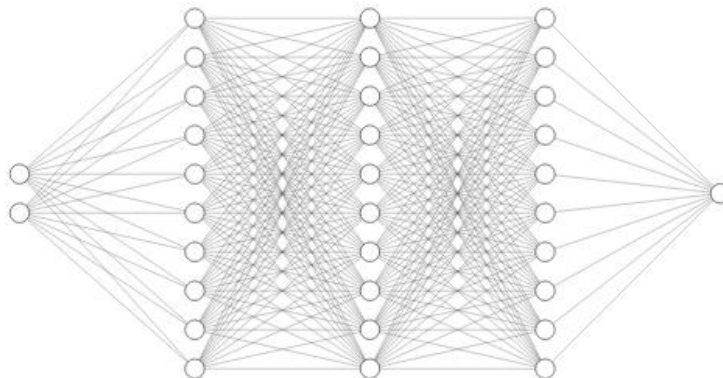
```
1 X = torch.FloatTensor([[0, 0], [0, 1], [1, 0], [1, 1]]).to(device)
2 Y = torch.FloatTensor([[0], [1], [1], [0]]).to(device)
```

0. 인공 신경망

XOR 실습

- 다층 퍼셉트론 설계
- 입력층, 은닉층1, 은닉층2, 은닉층3, 출력층을 가지는 은닉층이 3개인 인공 신경망
- 밑의 그림은 설계한 인공 신경망을 시각화한 것

```
1 model = nn.Sequential(  
2     nn.Linear(2, 10, bias=True), # input_layer = 2, hidden_layer1 = 10  
3     nn.Sigmoid(),  
4     nn.Linear(10, 10, bias=True), # hidden_layer1 = 10, hidden_layer2 = 10  
5     nn.Sigmoid(),  
6     nn.Linear(10, 10, bias=True), # hidden_layer2 = 10, hidden_layer3 = 10  
7     nn.Sigmoid(),  
8     nn.Linear(10, 1, bias=True), # hidden_layer3 = 10, output_layer = 1  
9     nn.Sigmoid()  
10 ).to(device)  
11 criterion = torch.nn.BCELoss().to(device)  
12 optimizer = torch.optim.SGD(model.parameters(), lr=1) # modified learning rate from 0.1 to 1
```



0. 인공 신경망

XOR 실습

- 이전과 다르게 epoch 가 진행될 때마다 비용이 줄어드는 것을 확인할 수 있다.

```
1 for epoch in range(10001):
2     optimizer.zero_grad()
3     # forward 연산
4     hypothesis = model(X)
5
6     # 비용 함수
7     cost = criterion(hypothesis, Y)
8     cost.backward()
9     optimizer.step()
10
11     # 100의 배수에 해당되는 에포크마다 비용을 출력
12     if epoch % 1000 == 0:
13         print(epoch, cost.item())
```

```
0 0.00011297257151454687
1000 9.459738794248551e-05
2000 8.12893922557123e-05
3000 7.120047666830942e-05
4000 6.32575829513371e-05
5000 5.6864570069592446e-05
6000 5.166375922271982e-05
7000 4.731238732347265e-05
8000 4.358691876404919e-05
9000 4.041282954858616e-05
10000 3.767089583561756e-05
```

0. 인공 신경망

XOR 실습

- 결과 역시 XOR 문제를 잘 해결한 것을 확인할 수 있다.

```
1 with torch.no_grad():
2     hypothesis = model(X)
3     predicted = (hypothesis > 0.5).float()
4     accuracy = (predicted == Y).float().mean()
5     print('모델의 출력값(Hypothesis): ', hypothesis.detach().cpu().numpy())
6     print('모델의 예측값(Predicted): ', predicted.detach().cpu().numpy())
7     print('실제값(Y): ', Y.cpu().numpy())
8     print('정확도(Accuracy): ', accuracy.item())
```

모델의 출력값(Hypothesis): [[2.3310218e-05]

[9.9996090e-01]

[9.9996424e-01]

[5.2368901e-05]]

모델의 예측값(Predicted): [[0.]

[1.]

[1.]

[0.]]

실제값(Y): [[0.]

[1.]

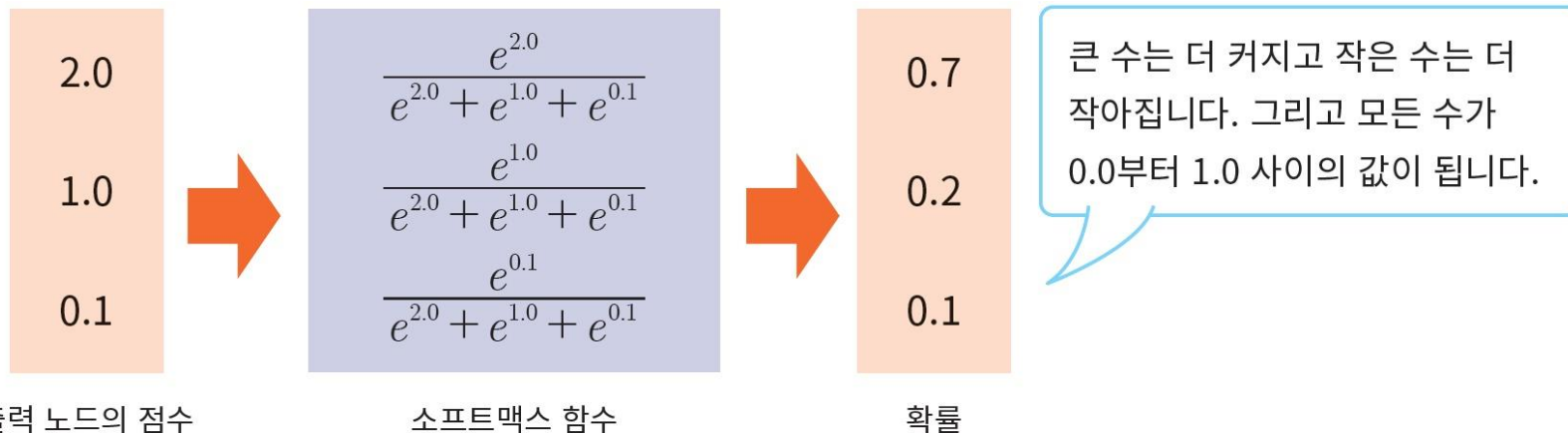
[1.]

[0.]]

정확도(Accuracy): 1.0

0. 새로운 손실 함수

- 출력층의 노드에 소프트맥스 함수를 사용하고 손실 함수로 교차 엔트로피를 많이 사용한다
- 소프트 맥스 활성화 함수



- 교차 엔트로피는 2개의 확률분포 간의 거리를 측정한 것이다.
- 교차 엔트로피는 2개의 확률 분포 p, q 에 대해서 다음과 같이 정의된다.

$$H(p, q) = - \sum_x p(x) \log q(x)$$

- 교차 엔트로피가 크면, 2개의 확률 분포가 많이 다른 것이다. 교차 엔트로피가 작으면 2개의 확률 분포가 거의 일치한다고 볼 수 있다.

0. 교차 엔트로피 손실 함수

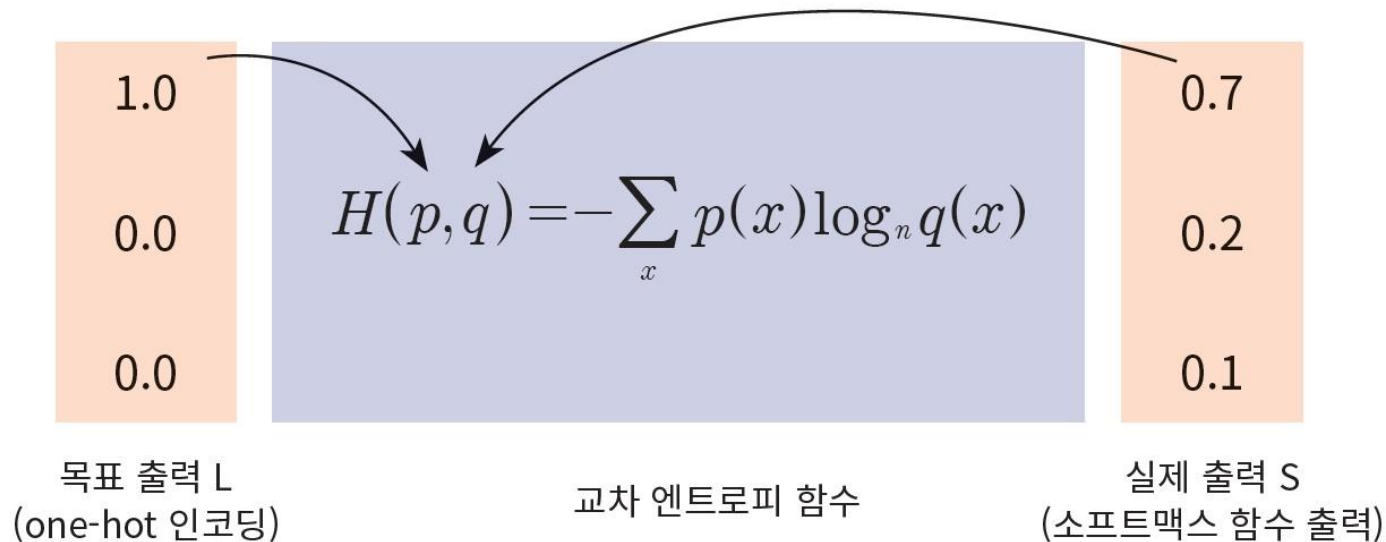


그림 14-9 교차 엔트로피 함수

$$\begin{aligned}
 H(p, q) &= - \sum_x p(x) \log_n q(x) \\
 &= - (1.0 * \log 0.7 + 0.0 * \log 0.2 + 0.0 * \log 0.1) \\
 &= 0.154901
 \end{aligned}$$

0. 교차 엔트로피 계산

입력 샘플	실제 출력			목표 출력		
샘플 #1	0.1	0.3	0.6	0	0	1
샘플 #2	0.2	0.6	0.2	0	1	0
샘플 #3	0.3	0.4	0.3	1	0	0

- ▷ 첫 번째 샘플에 대하여 교차 엔트로피를 계산해보자. $-(\log(0.1) * 0 + \log(0.3) * 0 + \log(0.6) * 1) = -(0 + 0 - 0.51) = 0.51$ 이 된다.
- ▷ 두 번째 샘플의 교차 엔트로피는 $-(\log(0.2) * 0 + \log(0.6) * 1 + \log(0.2) * 0) = -(0 - 0.51 + 0) = 0.51$ 이다.
- ▷ 세 번째 샘플의 교차 엔트로피는 $-(\log(0.3) * 1 + \log(0.4) * 0 + \log(0.3) * 0) = -(-1.2 + 0 + 0) = 1.20$ 이다.
- ▷ 따라서 3개 샘플의 평균 교차 엔트로피 오류는 $(0.51 + 0.51 + 1.20) / 3 = 0.74$ 가 된다.

0. 인공 신경망

MNIST 성능 향상

- 다층 퍼셉트론을 사용해 MNIST 모델의 성능을 향상시킬 수 있다.

```
1 import torch
2 import torchvision.datasets as dsets
3 import torchvision.transforms as transforms
4 from torch.utils.data import DataLoader
5 import torch.nn as nn
6 import matplotlib.pyplot as plt
7 import random
8
9 USE_CUDA = torch.cuda.is_available() # GPU를 사용가능하면 True, 아니라면 False를 리턴
10 device = torch.device("cuda" if USE_CUDA else "cpu") # GPU 사용 가능하면 사용하고 아니면 CPU 사용
11 print("다음 기기로 학습합니다:", device)
12
13 # for reproducibility
14 random.seed(777)
15 torch.manual_seed(777)
16 if device == 'cuda':
17     torch.cuda.manual_seed_all(777)
18
19 # MNIST dataset
20 mnist_train = dsets.MNIST(root='MNIST_data/',
21                           train=True,
22                           transform=transforms.ToTensor(),
23                           download=True)
24
25 mnist_test = dsets.MNIST(root='MNIST_data/',
26                          train=False,
27                          transform=transforms.ToTensor(),
28                          download=True)
29
30
31
```

0. 인공 신경망

MNIST 성능 향상

- 다층 퍼셉트론을 사용해 MNIST 모델의 성능을 향상시킬 수 있다.

```
1 # MNIST data image of shape 28 * 28 = 784
2 linear = nn.Sequential()
3 linear.add_module('fc1', nn.Linear(28*28*1, 100))
4 linear.add_module('relu1', nn.ReLU())
5 linear.add_module('fc2', nn.Linear(100, 100))
6 linear.add_module('relu2', nn.ReLU())
7 linear.add_module('fc3', nn.Linear(100, 10))
8 # MNIST data image of shape 28 * 28 = 784
9
10 # hyperparameters
11 training_epochs = 15
12 batch_size = 100
13
14 # 비용 함수와 옵티마이저 정의
15 criterion = nn.CrossEntropyLoss().to(device) # 내부적으로 소프트맥스 함수를 포함하고 있음.
16 optimizer = torch.optim.SGD(linear.parameters(), lr=0.1)
17
18 # dataset loader
19 data_loader = DataLoader(dataset=mnist_train,
20                           batch_size=batch_size, # 배치 크기는 100
21                           shuffle=True,
22                           drop_last=True)
```

0. 인공 신경망

MNIST 성능 향상

- 단층 퍼셉트론보다 비용이 작아진 것을 확인할 수 있다.

```
1 for epoch in range(training_epochs): # 앞서 training_epochs의 값은 15로 지정함.
2     avg_cost = 0
3     total_batch = len(data_loader)
4
5     for X, Y in data_loader:
6         # 배치 크기가 100이므로 아래의 연산에서 X는 (100, 784)의 텐서가 된다.
7         X = X.view(-1, 28 * 28).to(device)
8         # 레이블은 원-핫 인코딩이 된 상태가 아니라 0 ~ 9의 정수.
9         Y = Y.to(device)
10
11         optimizer.zero_grad()
12         hypothesis = linear(X)
13         cost = criterion(hypothesis, Y)
14         cost.backward()
15         optimizer.step()
16
17         avg_cost += cost / total_batch
18
19     print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))
20
21 print('Learning finished')
```

```
Epoch: 0001 cost = 0.628847122
Epoch: 0002 cost = 0.239227369
Epoch: 0003 cost = 0.175555333
Epoch: 0004 cost = 0.138709322
Epoch: 0005 cost = 0.115421154
Epoch: 0006 cost = 0.096607938
Epoch: 0007 cost = 0.083727077
Epoch: 0008 cost = 0.072155662
Epoch: 0009 cost = 0.063731015
Epoch: 0010 cost = 0.056700680
Epoch: 0011 cost = 0.050654452
Epoch: 0012 cost = 0.044952631
Epoch: 0013 cost = 0.040506925
Epoch: 0014 cost = 0.036337849
Epoch: 0015 cost = 0.032280162
Learning finished
```

0. 인공 신경망

MNIST 성능 향상

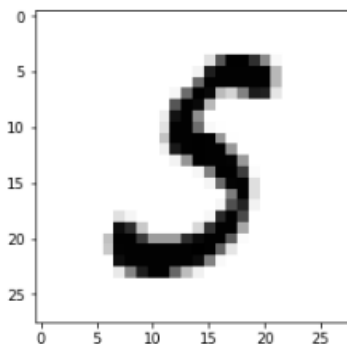
- 정확도 역시 89% -> 97% 로 증가하였다.

```
1 # 테스트 데이터를 사용하여 모델을 테스트한다.
2 with torch.no_grad(): # torch.no_grad()를 하면 gradient 계산을 수행하지 않는다.
3     X_test = mnist_test.test_data.view(-1, 28 * 28).float().to(device)
4     Y_test = mnist_test.test_labels.to(device)
5
6     prediction = linear(X_test)
7     correct_prediction = torch.argmax(prediction, 1) == Y_test
8     accuracy = correct_prediction.float().mean()
9     print('Accuracy:', accuracy.item())
10
11 # MNIST 테스트 데이터에서 무작위로 하나를 뽑아서 예측을 해본다
12 r = random.randint(0, len(mnist_test) - 1)
13 X_single_data = mnist_test.test_data[r:r + 1].view(-1, 28 * 28).float().to(device)
14 Y_single_data = mnist_test.test_labels[r:r + 1].to(device)
15
16 print('Label: ', Y_single_data.item())
17 single_prediction = linear(X_single_data)
18 print('Prediction: ', torch.argmax(single_prediction, 1).item())
19
20 plt.imshow(mnist_test.test_data[r:r + 1].view(28, 28), cmap='Greys', interpolation='nearest')
21 plt.show()
```

Accuracy: 0.9725000262260437

Label: 5

Prediction: 5



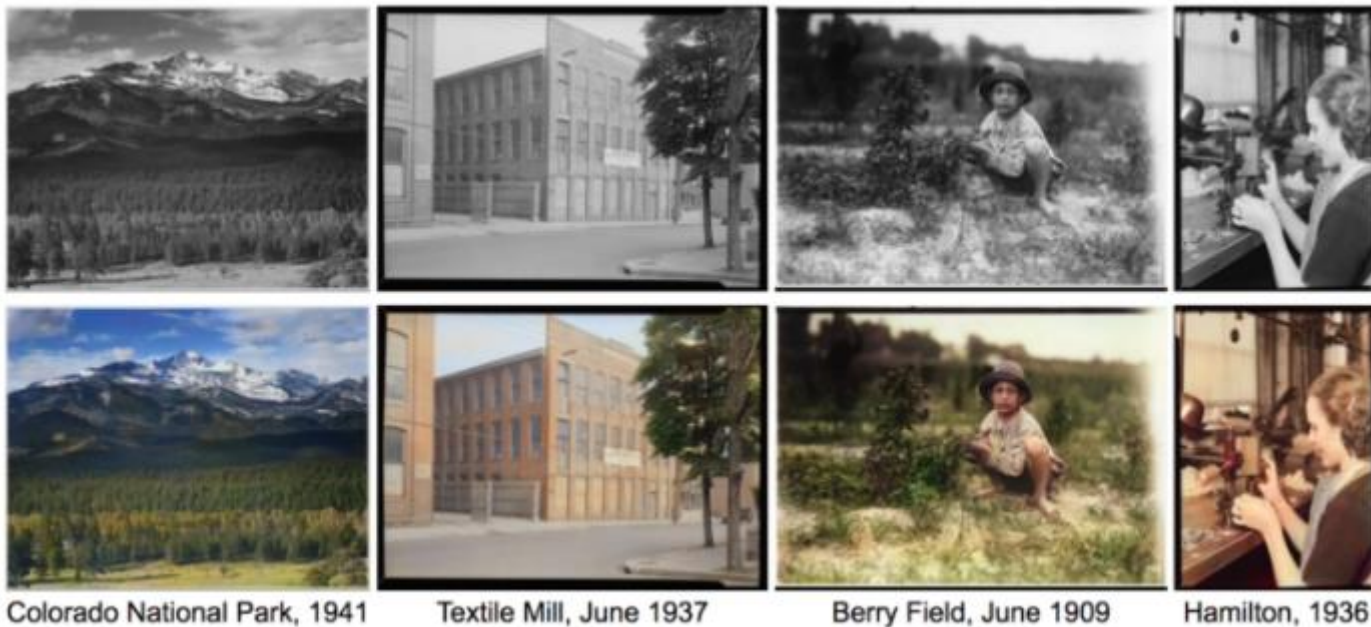
1. CNN 적용 사례

- 정치인 재연



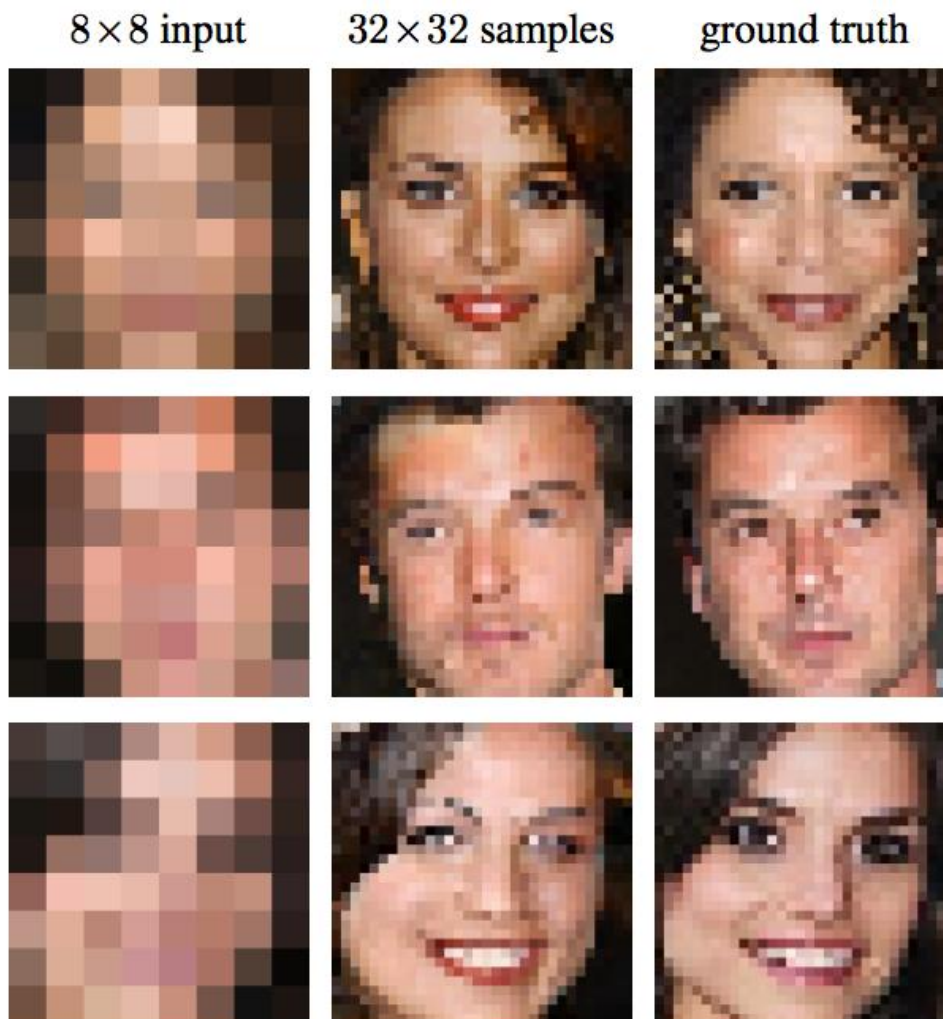
1. CNN 적용 사례

- 흑백 사진과 영상에 색 복원



1. CNN 적용 사례

- 픽셀 복원



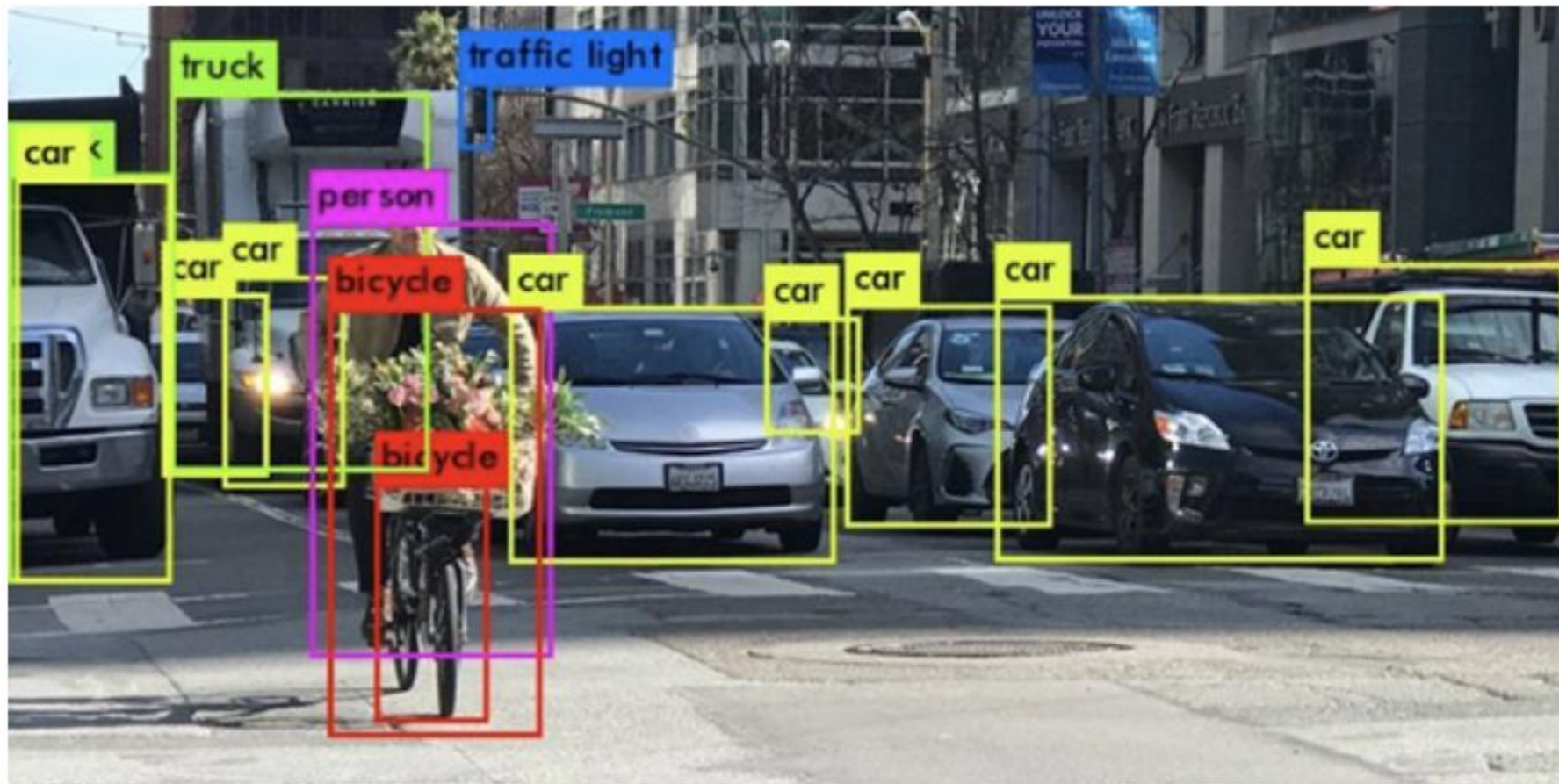
1. CNN 적용 사례

- 실시간으로 여러 사람의 움직임을 추정



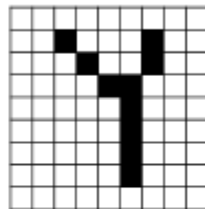
1. CNN 적용 사례

- 이미지 객체 인식



2. CNN 이란?

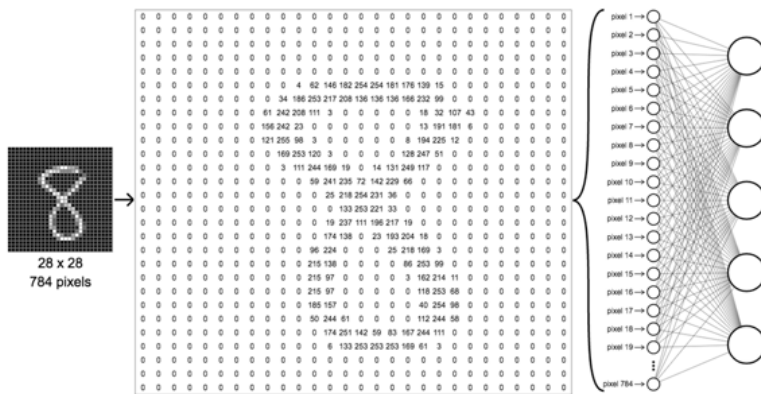
- 기존 신경망 이미지 처리의 한계
 - 공간적 / 지역적 정보의 유실



↓ 변환



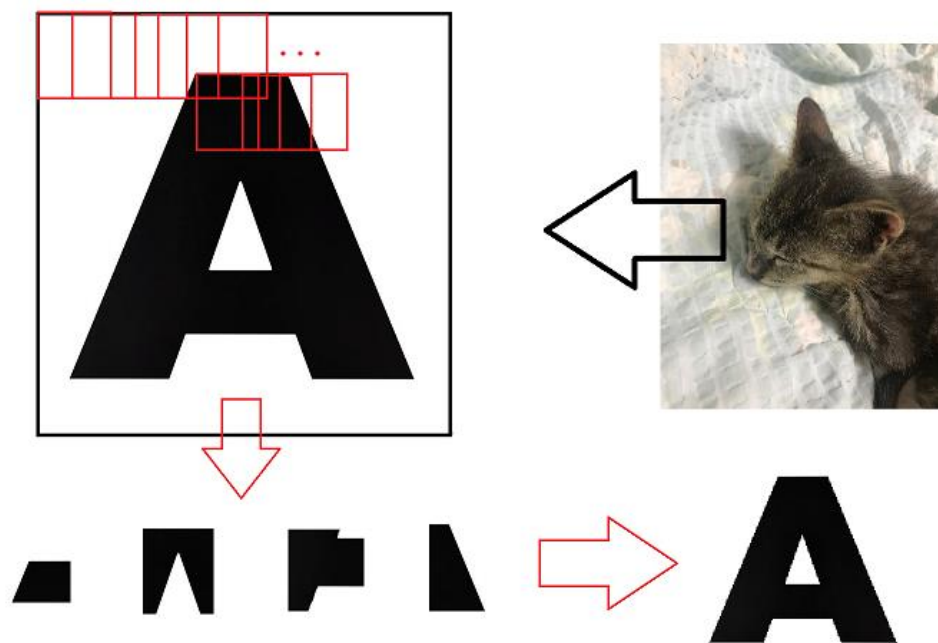
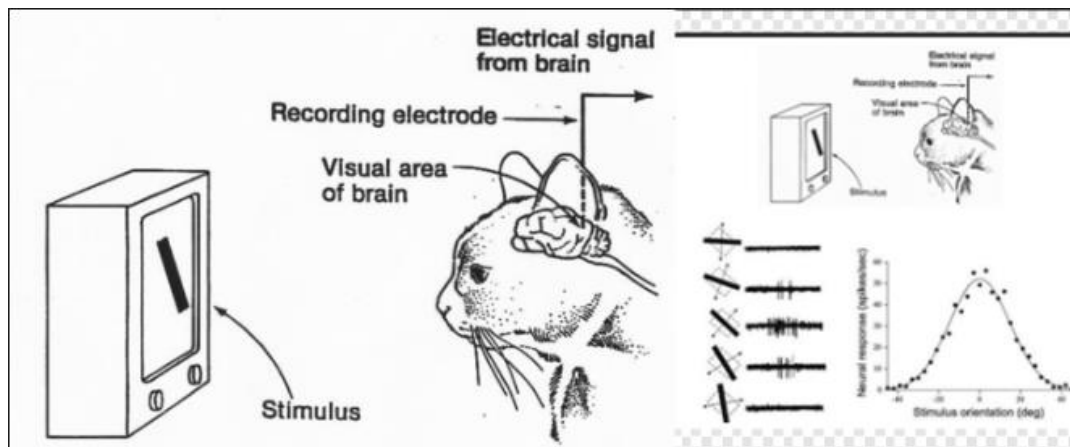
- 이미지의 크기가 커질수록 증가하는 가중치 학습 크기



$$28 \times 28 = 784$$

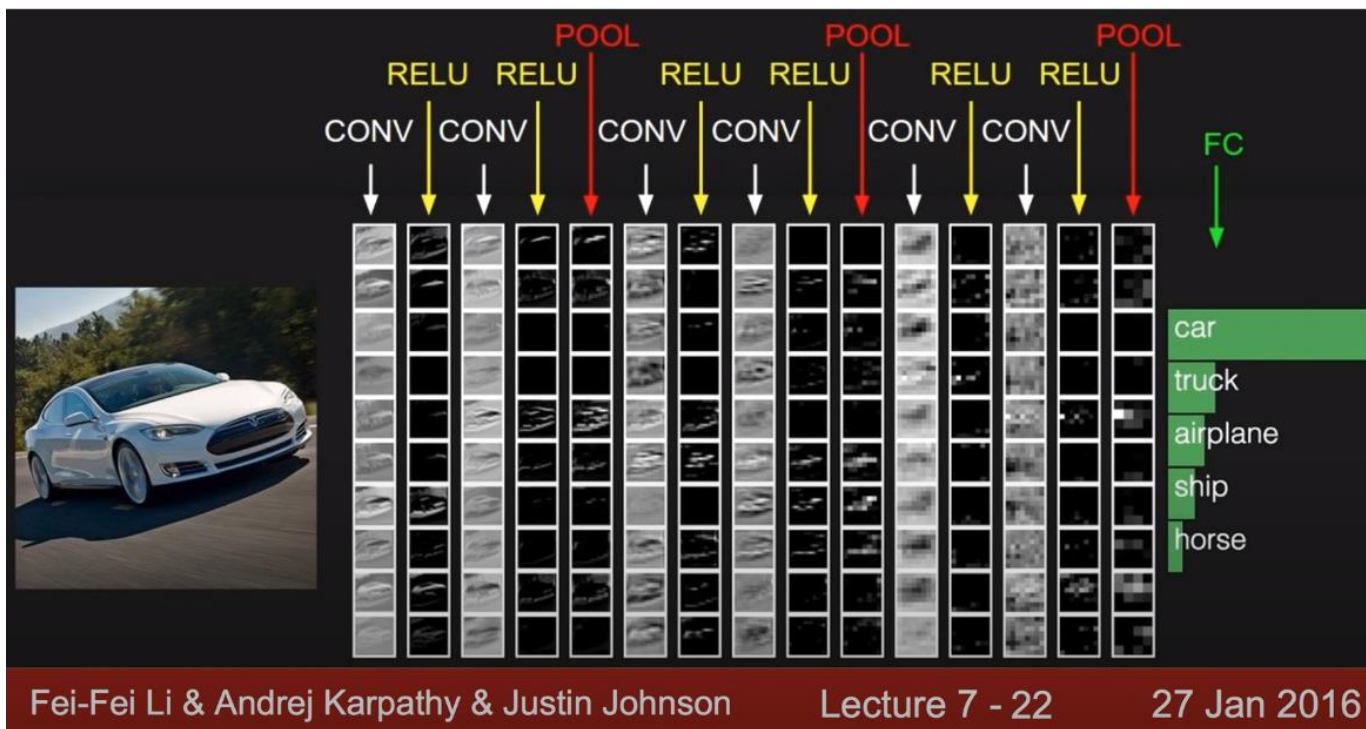
$$1024 \times 768 = 786,432$$

2. CNN 이란?

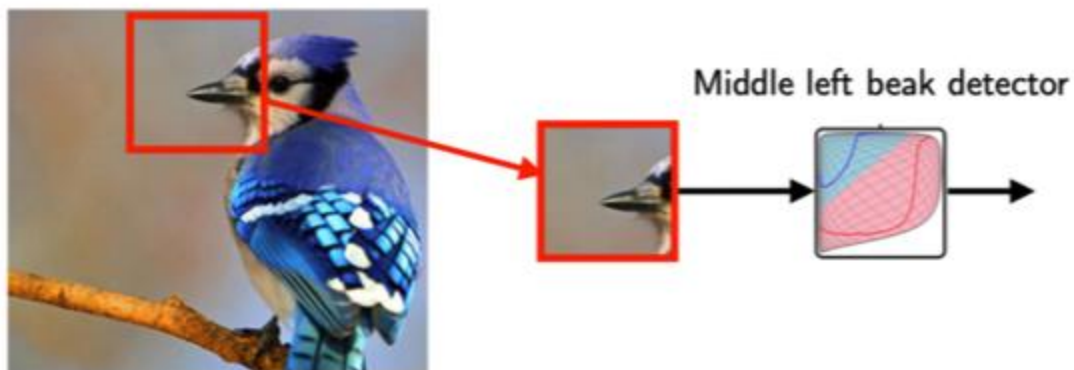
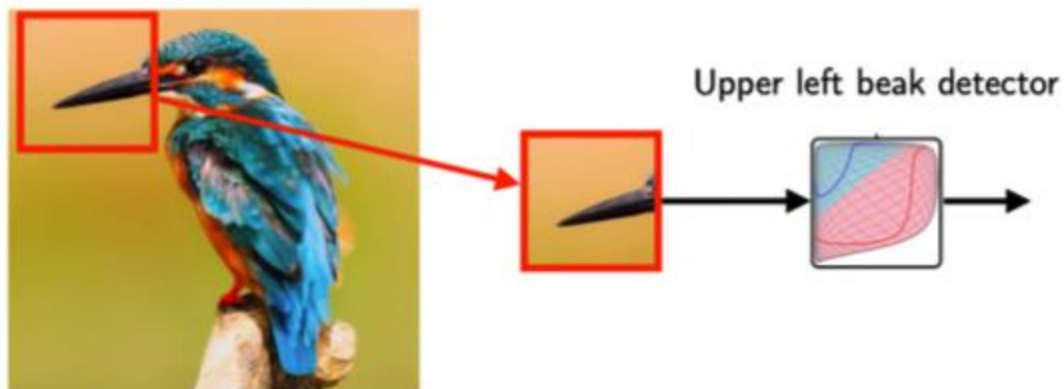


2. CNN 이란?

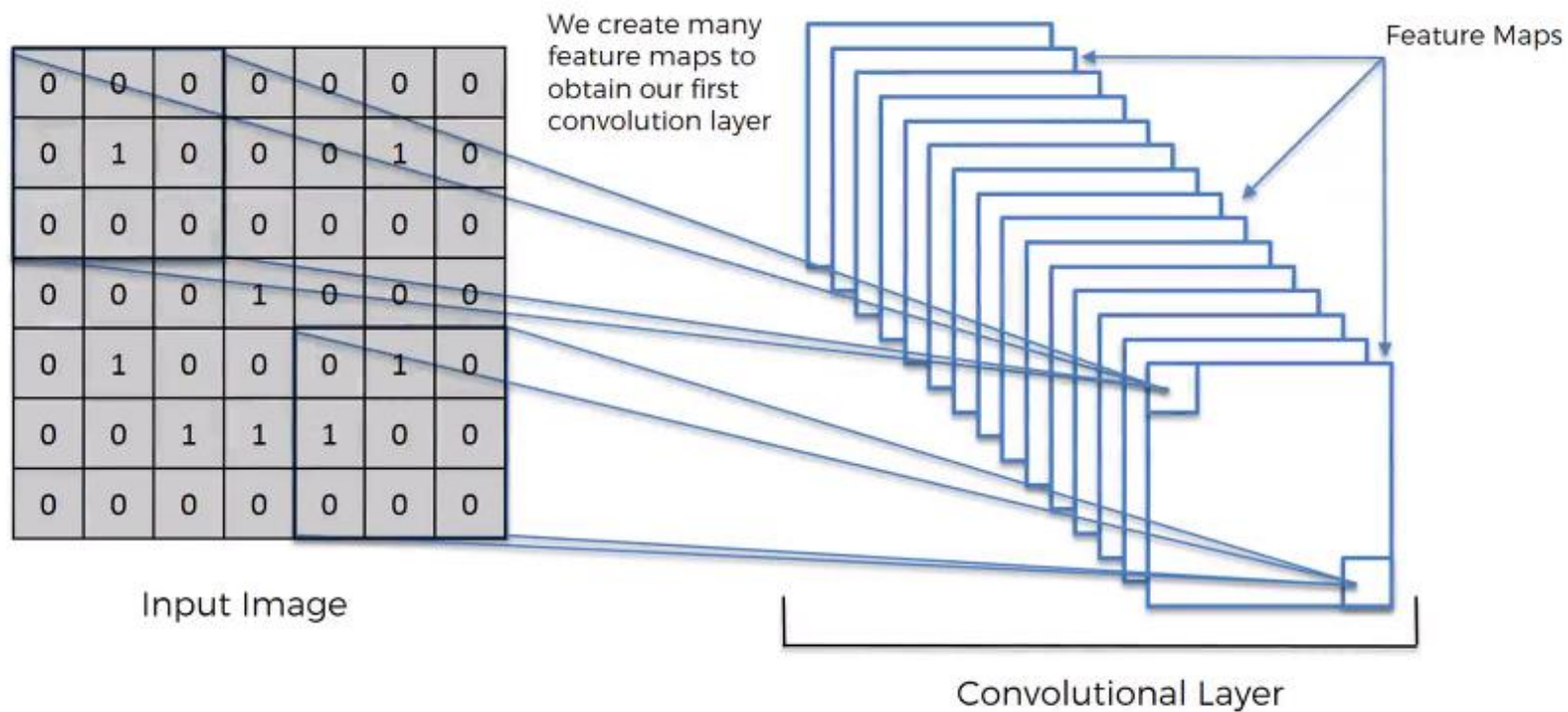
- Convolutional Neural Networks(CNN)
 - 시각적 영상을 분석하는 데 사용되는 인공신경망의 한 종류
 - 영상 및 동영상 인식, 추천 시스템, 영상 분류, 의료 영상 분석, 자연어 처리 등 다양한 분야에서 사용



2. CNN 이란?



2. CNN 이란?



2. CNN 이란?

- Convolution에서 Feature Map을 만들어 내는 과정

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

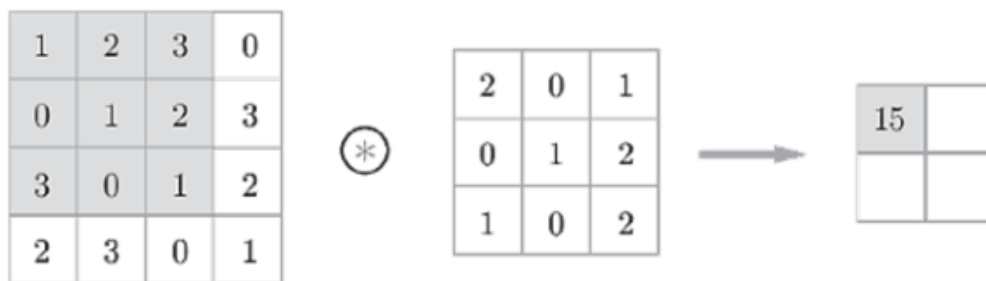
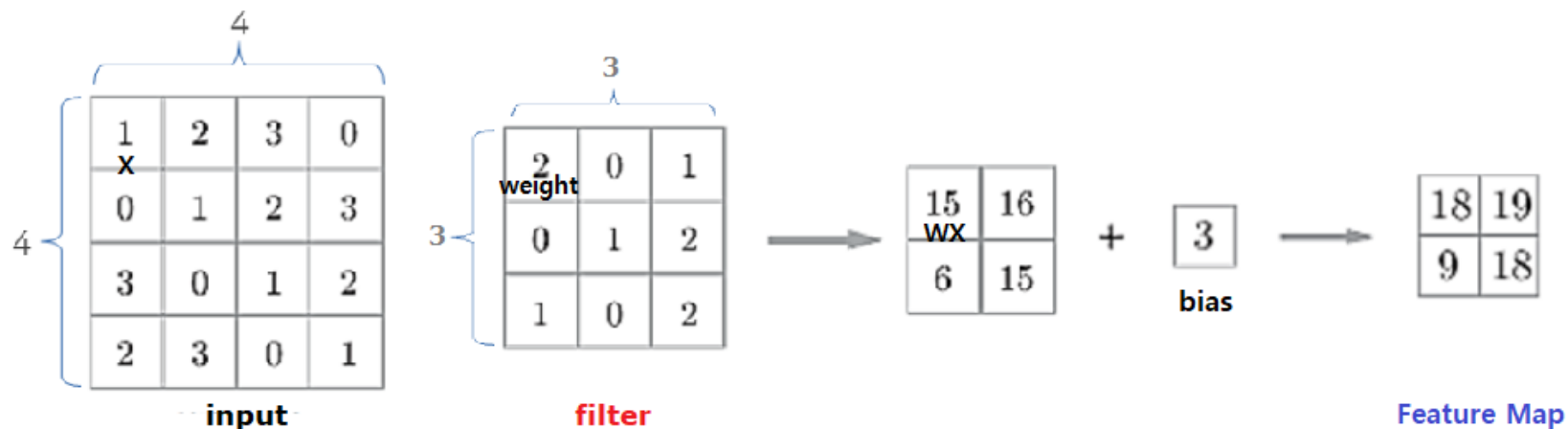
Image

4		

Convolved
Feature

2. CNN 이란?

- Convolution에서 Feature Map을 만들어 내는 과정



$$1 \times 2 + 2 \times 0 + 3 \times 1 + \dots + 0 \times 0 + 1 \times 2 = 15$$

2. CNN 이란?

- Convolution에서 Feature Map을 만들어 내는 과정



-1	0	+1
-2	0	+2
-1	0	+1

**Sobel-X
(vertical)**

+1	+2	+1
0	0	0
-1	-2	-1

**Sobel-Y
(horizontal)**



2. CNN 이란?

- Padding이 필요한 이유
 - 모서리의 정보들의 연산이 가운데 정보들의 연산보다 적음

7

1	2	3	3	3	2	1
2	4					
3	6	9				
3			9			

7

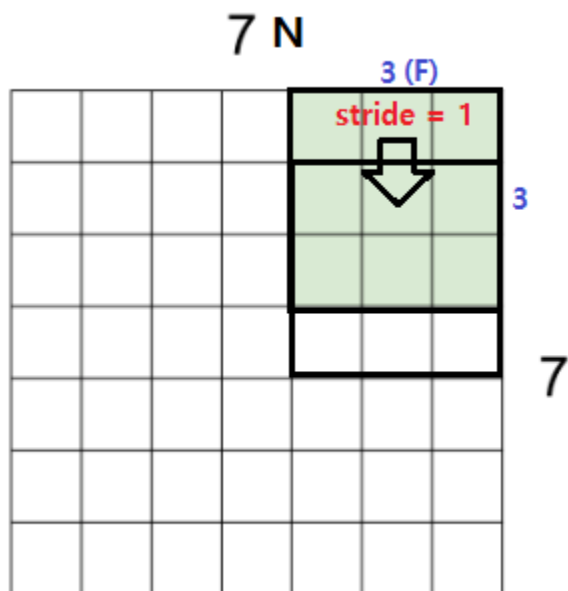
filter : 3X3, stride : 1

2. CNN 이란?

- Padding이 필요한 이유

- 전체 이미지 사이즈와 Filter 사이즈, Stride를 알면 Filter map의 사이즈 계산이 가능
- Filter Map이 작아지는 특징이 있음

※ Stride - Filter가 계산하며 이동하는 칸 수



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

Output size:
 $(N - F) / \text{stride} + 1$

e.g. $N = 7, F = 3$:

stride 1 => $(7 - 3) / 1 + 1 = 5$

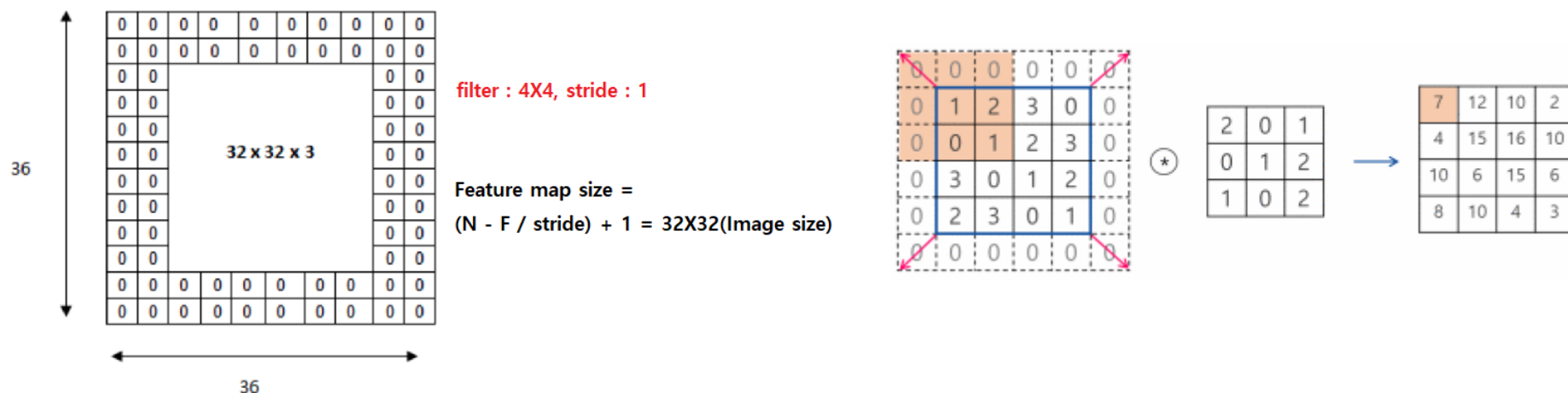
stride 2 => $(7 - 3) / 2 + 1 = 3$

stride 3 => $(7 - 3) / 3 + 1 = 2.33 \therefore 1$

2. CNN 이란?

• Padding이란?

- 앞서 설명한 문제점을 방지하고자 합성곱 연산 전에 선 처리하는 작업
- 입력 데이터 주변을 특정 값(Hyper-Parameter)로 채워 공간적 크기를 조절하는 방법
- 주로 Hyper-Parameter를 0으로 하는 Zero-Padding을 많이 사용
- 덮는 층을 얼마나 두껍게 할지를 조절하여 Feature map을 원하는 사이즈로 조절 가능



※ Padding에서 0의 의미 : 의미 없는 정보, 가중치(W)는 0의 정보를 무시하게 된다.

2. CNN 이란?

- 출력 크기 계산 공식

$$(OH, OW) = \left(\frac{H + 2P - FH}{S} + 1, \frac{W + 2P - FW}{S} + 1 \right)$$

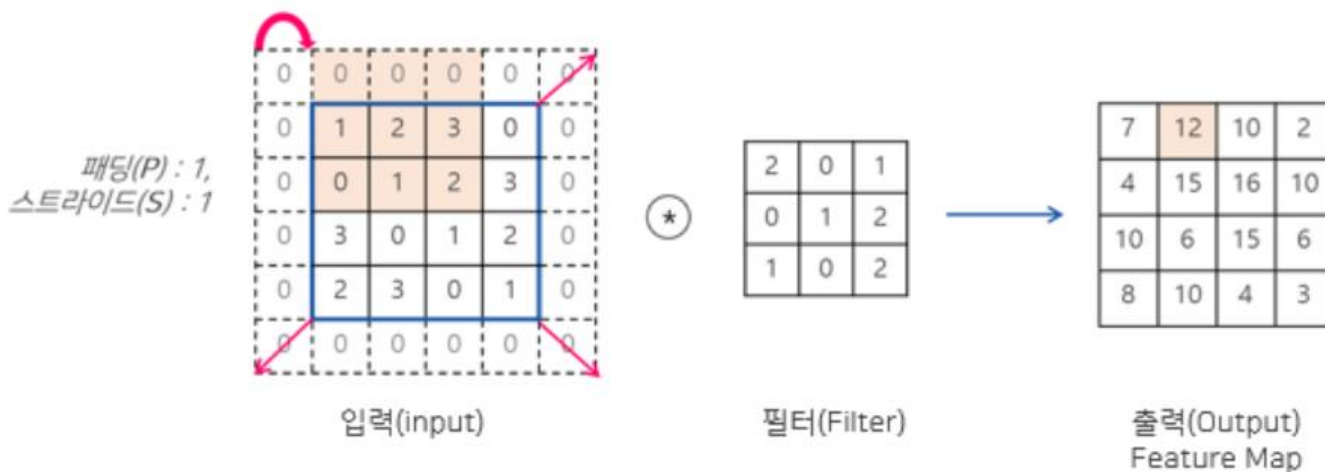
(H, W) : 입력 크기(input size)

(FH, FW) : 필터 크기 (filter/kernel size)

S : 스트라이드(stride)

P : 패딩(padding)

(OH, OW) : 출력 크기 (output size)

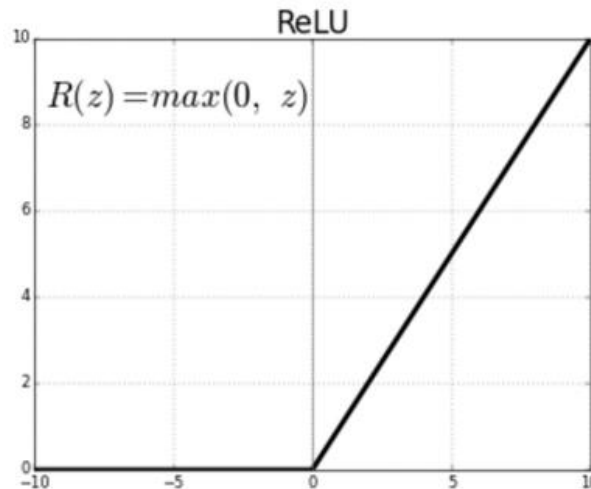


$$(OH, OW) = \left(\frac{4 + 2 \times 1 - 3}{1} + 1, \frac{4 + 2 \times 1 - 3}{1} + 1 \right) = (4, 4)$$

2. CNN 이란?

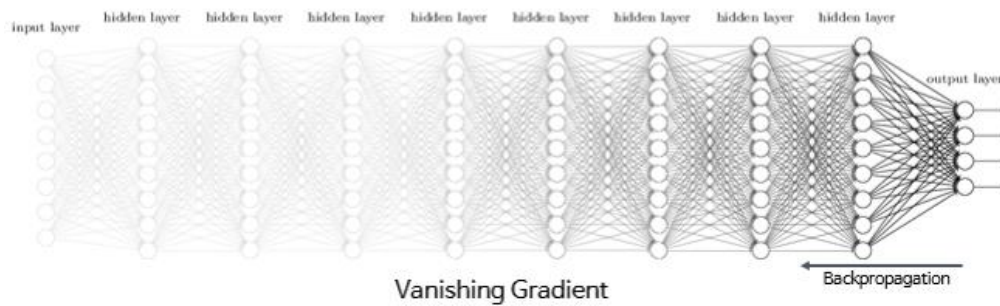
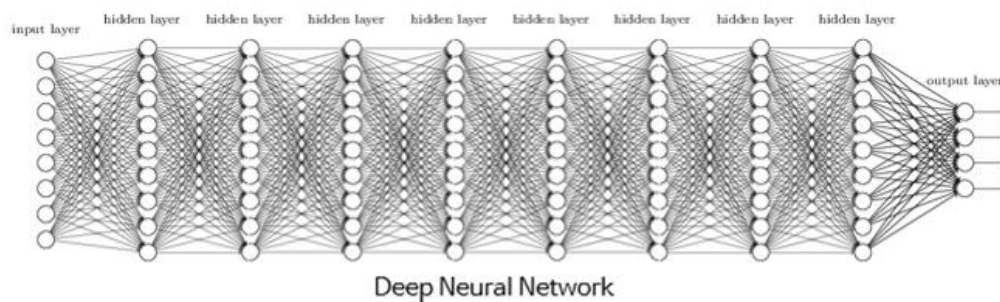
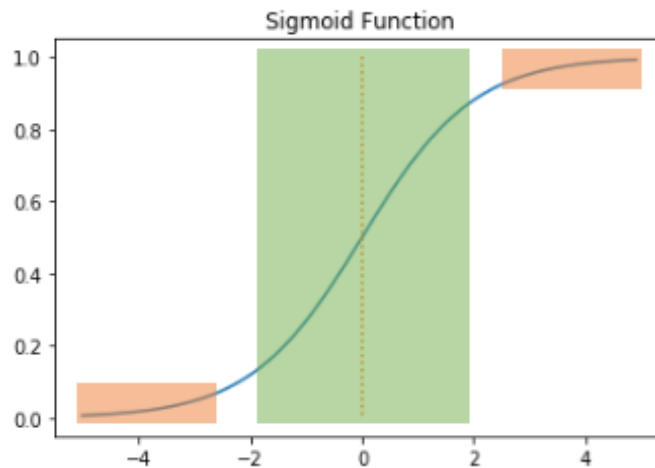
• 활성화 함수(Activation Function)

- 필터 들을 통해 특징맵이 추출되면 이 특징맵에 활성화 함수를 적용하여 값을 활성화 시킨다.
- 예를 들면, 곡선 값의 특징을 가지고 있는 필터를 통해 입력 데이터를 적용시키면 정량적인 값이 나오는데 이러한 값들을 "곡선 특징을 얼마나 어떻게 가지고 있는지", "없다" 로 바꾸어 주는 과정이 필요하다.
- CNN 에서는 주로 ReLU 함수를 사용한다.
- Sigmoid, Tanh 함수대신 ReLU 함수를 사용하는 이유
 - 1) 층을 깊게 가져가는 CNN 특성상 역전파 과정에서 Gradient Vanishing 문제가 발생
 - 2) 속도가 다른 활성화 함수들 보다 빠름



2. CNN 이란?

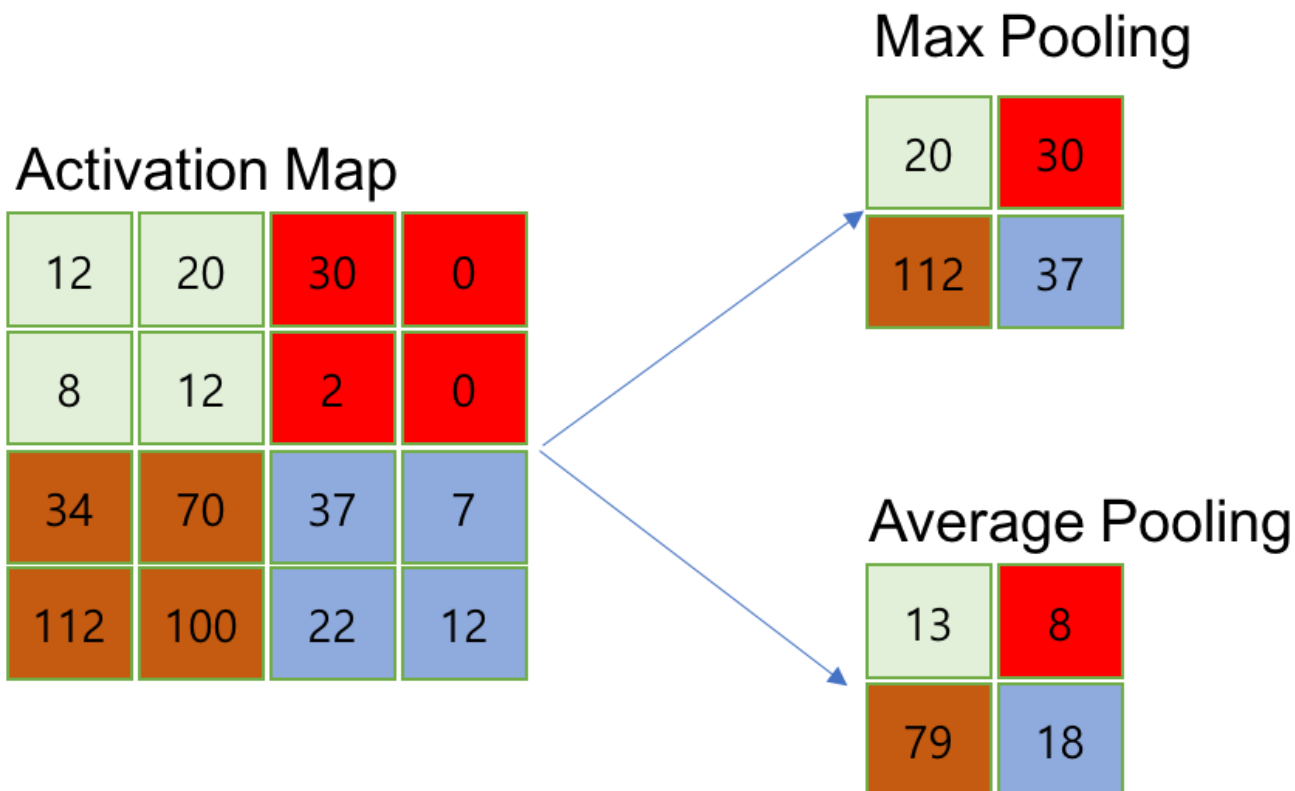
- Gradient Vanishing



2. CNN 이란?

• Pooling이란?

- Convolution Layer의 출력 데이터를 입력으로 받아 출력 데이터의 크기를 줄이거나 특정 데이터를 강조하는 용도로 사용
- Max, Average, Min Pooling 이 있음 ※ 주로 Max Pooling 사용



2. CNN 이란?

- Pooling Layer의 특징

- Parameter가 없어 학습이 이루어지지 않고, Hyper-Parameter만 조정가능

- ※ Max Pooling과 Average Pooling의 차이점

- Max Pooling

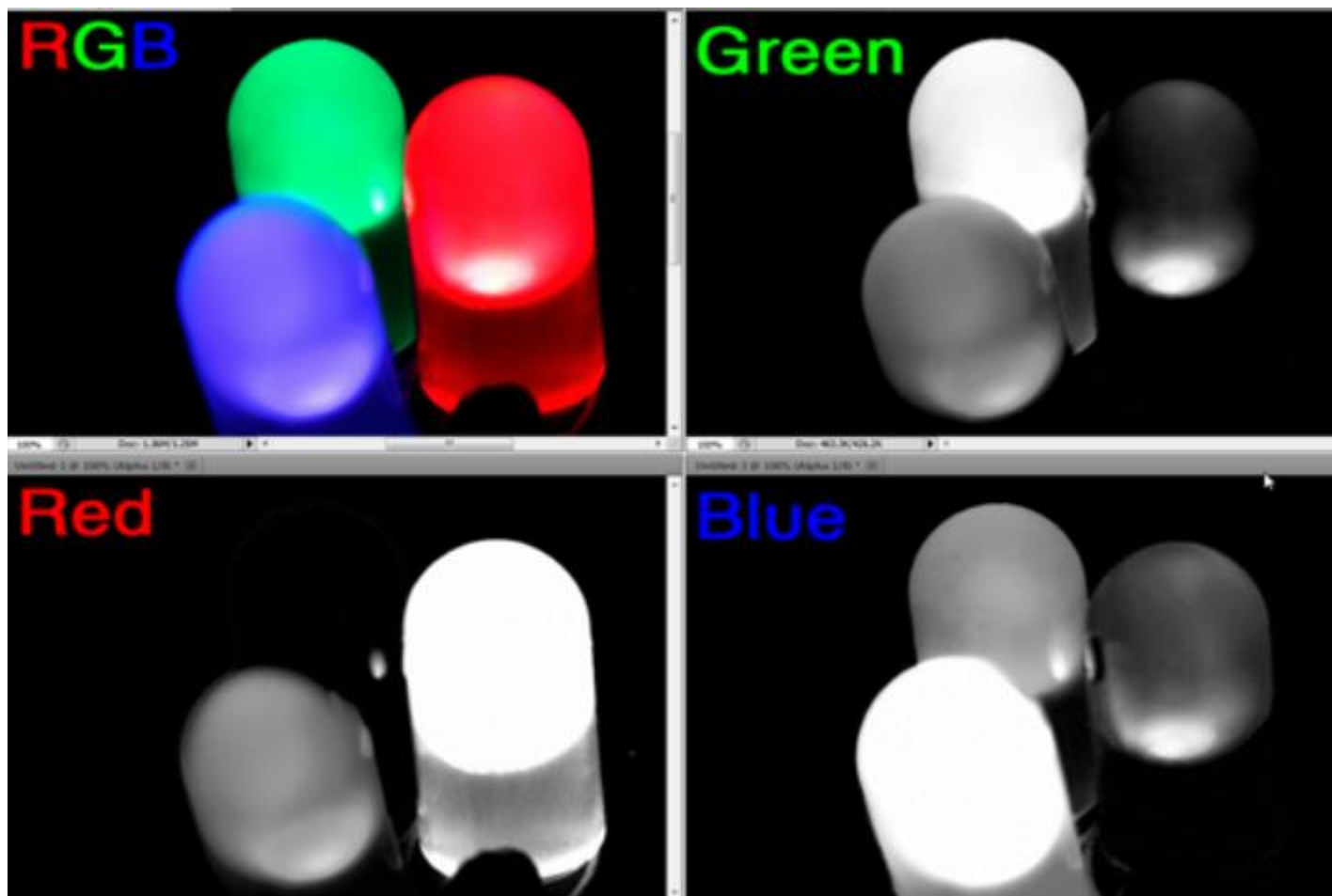
- Input Layer에 가까울 수록 필요 없는 정보 혹은 중복된 정보들이 많음
- 즉 Noise가 많아 Max Pooling을 사용하여 Special한 정보만 추출

- Average Pooling

- Output Layer에 가까울 수록 필요한 정보 혹은 의미 있는 정보들이 많음
- 이러한 정보들의 유실을 막기 위해 모든 정보의 평균을 구하는 Average Pooling 사용

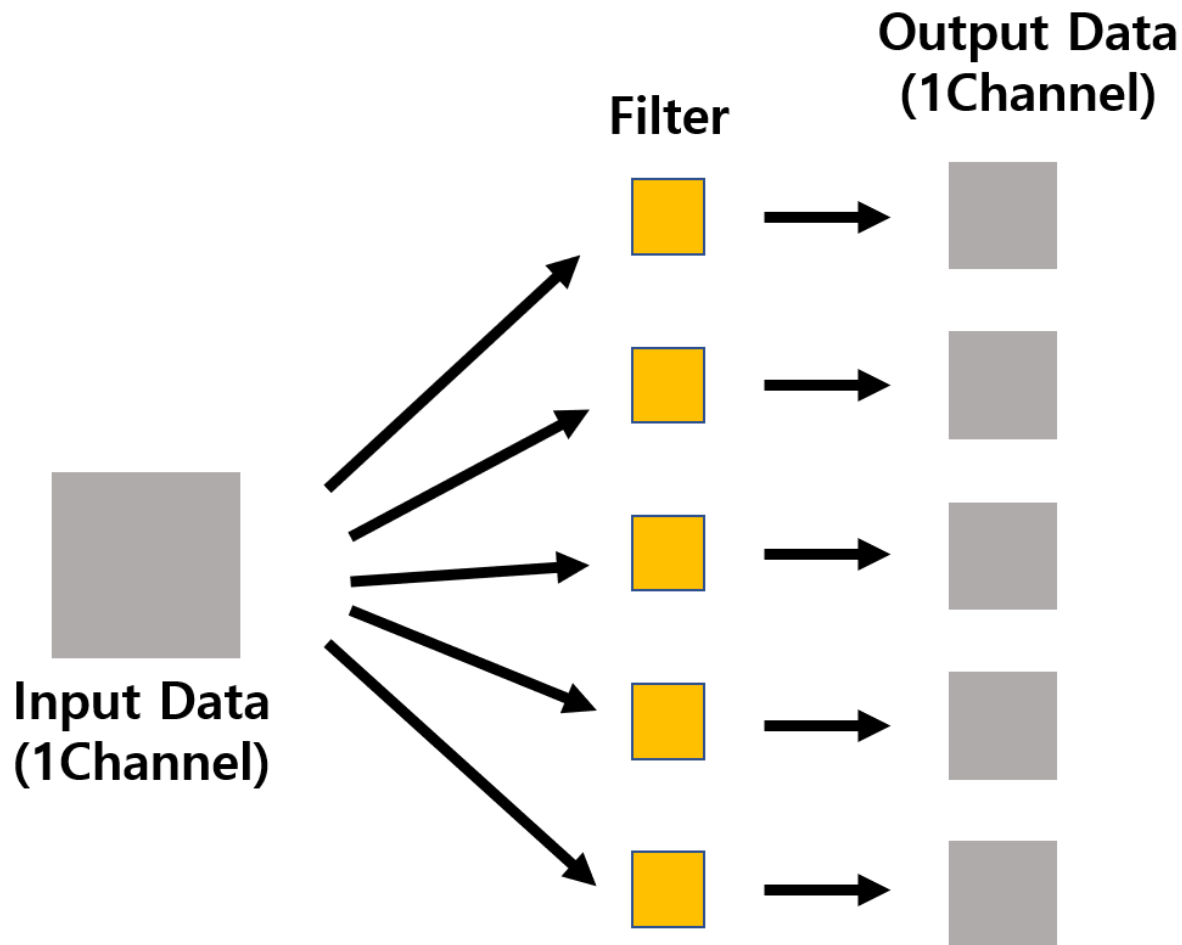
2. CNN 이란?

- Channel



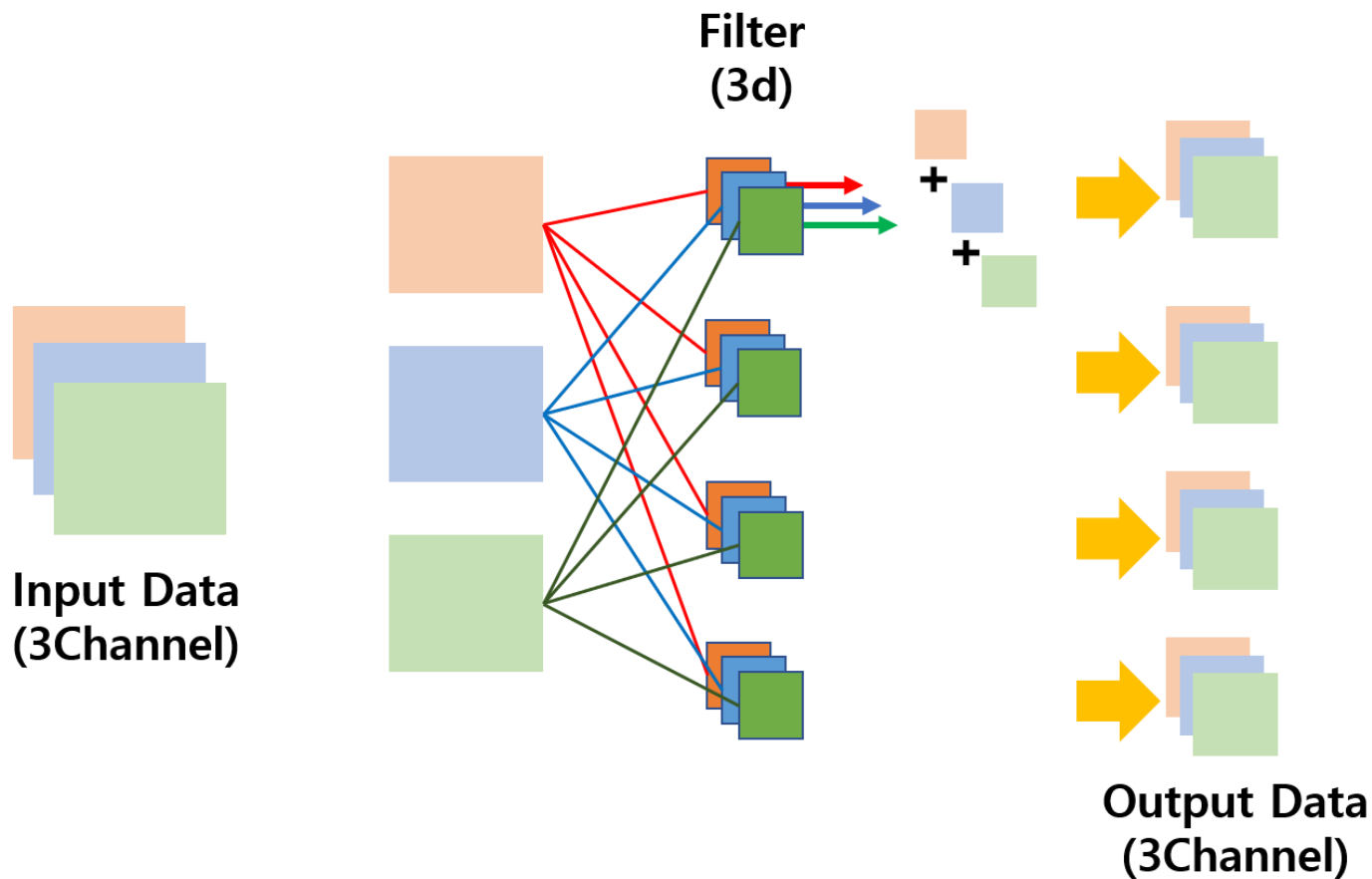
2. CNN 이란?

- Channel / 흑백인 경우

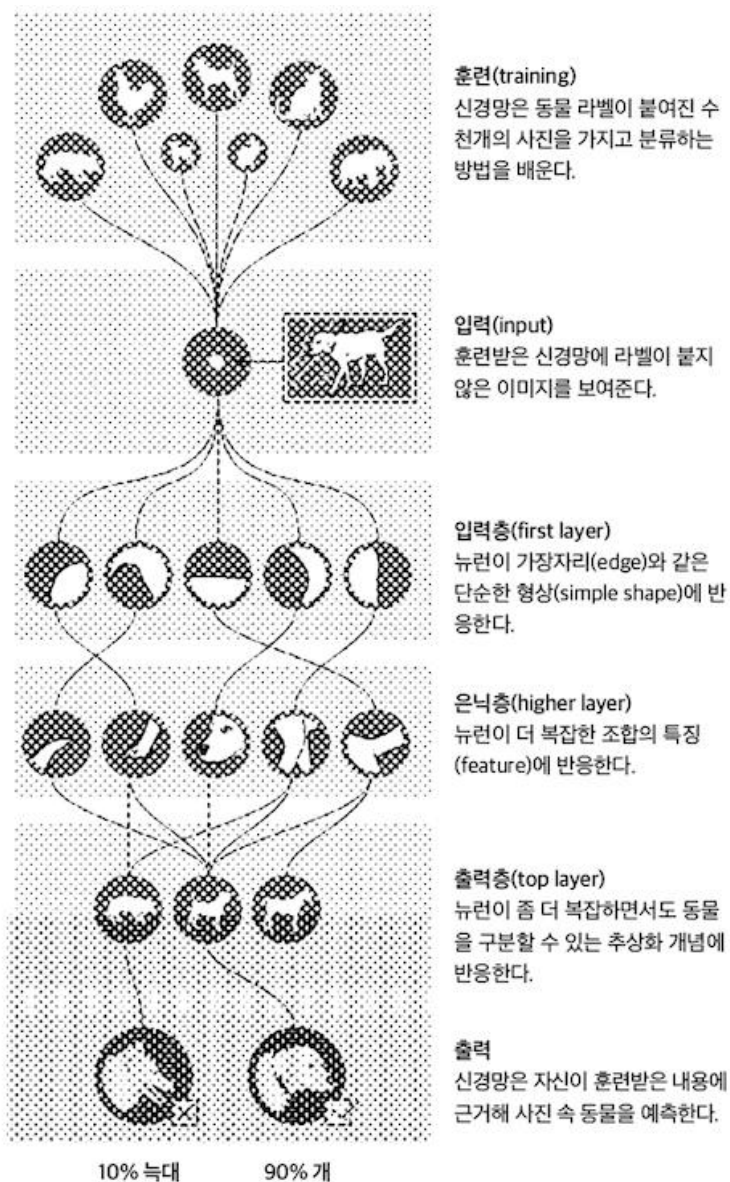


2. CNN 이란?

- Channel / 컬러인 경우

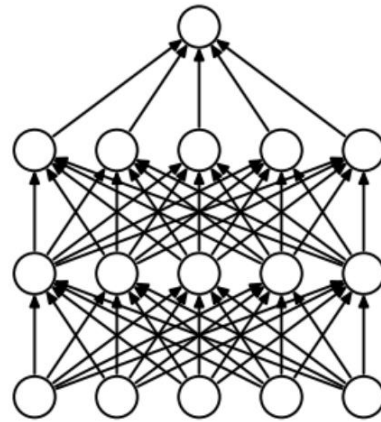


2. CNN 이란?

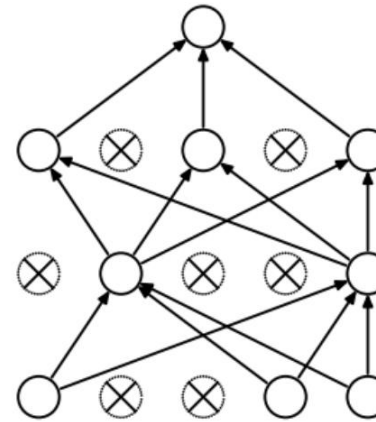


3. Dropout

- Dropout이란?



(a) Standard Neural Net

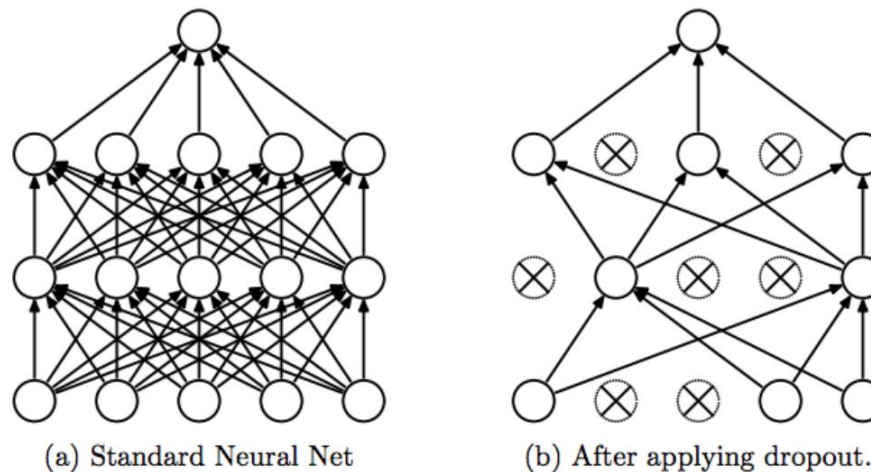


(b) After applying dropout.

- 그림 (a)에 대한 학습을 진행할 때, 망에 있는 모든 Node들에 대한 학습을 수행하지 않고,
- 그림 (b)와 같이 일부 Node를 무작위로 생략하고 학습을 수행.
- 생략되는 Node의 비율은 사용자가 임의로 정해야 함.
- 일정한 Batch 구간 학습 후 생략되는 Node를 재 선정

3. Dropout

- Dropout효과



- Voting 효과

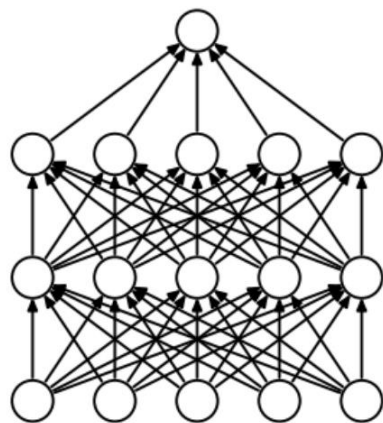
- 일정한 batch 구간 동안 줄어든 망을 이용해 학습 시 줄어든 망에 대하여 오버피팅 되고,
- 다음 Batch 구간 동안은 또 다른 줄어든 망에 대해 오버피팅 되어 Voting에 의한 평균 효과를 얻을 수 있음.

- Co-adaptation을 피하는 효과

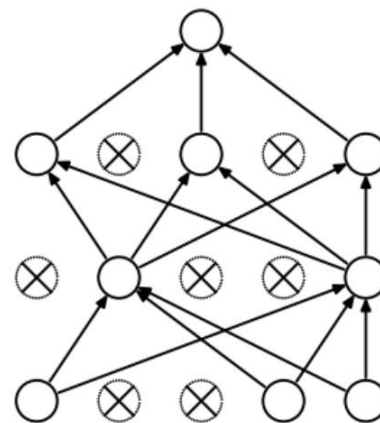
- 특정 노드의 가중치, 바이어스가 큰 값을 가지게 되면 특정 노드의 영향이 커져 다른 노드의 학습 속도가 느려지거나, 학습 진행이 잘 되지 않을 수 있음.
- 즉 특정 노드에 영향을 받아 노드들이 서로 동조화(Co-adaptation) 되는 것을 피할 수 있음.

3. Dropout

- Dropout 적용 방법



(a) Standard Neural Net



(b) After applying dropout.

- 이번 Batch때 사용하지 않을 Node의 가중치를 0으로 임시 조정
- 또는 사용하지 않을 Node에 표식을 하여 표식이 있을 시 건너 뛰기

※ 테스트시에는 설정한 Dropout 확률 값 만큼 결과 값에 곱해줄 필요가 있음

4. 실습

Fashion MNIST

- 옷, 신발 등의 사진을 분류하는 문제

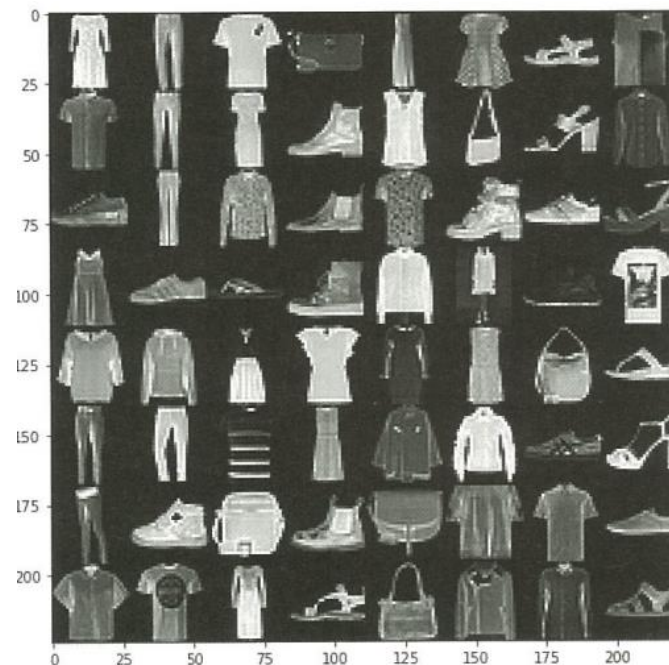
```
1 from torchvision import datasets, transforms, utils
2 from torch.utils import data
3
4 import matplotlib.pyplot as plt
5 import numpy as np
```

- 필요한 라이브러리 임포트

```
1 transform = transforms.Compose([
2     transforms.ToTensor()
3 ])
```

```
1 trainset = datasets.FashionMNIST(
2     root      = './.data/',
3     train     = True,
4     download  = True,
5     transform = transform
6 )
7 testset = datasets.FashionMNIST(
8     root      = './.data/',
9     train     = False,
10    download  = True,
11    transform = transform
12 )
```

- 데이터 다운로드



4. 실습

Fashion MNIST

- 학습에 사용하기 편리하게 데이터 전처리

```
1 batch_size = 16
2
3 train_loader = data.DataLoader(
4     dataset      = trainset,
5     batch_size   = batch_size
6 )
7 test_loader = data.DataLoader(
8     dataset      = testset,
9     batch_size   = batch_size
10 )
```

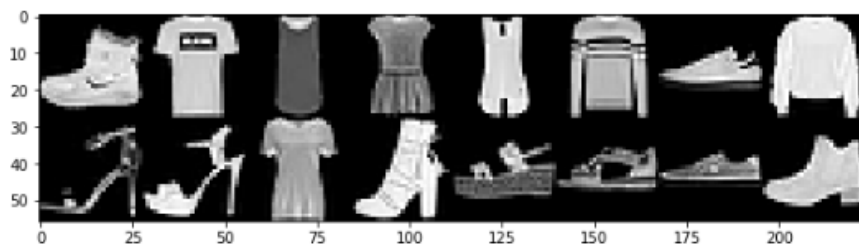
```
1 dataiter      = iter(train_loader)
2 images, labels = next(dataiter)
```

4. 실습

Fashion MNIST

- 학습 데이터 확인

```
1 img = utils.make_grid(images, padding=0)
2 npimg = img.numpy()
3 plt.figure(figsize=(10, 7))
4 plt.imshow(np.transpose(npimg, (1,2,0)))
5 plt.show()
```



- 라벨링 확인

```
1 print(labels)
tensor([9, 0, 0, 3, 0, 2, 7, 2, 5, 5, 0, 9, 5, 5, 7, 9])
```

```
1 CLASSES = {
2     0: 'T-shirt/top',
3     1: 'Trouser',
4     2: 'Pullover',
5     3: 'Dress',
6     4: 'Coat',
7     5: 'Sandal',
8     6: 'Shirt',
9     7: 'Sneaker',
10    8: 'Bag',
11    9: 'Ankle boot'
12 }
13
14
15 for label in labels:
16     index = label.item()
17     print(CLASSES[index])
```

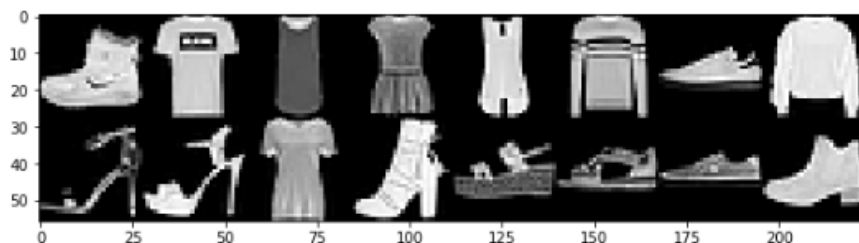
```
Ankle boot
T-shirt/top
T-shirt/top
Dress
T-shirt/top
Pullover
Sneaker
Pullover
Sandal
Sandal
T-shirt/top
Ankle boot
Sandal
Sandal
Sneaker
Ankle boot
```

4. 실습

Fashion MNIST

- 학습 데이터 확인

```
1 img = utils.make_grid(images, padding=0)
2 npimg = img.numpy()
3 plt.figure(figsize=(10, 7))
4 plt.imshow(np.transpose(npimg, (1,2,0)))
5 plt.show()
```



- 라벨링 확인

```
1 print(labels)
tensor([9, 0, 0, 3, 0, 2, 7, 2, 5, 5, 0, 9, 5, 5, 7, 9])
```

```
1 CLASSES = {
2     0: 'T-shirt/top',
3     1: 'Trouser',
4     2: 'Pullover',
5     3: 'Dress',
6     4: 'Coat',
7     5: 'Sandal',
8     6: 'Shirt',
9     7: 'Sneaker',
10    8: 'Bag',
11    9: 'Ankle boot'
12 }
13
14
15 for label in labels:
16     index = label.item()
17     print(CLASSES[index])
```

```
Ankle boot
T-shirt/top
T-shirt/top
Dress
T-shirt/top
Pullover
Sneaker
Pullover
Sandal
Sandal
T-shirt/top
Ankle boot
Sandal
Sandal
Sneaker
Ankle boot
```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 필요한 라이브러리 импорт

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torch.nn.functional as F
5 from torchvision import transforms, datasets
6
7 USE_CUDA = torch.cuda.is_available()
8 DEVICE = torch.device("cuda" if USE_CUDA else "cpu")
9 EPOCHS = 30
10 BATCH_SIZE = 64
```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 데이터셋 불러오기

```
1 transform = transforms.Compose([
2     transforms.ToTensor()
3 ])
4 trainset = datasets.FashionMNIST(
5     root      = './.data/',
6     train     = True,
7     download  = True,
8     transform = transform
9 )
10 testset = datasets.FashionMNIST(
11     root      = './.data/',
12     train     = False,
13     download  = True,
14     transform = transform
15 )
16
17 train_loader = torch.utils.data.DataLoader(
18     dataset    = trainset,
19     batch_size = BATCH_SIZE,
20     shuffle    = True,
21 )
22 test_loader = torch.utils.data.DataLoader(
23     dataset    = testset,
24     batch_size = BATCH_SIZE,
25     shuffle    = True,
26 )
```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분

입력 `x` 는 [배치크기, 색, 높이, 넓이] 로 이루어져 있습니다. `x.size()` 를 해보면 `[64, 1, 28, 28]` 이라고 표시되는 것을 보실 수 있습니다. Fashion MNIST에서 이미지의 크기는 28 x 28, 색은 흑백으로 1 가지 입니다. 그러므로 입력 `x`의 총 특성값 갯수는 $28 \times 28 \times 1$, 즉 784개 입니다.

우리가 사용할 모델은 3개의 레이어를 가진 인공신경망 입니다.

```

1 class Net(nn.Module):
2     def __init__(self):
3         super(Net, self).__init__()
4         self.fc1 = nn.Linear(784, 256)
5         self.fc2 = nn.Linear(256, 128)
6         self.fc3 = nn.Linear(128, 10)
7
8     def forward(self, x):
9         x = x.view(-1, 784)
10        x = F.relu(self.fc1(x))
11        x = F.relu(self.fc2(x))
12        x = self.fc3(x)
13        return x

```

```

1 model = Net().to(DEVICE)
2 optimizer = optim.SGD(model.parameters(), lr=0.01)

```


4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분

```

1 def train(model, train_loader, optimizer):
2     model.train()
3     for batch_idx, (data, target) in enumerate(train_loader):
4         # 학습 데이터를 DEVICE의 메모리로 보냄
5         data, target = data.to(DEVICE), target.to(DEVICE)
6         optimizer.zero_grad()
7         output = model(data)
8         loss = F.cross_entropy(output, target)
9         loss.backward()
10        optimizer.step()

```

```

1 def evaluate(model, test_loader):
2     model.eval()
3     test_loss = 0
4     correct = 0
5     with torch.no_grad():
6         for data, target in test_loader:
7             data, target = data.to(DEVICE), target.to(DEVICE)
8             output = model(data)
9
10            # 모든 오차 더하기
11            test_loss += F.cross_entropy(output, target,
12                                       reduction='sum').item()
13
14            # 가장 큰 값을 가진 클래스가 모델의 예측입니다.
15            # 예측과 정답을 비교하여 일치할 경우 correct에 1을 더합니다.
16            pred = output.max(1, keepdim=True)[1]
17            correct += pred.eq(target.view_as(pred)).sum().item()
18
19    test_loss /= len(test_loader.dataset)
20    test_accuracy = 100. * correct / len(test_loader.dataset)
21    return test_loss, test_accuracy

```

4. 실습

Fashion MNIST

- 인공지능망으로 패션 아이템 구분
- 86.93%의 정확도를 보여준다.

```

1 for epoch in range(1, EPOCHS + 1):
2     train(model, train_loader, optimizer)
3     test_loss, test_accuracy = evaluate(model, test_loader)
4
5     print('[{}] Test Loss: {:.4f}, Accuracy: {:.2f}%'.format(
6         epoch, test_loss, test_accuracy))

```

```

[1] Test Loss: 0.8437, Accuracy: 69.50%
[2] Test Loss: 0.6743, Accuracy: 75.98%
[3] Test Loss: 0.5830, Accuracy: 79.28%
[4] Test Loss: 0.5413, Accuracy: 80.55%
[5] Test Loss: 0.5131, Accuracy: 81.77%
[6] Test Loss: 0.5202, Accuracy: 80.67%
[7] Test Loss: 0.4880, Accuracy: 82.71%
[8] Test Loss: 0.4806, Accuracy: 83.13%
[9] Test Loss: 0.4625, Accuracy: 83.45%
[10] Test Loss: 0.4596, Accuracy: 83.49%
[11] Test Loss: 0.4620, Accuracy: 83.78%
[12] Test Loss: 0.4398, Accuracy: 84.48%
[13] Test Loss: 0.4384, Accuracy: 84.58%
[14] Test Loss: 0.4306, Accuracy: 84.76%
[15] Test Loss: 0.4295, Accuracy: 84.92%

```

```

[16] Test Loss: 0.4197, Accuracy: 85.25%
[17] Test Loss: 0.4381, Accuracy: 84.70%
[18] Test Loss: 0.4056, Accuracy: 85.57%
[19] Test Loss: 0.4151, Accuracy: 85.27%
[20] Test Loss: 0.4023, Accuracy: 85.72%
[21] Test Loss: 0.4096, Accuracy: 85.52%
[22] Test Loss: 0.3959, Accuracy: 85.98%
[23] Test Loss: 0.3995, Accuracy: 85.99%
[24] Test Loss: 0.4055, Accuracy: 85.72%
[25] Test Loss: 0.3880, Accuracy: 86.21%
[26] Test Loss: 0.3828, Accuracy: 86.40%
[27] Test Loss: 0.4017, Accuracy: 85.58%
[28] Test Loss: 0.3839, Accuracy: 86.20%
[29] Test Loss: 0.3696, Accuracy: 86.93%
[30] Test Loss: 0.3715, Accuracy: 86.79%

```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 앞서 배운 정규화와 드롭아웃을 통해 정확도를 높여보자

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torch.nn.functional as F
5 from torchvision import transforms, datasets
```

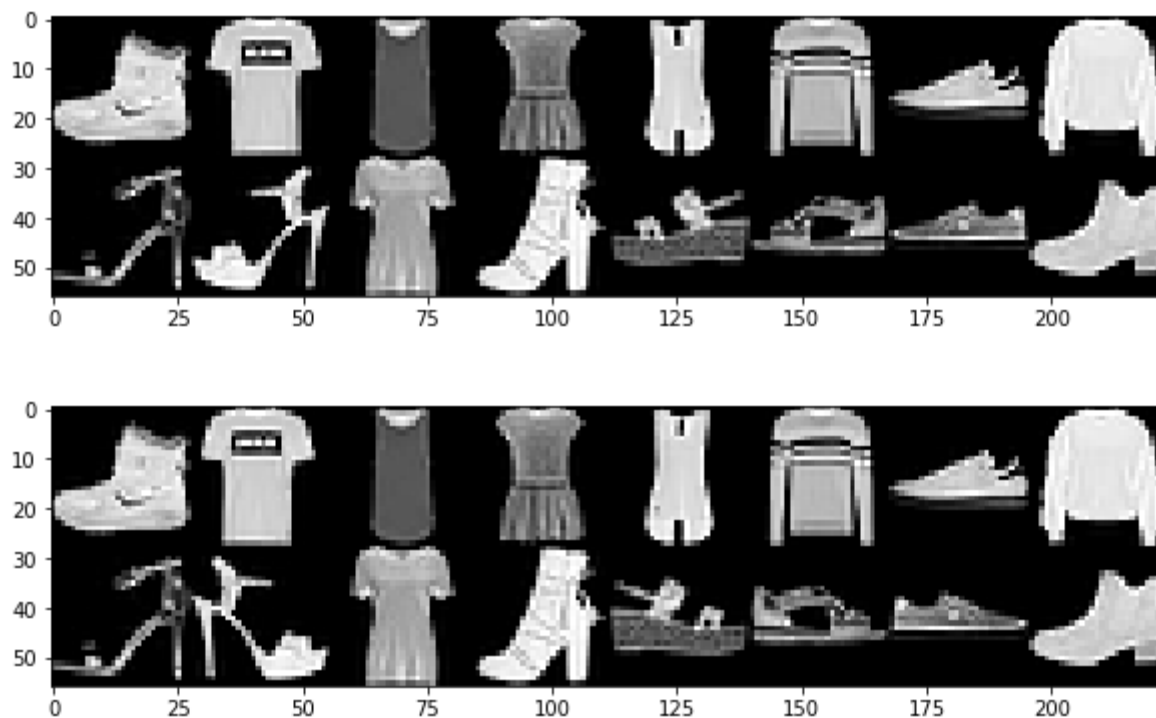
```
1 USE_CUDA = torch.cuda.is_available()
2 DEVICE = torch.device("cuda" if USE_CUDA else "cpu")
```

```
1 EPOCHS = 50
2 BATCH_SIZE = 64
```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 앞서 배운 정규화와 드롭아웃을 통해 정확도를 높여보자
- 데이터셋의 모양을 바꾸고 노이즈를 추가하여 데이터의 양을 증가시킨다.



4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 앞서 배운 정규화와 드롭아웃을 통해 정확도를 높여보자
- 데이터셋의 모양을 바꾸고 노이즈를 추가하여 데이터의 양을 증가시킨다.

```
1 train_loader = torch.utils.data.DataLoader(  
2     datasets.FashionMNIST('./.data',  
3         train=True,  
4         download=True,  
5         transform=transforms.Compose([  
6             transforms.RandomHorizontalFlip(),  
7             transforms.ToTensor(),  
8             transforms.Normalize((0.1307,), (0.3081,))  
9         ])),  
10    batch_size=BATCH_SIZE, shuffle=True)  
11 test_loader = torch.utils.data.DataLoader(  
12     datasets.FashionMNIST('./.data',  
13         train=False,  
14         transform=transforms.Compose([  
15             transforms.ToTensor(),  
16             transforms.Normalize((0.1307,), (0.3081,))  
17         ])),  
18    batch_size=BATCH_SIZE, shuffle=True)
```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 앞서 배운 정규화와 드롭아웃을 통해 정확도를 높여보자
- 모델에 드롭아웃 추가

```

1 class Net(nn.Module):
2     def __init__(self, dropout_p=0.2):
3         super(Net, self).__init__()
4         self.fc1 = nn.Linear(784, 256)
5         self.fc2 = nn.Linear(256, 128)
6         self.fc3 = nn.Linear(128, 10)
7         # 드롭아웃 확률
8         self.dropout_p = dropout_p
9
10    def forward(self, x):
11        x = x.view(-1, 784)
12        x = F.relu(self.fc1(x))
13        # 드롭아웃 추가
14        x = F.dropout(x, training=self.training,
15                      p=self.dropout_p)
16        x = F.relu(self.fc2(x))
17        # 드롭아웃 추가
18        x = F.dropout(x, training=self.training,
19                      p=self.dropout_p)
20        x = self.fc3(x)
21        return x

```

```

1 model = Net(dropout_p=0.2).to(DEVICE)
2 optimizer = optim.SGD(model.parameters(), lr=0.01)

```

4. 실습

Fashion MNIST

- 인공신경망으로 패션 아이템 구분
- 학습 함수, 평가 함수

```

1 def train(model, train_loader, optimizer):
2     model.train()
3     for batch_idx, (data, target) in enumerate(train_loader):
4         data, target = data.to(DEVICE), target.to(DEVICE)
5         optimizer.zero_grad()
6         output = model(data)
7         loss = F.cross_entropy(output, target)
8         loss.backward()
9         optimizer.step()

```

```

1 def evaluate(model, test_loader):
2     model.eval()
3     test_loss = 0
4     correct = 0
5     with torch.no_grad():
6         for data, target in test_loader:
7             data, target = data.to(DEVICE), target.to(DEVICE)
8             output = model(data)
9             test_loss += F.cross_entropy(output, target,
10                                         reduction='sum').item()
11
12             # 맞춘 갯수 계산
13             pred = output.max(1, keepdim=True)[1]
14             correct += pred.eq(target.view_as(pred)).sum().item()
15
16     test_loss /= len(test_loader.dataset)
17     test_accuracy = 100. * correct / len(test_loader.dataset)
18     return test_loss, test_accuracy

```

4. 실습

Fashion MNIST

- 인공지능망으로 패션 아이템 구분
- 88.98% 정확도를 보여준다. 2% 정확도 상승

```

1 for epoch in range(1, EPOCHS + 1):
2     train(model, train_loader, optimizer)
3     test_loss, test_accuracy = evaluate(model, test_loader)
4
5     print('{[{}]} Test Loss: {:.4f}, Accuracy: {:.2f}%'.format(
6         epoch, test_loss, test_accuracy))

```

```

[1] Test Loss: 0.6486, Accuracy: 77.37%
[2] Test Loss: 0.5417, Accuracy: 80.26%
[3] Test Loss: 0.4887, Accuracy: 82.27%
[4] Test Loss: 0.4639, Accuracy: 83.17%
[5] Test Loss: 0.4424, Accuracy: 84.02%
[6] Test Loss: 0.4253, Accuracy: 84.70%
[7] Test Loss: 0.4214, Accuracy: 84.63%
[8] Test Loss: 0.4077, Accuracy: 85.06%
[9] Test Loss: 0.4085, Accuracy: 84.99%
[10] Test Loss: 0.3914, Accuracy: 85.86%
[11] Test Loss: 0.3899, Accuracy: 85.90%
[12] Test Loss: 0.3774, Accuracy: 86.36%
[13] Test Loss: 0.3809, Accuracy: 86.33%
[14] Test Loss: 0.3718, Accuracy: 86.55%
[15] Test Loss: 0.3686, Accuracy: 86.57%
[16] Test Loss: 0.3628, Accuracy: 86.86%
[17] Test Loss: 0.3622, Accuracy: 86.67%
[18] Test Loss: 0.3585, Accuracy: 87.04%
[19] Test Loss: 0.3562, Accuracy: 86.94%
[20] Test Loss: 0.3496, Accuracy: 87.51%
[21] Test Loss: 0.3502, Accuracy: 87.31%
[22] Test Loss: 0.3437, Accuracy: 87.68%
[23] Test Loss: 0.3459, Accuracy: 87.26%
[24] Test Loss: 0.3544, Accuracy: 87.20%
[25] Test Loss: 0.3383, Accuracy: 87.98%

```

```

[26] Test Loss: 0.3387, Accuracy: 88.03%
[27] Test Loss: 0.3408, Accuracy: 87.90%
[28] Test Loss: 0.3329, Accuracy: 88.07%
[29] Test Loss: 0.3393, Accuracy: 87.80%
[30] Test Loss: 0.3296, Accuracy: 88.25%
[31] Test Loss: 0.3277, Accuracy: 88.31%
[32] Test Loss: 0.3305, Accuracy: 88.23%
[33] Test Loss: 0.3253, Accuracy: 88.37%
[34] Test Loss: 0.3269, Accuracy: 88.26%
[35] Test Loss: 0.3261, Accuracy: 88.30%
[36] Test Loss: 0.3194, Accuracy: 88.87%
[37] Test Loss: 0.3207, Accuracy: 88.75%
[38] Test Loss: 0.3186, Accuracy: 88.55%
[39] Test Loss: 0.3216, Accuracy: 88.41%
[40] Test Loss: 0.3164, Accuracy: 88.50%
[41] Test Loss: 0.3183, Accuracy: 88.45%
[42] Test Loss: 0.3139, Accuracy: 88.77%
[43] Test Loss: 0.3161, Accuracy: 88.66%
[44] Test Loss: 0.3112, Accuracy: 88.89%
[45] Test Loss: 0.3105, Accuracy: 89.05%
[46] Test Loss: 0.3138, Accuracy: 88.82%
[47] Test Loss: 0.3135, Accuracy: 88.73%
[48] Test Loss: 0.3128, Accuracy: 88.98%
[49] Test Loss: 0.3183, Accuracy: 88.52%
[50] Test Loss: 0.3213, Accuracy: 88.53%

```


4. 실습

Fashion MNIST

- CNN(Convolutional Neural Network)를 사용하여 패션아이템 구분 성능 높이기
- 필요한 라이브러리 임포트

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torch.nn.functional as F
5 from torchvision import transforms, datasets
6 USE_CUDA = torch.cuda.is_available()
7 DEVICE = torch.device("cuda" if USE_CUDA else "cpu")
8 EPOCHS = 40
9 BATCH_SIZE = 64
```

4. 실습

Fashion MNIST

- CNN(Convolutional Neural Network)를 사용하여 패션아이템 구분 성능 높이기
- 데이터 셋 불러오기

```
1 train_loader = torch.utils.data.DataLoader(  
2     datasets.MNIST('./.data',  
3         train=True,  
4         download=True,  
5         transform=transforms.Compose([  
6             transforms.ToTensor(),  
7             transforms.Normalize((0.1307,), (0.3081,))  
8         ])),  
9     batch_size=BATCH_SIZE, shuffle=True)  
10 test_loader = torch.utils.data.DataLoader(  
11     datasets.MNIST('./.data',  
12         train=False,  
13         transform=transforms.Compose([  
14             transforms.ToTensor(),  
15             transforms.Normalize((0.1307,), (0.3081,))  
16         ])),  
17     batch_size=BATCH_SIZE, shuffle=True)
```

4. 실습

Fashion MNIST

- CNN(Convolutional Neural Network)를 사용하여 패션아이템 구분 성능 높이기
- 모델 구축

```

1 class Net(nn.Module):
2     def __init__(self):
3         super(Net, self).__init__()
4         self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5         self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6         self.conv2_drop = nn.Dropout2d()
7         self.fc1 = nn.Linear(320, 50)
8         self.fc2 = nn.Linear(50, 10)
9
10    def forward(self, x):
11        x = F.relu(F.max_pool2d(self.conv1(x), 2))
12        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13        x = x.view(-1, 320)
14        x = F.relu(self.fc1(x))
15        x = F.dropout(x, training=self.training)
16        x = self.fc2(x)
17        return x

```

```

1 model = Net().to(DEVICE)
2 optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

```

4. 실습

Fashion MNIST

- CNN(Convolutional Neural Network)를 사용하여 패션아이템 구분 성능 높이기
- 학습 함수와 평가 함수

```

1 def train(model, train_loader, optimizer, epoch):
2     model.train()
3     for batch_idx, (data, target) in enumerate(train_loader):
4         data, target = data.to(DEVICE), target.to(DEVICE)
5         optimizer.zero_grad()
6         output = model(data)
7         loss = F.cross_entropy(output, target)
8         loss.backward()
9         optimizer.step()
10
11     if batch_idx % 200 == 0:
12         print('Train Epoch: {} [{}/{}] ({:.0f}%) #tLoss: {:.6f}'.format(
13             epoch, batch_idx * len(data), len(train_loader.dataset),
14             100. * batch_idx / len(train_loader), loss.item()))

```

```

1 def evaluate(model, test_loader):
2     model.eval()
3     test_loss = 0
4     correct = 0
5     with torch.no_grad():
6         for data, target in test_loader:
7             data, target = data.to(DEVICE), target.to(DEVICE)
8             output = model(data)
9
10            # 배치 오차를 합산
11            test_loss += F.cross_entropy(output, target,
12                                       reduction='sum').item()
13
14            # 가장 높은 값을 가진 인덱스가 바로 예측값
15            pred = output.max(1, keepdim=True)[1]
16            correct += pred.eq(target.view_as(pred)).sum().item()
17
18     test_loss /= len(test_loader.dataset)
19     test_accuracy = 100. * correct / len(test_loader.dataset)
20     return test_loss, test_accuracy

```

4. 실습

Fashion MNIST

- CNN(Convolutional Neural Network)를 사용하여 패션아이템 구분 성능 높이기
- 정확도가 99% 까지 나오는 것을 확인할 수 있다.

```

1 for epoch in range(1, EPOCHS + 1):
2     train(model, train_loader, optimizer, epoch)
3     test_loss, test_accuracy = evaluate(model, test_loader)
4
5     print('{:} Test Loss: {:.4f}, Accuracy: {:.2f}%'.format(
6         epoch, test_loss, test_accuracy))

```

```

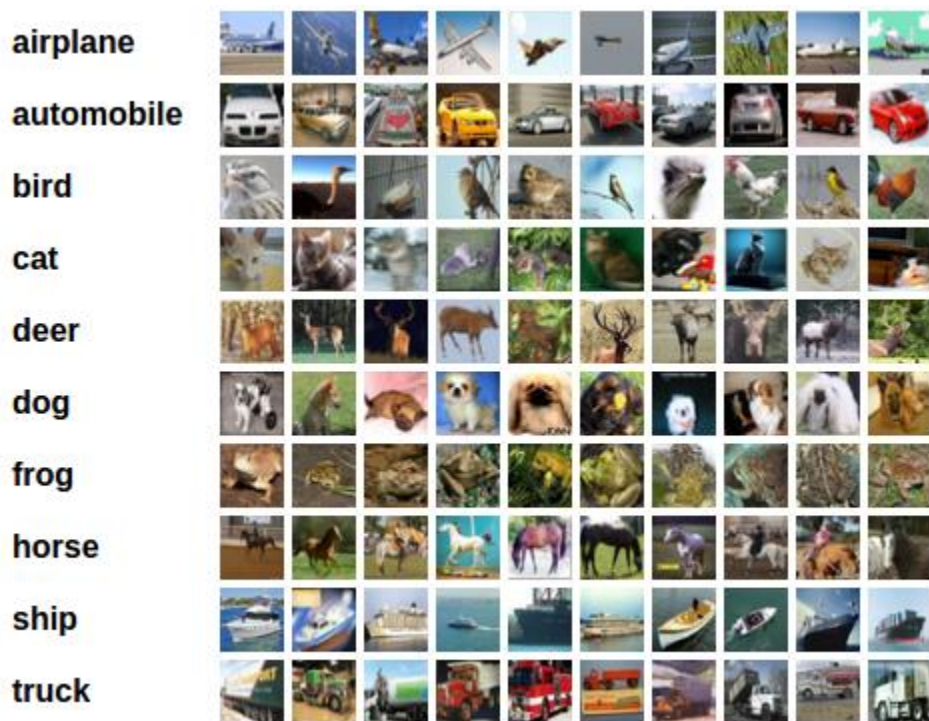
[37] Test Loss: 0.0310, Accuracy: 99.02%
Train Epoch: 38 [0/60000 (0%)] Loss: 0.227352
Train Epoch: 38 [12800/60000 (21%)] Loss: 0.249107
Train Epoch: 38 [25600/60000 (43%)] Loss: 0.250444
Train Epoch: 38 [38400/60000 (64%)] Loss: 0.073755
Train Epoch: 38 [51200/60000 (85%)] Loss: 0.227937
[38] Test Loss: 0.0276, Accuracy: 99.15%
Train Epoch: 39 [0/60000 (0%)] Loss: 0.194536
Train Epoch: 39 [12800/60000 (21%)] Loss: 0.051924
Train Epoch: 39 [25600/60000 (43%)] Loss: 0.095845
Train Epoch: 39 [38400/60000 (64%)] Loss: 0.036518
Train Epoch: 39 [51200/60000 (85%)] Loss: 0.172633
[39] Test Loss: 0.0292, Accuracy: 99.09%
Train Epoch: 40 [0/60000 (0%)] Loss: 0.076790
Train Epoch: 40 [12800/60000 (21%)] Loss: 0.155643
Train Epoch: 40 [25600/60000 (43%)] Loss: 0.040076
Train Epoch: 40 [38400/60000 (64%)] Loss: 0.114211
Train Epoch: 40 [51200/60000 (85%)] Loss: 0.232939
[40] Test Loss: 0.0283, Accuracy: 99.09%

```

4. 실습

이미지 분류

- CIFAR-10 데이터 셋 / R, G, B 3채널의 32*32 크기의 이미지 데이터
- 총 10가지의 클래스로 이루어진 데이터셋



4. 실습

이미지 분류

- 데이터 셋 불러오기와, 정규화

```

1 import torch
2 import torchvision
3 import torchvision.transforms as transforms
4
5 transform = transforms.Compose(
6     [transforms.ToTensor(),
7      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
8
9 #데이터 불러오기, 학습여부 o
10 trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
11                                         download=True, transform=transform)
12
13 #학습용 셋은 섞어서 뽑기
14 trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
15                                           shuffle=True, num_workers=2)
16
17 #데이터 불러오기, 학습여부 x
18 testset = torchvision.datasets.CIFAR10(root='./data', train=False,
19                                         download=True, transform=transform)
20
21 #테스트 셋은 굳이 섞을 필요가 없음
22 testloader = torch.utils.data.DataLoader(testset, batch_size=4,
23                                           shuffle=False, num_workers=2)
24
25 #클래스들
26 classes = ('plane', 'car', 'bird', 'cat',
27            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

```

Downloading <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz> to ./data#cifar-10-python.tar.gz

170500096it [00:25, 6725551.93it/s]

Extracting ./data#cifar-10-python.tar.gz to ./data
Files already downloaded and verified

4. 실습

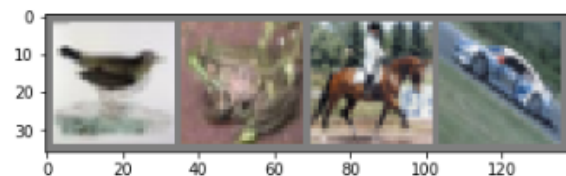
이미지 분류

- 이미지 데이터셋 확인

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 #이미지 확인하기
5
6 def imshow(img):
7     img = img / 2 + 0.5     # 정규화 해제
8     npimg = img.numpy()
9     plt.imshow(np.transpose(npimg, (1, 2, 0)))
10    plt.show()
11
12
13 # 학습용 이미지 뽑기
14 dataiter = iter(trainloader)
15 images, labels = dataiter.next()
16
17 # 이미지 보여주기
18 imshow(torchvision.utils.make_grid(images))
19
20 # 이미지별 라벨 (클래스) 보여주기
21 print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

```



bird frog horse car

4. 실습

이미지 분류

- CNN 정의하기 / Loss와 Optimizer 정의하기

```

1 import torch.nn as nn
2 import torch.nn.functional as F
3
4
5 class Net(nn.Module):
6     def __init__(self):
7         super(Net, self).__init__()
8
9         #input = 3, output = 8, kernel = 5
10        self.conv1 = nn.Conv2d(3, 6, 5)
11        #kernel = 2, stride = 2, padding = 0 (default)
12        self.pool = nn.MaxPool2d(2, 2)
13        self.conv2 = nn.Conv2d(6, 16, 5)
14        #input feature, output feature
15        self.fc1 = nn.Linear(16 * 5 * 5, 120)
16        self.fc2 = nn.Linear(120, 84)
17        self.fc3 = nn.Linear(84, 10)
18
19        # 과 계산
20        def forward(self, x):
21            x = self.pool(F.relu(self.conv1(x)))
22            x = self.pool(F.relu(self.conv2(x)))
23            x = x.view(-1, 16 * 5 * 5)
24            x = F.relu(self.fc1(x))
25            x = F.relu(self.fc2(x))
26            x = self.fc3(x)
27            return x
28
29
30 net = Net()

```

```

1 import torch.optim as optim
2
3 criterion = nn.CrossEntropyLoss()
4 optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

```

4. 실습

이미지 분류

- 학습하기

```

1  for epoch in range(2): #데이터셋 2번 받기
2
3      running_loss = 0.0
4      for i, data in enumerate(trainloader, 0):
5          # 입력 받기 (데이터 [입력, 라벨(정답)]으로 이루어짐)
6          inputs, labels = data
7
8          #학습
9          #optimizer의 기울기를 0으로 만들기 (변화도가 누적되지 않게 하기 위해)
10         optimizer.zero_grad()
11         # output 구하기
12         outputs = net(inputs)
13         # loss 계산
14         loss = criterion(outputs, labels)
15         #backpropagation (기울기 계산)
16         loss.backward()
17         #업데이트
18         optimizer.step()
19
20         # 결과 출력
21         running_loss += loss.item()
22         if i % 2000 == 1999: # print every 2000개 마다
23             print('[%d, %5d] loss: %.3f' %
24                   (epoch + 1, i + 1, running_loss / 2000))
25             running_loss = 0.0
26
27     print('Finished Training')
28
29     #여기에 학습한 모델 저장
30     PATH = './cifar_net.pth'
31     torch.save(net.state_dict(), PATH)

```

```

[1, 2000] loss: 2.163
[1, 4000] loss: 1.811
[1, 6000] loss: 1.673
[1, 8000] loss: 1.570
[1, 10000] loss: 1.496
[1, 12000] loss: 1.445
[2, 2000] loss: 1.381
[2, 4000] loss: 1.346
[2, 6000] loss: 1.322
[2, 8000] loss: 1.314
[2, 10000] loss: 1.280
[2, 12000] loss: 1.248

```

Finished Training

4. 실습

이미지 분류

- 테스트하기 / 4개의 이미지 중 3개를 맞춘 것을 확인할 수 있다.

```

1 dataiter = iter(testloader)
2 images, labels = dataiter.next()
3
4 # 실험용 데이터와 결과 출력
5 imshow(torchvision.utils.make_grid(images))
6 print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
7
8 # 학습한 모델로 예측과 뽑아보기
9 net = Net()
10 net.load_state_dict(torch.load(PATH))
11 outputs = net(images)
12 _, predicted = torch.max(outputs, 1)
13 print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
14                               for j in range(4)))

```



```

GroundTruth:   cat  ship  ship plane
Predicted:     cat  ship   car plane

```

4. 실습

이미지 분류

- 모델 평가
- 모델의 정확도는 56%이며,
- 클래스별 정확도는 밑의 코드와 같다.

```

1 correct = 0
2 total = 0
3 with torch.no_grad():
4     for data in testloader:
5         images, labels = data
6         outputs = net(images)
7         _, predicted = torch.max(outputs.data, 1)
8         total += labels.size(0)
9         correct += (predicted == labels).sum().item()
10
11 print('Accuracy of the network on the 10000 test images: %d %%' % (
12     100 * correct / total))

```

Accuracy of the network on the 10000 test images: 56 %

```

1 # 각 분류(class)에 대한 예측과 계산을 위해 준비
2 correct_pred = {classname: 0 for classname in classes}
3 total_pred = {classname: 0 for classname in classes}
4
5 # 변화도는 여전히 필요하지 않습니다
6 with torch.no_grad():
7     for data in testloader:
8         images, labels = data
9         outputs = net(images)
10        _, predictions = torch.max(outputs, 1)
11        # 각 분류별로 올바른 예측 수를 모읍니다
12        for label, prediction in zip(labels, predictions):
13            if label == prediction:
14                correct_pred[classes[label]] += 1
15                total_pred[classes[label]] += 1
16
17
18 # 각 분류별 정확도(accuracy)를 출력합니다
19 for classname, correct_count in correct_pred.items():
20     accuracy = 100 * float(correct_count) / total_pred[classname]
21     print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

```

```

Accuracy for class: plane is 71.3 %
Accuracy for class: car is 77.3 %
Accuracy for class: bird is 33.4 %
Accuracy for class: cat is 55.5 %
Accuracy for class: deer is 42.2 %
Accuracy for class: dog is 22.3 %
Accuracy for class: frog is 81.8 %
Accuracy for class: horse is 64.7 %
Accuracy for class: ship is 64.7 %
Accuracy for class: truck is 53.8 %

```

4. 실습

이미지 분류

- 모델 성능 향상 방법
- CNN 수정
- 조금 더 복잡한 CNN 모델 구축
- Dropout 기법 사용
- 학습 횟수 증가

```

1 class Net(nn.Module):
2     def __init__(self):
3         super(Net, self).__init__()
4         self.conv1 = nn.Conv2d(3, 32, 5)
5         self.conv2 = nn.Conv2d(32, 64, 5)
6         self.conv3 = nn.Conv2d(64, 128, 5)
7
8         self.pool = nn.MaxPool2d(2, 2, 1)
9
10        self.fc1 = nn.Linear(128 * 2 * 2, 256)
11        self.fc2 = nn.Linear(256, 128)
12        self.fc3 = nn.Linear(128, 84)
13        self.fc4 = nn.Linear(84, 10)
14
15        self.dropout1 = nn.Dropout(p=0.5, inplace=False)
16
17    def forward(self, x):
18        x = F.relu(self.conv1(x))
19        x = self.pool(x)
20
21        x = F.relu(self.conv2(x))
22        x = self.pool(x)
23
24        x = F.relu(self.conv3(x))
25        x = self.dropout1(x)
26        x = self.pool(x)
27
28        x = x.view(-1, 128 * 2 * 2)
29        x = F.relu(self.fc1(x))
30        x = self.dropout1(x)
31        x = F.relu(self.fc2(x))
32        x = F.relu(self.fc3(x))
33        x = self.fc4(x)
34        return x

```

4. 실습

이미지 분류

- 모델 성능 향상 방법
- CNN 수정
- 조금 더 복잡한 CNN 모델 구축
- Dropout 기법 사용
- 학습 횟수 증가

```

1  for epoch in range(20): #데이터셋 2번 받기
2
3      running_loss = 0.0
4      for i, data in enumerate(trainloader, 0):
5          # 입력 받기 (데이터 [입력, 라벨(정답)]으로 이루어짐)
6          inputs, labels = data
7
8          #학습
9          #optimizer의 기울기를 0으로 만들기 (변화도가 누적되지 않게 하기 위해)
10         optimizer.zero_grad()
11         # output 구하기
12         outputs = net(inputs)
13         # loss 계산
14         loss = criterion(outputs, labels)
15         #backpropagation (기울기 계산)
16         loss.backward()
17         #업데이트
18         optimizer.step()
19
20         # 결과 출력
21         running_loss += loss.item()
22         if i % 2000 == 1999: # print every 2000개 마다
23             print('[%d, %5d] loss: %.3f' %
24                   (epoch + 1, i + 1, running_loss / 2000))
25             running_loss = 0.0
26
27     print('Finished Training')
28
29     #여기에 학습한 모델 저장
30     PATH = './cifar_net.pth'
31     torch.save(net.state_dict(), PATH)

```

4. 실습

이미지 분류

- 정확도가 74% 까지 상승한 것을 확인할 수 있다.

```

1 correct = 0
2 total = 0
3 with torch.no_grad():
4     for data in testloader:
5         images, labels = data
6         outputs = net(images)
7         _, predicted = torch.max(outputs.data, 1)
8         total += labels.size(0)
9         correct += (predicted == labels).sum().item()
10
11 print('Accuracy of the network on the 10000 test images: %d %%' % (
12     100 * correct / total))

```

Accuracy of the network on the 10000 test images: 74 %

```

1 # 각 분류(class)에 대한 예측과 계산을 위해 준비
2 correct_pred = {classname: 0 for classname in classes}
3 total_pred = {classname: 0 for classname in classes}
4
5 # 변화도는 여전히 필요하지 않습니다
6 with torch.no_grad():
7     for data in testloader:
8         images, labels = data
9         outputs = net(images)
10        _, predictions = torch.max(outputs, 1)
11        # 각 분류별로 올바른 예측 수를 모읍니다
12        for label, prediction in zip(labels, predictions):
13            if label == prediction:
14                correct_pred[classes[label]] += 1
15                total_pred[classes[label]] += 1
16
17 # 각 분류별 정확도(accuracy)를 출력합니다
18 for classname, correct_count in correct_pred.items():
19     accuracy = 100 * float(correct_count) / total_pred[classname]
20     print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

```

```

Accuracy for class: plane is 77.3 %
Accuracy for class: car is 83.1 %
Accuracy for class: bird is 70.3 %
Accuracy for class: cat is 59.4 %
Accuracy for class: deer is 64.7 %
Accuracy for class: dog is 58.6 %
Accuracy for class: frog is 82.5 %
Accuracy for class: horse is 80.2 %
Accuracy for class: ship is 83.6 %
Accuracy for class: truck is 82.3 %

```

Q & A

Thank you