[2026 winter seminar] – 딥러닝 홀로서기



Pusan National University
Dept. of Electronic Engineering
Computer Vision & Signal Processing Lab.

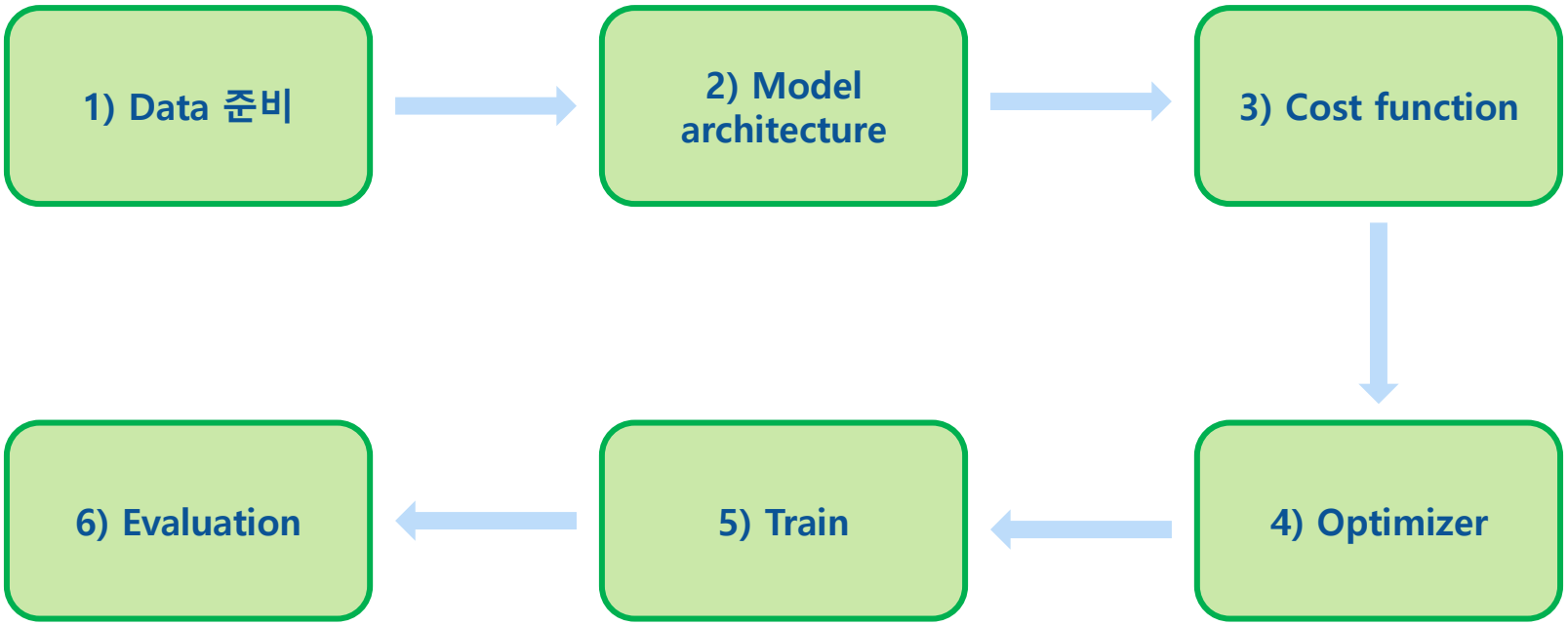


Presented By
전자공학전공 202324203 김우형

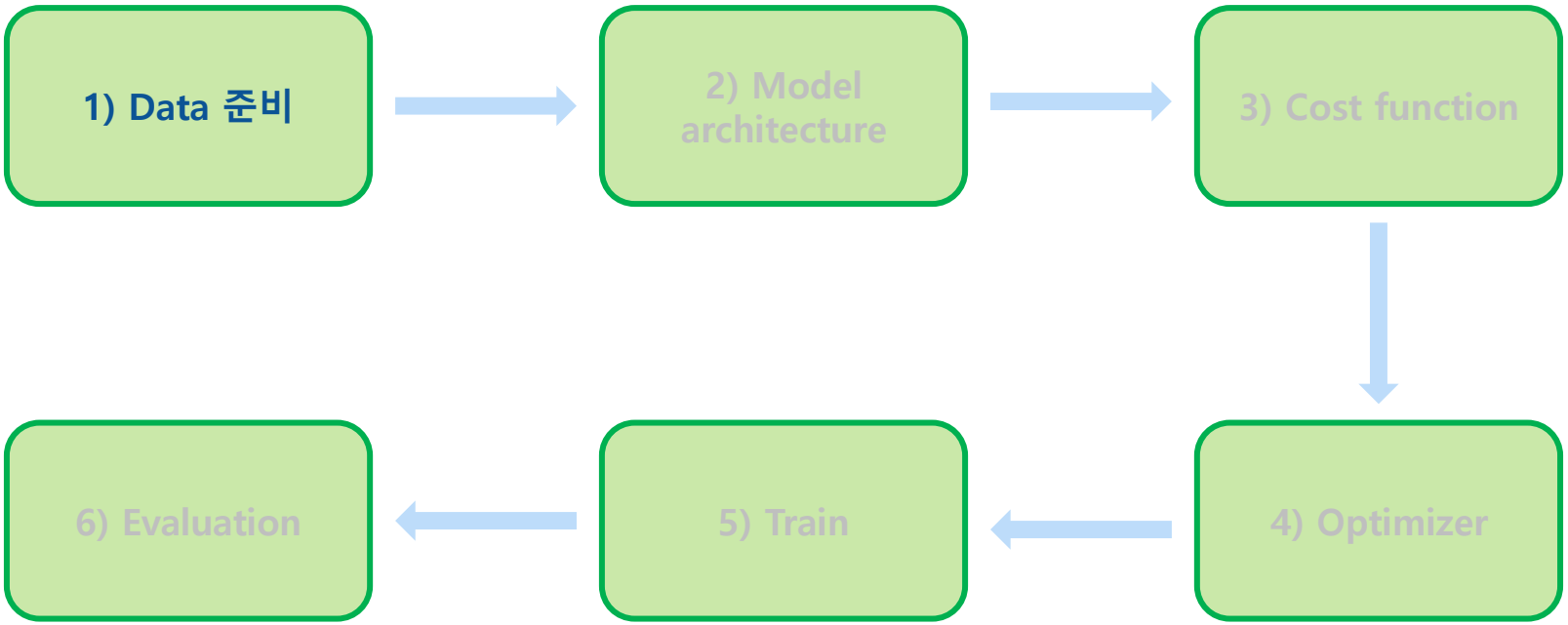
Outline

- **Deep learning Process**
  - **Data 준비**
  - **Model architecture**
  - **Cost Function**
  - **Optimizer**
  - **Train**
  - **Evaluation**
- **CIFAR 10 이미지 분류**
  - **MLP를 이용한 CIFAR 10 이미지 분류**
  - **CNN을 이용한 CIFAR 10 이미지 분류**
- **Attention Is All You Need**
  - **Introduction**
  - **Background**
  - **Model Architecture**

Deep Learning process

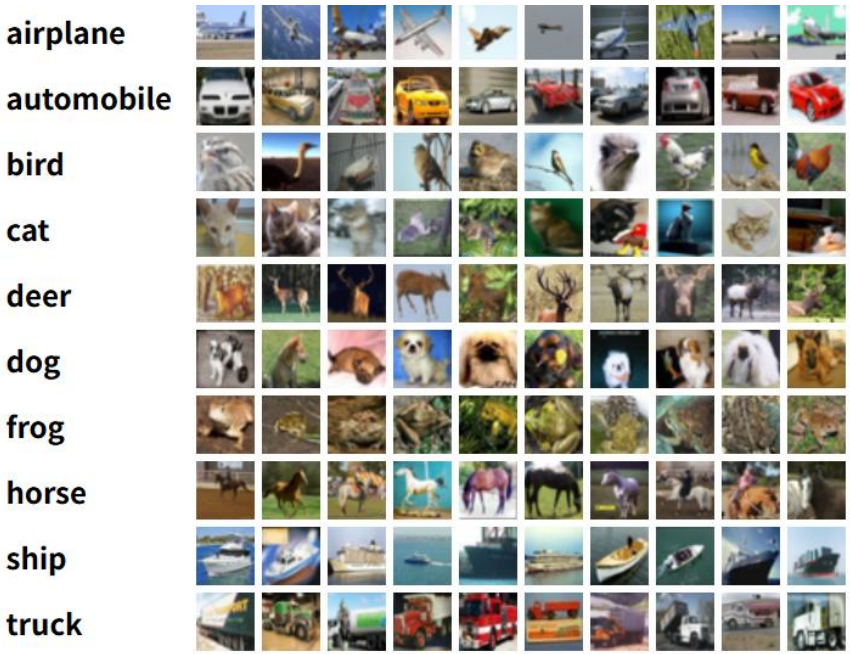


Deep Learning process



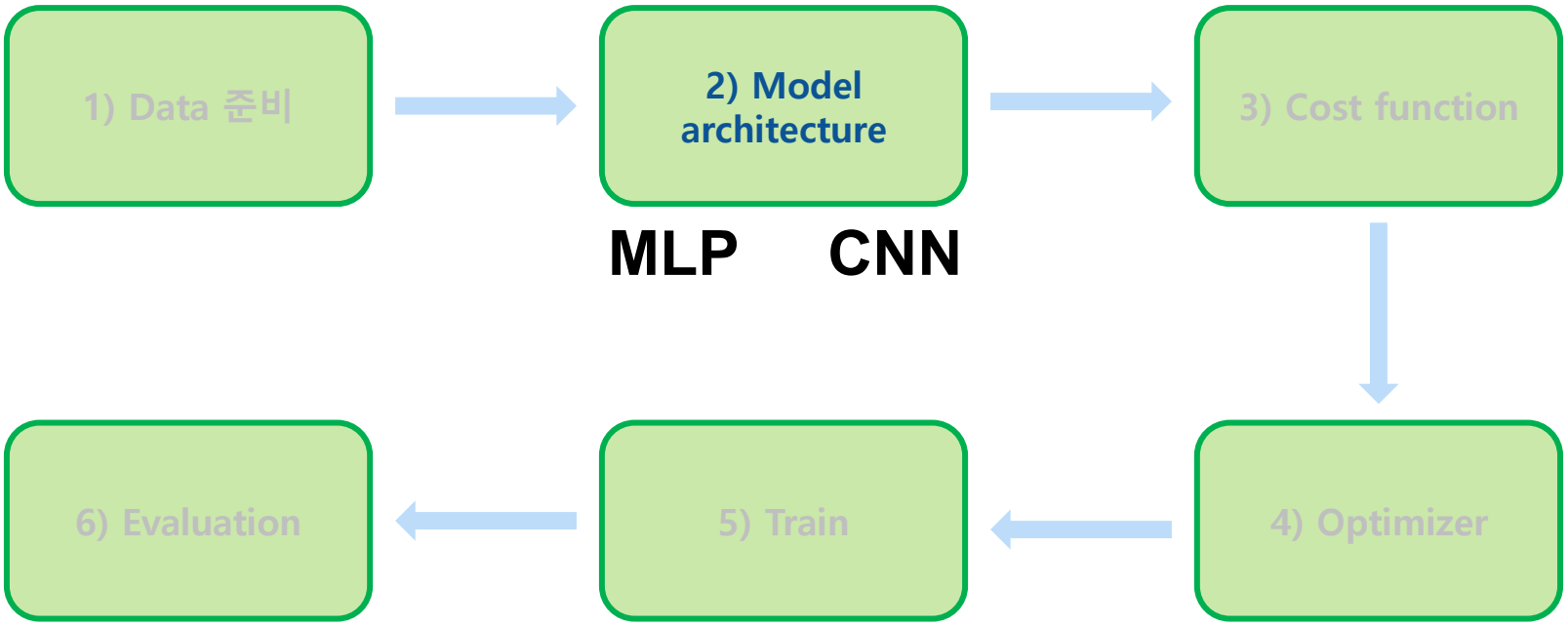
CIFAR 10

- 10가지 종류의 사물/동물이 담긴 작고 흐릿한 32 x 32 크기의 컬러 이미지 데이터셋
- 데이터 수 : 60,000장



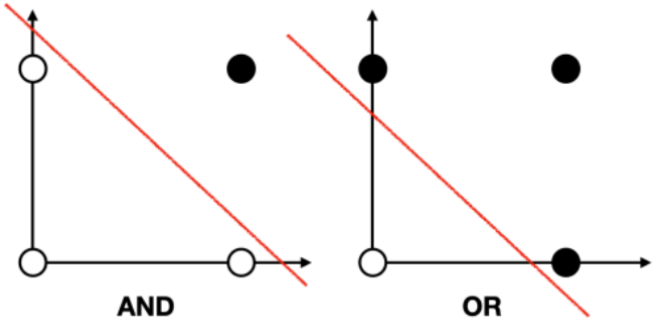
- **Train set**
  - 학습시키는 Dataset
  - 40,000장
- **Validation set**
  - Overfitting을 감지
  - 10,000장
- **Test set**
  - 한 번도 본적 없는 data로 성능 평가
  - 10,000장

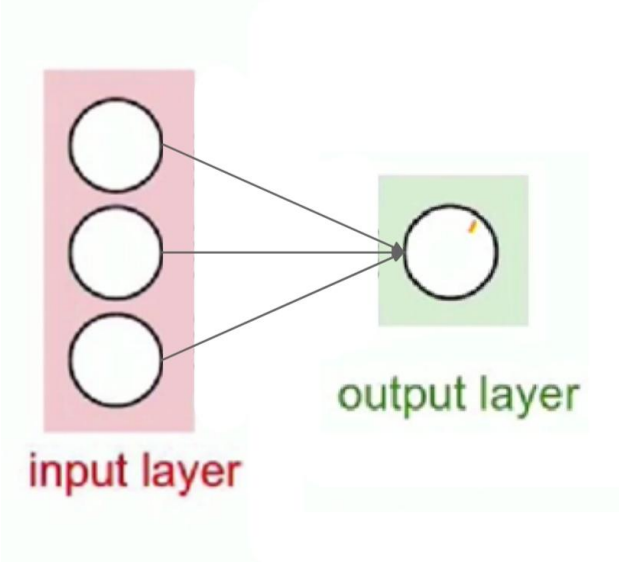
Deep Learning process

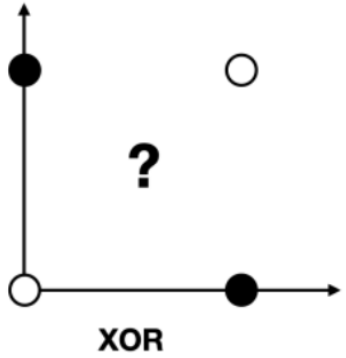


Single-Layer Perceptron

- 입력층과 출력층만 존재
- AND, OR과 같은 단순한 문제 해결 가능

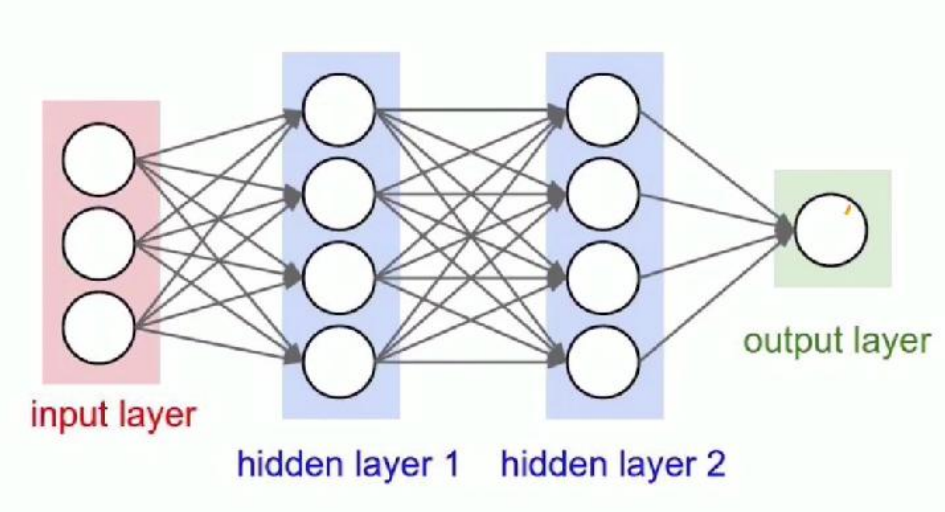


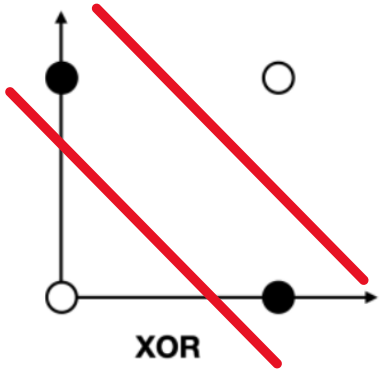




Multi-Layer Perceptron (MLP)

- 여러 개의 layer를 쌓아 올린 가장 기본적인 형태의 인공 신경망
- 입력층과 출력층 사이에 은닉층 존재
- 은닉층에서 Activation Function을 통과 시킴으로써 XOR과 같은 복잡한 문제 해결 가능



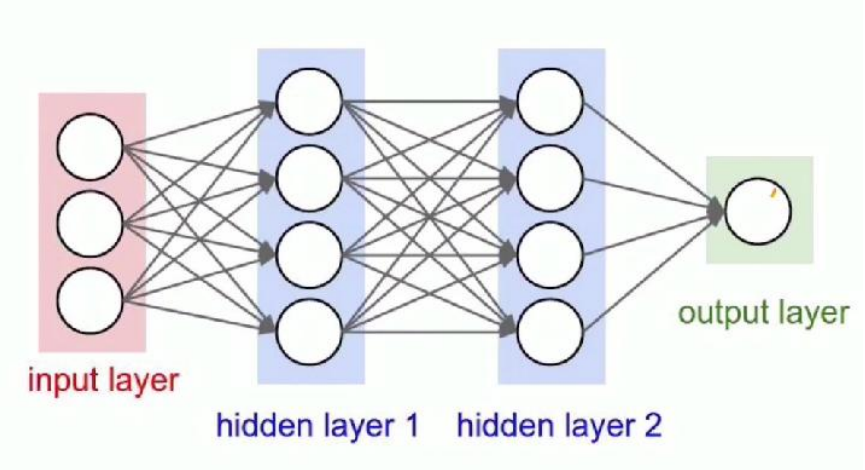


MLP의 한계 1)

- 보편 근사 정리
  - MLP의 hidden unit을 무한히 키우면 어떤 function이라도 approximation 가능

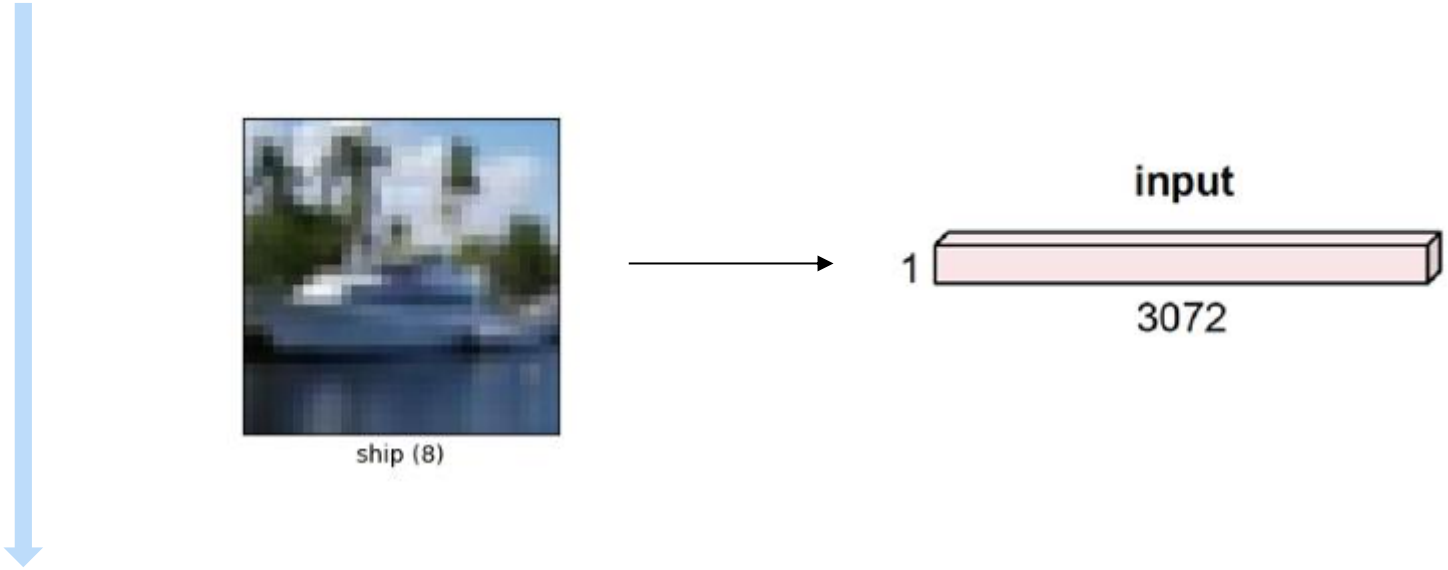
- MLP는 모든 픽셀 하나하나가 다음 층의 뉴런과 모두 연결 돼야함

- 파라미터 수 엄청나게 증가
  - ➔ Overfitting
  - ➔ 연산량 증가



MLP의 한계 2)

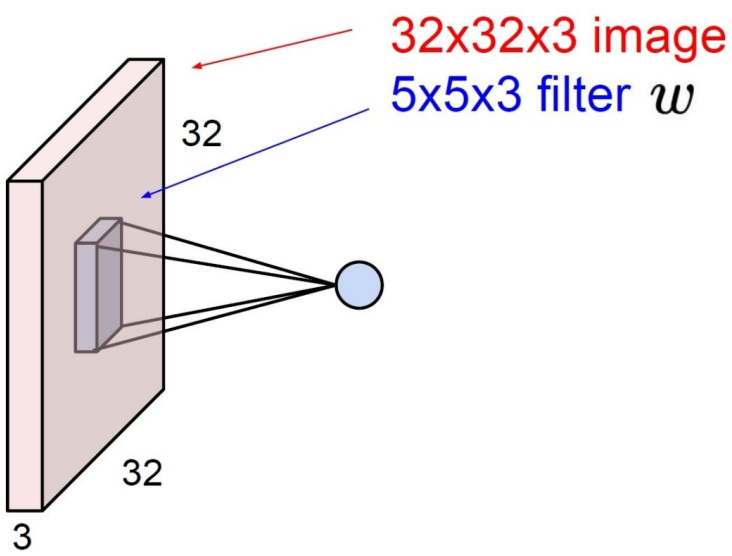
- Input 이미지를 긴 벡터로 펴버림

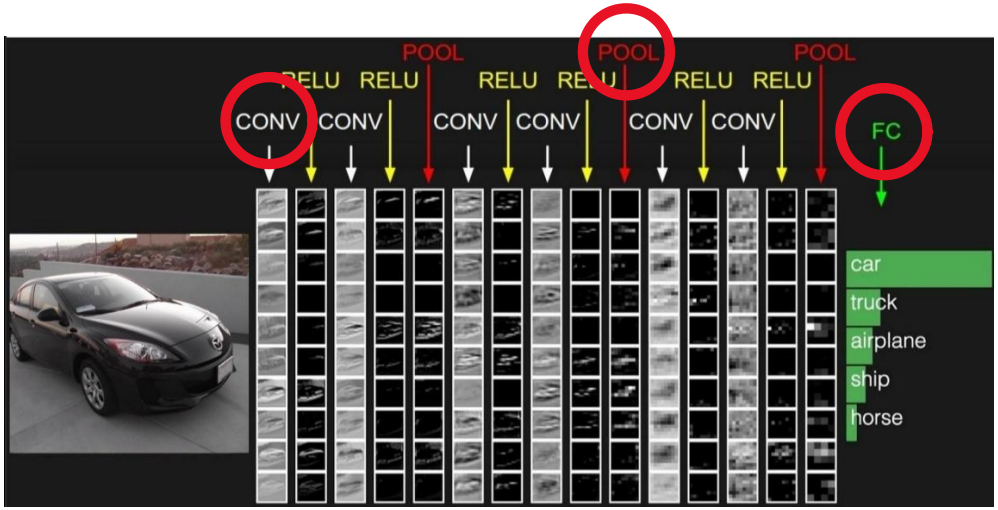


- 옆 픽셀과의 관계(공간적 구조)가 사라져 이미지 파악이 어려워짐

Convolution Neural Network (CNN)

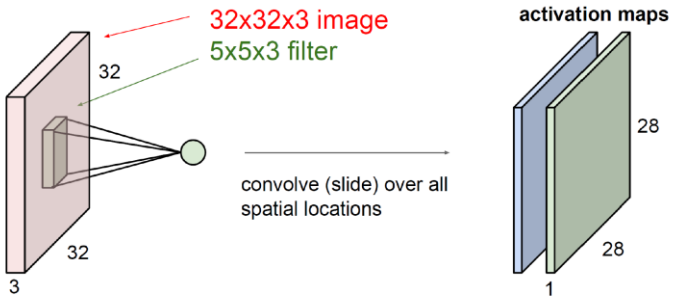
- 이미지와 필터를 합성곱하여 feature를 찾아내고, 이를 통해 최종 클래스를 분류하는 모델



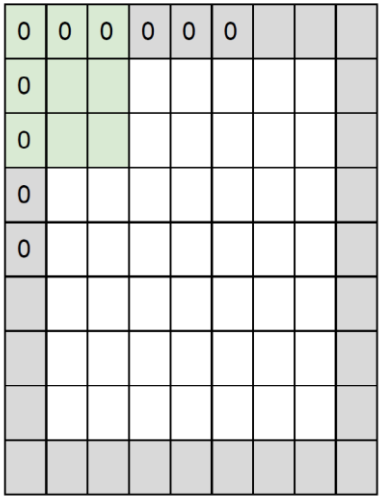


CNN 구조

- **Convolution layer**
  - **filter를 이미지에 convolution하여 Activation map (feature) 추출**

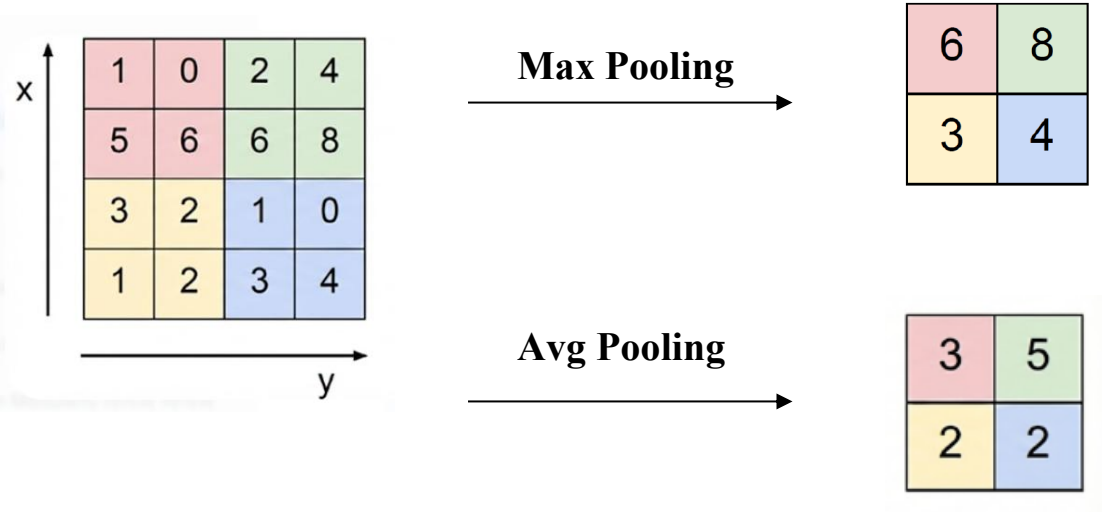


  - **padding : 이미지에 가짜 픽셀을 둘러주는 기법**
    - 이미지 크기 유지 -> layer를 깊게 쌓을 수 있음
    - 모서리까지 정보를 유지할 수 있음



CNN 구조

- **Pooling layer**
  - **Activation map의 공간을 줄여 의미 있는 정보만 남김**



CNN 구조

- **Fully Connected Layer (FC layer)**
  - 최종 Activation map을 한 줄로 펴서 정답을 도출
    - 공간 구조를 명확하게 보존하지 않아서, 위치 변화에 민감해지기 쉬움
    - MLP 특성상 parameter 수가 많음

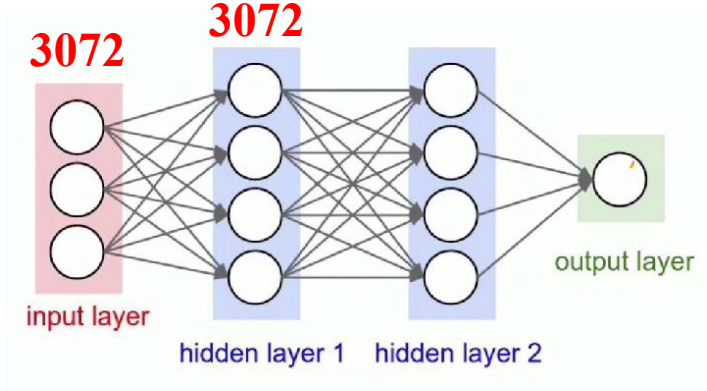
- **GAP (Global Average Pooling)**
  - 공간 정보를 평균 내어, 위치정보는 버리고 특징의 존재 여부만 남김
  - GoogleNet / ResNet

MLP의 한계 해결

- **MLP**
  - 다음 층을 만들기 위해 필요한 가중치 : 3072^2개

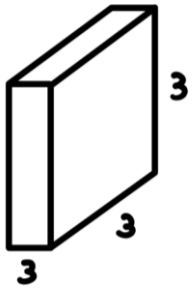
32 x 32 x 3



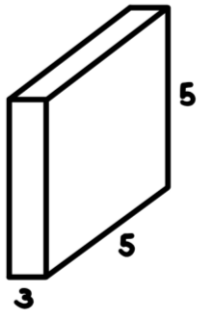


- **CNN**
  - 다음 층을 만들기 위해 필요한 가중치

3 x 3 x 3 = 27 개

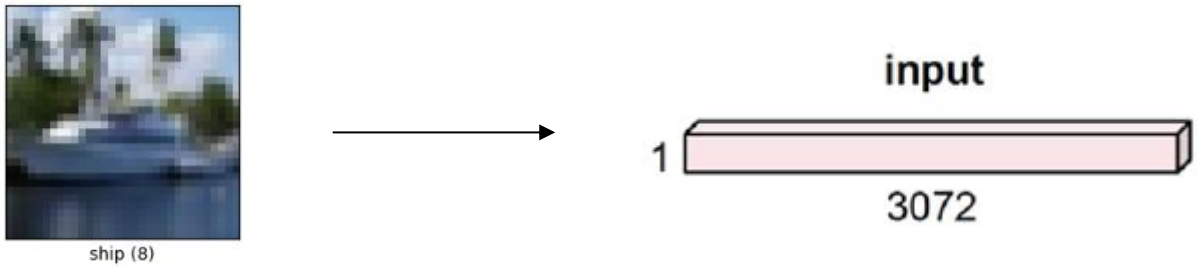


5 x 5 x 3 = 75 개

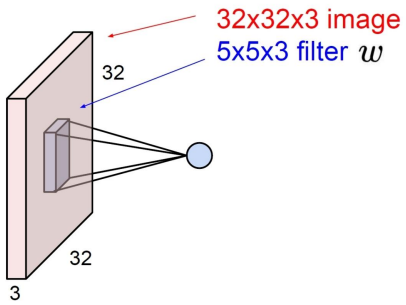


MLP의 한계 해결

- **MLP**
  - 이미지를 긴 벡터 형태로 받아 옆 픽셀과의 관계가 사라짐

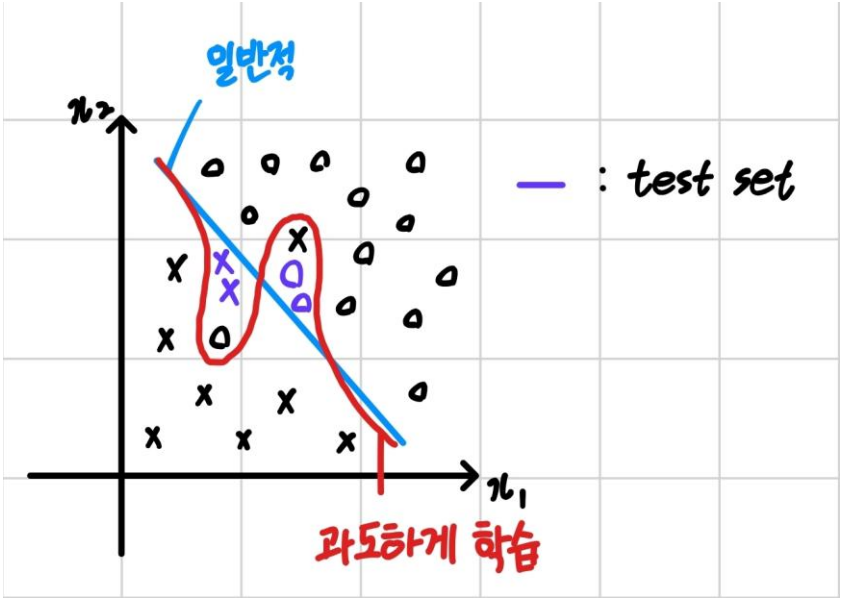


- **CNN**
  - 차원 변형 없이 입력 차원을 [batch, channel, height, width]로 받음
  - 이미지의 형태를 유지한 채 인접한 픽셀 간의 패턴을 파악하는 필터 사용



Deep Learning 학습의 문제(1)

- **Overfitting**
  - Model이 Trainset을 과하게 학습하여 Testset에서는 성능이 떨어지는 현상
  - Model capacity를 늘리면 더 복잡한 현상을 예측할 수 있지만 trainset만 잘 구분하고 test set에서는 성능을 못 낼 수 있음



Deep Learning 학습의 문제(1)

- **Overfitting 해결책**

1) L2 Regularization

- **Loss = Loss + R(λ)**
- **R(λ) = λ||ω|| : 패널티항**

➔ **Model은 loss를 줄이기 위해 가중치를 작게 만듬**

➔ **Model이 단순해지게 됨**

Deep Learning 학습의 문제(1)

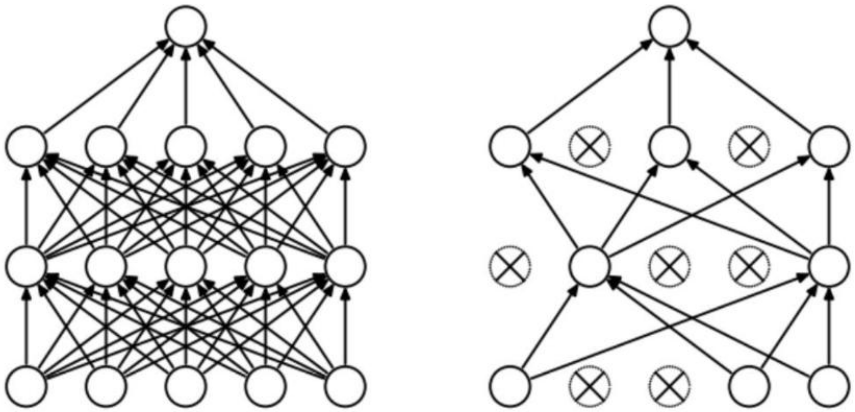
- **Overfitting 해결책**

2) Dropout

- **Node를 확률적으로 비활성화 시켜 일시적으로 model capacity를 줄일 수 있음**

  ➔ 특정 뉴런의 의존도를 낮춤

  ➔ 여러 model로 예측하고 결과를 합치는 앙상블 효과를 낼 수 있음



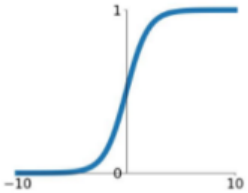
(a) Standard Neural Net (b) After applying dropout.

Deep Learning 학습의 문제(2)

- **Gradient Vanishing**
  - **Backpropagation 과정에서 입력으로 갈수록 gradient가 0에 수렴하는 현상**

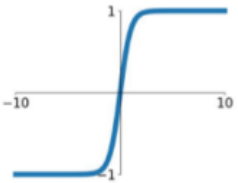
Sigmoid

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

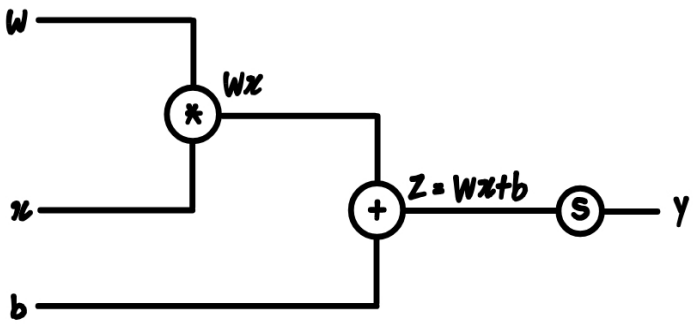


tanh

$$\tanh(x)$$



- **Backpropagation algorithm**



Cost function : y에 대한 함수

$$\frac{\partial cost}{\partial w} = \frac{\partial cost}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w}$$

Deep Learning 학습의 문제(2)

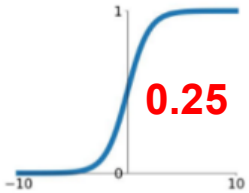
- **Gradient Vanishing 해결책**

1) Xavier Initialization

- gradient vanishing이 일어나지 않게 = wx+b가 0근처에 있도록 가중치 초기화

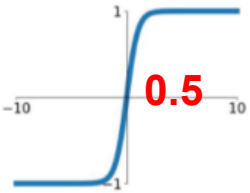
Sigmoid

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



tanh

$$\tanh(x)$$



2) Batch Normalization

- 입력 데이터의 분포를 평균 0, 분산 1로 강제 조정하여 학습을 빠르고 안정적으로 만드는 기법

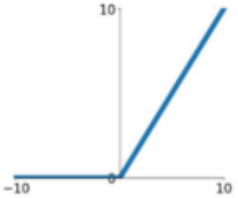
Deep Learning 학습의 문제

- **Gradient Vanishing 해결책**

3) 다른 Activation function 사용

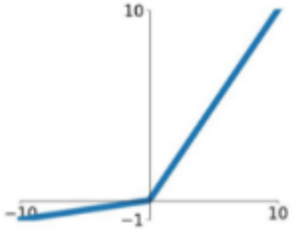
ReLU

$$\max(0, x)$$



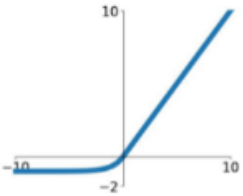
Leaky ReLU

$$\max(0.1x, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



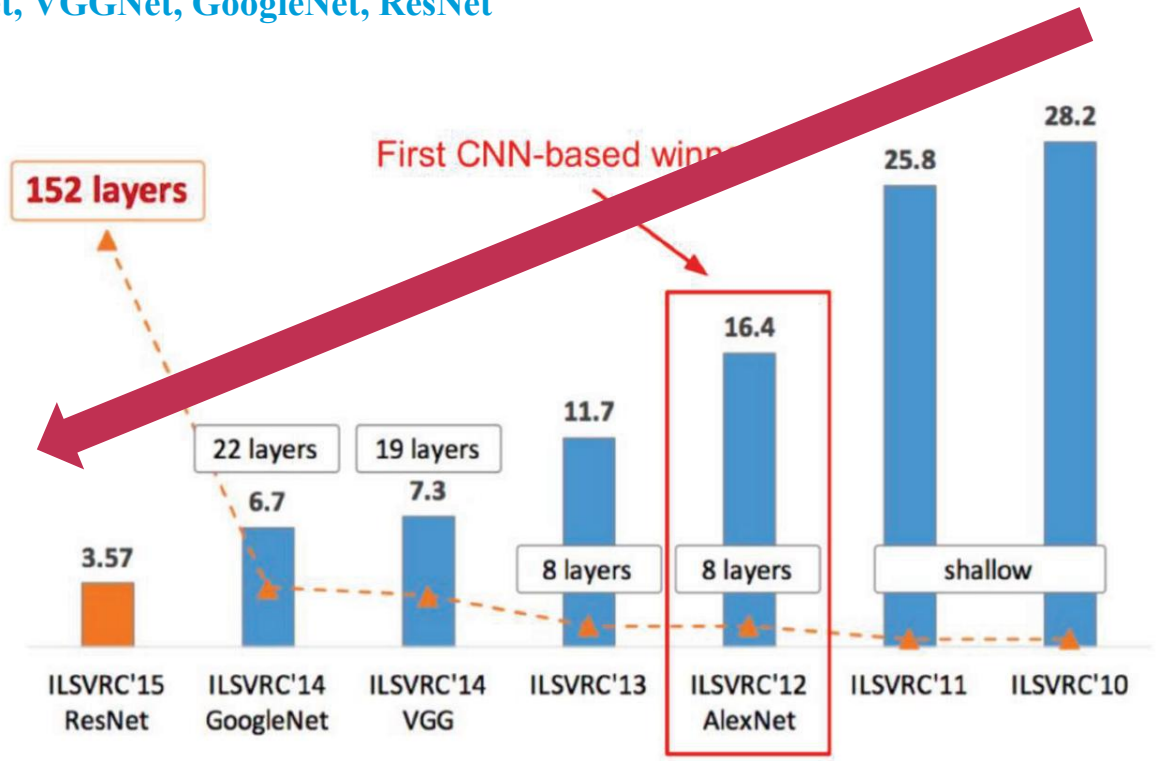
- **ReLU의 경우 Xavier Initialization을 사용하면 입력 값의 절반이 사라짐**

  ➔ 층이 깊어질수록 정보가 사라짐

  ➔ He Initialization : 죽은 절반을 보상하기 위해 초기 입력의 분산을 2배로 키움

CNN의 발전

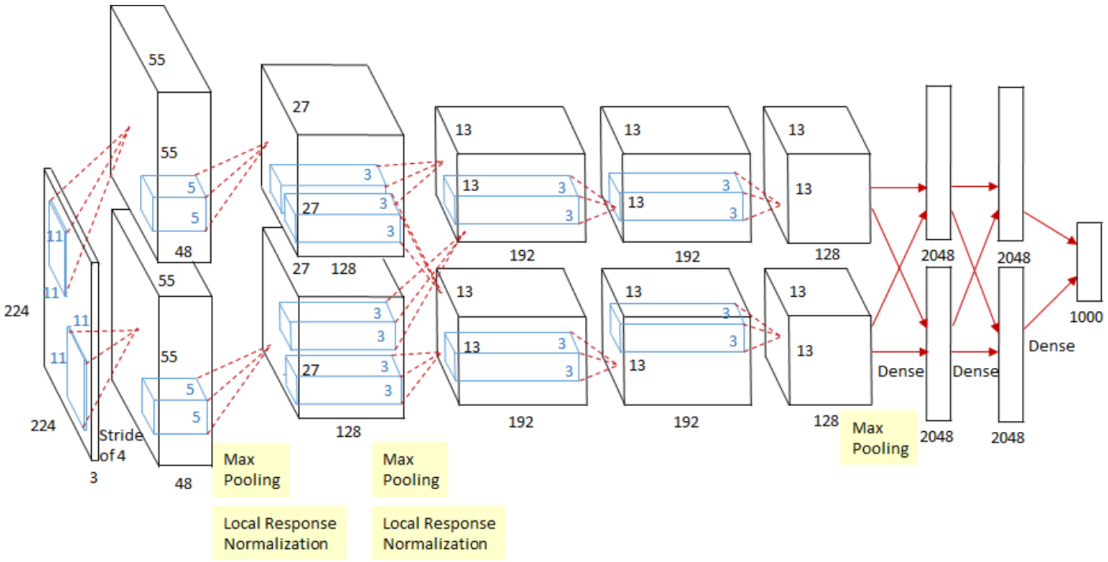
- **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**
  - CNN 성능을 발전시킨 모델들을 배출한 대회
  - 딥러닝 모델들이 발전한 과정
    - AlexNet, VGGNet, GoogleNet, ResNet



AlexNet

- LeNet-5
  - Backpropagation 처음 사용
  - Sigmoid / Tanh
  - CPU
    - 층을 깊게 쌓지 못함

- AlexNet
  - GPU 사용
  - 첫 ReLU 사용
    - Gradient vanishing 해결
  - 첫 Dropout 사용
  - Data Augmentation 사용
    - Overfitting 방지



➔ 층을 깊게 쌓을 수 있게 됨

VGGNet

- 모든 convolution layer에서 가장 작은 3 x 3 filter만 사용

  ➔ parameter 수 감소

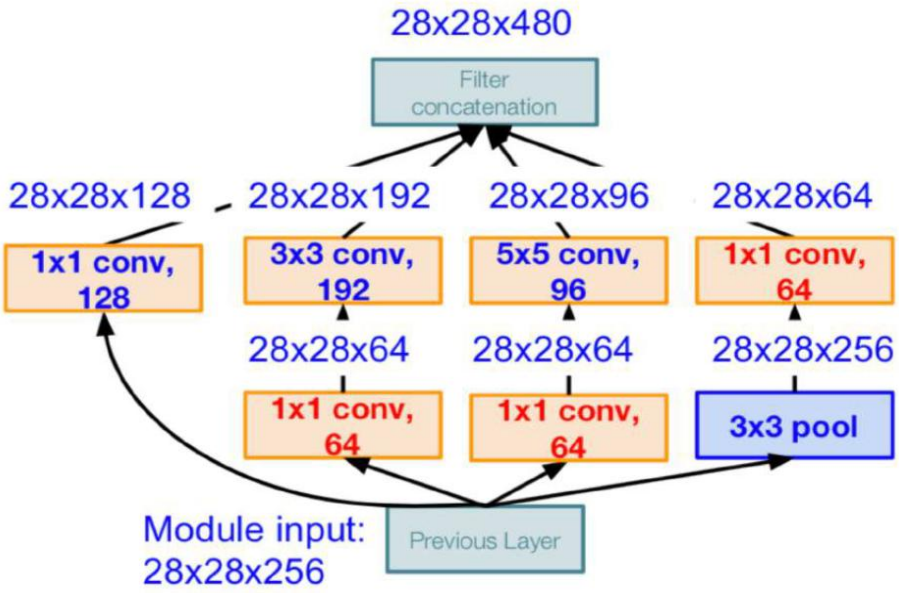
  ➔ 5x5, 7x7 filter와 비교하여 같은 Receptive Field 대비 더 많은 비선형 함수를 적용 가능

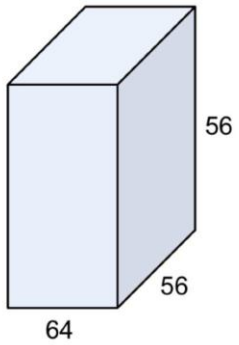
- FC layer에서는 activation map을 길게 늘려 파라미터 수가 크게 증가

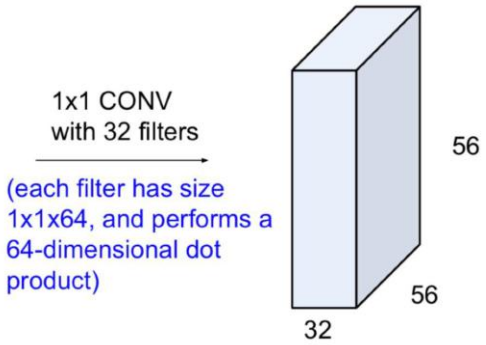


GoogleNet

- **Inception module 사용**
  - **한 layer에 여러 size의 filter를 씌움**
    - 작은 특징부터 큰 특징까지 한 번에 잡아낼 수 있음
  - **각 convolution의 padding의 두께를 다르게 하여 이미지크기는 유지**
  - **Bottleneck layer를 추가함으로써 정보는 유지하면서 연산량은 감소**

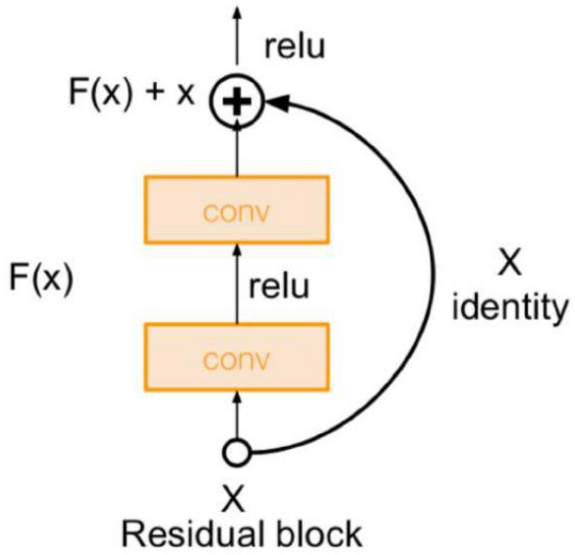


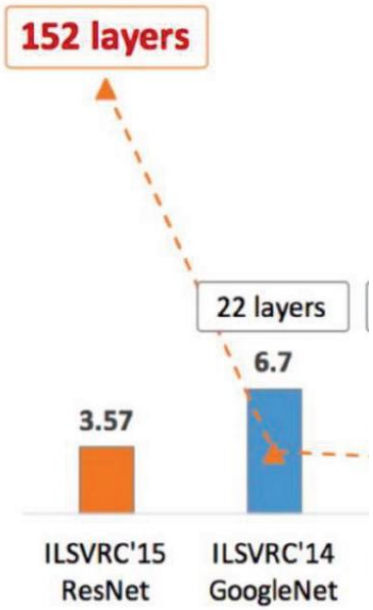




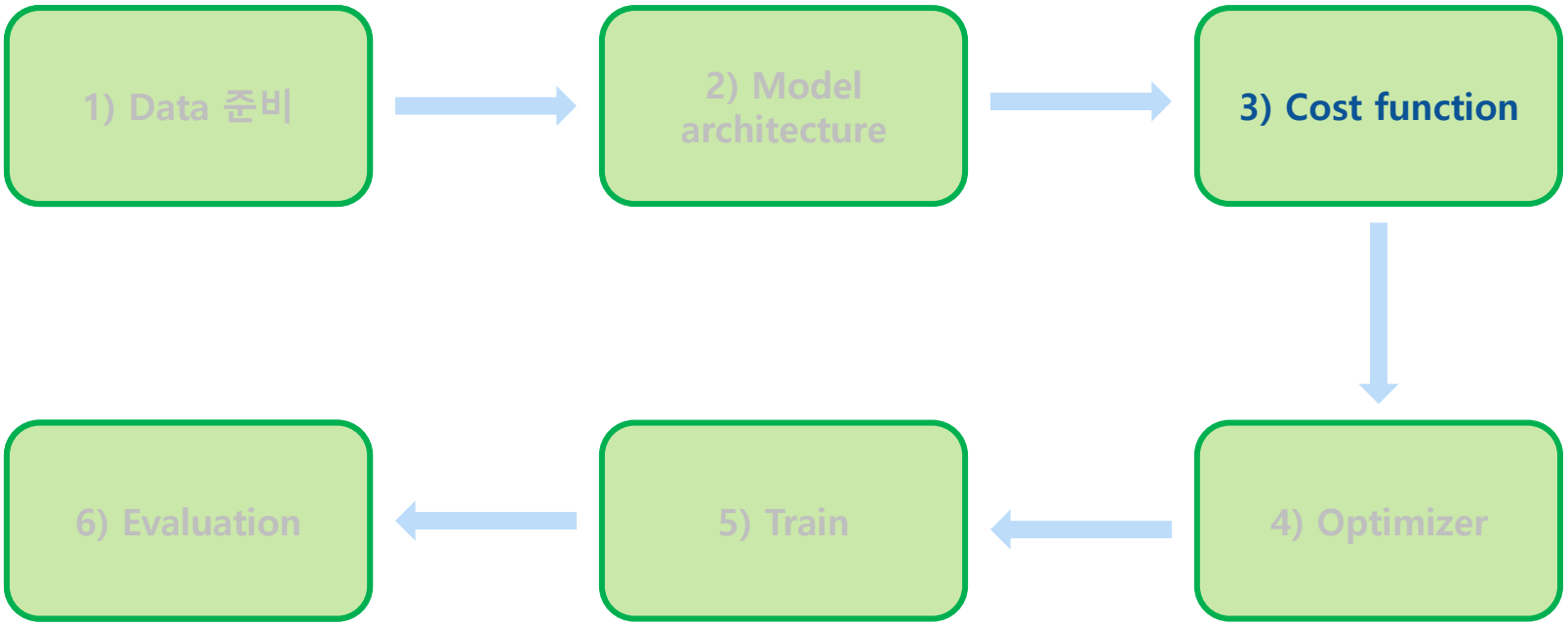
ResNet

- Gradient vanishing 때문에 layer를 더 깊게 쌓을 수 없었음
  - Skip Connection : 입력 x를 뒷 층에 더하기
- GoogleNet의 병렬구조를 버리고 직렬구조 선택
  - 단순 구조





Deep Learning process



3) Cost function

- 모델의 예측값과 실제값 사이의 차이를 나타내는 지표
- 학습의 방향을 결정하는 절대적인 기준

- **MSE**
  - **Regression 문제에서 주로 사용**
  - **오차** $H(x) - y$
  - **제곱 오차** $(H(x) - y)^2$
  - **평균 제곱 오차** $\frac{1}{n}\sum_{i=1}^{n}(H(x_i) - y_i)^2$

3) Cost function

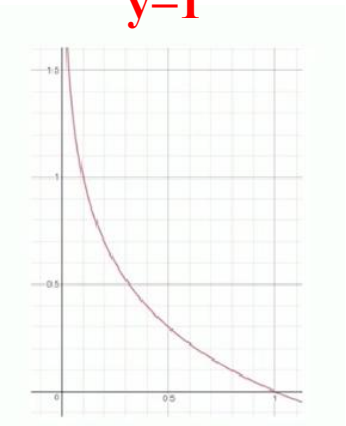
- **Binary Cross Entropy (BCE)**

  - **class가 2개**

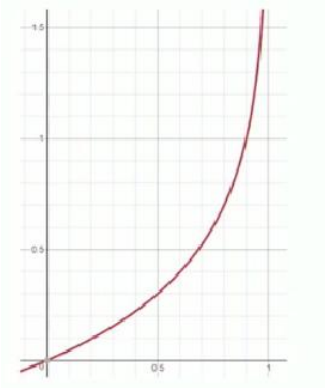
$$c(\hat{y}, y) = \begin{cases} -log(\hat{y}) & : y=1 \\ -log(1-\hat{y}) & : y=0 \end{cases} = -ylog\hat{y} + (1-y)\,log(1-\hat{y})$$

  - y : 실제값
  - $\hat{y}$: 모델의 예측 값 (sigmoid를 통과해 0~1 사이의 값)

y=1



y=0



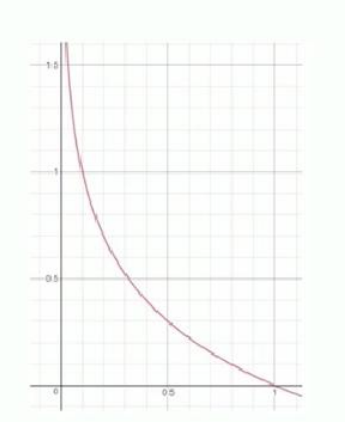
3) Cost function

- **Cross Entropy**

  - **class가 3개 이상**

$$c(\hat{y}, y) = \sum -y\log(\hat{y})$$

  - y : 실제값
  - $\hat{y}$: 모델의 예측 값
  - $\hat{y}_i$: Logit = 확률로 변하기 전의 class 각각의 예측 값

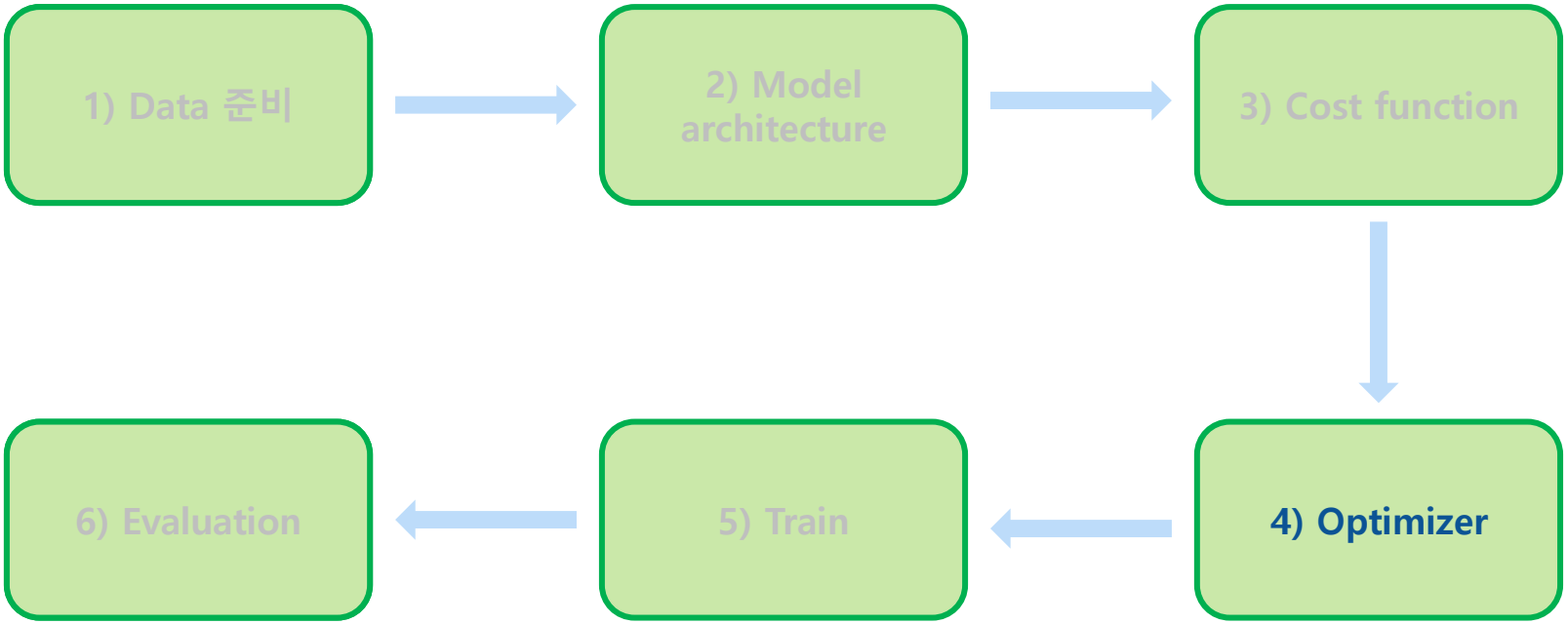


$$Softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}}$$

	강아지	고양이	쥐
실제값	[1, 0, 0]	[0, 1, 0]	[0, 0, 1]

예측값 `[0.1, 0.7, 0.2]`

Deep Learning process



4) Optimizer

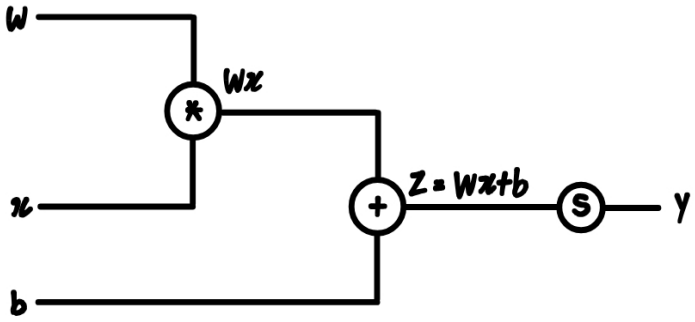
- **Optimizer**
  - Loss를 최소화하는 방향으로 가중치를 수정하는 알고리즘

- **Gradient Descent** (Optimizer.step())
  - global minimum으로 가기 위해 가중치를 수정하는 과정

$$\theta = \theta - \eta \nabla J(\theta)$$

- θ : weight
- η : learning rate
- J(θ) : Loss function

- **Back Propagation** (loss.backward())
  - Loss 의 기울기(Gradient) 구하는 도구



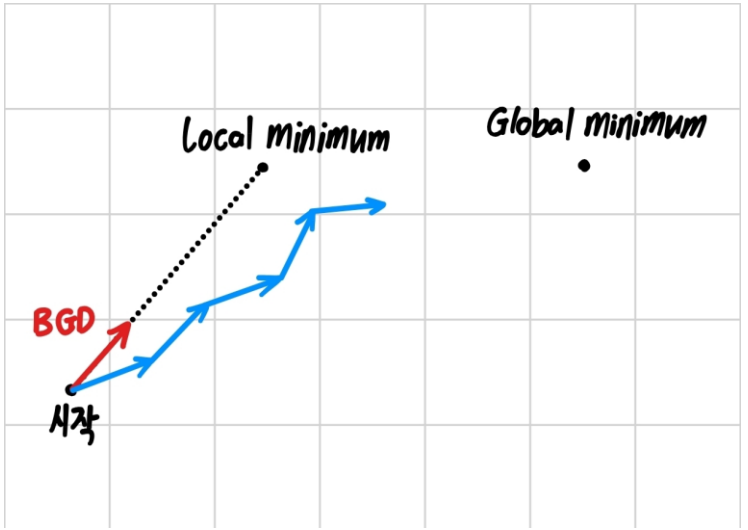
Cost function : y에 대한 함수

$$\frac{\partial cost}{\partial w} = \frac{\partial cost}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w}$$

4) Optimizer

- **Batch Gradient Descent (BGD)**
  - 모든 train set에 대한 gradient 계산한 뒤 평균을 내어 가중치 수정

- **Stochastic Gradient Descent (SGD)**
  - mini-batch에 대해서만 gradient 계산한 뒤 평균을 내어 가중치 수정
    - ➔ 1step에 대한 계산 속도가 빨라 학습 속도가
    - ➔ RAM 용량이 여유로워지게 됨



4) Optimizer

- **SGD의 한계 1)**

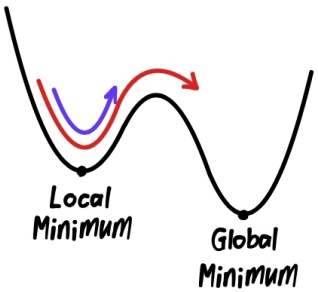
$$\theta = \theta - \eta \nabla J(\theta)$$

  - Minimum에 도달하지 못하고 주변을 서성거림
  - Local Minimum을 탈출하지 못하는 한계 존재

- **Momentum**

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

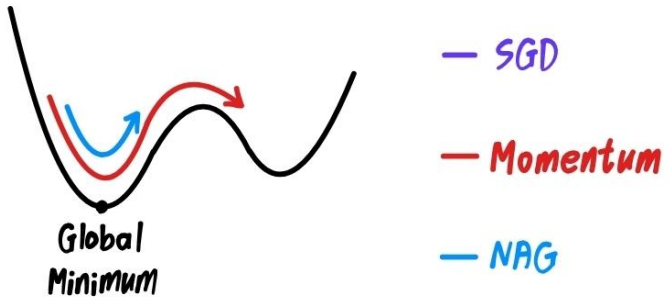
  - v_{t-1} : 직전에 움직이던 속도
  - Υ : 0~1사이의 decay(감쇠)상수



- **NAG**

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

  - 관성만큼 미리 가본 뒤 global minimum을 지나가는 것을 막아줌



4) Optimizer

- SGD의 한계 2)

$$\theta = \theta - \eta \nabla J(\theta)$$

  - gradient큰 parameter는 learning rate를 작게
    gradient가 작은 parameter는 learning rate 크게 적용해야 함

- Adagrad

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

$$G_t = G_{t-1} + (\nabla_\theta J(\theta_t))^2$$

  - G_t : gradient의 크기를 축적한 값
  - ε : 0이 되지 않기 위한 상수
    - G_t가 계속 축적되면 lr이 0에 수렴

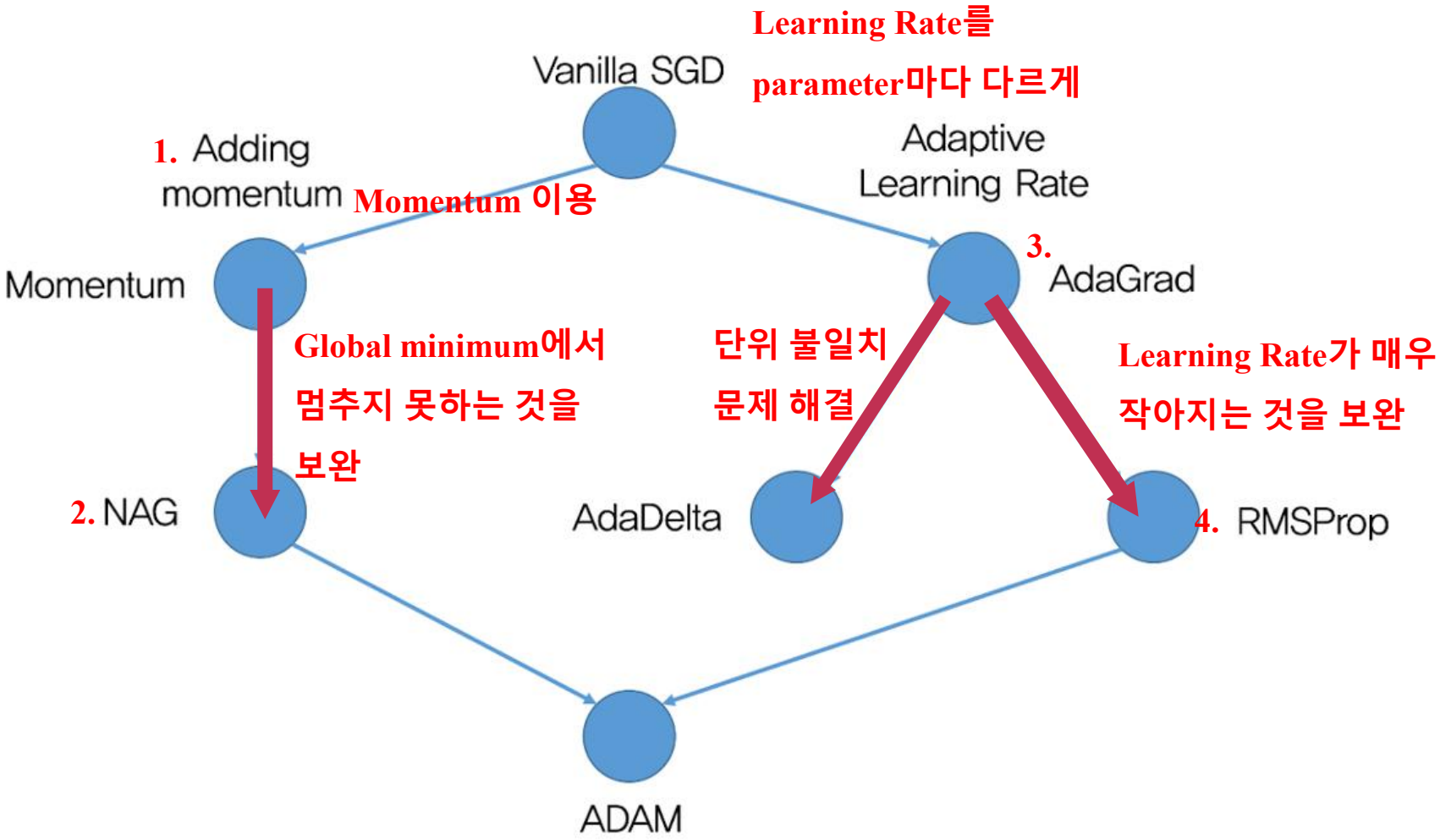
- RMSprop

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

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

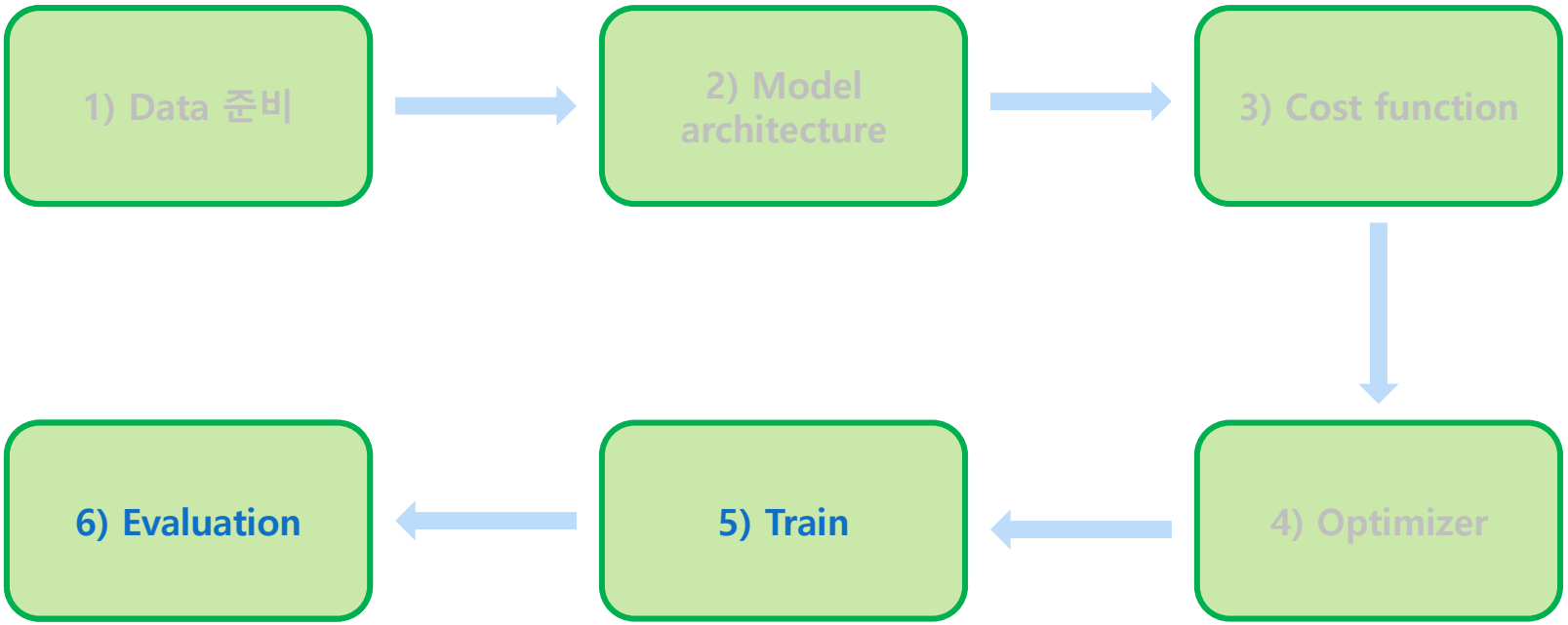
  - ϒ : 이전 gradient에서 남길 부분을 정하는 상수
    - G_t의 크기 유지 가능

4) Optimizer



NAG 장점 + RMSProp 장점

Deep Learning process



Train & Evaluation

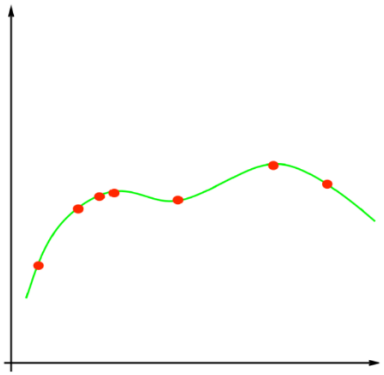
- Train set : Model이 실제로 학습을 하는데 사용하는 Data

- Validation set : 학습 방향을 조정하는데 사용하는 Data

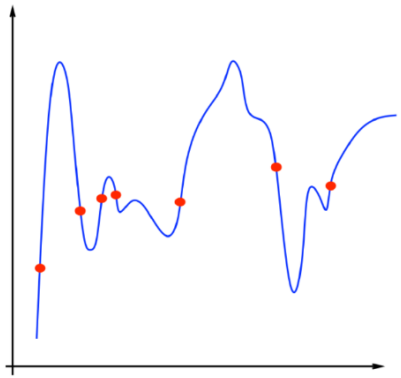
  - Overfitting 방지

- Test set : 최종 모델의 성능을 평가하기 위해 사용하는 Data

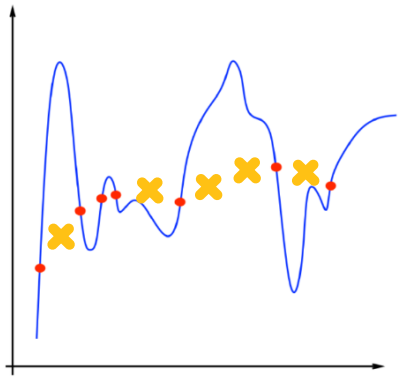
  - 한 번도 사용되지 않은 Dataset



<실제 Dataset>



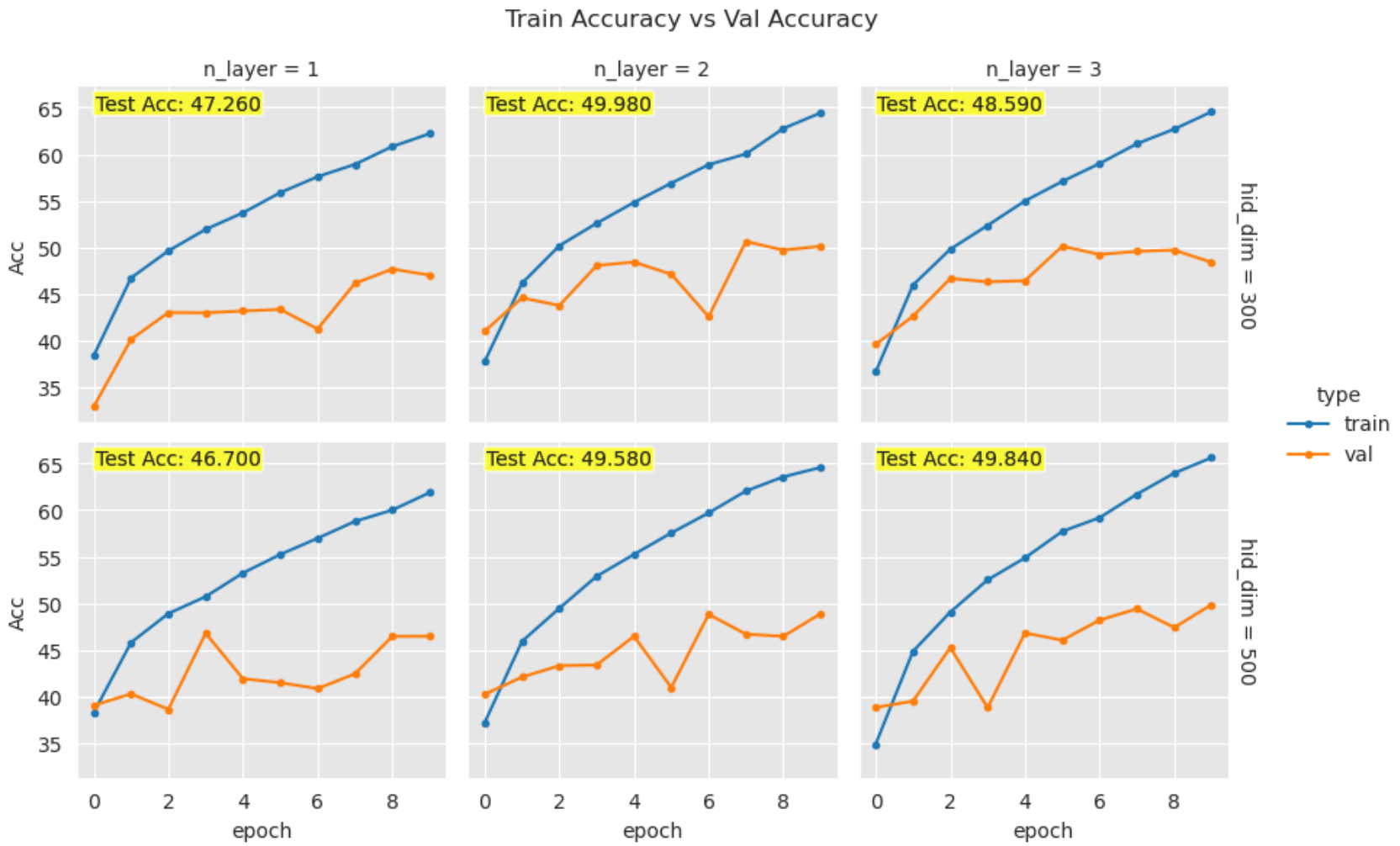
<Overfitting 된 경우>



<Validation set 도입>

CIFAR 10 with MLP

Initial value

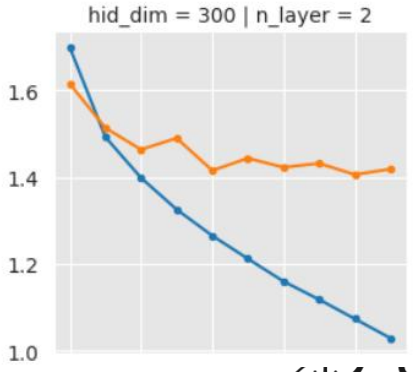


Experiment 1

- **Activation Function (dropout = 0.2)**

Act / n_layer / hid_dim	Sigmoid	Tanh	relu	Elu	Leaky_relu	Gelu
1 / 300	43.990	39.450	47.260	44.950	44.590	46.450
1 / 500	44.660	42.510	46.700	46.670	46.480	45.580
2/ 300	46.600	44.980	49.980	48.880	48.010	49.010
2 / 500	46.370	45.370	49.580	50.050	49.510	46.720
3 / 300	46.980	47.190	48.590	49.900	50.840	51.620
3 / 500	45.370	46.840	49.840	50.550	50.620	50.370

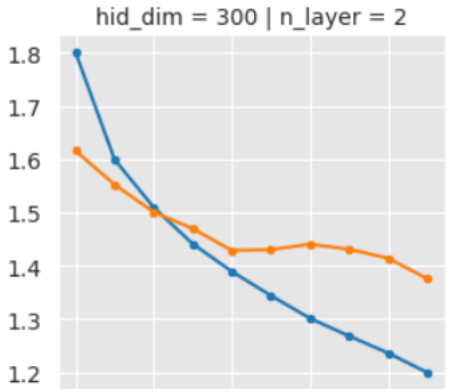
- **Overfitting 크게 발생해 dropout을 수정 (0.2 -> 0.5)**



Experiment 1

- **Activation Function (dropout = 0.5)**

n_layer / hid_dim	Sigmoid	Tanh	relu	Elu	Leaky_relu	Gelu
1 / 300	43.940	46.260	51.000	48.750	51.020	46.450
1 / 500	44.910	46.980	51.450	49.290	51.280	45.580
2/ 300	42.820	44.870	51.120	47.200	50.470	50.510
2 / 500	46.390	46.610	52.100	48.540	51.370	50.560
3 / 300	43.350	45.730	51.520	47.460	50.530	45.330
3 / 500	44.990	47.670	49.720	49.200	48.980	51.350



Experiment 2

- **Optimizer**

n_layer / hid_dim	SGD	Momentum	Nag	RMSprop	Adagrad	Adadelta	Adam
1 / 300	23.380	34.450	34.470	51.080	40.940	22.790	51.290
1 / 500	24.430	34.980	34.960	50.890	42.820	23.510	51.950
2/ 300	35.140	47.100	46.870	51.310	45.080	26.590	52.960
2 / 500	36.560	47.680	47.680	51.970	47.370	29.540	53.380
3 / 300	31.340	43.140	43.210	51.560	41.360	22.000	51.800
3 / 500	32.360	42.980	43.090	51.060	43.130	24.130	52.860

Experiment 3

- **hid_dim**

hid_dim	Test_acc
128	46.560
256	49.750
512	51.580
1024	52.990
2048	54.190
4096	52.120
8192	53.620

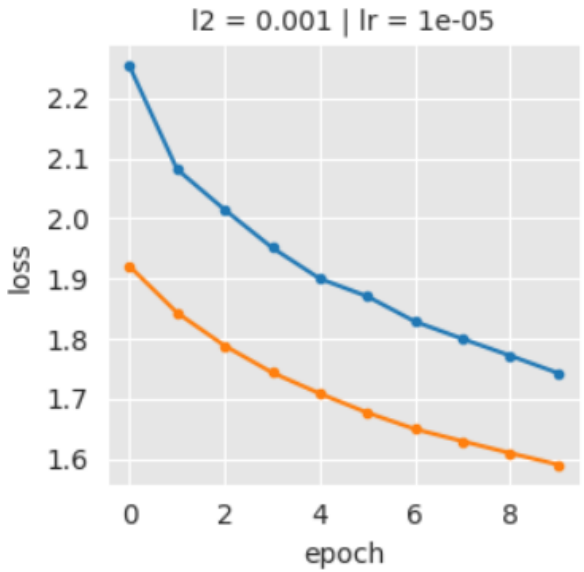
underfitting

overfitting

Experiment 4 (lr vs l2)

- **Epoch = 10 (Adam)**

lr/ l2	Test_acc
0.00001 / 0.00001	44.120
0.00001 / 0.0001	43.870
0.00001 / 0.001	42.760
0.0001 / 0.00001	52.450
0.0001 / 0.0001	51.840
0.0001 / 0.001	51.960
0.0005 / 0.00001	53.770
0.0005 / 0.0001	53.300
0.0005 / 0.001	52.620



- loss가 계속 감소하여 학습이 부족한 것으로 판단 -> Epoch 늘려보기

Experiment 4 (lr vs l2)

- Epoch = 30 (Adam)

lr/ l2	Test_acc
0.00001 / 0.00001	50.610
0.00001 / 0.0001	50.900
0.00001 / 0.001	50.570
0.0001 / 0.00001	55.410
0.0001 / 0.0001	55.540
0.0001 / 0.001	55.730
0.0005 / 0.00001	55.460
0.0005 / 0.0001	54.860
0.0005 / 0.001	54.680

- Epoch = 30 (RMSprop)

lr/ l2	Test_acc
0.00001 / 0.00001	50.920
0.00001 / 0.0001	51.330
0.00001 / 0.001	51.290
0.0001 / 0.00001	56.070
0.0001 / 0.0001	53.650
0.0001 / 0.001	54.150
0.0005 / 0.00001	55.000
0.0005 / 0.0001	54.740
0.0005 / 0.001	54.790

Experiment 5 (Batch size)

- **RMSprop**

Train_batch_size	Test_acc
64	54.700
128	55.670
256	55.050
512	55.220

- **Adam**

Train_batch_size	Test_acc
64	54.950
128	55.830
256	55.190
512	55.350

- batch_size가 작은 경우 : global minimum에 도달해도 안착하지 못하고 서성거림
- batch_size가 큰 경우 : local minimum에 빠지면 탈출하지 못함

Experiment 6

- Node 수
- 기존
  - hid_dim -> hid_dim
    - 장점 : 정보 손실이 거의 없음
    - 단점 : parameter 수가 많아 overfitting이 될 수 있음
- 변경
  - hid_dim -> hid_dim / 2
    - 장점 : parameter 수 감소 / 중요하지 않은 정보는 버림
    - 단점 : 중요한 정보까지 손실 될 수 있음

Experiment 6

- **Adam**

Hid_dim / n_layer	Test_acc	
1024 / 6	48.940	1024 -> 512 -> 256 -> 128 -> 64 -> 10
1024 / 5	53.570	1024 -> 512 -> 256 -> 128 -> 10
1024 / 4	55.500	1024 -> 512 -> 256 -> 10
2048 / 6	54.680	2048 -> 1024 -> 512 -> 256 -> 128 -> 10
2048 / 5	54.830	2048 -> 1024 -> 512 -> 256 -> 10
2048 / 4	55.840	2048 -> 1024 -> 512 -> 10
4096 / 6	54.980	4096 -> 2048 -> 1024 -> 512 -> 256 -> 10
4096 / 5	55.300	4096 -> 2048 -> 1024 -> 512 -> 10
4096 / 4	55.700	4096 -> 2048 -> 1024 -> 10

Experiment 6

- **RMSprop**

Hid_dim / n_layer	Test_acc	
1024 / 6	49.670	1024 -> 512 -> 256 -> 128 -> 64 -> 10
1024 / 5	54.130	1024 -> 512 -> 256 -> 128 -> 10
1024 / 4	55.110	1024 -> 512 -> 256 -> 10
2048 / 6	54.620	2048 -> 1024 -> 512 -> 256 -> 128 -> 10
2048 / 5	55.920	2048 -> 1024 -> 512 -> 256 -> 10
2048 / 4	55.730	2048 -> 1024 -> 512 -> 10
4096 / 6	55.830	4096 -> 2048 -> 1024 -> 512 -> 256 -> 10
4096 / 5	55.980	4096 -> 2048 -> 1024 -> 512 -> 10
4096 / 4	55.370	4096 -> 2048 -> 1024 -> 10

Scheduler

- 학습이 진행됨에 따라 Learning Rate를 조절하여 Global minimum에 근접하게 하는 기법

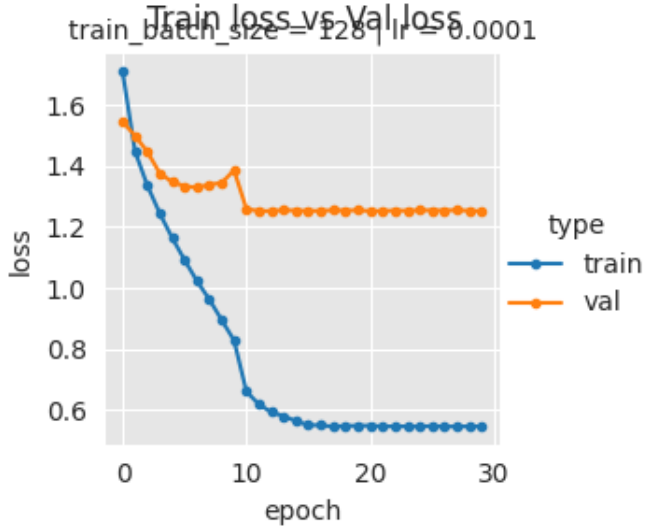
```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=2, factor=0.05)
```

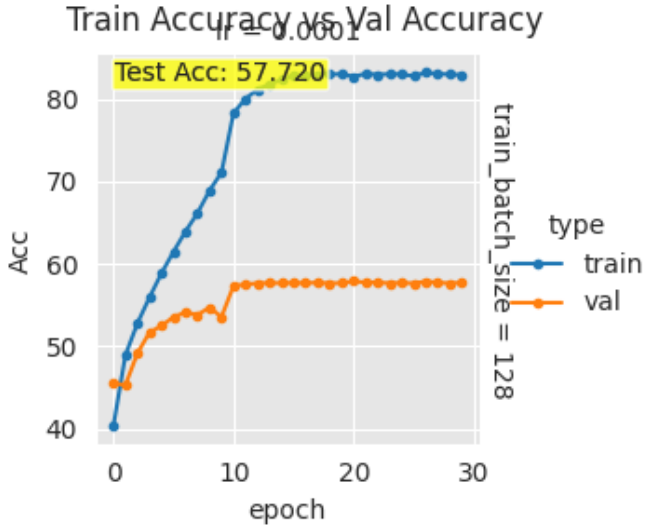
```
scheduler.step(val_loss)
```

- val_loss가 epoch 2번동안 줄어들지 않으면 learning rate * 0.05

Experiment 6

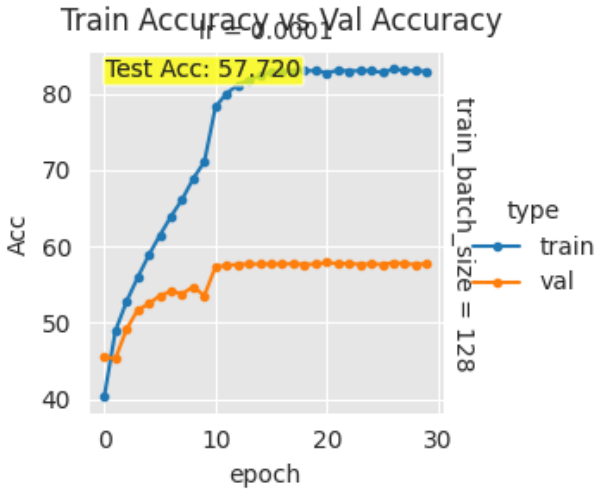
- 결론
  - Activation function = ReLU
  - n_layer : 2
  - hid_dim : 2048
  - Epoch = 30
  - lr = 0.0001
  - Optimizer = RMSprop
  - l2 = 0.00001
  - Train_Batch_size : 128
  - Scheduler 사용





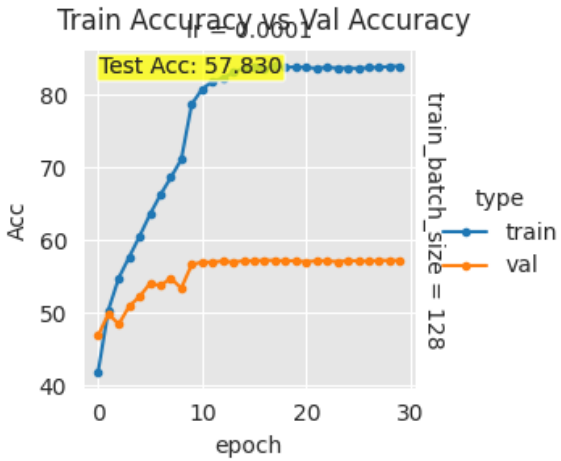
Experiment 7

- Xavier Initialization (O)



- Xavier Initialization (x)

(He Initialization)

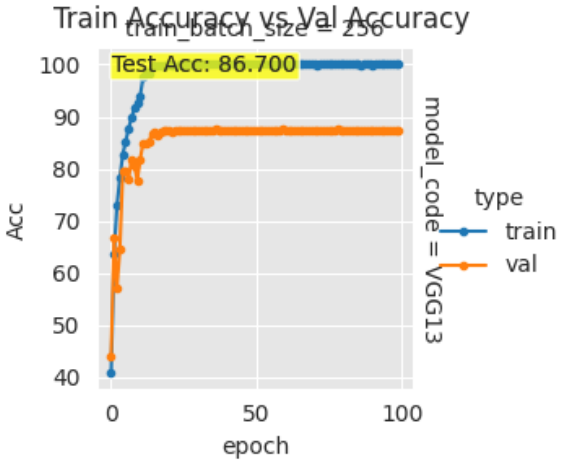


- ReLU와 Xavier Initialization을 동시에 사용하였을 때 보다 He Initialization을 사용하였을 때 성능이 더 높아지는 것을 확인 가능

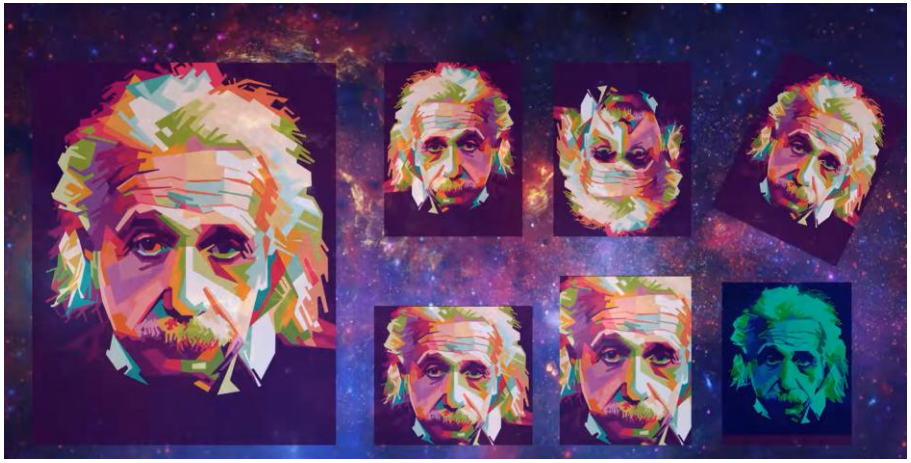
CIFAR 10 with CNN

Data Augmentation

- 여러 Hyperparameter tuning을 진행했지만 overfitting이 해결되지 않아 성능이 86%에서 오르지 않음



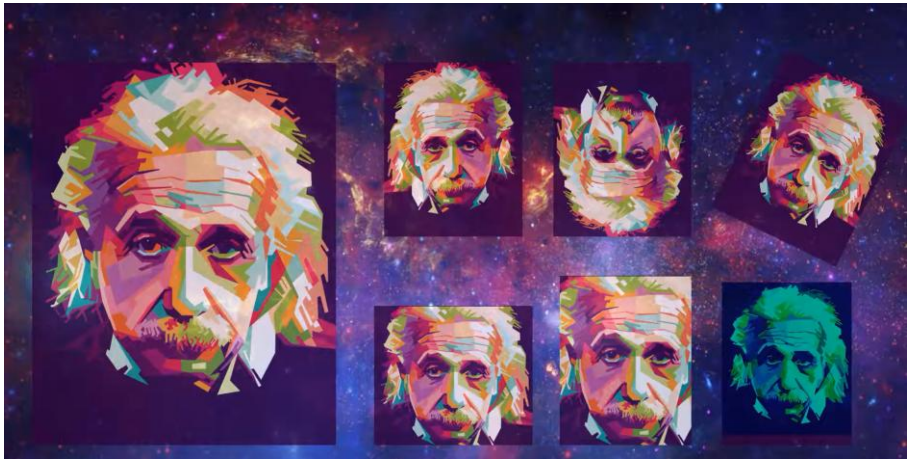
- Data Augmentation
  - 기존 Data를 수정하여 새로운 Data를 얻어내는 기법
    1) Data수가 부족하여 추가 Data를 확보하고 싶은 경우
    2) Overfitting 방지



Data Augmentation

- **Crop / Horizontalflip**
  - 성능 향상
  - 객체의 본질을 해치지 않으면서 다양성 증가

- **Verticalflip / Rotation / Scale**
  - 성능 감소
  - 객체의 본질을 훼손



Initial value

- **Model Related**
  - in_channels = 3
  - out_dim = 10
  - activation function = ReLU
- **Regularization**
  - batch normalization = True
  - dropout = 0
  - Xavier Initialization = False
- **Optimizer & Training**
  - optimizer = Adam
  - learning rate = 0.0001
  - epoch = 30
  - train_batch_size = 256
  - test_batch_size = 1024

```
name_var1 = 'l2'
name_var2 = 'model_code'
list_var1 = [0.000001,0.000005,0.00001, 0.00005]
list_var2 = ['VGG11','VGG13''VGG16', 'VGG19']
```

Initial Value

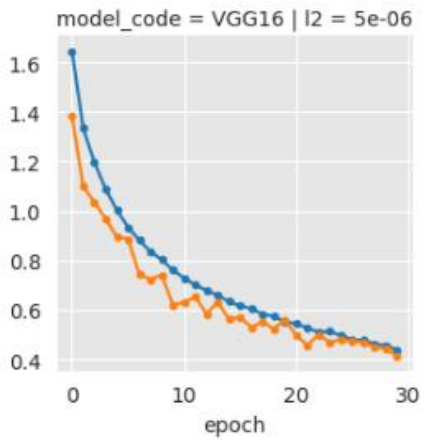
- **VGGNet vs l2**

VGGNet \ l2	0.000001	0.000005	0.00001	0.00005
VGG11	81.910	81.090	82.330	81.360
VGG13	83.540	83.510	83.350	84.550
VGG16	84.050	85.430	84.540	82.880
VGG19	83.340	84.030	84.610	84.980

- **Data Augmentation을 적용**

➔ **test_acc가 70%대에서 80%로 상승**

➔ **Overfitting 완화**

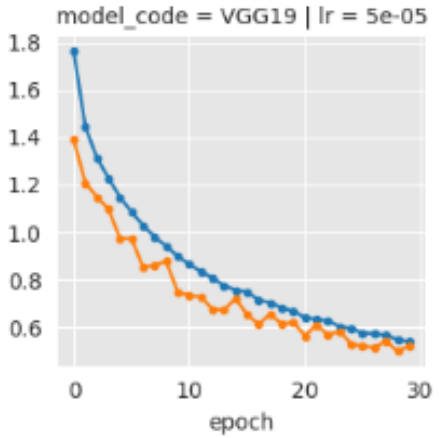


Experiment 1

- **Learning rate**

  - **Epoch=30**

VGGNet(l2) \ lr	0.00001	0.00005	0.0001	0.0005	0.001
VGG13($5*10^{-5}$)	71.960	81.180	83.650	87.090	85.900
VGG16($5*10^{-6}$)	71.500	81.770	83.330	85.880	85.780
VGG19($5*10^{-5}$)	69.820	82.040	84.630	83.720	84.430



- **loss가 계속 감소하여 학습이 부족한 것으로 판단 -> Epoch 늘려보기**

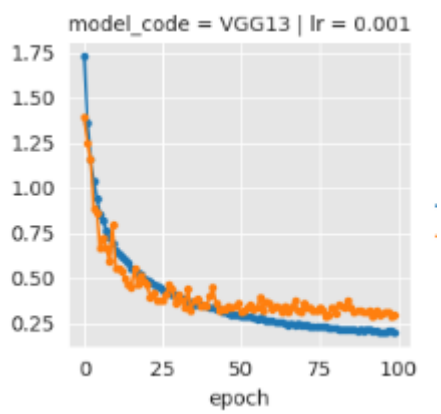
Experiment 1

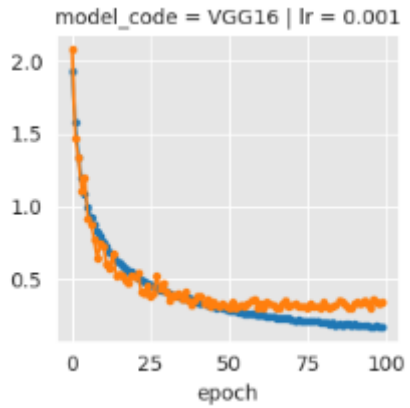
- **Learning rate**

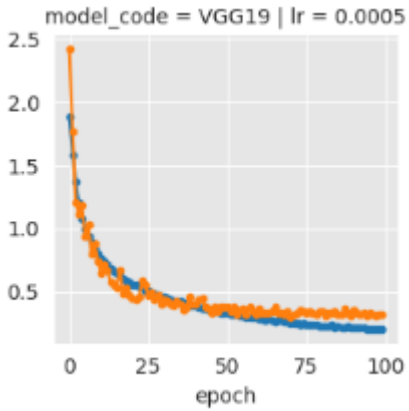
  - **Epoch=100**

VGGNet(l2) \ lr	0.00001	0.00005	0.0001	0.0005	0.001
VGG13($5*10^{-5}$)	79.410	85.480	87.120	90.060	90.840
VGG16($5*10^{-6}$)	78.870	86.180	88.390	90.500	90.720
VGG19($5*10^{-5}$)	78.440	85.780	88.890	90.720	89.330

- **epoch을 늘릴수록 test_acc가 증가하는 경향을 보임**







Experiment 1

- 결론 (Model_code : l2 / lr)
  - VGG 13 : 5 * 10^{-5} / 0.001
  - VGG 16 : 5 * 10^{-6} / 0.001
  - VGG 19 : 5 * 10^{-5} / 0.0005
  - epoch = 100

Experiment 2

- **FC layer + dropout**

VGGNet \ dropout	0	0.1	0.2	0.3	0.4	0.5	0.6
VGG13	90.330	89.810	90.320	90.170	90.600	90.340	90.550
VGG16	90.760	90.370	91.020	90.940	90.570	90.380	90.260
VGG19	90.070	90.560	90.520	90.420	90.020	90.460	89.570

- Epoch을 높이니 overfitting이 생긴다고 판단하여 FC layer에 적절한 dropout 걸어주기
- dropout 이 성능에 큰 영향을 끼치진 못하지만 어느 정도의 성능은 향상

Experiment 3

- **Train_batch_size**

VGGNet \ Batch_Size	32	64	128	256	512
VGG13	89.660	89.700	90.300	90.370	89.320
VGG16	90.530	90.700	91.390	90.660	89.940
VGG19	89.680	89.840	90.390	90.430	90.040

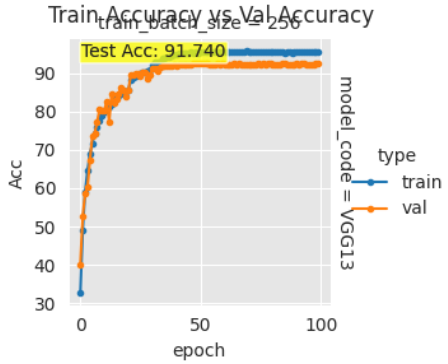
Experiment 2,3

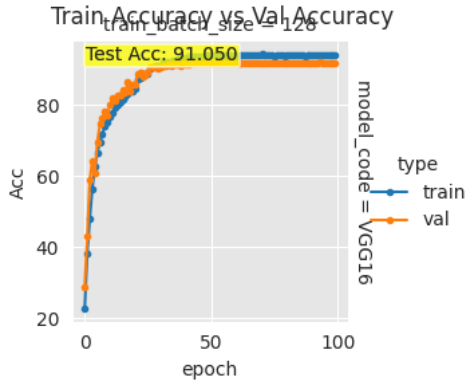
- **결론 (Model_code : l2 / lr / dropout / train_batch_size)**
  - **VGG 13 :** $5 * 10^{-5}$ / 0.001 **/ 0.4 / 256**
  - **VGG 16 :** $5 * 10^{-6}$ / 0.001 **/ 0.2 / 128**
  - **VGG 19 :** $5 * 10^{-5}$ / 0.0005 **/ 0.1 / 256**
  - epoch = 100

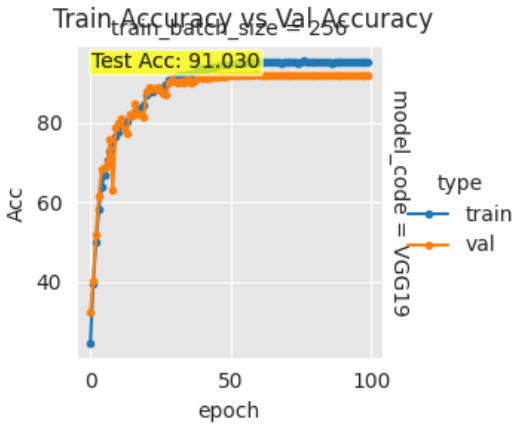
Experiment 4

- **Epoch늘리고 Scheduler까지 사용한 결과**

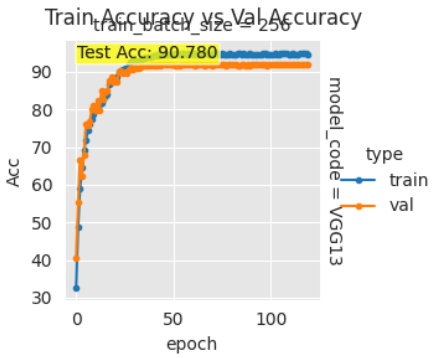
  - **epoch: 100**

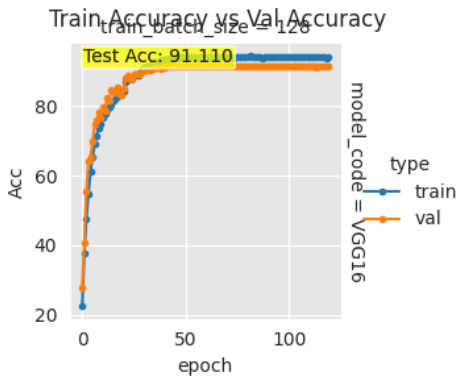


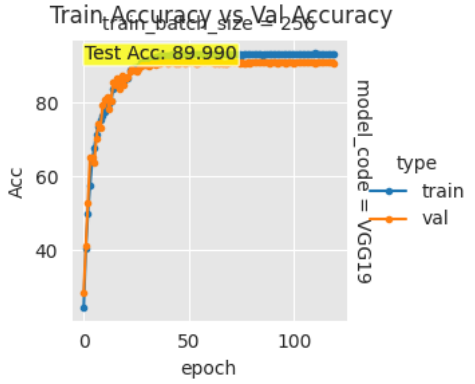




  - **epoch: 120**

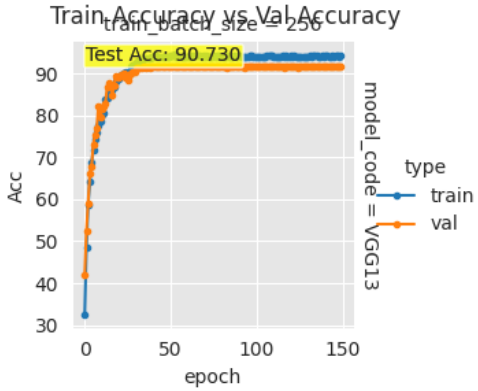


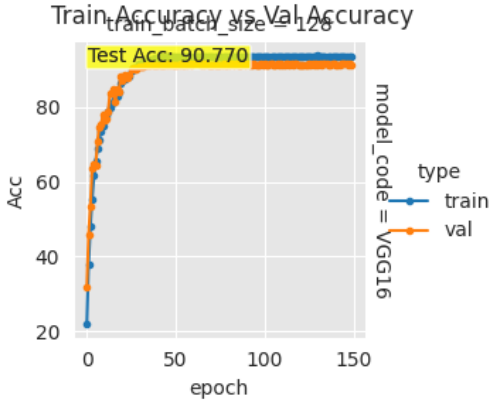


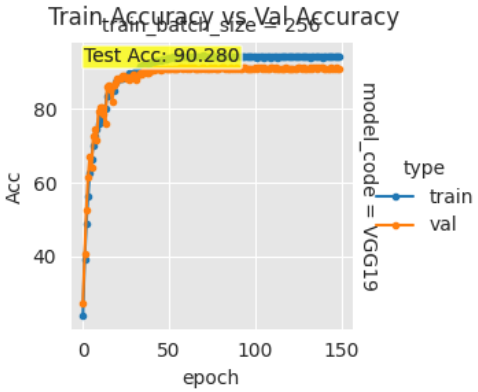


Experiment 4

- **Epoch늘리고 Scheduler까지 사용한 결과**
  - **epoch: 150**



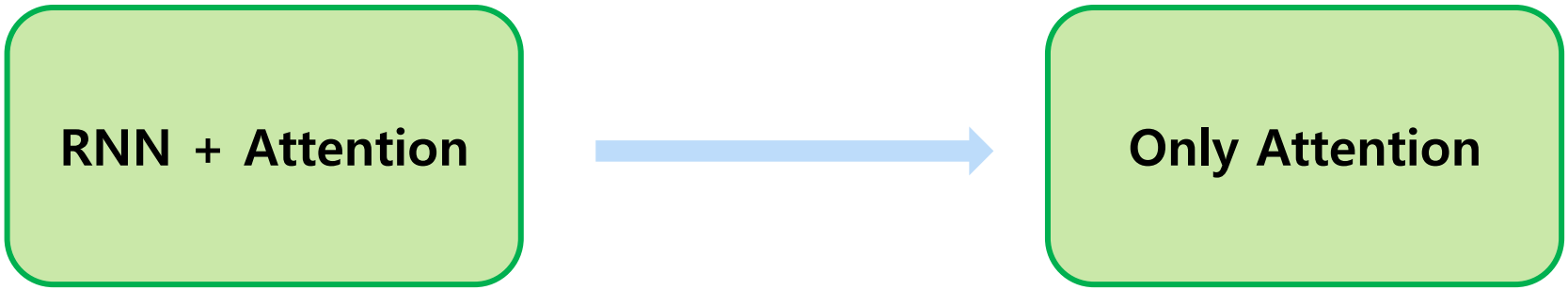




Attention Is All You Need [NeurIPS 2017]

Introduction

- **Motivation**
  - **RNN의 문제 : parallelization의 한계**
    - 병렬계산에 비해 학습시간이 기하급수적으로 길어짐



Background

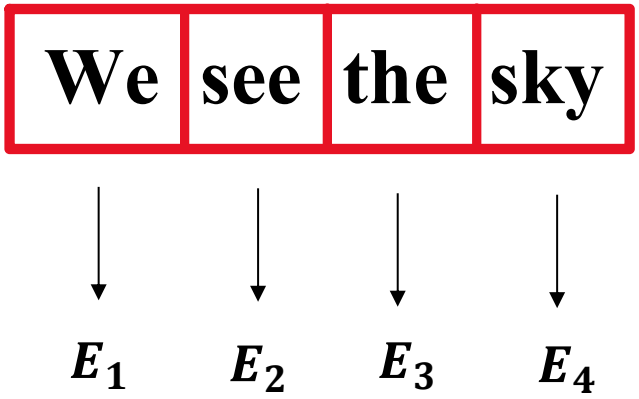
- **Parallel computing을 위한 여러 시도들**
  - **CNN 기반 Model (Neural GPU / ByteNet / convS2S)**
  - **거리가 먼 정보일수록 연산량이 늘어남**

- **Transformer**
  - **연산을 상수 번만 작업해서 처리**
  - **Transformer는 최초로 RNN과 CNN 없이 오직 self-attention에만 전적으로 의존**
    - ➔ **Parallelization 가능**
    - ➔ **거리가 먼 정보여도 즉시 처리 가능**

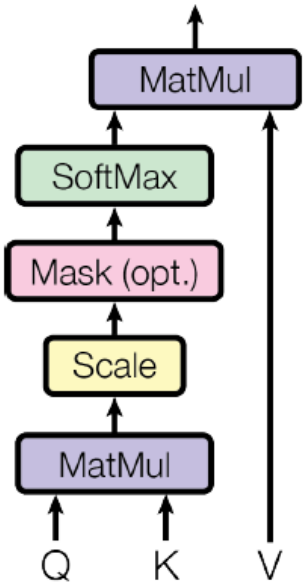
Background

- **Self-attention : 단일 시퀀스 내의 서로 다른 위치를 연관 지어, 그 시퀀스의 표현을 계산하는 매커니즘**

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



Scaled Dot-Product Attention



- **E = Token Embedding(의미) + Positional Embedding(위치)**

Background

- Query = E * W_Q : 자신과 관련된 정보를 찾기 위한 질문
  - W_Q : 가중치 행렬 (훈련 가능한 parameter)

- Key = E * W_k : Query의 식별자
  - W_k : 가중치 행렬 (훈련 가능한 parameter)

- Value = E * W_v : 각 token의 실제 값
  - W_v : 가중치 행렬 (훈련 가능한 parameter)

Background

- 각 Key 와 Query 내적 = 유사도
  - softmax : 합이 1인 확률 분포로 변환
  - d_k : gradient vanishing 방지
  - Masking : cheating 방지

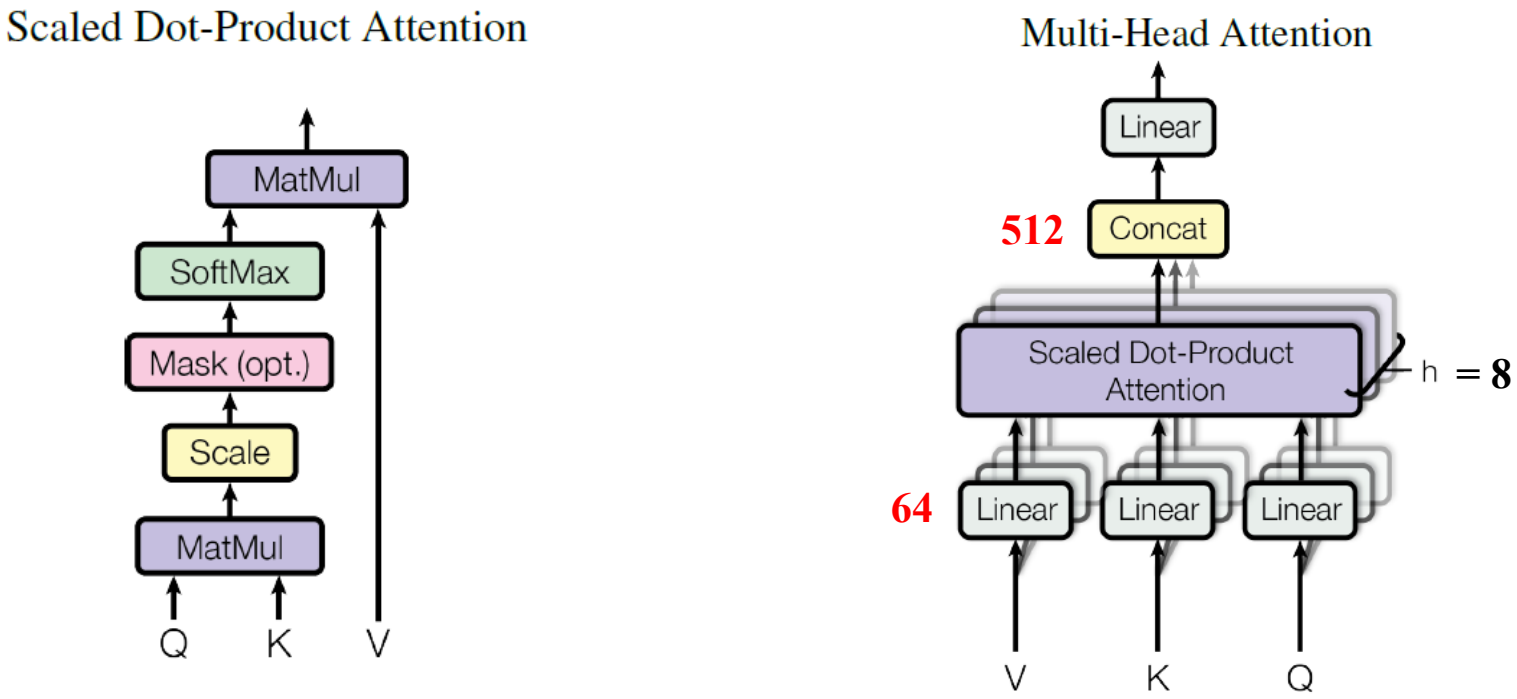
$$\text{Attention (Q,K,V)} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

		we	see	the	sky
		Q_1	Q_2	Q_3	Q_4
we	K_1	10.0	14.5	2.0	5.0
see	K_2	$-\infty$	8.0	3.5	18.0
the	K_3	$-\infty$	$-\infty$	9.0	8.5
sky	K_4	$-\infty$	$-\infty$	$-\infty$	12.0

- 확률과 V 가중합 = context vector
  - 각 token의 문맥의 의미가 담긴 vector

Background

- **Multi-Head Attention**
  - 여러 Head가 다른 특징(문법, 의미, 위치)에 집중하여 명확한 정보 추출



Model Architecture

- **Encoder – Decoder Structure**
- **Feed Forward**
  - Context vector를 더 깊이 있는 정보로 만듬
- **Add (Residual Connection)**
  - 원래 입력값을 잊지 않기 위해 이전 입력값을 더해 줌
- **Norm**
  - 각 벡터의 값들을 평균 0, 분산 1이 되도록 정규화

