『CVSP 기초 학습』

Deep Learning: Core



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



Presented By
김준수

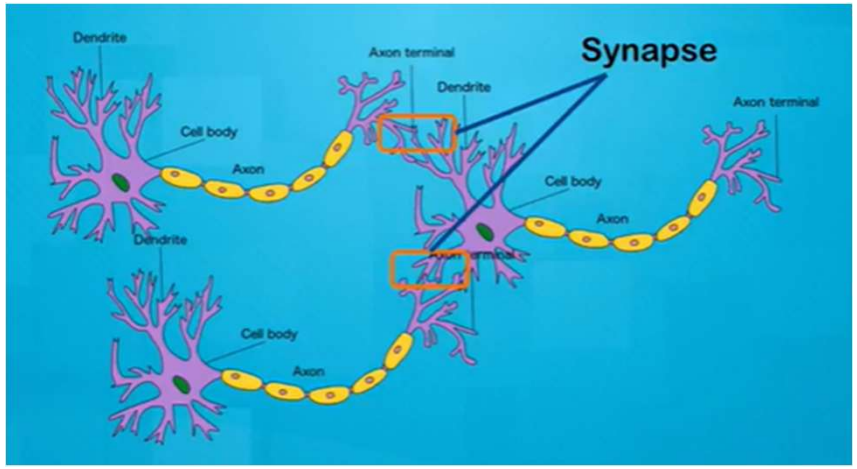
Outline

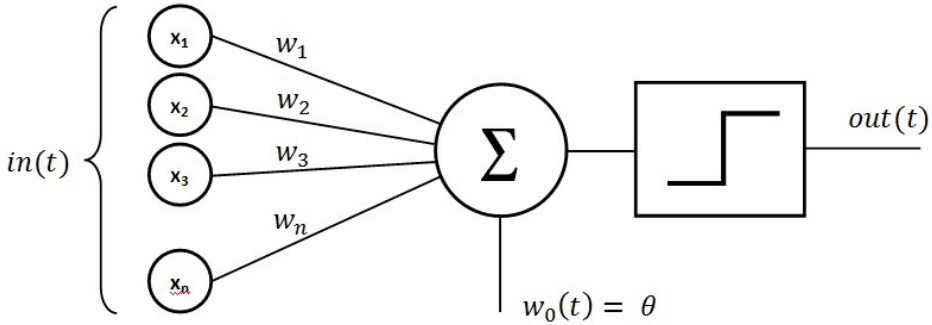
- **Basic Course (Stand Alone Deep Learning)**
  - Basic Architectures (Regression / Loss / Optimization)
  - Fundamental Neural Networks (MLP / CNN)
- **Transformer (All You Need Is Attention)**
  - Recurrent Neural Network (RNN)
  - Transformer Architecture
- **Experiment**
  - ResNet & Multi Pipelined Model

Basic Course
(Stand Alone Deep Learning)

Basic Architecture

- **Perceptron**
  - **Start of Machine Learning**
    - 인공지능은 기계에게 지능을 부여하고자 하는 아이디어에서 시작.
    - 인간의 뇌 신경세포 구조를 모방하는 형태로 접근하고자 했음.
    - 여러 시그널이 복합적으로 계산되어 Axon을 타고 다음 뉴런으로 전달됨.



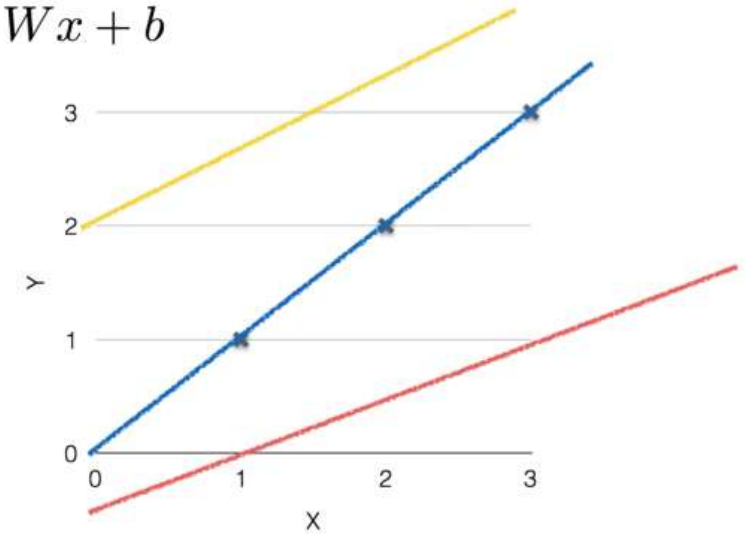


Basic Architecture

- **Perceptron**
  - **Hypothesis**
    - 2차원 좌표 평면 내에 존재하는 데이터들을 하나의 직선으로 주로 근사.
    - 반면, 직선을 그어낼 때, 어떻게 직선을 그을 것인가를 결정해야 함.
    - 직선을 긋는 기준은 기존 데이터들과 직선 간의 총합 거리가 가장 적은 것.
    - 이처럼 줄여나가고자 설정하는 값을 Cost 함수 또는 Loss 함수라 한다.

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

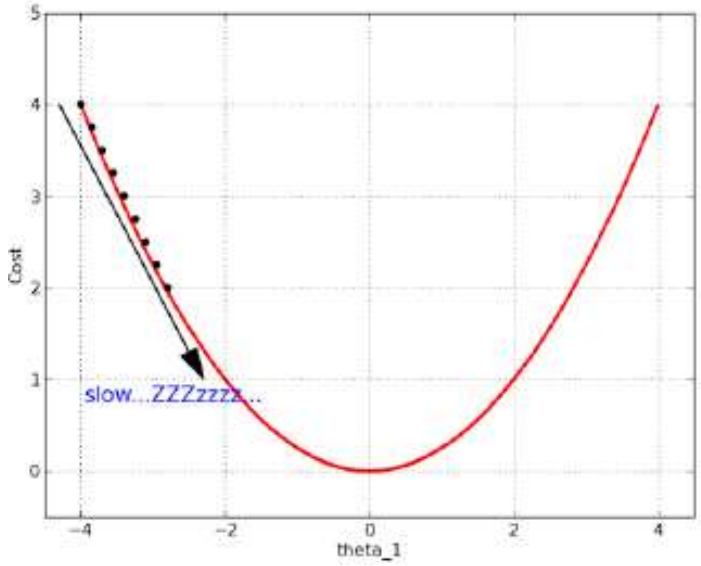
$$cost = \frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}) - y^{(i)})^2$$



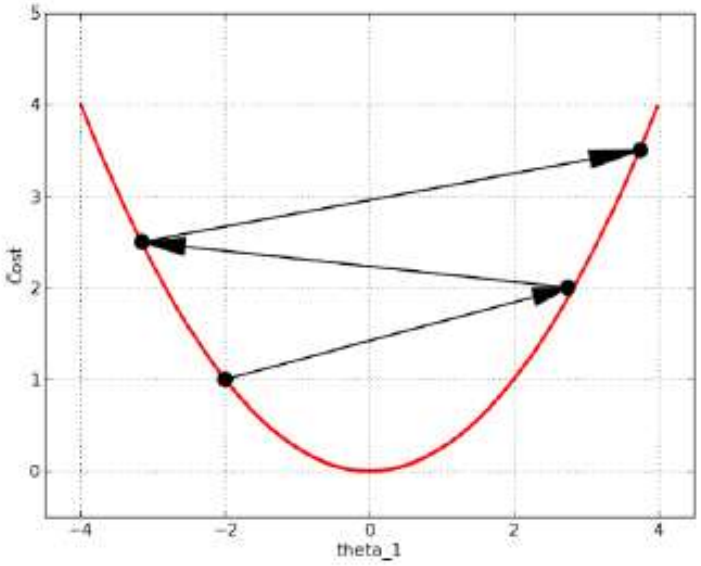
Basic Architecture

- **Perceptron**
  - **Gradient Descent**
    - W값에 대해 Cost함수를 편미분 시, 기울기 도출.
    - 편미분값의 일부를 기존 W에 뺌으로써 W를 업데이트.
    - Learning Rate를 효과적으로 설정하는 것이 중요.

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$



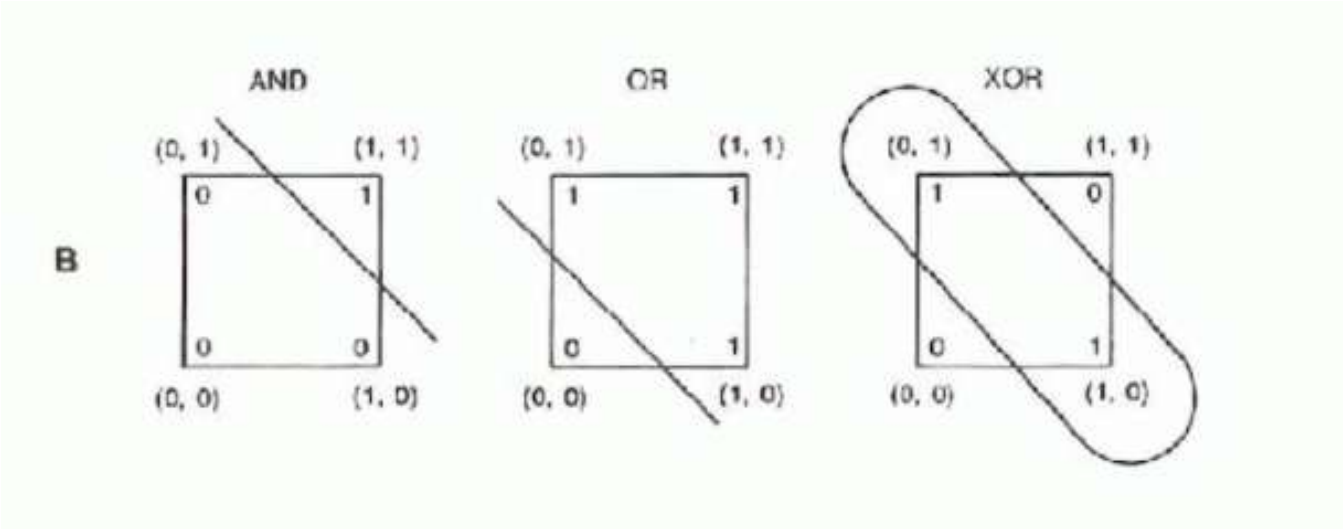
(c) Small Learning Rate



(d) Large Learning Rate

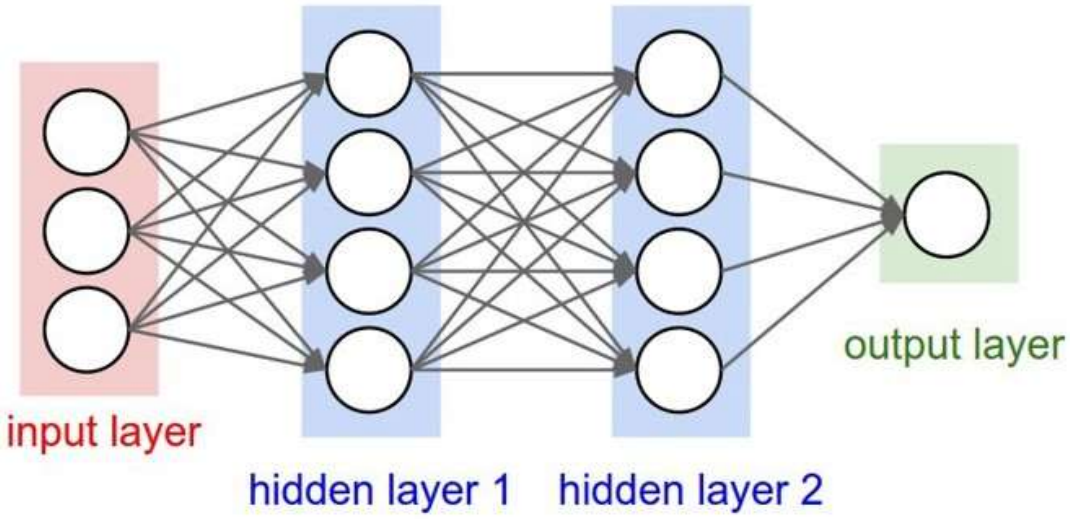
Basic Architecture

- **Multi Layer Perceptron (MLP)**
  - **Need of MLP**
    - Linear Classifier는 선형분리에 능통.
    - And나 Or의 문제는 잘 풀 수 있음.
    - 하지만 XOR같은 단일 선으로 풀 수 없는 문제는 풀 수 없음.



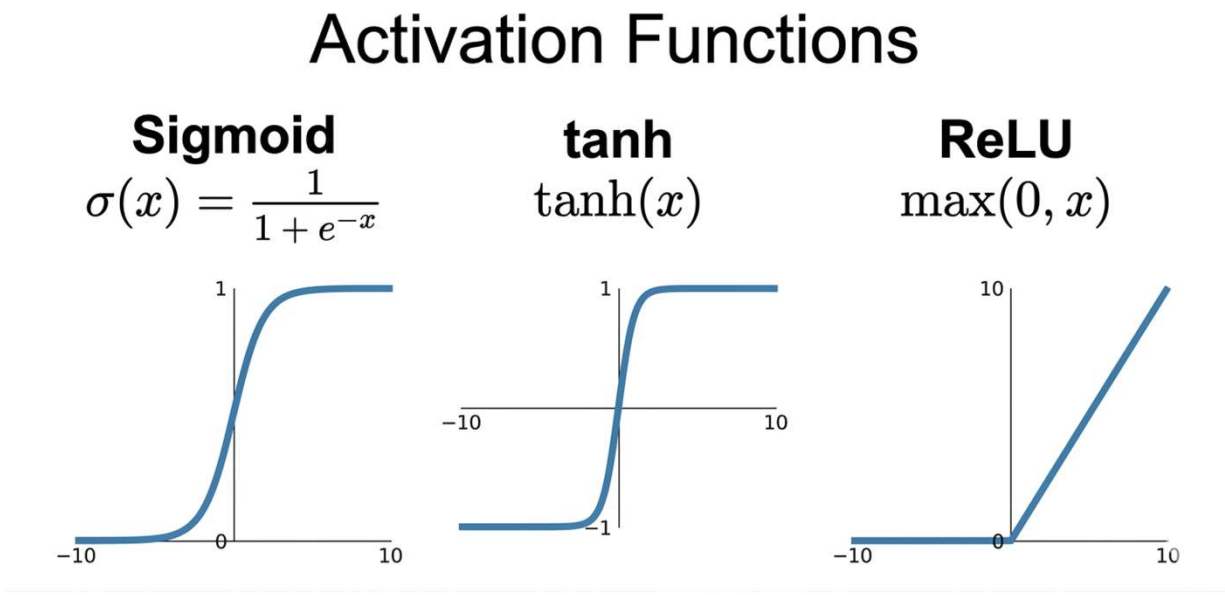
Basic Architecture

- **Multi Layer Perceptron (MLP)**
  - **Multi Layer Perceptron**
    - 단일 레이어로는 xor 문제를 풀 수 없다고 판단.
    - 따라서 레이어를 늘려서(Multi-Layer) 계산.
    - 선형 함수만 많이 쌓일 경우 하나의 선형함수로 표현돼 버리는 문제 존재.
    - 비선형성을 주기 위해 활성화 함수(Activation Function)을 Hidden Layer 사이마다 배치.



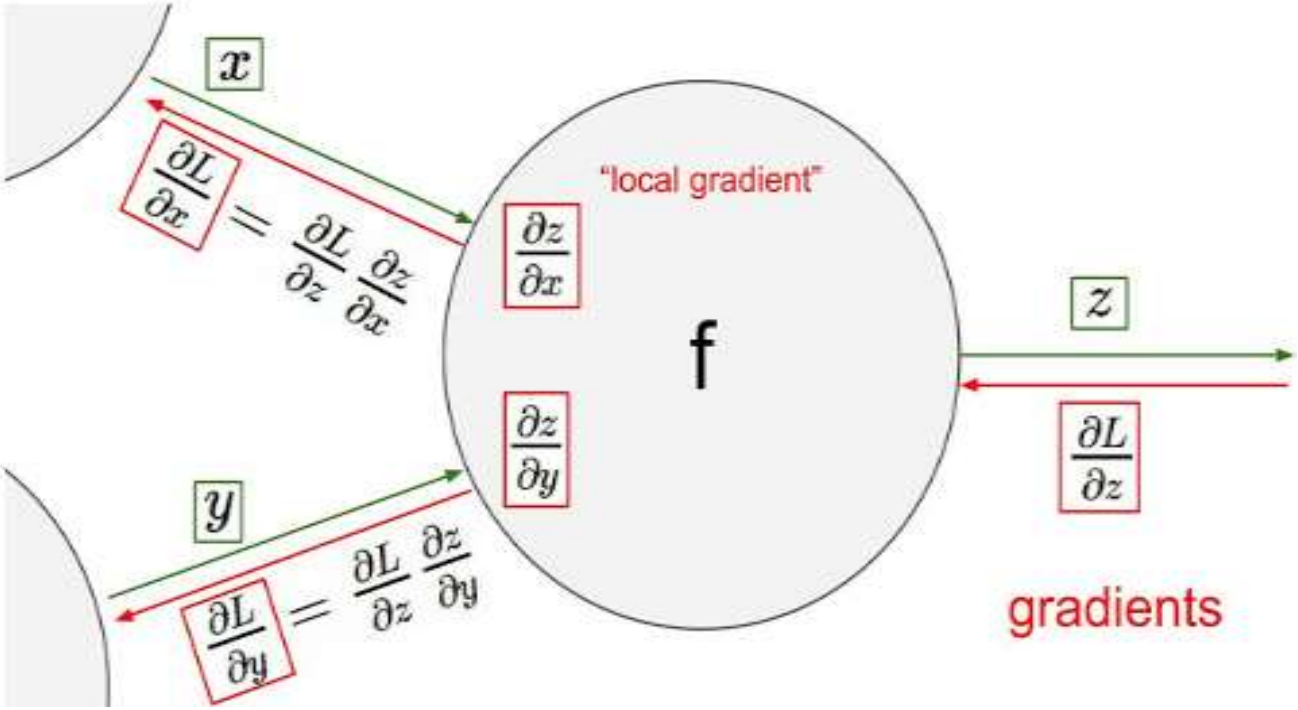
Basic Architecture

- **Multi Layer Perceptron (MLP)**
  - **활성화 함수 (Activation Function)**
    - 활성화 함수는 Sigmoid, tanh, ReLU가 존재.
    - Sigmoid, tanh는 레이어가 쌓일 수록 기울기가 사라지는 Gradient Vanishing 문제 발생.
    - ReLU를 통해 Gradient Vanishing 문제 해결.



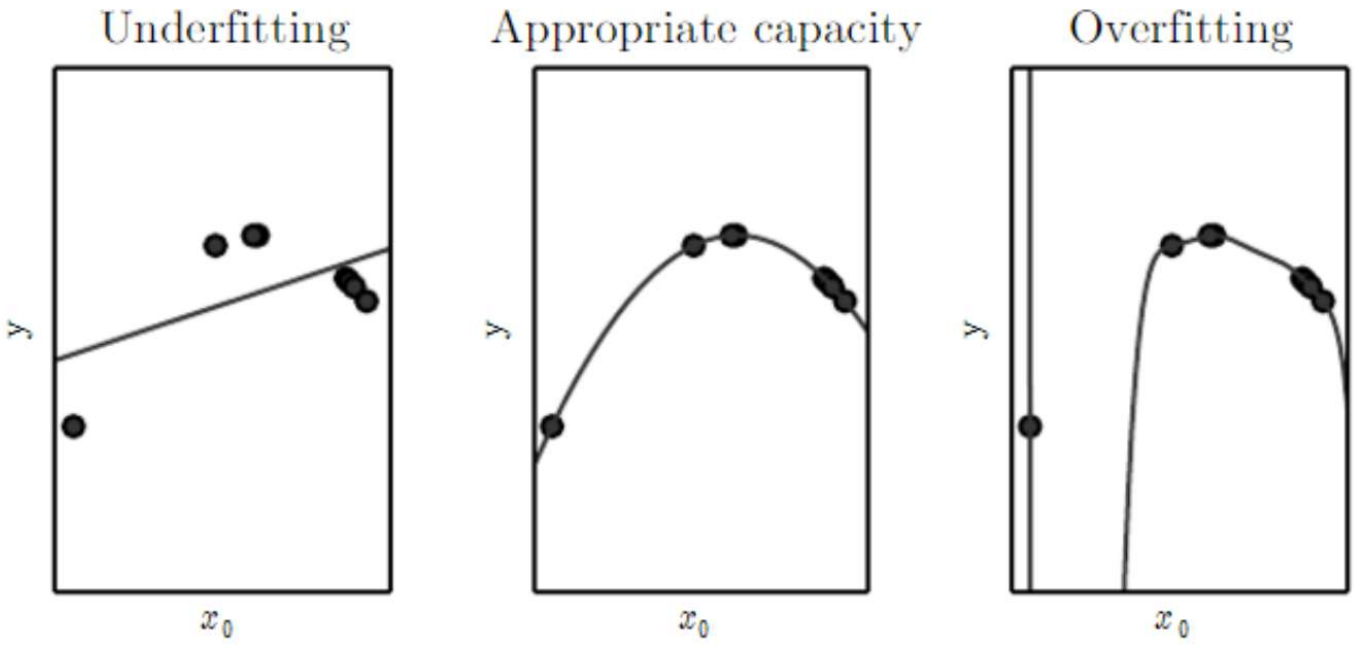
Basic Architecture

- **Multi Layer Perceptron (MLP)**
  - **Backpropagation**
    - 정답과 예상치와의 Loss를 이용하여 이전 레이어의 Gradient를 계산가능.
    - Gradient를 통해 기존의 가중치들을 업데이트 가능.



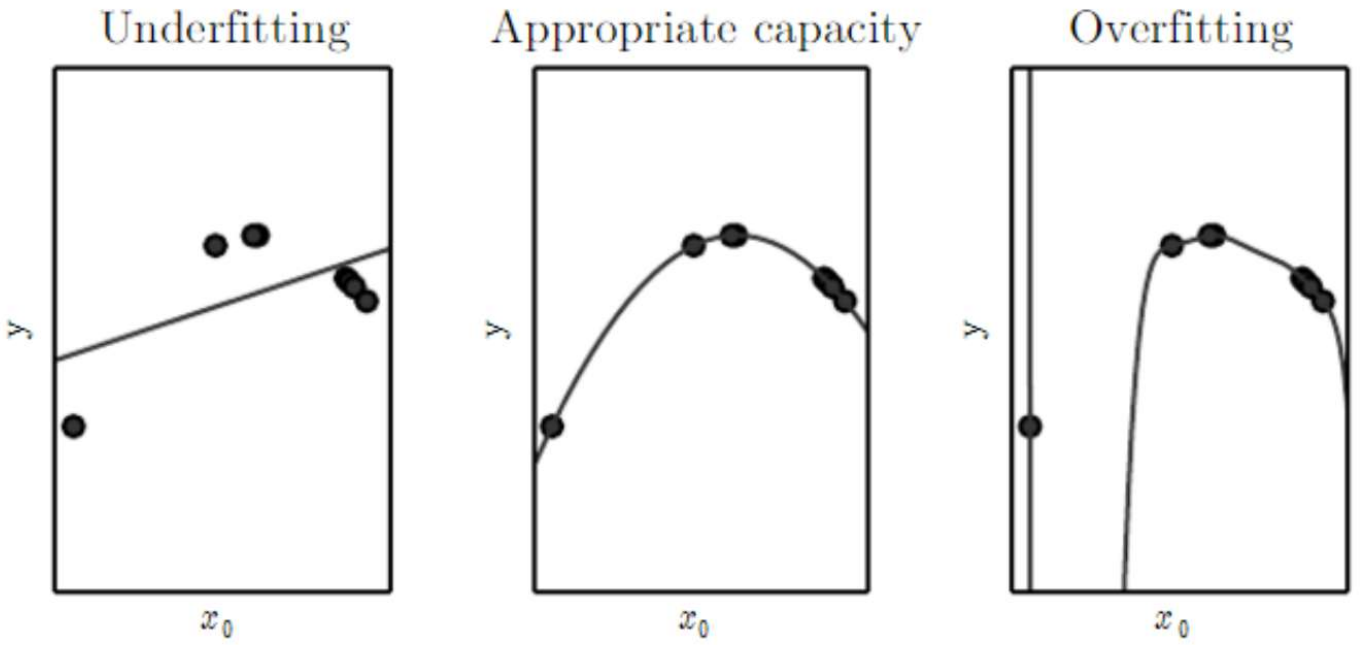
Basic Architecture

- **Overfitting & Regularization**
  - **Model Capacity**
    - 모델이 학습 데이터의 복잡한 패턴을 표현할 수 있는 능력.
    - 그런데, Model Capacity가 높으면 무조건 좋은 것인가?



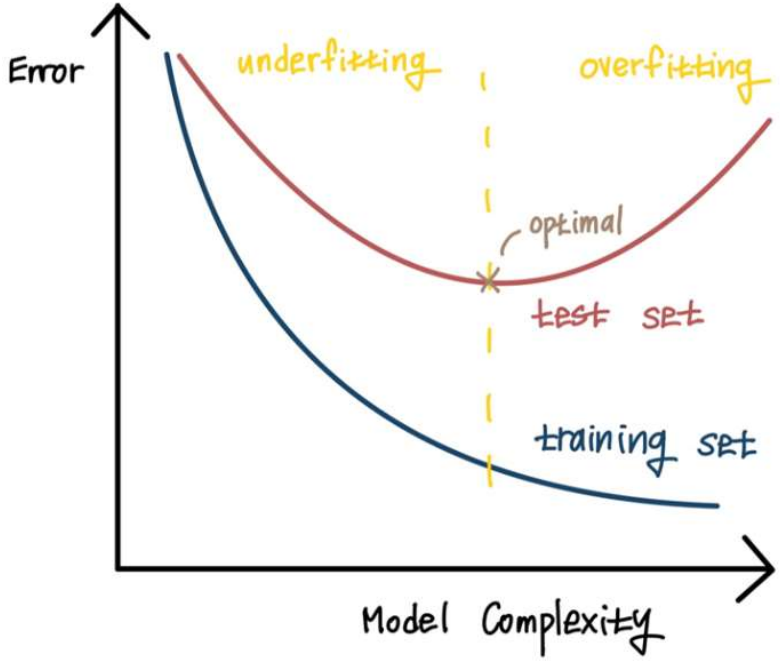
Basic Architecture

- **Overfitting & Regularization**
  - **Model Capacity**
    - 모델이 복잡한 상태에서 훈련 데이터를 통해 모델을 학습.
    - 학습데이터 내 필요없는 노이즈까지 학습하여 틀린 답을 낼 가능성이 커짐.
    - 즉, 훈련데이터에만 성능이 좋아 일반적인 성능이 떨어질 수 있음.(overfitting)
    - 따라서 Model Capacity와 학습 데이터 사이의 적절한 균형을 잡을 필요가 있음.



Basic Architecture

- **Overfitting & Regularization**
  - **Underfitting**
    - 데이터에 대한 학습이 너무 단순화 되어, 에러가 많이 나오는 상황.
  - **Overfitting**
    - 학습 데이터(Train set)에 대해 과하게 학습된 상황.
    - 학습 데이터 이외의 상황엔 모델이 잘 작동하지 못하게 됨.

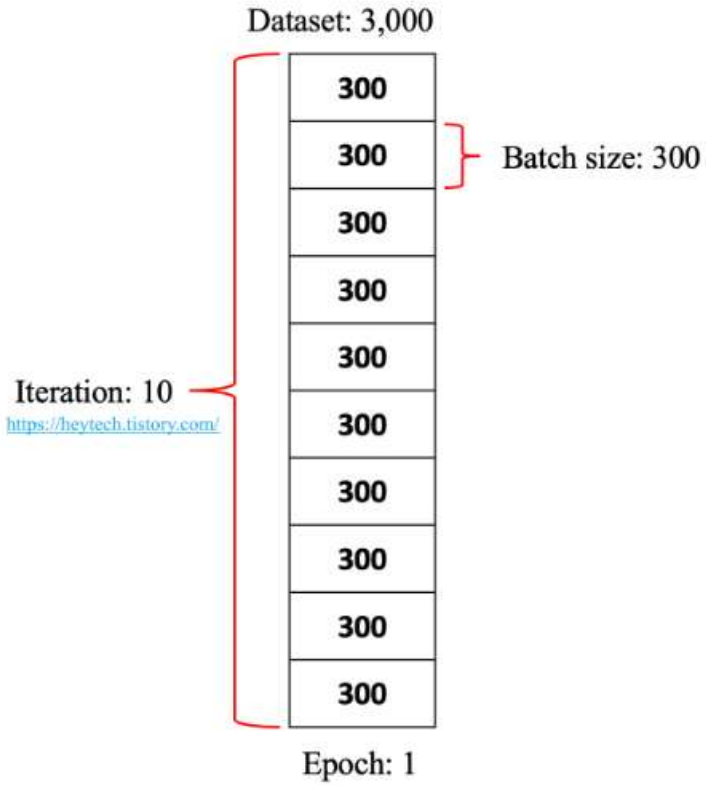


Basic Architecture

- **Overfitting & Regularization**
  - **Data Set**
    - Overfitting이 얼마나 일어나는지 확인하기 위해 학습할 때 데이터셋을 나눈다.
    - Train set, Validation set, Test set 으로 구성.
  - **Train set**
    - Wx + b 에서 W와 b의 파라미터를 조절하는 데 쓰이는 데이터 셋.
    - 주요한 학습을 이 데이터에서 진행.
  - **Validation set**
    - Train set에 너무 많은 epoch로 학습시키면 overfitting이 일어남.
    - Validation set을 통해 overfitting인지 확인.
    - 하이퍼 파라미터를 조절.
  - **Test set**
    - Validation set을 통해 최적화 된 모델의 실 성능을 검증하기 위한 데이터.

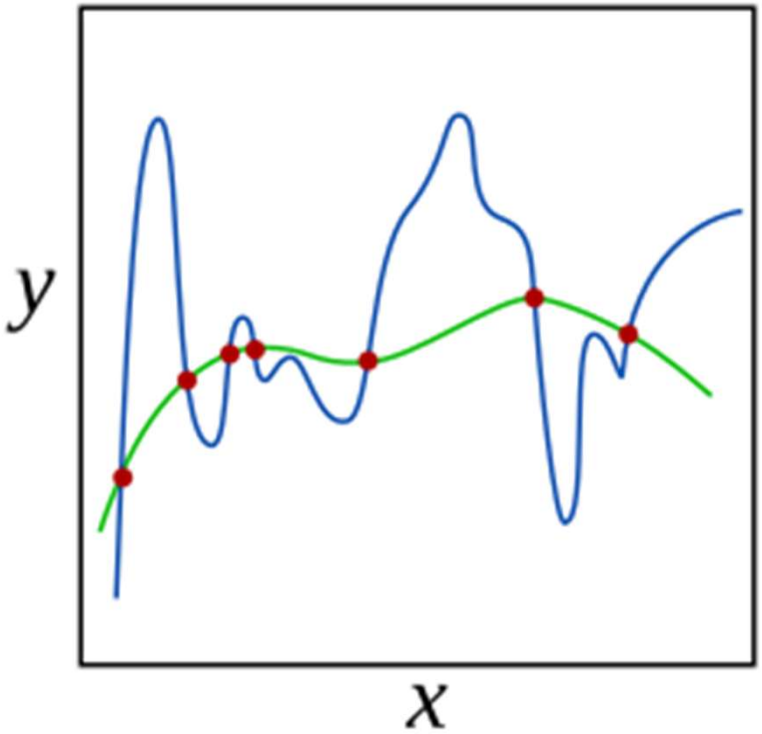
Basic Architecture

- **Overfitting & Regularization**
  - **Train dataset**
    - 머신러닝의 데이터는 보통 매우 많으므로 최적화 위해 과정 나눔.
    - Batch Size, Iteration, Epoch으로 나뉜다.



Basic Architecture

- **Overfitting & Regularization**
  - **Regularization**
    - Overfitting을 줄이는 방법 중 하나로 Regularization이 있다.
    - Train Loss가 증가 가능하지만, Validation Loss, Test Loss를 감소시키는 기법.
    - L1/L2 Regularization, Dropout, Batch Normalization 등이 있다.



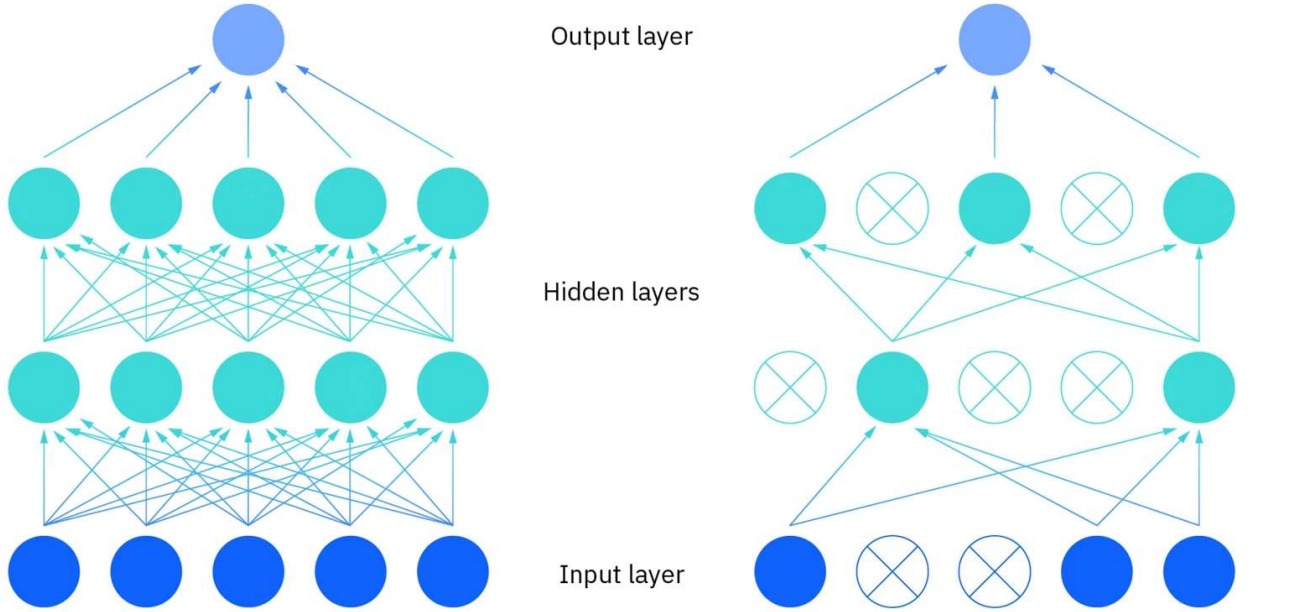
Basic Architecture

- **Overfitting & Regularization**
  - **L1 \ L2 Regularization**
    - 기존의 Loss에 L1\L2 Norm을 더하여 Loss가 작더라도 W가 크면 Cost가 커짐.
    - 즉 Loss에만 비중을 두는 것이 아니라, W에도 비중이 생김.

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y}_i) + \frac{\lambda}{2}|w|^2\}$$

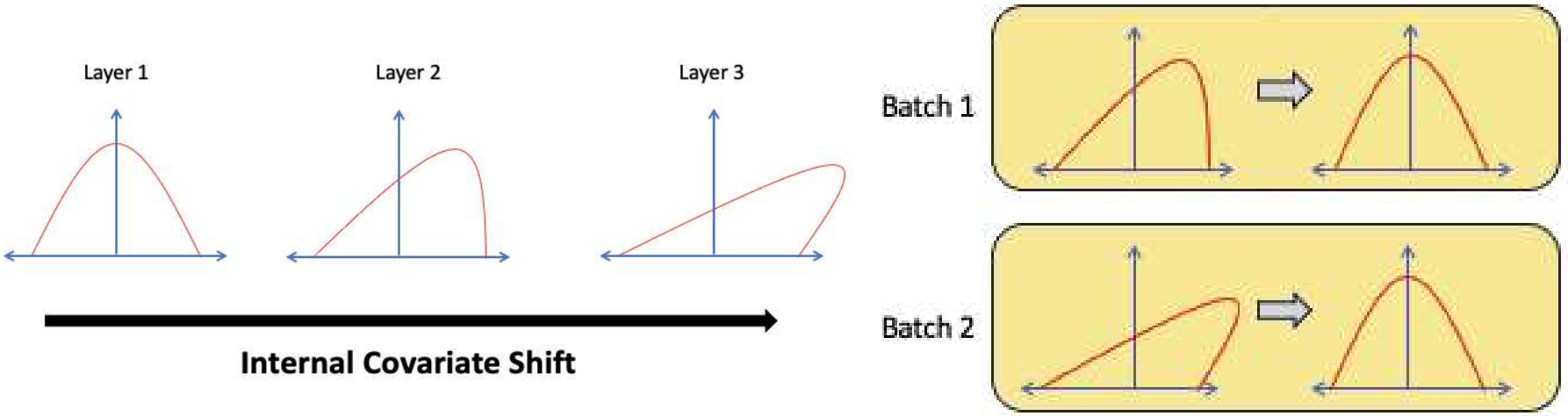
Basic Architecture

- **Overfitting & Regularization**
  - **Dropout**
    - 모든 노드들을 사용하는 것이 아니라, 일부 노드들을 0으로 끄는 작업.
    - 특정 노드가 과도하게 학습되어 강해지는 것을 방지함.
    - dropout은 무작위로 여러 구조를 만들어 general하게 학습시키는 효과가 있음.



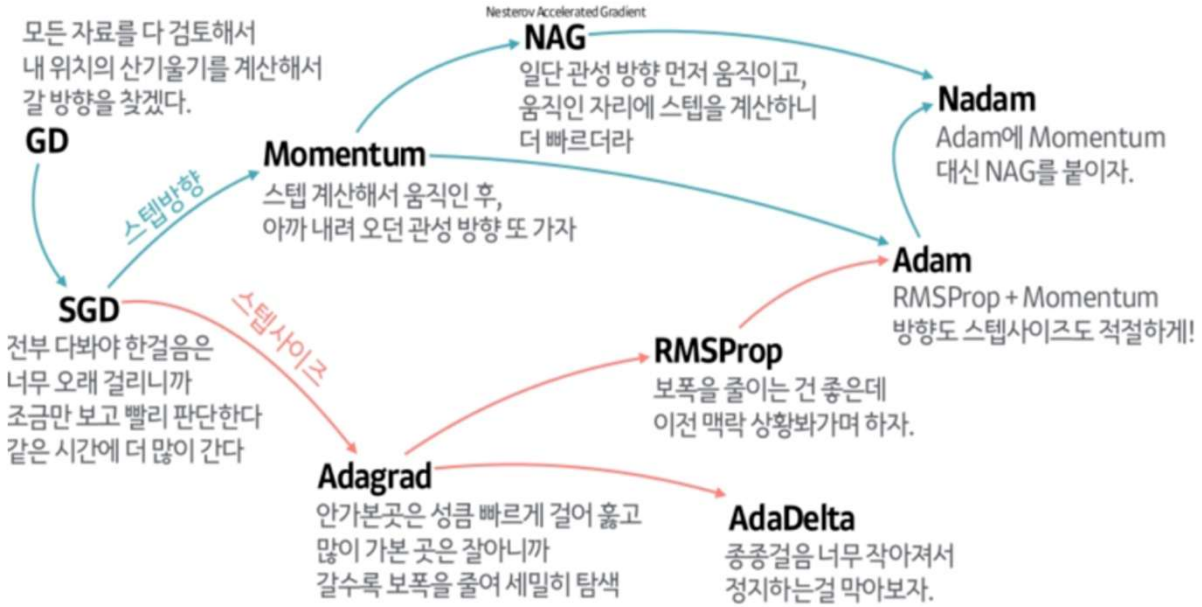
Basic Architecture

- **Overfitting & Regularization**
  - **Batch Normalization(BN)**
    - 레이어를 지날 때마다 출력 값의 데이터 분포가 원본의 데이터 분포와 점차 달라짐.
    - 데이터 분포 변화가 축적되면 기울기가 0으로 수렴되는 데이터들이 점차 생겨 Gradient vanishing를 야기하게 된다.
    - 따라서 기울어진 분포를 가우시안 분포로 만드는 과정 거침.
    - Gradient Vanishing 때문에 LR을 작게 해야만 했던 문제 해결, 학습 속도 개선.



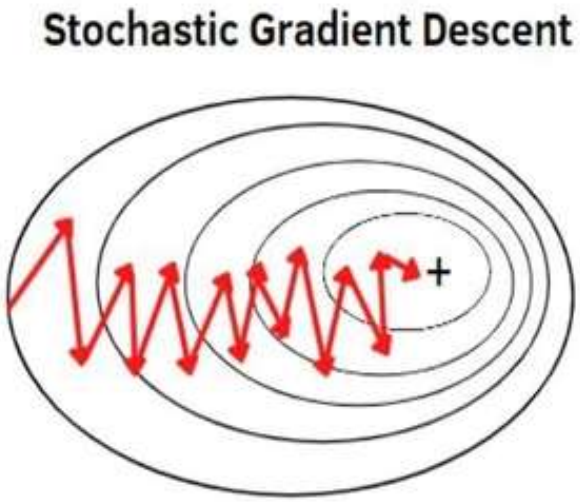
Basic Architecture

- **Optimization & Optimizer**
  - **Optimization**
    - Optimization = Loss function의 최솟값을 찾아 가는 과정
    - Optimizer = Optimization 과정을 효과적으로 수행하는 알고리즘



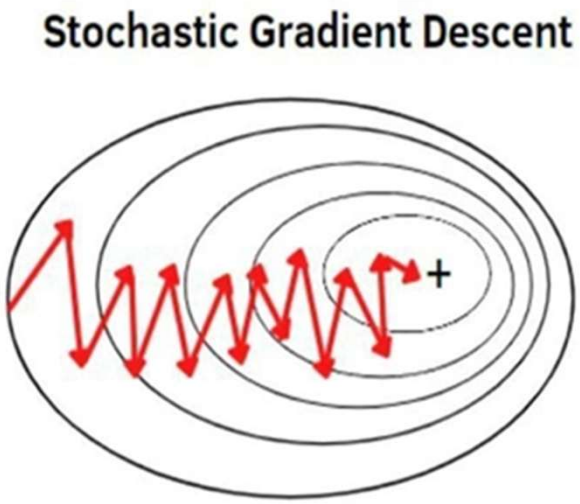
Basic Architecture

- **Optimization & Optimizer**
  - **SGD**
    - 기존 Gradient Descent는 전체 데이터에 대해 gradient를 계산하였음.
    - 데이터 수가 많을 수록 계산량이 늘어 업데이트 속도가 느려지는 문제 직면.
    - 따라서 일부 데이터(Mini-batch)에 대해서만 Gradient를 계산
    - 빠른 속도와 적은 RAM 사용량을 확보 가능했다.



Basic Architecture

- **Optimization & Optimizer**
  - **SGD**
    - 허나 SGD에도 크게 두 가지 문제가 존재.
      - Local Minimum에 멈춰있는 문제
      - Step Size가 모든 파라미터마다 같아 일부 파라미터가 학습이 부족한 채 종료.



Basic Architecture

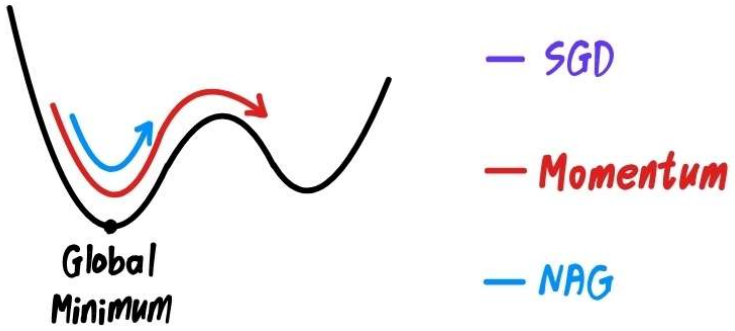
- **Optimization & Optimizer**
  - **Momentum**
    - Local minimum에 갇혀 있을 상황에, 추가적으로 힘을 주어 빠져나오게 함.
    - 마치 관성처럼 이전에 이동하던 속도와 현재의 gradient를 합하여 step.
  - **NAG**
    - Momentum을 먼저 계산한 후, 그 직후 시점에서 gradient를 계산한 것을 더함.
    - Global minimum을 넘어가는 overshooting 문제를 억제할 수 있음.

4) Optimizer

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

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





Basic Architecture

- **Optimization & Optimizer**
  - **AdaGrad**
    - 기존에 많이 업데이트 됐던 파라미터의 Learning Rate를 줄임.
    - 업데이트가 부족한 파라미터의 업데이트를 더 유도할 수 있다.
  - **RMSProp**
    - Gradient를 계속해서 누적합시, 결국 분모가 커져 Learning Rate가 0으로 수렴하게 되는 결과 발생.
    - 기존 누적 메커니즘과 달리, Gt-1과, 현재 기울기에 0~1의 계수를 부여하여 누적된 합계가 너무 커지지 않게 함.

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

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

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

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

Basic Architecture

- **Optimization & Optimizer**
  - **ADAM**
    - NAG와 RMSProp의 장점을 합친 것.
    - Momentum과 Adaptive Learning Rate(ALR)를 통해 Learning Rate를 조절.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

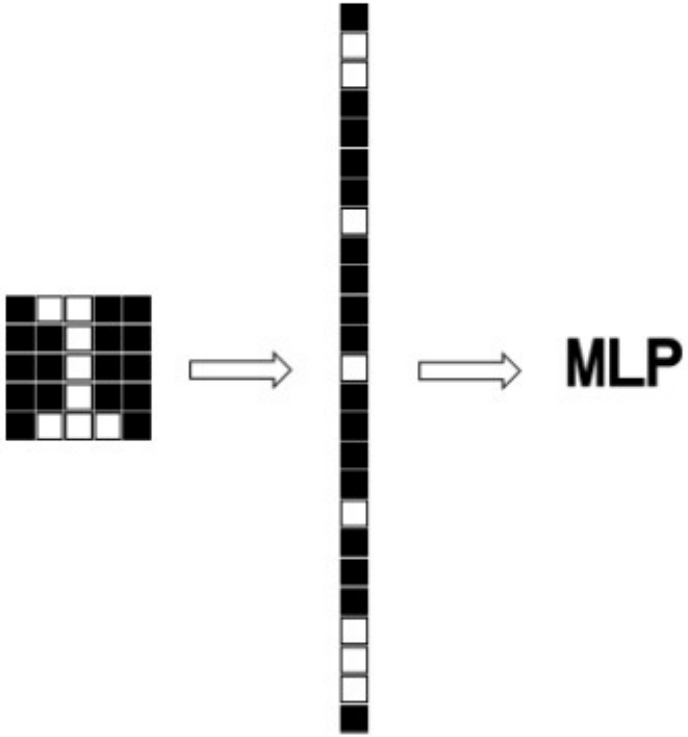
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

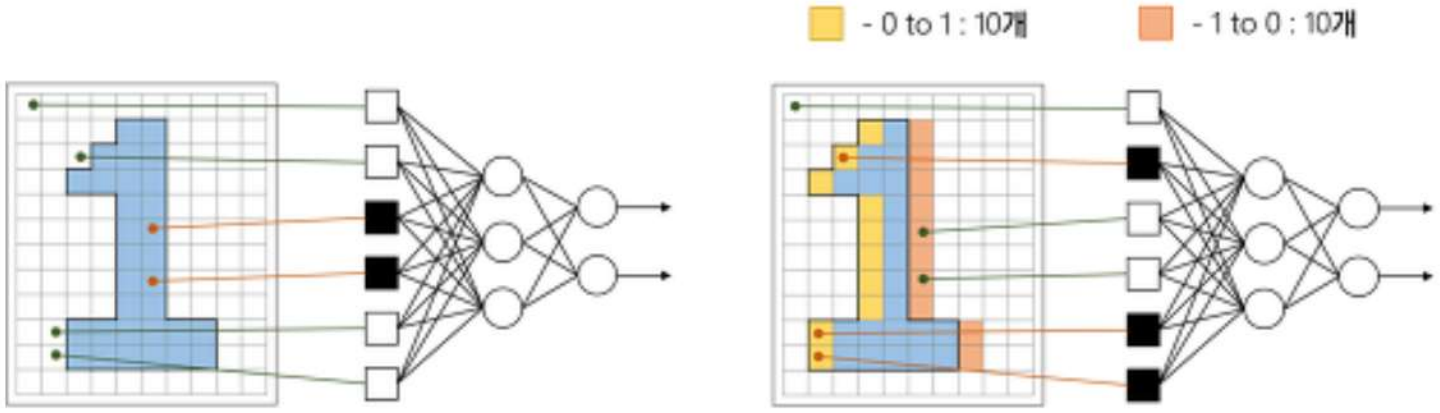
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **MLP의 한계**
    - MLP가 이미지를 처리할 때, 각 픽셀을 일렬로 펼쳐 입력으로 사용.
    - 이는 이미지의 공간적인 정보를 파괴.
    - 공간적 특징을 고려하지 않는 무의미한 파라미터 연결을 야기.



Fundamental Neural Networks

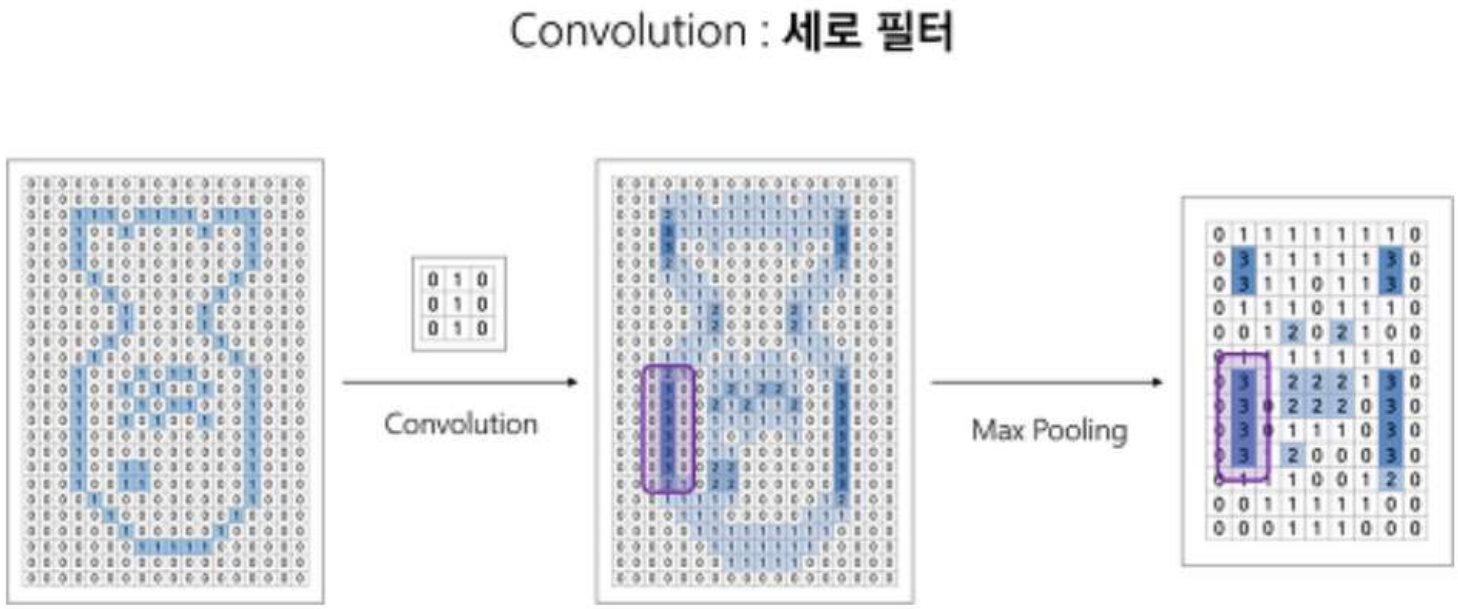
- **CNN (Convolutional Neural Network)**
  - **MLP의 한계**
    - 같은 모양의 이미지지만 왼쪽으로 한 픽셀만 움직였을 때의 변화.
    - 한 픽셀만 움직이는데도, 인풋값이 20개가 바뀌어 학습된 weight가 무의미함.



한 칸씩만 움직였는데
변화하는 인풋값이 20개

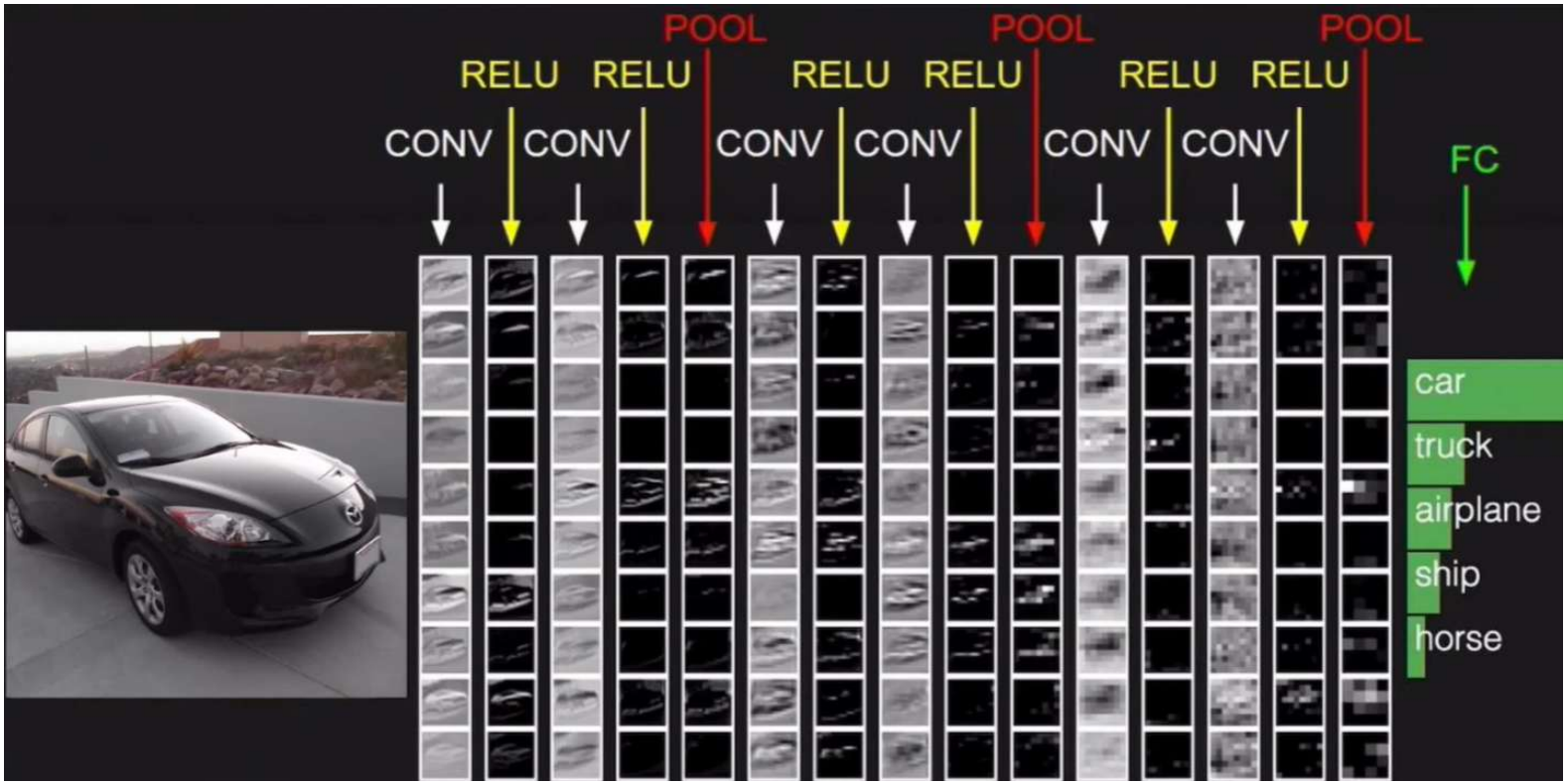
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **CNN의 도입**
    - 따라서 이미지의 구조를 보존하기 위해, 필터를 움직이면서 특징을 분석.
    - 어떤 특징이 어느 위치에 있는지 확인할 수 있음.



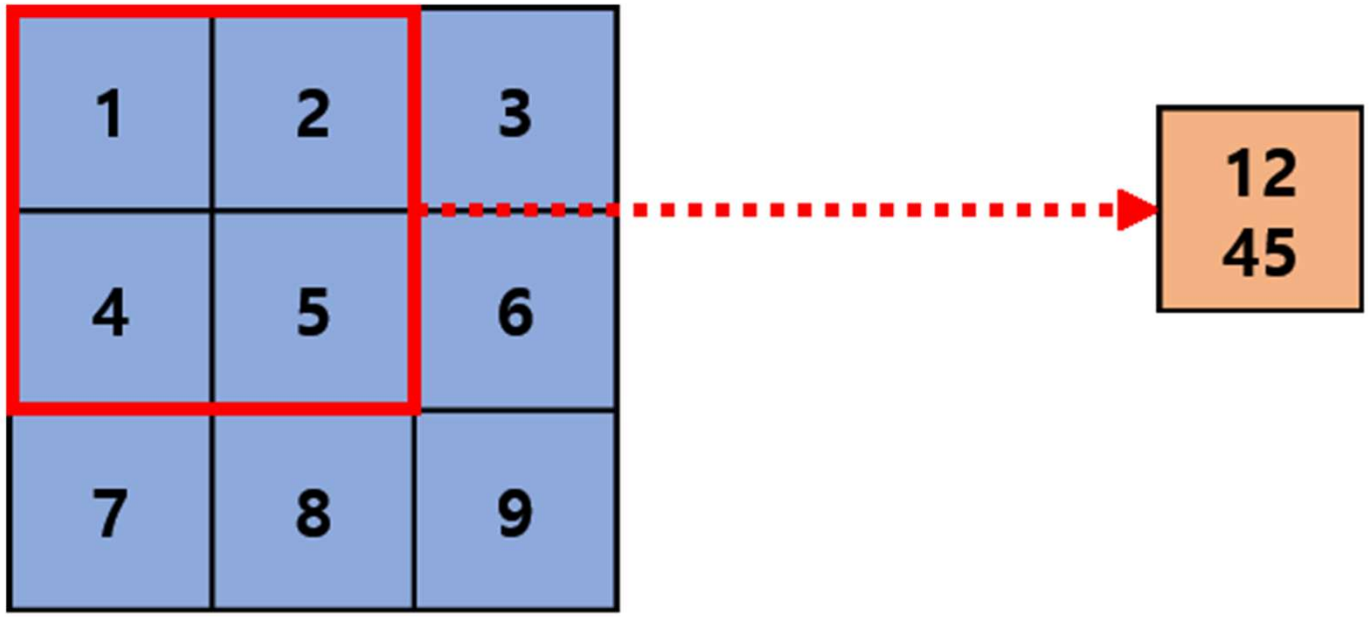
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **CNN의 구조**
    - Conv – ReLU – Pool 레이어의 반복.
    - 마지막에 FC로 마무리.



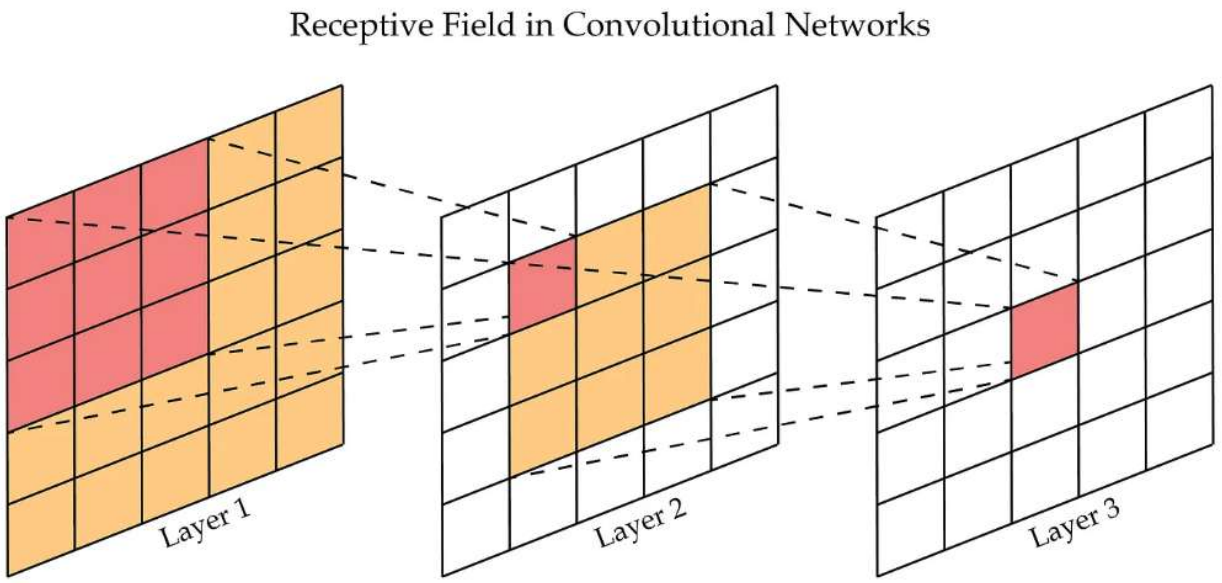
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **Convolution Layer**
    - N x N의 마스크를 씌워서 슬라이딩하여 Dot product.
    - 마스크 하나 당 depth는 하나. 즉, n개의 마스크를 통해 conv시 다음 레이어의 인풋 채널은 n.



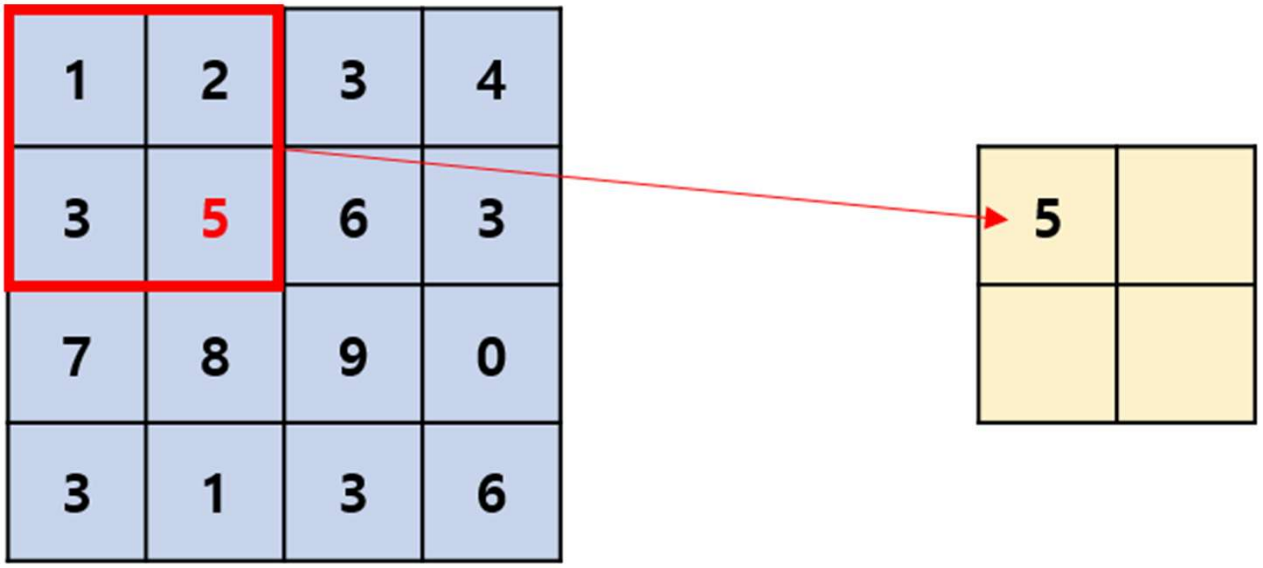
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **Receptive Field**
    - 출력레이어의 뉴런 하나에 영향의 미치는 입력레이어 뉴런의 공간크기.
    - 즉, 각 레이어에서 마스크가 원본이미지를 보는 크기.
    - Layer가 깊어질 수록 receptive field가 커지게 되는 것이 바람직.



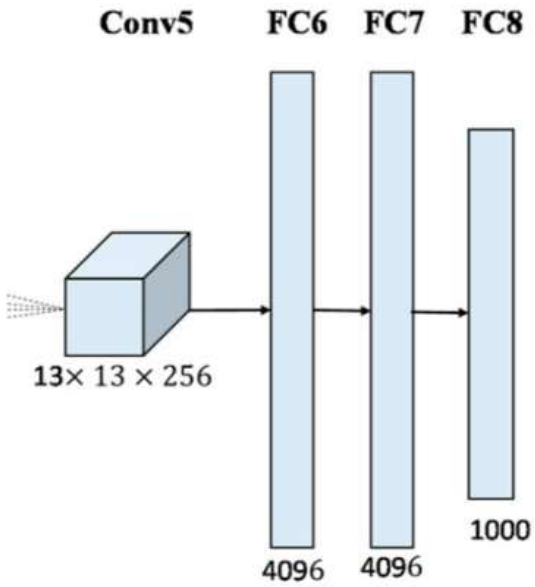
Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **Pooling**
    - 따라서, 효율적으로 receptive field를 키우기 위해 이미지 크기를 압축.
    - Pooling mask를 이용해 계산하여 이미지를 압축.



Fundamental Neural Networks

- **CNN (Convolutional Neural Network)**
  - **Fully Connected layer(FC)**
    - 여러 Channel로 된 layer들을 1차원 벡터로 펼쳐내는 과정.
    - Fc 이후, Softmax를 통해 결론을 내기 위한 작업.



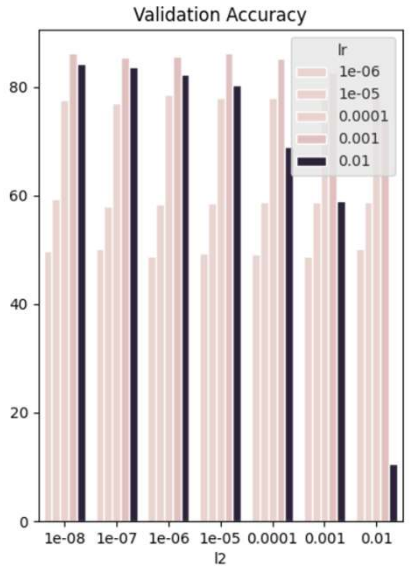
Fundamental Neural Networks

- **Cifar 10 by VGGNet**
  - **Practice Result**

<Val Acc(%)>

l2 \ lr	0.0001	0.001	0.01
1e-06	78.62	85.60	82.34
1e-05	77.94	**86.23**	80.33
0.0001	78.03	85.28	68.89

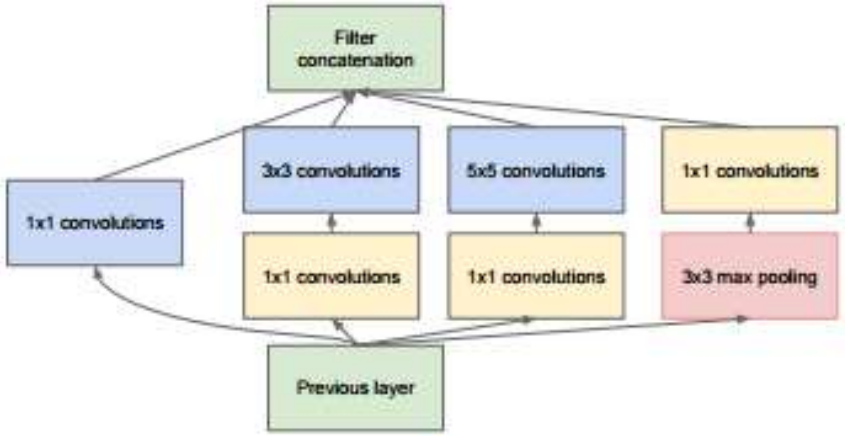
Optimizer: Adam
Batch Normalization: O
Epochs : 100
Model : VGG19
Active Function : ReLU



- 실습 코드를 간단하게 돌려본 결과.
- LR : 0.001, L2 : 1e-05의 Test Acc : 85.74%

Fundamental Neural Networks

- **GoogLeNet & ResNet**
  - **VGGNet의 한계**
    - 깊은 depth의 컨볼루션 레이어 결과들을 Fc레이어에서 flatten하여 계산.
    - 이 과정에서 파라미터 수가 폭발적으로 증가하여 메모리 부족 문제 발생.
  - **GoogLeNet의 등장**
    - 1x1 컨볼루션 필터를 통해 채널 수를 줄이고 다음 컨볼루션 필터 통과.
    - 채널 수를 줄이고 다음 컨볼루션을 거침으로써 연산량 크게 줄임.

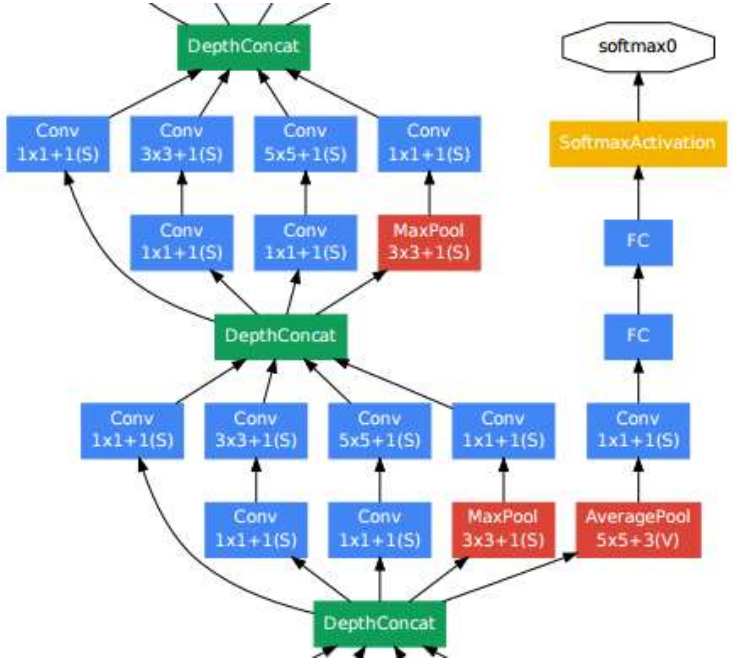


(b) Inception module with dimensionality reduction

Figure 2: Inception module

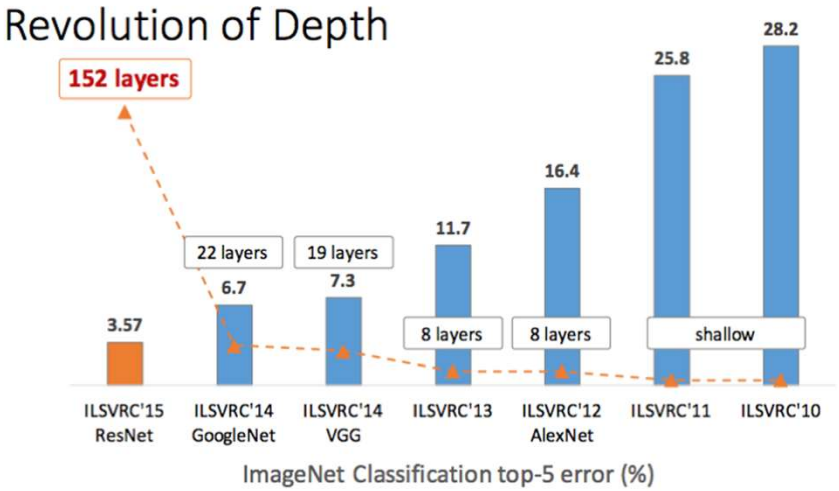
Fundamental Neural Networks

- **GoogLeNet & ResNet**
  - **GoogLeNet의 등장**
    - 레이어가 깊어질 수록 역전파 시, 앞 레이어의 Gradient Vanishing 문제 발생.
    - 따라서, Gradient Vanishing 되기 전, 중간마다 Gradient를 추가해 주는 모듈 추가.



Fundamental Neural Networks

- **GoogLeNet & ResNet**
  - **ResNet의 등장**
    - 하지만 이전 모델들은 여전히 레이어가 깊어질 수록, 학습이 어려워 진다는 문제 존재.
    - 이를 개선하고자 skip connection이라는 기술 도입.
    - 레이어 사이의 변화량이 적어지게 유도하여 레이어 간 서서히 학습되게 유도.
    - 이를 통해 레이어가 대량으로 늘어나도 에러가 크지 않을 수 있었음.



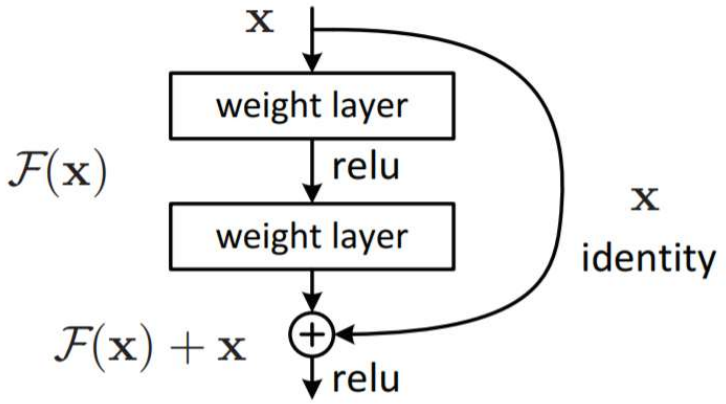
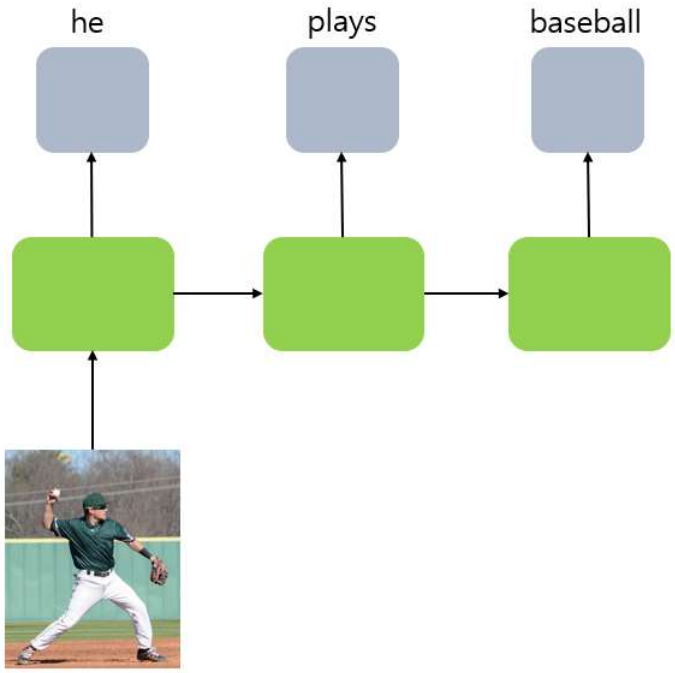


Figure 2. Residual learning: a building block.

Transformer
(RNN & All You Need Is Attention)

Transformer

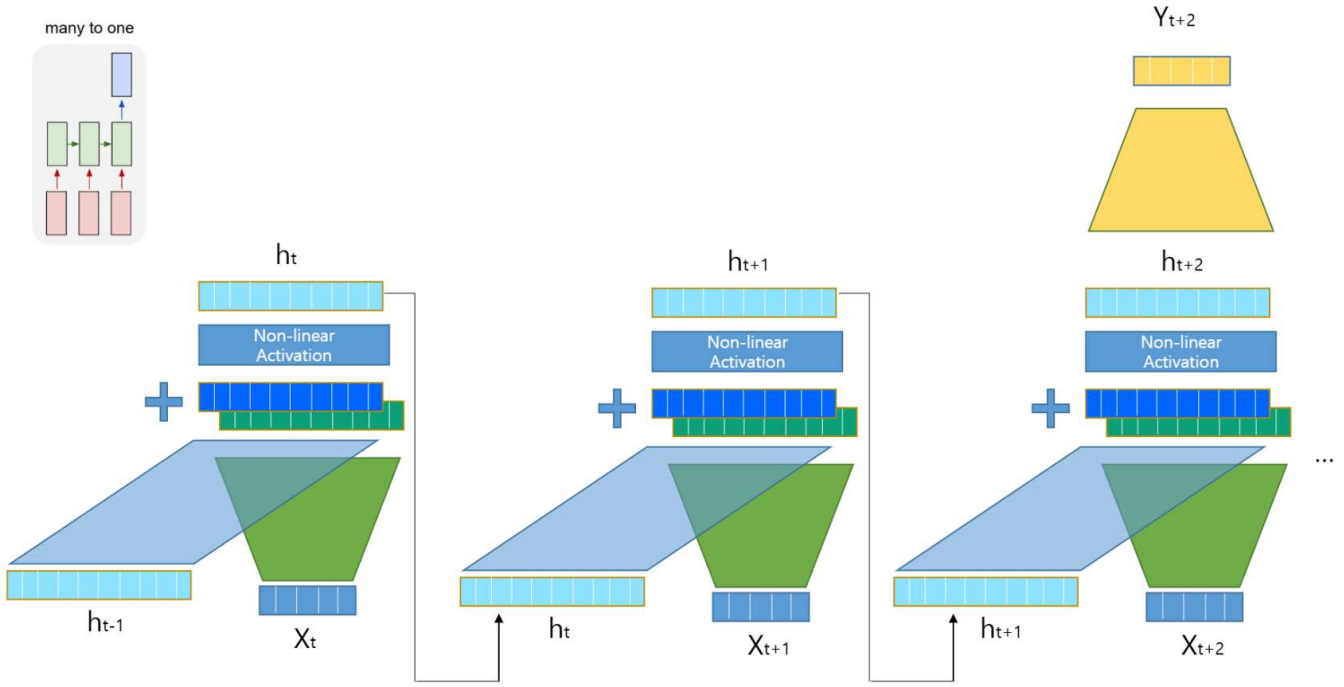
- **Recurrent Neural Network (RNN)**
  - **RNN의 등장**
    - Sequential한 데이터를 다루기 위한 모델
    - 과거의 정보를 사용하여 현재 및 미래의 입력에 대한 신경망의 성능 개선.



일 대 다(one-to-many)

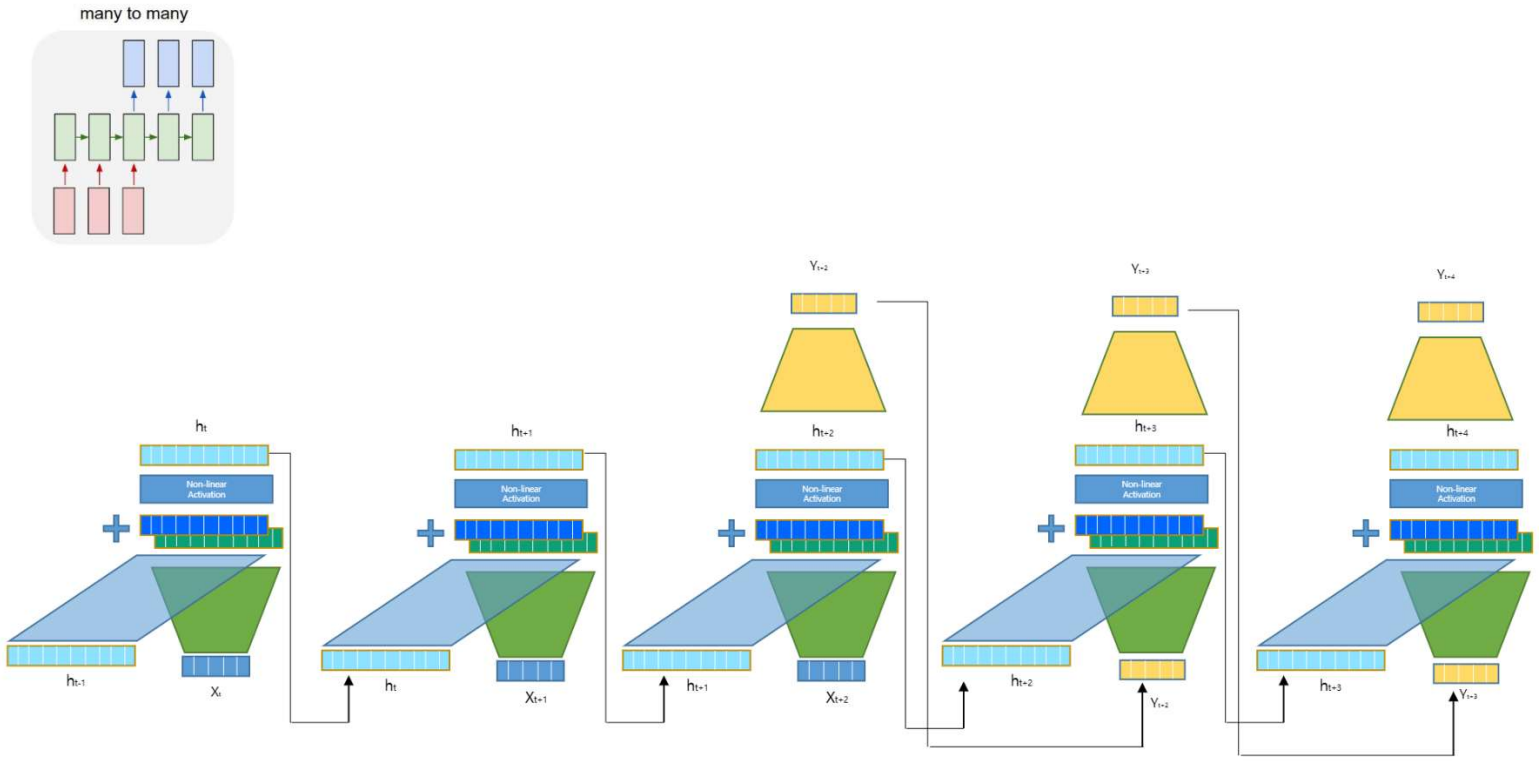
Transformer

- **Recurrent Neural Network (RNN)**
  - **RNN의 메커니즘**
    - 이전 레이어의 결과를 입력으로 가져와 계산. (파란색 벡터)
    - 추가적으로 현재의 입력 또한 계산. (초록색 벡터)
    - 계산된 결과(하늘색 벡터)를 통해 원하는 시점에서 출력으로 도출.(노란색 벡터)
    - 원하는 형태의 출력으로 만들어 내는 모듈(노란색 모듈) 필요.



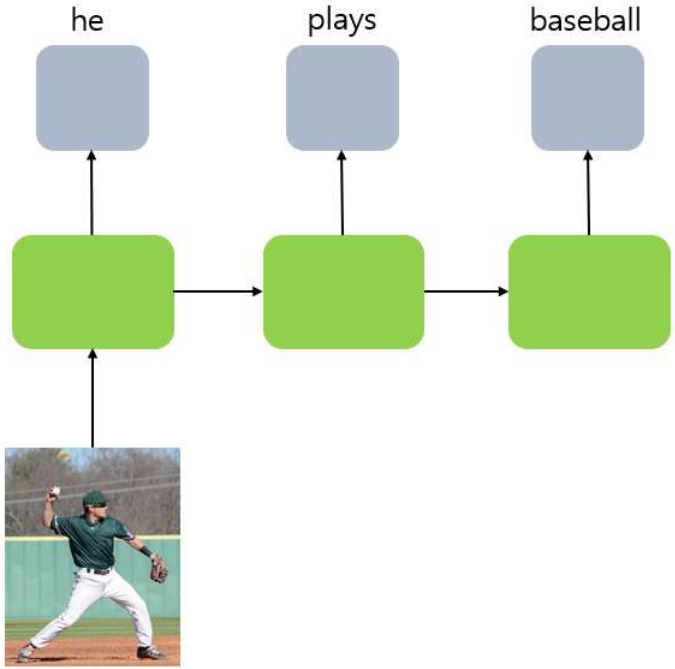
Transformer

- **Recurrent Neural Network (RNN)**
  - **RNN의 메커니즘**
    - 목적에 따라 여러가지 구조로 나뉨.
    - Many to One, Many to Many, One to Many 등등



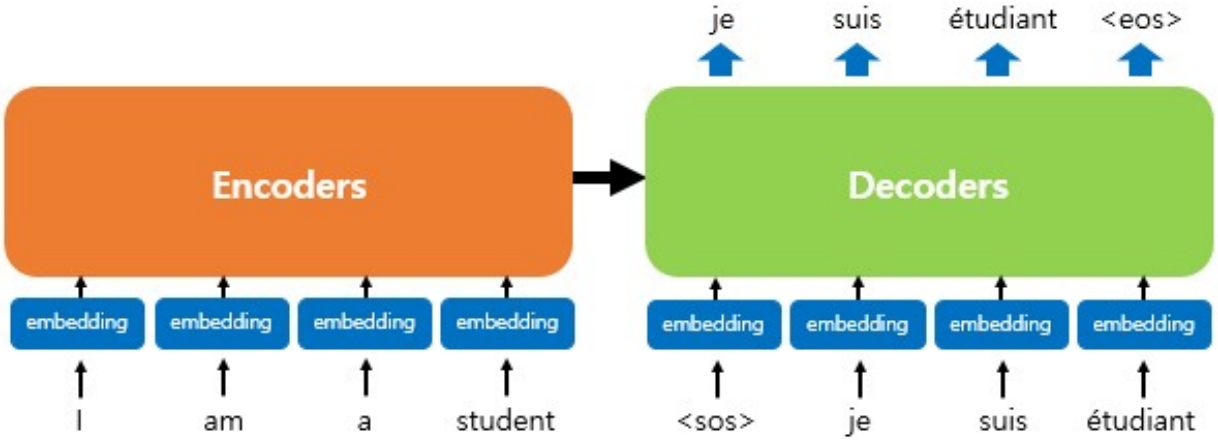
Transformer

- **Transformer란?**
  - **RNN의 한계**
    - RNN들은 직렬, 순차적 계산구조.
    - 이전 단계에서 계산이 되어야만 다음 단계에서 계산이 가능.
    - 문장이 길어질 수록 계산시간과 메모리 사용량이 커지는 문제 직면.
    - 이를 해결하고자 병렬적 계산으로 연산을 한번에 계산하려 함.



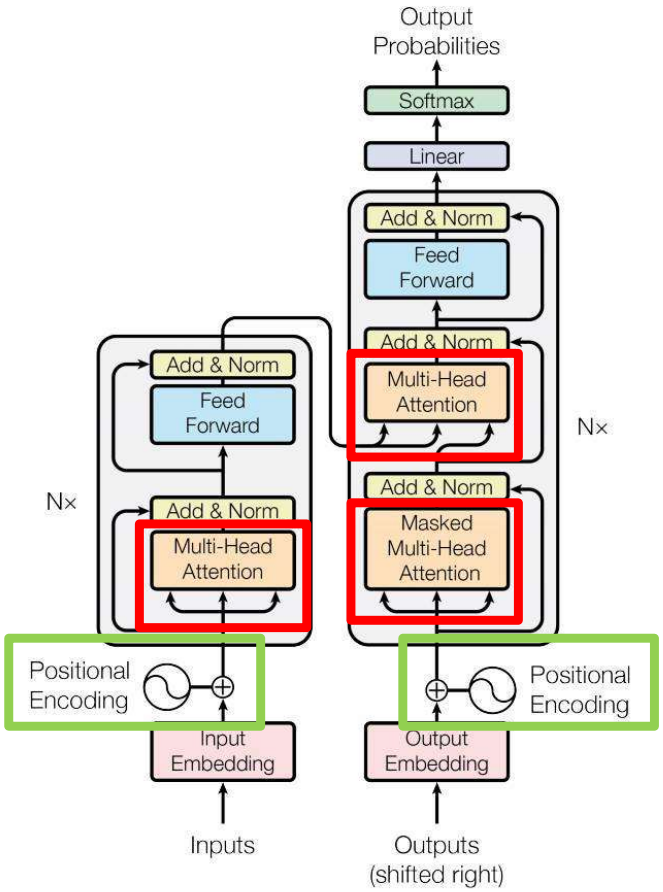
Transformer

- **Transformer의 전체 구조**
  - **인코더와 디코더**
    - Transformer는 크게 Encoder와 Decoder로 구성됨.
    - 영어를(input) 프랑스어로(output) 번역하는 상황을 가정.
    - 인코더에선 영어 문장의 문맥 파악.
    - 디코더에선 기존에 생성한 프랑스어 문장의 문맥을 파악.
    - 또한 인코더에서 나온 정보와 영어 문장의 문맥정보를 바탕으로 다음 단어에 대한 정보 산출.



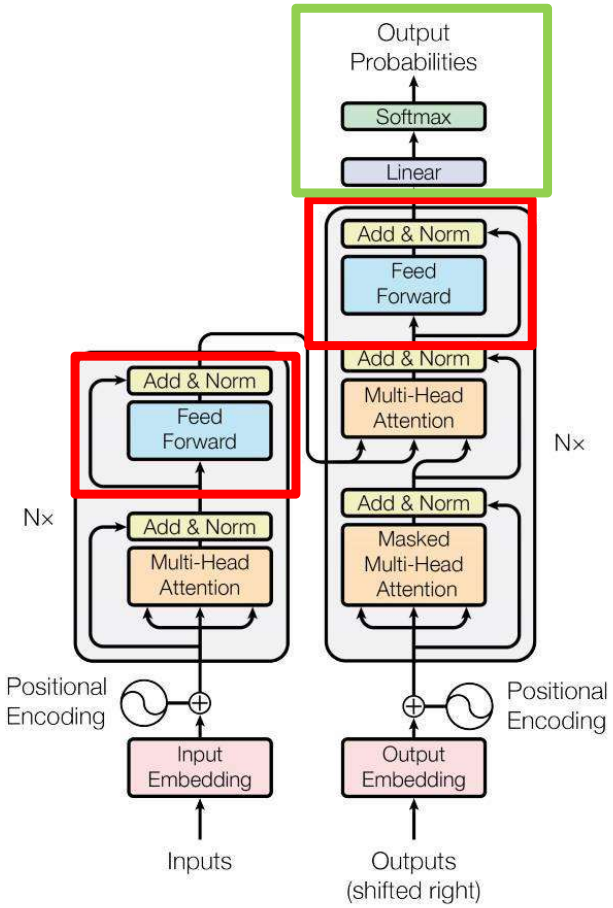
Transformer

- **Transformer의 전체 구조**
  - **Transformer의 순차적인 흐름**
    - 인풋 문장과 이전에 만들어 낸 아웃풋 문장 속 단어 순서를 정하는 Positional Encoding.
    - 문맥을 파악하고 정보를 추출하는 Multi-Head Attention.



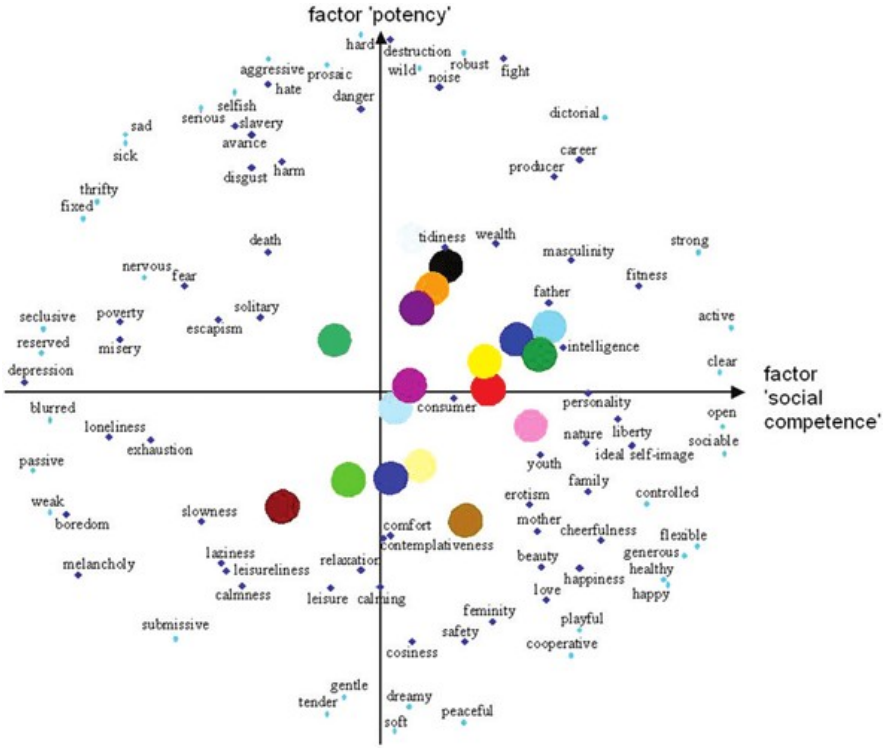
Transformer

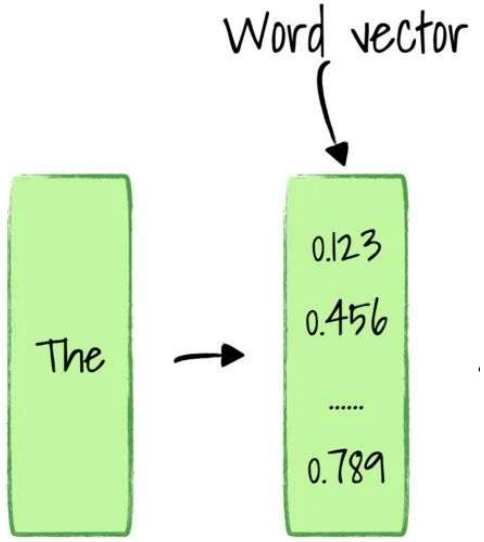
- **Transformer의 전체 구조**
  - **Transformer의 순차적인 흐름**
    - Multi-Head Attention이 끝난 후 Residual Connection과 FC를 실행.
    - 디코더에서 추출한 정보를 FC하고, Softmax하여 최종적으로 나올 단어 결정.



Transformer

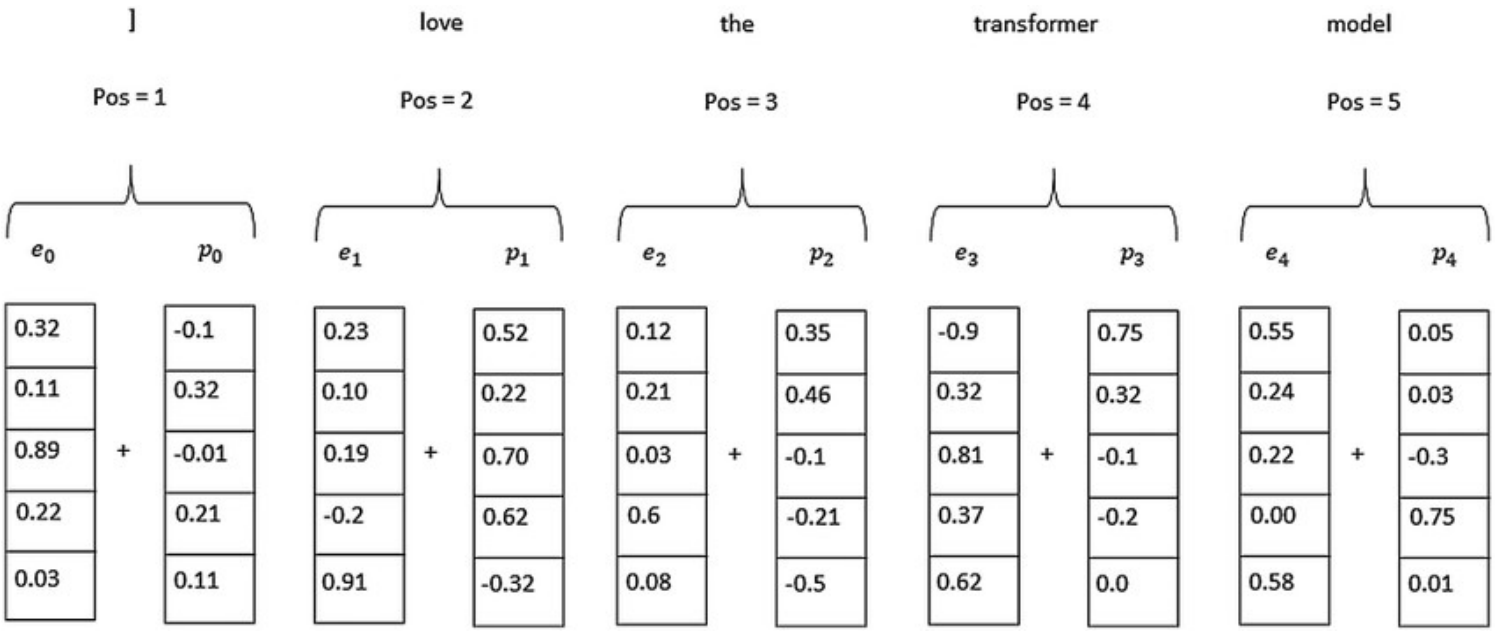
- **Positional Encoding**
  - **Embedding Vector**
    - 자연어 단어를 벡터로 수치화 한 것.
    - 문장 내 단어들을 수치적으로 계산할 수 있게 됨.





Transformer

- **Positional Encoding**
  - **Why Positional Encoding?**
    - Transformer는 상술했듯, RNN을 모두 제거한 구조.
    - RNN을 제거하여 문장 내 단어들의 위치 정보를 알 수 없게 됨.
    - 따라서, 문장 내 단어들의 위치 정보를 미리 넣어주는 과정을 거침.



Transformer

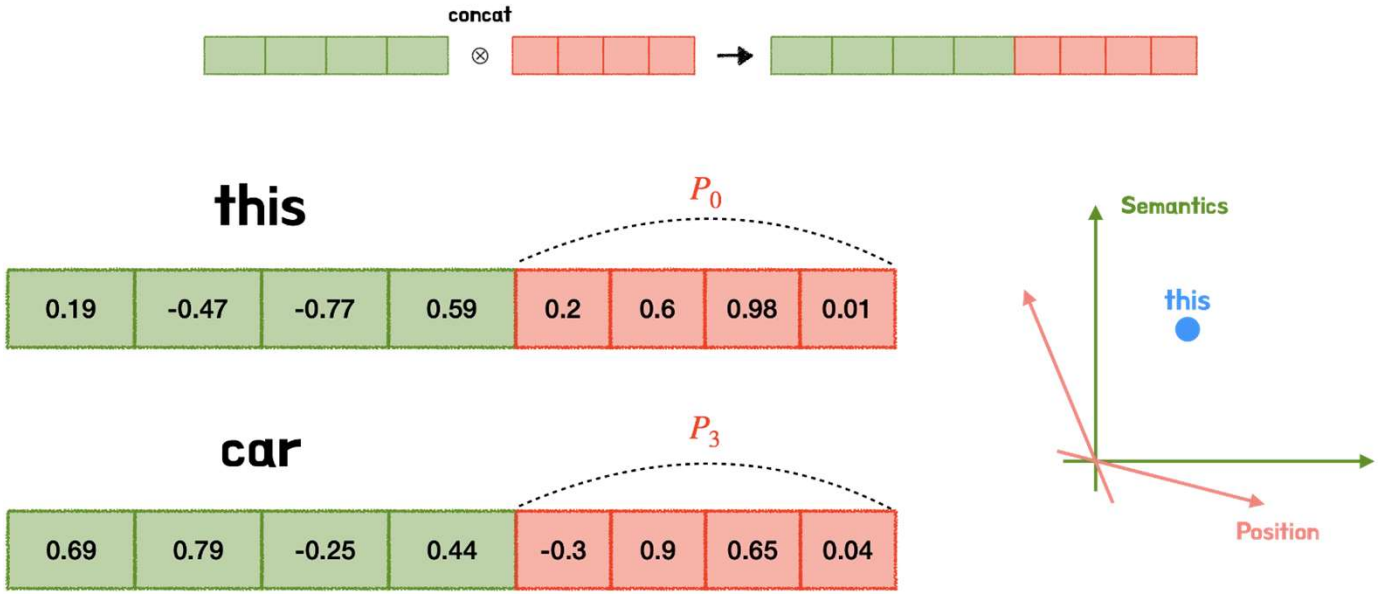
- **Positional Encoding**
  - **Positional Encoding**
    - 임베딩 벡터에 위치 벡터 값들을 그대로 더함.
    - Cos, Sin 함수의 주파수를 달리 하며 위치 정보를 기입.

$$PE_{(pos,\,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,\,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Transformer

- **Positional Encoding**
  - **Why Adding?**
    - 임베딩 벡터에 위치 벡터를 더하면 단어의 의미가 훼손되지 않나?
    - 차라리 concat을 하는 것이 낫지 않나?



Transformer

- **Positional Encoding**
  - **Why Adding?**
    - Concat시 벡터 차원이 두배가 되어 연산량이 매우 커지게 됨.
    - 행렬 공식상 (A+B)W = AW + BW 이므로, 가중치는 의미와 위치 모두를 학습 가능.



Transformer

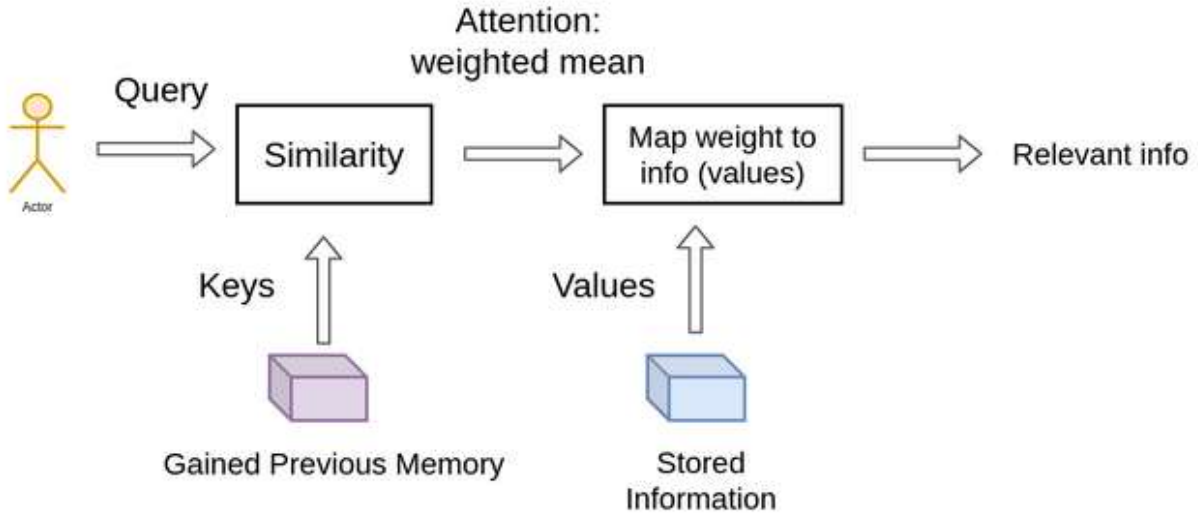
- **Positional Encoding**
  - **Why Adding?**
    - 임베딩 벡터에 더하기 전 약 20배정도 스케일링하여 임베딩 벡터의 크기를 키움.
    - 위치 벡터의 크기는 -1~1 이므로 의미 정보에 큰 영향을 미치지 못함.

3.4 Embeddings and Softmax

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [30]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

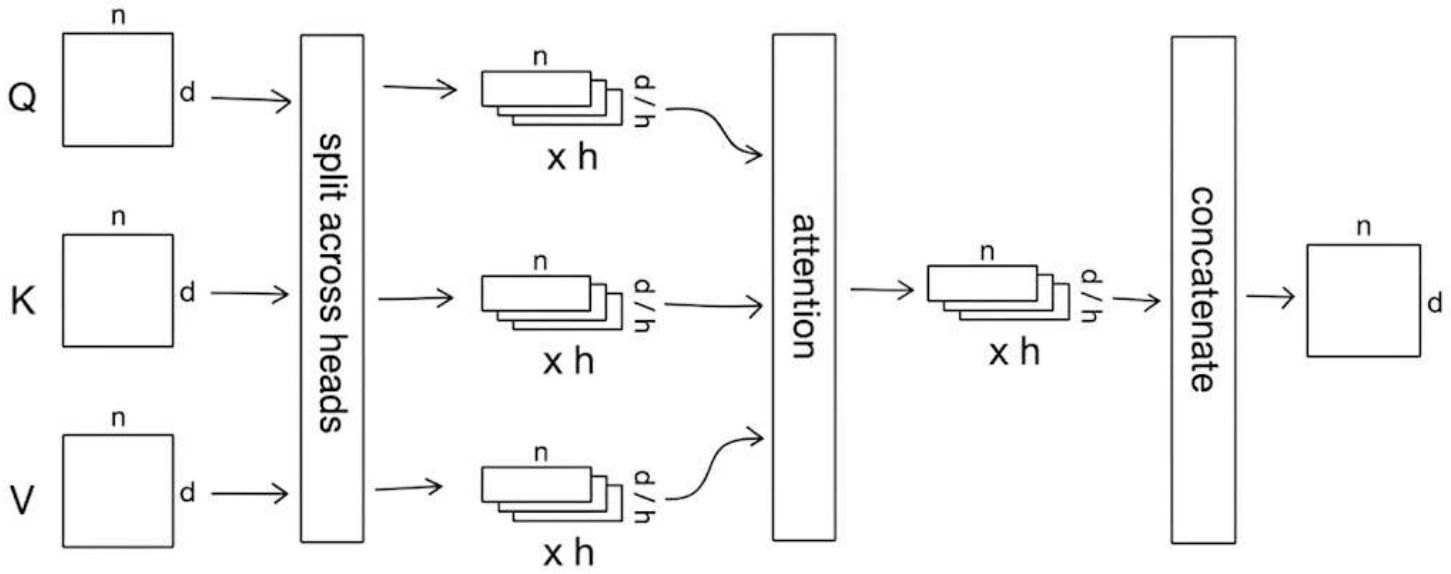
Transformer

- **Multi-Head Attention**
  - **Attention**
    - 문장을 Query(Q), Key(K), Value(V)를 이용하여 이해하고 예측하는 과정.
    - Transformer에선 RNN을 전부 제거하고 Attention만으로 문장을 구성.
      - Query란 문장 내 단어들이 문맥을 파악하기 위해 스스로 정보를 요구하는 숫자.
      - Key란 Query가 문장 내 어떤 단어와 연관성이 높은 지 판별.
      - Value란 단어들의 실제 의미를 담고 있는 정보.



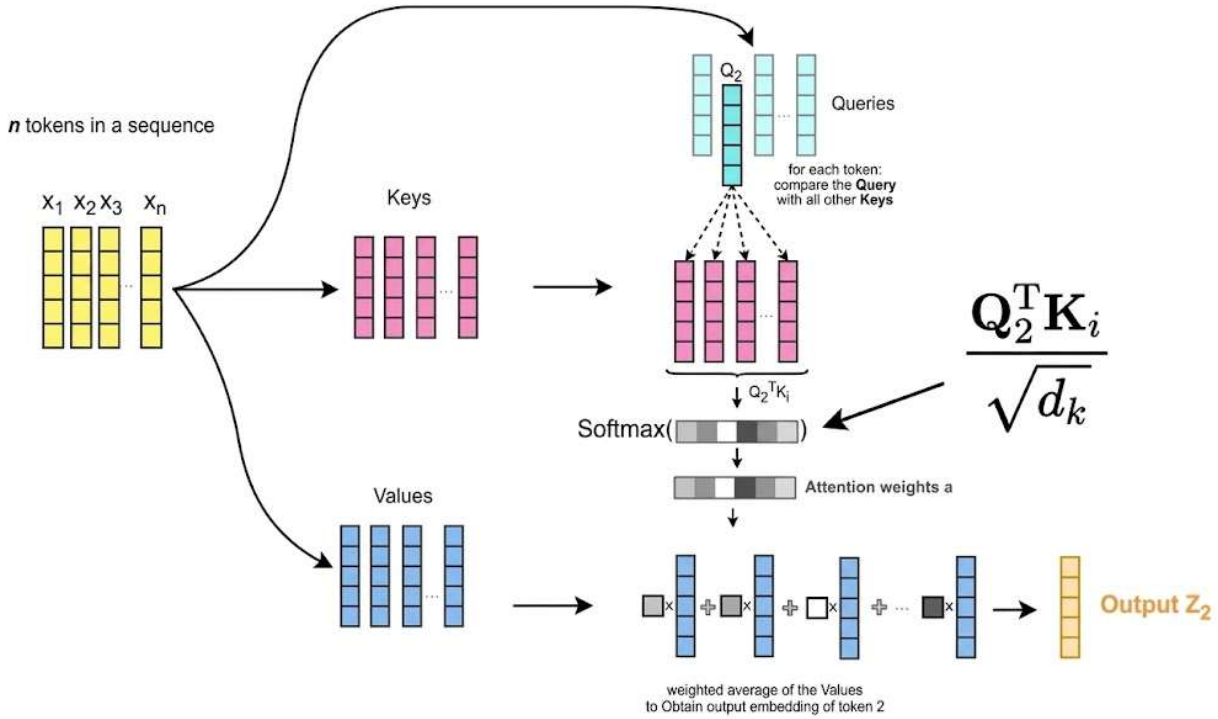
Transformer

- **Multi-Head Attention**
  - **Benefit of Multi-Head Attention**
    - 기존의 높은 차원의 인풋을 작은 차원의 헤드 여러 개로 나누어 feature 추출.
    - 다양한 feature들을 같은 연산량에서 확인할 수 있는 장점.



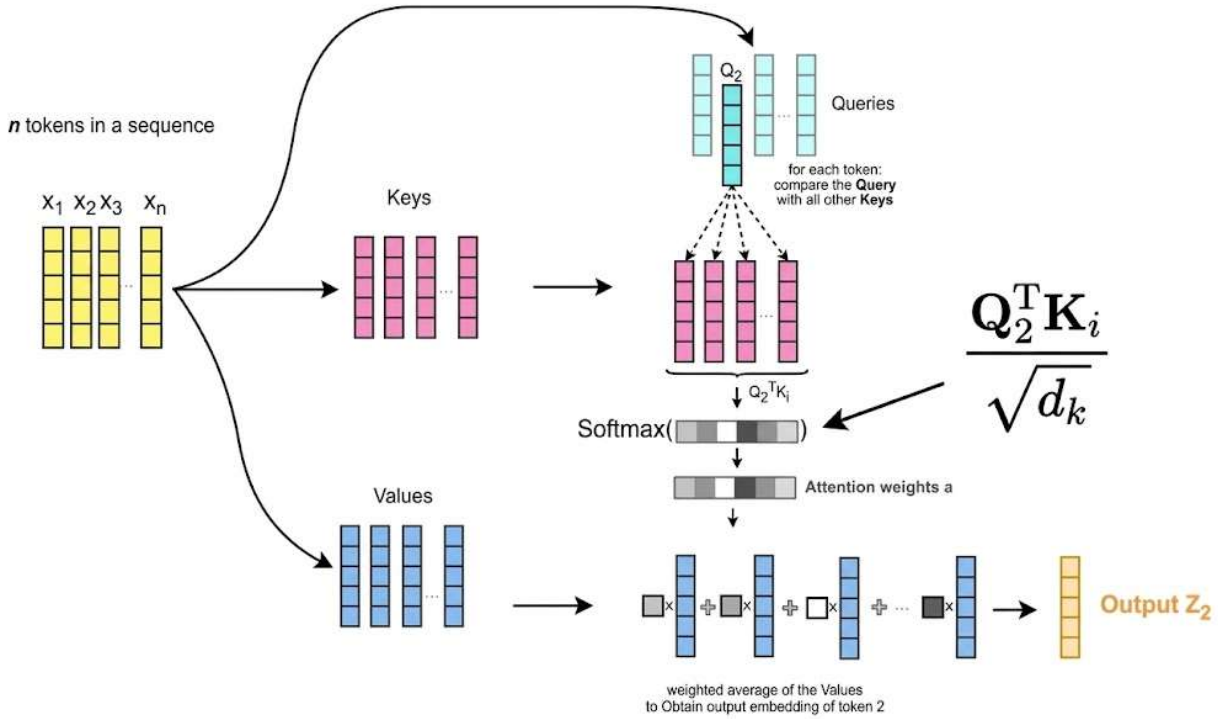
Transformer

- **Multi-Head Attention**
  - **Scaled Dot-Product Attention**
    - 임베딩 벡터를 세 개의 가중치를 통과시키면 Q, K, V가 생성된다.
    - Q와 K를 가중합하여 Attention Score를 만들어 낸다.



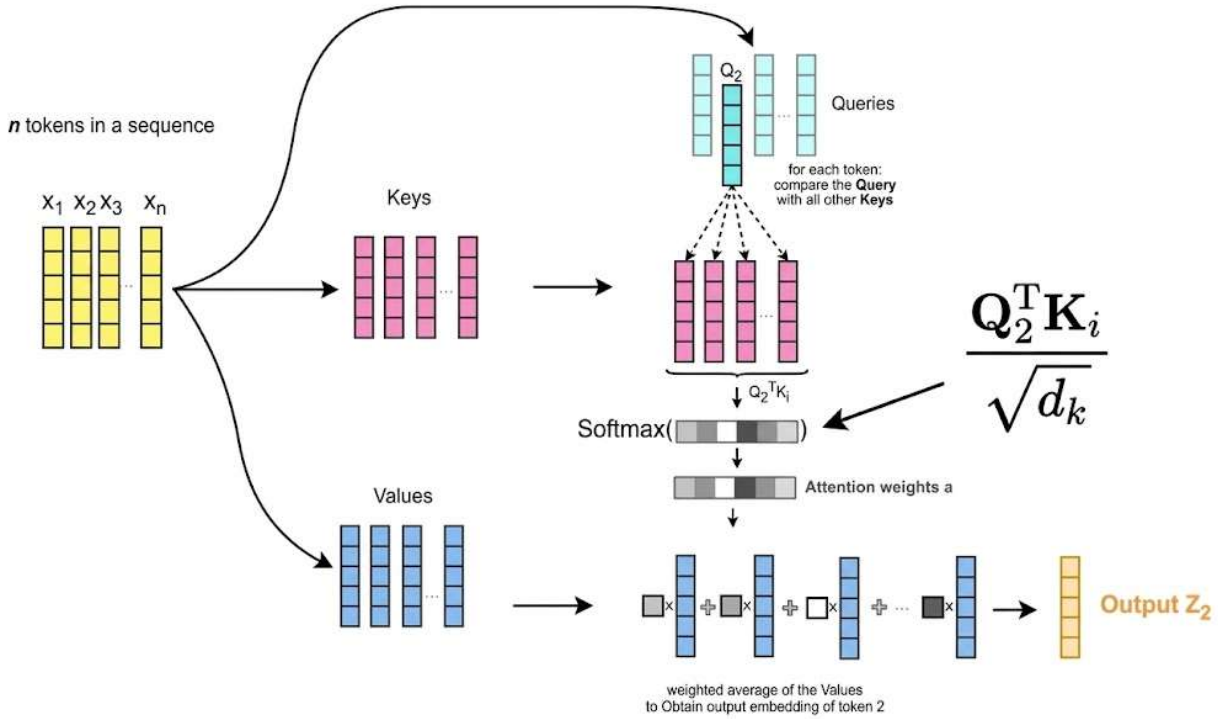
Transformer

- **Multi-Head Attention**
  - **Scaled Dot-Product Attention**
    - Attention Score를 Softmax에 넣기 전, 값이 커지면 기울기가 사라질 가능성이 있음.
    - 따라서 그 값에 $\sqrt{d_k}$를 나누고 Softmax를 거치어 Attention Weight 생성.



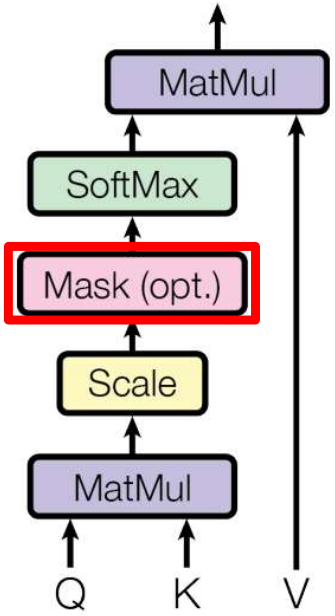
Transformer

- **Multi-Head Attention**
  - **Scaled Dot-Product Attention**
    - 마지막으로 그 값을 V와 가중합함으로써 Context Vector 결정.



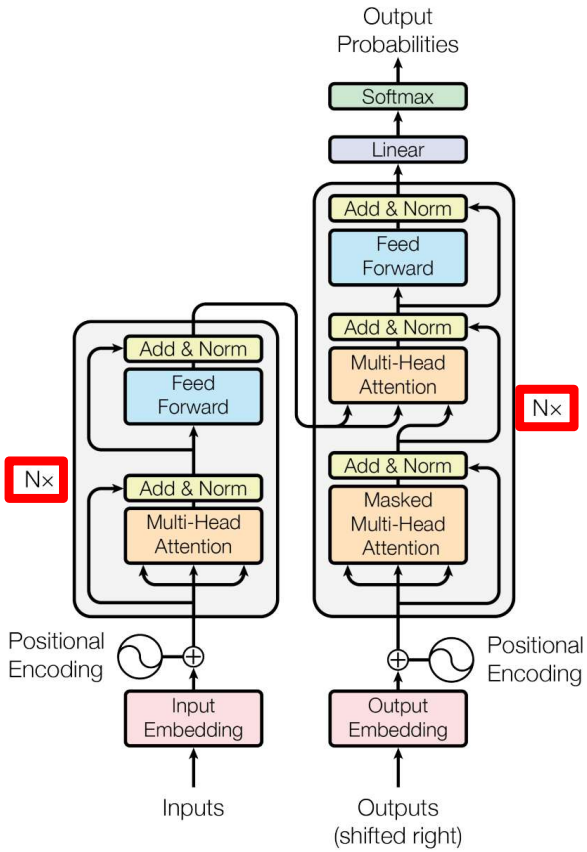
Transformer

- **Multi-Head Attention**
  - **Masking Layer**
    - 후술할 Decoder Self Attention에선 추가적으로 Masking Layer를 거침.
    - 이는 머신을 학습시킬 때 미래 단어에 대한 정보를 가리기 위함.



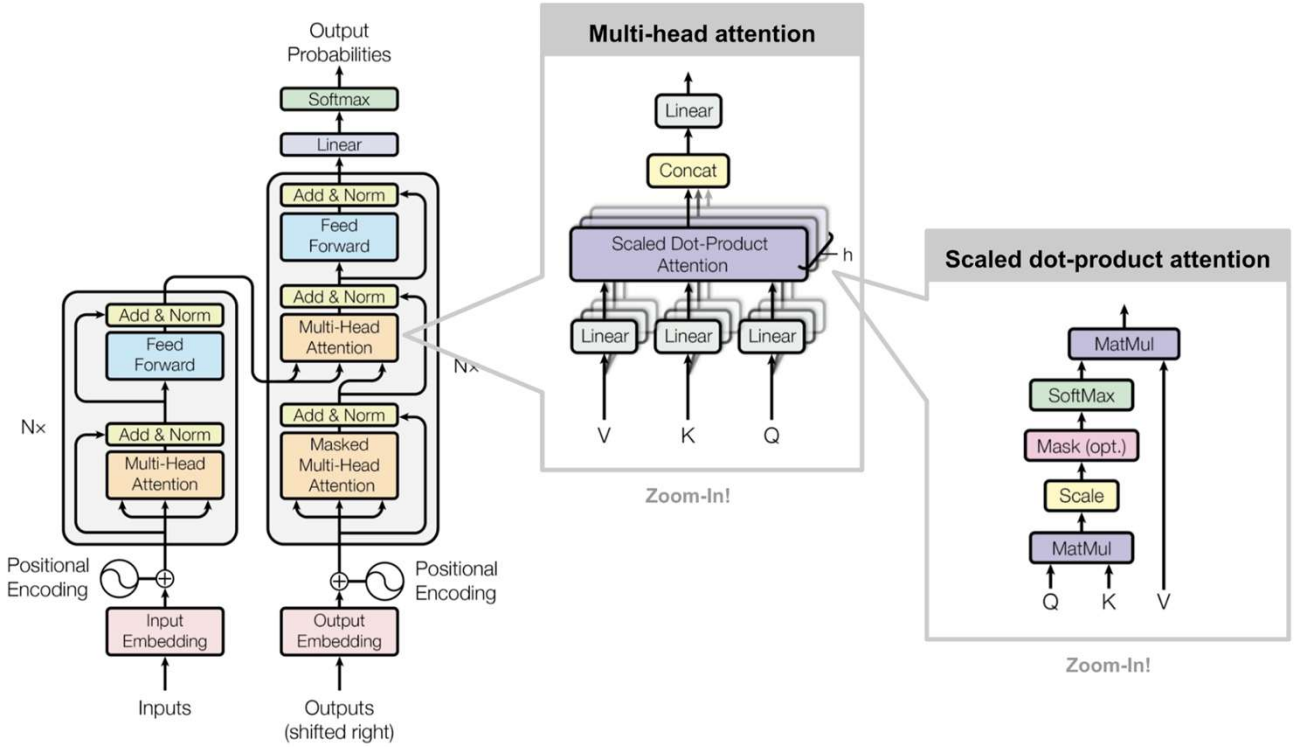
Transformer

- **Multi-Head Attention**
  - **Layer Amount of Attentions**
    - 이는 하나의 헤드 안에서 N만큼 해당 레이어(회색 영역)를 반복한 것.
    - 레이어를 여러 개 쌓음으로써 model capacity 증가.



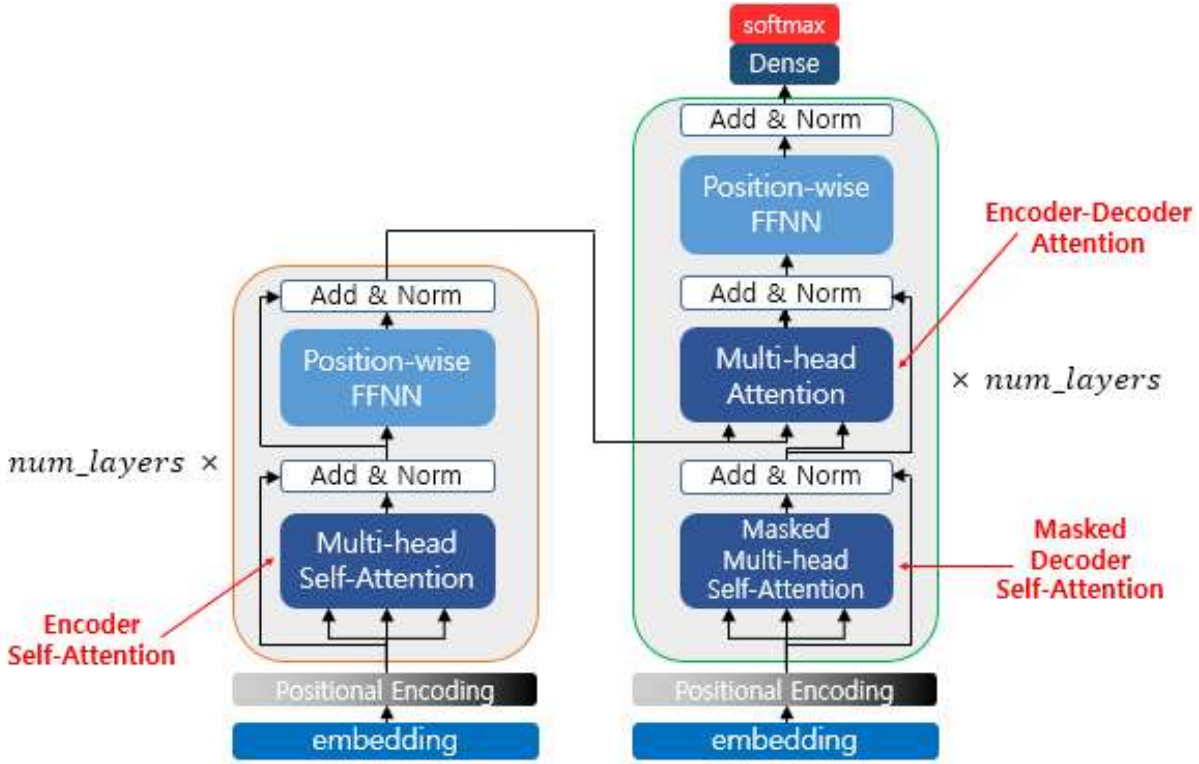
Transformer

- **Multi-Head Attention**
  - **Architecture of Transformer**
    - 정리하면 다음과 같이 하나의 레이어 속 하위 레이어들이 존재.



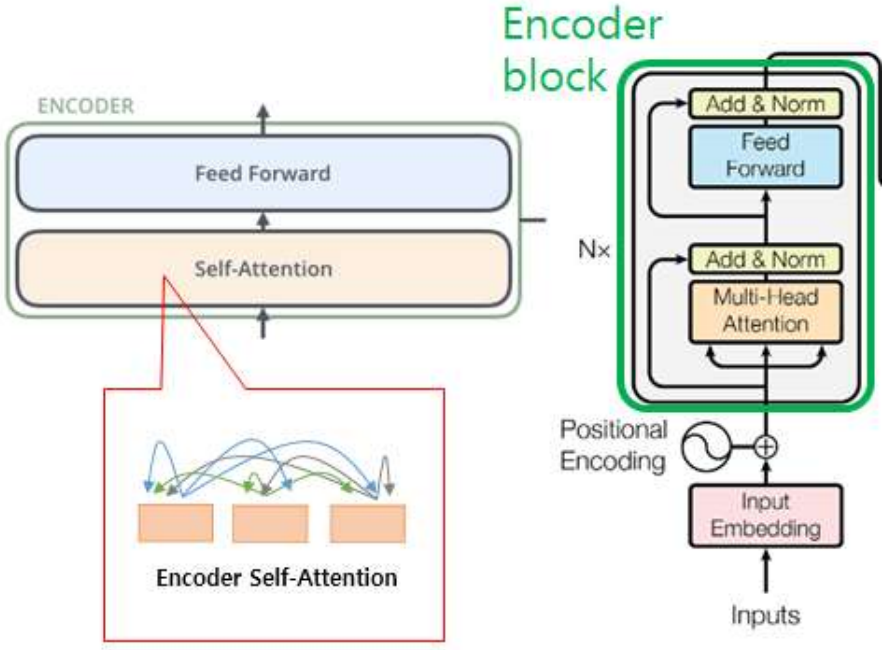
Transformer

- **Multi-Head Attention**
  - **Three ways of Multi-Head Attention**
    - Multi-Head Attention은 다른 위치별로 다른 기능을 수행.
    - Encoder Self Attention, Decoder Self Attention, Encoder-Decoder Attention으로 구성.



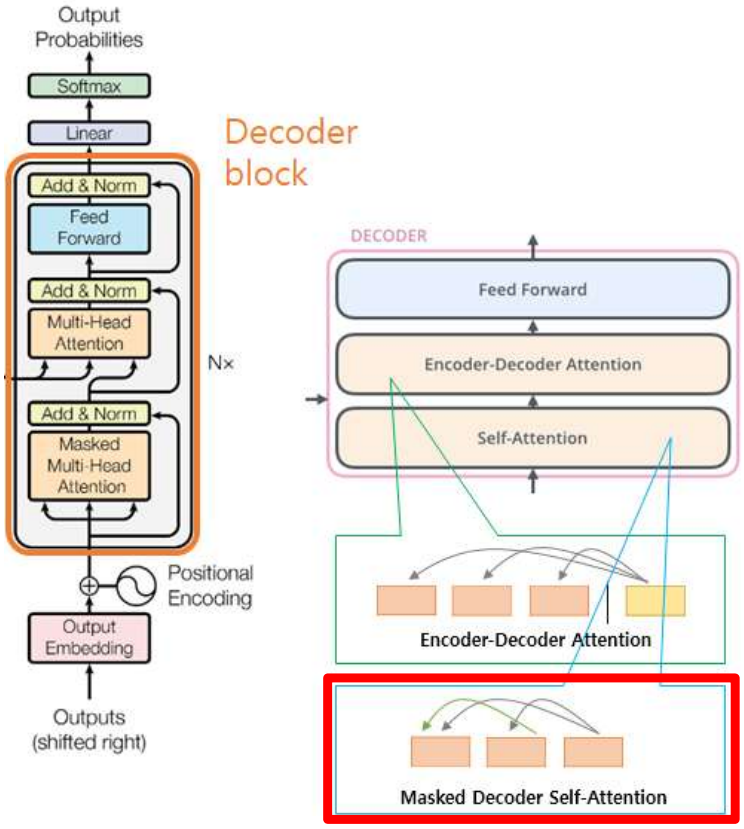
Transformer

- **Multi-Head Attention**
  - **Encoder Self Attention**
    - 인풋 문장들의 문맥을 파악.
    - 처음 인코더 Layer에서 Q, K, V는 임베딩-위치 벡터에서 가져옴.
    - 이후 레이어부터 Q, K, V 모두 이전 인코더 출력에서 가져옴.



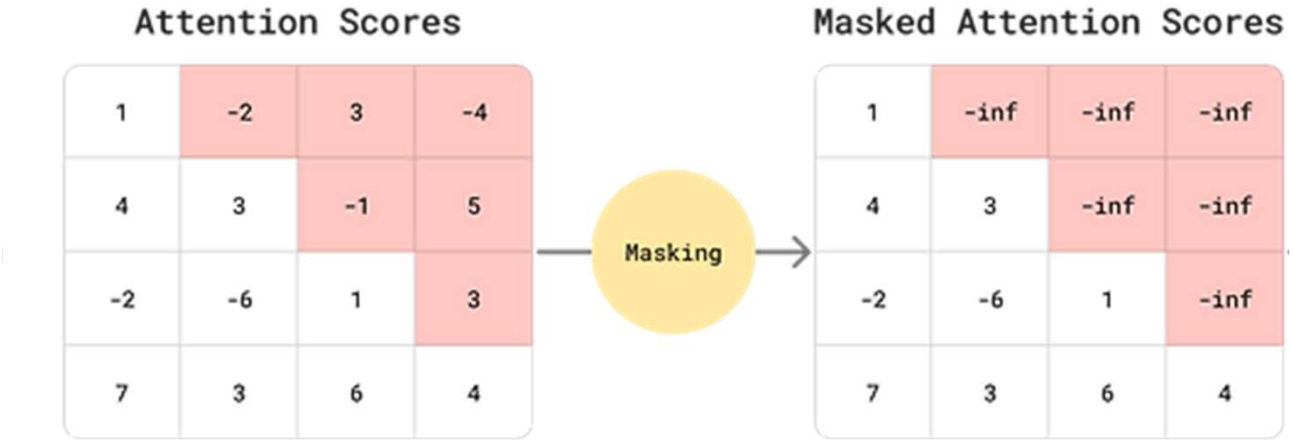
Transformer

- **Multi-Head Attention**
  - **Decoder Self Attention**
    - 아웃풋 문장들의 문맥을 파악.
    - 인코더 셀프 어텐션과 같은 레이어 형태 반복.
    - 추가적으로 마스킹 레이어를 거침.



Transformer

- **Multi-Head Attention**
  - **Masking Layer of Decoder Self Attention**
    - 디코더는 단어를 순차적으로 생성해 나가므로, 미래의 정보를 참조해선 안된다.
    - 따라서 Attention Score에서 미래의 단어에 해당하는 부분은 $-\infty$으로 만듦.



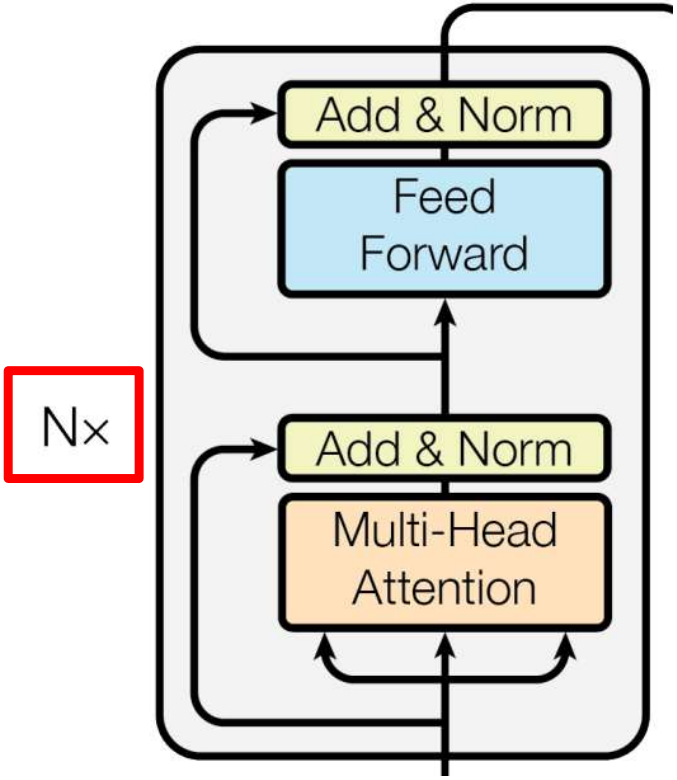
Transformer

- **Multi-Head Attention**
  - **Masking Layer of Decoder Self Attention**
    - $-\infty$ 만듦으로써 이후의 softmax 과정에서 미래 정보를 0으로 만들 수 있음.
    - 이후 V와 가중합함으로써 Context Vector(H)를 만들게 됨.

	<start>	Ice	cream	is	delicious	<end>
<start>	0.99	0	0	0	0	0
Ice	0.004	0.98	0	0	0	0
cream	0.002	0.01	0.95	0	0	0
is	0.001	0.002	0.011	0.99	0	0
delicious	0.002	0.021	0.03	0.01	0.998	0
<end>	0.003	0.001	0.003	0.002	0.012	0.97

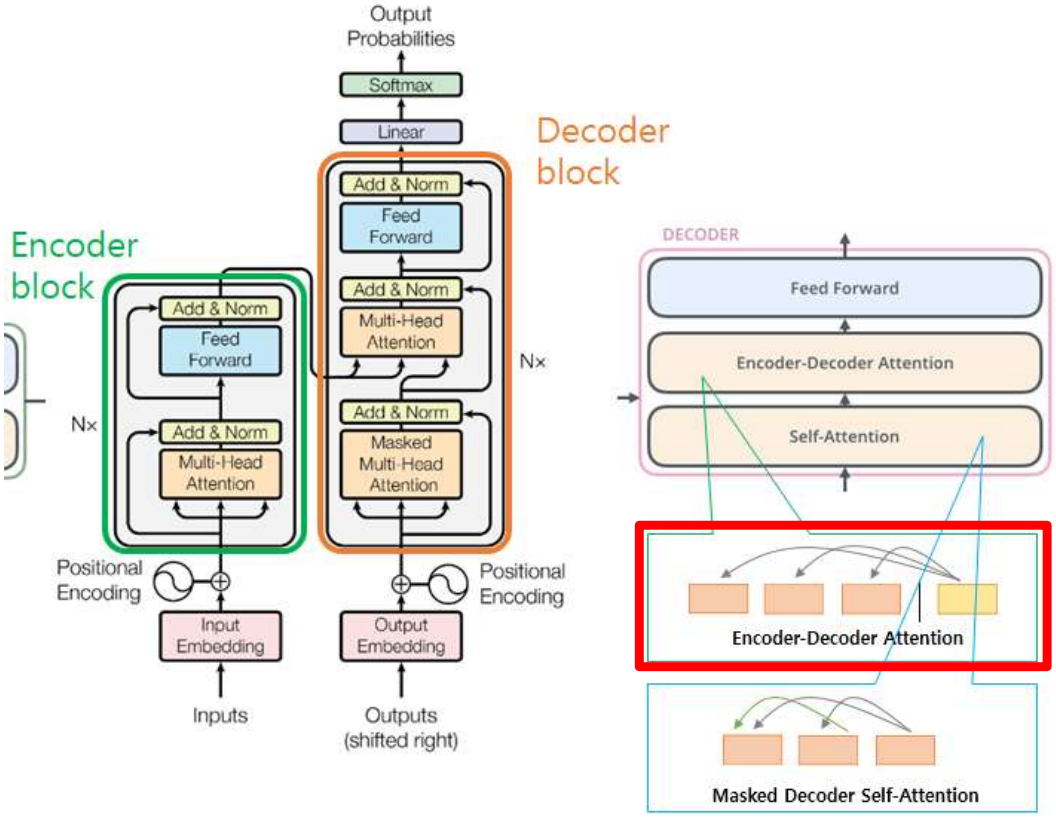
Transformer

- **Multi-Head Attention**
  - **Masking Layer of Decoder Self Attention**
    - 이 H는 다음 디코더 레이어에서 다음 Q, K, V로 변형되고 일련의 과정을 반복.



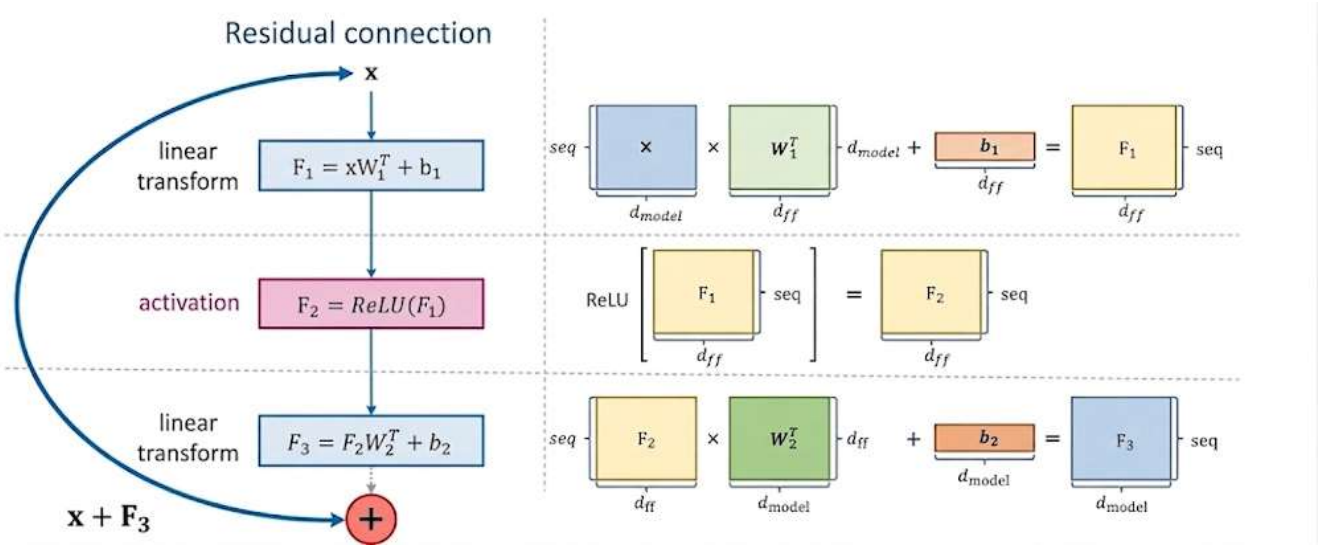
Transformer

- **Multi-Head Attention**
  - **Encoder-Decoder Attention**
    - 디코더에서 나온 문장의 정보와 인코더에서 나온 인풋 문장의 문맥을 융합.
    - 디코더 결과물에서 Q, 인코더 결과물에서 K,V를 만듦.



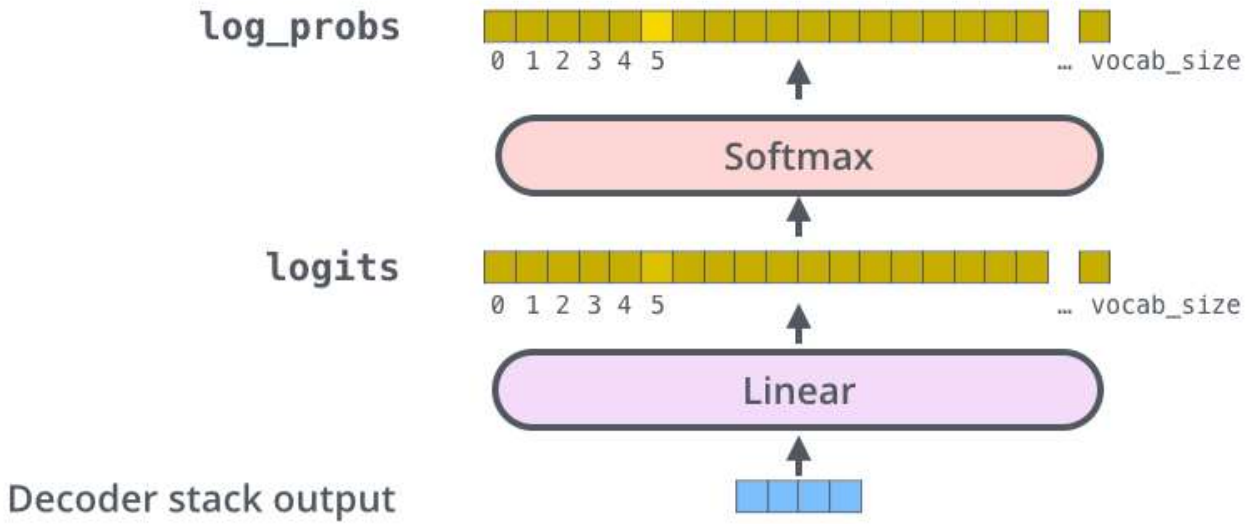
Transformer

- **Feed-Forward & Residual Connection**
  - **Feed-Forward**
    - Multi-Head Attention을 거친 결과들에 비선형성을 추가.
    - 차원의 변형 (512 – 2048 – 512)과, 그 사이 활성화 함수 거침.
  - **Residual Connection**
    - Feed-Forward 결과와 그 이전 값을 Add.
    - 이후, 데이터 분포 정렬하기 위해 Normalization.



Transformer

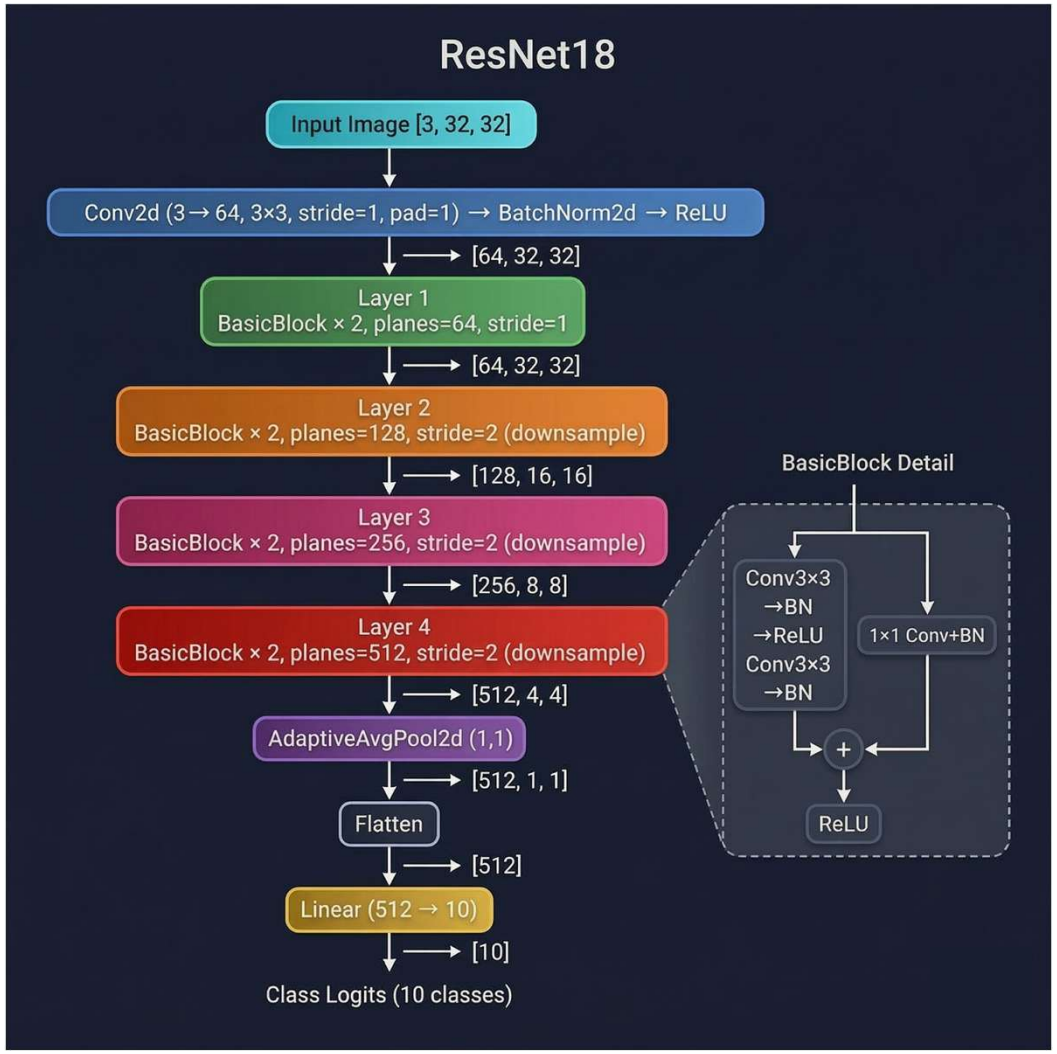
- **Post Processing**
  - **FC**
    - 모든 어텐션이 끝난 후, 512차원의 결과물을 번역하고자 하는 언어의 단어사전 크기만큼 차원을 늘림.
    - 단어가 30000개라면, 30000차원으로 확장 후 FC.
  - **Softmax**
    - 마지막으로 Softmax를 통하여 어떤 단어를 선택할 지 결정.



Experiment
(ResNet VS Multi PipeLined Model)

Experiment

- **Cifar 10 by ResNet**
  - **Aim of Experiment**
    - ResNet의 실습코드를 통해 Cifar 10의 Test Acc를 올리는 것을 목표.
    - ResNet의 구조는 다음과 2개의 Basicblock을 가진 4개의 레이어로 구성.
    - Basicblock은 두번의 3x3 Conv와 그 사이에 ReLU, BatchNorm 구성.



Experiment

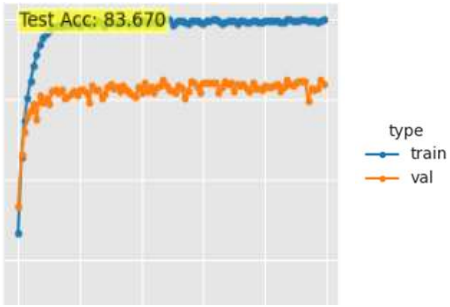
- **Cifar 10 by ResNet**
  - **Experiment Result**

<Val Acc(%)>

l2 \ lr	0.0005	0.00075	0.001
1e-05	83.02	**84.01**	82.45
5e-05	82.47	83.86	81.01
0.0001	80.02	83.12	82.63

Optimizer: Adam
Batch Normalization: O
Epochs : 100
Active Function : ReLU

Test Acc



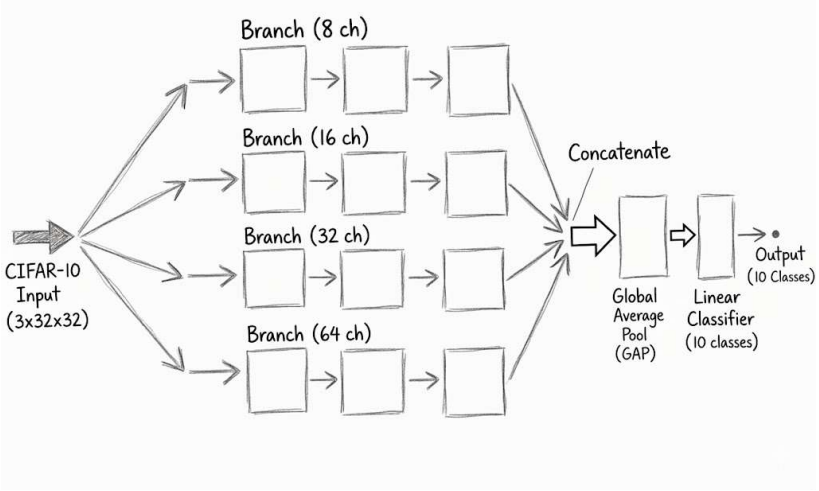
- Test 결과 약 83.7%의 Acc를 확인할 수 있었다.

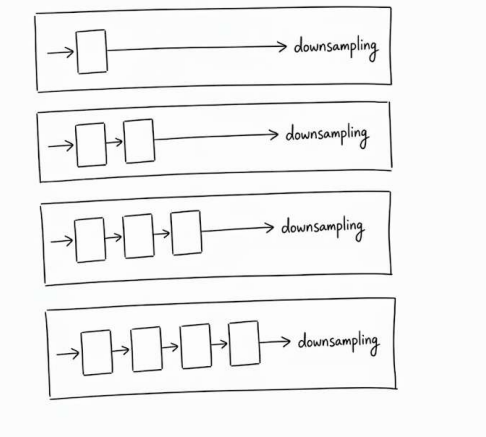
Experiment

- **Cifar 10 by Multi PipeLined Model**
  - **How about these?**
    - ResNet의 성능을 조금 더 올리기 위해서 어떻게 접근하는 게 나을까 생각.
    - 현재 ResNet의 구조는 직렬로 여러 블록 레이어들을 깊게 쌓는 구조.
    - 각각의 Feature들을 전담 분석하는 레이어들을 병렬적으로 넓게 쌓으면?
    - '직렬 구조의 파이프라인을 여러 개 두어, 각각 스스로의 Feature를 전담해서 맡은 다음, 그 값들을 모아서 판단하면 어떨까?' 라는 생각으로 접근.

Experiment

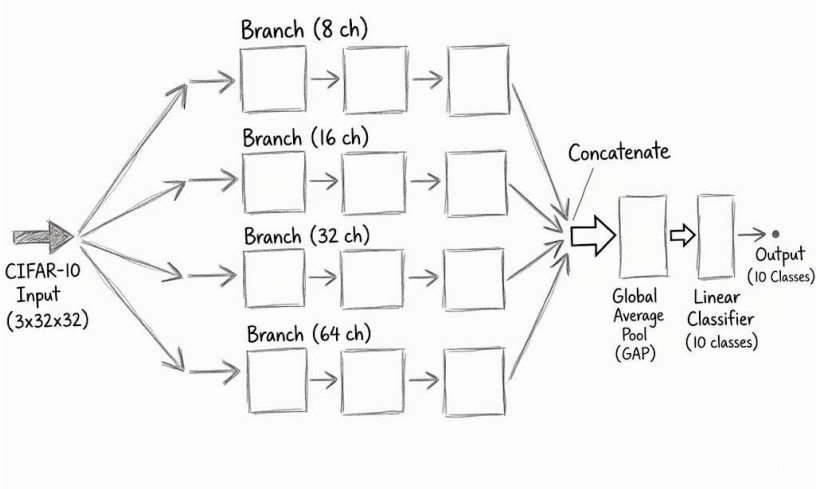
- **Cifar 10 by Multi PipeLined Model**
  - **Architecture**
    - 4개의 다른 채널을 가진 단독적인 파이프라인 구조.
    - 모든 파이프라인은 3개의 스테이지를 가지며, 스테이지 속에 레이어들이 존재.
    - 하나의 스테이지가 끝나면, 다운샘플링을 진행. 단, 마지막 스테이지에선 제외.
    - 채널이 늘어나 복잡할 수록 학습이 어려울 거라 생각하여, 채널이 깊을수록 레이어 개수도 함께 늘어나는 구조를 채용.

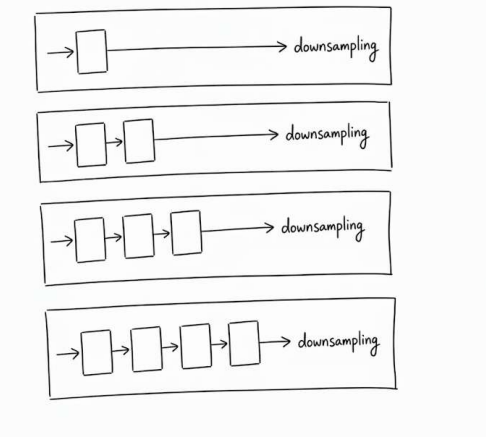




Experiment

- **Cifar 10 by Multi PipeLined Model**
  - **Architecture**
    - 각각의 레이어에선 Residual connection과 ReLU.
    - 각각의 파이프라인에서 나온 결과들을 Concat하여 960채널의 텐서로 합침.
    - Global Average Pooling을 통해 압축하여 1x1 960채널의 텐서로 파라미터 수를 절감.
    - 이후 Flatten하여 벡터화 후 Classifier를 통해 10개의 클래스에 대한 점수 측정.





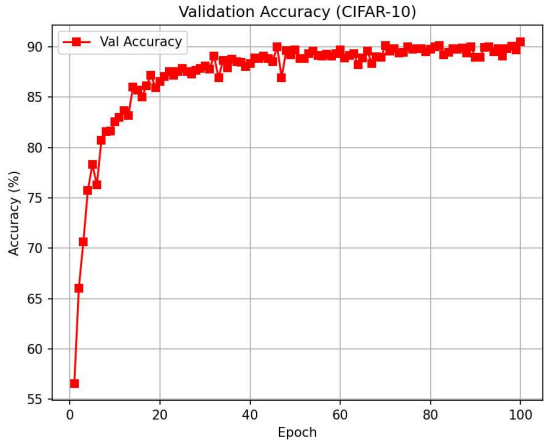
Experiment

- **Cifar 10 by Multi PipeLined Model**
  - **Experiment Result**

<Val Acc(%)>

lr \ batch	64	128
0.0001	89.22	88.50
0.0005	90.76	89.22
0.001	**91.62**	**89.80**
0.005	86.88	89.70

Optimizer: Adam
Batch Normalization: O
Epochs : 100
Active Function : ReLU



- Test 결과 Batch = 64 : Test Acc = 90.72 %, Batch = 128 : Test Acc = 89.35 %로 측정.
- Batch = 64일 때 에포크 당 약 17초, 128일 때 약 9초가 소요되어 속도와 정확도 사이의 Trade-Off가 존재.

Experiment

- **Cifar 10 by ResNet Light**
  - **Why New Model is Better?**
    - 새 모델이 ResNet보다 좋은 이유를 분석하고자 한다.
    - ResNet은 파라미터가 1100만개이며, Multi PipeLined Model은 약 60만개이므로, 파라미터가 많아서 오버피팅이 일어났을 것이란 가능성을 제기.
    - 따라서 ResNet의 파라미터를 줄여서 성능이 하락한다면 파라미터 개수의 문제가 아니므로 모델의 성능이 단순히 좋아서 오른 것으로 추측 가능.
    - ResNet 모델 경량화를 통해 파라미터를 약 70만개로 줄일 수 있었음.
      - 기존 ResNet은 스테이지 별 채널 수가 64 – 128 – 256 – 512로 증가
      - 경량화 모델은 스테이지 별 16 – 32 – 64 – 128 로 증가

Experiment

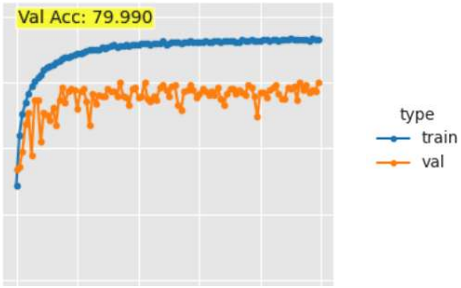
- **Cifar 10 by ResNet Light**
  - **Experiment Result**

<Val Acc(%)>

l2 \ lr	0.001	0.00125	0.0025
0.001	76.13	77.46	77.47
0.005	75.44	**79.99**	76.24

Optimizer: Adam
Batch Normalization: O
Epochs : 100
Active Function : ReLU

Train Acc vs Val Acc



- 실험 결과 Accuracy가 줄어들었으므로, ResNet이 파라미터의 개수가 적어서 오버피팅이 일어난 것은 아니라고 짐작 가능.

Experiment

- **Conclusion**
  - **다른 측면에서의 분석**
    - ResNet은 특성 상 레이어를 깊게 쌓을 수록 성능이 오르는 구조.
    - 그러나 실습의 코드에선 레이어가 16개밖에 없어 최대 성능을 촉진하지 못했을 가능성도 존재한다.
    - 하지만 그런 측면에서, 파이프라인 당 레이어가 최대 12개 밖에 되지 않으므로 병렬 파이프라인 구조가 유의미한 성능을 내었다고 판단할 수 있었다.

Conclusion

- 결론
  - 느낀 점
    - 여태껏 해보지 못했던 딥러닝의 기초를 학습할 수 있어서 좋은 경험이었다.
    - 생각보다 발전이 매우 빠르게 되고 있는 모습에 빨리 따라 잡을 수 있도록 열심히 해야겠다고 느꼈다.
  - 앞으로의 목표
    - 연구주제를 구체화해서 졸업과제 및 차후 연구를 시작하도록 할 예정입니다.