26.01.29

Standalone - DeepLearning



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



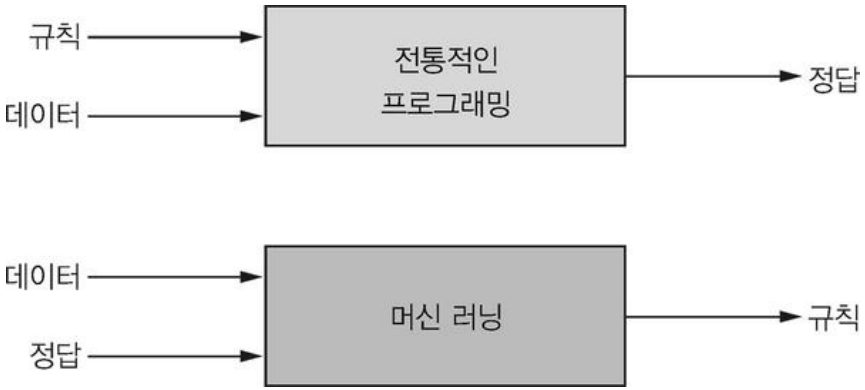
Presented By
조범권

Outline

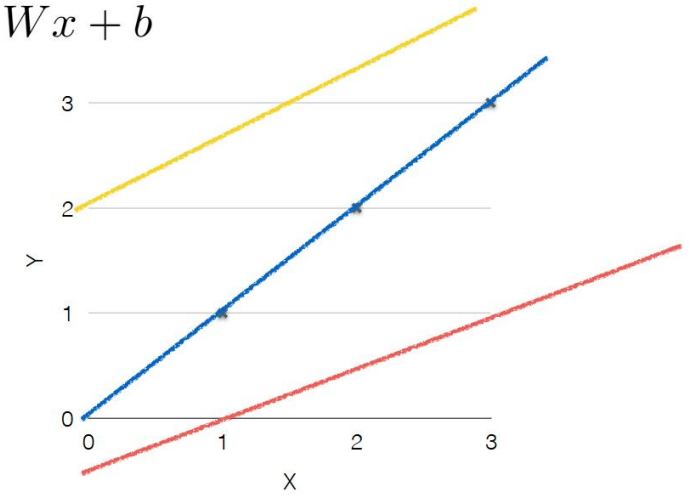
- Introduction
- The Hurdles of Deep Learning
- CNN Architectures
- Transformer
- Assignment#4

Introduction

- **Deep Learning**
  - 규칙을 코딩하지 않고, 데이터와 정답을 주고 규칙을 찾기 위함
    - 입력 x와 정답 y가 주어질 때, 이것들을 가장 잘 표현하는 가설을 찾는 것

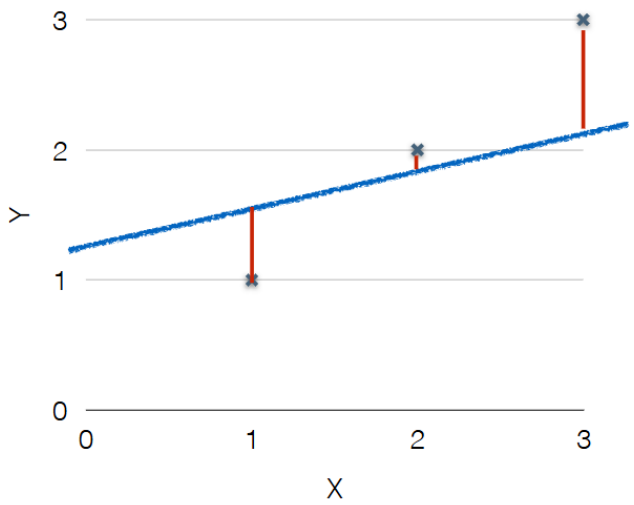


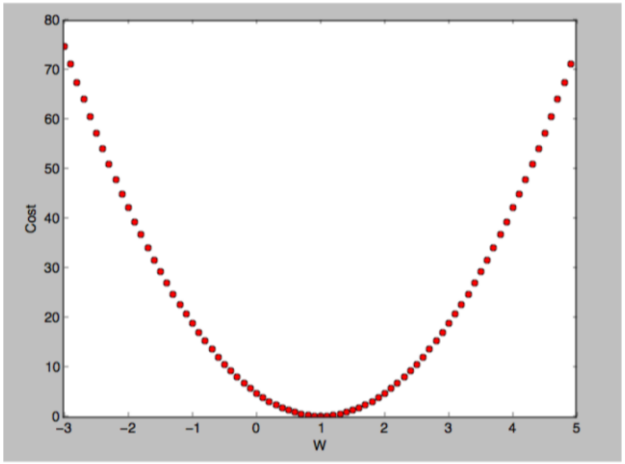
$$H(x) = Wx + b$$



Introduction

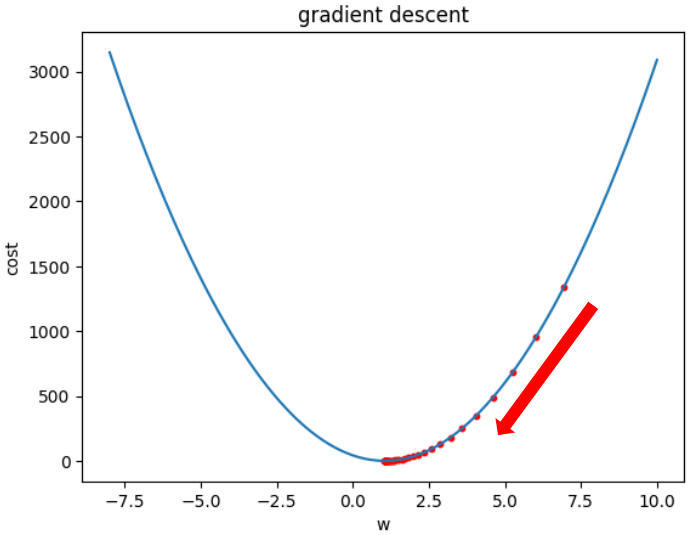
- **Cost Function**
  - 최적의 가설을 찾기 위한 판단 기준
    - 예측값과 실제값의 차이를 수치화
    - 최적 Hypothesis = 최소 Cost
      - ex) MeanSquareError: $cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x_i) - y_i)^2$





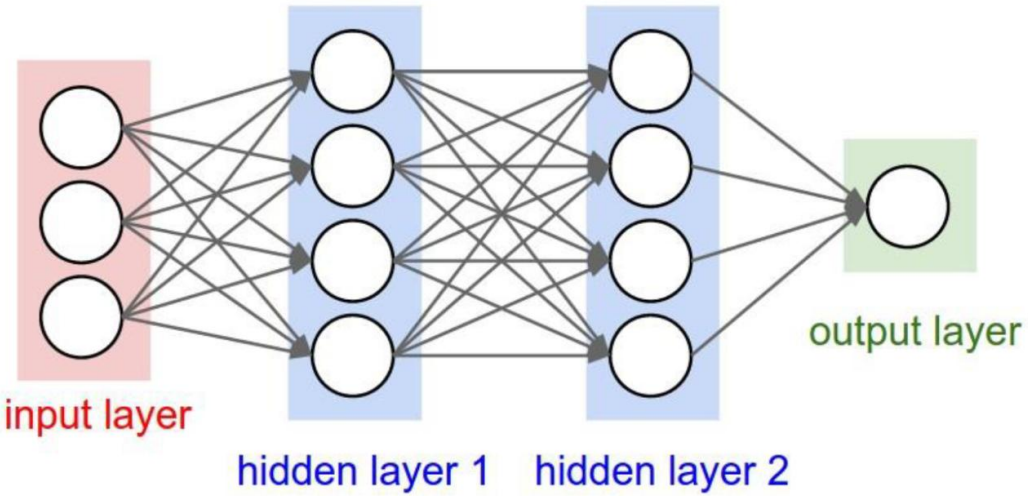
Introduction

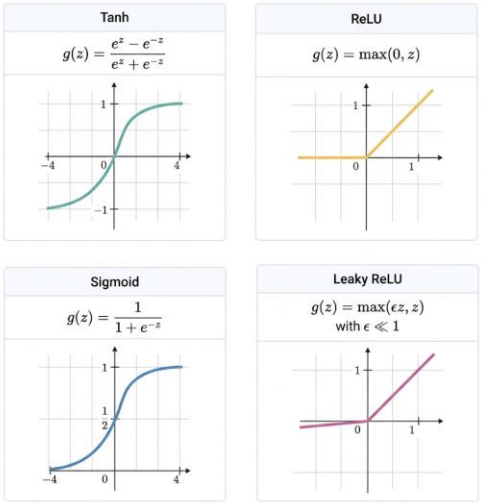
- **Gradient Descent**
  - 최소 Cost를 찾는 방법
    - Cost Function의 경사를 따라 내려가서 최소 Cost인 지점을 찾음
      - 무작위 시작점에서 출발
      - 그 지점에서 Cost Function의 미분값에 Step size 곱해서 이동
        - ✓ $W := W - \alpha \frac{\partial}{\partial x} cost(W) = W - \alpha \frac{1}{m} \sum_{i=1}^{m} (Wx_i - y_i)x_i$
      - Minimum에 수렴할 때 까지 반복



Introduction

- **Multilayer Perceptron (MLP)**
  - Input / Output Layer 사이에 1개 이상의 Hidden Layer가 있는 구조
    - Activation Function을 통해 비선형성을 부여, 복잡한 예측을 가능하게 함
    - 그러나 초반에는 구조가 너무 복잡해 gradient 계산 불가능 하다고 판단

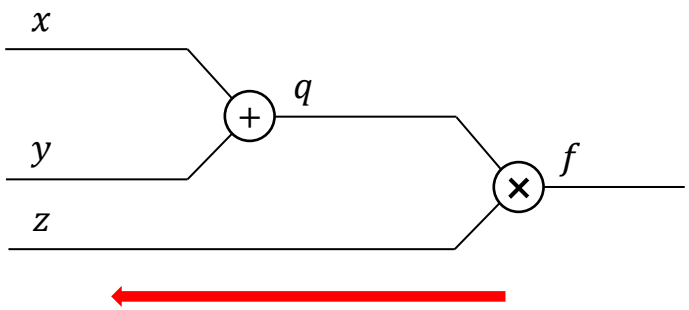




Introduction

- **Backpropagation**
  - Multi-layer에서 각 층의 gradient를 계산하는 방법
    - 예측값과 실제값의 loss와 gradient를 계산
    - Chain rule을 이용하여, Backward로 각 파라미터의 gradient 계산

간단한 예시



$$f(x,y,z) = qz = (x+y)z$$

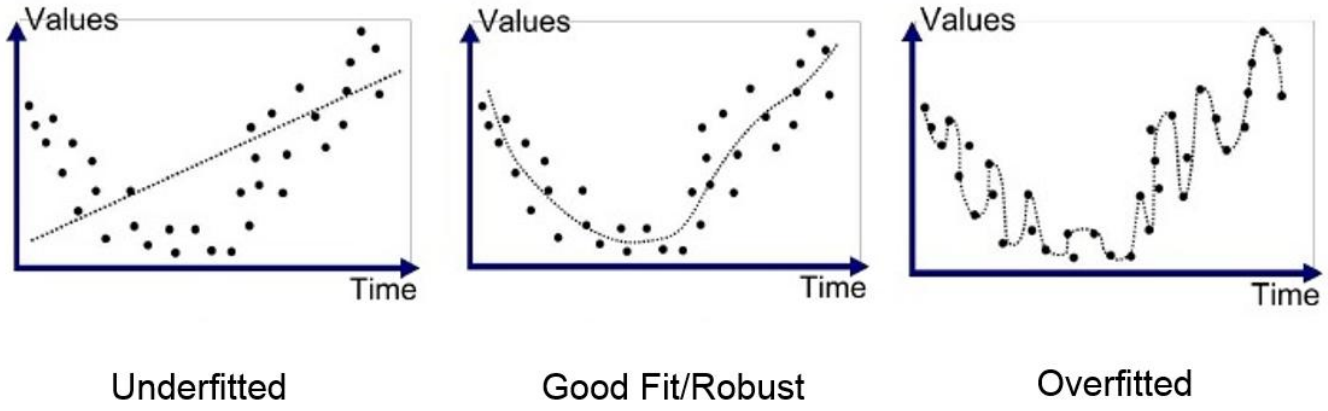
$$\frac{\partial f}{\partial q} = \mathrm{z}\,, \frac{\partial f}{\partial \mathrm{z}} = \mathrm{q}$$

$$\frac{\partial \mathrm{q}}{\partial \mathrm{x}} = 1\,, \frac{\partial \mathrm{q}}{\partial \mathrm{y}} = 1$$

$$\frac{\partial f}{\partial \mathrm{x}} = \frac{\partial f}{\partial \mathrm{q}}\frac{\partial \mathrm{q}}{\partial \mathrm{x}} = \mathrm{z}\,, \frac{\partial f}{\partial \mathrm{y}} = \frac{\partial f}{\partial \mathrm{q}}\frac{\partial \mathrm{q}}{\partial \mathrm{y}} = \mathrm{z}$$

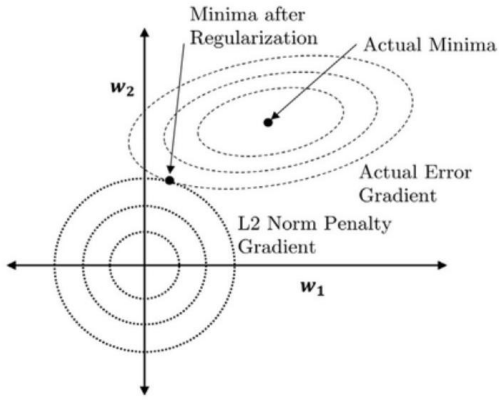
The Hurdles of Deep Learning

- **Overfitting**
  - 모델이 학습 데이터를 과도하게 학습하여 실제 데이터에는 맞지 않는 문제
    - Model Capacity가 늘어날 수록, 더 복잡한 예측이 가능
    - 이로인해 학습 데이터의 노이즈마저 학습해버림

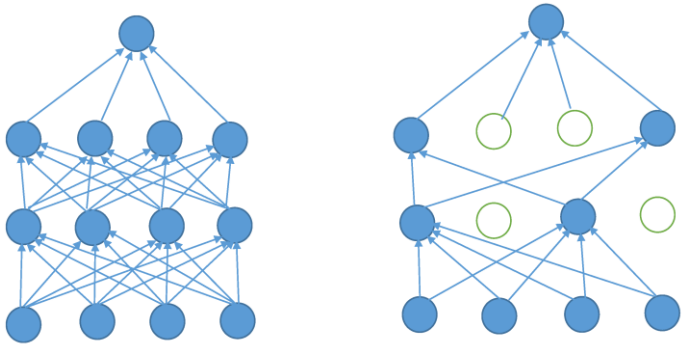


The Hurdles of Deep Learning

- **Overfitting 억제**
  - **L2 Regularization**
    - Loss에 weights의 크기 제곱에 비례하는 페널티항 추가
      - $Loss_{L2}(W) = Loss(W) + \alpha \cdot \|W\|_2^2$
        - ✓ W: 모델의 weights 벡터 (파라미터들)
        - ✓ $\|W\|_2^2$: weight들의 제곱합
        - ✓ α: 규제 강도를 조절하는 하이퍼파라미터
      - 이를 통해 weight들의 크기를 억제하여 노이즈에 둔감해짐

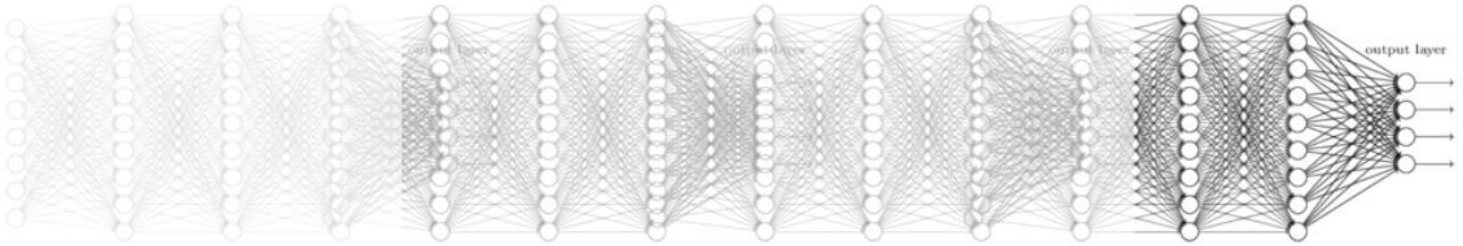


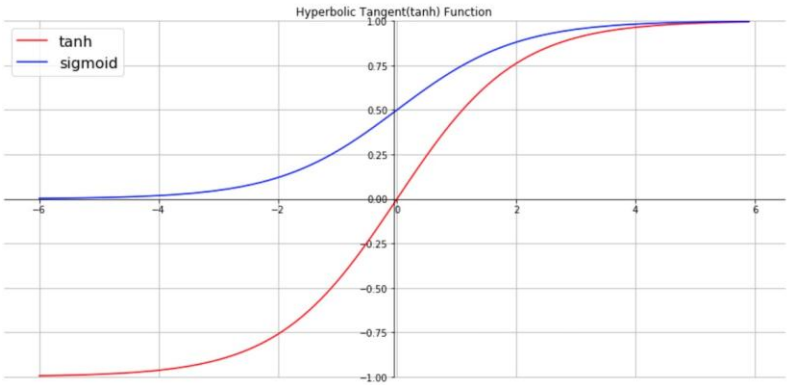
  - **Dropout**
    - 학습할 때 p의 확률로 뉴런을 비활성화
      - 모델이 특정 뉴런에 과의존하는 문제를 방지
        - ✓ 각 뉴런을 효율적으로 이용
        - ✓ 다양한 패턴을 학습 (앙상블과 유사)

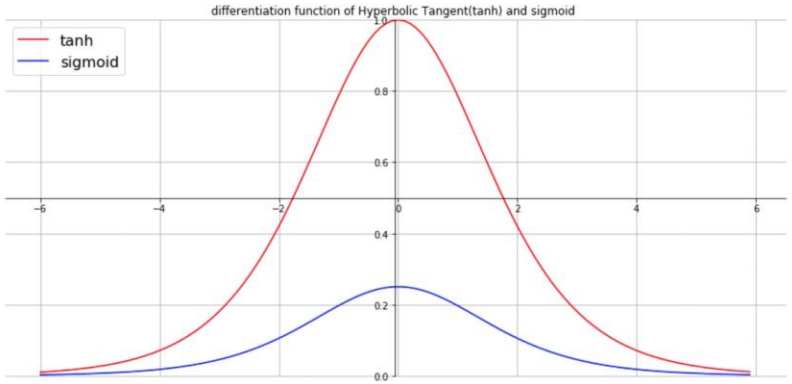


The Hurdles of Deep Learning

- **Gradient Vanishing**
  - Backpropagation 과정에서 입력층으로 갈 수록, Gradient가 매우 작아지는 현상
    - Activation Function으로 Sigmoid 계열 (logistic, tanh)을 사용할 때
      - 미분 값이 1이하, 양 끝단에선 미분 값이 거의 0
      - Backpropagation 과정에서 1보다 작은 값들이 계속 곱해져서 Gradient가 작아짐

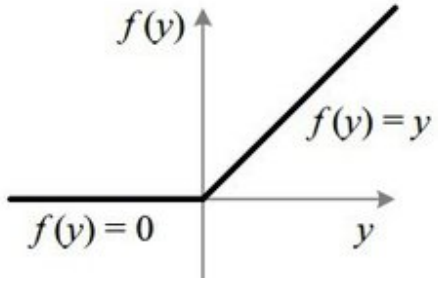




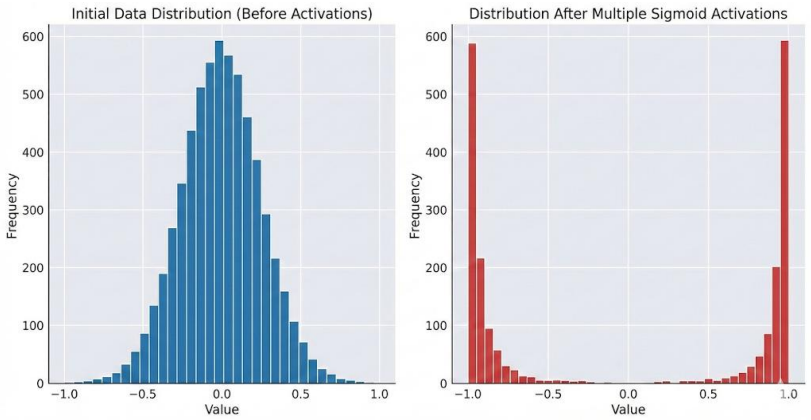


The Hurdles of Deep Learning

- **Gradient Vanishing 방지**
  - **ReLU**
    - input이 커져도 미분 값이 1
      - Gradient를 보다 잘 보존할 수 있음

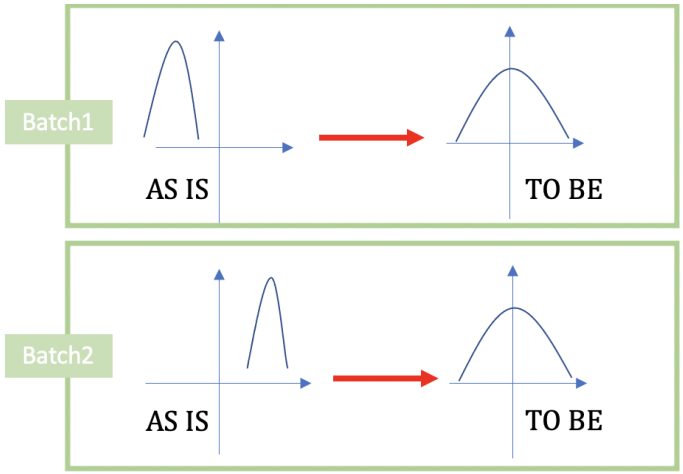


  - **Xavier Initialization**
    - 특정 분산을 갖는 분포에서 무작위로 샘플링하여 가중치를 초기화
    - 입력과 출력의 분산을 일정하게 유지
      - Activation function으로 Sigmoid 계열을 사용할 때,
        층이 깊어질수록 분포가 양 극단으로 치우치고 Gradient Vanishing이 심화



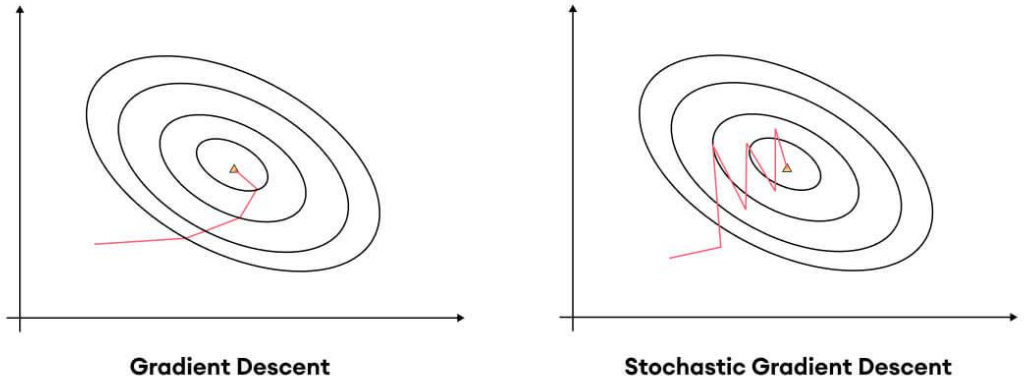
The Hurdles of Deep Learning

- **Gradient Vanishing 방지**
  - **Batch Normalization**
    - Activation function 입력의 분포를 평균 0, 분산 1인 정규분포 형태로 재조정 후 학습 가능한 파라미터로 Scale, Shift
      - Sigmoid 계열에서는 기울기가 살아있는 구간으로 유지
      - ReLU에서는 Activation을 하는 의미가 없는 데이터 분포를 완화
        - ✓ 데이터들이 모두 0보다 작아서 ReLU를 거치면 Gradient가 0이 되는 분포 또는 데이터들이 모두 0보다 커서 ReLU를 거쳐도 아무런 변화가 없는 분포
        - ✓ 이를 통해 비선형성을 잃는 문제 또한 해결



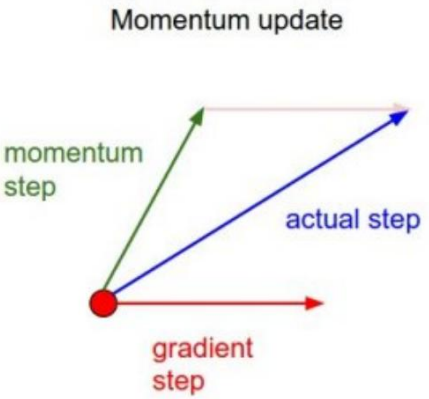
The Hurdles of Deep Learning

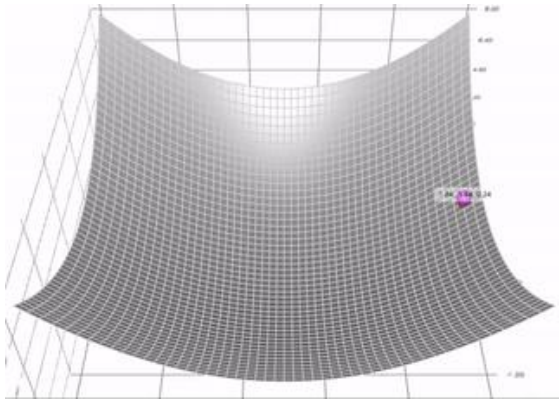
- **Optimizer**
  - **Stochastic Gradient Descent (SGD)**
    - 모든 training dataset에서 무작위로 추출한 일부(mini batch)에 대해 gradient 계산 후 스텝
      - Batch Gradient Descent보다 메모리가 작아도 되고, 스텝 밟는게 빠르며 수렴 속도도 빠름
      - 학습에 노이즈가 많지만 오히려 local minima를 피할 수 있음
    - 그러나 local minima에서 빠져나올 수 없다는 한계가 존재



The Hurdles of Deep Learning

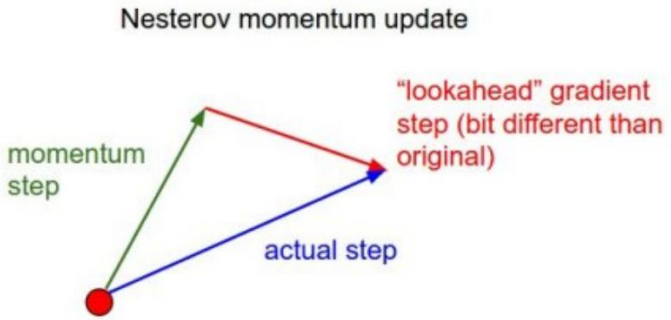
- **Optimizer - Adding Momentum**
  - **Momentum**
    - 관성(momentum)으로 local minima 빠져나오기
    - $\theta = \theta - v_t$
    - $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$
      - v_t: momentum
      - θ: Parameter set of the model
      - η: learning rate
      - γ: decaying constant, v_{t-1}을 억제
    - 불필요한 노이즈도 해결
    - 그러나 global minima에서 멈추지 못함





The Hurdles of Deep Learning

- **Optimizer - Adding Momentum**
  - **Nesterov Accelerated Gradient (NAG)**
    - 관성에 의해 이동할 위치에서 gradient 계산
    - $\theta = \theta - v_t$
    - $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
      - v_t: momentum
      - θ: Parameter set of the model
      - η: learning rate
      - γ: decaying constant, v_{t-1}을 억제



    - minima를 지나가면 속도를 줄일 수 있음
    - 그러나 step size가 모든 파라미터에서 동일하다는 한계가 존재

The Hurdles of Deep Learning

- **Optimizer - Adaptive Learning Rate**
  - **Adaptive Gradient (AdaGrad)**
    - gradient의 제곱을 누적
    - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta J(\theta_t)$
    - $G_t = G_{t-1} + \nabla_\theta J(\theta_t) \odot \nabla_\theta J(\theta_t)$
      - G_t: 과거 gradient의 제곱 누적
      - θ: Parameter set of the model
      - η: learning rate
      - ϵ: 0으로 나누는 것을 방지하기 위한 아주 작은 값
    - 많이 변화한 파라미터는 step size가 많이 줄어들고 조금 변화한 파라미터는 step size가 조금 줄어듦
    - 그러나 G_t가 계속 커져서 step size가 0에 수렴

The Hurdles of Deep Learning

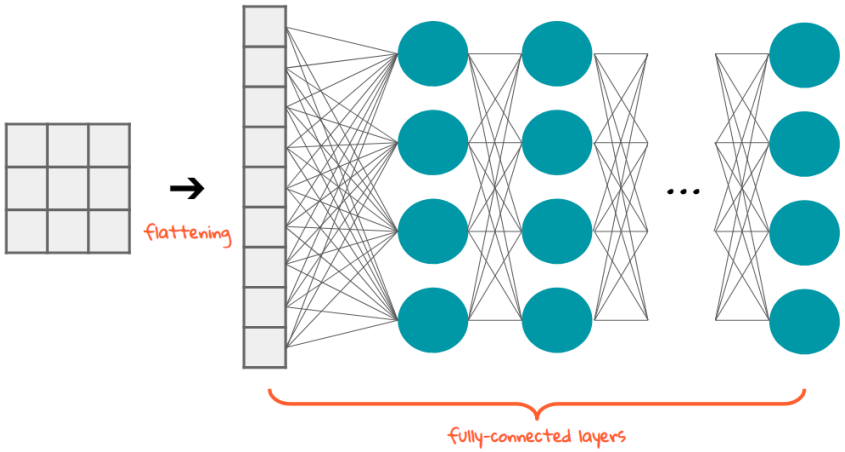
- **Optimizer - Adaptive Learning Rate**
  - **RMSProp**
    - decaying constant를 곱해서 step size가 너무 빨리 0에 수렴하지 않도록
    - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta J(\theta_t)$
    - $G_t = \gamma G_{t-1} + (1-\gamma)\left(\nabla_\theta J(\theta_t)\right)^2$
      - G_t: 과거 gradient의 제곱 누적 (지수적 감쇠)
      - θ: Parameter set of the model
      - η: learning rate
      - ϵ: 0으로 나누는 것을 방지하기 위한 아주 작은 값
      - γ: decaying constant, step size의 수렴 속도를 억제
    - decaying constant는 일반적으로 0.9로 설정

The Hurdles of Deep Learning

- **Optimizer - Combined**
  - **ADAM**
    - Momentum과 Adaptive Learning rate를 모두 적용 (Momentum + RMSProp)
    - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t$
    - $\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$
      - 초기에는 m_t, v_t가 0에 가까워져 학습이 느려짐, 이를 방지하기 위해 보정
    - $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
      - momentum
    - $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
      - adaptive learning rate

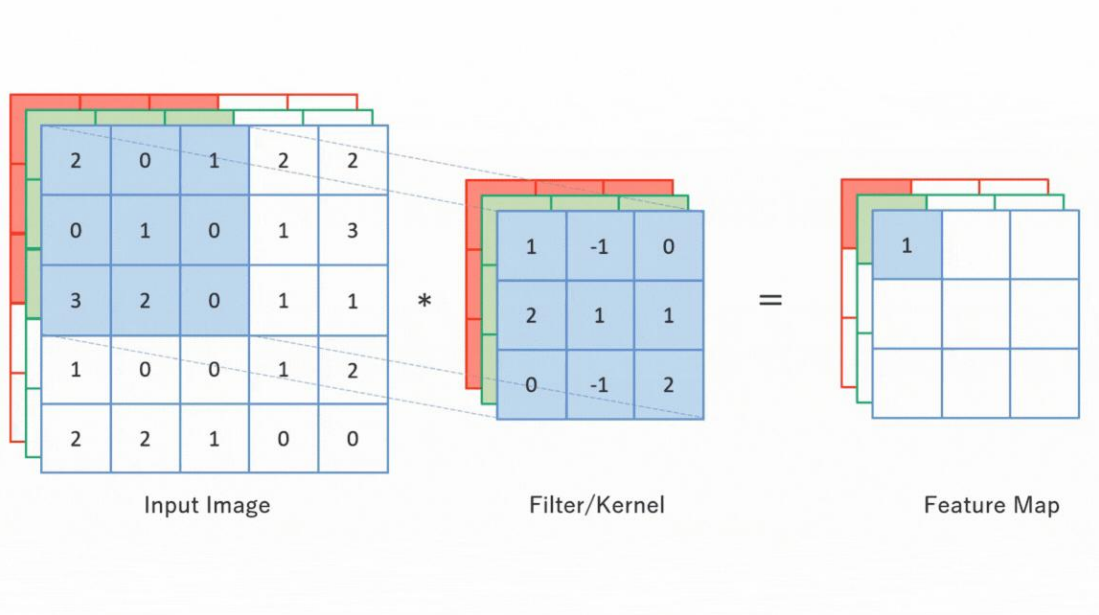
CNN Architectures

- **Problems of MLP**
  - 입력 이미지를 Flatten해서 입력
    - Ex) 3×3 → 9×1
    - 이로 인해 공간적인 정보가 손실됨
      - 가까이 있는 픽셀인지 멀리있는 픽셀인지를 알 수 없음
  - Fully Connected 되어있음
    - 파라미터 수가 매우 많음
      - 일부는 의미가 없는 파라미터
        - ✓입력값을 그대로 출력만 하기도 함
      - 학습이 오래 걸리고, Overfitting을 유발



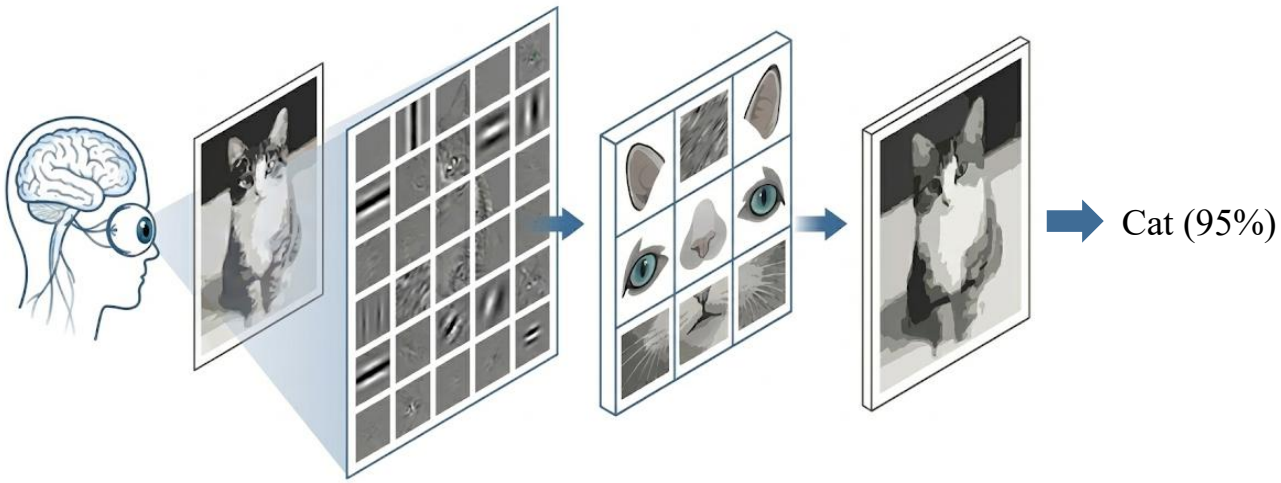
CNN Architectures

- **CNN**
  - **Convolution**
    - Filter를 이미지 위에서 슬라이딩 하며 Spatial Feature를 추출
      - 서로 다른 Weight를 가진 같은 크기의 Filter 여러 개를 적용해서 다양한 Feature를 추출
      - 이를 통해 MLP에 비해 더 적은 수의 파라미터로도 좋은 성능을 낼 수 있음



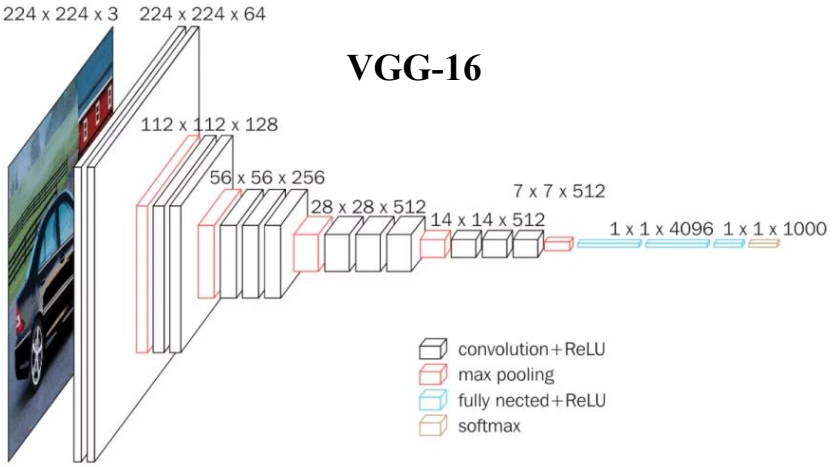
CNN Architectures

- **CNN**
  - 인간이 시각 정보를 처리하는 방식을 모방
    - 얕은 쪽: 비슷한 위치에 있는 물체들끼리 인식 (Low Level Features)
      - ex) 경계, 색상 점, 질감 등
    - 깊은 쪽: Low Level Features를 추상화해서 처리함 (High Level Features)



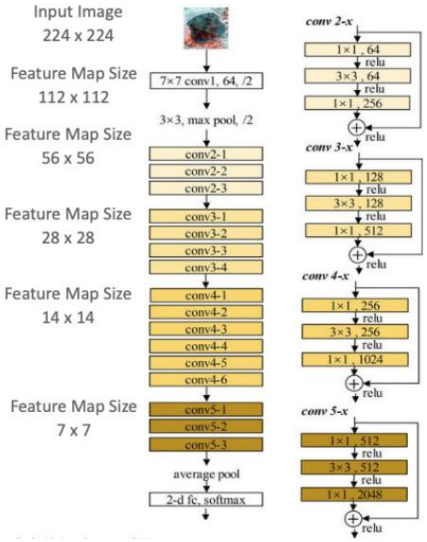
CNN Architectures

- **VGGNet**
  - **VGG-16**: Convolution Layer 13개 + Fully Connected Layer 3개
  - 작은 필터, 깊은 구조
    - 같은 크기의 Receptive field를 얻는 데에 필요한 Parameter가 감소
    - 여러 번의 활성함수를 거치며 비선형성이 추가됨
  - Gradient Vanishing 문제로 19 Layer 보다 깊은 구조 불가능
  - FC Layer에서 많은 파라미터 사용
    - 1.3억 개 이상의 파라미터 중, 약 90% 차지



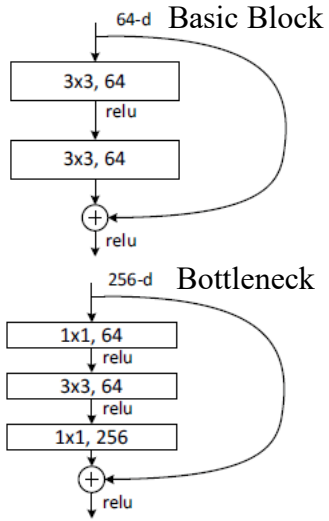
CNN Architectures

- **ResNet**
  - 18 Layer ~ 152 Layer (Fully Connected Layer 1개)
  - 매우 깊은 구조, Residual Block 쌓아놓은 구조
    - 50 Layer 이상의 구조에서는 Bottleneck 이용해서 연산 효율 높임
    - Residual Connection으로 Gradient Vanishing을 해결
  - Global Average Pooling으로 여러 개의 FC Layer를 사용하지 않음



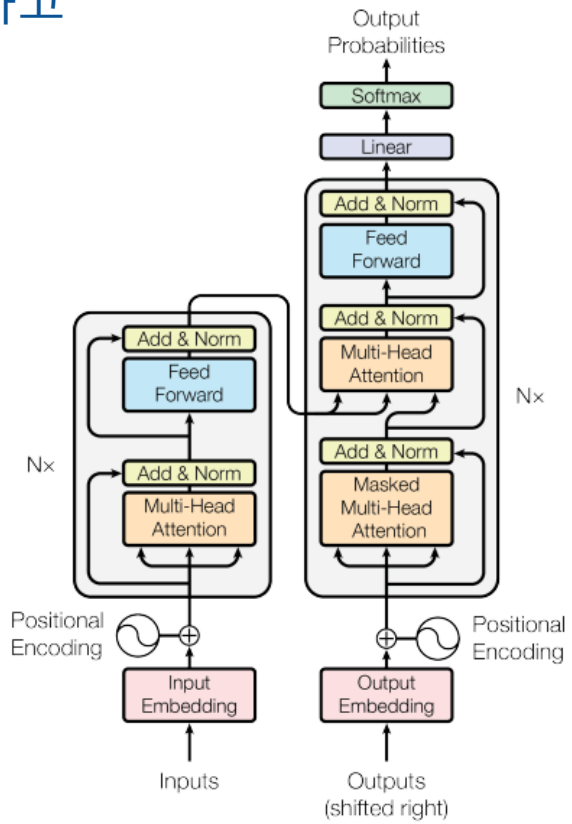
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Residual Block



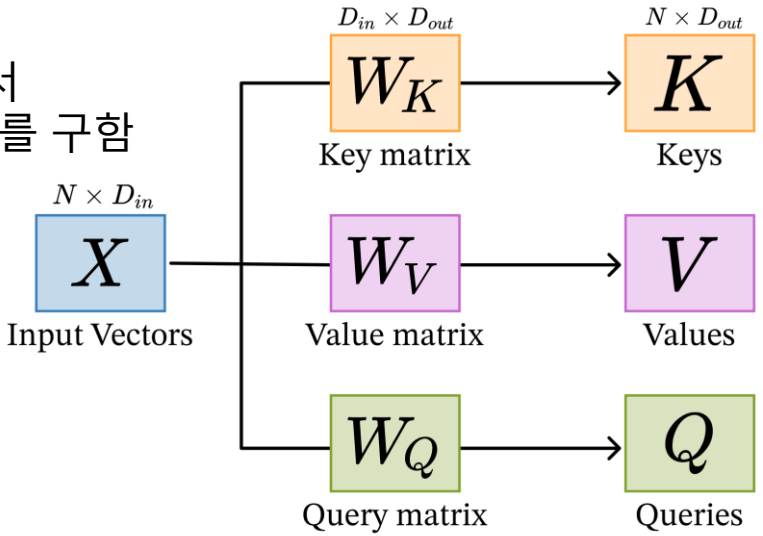
Transformer

- **Background**
  - Attention을 이용하여 입력 토큰 간의 문맥을 파악하고 다음에 올 토큰을 예측하는 모델
    - Token
      - 텍스트를 처리하는 최소 단위
    - Embedding
      - 토큰을 고차원 공간의 벡터로 변환한 것



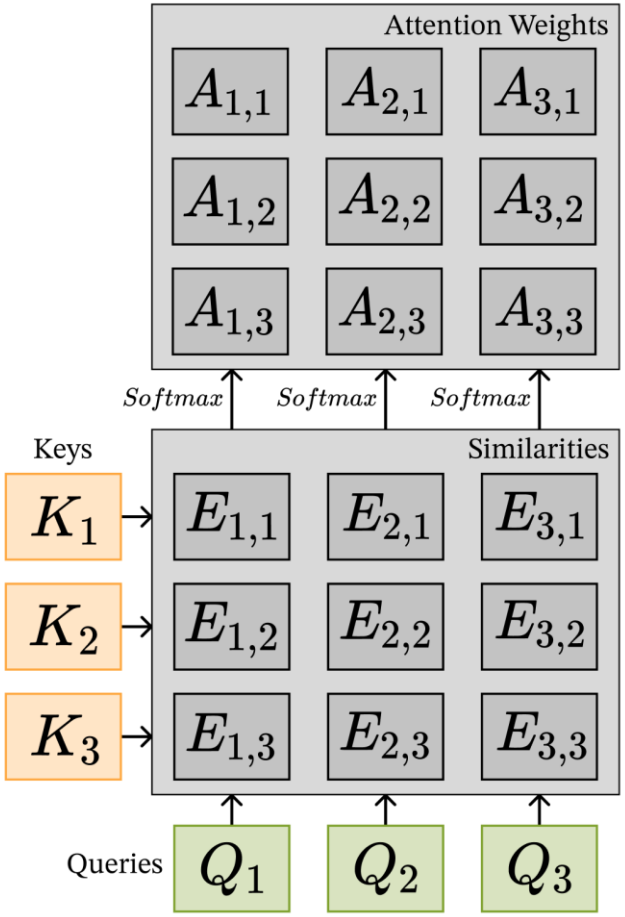
Transformer

- **Attention**
  - Query와 Key의 유사도를 측정하여, 그 점수만큼 Value를 가져오는 과정
    - 검색어와 유사한 인덱스를 가진 내용을 많이
      검색어와 덜 유사한 인덱스를 가진 내용을 적게
    - $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{D_{out}}}\right)V$
  - Input Projection
    - Input Vectors에 각각의 Weight matrix를 곱해서
      Attention 계산에 필요한 Keys, Values, Queries를 구함
      - 인덱스, 내용, 검색어의 역할
      - 각각의 Weight matrix는 학습 가능



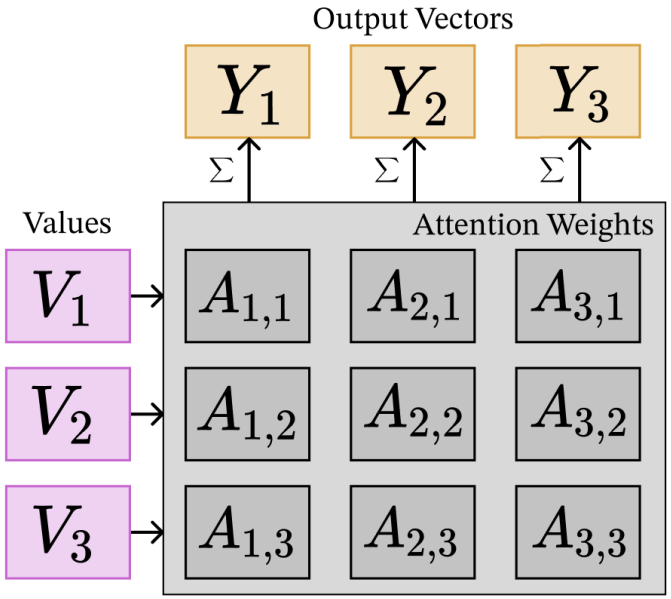
Transformer

- **Self-Attention**
  - Computation
    - Queries와 Keys를 내적 후 $\sqrt{D_{out}}$ 으로 나눠 Similarities(Attention Score)를 구함
      - $E = QK^T/\sqrt{D_{out}}$ $[N \times N]$
      - 차원이 너무 커지면 내적값의 분산이 너무 커져 softmax 함수의 양 극단으로 감 (Gradient Vanishing)
        - ✓이를 방지하기 위해 $\sqrt{D_{out}}$ 으로 나눔
    - 각각을 Softmax에 통과시켜 Attention Weights를 계산



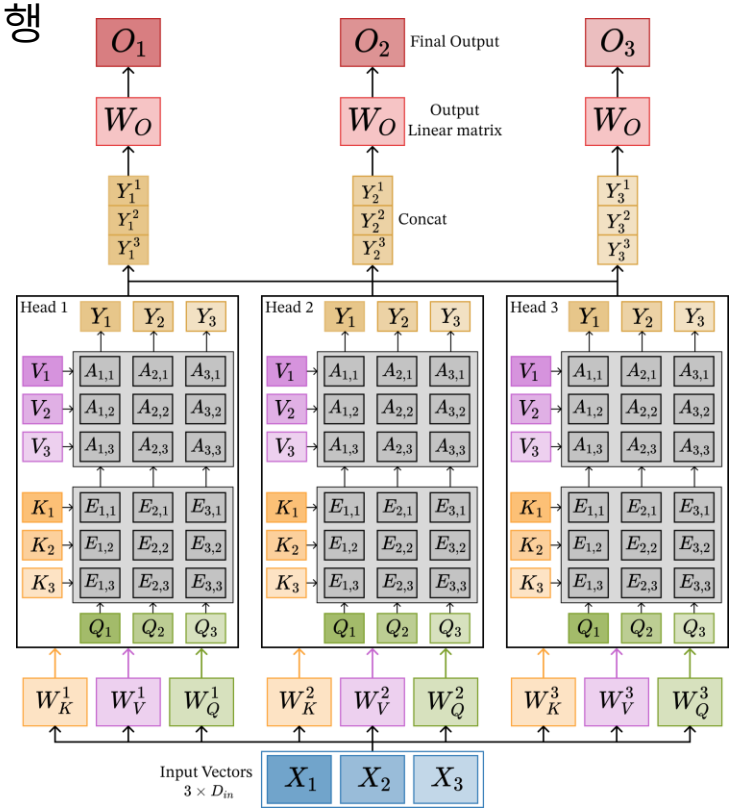
Transformer

- **Self-Attention**
  - Outputs
    - Attention weights에 따라 Values를 Weighted Sum
    - 계산 결과로 Output Vectors를 얻을 수 있음
      - 입력 시퀀스 토큰들의 전체 맥락이 반영된 임베딩
    - FFN 및 Residual Connection, LayerNorm을 거쳐 다음 레이어로 전달 되거나, 최종적으로 예측에 사용



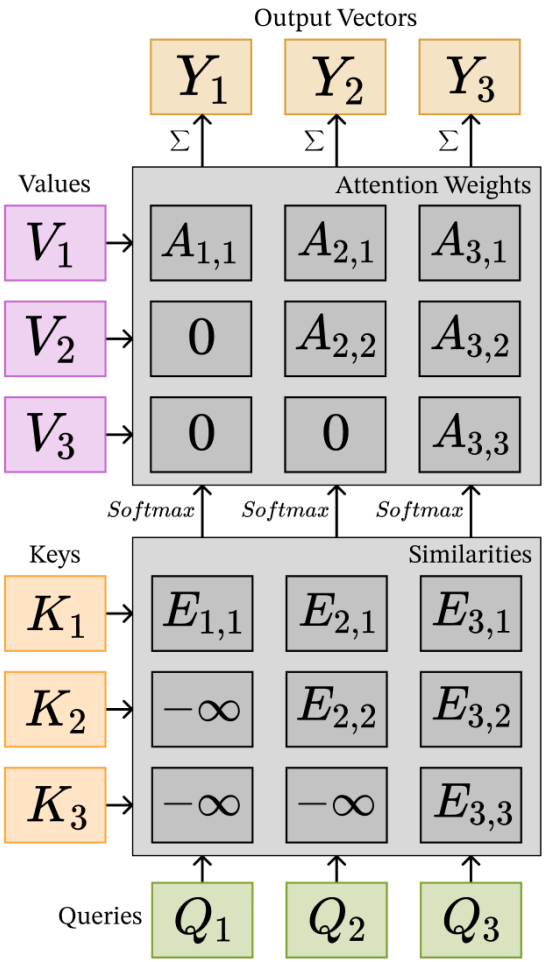
Transformer

- **Multiheaded Self-Attention**
  - Head 개수 만큼 Key matrix, Value matrix, Query matrix를 생성
    - 이를 바탕으로 독립적인 Self-Attention을 병렬 수행
    - 각각의 출력을 concat 한 뒤 W_O와 곱해서 정보를 혼합하고 최종 출력을 얻음
      - W_O는 단일 matrix
    - 다양한 관점(문법, 의미 등)에서 서로 다른 위치의 정보를 동시에 참조 가능



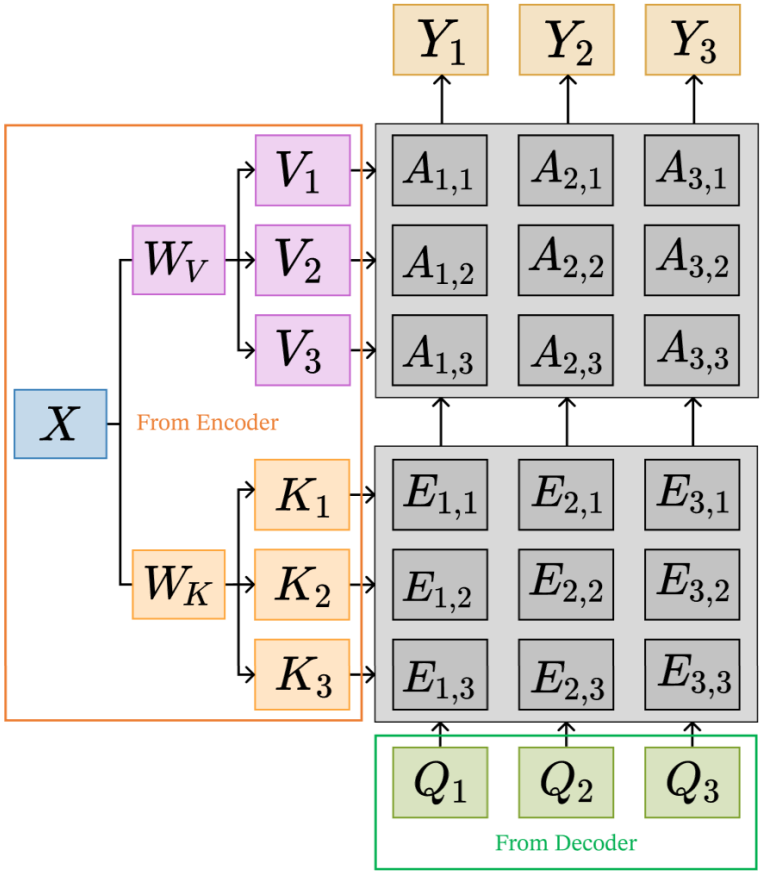
Transformer

- **Masked Self-Attention**
  - 디코더 학습을 위해 마스킹
    - Teacher Forcing
      - 학습 효율을 위해 Target Sequence를 한 번에 입력
    - 마스킹을 하지 않고 Self-Attention을 진행하면 정답을 참조 (자신이 예측할 미래를 참조)
      - 인과관계가 깨져 학습이 제대로 되지 않음
    - Attention Score의 미래 시점 위치에 -∞ 마스킹 적용
      - ex) Q1에서 K2, K3 / Q2에서 K3
      - Softmax 통과 시 해당 위치의 확률이 0이 되어 유출 차단



Transformer

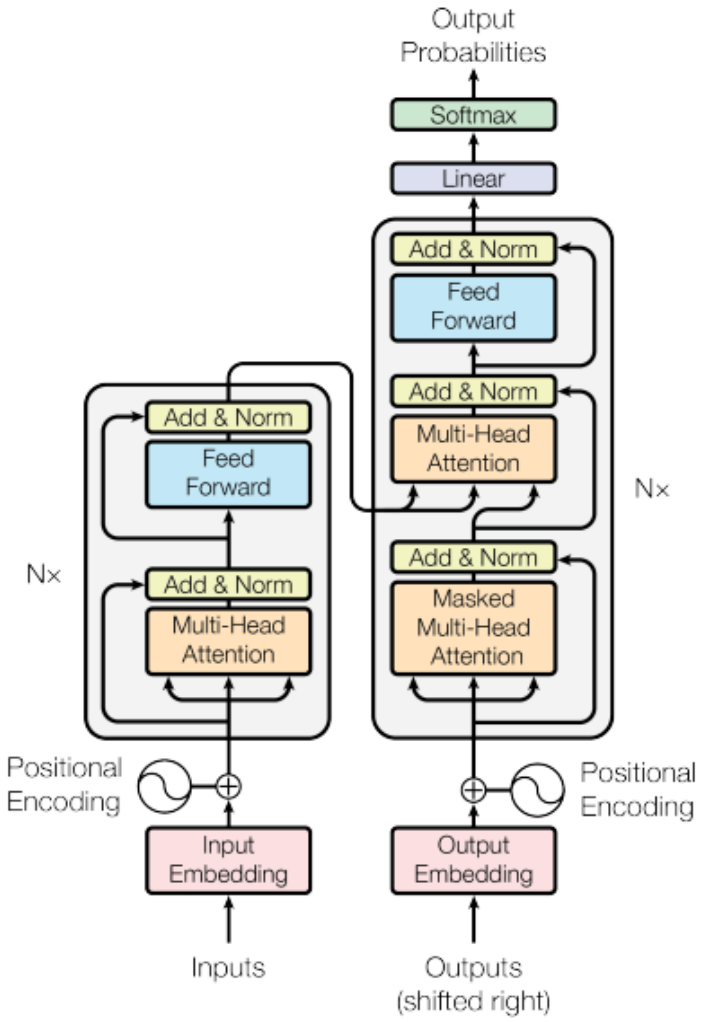
- **Cross-Attention**
  - Encoder의 최종 출력을 Decoder에서 이용
    - 인코더의 출력에 디코더의 W_K, W_V 곱해서 Keys, Values 생성
    - 디코더의 현재 상태에 W_Q 곱해서 Query 생성
    - 디코더가 현재 시점에서 어떤 인코더 정보에 집중해야 하는지 결정



Transformer

- **Feed Forward Neural Network (FFNN)**
  - 각 Position별로 Attention을 통해 얻은 선형적인 결과에 비선형성을 추가

- **Positional Encoding**
  - 각 단어의 임베딩에 위치 정보를 더함

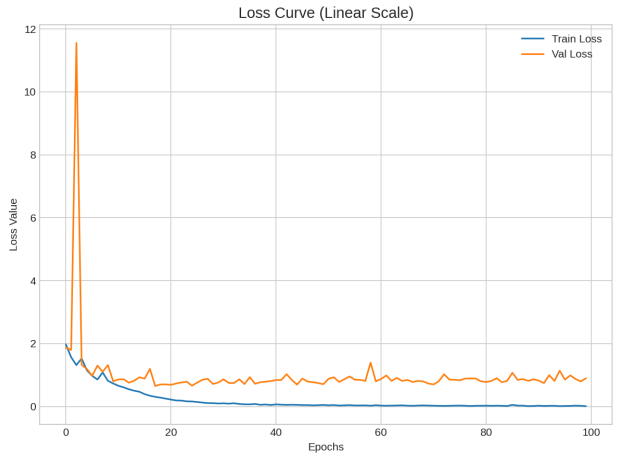


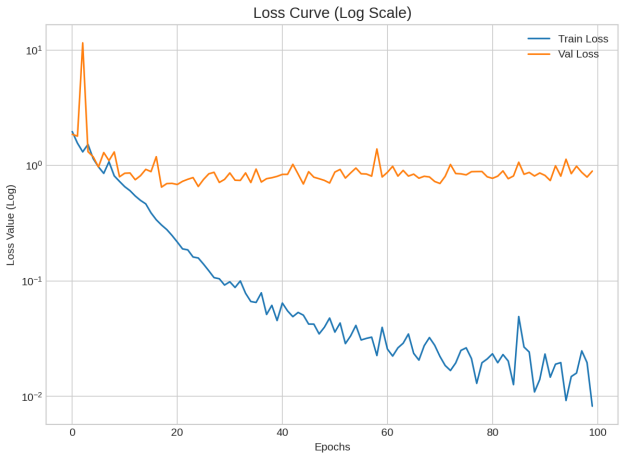
Transformer

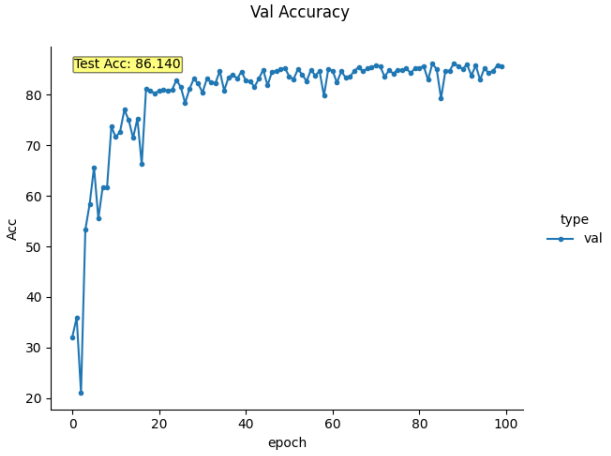
- **Comparison with CNN**
  - CNN
    - 입력이 무엇이든 같은 필터, 가중치 적용
    - 이미지나 텍스트의 지역적 특성을 가정 (작은 크기의 filter 사용)
      - 적은 데이터로도 학습이 잘 됨
  - Transformer (Attention)
    - 입력 데이터에 따라 Attention Weight가 변화
    - 지역성이나 순서에 대한 가정이 거의 없음
      - 데이터가 적을 때는 학습이 어렵지만, 대규모 데이터가 주어지면 CNN보다 더 높은 성능

Assignment #4

- **CIFAR10 with VGGNet - Part 1**
  - exp1~3 (Model Code / Learning rate / L2 / Dropout / Optimizer)
    - Batch size: 256, Epoch: 100
      Activation Function: ReLU, BatchNorm: True
    - Model Code: VGG19 / Learning rate 1e-3 / L2: 0 / Dropout: 0 / Optimizer: Adam
      에서 가장 높은 Test Acc 86.14% 확인

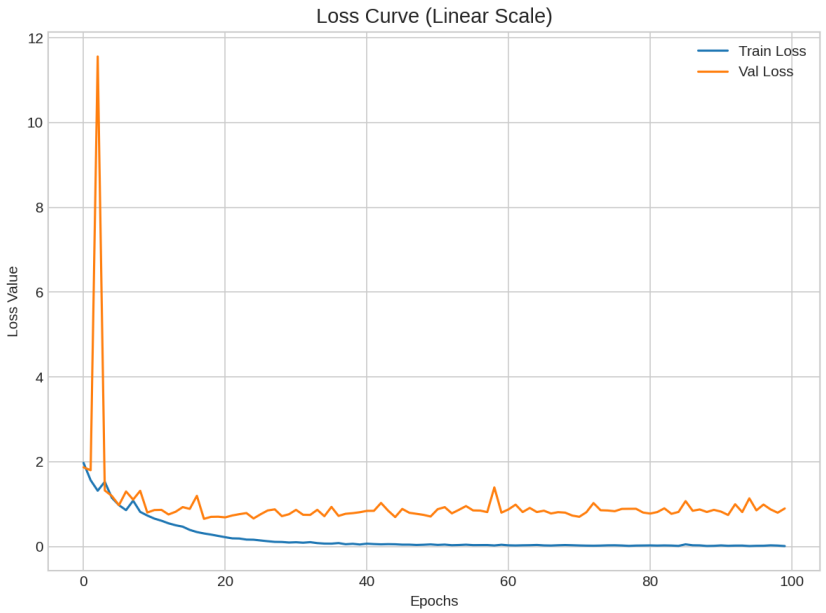


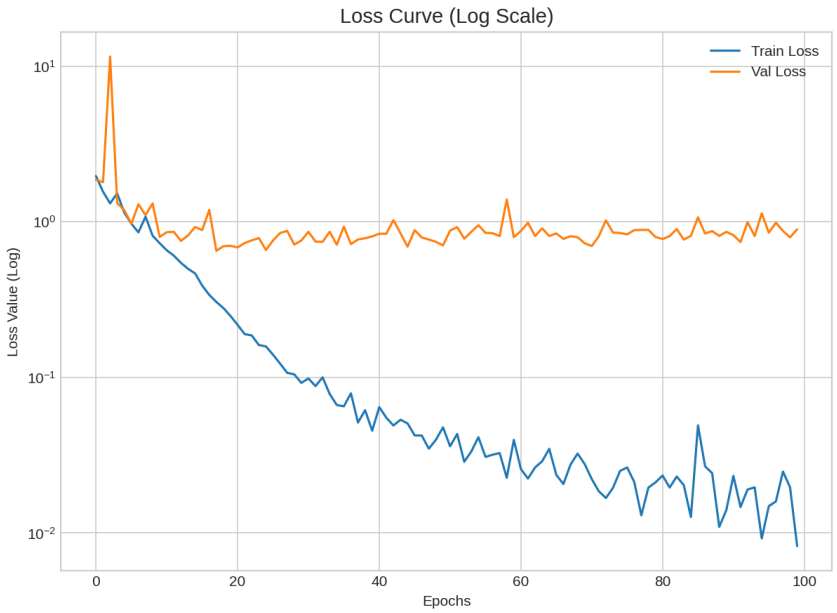




Assignment #4

- **CIFAR10 with VGGNet - Analysis 1**
  - 현재까지 가장 높은 Test Accuracy는 86.14%
    - Train Loss는 0에 가까운 수치를 보여주고 있으나,
      Validation Loss는 이에 못 미치고 차이가 벌어지는 모습을 보여주고 있음
    - Overfitting이 발생하고 있다고 판단





Assignment #4

- **CIFAR10 with VGGNet - Analysis 1**
  - Overfitting 해결을 위한 시도
    - Model Capacity 축소 (Layer 수 조절) → 가장 깊은 VGG19에서 최고 성능 달성
      - 현재 Task에는 High Capacity가 필요하다고 생각
    - Regularization (L2, Dropout) → 적용 시 오히려 성능 하락
      - 인위적인 제약이 모델의 표현력을 저해한다고 생각
  - 모델을 제한하지 않고 Overfitting을 방지하기 위해 Data Augmentation 적용

Optimizer: Adam
Learning rate: 1e-3

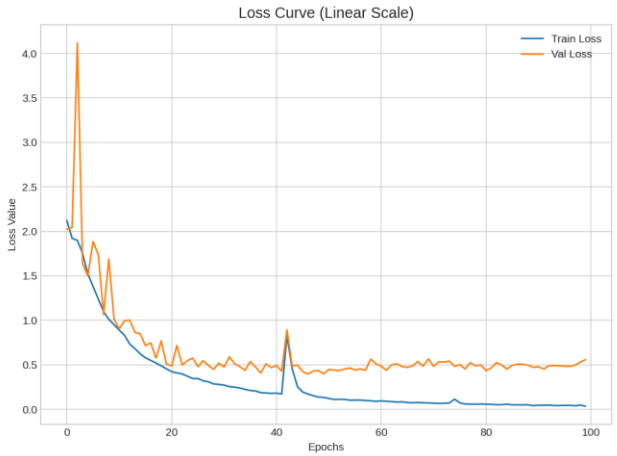
Model Code	Test Acc (%)
VGG11	82.98
VGG13	85.86
VGG16	86.13
VGG19	**86.14**

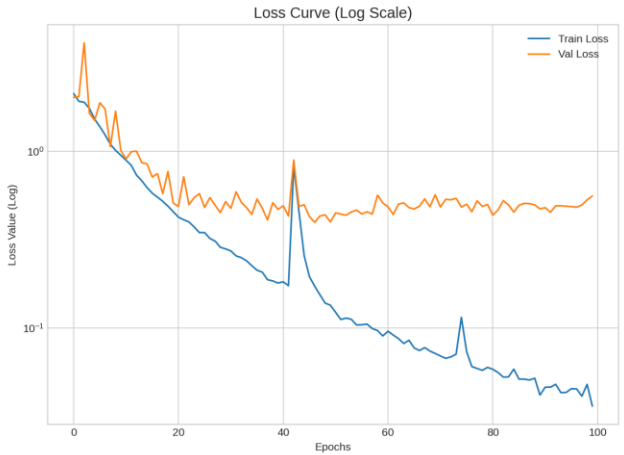
Model Code: VGG19, LR: 1e-3

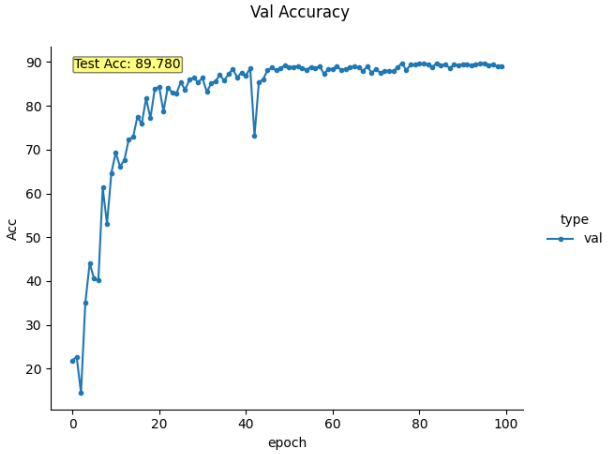
L2	Dropout	Test Acc (%)
0	**0**	**86.14**
0	0.3	85.50
0	0.5	84.77
1e-2	0	80.15
1e-2	0.3	73.58
1e-2	0.5	79.45
1e-4	0	84.50
1e-4	0.3	84.66
1e-4	0.5	84.74

Assignment #4

- **CIFAR10 with VGGNet - Part 2**
  - exp4 (Data Augmentation)
    - RandomCrop, RandomHorizontalFlip 적용
    - Test Acc 89.78% 확인 (직전 실험 대비 3.64% 증가)
    - Train Loss와 Validation Loss 사이의 Gap도 줄어든 것을 확인

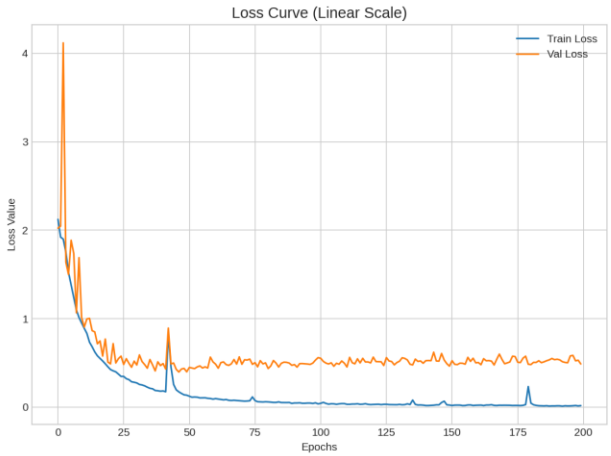


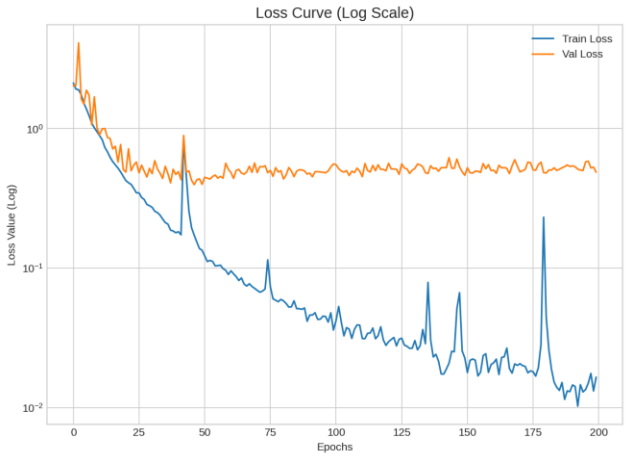


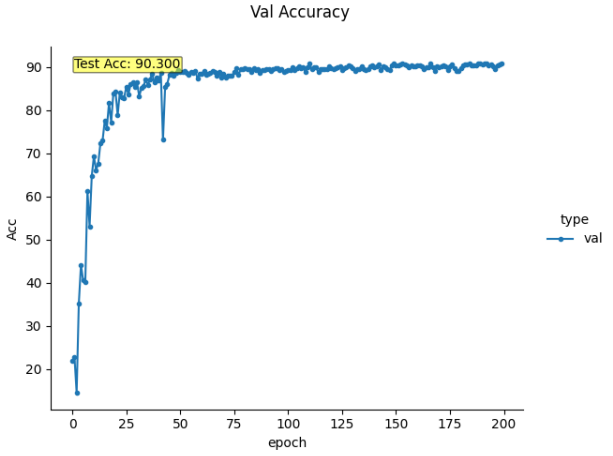


Assignment #4

- **CIFAR10 with VGGNet - Part 2**
  - exp4.1 (Data Augmentation)
    - RandomCrop, RandomHorizontalFlip 적용
    - Data Augmentation으로 인해 학습 속도가 느려졌을 것이라 판단, Epoch를 200까지 올림
    - Test Acc 90.30% 확인 (exp4 대비 0.52% 증가)







Assignment #4

- **CIFAR10 with VGGNet - Analysis 2**
  - exp4.1의 클래스별 Train/Validation Accuracy를 분석
    - 특정 클래스(Bird, Cat, Dog)에서 Generalization Gap이 큰 것을 확인
      - Overfitting이 특정 클래스에 집중된다고 판단

Class	Train Acc (%)	Val Acc (%)	Gap (%)
Airplane	99.75	90.28	9.47
Automobile	99.97	96.00	3.97
Bird	99.85	89.45	10.40
Cat	99.30	82.35	16.95
Deer	99.53	90.19	9.34
Dog	99.62	86.15	13.47
Frog	99.40	91.43	7.98
Horse	99.98	92.06	7.91
Ship	99.92	96.08	3.85
Truck	99.85	94.86	4.99
Overall	99.71	90.8	8.84

Assignment #4

- **CIFAR10 with VGGNet - Analysis 2**
  - **Class-Adaptive Data Augmentation**
    - Overfitting이 심한 특정 클래스(Gap > 10%)에 Data Augmentation을 강하게 적용하여, 모델의 일반화 성능 향상을 유도
      - 일반 클래스: Standard Augmentation (RandomCrop + RandomHorizontalFlip)
      - 과적합 클래스: Strong Augmentation (Standard Augmentation + α)
    - Strong Augmentation을 적용한 클래스의 Gap이 줄어들고, 동시에 Train Accuracy가 포화 상태가 되지 않은 경우가 가장 이상적이라 생각
      - 이 경우에 Epoch를 늘려주면 Gap이 유지되면서 Train loss와 Validation loss가 감소할 것이라는 가설
    - 여러 번의 실험을 통해 Strong Augmentaion의 적절한 정도를 찾아야 함.
      - 과도한 Augmentation 적용 시, 모델이 분류의 기준으로 학습해야 하는 클래스의 핵심 특성이 손실 되어 성능이 저하 될 수 있음

Assignment #4

- **CIFAR10 with VGGNet - Part 3**
  - exp5 (Class-Adaptive Data Augmentation)
    - Epoch 200에서 가장 이상적인 결과를 찾고 Epoch를 300으로 늘려보며 가설 검증
      - exp5.7: Standard Augmentation + RandAugment (num_ops=2, magnitude=5) + RandomErasing
      - exp5.10: Standard Augmentation + RandAugment (num_ops=3, magnitude=6) + RandomErasing
    - 예상과 달리 Gap이 잘 유지되지 않고, Train Acc가 포화되지 않음

Index	Epoch	Bird			Cat			Dog			Test Acc
		Train	Val	Gap	Train	Val	Gap	Train	Val	Gap	
exp4.1	200	99.85	89.45	10.40	99.30	82.35	16.95	99.62	86.15	13.47	90.30
exp5.7.0	200	98.70	90.15	8.55	98.55	89.27	9.28	98.55	89.27	9.28	90.20
exp5.7.1	300	98.47	85.87	12.60	96.93	82.85	14.08	96.90	87.09	9.82	90.61
exp5.10.0	200	97.82	85.97	11.85	94.65	82.55	12.11	93.25	81.24	12.00	90.03
exp5.10.1	300	97.80	85.27	12.52	96.78	87.06	9.72	94.90	82.75	12.15	90.53

Assignment #4

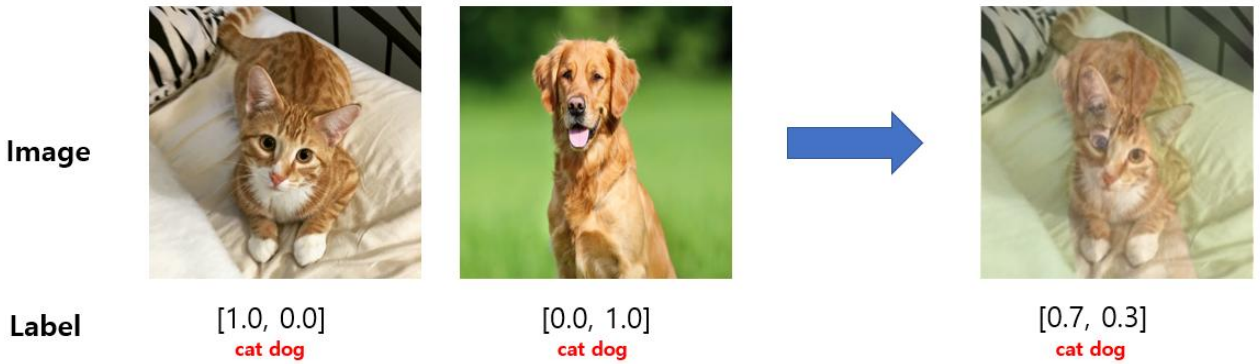
- **CIFAR10 with VGGNet - Analysis 3**
  - **Class-Adaptive Data Augmentation – 문제점 및 한계**
    - Gap이 잘 유지되지 않음
    - 또한 Train Accuracy가 100%에 가깝게 포화되지 않음
      - 그로 인해 만약 Gap이 유지되더라도, 의미있는 성능 향상을 이루지 못함
    - 고정된 Augmentation을 사용하니, 학습 도중에 다른 클래스의 Gap이 벌어지기도 함

Method	Bird			Cat			Dog			Deer			Test Acc
	Train	Val	Gap	Train	Val	Gap	Train	Val	Gap	Train	Val	Gap	
exp4.1	99.85	89.45	10.40	99.30	82.35	16.95	99.62	86.15	13.47	99.53	90.19	9.34	90.30
exp5.3.0	98.70	90.15	8.55	98.55	89.27	9.28	98.55	89.27	9.28	98.49	87.40	11.09	90.22

    - Hyperparameter로 사용할 수 있는 변수가 많아서 학습에 많은 시간과 자원이 필요
      - 사용할 Augmentation의 종류와 개수, Augmentation 각각의 하이퍼파라미터, …
    - 지금은 Seed가 고정되어 있어 문제가 발생하고 있지 않지만,
      Seed가 고정되어 있지 않을 경우 매번 Overfitting이 발생하는 클래스가 바뀔 수 있음

Assignment #4

- **CIFAR10 with VGGNet - Analysis 3**
  - **Adaptive mixup**
    - **mixup** (ICLR 2018)
      - 학습 데이터에서 두 개의 데이터를 혼합하여 새로운 학습 데이터를 만듦
        - ✓이미지 데이터의 투명도 λ를 조절하여 혼합
          label 데이터 또한 투명도 λ의 비율에 맞춤
        - ✓$\hat{x} = \lambda x_i + (1-\lambda)x_j$ / $\hat{y} = \lambda y_i + (1-\lambda)y_j$
        - ✓λ가 0.5에 가까울 수록 모델이 정답을 맞추기 어려워짐
        - ✓mixup의 강도를 하이퍼파라미터 α로 조절
          α가 클 수록 λ가 0.5에 가까운 값이 나올 확률 증가



Assignment #4

- **CIFAR10 with VGGNet - Analysis 3**
  - **Adaptive mixup**
    - 매 에포크마다 클래스별 Loss Gap을 구해서 다음 에포크의 mixup 강도 α를 업데이트
      - Gap이 클 수록 α가 커지게 구현
      - 이로인해 Augmentation이 더 강해지고 Overfitting을 완화할 수 있도록 함
    - Overfitting이 일어나는 클래스가 변하더라도 그에 맞춰 mixup 강도가 변화
      - Seed를 고정시킬 필요 X
      - 매번 학습이 끝나고 Overfitting이 일어나는 클래스를 비교할 필요 X
    - 사용하는 Hyperparameter는 5개
      - base_alpha, gap_sensitivity, max_alpha, alpha_momentum, warmup_epochs

Assignment #4

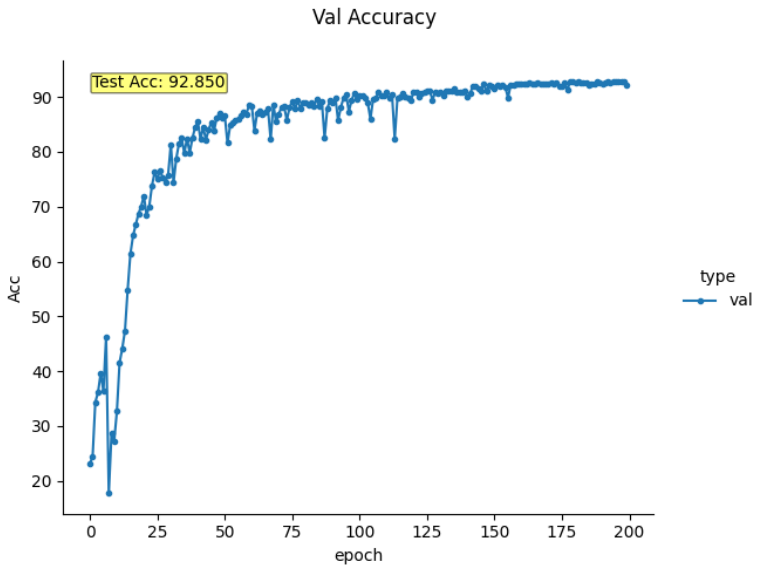
- **CIFAR10 with VGGNet - Analysis 3**
  - **Learning rate Scheduler**
    - Augmentation을 강하게 적용했을 때,
      Train Accuracy가 100%에 가깝게 포화되지 않는 문제가 있었음
    - Learning rate Scheduler를 통해 더 쉽게 Train Accuracy가 100% 가까이 오를 수 있게 개선

Assignment #4

- **CIFAR10 with VGGNet - Part 4**
  - exp9.5.1 (Adaptive Mixup)
    - Base Alpha: 0.2, Alpha Momentum: 0.6, Gap Sensitivity: 2.0, Warmup Epochs: 50
    - Test Acc 92.85% 확인 (exp4.1 대비 2.55% 증가)
    - Train Acc는 100% 가까이 포화되고, 대부분의 클래스에서 Gap이 감소함을 확인

exp4.1 클래스별 Train, Val Acc 및 Gap

Class	Train Acc (%)	Val Acc (%)	Gap (%)
Airplane	99.75	90.28	9.47
Automobile	99.97	96.00	3.97
Bird	99.85	89.45	10.40
Cat	99.30	82.35	16.95
Deer	99.53	90.19	9.34
Dog	99.62	86.15	13.47
Frog	99.40	91.43	7.98
Horse	99.98	92.06	7.91
Ship	99.92	96.08	3.85
Truck	99.85	94.86	4.99
Overall	99.71	90.8	8.84



Class	Train Acc (%)	Val Acc (%)	Gap (%)
Airplane	99.95	94.94	5.01 ↓↓
Automobile	99.92	97.17	2.75 ↓
Bird	99.82	90.55	9.28 ↓
Cat	99.78	88.57	11.21 ↓↓↓
Deer	99.88	92.46	7.42 ↓
Dog	99.67	87.28	12.39 ↓
Frog	99.83	94.21	5.61 ↓
Horse	99.90	94.07	5.83 ↓
Ship	99.97	94.90	5.07 ↓
Truck	99.85	95.27	4.58 ≈
Overall	99.86	92.91	6.92 ↓

exp4.1 대비 ↓ : 1% 이상 / ↓↓ : 3% 이상 / ↓↓↓ : 5% 이상

Assignment #4

- **CIFAR10 with VGGNet - Best Result**
  - Model Code: VGG19, Batch size: 256, Epoch: 200, Learning rate: 1e-3
    Activation Function: ReLU, BatchNorm: True, Optimizer: Adam
  - L2 Regularization: 0, Dropout: 0
  - Scheduler type: ReduceLROnPlateau, Scheduler factor: 0.5, Scheduler patience: 10
  - Base Alpha: 0.2, Gap Sensitivity: 2.0, Max Alpha: 2.0,
    Alpha Momentum: 0.6, Warmup Epochs: 50
  - Test Accuracy: 92.85 %

Assignment #4

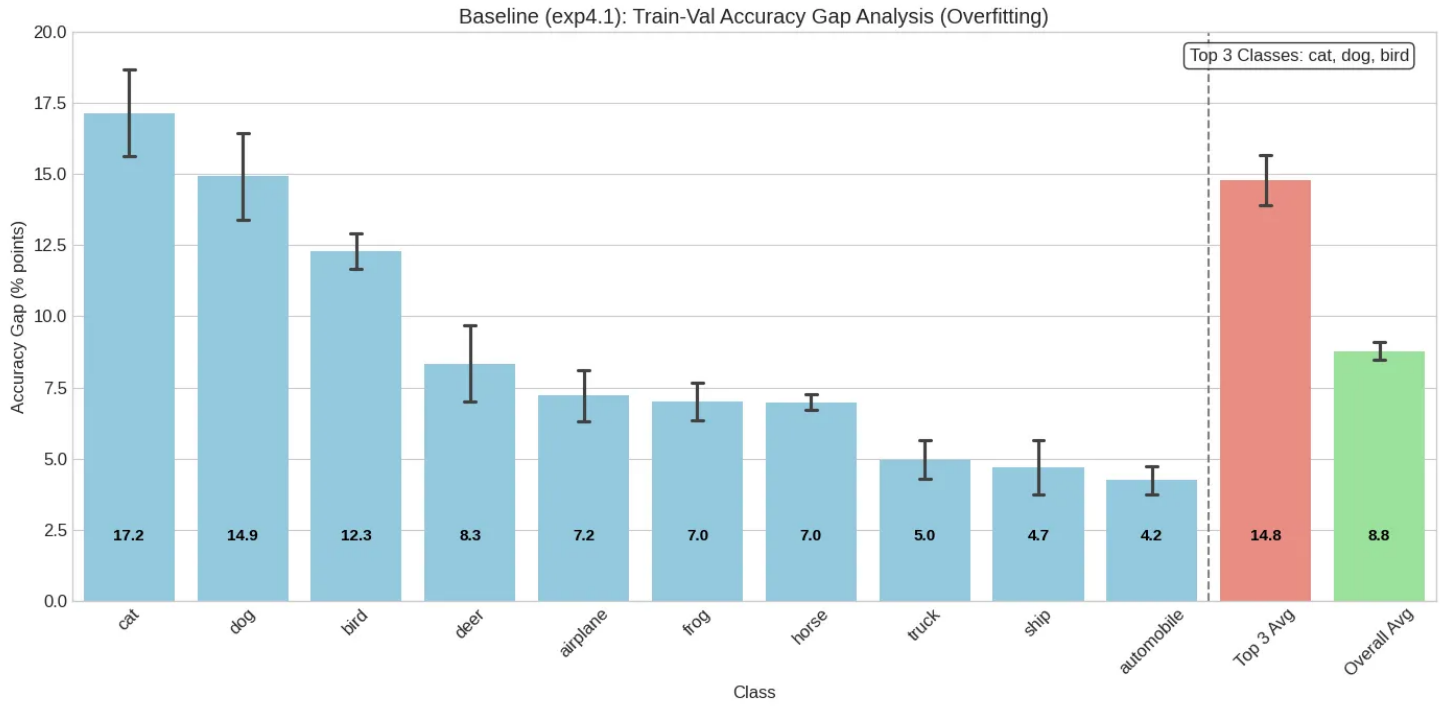
Supplement

Assignment #4

- **CIFAR10 with VGGNet - extra experiment**
  - Baseline / Mixup / Adaptive Mixup
    - 세 경우 모두 Mixup 이외의 하이퍼파라미터는 모두 동일
      (Learning rate Scheduler 포함)
    - 서로 다른 5개의 Seed에서의 실험 결과를 바탕으로 분석
    - Baseline은 이전 실험 당시 가장 성능이 잘 나왔던 exp4.1의 하이퍼파라미터
      + Learning rate Scheduler
    - Mixup은 3가지 alpha값 중 가장 성능이 잘 나온 alpha=0.2
    - Adaptive Mixup은 이전 실험 당시 가장 성능이 잘 나왔던 exp9.5.1의 하이퍼파라미터
    - 이를 통해 Adaptive Mixup의 일반화 성능을 파악해보고자 함

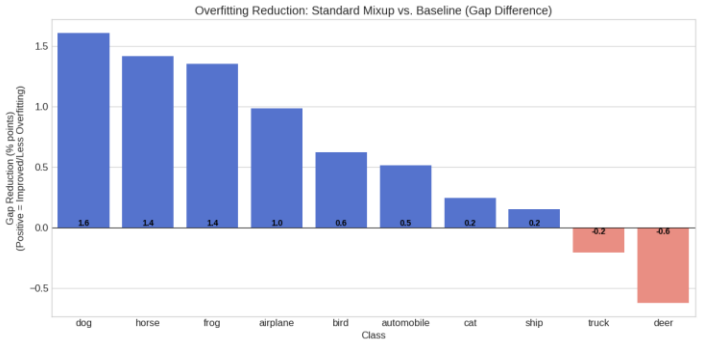
Assignment #4

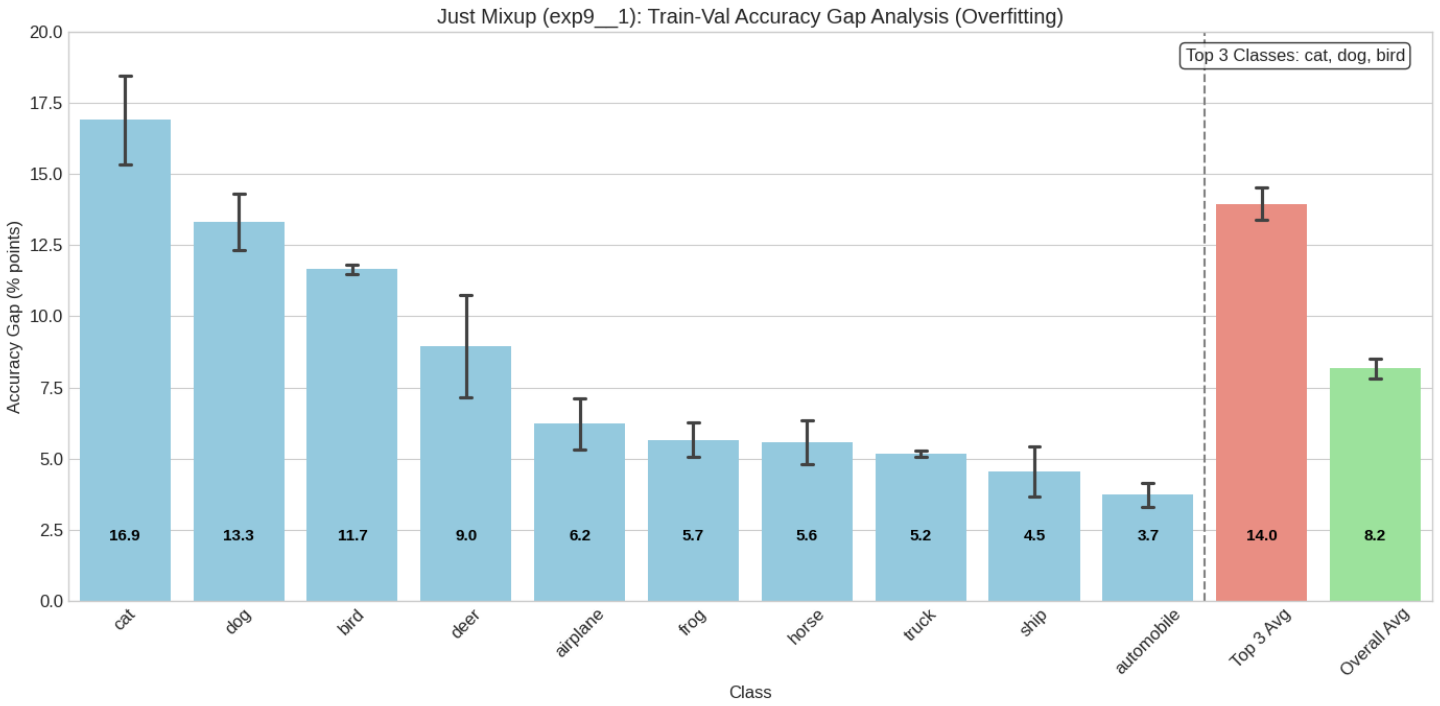
- **CIFAR10 with VGGNet - extra experiment**
  - Baseline 클래스별 Acc Gap 평균



Assignment #4

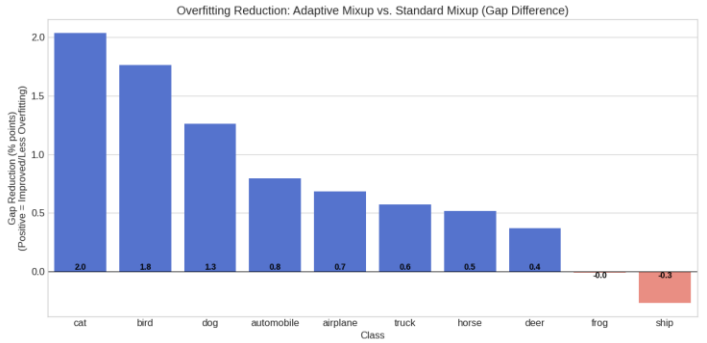
- **CIFAR10 with VGGNet - extra experiment**
  - Mixup 클래스별 Acc Gap 평균
    - Baseline 대비 대부분 Gap이 감소한 것을 확인

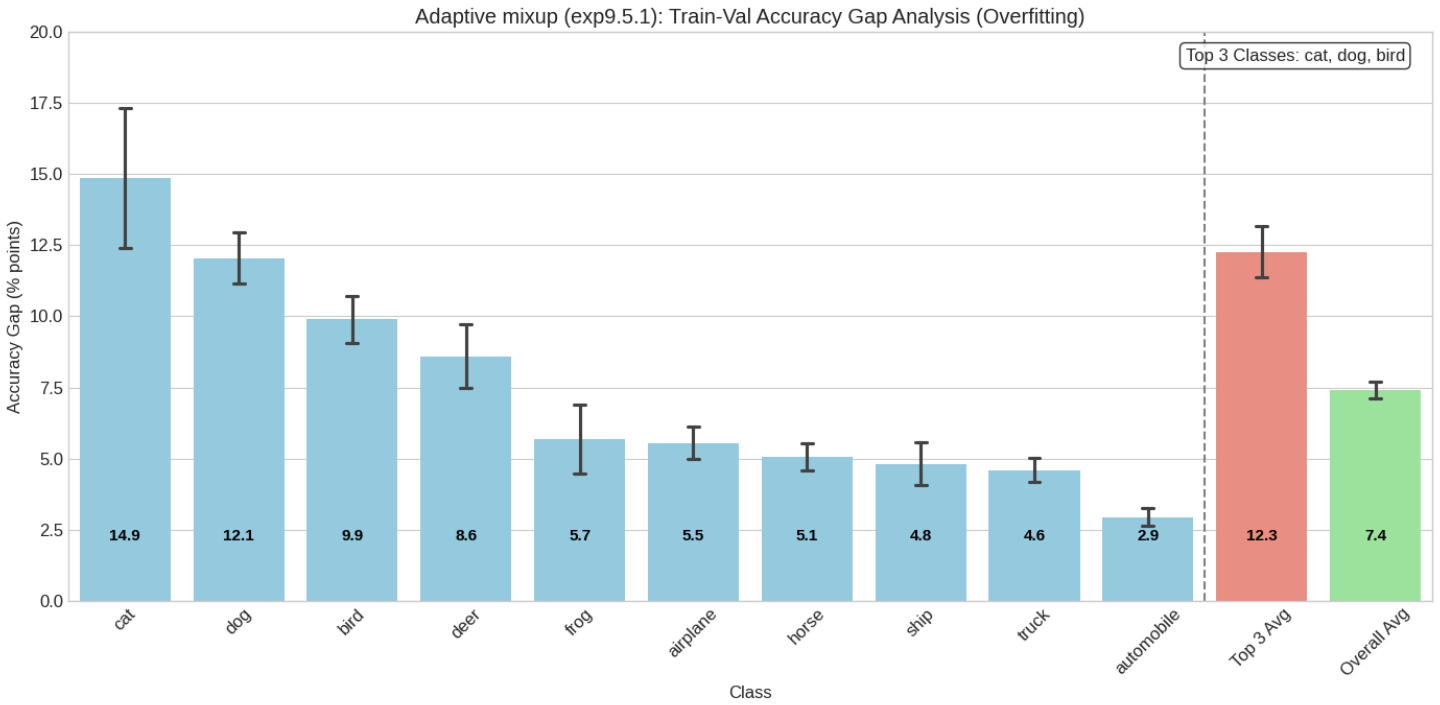




Assignment #4

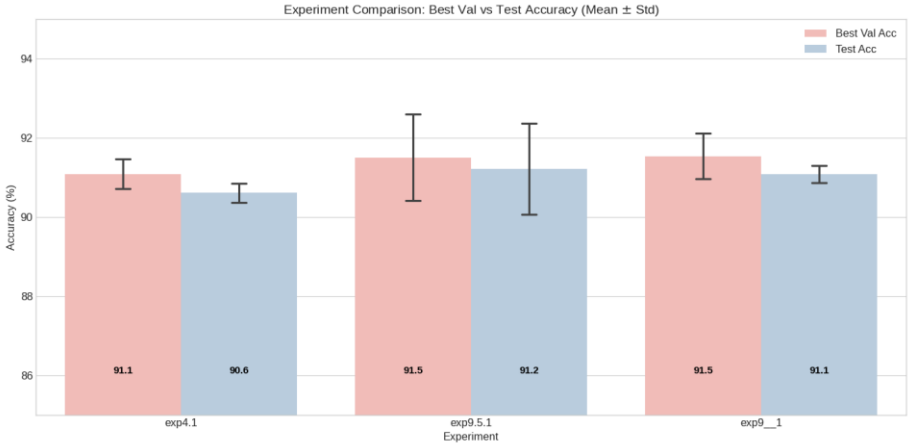
- **CIFAR10 with VGGNet - extra experiment**
  - Adaptive Mixup 클래스별 Acc Gap 평균
    - Mixup 대비 대부분 Gap이 감소한 것을 확인



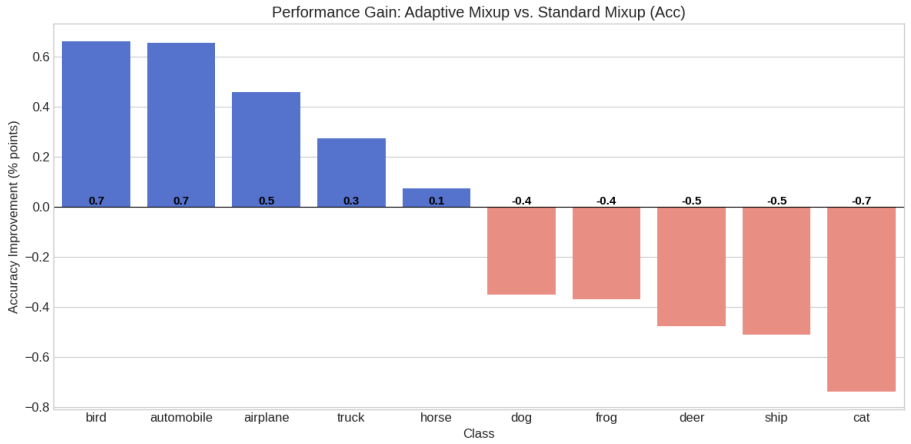


Assignment #4

- **CIFAR10 with VGGNet - extra experiment**
  - Baseline / Mixup / Adaptive Mixup Acc 평균
    - Mixup과 Adaptive Mixup 둘 다 Baseline 대비 약간의 성능 향상을 보임
    - 그러나 Mixup과 Adaptive Mixup은 큰 차이가 없는 것을 확인
    - 클래스별 Acc를 비교해본 결과
  - Random Seed 일반화 성능은 Mixup과 Adaptive Mixup이 큰 차이가 없다는 결론



Baseline



Adaptive Mixup

Mixup

Assignment #4

- **CIFAR10 with VGGNet**
  - **Adaptive mixup - 구현**
    - gaps = val_class_loss - train_class_loss
    - target_alphas = base_alpha + gap_sensitivity * ReLU(gaps)
      - gap이 음수일 때는 기본 alpha값이 되도록 구현
      - gap_sensitivity로 강도 조절 (Optimizer의 Learning rate와 유사)
    - target_alphas = clamp(target_alphas, min=base_alpha, max=max_alpha)
      - alpha가 너무 크거나 작아지는 것을 방지하기 위함
    - class_alphas = alpha_momentum * class_alphas + (1 - alpha_momentum) * target_alphas
      - momentum을 적용해서 alpha값이 일관성을 갖고, 불필요한 노이즈를 줄이기 위함 (Optimizer의 momentum과 유사)
    - warmup_epochs 동안에는 target_alphas를 base_alpha로 유지
      - 학습 초반, target_alphas가 크게 바뀌어 학습이 불안정해지는 것을 방지하기 위함

Assignment #4

- **CIFAR10 with VGGNet**
  - **Adaptive mixup - 구현**
    - eff_alphas = max(alpha_a, alpha_b)
      - 랜덤으로 뽑힌 두 개의 클래스 중 더 큰 alpha를 기준으로 mixup을 진행
    - lambda = Beta(eff_alphas, eff_alphas).sample()
      - Beta distribution을 이용해서 lambda값을 선정
      - alpha가 클 수록 0.5 근처의 lambda가 나올 확률이 높아짐
        alpha가 클 수록 모델이 문제를 맞추기 어려워져서 Overfitting이 완화될 것이라 생각

