Computer Vision and Signal Processing Lab.
CS231 및 CNN 코드

–세미나 발표–



Pusan National University
Dept. of Geology
Dept. of Electronic Engineering



Presented By
김소형

Outline

- Image Classification
- Deep learning Concept
- Convolutional Neural Networks
- Training Neural Networks
- Code Review
- Conclusion

Image Classification

- Image Classification
  - Image Classification: A core task in Computer Vision
  - Data-Driven Approach: 데이터 중심 접근 방법
  - Parametric Approach: 파라미터 기반 접근 방법

Image Classification

- Image Classification
  - Image Classification: A core task in Computer Vision
    - Challenge
      - Semantic Gap: 인간이 인식하는 의미와 컴퓨터의 차이
      - Viewpoint variation: 보는 방향이 조금만 달라져도 픽셀 값이 변화

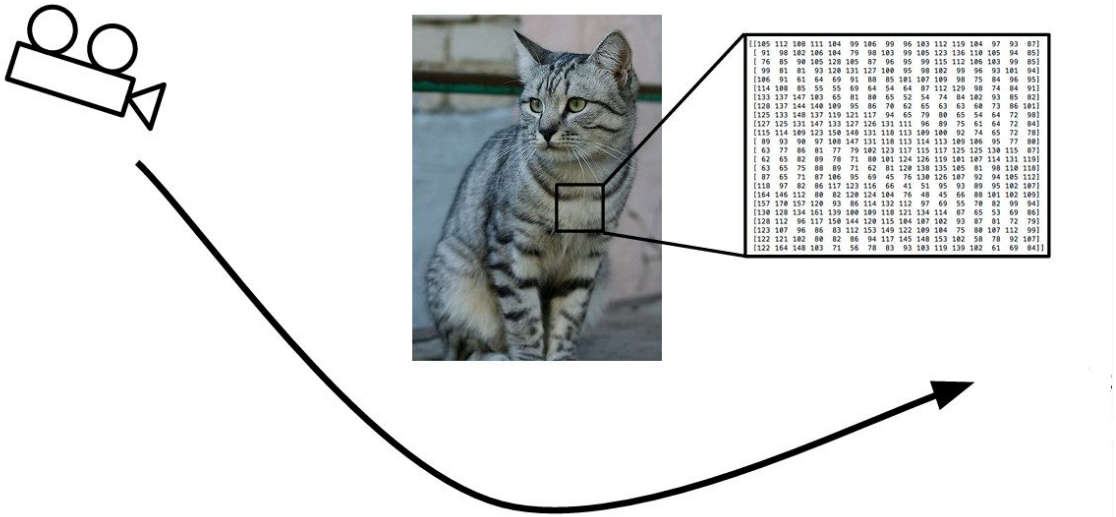
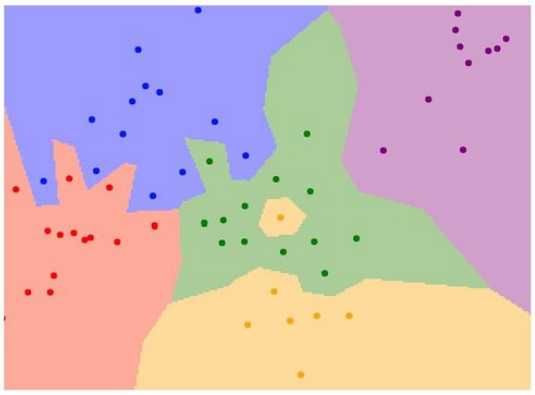


Image Classification

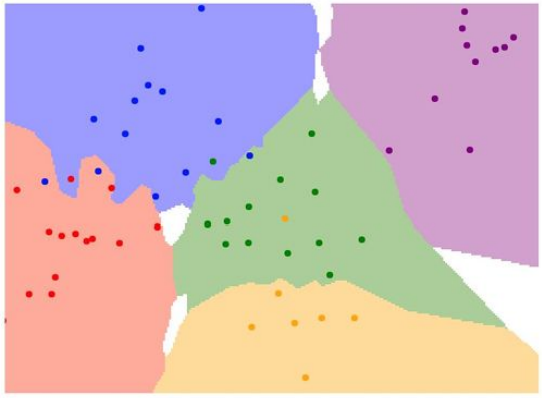
- Image Classification
  - Data-Driven Approach: 데이터 중심 접근 방법
    - Train time
      - 학습이미지를 단순 저장하므로 짧은 시간 소요
    - Test time
      - 학습이미지와 input이미지를 일일이 비교하므로 비교적 많은 시간 소요
    - 문제점
      - Test는 사용자가 하기에 시간이 오래 걸리면 좋지 못한 Classifier

Image Classification

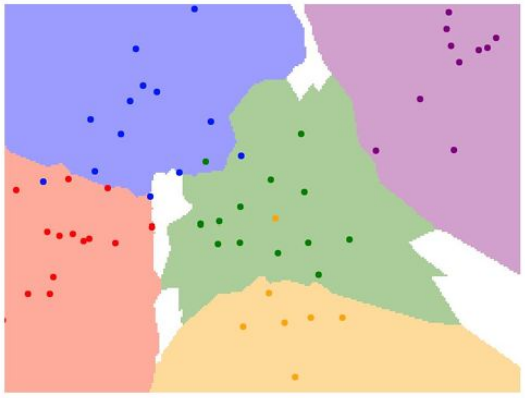
- Image Classification
  - Data-Driven Approach: 데이터 중심 접근 방법
    - K-NN 알고리즘: 임의의 지점에서 가장 가까운 point의 class로 분류
      - K는 가까운 point의 개수, 하나의 class로 특정짓지 못할 때 흰색으로 처리



K = 1



K = 3



K = 5

Image Classification

- Image Classification
  - Data-Driven Approach: 데이터 중심 접근 방법
    - K-NN 알고리즘: 임의의 지점에서 가장 가까운 point의 class로 분류
      - problem: Curse of dimensionality
        - ✔ 차원이 증가함에 따라 데이터가 기하급수적으로 늘어나므로 학습시간 대폭 증가

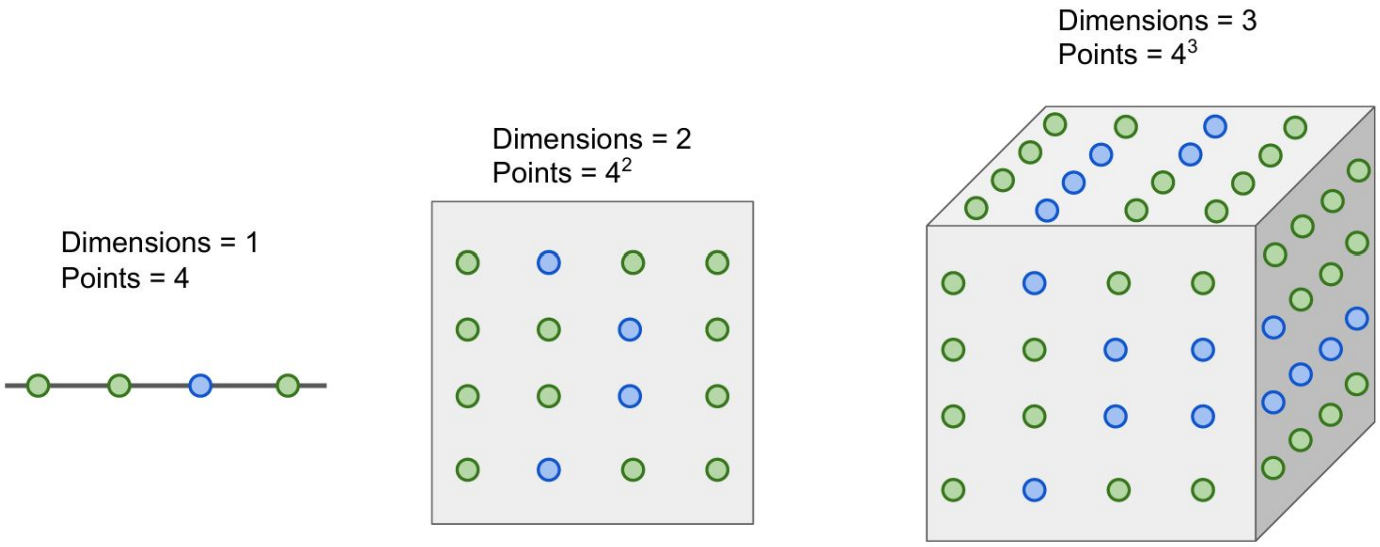


Image Classification

- Image Classification
  - Parametric Approach: 파라미터 기반 접근 방법
    - Linear Classifier
      - 학습이미지를 저장하지 않고 파라미터를 도입하여 이를 학습

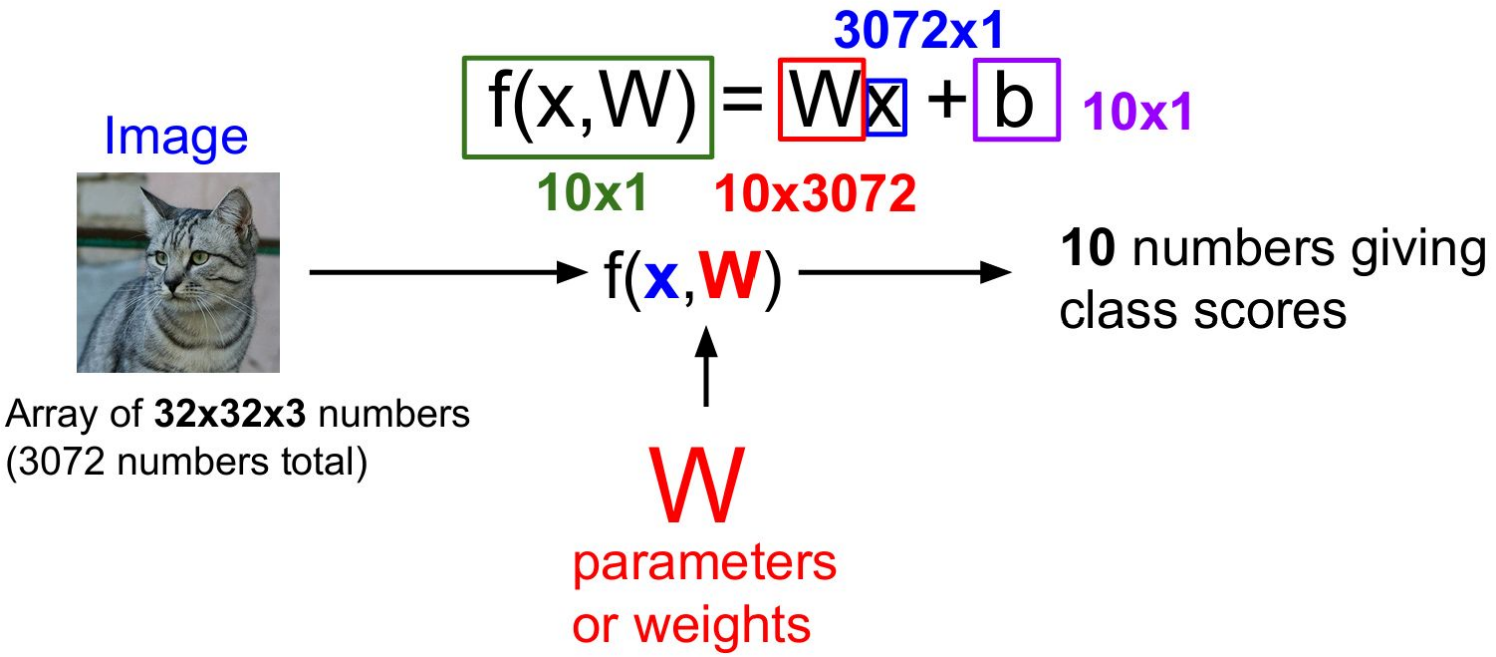
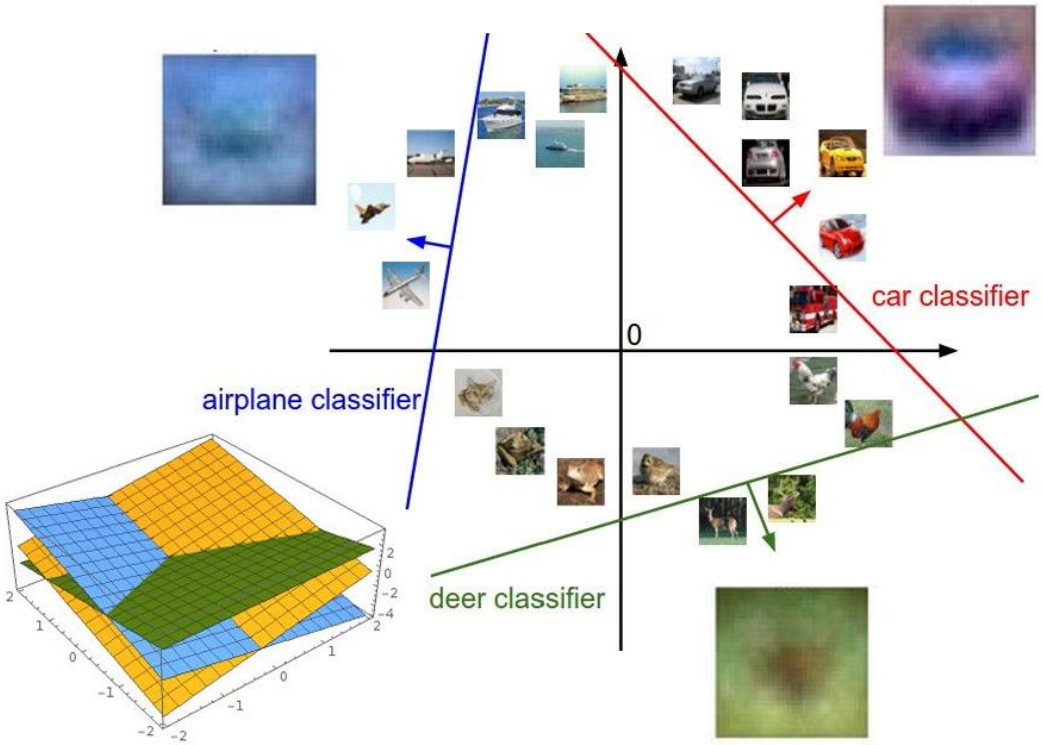


Image Classification

- Image Classification
  - Parametric Approach: 파라미터 기반 접근 방법
    - Linear Classifier
      - Feature map에서 feature가 유사할수록 가까이 위치

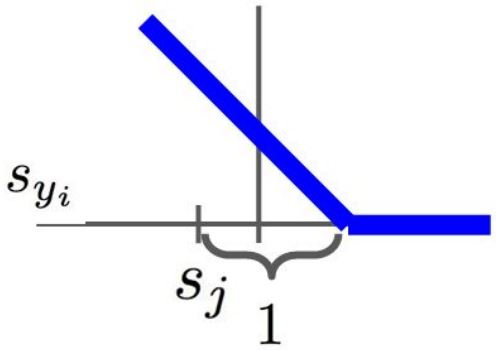


Deep learning Concept

- Deep learning Concepts
  - Loss function: 모델의 예측이 얼마나 틀렸는지를 측정하는 함수
  - Backpropagation: Loss값을 줄이기 위해 각 가중치에 대한 gradient를 계산하는 알고리즘
  - Optimizer: Backpropagation으로 계산된 gradient를 사용해 가중치를 실제로 업데이트하는 방법

Deep learning Concept

- Deep learning Concepts
  - Loss function
    - Hinge Loss
      - Multiclass SVM: Hinge loss를 최소화하도록 학습하는 모델



$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Deep learning Concept

- Deep learning Concepts
  - Loss function
    - Hinge Loss
      - Multiclass SVM: Hinge loss를 최소화하도록 학습하는 모델





cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Deep learning Concept

- Deep learning Concepts
  - Loss function
    - Cross Entropy Loss
      - Softmax: 여러 클래스 중 어떤 클래스에 속할 확률인지를 출력

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \qquad L_i = -\log P(Y = y_i|X = x_i)$$

Softmax function

$$L_i = -\log(\frac{e^{sy_i}}{\sum_j e^{s_j}})$$



unnormalized probabilities

		exp		normalize		
cat	**3.2**	→	**24.5**	→	**0.13**	→ L_i = -log(0.13) = **0.89**
car	5.1		164.0		0.87	
frog	-1.7		0.18		0.00	

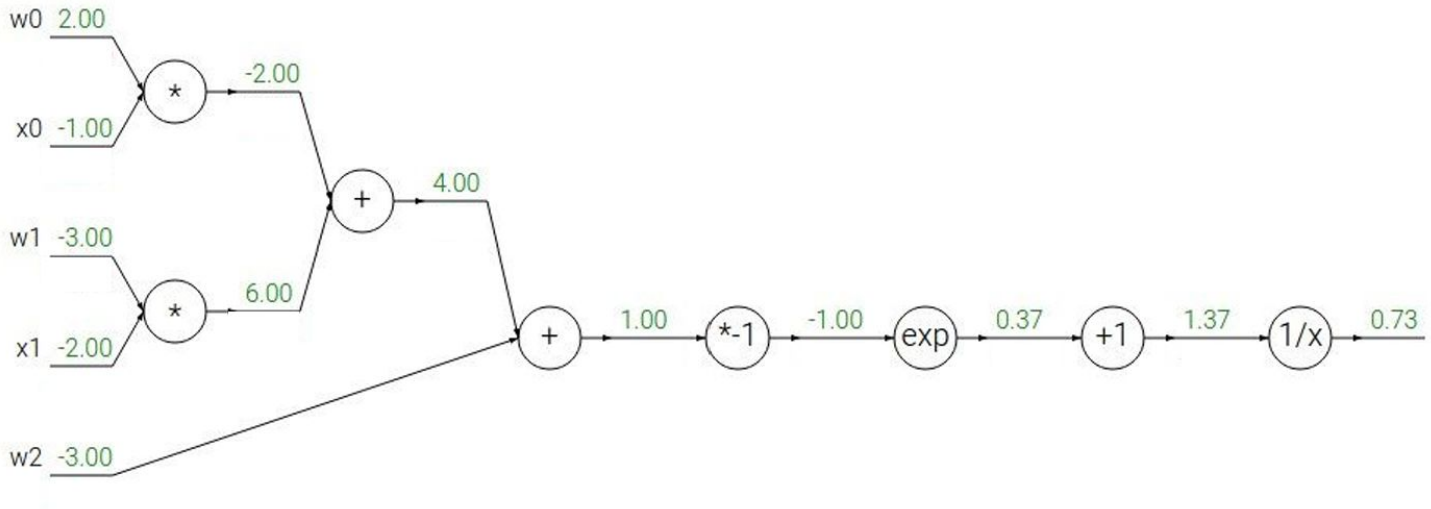
unnormalized log probabilities

probabilities

Deep learning Concept

- Deep learning Concepts
  - Backpropagation
    - Computational graph
      - gradient를 효율적이고 체계적으로 계산하기 위한 도구

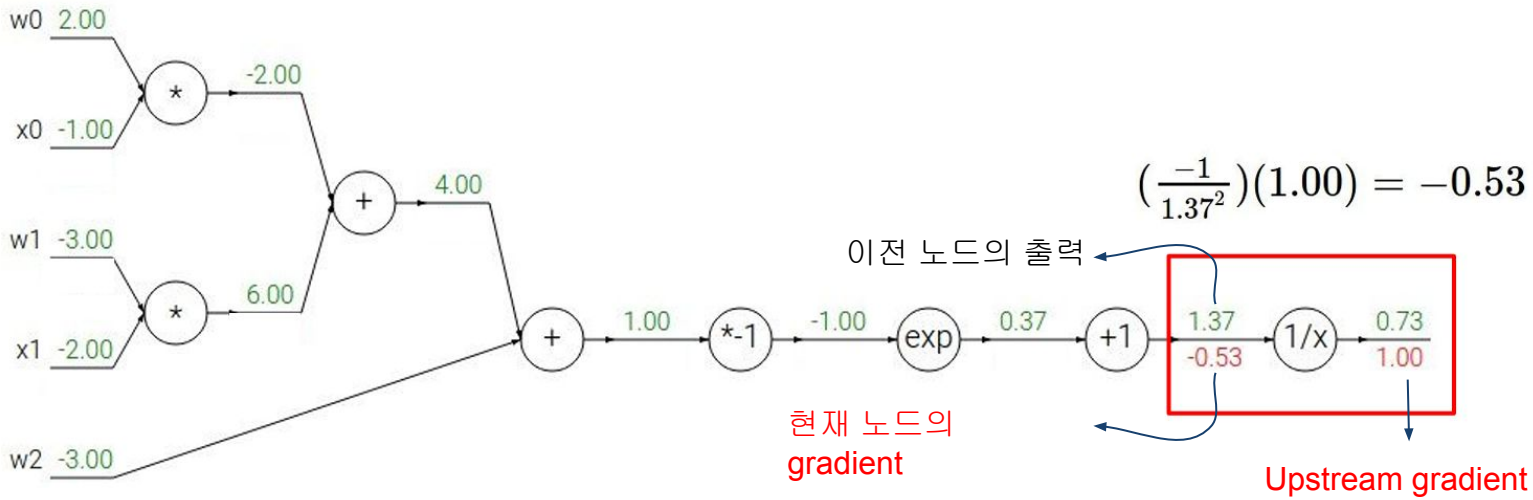
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Deep learning Concept

- Deep learning Concepts
  - Backpropagation
    - Computational graph
      - (Upstream gradient) x (Local gradient) = 현재 노드의 gradient
        - ✔ Local gradient: 현재 노드의 미분식에 이전 노드의 출력을 대입

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

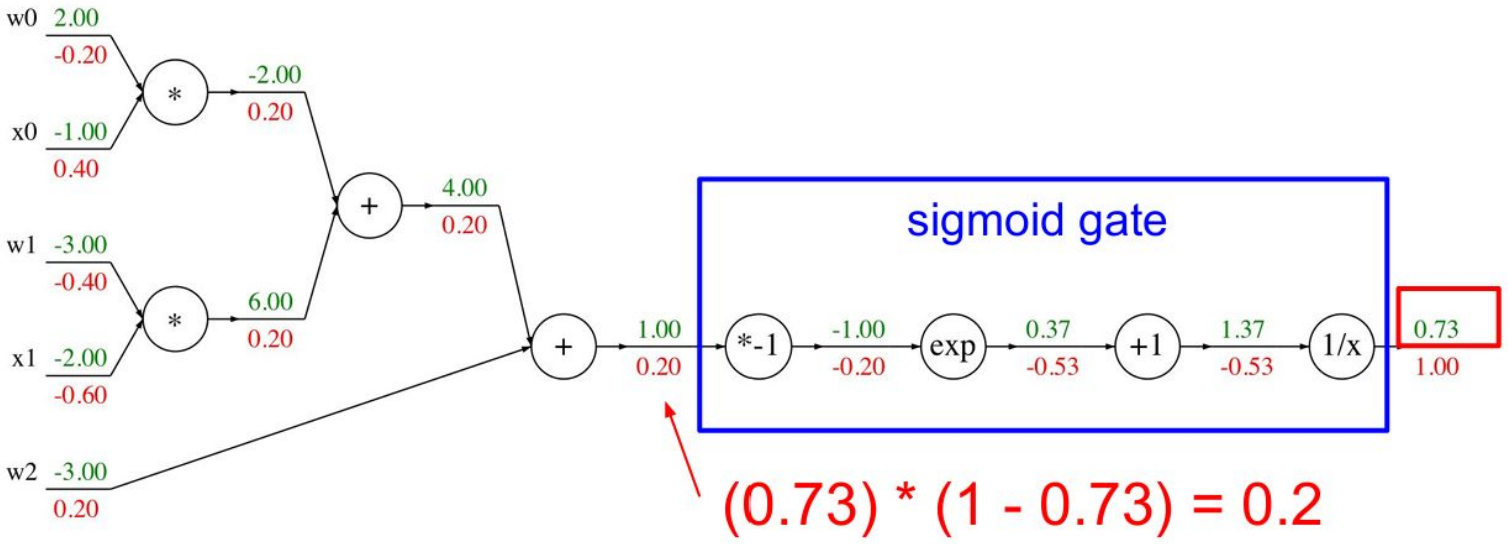


Deep learning Concept

- Deep learning Concepts
  - Backpropagation
    - Computational graph

여러 노드가 하나의 함수로 표현된다면 연산을 한 번에 수행 가능

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



Deep learning Concept

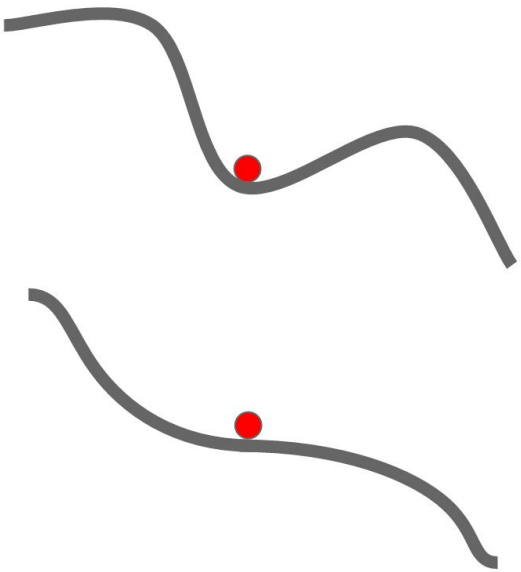
- Deep learning Concepts
  - Optimizer
    - Gradient Descent(GD)
      - 전체 데이터의 loss 계산 후, 한 번에 업데이트
    - Vanilla SGD
      - 하나의 데이터 샘플로 loss 계산 후, 업데이트. 이 후 반복
    - SGD + Momentum
      - 이전 gradient 방향을 관성처럼 이어받아, 안정적이고 더 빠른 수렴
    - Nesterov Momentum
      - 기존 momentum의 lookahead 버전
    - AdaGrad
      - 파라미터마다 learning rate 다르게 조정
    - RMSProp
      - AdaGrad에서 learning rate가 너무 작아지는 것을 방지
    - Adam
      - RMSProp + Momentum + Bias Correction(초기 학습 보정)

Deep learning Concept

- Deep learning Concepts
  - Optimizer
    - Vanilla SGD
      - 하나의 데이터 샘플로 loss 계산 후, 업데이트. 이 후 반복
        - ✔ problem:local minima이나 saddle point에서 잘 빠져나오지 못함

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

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

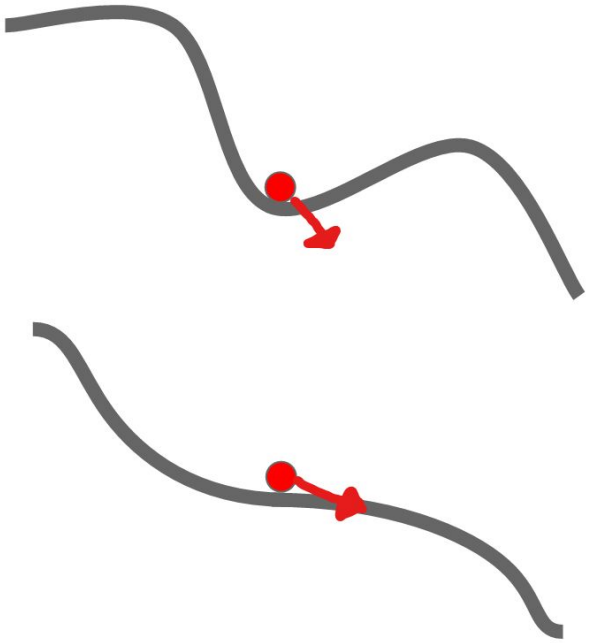


Deep learning Concept

- Deep learning Concepts
  - Optimizer
    - SGD + Momentum
      - 이전 gradient 방향을 관성처럼 이어받아, 안정적이고 더 빠른 수렴

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

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



Deep learning Concept

- Deep learning Concepts
  - Optimizer
    - RMSProp
      - AdaGrad에서 learning rate가 너무 작아지는 것을 방지

AdaGrad

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



RMSProp

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

Deep learning Concept

- Deep learning Concepts
  - Optimizer
    - Adam
      - RMSProp + Momentum + Bias Correction(초기 학습 보정)
        - ✔ Bias Correction: 초기 momentum이 값이 너무 작아지는 것을 방지

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

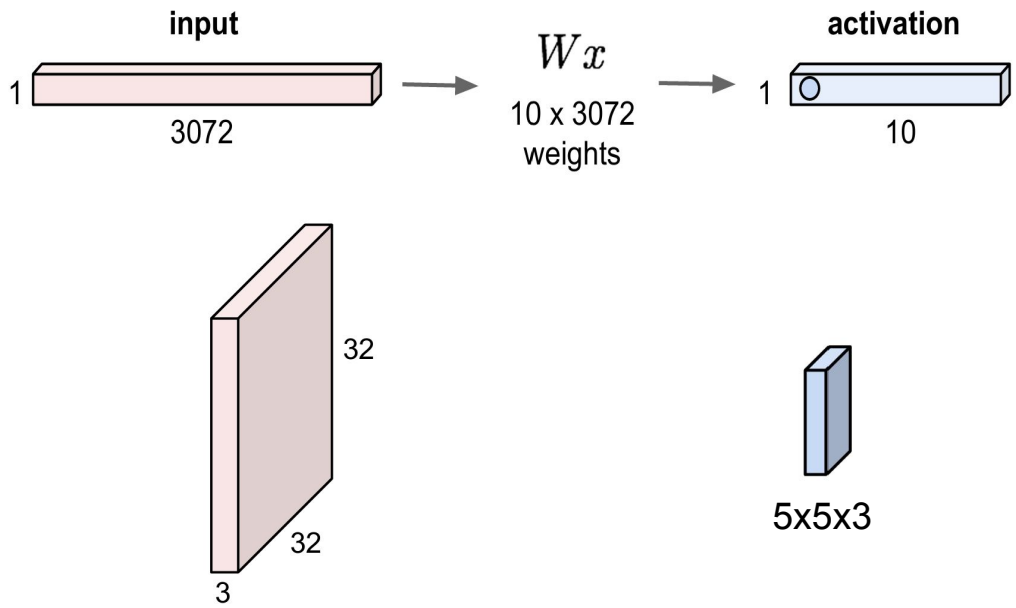
AdaGrad / RMSProp

Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 등장 배경
  - CNN의 동작 과정
  - CNN Architectures

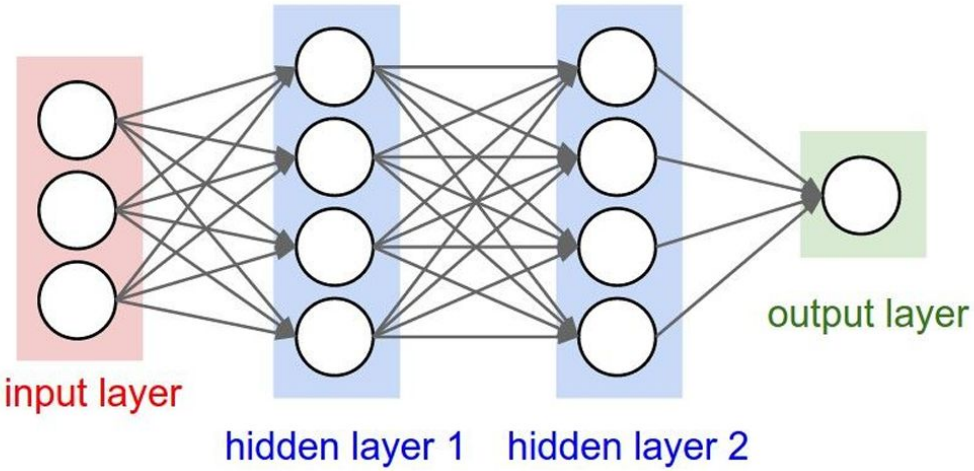
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 등장 배경
    - MLP의 한계
      - input의 기존 구조를 변형
        - ✔ CNN은 기존 구조는 유지하며 필터를 도입



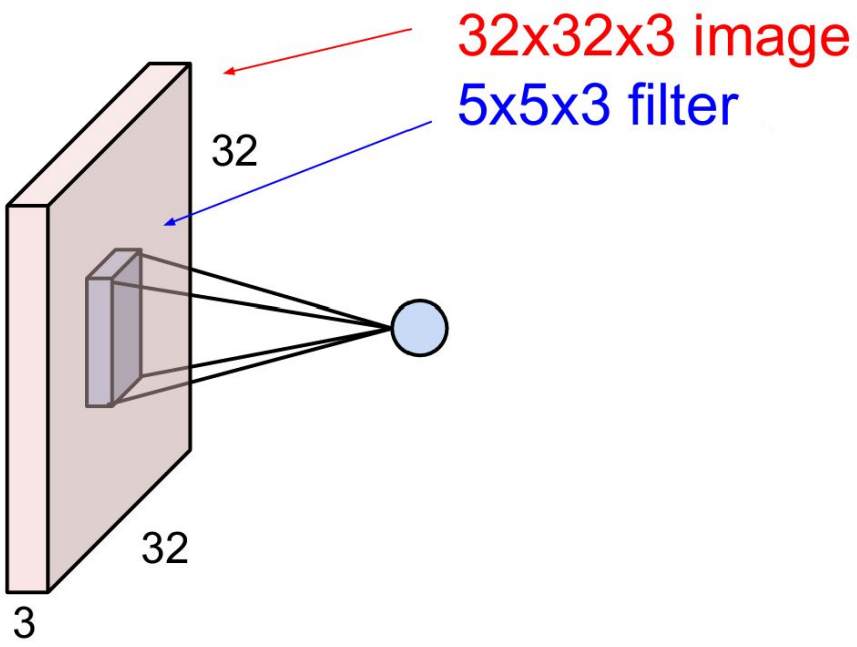
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 등장 배경
    - MLP의 한계
      - input의 차원이 커질수록 파라미터 수의 폭발적 증가
        - ✔ CNN은 필터를 도입함으로써 파라미터의 재사용이 가능해짐



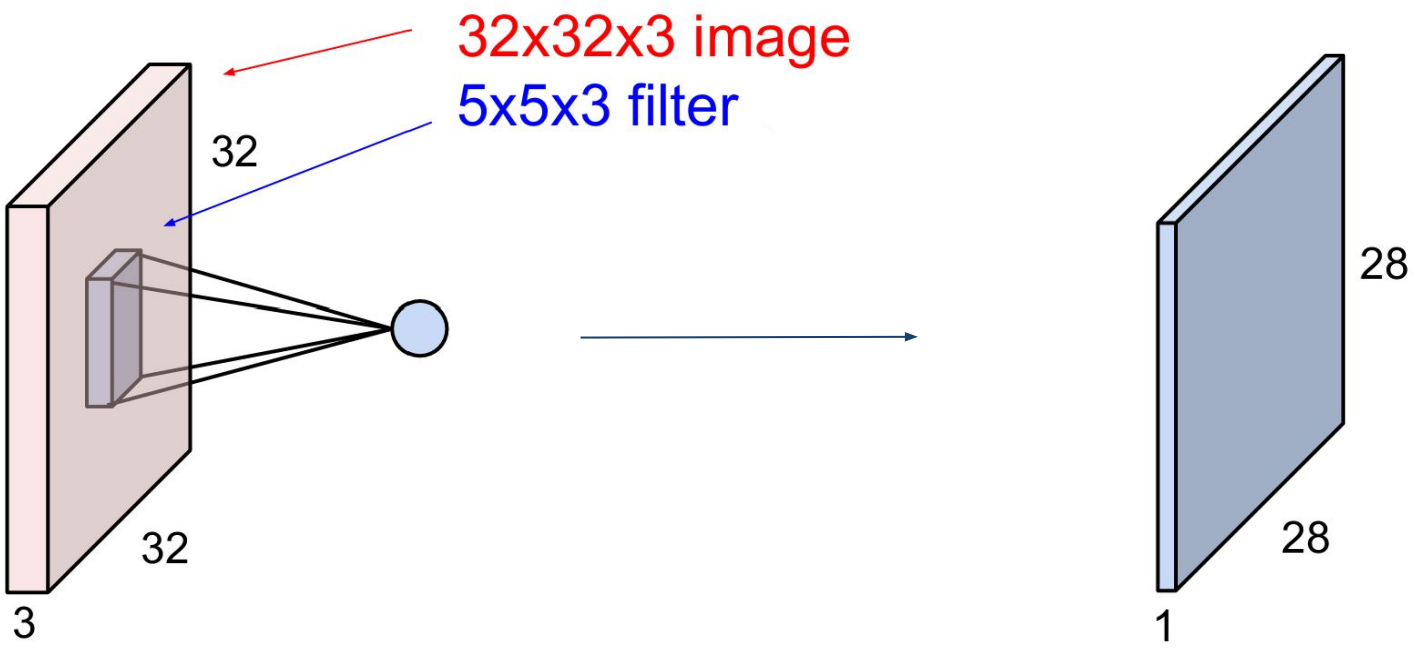
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Convolutional layer
      - 필터 하나가 이미지에 슬라이딩하며 지역적인 특징 추출



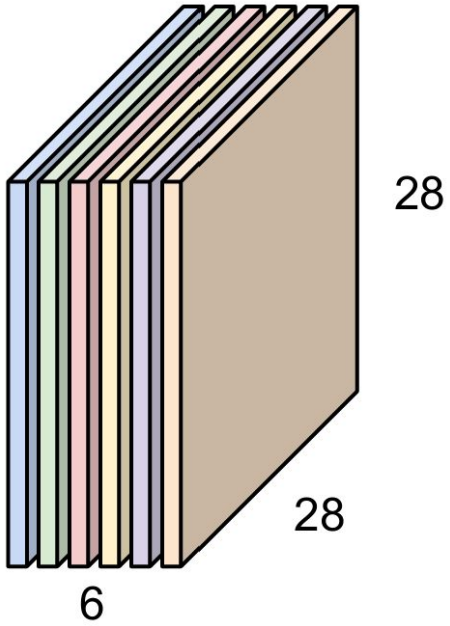
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Convolutional layer
      - 필터 하나가 이미지에 슬라이딩하며 activation map 형성



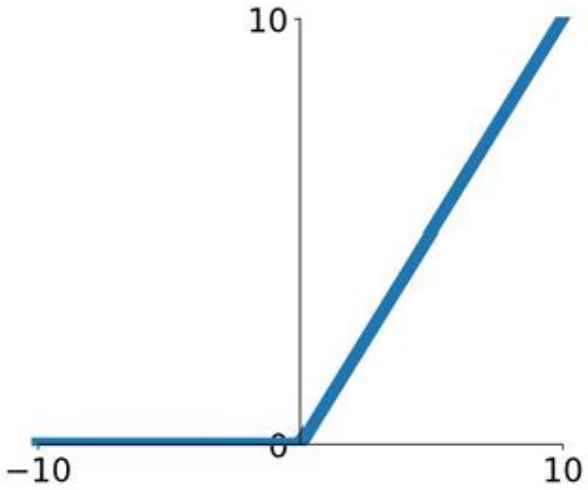
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Convolutional layer
      - 여러 필터를 사용하여 여러 개의 activation map 형성



Convolutional Neural Networks

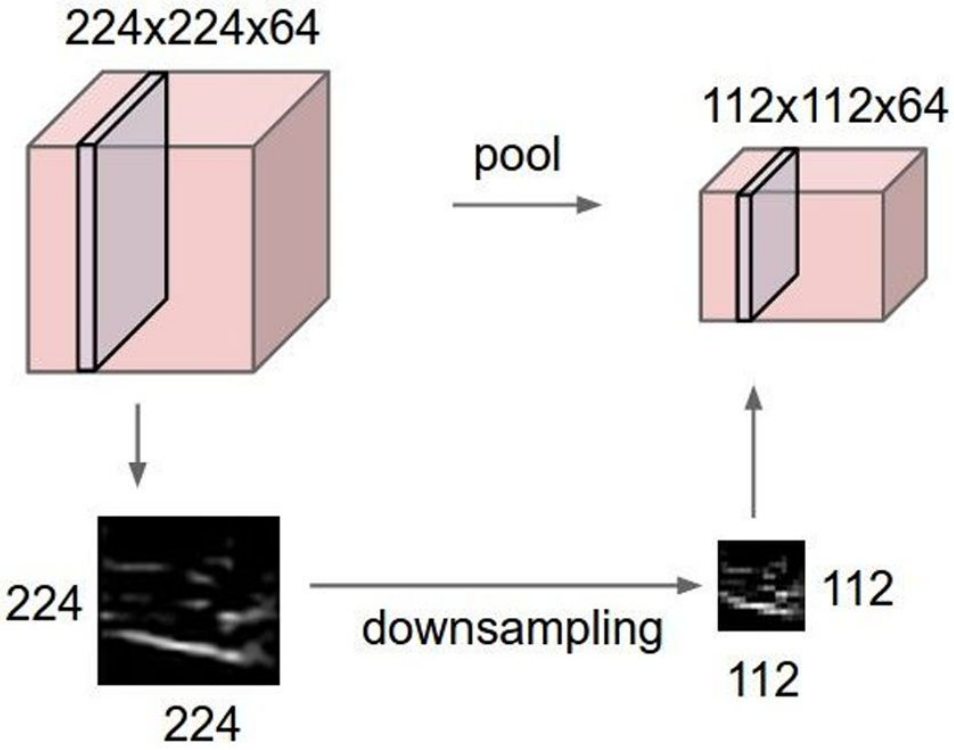
- Convolutional Neural Networks
  - CNN의 동작 과정
    - ReLU
      - activation map에 비선형성 추가



$$f(x) = \max(0,x)$$

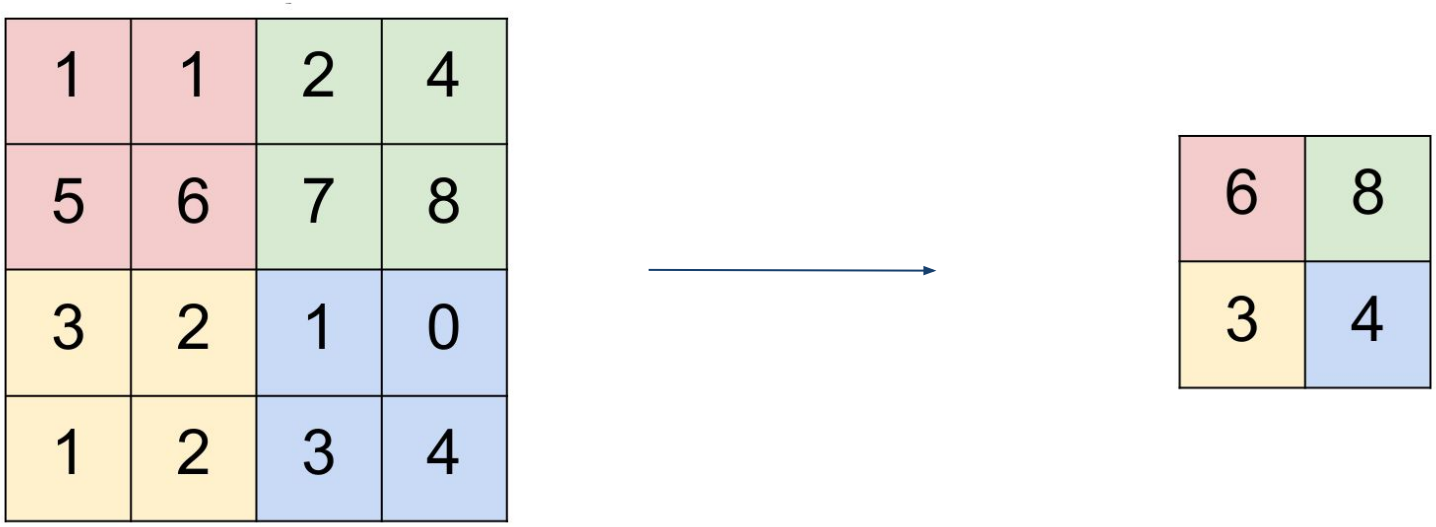
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Pooling layer
      - 각 activation map의 크기를 줄이며 중요한 정보만 유지



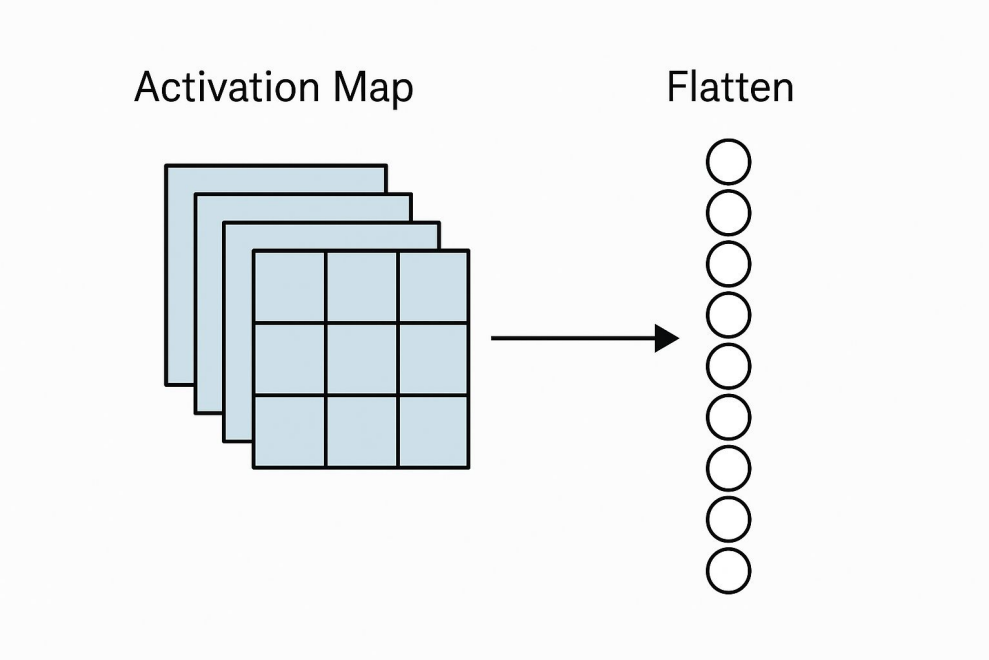
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Pooling layer
      - 일반적으로 Max Pooling 사용



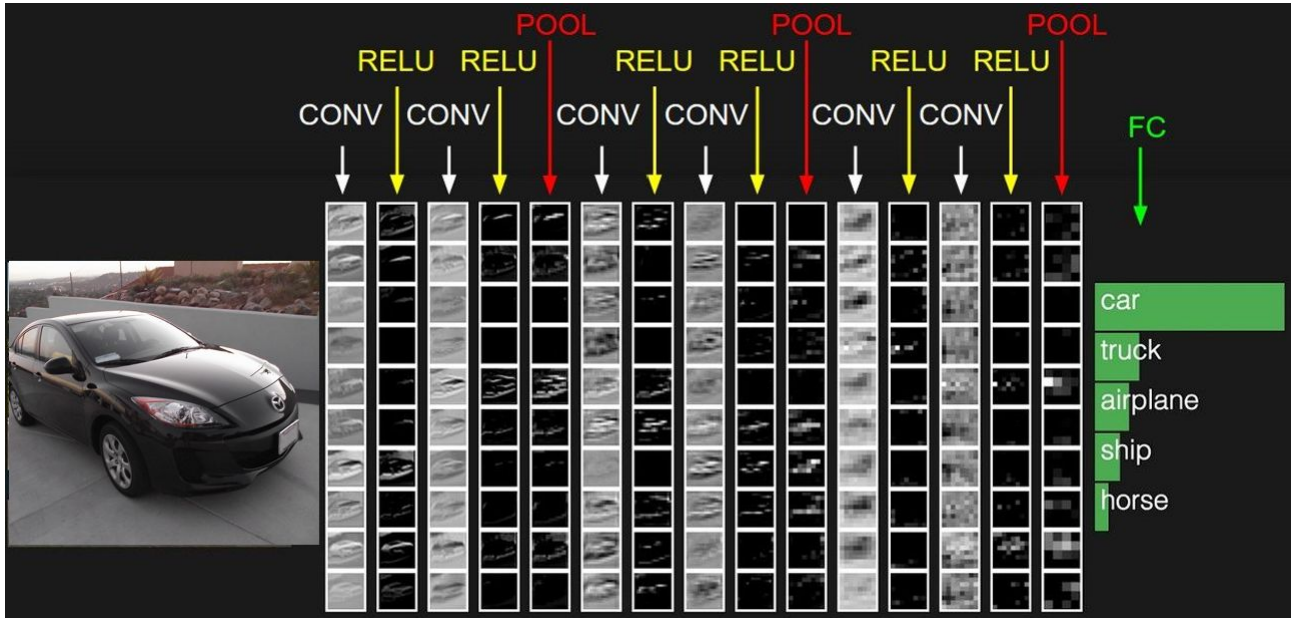
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - Flatten
      - 출력 전 마지막 activation map을 1D 벡터로 펼침
        - ✔ FC layer에 넣기 위함



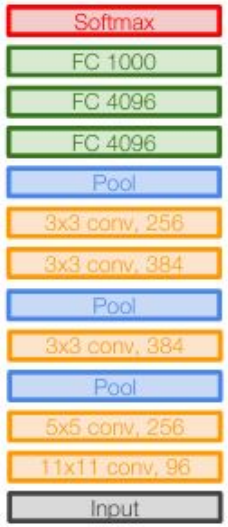
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN의 동작 과정
    - FC layer
      - 추출된 특징을 바탕으로 최종 판단



Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN Architectures
    - AlexNet(2012)
      - 최초의 CNN기반 ImageNet 우승자
        - ✔ error rate: 16.4%



AlexNet

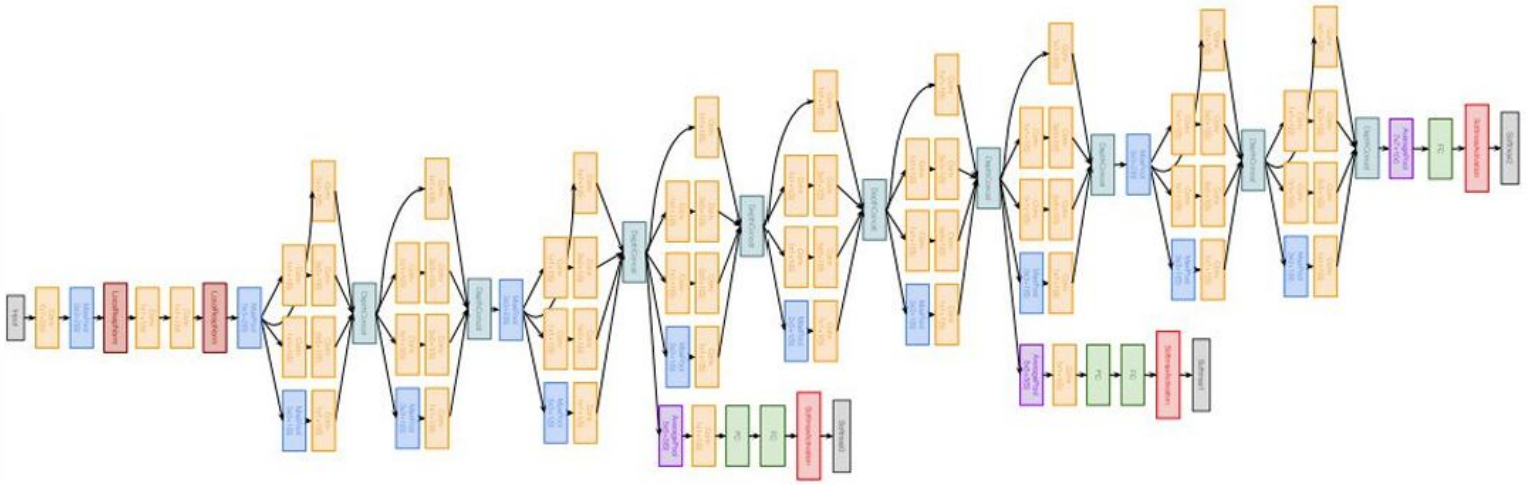
Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN Architectures
    - VGG Net(2014)
      - Alex Net에서 좀 더 깊고 정돈된 구조
        - ✔ error rate: 7.3% (VGG19)



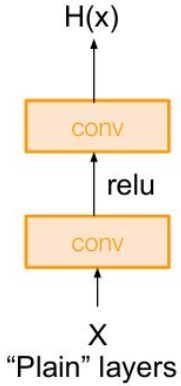
Convolutional Neural Networks

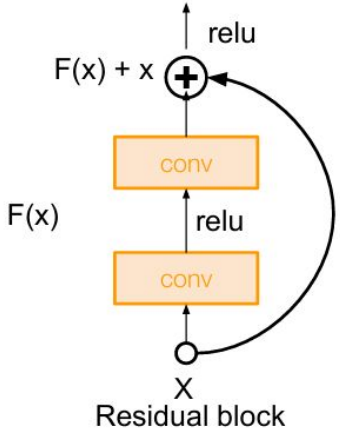
- Convolutional Neural Networks
  - CNN Architectures
    - GoogleNet(2014)
      - 여러 종류의 필터를 병렬로 적용
        - ✔ error rate: 6.7%

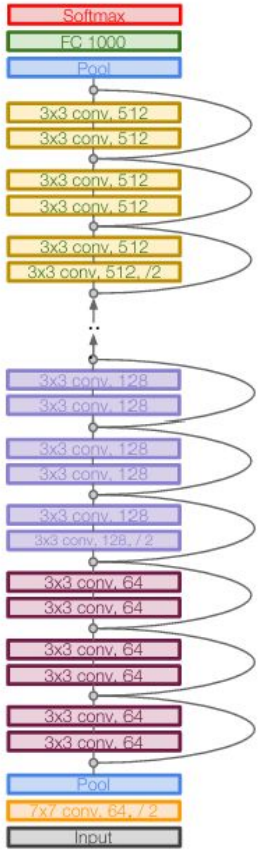


Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN Architectures
    - ResNet(2015)
      - skip connection: 전체 함수 학습 대신 변화량을 학습
        - ✔ error rate: 3.57%

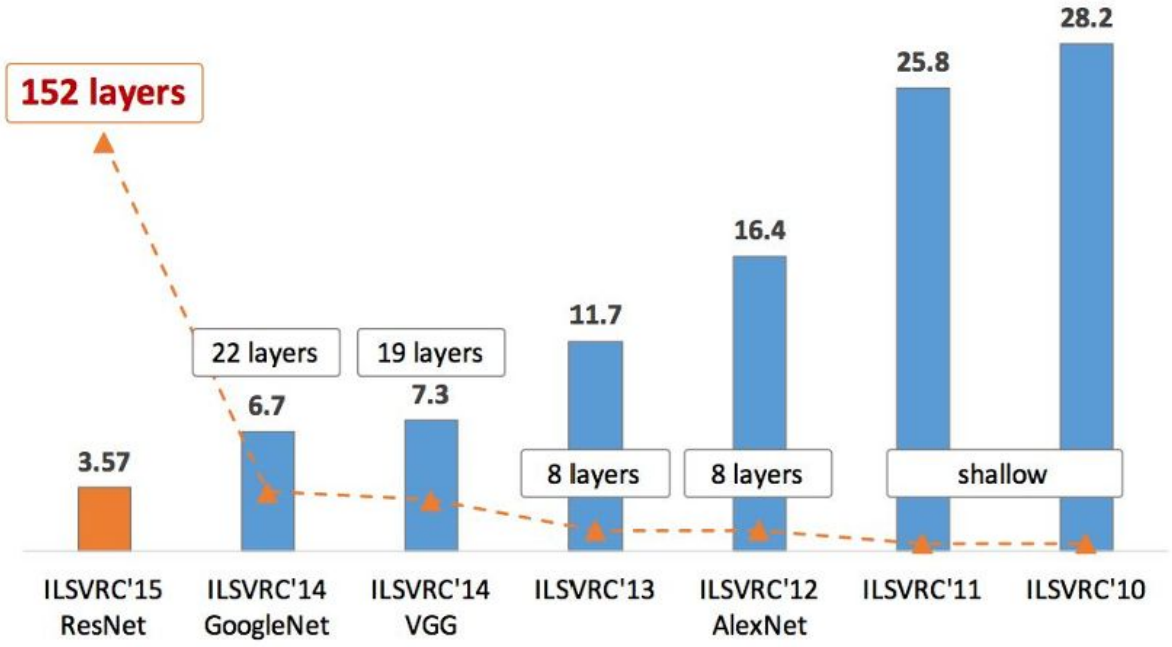






Convolutional Neural Networks

- Convolutional Neural Networks
  - CNN Architectures
    - error rate 비교 그래프



Training Neural Networks

- Training Neural Networks
  - Activation Functions
  - Data Preprocessing
  - Weight Initialization
  - Batch Normalization

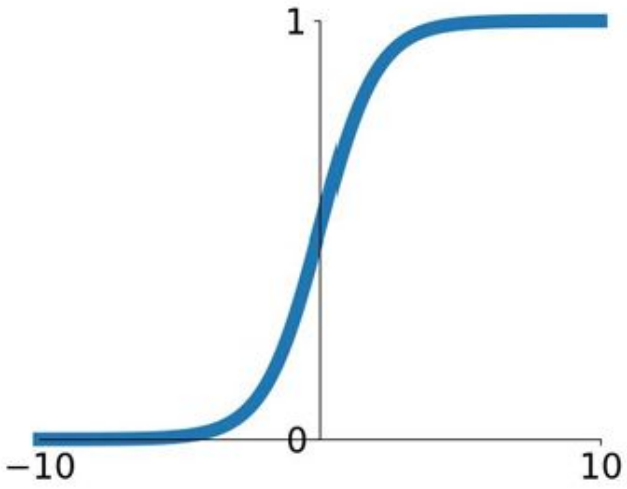
Training Neural Networks

- Training Neural Networks
  - Activation Functions
    - Activation Function이 없다면 비선형성을 주지 못해 layer를 아무리 깊게 쌓아도 의미가 없게 됨

$$z_2 = W_2(W_1x + b_1) + b_2$$
$$= (W_2W_1)x + (W_2b_1 + b_2)$$
$$= W'x + b'$$

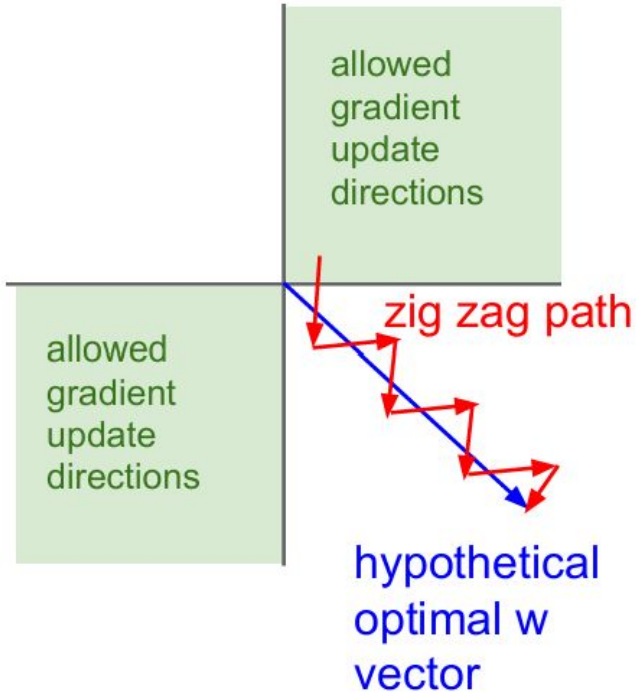
Training Neural Networks

- Training Neural Networks
  - Activation Functions
    - Sigmoid
      - Zero-Centered가 아니고 Gradient Vanishing 문제가 있음



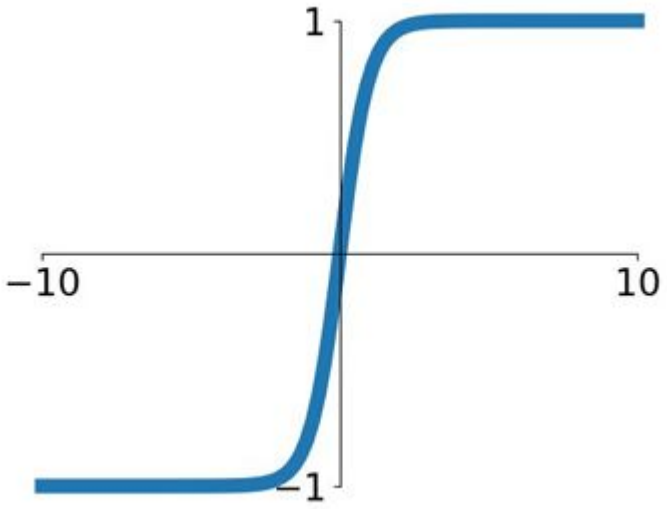
Training Neural Networks

- Training Neural Networks
  - Activation Functions
    - Sigmoid
      - Zero-Centered가 아닐 때의 문제점
        - ✔ 업데이트 방향이 국한되어 수렴속도가 늦어짐



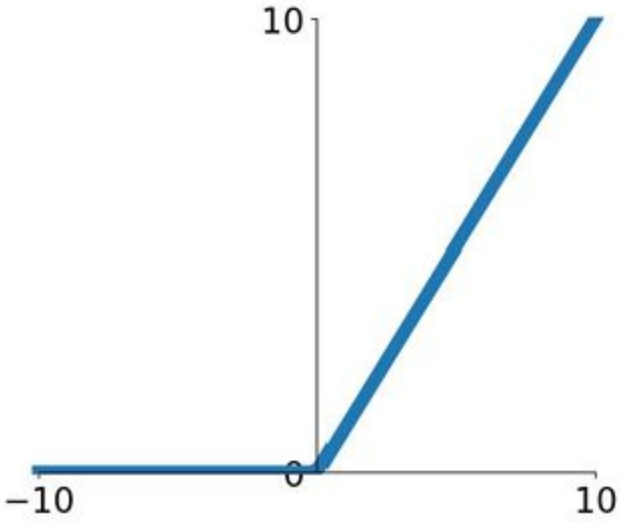
Training Neural Networks

- Training Neural Networks
  - Activation Functions
    - tanh
      - Zero-centered지만 여전히 Gradient Vanishing 남아있음



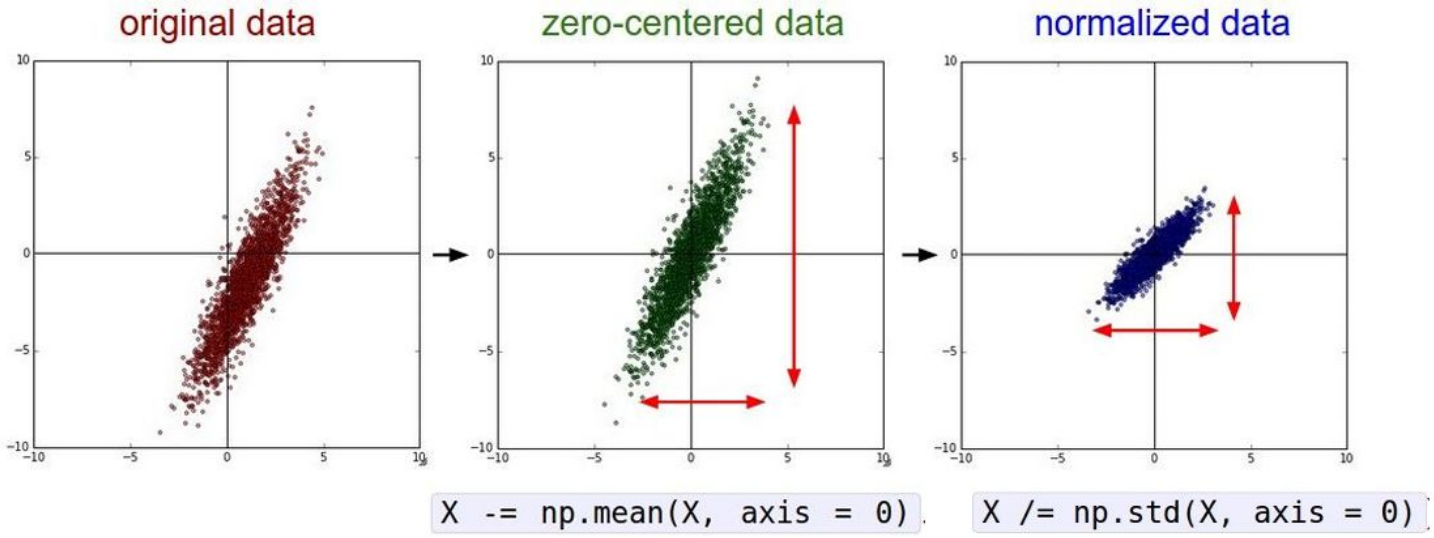
Training Neural Networks

- Training Neural Networks
  - Activation Functions
    - ReLU
      - Gradient Vanishing해결 및 연산 단순화
        - ✔ sigmoid와 tanh에 비해 max의 연산은 단순함



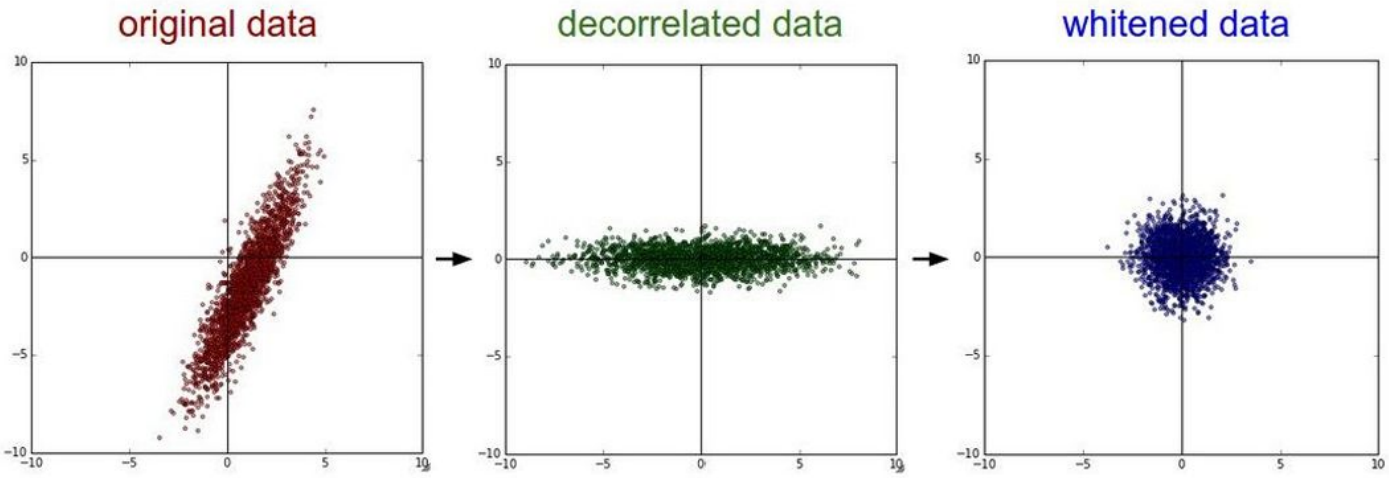
Training Neural Networks

- Training Neural Networks
  - Data Preprocessing
    - zero-centered: 데이터의 평균을 0으로 맞춤
      - 입력 값이 한 쪽으로 치우치지 않도록 하기 위함
    - normalized: 데이터를 일정한 범위 또는 분산으로 맞춤
      - feature간 스케일을 통일하기 위함



Training Neural Networks

- Training Neural Networks
  - Data Preprocessing
    - decorrelated(PCA): 데이터의 주요 축을 따라 차원 축소
      - 데이터의 분산을 최대화하는 주성분을 찾아 중요한 feature 추출
    - whitened: decorrelated data를 각 주성분의 분산을 동일하게 맞추는 과정
      - feature들간의 독립성과 균형을 만들기 위함

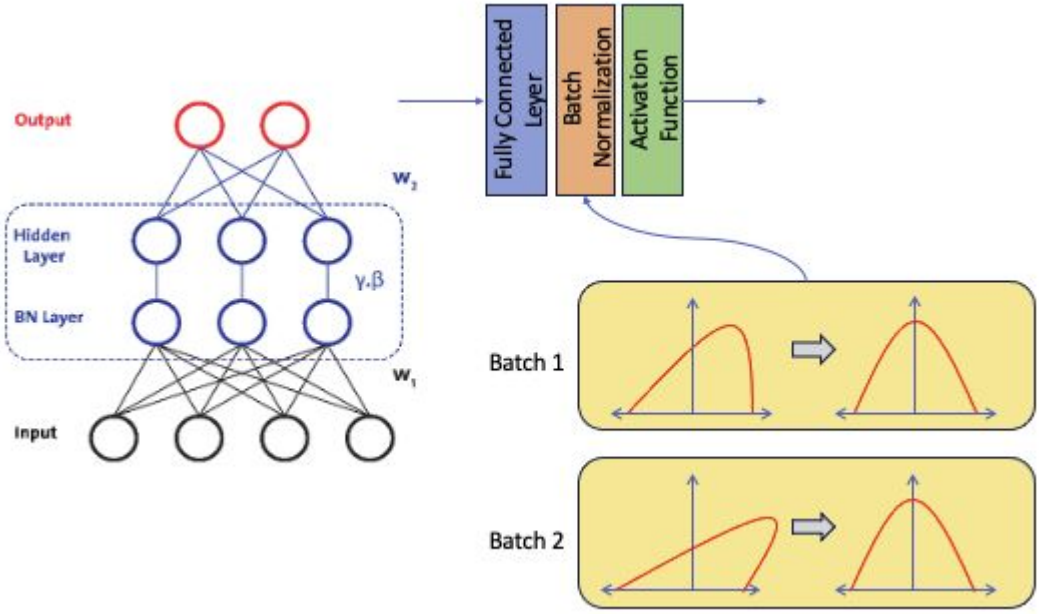


Training Neural Networks

- Training Neural Networks
  - Weight Initialization
    - 초기값이 너무 작으면, 역전파시 gradient가 점점 0으로 수렴
    - 초기값이 너무 크면, gradient가 폭발
    - 모든 weight를 0으로 초기화 하면, 모든 뉴런이 동일하게 학습
    - 초기값이 부적절하면, optimizer가 좋은 방향을 찾기 어려움
      - Xavier initialization, He initialization

Training Neural Networks

- Training Neural Networks
  - Batch Normalization
    - Internal Covariate Shift: 학습이 진행되면서 앞쪽 weight가 바뀌면 그에 따라 뒷 레이어로 전달되는 값들의 분포도 계속 바뀜
      - ⁘문제점: 데이터의 분포가 layer마다 달라지면 일관적인 학습을 하지 못해 효율이 떨어짐



Training Neural Networks

- Training Neural Networks
  - Batch Normalization
    - 데이터를 각 배치 별로 평균과 분산을 이용해 정규화 하는 것
    - γ(scaling)과 β(shifting)를 파라미터로 두고 이를 학습시킴

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Code Review

- Code Review
  - 주요 코드 분석 - CNN cifar10
    - x.view(x.size(0), -1)
      - CNN의 4D 출력을 2D로 평탄화
    - x = self.classifier(x)
      - 평탄화된 CNN의 출력 벡터를 MLP classfier에 넣어 최종 분류

```
class CNN(nn.Module):

    def __init__(self, model_code, in_channels, out_dim, act, use_bn):
      super().__init__()

      if act == 'relu':
        self.act = nn.ReLU()
      elif act == 'tanh':
        self.act = nn.Tanh()
      elif act == 'sigmoid':
        self.act = nn.Sigmoid()
      else:
        raise ValueError("Not a valid activation function code")

      self.layers = self._make_layers(model_code, in_channels, use_bn)
      self.classifier = nn.Sequential(nn.Linear(512,256),
                                      self.act,
                                      nn.Linear(256,out_dim))

    def forward(self, x):
      x = self.layers(x)
      x = x.view(x.size(0), -1)
      x = self.classifier(x)
      return x
```

Code Review

- Code Review
  - 주요 코드 분석 - CNN cifar10
    - net.cuda()
      - CNN 네트워크를 GPU에 올림
    - if optim문
      - 여러 Optimizer 중 선택해서 사용하기 위함

```
def experiment(partition, args):

    net = CNN(model_code = args.model_code,
              in_channels = args.in_channels,
              out_dim = args.out_dim,
              act = args.act,
              use_bn = args.use_bn)
    net.cuda()

    criterion = nn.CrossEntropyLoss()
    if args.optim == 'SGD':
        optimizer = optim.SGD(net.parameters(), lr=args.lr, weight_decay=args.l2)
    elif args.optim == 'RMSprop':
        optimizer = optim.RMSprop(net.parameters(), lr=args.lr, weight_decay=args.l2)
    elif args.optim == 'Adam':
        optimizer = optim.Adam(net.parameters(), lr=args.lr, weight_decay=args.l2)
    else:
        raise ValueError('In-valid optimizer choice')
```

Code Review

- Code Review
  - 주요 코드 분석 - CNN cifar10
    - argparse.ArgumentParser()
      - 코드를 수정하지 않고 하이퍼파라미터를 지정해주기 위함

```
# ====== Random Seed Initialization ====== #
seed = 123
np.random.seed(seed)
torch.manual_seed(seed)

parser = argparse.ArgumentParser()
args = parser.parse_args("")
args.exp_name = "exp1_lr_model_code"

# ====== Model ====== #
args.model_code = 'VGG11'
args.in_channels = 3
args.out_dim = 10
args.act = 'relu'

# ====== Regularization ======= #
args.l2 = 0.0001
args.use_bn = True

# ====== Optimizer & Training ====== #
args.optim = 'RMSprop' #'RMSprop' #SGD, RMSprop, ADAM...
args.epoch = 40
args.train_batch_size = 64
args.test_batch_size = 1024
```

Code Review

- Code Review
  - 실험 과정 - CNN cifar10(SGD)
    - Optimizer를 SGD로 설정
    - learning rate를 0.1, 0.01, 0.001로 실험
      - 0.1에서 가장 높은 성능을 보임
    - learning rate를 0.1로 고정 후 epoch을 20에서 30, 40으로 점차 높여봄
      - epoch을 늘릴수록 성능이 높아짐
    - momentum을 추가하여 실험 진행
      - 오히려 성능 저하, 하지만 epoch을 늘렸을 때 최고 성능은 더 높았음
    - epoch을 40으로 해도 train loss가 남아있어 lr scheduling 적용
      - train loss가 0이 됨을 확인하였고, 성능도 더 높아짐
    - 순수 SGD는 epoch에 따른 변화가 미미함을 관찰

Code Review

- Code Review
  - 실험 결과 - CNN cifar10(SGD)

model	act	L2 reg	bn	optim	epoch	lr	moment um	test_ac c	lr scheduling
VGG11	ReLU	0.0001	True	SGD	20	0.1	x	82.07	x
VGG11	ReLU	0.0001	True	SGD	20	0.01	x	78.1	x
VGG11	ReLU	0.0001	True	SGD	20	0.001	x	76.78	x
VGG11	ReLU	0.0001	True	SGD	30	0.1	x	82.21	x
VGG11	ReLU	0.0001	True	SGD	40	0.1	x	82.23	x
VGG11	ReLU	0.0001	True	SGD	20	0.1	0.9	79.06	x
VGG11	ReLU	0.0001	True	SGD	30	0.1	0.9	77.86	x
VGG11	ReLU	0.0001	True	SGD	40	0.1	0.9	83.97	o

Code Review

- Code Review
  - 실험 과정 - CNN cifar10(RMSProp)
    - Optimizer를 RMSProp으로 설정
    - learning rate를 0.1, 0.01, 0.001로 실험
      - 0.001에서 가장 높은 성능을 보임 -SGD와의 차이
    - learning rate를 0.001로 고정 후 epoch을 20에서 30, 40으로 점차 높여봄
      - epoch을 늘릴수록 성능이 높아짐
    - epoch을 40으로 해도 train loss가 남아있어 lr scheduling 적용
      - train loss가 0이 됨을 확인하였고, 성능도 더 높아짐

Code Review

- Code Review
  - 실험 결과 - CNN cifar10(RMSProp)

model	act	L2 reg	bn	optim	epoch	lr	test_acc	lr scheduling
VGG11	ReLU	0.0001	True	RMS	20	0.1	10.0	x
VGG11	ReLU	0.0001	True	RMS	20	0.01	69.75	x
VGG11	ReLU	0.0001	True	RMS	20	0.001	79.68	x
VGG11	ReLU	0.0001	True	RMS	30	0.001	79.35	x
VGG11	ReLU	0.0001	True	RMS	40	0.001	79.77	x
VGG11	ReLU	0.0001	True	RMS	40	0.001	84.12	o

Code Review

- Code Review
  - 실험 과정 - CNN cifar10(Adam)
    - Optimizer를 Adam으로 설정
    - learning rate를 0.1, 0.01, 0.001로 실험
      - 0.001에서 가장 높은 성능을 보임
    - learning rate를 0.001로 고정 후 epoch을 20에서 30, 40으로 점차 높여봄
      - epoch을 늘릴수록 성능이 높아짐
    - epoch을 40으로 해도 train loss가 남아있어 lr scheduling 적용
      - train loss가 0이 됨을 확인하였고, 성능도 더 높아짐

Code Review

- Code Review
  - 실험 결과 - CNN cifar10(Adam)

model	act	L2 reg	bn	optim	epoch	lr	test_acc	lr scheduling
VGG11	ReLU	0.0001	True	Adam	20	0.1	10.0	x
VGG11	ReLU	0.0001	True	Adam	20	0.01	69.22	x
VGG11	ReLU	0.0001	True	Adam	20	0.001	79.37	x
VGG11	ReLU	0.0001	True	Adam	30	0.001	80.28	x
VGG11	ReLU	0.0001	True	Adam	40	0.001	81.04	x
VGG11	ReLU	0.0001	True	Adam	40	0.001	84.21	o

Code Review

- Code Review
  - 실험 과정 - CNN cifar10
    - 동일한 모델에서 세 optimizer의 최고 성능이 약 84%로 거의 비슷한 성능을 보임
    - 모델의 한계로 판단하고 VGG 13, VGG 16, VGG 19와 비교
    - 다른 하이퍼파라미터들은 VGG 11실험의 최고성능에서 고정
    - Adam이 가장 발전된 optimizer로 판단되어 Adam만 시행

Code Review

- Code Review
  - 실험 결과 - CNN cifar10(Adam)
    - VGG 13, VGG 16, VGG 19

model	act	L2 reg	bn	optim	epoch	lr	test_acc	lr scheduling
VGG11	ReLU	0.0001	True	Adam	40	0.001	84.21	o
VGG13	ReLU	0.0001	True	Adam	40	0.001	86.61	o
VGG16	ReLU	0.0001	True	Adam	40	0.001	86.23	o
VGG19	ReLU	0.0001	True	Adam	40	0.001	86.77	o
VGG11	ReLU	0.0001	False	Adam	40	0.001	75.45	o

Code Review

- Code Review
  - 실험 결과 분석
    - SGD는 단점이 많고 초기에 Optimizer라 성능이 안 좋을 줄 알았으나, task가 간단해서 그런지 RMSProp이나 Adam 성능에 차이가 없었음
    - Optimizer에 따라 좋은 learning rate가 다르다는 것을 알 수 있었음. SGD는 0.1에서 좋은 성능을 보였고 RMSProp과 Adam은 0.001에서 좋은 성능을 보임
    - 단순히 epoch만 늘려서는 train loss가 0이 되지 않았고 scheduling으로 이를 해결할 수 있었음
    - VGG11과 VGG19의 epoch당 train time은 2배 이상으로 상당히 많이 차이 남

Conclusion

- 느낀 점
  - 딥러닝을 공부하면서 이 분야는 단순 암기나 수동적인 수용이 아닌 개념 하나하나에 대해 의문을 품고, 논리적으로 비판하는 사고력이 필요하다는 것을 느낌. 이론과 코드를 처음 배울 때 막막했지만, 실제 코드로 실험을 진행하며 추상적인 개념들이 조금 구체화되는 느낌을 받음.

- 향후 계획
  - 학부졸업까지 꽤 남았지만, '시간이 많으니 천천히 하자' 가 아닌 '빠르게 적응하고 성장하여 나의 경쟁력을 올리자'라는 마인드 장착.
  - 올해 목표는 국내 논문 1편 제출.