

DEEP LEARNING

– Basics to Trends –

Version 0.1

SEUNGHEE HAN

mod96@naver.com

February 13, 2020

Abstract

- 이 필기록은 2020년 1월 6일부터 21일까지 진행된 고려대학교 수학과 오승상 교수님의 ‘Deep Learning for beginners and experts’ 세미나를 기저로 하여 작성되었습니다. 처음에는 필기용으로 작성하였지만, 그저 그런 필기로만 남겨두면 기억속에 묻힐 것 같아 다듬고 내용을 추가하여 문서화하게 되었습니다. 좋은 기회를 만들어주신 오승상 교수님께 감사드립니다.
- 저는 고려대학교에서 물리학을 본전공으로 수학을 이중전공 중인 4학년 학부생입니다. 식의 유도 과정이나 algorithm 들을 이해하기 위해 해당되는 논문들을 참조하기는 했습니다만, 제대로 이해하지 못하고 오류가 생겼을 가능성이 있습니다. 이런 오류를 발견하여 제 메일 mod96@naver.com 으로 보내주신다면 매우 감사드리겠습니다.
- 우선은 Learning 에 대한 이론들을 기준으로 작성되었으나, 추후에 하나하나 코딩을 하면서 적절한 예시 코드들도 삽입할 예정입니다. 표지의 version 을 기준으로 업데이트 되는 것을 봐주시면 되겠습니다.

Contents

1 Lecture 1 : Deep Neural Network I	2
1.1 Introduction	2
1.2 Machine Learning Motivation	3
1.3 Optimization	7
1.4 Traditional Machine Learning	11
2 Lecture 2 : Deep Neural Network II	14
2.1 Theoretical Background of Neural Network	14
2.2 Nonlinearity	16
2.3 Optimization II	17
2.4 Overfitting	19
2.5 Weight Initialization	24
2.6 Confusion Matrix and Test Score	24
3 Lecture 3 : Convolutional Neural Network	30
3.1 About CNN	30
3.2 Convolution	32
3.3 Other Things of CNN	38
3.4 CNN Architectures	39
4 Lecture 4 : Recurrent Neural Network	46
4.1 Motivation for using RNN	46
4.2 Backpropagation Through Time (BPTT)	50
4.3 Long Short-Term Memory (LSTM)	51
5 Lecture 5 : Variational Auto-Encoder	55
5.1 PCA	55
5.2 Auto Encoder	60
5.3 Variational AutoEncoder	65
5.4 Generative Adversarial Network (GAN)	71
5.5 GANs - DCGAN, cGAN, CycleGAN, PGGAN, WGAN	77
5.6 Semi-supervised GAN	88
6 Lecture 6 : Deep Reinforcement Learning I : RL	90
6.1 Markov Decision Process	91
6.2 Bellman Equation	97
6.3 Dynamic Programming	102
6.4 Introduction to Reinforcement Learning	107
6.5 Monte Carlo Method	109
6.6 Temporal Difference Learnings	112
7 Lecture 7 : Deep Reinforcement Learning II : DRL	118
7.1 Introduction	118
7.2 Deep Q -Network (DQN)	118
7.3 Policy Gradient Method : REINFORCE	123
7.4 Actor-Critic Method	126

7.5 Policy Gradient DRL	133
8 Lecture 8 : Natural Language Processing	143
8.1 Latural Language Processing (NLP)	143
8.2 Word2vec	147
8.3 Seq2Seq	156
9 Lecture 9 : Visual Attention	163
9.1 Image Captioning	163
9.2 Visual Attention	165
9.3 Recurrent Attention Model (RAM)	167
9.4 Deep Recurrent Visual Attention Model (DRAM)	170
9.5 Enriched Deep Recurrent Visual Attention Model (EDRAM)	172
10 Lecture 10 : Graph Representation Learning	178
10.1 Graph Representation Learning (GRL)	178
10.2 Node Embedding (shallow)	180
10.3 Graph Neural Network (GNN)	185

1 Lecture 1 : Deep Neural Network I

1.1 Introduction

1.1.1 Introduction to this seminar

이 세미나의 청중은 대학생 뿐만 아니라 연구원, 직장인 등으로 다양하다. 초반의 강의는 매우 쉽다. 아마 관심이 있었다면 다 아는 내용일 것이다. 하지만 그 후로는 매우 어려운 수학적 개념들이 등장하는데 이들은 수학적 지식, 특히 확률과정에 대한 지식이 없다면 이해하기가 어려울 것이다. 이 강의의 목적은 후에 관련 논문을 읽었을 때 혼자 학습할 수 있을 정도의 실력을 갖게 하는 것이다.

몇년 전 AlphaGo 와 이세돌의 대결로 세간이 떠들썩했다. 이전에도 AI 와 바둑기사의 대결은 있었지만, 인간이 AI 를 한 번 이겼다는 것으로 많은 이목을 집중시킬 수 있었다. 그런데 AlphaGo 는 어떻게 작동하는 것일까? 바둑이라는 게임에서 매 수마다 평균 선택할 수 있는 경우의 수는 250 가지이며, 한 판의 평균 착수는 150 이다. AlphaGo 는 가능한 모든 경우의 수를 고려하여 바둑을 둔 것 일까?

Learning, 혹은 Statistical Learning 은 확률론을 기반으로 하며 1960 년대부터 1980 년대 정도에 정립된 오래된 이론이다. 1990 년대에 들어서 컴퓨터 코딩을 이용하기 시작했고, 이것이 Machine Learning, 즉 Statistical Learning 을 기계가 알아들을 수 있게 만드는 것의 발판이 된다. 사실 machine learning 은 그 기반이 대부분 확률이 아니고, 그저 숫자를 normalize 하여 확률이론을 가져다 사용한 것 뿐이다. 그런데 그냥 해보니 잘 된 것이고, 잘 사용하는 것이다.

이후 2012 년 즈음이나 되어 게임 등에 적용되며 Machine Learning 이라는 것이 좋은 결과를 낸다는 것을 보여준다. 2015 년 이후에는 더 어려운 통계적 이론을 사용하며 더 어려운 문제들에 도전하고 있는데, 특히 NLP (Natural Language Processing) 의 연구가 활발하다.

Machien Learning에서 기본적인 것들은 우리도 조금만 배우면 충분히 할 수 있다. Python 을 예로 보자면, Google 에서 Tensorflow 라는 소스를 개발하면서 심지어는 고등학생도 조금만 따라하면 어느 정도 다를 수 있게 되었다. 또한 Keras, PyTorch 등이 있으며, Mathematica 는 Wolfram 에서 자체적으로 개발한 내장모듈로 쉽게 Machine Learning 을 사용할 수 있다.

최대한 쉽게 개념을 적는다고 적었지만, 학부수준의 다변수 미적분, 선형대수, 기초 확률론 등은 있다고 가정하고 필기록을 작성했다. 잘 정리된 교재를 공부하고 싶다면, 2019 년 2 학기에 수강한 강의 중 고려대학교 수학과 김영욱 교수님께서 추천해주신 다음과 같은 교재들을 참고하도록 하자.

1. 수학적 이론에 관해서는 Stanford University open course 에서 제공하는 CS229 - Ng 강의와 교재 introduction to statistical learning - James, Witten, Hastie, Tibshraum 를 추천한다. 학부 수준에서 가장 적당한 수준의 교재이다.
2. 대한원 수준의 심화된 교재로는 The elements of statistical learning - Trevor Hastie, Robert Tibshirani, Jerome Friedman 이 있다.
3. 쉬운 교재로는 introduction to machine learning, ethem alpaydin 이 있다.
4. 코딩을 위한 기본 교재로는 deep learning with python, jason brownlee 를 추천한다.

AI는 빠르게 발전하고 있기 때문에 이 분야에서 뭔가를 하고 싶다면 관심을 갖고 지속적으로 찾아봐야 한다. ‘Medium’이라는 사이트에서 data science 주제를 추천한다. 월 6천원을 내면 더 좋은 정보를 볼 수 있다. 의료와 그림에 관련하여 CNN-Convolution, 주식과 언어에 관련하여 RNN-recurrent, 게임에 관련하여 RL-reinforcement learning이라는 키워드가 있다. 자, 이제 본격적으로 AI 의 이론들을 알아보자.

1.1.2 Types of Learning

알파고는 CNN (Convolutional Neural Network) 과 RL (Reinforcement Learning) 을 사용한 대표적인 Machine 이다. 이 때 CNN 이란 그림을 컴퓨터가 인식할 수 있게 하는 방법인데, 작은 영역의 극점을 사용하여 그림의 특성을 파악하는 방식이다. 즉, 바둑판의 그림을 컴퓨터에 인식시켜 학습했다는 뜻이다. RL 은 강화학습이라고 부르는데, 이전의 선택이 후의 선택으로 영향을 주는 방식이다. 최적의 선택과정을 택하게 하는 방법으로서 RL 을 사용한다.

최초의 알파고는 기존에 있던 자료를 바탕으로 (Supervised Learning) 학습했다. AlphaGo Zero 는 알파고의 최종본으로, 바둑의 룰만 알려준 뒤 본인의 복사본과 게임을 하면서 스스로 학습하여 (Reinforcement Learning) 좋은 결과를 얻게 되었다.

Machine Learning 은 크게 3 가지로 나뉘며, Algorithm 들은 각각의 목적에 맞게 변형되어 사용된다.

1. **Supervised Learning** : 명시적인 정답이 주어진 상태에서 학습하는 방법이다. 즉 어떤 그림이 주어졌을 때 해당 그림이 ‘자동차’인지, ‘강아지’인지 등에 대한 정답을 기계에게 알려주고 학습을 시행하는 방법이다. 혹은 어떤 손글씨 그림이 주어졌을 때 해당 그림이 09 중 무엇인지 알려준 뒤 학습하는 방식이다.
2. **Unsupervised Learning** : 정답이 주어지지 않는다. 이 방법은 정해진 무엇인가를 맞추기 보다는 데이터의 숨겨진 특징이나 구조를 발견하는 데에 사용된다. 즉, 어떤 데이터에서 어떤 성질을 가지는지 모를 때 새로운 특징을 발견하는 데에 좋다.

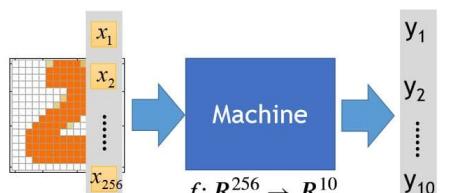
Unsupervised Learning 은 차원이 높아지면 데이터가 기하급수적으로 많이 필요하다. 이를테면 Regression 을 하는 경우 데이터 수가 5개 인데, 이를 선형으로 fitting 할지 이차곡선으로 fitting 할지 결정해야 하는 것은 불가하다. 데이터 수가 차원보다 매우 많은 경우 규칙을 찾을 가능성이 생긴다. 이 과정을 grouping 혹은 “Clustering”이라 한다. 패턴 공간에 주어진 유한 개의 패턴들이 서로 가깝게 모여서 무리를 이루고 있는 패턴 집합을 cluster (군집) 이라하고 무리지워 나가는 처리 과정을 clustering 이라 한다.

3. **Reinforcement Learning** : 규칙을 알려주고 Machine (Agent) 이 특정 결정을 내리면 그 행동에 대해 보상을 가하면서 학습하는 방식이다. 이 때 Machine 은 보상을 최대화 하도록 학습을 진행한다.

1.2 Machine Learning Motivation

1.2.1 Motivation

우리의 목적은 어떤 데이터를 기계에 집어넣었을 때 유의미한 결과를 얻는 것이다. 대표적으로 손글씨를 기계가 인식하도록 하기 위해 사용되는 MNIST 데이터의 경우 수많은 손글씨 사진과 그에 해당되는 정답이 쌍을 이루고 있다. 즉, 우리는 픽셀로 들어가는 데이터를 기계에 넣고, 결과적으로 그 사진이 어떤 숫자를 의미하는지 알아내야 한다.

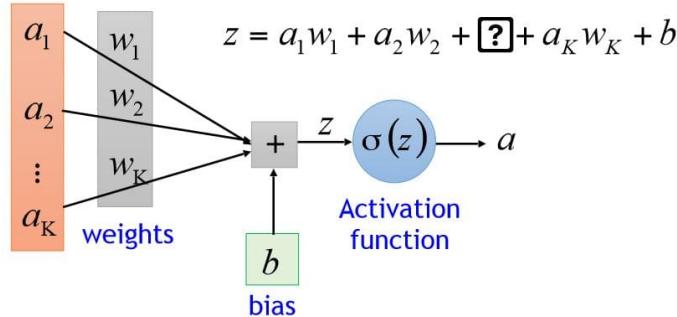


In deep learning, the function f is represented by neural network

이 경우 output y_1, y_2, \dots, y_{10} 은 각각 input data 가 나타내는 숫자가 1, 2, ..., 9, 0 일 확률이다. 가장 높은 값에 해당하는 숫자를 출력하면 기계는 사진을 보고 숫자를 인식한 것이 된다.

Perceptron 그렇다면 이 Machine 은 어떻게 구성되는가? 기본적으로 Machine 은 사람의 Neuron 을 모방한다. 그래서 Neural Network(NN) 라 불리는 것 이고, 복잡한 사람의 신경망을 모방한다. 다음은 가장 간단한, output dimension 이 1 인 1 개의 neuron 모델이다.

$$\text{Neuron} \quad f: R^K \rightarrow R$$



*playground.tensorflow.org 에서 deep learning 이란 어떤 것인지 간단하게 체험해볼 수 있다.

$$\text{output} = \sigma \left(\sum w_i a_i + b \right) = \sigma(WA + b)$$

우선 각각의 데이터마다 중요도가 다를 것으로 생각할 수 있다. 따라서 input data 각각에 다른 **weight** 를 곱해준다. 그렇게 선형결합된 값에 약간의 **bias** 를 준다. 데이터에 관계없이 우리가 모르는 영향이 있을 수도 있으므로 추가해주는 것이다. 그 후 비선형성을 추가하기 위해 Activation function 에 넣어준다. 그렇게 결과값을 출력하게 된다. 이 때 weight 와 bias 추후 모두 학습되는 **parameter** 이므로 최초에 어떤 값을 잡더라도 충분한 학습과정을 거치면 수정되어 적절한 값으로 수렴할 것이다.

신경망의 구조	Input Layer	학습할 데이터를 읽어들임
	Hidden Layer	weights+bias+activation functions로 학습
	Output Layer	최종결과 계산

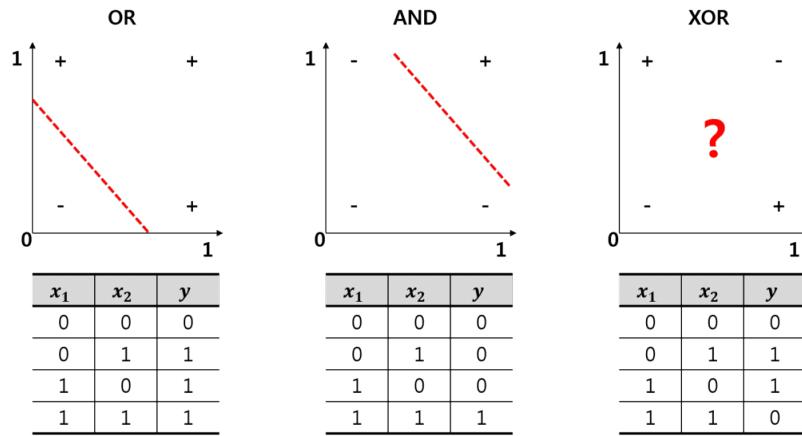
activation function 은 0 부터 1 의 값을 갖는 함수이어야 한다. 또한 다루기 쉬운 함수, 특히 미분했을 때 다루기 쉬운 함수이어야 한다. 후에 gradient descent 과정에서 자세하게 설명하도록 한다.

1.2.2 XOR Problem

XOR input x_1, x_2 둘 중 하나만 1 일 때 1 을 출력하는 논리회로이다.

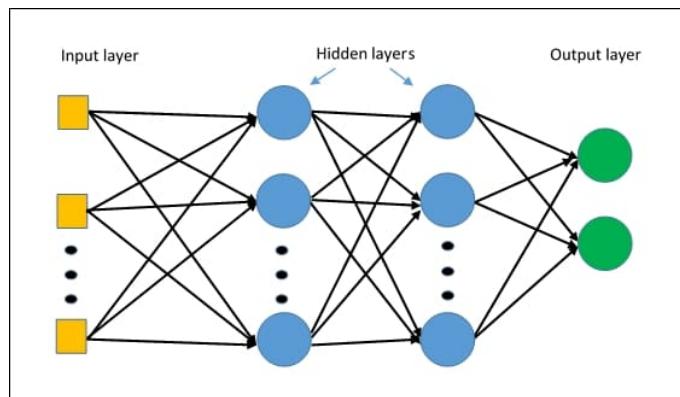
Inputs		Outputs
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

딥러닝의 초기에는 이 XOR이 가장 큰 문제 중 하나였다. linear regression 의 경우를 예로 들어보자.

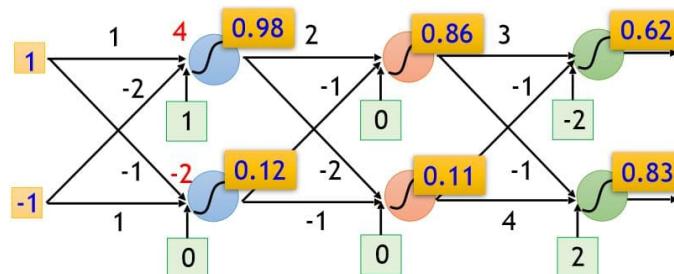


즉, AND 나 OR 은 linear regression 을 통해 쉽게 풀 수 있었지만, XOR 의 경우는 linear model로 풀 수가 없다. 1969 년 Minksy 와 Papert 는 하나의 perceptron 으로 비선형성을 추가했음에도 이 문제를 해결할 수 없다는 것을 증명했다. 이 때문에 딥러닝에는 하나의 빙하기가 찾아온다.

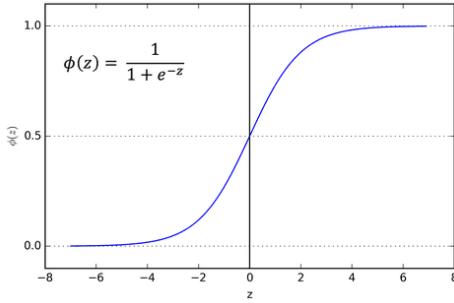
Multilayer Perceptron(MLP) 하나의 perceptron 으로는 한계가 있다는 것이 자명했다. 하지만 Geoffrey Hinton 교수는 layer 를 쌓아 MLP 를 만든다면 다양한 사용이 가능하며 XOR 문제도 해결할 수 있다는 것을 밝혀낸다.



이런 방식으로 layer를 쌓으면 계산량이 많아지지만 다양한 비선형함수를 만들 수 있게 된다. 층을 계속해서 쌓으면 이를 **Deep Neural Network(DNN)** 이라 한다. 원래는 컴퓨터의 연산량이 많아져 비효율적인 방식이라 취급되었지만, 점차 시대가 지나면서 이를 이용하는 것이 좋은 결과를 낼 수 있다는 것이 밝혀진다. 그렇다면 그 계산은 어떻게 되는가? DNN 에서, 1 개의 input layer, 1 개의 hidden layer, 1 개의 output layer, 그리고 모든 layer 의 차원이 2 이며 activation function 으로 sigmoid function 을 사용하는 경우의 예시가 다음과 같다.



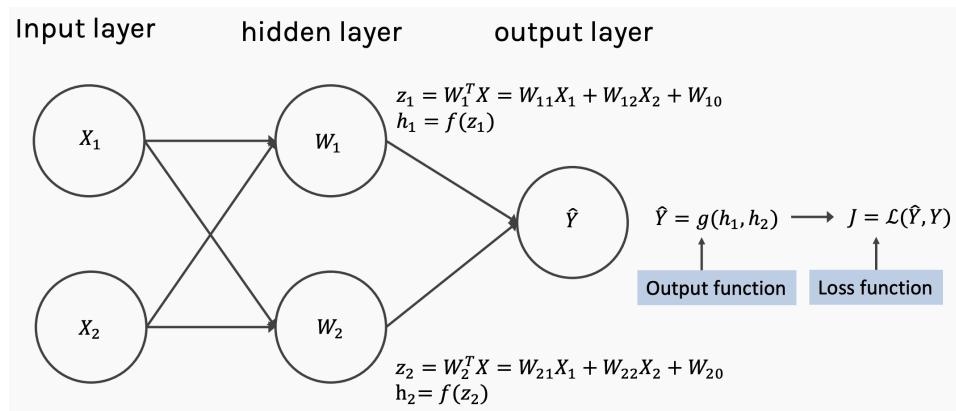
참고로, sigmoid function은 다음과 같다.



위에서는 weight, bias 값을 상정한 상태에서 (1,-1)의 input에 대한 output을 계산했다. 하지만 우리가 하고 싶은 것은 output 값을 알 때 적절한 weight, bias 값을 찾는 것이다. 이를테면 MNIST의 경우 '2'라는 답을 알고 있는 경우, 즉, output이 사실은 (0, 1, 0, 0, ...)으로 나와야 한다는 것을 알 때 weight와 bias를 적절히 조정하여 y_2 가 가장 크게 나오게 하는 것이다. 즉, Deep Learning 이런 weight, bias 값을 찾는 과정이다. 이 weight, bias를 **Learnable (trainable) Parameter θ** 라고 한다.

$$\theta = \{W^1, b^1, W^2, B^2, \dots, W^L, b^L\}$$

위의 과정을 일반화하면 다음과 같다.



이 때, 각 input 층에 1을 추가하고 Weight matrix 안에 bias를 넣어주면 Weight matrix 하나만으로 학습이 가능한 꼴이 된다. 이를테면 위에서 다른 예시의 첫 번째 weight 계산이 다음과 같았다면,

$$WX + b = \begin{pmatrix} 1 & -2 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

bias를 포함한 꼴은 다음과 같다.

$$W'X' = \begin{pmatrix} 1 & -2 & 1 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

Softmax Layer DNN에 사용되는 Layer는 매우 다양하다. 이 Layer의 경우 가장 기초적인 Layer 중 하나로, 해당 Layer에서 출력되는 값들의 총합으로 각각의 출력값을 나눠주면 총합이 1이기 때문에 마치 이것이 확률인 것처럼 계산이 적용된다. 물론 확률은 아니지만 확률론에서 발전된 이론들이 모두 적용된다.

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} \rightarrow \begin{pmatrix} e^{z_1} / \sum e^z \\ e^{z_2} / \sum e^z \\ e^{z_3} / \sum e^z \end{pmatrix}$$

이 Layer는 모든 hidden layer가 끝난 뒤 마지막에 넣어준다. 그렇게 되면 최종 값을 확률로서 다룰 수 있게 된다. Classification 문제의 경우 대부분 outputlayer 로서 softmax layer 를 사용하게 된다.

1.3 Optimization

Cost (loss) 특정 parameter θ 가 주어졌을 때, 어떤 input data 에 대한 결과값 output 이 실제 정답과 얼마나 가까운지를 표현하는 function 이다. Cost 는 그 목적에 따라 다양한 형태를 가지는데, 주로 **Mean Squared Error(MSE)** 혹은 **Cross Entropy** 를 사용한다. Total cost 는 각 input data 에 대한 cost (loss) 의 합이다.

$$C(\theta) = \sum L(\theta)$$

우리는 Machine 을 학습시키기 위해 적절한 Loss function 을 잡고 이를 최소화하거나, 최대화 한다. 이 때 데이터를 이용하는데, 데이터의 사이즈는 보통 너무 커서 지금의 컴퓨터로도 한번에 다루기가 힘들다. 또한 데이터를 한 번 사용했다고 해서 최적화된 값을 찾는 것은 어렵다. 때문에 우리는 최적화 과정에서 여러 번의 학습 과정을 거치는데, 이 때 등장하는 단어가 epoch, iteration, batch 이다.

- **epoch** : NN 은 output 값을 계산하는 forward pass 와, 최적화를 위해 반대로 거슬러 올라가며 parameter 를 수정하는 backward pass 로 나뉜다. 주어진 전체 데이터셋에 대해 forward pass 와 backward pass 가 끝나면 한번의 epoch 가 완료되었다고 한다. 이를테면 epoch=10 이라면 주어진 데이터셋에 대해 10 번의 학습을 진행하는 것 이다.
- **iteration** : 사용하는 데이터는 보통 매우 크기 때문에 한번에 다룰 수 없다. 때문에 데이터를 쪼개서 학습하게 되는데 이 때 몇 번 나누는가를 iteration 이라 한다.
- **batch size** : iteration 에 따라 나누어진 데이터 한 셋을 batch 라고 하며 이 것의 크기를 batch size 라 한다.

이를테면 5000 개의 데이터가 있다고 하자. epoch=20, batch_size=50 이라고 한다면 iteration 은 100 회이며 iteration 기준으로 봤을 때 2000 회의 학습이 진행된다.

1.3.1 Gradient Descent

Gradient Descent 는 특정 함수가 주어졌을 때 그 최소점/최대점을 찾는, 1차 근사 최적화 알고리즘이다. 최적화할 함수 $f(\mathbf{x})$ 가 주어졌을 때, 먼저 적절한 시작점 \mathbf{x}_0 를 정하고 다음과 같은 연산을 지속적으로 거친다.

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma_i \nabla f(\mathbf{x})$$

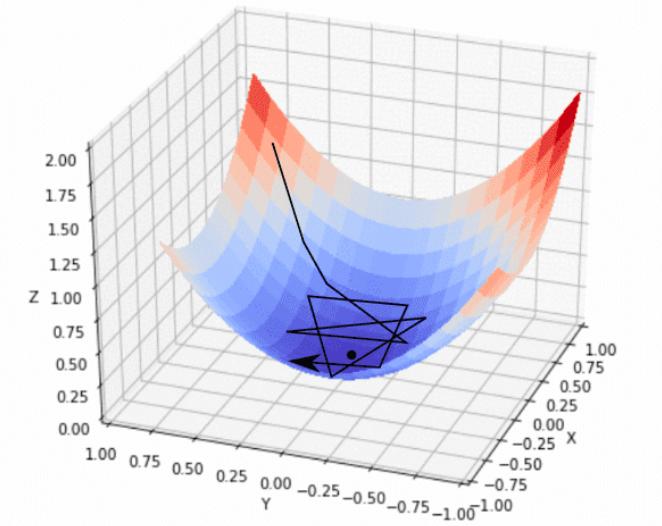
이 때 γ_i 는 한 번에 이동할 거리를 조절하는 매개변수이며, i 에 따라 달라질 수 있다. 이를테면 3차원 parabola $f(x, y) = x^2 + y^2$ 의 경우 그 gradient 는 자명하게도 다음과 같다.

$$\nabla f(x, y) = (2x, 2y)$$

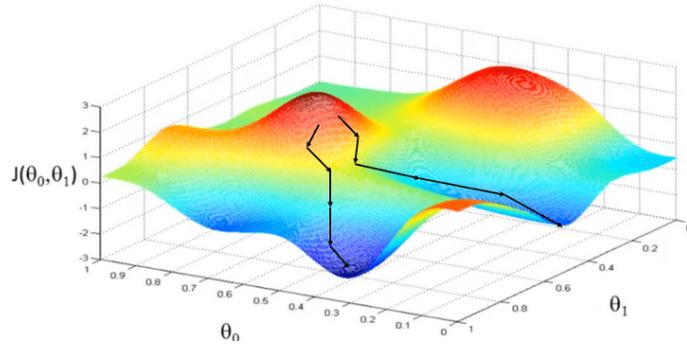
그리고 gradient descent 방식을 적용하면 다음과 같을 것이다.

$$\begin{aligned} x_{i+1} &= x_i - 2x_i \gamma_i \\ y_{i+1} &= y_i - 2y_i \gamma_i \end{aligned}$$

적절한 초기값에 따라 진행하는 경로를 보면 다음과 같다.



하지만 이 과정은 local minimum 밖에는 찾지 못한다. 즉, 우리가 원하는 것은 global minimum이나 이를 정확히 찾지는 못한다.



위의 예시 그림의 경우 어디서 parameter 초기값을 시작하느냐에 따라 서로 다른 local minimum 으로 수렴하는 것을 볼 수 있다.

1.3.2 Backpropagation

Geoffrey Hinton 교수는 Gradient descent 방법을 정형화하여 ‘backpropagation’이라는 이름으로 ML에 적용했다. 이전에 언급했듯이 output 값을 구하기 위해서는 forward process를 거쳐며, gradient로 parameter를 수정하기 위해서는 backpropagation을 거쳐야 한다. 수학적으로 어떤 함수의 gradient란 그 점에서 경사가 가장 심한 방향과 그 크기를 가진 벡터이다. 우리의 목적은 Loss function의 최소점이기 때문에 이 방향으로 하강한다. 단순히 빼는 것은 아니고, 학습률 η 를 곱해서 하강한다. 즉, 다음과 같은 과정을 갖는다.

$$\Theta_{n+1} = \Theta_n - \eta \frac{\partial}{\partial \theta} \text{Loss}(\theta) \Big|_{\Theta_n}$$

위에서 언급했던 가장 간단한 perceptron의 경우를 생각해 보자. input data \mathbf{x} , output \hat{y} , output data y 에 대해 다음과 같은 구조를 보여줬다.

$$\mathbf{x} \rightarrow z = \sum_i w_i x_i + 1 \cdot b \rightarrow \hat{y} = \frac{1}{1 + e^{-z}},$$

or, equivalently,

$$\hat{\mathbf{y}} = \sigma \left(\sum_i w_i x_i + 1 \cdot b \right) = \sigma(W\mathbf{x})$$

Loss function 으로 squared loss 를 사용할 경우 다음과 같다.

$$L = \frac{1}{2} \sum_i (\hat{y}_i - y_i)^2$$

이제 loss function 을 weight w_i 각각으로 편미분해야 한다.

$$\begin{aligned} \frac{\partial L}{\partial w_i} &= \frac{\partial}{\partial w_i} \left[\frac{1}{2} \sum_n \left(\sigma \left(\sum_j w_j x_j + b \right) - y_n \right)^2 \right] \\ &= \sum_n \frac{\partial z}{\partial w_i} \frac{\partial \hat{y}_n}{\partial z} \frac{\partial L}{\partial \hat{y}_n} \\ &= \sum_n x_i \cdot \frac{e^{-z}}{(1 + e^{-z})^2} (\hat{y}_n - y_n) \\ &= - \sum_n x_i \cdot \hat{y}_n (1 - \hat{y}_n) \cdot (y_n - \hat{y}_n) \end{aligned}$$

위의 과정을 Multilayer 경우의 연산으로 일반화 해보자. 2개의 Layer의 경우에 다음과 같은 구조를 가정해보자.

$$\begin{array}{ccc} \nearrow : & & \nearrow : \\ \mathbf{x} \rightarrow z_j = \sum_i w_{ji} x_i \rightarrow \hat{y}_j = \sigma(z_j) \rightarrow z'_k = \sum_j w'_{kj} \hat{y}_j \rightarrow \hat{y}'_k = \sigma(z'_k) \\ \searrow : & & \searrow : \end{array}$$

우선 뒤의 layer 한 층에 대한 partial derivative 를 계산해보자.

$$\frac{\partial L}{\partial \hat{y}_j} = \sum_k \frac{\partial z'_k}{\partial \hat{y}_j} \frac{dL}{dz'_k} = \sum_k w'_{kj} \frac{dL}{dz'_k} = \sum_k w'_{kj} \frac{d\hat{y}'_k}{dz'_k} \frac{\partial L}{\partial \hat{y}'_k}$$

이를 이용하면 다음과 같이 w_{ji} 에 대한 gradient 계산이 가능하다.

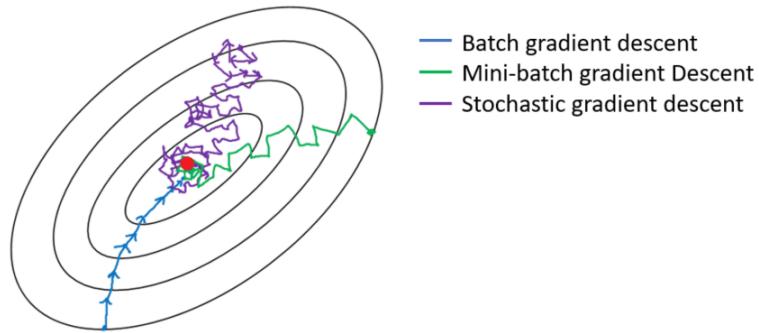
$$\frac{\partial L}{\partial w_{ji}} = \sum_n \frac{\partial z_j^n}{\partial w_{ji}} \frac{d\hat{y}_j^n}{dz_j^n} \frac{\partial L}{\partial \hat{y}_j^n} = \sum_n \frac{\partial z_j^n}{\partial w_{ji}} \frac{d\hat{y}_j^n}{dz_j^n} \sum_k w'_{kj} \frac{d\hat{y}'_k}{dz'_k} \frac{\partial L}{\partial \hat{y}'_k}$$

Gradient Descent 는 그 사용 방법에 따라 다양하게 나뉜다. 지금은 정말 많은 방법들이 나와있지만, 우선 Batch 에 따라 나누면 다음과 같다.

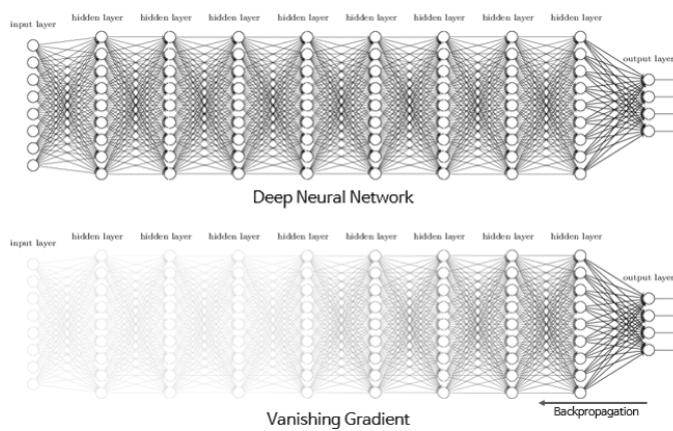
1. Batch Gradient Descent : 전체 training data 에 대해 한번에 Loss 를 계산한다. 즉, Total Cost 를 사용한다.

- 장점 : Loss function convex 라면 global minimum 을 보장한다.

- 단점 : 각 과정에서 연산이 많으므로 느리다.
2. Stochastic Gradient Descent (SGD) : 데이터셋에서 랜덤하게 하나의 데이터를 뽑아 Loss를 계산한다.
- 장점 : 빠르다.
 - 단점 : variance 가 크고 noise에 민감하다.
3. Mini-batch Gradient Descent : 특정 사이즈의 데이터만 뽑아 Loss를 계산한다.
- 장점 : 빠르다. 적당한 variance를 가진다.
 - 단점 : batch size는 hyperparameter인데, hyperparameter는 많을수록 안좋다. (보통 32에서 256 정도를 쓴다.)



Vanishing Gradient Problem 딥러닝에는 두번째 빙하기가 찾아온다. MLP의 경우 backpropagation 과정에서 sigmoid를 미분한 것이 계속해서 곱해지는데, sigmoid 함수의 특성상 미분값은 항상 0에 가깝고, 곱해질수록 값이 사라지는 문제가 생긴다. 이렇게 되면 학습이 더 이상 진행되지 않는 문제가 생긴다. output layer와 가까운 layer는 그나마 괜찮지만, front layer로 가까워질수록 gradient가 곱해지면서 0에 가깝게 되어 최종 값이 0이 되어버리는 것이다.

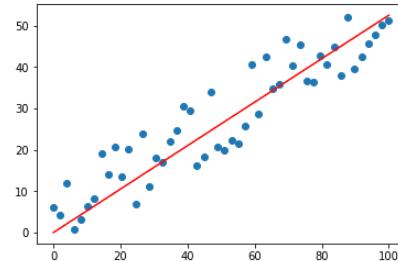


Geoffrey Hinton 교수는 ReLU 함수를 만들면서 이 문제를 해결하게 된다. 이는 후에 다루도록 한다.

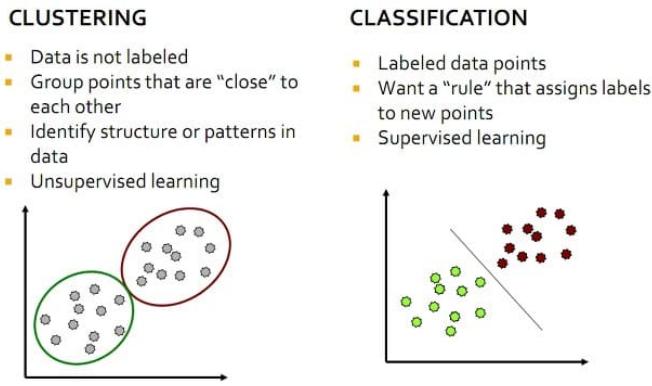
1.4 Traditional Machine Learning

Types of Learning Machine Learning 을 그 목적에 따라 나누면 다음과 같다.

1. Regression : 연속 변수 y 와 다른 변수들 x_1, x_2, \dots 들 간의 관계 혹은 함수를 찾는 문제이다.



2. Classification : 새로운 데이터가 들어오면 해당 데이터가 어떤 class에 해당하는지 출력하는 기계를 찾는 문제이다.
3. Clustering : 데이터를 같은 특징을 갖는 집단으로 나누는 문제이다.



즉, classification 은 training 과정에서 주어진 dataset 이 어떤 ‘class’ 인지 정답이 주어져있고 그 규칙을 찾아내는 반면, clustering 은 알 수 없는 규칙을 찾아내는 과정인 것 이다.

Linear Regression Machine Learning (그 의미를 확장했을 때) 의 가장 기본적인 방법이다. Neural Network 는 사용하지 않으며, input 과 output 이 linear 관계를 갖는다고 가정하고 학습하는 과정이다. 즉, 다음과 같은 구조를 갖는다.

Training Data : $\{(x_1, y_1), \dots, (x_m, y_m)\}$

Hypothesis : $y \approx h(x) = wx + b$

Cost : $L = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - y_i)^2$

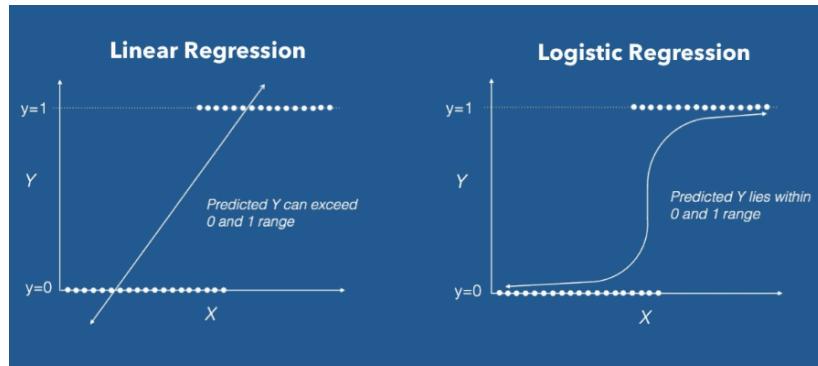
$x \rightarrow y = w_1x + w_2$

이 때 $w = w_1, b = w_2$ 라고 두고 gradient descent 방법을 사용하면 된다. 물론 적절한 초기값이 요구되며, η 와 η' 은 같을 수도 있고 다를 수도 있다. 보통은 같은 것으로 둔다.

$$\begin{aligned}w_1^{t+1} &= w_1^t - \eta \frac{\partial L}{\partial w_1} = w_1^t - \eta \cdot \frac{1}{m} \sum_{i=1}^m (w_1^t x_m + w_2^t - y_i) x_i \\w_2^{t+1} &= w_2^t - \eta' \frac{\partial L}{\partial w_2} = w_2^t - \eta' \cdot \frac{1}{m} \sum_{i=1}^m (w_1^t x_m + w_2^t - y_i)\end{aligned}$$

Logistic Regression (Binary Classification) $Y = \{0, 1\}$ 의 집합 안에서 출력값이 나오도록 하는 방법이다. 이는 주로 결과가 ‘참’ 인지 ‘거짓’ 인지 판별하는 문제에서 사용한다. Linear regression 을 사용하여 0.5 이상이면 1, 아니면 0 과 같은 방식을 사용하면 될 것이라 생각할 수도 있지만 이는 부정확한 방법이다.

우리는 activation function을 사용하여 regression 을 진행하게 된다. 즉, sigmoid function 으로 regression 시킨다. 이를 그림으로 나타내면 다음과 같다.



$$\text{Linear : } y_i = \theta^T x_i$$

$$\text{Logistic : } P(y_i = 1) = \sigma(\theta^T x_i)$$

실제로 logistic 모델에서 주는 값이 확률은 아니지만, 확률처럼 계산이 가능하다. Logistic Regression 의 경우 Cost function 으로 Cross Entropy 를 사용한다. 이런 편이 계산도 편하고 좋은 결과를 얻을 수 있기 때문이다. 구조를 정리하면 다음과 같다.

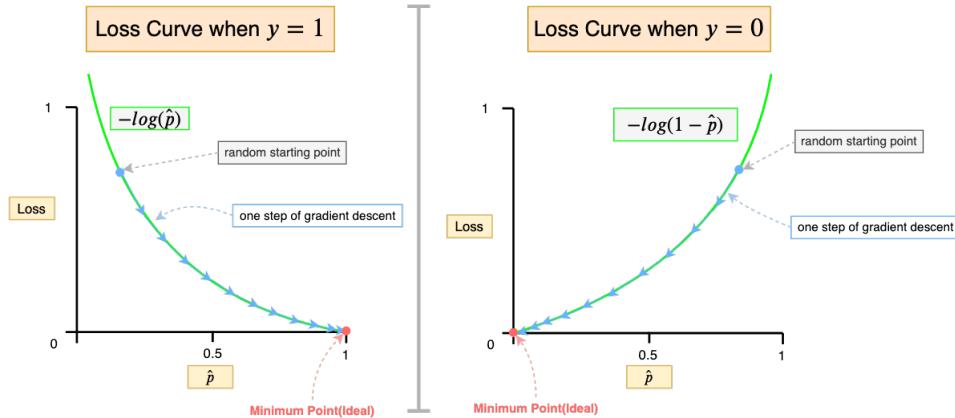
Training Data : (\mathbf{x}, y)

$$\text{Hypothesis : } y \approx h_\theta(\mathbf{x}) = \sigma \left(\sum w_k x_k \right) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

$$\begin{aligned}\text{Cost : } L(h_\theta(\mathbf{x}), y) &= \begin{cases} -\log h_\theta(\mathbf{x}) & \text{if } y = 1 \\ -\log (1 - h_\theta(\mathbf{x})) & \text{if } y = 0 \end{cases} \\&= -y \log h_\theta(\mathbf{x}) - (1 - y) \log (1 - h_\theta(\mathbf{x}))\end{aligned}$$

$$\mathbf{x} \rightarrow z = \sum w_k x_k \rightarrow y = h_\theta(\mathbf{x})$$

위의 error function 을 보면, $y = 1$ 이고 machine output 이 0에 가까우면 cost function 값이 매우 커지며 1에 가까우면 매우 작아짐을 알 수 있다. 또한 $y = 0$ 이고 machine output 이 1에 가까우면 cost function 값이 매우 커지며 0에 가까우면 매우 작아짐을 알 수 있다.



위의 구조는 데이터 수가 한개일 경우이다. 데이터 수가 m 일 경우 이전에서와 같이 합치면 된다.

Training Data : $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$

$$\text{Hypothesis} : y \approx h_{\theta}(\mathbf{x}) = \sigma\left(\sum w_k x_k\right) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

$$\text{Cost} : L(h_{\theta}(\mathbf{x}), y) = -\frac{1}{m} \sum_{i=1}^m \left[y_i \log h_{\theta}(\mathbf{x}_i) + (1 - y_i) \log (1 - h_{\theta}(\mathbf{x}_i)) \right]$$

$$\mathbf{x} \rightarrow z = \sum w_k x_k \rightarrow y = h_{\theta}(\mathbf{x})$$

Cross Entropy 를 사용할 경우 gradient 계산이 매우 쉬워진다. sigmoid function의 경우 앞서 살펴봤듯이 미분이 간단한 형태로 다음과 같이 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 인 것을 알고 있으므로, 다음과 같다.

$$\begin{aligned} \frac{\partial L}{\partial w_k} &= -\frac{1}{m} \sum_{i=1}^m \left[y_i \cdot \frac{1}{h_{\theta}(\mathbf{x}_i)} \frac{\partial h_{\theta}(\mathbf{x}_i)}{\partial w_k} - (1 - y_i) \frac{1}{1 - h_{\theta}(\mathbf{x}_i)} \frac{\partial h_{\theta}(\mathbf{x}_i)}{\partial w_k} \right] \\ &= -\frac{1}{m} \sum_{i=1}^m [y_i \cdot (1 - h_{\theta}(\mathbf{x}_i)) x_i^k - (1 - y_i) h_{\theta}(\mathbf{x}_i) x_i^k] \\ &= \frac{1}{m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}_i) - y_i) x_i^k \end{aligned}$$

따라서 parameter w_k 의 update 식은 다음과 같다.

$$w_k^{t+1} = w_k^t - \eta \frac{\partial L}{\partial w_k} = w_k^t - \eta \cdot \frac{1}{m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}_i) - y_i) x_i^k$$

Summary 기계학습은 모두 비슷한 과정을 거친다. 이를테면 어떤 환자의 건강검진 결과를 가지고 특정 질병이 발생할 확률을 얻는 Machine 을 만들고 싶다고 하자. 이 때 신경망을 구성하는 과정은 다음과 같다.

1. Data 를 수집한다. 환자의 기록은 비공개이기 때문에, 각자에게 동의를 얻어 수집하는 과정에서 시간과 돈이 많이 들 것으로 예상된다.
2. 수집한 Data 를 목적에 맞게 가공하고 정리한다. 수집한 Dataset 안에 정답이 있는 경우 Supervised Learning 이다. 우리는 질병이 발생할 확률을 얻어야 하기 때문에 각자가 그 질병에 걸렸는지도 조사를 했어야 한다.

3. 적당한 Model (= Architecture, Structure) 를 정한다. 즉, 다양한 DNN 의 방법들 중 어떤 것을 어떻게 사용할지 선택한다.
4. 위에서 정의한 Model 에 Data 를 넣어 결과값을 얻는다. Cost (Error, Loss function) 를 목적으로 맞게 적절히 정의한다. backpropagation 을 통해 Model 의 weight 를 조정하는 과정을 반복한다.

2 Lecture 2 : Deep Neural Network II

2.1 Theoretical Background of Neural Network

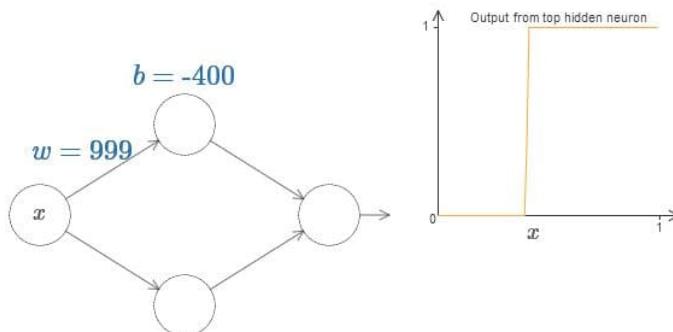
Universal Approximation Theorem : \mathbb{R}^n 안에 있는 임의의 연속함수는 비선형 activation function 을 갖는 뉴런 유한개로 구성된 하나의 hidden layer 을 갖는 neural network 로 적절히 근사될 수 있다. 또한, neuron 의 개수를 늘림으로서 근사를 더 좋게 만들 수 있다.

Cybenko-Hornik-Funabashi Theorem

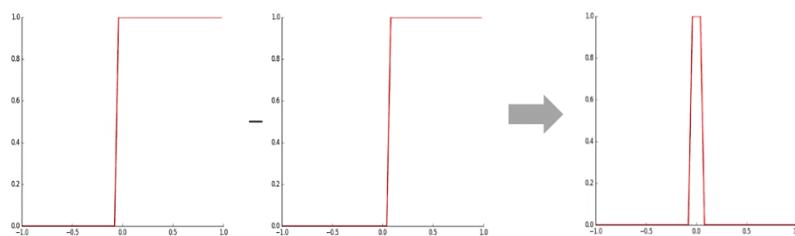
The sum of the following form can approximate any continuous function F on $[0, 1]^n$ to any degree of accuracy:

$$F(x) \approx \sum_k c_k \sigma \left(\sum_{i=1}^n w_{ki} x_i + b_k \right)$$

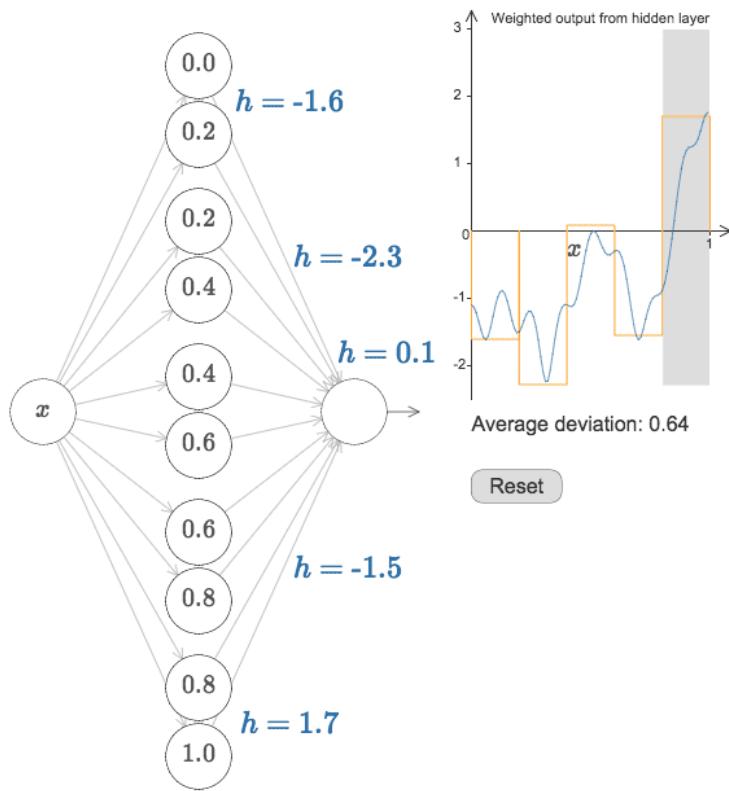
다음은 위의 이론에 대한 증명의 intuition 이다. 먼저 step function 을 만든다. activation function 으로 sigmoid 를 사용하고 그 steepness 를 계속해서 늘리면 된다.



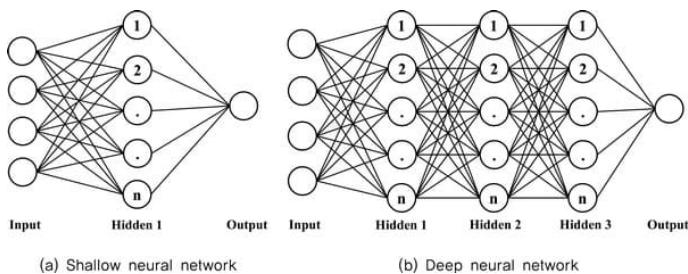
만약 임의의 연속 비선형함수를 적분할 때와 같이 미소구간으로 나누고, 이 미소구간을 모두 neural network 로 구현할 수 있다면 이들을 결합하여 원래의 비선형함수를 만들 수 있을 것이다. 즉, step function 두개를 합치면 특정 미소구간에서의 bump function 이 나오고, 이들을 모으면 임의의 비선형함수가 된다.



다음은 임의의 함수를 approximate 하기 위해 neural network를 구성한 모습이다. 위에서 언급했듯이 서로 붙어있는 쌍들은 하나의 bump function 을 나타낸다. 하지만 정확성을 높이기 위해서는 함수를 더 분할해서 더 많은 neuron 을 사용해야 할 것이다.



특정 비선형 함수를 높은 정확도를 가지고 근사하기 위해서는 보통 1 억개 이상의 neuron 이 필요하다. 즉, training 과정이 너무 느려지게 된다. 1 억개의 neuron 을 가진 single layer NN 과 비슷한 성능을 내기 위해서는 network 의 진행 가능 연결이 이와 비슷해야 한다. 그렇다면 layer 수가 적은, shallow network 와 layer 수가 많은, deep network 의 성능은 이와 비슷하지 않을까?



만약 둘의 노드 수가 같다고 한다면, deep network 쪽의 connection 이 훨씬 적어지며 input data 가 갈 수 있는 길은 훨씬 많게 된다. (즉, feature 가 다양한 방식으로 output 에 영향을 줄 수 있게 된다.) connection 각각에 weight parameter 가 할당된다는 것을 기억한다면, gradient descent 방법을 사용하기 위해 deep network 의 연산이 더 빨라진다는 것을 알 수 있다. 또한 약간의 노드 수를 늘림으로서 shallow network 보다 더 좋은 성능을 얻을 수 있다.

따라서 우리는 layer 를 늘리고 점점 더 deep 한 network 를 사용하게 된다. 이는 다시 vanishing gradient problem 으로 우리를 돌려놓게 되는데, 이제 ReLU 등에 대해 살펴볼 때가 되었다.

Geoffrey Hinton's Summary Geoffrey Hinton 교수는 매우 좋은 방법들을 제시하며 DL 이 나아가야 할 방향을 제시했다. 몇 가지를 살펴보고 넘어가자.

- 데이터셋이 적절한 결과를 내기에는 10^3 배 정도 작다.
⇒ Big Data, IoT
- 컴퓨터가 10^7 배 정도 느린다.
⇒ GPU, TPU
- Weight 를 초기화해야 하는데 어디서 시작해야 할지 잘 모르겠다.
⇒ RBM, Xavier initialization(2010), He initialization(2015, He라는 사람이 Xavier 변형)
- 잘못된 비선형함수를 사용했다.
⇒ ReLU(2007)

2.2 Nonlinearity

Activation Functions 앞서 Universal Approximation Theorem에서 언급했듯이 activation function으로서 비선형 함수가 필요하다. 이들의 종류를 살펴보면 다음과 같다. (Linear 함수를 사용할 경우 모든 Network가 하나의 선형 결합으로 귀결된다.)

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Copyright © Sebastian Raschka 2016
(<http://sebastianraschka.com>)

자주 사용되는 비선형함수들은 다음과 같다.

1. Sigmoid

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

앞서 살펴봤듯이 step function을 만들 수 있으며 함수값이 (0,1)에 존재하기 때문에 값이 explode하지 않는 장점이 있다. 또한 모든 영역에서 미분 가능이며 그 미분이 매우 좋은 모양으로 나온다.

$$\sigma'(z) = \sigma(z)[1 - \sigma(z)]$$

2. tanh

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{2}{1 + e^{-2z}} - 1 = 2\sigma(2z) - 1$$

3. ReLU

$$f(z) = \max(0, x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

0 보다 큰 영역에서 ReLU 를 미분할 경우 1 이 되어 vanishing gradient 문제가 해결된다. 또한 특정 구간에서 선형이기 때문에 계산이 간단하고 연산이 빠르다. 2015년 이후로 대부분의 문제에서 이것을 사용한다. 단점으로는 함수값이 explode 할 가능성도 있다. 또한 ReLU 를 살펴보면 0 보다 작은 것들은 모두 무시하는 것을 알 수 있다. training 과정에서 음수값이 나오면 weight 를 0 으로 학습해버려 노드가 사라지는 문제가 있다. 가끔은 0 보다 작은 데이터를 무시하면 안될 경우가 생기는데, 이때문에 Leaky ReLU 를 사용하기도 한다.

4. Leaky ReLU

$$f(z) = \max(0.1x, x) = \begin{cases} 0.1x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

2.3 Optimization II

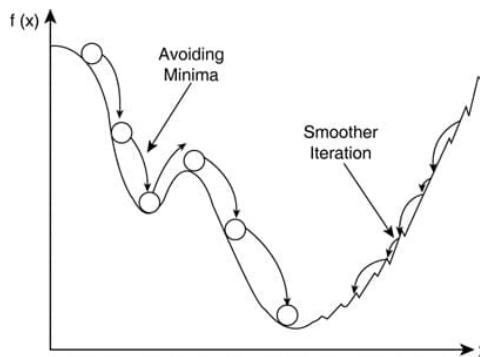
Gradient Descent II 이전에는 Batch 방식에 따라 Gradient Descent 를 살펴보았으며 보통 Mini-batch 방식을 사용한다고 했다. 여기에서는 더 추가되는 방식들을 다룬다.

1. Momentum

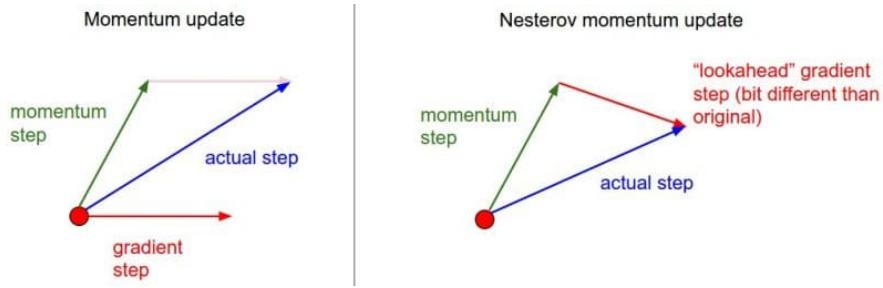
Momentum 방식은 말 그대로 Gradient Descent 과정에서 관성을 주는 것 이다. 현재 Gradient 를 통해 이동하고자 하는 벡터와, 과거에 이동했던 방향의 벡터가 합쳐진 벡터를 갖는다. 수식은 다음과 같다.

$$\Theta_{n+1} = m \cdot \Theta_n - \eta \cdot \nabla_{\Theta} L(\Theta) \Big|_{\Theta_n}$$

보통 momentum m 값으로 0.9를 사용한다. 이 방식을 사용하면 global minimum 을 찾을 수도 있을 것이라는 기대를 준다. 이를테면 너무 깊지는 않은 local minimum 으로 수렴하는 gradient descent 방식에서 momentum 방식을 추가한다면 이 local minimum 의 벽을 넘고 다른 minimum 을 찾기 시작할 것이다.



2. Nesterov Accelerated Gradient



Momentum 방식에서는 이동벡터 Θ_{n+1} 을 계산할 때 현재 위치에서의 gradient step 과 momentum step 을 합쳐 actual step 을 만든다. Nesterov 방식은 현 위치에서 momentum step 을 먼저 이동한 후, 그 위치에서의 gradient 를 구해서 gradient step 을 이동한다.

3. AdaGrad (adaptive gradient)

이전까지 학습률 η 는 hyperparameter 로서 고정되어 다뤄졌다. 큰 학습률은 빠른 학습을, 작은 학습률은 느리지만 정확한 학습을 보장하는 경향이 있다. 하지만 정해진 학습률을 가지고 계속 학습하다 보면 minimum 에 가까워졌을 때 진동하는 경우가 생기기도 한다. 때문에 처음에는 큰 학습률을 사용하다가 점차 학습률을 줄일 필요가 있는데, 이를 수식적으로 만든 것이 AdaGrad 이다.

$$\Theta_{n+1} = \Theta_n - \frac{\eta}{\sqrt{G_n + \epsilon}} \nabla_{\Theta} L(\Theta) \Big|_{\Theta_n}$$

$$G_n = \sum_{i=1}^n \left[\nabla_{\Theta} L(\Theta) \Big|_{\Theta_i} \right]^2$$

위에서 G_n 은 증가함수이다. 따라서 학습률은 계속해서 줄어든다는 것을 알 수 있다. 이 식은 증명하기보다는 여러 시도를 하던 중 경험적으로 ‘잘 되더라’ 라는 것 이다. 또한 보통 $\eta = 10^{-8}$ 를 사용하는데 이는 처음에 분모가 0 이 되는 것을 방지하기 위함이다.

4. RMSProp (root mean square propagation)

AdaGrad의 학습률이 급격히 줄어드는 문제를 해결한 방법이다.

$$\Theta_{n+1} = \Theta_n - \frac{\eta}{\sqrt{G_n + \epsilon}} \nabla_{\Theta} L(\Theta) \Big|_{\Theta_n}$$

$$G_n = \gamma \cdot G_{n-1} + (1 - \gamma) \left[\nabla_{\Theta} L(\Theta) \Big|_{\Theta_n} \right]^2$$

AdaGrad 의 식은 RMSProp 의 G_n 식에서 $\gamma \rightarrow 1$, $(1 - \gamma) \rightarrow 1$ 로 둔 것과 같다는 것을 알 수 있다. AdaGrad 는 계속해서 계산되는 gradient 값 모두에 동등한 중요도를 두었다면, RMSProp 은 지금 계산된 gradient 값과 이전에 계산된 값의 중요도 합을 1 로 두고 trade off 시킨 것이다. γ 의 기본값은 keras 기준 0.9 인데, 이는 ‘앞의 것들’ 에는 영향력을 줄이고, ‘지금 나온 것’ 에는 영향력을 좀 더 부여한다는 것 이다.

5. Adam (adaptive moment estimation)

RMSProp 과 Momentum 방식을 합친 방법이다. 기본적으로 $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

이다.

$$\hat{m}_\Theta = \frac{m_\Theta^{n+1}}{1 - (\beta_1)^{n+1}} \quad \text{where} \quad m_\Theta^{n+1} = \beta_1 m_\Theta^n + (1 - \beta) \nabla_\Theta L(\Theta) \Big|_{\Theta_n}$$

$$\hat{G}_\Theta = \frac{G_\Theta^{n+1}}{1 - (\beta_2)^{n+1}} \quad \text{where} \quad G_\Theta^{n+1} = \beta_2 G_\Theta^n + (1 - \beta_2) \left[\nabla_\Theta L(\Theta) \Big|_{\Theta_i} \right]^2$$

$$\Theta_{n+1} = \Theta_n - \eta \frac{\hat{m}_\Theta}{\sqrt{\hat{G}_\Theta + \epsilon}}$$

보통 Adam 을 사용한다.

Training and Test sets DataSet 이 주어졌을 때, 가장 처음으로 이들을 용도에 따라 나눠야 한다. 순서대로 보자.

- dataset의 순서를 랜덤으로 섞는다. 만약 dataset 이 처음 1000 개는 장미, 다음 1000 개는 국화, ... 등으로 되어있다면 그 데이터를 그대로 넣으면 편향이 심하게 생길 것 이다.
- training 70%, test 30% 로 나누거나, training 60%, validation 20%, test 20% 로 나눈다. 만약 100% 로 training 한 후 그 데이터로 test를 한다면 machine의 test score가 매우 좋을 것 이다. 하지만 이후 새로운 데이터를 그 machine 에 넣는다면 잘 맞추지 못할 것 이다. validation 의 경우 **Cross-Validation** 방법을 이용하고는 한다.

training data 와 validation data 를 합한 것을 \mathfrak{D} 라 하자. 그렇다면

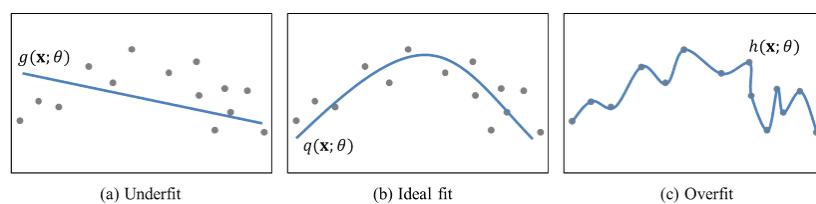
$$\mathfrak{D} \rightarrow \{\mathfrak{D}_1, \dots, \mathfrak{D}_k\}$$

와 같이 random 분할 한 뒤 \mathfrak{D}_i 만을 선택해 이를 제외한 data 로 training 하고 \mathfrak{D}_i 를 이용해 score 를 낸다. $i = 1, 2, \dots, k$ 에 대해 이 과정을 반복한 뒤 score 분포의 MSE 를 사용한다. 이것이 Cross-Validation 이다.

- 간혹 원래 데이터 수가 부족해서 원래 데이터를 VAE 나 Augment 해서 그 수를 늘려 사용하는 경우가 있다. 1000 개의 데이터를 100 배로 만들어 10^5 개의 데이터가 생겼다면, 이 10^5 개에서 training 과 test set 을 나누면 안된다. 기존의 데이터에서 얻은 데이터는 기존의 데이터와 유사하기 때문에 test set 으로 사용할 수 없다. 따라서 1000 개에서 먼저 set 을 나누고, training set 700 개에서 VAE 나 Augment 방식을 사용해야 할 것 이다.

2.4 Overfitting

Bias-Variance Tradeoff Bias, Variance 는 Prediction error 의 종류로, 이들을 이해해야 overfitting 과 underfitting 을 이해할 수 있다. underfitting 은 단순히 성능이 좋지 않은 문제로 해결하기 쉽다. 우리는 overfitting 문제를 주로 해결하게 될 것이다.



위의 그림을 고려하여 생각해 보자.

- **Bias** : 원래 데이터 (training data) 와 fitting된 값 (machine) 의 차이를 의미한다. 물론 우리는 machine learning 방식을 사용하기 때문에 fitting된 값이란 여러 fitting 들의 평균값을 말한다. 이것이 크다는 것은 machine의 성능이 좋지 못하다는 것을 의미한다. 즉, underfitting 되었다는 것을 의미한다. 이는 network 자체가 잘못 설계된 경우로, hyperparameter 를 수정하면서 쉽게 해결할 수 있다.
- **Variance** : fitting 된 값 (machine) 들 서로간의 차이를 의미한다. 이것이 크다는 것은 training 할 때마다 매우 다른 결과를 얻는다는 것이다. 즉, training 과정에서 overfitting 이 발생했다는 것을 의미한다. overfitting 이 발생했을 경우 training 결과의 bias 는 매우 낮겠지만, test data 를 machine 에 넣으면 error 가 매우 클 것이다. 즉, 주어진 데이터에 대해서는 매우 꼭 맞겠지만, 다른 데이터가 들어오면 틀린 결과를 얻는다.

그렇다면 이들을 어떻게 해결해야 할까? 다음은 간단히 정리한 해결 방안이다.

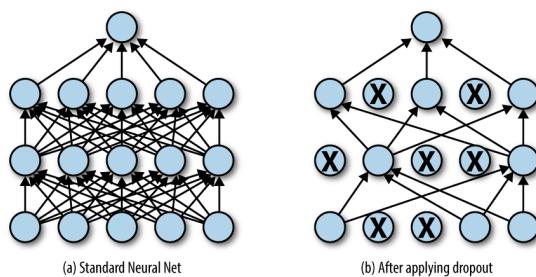
- Underfitting

1. 더 오래 학습한다.
2. 모델을 더 복잡하게 만든다.
3. feature 를 추가한다.
4. Regularization 을 줄인다.
5. 다 안되면, 새로 만든다.

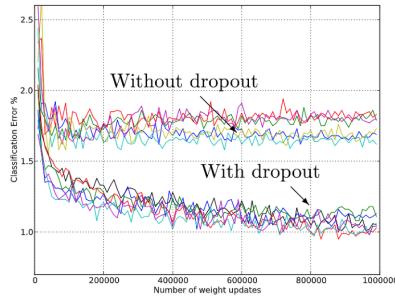
- Overfitting

1. 더 많은 데이터를 얻는다.
2. feature 를 줄인다.
3. Regularization 을 늘린다.
4. 다 안되면, 새로 만든다.

Dropout 각각의 mini-batch 에 대하여 forward pass 를 거칠 때, 미리 정해준 비율만큼의 neuron 을 생략한다. 물론, node 를 생략하는 과정은 확률에 근거하며, 해당 node 의 weight 를 0 으로 두면 node 가 사라지는 효과가 난다.

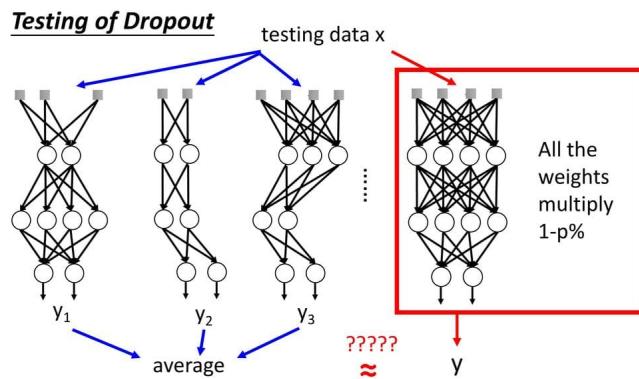


이는 parameter 를 줄여주는 효과가 있으므로 Overfitting 을 방지한다. 또한 connection 의 개수는 node 가 줄어든 것 보다 훨씬 많이 줄어든다. 즉, 계산량이 매우 많이 줄어 연산이 빨라진다.



Dropout은 Ensemble Learning과 닮아있는데, 각 mini batch마다 (아마도) 서로 다른 network를 가지고 학습할 것 이므로 그렇게 볼 수도 있다.

Dropout is a kind of ensemble.



참고로, Dropout parameter p 는 Keras의 경우 dropout probability를, TensorFlow의 경우 keep probability를 말한다. 또한 training에서는 dropout을 사용하지만, test에서는 dropout을 사용하지 않는다. 그런데 영향력을 살펴보면 test 할 때는 weight에 적절한 값을 곱해줘야 한다.

다음은 Dropout을 약간 변형한 아이디어들이다.

- Dropout은 Maxout과 잘 맞는다.
- Dropconnect : node를 drop하는 것이 아니라, connection을 지운다.
- Annealed Dropout : epoch마다 dropout rate를 줄인다.
- Standout : 각 neuron마다 dropout rate를 다르게 준다.

Regularization Weight의 크기를 제한함으로서 overfitting을 방지하는 방법이다.

1. L1 Regularization (L1 norm)

$$L_{L1} = \frac{1}{2m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 + \frac{\lambda}{m} \sum_{j=1}^k |w_j|$$

Weight의 크기가 기존에 알던 loss function L 에 더해져 있다. 따라서 gradient descent 방법을 사용하면 L 도 줄어들지만, weight의 크기 $|w_j|$ 도 줄어들게 된다.

2. L2 Regularization (Weight decay, L2 norm)

$$L_{L2} = \frac{1}{2m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 + \frac{\lambda}{2m} \sum_{j=1}^k w_j^2 = L + \frac{\lambda}{2m} \sum_{j=1}^k w_j^2$$

L1 과 다른 것은 Weight 가 제곱되어 더해졌다는 것 이다. 때문에 미분이 가능해 계산이 쉽다. 이를 이용하면 다음과 같이 계산이 가능하다.

$$\begin{aligned} \Rightarrow w_j(n+1) &= w_j(n) - \eta \frac{\partial L_{L2}}{\partial w_j} \Big|_{w_j(n)} = w_j(n) - \eta \left[\frac{\partial L}{\partial w_j} + \frac{\lambda}{m} w_j(n) \right] \\ &= \left(1 - \frac{\eta\lambda}{m}\right) w_j(n) - \eta \frac{\partial L}{\partial w_j} \end{aligned}$$

이 때 $-\frac{\eta\lambda}{m}$ 으로 인해 Weight가 decay 하게 된다.

Normalization and Standardization 서로 다른 단위를 가진 여러 feature 들을 무차원화시키는 방법이다. 통계에서는 ‘정규화’ 라는 주제 아래 각각 ‘최소 최대 정규화’ 와 ‘표준점수’ 라는 이름으로 묶인다.

1. Normalization (into [0,1])

$$x'_j = \frac{x_j - \min(x)}{\max(x) - \min(x)}$$

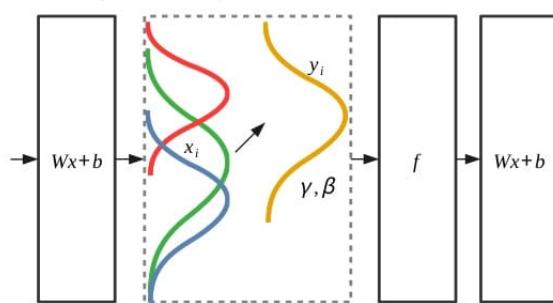
2. Standardization

$$x'_j = \frac{x_j - \text{mean}(x)}{\text{std}(x)} = \frac{x_j - \mu}{\sigma}$$

Batch Normalization (BN) 특정된 하나의 layer 를 A 라고 하고 관찰해보자. **Internal Covariate Shift** 란 training 과정에서 A 직전 layer 의 parameter 가 수정되면서 결과적으로 A 에 들어오는 layer input 의 분포가 달라지는 것을 말한다. 이렇게 되면 input data 에 Normalization이나 Standardization 을 가했어도 layer input 들은 분포가 계속 바뀌고 제각각이기 때문에 학습이 더뎌진다. 따라서 우리는 Batch Normalization 을 하게 된다.

Batch normalization

Ensure the output statistics of a layer are fixed.



즉, A 이전 layer 들에서 나온 것들이 weight 를 거쳐 바로 A 의 activation function 에 들어오는 것이 아니라, weight 를 거친 것들을 standardization 해서 다시 (learnalbe parameter γ, β 를 이용해) 선형결합한 뒤 activation function 에 넣어주는 구조가 된다. BN algorithm 은 다음과 같다.

Input:	Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
	Parameters to be learned: γ, β
Output:	$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
	$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ // mini-batch mean
	$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance
	$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize
	$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$ // scale and shift

이 때 A 이전 layer 들에서 나온 것들이 weight 된 후 $B = \{x_1, \dots, x_m\}$ 으로 들어온다. 이들의 분포를 이용해 standardization 된 \hat{x}_i 가 만들어지고, 이를 선형결합해 y_i 가 A 의 activation function 에 들어가게 된다. 이제 BN 을 추가하는 알고리즘에 대해 알아보자.

Input:	Network N with trainable parameters Θ ;
	subset of activations $\{x^{(k)}\}_{k=1}^K$
Output:	Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$
1:	$N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: for $k = 1 \dots K$ do	
3:	Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. I)
4:	Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: end for	
6:	Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^K$
7:	$N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: for $k = 1 \dots K$ do	
9:	// For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10:	Process multiple training mini-batches \mathcal{B} , each of size m , and average over them:
	$E[x] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$
	$\text{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$
11:	In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma, \beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: end for	

우선 BN을 추가하기 전에 $N_{\text{BN}}^{\text{inf}}$ 라는 초기 network 를 저장한다. 그리고 BN 을 추가하여 training 작업을 진행할 $B_{\text{BN}}^{\text{tr}}$ 을 저장한다. 이제 $B_{\text{BN}}^{\text{tr}}$ 에 대해 BN 을 추가하기로 정한 $1, 2, \dots, K$ node 들에 대해

$y^{(k)} = \text{BN}_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$ 알고리즘을 추가한다. 그리고 parameter $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}|_{k=1, \dots, K}$ 에 대해 학습을 진행한다. 이제 N_{BN}^{inf} 를 B_{BN}^{tr} 로 지정한다.

2.5 Weight Initialization

Weight Initialization 기존에는 Weight의 초기값으로서 0, 1, $\text{rand}(0,1)$ 등을 사용했다. 하지만 이들은 다음과 같은 문제를 야기했다.

- Zero Initialization : weight 가 symmetric하게 되어 network 가 linear matrix로 귀결된다.
- Random Initialization : Weight Matrix W 의 standard deviation이 1과 같이 높은 값일 경우 $|W \cdot x + b|$ 가 매우 커지게 되고 따라서 $\sigma(W \cdot x + b)$ 는 1이나 0이 된다. 따라서 vanishing gradient 문제를 야기한다. 혹은 Weight Matrix W 의 standard deviation이 0과 같이 낮은 값일 경우 $W \cdot x + b$ 는 0에 가깝게 되고 따라서 $\sigma(W \cdot x + b)$ 는 0.5 근방이 되어 적절한 학습이 이루어지지 않는다.

이러한 문제를 해결하기 위해 고안된 Initialization은 다음과 같다. 주된 아이디어는 이전 layer에서 들어오는 것들이 weight 되어 합쳐질 때, 그 값을 적절히 줄이는 것이다.

LeCun Normal Initialization

$$W \sim N\left(0, \frac{1}{\text{node}_{in}}\right)$$

Xavier Normal Initialization

$$W \sim N\left(0, \frac{2}{\text{node}_{in} + \text{node}_{out}}\right)$$

He Normal Initialization

$$W \sim N\left(0, \frac{2}{\text{node}_{in}}\right)$$

LeCun Uniform Initialization

$$W \sim U\left(-\sqrt{\frac{1}{\text{node}_{in}}}, \sqrt{\frac{1}{\text{node}_{in}}}\right)$$

Xavier Unifrom Initialization

$$W \sim U\left(-\sqrt{\frac{6}{\text{node}_{in} + \text{node}_{out}}}, \sqrt{\frac{6}{\text{node}_{in} + \text{node}_{out}}}\right)$$

He Uniform Initialization

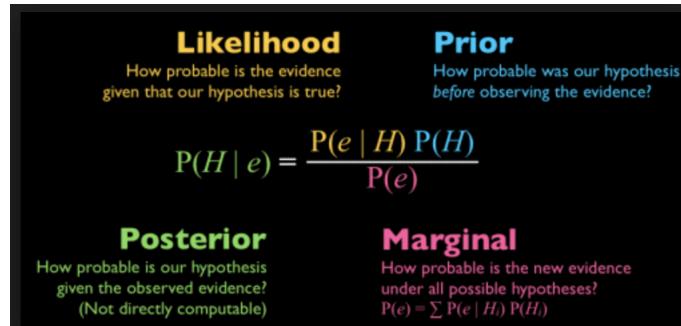
$$W \sim U\left(-\sqrt{\frac{6}{\text{node}_{in}}}, \sqrt{\frac{6}{\text{node}_{in}}}\right)$$

이 때 node_{in} 은 해당 layer에 들어오는 node의 개수 (layer input의 개수), node_{out} 은 해당 layer에서 나가는 node의 개수 (layer output의 개수)이다.

2.6 Confusion Matrix and Test Score

2.6.1 Bayes hypothesis

어떤 피의자에게 판결을 내리기 위해 통계를 사용한다고 가정하자. 우리는 그 도시의 범죄자에 대한 비율과 그 중 잡힌 범죄자에 대한 비율을 알고 있다. 또한 현장에서 발견된 증거 X 가 있다. 두 가설 H_0 (귀무가설)과 H_1 (대립가설)이 있고 prior가 있다. (이들을 알고 있다는 가정이 필요하다.)



Bayes 의 방법은 $X = x$ 를 얻었을 때 이로부터 update 한 두 가설의 posterior 를 비교하여 둘 가운데 하나를 선택한다. 즉 다음의 경우에 H_1 을 선택한다:

$$P(H_1 | X = x) > P(H_0 | X = x)$$

이는 다음과 같이 정리가 된다.

$$L(x) = \frac{P(X = x | H_1)}{P(X = x | H_0)} > \frac{P(H_0)}{P(H_1)} = \xi \quad (\text{threshold})$$

이 때 $L(x)$ 를 우도비 (likelihood ratio) 라고 부른다. 즉, 피의자가 범인이 아니라고 가정했을 때 ($=P(X = x | H_0)$) 보다 범인이라고 가정했을 때 ($=P(X = x | H_1)$) 의 확률이 높다면 우리는 피의자가 범인이라고 판결한다.

한편 우리는 억울하게 범인으로 결정되는 일이 적기를 바란다. 따라서 다음과 같은 변수를 정의한다.

$$\alpha = P(\text{reject } H_0 | H_0)$$

즉, 사실은 H_0 인데 H_0 를 기각하는 확률이 작기를 바란다. 그런데 이를 줄이다 보면 다음과 같은 변수가 커진다.

$$\beta = P(\text{reject } H_1 | H_1)$$

즉, 사실은 범인인데 범인이 아니라는 판결이 늘어나게 된다. 따라서 적당한 중심을 잡아야 한다. α 와 β 중 어느 것이 작아지는 것이 더 중요한가는 경우에 따라 달라진다. 병에 걸렸는지 안걸렸는지 검증하는 screening test 의 경우는 병에 걸렸는데 걸리지 않았다고 판결하는 경우를 최소화하기를 바란다.

인간의 지식은 실세계의 상황을 여러 개의 법칙을 세워 언어로서 설명하고 이해하도록 만든 것이다. 그러나 인간의 경험(관측량)을 모두 법칙으로 설명하는 것은 불가능하다. 인간이 구체적으로 인지할 수 있는 법칙의 수는 실세계의 다양성에 비해 너무 모자란다. 이를 Knowledge Acquisition Bottleneck 이라 한다. 이에 대한 대안은 법칙 (즉, 함수) 대신에 확률을 사용한다. 관측량 x 가 갖는 값들의 확률분포 $P(x)$ 를 제시하는 방법을 사용한다.

2.6.2 Confusion Matrix

Performance Measurement 우선 machine 을 만든다. 해당 machine 이 얼마나 정확한지, 오차를 갖는지 확인하기 위해서는 performance measure 를 해야 한다. 이전에도 언급했듯이 우선 약간의 Training 을 거친 machine 은 아직 모든 regression 이 된 것이 아니기 때문에 validation data 로 해당 machine 을 검증한다. 최종적으로 완성된 machine이 어떤가는 test data 를 사용한다. 결과 분석 지표로서 몇 가지를 고려해 보자.

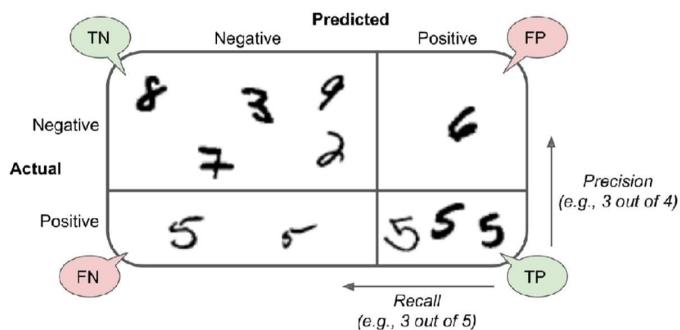
- Confusion matrix, Binary table 에 대해서 (즉, 결과가 True/False 일 경우) TP, FP, FN, TN 이 있다.

- Accuracy
- Sensitivity: Recall, True Positive Rate
- Specificity
- Precision: Positive Predictive Rate
- False Positive Rate: Type II Error
- False Negative Rate: Type I Error

Confusion Matrix Machine이 예측한 것과 실제 값이 맞는지 확인하는 Matrix이다.

	cotton crop	damp gray soil	gray soil	red soil	soil with vegetation stubble	very damp gray soil	
actual class	215	0	2	0	5	2	224
cotton crop							
damp gray soil	0	135	34	0	2	40	211
gray soil	0	16	368	1	0	12	397
red soil	1	0	2	458	0	0	461
soil with vegetation stubble	3	0	1	20	183	30	237
very damp gray soil	0	36	12	0	8	414	470
	219	187	419	479	198	498	
predicted class							

이들 중 하나의 feature를 골라 True 와 False 로 나누면 다음과 같은 계산이 가능하다. 다음과 같은 예시를 살펴보자.



정답은 5가 아니고(=Actual Negative) machine도 5가 아니라고(=Predicted Negative) 했을 경우 ‘TN’ (True Negative 즉, Negative인 것을 맞춘) 이다. 우측 하단의 ‘TP’의 경우를 이와 같이 해석하자면, ‘True Positive’ 즉, 정답은 5이고 machine도 5라고 맞춘 것이다. 이와 같은 표를 일반적으로 다음과 같이 표현한다. (혹은, Binary table이라고도 한다.)

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Accuracy (ACC) 좋은 지표는 아니지만 가장 먼저 보는 지표이다. 전체 데이터에서 맞춘 것의 비율을 본다.

$$\text{Accuracy} = \frac{\text{맞춘 개수}}{\text{전체 data 개수}} = \frac{\text{TN+TP}}{\text{total}}$$

또한 $\text{Error}=1-\text{Accuracy}$ 이다. 하지만 Accuracy만 보면 문제가 크다. 이를테면 희귀병 진단 검사에서 모집단은 크고 희귀병 집단은 매우 작다. 이 경우 그냥 ‘모두 정상이다!’ 라고 한다면 Accuracy는 매우 높을 것이다. 따라서, 중요한 것은 병에 걸린 사람 중에서 얼마나 제대로 진단했는가가 중요하게 된다. 즉, 우리는 FP, FN의 확률을 최소화하고 싶다. 물론, 고전통계에서 살펴봤듯이, 둘 다 0으로 수렴하게 만들 수는 없다.

Specificity (TNR) : True negative ratio, SPC, Selectivity

Specificity는 다음과 같다.

$$\mathbb{P}(N|No) = \frac{\text{TN}}{\text{TN+FP}}$$

그림에서 보면 다음과 같다.

$\frac{\text{TN}}{\text{TN+FP}}$	(Alarm) P Positive 판단	N Negative 판단
Cr 범죄자	TP True Positive	FN False Negative
No 정상인	FP False Positive	TN True Negative

고전통계와 비교하면 다음과 같다. (FPR)

$$\alpha = 1 - \text{Specificity} = \mathbb{P}(\text{accept } H_0 | H_1) : \text{False Positive Rate, FPR}$$

이는 Type I error이다.

Sensitivity (TPR) : True positive ratio, Recall

범죄자를 Cr, 정상인을 No 라고 하자. Sensitivity는 다음과 같다.

$$\mathbb{P}(P|Cr) = \frac{TP}{TP+FN}$$

그림에서 보면 다음과 같다.

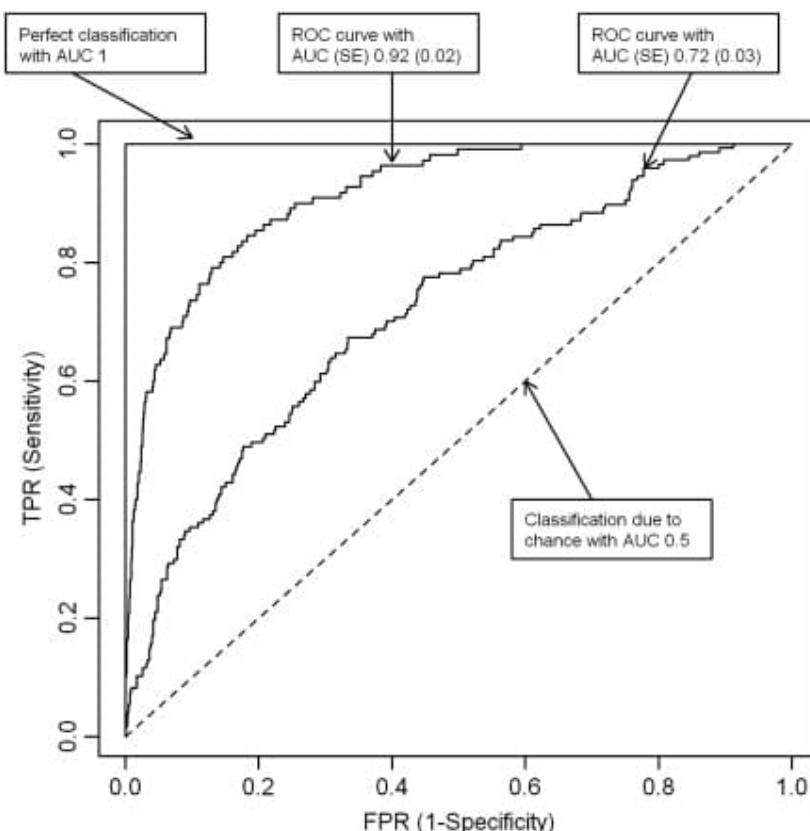
	$\frac{TP}{TP + FN}$	(Alarm) P Positive 판단	N Negative 판단
Cr 범죄자	TP True Positive	FN False Negative	
No 정상인	FP False Positive	TN True Negative	

고전통계와 비교하면 다음과 같다. (FNR)

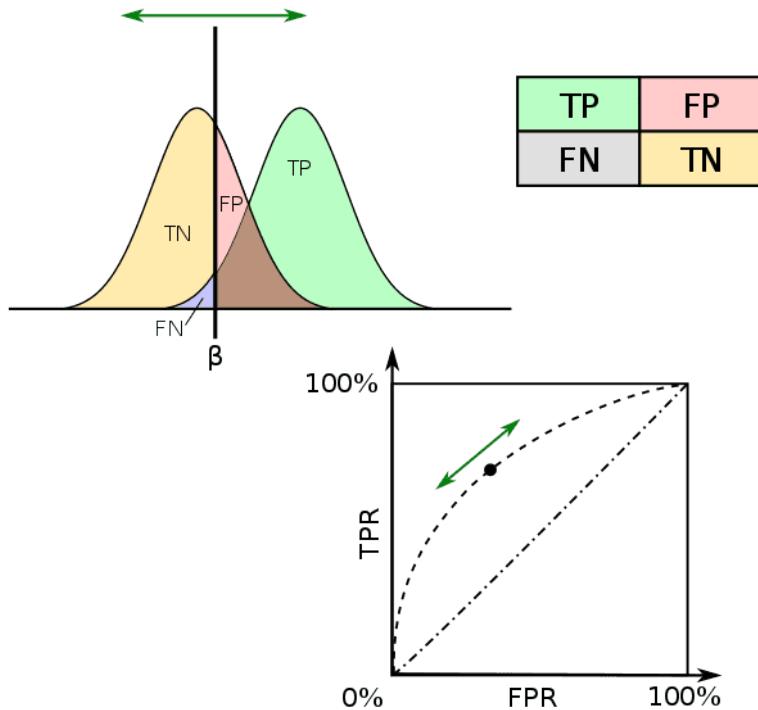
$$\beta = 1 - \text{Sensitivity} = \mathbb{P}(\text{reject } H_0 | H_0) : \text{False Negative Rate, FNR}$$

이는 Type II error이다.

ROC curve (Receiver Operating Characteristics) ROC 곡선은 주어진 모델(machine)에 대해서 가능한 모든 역치에서의 두 지표 TPR과 FPR의 관계를 그래프로 나타낸다. 즉, Sensitivity(TPR) vs 1-Specificity(1-TNR=FPR)의 그림이다.



우리는 FPR이 낮으면서 TPR이 높기를 바란다. 위의 예시에서 볼 수 있듯이, 그래프가 위쪽 사각형에 가까워질수록 좋은 성능을 보여준다. 여러가지 hyperparameter를 통해, 여러가지 threshold를 통해 ROC curve를 그리고 가장 정확한 machine을 선택하도록 한다. 만약 그래프가 아래로 많이 내려간다면, 결과를 한 번 뒤집으면 된다. 참고로, AUC (Area Under Curve) 란 이 그래프 아래의 면적을 나타낸다.



우측의 분포는 실제 positive인 것의 분포 $f_1(x)$ 이고, 좌측은 실제 negative인 것의 분포 $f_0(x)$ 이다. machine은 한계점 β 를 설정하여 이보다 우측에 있으면 positive라고 말하고, 좌측에 있으면 negative라고 말하기로 했다. FPR과 TPR은 다음과 같이 계산 할 수 있다.

$$\text{FPR}(\beta) = \int_{\beta}^{\infty} f_0(x)dx$$

$$\text{TPR}(\beta) = \int_{\beta}^{\infty} f_1(x)dx$$

따라서 ROC curve는 β 에 따라 달라질 것으로 예상할 수 있다.

만든 Machine을 사용할 때의 정확도 지표를 고려하자. 이제 분모가 달라진다.

1. **Positive predictive value (Precision)** : 범죄자로 판정된 사람 가운데서 실제 범죄자의 비

$$\mathbb{P}(Cr|P) = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

그림에서 보면 다음과 같다.

		$\frac{TP}{TP + FP}$	(Alarm) P Positive 판단	N Negative 판단
		Cr 범죄자	TP True Positive	FN False Negative
	Cr 범죄자			
	No 정상인	FP False Positive	TN True Negative	

2. Negative predictive value : 정상인으로 판정된 사람 가운데서 실제 정상인의 비

$$\mathbb{P}(No|N) = \frac{TN}{TN+FN}$$

그림에서 보면 다음과 같다.

		$\frac{TN}{TN + FN}$	(Alarm) P Positive 판단	N Negative 판단
		Cr 범죄자	TP True Positive	FN False Negative
	Cr 범죄자			
	No 정상인	FP False Positive	TN True Negative	

F_1 Score Classify 문제에서 간혹 사용하는 지표이다.

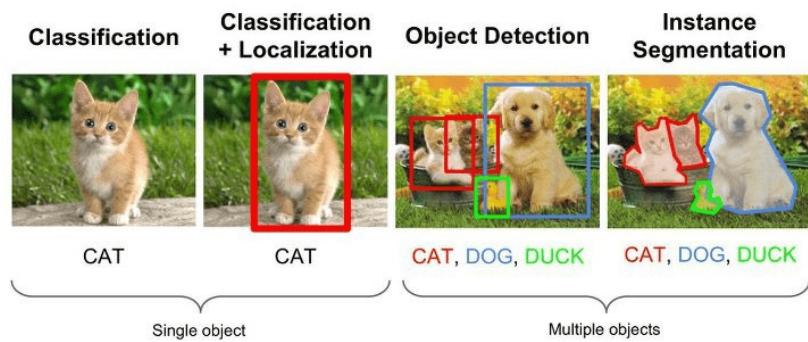
$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{TPR}} = \frac{2}{\frac{TP+FP}{TP} + \frac{TP+FN}{TP}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

3 Lecture 3 : Convolutional Neural Network

이전까지는 딥러닝의 기본적인 것들에 대해 살펴봤다. 이제부터는 그 기본에서 시작한 다른 중요한 기술들을 배울 것이다. 여기서는 사진을 input 으로 하는 machine 에 대해 다룰 것이다. 100×100 픽셀을 갖는 사진을 input 으로 하려면 input dimension 이 매우 커져버린다. 사진의 해상도가 높아지면 input dimension 은 제곱수로 빠르게 늘어날 것이다. 이를 바로 machine 에 넣어 학습하는 것은 불가능하다. CNN 은 사진 데이터를 적절히 변형해 학습을 가능하게 한다.

3.1 About CNN

CNN 은 shared weights architecture 와 translation invariance characteristics 에 기반하여 사진을 분석하는 방법이다. 그 방식은 다양하게 적용되는데, 다음 그림을 보자.

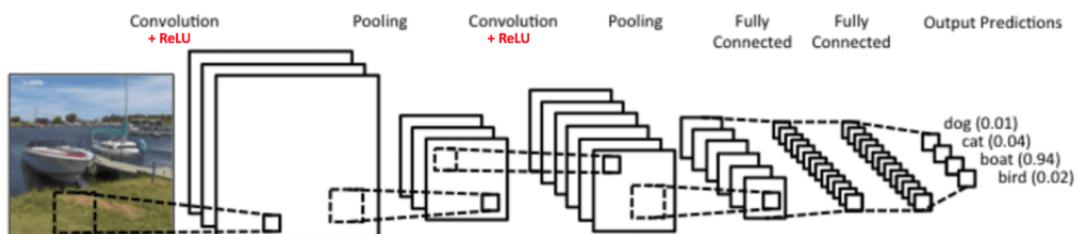


기본적인 아이디어는 ‘input dimension이 너무 높아 parameter가 너무 많다. weight 개수를 줄이자!’ 이다. 후에 설명되겠지만, weight sharing이라는 방법으로 parameter를 효과적으로 줄일 수 있다. 또한 사진은 찍는 구도에 따라 물체가 회전되어있을 수 있고, 크게 혹은 작게 찍혀있을 수 있다. 이런 다양한 구도에 관계없이 기능을 잘 하는 것도 관건이다. CNN 기법은 translation invariance 특징을 가지기 때문에 이 조건도 만족하게 된다.

후에 다뤄질 RNN과 함께 사용될 경우, 사진에 적합한 설명을 출력하는 것도 가능하다.



LeNet Yann LeCun은 최초의 CNN인 LeNet을 만들었다. 다음은 LeNet의 요약된 구조이다.

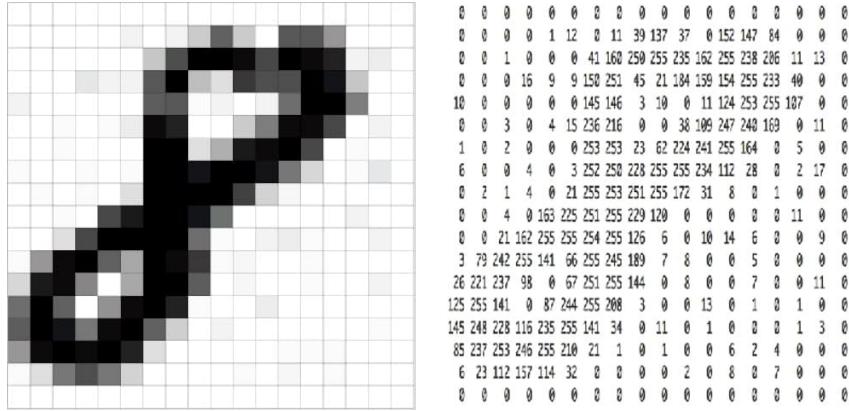


위의 그림에 해당하는 machine은 4개의 class에 대한 classification 을 출력한다.

Main Topics : LeNet 은 크게 4가지 단계로 이루어져있다. 이들은 CNN에서 중점적으로 다룰 내용들이다.

- Convolution (shared weights - parameter 개수를 줄임, dimension reduction - 정보를 최대한 유지하며 dimension 줄임)
- Nonlinearity (ReLU)
- Pooling (특히, Max Pooling - translation invariance, dimension reduction)
- FC (Fully connected layer)

Input Image CNN에 입력되는 데이터는 이미지이다. 모든 이미지는 색을 표현하는 숫자로 이루어진 행렬로 나타내어질 수 있는데, 특히 RGB로 표현된 이미지의 경우는 각 픽셀에 0부터 255까지 색상을 표현하는 숫자가 할당되며 이 행렬은 R, G, B의 3개 층(channel)을 이룬다. 흑백 2D 이미지는 RGB와는 달리 한개의 층을 가지며, 각 픽셀의 명도에 해당하는 0부터 255까지의 숫자를 가진다.



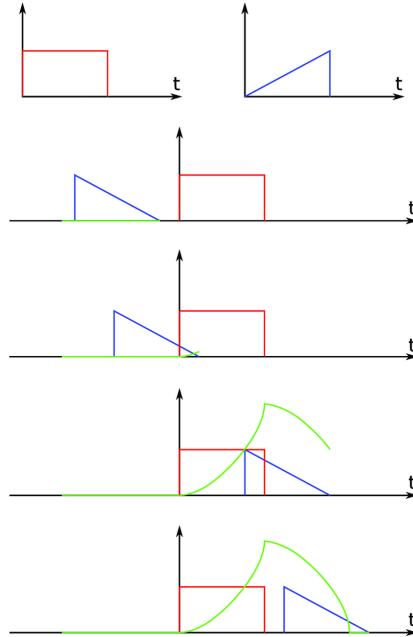
3.2 Convolution

3.2.1 Convolution in math

Convolution이란 무엇인가? 수학에서 사용하는 정의는 다음과 같다.

$$(f * g)(c) = \sum_a f(a) \cdot g(c - a)$$

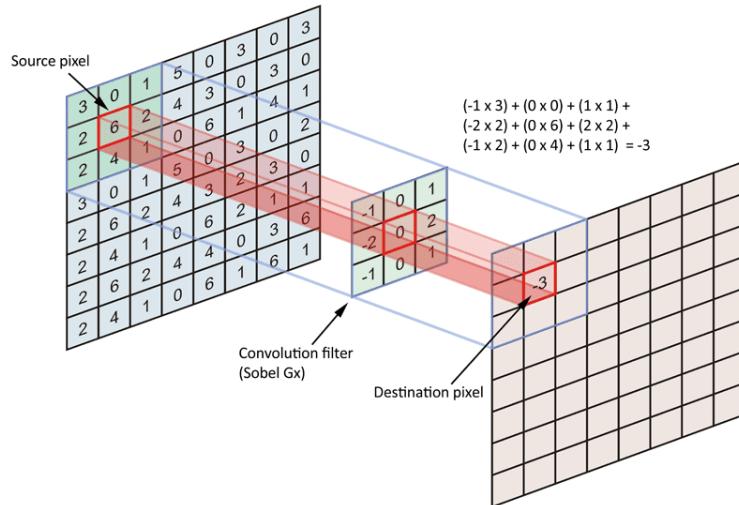
그동안 우리는 fourier transform을 사용하면 계산이 쉬워지기 때문에 convolution을 사용했다. 그렇다면 convolution의 도형적 의미는 어떨까? 다음의 그림을 보자.



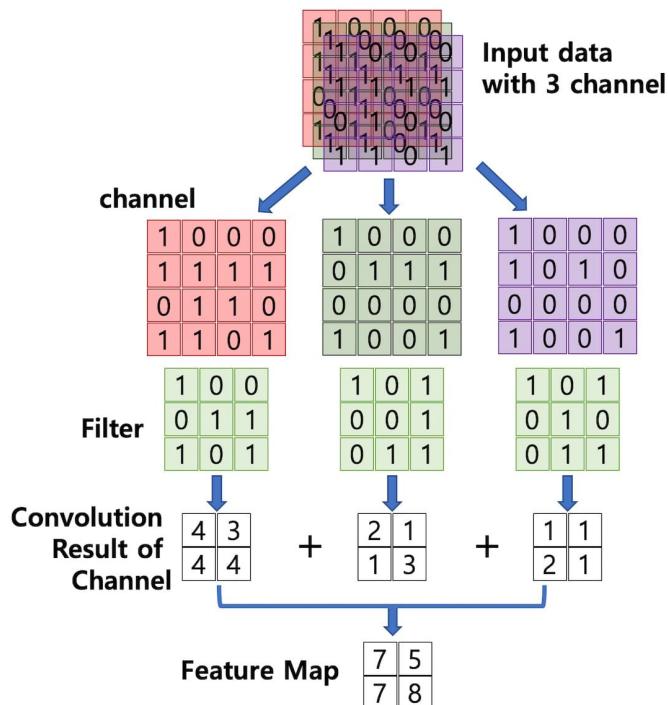
즉, convolution은 $g(-a)$ 를 c 만큼 평행이동시키면서 $f(a)$ 와 겹치는 면적 (적분)이다. 우리는 image processing을 위해 2차원 convolution을 사용한다. DL에서의 convolution은 수학적 정의와는 다르다. 하지만 작용하는 모습이 비슷하기에 이렇게 이름을 붙였다.

3.2.2 Convolution in DL

기존에 알던 machine을 사용한다면 input matrix에서 output matrix로 사상할 때 $\text{dim}(\text{input}) \times \text{dim}(\text{output})$ 만큼의 많은 weight가 필요하다. Convolution을 사용하면 weight를 매우 적게 줄일 수 있다. 기본적으로 작은 영역에서만 0이 아닌 값을 가지는 (supported) local 함수, 즉 **kernel (filter)** 을 사용하여 convolve 한다. 이 kernel이 덮는 영역의 값들만 weighted sum 하며 적어나간다. 아래 예시의 경우에는 kernel의 차원, 즉 9개의 parameter만 필요하게 된다.



위의 그림을 보면, image matrix의 가능한 모든 점에 대해 kernel에 대한 weight를 적용해 convolve 한다. kernel의 크기가 클 수록 데이터의 수는 적어질 것이다. 이 때 kernel이 1회 움직일 때 움직이는 픽셀의 수를 **stride** 라고 한다. 만약 5×5 크기의 channel에 3×3 크기의 kernel을 stride 1로 적용한다면 3×3 크기의 결과값이 나올 것이다.



위의 그림에서는 모든 channel에 같은 filter를 사용했지만, 원칙적으로는 각각의 channel (즉, R, G, B channel)에 대해 용도에 따라 다른 filter (kernel) matrix를 사용한다. 그리고 결과로 나온 모

든 matrix를 더한다. 그렇게 되면 3장의 channel matrix 이었던 것이 1장의 matrix로 변한다. 이것이 **Feature Map** 이다. 그런데 크기가 2×2 보다 크거나 같은 filter를 거치면 데이터의 개수가 줄어드는 문제가 있다. 이 때문에 **zero-padding**을 하기도 한다. 즉, 원래 데이터의 둘째에 '0'을 추가하여 결과값의 크기가 원래 데이터의 크기와 같게 만드는 것이다.

Layer Size CNN의 구조를 만들기 위해서는 Layer의 사이즈를 계산할 수 있어야 한다. 예시를 들어보자. $32 \times 32 \times 3$ image가 있다고 하자. 각 channel마다 다른 filter를 사용할 것이며 5×5 크기의 filter를 사용할 것이라면 $5 \times 5 \times 3$ 의 filter가 필요하다. bias를 추가한다고 하면 이 한 번의 작업에 필요한 parameter 개수는 $5 \times 5 \times 3 + 1 = 76$ 개이다. stride=1로 하면 feature map matrix 하나는 28×28 크기를 갖는다. 만약 feature map의 channel을 6개 얻고 싶다면 이 작업을 6개 반복하게 되며, 따라서 $76 \times 6 = 456$ 개의 parameter가 필요하며 결과적으로 $28 \times 28 \times 6$ 크기의 feature map을 얻게 된다.

input의 사이즈가 $W_1 \times H_1 \times D_1$ 이라고 하자. filter의 개수가 K , filter의 한 변의 크기 F , depth D_1 , stride S , zero padding P 라고 하자. 이 때 새로 만들어지는 layer 사이즈 $W_2 \times H_2 \times D_2$ 는 다음과 같다.

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

자명하게도, feature map의 channel 개수는 사용하는 kernel set의 개수만큼 나올 것이며 이것이 D_2 가 된다. 또한 굳이 D 를 사용하는 이유는, 이것이 **Depth**이며 사용되는 kernel set의 개수를 의미하기 때문이다.

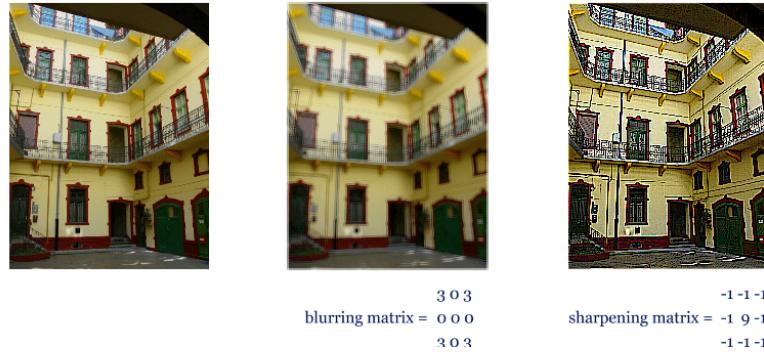
Kernel Size kernel의 크기는 어떤 역할을 하는가? stride 1, zero padding, filter 개수 C 라고 했을 때 7×7 filter와 5×5 filter의 차이는 다음과 같다. (그림에서 5×5 가 아니라 3×3 임.)

Layer 한 개, 7×7 filter	Layer 세 개, 5×5 filter
Weight의 개수	Weight의 개수
$C \times (7 \times 7 \times C) = 49 C^2$	$3 \times C \times (3 \times 3 \times C) = 27 C^2$
계산 복잡도	계산 복잡도
$(H \times W \times C) \times (7 \times 7 \times C)$ $= 49 HWC^2$	$3 \times (H \times W \times C) \times (3 \times 3 \times C)$ $= 27 HWC^2$
parameter 개수 적음 비선형성 높음	

만약 7×7 크기의 channel에서 3×3 크기의 kernel을 3번 사용한다면 모든 곳의 정보를 얻을 수 있다. 반면 7×7 크기의 kernel을 1번 사용하면 모든 곳의 정보를 얻을 수 있다. 비슷한 성능이지만, 3×3 의 계산이 적고 때 layer마다 비선형성을 추가했으므로 비선형성 또한 높다. 따라서 3×3 을 사용하는 것이 좋다고 말할 수 있다.

Feature Extraction 수학적으로 의미를 따져봤을 때 convolution은 '평균으로 분산시키기', 'smoothing 기법' 등의 의미를 갖는다. CNN에서는 이 작용소(operator), 즉 kernel이 가지는 Feature Extraction 기능을 사용한다. 수학에서의 operator는 근본적으로 내적이기 때문에 이것이 가능한 것이다. 정말 그럴까? 다음 예시를 고려해 보자.

```
Convolution matrix over pixels
<filter id="blur">
  <feConvolveMatrix order="3" kernelMatrix="3 0 3 0 0 0 3 0 3"/>
</filter>
...
<image ... filter="#blur"/>
```

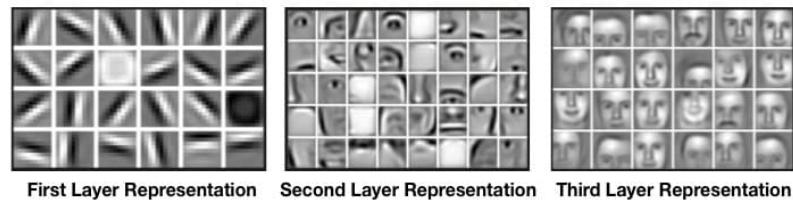


즉, 사진에 특정 kernel matrix를 이용하면 detect하는 모양이 달라진다. 물체의 외곽선만 detect하는 matrix, gaussian matrix 등 kernel의 형태에 따라 특정 feature가 detect 된다. 즉, 사진의 pattern을 찾고 이를 학습하기 위해서 filter를 적절히 조정해 나가게 된다. 몇 가지 예시가 다음과 같다.

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Edge detection을 보면 matrix 합이 0이라는 것을 알 수 있다. 때문에 평균값이 0으로 수렴하며 이는 검은색을 나타낸다. 다른 예시의 경우 matrix 합이 1이며 따라서 원래의 색체를 유지한다는 것을 알 수 있다. 물론 이 예시는 초기값이며 학습을 통해 수정해나가는 것이다.

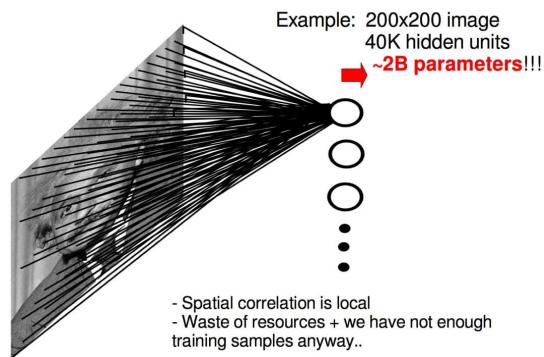
수정된 Convolution 스텝은 각각 해당 스텝에서 적절한 feature를 뽑아낸다. 그 feature를 직접 보면 우리가 알아챌 수도 있지만, 가끔은 이해할 수 없는 것들이 나오고는 한다. 아래의 예시를 보자.



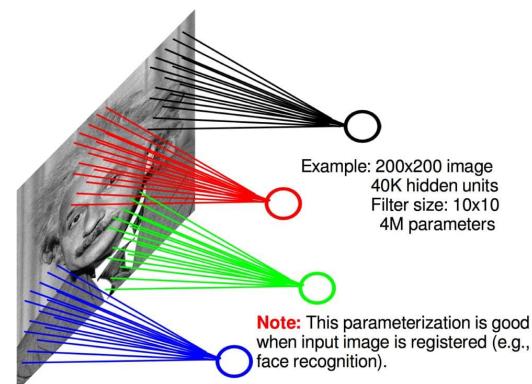
첫 번째 layer는 단순히 작은 직선 모양을 찾아낸다. 그 다음 layer는 사용하는 정보가 더 넓다. 따라서 첫 layer 보다는 약간 디테일한 부분, 예를 들면 눈, 코, 입 등의 모양을 찾아낸다. 마지막은 전체적인 얼굴 모양과 같이 높은 복잡도의 특징을 찾는다.

3.2.3 Why CNN?

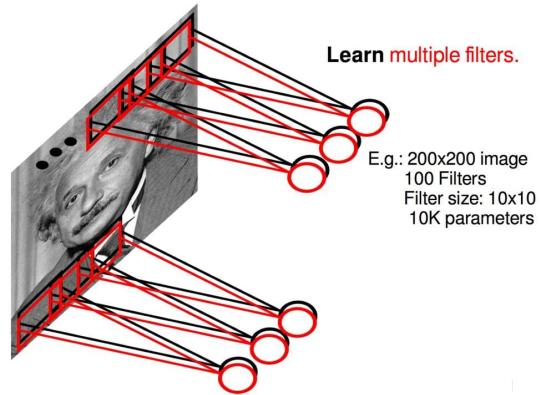
1. fully connected NN 만약 우리가 그림을 다룰 때 CNN을 사용하지 않고 모든 영역에 weight를 같게 하여 fully connected 된 layer를 사용한다면 계산 수가 너무 많을 뿐 아니라 그림의 특징이 잘 뽑히지 않는다.



2. locally connected NN 그림의 section을 적절히 나누어 locally connected된 layer를 사용한다면 위의 단점을 어느 정도는 보완할 수 있다.

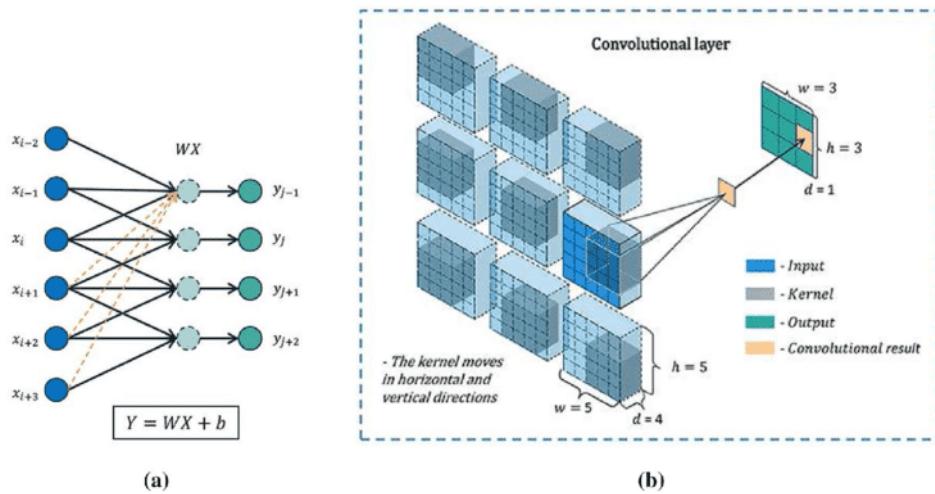


3. CNN 하지만 그림이 registered 되어있을 때에만 좋은 결과를 보여준다. 즉, 그림을 쪼갤기 때문에 전반적인 특징을 잘 잡아내지 못한다. 따라서 weight sharing이라는 방법을 사용한다. 같은 weight를 여러 다른 영역에서 사용하는 것이다.

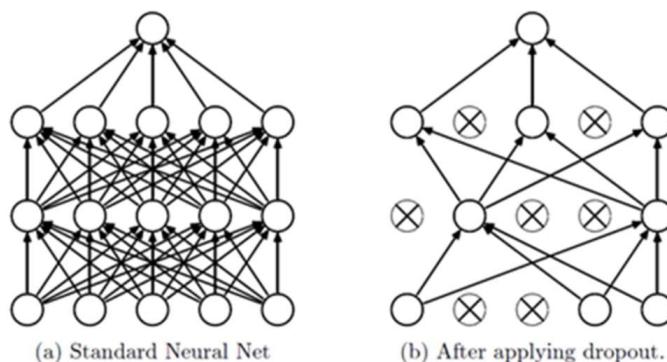


검은색 선은 한번의 weight sharing이고, 빨간색 선은 이 방법을 한 번 더 반복한 것이다. 즉, weight sharing filter를 여러 번 사용하여 더 정확한 결과를 얻는다. 이것이 우리가 살펴 본 CNN이다. 다만 여기까지 봤을 때 network가 너무 많이 연결된 듯 하다. 이때문에 overfitting이 일어나기도 하는데, 몇 가지 방법으로 이를 방지할 수 있다.

1. Local Connectivity : 모든 pixel을 다음 network로 연결하는 대신 일부분만 연결한다. 이는 parameter 개수를 직접적으로 줄여준다.

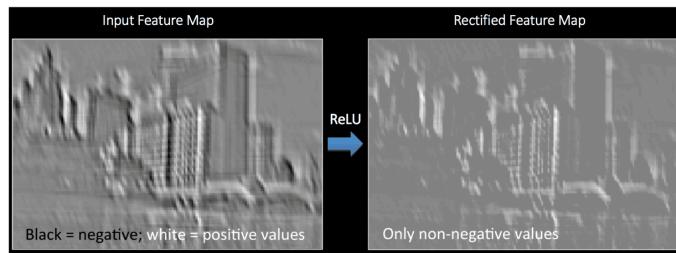


2. Parameter Sharing : 각 채널 안에서 같은 filter를 사용해서 parameter 개수를 줄여준다. 이전에 feature map에서 설명했듯이, 이렇게 해서 feature map들의 stack이 생긴다.
3. Dropout : 특정 비율에 해당하는 랜덤한 뉴런을 찾아 생략한다. 이는 parameter 개수를 줄여주는 효과가 있다.



3.3 Other Things of CNN

Nonlinearity 기존 이미지에 convolution하여 feature map을 얻었다. 해당 과정은 선형이기 때문에, network가 하나의 선형결합으로 귀결되지 않으려면 비선형성을 추가해야 한다. 따라서 얻은 feature map 각각의 pixel에 activation function을 적용하게 된다. ReLU를 가장 많이 사용하는데, 이 경우 음수값을 모두 0으로 만든다는 것을 알고 있다.



즉, kernel은 중요한 정보를 양수로 보내고 중요하지 않은 정보는 음수로 보낸다. 그 후 ReLU를 적용하면 중요하지 않은 정보를 버리고 비선형성을 갖게 된다.

Pooling 각각의 feature map의 차원을 줄이면서도 (마지막에 합쳐주지 않음. 즉, channel 개수는 유지) feature의 중요한 정보는 잃지 않는 방법이다. Max, Avg, Sum 등의 방법이 있다. CNN에서는 보통 Max Pooling을 사용하며, 이에 대한 예시가 다음과 같다.

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

2×2 Max-Pool →

20	30
112	37

Pooling은 parameter가 아니다. 즉, hyperparameter로 최초에 정해줘야 하며 고정된 값이다. Pooling은 역과정이 불가하기 때문에, 데이터를 잃어버린다. 하지만 중요하다고 여기는 정보는 잃지 않으며 연산이 빨라지기 때문에 어떤 일을 하느냐에 따라 상황에 맞춰 사용하면 된다.

kernel 2×2 , stride 2 의 convolution을 사용하면 위 그림과 유사한 과정으로 차원을 줄이는데, 왜 Pooling을 사용할까? 몇 가지 장점을 살펴보자.

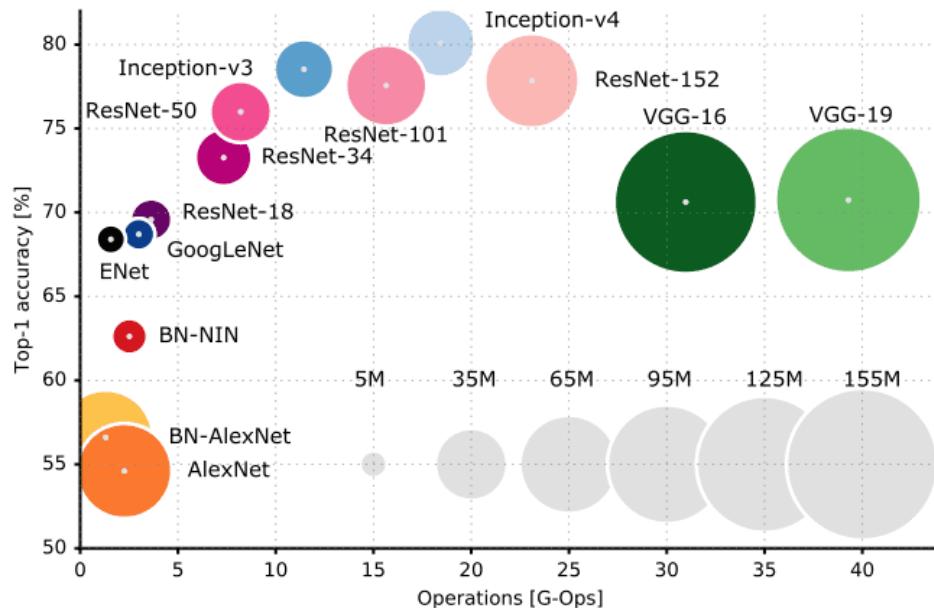
- Pooling은 input의 feature들의 차원을 낮춰줘 다루기 쉽게 한다. 즉, parameter의 개수가 줄고, 계산 부하가 줄며, overfitting을 피할 수 있다.
- 작은 변화의 영향을 덜 받게 할 수 있는데, 다르게 말하자면 noise 혹은 약간의 손떨림으로 인한 사진의 blur 등의 영향을 희석시킬 수 있다. 이를테면 2×2 avg 를 사용할 경우, 1 ~ 2 pixel의 이동 정도는 영향이 아예 없게 된다.
- Scale invariant한 방식으로 image를 표현할 수 있다. image 안의 일부분에 놓인 것도 같은 것으로 인식해서 찾아낸다. 즉, 그림을 이동시키거나 회전시켜도 괜찮다.

FC (Fully Connected layer) 앞서 convolution의 방법으로 그림의 특징을 뽑아내고, activation function으로 비선형성을 추가하고, pooling으로 데이터를 다루기 쉽게 하는 것을 보았다. 하지만 학습을 위해서는 우리가 얻어낸 feature를 기준에 알던, traditional한 machine에 넣을 필요가 있다. 때문에 우리는 convolution, pooling 등이 끝난 후 FC에 연결하게 된다. FC는 softmax를 output layer로 갖는

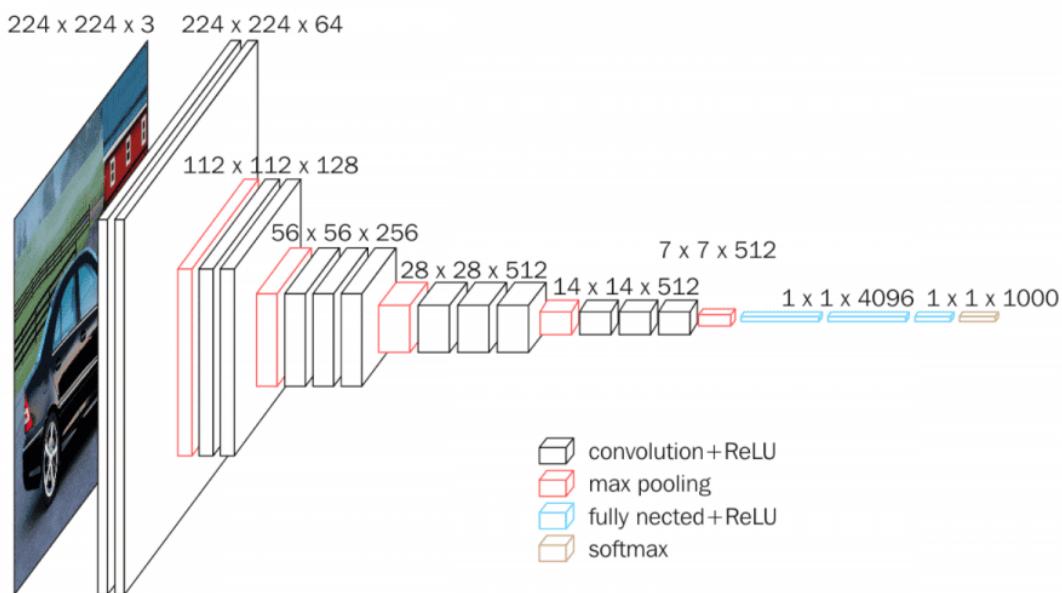
MLP (Multi Layer Perceptron) 으로, 모든 neuron은 자신의 전 단계의 모든 neuron과 연결되어 있다. 어떻게 보면, 앞선 convolution, pooling 등의 layer는 feature를 뽑아내고, FC는 classification의 기능을 한다고 말할 수 있다.

3.4 CNN Architectures

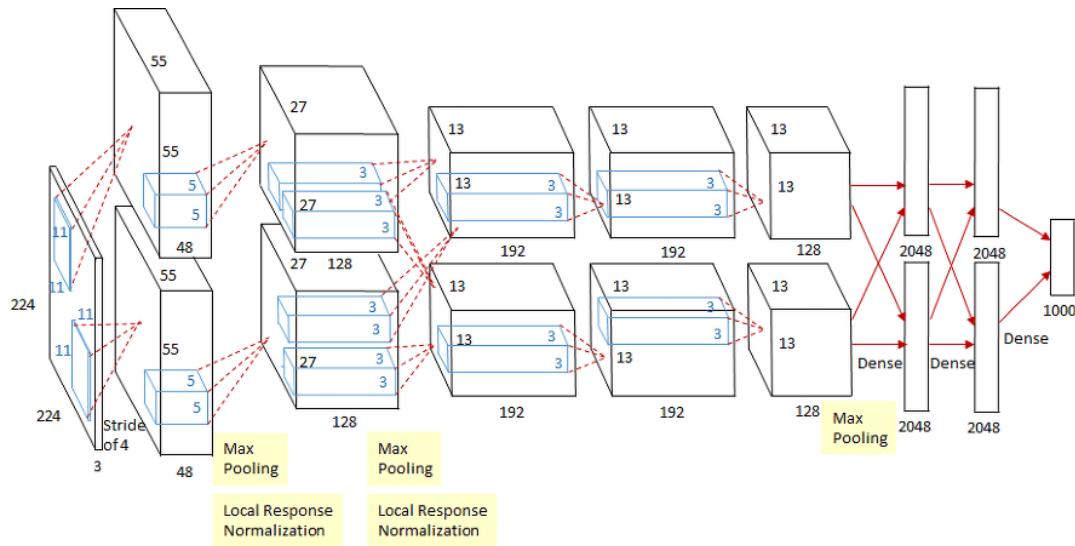
다음은 ILSVRC (Large Scale Visual Recognition Challenge) 의 top accuracy machine 들이다. 원의 크기는 parameter의 개수이다.



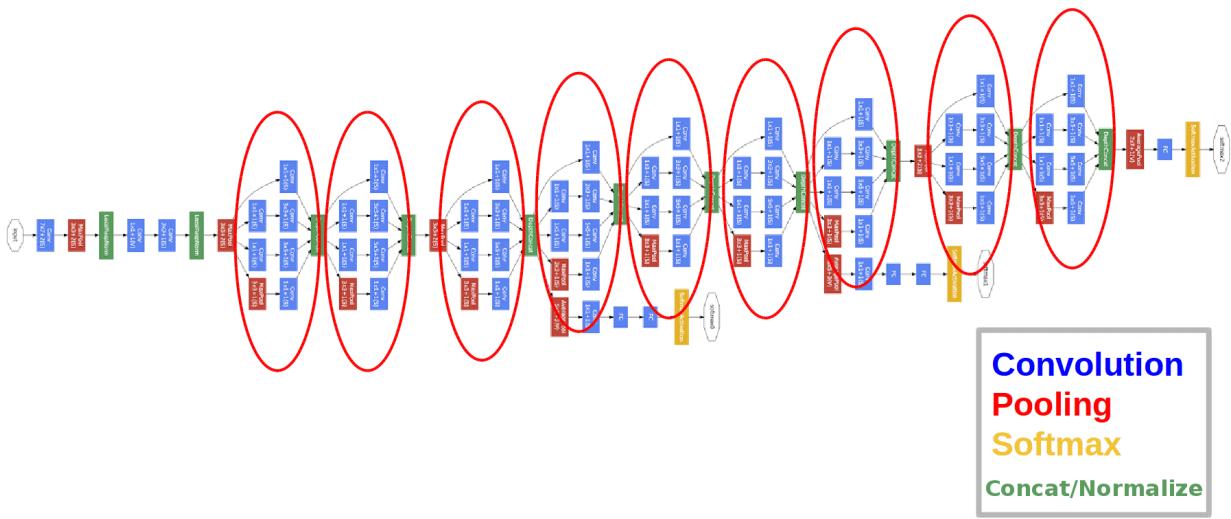
VGG-16 (Simonyan and Zisserman) 이 구조는 매우 간단하고 다루기 편하기 때문에 지금까지도 많이 쓰인다. 특히 VGG-16 모델은 이미 수많은 데이터로 학습이 되어있기 때문에 학습된 모델을 가져와 마지막 softmax만 내가 원하는 것으로 바꿔 그 부분만 학습하면 된다. 그렇게 하면 적은 데이터를 가지고도 내가 원하는 결과를 얻을 수 있다.



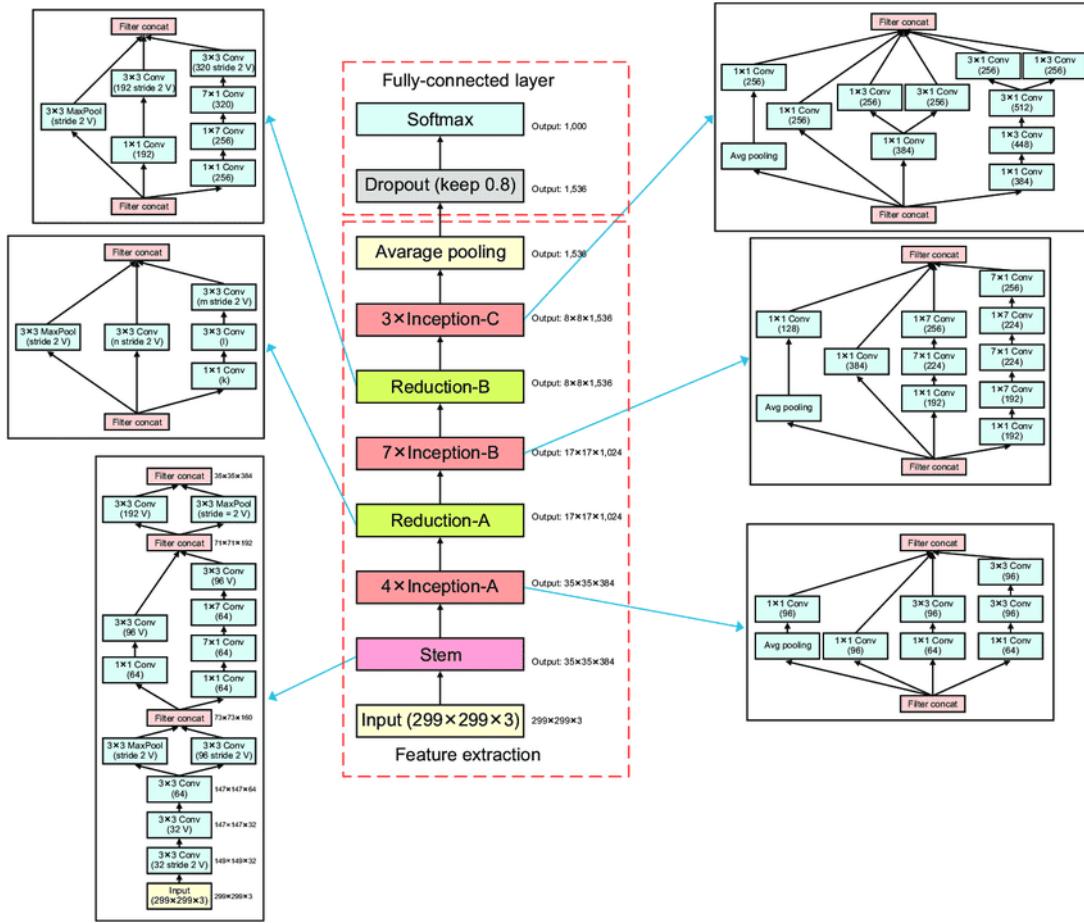
AlexNet (Alex Krizhevsky)



GoogLeNet (Inception v1) 매우 복잡해 보이는데, ILSVRC 표를 보면 알 수 있듯이 parameter 개수가 매우 적다. 이는 FC에 들어가기 전 처리 작업을 잘 해놨기 때문인데, 다음 그림에서 보이는 반복 작업이 그것이다.



Inception v4



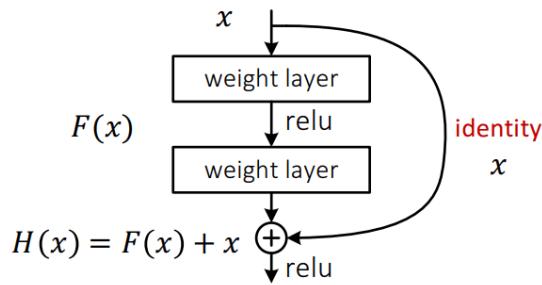
위의 그림을 보면 1×1 conv라는 것이 많이 보인다. 이것을 왜 사용할까? 그리고 identity와는 무엇이 다를까? 1×1 filter는 다음과 같은 역할을 한다.

1. 하나의 filter set은 각 channel마다 다른 값을 곱해주어 channel마다의 중요도를 구분해준다. 즉, 하나의 channel을 한 묶음으로 보아 channel끼리의 FC라고 볼 수 있다.
2. 비선형성을 증가시킨다.
3. input channel과 output channel의 수가 같으면 차원의 변화가 없다. 물론 set의 개수를 조절해 feature map의 channel 개수를 조절하기도 한다.

Inception-B 모듈을 보면 1×7 과 7×1 filter를 따로 적용한 것을 알 수 있다. 왜 7×7 filter를 한번에 적용하지 않았을까? 이는 parameter 개수를 보면 알 수 있는데, 1×7 과 7×1 filter를 따로 적용할 경우 14개의 parameter만 있으면 되지만 7×7 filter를 한번에 적용할 경우 49개의 parameter가 필요하다. 또한 따로 적용하면 비선형성이 증가한다.

Inception-C 모듈을 보면 하나의 input을 1×3 과 3×1 로 나누어 적용한 후 합친 것을 볼 수 있다. 이는 ‘가로’와 ‘세로’를 따로 보면 parameter를 줄이는 장점도 있다.

ResNet 다음의 과정을 152번 쌓은 구조이다.



ResNet은 기본적으로 다음과 같은 특징을 갖는다.

- 간단한 디자인이며, deep 하기만 하다.
- 거의 모두가 3×3 kernel 을 사용한다.
- Batch normalization 을 사용했다.
- pooling(일부 있음), FC, dropout이 없다.
- shortcut mapping, 즉 identity가 한 단계 건너뛰는 작업을 사용한다. 이 때문에 정보가 지속적으로 유지될 수 있다. 이를 **residual** 이라 한다. 후에 계속 사용되는 용어이니 기억해두도록 하자.

매우 간단하고 별로 특이한 방법을 사용하지 않았음에도 불구하고 ILSVRC 기록을 보면 정확도가 매우 높다. 계산과정을 살펴보면, 다음과 같은 특징을 갖는 것을 알 수 있다.

1. Very smooth forward pass

$$\begin{aligned} x_{l+1} &= x_l + F(x_l) \\ &= x_{l-1} + F(x_{l-1}) + F(x_l) \\ &= \dots \\ &= x_n + \sum_{i=n}^{l-1} F(x_i) \end{aligned}$$

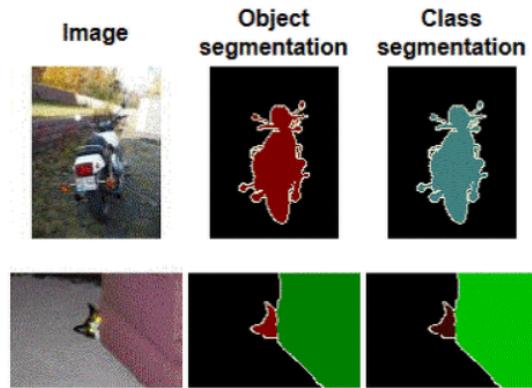
2. Very smooth backpropagation

$$\frac{\partial L}{\partial x_n} = \frac{\partial L}{\partial x_L} \frac{\partial x_L}{\partial x_n} = \frac{\partial L}{\partial x_L} \left(1 + \frac{\partial}{\partial x_n} \sum_{i=n}^{l-1} F(x_i) \right)$$

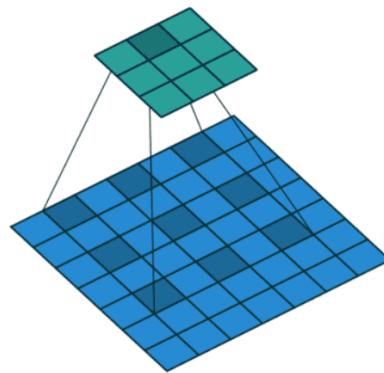
이 때문에 activation을 바꿔도 gradient vanishing이 생기지 않는다.

이 방법이 성공하자 이를 응용하여 Naive residual block, Bottleneck residual block 등이 나오게 되었다.

Dilated Convolution (atrous convolution) segmentation을 위한 방법이다. object segmentation은 이미지에서 '어떤 물체가 있다는 사실'을 픽셀 단위로 예측하는 것이고, class segmentation은 어떤 물체가 '어떤 클래스에 속하는지'까지 예측하는 것이다.



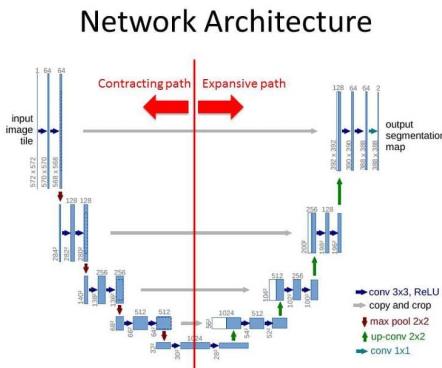
Receptive Field란 kernel이 한번에 얻을 수 있는 정보의 영역이다. Dilated Convolution은 필터 내부에 zero padding을 추가해 강제로 receptive field를 늘리는 방법이다. 다음 그림을 보면 초록색이 filter인데, 파란색 그림(input)에 바로 convolution하지 않고 zero padding을 추가해 진한 파란색 그림으로 만든 후 convolution 한다. filter는 진한 파란색 부분에만 적용되며 나머지 부분은 0으로 곱해진다.



receptive field란 필터가 한 번의 보는 영역으로 볼 수 있는데, 결국 필터를 통해 어떤 사진의 전체적인 특징을 잡아내기 위해서는 receptive field는 높으면 높을수록 좋다. 그렇다고 필터의 크기를 너무 크게하면 연산의 양이 크게 늘어나고, 오버피팅의 우려가 있다.

Dilated Convolution으로 Pooling을 수행하지 않고도 receptive field의 크기를 크게 가져갈 수 있기 때문에 spatial dimension의 손실이 적고, 대부분의 weight가 0이기 때문에 연산의 효율도 좋다. 공간적 특징을 유지하는 특성 때문에 Dilated Convolution은 특히 Segmentation에 많이 사용된다.

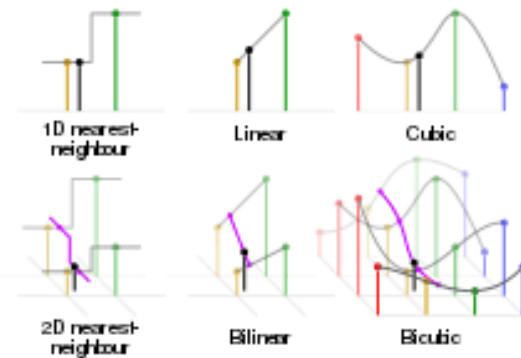
Up Sampling 위에서 Dilated Convolution을 통해 물체가 어디에 있는지 알았다면, 이를 다시 사진의 크기로 표현해야 할 때가 있다. 즉, 사진에서 feature를 얻어내는 과정이 down sampling이고 지금 까지 우리가 다뤘던 것이라면, up sampling은 역과정이다. 물론, convolution 과정에서 정보를 잃었기 때문에 완전한 역과정은 불가능하다. 후에 다루겠지만, Auto Encoder에서 이를 중요하게 사용한다.



이 방법은 크게 두가지로 나눌 수 있다.

1. Non-learnable interpolation methods

- nearest neighbor : 가까운 곳을 기준값으로 채운다.
- bi-linear : 값 사이에 직선을 생성해 그것을 기준으로 채운다.
- bi-cubic : 점 4개를 한 세트로 하여 3차식을 생성해 그것을 기준으로 채운다.



이들은 오래전부터 사용된 manual 적인 방법이다.

2. Learnable NN up-sampling

- Transposed convolution (deep convolution)
- Fractionally-stride convolution

이 둘을 자세하게 다뤄보자.

Transposed Convolution Convolution의 중요한 점은 positional connectivity, 즉 공간 정보가 붙어서 유지된다는 것이다. 또한, convolution layer를 지나면서 차원이 줄고 결국 many to one relationship이라는 것이다. 반면 이 역과정은 one to many relationship이 되어야 한다. 완전한 역과정은 불가능하기 때문에, 이 과정에서 학습이 필수적이다.

Transposed Convolution은 convolution 과정을 나타내는 하나의 matrix를 찾는 것을 우선으로 한다. 즉, 원래는 kernel을 움직여가면서 convolution을 시행했지만, 이 과정을 나타내는 하나의 matrix

를 만들 수 있다. 다음과 같은 3×3 kernel 과 4×4 input, 그에 따른 2×2 output 을 고려하자.

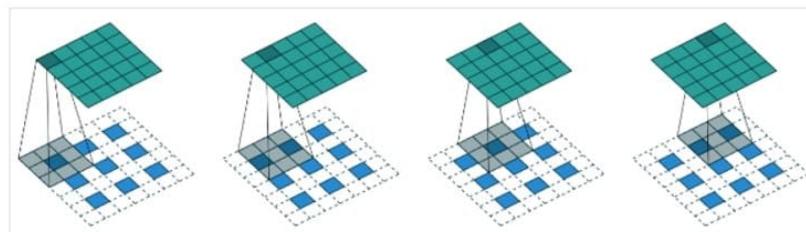
$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} * \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} & x_{16} \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}$$

이를 matrix 연산으로 표현하면 다음과 같이 $MX = Y$ 의 꼴이 된다.

$$\begin{bmatrix} w_1 & w_2 & w_3 & 0 & w_4 & w_5 & w_6 & 0 & w_7 & w_8 & w_9 & 0 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & w_4 & w_5 & w_6 & 0 & w_7 & w_8 & w_9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & w_3 & 0 & w_4 & w_5 & w_6 & 0 & w_7 & w_8 & w_9 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_1 & w_2 & w_3 & 0 & w_4 & w_5 & w_6 & 0 & w_7 & w_8 & w_9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{15} \\ x_{16} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_3 \\ y_4 \end{bmatrix}$$

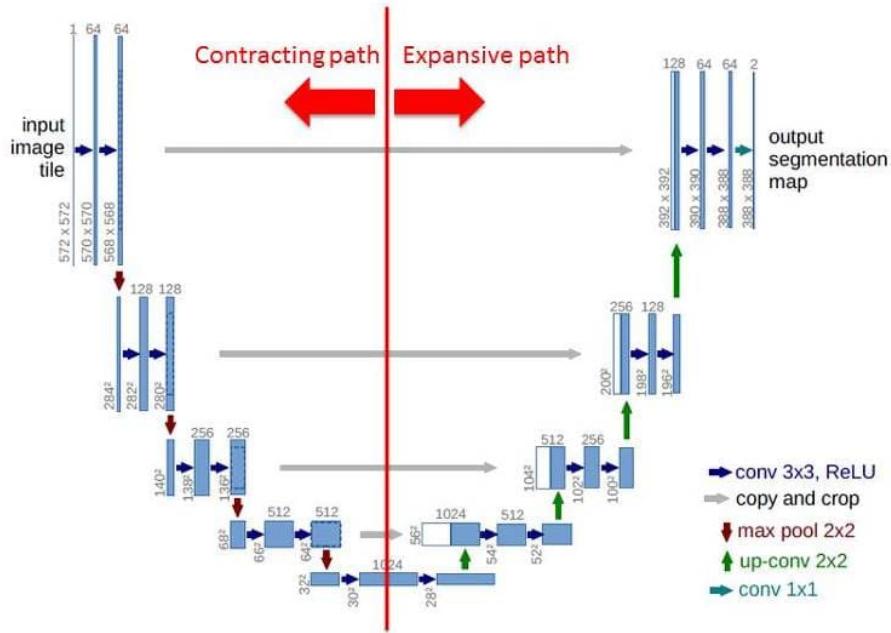
이제 새로운 ‘transposed convolution’ matrix M'^T 를 만든다. 물론 이것이 위에서 구한 matrix M 의 inverse 이면 좋겠지만, 이는 거의 불가능하다. 때문에 적당한, 역행렬로 근사될 수 있는 matrix 로 만드는 것이 학습의 목표가 된다. 즉, 초기값은 위에서 만든 matrix M 을 transpose 한 M^T 으로 하고, 결과값이 convolution input 과 같아지는 방향으로 학습한다.

Fractionally-Strided Convolution 기존에 5×5 의 크기를 갖는 input 에서 3×3 크기의 kernel 를 stride 1 으로 적용하면 3×3 크기의 output이 나온다는 것을 알고 있다. 그렇다면 3×3 에서 5×5 크기를 만들 수 있을까? 다음은 $\frac{1}{2}$ -strided 3×3 fractionally-strided convolution from 3×3 to 5×5 예시이다.



U-Net 은 segmentation을 위해 upsampling을 사용하는 machine이다. 주로 의학분야에서 사용한다.

Network Architecture



4 Lecture 4 : Recurrent Neural Network

4.1 Motivation for using RNN

Why RNN? 이전의 Feed-forward network의 경우에는 다음과 같은 단점이 있다.

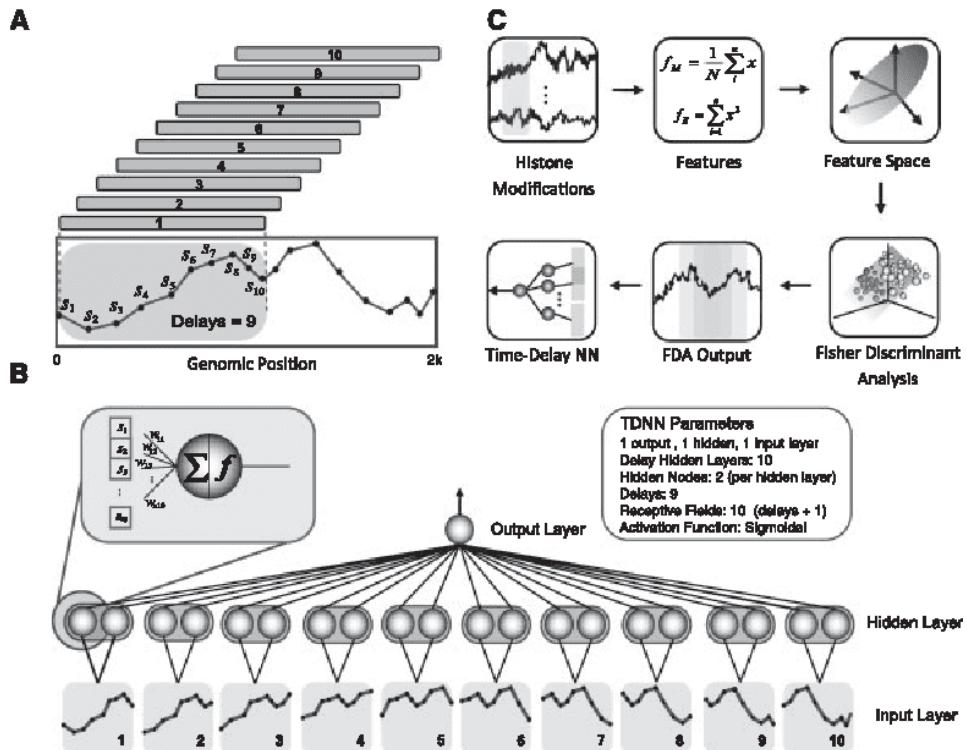
- 정보가 한 방향으로만 흐른다.
- 각 노드에서 하나의 input은 하나의 output만을 생성한다.
- 시간이나 이전 상태에 대한 기억의 개념이 없다.

반면, 이제부터 공부할 Recurrent network는 다음과 같은 특징이 있다.

- 정보가 다양한 방향으로 흐른다.
- 노드가 다른 노드로도 있지만, 자기 자신으로도 간다.
- 시간, 이전 상태에 대한 기억이 있다.

4.1.1 Time Delay Neural Network (TDNN)

일반적인 MLP와 유사하나, input 을 시간에 따라 ($t, t - 1, t - 2$) 등과 같이 넣는 방법이다.

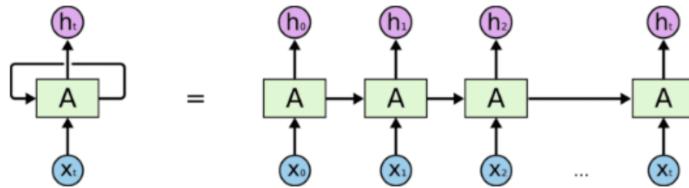


sliding window가 움직이면서 정해진 범위의 시간 영역에 해당하는 데이터만 추려 하나의 데이터로서 machine에 넣는다. 위 그림의 경우에는 10개의 discrete time dependent data를 하나의 벡터로 묶어 input으로 사용하게 된다. 단위시간이 day 이고 주가 예측 machine을 만드는 것이라면 이 machine은 10일간의 주가를 하나의 데이터로 사용하는 것이다. sliding window가 지나면서 데이터가 계속 겹치므로, 과거에 대한 정보가 있다고 할 수 있다. 하지만 이 방법은 그리 성공적이지 못하다.

- window의 크기를 찾는 것이 성공여부를 결정한다. 만약 window가 너무 작다면 메모리 기능이 거의 없을 것이다. 만약 window가 너무 크다면 parameter가 너무 많아지며, 너무 오래전 데이터는 무의미한 노이즈가 되어 섞이게 될 것이다.
- Atari 게임과 같이 - 바로 이전 몇 프레임에서 미사일이 어디 있는지 정도의 데이터만 필요한 적은 메모리를 요하는 곳에서는 작동하지만, 주식 (실제로는 몇 초마다 값이 바뀌므로)과 같이 긴 시간 간격의 데이터를 처리하는 곳에는 작동하지 않는다.
- input data가 시간간격에 따라 균등하게 분포하리라는 보장이 없다. TDNN은 고정된 시간간격을 사용하기 때문에, 이와 같은 변화에 반응하지 못한다.
- 기본적으로 MLP의 구조를 갖는다. 비록 과거의 데이터를 사용한다고 하더라도, 일단 window에서 추출되면 모두 하나의 벡터로 묶여 구분되지 않는다. 즉, window로 자른 벡터들의 순서를 바꿔도 machine에 차이가 없다. 때문에 시간에 대한 개념이 없다.

4.1.2 Recurrent Neural Network (RNN)

feed-forward network는 현재의 input으로 그 다음 결정을 내린다. 반면 RNN은 현재의 input은 물론 과거의 input을 반복적으로 (recurrently) 사용한다. 따라서 이 방법은 sequential data를 다룰 수 있게 된다. 기본적인 RNN의 구조는 다음과 같다.



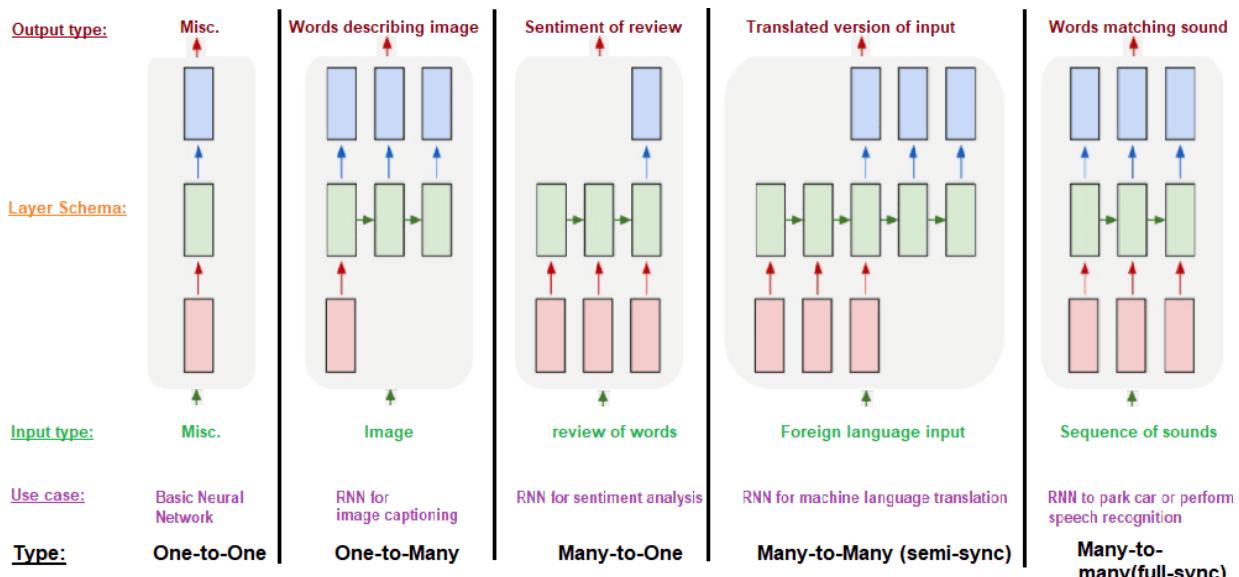
An unrolled recurrent neural network.

RNN은 순차적으로 들어오는 data element를 한번에 하나씩 처리하며 여기서 나온 **hidden state vector**를 유지시켜 마치 메모리를 갖는 것처럼 작동한다. 좌측 그림에서 보이듯이 A에서 나와 다시 A로 들어가는 것이 바로 hidden state vector이다. 이를 시간 순서로 보면 우측 그림이 되는데, 시간 t 에서 생성된 hidden state vector는 그 다음 시간 $t+1$ 로 들어가게 된다.

보통은 sequential data에 대해 TDNN를 시도하고, 잘 되지 않으면 RNN의 발전된 형태인 LSTM을 사용한다.

RNN은 문자, 단어, 구, 문장, 문단 등을 모델로 하는 ‘자연어 모델’에 많이 쓰인다. 이를테면 모바일에서 입력 단어 예측, 키워드 바탕으로 신문 기사 작성 등에 사용된다. 또한 안전관리 및 군사적 목적으로 영상에서 사물을 추적하거나, 주가 예측, 환율 예측, 번역, 음성 인식, 대화, 건강상태 인지 등의 방대한 응용 분야가 있다. 이들을 살펴보면 시간에 따라 지속적으로 들어오는 데이터에 대한 학습이라는 것을 알 수 있다.

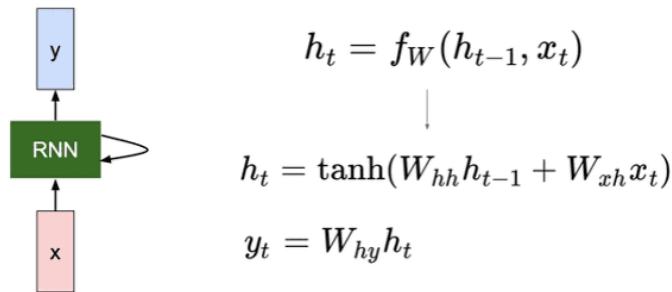
다음은 RNN의 대표적인 종류들이다.



이에 대한 예시들이 다음과 같다.

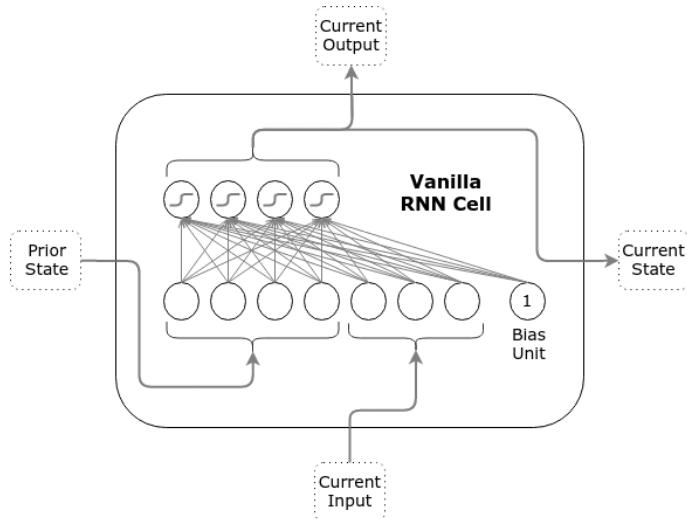
- Many to Many - Sequence Learning : 주가 예측, 상품 추천, 단어 예측 등
- Many to One - Sequence Classification : DNA 열 분류 (DNA Sequence가 ACGT 열로 주어졌을 때 coding region인가 아닌가 판별), 이상 검출 (data의 특정 열이 정상인가 아닌가), 감정판별 등

Simple RNN (Vanilla RNN) 가장 기본적인 RNN이다. 핵심은 hidden unit 안에 들어 있는 state vector가 과거의 정보를 간직하고 있다는 것이다. Weight 행렬은 모든 시간 스텝에서 같은 값으로 학습된다.

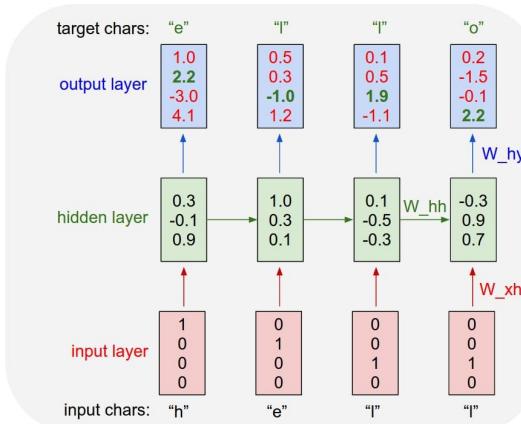


우선 시각 t 에 machine으로 input x_t 가 들어온다. W_{hh} 는 machine의 이전상태인 h_{t-1} 에 대한 weight이고, W_{xh} 는 input x_t 에 대한 weight이며, 이들을 선형결합한 후 activation function에 적용해 시각 t 에서 machine의 상태인 state vector h_t 를 얻는다. 또한 machine은 시각 t 에서 output $y_t = W_{hy}h_t$ 를 출력한다. 이제 machine의 시각 t 에서의 상태 h_t 를 알고 있으니, 그 다음 시각 $t+1$ 에서 새로운 input x_{t+1} 이 들어오면 이 과정이 반복된다. 우리의 목표는 예측값 y_t 를 잘 맞추는 Weight 행렬을 찾는 것이다.

다음 그림은 hidden state 크기가 4, input 크기가 3인 simple RNN을 알아보기 쉽게 그린 것이다.



Language Model 구글 검색창에 ‘hell’라고만 입력해도 그 다음 문자인 ‘o’가 나온다. 이렇게 이전에 입력된 단어로 다음 단어를 예측하는 모델을 예시로 다음 그림을 살펴보자.



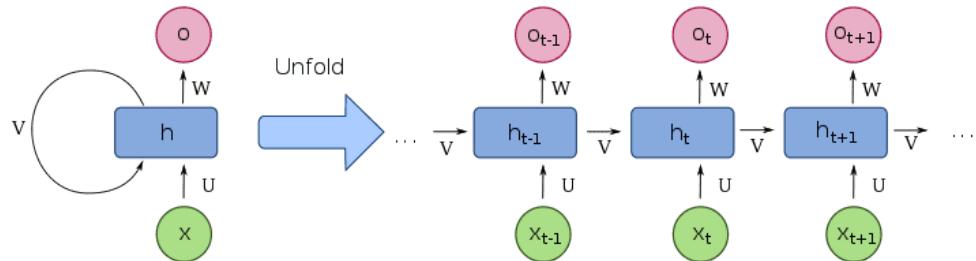
처음 시각에 들어간 ‘h’에 대해 target은 ‘e’가 되고, 이 old state는 다음 input으로 들어간다. 최종적으로 이 모델은 ‘o’를 출력하도록 만들기를 원하는 것이다. 문장 번역의 경우 하나의 time step으로 한

문장을 사용할 수도 있고, 단어마다 잘라서 하나의 time step으로 사용할 수도 있다. 이제 이를 어떻게 학습하는지 알아보자.

4.2 Backpropagation Through Time (BPTT)

이전에는 단순히 Backpropagation의 방법으로 NN의 parameter들을 최적화 해왔다. 하지만 RNN은 sequential input에 맞게 설계되었기 때문에 BPTT의 방법을 사용해야 한다. 우선 우리가 가진 parameter들은 다음과 같다:

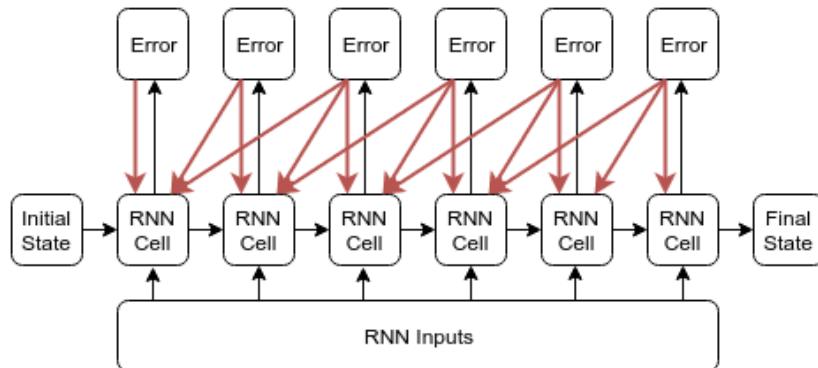
- $U = W_{xh} : x \rightarrow h$
- $V = W_{hh} : h \rightarrow h$
- $W = W_{hy} : h \rightarrow y$



unfold 한 구조에서 weight share가 되지 않는 forward pass를 있다고 생각하고, 모든 cell의 matrices를 다른 것으로 본다. 그리고 output은 시각 t 의 것 하나만 본다. 즉, $U_0, V_0, U_1, V_1, \dots, U_{t-1}, V_{t-1}, U_t, V_t, W$ 의 wieght에 대한 forward pass를 한다. 그리고 모든 weight들에 대한 gradient를 계산해서 더하거나, 평균을 내어 원래 matrix를 업데이트 한다:

$$\nabla U = \sum_{i=0}^t \nabla U_i$$

$$\nabla V = \sum_{i=0}^t \nabla V_i$$

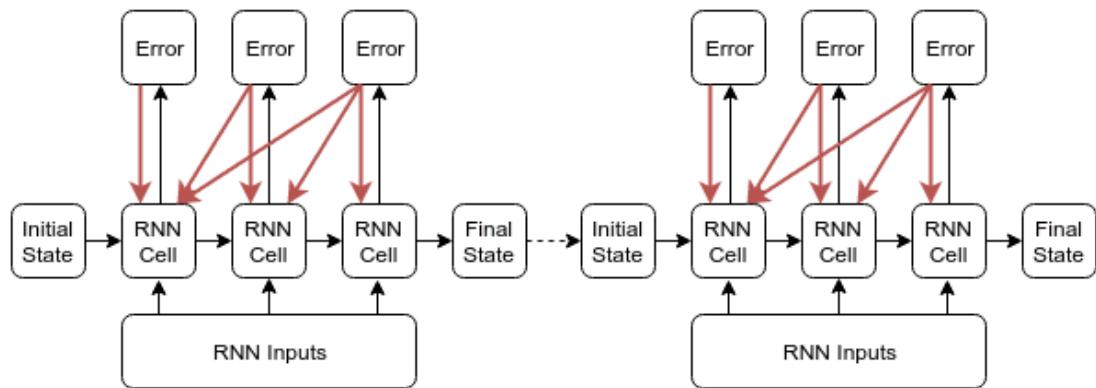


만약 input sequence가 길어진다면, 뒤로 갈수록 학습이 느려질 것이다. 또한 gradient vanishing 문제도 발생하게 된다. 때문에 적절한 시간 길이를 잘라서 해당 부분만 학습하는 방법이 개발되게 된다.

Truncated BPTT (TBPTT) sequence의 문맥, 흐름을 파악하기 위해 충분히 길고, 학습하기에 적절하도록 충분히 짧은 시간 간격을 정해야 한다. 해당 방법은 backpropagation을 하기 위해 gradient를 직접 계산하기 보다는 대략 estimate해서 적용한다. TBPTT에서는 다음과 같은 hyper parameter를 정해줘야 한다:

- **Lookback** : backpropagation 어림을 하기 위해 몇 시간 간격을 볼 것 인지.

우선 lookback 만큼 network를 unfold 한다. 각 cell마다의 weight는 모두 같을 것이다. 이제 여기서 BPTT를 시행한다. parameter의 개수는 그대로이고, 계산량은 lookback만큼 늘어난다. TDNN의 경우에는 window 크기만큼 parameter가 늘어났지만, 여기서는 parameter가 늘어나지 않는다는 것이 핵심이다.



위의 그림은 3 시간 간격만큼 자르는 TBPTT를 시행한다. 3 시간 간격만큼 자른 후에는 해당 구간에서 BPTT를 시행한다. 보통 lookback은 8에서 200 정도의 값을 사용하는데, 일단 lookback 을 늘려 최적의 성능을 얻으면 그 이상으로 늘려도 좋은 성능을 얻을 수 없다.

- **Lookahead** : forward pass를 시행할 때 몇 시간 간격을 볼 것 인지

State Maintenance 아직 연구가 많이 진행되지 않아 자료가 없다. 적절히 사용되면 가능성이 있을 듯 한 주제이다. Keras에서는 RNN의 state를 유지하기 위한 두 가지 방법을 제공한다.

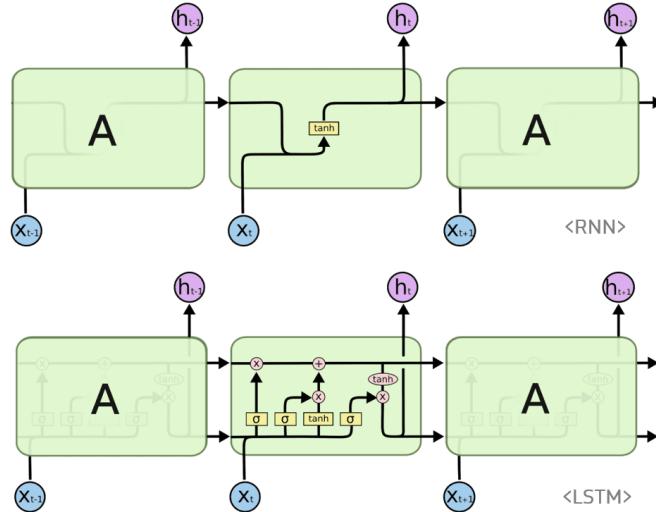
1. Default mode : 각 input sequence를 실행할 때 lookback 시간만큼 state를 유지한다.
2. Stateful mode : 하나의 batch에서 i 번째 데이터를 이용해 출력된 final state가 다음 batch의 i 번째 데이터에 해당하는 machine의 initial state로 들어간다. 이를 위해서는 batch마다 데이터 수와 특정 순서가 모두 같아야 한다.

이를테면 lookahead를 10, batch size 10 으로 학습한다고 가정하자. 이렇게 되면 한번의 batch에 input이 100개 들어갈 것이다. 그러면 첫번째 batch의 첫 번째 sequence의 input hidden state vector는 random initialize 해야 한다. 그러면 output hidden state vector가 나오는데, 이를 두번째 batch의 첫 번째 sequence input hidden state vector로 사용한다. 이렇게 되면 batch를 작게 잘랐는데도 불구하고 더 먼 과거의 정보도 알 수 있다.

제약조건이 많기 때문에, 정확하게 dimension을 계산한 후 적용해야 한다.

4.3 Long Short-Term Memory (LSTM)

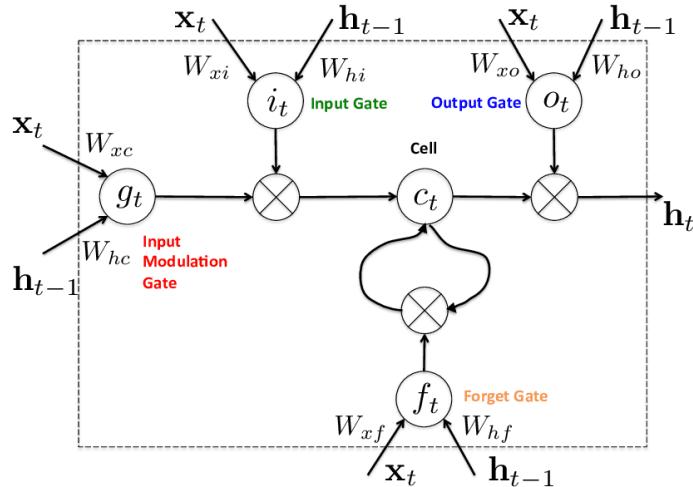
Long-term Dependency RNN을 사용할 때, 만약 서로 관련있는 데이터와 결과값이 너무 멀리 떨어져있다면, 학습이 제대로 진행되지 않을 것이다. 이를테면 ”누가 나오면 박수 치세요” 라고 말한 뒤 1분이 지나 박수소리가 들렸다면, 학습이 되지 않을 것이다. 이를 해결하기 위해 개발된 것이 LSTM이다. 또한, LSTM은 simple RNN의 vanishing/exploding gradient 문제도 해결한다.



4.3.1 LSTM

sparse DRNN의 특수한 케이스로 현재까지 나온 시간 모델 중 가장 성능이 좋고 많이 쓰이고 있다. 기본적인 RNN 구조에 input, output, forget gate라는 개념을 도입하고 이를 학습시킨다. 즉 어떤 RNN 유닛은 short term만 학습하고, 다른 RNN 유닛은 오직 long term만 학습한다.

LSTM은 RNN에서 input, output, forget gate가 추가되어 long term dependency를 잘 학습할 수 있는 모델이다. 이때 RNN에는 activation function이 없다. 그래서 recursive하게 Time Through Back Propagation 을 하여도 vanishing gradient 또는 exploding gradient가 일어나지 않는다.



$$\begin{aligned}
 i_t &= \sigma(W_{xi}^i x_t + W_{hi}^i h_{t-1} + b_h^i) && \text{input gate} \\
 f_t &= \sigma(W_{xf}^f x_t + W_{hf}^f h_{t-1} + b_h^f) && \text{forget gate} \\
 o_t &= \sigma(W_{xo}^o x_t + W_{ho}^o h_{t-1} + b_h^o) && \text{output gate} \\
 g_t &= \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1} + b_h^g) \\
 c_t &= f_t \circ c_{t-1} + i_t \circ g_t \\
 h_t &= o_t \circ \tanh(c_t)
 \end{aligned}$$

where \circ is an elementwise product. 이를 정리하면 다음과 같다.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

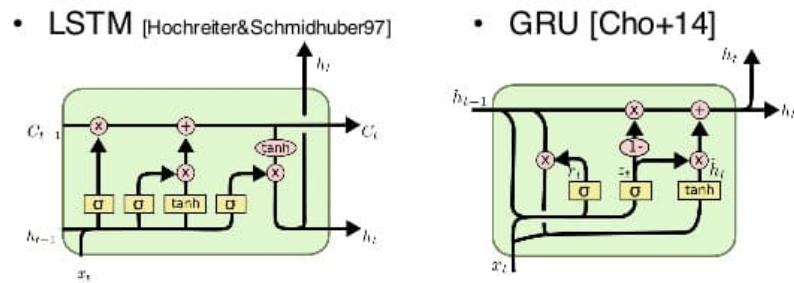
input을 강하게 하려면 sigmoid를 identity로 하고, 약하게 하려면 0으로 한다. forget, output도 마찬가지이다. g 는 RNN의 state vector를 만드는 과정과 유사하다. 이를테면 언어학습을 위해 문장이 올 때, "주어가 나왔음"과 같은 것은 cell state의 담당이고, "He"의 의미가 나왔다는 것은 hidden state의 담당이다. cell state를 계산하는 과정을 보면 이해가 쉽다. 그전단계의 cell state c_{t-1} 을 가지고갈지 말지는 f_t 가 결정한다. 그리고 RNN의 hidden state 역할인 g_t 를 cell state 안에 받아들일 것인지 말지는 i_t 가 결정한다.

LSTM은 주로 덧셈의 구조로 이루어져있기 때문에 gradient vanishing/exploding 문제가 발생하지 않는다.

4.3.2 Gated Recurrent Unit (GRU)

LSTM의 cell을 단순화시킨 방법이다. cell state와 hidden state를 하나로 묶어 hidden state로 사용했고, forget gate와 input gate를 하나의 update gate z_t 로 묶은 모습이다. 또한 reset gate r_t 를 통해 forget 기능을 사용했다.

LSTM and GRU



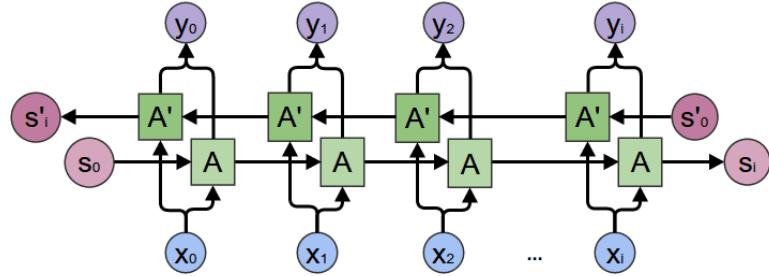
$$\begin{aligned}
 f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
 i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
 \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
 C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
 o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
 h_t &= o_t * \tanh(C_t)
 \end{aligned}
 \quad
 \begin{aligned}
 z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
 r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
 \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\
 h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
 \end{aligned}$$

Tohoku University, Inui and Okazaki Lab. (Biases are omitted.)
Sosuke Kobayashi

4.3.3 Types of LSTMs

다음은 LSTM의 다양한 변형들이다.

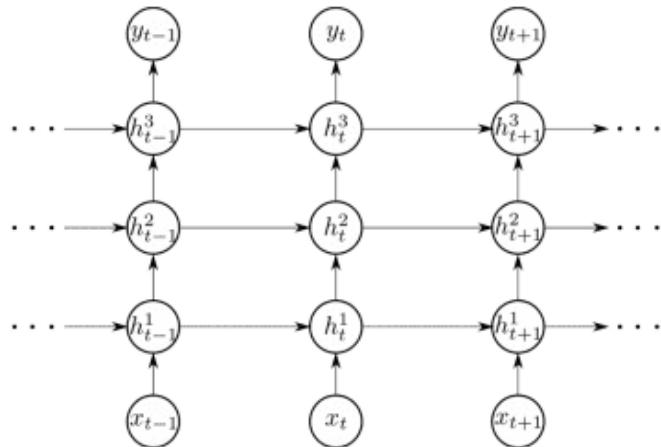
1. Vanilla LSTM : LSTM을 매우 단순화한 구조로, memory cell 부분만 사용한다.
2. Bidirectional LSTM : input sequence를 앞으로, 뒤로 넣어 학습한다.



따로 두 번 하는 것이 아니라, 앞, 뒤로 넣어진 sequential data는 다음과 같은 output을 만든다.

$$y_t = \sigma(W'h'_t + Wh_t)$$

3. Stacked LSTM : 하나의 LSTM layer 위에 다른 LSTM layer를 쌓은 구조이다. 즉, input을 받은 layer의 output을 input으로 하는 LSTM을 하나 쌓았다. 이 LSTM은 MLP로 들어간다.



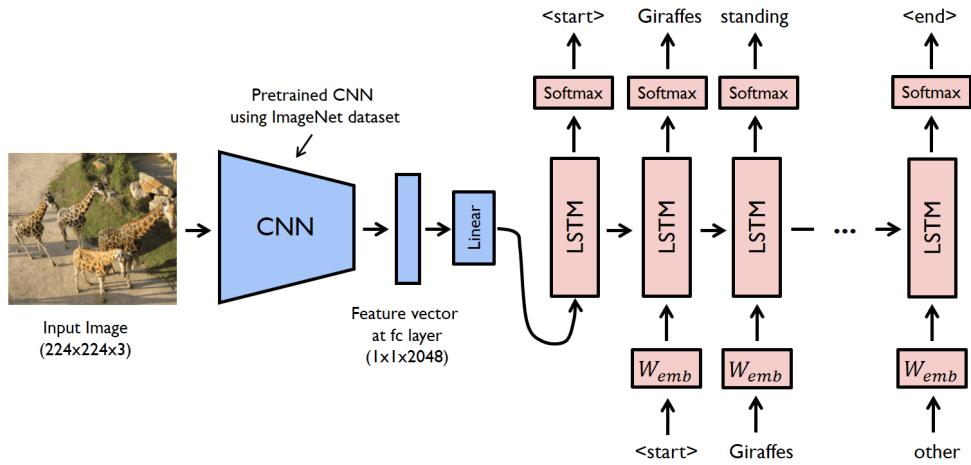
첫번째 있는 LSTM은 전에 언급했듯이 다음의 구조를 갖는다.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

가운데에 있는 LSTM은 다음과 같은 구조를 갖는다.

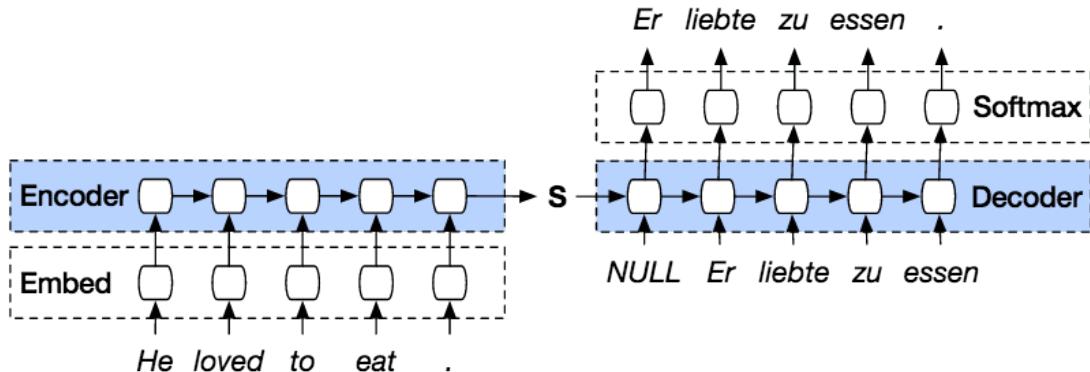
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W' \begin{pmatrix} h'_{t-1} \\ h_t \end{pmatrix}$$

4. CNN-LSTM : 사진의 맵락을 파악하여 caption을 넣기 위한 방법이다. 미리 training 해둔 CNN machine (이를테면 VGG-16)에서 FC를 제거하고 그 자리에 LSTM을 넣는다. Convolution 과정에서 최종적으로 flatten 하여 linear한 feature vector를 얻고, 이를 최초 LSTM의 initial input으로 한다. 그 다음부터는 이전에 Language Model에서 살펴본 것처럼 진행한다.



5. Encoder-Decoder LSTM : 하나의 LSTM은 encoding을 하고, 다른 하나의 LSTM은 decoding 한다.
6. Generative LSTM : input sequence에서 구조를 익히고 이와 비슷한, 새로운 sequence를 생성한다.

Seq2Seq Sequential data를 받아 Sequential output을 내보내는 RNN 이다. 다음은 번역을 위한 machine 이다. 후에 자세히 다룬다.

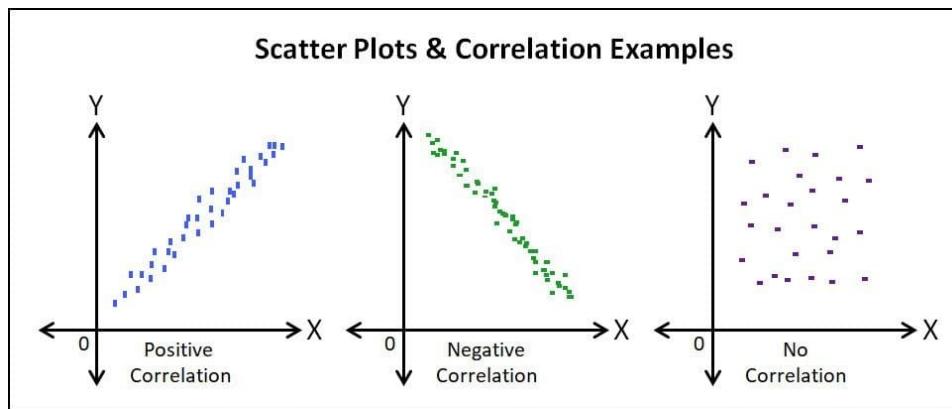


5 Lecture 5 : Variational Auto-Encoder

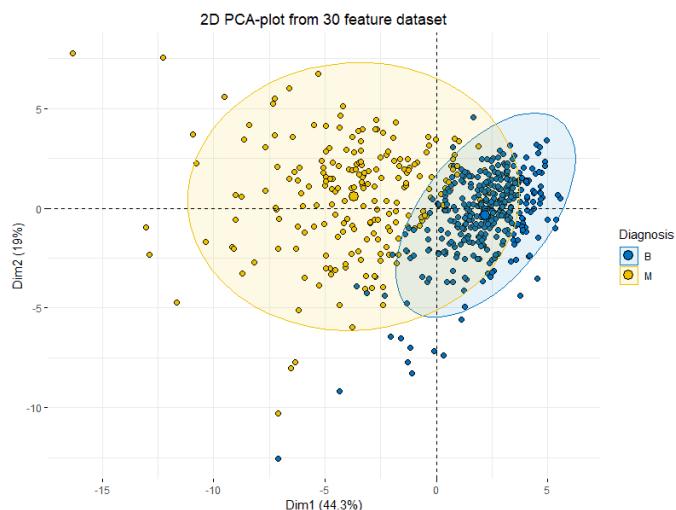
5.1 PCA

5.1.1 Feature Selection

어떤 feature가 유의미한지 파악하는 데에는 몇 가지 방법이 있다. 우선 어떤 feature 두개가 주어졌을 때, 둘의 correlation (두 벡터의 내적)을 계산하여 상관관계를 알아낼 수 있다. correlation이 커서 어떤 경향성 (즉, 한 variable이 커지면 다른 하나가 커지거나 작아지는 것)이 나타나는 것을 확인할 수 있으면 feature을 살린다. 이 방법을 **Feature selection**이라 한다.



다음으로는 feature의 축을 새로 정하는 방법이 있는데, 이를 **Principle Component Analysis (PCA)**라 한다. 서로 연관 가능성이 있는 고차원 공간의 표본들을 선형 연관성이 없는 저차원 공간(주성분)의 표본으로 변환하기 위해 직교 변환을 사용하는 방법이다. 우선 기초적인 수학을 다뤄보자.



Eigenvalues 다음과 같이 일반적인 이차식을 보자.

$$f(\mathbf{x}) = ax^2 + 2bxy + cy^2 = \mathbf{x}^T A \mathbf{x}$$

단위 원 (즉, $\mathbf{x}^T \mathbf{x} = 1$) 위에서 $f(\mathbf{x})$ 의 최대 최소값은 얼마일까? 간단히 Lagrange multiplier 를 이용하자.

$$\nabla f(\mathbf{x}) = \nabla \mathbf{x}^T A \mathbf{x} = 2A\mathbf{x}$$

따라서 방정식으로 쓰면 다음과 같다.

$$A\mathbf{x} - \lambda\mathbf{x} = 0$$

$$\mathbf{x}^T \mathbf{x} - 1 = 0$$

이 때 λ 는 A 의 고유값이고 방정식에 해당하는 $\mathbf{x} = \mathbf{v}$ 는 고유벡터이다. 이 고유값과 고유벡터를

이용하면 다음과 같이 spectral decomposition 된다.

$$A[\mathbf{v}_1, \dots, \mathbf{v}_n] = [\lambda_1 \mathbf{v}_1, \dots, \lambda_n \mathbf{v}_n] = [\mathbf{v}_1, \dots, \mathbf{v}_n] \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} = VD$$

$$\mathbf{x} = y_1 \mathbf{v}_1 + \dots + y_n \mathbf{v}_n = V\mathbf{y}$$

$$\Rightarrow \mathbf{x}^T A \mathbf{x} = \mathbf{y}^T (V^T V) D (V^T V) \mathbf{y} = \mathbf{y}^T D \mathbf{y}$$

Singular Values 다음과 같은 행렬의 사상을 생각해 보자.

$$\mathbf{y} = Ax = \begin{pmatrix} 0 & -2 \\ 3 & 0 \end{pmatrix} \mathbf{x}$$

이 때 $\mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ 혹은 $\mathbf{x} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ 을 생각해 보면, 여기서의 행렬 A 는 x 축 방향은 3 배 늘려서 y 축에 가져다 놓고, y 축 방향은 2 배 늘려서 -x 축에 가져다 놓은 것으로 볼 수 있다. 즉 선형사상 A 는 $\mathbf{x} \in \mathbb{R}^n$ 에서 $A\mathbf{x} \in \mathbb{R}^m$ 으로 사상한다. 우리는 $|\mathbf{x}| = 1$ 인 벡터들 중 $|A\mathbf{x}|$ 의 최대와 그 때의 \mathbf{x} 에 관심이 있다. 즉, $\mathbf{x}^T \mathbf{x} = 1$ 일 때 다음을 찾는다.

$$|A\mathbf{x}|^2 = \mathbf{x}^T A^T A \mathbf{x}$$

이는 eigenvalue 문제와 같지만, 우리가 고려하는 $A^T A$ 는 대칭행렬이기 때문에 아주 쉬운 문제이다. 위에서 사용한 notation을 그대로 사용하면 다음과 같다.

$$A^T A \mathbf{v}_i = \lambda \mathbf{v}_i$$

$$\Rightarrow |A\mathbf{v}_i|^2 = \lambda_i$$

이 때 $|A\mathbf{v}_i| = \sigma_i$ 라고 두고, $A\mathbf{v}_i$ 를 normalize하여 $\mathbf{u}_i = A\mathbf{v}_i / \sigma_i$ 라고 하면 \mathbf{u}_i 는 orthonormal한 basis이다. 위의 eigen equation을 다시 쓰면 다음과 같다.

$$A^T A \mathbf{v}_i = \lambda \mathbf{v}_i$$

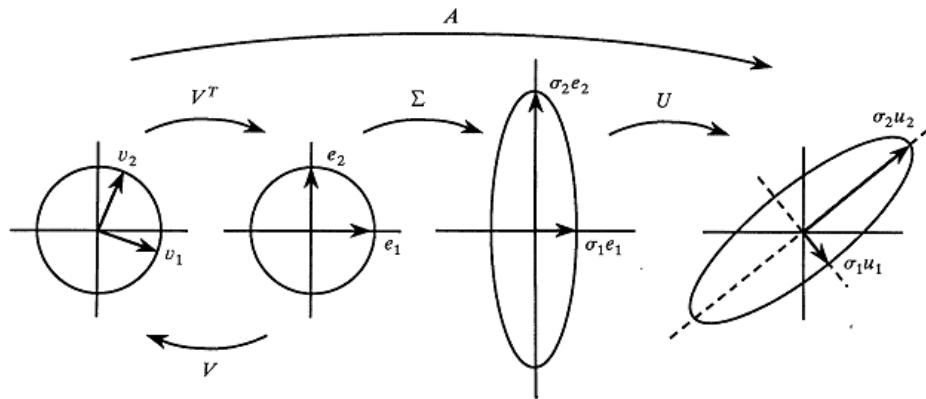
$$\Rightarrow A^T (\sigma_i \mathbf{u}_i) = \sigma_i^2 \mathbf{v}_i$$

$$\Rightarrow A^T \mathbf{u}_i = \sigma_i \mathbf{v}_i$$

$$\Rightarrow \sigma_i \mathbf{u}_i = A \mathbf{v}_i$$

이 관계식을 정리하면 다음과 같이 singular value decomposition (SVD) 된다. (or, compact decomposition)

$$A = U \Sigma V^T = [\mathbf{u}_1, \dots, \mathbf{u}_n] \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_n^T \end{bmatrix}$$



이제 A 의 inverse를 구해야 한다. U 와 V 는 orthonormal basis 들이므로 Σ 만 inverse로 적절히 변형하면 된다. 이는 다음과 같다. 물론 이는 pseudo-inverse 이다.

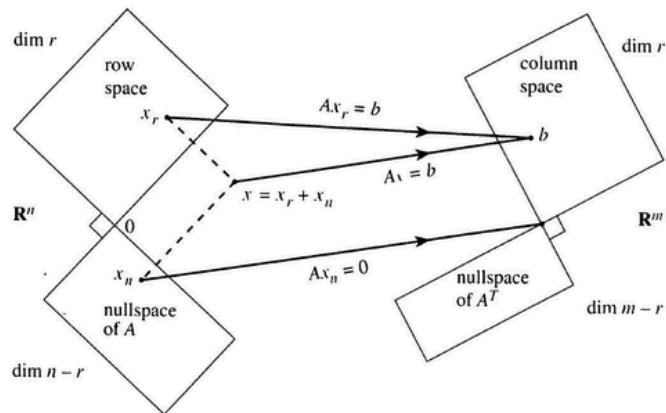
$$A^+ = V\Sigma^{-1}U^T = [v_1, \dots, v_n] \begin{bmatrix} 1/\sigma_1 & & \\ & \ddots & \\ & & 1/\sigma_n \end{bmatrix} \begin{bmatrix} u_1^T \\ \vdots \\ u_n^T \end{bmatrix}$$

5.1.2 PCA

다음과 같은 행렬 사상을 생각해 보자.

$$Ax = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

행렬 A 는 벡터 x 의 a 좌표만을 반환하며, b 는 남지 않는다. a 를 가로축, b 를 세로축으로 둔다면 행렬 A 에 의한 사상은 가로축으로의 사영이다. 만약 $a = 0$ 이라면 b 값이 어떻던 간에 결과같은 $\vec{0}$ 이 되며, 이런 벡터를 ‘행렬 A 의 kernel’ 혹은 ‘null space’라고 한다. 즉, 우리가 고려한 세로축 위의 모든 벡터들이 ‘null space’ 인 것이다. 반대로 가로축은 행렬 A 의 ‘row space’ 이다. 즉, 벡터가 자기 자신으로 사상되는 집합이다. 이를 일반화하여 나타내면 다음과 같다.



PCA 다음과 같이 5개의 데이터가 주어졌다고 하자.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \dots, \begin{bmatrix} x_5 \\ y_5 \end{bmatrix}$$

이 데이터들을 행렬 A' 으로 모아보자.

$$A' = \begin{bmatrix} x_1 & x_2 & \dots & x_5 \\ y_1 & y_2 & \dots & y_5 \end{bmatrix}$$

이 데이터들을 지나는 이차곡선을 찾고, 그 이차곡선에 대한 중심점과 그 축 (principal axis)을 찾고자 한다. 우선은 데이터들의 중심점을 찾아야 한다. 이는 각 데이터에 평균값을 뺀으로서 얻을 수 있다. 즉, 고려하고자 하는 행렬 A 는 다음과 같다.

$$A = \begin{bmatrix} x_1 - \bar{x} & x_2 - \bar{x} & \dots & x_5 - \bar{x} \\ y_1 - \bar{y} & y_2 - \bar{y} & \dots & y_5 - \bar{y} \end{bmatrix}$$

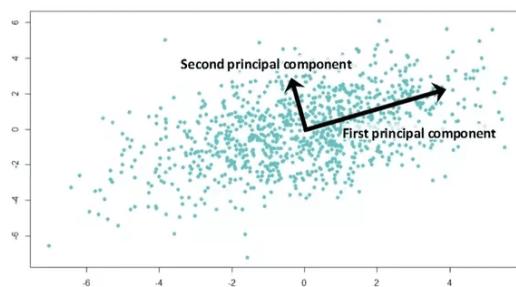
이제 행렬 A 에 의한 사상을 생각해 보자.

$$A\mathbf{a} = \begin{bmatrix} x_1 - \bar{x} & x_2 - \bar{x} & \dots & x_5 - \bar{x} \\ y_1 - \bar{y} & y_2 - \bar{y} & \dots & y_5 - \bar{y} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_5 \end{bmatrix}$$

우리는 이 값이 최대가 되는 점들을 찾고 싶다. 즉, 이차곡선의 모양이 어떻게 되는지 알고 싶다. 이를 위해 위의 사상을 제곱해보자.

$$\begin{aligned} \mathbf{a}^T A^T A \mathbf{a} &= [a_1 \ a_2 \ \dots \ a_5] \begin{bmatrix} x_1 - \bar{x} & y_1 - \bar{y} \\ x_2 - \bar{x} & y_2 - \bar{y} \\ \vdots & \vdots \\ x_5 - \bar{x} & y_5 - \bar{y} \end{bmatrix} \begin{bmatrix} x_1 - \bar{x} & x_2 - \bar{x} & \dots & x_5 - \bar{x} \\ y_1 - \bar{y} & y_2 - \bar{y} & \dots & y_5 - \bar{y} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_5 \end{bmatrix} \\ &= a_1^2(x_1 - \bar{x})^2 + a_2(x_2 - \bar{x})^2 + \dots \end{aligned}$$

즉, 이것의 최대값을 구하는 것은 경계의 이차곡선을 구하는 것이다. 그런데 이는 SVD의 문제와 같다. 따라서 벡터 $\mathbf{u}_i = A\mathbf{a}_i/\sigma_i$ 를 구하고, SVD의 문제를 풀면 된다. 간단한 결과 모형이 다음과 같다.



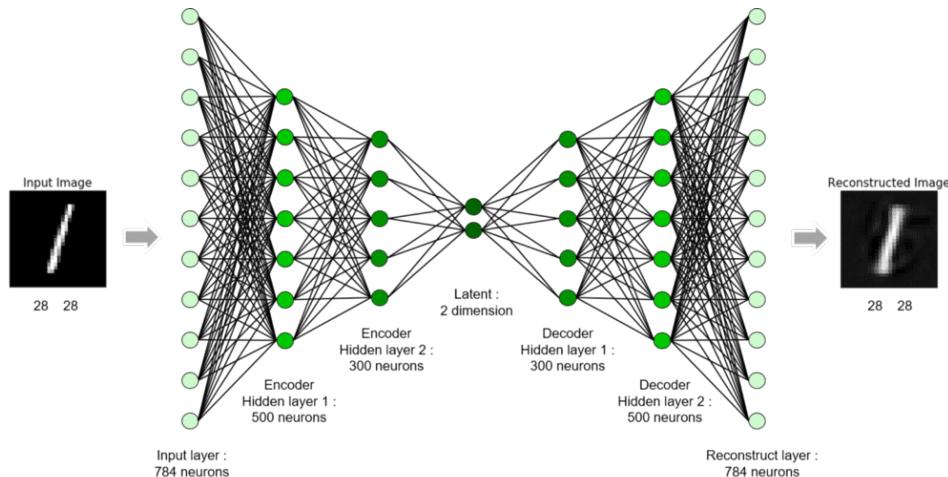
PCA를 시행할 때 주의해야 할 점이 몇 가지 있다. 비록 PCA가 데이터의 밀도를 잘 고려하는 방법이지만, 특정 데이터의 값이 너무 크다면 principal axis가 잘 안 잡힐 수 있다. 이런 데이터들을 outlier라고 한다.

다음으로는 데이터의 단위를 고려해야 한다. feature에 키와 손가락의 길이가 cm로 주어졌다고 생각해보자. 일반적으로 키가 손가락보다 매우 크기 때문에, 손가락 feature는 무시될 가능성이 있다. (즉, 손가락 데이터에 대한 principal axis 방향이 작아진다.) 따라서 모든 데이터의 중요성을 고려하여 크기를 조절하거나, 모든 데이터의 단위를 없애고 normalize해야 한다. 이는 Lecture 2에서 다뤘다.

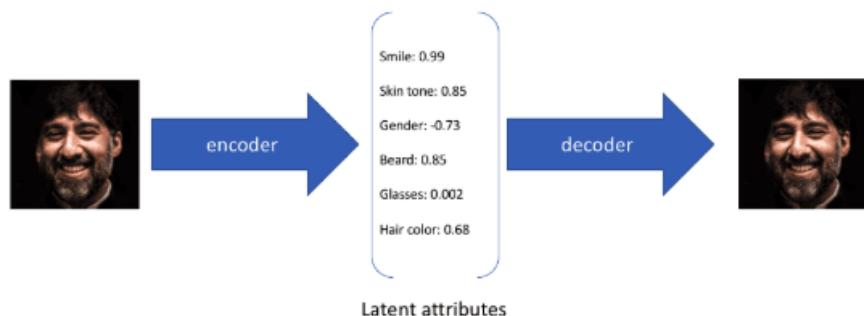
5.2 Auto Encoder

5.2.1 Auto Encoder (AE)

data를 encoding 하고 그것을 다시 decoding 하여 다시 만드는, unsupervised learning의 종류이다. 이 과정은 사람의 개입이 없는 'auto' 방식이다.



그 과정을 살펴보면, 우선 input data를 훨씬 작은 차원의 **latent vector (code)**로 바꾼다. latent vector는 각 성분마다 데이터를 성분을 담고 있다고 생각하면 된다.



그 후 이를 upsampling, 혹은 transposed convolution 방식으로 다시 생성한다. 기계를 학습하기 위해서는 input dataset이 모두 비슷한 것들이어야 한다. 만약 새, 고양이, 비행기 등이 섞여있는 dataset을 학습한다면 제대로 되지 않을 것이다. 위의 그림처럼 '1'이라는 숫자를 학습한다면, 첫번째로는 이 사진과 관련된 저차원벡터 'latent vector'를 얻을 수 있다. 두번째로는 원래 데이터와 비슷한, 그러나 원래 데이터와는 다른 'reconstructed output'을 얻을 수 있다. 물론 reconstructed output은 원래 데이터와 연관이 있으며, noise 가 많이 추가된 형태이다.

목표는 output \mathbf{y}^k 가 input \mathbf{x}^k 와 같아지는 것 이므로, 다음과 같은 Loss function이 사용 가능하다 (euclidean matrix norm을 사용한다):

$$\text{MSE} : L(\theta) = \frac{1}{2} \sum_k \|\mathbf{y}^k - \mathbf{x}^k\|_2^2 = \frac{1}{2} \sum_k \sum_i (y_i^k - x_i^k)^2$$

$$\text{Binary Cross-entropy} : - \sum_k \sum_i [x_i^k \log y^k + (1 - x_i^k) \log(1 - y^k)]$$

이제 standard backpropagation 으로 학습을 진행하면 된다.

학습하여 얻어진 AE machine은 다양하게 이용될 수 있다. latent vector는 원래 이미지의 특징을 잘 담고 있다고 할 수 있다. 따라서 다른 machine이 classification, clustering, anomaly detection을 할 때 input으로 원래 이미지(high dimension)를 넣는 것이 아닌, 추출된 latent vector를 넣어 학습시킬 수 있다.

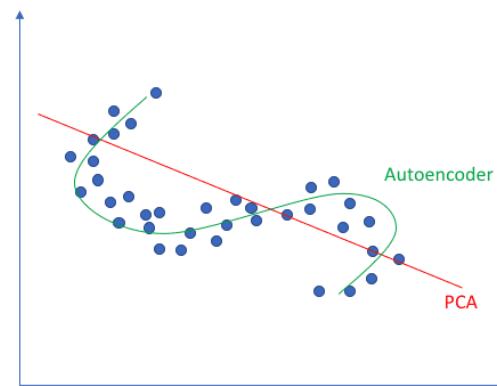
또한 latent vector는 뭔지 모르지만 data의 중요한 feature를 담고 있다. 이를 visualize 하면 우리가 알아보기 쉽게 clustering도 가능하다.

PCA vs AE 두 방법 모두 차원을 낮추는 방법이다.

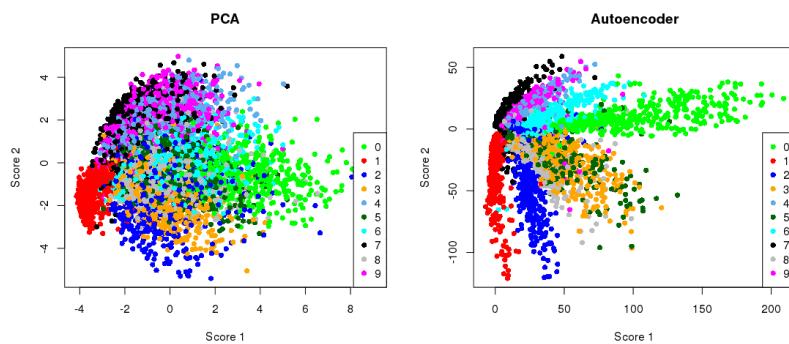
- PCA : AE의 linear한 모형이다. 낮은 차원의 hyper-plane을 찾는 것을 목적으로 하며 원래의 데이터의 선형 정보를 갖는다.
- AE : 데이터의 비선형 정보를 담는, 강력한 방법이다.

다음과 같은 파란색 점으로 표시된 데이터를 1차원으로 낮추고 싶다. 즉, 데이터를 잘 표현하는 선을 찾고 싶다. PCA를 적용했을 때와 AE를 사용했을 때의 결과가 다음과 같다.

Linear vs nonlinear dimensionality reduction

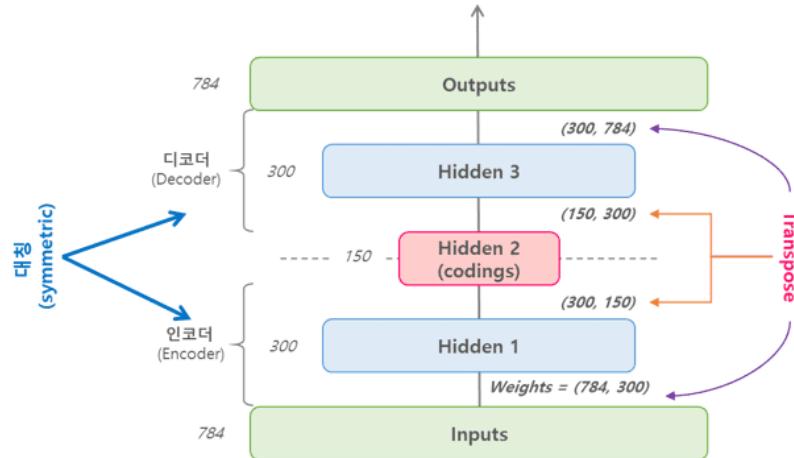


다음은 MNIST data에 대해 2차원 데이터로 낮추는 PCA와 AE의 결과이다.



5.2.2 Variations of AE

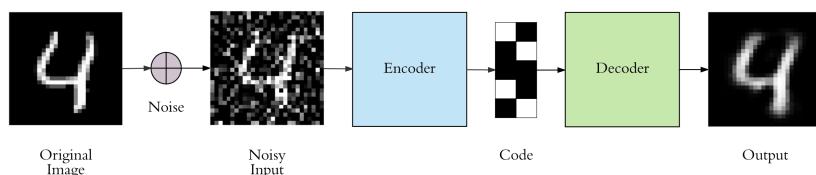
Tied Weight AE 이 방식의 Decoder는 Encoder와 Symmetric한 구조를 갖는다. 즉, Encoder의 첫 번째 hidden layer의 차원이 (n, m) 이었다면 Decoder의 마지막 hidden layer의 차원은 (m, n) 이어야 한다. 또한 parameter는 서로 같다. 따라서 decoding에서 parameter가 추가로 필요하지 않다.



이 방식은 필수적이지 않다. 사실 결론적으로 필요한 것은 input size와 output size가 같기만 하면 된다. 이 방식을 사용할 경우는 weight를 절반으로 줄여 학습을 빠르게 하고 overfitting을 방지할 수 있다.

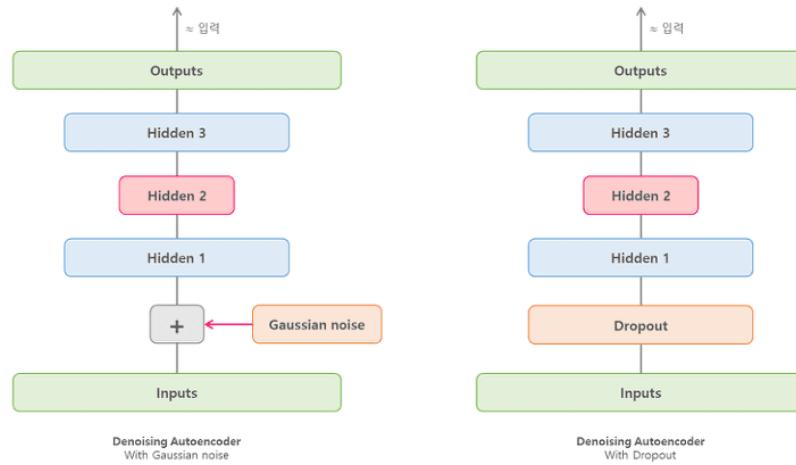
Encoder-Decoder 기본적으로 Auto-Encoder는 input과 output이 정확히 같아지는 것을 목표로 한다. 반면 Encoder-Decoder는 input의 feature를 갖고 있는 새로운 output을 만드는 것을 목표로 한다. 이를테면, 문장이 input으로 들어갔을 때 이를 표현하는 그림을 얻는 machine을 만들 수 있다.

Denoising AutoEncoder 본래 AE는 input을 넣고 target을 input으로 하여 비교하는 방식이었다면, Denoising AutoEncoder는 input에 noise를 추가하여 input으로 넣고, noise가 없는 input을 target으로 한다. 보통 noise로 normal random을 사용한다.



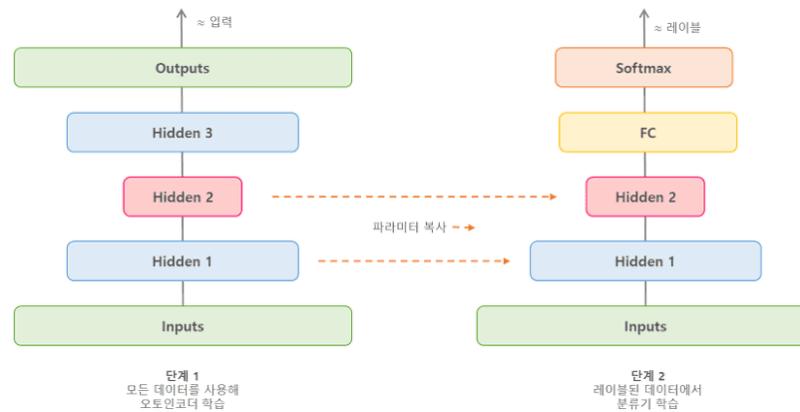
이렇게 학습을 하면, machine은 noise를 제거하는 방향으로 진행하게 된다.

noise를 추가하는 방식은 다양한 방법이 있다. normal random을 추가하거나, 일부 데이터를 없애거나 (dropout), 색을 없앨 수도 있다.



Stacked AutoEncoder input data의 dimension이 너무 큰 경우, 바로 AE를 적용해버리면 parameter가 너무 많아 학습이 잘 진행되지 않는다. Stacked AutoEncoder는 AE의 과정을 순차적으로 나눠 학습하는 방법이다. 이를테면 input dimension이 5000일 때, 우선 5000을 2500으로 줄이고 다시 2500으로 늘리는 AE를 만든다. 그럼 이제 2500에 대한 답을 가지고 있으므로, 다음에는 2500을 1000으로 줄이고 다시 2500으로 늘리는 AE를 만든다. 이런 방식을 반복하고, 최종적으로 모든 과정을 합치면 적절한 latent vector를 만드는 AE를 얻을 수 있다.

Classification using Stacked AE Stacked AE를 만든 후에 Decoder를 없앤 후 Classification machine을 붙인다. 사진에 대한 classification에서도 이를 사용할 수 있다.



우선은 Encoder에 해당하는 parameter는 고정된 값으로 하여 FC만 학습을 진행한다. 적절히 학습이 진행되었다고 한다면, Encoder 일부 layer도 parameter도 학습에 포함시켜 학습을 추가로 진행한다.

Semi-Supervised Learning label이 붙어진 데이터가 1만개, label이 붙어지지 않은 데이터 6만개가 있다고 하자.

- **Transfer Learning** : 이전에 학습된, 우리의 목적과 비슷한 역할을 하는 machine을 가져와 낮은 layer의 parameter만 고정하고 나머지를 학습한다.
- **AE method** : 7만개의 모든 데이터를 이용해 unsupervised learning을 하고, 만들어진 machine 위에 약간의 layer를 추가하고 낮은 layer의 parameter는 그대로 사용하여 label이 붙은 데이터 1만개로 학습을 진행한다.

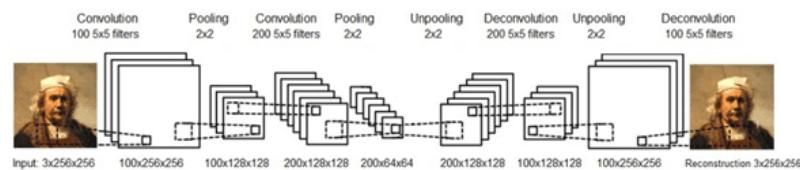
Anomaly Detection ‘일반적인’ 데이터와 다른, ‘stand out’ 을 골라내는 방법이다. 이를테면 MNIST 데이터에서 몇 개의 label을 바꿔 unusual data를 추려내고 싶다고 하자. 다음과 같은 간단한 과정으로 목표를 달성할 수 있다.

1. 우선 모든 데이터를 이용해 AE를 만든다.

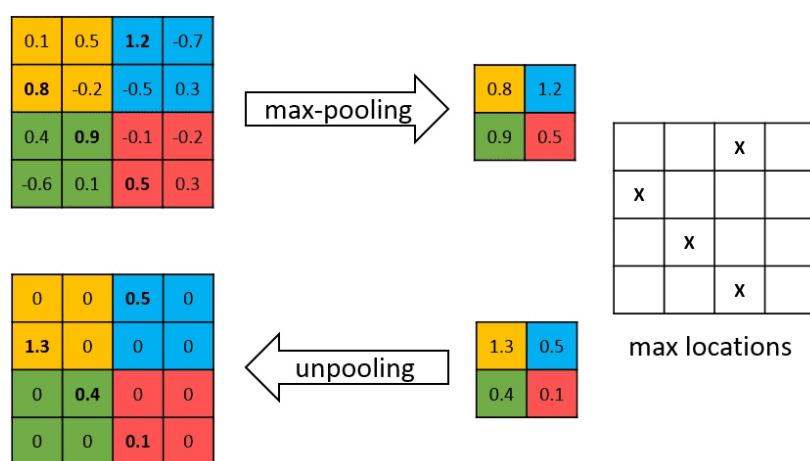
2. loss, cost, error 가 큰 데이터를 고른다.

물론 자연의 대부분의 데이터는 label이 없다. 이를테면 공장에서 기계를 돌리는데, 기계가 잘못 작동 할 때마다 소리를 다르게 낸다고 하자. Supervised Learning 을 사용하고자 하여 문제가 되는 소리를 직접 찾아 label 을 붙이는 작업을 해보면, 이상하다고 규정할 사례가 매우 적고 이를 구분하는 것 또한 사람이 하기 어려운 경우가 많다. Semi-Supervised Learning 을 이용한 Anomaly Detection 을 이용하면 어느정도는 가능하다. 최근에는 거의 모든 자동차에 이것이 부착되어 자동차 어디에 이상이 있는지 알려준다.

Convolutional Auto-Encoder - DeepPainter 2016년 Eli David 와 Nathan S. Netanyahu 는 작품을 기계에 넣으면 그 작품을 만든 화가를 맞추는 machine을 만들었다. 통상적으로 화가들의 그림은 100점 이내로 적고, 이 때문에 data augmentation 방법을 적용해도 overfitting이 발생하기 때문에 일반적인 CNN 방법은 사용할 수가 없다. DeepPainter의 방법은 다음과 같다.



1. Stacked AE 를 구성하여 AE를 학습시킨다. 또한 Denoising AE에서 20% pixel dropout 을 사용 했다.
2. 이 때 Decoder에서 사용되는 unpooling 방식은 다음과 같다.



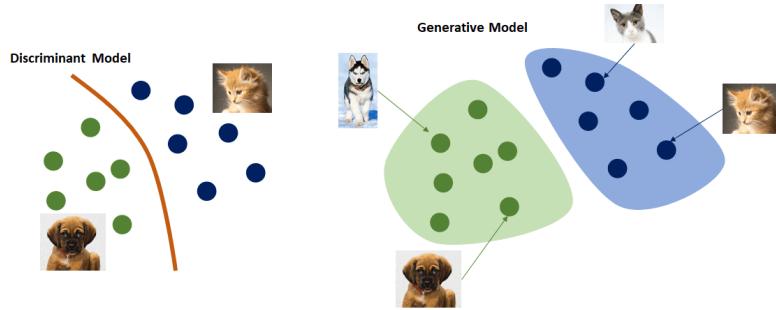
즉, pooling 과정에서는 max pooling을 사용하는데, 이 때 어느 부분이 max pooling 값으로 사용되었는지 max location을 저장하고, unpooling 시킬 때 이 위치를 그대로 사용하고 다른 곳은 0으로 채운다.

3. 학습된 AE에서 decoder는 없애고, encoder에 FC 를 연결하여 화가의 이름을 label한 후 classification으로 FC만 학습시킨다. cross validation을 사용했으며, 결과적으로 이 machine의 정확도는 96.52% 나 되었다.

5.3 Variational AutoEncoder

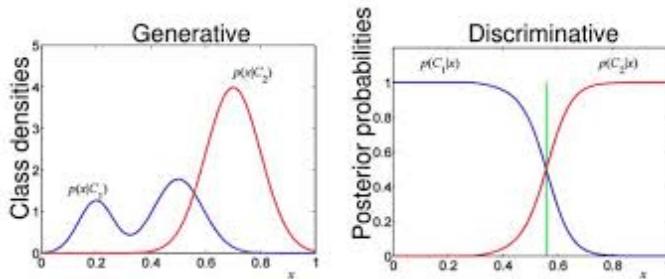
5.3.1 Motivations

Discriminative vs Generative models 머신러닝의 분류 알고리즘은 결론적으로 데이터간의 구분선을 찾는 과정이었다. 다음 그림을 살펴보자. Discriminative 과정은 ‘개’라는 class와 ‘고양이’라는 class를 구분하는 모델을 만든다면, Generative 과정은 이들을 구분해내는 것을 넘어 이들을 이해하고 있기 때문에 새로운 데이터를 생성할 수 있는 능력을 가진다.



어떤 입력 데이터 $X = \{x_1, \dots, x_m\}$ 과 출력 label $Y = \{y_1, \dots, y_m\}$ 값이 주어졌다고 하자.

- Discriminative model 은 데이터로부터 **conditional probability distribution** $P(y|x)$ 를 얻는다. classification 문제나, regression 문제의 경우 이 model은 좋은 결과를 낸다. 아래 그림을 보면, Discriminative model의 경우 C_1 으로 판별할 때 C_2 로 판별할 확률은 거의 0이다. 그리고 C_2 으로 판별할 때 C_1 로 판별할 확률은 거의 0이다. 때문에 Classification 문제에서 유용하게 사용된다.
- Generative model 은 데이터로부터 **joint probability distribution** $P(x,y)$ 를 얻으며 Bayes 통계를 근거로 $P(y|x)$ 를 예측한다. 이 model은 joint probability distribution 을 얻기 때문에 새로운 데이터를 생성할 수 있다.



Bayes Rule 간단히 짚고 넘어가면 다음과 같다. decision theory에서는 loss라는 함수를 정의하고, 각 decision에 따른 loss 값을 계산하여 데이터를 얻는다. 만약 loss가 작아야 한다면, 실험(이를테면 병의 치료를 위한)을 하는 대상자들을 실험군과 대조군으로 나눈 다음, 적절한 decision을 적용한다. 반복된 실험에서 얻게 되는 data는 두 가지로 나뉜다. 기술 descriptive 통계는 수집된 데이터의 요약 정보(치료 비용, 치료 효과 등)이며, 추측 inferential 통계는 확률을 이용한 정보를 추출(어떤 약품이 치료 효과가 큰지)하는 것이다.

통계량은 random sample의 함수이며, 여기에는 point statistics, interval statistic이 있다. 어떤 값이 통계량이 되는지 추정하는 것을 통계적 추정(estimation)이라고 하며, sample value에 따라 결과값을

내보내주는 함수를 추정량(estimator)라고 한다. 또한 estimator에 sample value를 대입하여 얻은 값을 추정값(estimate)이라고 한다. 최종적으로 분포를 추측하는 것을 회귀분석(Regression)이라고 한다. 검정(Classification)이란 특히 좁은 의미로 가설검정(hypothesis testing)을 말하며, 이는 특정 명제가 yes인지 no인지 확인하는 것 이다. machine learning에서, 통계에서 모든 문제는 regression 혹은 classification 이다.

다음은 data D , hypothesis H (='hypothesis is true')에 대한 Bayes 정리이다.

$$\mathbb{P}(Y|X) = \frac{\mathbb{P}(X|Y)\mathbb{P}(Y)}{\mathbb{P}(X)}$$

이 때, 간단히 용어를 정리하자면 다음과 같다.

- Posterior probability : $\mathbb{P}(y|x)$
- Likelihood : $\mathbb{P}(x|y)$
- Prior probability : $\mathbb{P}(y)$

다음의 예시를 생각하자. X는 동전의 종류가 A, B, C일 확률이며, Y는 각 동전을 던졌을 때 앞면이 나올 확률이다. 동전을 임의로 꺼내어 던졌을 때 앞면이 나왔다면 이 동전이 A, B, C일 확률은 각각 얼마인가?

	A	B	C
$\mathbb{P}(X)$	0.4	0.4	0.2
$\mathbb{P}(D)$	0.5	0.6	0.9

Bayes의 이론을 사용하여 A에 대한 확률을 구하면 다음과 같다. 어떤 것이 condition으로 들어있는지를 유의해야 한다.

$$\begin{aligned} \mathbb{P}(A|D) &= \frac{\mathbb{P}(A \cap D)}{\mathbb{P}(D)} \\ &= \frac{\mathbb{P}(D|A)\mathbb{P}(A)}{\mathbb{P}(D|A)\mathbb{P}(A) + \mathbb{P}(D|B)\mathbb{P}(B) + \mathbb{P}(D|C)\mathbb{P}(C)} \\ &= \frac{0.5 \times 0.4}{0.5 \cdot 0.4 + 0.6 \cdot 0.4 + 0.9 \cdot 0.2} \end{aligned}$$

사실은 다음과 같은 표를 사용하면 편리하다.

hypothesis H	prior $\mathbb{P}(H)$	likelihood $\mathbb{P}(D H)$	Bayes num. $\mathbb{P}(D H)\mathbb{P}(H)$	posterior $\mathbb{P}(H D)$
A	0.4	0.5	0.2	0.3226
B	0.4	0.6	0.24	0.3871
C	0.2	0.9	0.18	0.2903

다시 한 번 Remark 하자. 둘의 관계를 모르는 확률분포가 각각 있다면 joint 분포는 알 수 없다. 다만 만약 둘을 같은 sample에서 independent하게 뽑는다면(즉, i.i.d.) 라면 단순히 확률분포를 곱함으로서 그 joint 분포를 알 수 있다.

AE에서 우리의 목적은 데이터의 정보를 거의 그대로 간직한 차원이 낮은 latent vector 를 만드는 것 이었다. 즉, Encoder 만의 역할이 중요했다.

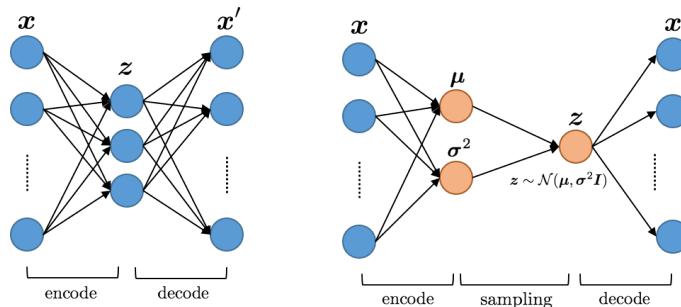
Motivation for VAE and GAN 이제 우리의 목적은 데이터가 적은 상태에서 ML을 하기 위해 데이터 수를 늘리는 것이다. 즉, 원래의 데이터와 비슷하지만 같지는 않는, 새로운 데이터를 ML을 통해 만들어내는 것이다. 정량적으로 말하자면, 원래의 데이터 X 로부터 $\mathbb{P}(X)$ 를 얻고 이 분포로부터 새로운 데이터 X' 을 만들어내는 것이다.

Generative model (생성모델)은 다음 두 과정으로 새로운 데이터 X' 을 만들어낸다.

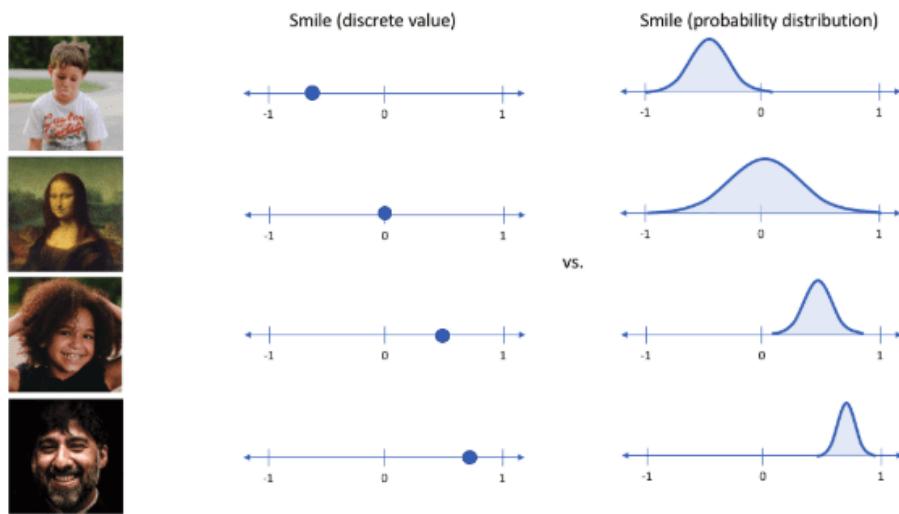
1. Z 를 latent variable 이라고 하자. 우선 $\mathbb{P}(Z)$ 를 normal distribution 이라고 가정한 뒤 $\mathbb{P}(X, Z) = \mathbb{P}(X|Z)\mathbb{P}(Z)$ 를 얻어낸다.
2. $\mathbb{P}(X, Z)$ 로부터 marginal distribution $\mathbb{P}(X)$ 를 얻어낸다. 이제 $\mathbb{P}(X)$ 로부터 새로운 데이터를 생성한다. 이들의 대표적인 예시는 VAE 와 GAN 인데, 이 둘은 다음의 특징을 가진다.
 - VAE 는 실재 분포 $\mathbb{P}_{data}(X)$ 과 유사하다고 생각되는 $\mathbb{P}_{model}(X)$ 를 데이터로부터 먼저 학습한다. 그 후 여기서 새로운 데이터 x' 을 만들어낸다.
 - GAN 은 $\mathbb{P}_{model}(X)$ 을 구할 필요 없이 바로 x' 을 만들어낸다.

5.3.2 Variational Auto-Encoder (VAE)

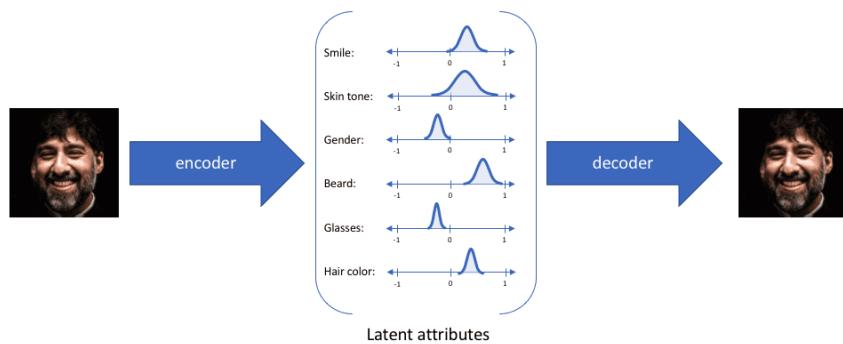
이전에 다루었던 AE의 경우, 하나의 latent vector에 대해 하나의 결과값만이 존재했다. decoder는 latent vector를 원래 데이터와 같도록 만드는 역할만을 수행했다. 반면 VAE는 latent space에서 latent vector를 생성하는 과정에서 확률론적 방법을 사용한다. 또한, encoder를 단순히 하나의 input에 대해 하나의 latent vector를 만들도록 하는 것이 아닌, latent attribute의 확률분포를 설명하도록 설계한다.



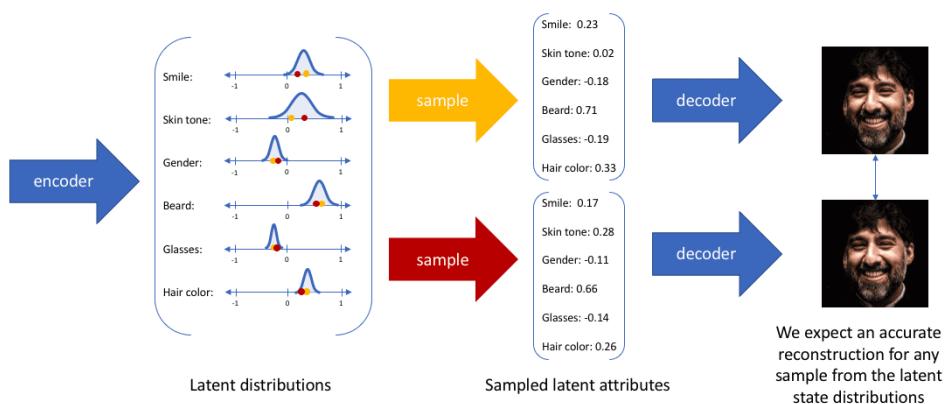
이 때, μ 와 σ 는 각각 mean vector, standard deviation vector 인데, latent vector 각 성분에 대한 그것이다. 이들을 계산하는 과정은 batch 몇 번으로 평균과 분산을 계산하는 것이 아니다. 이들도 hidden layer에서 나오는 값들이며 학습과정을 통해 찾아내는 값들이다. 즉, encoder에서 나온 벡터 두 개가 mean vector와 std vector이며, decoder에서는 이를 ‘평균’과 ‘분산’의 의미로 사용하게 되는 것이다. 이를테면 latent vector에서 ‘smile’을 나타내던 성분을 AE와 VAE의 관점으로 보면 다음과 같다.



AE의 경우는 하나의 값을 얻어 유일한 latent vector를 얻지만, VAE는 확률분포에 근거해서 값을 산출한다.



VAE는 확률분포에 근거해서 값을 산출하기 때문에, latent vector는 항상 달라진다.



Loss function 물론, AE때와 마찬가지로 input 값과 output 값이 정확히 같아지기를 바란다. 즉, 목표는 다음과 같다.

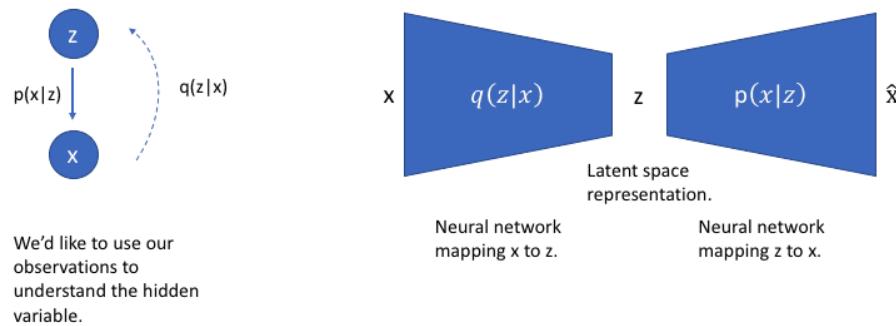
1. target을 input으로 하여 계산된 $MSE \sum_i(y_i - x_i)^2$, 혹은 cross-entropy $\mathbb{E}_{q(z|x)} \log p(x|z)$ 를 최소화 한다. 즉, input과 output이 같아지도록 한다.
2. data를 x , latent vector element를 z 라고 하자. latent vector 각각의 성분에 대해, 우리가 학습한 분포 $q(z|x)$ 가 정규분포 $N(0, 1)$ 과 같아지도록 한다. 따라서 latent vector 각각의 성분에 대한 학습한 분포 $q_k(z|x)$ 와 정규분포 $N(0, 1)$ 에 대한 KL-divergence를 사용한다. (KL-divergence : 확률분포간의 간격. not metric nor symmetric)

따라서 Loss function 은 다음과 같다.

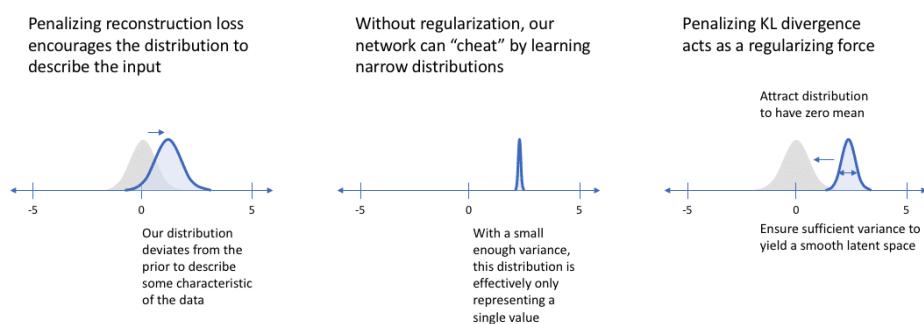
$$L(x, y) + \beta \sum_k D_{KL}(q_k(z|x) \| N(0, 1)) \quad \text{MSE} + \text{KL-divergence}$$

이 때, KL-divergence는 다음과 같다.

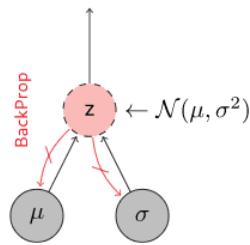
$$D_{KL}(P \| Q) = \mathbb{E}_x \log \frac{P(x)}{Q(x)}$$



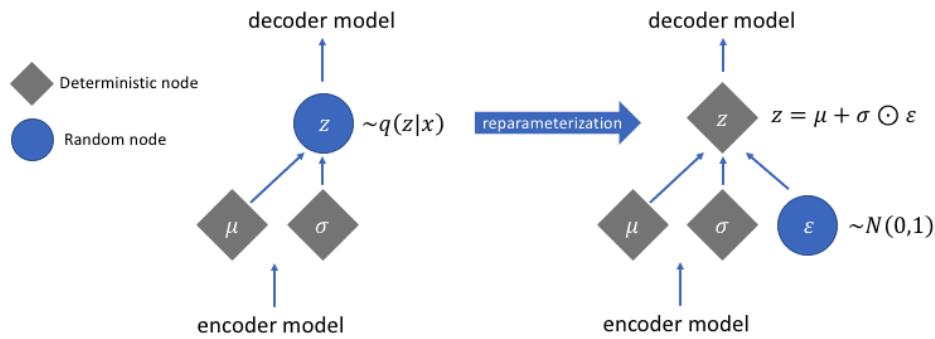
여기서 $L(x, y)$ 는 우리가 찾는 분포가 찾고자 하는 실제 분포에 가깝게 해준다. 만약 KL-divergence term이 없었다면, 데이터에만 최적화된, overfitting된 분포를 얻었을 것이다. 이 설명이 다음 그림에 나타나 있다.



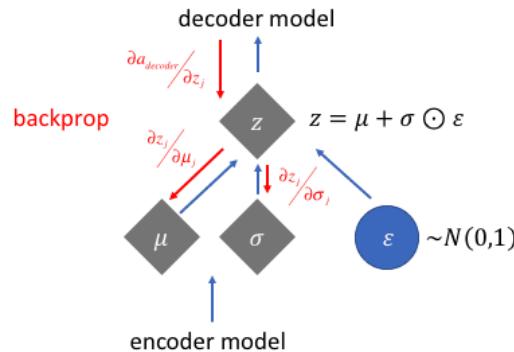
Backpropagation 하지만 우리가 지금까지 설명한 모델로 학습을 진행할 수가 없다. random sampling이라는 과정에서 backpropagation 이 불가능하기 때문이다.



이 때문에 Reparameterization trick 을 사용하게 된다. 즉, mean과 std는 그대로 선형결합되어 여기 사용되는 상수를 이용해 분포를 normal 하게 만들어주는 것이다.

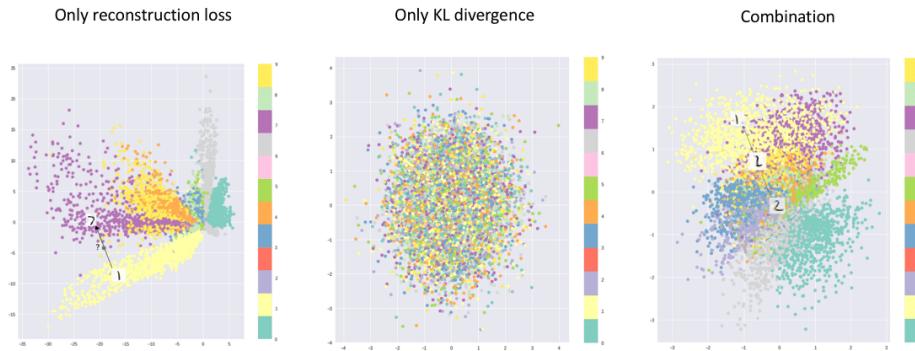


실제 계산은 다음과 같다.

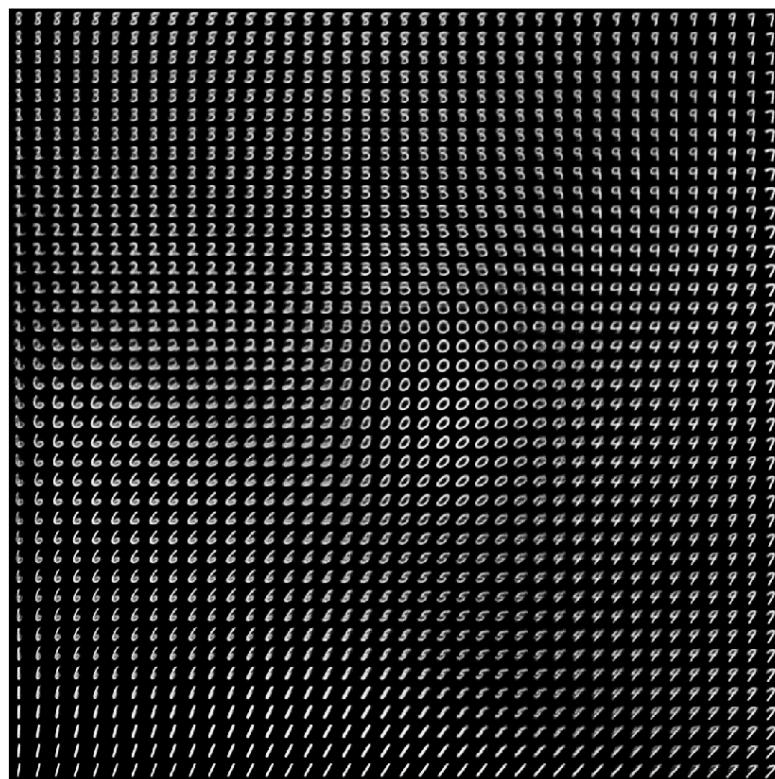


다시 언급하지만, AE는 input과 output을 같게 하는 기능을 이용해 데이터 차원을 낮추는 것에서는 아주 잘 작동했지만, latent space가 continuous 하지 않았기에 그 응용적 한계가 명확했다. 반면 VAE는 다음 그림에서 보이듯이 연속적인 latent space를 갖는데, 때문에 데이터를 연속적인 latent space에서 재생성할 수 있었다.

아래의 그림은 VAE에서 loss function으로 무엇을 사용했는지에 따른 clustering 결과이다. reconstruction error만 사용한 경우 AE와 유사한데, 즉 latent space에서 각 data 사이의 gap은 machine이 표현하지 않는다. KL divergence만 사용한 경우는 분포가 균일하기는 하지만 clustering이 전혀 되지 않았다. VAE는 마침내 clustering도 적절하게 하고, data 사이의 gap에 적절한 데이터를 생성하면서 latent space를 연속적으로 만든다.

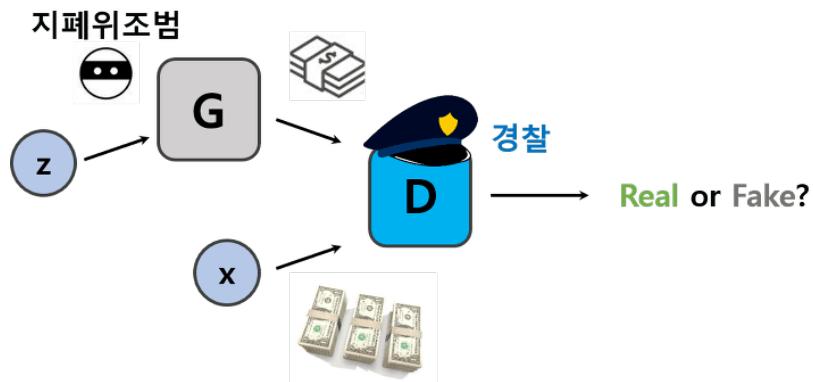


다음 그림은 MNIST 데이터를 적절한 cluster와 gap을 주어 섞어놓은 2D image이다. 다음과 같이 discontinuous 한 (즉, cluster 사이에 gap이 있는) 데이터를 AE에 넣을 경우 해당 gap을 어떻게 처리할지 결정하지 못하기 때문에 말도 안되는 결과를 출력한다. 사실 이 그림은 VAE의 latent 성분을 decoding 했을 때의 결과들인데, 이를 통해 VAE가 gap을 어떻게 처리하는지 알 수 있다.



5.4 Generative Adversarial Network (GAN)

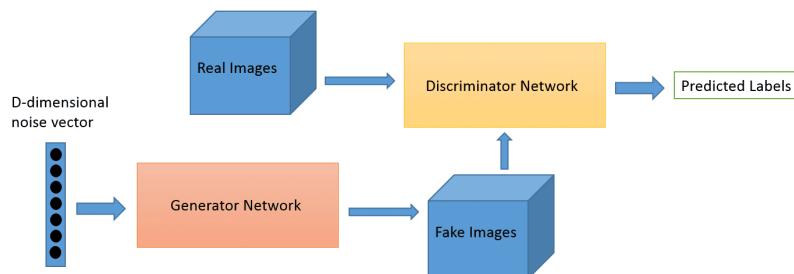
VAE는 데이터를 학습시키고, 하나의 데이터를 넣으면 그것과 유사한 것을 만들어낸다. 하지만 GAN은 확률분포와 연관이 없기 때문에, 침실 사진 수만장을 학습시켜도 그것과 겹치지 않는 새로운 침실 사진을 만들어낸다.



- Generative : generative model 을 학습한다. 즉, 데이터를 생성하는 구조이다.
- Adversarial : generator G (가짜 데이터를 생성하는 것) 와 discriminator D (진짜인지 가짜인지 구분하는 것) 가 경쟁학습을 한다.
- Network : DNN

GAN 은 Ian Goodfellow가 2014년 만든 작품으로, supervised learning에 치중되어 있던 DL 산업을 unsupervised learning 으로 돌려놓았다. GAN은 손쉽게 원래 데이터와는 다른 가짜 데이터를 생성해내기 때문에 이를 이용한 연구가 활발하게 진행되고 있다. GAN 방법은 non-convergence (convergence 보장이 안됨) 와 mode collapse (기본적으로 discriminative를 속이는 것이 목적이기에 하나가 잘먹히면 계속 개만 만들어냄) 문제로 인해 구현하기 어려운데, 최근에는 DCGAN, WassersteinGAN 등이 새로이 등장하고 있다. Yann LeCun 은 2016 세미나에서 GAN을 ‘근 10년간 가장 흥미로운 ML 주제’라고 표현하기도 했다.

Architecture GAN 의 기본적인 구조를 보면 다음과 같다.



우선, **Generator** G 는 discriminator를 속이기 위해, real data x 와 유사한 sample들을 만들어내야 한다. 이 때 Generator에는 Latent random variable ($N(0, 1)$ 이나 $U(-1, 1)$ 를 따르는) z (x 의 정보를 담기 위해 충분한 정도의 크기) 가 들어가 fake data $G(z)$ 가 생성된다. **Discriminator** D 는 들어온 input이 real data x 인지, fake data $G(z)$ 인지 구분한다. 두 machine을 경쟁적으로, 교대로, 서로가 서로를 도와주면서 학습시킨다. 이 과정을 위한 **Objective function** 은 다음과 같이 정의된다.

$$\min_G \max_D L(D, G)$$

즉, G 입장에서는 $L(D, G) |_{D \text{ fixed}}$ 를 최소화하고, D 입장에서는 $L(D, G) |_{G \text{ fixed}}$ 를 최대화 한다. 이 때 $L(D, G)$ 는 다음과 같다.

$$L(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log D(\mathbf{x}) \right] + \mathbb{E}_{\mathbf{z} \sim p_g(\mathbf{z})} \left[\log (1 - D(G(\mathbf{z}))) \right]$$

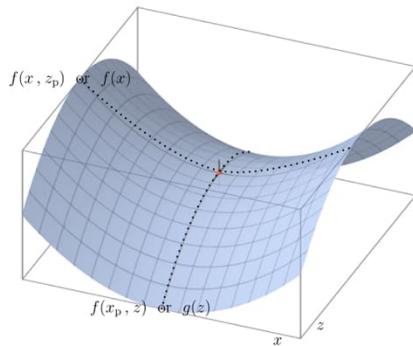
최종단계에서 sigmoid나 softmax를 취한다고 하자.

Discriminator D 는 **reward** $L(D, G)$ 를 최대화하려고 한다. 즉, $D(\mathbf{x})$ (진짜를 진짜라고 판별하는 경우) 를 1로 가깝게 하며 $D(G(\mathbf{z}))$ (가짜를 진짜라고 판별하는 경우) 를 0로 가깝게 하려고 한다.

Generator G 는 $L(D, G)$ 를 최소화하려고 한다. 즉, $D(G(\mathbf{z}))$ 를 1로 만들고자 한다. $D(\mathbf{x})$ 는 건들 수 없다.

이는 minimax two-player game 혹은 minimax problem 이라고 하는데, Nash 균형에 의하면 이런 경쟁 게임에서는 다음과 같은 경우에 균형이 만들어진다.

$$p_{data} = p_g \quad \text{i.e. } D(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} = \frac{1}{2} \quad \forall \mathbf{x}$$

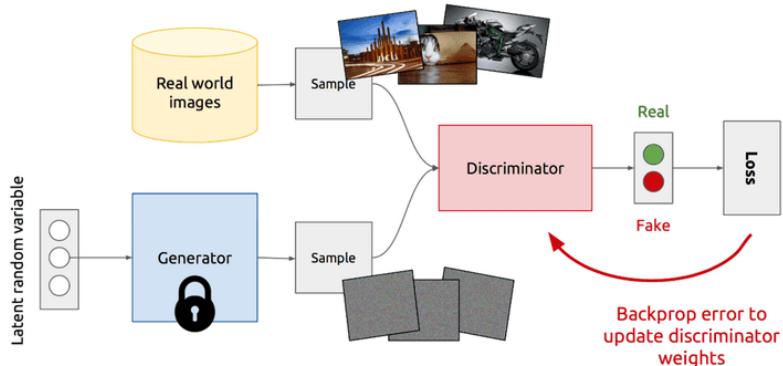


아주 좋은 경우 위의 그림과 같이 되겠지만, 사실 보통의 경우 이렇지 못하다.

Backpropagation D 와 G 각각을 training 하는 과정은 다음과 같다.

1. **training D** : Generator의 parameter θ_g 를 고정시킨 후 real data \mathbf{x} 와 generated data $G(\mathbf{z})$ 를 이용하여 gradient descent 를 시행한다. gradient 는 다음과 같다. 증가시키는 방향으로 gradient 를 적용한다. (gradient ascend)

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left\{ \log D(\mathbf{x}^{(i)}) + \log [1 - D(G(\mathbf{z}^{(i)}))] \right\}$$



학습할 때 real data 는 식의 앞부분을 담당하고, fake data는 뒷부분을 담당하게 될 것이다.

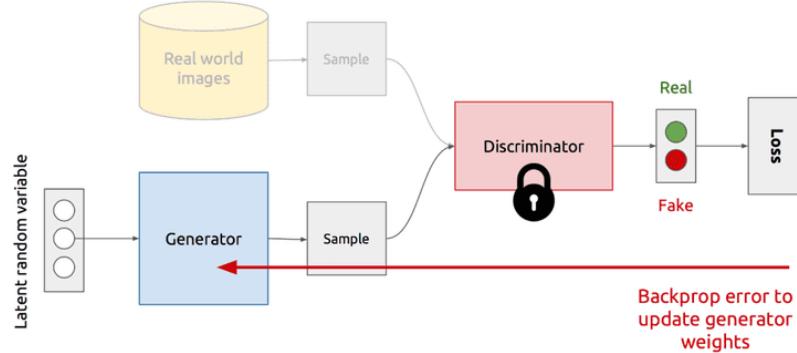
2. **training G** : Discriminator의 parameter θ_d 를 고정시킨 후 sample noise vector \mathbf{z} 를 이용하여 gradient descent 를 시행한다. gradient 는 다음과 같다.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log [1 - D(G(\mathbf{z}^{(i)}))]$$

혹은 다음을 사용하기도 한다.

$$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log D(G(\mathbf{z}^{(i)}))$$

아래식의 성능이 더 좋다.



이를 pseudo code로 작성한 algorithm이 다음과 같다. discriminator의 성능이 좋아야 generator의 학습 방향이 정해지기 때문에 보통은 discriminator를 먼저 학습한다. 또한 discriminator를 k 번 학습한 후 generator를 1번 학습하는 과정으로 반복하기도 한다.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```

for number of training iterations do
  for  $k$  steps do
    • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
    • Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
    • Update the discriminator by ascending its stochastic gradient:
  
```

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

end for

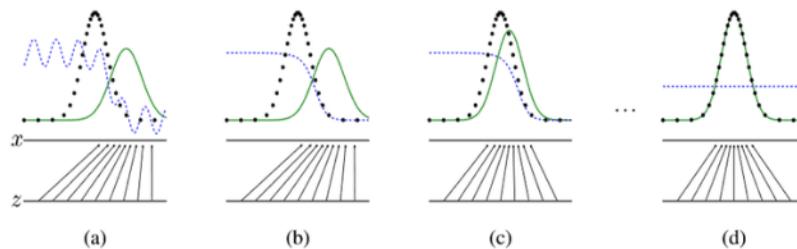
- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

위 과정을 반복적으로 수행하다 보면, D 와 G 가 발전을 거듭하고 결국 평형상태에 도달하게 된다.



검은 점이 실재 데이터, 파란선은 discriminator, 초록선은 generator 이다. (a) 처음에 만들어진 generator의 fake data는 영만이지만 generator 역시 마찬가지이다. discriminator는 $D(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}$ 를 계산하며 update 한다. (b) fake data가 몰려있는 곳에서 discriminator $D(\mathbf{x})$ 는 작아지며 real data가 몰려있는 곳에서는 커진다. (d) 최종적으로 generator는 real data와 거의 같은 fake data를 생성하며 discriminator는 구분하지 못한다. 여기가 평형상태이다.

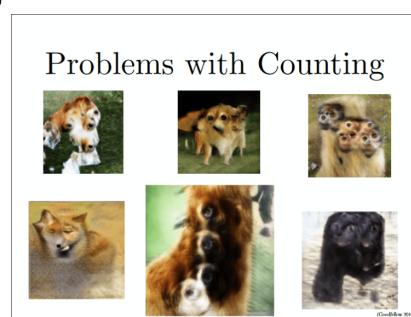
Pros and Cons of GANs

- Pros:

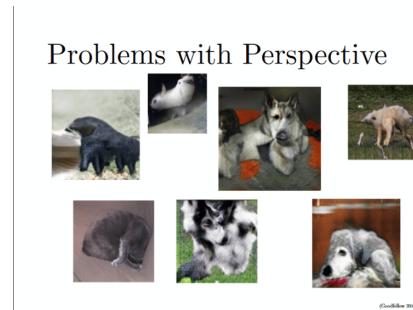
- Unsupervised Learning 이기 때문에 label 작업이 필요 없다.
- high-quality의 데이터를 만들어낸다.
- optimization은 기준에 알던 것을 그대로 사용하면 된다.
- explicit intractable integral 이 필요없다.
- Markov chain 이 필요없다.

- Cons:

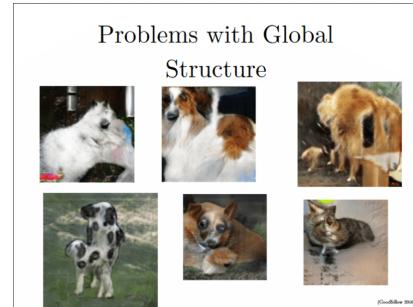
- converge 보장이 없다.
이는 minimax problem에서 나타날 수 밖에 없는 문제이다. G 와 D 의 학습이 독립적으로, 번갈아서 일어나기 때문에 일어나는 문제이다.
- diminished gradient problem
만약 discriminator가 너무 강해 generator로부터 온 fake data를 모두 구분한다면, generator의 training function은 0이 될 것이다. 이는 vanishing gradient 문제이다. 만약 discriminator가 너무 약하다면, generator는 대충 만들어도 discriminator가 구분하지 못하니 학습이 되지 않을 것이다.
- Mode collapse
가장 큰 문제이다. optimization 과정에서 local minimum을 찾고 다른 곳으로 가지 않는 문제라고 볼 수 있다. generator가 적절한 결과 하나를 찾아내서 계속 그것만 생성하는 경우도 여기에 해당한다.
- 다음 사진들은 GAN이 만든 것인데, 실제로는 절대 불가능한 것들이다.
 1. 개수에 대한 개념이 없다.



2. 입체에 대한 개념이 없다.

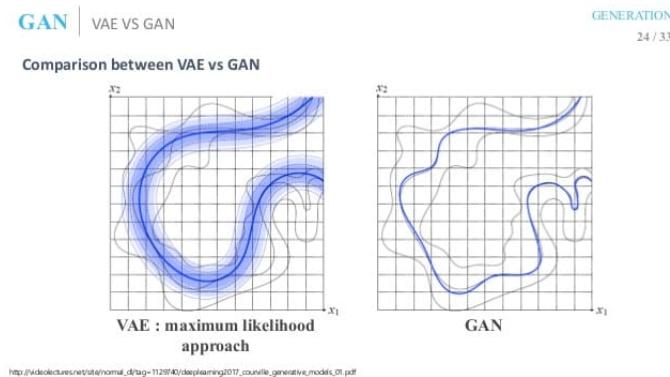


3. local하게 보면 맞는 듯 하나 global하게 보면 틀리다.

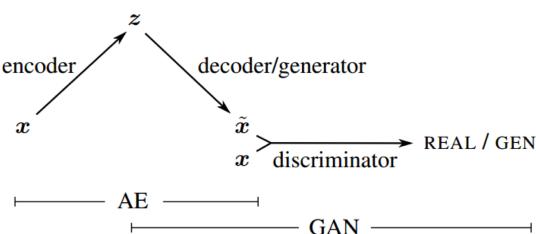


- $p_g(G(\mathbf{z}))$ 의 정확한 식이 없다.
- convergence 보장이 없기 때문에 training 과정을 유심히 관찰하여 언제 멈출지를 결정해줘야 한다.
- 평가 방법이 없어 다른 모델과 비교가 어렵다.

VAE-GAN 은 VAE와 GAN을 합친 모델이다. VAE는 넣는 사진과 유사한 데이터를 얻는 machine 이다. 확률적으로 연속적인 공간을 사용하기 때문에 사진이 흐려지는 문제가 있다. GAN은 확률에 근거하지 않기 때문에 매우 sharp한 데이터를 얻는다.



VAE-GAN은 VAE를 generator로 사용한다. 이렇게 하면 GAN에서 뜬금없는 결과가 나오는 것은 방지할 수 있다. 하지만 학습량이 너무 많고 오래걸리는 단점이 있다.



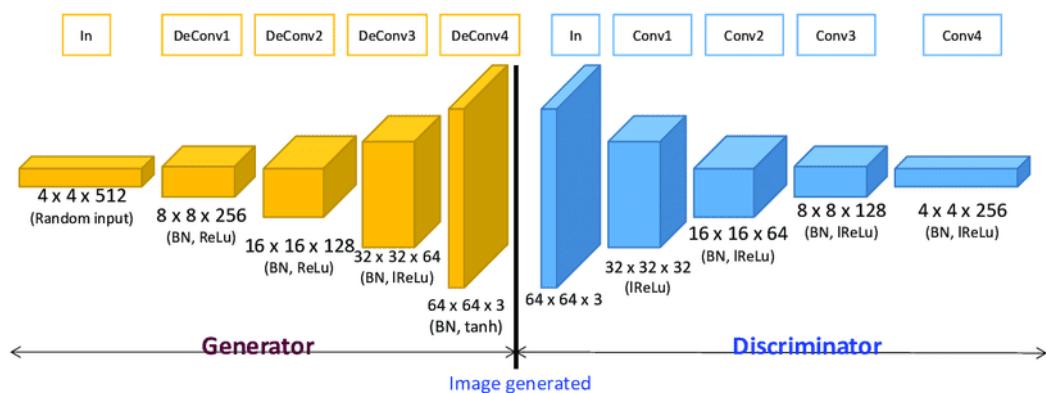
5.5 GANs - DCGAN, cGAN, CycleGAN, PGGAN, WGAN

다음 사이트에서 GAN의 초기부터 2018년 9월까지의 GAN 계보를 볼 수 있다.

<https://github.com/hindupuravinash/the-gan-zoo/blob/master/gans.tsv>

Vanilla GAN 초기의 GAN이며 결과가 좋지 못했다. generator, discriminator의 network는 FC로 되어있다.

Deep Convolutional GAN (DCGAN) Convolution을 이용한, 가장 성공적이고 유명한 GAN의 종류이다. max pooling, FC layer가 없다는 특징을 찾아볼 수 있다.



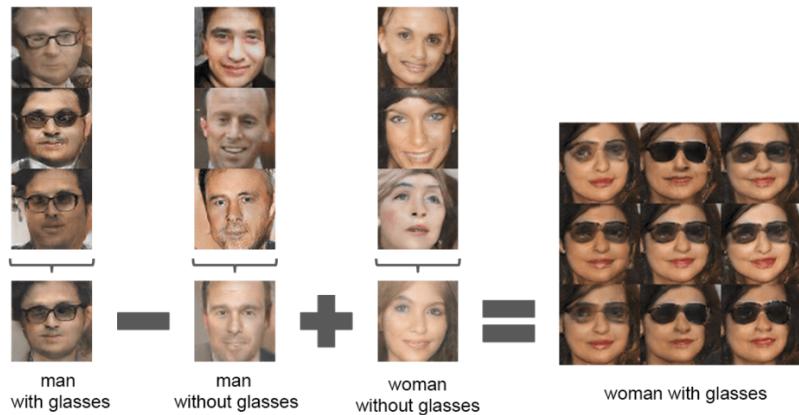
Generator는 up-sampling을 위해 transposed convolution을 사용했고, Discriminator는 down-sampling 과정에서 convolution 만을 사용했다. 만약 max pooling 을 사용한다면 정보를 잃는 것인데, 이는 discriminator의 input으로 살짝 찌그러진 이미지가 들어와도 구분을 못한다는 것이다. 즉, generator 가 약간 이상한 image를 만들어도 잘 구분하지 못한다는 것이다.

DCGAN 논문의 가이드라인을 보면 다음과 같다.

- pooling layer를 전혀 쓰지 않고, discriminator의 경우 strided convolution (전형적인 down-sampling convolution)을, generator의 경우 fractional-strided convolution (transposed convolution for up-sampling)을 사용한다.
- 각각의 output layer를 제외한 모든 곳에 Batch normalization을 넣는다.
- FC 는 전혀 넣지 않는다.
- generator의 경우 output으로 tanh 를 사용하며 이외에는 ReLU 를 사용한다.
- discriminator의 경우 output으로 sigmoid를 사용하며 이외에는 LeakyReLU 를 사용한다.

DCGAN에는 재미있는 특징이 있다.

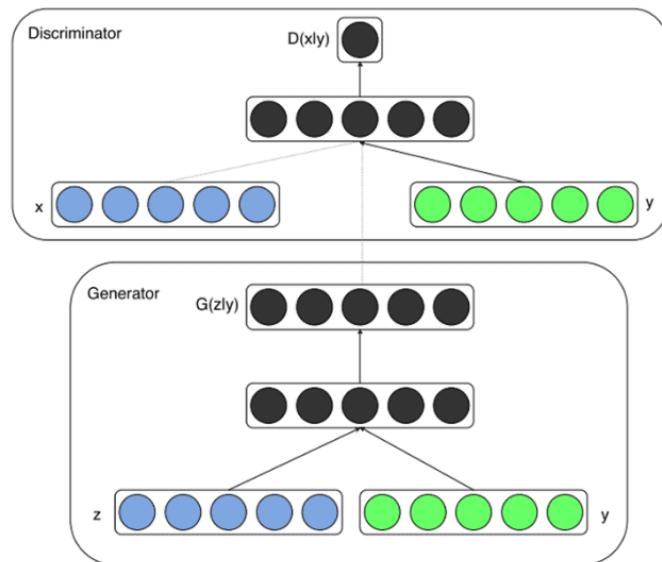
1. Latent vector 연산이 가능하다. 즉, 특정 feature를 latent vector 사칙연산을 통해 섞을 수 있다.



2. generator의 input인 z 값 (latent space) 을 조금씩 바꿔가면서 (walking) 생성되는 output 이미지의 예시인데, 아주 부드럽게 바뀌는 것으로 보아 latent space의 연속성도 확인할 수 있다.



Conditional GAN (cGAN) 원래 Generator에 들어가는 값은 label이 없는 random variable, 즉 latent variable z 이다. cGAN은 z 에 label y 를 추가해 generator에 넣으면, discriminator에 real data x , fake data $G(z)$ 들어갈 때도 같이 들어간다. 물론 이제는 label이 추가됐으므로 $G(z|y)$ 라고 써주어야 한다.



이 때 주의해야 할 것은, \mathbf{z} 는 정규분포나 균등분포에 불과하며, $G(\mathbf{z}|\mathbf{y})$ 가 실재 분포에 가까워지도록 generator를 학습하는 것이다. 그러면 label \mathbf{y} 를 바꾸는 것 만으로 generating 되는 결과들이 $G(\mathbf{z}|\mathbf{y})$ 의 분포 안에서 적절히 바뀌게 된다.

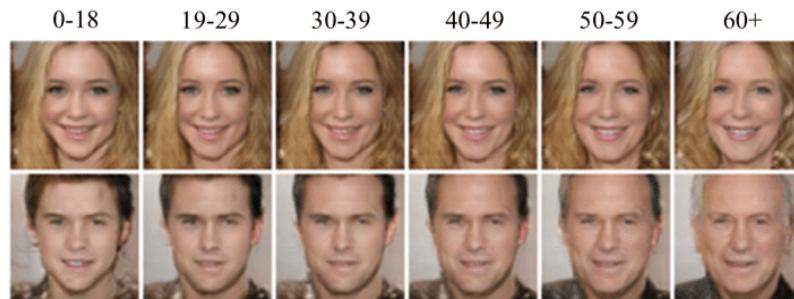
Objective function은 다음과 같다.

$$\min_G \max_D L(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log D(\mathbf{x}, \mathbf{y}) \right] + \mathbb{E}_{\mathbf{z} \sim p_g(\mathbf{z})} \left[\log (1 - D(G(\mathbf{z}|\mathbf{y}))) \right]$$

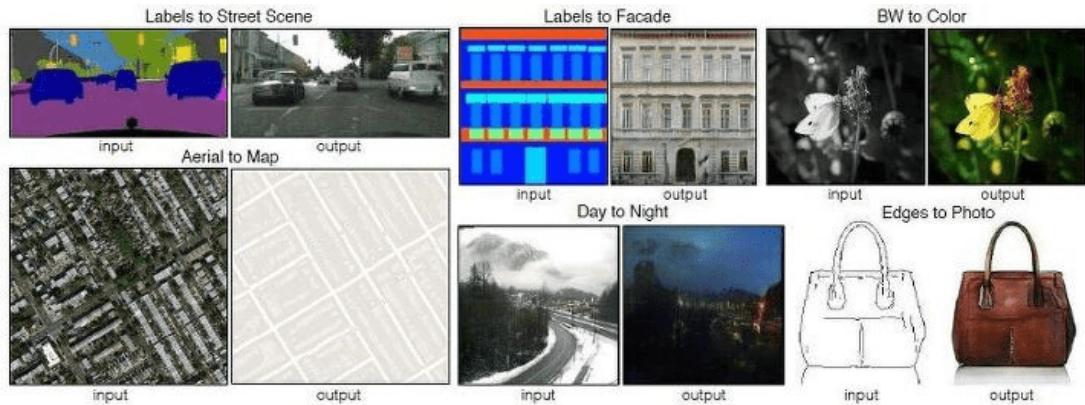
cGAN을 최초로 발표한 논문에서는 MNIST data에 대하여 condition \mathbf{y} 를 각 entry 순서대로 0, 1, 2,..., 9 의 label을 붙였는데, 이에 따라 generator에 latent variable \mathbf{z} 와 특정 entry만 1로 넣은 label vector \mathbf{y} 를 넣으면 다음과 같은 결과가 나온다.



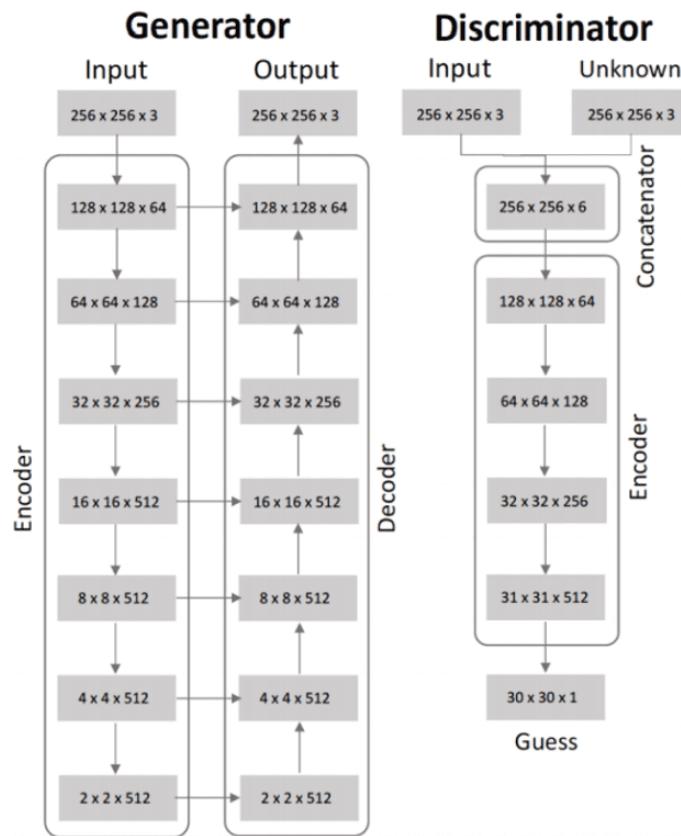
이를테면 각 데이터에 ‘나이’에 해당하는 label을 추가한다고 하자. 학습용 data에는 나이 label이 모두 붙어있다고 가정하자. 그러면 새로운 데이터를 만들 때 label을 넣으면 원하는 나이대의 사진을 얻을 수 있다.



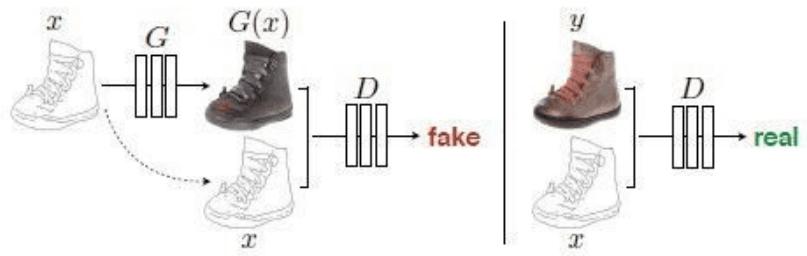
Pix2Pix with cGAN 간혹 이미 있는 이미지를 다른 스타일의 이미지로 변형해야 하는 경우가 있다. 이를테면 스케치에 색을 칠하거나, 낮 사진을 밤으로 바꾸거나, 여름의 사진을 눈쌓인 겨울로 바꾸고 싶은 경우가 그 예시이다.



기존의 GAN은 generator의 input으로서 latent variable \mathbf{z} , 즉 ‘random’ 값을 받았다. Pix2Pix는 cGAN의 방식을 쓰며 generator는 latent variable \mathbf{z} 가 아닌, 변형하고자 하는 이미지 \mathbf{x} 를 입력으로 받는다. 이를테면 스케치를 입력하면 채색을 해주는 machine을 만들고 싶다고 하자. 그렇다면 우리는 우선 스케치와 채색된 결과가 pair로 연결된 dataset이 필요하다. 즉, input \mathbf{x} , target \mathbf{y} 가 paired 된 tupled dataset 이 필요하며 사람 채색을 하는 trained machine의 결과가 같이 나와있다.



사물을 변형한다고 하자. 그림이 조금 헛갈리는데, 스케치 \mathbf{x} 는 label 이다. generator G 에는 latent variable \mathbf{z} 와 label \mathbf{x} 가 함께 들어가 fake data $G(\mathbf{z}, \mathbf{x})$ 가 생성되며, 이 fake data $G(\mathbf{z}, \mathbf{x})$, label \mathbf{x} , label \mathbf{x} 에 해당하는 real data \mathbf{y} 가 discriminator에 들어간다.



여기서 Objective function은 다음과 같다. 기본적으로 cGAN의 구조이지만 label인지 input인지 모를 label x 가 사용되었다는 것, generator에서 나온 fake data $G(z, x)$ 가 기본적으로 label x 에 해당하는 real data y 와 유사해야한다는 것에 초점을 맞추어 변형하면 다음과 같다.

$$\min_G \max_D L_{\text{cGAN}}(D, G) + \lambda L_{L_1}(G)$$

where

$$L_{\text{cGAN}}(D, G) = \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \left[\log D(\mathbf{x}, \mathbf{y}) \right] + \mathbb{E}_{\mathbf{x}, \mathbf{z} \sim p_g(\mathbf{z})} \left[\log (1 - D(G(\mathbf{x}, \mathbf{z}))) \right]$$

and

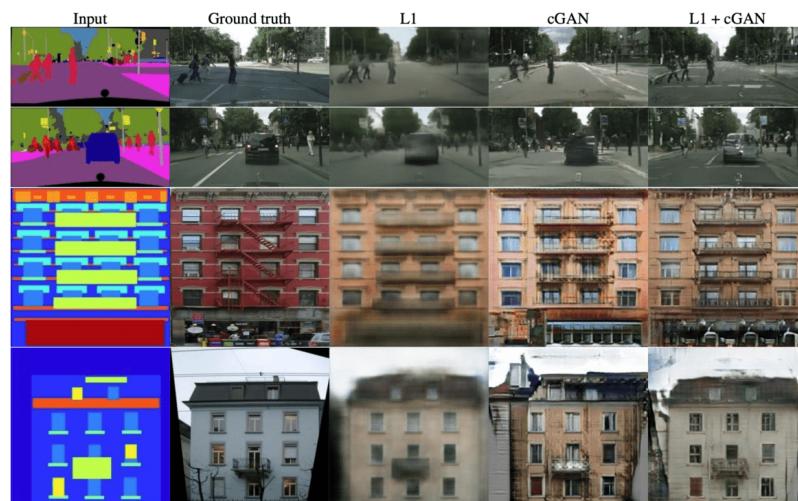
$$L_{L_1}(G) = \mathbb{E}_{\mathbf{x}, \mathbf{y}, \mathbf{z}} \left[\| \mathbf{y} - G(\mathbf{x}, \mathbf{z}) \|_1 \right]$$

*norm 1 (L1 norm) 은 절대값들의 합, norm 2 (L2 norm) 는 제곱의 합의 제곱근이다.

논문에서 소개된 Pix2Pix의 구조를 살펴보면 다음과 같다.

- generator는 스케치 x 를 input으로 하고 real data y 로 하는 U-Net 구조이다.
- real data y 와 스케치 x 를 넣으면 ‘real’ 을 반환하도록 하며, 스케치 x 를 generator에 넣어 얻은 $G(\mathbf{x}, \mathbf{z})$ 와 스케치 x 를 넣으면 ‘fake’를 반환하도록 하는 discriminator를 학습시킨다.
- discriminator를 학습하는 과정에서는 이미지를 쪼개서 patch 끼리 비교하는 PatchGAN 을 사용한다.

실제 사진을 다양하게 단순화 혹은 변형한 다음과 같은 input x 를 생각해보자. L1 norm만을 사용한 CNN (U-Net) 은 흐린 결과를 준다. cGAN은 다양한 feature를 추가하는 등 완전히 새로운 데이터를 만들어낸다. L1 norm을 사용한 U-Net과 cGAN을 함께 사용하면 원본과 비슷한, 선명한 결과를 준다.



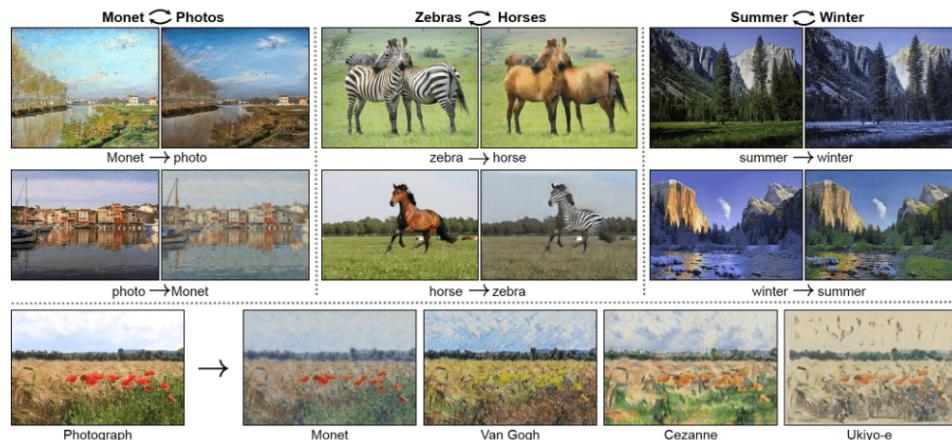
Generator의 objective function에는 L1 loss 가 포함되어있는데, 이는 real data y 와 U-Net으로 생성된 $G(x, z)$ 의 거리를 최소화하는 역할이기 때문에 통상적으로 이미지의 평균 성분, 즉 저주파에 집중하게 된다.

그렇다면 Discriminator의 objective 에서는 고주파 성분에 집중할 필요하 있다. 통상적인 GAN에서는 이미지 전체에 대한 score를 사용했다. PatchGan은 특정 크기의 patch 단위로 'real' 과 'fake' 를 판별하고, 그 결과에 평균을 취하는 방식이다. 서로 면 patch일수록 서로 연관이 없어지는 경향이 있다. 대다수의 patch에 대해 진짜같은 이미지를 생성할 수 있다면, generator의 성능은 더 올라갈 수 있을 것이라 가정할 수 있다. 그리고 Pix2Pix 개발팀은 이를 실험적으로 검증했다.

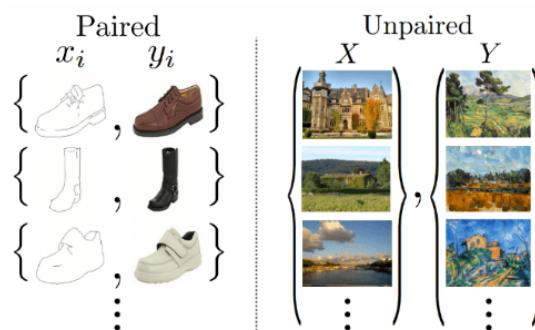
Patch의 크기를 너무 작게 쪼개면 흐린, 좋지 못한 결과를 준다. 너무 큰 것도 좋지 못한 결과를 준다. 다음 그림에서 보이듯이 70 patch를 사용했을 경우 가장 좋은 결과를 냈다고 한다.



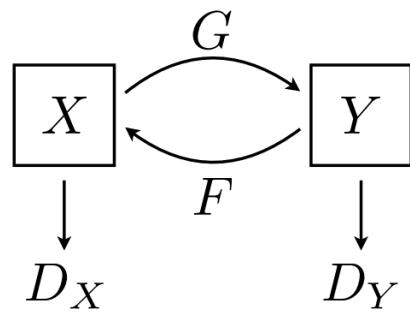
CycleGAN 사진의 기본 구조는 유지하되, 그 중 어떤 특징만 바꿔주는 방법이다. 이는 Pix2Pix와 같은 기능이지만, paired set이 없어도 된다는 장점이 있고, Pix2Pix 보다는 성능이 (당연히) 떨어지지만 unpaired 기법 중에서는 가장 성능이 좋다.



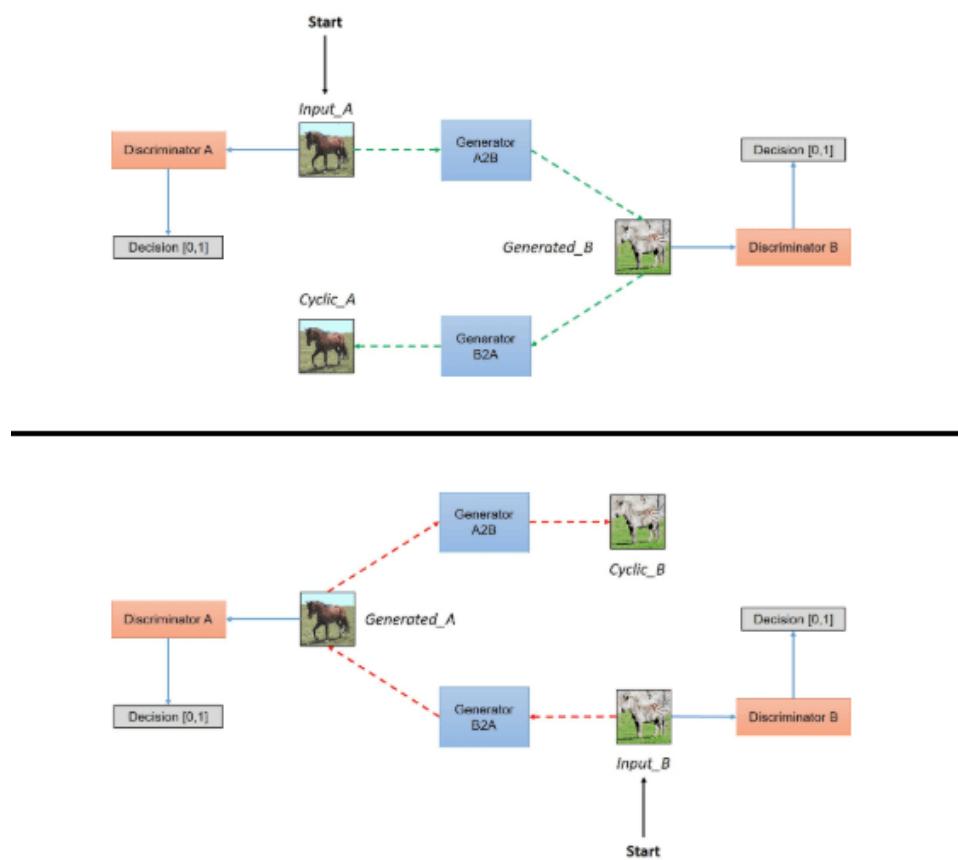
Pix2Pix 는 paired dataset이 있어야 한다. 신발 스케치를 input으로 하고자 한다면 그 스케치와 실재 그림이 있어야 한다. 하지만 일반적인 경우 이것은 불가능하다. CycleGan은 dataset이 unpaired 되어있어도 학습할 수 있는 방법이다.



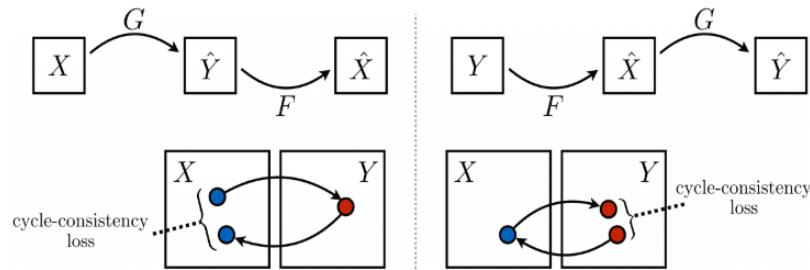
CycleGan은 X 라는 input set에서 Y 라는 output set의 style로 바꿔주는 방식이다. 그 구조를 보면 이와같이 단순하다.



G 는 X 도메인 (즉, X dataset이 존재하는 도메인)에 존재하는 사진을 Y 도메인으로 변환하며, 반대로 F 는 Y 도메인의 사진을 X 도메인으로 변환한다. D_X 와 D_Y 는 각각 X 도메인과 Y 도메인을 위한 판별자이다. 즉, generator G 는 discriminator D_Y 와 GAN을 구성하며, generator F 는 discriminator D_X 와 GAN을 구성한다. 이를 펼쳐보면 다음과 같이 볼 수도 있다.



하지만 X 에서 사상된 $G(X)$ 가 얼마나 Y 도메인에 잘 갔는지는 알 길이 없다. 물론 D_Y 가 존재하기는 하지만, 그 자체로 충분하지는 않다고 한다. 따라서 CycleGAN은 input X 를 $G(X)$ 로 보낸 후 다시 원래로 돌려 ($F(G(X))$) 이것이 원래 input X 와 얼마나 같은지를 평가해 학습을 진행한다. (이를 Cycle Consistency 라 한다.) 때문에 결론적으로 사용하지 않을 F 가 존재하는 것이며, 한 번 갔다가 다시 와야하기 때문에 parameter는 2배가 되는 것이다.



물론 Cycle Consistency는 inverse problem 이기 때문에 악영향을 줄 수도 있다고 생각할 수 있다. 그러나 이것이 최선이며, 결과적으로는 좋은 모델을 만든다.

Objective function은 다음과 같다.

$$\min_G \max_D [L_{\text{GAN}}(G, D_Y, X, Y) + L_{\text{GAN}}(F, D_X, Y, X) + \lambda L_{\text{cyc}}(G, F)]$$

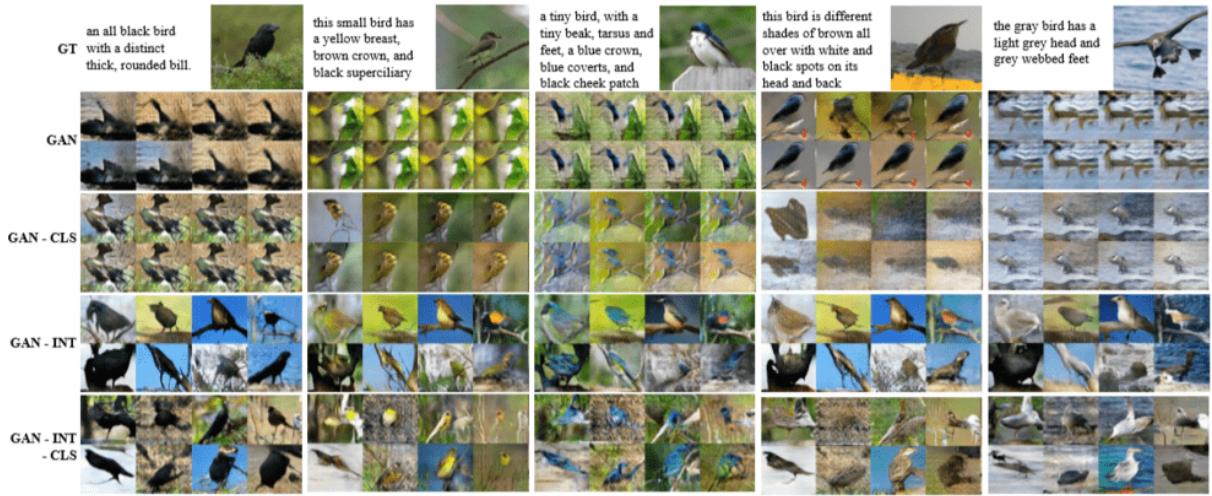
또한 각각의 loss는 다음과 같다.

$$\begin{aligned} L_{\text{GAN}}(G, D_Y, X, Y) &= \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{x})} \left[\log D_Y(\mathbf{y}) \right] + \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log (1 - D_Y(G(\mathbf{x}))) \right] \\ L_{\text{GAN}}(F, D_X, Y, X) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log D_X(\mathbf{x}) \right] + \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{y})} \left[\log (1 - D_X(F(\mathbf{y}))) \right] \\ L_{\text{cyc}}(G, F) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\| F(G(\mathbf{x})) - \mathbf{x} \|_1] + \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{y})} [\| G(F(\mathbf{y})) - \mathbf{y} \|_1] \end{aligned}$$

첫 번째 term은 정방향의 G 에 대한 GAN loss, 두 번째 term은 역방향의 F 에 대한 GAN loss, 마지막 term은 cycle consistency에 대한 L1 norm term이다. L1 norm은 pixel by pixel 방식으로 계산을 진행하는데, ‘어떤 부분을’ 바꿔야 하는지에 대한 정보가 없기 때문에 다음과 같은 문제가 발생하기도 한다.

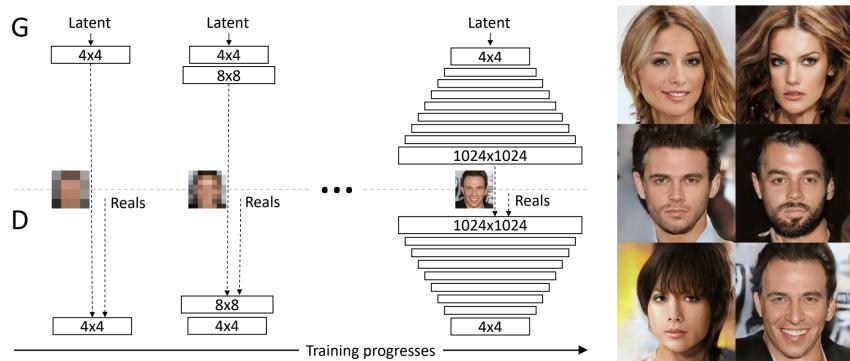


Text2Image with cGAN 텍스트가 input으로 들어오면 이를 image로 표현하는 machine이다.



GAN의 경우 계속 비슷한 것만 generate 하는 것을 볼 수 있는데, 전에 언급한 mode collapse 이다.

Progressive growing GAN (PGGAN) 연예인들의 사진을 통해 학습하고, 그들의 특징을 얻어 새로운 가상의 인물사진을 만들어내는 machine이다.



처음부터 너무 고차원의 학습을 진행할 수가 없기 때문에 stacked AE와 비슷한 방식을 사용한다. 최초에는 real image를 모두 4×4 pixel로 변환하고 이를 GAN으로 학습한다. 여기서 학습된 generator과 discriminator 모두 8×8 convolution layer를 한 층 쌓는다. 그리고 8×8 로 변형된 real image를 통해 모든 layer를 또다시 학습한다. 이런 방식으로 1024×1024 까지 늘리면 최종적으로 high-quality image를 얻게 된다.

Wasserstein GAN (WGAN) 해석학에 유명한 연구소에서 만든 방법으로, 상당히 어렵다. 우선 확률분포의 거리를 정의해야 한다.

- Total Variation distance

$$\delta(\mathbb{P}, \mathbb{Q}) = \sup_{A \in \Sigma} |\mathbb{P}(A) - \mathbb{Q}(A)|$$

where

$$\mathbb{P}(A) = \int_A P(x) dx$$

식에서 볼 수 있듯이 이 정의는 metric이다. (그래서 distance라고 한다.) 두 확률분포가 gaussian 모형이라면, ‘면적을 뺀 절대값의 최대값’을 나타내므로 이는 ‘안겹치는 부분의 면적’과 같다.

따라서 다음과 같이 변형된다.

$$\delta(\mathbb{P}, \mathbb{Q}) = \sup_{A \in \Sigma} |\mathbb{P}(A) - \mathbb{Q}(A)| = \frac{1}{2} |\mathbb{P}(A) - \mathbb{Q}(A)|$$

이 때 Σ 는 \mathbb{P} 와 \mathbb{Q} 가 만나는 부분부터 끝(∞ or $-\infty$) 이다.

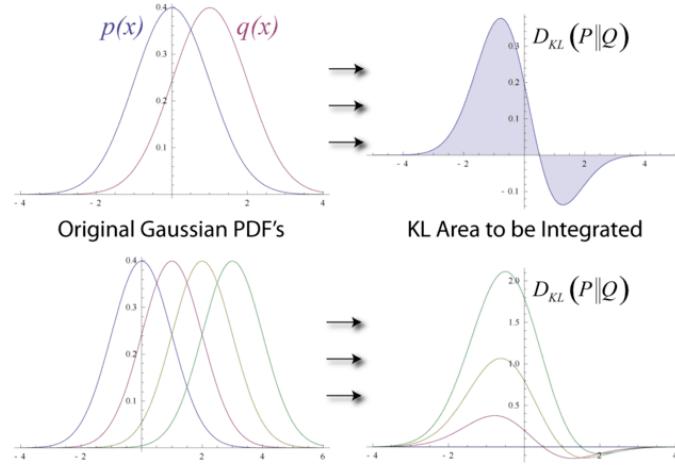
- Kullback-Leibler divergence

$$\begin{aligned} D_{KL}(\mathbb{P} || \mathbb{Q}) &= E_{x \sim \mathbb{P}} \left[\log \frac{P(x)}{Q(x)} \right] \\ &= \int_x P(x) \log \frac{Q(x)}{P(x)} dx \end{aligned}$$

discrete 의 경우에는 entropy form이 가능하다.

$$\begin{aligned} D_{KL}(\mathbb{P} || \mathbb{Q}) &= E_{x \sim \mathbb{P}} \left[\log \frac{P(x)}{Q(x)} \right] \\ &= - \sum_i P(x_i) \log \frac{Q(x_i)}{P(x_i)} \\ &= \sum_i P(x_i) \log P(x_i) - \sum_i P(x_i) \log Q(x_i) \\ &= -H(\mathbb{P}) + H(\mathbb{P}, \mathbb{Q}) \end{aligned}$$

식을 살펴보자. 식에서 $P(x)$ 가 없었다면 $\log \frac{Q(x)}{P(x)}$ 의 꼴이 완전히 대칭적이어서 그 적분이 0 이 될 것이다. $P(x)$ 를 곱해졌기 때문에 대칭상태에서 $P(x)$ 가 클 때의 값을 높여준다. 때문에 KL-divergence는 그 값이 0 이상이다.

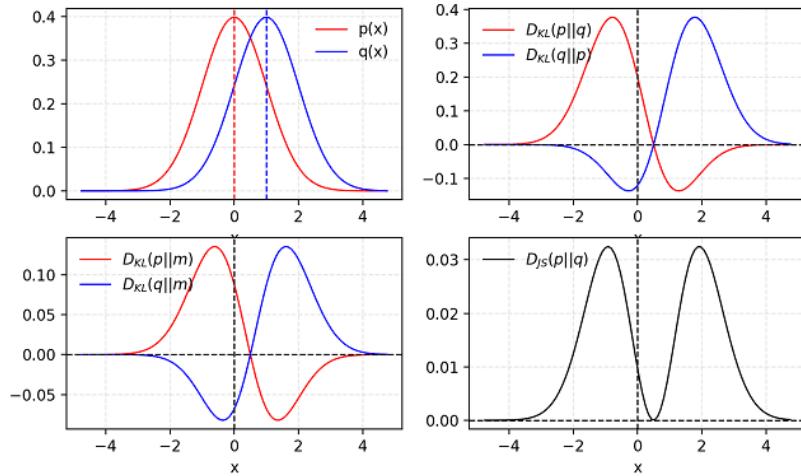


두번째 그림은 다양한 \mathbb{Q} 에 대하여 $P(x) \log \frac{Q(x)}{P(x)}$ 를 그린 것이다. 적분하여 각각의 $D_{KL}(\mathbb{P} || \mathbb{Q})$ 를 얻게 된다.

- Jensen-Shannon divergence

$$D_{JS}(\mathbb{P} || \mathbb{Q}) = \frac{1}{2} D_{KL} \left(\mathbb{P} \middle\| \frac{\mathbb{P} + \mathbb{Q}}{2} \right) + \frac{1}{2} D_{KL} \left(\mathbb{Q} \middle\| \frac{\mathbb{P} + \mathbb{Q}}{2} \right)$$

KL divergence에서 symmetric을 만든 형태이다.



discrete한 경우에 $\mathbb{M} = \frac{\mathbb{P} + \mathbb{Q}}{2}$ 이라 두면 다음과 같이 entropy form으로 변형된다.

$$\begin{aligned} D_{JS}(\mathbb{P}||\mathbb{Q}) &= \frac{1}{2}D_{KL}(\mathbb{P}||\mathbb{M}) + \frac{1}{2}D_{KL}(\mathbb{Q}||\mathbb{M}) \\ &= \frac{1}{2}(-H(\mathbb{P}) + H(\mathbb{P}, \mathbb{M})) + \frac{1}{2}(-H(\mathbb{Q}) + H(\mathbb{Q}, \mathbb{M})) \\ &= -\sum \frac{\mathbb{P}}{2} \log \mathbb{M} - \sum \frac{\mathbb{Q}}{2} \log \mathbb{M} + \frac{1}{2} \left[\sum \mathbb{P} \log \mathbb{P} + \sum \mathbb{Q} \log \mathbb{Q} \right] \\ &= -\sum \mathbb{M} \log \mathbb{M} + \frac{1}{2} \left[\sum \mathbb{P} \log \mathbb{P} + \sum \mathbb{Q} \log \mathbb{Q} \right] \end{aligned}$$

이 때 확률변수 X 를 $X \sim \mathbb{M}$ 라 하고 확률변수 Z 를 \mathbb{P} 와 \mathbb{Q} 의 binary indicator 라고 하면 다음과 같이 mutual information 으로 표현된다.

$$D_{JS}(\mathbb{P}||\mathbb{Q}) = H(X) - H(X|Z) = I(X; Z)$$

KL-divergence는 확률분포간 얼마나 떨어져 있는가를 나타내는 매우 효과적인 도구이며 가장 많이 사용된다. 하지만 \mathbb{P} 와 \mathbb{Q} 가 너무 많이 떨어져있으면 적당히 멀리 떨어져 있을 때와 구분이 안된다. 즉, \mathbb{P} 의 영역에서 \mathbb{Q} 의 꼬리만 있으면, KL-divergence는 \mathbb{Q} 가 더 멀어져도 값이 변하지 않는다. 때문에 \mathbb{Q} 가 너무 많이 떨어져있으면 학습이 너무 느리다. 이 문제를 보정하고자 하는 것이 WGAN 이다.

Wasserstein (Earth-Mover) distance는 다음과 같다.

$$W(\mathbb{P}, \mathbb{Q}) = \inf_{\gamma \in \Pi(\mathbb{P}, \mathbb{Q})} \mathbb{E}_{(x,y) \sim \gamma} [||x - y||]$$

직관적으로, $\gamma(x, y)$ 는 \mathbb{P} 를 \mathbb{Q} 로 옮기기 위해 x 를 얼마나 y 로 변환해야 하는지를 말해준다. 또한 inf 를 붙이면서 그들 중 ‘최소’를 말하게 된다. 이 형태는 L1 norm 이기 때문에 미분하기 어렵기 때문에 논문에서는 사용하기 좋은 형태로 바꾸게 된다.

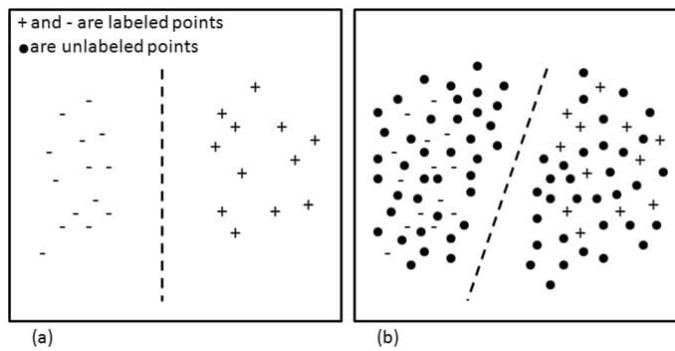
$$\max_{w \in W} \mathbb{E}_{x \sim \mathbb{P}_r} [f_w(x)] - \mathbb{E}_{z \sim p(z)} [f_w(g_\theta(z))]$$

이 때 f_w 는 어떤 K 에 대하여 K -Lipschitz이며 parametrized 되어있고, \mathbb{P}_r 은 data의 distribution, \mathbb{P}_θ 는 generated model의 distribution 이다.

5.6 Semi-supervised GAN

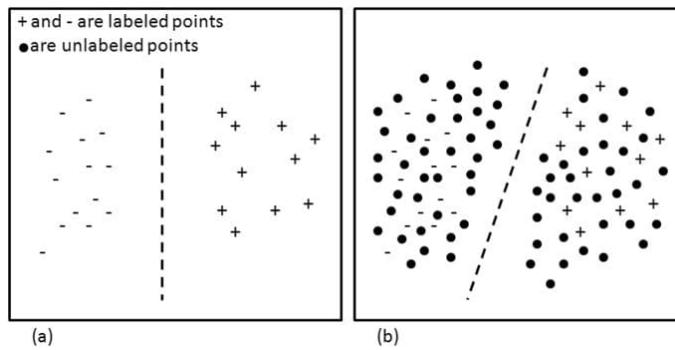
- Supervised Learning : label이 붙은 train dataset (x_i, y_i) 가 들어와 이로부터 학습을 진행한다. 그 뒤 test dataset에서 classification을 진행한다.
- Unsupervised Learning : 모든 dataset (x_i) 에는 label이 붙어있지 않으며 여기서 cluster 들을 찾아냄으로서 패턴을 알아낸다.
- Semi-supervised Learning : label이 일부 있는 dataset을 이용하기 위한 방법이다. 일부 초록색으로 알고 있고 일부 빨간색으로 알고 있다면 clustering 후 다른 것들에 classification을 할 수 있을 것이다.

machine을 만들기 위한 dataset을 가지고 있다고 하자. 그런데 일부는 label이 붙어있고, 일부는 붙어 있지 않다. 이럴 때는 어떻게 해야 할까? 다음 예시를 보자.



label이 붙어있는 데이터는 적기 때문에 이를 사용했을 때 얻을 수 있는 결과는 그다지 좋지 못하다. 때문에 최대한 많은 데이터를 활용하는데, unlabeled data까지 활용하는 방법이 (b)이다. 즉, 먼저 clustering을 하고 label이 붙은 것을 이용해 class를 나눈다.

Semi-supervised GAN (SGAN) 기존 GAN의 discriminator는 real과 fake를 binary classification 하는 모델이었다. 때문에 학습 후 discriminator를 버리고 진짜같은 fake를 만들어내는 generator만 썼다. 여기서는 discriminator를 좋게 만들어 후에 generator를 버리고 discriminator를 사용하는 것이 목적인데, 그 구조가 다음과 같다.



generator가 하는 일은 이전과 같다. 진짜같은 fake data $G(\mathbf{z})$ 를 만드는 것이다. 물론 여기에는 label이 붙어있지 않다. 그리고 real data 와 함께 discriminator에 들어가는데, real data에는 labeled data (\mathbf{x}, \mathbf{y}) 와 unlabeled data \mathbf{x} 가 있다. discriminator는 이들을 받아 ‘real’이라고 판명내릴 경우 ‘class’ 까지 판별해서 결과를 출력하며, ‘fake’ 인 경우는 따로 출력한다. 즉, class가 k 개인 경우 discriminator 가 출력하는 값은 $k + 1$ 개인 것이다.

만약 어떤 값 \mathbf{x} 가 D 로 들어와 결과적으로 $(l_1, l_2, \dots, l_k, l_{k+1})$ 의 값이 나왔다면, D 가 이것을 ‘fake’라고 출력할지, ‘real class i ’라고 출력할지는 다음과 같은 확률에 근거해서 판단한다.

$$p_{\text{model}}(\mathbf{y} = k+1 | \mathbf{x}) = \frac{e^{l_{k+1}}}{\sum_{j=1}^{k+1} e^{l_j}}$$

$$p_{\text{model}}(\mathbf{y} = i | \mathbf{x}) = \frac{e^{l_i}}{\sum_{j=1}^{k+1} e^{l_j}} \quad 0 < i < k+1$$

Objective function은 다음과 같다.

- Discriminator loss

$$L_D = L_{D(\text{supervised})} + L_{D(\text{unsupervised})}$$

where

$$L_{D(\text{supervised})} = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}(\mathbf{x}, \mathbf{y})} \left[\log(p_{\text{model}}(\mathbf{y} | \mathbf{x}, \mathbf{y} < k+1)) \right]$$

$$L_{D(\text{unsupervised})} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \left[\log(1 - p_{\text{model}}(\mathbf{y} = k+1 | \mathbf{x})) \right] - \mathbb{E}_{\mathbf{x} \sim G} \left[\log(p_{\text{model}}(\mathbf{y} = k+1 | \mathbf{x})) \right]$$

$L_{D(\text{supervised})}$ 은 supervised, 즉 이미 label을 알고 있는 상태의 real data이다. discriminator가 원래 label \mathbf{y} 를 맞출 확률이 높을수록 log값의 크기는 작아지며 결과적으로 $L_{D(\text{supervised})}$ 는 작아진다.

$L_{D(\text{unsupervised})}$ 의 첫 항은 unlabeled real data에 대한 식이다. discriminator가 이를 $k+1$, 즉 ‘fake’라고 판별할 확률이 높으면 $1-p$ 는 작아지고, 결국 loss는 커진다.

$L_{D(\text{unsupervised})}$ 의 두번째 항은 fake data에 대한 식이다. discriminator가 이를 $k+1$, 즉 ‘fake’라고 판별할 확률이 높으면 loss는 작아진다.

- Generator loss

$$L_G = L_{G(\text{cross-entropy})} + L_{G(\text{feature matching})}$$

where

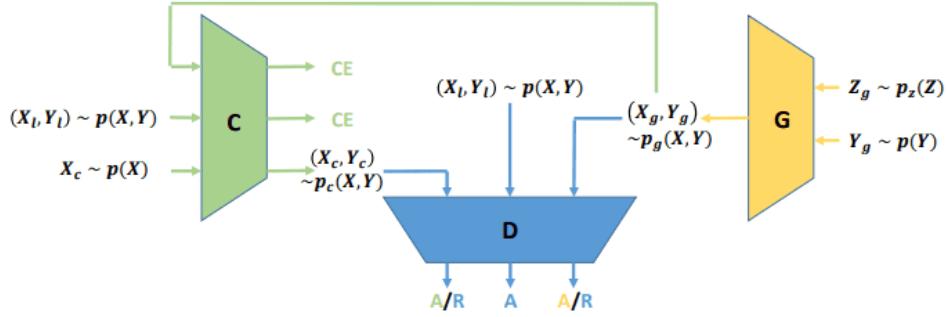
$$L_{G(\text{cross-entropy})} = -\mathbb{E}_{\mathbf{x} \sim G} \left[\log(1 - p_{\text{model}}(\mathbf{y} = k+1 | \mathbf{x})) \right]$$

$$L_{G(\text{feature matching})} = \left\| \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [f(\mathbf{x})] - \mathbb{E}_{\mathbf{x} \sim G} [f(\mathbf{x})] \right\|_2^2$$

$L_{G(\text{cross-entropy})}$ 는 D 를 잘 속이는지에 대한 장치이다. G 에서 나온 fake data \mathbf{x} 가 D 에서 $k+1$, 즉 ‘fake’로 판명될 확률이 높다면 $1-p$ 는 작아지고, 결국 loss는 커진다.

$L_{G(\text{feature matching})}$ 는 real image의 평균값과 generated image의 평균값의 L2 norm 재곱이다. 즉, generator는 discriminator를 잘 속여야 하기도 하지만, real image와 비슷한 분포에서 나와야 한다는 것이다.

Triple GAN SGAN 보다 조금 더 나아가서, generator가 만든 것도 classifier에 넣고 unlabeled도 classifier에 넣어서 classify도 하고 SGAN도 돌리는 구조이다.



여기서 A는 acceptance, R은 rejection, CE는 cross entropy loss for supervised learning 이다. 우선 G 에서 나온 (X_g, Y_g) , C 에서 나온 (X_c, Y_c) , 원래 갖고 있던 (X_l, Y_l) 모두를 D 에 넣고 discriminator를 학습한다. 그 다음은 (X_g, Y_g) , (X_l, Y_l) , 원래 갖고 있던 unlabeled data x_c 를 C 에 넣고 classifier를 학습한다. 이 때 x_c 가 classifier를 거치고 discriminator에서 accept 되었는지 reject 되었는지를 loss에 포함한다. 마지막으로 generator가 discriminator를 잘 속였는지 판별해 generator를 학습한다.

Algorithm 1 Minibatch stochastic gradient descent training of Triple-GAN in SSL.

for number of training iterations **do**

- Sample a batch of pairs $(x_g, y_g) \sim p_g(x, y)$ of size m_g , a batch of pairs $(x_c, y_c) \sim p_c(x, y)$ of size m_c and a batch of labeled data $(x_d, y_d) \sim p(x, y)$ of size m_d .
- Update D by ascending along its stochastic gradient:

$$\nabla_{\theta_d} \left[\frac{1}{m_d} \left(\sum_{(x_d, y_d)} \log D(x_d, y_d) \right) + \frac{\alpha}{m_c} \sum_{(x_c, y_c)} \log(1 - D(x_c, y_c)) + \frac{1 - \alpha}{m_g} \sum_{(x_g, y_g)} \log(1 - D(x_g, y_g)) \right].$$

- Compute the unbiased estimators $\tilde{\mathcal{R}}_{\mathcal{L}}$ and $\tilde{\mathcal{R}}_{\mathcal{P}}$ of $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{P}}$ respectively.
- Update C by descending along its stochastic gradient:

$$\nabla_{\theta_c} \left[\frac{\alpha}{m_c} \sum_{(x_c, y_c)} p_c(y_c|x_c) \log(1 - D(x_c, y_c)) + \tilde{\mathcal{R}}_{\mathcal{L}} + \alpha_{\mathcal{P}} \tilde{\mathcal{R}}_{\mathcal{P}} \right].$$

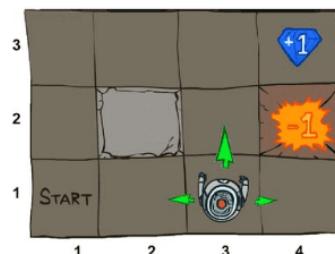
- Update G by descending along its stochastic gradient:

$$\nabla_{\theta_g} \left[\frac{1 - \alpha}{m_g} \sum_{(x_g, y_g)} \log(1 - D(x_g, y_g)) \right].$$

end for

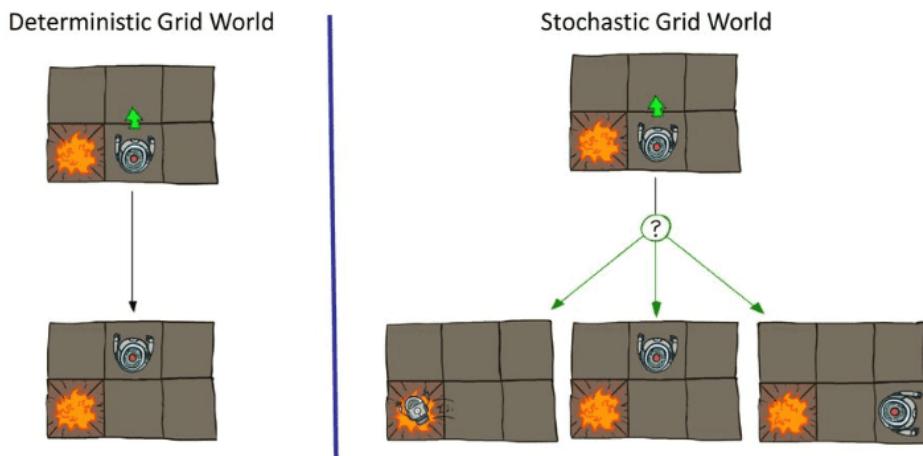
6 Lecture 6 : Deep Reinforcement Learning I : RL

Grid World 강화학습에서 자주 쓰이는 예시이다. 다음과 같은 게임이 있다고 하자.



agent 가 움직이는데, 단계별로 선택되는 각 action 은 계획한대로 되지 않고 확률적으로 움직인다. 이를테면 ‘North’ action 을 선택하면 North(80%), West(10%), East(10%) 의 확률로 움직이며, 움직이려는 경로에 벽이 있으면 움직이지 않는다. 각 action을 취할 때 마다 약간의 negative reward 를 받고, 표시된 곳으로 가면 아주 큰 reward (+1 혹은 -1) 을 받는다. reward -1 부분에서는 agent 가 죽으면서 게임이 끝난다. 우리의 목표는 reward 를 최대화하는 것이다.

Deterministic grid world 에서 agent는 정확히 계획한대로 움직인다. 그러나 위에서 고려한, 그리고 대부분의 경우에 적용되는 Stochastic grid world 에서는 확률적으로 action을 취한다. 목표를 달성하기 위해서는 어떤 방법을 사용해야 할까?



6.1 Markov Decision Process

6.1.1 Markov Property

확률과정론 (stochastic process) 에서 배우는 내용이다. stochastic process 에서 미래 상황에 대한 conditional probability 를 구할 때 현재 상태에만 관련이 있다면 이는 **Markov Property** 를 가지고 있다고 한다. 또한 이것을 **Markov Process** 라고 한다. Markov Process 중 discrete 한 시간 간격을 갖는 것을 **Markov Chain**, 연속적인 시간을 갖는 것을 **Continuous Markov Process** 라고 한다. Markov Chain 에서 Markov Property를 수식으로 정리하면 다음과 같다. (우리는 암묵적으로 time-homogeneous, 즉 시각 t 에 상관없는 모델을 사용한다.)

$$\mathbb{P}[S_{t+1} = s' | S_t = s] = \mathbb{P}[S_{t+1} = s' | S_0 = s_0, S_1 = s_1, \dots, S_t = s]$$

여기서 $\{S_t\}$ 는 Markov Chain을 나타내며, s 는 state space에 있는 state 값이다.

Markov Chain은 state space \mathcal{S} 와 transition probability matrix \mathcal{P} 의 tuple $(\mathcal{S}, \mathcal{P})$ 이다.

- \mathcal{S} : state 의 set
- \mathcal{P} : 어떤 시각 t 에서 state s 인 것이 다음 시각 $t + 1$ 에 어떤 state로 갈지 결정하는 확률을 matrix form 으로 나타낸 것 이다. Markov Chain의 경우는 자기 자신으로 갈 수 있다.

$$\mathcal{P}_{ij} = \mathbb{P}[S_{t+1} = s_j | S_t = s_i]$$

다음은 Sheldon M.Ross 에서 발췌한 예시문제이다.

For states $i, j, k, l \neq j$, let

$$P_{ij/k}^n = P\{X_n = j, X_l \neq k, l = 1, \dots, n-1 | X_0 = i\}.$$

- (a) Explain in words what $P_{ij/k}^n$ represents.
- (b) Prove that, for $i \neq j$, $P_{ij}^n = \sum_{k=0}^n P_{ii}^k P_{ij/i}^{n-k}$.

solution :

- (a) $P_{ij/k}^n$ is the probability of being in j at time n , starting in i at time 0, while avoiding k .
- (b) Let N denote the time at which X_k is last i before time n . Then since $0 \leq N \leq n$,

$$\begin{aligned} P_{ij}^n &= P\{X_n = j | X_0 = i\} \\ &= \sum_{k=0}^n P\{X_n = j, N = k | X_0 = i\} \\ &= \sum_{k=0}^n P\{X_n = j, X_k = i, X_l \neq i : k+1 \leq l \leq n | X_0 = i\} \\ &= \sum_{k=0}^n P\{X_n = j, X_l \neq i : k+1 \leq l \leq n | X_0 = i, X_k = i\} P\{X_k = i | X_0 = i\} \\ &= \sum_{k=0}^n P_{ii}^k P_{ij/i}^{n-k} \end{aligned}$$

6.1.2 Markov Decision Process (MDP)

모든 state가 Markov Property 를 가지는 의사결정의 수학적 모델이다. 이는 다음과 같이 4개의 tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R)$ 로 표현된다. (우리는 \mathcal{S}, \mathcal{A} 가 유한하다고 가정하고 문제를 푼다.)

- \mathcal{S} : state 의 set
- \mathcal{A} : action 의 set
- \mathcal{P} : transition probability matrix. state s 에서 action a 를 선택해 state s' 가 될 확률은 다음과 같다.

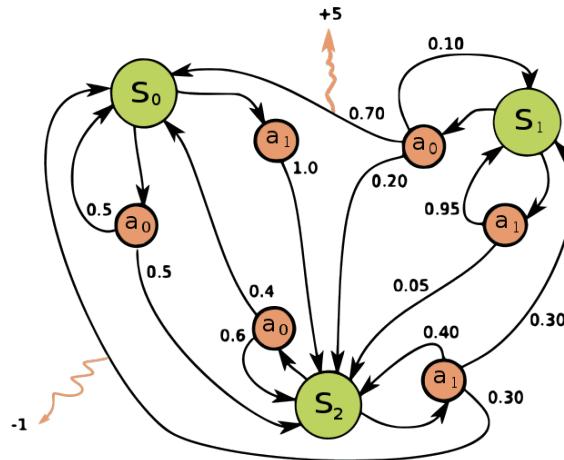
$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] = T(s, a, s')$$

이때 T 는 transition function 이다.

- R : reward function (reward set \mathcal{R})

$R_{ss'}^a$ 는 state s 에서 action a 를 거쳐 state s' 로 갔을 때 얻는 reward 이다. 우리는 reward 가 최대가 되기를 바란다.

다음의 예시를 고려해 보자.



S_0, S_1, S_2 에서 취할 수 있는 action 은 a_0 와 a_1 의 두가지 이다. 즉, $\mathcal{A} = \{a_0, a_1\}$ 이다. S_0 에서 action a_0 를 취하는 경우 다시 S_0 로 돌아올 확률은 0.5, S_2 로 갈 확률은 0.5 이다. 이것이 transition probability 이다. 반면 S_0 에서 action a_1 을 취하면 반드시 S_2 로 간다.

S_1 에서 action a_0 를 취하면 0.7의 확률로 S_0 로 가게 되며 reward +5 를 받는다. 반면 S_2 에서 action a_1 을 취하면 0.3 의 확률로 S_0 로 가며 reward -1 을 받는다.

위에서 언급한 notation을 사용해 보자. $P_{s_2 s_1}^{a_1} = 0.3$ 이며 $R_{s_1 s_0}^{a_0} = +5$ 라는 것 등을 알 수 있다. 단원의 처음 부분에 언급했던 grid world 의 MDP 는 다음과 같다.

- State set

$$\mathcal{S} = \{(1, 1), (1, 2), (1, 3), \dots, (4, 2), (4, 3)\}$$

- Action set

$$\mathcal{A} = \{\text{north}, \text{south}, \text{east}, \text{west}\}$$

- State transition probability

$$P_{(3,1)(3,2)}^{\text{north}} = 0.8$$

$$P_{(3,1)(2,1)}^{\text{north}} = 0.1$$

$$P_{(3,1)(4,1)}^{\text{north}} = 0.1$$

⋮

- Reward

$$R_{(3,3)(4,3)}^{\text{east}} = +1$$

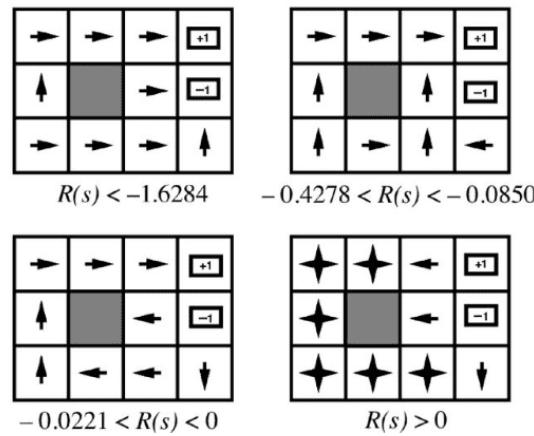
$$R_{(3,2)(4,2)}^{\text{north}} = -1$$

$$R_{(4,1)(4,2)}^{\text{north}} = -1$$

$$R_s^a = c \quad \text{for the others}$$

이 때 c 는 작은 음수값이다. 잘 보면 ‘north’ 라는 action 을 취하고자 했음에도 ‘east’ 로 가는 확률은 존재한다. 즉, stochastic grid world 이다.

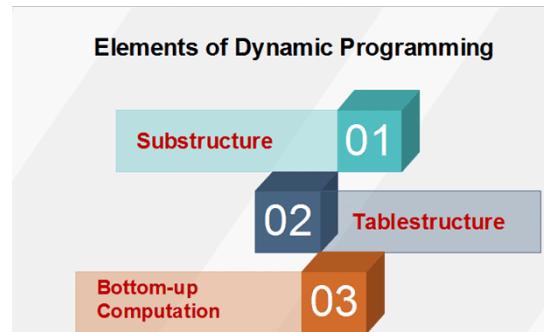
우리는 각 state에서 어떤 action을 선택할지에 대한 최적의 가이드라인을 찾고 싶다. 즉, 각 state에서 어떤 action을 선택할지에 대한 policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ 중, 우리의 목적에 맞는 (return을 최대로 하는) 최적화된 policy $\pi_* : \mathcal{S} \rightarrow \mathcal{A}$, 즉 **Optimal Policy**를 찾고 싶다. 다양한 c 값에 따른 최종 결과는 다음과 같다.



좌상단의 경우 한번 움직일 때마다의 reward가 음으로 너무 크기 때문에 게임을 최대한 빨리 끝내는 것이 손해가 적다. 우상단의 경우 최적의 경우이며, 좌하단은 c 의 크기가 매우 작아 오래 움직이더라도 안전한 방향으로 action을 정하게 된다. (3,2)를 보면 흥미로운데, north action을 취할 경우 -1로 빠질 확률이 존재하기 때문에 벽이 존재함에도 불구하고 west action을 취하게 된다.

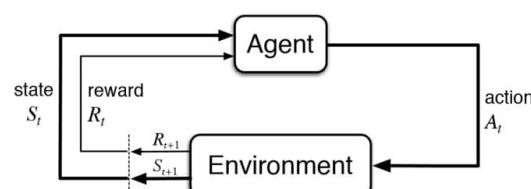
이러한 Optimal Policy는 어떻게 찾을까? 우리가 여기서 살펴볼 solution method는 다음과 같이 두개이다.

- MDP를 알고 있을 경우, 즉 완벽한 정보가 갖춰져 있을 경우 **Dynamic Programming (DP)**를 사용한다.



하지만 state가 백만개 넘게 늘어날 경우 계산이 매우 느리다.

- MDP를 모를 경우, 즉 불완전한 정보만이 있을 경우 **Reinforcement Learning (RL)**을 사용한다.



agent에게 명령(action)을 줄 수는 있지만 그 결과가 정확히 어떨지는 계산할 수가 없다. 이 때는 많은 sample 들에 대해 실험하여 경험적으로 알아볼 수밖에 없다.

MDP에서 $p(s', r|s, a)$ 를 안다는 것은 MDP를 다 아는 것이다. 모든 transition (s, a, s', r) 에 대하여 $p(s', r|s, a)$ (state s 에서 action a 를 선택해 s' 으로 가며 reward r 을 받을 확률)를 아는 경우 이를 이용해 다음과 같은 계산이 가능하다.

- state-transition probability

$$P_{ss'}^a = p(s'|s, a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

- expected reward for state-action pair

$$R_s^a = r(s, a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} \left[r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \right]$$

- expected reward for state-action-next.state triple

$$R_{ss'}^a = r(s, a, s') = \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{r \in \mathcal{R}} rp(s', r | s, a)}{p(s'|s, a)}$$

Reward R_t 는 agent가 step t 에서 얼마나 잘 했는지를 표시해주는 feedback 신호이다. agent의 일은 reward의 최종합을 크게 만들어주는 것이며, RL의 경우 다음을 가정한다.

Reward Hypothesis

All goals can be described by the maximization of the expected value of the cumulative sum of rewards

예시로, Packman은 매 순간, Go game은 게임이 끝나고, Ping-pong은 점수가 날 때마다 reward가 주어진다.

Return G_t 는 step t 로부터 그 후에 남은 reward를 지수적으로 낮춰놓은 값이다. (discounted)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$$

이 때 **Discount** γ 는 $[0, 1]$ 구간의 값을 가지는데, 이를 곱하는 이유는 미래가 어떻게 될지 불확실하기 때문이다. 만약 0에 가까우면 근시안적이고 1에 가까우면 먼 곳까지 예측하게 된다. 많은 MDP는 이와같이 discounted 되어있는데, 그 이유는 다음과 같다.

- 계산의 편의성을 위해 (return이 무한대가 되는것을 방지하기 위해)
- 미래에 대한 불확실성 때문에 (따라서 exponentially decay 하도록 설계)
- 실제로, 지금 바로 얻는 reward가 나중에 올 reward 보다 영향력이 크다.

가끔은 undiscounted 를 사용하기도 한다. step이 그리 길지 않은 경우에 해당한다.

Policy π 는 일반적으로 주어진 state에서 action의 probability distribution 이다. 즉, state s 에 대하여 policy π 를 따를 때 하나의 정해진 action이 있는 것이 아닌, stochastic하게 action을 정하는 경우 다음과 같다.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

π 는 agent가 특정 state s 에서 어떤 최적의 action a 를 취해 total reward 가 커지도록 한다. MDP policy는 markov이기 때문에 과거의 state와는 상관이 없으며, 현재 state만이 중요하다. Deterministic Policy, 즉 state s 에 대해 policy π 를 따를 때 하나의 a 가 대응될 경우는 $\pi(s) = a$ 이다.

Stochastic/Deterministic Policy는 action을 취하는 방법을 나타낸 것으로, 특정 action을 취했을 때 어떤 방식으로 다음 state가 결정되는지를 관할하는 Stochastic/Deterministic MDP와는 관련이 없다.

MDP를 완전히 알고 있으면, deterministic optimal policy가 존재한다. (즉, 특정 state에서 어떤 action을 취해야 하는지에 대한 명확한 답이 있다.) 반면, MDP를 모른다면 ϵ -greedy policy (stochastic policy) 방법이 필요하다. ($1-\epsilon$: choose the optimal value, ϵ : choose randomly) 물론, optimal policy는 deterministic이겠지만, 기본적으로 모든 policy는 stochastic이라고 가정한다. 이에 대해서는 후에 자세히 설명한다.

State-value and Action-value functions Value function은 특정 state s 에서 policy π 를 따랐을 때 return의 기대값을 예측한다. 그것이 state s 일 때 부터를 기준으로 하는지, state s 에서 action a 를 따랐을 때 부터를 기준으로 하는지에 따라 State-value function과 Action-value function으로 나뉜다.

- **State-value function** $v_\pi(s)$ 는 state s 에서 시작해 policy π 를 따랐을 때 기대되는 return 값이다.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

- **Action-value function (quality)** $q_\pi(s, a)$ 는 state s 에서 action a 를 취한 후 policy π 를 따랐을 때 기대되는 return 값이다.

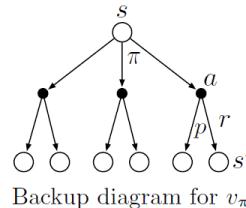
$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

- Action-value function은 State-value function의 기준에서 action a 만 결정되었을 뿐이다. 따라서 State-value function과 Action-value function은 다음의 관계를 가진다. 즉, state s 에서 state-value function은 s 에서 가능한 모든 action a 에 대한 action-value function들의 평균이다.

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a)$$

proof :

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \sum_a \mathbb{P}[A_t = a|S_t = s] \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \sum_a \pi(a|s) q_\pi(s, a) \end{aligned}$$



이 때 검은 점은 action, 흰 점은 state이다. 위의 식에 따르면 action-value function 을 알 때 state-value function 도 계산할 수 있으므로 action-value function $q_\pi(s, a)$ 을 먼저 알아내야 할 것 같다. 하지만 DP 에서는 계산량 때문에 이것을 사용할 수 없다. 이는 뒤에서 설명한다.

- Advantage function $A_\pi(s, a)$ 는 위 두 함수의 차이다.

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s)$$

즉, action a 를 골랐을 때의 return^o (s 에서 갈 수 있는 모든 action 각각에 대한 return^o) 평균보다 더 좋은가를 나타내는 function 이다.

6.2 Bellman Equation

6.2.1 Bellman Expectation Equation

Bellman Expectation Equation 1 : state-value function

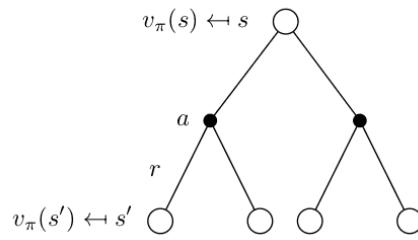
$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s', r} [r + \gamma v_\pi(s')] p(s', r|s, a) \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

즉, State-value function $v_\pi(s)$ 을 immediate reward R_{t+1} 과 discounted successor state-value $\gamma v_\pi(S_{t+1})$ 로 분리하는 recursive equation 이다.

proof :

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \sum_a \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \mathbb{P}[A_t = a | S_t = s] \\ &= \sum_a \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \pi(a|s) \\ &= \sum_a \pi(a|s) \sum_{s', r} \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \\ &\quad \times \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a] \\ &= \sum_a \pi(a|s) \sum_{s', r} \left[\mathbb{E}_\pi[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \right. \\ &\quad \left. + \gamma \mathbb{E}_\pi[G_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \right] p(s', r|s, a) \\ &= \sum_a \pi(a|s) \sum_{s', r} [r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']] p(s', r|s, a) \\ &= \sum_a \pi(a|s) \sum_{s', r} [r + \gamma v_\pi(s')] p(s', r|s, a) \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

위의 증명을 시각화 하면 다음과 같이 볼 수도 있다.



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

한편, Action-value function $q_\pi(s, a)$ 에도 비슷한 논의가 적용된다.

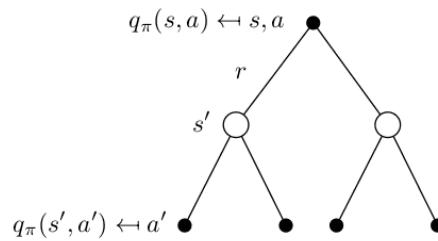
Bellman Expectation Equation 2 : action-value function

$$\begin{aligned} q_\pi(s, a) &= \sum_{s', r} p(s', r|s, a) [r + \gamma \sum_{a'} q_\pi(s', a') \pi(s', a')] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

proof :

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s'} \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \\ &\quad \times \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a] \\ &= \sum_{s'} \left[\mathbb{E}_\pi[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \right. \\ &\quad \left. + \gamma \mathbb{E}_\pi[G_{t+1} | S_t = s, A_t = a, S_{t+1} = s', R_{t+1} = r] \right] p(s', r | s, a) \\ &= \sum_{s'} \left[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \right] p(s', r | s, a) \\ &= \sum_{s'} p(s', r | s, a) \left[r + \gamma \sum_{a'} \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s', A_{t+1} = a'] \mathbb{P}[A_{t+1} = a' | S_{t+1} = s'] \right] \\ &= \sum_{s'} p(s', r | s, a) \left[r + \gamma \sum_{a'} q_\pi(s', a') \pi(a' | s') \right] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

위의 증명을 시각화 하면 다음과 같이 볼 수도 있다.



$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

6.2.2 Finding Optimal Policy

Optimal Value functions and Policy MDP 가 ‘풀렸다’ 라고 말할 수 있을 때는 언제인가? 이는 Optimal Value function 을 구해졌을 때이다. Optimal Value function 은 MDP 의 가장 좋은 성능을 특정한다.

- Optimal state-value function

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

- Optimal action-value function

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

Policy는 원래 order가 없지만, 다음과 같이 order를 정의해보자.

$$\pi' \geq \pi \quad \text{if} \quad v_{\pi'}(s) \geq v_{\pi}(s) \quad \forall s$$

그러면 다음의 Theorem을 만들 수 있다.

Any MDP satisfies the followings :

- There exists an **optimal policy** $\pi_* \geq \pi$ for all π
- All optimal policies achieve the optimal state-value function $v_{\pi_*}(s) = v_*(s)$
- All optimal policies achieve the optimal action-value function $q_{\pi_*}(s, a) = q_*(s, a)$

위의 theorem을 이용하면, action-value function 을 이용해 optimal policy를 찾을 수 있다.

Finding an optimal policy

Any optimal policy can be found by maximizing over $q_*(s, a)$:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

If we know $q_*(s, a)$, we immediately have the optimal policy:

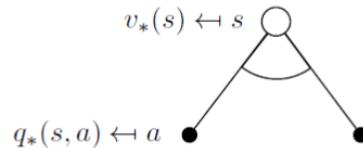
$$\pi_*(s) = \operatorname{argmax}_a q_*(s, a)$$

이 때 $\operatorname{argmax}_a f(a)$ 란 $f(a)$ 를 최대로 만드는 a 라는 뜻이다. 또한, state-value function은 다음과 같이 구할 수 있다.

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s, a)$$

proof :

$$\begin{aligned} v_*(s) &= v_{\pi_*}(s) \\ &= \sum_a \pi_*(a|s) q_{\pi_*}(s, a) \\ &= \max_{a \in \mathcal{A}(s)} q_*(s, a) \end{aligned}$$



6.2.3 Bellman Optimality Equation

위의 식과 Bellman expectation equation 을 사용하면 $v_*(s)$ 를 얻는다.

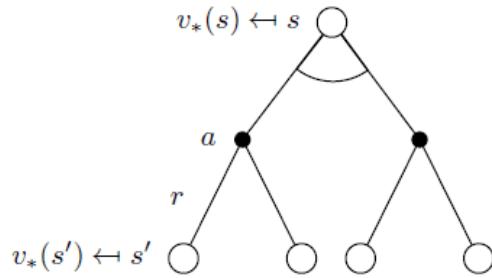
Bellman Optimality Equation 1 : state value function

$$v_*(s) = \max_a [R_s^a + \gamma \sum_{s'} P_{ss'}^a v_*(s')]$$

proof :

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a) \\ &= \max_a \left[R_s^a + \gamma \sum_{s'} p(s'|s, a) \sum_{a'} q_{\pi_*}(s', a') \pi_*(a'|s') \right] \\ &= \max_a \left[R_s^a + \gamma \sum_{s'} p(s'|s, a) \max_{a'} q_*(s', a') \right] \\ &= \max_a \left[R_s^a + \gamma \sum_{s'} P_{ss'}^a v_*(s') \right] \end{aligned}$$

위의 증명을 시각화 하면 다음과 같이 볼 수도 있다.



마찬가지 방법으로 $q_*(s, a)$ 를 얻는다.

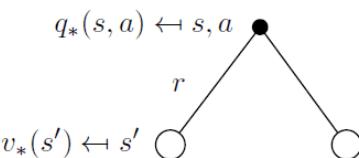
Bellman Optimality Equation 2 : action value function

$$q_*(s, a) = R_s^a + \gamma \sum_{s'} P_{ss'}^a \max_{a'} q_*(s', a')$$

proof :

$$\begin{aligned} q_*(s, a) &= R_s^a + \gamma \sum_{s'} p(s'|s, a) \sum_{a'} q_{\pi_*}(s', a') \pi_*(a'|s') \\ &= R_s^a + \gamma \sum_{s'} p(s'|s, a) \max_{a'} q_*(s', a') \end{aligned}$$

위의 증명을 시각화 하면 다음과 같이 볼 수도 있다.



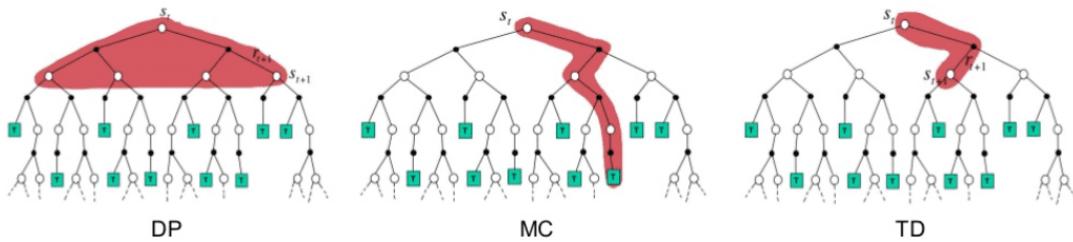
MDP 를 정확히 알고 있으면, 즉 $p(s', r|s, a)$ 를 알고 있으면 Dynamic Programming 을 통해 iteration 을 돌려 위의 식들을 계산할 수 있다. 정리하자면 다음과 같다.

Summary for Dynamic Programming

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a) \\ q_*(s, a) &= r(s, a) + \gamma \sum_{s'} p(s'|s, a) v_*(s') \\ \pi_*(s) &= \operatorname{argmax}_a q_*(s, a) \end{aligned}$$

MDP 를 모르는 RL 의 경우는 $v_*(s')$ 를 찾아도 $q_*(s, a)$ 를 못찾으므로 $q_*(s, a)$ 를 우선적으로 찾는다.

Backup future state value 를 이용해 state value 를 update 하는 것을 말한다. **full backup** 은 다음 세션을 모두 계산하는 반면 **sample backup** 은 sample에 대해서만 이를 시행한다. DP 는 ‘full backup’ 을 사용하며, MC (Monte Carlo) 나 TD (Temporal Difference) 같은 RL 은 ‘sample backup’ 을 사용한다. MC 는 return 을 사용하기 때문에 끝까지 가고, TD 는 몇 개의 step 만 시행하여 학습한다.



6.3 Dynamic Programming

Dynamic Programming (DP) 1953년, Bellman 이 명명한 방법이다. 이것이 사용하는 method 들은 기본적으로 다음 3가지 이다:

1. **Substructure** : 복잡한 문제를 간단한 여러 개의 문제로 나누어 푼다.
2. **Table Structure** : 각 하위 문제의 답을 계산한 뒤, 그 답을 저장하여 후에 같은 문제가 나왔을 경우 불러와서 사용한다. 이러한 방법으로 계산 회수를 줄일 수 있다.
3. **Bottom-up Computation** : 문제를 여러 개의 하위 문제 (substructure) 로 나누어 푼 다음, 그것들을 결합하여 최종 목표에 도달한다.

위와 같은 method 들을 사용하기 때문에, DP 는 그 사용에 제한이 있다. DP 는 다음과 같은 특징을 가진 문제들에서만 사용될 수 있다:

- **Optimal Substructure** : 최종 최적해는 그 하위 문제들의 최적해를 이용해 구할 수 있다.
이를테면, 최단거리 문제의 경우 A 에서 C 까지 가는데 B 를 거치지 않아도 된다고 하면, AB 의 거리와 BC 의 거리를 통해 AC 최단거리를 구할 수 없으므로 해당이 안된다. 물론 반드시 B를 거쳐야 하는 문제였다면 해당이 될 것이다.
- **Overlapping Sub-problems** : 같은 하위 문제의 최적해는 다시 계산할 필요 없이 저장 후 계속 사용할 수 있다.
만약 하위 문제가 계속해서 바뀌거나, 공통적인 하위 문제를 발견할 수 없다면 이 방법을 사용할 수 없다.

MDP 문제는 두 조건 모두 만족시킨다. 따라서 고전적으로 MDP 문제에 Dynamic Programming 을 적용했다.

6.3.1 Value Iteration

Bellman Optimal equation을 상기해 보자.

$$v_*(s) = \max_a \sum_{s'} p(s'|s, a) [r + \gamma v_*(s')]$$

이를 이용하여 state value function 을 업데이트 할 것이다. Iteration 은 다음의 과정을 거친다. V 는 지속적으로 업데이트 되는 state value function 계산값이다.

1. 모든 s 에 대하여 $V_0(s) = 0$ 초기화를 한다. (혹은 random initialize 할 수도 있다.)

2. 임의의 step k 에 대하여 state 가 s 일 때 가능한 모든 action a 에 대해

$$\sum_{s'} P_{ss'}^a [r + \gamma V_k(s')]$$

를 계산하고 최대값을 찾아 $V_{k+1}(s)$ 로 업데이트 한다 :

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a [r + \gamma V_k(s')]$$

이 작업을 $V(s)$ 가 수렴할 때 까지 한다. 즉, $V_*(s)$ 가 되도록 한다. 수치해석학을 배워본 적이 없다면 이 방법이 어떻게 해서 최적해를 구할 수 있는 것인지 의아해 할 수 있다. 물론, $P_{ss'}^a$ 를 모두 알고 있으므로 Bellman equation 을 모든 state 와 action 에 대해 만들 수 있고 이를 연립하여 풀 수도 있다. 하지만 이는 매우 비효율적인 방법이다. Gauss elimination 을 사용한다 해도 모든 parameter 개수의 3승에 달하는 계산을 해야 하기 때문이다.

이러한 recursive 업데이트 작업은 두가지 방식 중 하나를 선택하여 진행한다.

- **synchronous backups** : 모든 s 에 대해 $V_{k+1}(s)$ 를 계산하고 한번에 업데이트 한다. 즉, $k + 1$ 단계에서는 k 단계의 값만 사용한다.
- **asynchronous backups** : 하나의 s 에 대해 $V_{k+1}(s)$ 를 계산하고 하나씩 업데이트 한다. 이렇게 하면 먼저 업데이트 된 다른 state 값들을 가져다가 사용할 수 있다. 많은 경우 이렇게 하면 수렴이 빨라진다.

3. Optimal policy π_* 를 계산한다.

$$\begin{aligned} \pi_*(s) &= \operatorname{argmax}_a q_*(s, a) \\ &= \operatorname{argmax}_a \sum_{s'} p(s'|s, a) [r + \gamma \max_{a'} q_*(s', a')] \\ &= \operatorname{argmax}_a \sum_{s'} p(s'|s, a) [r + \gamma V_*(s')] \end{aligned}$$

다음은 위 과정을 나타낸 pseudo code 이다.

```

Initialize  $V$  arbitrarily, e.g.,  $V(s) = 0$ , for all  $s \in \mathcal{S}^+$ 

Repeat
   $\Delta \leftarrow 0$ 
  For each  $s \in \mathcal{S}$ :
     $v \leftarrow V(s)$ 
     $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$ 
     $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
  until  $\Delta < \theta$  (a small positive number)

Output a deterministic policy,  $\pi$ , such that
 $\pi(s) = \operatorname{argmax}_a \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$ 

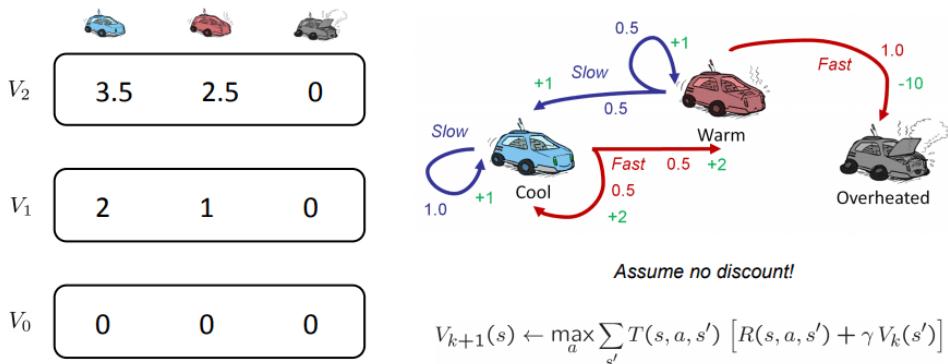
```

이 계산방법은 DP 에서 가장 간단하다. 모든 discounted finite MDP 에 대해서 수렴성이 보장되지만 몇 가지 단점과 주의할 점이 있다.

- best action 이 나왔는데도 불구하고 iteration 은 계속 진행될 수 있다. 즉, policy 는 이미 optimal 하지만 value function 이 수렴하는 것을 찾기 위해 계속해서 진행될 수가 있다.
- $O(S^2A)$ per iteration 으로 매우 느리다.
- ‘convergence’ 는 $|V_{k+1}(s) - V_k(s)| < \epsilon$ 이다.

1. Example : A robot car with 3 states, 2 actions

- 3 states : cool, warm, overheated
- 2 actions : slow, fast
- going faster, gets double reward



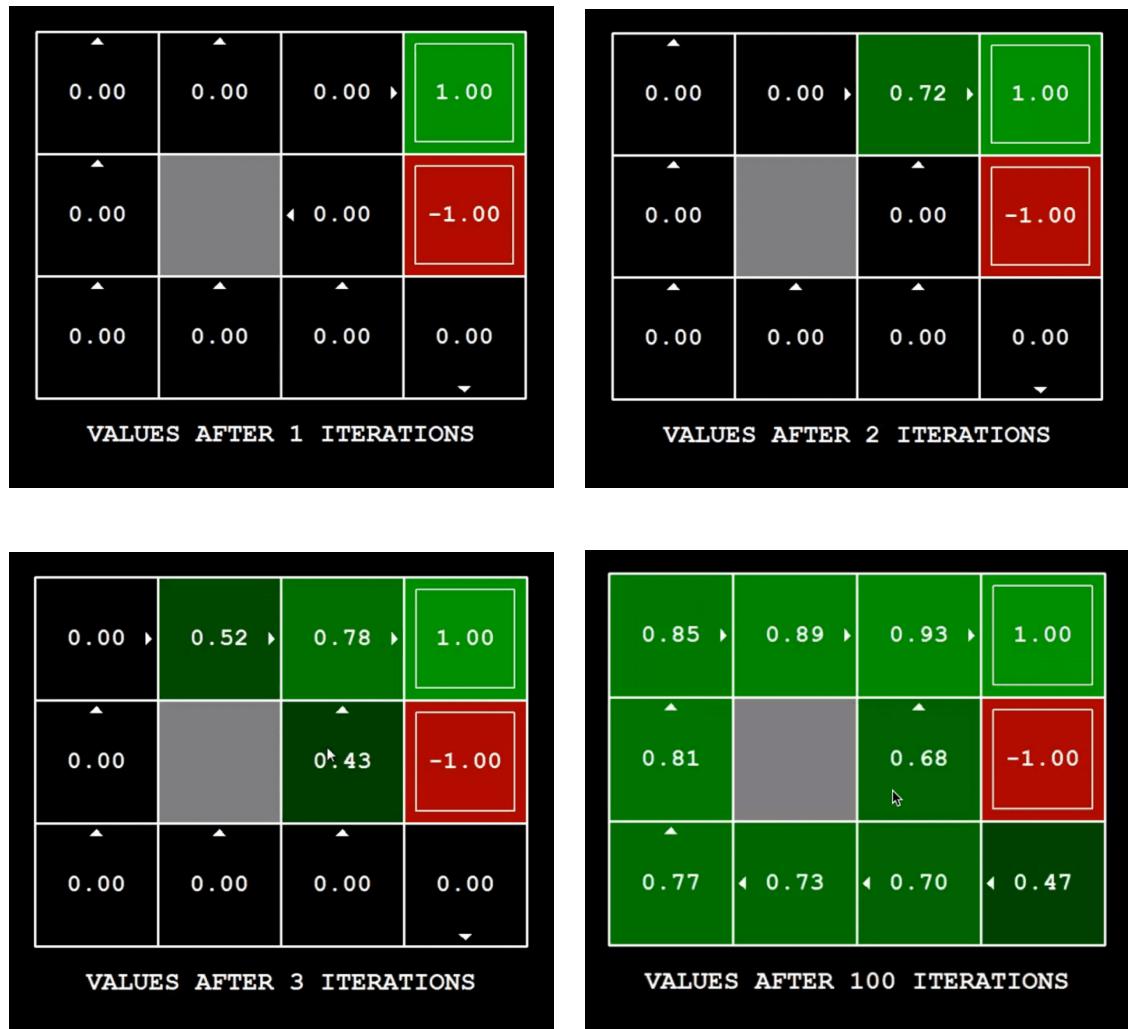
V_0 는 초기화된 값들이다. cool 상태에서 시작할 경우 fast를 선택하는 것이 slow보다 maximize 하므로 $V_1(\text{cool})=0.5 \times 2 + 0.5 \times 2$ 이다. warm 상태에서 시작할 경우는 slow를 선택하는 것이 maximize 하므로 $V_1(\text{warm})=0.5 \times 1 + 0.5 \times 1$ 이다.

V_1 에서 V_2 로 가는 업데이트를 보자. warm 상태에서 slow를 선택할 경우 value는 $0.5 \times (1+2) + 0.5 \times (1+1)$ 이며 fast를 선택할 경우 value는 $1 \times (-10+0)$ 이다. 따라서 $V_1(\text{warm})=2.5$ 이다.

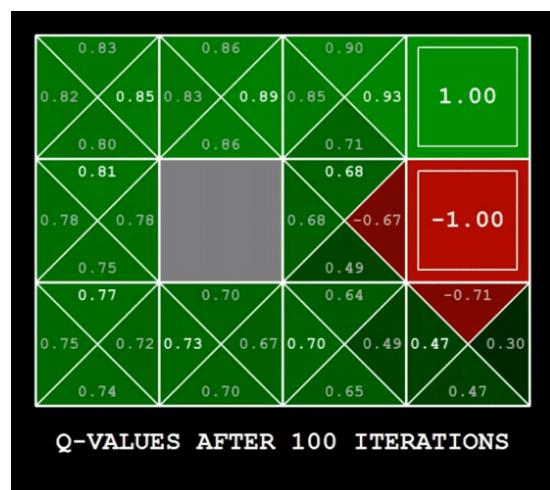
이와 같이, 업데이트 과정에서는 표의 하단 정보가 모두 있어야 한다. 계속해서 하위문제의 답을 저장하고 불러다가 사용하기 때문이다. 또한 위에서 사용한 방법은 synchronous backup인데, 만약 $V_1(\text{warm})$ 을 계산할 때 $V(\text{cool})$ 의 값으로 $V_0(\text{cool})$ 이 아닌 $V_1(\text{cool})$ 을 사용했다면 asynchronous backup 이다.

2. Example : Grid world

앞서 소개한 Grid world 를 value iteration 한 결과이다.



하지만 위의 표는 state-value function 값을 나타낸 것으로, 정확히 어떤 action 을 취해야 하는지 알 수 없다. 때문에 이를 이용해 Q-value 를 계산하고 이에 따라 policy 를 결정한다.



이 정보를 갖고 있다면 어떤 action 을 취해야 하는지 알 수 있다. 때문에 최종적으로 Q-table을 얻는 것이 중요하다.

6.3.2 Policy Iteration

Value Iteration 이 느린 것을 보완하기 위해, policy π 를 바꾸면서 지속적으로 state value $V_{k+1}(s)$ 를 업데이트 한다. 그 과정은 다음과 같다.

1. 모든 s 에 대하여 $V_0(s) = 0$ 초기화를 한다. (혹은 random initialize를 할 수도 있다.) 또한 policy π 도 deterministic policy 로 적절히 초기화 한다.
2. **Policy Evaluation (PE)** 미리 정한 deterministic policy π 에 대해 다음과 같이 업데이트를 진행한다.

$$V_{k+1}(s) = \sum_{s'} p(s'|s, \pi(s)) [r + \gamma V_k(s')]$$

이 업데이트는 적당히 수렴할 만큼 진행한다. 수렴한 결과를 $V_\pi(s)$ 라고 하자. 즉, policy π 에 맞춰 수렴한 state value 이다.

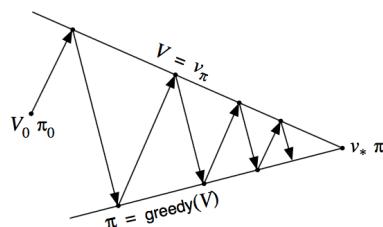
3. **Policy Improvement (PI)** : V_π 를 이용하여 π 를 π' 으로 업데이트 한다.

$$\pi'(s) = \operatorname{argmax}_a \sum_{s'} p(s'|s, a) [r + \gamma V_\pi(s')]$$

4. 이 방식으로 iteration 을 진행하면,

$$q_\pi(s, \pi'(s)) \geq V_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a)$$

이기 때문에 π' 은 계속해서 π 보다 좋아진다. 만약 $\pi = \pi'$ 이라면 optimal에 도달한 것 이다. 또한 유한한 MDP의 경우 유한한 policy가 존재하는데, 따라서 유한한 iteration 안에 optimal에 도달할 것을 기대할 수 있다.



```

Initialize a policy  $\pi'$  arbitrarily
Repeat
   $\pi \leftarrow \pi'$ 
  Compute the values using  $\pi$  by
    solving the linear equations
     $V^\pi(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^\pi(s')$ 
  Improve the policy at each state
   $\pi'(s) \leftarrow \operatorname{argmax}_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s'))$ 
Until  $\pi = \pi'$ 

```

Policy Improvement Theorem

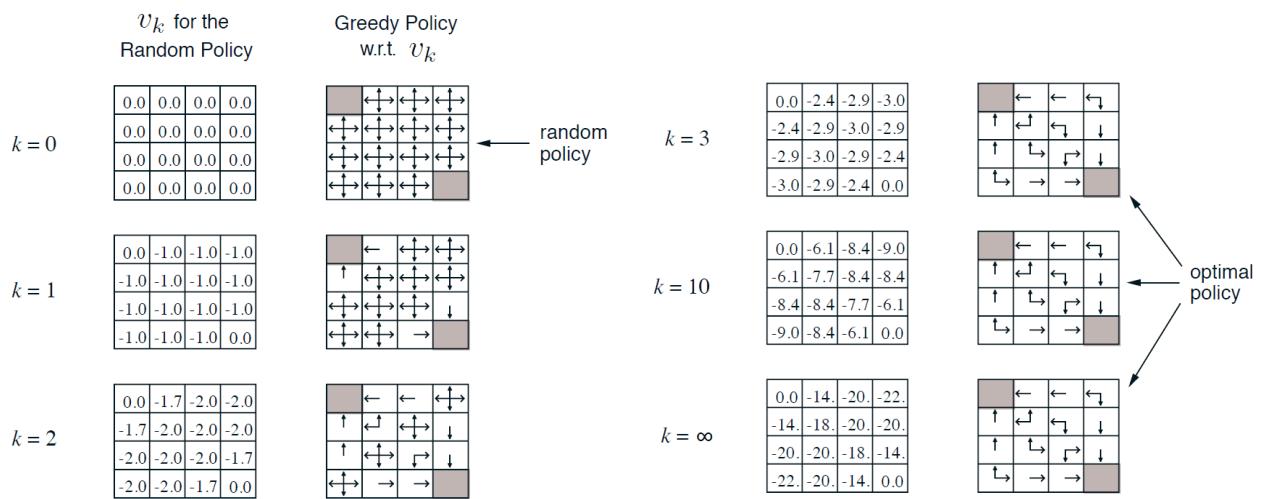
Let π and π' be two policies.

If $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ for all $s \in \mathcal{S}$, then $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$.

proof :

$$\begin{aligned}
 v_{\pi}(s) &\leq q_{\pi}(s, \pi'(s)) \\
 &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) | S_t = s] \\
 &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, \pi'(S_t + 1)) | S_t = s] \\
 &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_{\pi}(S_{t+2})] | S_t = s] \\
 &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_{\pi}(S_{t+2}) | S_t = s] \\
 &\vdots \\
 &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s] \\
 &= v_{\pi'}(s)
 \end{aligned}$$

Example : small grid world

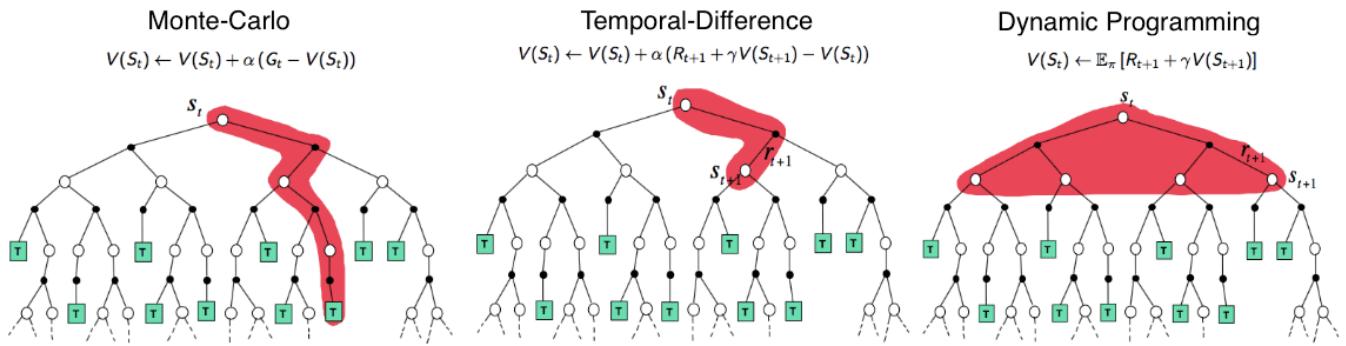


policy 는 이미 $k = 3$ 부터 변하지 않는다. 즉, iteration 3 회 만에 optimal policy 를 찾은 것이다.

6.4 Introduction to Reinforcement Learning

RL 은 deep learning 은 아니다. computer science (cs) 분야에서 고전적으로 사용하던 기법 중 하나일 뿐이다. 이전에도 언급했듯이, DP 와 RL 의 차이는 다음과 같다.

- DP : MDP 를 알고 있을 때 value function table 을 **full backup** 하여 iteration 한다. state 개수가 몇 백만 정도일 때 까지는 잘 작동한다. 하지만 더 큰 수의 state 가 있으면 이 방법으로 풀 수가 없다. 또한 $q(s, a)$ 를 구하는 것이 직관적으로 좋지만 각 state 에 대한 action 을 고려하여 더 많은 계산을 해야 하기 때문에 $v(s)$ 를 통해 계산을 진행한다. ($|{(s, a)}| \gg |s|$)
- RL : MDP 를 모를 때 **sample backup** 을 사용하여 Bellman optimal equation 을 근사적으로 푸다. 즉, sample 에 나타난 value function 들에 대해서만 계산을 진행한다. 따라서 sample 의 크기 정도의 계산만을 진행하며 state 가 늘어나도 마찬가지이다. 물론 이 때문에 각 state 가 몇 번 업데이트 되었는가는 서로 다르다. 여기에는 MC, TD (Sarsa, Q-learning) 등이 있다.



Bellman equation이 다음과 같음을 다시 한 번 상기하자.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

즉, DP 는 MDP 에 대한 모든 정보를 바탕으로 기대값을 직접 계산할 수 있으나, RL 에서는 이것을 할 수 없기 때문에 각 sample 에 대한 값으로 업데이트를 진행한다. MC 에서는 G_t 가 target 이며 (첫째줄), TD 에서는 $R_{t+1} + \gamma v_\pi(S_{t+1})$ 가 target 이다. (둘째줄)

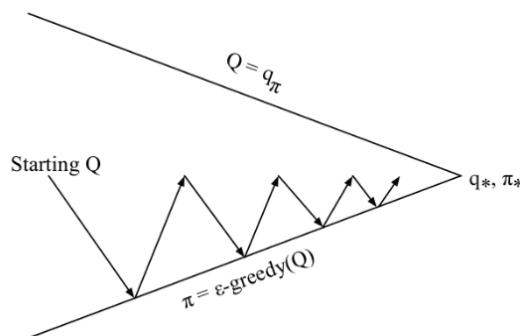
DP 와 RL 모두 Bellman equation 을 사용한 **tabular updating method** (각 state에 대해 V_k 를 업데이트)이며 이를 반복적으로 수행해 optimal policy 를 찾아나간다. DP 의 경우 한번의 iteration에서 full backup 을 사용해 모든 값을 업데이트 한다. 따라서 계산량이 많은 q 보다는 v 를 사용하는 것이다.

반면 RL 에서는 한번의 iteration에서 sample 을 뽑아 해당 sample 에 나타나는 값들만 업데이트 한다. 또한 여기에서 v 는 optimal policy 계산에 적합하지 않기 때문에 q 를 사용한다.

$$\pi_*(s) = \operatorname{argmax}_a q_*(s, a)$$

DP 는 state 가 늘어날 수록 state variable 수가 지수적으로 증가해 curse of dimensionality 문제가 있었다면, RL 은 sample backup 을 사용하기 때문에 일정한 량의 계산만 진행하면 된다.

Generalized Policy Iteration (GPI) 기존의 Policy Iteration 의 경우 정해진 policy 에 대해 value function 을 모두 계산하고 (PE) 이를 이용해 다음 policy 를 계산 (PI) 했다. 이 과정은 full backup 이고 계산이 많이 필요했다. GPI 는 이 과정과 유사하나, PE 과정에서 정확한 값을 계산하지 않고 계속해서 sample 만 뽑아 적당히 근사시킨다. 즉, current policy 에 적당히 부합하도록 value function 을 근사시키고 그 다음 policy 는 이를 통해 계산하는 것이다. 거의 대부분의 RL 은 이 과정으로 설명될 수 있다.



6.5 Monte Carlo Method

Monte Carlo Method (MC) tabular updating 방식이며 MDP transition에 대해 몰라도 되는 model-free 방식이다. MC는 episode-by-episode (즉, 게임이 끝날 때 까지가 하나의 episode) 방식으로 GPI를 사용하는데, sample로 뽑은 전체 경로로부터 학습을 진행한다.

$$\text{Value} = \text{average of returns } G_t \text{ of sampled episodes}$$

즉, 시작부터 끝까지 가본 뒤에 얻은 return 값들을 평균 내어 estimation을 진행한다. 이 때 사용되는 estimation은 q_π 이며 ϵ -greedy policy improvement를 사용한다. MC는 episode-by-episode이기 때문에, 즉 시작부터 끝까지를 한 번의 episode로 하기 때문에 Markov property가 아닌 stochastic decision process도 어느 정도 적용된다. 그러나 이 때문에 끝이 존재하며 discrete한 모델의 경우에만 적용이 가능하다.

6.5.1 MC Prediction (Policy Evaluation, PE)

목표는 policy π 를 사용한 모든 episode에서 q_π 를 학습하는 것이다. state S_0 에서 시작해 timestep T 까지 진행하는 다음의 모델을 가정하자.

- entire trajectory :

$$S_0, A_0, R_1, S_1, A_1, \dots, R_T$$

- return :

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$$

목표는 q_π 를 학습하는 것이다. 정의에 따르면 action-value function은 다음과 같다.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

단, 여기서는 실제 기대값을 계산하는 것이 아닌 (원래는 각 값들에 transition probability를 곱해야 기대값이 된다), 경험적인 값들의 산술 평균값을 사용하는 것이다.

업데이트할 때, state s 를 기준으로 하는데 게임을 진행하다 보면 이 상태가 다시 나오기도 한다. 때문에 업데이트를 어떻게 할지를 선택해 줘야 한다. First-Visit은 state s 가 처음 나왔을 때 한 번만 해주고 후에는 무시하는 것이고, Every-Visit은 state s 가 나올 때마다 업데이트를 해주는 것이다. First-Visit으로 바둑에 대한 학습을 있다고 생각해 보자. 바둑은 한번의 대국에 대략 200번을 둔다고 한다. 그렇다면 우리는 한 번의 episode에 200개 정도의 state variable을 업데이트하는 것이다.

한번의 episode에 대해 평균을 계산하는 과정은 다음과 같다.

1. increment count : state s 와 action a 가 나올 때마다 그 회수를 count 한다.

$$n(s, a) \leftarrow n(s, a) + 1$$

2. increment total return : state s 와 action a 가 나올 때마다 total return을 업데이트 한다. timestep t 에서 $S_t = s, A_t = a$ 라고 하면 다음과 같다.

$$\text{Sum}(s, a) \leftarrow \text{Sum}(s, a) + G_t$$

3. value of (s, a) is estimated by mean return : $Q(s, a)$ 는 state s 와 action a 가 나올 때마다 합해준 return 값을 state s 와 action a 가 나온 회수로 산술평균한 값이다.

$$Q(s, a) = \frac{\text{Sum}(s, a)}{n(s, a)}$$

law of large numbers에 의하면 그 수렴성이 보장된다.

$$\lim_{n(s,a) \rightarrow \infty} Q(s, a) = q_\pi(s, a)$$

이 방법은 각각의 s 의 모든 경우에 대해 계산해야 한다. 계산량을 줄이기 위한 방법으로 다음을 사용해 보자.

General principle of incremental mean

$$\mu_k = \frac{1}{k} \sum_{i=1}^k x_i = \frac{1}{k} \left(\sum_{i=1}^{k-1} x_i + x_k \right) = \frac{1}{k} ((k-1)\mu_{k-1} + x_k) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

즉, 이미 계산한 이전단계의 평균을 저장해놓고 계속 사용하자는 것이다. 이를 이용하면 다음과 같은 업데이트가 가능하다. 이전에 소개한 식은 하나의 state s 와 action a 에 초점을 맞췄다면, 이번에는 하나의 episode에서 timestep $t = 0, 1, 2, \dots$ 이 진행되면서 나오는 각각의 state/action에 대한 업데이트에 초점을 맞춘다.

Incremental Monte Carlo updates

$$\begin{aligned} n(S_t, A_t) &\leftarrow n(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{n(S_t, A_t)} [G_t - Q(S_t, A_t)] \end{aligned}$$

이 과정은 원래 값 $Q(S_t, A_t)$ 에서 조금씩 움직여주는 것으로 생각할 수 있다. $n(S_t, A_t)$ 는 점점 커지므로, 업데이트는 계속해서 느려질 것이다. 생각해보면 이는 과거의 정보와 지금의 정보가 모두 동등한 영향을 갖고 있다는 가정 하에서 이루어진 계산이었다. 따라서 우리는 현재의 데이터에 좀 더 가중치를 주고자 다음과 같은 식을 사용한다.

constant- α MC Policy Evaluation

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha [G_t - Q(S_t, A_t)] \\ &= (1 - \alpha)Q(S_t, A_t) + \alpha G_t \end{aligned}$$

첫번째 줄을 보면 이는 DL에서 살펴봤던 Gradient Descent와 비슷한 모양인데, G_t 를 target으로 하여 $Q(S_t, A_t)$ 가 다가가는 모양이다. 즉, 우리는 target G_t 를 향하는 학습을 진행하는 것이다.

6.5.2 MC Control (ϵ -Greedy Policy Improvement, PI)

우선 Exploitation과 Exploration 성질에 대해서 알아보자.

- **Exploitation**: 새로운 sample을 찾을 때 이미 가지고 있는 정보를 통해 가장 높은 q 값을 주는 action sample을 고른다. 하지만 계속 현재 상황에서 최고의 것만 보고 판단을 할 경우에는 optimal로 가는 것이 아니고 local optimum에 빠질 위험이 있다.
- **Exploration**: 새로운 sample을 찾을 때 $1 - \epsilon$ 의 확률로 이미 가지고 있는 정보를 통해 가장 높은 q 값을 주는 sample을 고른다. ϵ 의 확률로는 random 한 action을 고른다. 이는 단기적으로

봤을 때 손해라고 보여지지만, 새로운 영역을 탐험하면서 더 좋은 policy 를 찾을 가능성도 있다. 이것이 ϵ -greedy policy improvement 방법이다.

기존 policy π 의 경우 다음과 같을 것이다.

$$\pi(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a Q_\pi(s, a) \\ 0 & \text{otherwise} \end{cases}$$

하지만 π 는 optimal policy 가 아니기에 이를 수정해야 할 필요가 있고 이것이 policy improvement 방식이었다. MC 는 여기에 대해 Exploration 의 방법을 사용하는데, 이는 최적의 greedy action (가장 높은 Q 값을 주는 sample) 을 $1 - \epsilon$ 의 확률로 고르고 ϵ 의 확률로 random action 을 고르는 방법이다. 즉, 전체 action 이 m 개 있다면 각 action 에 대해 random 으로 고를 확률은 $\frac{\epsilon}{m}$ 이 된다. policy 는 다음과 같이 선택된다.

$$\pi'(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{m} & \text{if } a = \operatorname{argmax}_{a'} Q_\pi(s, a') \\ \frac{\epsilon}{m} & \text{otherwise (} m - 1 \text{ actions)} \end{cases}$$

우선 policy π 에서 가장 높은 q 값을 주지 않는 $m - 1$ 개의 action 들은 각각 $\frac{\epsilon}{m}$ 의 확률을 가진다. 그리고 policy π 에서 가장 높은 q 값을 주는 1개의 action 에는 $1 - \frac{\epsilon}{m}(m - 1) = 1 - \epsilon + \frac{\epsilon}{m}$ 의 확률이 주어진다.

DP 에서 policy iteration 을 다룰 때 Policy Improvement Theorem 이라는 이론배경이 있었다. ϵ -Greedy Policy 도 이론적 배경이 있다.

ϵ -Greedy Policy Improvement Theorem

For any ϵ -greedy policy π , ϵ -greedy policy π' is always improved:

$$v_{\pi'}(s) \geq v_\pi(s)$$

proof :

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a q_\pi(s, a) \mathbb{P}[\pi'(s) = a | s] \\ &= \frac{\epsilon}{m} \sum_a q_\pi(s, a) + (1 - \epsilon + \frac{\epsilon}{m}) \max_a q_\pi(s, a) \\ &= \frac{\epsilon}{m} \sum_a q_\pi(s, a) + (1 - \epsilon + \frac{\epsilon}{m}) \sum_a \frac{\pi(a|s) - \frac{\epsilon}{m}}{1 - \epsilon} q_\pi(s, a) \\ &\geq \sum_a \pi(a|s) q_\pi(s, a) \\ &= v_\pi(s) \end{aligned}$$

이 때 $\max_a q_\pi(s, a)$ 를 변형시킬 때 사용된 식을 보면 다음과 같다.

$$\sum_a \frac{\pi(a|s) - \frac{\epsilon}{m}}{1 - \epsilon} q_\pi(s, a) = \frac{(1 - \epsilon + \frac{\epsilon}{m}) - \frac{\epsilon}{m}}{1 - \epsilon} \max_a q_\pi(s, a) + \sum_{a \neq \operatorname{argmax}_a q_\pi(s, a)} \frac{\frac{\epsilon}{m} - \frac{\epsilon}{m}}{1 - \epsilon} q_\pi(s, a) = \max_a q_\pi(s, a)$$

다음은 ϵ -greedy policy improvement 에 대한 pseudo code 이다.

```

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :
 $Q(s, a) \leftarrow$  arbitrary
 $Returns(s, a) \leftarrow$  empty list
 $\pi \leftarrow$  an arbitrary  $\epsilon$ -soft policy

Repeat forever:
  (a) Generate an episode using  $\pi$ 
  (b) For each pair  $s, a$  appearing in the episode:
     $R \leftarrow$  return following the first occurrence of  $s, a$ 
    Append  $R$  to  $Returns(s, a)$ 
     $Q(s, a) \leftarrow \text{average}(Returns(s, a))$ 
  (c) For each  $s$  in the episode:
     $a^* \leftarrow \arg \max_a Q(s, a)$ 
    For all  $a \in \mathcal{A}(s)$ :
       $\pi(s, a) \leftarrow \begin{cases} 1 - \epsilon + \epsilon / |\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon / |\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$ 

```

주의할 점은 최초 policy 초기화에서 모든 action에 대한 probability는 0이 되지 않도록 해야 학습의 가능성이 있다는 것이다. (non-empty probabilities) ϵ -greedy policy improvement는 수렴을 보장하지 않으나 (즉, 수렴하지 않는 반례가 있다), 대부분의 경우 수렴한다.

*GLIE (Greedy in the Limit with Infinite Exploration) : epsilon 값을 점점 줄이는 방법을 사용하는데, 이는 convergence를 보장한다.

6.6 Temporal Difference Learnings

Temporal Difference Learnings (TD) 이 역시 tabular updating 방식이며 MDP transition에 대해 몰라도 되는 model-free 방식이다. Policy improvement 방법으로서 GPI (ϵ -greedy PI 등) 을 사용하나 MC 와는 다르게 하나의 episode에서 각각의 one-step 방식으로 사용한다. target 으로 실재값이 아닌 추측값으로, estimator로 estimate를 구하는 방식이다. 기본적인 방식은 MC 와 같으나 Policy Evaluation 모양이 다르다.

6.6.1 TD Prediction (Policy Evaluation, PE)

MC의 경우 target으로서 G_t 를 사용했다. 즉, 식이 다음과 같았다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]$$

G_t 를 계산해야 했기에 하나의 episode가 모두 끝나야 학습이 가능했다. 여기서 우리는 deterministic policy를 이용하여 G_t 를 근사한다. 즉, policy를 사용하는 경우 G_t 는 다음과 같이 근사된다.

$$G_t = R_{t+1} + \gamma G_{t+1} \approx R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$$

하지만 우리가 가진 policy는 optimal 하다는 보장이 없다. 때문에 policy에 의존하지 않고 근사하는 경우 다음과 같다.

$$G_t \approx R_{t+1} + \gamma \max_a q(S_{t+1}, a)$$

위의 두 가지 근사를 이용하여 다음과 같은 TD 방법이 만들어진다.

- **Sarsa** : On-Policy TD Control - *target $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$ 은 현재 policy에 의한 값이다. (Sarsa 인 이유 : state - action - reward - state - action 을 한 세트로 업데이트를 진행한다.)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

*behavior policy는 A_{t+1} 이다. 다음은 Sarsa의 pseudo code이다.

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
    Initialize  $s$ 
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
    Repeat (for each step of episode):
        Take action  $a$ , observe  $r, s'$ 
        Choose  $a'$  from  $s'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
         $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ 
         $s \leftarrow s'; a \leftarrow a'$ ;
    until  $s$  is terminal

```

***Target policy** : 학습의 대상이 되는 policy, ***Behavior policy** : behavior 를 결정하는 policy, 즉 episode 를 생성하는 policy.

- **Q-learning** : Off-Policy TD Control - target $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ 이 현재 policy 와 관련이 없다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

식에서 다음 action 이 정해지지 않았기 때문에 (즉, a 가 확정되지 않았기 때문에 behavior policy 가 아닌 그냥 계산값이다.) behavior policy 가 없다. 다음은 Q-learning 의 pseudo code 이다.

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
    Initialize  $s$ 
    Repeat (for each step of episode):
        Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
        Take action  $a$ , observe  $r, s'$ 
         $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
         $s \leftarrow s'$ ;
    until  $s$  is terminal

```

TD 는 one-step 으로 진행하기 때문에 Markov property 가 있는 모델에만 적용되며, 게임이 끝나지 않아도 학습은 시행할 수 있다. 또한 continuous process 에도 적용이 가능하다. 또한 통상적으로는 MC 보다 TD 가 빠르다.

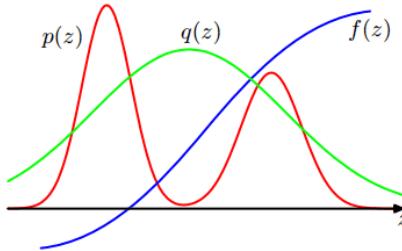
On-Policy vs Off-Policy

- **On Policy** 현재 사용중인 policy 를 이용하여 결정을 내린다. 즉, Episode 를 탐험하는 policy 와 PI, PE 하는 policy 가 같다.
- **Off Policy** Episode 를 탐험하는 policy 와 PI, PE 하는 policy 가 다르다. Q-learning 의 경우 PE 하기 위해 target $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ 에서 action a 는 $Q(S_{t+1}, a)$ 를 maximize 하는 것으로 고른다.

On Policy 의 경우 현재 진행중인 policy 만 고려하며 과거의 policy 는 사용하지 않는다. 반면 Off Policy 의 경우 과거에 만들어졌던 정보들을 사용할 가능성이 있고, 심지어는 외부에서 만든 데이터도 사용할 수도 있다. 때문에 Off Policy 는 exploration 이 잘 되어있다. On Policy 는 exploration 을 늘리기 위해 일반적인 GPI 대신 ε -greedy 방식을 사용할 수 있다.

6.6.2 Importance Sampling

다음과 같은 함수와 확률분포에 대해, $f(z)$ 의 $p(z)$ 에 대한 기대값 $\mathbb{E}_p[f(z)]$ 를 계산해야 한다고 하자.



어떤 함수 $f(z)$ 의 $p(z)$ 에 대한 기대값을 계산해야 하는데 $p(z)$ 분포가 다루기 어려운 함수일 때, $q(z)$ 에 대한 expectation 으로 바꿔 계산하는 방법이다. 이를 이용하면 위에서 소개한 policy evaluation 계산을 쉬운 방법으로 바꿀 수 있다.

Importance Sampling

$$\mathbb{E}_p[f(X)] = \mathbb{E}_q \left[\frac{P(X)}{Q(X)} f(X) \right]$$

where we call $\frac{P(X)}{Q(X)}$ the **Importance-sampling ratio**.

proof :

$$\text{Discrete : } \mathbb{E}_p[f(X)] = \sum P(X)f(X) = \sum Q(X)\frac{P(X)}{Q(X)}f(X) = \mathbb{E}_q \left[\frac{P(X)}{Q(X)}f(X) \right]$$

$$\begin{aligned} \text{Continuous : } \mathbb{E}_p[f(X)] &= \int f(x) \frac{p(x)}{q(x)} q(x) dx = \mathbb{E}_q \left[\frac{P(X)}{Q(X)} f(X) \right] \\ &\approx \frac{1}{N} \sum_n \frac{p(x_n)}{q(x_n)} f(x_n) \quad \text{where } x_n \sim q(x) \end{aligned}$$

어떤 target policy π 가 주어졌을 때 이에 따라 하나의 episode trajectory $T = \{A_t, S_{t+1}, A_{t+1}, \dots, S_T\}$ 가 발생할 확률은 다음과 같다.

$$\begin{aligned} \mathbb{P}[T | S_t = s, A \sim \pi] &= \mathbb{P}[T | S_t, A_t] \mathbb{P}[A_t | S_t] \\ &= \mathbb{P}[T | S_t, A_t, S_{t+1}] \mathbb{P}[S_t, A_t, S_{t+1} | S_t, A_t] \mathbb{P}[A_t | S_t] \\ &\vdots \\ &= \mathbb{P}[S_T | S_{T-1}, A_{T-1}] \cdots \mathbb{P}[S_{t+1} | A_t, A_t] \mathbb{P}[A_t | S_t] \\ &= \prod_{k=t}^{T-1} \pi(A_k | S_k) \mathbb{P}[S_{k+1} | S_k, A_k] \end{aligned}$$

behavior policy μ 가 주어지면 importance-sampling ratio 는 다음과 같다.

$$\rho_t^T = \frac{\prod_{k=t}^{T-1} \pi(A_k | S_k) \mathbb{P}[S_{k+1} | S_k, A_k]}{\prod_{k=t}^{T-1} \mu(A_k | S_k) \mathbb{P}[S_{k+1} | S_k, A_k]} = \prod_{k=t}^{T-1} \frac{\pi(A_k | S_k)}{\mu(A_k | S_k)}$$

behavior policy 가 존재해야 가능하므로 on-policy 방식인 MC 와 Sarsa 에 대해 이를 적용해 본다.

1. MC

target G_t 는 그 정의에 따르면 다음과 같다.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \\ &= \sum_{k=0}^{T-t-1} \gamma^k R_{t+1+k} \end{aligned}$$

이는 본래 π 에 따라 계산되는 값이지만, behavior policy 에 따른 것으로 바꾸기 위해 다음과 같이 정의한다.

$$G_t^{\pi/\mu} = \rho_t^T G_t = G_t \prod_{k=t}^{T-1} \pi(A_k|S_k) \mathbb{P}[S_{k+1}|S_k, A_k]$$

결과적으로 Off-policy MC 는 다음과 같은 PE 식을 갖게 된다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [G_t^{\pi/\mu} - Q(S_t, A_t)]$$

하지만 이 것은 variance 를 매우 높일 가능성이 있다.

2. Sarsa

이 또한 마찬가지로 target 에 importance-sampling ratio 를 곱해 사용하면 Off-policy Sarsa 가 된다. 이 경우는 one step 에 대해 update 하기 때문에 one step 에 대한 ratio 만 곱해준다.

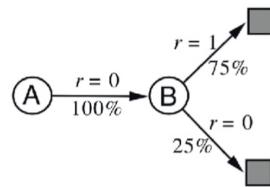
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})) - Q(S_t, A_t) \right]$$

Bias vs Variance MC 의 경우 target 이 실재 데이터이기 때문에 이 과정을 무한히 반복한다면 estimation 값이 실재값과 같아질 것이다. 즉, target G_t 는 참값 $q_\pi(S_t, A_t)$ 의 unbiased estimate 이다. TD 의 경우 target 이 실재 데이터가 아니기에 이 과정을 무한히 반복한다고 해도 실재값으로 가지 않을 가능성이 있다. 즉, target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ 은 참값 $q_\pi(S_t, A_t)$ 의 biased estimate 이다.

한편 MC 는 random 으로 episode 를 고르기 때문에 variance 가 매우 크다. TD 는 one step 단위로 학습을 진행하기 때문에 variance 가 매우 작다. 때문에 쓰임에 맞는 모델을 사용하면 된다.

6.6.3 Applications

AB example : MC vs TD A 에서 어떤 action 을 취하면 반드시 B 로 가며 reward 0, B 에서 어떤 action 을 취하면 0.75 의 확률로 reward 1 을 받으며 게임이 끝나고, 0.25 의 확률로 reward 0 으로 게임이 끝나는 게임이 있다.



그리고 다음과 같은 8 개의 episode 를 얻었다고 가정하자.

$$\{A, 0, B, 0\}, \{B, 1\}, \{B, 1\}, \{B, 1\}, \{B, 1\}, \{B, 1\}, \{B, 1\}, \{B, 0\}$$

8개의 episode 중 6개의 경우에서 1 의 return 을 얻었으므로 $V(B)$ 의 estimate 값은 0.75 일 것이다. $V(A)$ 는 MC 와 TD 의 방법으로 각각 계산해보면 다음과 같다.

- MC

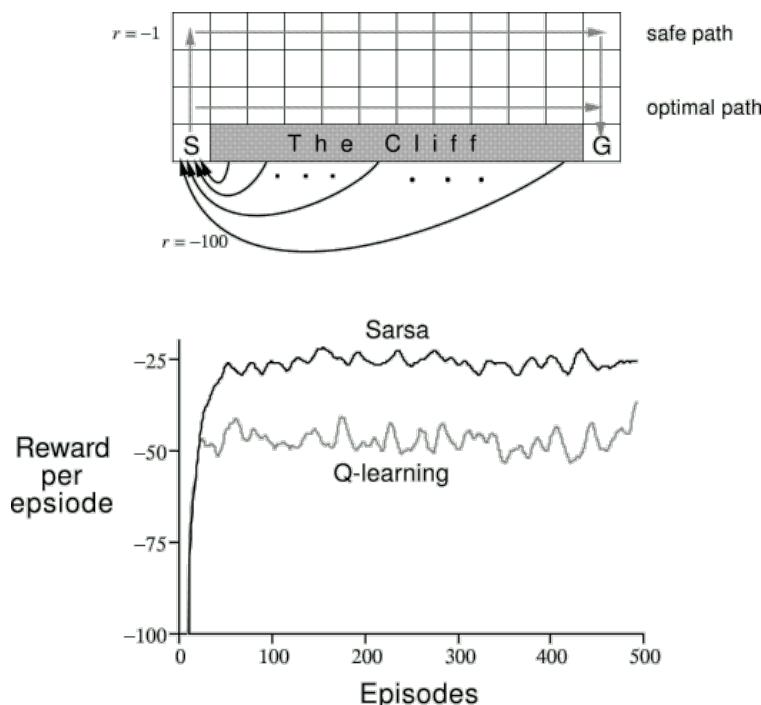
$$V(A) \leftarrow V(A) + \alpha[G_t - V(A)] = 0$$

- TD

$$V(A) \leftarrow V(A) + \alpha[R_{t+1} + \gamma V(B) - V(A)] = 0 + 1 \cdot [0 + 1 \cdot 0.75 - 0] = 0.75$$

두 결과가 다르다. 직관적으로 봐도 TD 가 맞아보인다. 통상적으로 TD 가 더 좋은 결과를 낸다. 물론 아직까지 수학적인 근거는 없다.

Sarsa vs Q-learning Cliff walking의 예시를 고려해보자. Grid world에서 agent가 움직이는데, Cliff 영역에 도달할 때마다 초기위치 S 로 돌아오며 reward -100 을 받는다. 하나의 action을 취할 때마다 얻는 reward는 -1 이다.

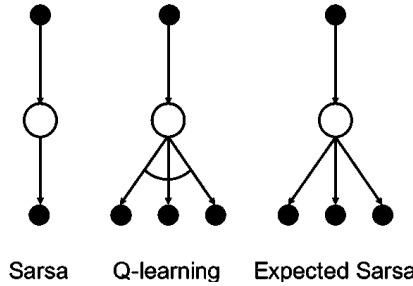


Sarsa 는 정해진 policy에 대한 최적의 action을 찾기 때문에 더 안전한 safe path를 찾고, Q-learning은 각 action마다 Off-Policy이기 때문에 최단거리인 optimal path를 찾는 모습을 볼 수 있다. 물론 이 길은 Cliff로 떨어질 위험이 있기 때문에 return은 Sarsa 가 더 높은 것을 볼 수 있다.

Expected Sarsa Q-learning에서 maximum을 사용하는 대신 expectation 값을 사용하는 방법이다. 때문에 $t+1$ 의 action이 식에 들어가 'Sarsa'의 이름이 붙는다. 이는 Sarsa 보다 성능이 좋기는 하지만, 계산량이 너무 많아진다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)]$$

다음은 Sarsa, Q-learning, Expected Sarsa를 비교한 그림이다.



Sarsa 는 $S_t, A_t, R_t, S_{t+1}, A_{t+1}$ 가 policy 에 따라 정해져 있는 모습이다. Q-learning 은 Off policy 이기 때문에 S_t, A_t, R_t, S_{t+1} 까지는 주어지지만 A_{t+1} 는 maximum 을 이용해 찾는 모습이다. Expected Sarsa 는 policy 에 따라 A_{t+1} 에 대한 값을 평균내는 모습이다.

Double Q-learning 기존의 Q-learning 에서 사용한 $\max_a Q(S_{t+1}, a)$ 는 Q 값이 높아지는 bias 를 초래했다. (maximization bias) 이는 $q(s, a)$ 로의 수렴을 느리게 했다. 이를 극복하기 위해 Double Q-learning 은 두개의 Q-Value function 을 사용한다. 다음 알고리즘을 살펴보자.

```

Initialize  $Q_1(s, a)$  and  $Q_2(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily
Initialize  $Q_1(\text{terminal-state}, \cdot) = Q_2(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
        Choose  $A$  from  $S$  using policy derived from  $Q_1$  and  $Q_2$  (e.g.,  $\varepsilon$ -greedy in  $Q_1 + Q_2$ )
        Take action  $A$ , observe  $R, S'$ 
        With 0.5 probability:
            
$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left( R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A) \right)$$

        else:
            
$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left( R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A) \right)$$

         $S \leftarrow S'$ 
    until  $S$  is terminal
  
```

TD(n) n -step return 을 사용한 강화학습 방법의 종류이다.

$$\begin{aligned} G_t^{(1)} &= R_{t+1} + \gamma V(S_{t+1}) && : \text{TD target} \\ G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n V(S_{t+n}) \\ G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T && : \text{MC target} \end{aligned}$$

이에 따라 n -step TD learning 은 다음과 같은 update 식을 갖는다.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)]$$

TD(λ) n -step 에서 n 을 어떻게 정하는지가 문제이다. TD(λ) 는 n 을 hyperparameter 로 남겨두지 않고 geometrix weight 를 취해 다음과 같이 만든다.

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

이 때 사용된 geometric form 은 합하면 1이라 평균이 바뀌지 않으며, TD(n) 방법보다 그 성능이 좋다.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^\lambda - V(S_t)]$$

이 때 TD(0)=TD 이며, TD(1) 은 every-visit MC 와 유사하다. 따라서 TD(0.5) 는 TD 와 MC 를 대략 섞은 것이라 할 수 있다.

7 Lecture 7 : Deep Reinforcement Learning II : DRL

7.1 Introduction

RL vs Deep RL

- Reinforcement Learning (RL)

- Tabular updating method : state-action value function $Q(s, a)$ 에 대한 table을 iteration을 통해 업데이트하면서 optimal policy 를 찾아나간다. 이론적 배경으로서 Bellman equation 을 사용한다. (MC, TD)
- state, action space 가 백만 이하일 때 적용할 수 있다.

- Deep Reinforcement Learning (DRL)

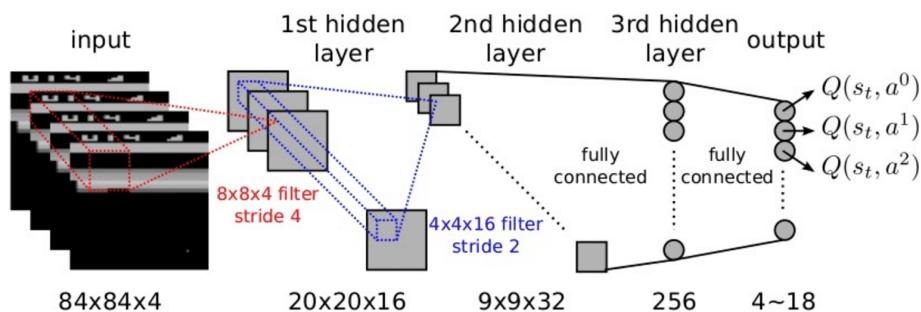
- Function approximation method : DNN 을 통해 state-action value function 을 근사해 나간다. 이 때 어떤 function 을 이용하는가에 따라 method 가 나뉜다.
 - * $Q(s, a) \leftarrow$ Deep Q-Network (DQN)
 - * $\pi(a|s) \leftarrow$ Policy Gradient
 - * Both \leftarrow Actor-Critic (A3C)
- 발전된 형태의 DRL 의 경우는 state, action space 는 얼마든지 커도 되며 심지어 연속이어도 된다.
- DL 을 label 없이, RL 을 table 없이 하는 것과 같다.

Why Deep RL? Deep Learning 과 Reinforcement Learning 을 합친 말이다. DQN 와 AlphaGo 가 성공했듯이, 게임, 로봇, 자율주행, 금융 등에서 Bigdata 의 힘과 DL algorithm 의 힘으로 매우 많은 분야의 decision process 를 풀 수 있다. RL 에서는 curse of dimensionality 문제가 있었지만, DL 은 자동으로 차원을 줄여주기 때문에 효과가 좋다.

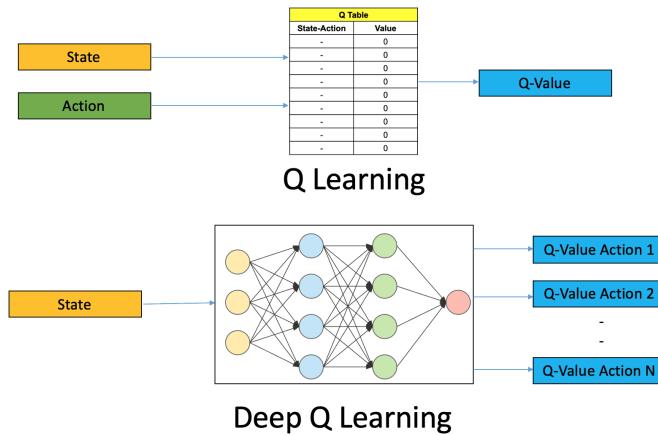
처음 성공한 것은 Atari game 에 적용한 DQN (DeepMind 2013) 이었고, 다음은 hybrid DRL system (CNN + RL) 을 이용한 AlphaGo 였다. David Silver 는 DL 과 RL 을 합친 것을 AI 라 불렸다.

7.2 Deep Q-Network (DQN)

Deep Q-Network 2013년 DeepMind 에서 처음 사용되었다. CNN 과 Q-learning 을 합친 구조인데, CNN 은 pixel image 를 받아 approximated Q-value 를 출력한다.



state 와 action 을 받아 각 step 에서 learning 하던 Q-learning 과는 달리 DQN 은 Q-value 를 적절한 수의 parameter θ 를 통해 근사했다.



이 때 action space 는 CNN 의 출력값이므로 discrete (finite and not large) 하며, state 는 크거나 연속이어도 상관 없다. 좋은 action을 찾기 위해서는 forward pass 만을 사용해야 한다.

Naive DQN : CNN+DL 만 사용한 경우. 우선 Q -learning 의 Policy Evaluation 을 다시 살펴보자.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

이 때 target은 $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ 이었다. **Naive DQN** 은 같은 target을 사용하지만 action 은 parameter 로 표현되기 때문에 $r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta)$ 가 되며, MSE loss 를 최소화 하는 방향으로 gradient descent 가 진행된다.

$$L(\theta) = [r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta)]$$

Naive DQN 은 3개의 convolution layer 와 2개의 FC 로 이루어져 있다. 그러나 문제가 있다. FC 하나만 가지고 시행한 것 보다 성능이 안좋다.

	Breakout	R. Raid	Enduro	Sequest	S. Invaders
Naive DQN	3.2	1453.0	29.1	275.8	302.0
Linear	3.0	2346.9	62.0	656.9	301.3

이를테면, 로봇이 걷는 machine 을 만들고 싶다면 기계의 순간순간 모습들을 사진으로 만들어 DQN 으로 넣어주는 방법이 있을 것 이다. 그러나 해당 사진이 기울어지는 중 인지 다시 서는 중 인지 알 수가 없다. 이렇게 구분이 되지 않으면 해당 state 안넘어질 것 같다고 판단하게 되고 그 다음, 정말 넘어지게 될 state 가 걸려내지지 않는다. 즉, 근접한 데이터간의 상호관계 (correlation) 에 의해 학습이 좋지 못하다.

또는 target $r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta)$ 이 Q network 에 의존하기 때문에 학습이 좋지 않기도 하다. Q network 를 업데이트할 때마다 target 이 바뀌므로 target 은 non-stationary 하며 따라서 estimator 가 발산하기도 한다.

DQN 다음의 특징들을 살펴보자.

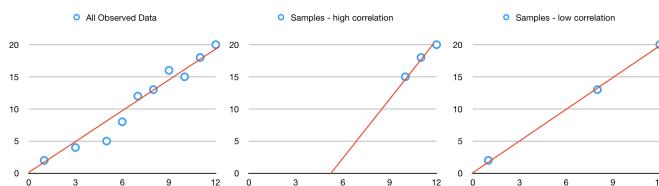
- 다음의 4가지 방법으로 action-value function 근사과정을 안정시켰다.

1. CNN 을 사용했다. : DQN 은 최근 4개 frame 을 하나의 데이터로 사용했는데, grey scale 84×84 pixel 이니 (물론 원래 이미지는 128 color, 210×160 pixel 이나 줄여서 이정도이다.) 총 28224 dimension input 이다. 따라서 screen 의 상태, 즉 state 는 2^{28224} 개인데, 이는

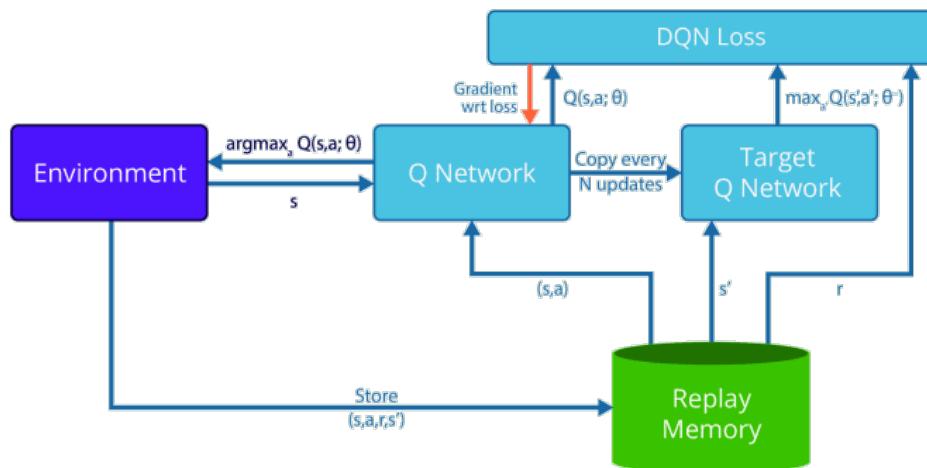
Q-learning 으로 풀 수 없는 문제이다. 따라서 dimension 을 낮춰줄 CNN 을 사용하게 되었다. 여기서는 512 float data 로 차원을 줄였다. 게임에서는 미세한 pixel 이동이 중요하기 때문에, pooling은 사용하지 않았다.

2. Experience replay (Replay buffer) : sample 간의 상호관계를 제거하기 위해 사용되었다.
 3. Target network : target \circlearrowleft non-stationary 인 것을 극복하기 위해 사용되었다.
 4. Clipping rewards : 각각의 게임 score 가 다른 scale 을 가지고 있을 수 있으므로, positive reward 는 모두 +1 로 고정시키고 negative reward 는 모두 -1 로 고정시켰다.
- 게임의 pixel 과 score 만을 input 으로 하여 end-to-end RL 방식을 만들었다. 즉, 최소한의 정보 만이 필요했다.
 - 같은 네트워크 구조로 다양한 역할을 하는 machine 을 만들 수 있다. 실제로 이들은 다양한 Atari 게임들에 대해 같은 알고리즘을 적용시켜 대부분 사람을 뛰어넘는 결과를 냈다.

Experience Replay (Replay buffer) Agent 는 한번의 게임에서 강력하게 연관되어있는 경험을 토대로 학습을 진행하는데, 때문에 네트워크는 이 biased data 에 맞게 학습을 진행한다. 즉, 4개의 연결된 frame 만을 가지고 학습을 진행하면 매 학습이 그 순간에 biased 되어있을 것이다.



Experience Replay 는 Replay buffer (십만 이상의 크기를 가진)에 agent의 경험과 게임에서 일어난 state, action, reward, transition $\{(s, a, r, s')\}$ 을 저장한 후, 나중에 랜덤하게 minibatch 를 뽑아 학습을 진행한다. 이는 sample 간의 상호관계를 제거하고, minibatch 를 사용했기에 learning speed 를 올리고, data 를 계속 재사용하면서 효율을 높인다.



구조를 보면, Environment에서 작동하고 있는 Agent 는 새로운 데이터 $\{(s_t, a_t, r_{t+1}, s'_{t+1})\}$ 를 바로 사용하는 것이 아니라 Replay Memory 에 저장한다. Replay Memory 의 크기는 일정하며 (이를테면 $N = 10,000$) 그 이상의 정보가 들어오면 가장 오래된 것을 버린다. 그리고 여기서 minibatch (이를테면 $B = 200$) 를 뽑아 Q-Network 에 주고, 이를 통해 학습한 후 (학습할 때 target 은 Target Q-Network 의 parameter 를 사용한다.) 다음 action $a_{t+1} = \text{argmax}_a Q(s_{t+1}, a; \theta)$ 를 Agent 에게 줘 다음 action 을 결정한다.

Target Network 만약 target function 이 빠르게 바뀐다면, training 이 잘 되지 않을 것이다. 즉, Q -Network = Target Q -Network 라면 training이 잘 되지 않을 것이다. Target Network 는 Target \hat{Q} -network 에서 parameter $\hat{\theta}$ 를 고정시키고 Behavior Q -network 의 parameter θ 를 업데이트 한다. 수많은 training이 진행되면 (이를테면 $N = 1000$ 번의 iteration 후) Target \hat{Q} -network 를 Behavior Q -network 로 치환한다. Batch 크기를 B 라 하자.

- Forward Pass Loss function

$$L(\theta) = \frac{1}{B} \sum_{|\{i\}|=B} \left[r_{i+1} + \gamma \max_a \hat{Q}(s_{i+1}, a; \hat{\theta}) - Q(s_i, a_i; \theta) \right]^2$$

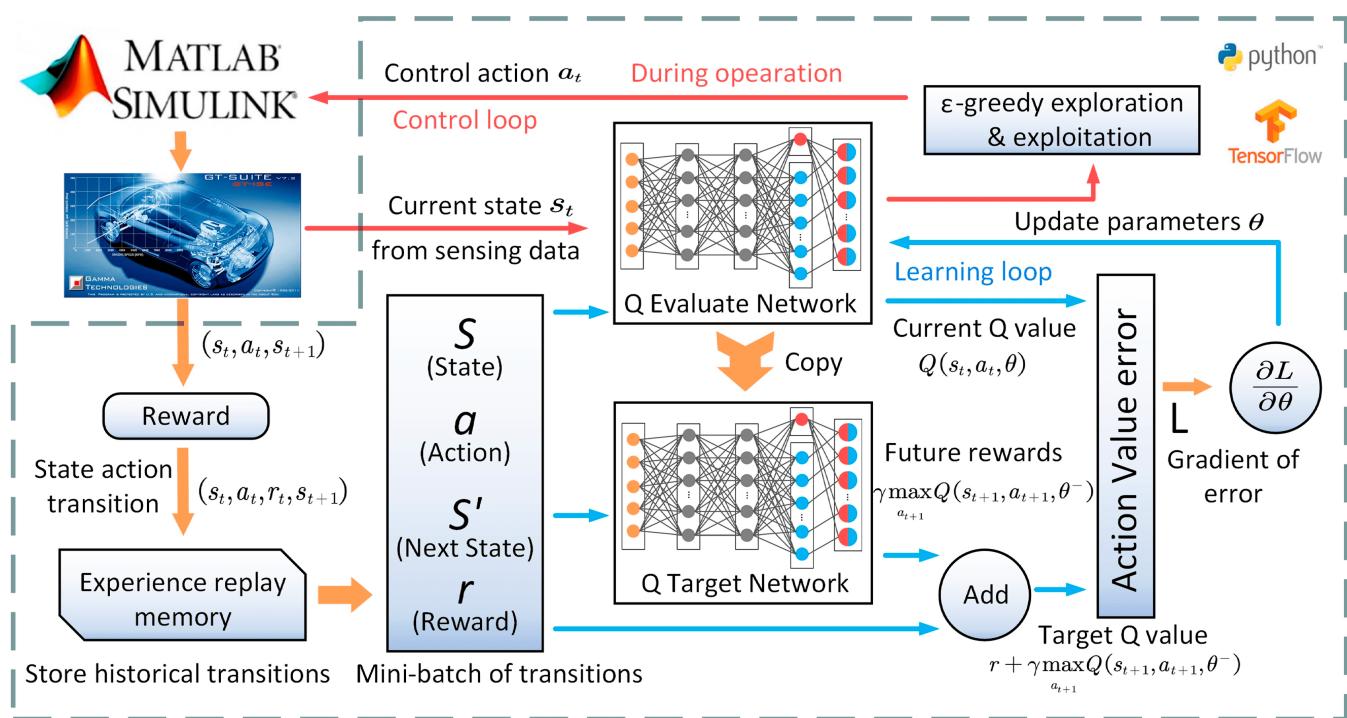
- Backward Pass

$$-\nabla_{\theta} L(\theta) = \frac{2}{B} \sum_{|\{i\}|=B} \left[r_{i+1} + \gamma \max_a \hat{Q}(s_{i+1}, a; \hat{\theta}) - Q(s_i, a_i; \theta) \right] \nabla_{\theta} Q(s_i, a_i; \theta)$$

update 식은 다음과 같다.

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} L(\theta)$$

위의 과정을 정리한 그림이다.



DQN 의 알고리즘을 살펴보자.

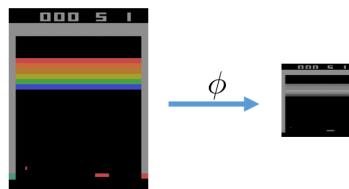
Algorithm 1: deep Q-learning with experience replay.

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
For episode = 1,  $M$  do
    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
    For  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the
        network parameters  $\theta$ 
        Every  $C$  steps reset  $\hat{Q} = Q$ 
    End For
End For

```

우선 모든 것을 초기화하고, episode 하나마다 For 문을 시행한다. s_1 은 처음 게임의 화면이고 pre-processed sequence ϕ 는 원래 128 color, 210×160 pixel 이었던 것을 흑백의 84×84 로 바꿔준 것이다.



그 다음 for 문에서는 action a_t 를 ϵ -greedy 방법으로 고르고, 이를 게임에서 실행한 후 reward r_t 와 다음 이미지 x_{t+1} 을 받아 다음 state 와 preprocess 를 계산한다. 이들을 replay memory 에 저장한다. 그후 minibatch 를 sampling 하고 학습을 진행한다.

Double DQN : Double Q -learning 을 DQN 에 접목한 것이다. DQN 의 loss 가

$$L(\theta) = \frac{1}{B} \sum_{|\{i\}|=B} \left[r_{i+1} + \gamma \max_a \hat{Q}(s_{i+1}, a; \hat{\theta}) - Q(s_i, a_i; \theta) \right]^2$$

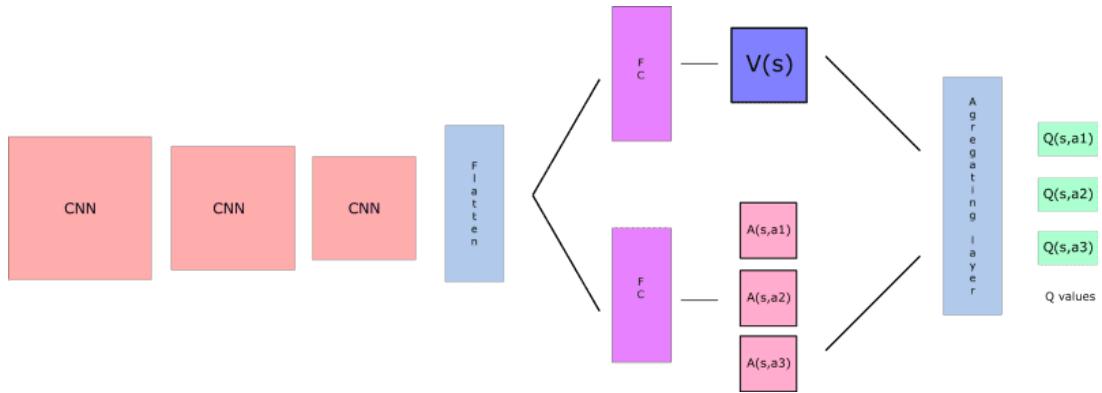
이었다면, Double DQN 의 loss 는 다음과 같다. 똑같은 식인데, 한번 더 꼬았다고 생각하면 된다. Target network 는 second network 처럼 작동한다.

$$L(\theta) = \frac{1}{B} \sum_{|\{i\}|=B} \left[r_{i+1} + \gamma \hat{Q}\left(s_{i+1}, \operatorname{argmax}_a Q(s_{i+1}, a; \theta); \hat{\theta}\right) - Q(s_i, a_i; \theta) \right]^2$$

Dueling DQN : Q -network 를 value function $V(s)$ (action-independent) + advantage function $A(s, a)$ (action-dependent) 로 나눠 따로 output 으로 받은 뒤 합한다.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

*Recall - Advantage function : action a 를 골랐을 때의 return이 (s 에서 갈 수 있는 모든 action 각각에 대한 return 의) 평균보다 더 좋은가를 나타내는 function 이다. 게임은 순간순간 최적의 action 이 중요하기 때문에 이렇게 하면 성능이 더 좋다.



7.3 Policy Gradient Method : REINFORCE

Policy Gradient Method (PGM) DQN 의 경우 action space 가 너무 크면 연산량이 너무 많다. 반면 PGM 의 경우는 policy 를 직접 구하기 때문에 연산이 빠르다. 기본적으로 PGM 은 **optimization parameterized policy π** 를 구하기 위해 total reward 의 기대값을 최대화 시키는 방향으로 gradient descent 를 시행한다. Q -value function 에 대한 어떤 정보도 필요 없으며, DQN 처럼 experience replay 가 필요하지도 않다. network weight θ 는 policy π 를 구하기 위한 parameter 이다.

- Trajectory :

$$\tau = s_1, a_1, r_2, s_2, a_2, \dots, s_T$$

- Total reward : $r(\tau)$

일 때, **Objective function** 은 다음과 같다.

$$L(\theta) = \mathbb{E}_\theta[r(\tau)] = \int p(\tau; \theta)r(\tau)d\tau$$

이 때, $p(\tau; \theta) = \pi_\theta(\tau)$, 즉 trajectory τ 가 실행될 확률분포 이다. update 식은 다음과 같다.

$$\theta \leftarrow \theta + \alpha \nabla_\theta L(\theta)$$

gradient 를 구하기는 조금 어렵다.

Policy Gradient Theorem

The derivative of the expected reward is the expectation of the product of the reward and the summed gradient of log of the policy π_θ :

$$\nabla_\theta \mathbb{E}_\theta[r(\tau)] = \mathbb{E}_\theta \left[r(\tau) \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \right]$$

proof :

$$\begin{aligned}
 \nabla_{\theta} \mathbb{E}_{\pi}[r(\tau)] &= \nabla_{\theta} \int \pi_{\theta}(\tau) r(\tau) d\tau \\
 &= \int \nabla_{\theta} \pi_{\theta}(\tau) r(\tau) d\tau \\
 &= \int \pi_{\theta}(\tau) \frac{\nabla_{\theta} \pi_{\theta}(\tau)}{\pi_{\theta}(\tau)} r(\tau) d\tau \\
 &= \int \pi_{\theta}(\tau) r(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau) d\tau \\
 &= \mathbb{E}_{\pi_{\theta}}[r(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)]
 \end{aligned}$$

parameter θ 는 policy 에만 관련이 있다는 것을 기억하자. 또한 다음과 같은 변형이 된다. 첫 줄의 식은 Markov property 를 사용한 것이다.

$$\begin{aligned}
 \nabla_{\theta} \log \pi_{\theta}(\tau) &= \nabla_{\theta} \log \left[p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t | s_t) p(s_{t+1} | s_t, a_t) \right] \\
 &= \nabla_{\theta} \left[\log p(s_1) + \log \left[\prod_{t=1}^T \pi_{\theta}(a_t | s_t) \right] + \log \left[\prod_{t=1}^T p(s_{t+1} | s_t, a_t) \right] \right] \\
 &= \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(s_t | s_t)
 \end{aligned}$$

따라서 우리는 pdf $p(\tau; \theta)$ 나 transition probability $p(s_{t+1} | s_t, a_t)$ 를 알 필요가 없다. 하지만 아직 expectation이 붙어있다. 때문에 **Markov Chain Monte Carlo (MCMC)** 방법을 사용하는데, 이는 간단히 말해 minibatch 를 통해 매우 많은 경로를 찾은 후 여기서 얻은 값들의 산술평균을 구하는 것이다. 또한 reward $r(\tau)$ 는 불규칙한 경로선택 때문에 high variance를 가지고 있다. 때문에 우리는 discounted return G_t 를 사용할 수도 있을 것이다.

MCMC 연속 확률분포를 따르는 변수를 sampling 하는 과정은 특히 어렵다. 또한 고차원 문제는 Rejection sampling, Importance sampling 등의 방법으로 해결하기 어렵다.

우선 Markov Chain 은 이전에 다뤘듯 다음 상태를 결정할 때 현재 상태만이 영향을 주는 확률과정 모델이다. 즉, 현재 상태가 $S_t = s$ 라면 다음 상태 S_{t+1} 을 결정하기 위해서 S_{t-1} 과 같은 정보가 필요 없다는 것이다. 또한 Monte Carlo 방법은 우리가 직접 계산하기 어려운 확률과정 문제에 대하여 컴퓨터가 직접 시뮬레이션하여 데이터를 얻고 이를 통해 답을 얻는 것이다.

MCMC 알고리즘은 우리가 샘플을 얻으려고 하는 목표분포를 Stationary Distribution 으로 가지는 Markov Chain 을 우선 만든다. 이 확률과정의 시뮬레이션을 가동하고, 초기값에 영향을 받는 burn-in period 를 지나고 나면, 목표분포를 따르는 샘플이 만들어진다.

이를테면 통계물리에서 배우는 lattice Ising Model 에 대한 Metropolis Algorithm 의 경우 온도에 따른 평균 에너지를 계산하기 위해 2^{N^2} 개의 입자에 대해 각각 exchange constant 를 계산하기 보다는 특정 입자의 spin 을 뒤집어 에너지가 낮아지면 그대로 두고, 에너지가 높아지면 확률적으로 그대로 두거나 다시 뒤집는다. 이 방법은 Ising Model 을 푸는데에 빠르고 좋은 방법은 아니지만, 컴퓨터가 직접 시뮬레이션 하기 때문에 기존 Ising Model 이 제시한 방법보다는 매우 유용하다.

우리는 π_{θ} 에 대한 평균값을 계산하기 위해 π_{θ} 를 따라 매우 많은 시뮬레이션을 진행하고 각각의 경로마다의 $\sum_{t=1}^T G_t \nabla \log \pi_{\theta}(a_t | s_t)$ 를 계산한다. 결과적으로 이것들의 합을 시뮬레이션 진행 회수 M 으로 나눠주면 원하는 평균을 얻게 된다.

REINFORCE (Monte Carlo Policy Gradient) Objective function 의 Gradient 는 위에서 계산했듯이 다음과 같다.

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=1}^T G_t \nabla \log \pi_\theta(a_t | s_t) \right]$$

그 알고리즘은 매우 간단하다.

1. state s 에서 시작해서 policy π_θ 를 따르는 시뮬레이션을 M 회 시행한다.
2. gradient 를 계산한다.

$$g_\theta = \frac{1}{M} \sum_{i=1}^M \left[\sum_{t=1}^T G_t^{(i)} \nabla \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \right]$$

3. policy update

$$\theta \leftarrow \theta + \alpha g_\theta$$

REINFORCE with baseline 일반 REINFORCE 는 high variance data 탓에 parameter 안정화가 어렵다. 따라서 G_t 의 variance 를 줄이는 방향으로 baseline 을 넣어주게 되는데, 원래 gradient 값이 변하지도 않아야 한다. 즉, action 하고 관련이 없는 값 $b(s_t)$ 를 넣어주면 된다.

Objective function의 Gradient 는 다음과 같다.

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=1}^T (G_t - b(s_t)) \nabla \log \pi_\theta(a_t | s_t) \right]$$

이 때 baseline $b(s_t = s)$ 에 해당하는 항은 0이 된다:

$$\begin{aligned} \mathbb{E}_{\pi_\theta} \left[\sum_{t=1}^T b(s) \nabla_\theta \log \pi_\theta(a_t | s_t) \right] &= \sum_{t=1}^T \int \pi(a_t | s_t) b(s) \nabla_\theta \log \pi_\theta(a_t | s_t) d\tau \\ &= \int \sum_{t=1}^T b(s) \nabla_\theta \pi(a_t | s_t) d\tau \\ &= b(s) \int \nabla_\theta \pi(\tau) d\tau \\ &= b(s) \nabla_\theta \cdot 1 \\ &= 0 \end{aligned}$$

variance 는 어떻게 되었을까? b 는 θ 와 관련이 없도록 주의하며 계산을 해보자. $\nabla_\theta \log \pi_\theta(\tau) = g(\tau)$ 라고 하고 계산해 보자.

$$\begin{aligned} \text{Var} &= \mathbb{E}_{\pi_\theta} \left[(\nabla_\theta \log \pi_\theta(\tau) (r(\tau) - b))^2 \right] - \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(\tau) (r(\tau) - b)]^2 \\ \frac{d\text{Var}}{db} &= \frac{d}{db} \left[\mathbb{E}_{\pi_\theta} [g(\tau)^2 (r(\tau) - b)^2] - \mathbb{E}_{\pi_\theta} [g(\tau) r(\tau)] \right] \\ &= \frac{d}{db} \mathbb{E}_{\pi_\theta} [g(\tau)^2 (r(\tau) - b)^2] \\ &= -2 \mathbb{E}_{\pi_\theta} [g(\tau)^2 r(\tau)] + 2b \mathbb{E}_{\pi_\theta} [g(\tau)^2] \end{aligned}$$

위의 값이 0이 되기 위해서 (변화량이 0이 되면 bias를 완전히 없앨 수 있다.) 이상적인 b 의 값은 다음과 같다.

$$b = \frac{\mathbb{E}_{\pi_\theta}[g(\tau)^2 r(\tau)]}{\mathbb{E}_{\pi_\theta}[g(\tau)^2]}$$

이는 gradient magnitude로 weighted 된 expected reward이다. 계산이 번거롭기 때문에 우리는 $b(s_t = s)$ 로서 state value function $V(s) = \mathbb{E}_{\pi_\theta}[G_t | S_t = s]$ 를 자주 사용한다. 이렇게 되면 value function도 학습을 진행해야 하기 때문에 parameter가 늘어나며 (\mathbf{w}), 다음 알고리즘에 그 과정이 나와있다.

REINFORCE with Baseline (episodic)

```

Input: a differentiable policy parameterization  $\pi(a|s, \theta), \forall a \in \mathcal{A}, s \in \mathcal{S}, \theta \in \mathbb{R}^n$ 
Input: a differentiable state-value parameterization  $\hat{v}(s, \mathbf{w}), \forall s \in \mathcal{S}, \mathbf{w} \in \mathbb{R}^m$ 
Parameters: step sizes  $\alpha > 0, \beta > 0$ 

Initialize policy weights  $\theta$  and state-value weights  $\mathbf{w}$ 
Repeat forever:
    Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot| \cdot, \theta)$ 
    For each step of the episode  $t = 0, \dots, T - 1$ :
         $G_t \leftarrow$  return from step  $t$ 
         $\delta \leftarrow G_t - \hat{v}(S_t, \mathbf{w})$ 
         $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$ 
         $\theta \leftarrow \theta + \alpha \gamma^t \delta \nabla_{\theta} \log \pi(A_t | S_t, \theta)$ 

```

다음 단원에서 소개하겠지만, 여기서 $v(S_t; \mathbf{w})$ 는 ‘critic’의 역할이고, $\pi(A_t | S_t, \theta)$ 는 ‘actor’의 역할이다.

DQN vs Policy Gradient ATARI DQN은 Q-learning과 CNN을 이용했다. AlphaGo는 Policy Gradient와 Monte Carlo Tree Search(MCTS). 사실, 대부분의 사람들은 여러가지 장점 때문에 Policy Gradient를 선호한다. 우선 expected reward를 직접적으로 최적화하며 Q -value function 없이 최적화 action을 찾아낸다. 또한 Experience replay를 사용하지 않고, DQN보다 코드가 훨씬 단순하다.

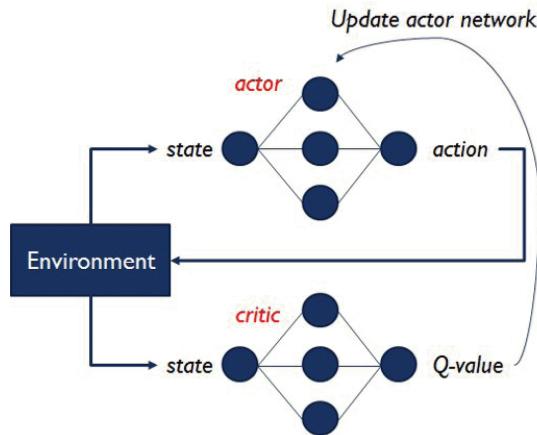
7.4 Actor-Critic Method

7.4.1 Actor-Critic Method

다음 두개의 network로 구성된다:

- **Critic**: (DQN을 사용하여) value function $V(s; \phi)$ 또는 $Q(s, a; \phi)$ 에 대한 parameter ϕ 를 update 한다.
- **Actor**: Policy Gradient를 사용하여 policy function $\pi_\theta(a|s)$ 에 대한 parameter θ 를 update 한다.

근본적으로는 GPI(Generalized Policy Iteration)와 구조가 같다. Critic은 PE의 역할이고 Actor는 PI의 역할이다.



이전에 살펴본 REINFORCEMENT with baseline (여기서부터는 사실 actor-critic 이다.)에서 봤듯이, policy 를 업데이트 하는 과정에서 value function 을 같이 업데이트 하면 더 효율적인 학습이 가능하다. Critic 에서 value function 을 학습하고, Actor 에서 action 을 찾을 때 이를 사용한다.

Policy Gradient 는 다양한 form 을 가지며 이를 정리하면 다음과 같다. REINFORCE 는 actor-critic 이 아니지만, 식을 비교해보면 결국 모든 방법들은 여기서 시작해서 G_t 를 어떻게 근사했으며 bias 를 없애기 위해 어떤 방식을 사용했는가에 지나지 않는다. one-step 에서 진행한다면 sum 기호를 제거하면 된다.

$$\begin{aligned}
 \nabla_{\theta} L(\theta) &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot G_t \right] && \text{REINFORCE} \\
 &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot (G_t - V_{\phi}(s_t)) \right] && \text{REINFORCE with baseline} \\
 &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot Q_{\phi}(s_t, a_t) \right] && \text{Q-value Actor-Critic} \\
 &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot A_{\phi, \phi'}(s_t, a_t) \right] && \text{Advantage Actor-Critic} \\
 &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot (r_{t+1} + \gamma V_{\phi}(s_{t+1}) - V_{\phi}(s_t)) \right] && \text{TD Actor-Critic}
 \end{aligned}$$

여기서 Actor 역할 (action 을 더 좋게 하는 것) 을 하는 것이 $\nabla \log \pi_{\theta}(a_t | s_t)$ 이며, Critic 역할 (value function에 관한 것) 을 하는 것이 그 뒤에 붙은 것들이다. TD Actor-Critic 의 경우는 다음 상황에 대한 정보가 식에 함께 들어간다. (bootstrapping) 때문에 Experience replay 방법을 사용해야 할 것으로 생각할 수 있다.

Advantage Actor-Critic 은 다음과 같은 방식으로 두개의 parameter 를 사용한다. Q 와 V 각각에 parameter 를 사용하는 것을 볼 수 있다.

$$A_{\phi, \phi'}(s_t, a_t) = Q_{\phi}(s_t, a_t) - V_{\phi'}(s_t)$$

TD Actor-Critic 의 경우 Advantage Actor-Critic 에서 parameter 를 조절한 것으로 볼 수도 있고,

REINFORCE with baseline 을 근사한 것으로도 볼 수 있다.

$$\begin{aligned} A_{\pi_\theta}(s_t, a_t) &= Q_{\pi_\theta}(s_t, a_t) - V_{\pi_\theta}(s_t) \\ &= \mathbb{E}_{\pi_\theta} \left[r_{t+1} + \gamma V_{\pi_\theta}(s_{t+1}) \middle| s_t, a_t \right] - V_{\pi_\theta}(s_t) \\ &= \mathbb{E}_{\pi_\theta} \left[r_{t+1} + \gamma V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t) \middle| s_t, a_t \right] \end{aligned}$$

or,

$$G_t - V_\phi(s_t) \approx r_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

이들 중 REINFORCE with baseline 과 TD Actor-Critic 을 각각 기준으로 하여 Critic 과 Actor 가 어떤 방식으로 학습하는지 알아보자.

Critic 다양한 target 을 대상으로 value function 을 근사한다.

- REINFORCE with baseline target G_t 는 ϕ 와 관련이 없다.

$$\begin{aligned} \text{MSE} &\sim (G_t - V_\phi(s_t))^2 \\ \rightarrow \Delta\phi &= \beta(G_t - V_\phi(s_t)) \nabla_\phi V_\phi(s_t) \end{aligned}$$

이 때 β 는 적절한 학습률이다. 따라서 coding 은 다음과 같은 순서로 진행한다.

$$\begin{aligned} \delta_t &\leftarrow G_t - V_\phi(s_t) \\ \phi &\leftarrow \phi + \beta \delta \nabla_\phi V_\phi(s_t) \end{aligned}$$

- TD Actor-Critic target $r_{t+1} + \gamma V_\phi(s_{t+1})$ 는 물론 ϕ 에 의존하지만 미래의 정보를 가져온 것 이므로 상수취급한다.

$$\Delta\phi = \beta(r_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)) \cdot \nabla_\phi V_\phi(s_t)$$

이 때 β 는 적절한 학습률이다. 따라서 coding 은 다음과 같은 순서로 진행한다.

$$\begin{aligned} \delta_t &\leftarrow r_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \\ \phi &\leftarrow \phi + \beta \delta \nabla_\phi V_\phi(s_t) \end{aligned}$$

Actor policy gradient 를 근사한다.

- REINFORCE with baseline : 이전 단원에서 다루었듯이, Monte Carlo 방법을 이용하여 다음의 업데이트를 진행한다.

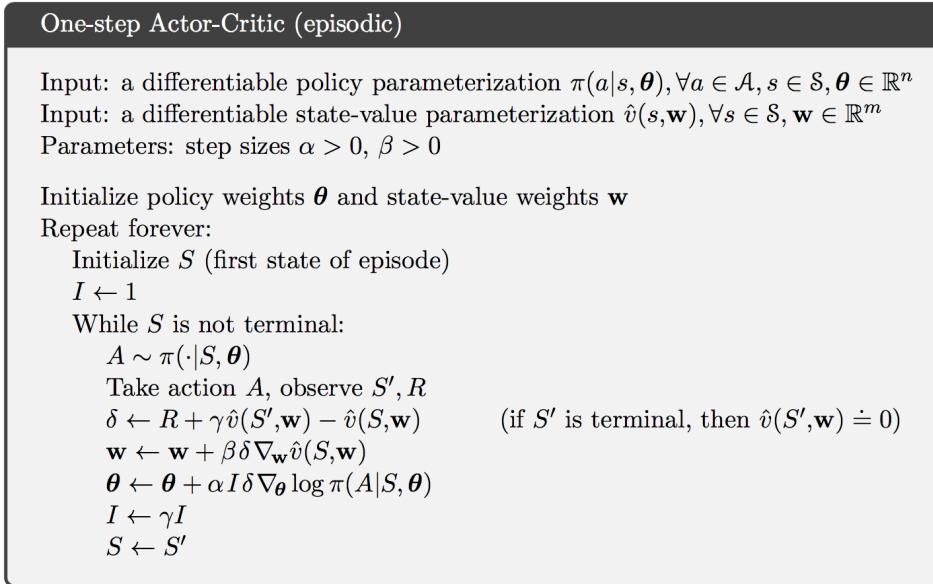
$$\theta = \theta + \alpha \nabla_\theta \mathbb{E}[r(\tau)]$$

이 때 α 는 적절한 학습률이다. δ 는 이미 위에서 정의했으므로 coding 식은 다음과 같다.

$$\theta \leftarrow \theta + \alpha \frac{1}{M} \sum_{i=1}^M \left[\sum_{t=1}^T \delta_t^{(i)} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \right]$$

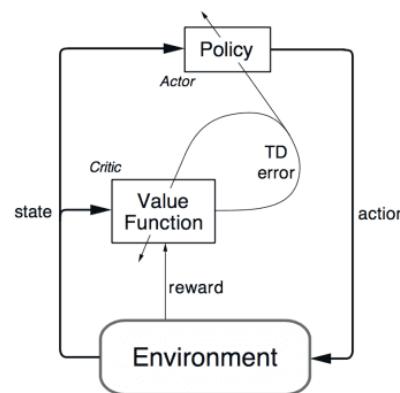
- TD Actor-Critic : target 만 달라졌을 뿐, REINFORCE with baseline 의 식과 완전히 같다.

TD method 를 사용한 one-step Actor-Critic algorithm 이 다음과 같다. 게임이 한번 끝날 때 까지 하는 것이 아닌, 한 step 에 대한 best action 을 찾는 algorithm 이다.



Summary

- REINFORCE with baseline : unbiased 이기 때문에 minimum을 찾을 수 있다. (물론 local인지 global 인지 모름) 하지만 기본적으로 variance 가 크고 게임이 끝날 때 까지 진행해야 하기 때문에 학습이 느리다.
- TD : 학습이 빠르지만 estimation 으로 estimation 하기 때문에 (bootstrapping) 적절한 step 을 조절해줘야 한다.
- Actor-Critic : Critic 으로는 bootstrapping 방식을 사용하여 속도를 높이고 Actor 로는 Policy Gradient 방법을 사용한다. 굳이 network 를 나눌 필요는 없으며 하나의 network 에서 두 값 다 뽑아내도 된다.

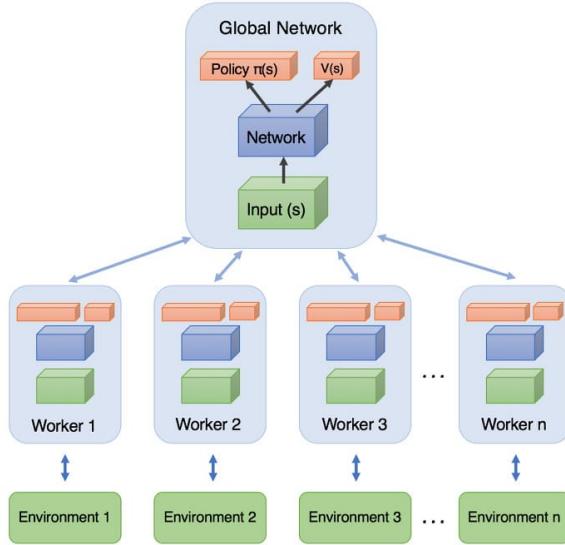


7.4.2 Asynchronous Advantage Actor-Critic (A3C)

A 3개, C 1개 라서 A3C라고 이름 붙었다. Google DeepMind에서 2016년 선보인 기술로, 데이터의 상호연관 (temporal correlation)을 없애기 위해 experience replay를 이용하는 대신 간단한 여러개의 agent들을 병렬연결한 후 이들을 asynchronous 방법으로 학습하는 것이다. 병렬연결된 agent들은

각자 다양한 경험을 하기 때문에 데이터간의 연관성이 사라지고 더 stationary 한 process 를 만든다. action space 가 continuous 해도 잘 작동하기에 DRL 을 한단계 상승시켰다고 볼 수 있다.

A3C 는 Global Network 하부에 병렬로 연결된 agents (실제 게임을 하는 worker) 구조로 되어 있으며 이들은 각자의 environment (게임 emulation 환경) 를 가지고 있다. 우선 Global Network 를 모든 agent에 복사하고, agent 마다 학습을 진행해 계산된 gradient 를 global network로 적용시킨다.

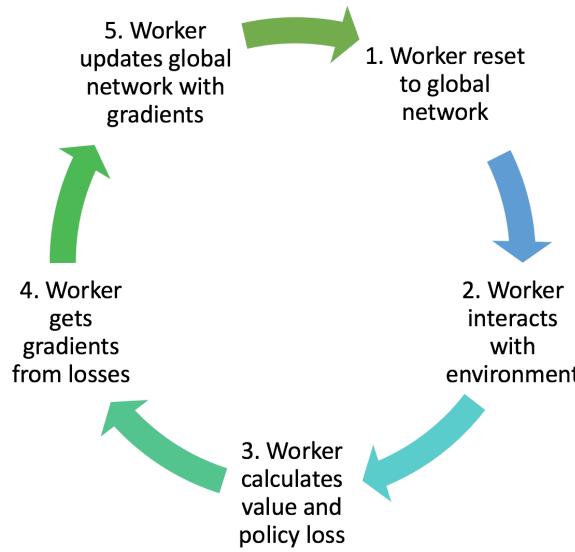


simple RL agent 의 경우에는 (특히 online 학습에서는) non-stationary 였으며 데이터간 연관이 짙었다. Asynchronous 방법은 다음과 같은 장점이 있다:

- agent 들이 각자 독립이기 때문에 시간에 대한 연관이 줄어든다.
- DQN 처럼 experience replay 를 할 필요가 없다. 따라서 메모리와 연산량이 줄어든다.
- 서로 다른 exploration policy 들이 다양한 경험을 준다. 때문에 데이터가 stationary 하고 견고해 진다.
- on-policy RL 도 가능하다.
- 대부분의 DL method 와 달리, GPU 대신 multi-core CPU 를 활용할 수 있다.

Asynchronous Implementation A3C 에서 학습하는 과정은 다음과 같다.

1. 각각의 agent 들은 각각 병렬로 학습한다. global network 는 shared actor and critic parameter θ, ϕ 를 갖고 있으며 agent 들은 asynchronously 이들을 업데이트 한다. 즉, 각 agent 는 자신의 한 batch 가 끝나는대로 업데이트를 진행한다.
2. 각각의 업데이트 사이에, agent 들은 shared network 의 사본을 가지며 local agent policy 에 맞게 적절한 시간 간격 t_{\max} 만큼 시뮬레이션 한다.
3. 시간 간격 t_{\max} 만큼의 시뮬레이션이 끝나면 agent 들은 gradient 를 계산하고 shared parameter 를 업데이트 한다.
4. 해당 방식으로 학습하면 agent 가 늘어날 수록 linear 하게 학습 속도가 늘어난다.



A3C 에서는 advantage actor-critic 방식을 약간 변형하여 사용한다. 즉, target 으로서 advantage function $A(s, a) = Q(s, a) - V(s)$ 를 사용한다. 이는 단순히 agent의 action 이 좋았는지 나빴는지를 평가할 뿐 아니라, 기대한 것에 비해 얼마나 더 좋아졌는지를 평가한다. 즉, advantage 가 커질수록 더 좋은 action 을 선택하게 되는 것이다.

A3C 에서는 일반적인 advantage actor-critic 과 달리 Q 를 계산하지 않고 이를 추정하기 위한 n -step discounted return $G_t^{(n)}$ 을 사용한다. 즉, advantage function 의 추정값은 $G_t^{(n)} - V(S_t)$ 이다. 다음을 살펴보자.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ q_\pi(s, a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \end{aligned}$$

이를 이용하여 다음과 같이 근사한다.

$$\begin{aligned} Q(S_t, A_t) &= \mathbb{E}[G_t | S_t, A_t] \\ &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \cdots | S_t, A_t\right] \\ &\approx R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \mathbb{E}[G_{t+n} | S_{t+n}] \\ &= \sum_{i=0}^{n-1} \gamma^i R_{t+i} + \gamma^n V(S_{t+n}) \end{aligned}$$

다음은 A3C 에 대한 pseudo code 이다.

```

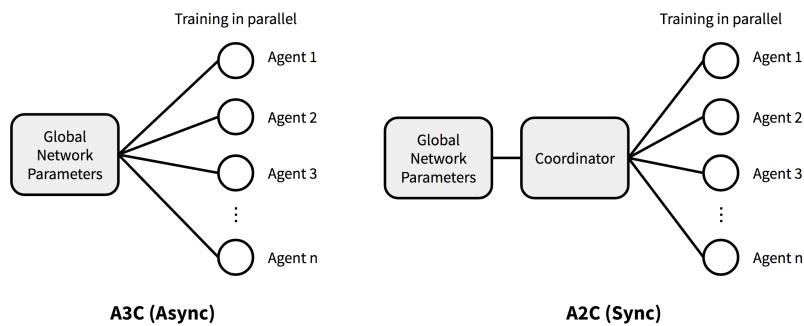
// Assume global shared parameter vectors  $\theta$  and  $\theta_v$  and global shared counter  $T = 0$ 
// Assume thread-specific parameter vectors  $\theta'$  and  $\theta'_v$ 
Initialize thread step counter  $t \leftarrow 1$ 
repeat
    Reset gradients:  $d\theta \leftarrow 0$  and  $d\theta_v \leftarrow 0$ .
    Synchronize thread-specific parameters  $\theta' = \theta$  and  $\theta'_v = \theta_v$ 
     $t_{start} = t$ 
    Get state  $s_t$ 
    repeat
        Perform  $a_t$  according to policy  $\pi(a_t | s_t; \theta')$ 
        Receive reward  $r_t$  and new state  $s_{t+1}$ 
         $t \leftarrow t + 1$ 
         $T \leftarrow T + 1$ 
    until terminal  $s_t$  or  $t - t_{start} == t_{max}$ 
     $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t; \theta'_v) & \text{for non-terminal } s_t // \text{Bootstrap from last state} \end{cases}$ 
    for  $i \in \{t - 1, \dots, t_{start}\}$  do
         $R \leftarrow r_i + \gamma R$ 
        Accumulate gradients wrt  $\theta'$ :  $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i | s_i; \theta')(R - V(s_i; \theta'_v))$ 
        Accumulate gradients wrt  $\theta'_v$ :  $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i; \theta'_v))^2 / \partial \theta'_v$ 
    end for
    Perform asynchronous update of  $\theta$  using  $d\theta$  and of  $\theta_v$  using  $d\theta_v$ .
until  $T > T_{max}$ 

```

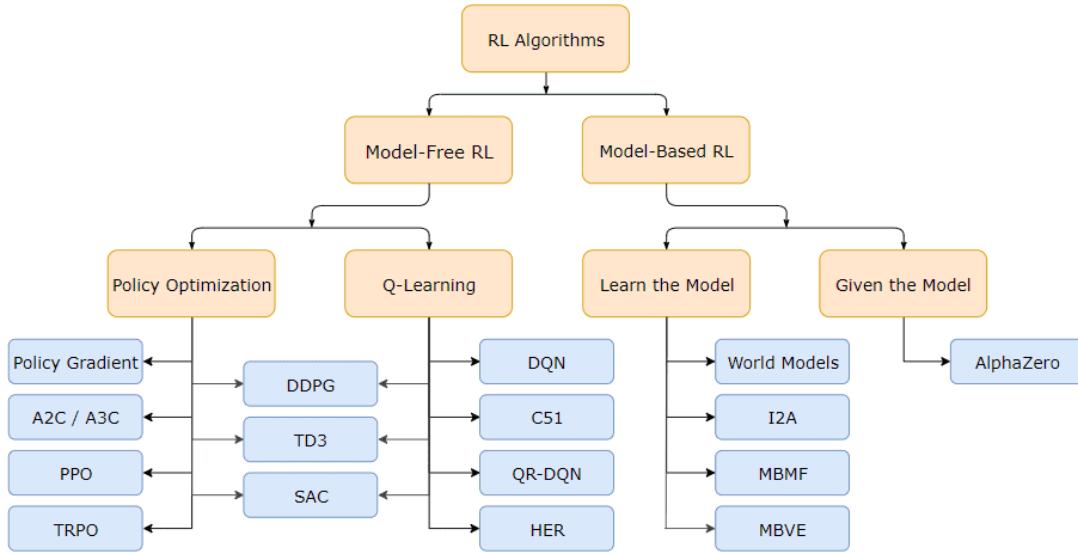
이를 보면 advantage function 을 추정하기 위한 step n 은 upper bound t_{max} 를 갖는다는 것을 알 수 있다. 또한 advantage actor critic 에서 사용했던 actor-critic optimization 을 그대로 사용하는 것을 알 수 있다. 각각의 worker 는 계산한 gradient 를 최종적으로 Global Network 에 적용한다.

Synchronous Deterministic Actor-Critic (A2C) A3C 를 synchronous, deterministic 하게 바꾼 방법이다. A3C 에서는 각각의 agent 들이 독립적으로 다른 policy 를 사용하여 global parameter 를 업데이트 하기 때문에 결과적으로 집계된 parameter 가 optimal 이 아닐 가능성이 있다.

A2C 에서는 **coordinator** 를 사용하여 agent 들이 모두 gradient 를 계산할 때까지 기다리고 모두가 끝나면 gradient 의 합을 global parameter 에 업데이트 시킨 뒤 다시 agent 들에게 작업을 할당해준다. 이는 training 을 더 일관적으로 만들며 빠르게 converge 하도록 한다. 이 방법은 GPU 를 사용함으로서 A3C 보다 훨씬 빠르고 정확하게 동작한다.



7.5 Policy Gradient DRL



7.5.1 Deterministic Policy Gradient (DPG)

Silver et al. (DeepMind 2014) 는 continuous action space 에도 적용할 수 있는 DPG 모델을 발표했다. Stochastic Policy Gradient 의 경우에는 stochastic policy probability function $\pi(a|s)$ 을 state 와 action 모두에서 적분하는 형태였다.

$$\nabla_{\theta} \mathbb{E}_{\theta}[r(\tau)] = \mathbb{E}_{\theta} \left[r(\tau) \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right]$$

이를 Q 에 대해 바꾸면 다음과 같다.

(Stochastic) Policy Gradient Theorem (Sutton et al., 1999)

Let $s \sim \rho_{\pi_{\theta}}$ and $a \sim \pi_{\theta}$. Object function and its gradient becomes:

$$\begin{aligned} L(\theta) &= \mathbb{E}_{\theta} [Q_{\pi_{\theta}}(s, a)] \\ &= \int \rho_{\pi_{\theta}}(s) \int \pi_{\theta}(a|s) Q_{\pi_{\theta}}(s, a) da ds \\ \nabla_{\theta} L(\theta) &= \int \rho_{\pi_{\theta}}(s) \int Q_{\pi_{\theta}}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) da ds \\ &= \mathbb{E}_{\theta} [Q_{\pi_{\theta}}(s, a) \nabla_{\theta} \log \pi_{\theta}(a|s)] \end{aligned}$$

Stochastic Policy Gradient Update :

$$\therefore \Delta \theta = Q_{\pi_{\theta}}(s, a) \nabla_{\theta} \log \pi_{\theta}(a|s)$$

DPG 의 경우에는 deterministic policy $a = \mu(s)$ (즉, 특정 state에 대해 확정된 action 존재) 를 state space 에서만 적분한다. 때문에 DPG 는 action space, state space 모두 연속이어야 된다.

Deterministic Policy Gradient Theorem (Silver et al., 2014)

Let $s \sim \rho_\mu$ and $a = \mu_\theta(s)$. Object function and its gradient becomes:

$$\begin{aligned} L(\theta) &= \mathbb{E}_\mu [Q_\mu(s, a)] \\ &= \int \rho_\mu(s) Q_\mu(s, a) ds \\ \nabla_\theta L(\theta) &= \int \rho_\mu(s) \nabla_\theta Q_\mu(s, a) ds \\ &= \int \rho_\mu(s) \nabla_a Q_\mu(s, a) \nabla_\theta \mu_\theta(s) ds \\ &= \mathbb{E}_\mu [\nabla_a Q_\mu(s, a)|_{a=\mu_\theta(s)} \cdot \nabla_\theta \mu_\theta(s)] \end{aligned}$$

Stochastic Policy Gradient Update :

$$\therefore \Delta\theta = \nabla_a Q_\mu(s, a)|_{a=\mu_\theta(s)} \cdot \nabla_\theta \mu_\theta(s)$$

$\rho_\mu(s')$ 은 stochastic process 의 일반적인 state distribution 이 아니라, γ 로 그 값을 줄인 **Discounted state distribution** 이다.

$$\rho_\mu(s') = \int \sum_{k=1}^{\infty} \gamma^{k-1} p_1(s) p^\mu(s \rightarrow s'; k) ds$$

이 때 $p_1(s)$ 는 초기에 state s 로 시작할 확률, $p^\mu(s \rightarrow s'; k)$ 는 state s 에서 시작해 policy μ 를 따랐을 때 k 번 째 step 에서 s' 을 방문할 확률이다.

드론 control 을 위해 DRL 을 사용한다고 하자. 기존 방법들의 경우 각 드론의 action 을 discrete 하게 자르는 방법을 생각해 볼 수 있다. 하지만 너무 잘게 자르면 space 가 너무 커져 학습을 진행할 수 없고, space 를 너무 작게 만들면 충분한 특성을 반영할 수 없다.

DPG 는 continuous action space 를 그대로 사용하기 때문에 이런 제약을 신경쓰지 않아도 된다. 이는 DQN 과 달리 maximum 을 계산하지 않아도 되는 Actor-Critic 방식을 사용하며 Q 가 미분 가능하기 때문에 Stochastic PG 보다 적은 sample 이 필요하기 때문에 가능하다.

Deep Deterministic Policy Gradient (DDPG) DeepMind 에서 2016 년 발표한, DPG 를 기본으로 한, model-free, off-policy 방법이며 DQN 을 continuous action space 로 가지고 온 것이라 볼 수 있다. DPG 와 마찬가지로 DDPG 는 action value function $Q(s, a)$ 의 expected gradient 를 state space 에서 적분함으로서 deterministic policy $\mu_\theta(s)$ 를 학습한다. 한편 DQN 과 마찬가지로 **Experience replay** 와 **Target network** 를 사용한다. 또한 Q -function 을 학습한다.

DDPG 는 Critic update 에 Q -function 을 사용하며 학습된 Q -function 으로 Actor 의 policy update 를 진행한다. Object function 은 다음과 같다.

$$L(\theta) = \mathbb{E}_\mu [Q_\mu(s, a)]$$

- Critic $Q_\phi(s, a)$ update (TD example) :

$$\Delta\phi = [r + \gamma \hat{Q}_\phi(s', \hat{\mu}_\theta(s')) - Q_\phi(s, a)] \nabla_\phi Q_\phi(s, a)$$

이는 REINFORCE 에서 다른 식과 정확히 같다. Target 은 \hat{Q} 모양으로 Target Network 에서 왔음을 강조하며, prime ' 표시는 $t + 1$ 을 의미한다.

- Actor $\mu_\theta(s)$ update(DPG) :

$$\Delta\theta = \nabla_a Q_\phi(s, a)|_{a=\mu_\theta(s)} \cdot \nabla_\theta \mu_\theta(s)$$

DDPG는 또한 target network를 천천히 변화시키기 위해 **soft update** 방식을 사용한다.

$$\begin{aligned}\hat{\phi} &\leftarrow \tau\phi + (1 - \tau)\hat{\phi} \\ \hat{\theta} &\leftarrow \tau\theta + (1 - \tau)\hat{\theta}\end{aligned}$$

DDPG는 deterministic policy 방식으로, ϵ -greedy 방법은 사용할 수 없다. 대신, exploration 성질을 늘리기 위해 noise \mathcal{N} 을 확률 ϵ 으로 추가해 **exploration policy** μ' 을 사용한다. policy가 deterministic이라서, 적절한 noise는 필수적이다.

$$a_t = \mu'(s_t) = \mu_\theta(s_t) + \mathcal{N}$$

다음은 DDPG의 pseudo code이다.

```

Randomly initialize critic network  $Q(s, a|\theta^Q)$  and actor  $\mu(s|\theta^\mu)$  with weights  $\theta^Q$  and  $\theta^\mu$ .
Initialize target network  $Q'$  and  $\mu'$  with weights  $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$ 
Initialize replay buffer  $R$ 
for episode = 1, M do
    Initialize a random process  $\mathcal{N}$  for action exploration
    Receive initial observation state  $s_1$ 
    for t = 1, T do
        Select action  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  according to the current policy and exploration noise
        Execute action  $a_t$  and observe reward  $r_t$  and observe new state  $s_{t+1}$ 
        Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $R$ 
        Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $R$ 
        Set  $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ 
        Update critic by minimizing the loss:  $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$ 
        Update the actor policy using the sampled policy gradient:
```

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

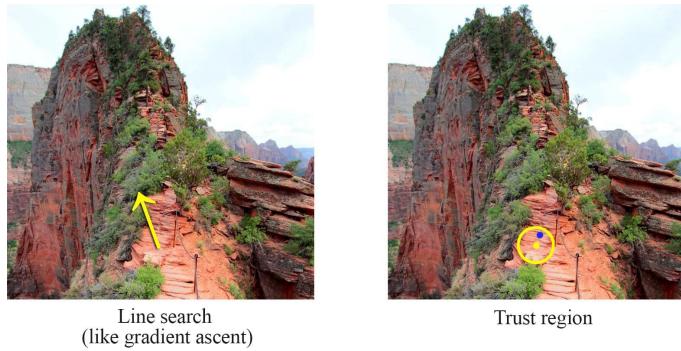
Update the target networks:

$$\begin{aligned}\theta^{Q'} &\leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}\end{aligned}$$

end for
end for

7.5.2 Trust Region Policy Optimization (TRPO)

DDPG는 continuous action space에서도 학습할 수 있다는 가능성을 열어주었지만, policy가 계속 좋아지는 보장이 있는 것은 아니었다. TRPO는 KL-divergence 조건 영역에 있는 **trust region**을 차용함으로서 이 문제를 해결했다. 즉, trust region 안에서 업데이트를 진행하면 성능이 상승한다는 보장이 된다는 것이다. 따라서 새로운 policy parameter를 업데이트 할 때 trust region 안에서 잡는다.



직관적으로 보자면 위의 그림과 같다. Line search 의 경우 step size 가 크면 학습이 잘못 될 가능성이 있다. 이는 gradient descent 에서 학습률을 너무 크게 한 경우와 비슷하다. 때문에 우리는 Trust region 을 설정해 그 안에서만 학습하게 된다. 수학적으로, 이 방법은 minimum 을 찾는 것이 보장된다. (물론 local 인지 global 인지 모른다.)

우리는 우선 최종결과를 직관적으로 이해하기 위해 DPG 에서 다른 Stochastic Policy Gradient Theorem 으로부터 출발한다. 우선 사용하는 function 을 advantage function 으로 바꾸면 $s \sim \rho_\pi$, $a \sim \pi$ 라고 할 때 objective function 과 gradient 는 다음과 같다.

$$\begin{aligned} L(\theta) &= \mathbb{E}_\theta [A_{\pi_\theta}(s, a)] \\ &= \sum_s \rho_\pi(s) \sum_a \pi_\theta(a|s) A_\pi(s, a) \\ \nabla_\theta L(\theta) &= \sum_s \rho_\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) A_\pi(s, a) \\ &= \sum_s \rho_\pi(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) A_\pi(s, a) \\ &= \mathbb{E}_\theta [\nabla_\theta \log \pi_\theta(a|s) A_\pi(s, a)] \end{aligned}$$

한편, 다음과 같은 objective function 을 고려해보자.

$$L(\theta) = \mathbb{E}_q \left[\frac{\pi_\theta(a|s)}{\pi_q(a|s)} A_{\pi_\theta}(s, a) \right]$$

어딘가 익숙하지 않은가? 바로 위에서 다른 Stochastic Policy Gradient Theorem 의 식을 q 로 Importance Sampling 한 것 이다. 식이 정확히 같기 때문에 당연히 gradient 는 같다:

$$\nabla_\theta \log \pi_\theta(a|s) \Big|_{\theta=q} = \frac{\nabla_\theta \pi_\theta(a|s) \Big|_{\theta=q}}{\pi_q(a|s)} = \nabla_\theta \left(\frac{\pi_\theta(a|s)}{\pi_q(a|s)} \right) \Big|_q$$

즉, Important Sampling 된 objective function 을 maximize 시키는 방향으로 학습을 진행해도 과정은 변하지 않는다. 이제 여기서 objective function 이 급격하게 변하는 것을 막기 위한 trust region 을 추가한다. 결과적으로 TRPO 의 objective function 은 다음과 같다.

$$\max_\theta \mathbb{E} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_\theta}(s, a) \right]$$

where $\mathbb{E} \left[D_{\text{KL}} [\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_\theta(\cdot|s)] \right] \leq \delta$

이렇게 KL Divergence 조건을 넣게 되면 새로운 policy π_θ 는 이전 policy $\pi_{\theta_{\text{old}}}$ 에 비해 급격하게 변하지 않는다. 왜 이렇게 했을까? advantage function은 policy가 얼마나 더 좋게 변하는지에 대한 직접적인 척도이기 때문에 advantage function에는 제한을 두기 않는다. 따라서 policy가 급격하게 변하는 것을 막기 위해, 확률분포의 급격한 변화를 막는다. 한편, Lagrangian duality principle을 이용하면 다음과 같이 한줄의 식으로 정리된다.

$$\max_{\theta} \mathbb{E} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_\theta}(s, a) \right] - \beta \mathbb{E} [D_{\text{KL}}[\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_\theta(\cdot|s)]]$$

이제 논문에 적힌 사고의 과정을 따라가보도록 하자. DRL의 목적은 expected return $\eta(\pi)$ 를 maximize하는 것이다. $\tau = \{s_0, a_0, \dots\}$ 라고 하자.

$$\eta(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t) \right]$$

한편 다음과 같이 식변환을 통해 old policy π_{old} 로부터 new policy π 에 대한 $\eta(\pi)$ 를 얻을 수 있다.

Preliminary for TRPO

$$\eta(\pi) = \eta(\pi_{\text{old}}) + \sum_s \rho_\pi(s) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a)$$

where

$$\rho_\pi(s) = \mathbb{P}(s_0 = s) + \gamma \mathbb{P}(s_1 = s) + \gamma^2 \mathbb{P}(s_2 = s) + \dots$$

proof :

1.

$$\begin{aligned} \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right] &= \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t [r_{t+1} + \gamma V_{\pi_{\text{old}}}(s_{t+1}) - V_{\pi_{\text{old}}}(s_t)] \right] \\ &= \eta(\pi) + \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^{t+1} V_{\pi_{\text{old}}}(s_{t+1}) - \sum_{t=0}^{\infty} \gamma^t V_{\pi_{\text{old}}}(s_t) \right] \\ &= \eta(\pi) - \mathbb{E}[V_{\pi_{\text{old}}}(s_0)] \\ &= \eta(\pi) - \eta(\pi_{\text{old}}) \\ \therefore \quad \eta(\pi) &= \eta(\pi_{\text{old}}) + \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right] \end{aligned}$$

2.

$$\begin{aligned} \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right] &= \sum_{t=0}^{\infty} \mathbb{E}_{\tau \sim \pi} [\gamma^t A_{\pi_{\text{old}}}(s_t, a_t)] \\ &= \sum_{t=0}^{\infty} \sum_s \mathbb{P}(s_t = s | \pi) \sum_a \pi(a|s) \gamma^t A_{\pi_{\text{old}}}(s, a) \\ &= \sum_s \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s | \pi) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a) \\ &= \sum_s \rho_\pi(s) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a) \end{aligned}$$

$$\therefore \eta(\pi) = \eta(\pi_{\text{old}}) + \sum_s \rho_\pi(s) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a)$$

하지만 이 식 중 $\rho_\pi(s)$ 는 π 에 의존한다. 즉, old policy로 new policy를 계산해야 하는데 그를 위해서 new policy가 필요하다는 것이다. 때문에 우리는 해당 항을 $\rho_{\pi_{\text{old}}}(s)$ 으로 대체하여 사용한다. 추후에 trust region으로 제한을 걸기 때문에 두 policy에 따른 분포는 크게 달라지지 않을 것으로 예상한다. 최종적으로 사용하는 loss function은 다음과 같다.

$$L_{\pi_{\text{old}}}(\pi) = \eta(\pi_{\text{old}}) + \sum_s \rho_{\pi_{\text{old}}}(s) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a)$$

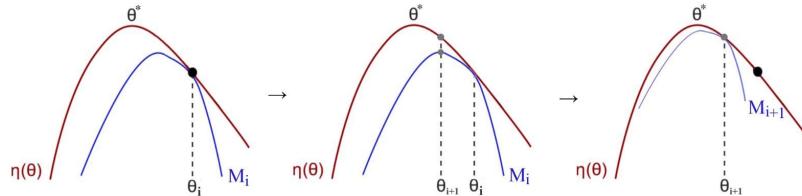
이 때 $\eta(\pi_{\text{old}})$ 는 고정값이기 때문에 우리는 다음과 같이 π' 을 찾는다.

$$\pi' = \operatorname{argmax}_\pi L_{\pi_{\text{old}}}(\pi) = \operatorname{argmax}_\pi \left[\sum_s \rho_{\pi_{\text{old}}}(s) \sum_a \pi(a|s) \right]$$

이것이 우리가 처음에 살펴본 objective function이다. 그 우리는 soft update 방식을 사용한다.

$$\pi_{\text{new}}(a|s) = (1 - \alpha)\pi_{\text{old}}(a|s) + \alpha\pi'(a|s)$$

이제 어떻게 KL Divergence 조건을 넣었는지 살펴보자. 우선 Minorization-Maximization (MM) algorithm을 알아야 한다. 이는 $\eta(\theta)$ 의 최대점을 찾는 알고리즘이다.



즉, 우리는 결과적으로 빨간 선에서의 최대값을 찾고 싶다. 각각의 iteration에서 우리는 파란 선, surrogate objective function M_i 를 찾아야 한다. 그 과정은 다음과 같다.

1. surrogate objective function은 다음 세 가지 조건을 만족해야 한다.

- 현재 policy parameter θ_i 에서 objective function η 를 근사하도록 한다.
- surrogate function은 objective function η 의 lower bound이어야 한다.
- η 보다 최적화하기 쉬워야 한다. (이를테면 2차함수)

2. 찾은 surrogate objective function M_i 의 최대점 θ_{i+1} 을 찾는다.

3. θ_{i+1} 를 이용하여 surrogate objective function M_{i+1} 을 찾는다.

위의 과정을 반복하면 policy는 지속적으로 높은 η 값을 갖게 된다. 그런데 policy는 유한하므로, 언젠가는 optimal 값으로 수렴할 것을 알 수 있다. 물론 global인지 local인지는 모른다.

논문에서는 다음과 같은 surrogate objective function M_i 를 제안했다 :

$$M_i(\pi) = L_{\pi_i}(\pi) - C \cdot D_{\text{KL}}^{\max}(\pi_i, \pi)$$

where

$$\begin{aligned}
 L_{\pi_i}(\pi) &= \eta(\pi_i) + \sum_s \rho_{\pi_i}(s) \sum_a \pi(a|s) A_{\pi_i}(s, a) \\
 C &= \frac{4\epsilon\gamma}{(1-\gamma)^2} \\
 D_{\text{KL}}^{\max}(\pi_i, \pi) &= \max_s D_{\text{KL}}[\pi_i(\cdot|s) || \pi(\cdot|s)] \\
 \epsilon &= \max_{s,a} |A_{\pi}(s, a)|
 \end{aligned}$$

위의 surrogate objective function ⚡] monotonic policy improvement 를 보장한다는 것을 증명해보자.

Monotonic Improvement Guarantee for General Stochastic Policies

$$\begin{aligned}
 \eta(\pi_i) &= M_i(\pi_i) \quad \text{and} \quad \eta(\pi) \geq M_i(\pi) \\
 \therefore \eta(\pi_{i+1}) &\geq \eta(\pi_i)
 \end{aligned}$$

proof :

1. 우선 Preliminary 증명의 1번 식을 가져오면 다음과 같다.

$$\eta(\pi) = \eta(\pi_{\text{old}}) + \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right]$$

또한 Preliminary 증명의 2번 과정 (ρ 의 정의를 사용한다) 을 loss function 에 적절히 사용하면 다음과 같다.

$$\begin{aligned}
 L_{\pi_{\text{old}}}(\pi) &= \eta(\pi_{\text{old}}) + \sum_s \rho_{\pi_{\text{old}}}(s) \sum_a \pi(a|s) A_{\pi_{\text{old}}}(s, a) \\
 &= \eta(\pi_{\text{old}}) + \sum_{t=0}^{\infty} \sum_s \mathbb{P}(s_t = s | \pi_{\text{old}}) \sum_a \pi(a|s) \gamma^t A_{\pi_{\text{old}}}(s, a) \\
 &= \eta(\pi_{\text{old}}) + \mathbb{E}_{\tau \sim \pi_{\text{old}}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right]
 \end{aligned}$$

2. 한편, n_t 를 $i < t$ 일 때 $a_i \neq a_i^{\text{old}}$ 인 회수라고 하자. 이제 stochastic process 에서 자주 사용한 기대값 쪼개기를 해보자.

$$\begin{aligned}
 \mathbb{E}_{\tau \sim \pi} [A_{\pi_{\text{old}}}(s_t, a_t)] &= \mathbb{P}[n_t = 0] \mathbb{E}_{\tau \sim \pi | n_t=0} [A_{\pi_{\text{old}}}(s_t, a_t)] + \mathbb{P}[n_t > 0] \mathbb{E}_{\tau \sim \pi | n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)] \\
 \mathbb{E}_{\tau \sim \pi_{\text{old}}} [A_{\pi_{\text{old}}}(s_t, a_t)] &= \mathbb{P}[n_t = 0] \mathbb{E}_{\tau \sim \pi_{\text{old}} | n_t=0} [A_{\pi_{\text{old}}}(s_t, a_t)] + \mathbb{P}[n_t > 0] \mathbb{E}_{\tau \sim \pi_{\text{old}} | n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)]
 \end{aligned}$$

$n_t = 0$ 이라는 것은 $i < t$ 인 i 모두에 대해 $a_i = a_i^{\text{old}}$ 라는 뜻이다. 때문에 위의 두 식을 빼면 다음과 같다.

$$\mathbb{E}_{\pi} [A_{\pi_{\text{old}}}(s_t, a_t)] - \mathbb{E}_{\pi_{\text{old}}} [A_{\pi_{\text{old}}}(s_t, a_t)] = \mathbb{P}[n_t > 0] [\mathbb{E}_{\pi | n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)] - \mathbb{E}_{\pi_{\text{old}} | n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)]]$$

3. (π, π_{old}) 가 α -coupled policy pair (def. $\mathbb{P}(a \neq a^{\text{old}}|s) \leq \alpha$ for all s) 라면 $\mathbb{P}[\pi, \pi_{\text{old}}$ agree at timestep $i] \geq 1 - \alpha$ 이며 따라서 다음과 같다.

$$\mathbb{P}[n_t > 0] \leq 1 - (1 - \alpha)^t$$

4. 또한 (π, π_{old}) 가 α -coupled policy pair 라면 advantage function에 대해 다음과 같은 부등식이 성립한다.

$$\begin{aligned} |A_{\pi_{\text{old}}}(s, a)| &= \left| \mathbb{E}_{a \sim \pi} [A_{\pi_{\text{old}}}(s^{\text{old}}, a)] \right| \\ &= \left| \mathbb{E}_{a \sim \pi} [A_{\pi_{\text{old}}}(s^{\text{old}}, a)] - \mathbb{E}_{a^{\text{old}} \sim \pi_{\text{old}}} [A_{\pi_{\text{old}}}(s^{\text{old}}, a^{\text{old}})] \right| \\ &= \left| \mathbb{P}[a \neq a^{\text{old}}|s] \mathbb{E}_{a \sim \pi, a^{\text{old}} \sim \pi_{\text{old}}} [A_{\pi_{\text{old}}}(s^{\text{old}}, a) - A_{\pi_{\text{old}}}(s^{\text{old}}, a^{\text{old}})] \right| \\ &\leq \alpha \cdot 2 \max_{s, a} |A_{\pi}(s, a)| \end{aligned}$$

5. 2 에 3 과 4 를 적용하면 다음과 같다.

$$\begin{aligned} \left| \mathbb{E}_{\pi} [A_{\pi_{\text{old}}}(s_t, a_t)] - \mathbb{E}_{\pi_{\text{old}}} [A_{\pi_{\text{old}}}(s_t, a_t)] \right| &\leq \mathbb{P}[n_t > 0] \left[\left| \mathbb{E}_{\pi|n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)] \right| \right. \\ &\quad \left. + \left| \mathbb{E}_{\pi_{\text{old}}|n_t>0} [A_{\pi_{\text{old}}}(s_t, a_t)] \right| \right] \\ &\leq 4\alpha(1 - (1 - \alpha)^t) \max_{s, a} |A_{\pi}(s, a)| \end{aligned}$$

6. 1 의 두 식을 빼고 5 를 적용하면 다음과 같다.

$$\begin{aligned} |\eta(\pi) - L_{\pi_{\text{old}}}(\pi)| &= \left| \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right] - \mathbb{E}_{\tau \sim \pi_{\text{old}}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi_{\text{old}}}(s_t, a_t) \right] \right| \\ &\leq \sum_{t=0}^{\infty} \gamma^t \left| \mathbb{E}_{\tau \sim \pi} [A_{\pi_{\text{old}}}(s_t, a_t)] - \mathbb{E}_{\tau \sim \pi_{\text{old}}} [A_{\pi_{\text{old}}}(s_t, a_t)] \right| \\ &\leq \sum_{t=0}^{\infty} \gamma^t 4\epsilon\alpha(1 - (1 - \alpha)^t) \\ &= 4\epsilon\alpha \left(\frac{1}{1 - \gamma} - \frac{1}{1 - \gamma(1 - \alpha)} \right) \\ &= \frac{4\epsilon\alpha^2\gamma^2}{(1 - \gamma)(1 - \gamma(1 - \alpha))} \\ &\leq \frac{4\epsilon\gamma}{(1 - \gamma)^2} \alpha^2 \end{aligned}$$

7. (Levin et al., 2009) Proposition 4.7 에 의하면 $\alpha = D_{\text{TV}}[\pi_i(\cdot|s)||\pi(\cdot|s)]$ 로 바꿔도 되며, (Pollard, 2000) Ch.3 에 의하면 $\alpha^2 \leq D_{\text{KL}}[\pi_i(\cdot|s)||\pi(\cdot|s)]$ 이므로 다음과 같은 부등식이 성립한다.

$$|\eta(\pi) - L_{\pi_{\text{old}}}(\pi)| \leq \frac{4\epsilon\gamma}{(1 - \gamma)^2} D_{\text{KL}}[\pi_i(\cdot|s)||\pi(\cdot|s)]$$

최종적으로 우리는 surrogate objective function 을 얻는다.

$$\eta(\pi) \leq L_{\pi_{\text{old}}}(\pi) - \frac{4\epsilon\gamma}{(1 - \gamma)^2} D_{\text{KL}}[\pi_i(\cdot|s)||\pi(\cdot|s)]$$

이로서 수학적으로 그 성능이 보장된 알고리즘을 얻었다. 위의 결과를 요약하면 다음과 같다.

```

Initialize  $\pi_0$ .
for  $i = 0, 1, 2, \dots$  until convergence do
    Compute all advantage values  $A_{\pi_i}(s, a)$ .
    Solve the constrained optimization problem
        
$$\pi_{i+1} = \arg \max_{\pi} [L_{\pi_i}(\pi) - CD_{\text{KL}}^{\max}(\pi_i, \pi)]$$

        where  $C = 4\epsilon\gamma/(1-\gamma)^2$ 
        and  $L_{\pi_i}(\pi) = \eta(\pi_i) + \sum_s \rho_{\pi_i}(s) \sum_a \pi(a|s) A_{\pi_i}(s, a)$ 
end for

```

하지만 위에서 구한 penalty coefficient C 를 그대로 사용하면 학습하는 step size 가 너무 작다. 때문에 우리는 KL divergence 조건을 따로 떼어 다음과 같이 Trust region constraint 를 만든다.

$$\max_{\theta} L_{\theta_{\text{old}}}(\theta) \quad \text{where} \quad D_{\text{KL}}^{\max}(\theta_{\text{old}} || \theta) \leq \delta$$

하지만 이렇게 하게 되면 KL divergence 의 max 값을 구하기 위해 계산해야 하는 양이 너무 많다. 대신에 우리는 **heuristic approximation** 으로 KL divergence 의 평균값을 사용하도록 한다. 또한 계산의 편의를 위해 Importance Sampling 된 모양을 사용하도록 한다. 결과적으로 다음과 같은 식이 만들어진다.

$$\begin{aligned} & \max_{\theta} \mathbb{E}_{s \sim \rho_{\theta_{\text{old}}}, a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_{\theta}}(s, a) \right] \\ & \text{s.t.} \quad \mathbb{E}_{s \sim \rho_{\theta_{\text{old}}}} \left[D_{\text{KL}}[\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_{\theta}(\cdot|s)] \right] \leq \delta \end{aligned}$$

이제 마지막 단계만이 남아있다. 이전에 언급했듯이, M_i 는 optimize 하기 쉬워야 한다. 그러나 아직까지도 연산량이 너무 많다. 때문에 $L_{\theta_{\text{old}}}(\theta)$ 를 Taylor 전개하여 1차항까지 사용하고, KL divergence 는 2차항까지 사용한다.

$$\begin{aligned} & \max_{\theta} \nabla_{\theta} L_{\theta_{\text{old}}}(\theta) \Big|_{\theta=\theta_{\text{old}}} (\theta - \theta_{\text{old}}) \\ & \text{s.t.} \quad \frac{1}{2} (\theta_{\text{old}} - \theta)^T H (\theta_{\text{old}} - \theta) \leq \delta \end{aligned}$$

where the **Fisher Information Matrix (FIM)** in the form of Hessian H is

$$H = \nabla^2 \mathbb{E}_{s \sim \rho_{\theta_{\text{old}}}} \left[D_{\text{KL}}[\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_{\theta}(\cdot|s)] \right] = \frac{\partial^2}{\partial_i \partial_j} \mathbb{E}_{s \sim \rho_{\theta_{\text{old}}}} \left[D_{\text{KL}}[\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_{\theta}(\cdot|s)] \right]$$

논문에서는 FIM 을 간접적으로 계산하기 위해 **Conjugate Gradient (CG)** algorithm 을 사용했다. 하지만 여기까지 노력했음에도 그 적용과 계산은 매우 어렵다.

7.5.3 Proximal Policy Optimization (PPO)

TRPO 는 성능이 좋지만, 실재 적용하고 코딩을 하기가 너무 난해하다. 이는 KL-divergence 조건 때문인데, PPO 는 이를 **clipped surrogate objective function** 을 통해 대체한다. 이는 계산량이 TRPO 보다 매우 적고 구현하기 쉽기 때문에 보통 이것을 사용한다.

TRPO 의 surrogate objective function 은 결과적으로 다음과 같았다.

$$\max_{\theta} \mathbb{E} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_{\theta}}(s, a) \right]$$

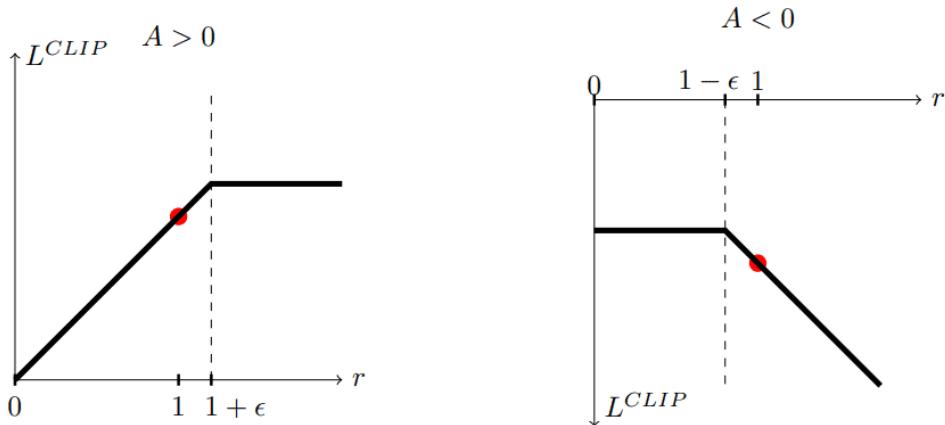
where $\mathbb{E} \left[D_{\text{KL}} [\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_{\theta}(\cdot|s)] \right] \leq \delta$

즉, policy distribution 이 급변하는 것을 방지하기 위해 KL divergence 조건을 넣은 것 이었다. PPO 는 목적이 같지만 KL divergence 를 사용하지 않고 clipping 방식을 사용하여 policy distribution 이 급변하는 것을 제한한다. 즉, policy distribution 비율을 $1 - \epsilon$ 과 $1 + \epsilon$ 사이로 제한한다. 그 objective function 은 다음과 같다.

$$L_t(\theta) = \mathbb{E}_t \left[\min \left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t, \text{clip} \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right] \cdot A_t \right) \right]$$

중앙값을 1 로 두고 $1 - \epsilon$ 과 $1 + \epsilon$ 의 값으로 clipping 하는 것은 $\theta = \theta_{\text{old}}$ 일 때 policy distribution 비율 값이 1 이기 때문이다. 이 objective function 은 TRPO 의 것의 lower bound 이기 때문에 이론적으로도 문제가 없다.

만약 $A > 0$ 이라면 action 이 좋은 것 이기 때문에 이를 반영하는 쪽으로 update 될 것 이고, $A < 0$ 으로 너무 치우친 policy 가 나왔다고 해도 clipping 에 의해 $1 - \epsilon$ 정도로 약간만 이동될 것이다.



논문에서는 이를 적용할 때 squared-error loss $L_t^{VF} = (V_{\theta}(s_t) - V_t^{\text{target}})^2$ 와 entropy bonus $S[\pi_{\theta}]$ 를 추가하여 다음과 같은 최종 식을 사용했다.

$$L_t^{\text{CLIP+VF+S}}(\theta) = \mathbb{E}_t \left[L_t(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_{\theta}](s_t) \right]$$

PPO 의 pseudo code 는 다음과 같다.

```

for iteration=1, 2, ... do
    for actor=1, 2, ..., N do
        Run policy  $\pi_{\theta_{\text{old}}}$  in environment for T timesteps
        Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
    end for
    Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
     $\theta_{\text{old}} \leftarrow \theta$ 
end for

```

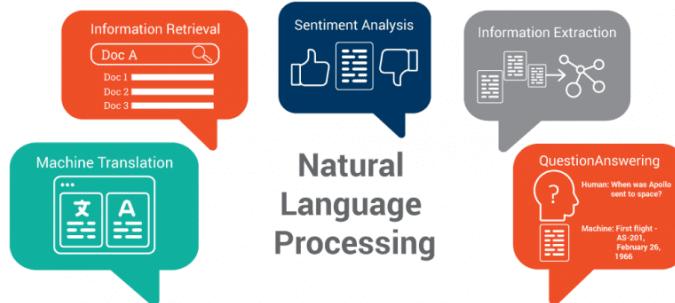
Summaray

Algorithm	Description	Model	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Model-Free	Off-policy	Discrete	Discrete	Sample-means
Q-learning	State-action-reward-state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State-action-reward-state-action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State-action-reward-state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State-action-reward-state-action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	Off-policy	Continuous	Continuous	Q-value
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage

8 Lecture 8 : Natural Language Processing

8.1 Latural Language Processing (NLP)

Natural Language Processing artificial intelligence 의 집약체로, 사람의 언어를 이해하고 재생산하며, computer 와 말하고 쓰는 방식으로 소통하기 위한 방법이다. 이는 computer science, artificial intelligence, computer language 와 관련이 있다.



NLP 는 다음과 같은 이유로 매우 어렵다.

- 하나의 input이 서로 다른 많은 의미를 가지고 있을 수 있다.
- 많은 서로 다른 input이 모두 같은 의미를 가지고 있을 수 있다.
- input의 component들 간의 상호관계가 명확하지 않다.

DL 이 나오기 전에도 NLP 는 연구가 되어왔는데, **bag of words** (document 안에서 나오는 단어가 이미 알고 있는 단어인지, 몇 번 나오는지 - 언어적 구조를 무시한다.) 를 이용하거나 각 단어가 전체 문단에서 가지는 중요도를 생각한다거나 하는 식이었다. 이런 방식은 curse of dimensionality 문제도 있었으며, 가지고 있던 data 안에 해당하는 단어가 없을 수도 있다는 문제도 있었다. 또한 이보다 어려운 문제들은 Machine Learning 의 기법으로서 하나하나 손으로 제작해야 했었다. DL 은 NLP 에 커다란 영향을 주었는데, 우선 단어들을 다루기 쉬운 벡터들로 변환시켜주는 **word2vec** machine 이 등장했다. 여기서부터는 bag of words 를 사용하지 않고 **word embedding** 을 사용하게 되었다. word2vec 은 다음의 두가지 특징을 가진다.

- 각각의 단어는 저차원의 벡터로 표현된다.
- 단어의 유사성은 벡터의 유사성으로 표현된다.

Vector representations of words

- Local (sparse) representation \leftarrow **one-hot encoding**

기존에는 각 단어에 하나의 vector component 를 할당해 단어를 저장했다. 해당되는 단어의 경우 그 component 를 1, 나머지를 0으로 했다. 단어가 많아지면 차원이 이에 비례해서 늘어나기 때문에 learning 에 사용되기는 어렵다.

$$\begin{aligned}
 \text{Rome} &= [1, 0, 0, 0, 0, 0, \dots, 0] \\
 \text{Paris} &= [0, 1, 0, 0, 0, 0, \dots, 0] \\
 \text{Italy} &= [0, 0, 1, 0, 0, 0, \dots, 0] \\
 \text{France} &= [0, 0, 0, 1, 0, 0, \dots, 0]
 \end{aligned}$$

이 경우 각 단어들은 서로 완전히 독립이다. 즉, 다음과 같다.

$$(\mathbf{w}_{\text{Rome}})^T \mathbf{w}_{\text{Paris}} = 0$$

- Distributed (dense) representation \leftarrow **word embedding** (Hinton, 1986)

vocabulary space 는 one-hot encoding 에서 보이듯 높은 차원을 가지고 있다. 이를 훨씬 낮은 차원의 continuous vector vocabulary subspace 로 변환한다. dimension 이 줄었지만 각 component 값이 실수이기 때문에 많은 단어들을 표현할 수 있다.

$$W(\text{"cat"}) = (0.2, -0.4, 0.7, \dots)$$

$$W(\text{"mat"}) = (0.0, 0.6, -0.1, \dots)$$

subspace 를 잘 잡으면 단어들에 들어있는 의미가 잘 반영되게 된다. (물론 그것이 어떤 의미 인지는 모르지만) 이 경우 단어들은 독립이 아닌데, 유사한 단어들끼리는 space 안에서 비슷한 위치에 있게 된다. 즉, 유사한 단어 끼리의 내적은 그 값이 다른 것 보다 크다.

그렇다면 dimension 을 줄이는 과정에서 단어의 유사성을 어떻게 반영시킬 것인가? 다음은 NLP 에서 사용되는 가장 중요한 가정이다.

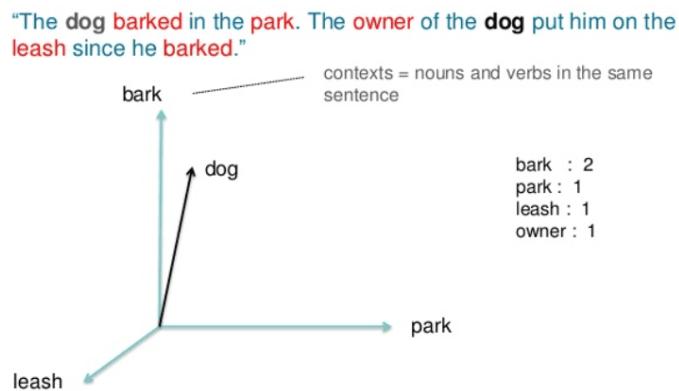
Distributional Hypothesis (Harris, 1954)

Similar words occur in similar contexts

즉, 비슷한 문맥에서 자주 같이 나오는 단어들은 단어의 유사성이 높다고 가정한다. 이 가정 하에 word embedding machine을 학습시킨다고 하자. ‘bycycle’과 ‘car’은 사람이 생각하기에 많이 유사해 보이지만, 아마도 ‘bycycle’과 ‘ride’가 함께 더 많이 나오기 때문에 이들의 유사성이 ‘bycycle’과 ‘car’보다 높을 것이다. 이 가정은 Statistical Semantics (이를테면 distributional semantic model) 연구의 근간이다.

Distributional Semantic Model 수많은 언어 data를 가지고 단어나 문맥 등의 언어 item 사이의 연관성을 밝히고 정리하는 연구분야이다. 다음과 같은 방식을 가진다.

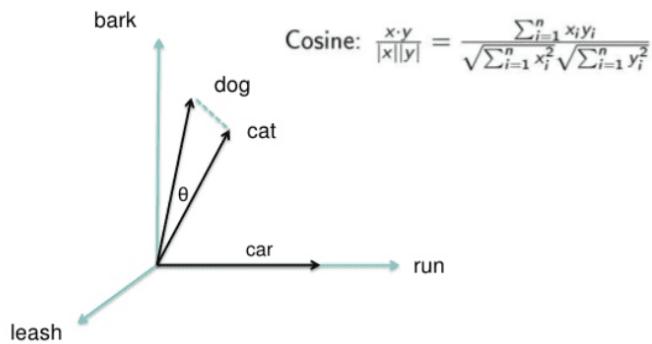
- Pre-process a corpus (target, context) : target은 벡터로 바꿔야 할 대상들, context는 바꾸는 기준들이다. 이를테면 context가 10개의 단어들이라면, target으로 하는 것들을 이 10개에 맞게 벡터로 변형해야 한다. 이를테면 “The god barked in the park. The owner of the dog put him on the leash since he barked”에서 ‘dog’라는 target을 살펴보면 context ‘bark’는 2번 같이 나왔고, ‘park’는 1번, ‘leash’는 1번, ‘owner’는 1번 나왔다.



- Count the target-context co-occurrences : 이러한 방식으로 특정 document를 context로 분석한 결과가 다음과 같다.

		contexts						
		leash	walk	run	owner	leg	bark	
targets		dog	3	5	1	5	4	2
		cat	0	3	3	1	5	0
		lion	0	3	2	0	1	0
		light	0	0	0	0	0	0
		bark	1	0	0	2	1	0
		car	0	0	4	3	0	0

세상에 존재하는 모든 문서들을 이런 방식으로 표를 작성한다고 하자. 이제 target들은 context의 좌표계로 표현된다. 이 때문에 비슷한 단어는 (즉, 동일한 context와 자주 함께 나오는 단어들은) 비슷한 위치의 벡터로 표현되게 된다.



혹은, 다음과 같은 추가 과정을 거치기도 한다.

- Weight the contexts (optional)
- Build the distributional matrix
- Reduce the matrix dimensions (optional)

Statistical language modeling training 으로 사용할 문장 dataset인 **training corpus** (말뭉치)에서 n 개의 단어로 구성된 문장 $s = w_1 w_2 \dots w_n$ 이 가지는 확률 $P(s)$ 를 계산하는 것이다. 이를테면 corpus 에서 ‘i am ...’ 문장이 ‘i must ...’ 보다 자주 많이 나온다면, ‘i’ 가 나왔을 때 그 다음으로 ‘am’ 이 나올 확률이 높을 것이다. (물론 방식에 따라 다르다.)

$$\begin{aligned} P(s) &= P(w_1 w_2 \dots w_n) \\ &= P(w_1) P(w_2 | w_1) P(w_3 | w_1 w_2) \dots P(w_n | w_1 w_2 \dots w_{n-1}) \\ &= \prod_{i=1}^n P(w_i | h_{i-1}) \end{aligned}$$

이 때 $h_{i-1} = w_1 w_2 \dots w_{i-1}$ 이다. 만약 h_{i-1} 이 너무 강하다면, corpus training 과정에서 $h_{i-1} w_i$ 를 관찰할 수 없을 가능성이 있다. 이를테면 corpus 에서 ‘i am a ...’ 라는 문장이 거의 없다고 하자. 그러나 ‘i am a student’ 라는 문장은 분명 있다. 때문에 ‘i am’ 까지 나왔을 때 ‘a student’ 가 이어져 원하는 문장을 얻을 확률은 거의 없을 것이다. 때문에 h_{i-1} 의 길이를 제한한다. 가장 간단한 것은 **n-gram** 으로, h_{i-1} 의 길이를 앞선 $n - 1$ 개의 단어만 가지고록 제한하는 것이다.

- **Uni-gram** : 현재 단어의 확률만 고려하는 방법으로, 문법상 맞지 않는 문장이 높은 점수를 얻을 가능성이 있다.

$$P(s) = \prod_{i=1}^n P(w_i)$$

- **Bi-gram**

$$P(s) = \prod_{i=1}^n P(w_i | w_{i-1})$$

- **Tri-gram**

$$P(s) = \prod_{i=1}^n P(w_i | w_{i-2} w_{i-1})$$

다음과 같은 예시를 고려해 보자. $\#$ 은 ‘space’ 를 뜻한다. 총 10,000 단어가 포함된 corpus 를 이용한 statistical language training 결과의 일부이다.

w_i	$P(w_i)$	w_{i-1}	$w_{i-1}w_i$	$P(w_i w_{i-1})$
I (10)	0.001	# (1000) that(10)	# I (7) that I (3)	0.007 0.3
talk (8)	0.0008	I (10) we (5)	I talk (2) we talk (1)	0.2 0.2
talks (7)	0.0007	he (9) she (4)	he talks (3) she talks (1)	0.333 0.25

이를테면 ‘I’ 의 경우 총 10회가 나왔으므로 $P(I) = 0.001$ 이다. ‘#’ 는 corpus 에서 총 1000 회 등장했으며 ‘# I’ 는 corpus 에서 총 7 회 등장했다. 따라서 $P(I|\#) = 0.007$ 이다. Uni-gram 을 사용할 경우 다음과 같다 :

$$P(I \text{ talk}) = P(I)P(\text{talk}) = 0.001 \cdot 0.0008$$

$$P(I \text{ talks}) = P(I)P(\text{talks}) = 0.001 \cdot 0.0007$$

즉, ‘I talks’ 라는 문법적으로 맞지 않는 문장에 대한 확률도 존재한다. 반면, Bi-gram 을 사용하면 다음과 같다 :

$$P(I \text{ talk}) = P(I|\#)P(\text{talk}|I) = 0.007 \cdot 0.2$$

$$P(I \text{ talks}) = P(I|\#)P(\text{talks}|I) = 0.007 \cdot 0.0$$

이 경우 문법적으로 맞지 않는 문장에 대해 어느 정도 걸러줄 것을 기대할 수 있다. 하지만 역시 복잡한 문장에 대해서는 그 기능을 다 하지 못할 것으로 보인다.

여기서 n 을 늘린다면 더 좋은 확률 결과값들을 얻을 것 이나, 이렇게 되면 vocabulary space 의 크기를 $|V|$ 라고 했을 때 $|V|^n$ 개의 가능한 조합이 있으므로 가능한 모든 조합에 대한 확률을 만드려면 data set이 매우 커질 것이다.

Brown Corpus 에서는 “He was happy” 가 6번 나오고 “She was joyful” 이 0번 나온다. 따라서 이 Brown Corpus dataset을 기준으로 하면 $P(\text{She was joyful})=0$ 인데, 사실 두 문장은 (물론 사람이 생각하기에) 매우 유사하다. 때문에 한번도 안나오는 조합에 대한 확률을 0이 아니도록 하기 위해 **smoothing** 이라는 것을 사용하기도 하는데, 좋은 성능을 주기에는 부족하다.

$$\text{add-one smoothing : } P(s) = \frac{c(s) + 1}{N + |V|}$$

이 때 $c(s)$ 는 s 에 대한 counting 이다.

8.2 Word2vec

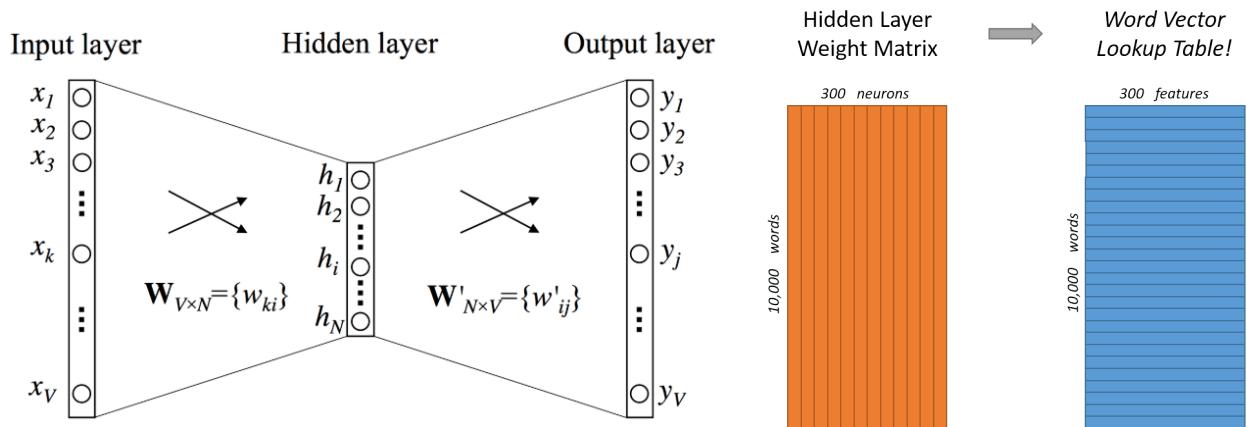
Word2vec 은 Google의 Mikolov 가 만든 word embedding software package 이다. 이 machine 은 2개의 linear FC (단, softmax 사용함) 로 이루어져 있으며 언어적 문맥을 이루는 단어들을 벡터로 바꿔준다. input은 아주 긴 text가 들어가며 output으로는 word vector 가 출력되는데, word vector space 차원은 대략 100에서 1000 사이이다. word vector space 차원을 늘리면 word embedding 의 품질이 좋아지지만, 특정 이상이 되면 잘 오르지 않는다. 문맥상으로 비슷한 단어들은 word space 에서 가까운 위치에 자리하게 된다. 우리는 다음의 구조와 방법들을 다룬다.

- word2vec의 Network models : CBOW, Skip-gram

- word2vec 을 Optimization 하는 방법이다. softmax를 그냥 사용하면 계산량이 너무 많기 때문에 이를 바꾸는 것 이다 : Hierarchical Softmax (복잡하다), Negative Sampling (더 효율적)
- Preprocessing pipelines : Dynamic Context Windows, Subsampling, Deleting Rare Words

8.2.1 Word2vec Models

Simple CBOW (simple continuous bag of words) 은 bi-gram model 이다. 즉, $P(v_{\text{output}}|v_{\text{input}})$ 을 학습시킨다. CBOW, Skip-gram 에도 비슷한 계산을 사용하기 때문에 이 구조를 먼저 알아보자. 두 개의 FC (linear, linear+softmax) 를 사용한 모습이다. 우리의 machine 은 ‘i’ 를 넣으면 ‘am’ 이 나오기를 바란다.



그 과정은 다음과 같다.

- one-hot encoded input \mathbf{x}_k 가 들어간다. (즉 $x_k = 1$ 이고 나머지는 0) V 는 vocabulary space dimension 이다.

$$\mathbf{x}_k = [x_1, x_2, \dots, x_V] \quad \text{only } x_k = 1$$

- hidden layer 를 계산한다.

$$\mathbf{h} = \mathbf{x}_k W = W_{(k,\cdot)} \quad \text{word vector for } v_k$$

or,

$$h_i = \sum_{q=1}^V x_q w_{qi} = w_{ki}$$

만약 machine 이 잘 학습되었다면, input 의 큰 dimension이 hidden layer 에서 줄어들고 적절한 word vector 가 만들어졌다는 뜻 이다. 즉, 우측 그림과 같이 위의 식에 의해 hidden layer weight matrix 의 각 row 는 각 word 에 해당하는 word vector 가 된다.

- output layer input (**score vector**) \mathbf{u} 를 계산한다.

$$\mathbf{u} = \mathbf{h} W'$$

or,

$$u_j = \mathbf{h} W'_{(\cdot,j)} = W_{(k,\cdot)} W'_{(\cdot,j)} = [WW']_{kj} = \sum_{i=1}^N h_i w'_{ij} = \sum_{i=1}^N w_{ki} w'_{ij}$$

4. Softmax 를 계산한다.

$$P(v_j|v_k) = y_j = \frac{e^{u_j}}{\sum_{j'=1}^V e^{u_{j'}}}$$

최종적으로 얻는 벡터 y 는 역시 one-hot encoding 으로 해석한다. 우리가 원하는 ‘i’ 에 대한 target 은 ‘am’ 이다. 즉, x_I (I (input) 에 해당하는 단어) 가 들어가면 y 에서 O (O (output) 에 해당하는 단어) 의 성분이 가장 커지기를 바란다.

Objective function 은 다음과 같다.

$$\max P(v_{\text{Output}}|v_{\text{Input}})$$

즉, 다음의 식을 최소화 하는 방향으로 gradient descent 방법을 적용한다.

$$-P(v_O|v_I) = -y_{j^*}$$

따라서 loss function 은 다음과 같다.

$$E = -\log y_{j^*} = -\log \left[\frac{e^{u_{j^*}}}{\sum_{j'=1}^V e^{u_{j'}}} \right] = -u_{j^*} + \log \sum_{j'=1}^V e^{u'_{j'}}$$

weight W' 에 대한 gradient 를 계산하면 다음과 같다. 이 때 $t_{j^*} = 1$ 이고 나머지 $t_j = 0$ 이다.

$$\begin{aligned} \frac{\partial E}{\partial u_j} &= y_j - t_j \equiv e_j = \text{prediction error} \\ \frac{\partial E}{\partial w'_{ij}} &= \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial w'_{ij}} = e_j h_i \\ \Rightarrow w'_{ij}^{\text{new}} &= w'_{ij}^{\text{old}} - \eta \frac{\partial E}{\partial w'_{ij}} = w'_{ij}^{\text{old}} - \eta e_j h_i \end{aligned}$$

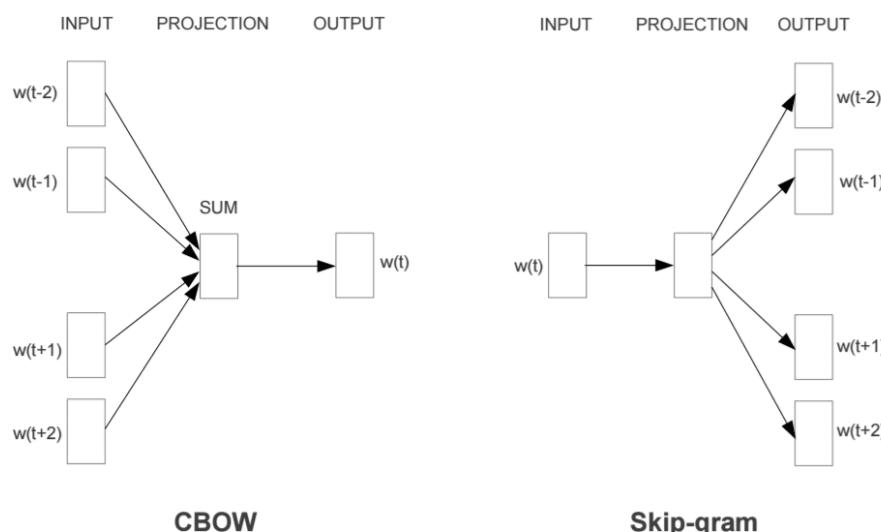
weight W 에 대한 gradient 를 계산하면 다음과 같다.

$$\begin{aligned} \frac{\partial E}{\partial h_i} &= \sum_{j=1}^V \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial h_i} = \sum_{j=1}^V e_j w'_{ij} \\ \frac{\partial E}{\partial w_{qi}} &= \frac{\partial E}{\partial h_i} \frac{\partial h_i}{\partial w_{qi}} = x_q \sum_{j=1}^V e_j w'_{ij} \\ \Rightarrow w_{qi}^{\text{new}} &= w_{qi}^{\text{old}} - \eta \frac{\partial E}{\partial w_{qi}} = w_{qi}^{\text{old}} - \eta x_q \sum_{j=1}^V e_j w'_{ij} \end{aligned}$$

이처럼 간단한 모델의 경우 softmax 를 사용하면 계산도 간단하고 최종결과를 확률로서 다룰 수 있기 때문에 꽤나 편하다. 하지만 여전히 FC를 사용하기 때문에 parameter 개수가 문제가 된다. 매우 큰 vocabulary space를 사용하면서도 학습이 잘 진행되게 하기 위해 Word2vec은 2가지 architecture 가 있다.

1. CBOW (continuous bag-of-words) 는 window를 통해 주변의 문맥적 단어를 보면서 현재 단어를 예측한다. 즉, 이전에는 바로 전 단어를 통해 다음 단어를 예측했다면, 이 방법은 단어의 앞뒤를 보면서 현재 단어를 예측한다. ‘5번째 앞’, ‘4번째 앞’, ..., ‘5번째 뒤’에 대해 독립적으로 수행하여 평균을 내기 때문에 문맥적 단어의 순서는 영향을 주지 않는다.
2. Skip-gram 은 현재 단어를 통해 주변의 문맥적 단어를 예측한다. 즉, 현재의 단어로 ‘5번째 앞’, ‘4번째 앞’, ..., ‘5번째 뒤’에 대해 독립적으로 예측한다. 현재의 단어에서 먼 것 보다 더 가까운 것에 더 많은 weight를 부여한다.

이들 모두 사실은 ‘word vector’를 생성하는 ‘input \rightarrow hidden weights W ’를 찾는 것인데, 학습하는 방식이 다른 것이다. CBOW는 빠르다는 장점이 있고, Skip-gram은 느리지만 성능이 좋다.



CBOW 다음과 같은 계산 과정을 거친다.

1. one-hot encoding 된 size C input context $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_C\}$ 를 생성한다. (context window size C)
2. Weight W 를 통해 embedded word vector $\{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_C\}$ 를 생성한다.

$$\mathbf{h}_i = \mathbf{x}_i W$$

3. embedded word vector 의 평균을 취한다.

$$\mathbf{h} = \frac{\mathbf{h}_1 + \mathbf{h}_2 + \dots + \mathbf{h}_C}{C} = \frac{1}{C} \sum_{c=1}^C \mathbf{x}_c W$$

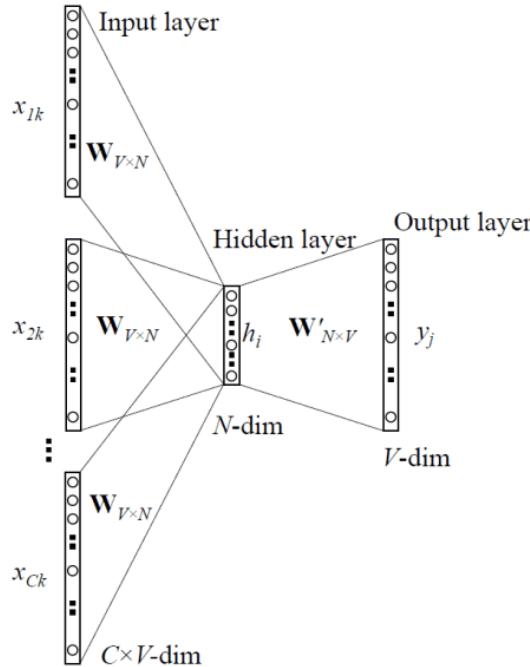
다른 방법으로, input context 를 평균내서 input으로 집어넣을 수도 있다.

4. score vector $\mathbf{u} = \mathbf{h}W'$ 를 계산한다.
5. score 를 probability $\mathbf{y} = \text{softmax}(\mathbf{u})$ 로 바꾼다.

6. $y = t$ 가 되기를 바란다. 이 때 t 는 actual word 의 one-hot encoding 이다. 따라서 loss 를 최소화시킨다.

$$E = -\log P(v_O | v_{I_1}, v_{I_2}, \dots, v_{I_C})$$

하나의 word 에 대한 computation 량은 $C \times N + N \times V$ 이다. C, N 은 작은 숫자인 반면 V 는 매우 큰 숫자임에 주목하자.

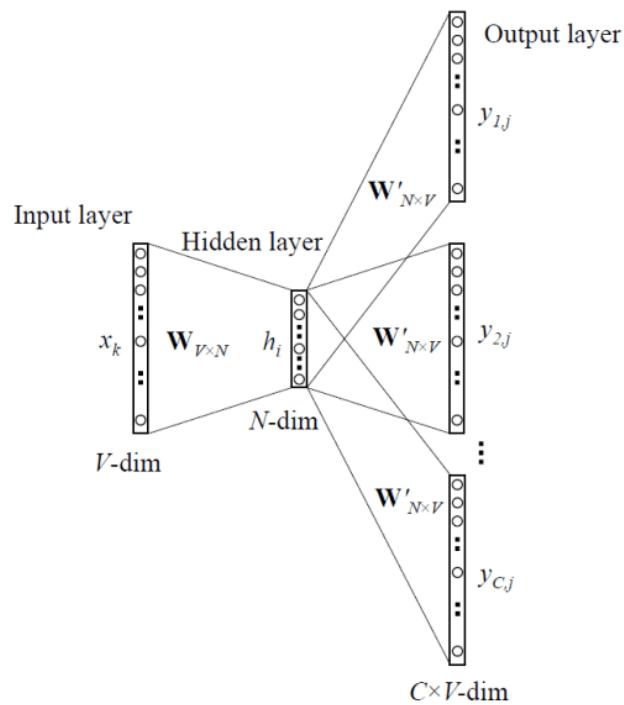


Skip-gram 다음과 같은 계산 과정을 거친다.

1. one-hot encoding 된 \mathbf{x} 를 생성한다.
2. Weight W 을 통해 embedded word vector $\mathbf{h} = \mathbf{x}W$ 를 계산한다.
3. score vector $\mathbf{u} = \mathbf{h}W'$ 를 계산한다. 이 때, 서로 다른 output layer 들은 같은 weight 를 공유한다.
4. score 를 probability $\mathbf{y} = \text{softmax}(\mathbf{u})$ 로 바꾼다.
5. $\mathbf{y} = \mathbf{t}_C$ 가 되기를 바란다. 최소화시키는 loss 는 각각의 target에 대해 적용되어야 한다.

$$\begin{aligned} E &= -\log P(v_{O_1}, v_{O_2}, \dots, v_{O_C} | v_I) \\ &= -\log \prod_{c=1}^C \frac{e^{u_{j_c^*}}}{\sum_{j'=1}^V e^{u_{j'}}} \\ &= -\sum_{c=1}^C u_{j_c^*} + C \log \left[\sum_{j'=1}^V e^{u_{j'}} \right] \end{aligned}$$

하나의 word 에 대한 computation 량은 $C \times N + C \times N \times V$ 이다.



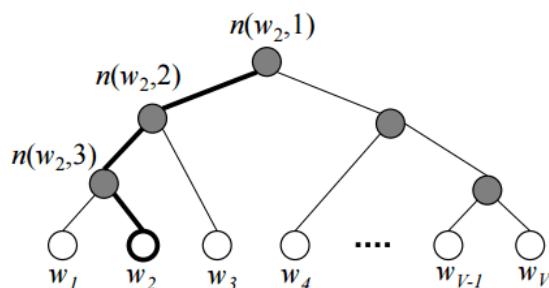
8.2.2 Word2vec Optimization

Hierarchical Softmax 위의 모델들을 살펴보면, W' 을 계산하는 과정에서 연산량이 엄청나게 늘어났다. 따라서 우리는 W' 을 제거하되, 마지막 결과가 확률로 나오게 변형한다. Hierarchical Softmax는 embedded word vector \mathbf{h} 에서 시작해 vocabulary space V 에 있는 단어들을 모두 표현하기 위해 (output y 를 출력하기 위해) binary tree (word2vec 의 경우 이를 **Binary Huffman Tree** 라 한다.)를 사용한다.

총 단계는 $\log V$ 에 비례하는데, 만약 tree 의 모든 단계가 binary 라면 $V = V^{\log_2 2} = 2^{\log_2 V}$ 이기 때문에 모든 단어를 표현할 수 있다. inner unit \mathbf{n} 의 개수 (\mathbf{n} 의 개수)는 $V - 1$ 개가 있다. 가장 위 ($\mathbf{n}(w, 1)$, root) 부터 시작해 각 단어 w_i (leaf) 로 가는 path (길이 $L(w_i)$) 가 유일하게 존재하며 이들은 w_i 가 output word w_O 가 될 확률을 계산하는 데에 사용된다.

$$P(w_i = w_O) = \sigma(\pm \mathbf{h} \cdot \mathbf{v}'_{n(w_i, 1)}) \cdot \sigma(\pm \mathbf{h} \cdot \mathbf{v}'_{n(w_i, 2)}) \cdots = \prod_{j=1}^{L(w_i)-1} \sigma(\pm \mathbf{h} \cdot \mathbf{v}'_{n(w_i, j)})$$

이 때 σ 는 sigmoid, 부호 + 는 왼쪽으로 갈 때이며 -는 오른쪽으로 갈 때를 의미한다.



위의 그림 예시의 경우 $L(w_2) = 4$ ($\approx \log V$) 인데, 이는 w_2 까지의 거리이다. $n(w_2, j)$ 는 w_2 까지 가는 경로에서 j 번째 unit 이다. $\mathbf{v}'_{n(w_i,j)}$ 는 각각의 inner unit $n(w_i, j)$ 에 해당하는 **output vector**이며 dimension N 이다. 이 벡터를 학습할 것인데, W' 와 그 역할이 비슷하다.

$$u_j = \mathbf{h} \cdot \mathbf{v}'_{n(w_i,j)}$$

$$\left(\Rightarrow P(w_i = w_O) = \prod_{j=1}^{L(w_i)-1} \sigma(\pm u_j) \right)$$

따라서 N 차원의 \mathbf{h} 를 $\log V$ 차원으로 사상한다. 이전에는 여기에 softmax 를 적용했는데, 이제 다음과 같이 계산한다.

$$P(w_2 = w_O) = \sigma(u_1)\sigma(u_2)\sigma(-u_3)$$

$$\sum_{i=1}^V P(w_i = w_O) = 1$$

즉, w_2 까지 가는 과정에서 u_1, u_2, u_3 를 거친다. 또한 모든 단어 w_i 로 가는 확률의 합은 1이 되어야 한다. ($\because \sigma(x) + \sigma(-x) = 1$: 그래서 sigmoid 를 사용했다.) 또한 softmax를 사용하는 기존 방법의 경우 하나의 y_j 값 만을 알고 싶다고 해도 모든 \mathbf{y} 의 모든 성분에 대한 지수 합이 필요했기에 계산량이 많았지만, 이 방법의 경우는 원하는 값에 해당하는 경로에서만 계산을 하면 된다.

Hierarchical Softmax 의 **Loss function** 은 다음과 같다. 여기서부터는 weight 와의 문자 중복을 피해 word vector 를 v 로 표기한다.

$$E = -\log P(v = v_O | v_1) = -\sum_{j=1}^{L(v)-1} \log \sigma(\pm u_j)$$

이전의 식과 같은 형태이지만 여기서는 확률의 계산이 달라진다. inner unit 각각의 output vector 에 대한 gradient 계산은 다음과 같다. ($\sigma'(x) = \sigma(x)(1 - \sigma(x))$ 를 기억하자.)

$$\begin{aligned} \frac{\partial E}{\partial u_j} &= -\frac{1}{\sigma(\pm u_j)} \cdot [\pm \sigma(\pm u_j)(1 - \sigma(\pm u_j))] \\ &= \mp(1 - \sigma(\pm u_j)) \\ &= \begin{cases} \sigma(u_j) - 1 & \text{if } + \\ \sigma(u_j) & \text{if } - \end{cases} \\ &= \sigma(u_j) - t_j \quad \text{where } t_j = 1 \text{ for } + \text{ and } 0 \text{ for } - \\ \frac{\partial E}{\partial \mathbf{v}'_{n(w_i,j)}} &= \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial \mathbf{v}'_{n(w_i,j)}} \\ &= (\sigma(u_j) - t_j) \mathbf{h} \end{aligned}$$

따라서 다음과 같다.

$$\begin{aligned} \mathbf{v}'_{n(w_i,j)}^{(\text{new})} &= \mathbf{v}'_{n(w_i,j)}^{(\text{old})} - \eta \frac{\partial E}{\partial \mathbf{v}'_{n(w_i,j)}} \\ &= \mathbf{v}'_{n(w_i,j)}^{(\text{old})} - \eta(\sigma(u_j) - t_j) \mathbf{h} \quad \text{where } j = 1, 2, \dots, L(v) - 1 \end{aligned}$$

또한 simple CBOW 에서 계산한 것을 참조하면 weight W 에 대한 gradient update 식은 다음과 같다.

$$\begin{aligned}\mathbf{w}_{(k,\cdot)}^{(\text{new})} &= \mathbf{w}_{(k,\cdot)}^{(\text{old})} - \eta \frac{\partial E}{\partial \mathbf{w}_{(k,\cdot)}} \\ &= \mathbf{w}_{(k,\cdot)}^{(\text{old})} - \eta \sum_{j=1}^{L(v)-1} \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{w}_{(k,\cdot)}} \\ &= \mathbf{w}_{(k,\cdot)}^{(\text{old})} - \eta x_k \sum_{j=1}^{L(v)-1} (\sigma(u_j) - t_j) \mathbf{v}'_{n(w_i,j)}\end{aligned}$$

parameter update에 사용되는 연산량은 $V \times N + N \times (V - 1)$ 이며, 각 단어에 사용되는 연산량은 $N \times \log V$ 이다. 즉, 연산량이 매우 줄어든다. 일반 Softmax와 Hierarchical Softmax 식을 비교하면 다음과 같다.

$$\begin{aligned}\text{Softmax} : P(v_j|v_I) &= y_j = \frac{e^{u_j}}{\sum_{j'=1}^V e^{u_{j'}}} \quad \text{where } u_{j'} = \sum_{i=1}^N h_i W'_{ij'} \\ \text{HS} : P(v = v_O|v_I) &= \prod_{j=1}^{L(v)-1} \sigma(\pm u_j) \quad \text{where } u_j = \mathbf{h} \cdot \mathbf{v}'_{n(v_i,j)}\end{aligned}$$

각 inner unit 의 역할은 tree 안에서 경로가 왼쪽으로 갈지 오른쪽으로 갈지 예측하는 것 이다. $\sigma(u_j)$ 는 이것의 예측값이다. 이 방법은 CBOW 와 Skip-gram 모두에 적용될 수 있는데, Skip-gram의 경우에는 한번에 하나의 단어에만 적용하는 구조로 만들면 된다.

Negative Sampling 이전의 방법은 iteration 마다 update 해줘야 하는 output vector 너무 많았다. 때문에 negative sampling 에서는 이들을 sampling 한다. Sample 은 output word v_O 와 noise distribution $P_n(v)$ 에 의해 랜덤으로 뽑힌 K 개의 negative sample $V_{\text{neg}} = \{v_1, v_2, \dots, v_K\}$ 로 이루어진다. 다음과 같이 corpus 를 만드는 과정을 고려해 보자. word space 의 크기는 10,000 이라 하자.

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. →	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. →	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. →	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. →	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

이를테면 (fox, quick) 을 training 하는 경우, “fox” 가 들어가면 “quick” 을 출력하게 하는 것은 Skip-gram 이고 “quick” 이 들어가 “fox” 가 나오도록 하는 것은 CBOW 이다. 그런데 이 때 각 단어의 one-hot vector 는 그 크기가 10,000 으로 매우 크다. Skip-gram 을 기준으로 보자면 10,000 차원 벡터에서 1개만 1로 만들고 나머지는 0으로 만들기 위해 수많은 weight 가 필요했다. Negative Sampling 은 0에 해당하는 단어를 적은 수로 (25개 정도) Sampling 하여 그것들에 해당하는 weight 만 업데이트 한다.

결과적으로 우리는 한 번의 iteration 에 대해, sample word $\{v_1, v_2, \dots, v_K, v_O\}$ 에 해당하는 $K + 1$ 차원 벡터 성분들에 output 1 혹은 0 을 출력하는 network 를 만든다. input word v_I 와 context v_C 로 된 input (v_I, v_C) 가 들어갔을 때, $P(D = 1|v_I, v_C)$ 는 input (v_I, v_C) 가 corpus data 에서 나왔을 확률 (즉, v_C 가 올바른 v_O 일 확률) 이며, $P(D = 0|v_I, v_C)$ 는 그렇지 않았을 확률 (즉, v_C 가 negative sample 에서 나왔을 확률) 이다.

$$P(D = 1|v_I, v_C) = \sigma(u_C) = \frac{1}{1 + e^{-u_C}}$$

이 때 u_C 는 simple CBOW 에서 봤던 것과 같이 다음과 같다.

$$\mathbf{u}_C = \mathbf{h}_C W' = \mathbf{v}_C W W'$$

Negative Sampling 의 **Loss function** 은 다음과 같다.

$$\begin{aligned} E &= -\log \left[P(D = 1|v_I, v_O) \prod_{v_j \in V_{\text{neg}}} P(D = 0|v_I, v_j) \right] \\ &= -\log \sigma(u_O) - \sum_{v_j \in V_{\text{neg}}} \log \sigma(-u_j) \end{aligned}$$

gradient 는 다음과 같다.

$$\begin{aligned} \frac{\partial E}{\partial u_j} &= \begin{cases} -\frac{1}{\sigma(u_O)} \cdot \left[\sigma(u_O)(1 - \sigma(u_O)) \right] \\ \frac{1}{\sigma(-u_j)} \cdot \left[\sigma(-u_j)(1 - \sigma(-u_j)) \right] \end{cases} \\ &= \sigma(u_j) - t_j \quad \text{where } t_O = 1 \text{ and } 0 \text{ for others} \end{aligned}$$

따라서 simple CBOW 에서 prediction error term 에 이 식을 넣어주면 업데이트 식이 나온다. 한 단어에 필요한 연산량은 $N \times (K + 1)$ 인데, 엄청나게 줄어든 것을 볼 수 있다. 이 방법은 CBOW 와 Skip-gram 모두에 적용될 수 있는데, Skip-gram 의 경우에는 한번에 하나의 단어에만 적용된다.

word2vec에 사용되는 **Noise distribution** $P_n(v)$ 는 unigram distribution 에 적당한 power (이를테면 $\frac{3}{4}$) 승 한 뒤 normalize 한 것 이다. 예시로, 100개의 단어 $(90v_1, 9v_2, 1v_3)$ 로 구성된 corpus 가 있다고 하자. unigram distribution은 다음과 같다.

$$P(v_1) = 0.9, P(v_2) = 0.09, P(v_3) = 0.01$$

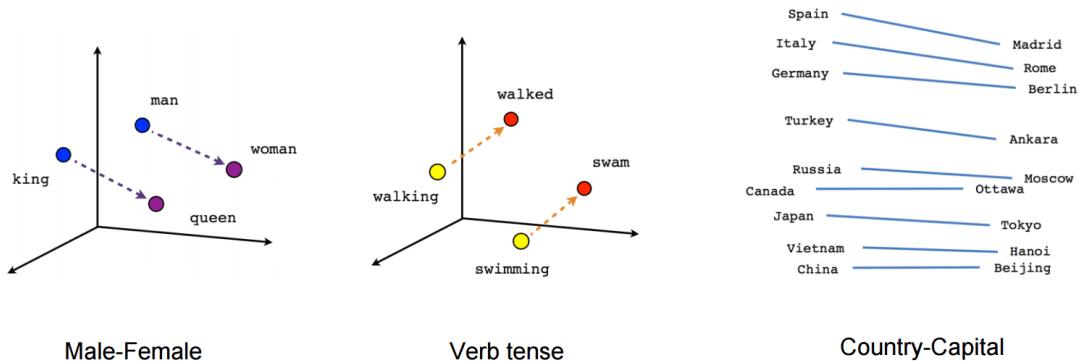
그렇다면 Noise distribution 은 다음과 같다.

$$P(v_1) = 0.825, P(v_2) = 0.146, P(v_3) = 0.029$$

이렇게 하면 잘 나오지 않는 단어에 대한 업데이트도 적절히 확률을 높여 수행할 수 있다. power는 적절히 조절해 원하는 만큼 확률을 조절할 수 있다.

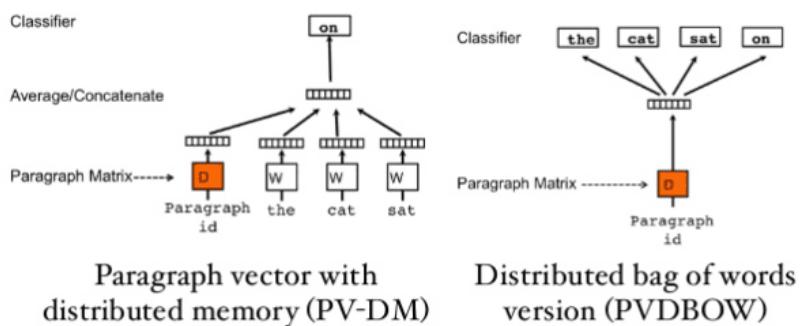
Subsampling 특정 빈도 (threshold t) 이상 등장하는 단어들은 제거하여 학습 속도를 높인다. 자주 등장하는 단어는 정보량을 적게 가지기 때문이다. 이를테면, 이전의 방법들은 ‘car’ 와 ‘drive’ 가 많이 같이 나와 둘의 유사도가 많이 높았지만, ‘car’ 와 ‘the’ 의 유사도가 더 높게 나왔다. 때문에 ‘the’ 와 같이 정보를 가지고 있지 않는 단어는 버릴 것이다. threshold 를 t 라 하면, $P(w) \geq t$ 이면 이를 적절히 줄여준다.

etc 참고로, word2vec에서는 비슷한 단어가 다음과 같은 방식으로 연관된다.



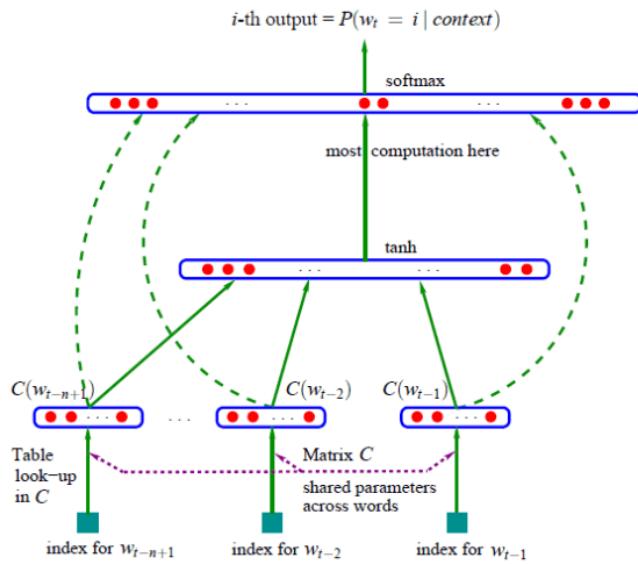
‘king’ 과 ‘queen’ 은 ‘남성’에서 ‘여성’으로 바뀌는 연관성을 가진다. 때문에 잘 학습된 word embedding model 이라면 ‘king’에서 ‘queen’ 으로 가는 만큼 ‘man’에서 움직였을 때 ‘woman’ 이 나올 것이다.

Sentence/Text Representation은 어떻게 할까? 만약 문장을 벡터로 잘 표현할 수 있다면 번역이 너무 간단할 것이다. 특정 의미를 가지는 문장을 말하면 그에 해당하는 벡터가 표현되고 여기서 언어만 고르면 되기 때문이다. 간단한 방법은 word2vec의 결과를 평균내는 것이다. 다른 방법으로는 Paragraph vector (Mikolov 2014) 가 있다. 기본적으로 word2vec의 변형인데, paragraph vector를 추가했다.



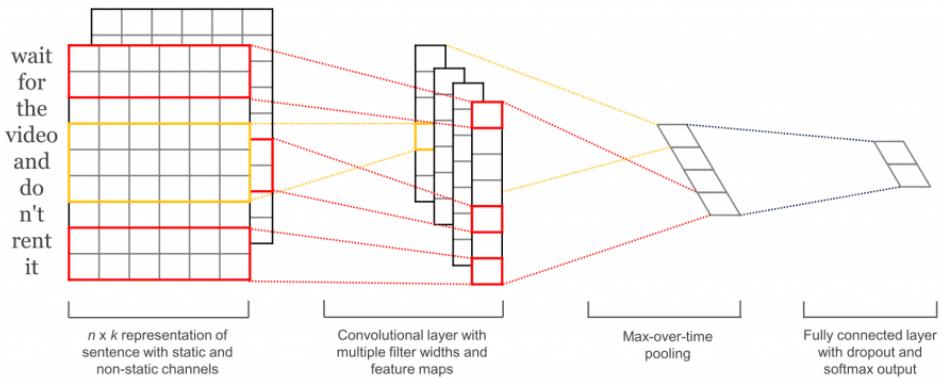
8.3 Seq2Seq

Feed-forward network (Bengio 2001) NLP에서 NN를 사용한 가장 단순한 모델이다. input으로 사용할 문장의 word embedding 을 나열하여 machine에 집어넣는다. 결과는 classification 이 될 수도 있고, 다른 나라의 언어 문장이 될 수도 있다. 성능이 좋지는 못하나, 간혹 사용된다.

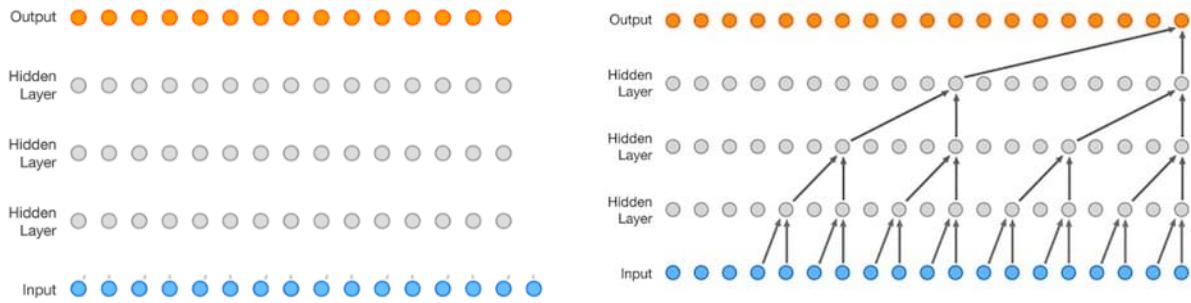


RNN (LSTM, GRU) 은 sequential data를 다루기 때문에 자연어처리에 적합할 것으로 보인다. 하지만 Recurrent model의 경우 word by word 방식으로 연산하기 때문에 병렬처리를 제한한다. (inhibits parallelization) RNN은 이전에도 언급했듯이 연관된 정보 사이의 거리가 멀 경우 과거의 정보를 잊는 특성 탓에 효과적이지 못하다. LSTM은 long range dependency problem을 해결하고자 cell state에 과거를 선택적으로 기억하고 잊는다. 하지만 이 또한 정보를 담은 단어가 너무 과거이면 잊기 때문에 그냥 사용하기에는 적합하지 않다.

CNN word embedding 된 벡터를 그림처럼 쌓아서 $M \times N$ matrix 를 만든다. 이 때 N 은 word embedding 의 차원이며, M 은 문장의 길이이다. 이 matrix의 row 하나는 하나의 단어를 표현하기 때문에 convolution filter 를 작게 잡으면 안되고, $n \times N$ 으로 잡아 하강시키며 convolution 한다. $n = 2$ 이라면 단어를 두개씩 보는 것 이고, $n = 3$ 이라면 단어를 세개씩 보는 것 이다.



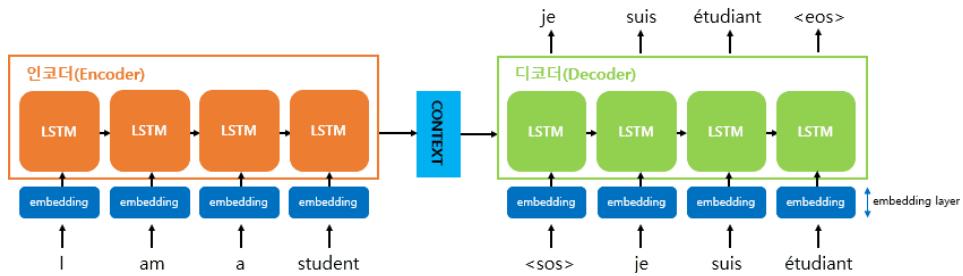
zero-padding 은 없기 때문에 convolution 후에 column 차원 M 이 조금 줄어든 벡터가 나오며, 여기에 activation을 적용한다. 그 뒤 이 벡터에서 max 값만 pooling 한다. (때문에 filter 개수가 매우 많아야 한다.) 이 때문에 문장의 길이 M 에 상관 없이 pooling 결과 벡터의 차원은 일정하다. 최종적으로 FC 를 거친다. 이 기술은 신약개발할 때도 사용하는데, 분자구조를 벡터로 바꿔 효과에 대한 classification 을 하기도 한다.



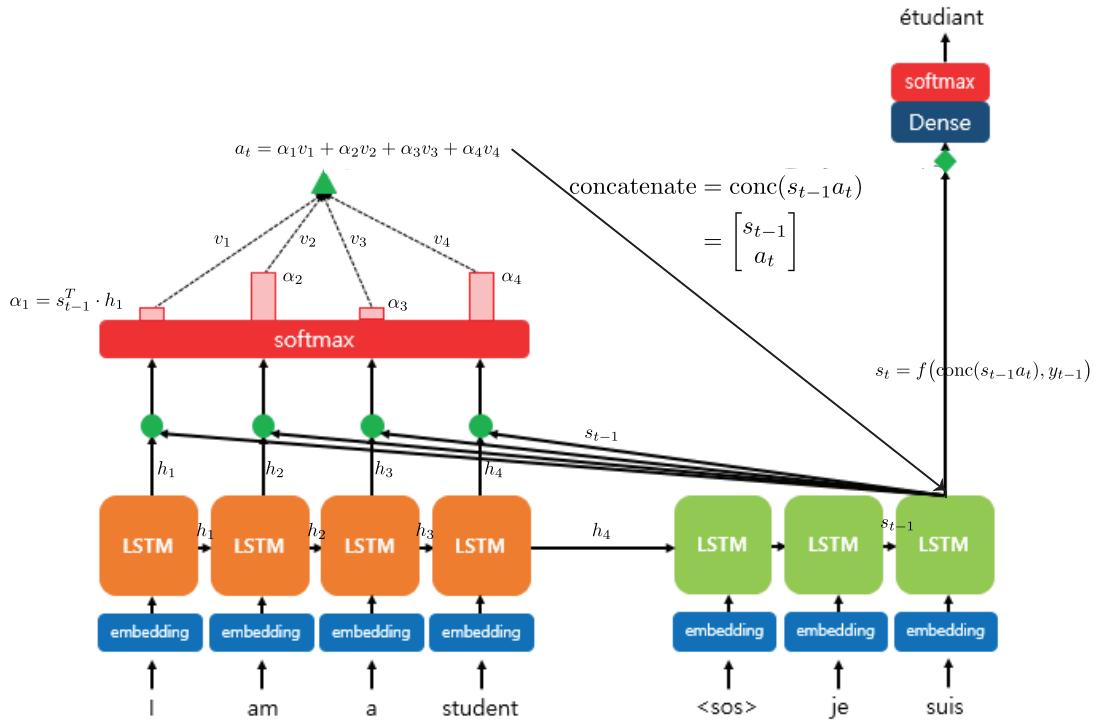
RNN은 시간개념이 있었지만, CNN 은 이와 무관하게 receptive field 에만 의존한다. 또한 CNN 은 parallel 계산이 가능하기 때문에 GPU 를 유용하게 사용할 수 있다.

8.3.1 Seq2Seq and Attention

Sequence-to-Sequence (Seq2Seq) 하나의 sequence 를 다른 sequence 로 보내는 방법이다. Encoder 는 문장을 단어 단위로 받아 하나의 vector 로 바꾼다. Decoder 는 이 vector 를 가지고 다음 단어를 예측한다. **Neural Machine translation (NMT)** 가 대표적인 예시이다. 다만, ‘과연 하나의 벡터가 얼마나 많은 정보를 가지겠느냐’ 하겠는게 문제이다.



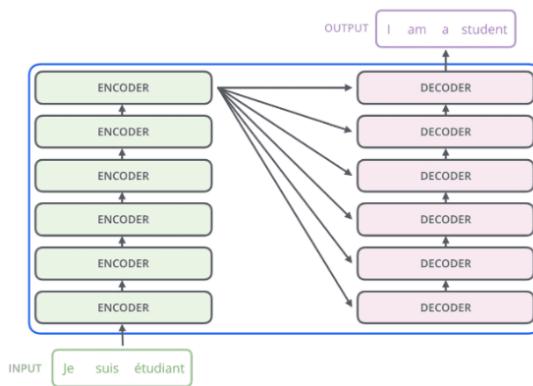
Attention 은 자연어 번역의 중대한 발전을 가져다 주었다. 기존 seq2seq 의 Encoder 는 input 의 모든 정보를 고정된 크기의 벡터로 변환했으며 그 벡터 하나로 문장을 generate 했는데, 이 방식은 문장이 길어질수록 뒤의 단어가 제대로 생성되지 않는 문제가 있었다. Attention 은 decoder가 input sequence를 참조하는 lookback 을 적용한다. 이 때 decoder는 input sequence hidden state 를 가져오는데, weighted average 를 취해 가져온다.



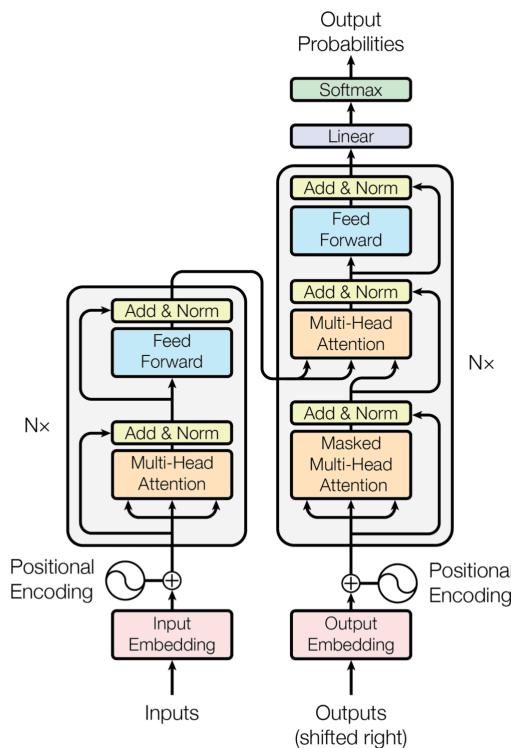
기존 LSTM 구조의 경우 이전 단계에서 받은 hidden state s_{t-1} 을 가지고 $s_t = f(s_{t-1}, y_{t-1})$ 을 생성했다. Attention 을 사용한 경우 s_t 를 만드는 과정이 조금 복잡하다. 우선 input sequence hidden state 를 가져와야 한다. 다만 필요한 과거의 정보는 선택적이다. 때문에 단어를 생성하기 전 **query** (이 경우 s_{t-1}) 를 hidden state 에 보내어 각 hidden state 에 해당하는 **key** (이 경우 h_i) 와 비교해서 어느 정보가 중요한지를 결정하고, **value** (이 경우 v_i) 벡터의 weighted sum 을 계산해 최종 단어를 생성한다.

8.3.2 Transformer

기존의 NMT 가 가지고 있던 global dependency 와 parallel computing 문제를 해결하기 위해 개발되었으며, 기본적으로 encoder-decoder 의 구조를 가지고 있다. input 과 output 의 global dependency 를 계산하기 위해 recurrent 나 convolution 을 사용하는 것이 아니라, **self-attention** 방식을 이용한다. 또한 이 방법은 병렬계산이 가능하게 한다. OpenAI, DeepMind의 AlphaStar 등에 사용되었다. 그 구조는 다음과 같이 단순하다.



Encoder, Decoder 를 자세하게 살펴보면 다음과 같은 구조를 갖고 있다. 그림은 각 layer의 sub-layer 를 나타낸 것이다.



encoder의 경우에는 다음과 같은 2개의 sub-layer 가 있다. Transformer 는 위의 구조를 6개 가지고 있다.

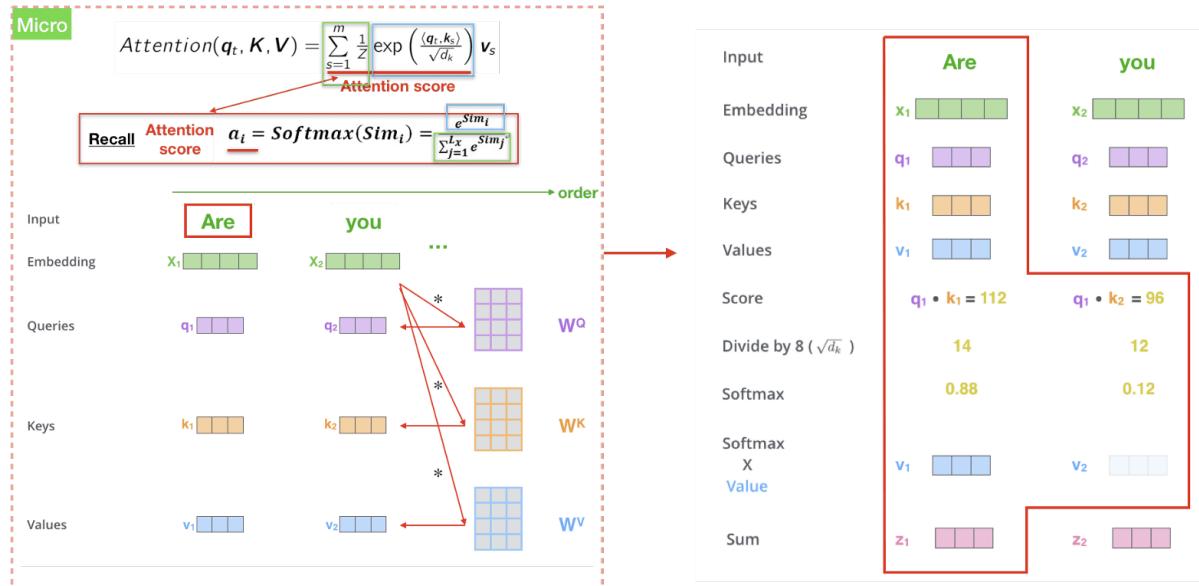
- stacked **self-attention** (Multi-Head Attention)
- **position-wise FFN** (Feed Forward Network)

decoder의 경우에는 다음과 같은 3개의 sub-layer 가 있다. RNN 은 아닌데, Decoder 에 들어가는 'Output Embedding' 은 이전 layer 에서 나온 단어의 Embedding 이다.

- masked multi-head attention
- **encoder-decoder attention** (Encoder로부터 가져오는 attention)
- position-wise FFN

각각의 sub-layer 연결에는 Residual connection (하나 건너뛰어 identity로 가는 것) 이 있으며, 이는 $\text{layerNorm}(x + \text{sublayer}(x))$ 으로 표현된다. 또한, 모든 sub-layer의 input dimension (word embedding dimension) 은 512 이다. input이 들어가 3개로 나뉘는 것은 query, key, value vector 이다.

Self-Attention 특정 단어를 encode 할 때 encoder가 input 문장 안의 다른 단어들을 살펴볼 수 있도록 도와준다. word embedded 벡터가 input 으로 들어가며 이것이 암호화되어 **sum** 으로 다온다. 하나의 Sum 을 계산하는 과정을 살펴보자. (그림에는 Embedding 과 query, key, value 의 차원이 다르게 나와있지만, single self-attention 의 경우 차원이 같다.)



우선 모든 input들에 대해 query, key, value 를 만들어야 한다. 이 때 weight matrix W^Q , W^K , W^V 를 모두가 공통적으로 사용한다.

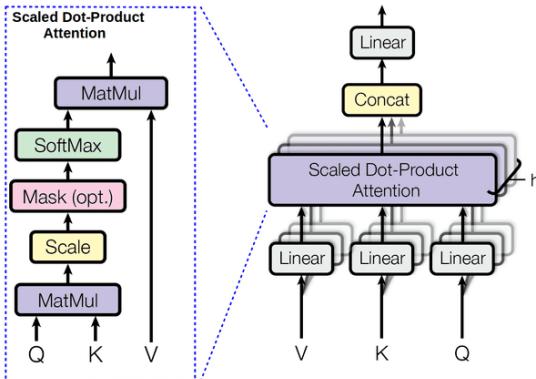
input vector : \mathbf{x}_i	dimension : 64
query vector : $\mathbf{q}_i = \mathbf{x}_i W^Q$	dimension : 64
key vector : $\mathbf{k}_i = \mathbf{x}_i W^K$	dimension : 64
value vector : $\mathbf{v}_i = \mathbf{x}_i W^V$	dimension : 64

모든 input에 대한 query, key, value 를 생성했으면, 본격적으로 하나의 input에 대한 Sum 계산을 시작한다. 우선 우리의 input ‘Are’에서 query 값을 다른 모든 input에 보내 각각의 key 값으로 Score 를 계산한다. 그리고 이 Score 에 대해 key dimension 의 루트값으로 나눠준다. 여기에 softmax 를 적용하고, 여기에 Value 를 곱한 것들을 모두 더해주면 Sum 값이 나온다:

$$\mathbf{Sum}_i = \sum_{j=1}^N \text{Softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{\dim_k}}\right) \mathbf{v}_j$$

이러한 일련의 과정을 수행하는 단위를 **Head** 라고 한다.

Multi-Head Attention Self Attention 에서는 embedded word 와 같은 차원의 query, key, value 를 계산했으나, 이 방법은 query, key, value 의 차원을 줄이고 최종 sum 을 concatenate 하는 방식이다.



input vector : \mathbf{x}_i	dimension : 512
query vector : $\mathbf{q}_i = \mathbf{x}_i W^Q$	dimension : 64
key vector : $\mathbf{k}_i = \mathbf{x}_i W^K$	dimension : 64
value vector : $\mathbf{v}_i = \mathbf{x}_i W^V$	dimension : 64

이 경우 Head 는 8개를 사용했다. 최종적으로 나오는 Sum

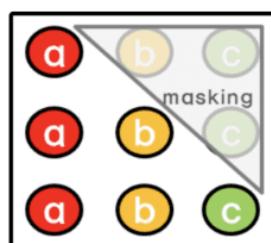
$$\mathbf{Sum}_i^h = \sum_{j=1}^N \text{Softmax} \left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{\dim_k}} \right) \mathbf{v}_j \quad \text{where } h = 1, 2, \dots, 8$$

는 차원이 64로, 이러한 Sum 이 8개 나오게 된다. 결과적으로 output으로 사용하는 것은 이들을 concatenate 하여 사용하게 된다.

$$\mathbf{Sum}_i = \begin{pmatrix} \mathbf{Sum}_i^1 \\ \mathbf{Sum}_i^2 \\ \vdots \\ \mathbf{Sum}_i^8 \end{pmatrix}$$

Masked Multi-head attention 기본적인 self attention 의 경우 예를 들어 ‘I kicked the ball’ 이라는 input 이 들어왔다고 할 때 ‘I’ ‘kicked’ ‘the’ ‘ball’ 각각에 해당하는 Sum 을 구하기 위해 모든 단어에 대한 score 를 계산하는 것을 볼 수 있다.

Masked 의 경우 현재 position 의 뒤에 있는 position 에 대한 attention 은 주지 않는다. 즉, ‘I’ ‘kicked’ 까지 나왔다면 3번째 단어를 예측하는 과정에서 ‘I’ 와 ‘kicked’ 의 query, key, value 만 사용할 수 있는 것 이다. ‘the’ 와 ‘ball’ 은 아직 나오지 않았으므로 attention 을 하지 않는다. 이는 현재 position 뒤쪽의 embedding 을 0 으로 두면 간단하게 처리된다. 미래의 정보를 가린다 하여 ‘Masking’ 이라 한다.



Position-wise FFN 모든 sublayer 에는 두개의 FC 가 linear 와 ReLU 를 사용하여 dimension을 $512 \rightarrow 2048 \rightarrow 512$ 로 변환하며 비선형성을 추가시켜주고 있으며 그 식은 다음과 같다.

$$\text{FFN}(\mathbf{z}) = \max(0, \mathbf{z}W_1 + b_1)W_2 + b_2$$

W_1 와 W_2 는 같은 sublayer 안의 모든 position 에서 같은 값을 사용한다.

Positional Encoding 위에서 언급한 layer들 에서는 문장 안의 단어 순서 정보가 전혀 고려되지 않은 것을 알 수 있다. 때문에 우리는 embedding vector 에 ‘positional encoding’ 이라는 벡터를 더한다. positional encoding 벡터는 sin, cos 함수를 사용한다.

BERT (Bidirectional Encoder Representations from Transformers) 이전에는 특정 machine 을 만들기 위해 해당한 문제에 관련된 데이터만을 가지고 있어야 했다. 모델은 해당 문제 하나에만 적용될 수 있었다. 이를테면 Seq2Seq 를 이용한 번역의 경우 ‘영어 문장’ 과 그것에 대응되는 ‘프랑스어 문장’ 이 있는 dataset 이 필요했다. BERT 는 ‘언어’ 가 가진 개념을 machine 에게 사전학습 시키고, 이를 조금 변형하여 모든 NLP 에 사용할 수 있는 모델이다.

인간에게도 사전학습이 중요하다. 이를테면, 우리는 이제껏 시험과 관련없는 수많은 글들을 읽었고 그로인해 언어의 구조를 이해하고 새로운 글을 읽을 수 있으며 또한 쓸 수도 있다. BERT 는 Transformer 와 유사한 구조를 사용하며, label 이 붙어있지 않은 수많은 언어를 토대로 학습을 진행한다. 학습된 모델은 사용하고자 하는 문제에 맞게 fine tuning 하여 사용한다. **GLUE** (General Language Understanding Evaluation) 과 **SQuAD** (Stanford Question Answering Dataset) 에서 사람보다 좋은 성능을 보여줬다.

9 Lecture 9 : Visual Attention

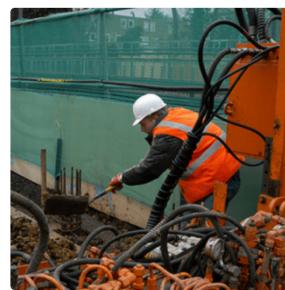
9.1 Image Captioning

Image Captioning 은 image 에 있는 물체와 행동을 토대로 해당 image 에 대한 설명을 text 형식으로 출력하는 과정이다. CNN 과 RNN 에 기저를 두고 있다.

- Encoder : CNN 이 image 의 공간 정보와 특징을 찾아낸다.
- Decoder : RNN 이 image 의 특징을 설명하는 word sequence 를 생성한다.



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."

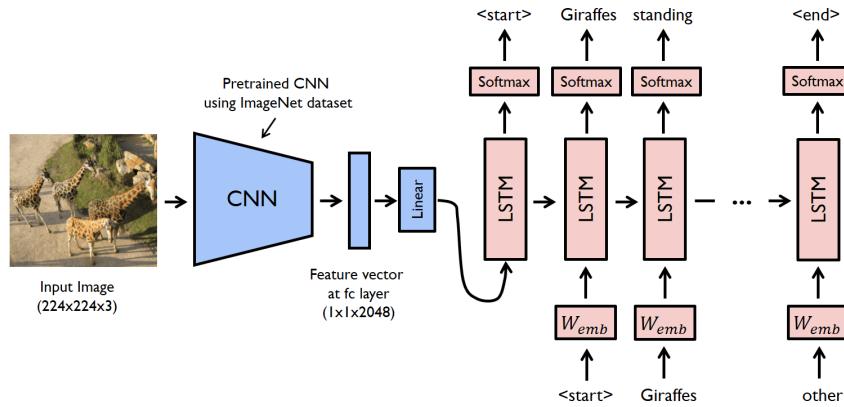


"two young girls are playing with lego toy."

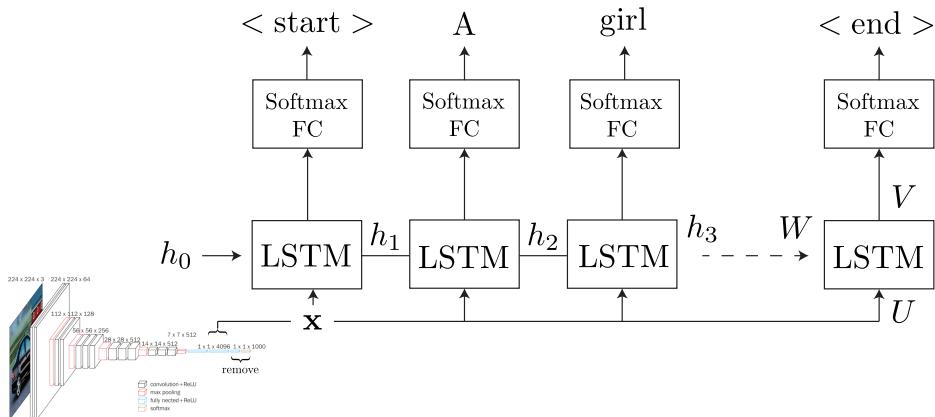
대표적인 dataset **COCO** 는 방대한 량의 image 가 연관된 5개의 caption 과 함께 저장되어있으며, 학습에 매우 유용하게 사용된다. 이런 작업은 어떻게 수행할 수 있을까? 우선 가장 간단한 CNN-LSTM Model 이 있다.

CNN-LSTM Model 우선 이미지를 통해 학습된 CNN (이를테면 VGG-16이나 Resnet)에서 마지막 FC (1000개 classification을 위한 FC)를 제거한다. convolution과 pooling을 통해 생성된 vector는 사진의 feature를 충분히 담고 있을 것이라 생각한다. 즉, CNN은 ‘encoder’로, 사진의 feature extractor로서 작동한다.

feature vector는 LSTM으로 들어간다. LSTM은 ‘decoder’로서, feature vector를 natural language로 변환시키도록 학습된다.



모든 Cell에 x 가 들어가는 것이 위의 그림처럼 y_{t-1} 이 들어가는 것 보다 더 좋다. CNN-LSTM에 사진을 넣으면 우선 CNN은 사진의 정보를 충분히 담고 있는 feature vector x 를 생성하여 모든 LSTM Cell에 보낸다.



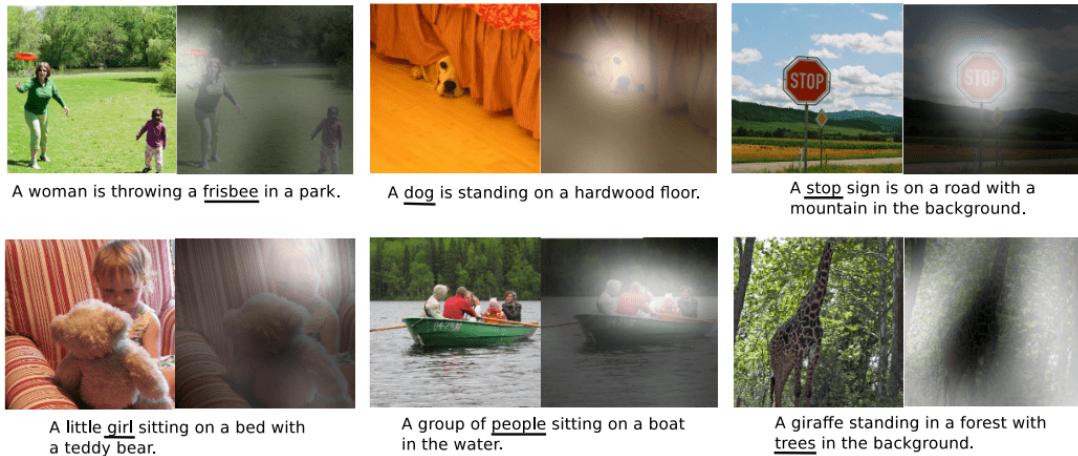
우선 hidden state vector h_0 가 x 와 함께 첫 LSTM Cell에 들어가면 다음 hidden state vector h_1 를 생성하며 $<\text{start}>$ 를 출력할 것이다. 그리고 다음 Cell에는 h_1 와 x 가 함께 들어가고 사진의 caption에 해당하는 첫 단어를 출력하게 된다. 즉, 다음과 같다 :

$$\begin{aligned} \mathbf{h}_t &= f(\mathbf{x}, \mathbf{h}_{t-1}) \\ \mathbf{y}_t &= g(\mathbf{h}_t) \quad \leftarrow \text{prediction} \end{aligned}$$

이 과정은 LSTM이 $<\text{end}>$ 를 출력할 때 까지 계속된다.

9.2 Visual Attention

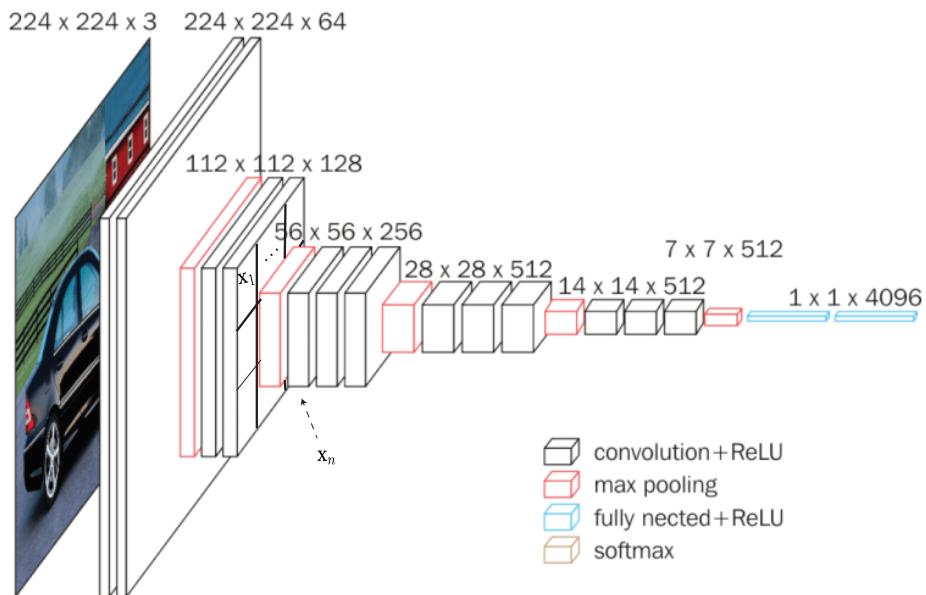
Visual Attention 사람은 image 를 보고 정보를 얻을 때, image 의 모든 영역을 균일하게 살펴보고 판단을 하기보다는 image 의 특정 부분을 집중하여 보고 (pays attention) 판단을 내린다. 아래 그림은 사진과 이 사진에 대한 caption 을 판단할 때 중요하게 보는 핵심 단어의 영역을 표시한 것 이다.



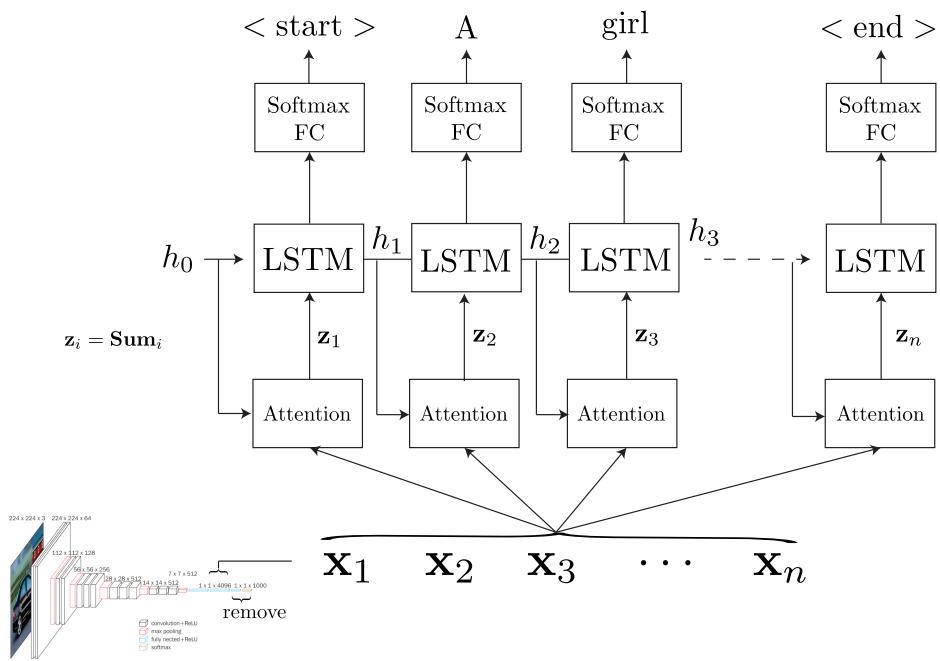
수학적으로, attention module 을 다음과 같이 사용한다 :

$$\mathbf{h}_t = f(\text{attention}(\mathbf{x}, \mathbf{h}_{t-1}), \mathbf{h}_{t-1})$$

이 때 \mathbf{h}_{t-1} 을 context, $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ 을 image feature 로 사용한다. 그 구조를 살펴보자. 우선 CNN 에서 image 를 n 개의 구간으로 분리해 각각에 대한 feature vector $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ 를 얻어낸다.



즉, 이전에는 feature vector x 만 LSTM 의 input 으로 넣었지만, 여기서는 x 를 LSTM 의 hidden state 초기값으로 들어가고, $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ 이 LSTM 에 물려있는 모든 Attention 으로 들어간다. h_{t-1} 는 이전에 어떤 단어가 나왔는지에 대한 query 가 되고 $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ 은 key 이며 동시에 value 가 된다. attention sum 은 다음 단어가 어떻게 나올지 적절한 key 를 골라야 하고 \mathbf{z}_t 가 된다.



attention 과정은 그 종류에 따라 둘로 나뉜다.

- **Soft attention** : h_{t-1} 와 \mathbf{x}_i 를 선형결합 후 activation에 넣어 a_{ti} 를 만든다 :

$$a_{ti} = f_{\text{att}}(\mathbf{x}_i, h_{t-1}) = \tanh(W h_{t-1} + W' \mathbf{x}_i)$$

그리고 여기에 softmax 를 취한다 :

$$a_{ti} = \text{softmax}(a_{t1}, a_{t2}, \dots, a_{tn})$$

마지막으로 value 에 곱해 sum 을 구한다 :

$$\mathbf{z}_t = \sum_{j=1}^n a_{tj} \mathbf{x}_j$$

softmax 를 사용하여 각각의 영향력을 줄였기 때문에 soft attention 이라 한다. 모든 과정이 연속이므로 Backpropagation 으로 학습을 진행한다.

- **Hard attention** : 위에서 마지막 sum 을 구하는 과정만 달라진다. 위에서는 확률로서 a_{tj} 를 직접 value 에 곱했다면, 여기서는 s_{tj} 가 1이 될 확률을 a_{tj} 로 하여 이것을 value 에 곱해준다 :

$$\mathbf{z}_t = \sum_{j=1}^n s_{tj} \mathbf{x}_j$$

이 때 확률은 다음과 같다 :

$$p(s_{tj} = 1 | s_{j < t}, \mathbf{x}) = a_{tj}$$

확률적으로 변수를 정하므로 REINFORCE 의 방법으로 학습을 진행한다.

9.3 Recurrent Attention Model (RAM)

RAM 은 sequential (RNN) decision process (RL) 을 이용한 visual attention 방법이다. Agent 는 image에서 물체의 위치를 찾아가는 action 을 취하게 된다.



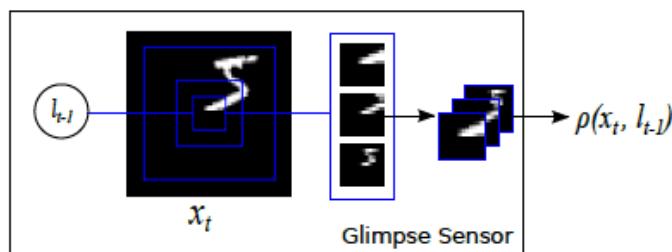
여기에는 다음의 4가지 network 가 있다 :

- **Glimpse Network** : Glimpse Sensor 를 이용해 만들었다.
- **Core Network** : Glimpse Network, LSTM 등을 이용해 만들었다.

LSTM 은 glimpse window 를 input 으로 하며 hidden state 를 근거로 다음 location 을 선택하며 dynamic environment 를 움직이게 한다. 이는 CNN 처럼 모든 영역에 대해 계산하지 않기 때문에 연산이 빠르다.

- **Location Network**
- **Action Network**

Glimpse Sensor 는 주어진 image x_t 의 glimpse location l_{t-1} 에 해당하는 몇 개의 서로 다른 크기의 patch 를 concatenate 한 $\rho(x_t, l_{t-1})$ 를 생성한다.



이 때 각 patch 를 추출하는 과정은 다음과 같다:

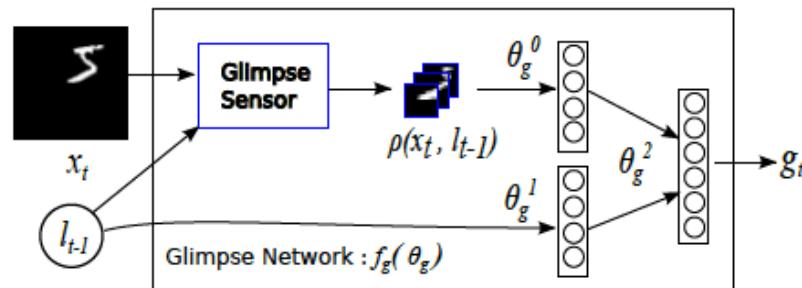
1. image x_t , glimpse location l_{t-1} 정보가 입력되면, patch 의 개수 k 는 고정이다. sensor 는 **bandwidth-limited sensor** ρ 를 가지고 있다.
2. convolution 을 이용하여 l_{t-1} 를 중심으로 하는 $s \times s$, $2s \times 2s$, ..., $ks \times ks$ pixel 크기의 patch 를 얻는다.
3. 얻은 patch 모두를 $s \times s$ pixel 크기로 줄인다. 처음부터 $s \times s$ 로 추출한 patch 는 high resolution 이지만, 다른 것들은 low resolution 을 갖게 된다.
4. 이들 모두를 concatenate 하여 $\rho(x_t, l_{t-1})$ 를 얻는다.



Glimpse Network 는 LSTM 의 input feature 가 되는 **glimpse feature vector** g_t 를 생성한다 :

$$g_t = f_g(x_t, l_{t-1}; \theta_g^0, \theta_g^1, \theta_g^2)$$

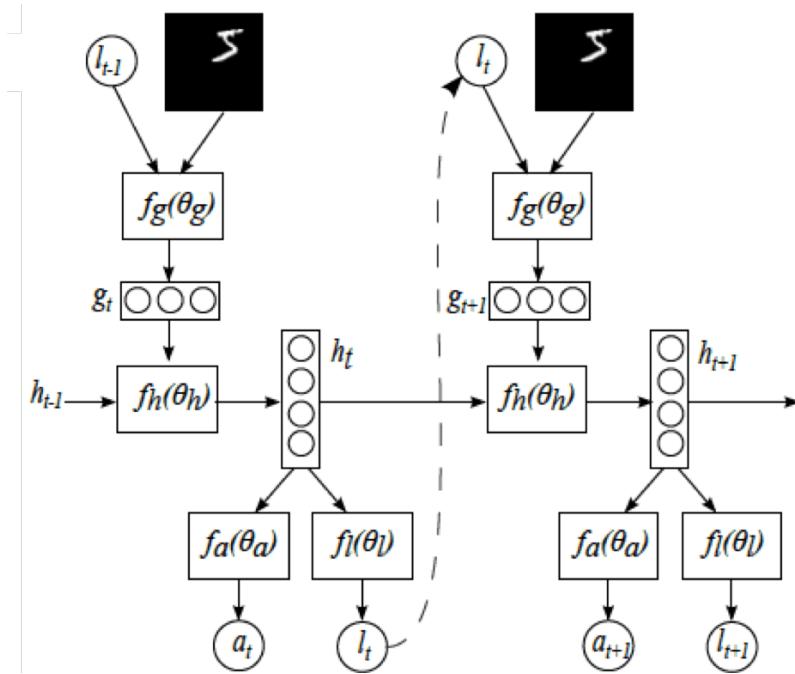
이 벡터는 원래 image x_t 보다 dimension 이 훨씬 작다.



우선 timestep t 에 해당하는 input image x_t 는 glimpse sensor 로 들어가고, 이전 단계에서 온 glimpse location l_{t-1} 은 glimpse sensor 와 FC θ_g^1 로 들어간다. glimpse sensor 에서 나온 $\rho(x_t, l_{t-1})$ 는 FC θ_g^0 으로 들어간다. 이들은 FC θ_g^2 에 들어가 glimpse feature vector g_t 로 출력된다. 이 때 g_t 는 다음과 같다 :

$$g_t = \text{ReLU} \left[\text{FC}_2 \left(\text{ReLU} \left[\text{FC}_0(\rho(x_t, l_{t-1})) \right] \right) + \text{FC}_2 \left(\text{ReLU} \left[\text{FC}_1(l_{t-1}) \right] \right) \right]$$

Core Network 는 LSTM 을 통해 hidden state $h_t = f_h(h_{t-1}, g_t; \theta_h)$ 를 생성한다. LSTM 은 glimpse feature vector g_t 를 input 으로 하며, 매 timestep t 의 hidden state 는 agent 에게 전해진다.



agent 는 hidden state 를 받아 Location Network 와 Action Network 를 작동시킨다:

- Location Network $f_l(h_t; \theta_l)$ 는 다음 timestep에 location sensor에 들어갈 glimpse location 을 확률에 따라 출력한다 :
- $$l_t \sim P(\cdot | f_l(h_t; \theta_l))$$
- Action Network $f_a(h_t; \theta_a)$ 는 environment action a_t 를 결정하는데, 이는 machine 의 목적에 따라 달라지며 environment 에 영향을 줄 수도 있다.

$$a_t \sim P(\cdot | f_a(h_t; \theta_a))$$

classification 의 경우에는 softmax output 을 사용하여 ouput 으로 classification 을 하며, motor control 같은 dynamic environment 의 경우에는 formulation이 복잡해진다.

Action Network 에서 action 을 수행하면, agent 는 다음 timestep에 대한 새로운 image x_{t+1} 와 reward r_{t+1} 을 받는다. 학습 방향은 return $R = \sum_{t=1}^T \gamma^{t-1} r_t$ 를 최대로 만들도록 한다.

Training RL 에서 POMDP (Partially Observable Markov Decision Process) 라고 불리는 방법이다. 이 방법에서는 environment를 부분적으로 관찰하기 때문에, agent 는 모든 trajectory $s_{1:t} = x_1, l_1, a_1, \dots, x_{t-1}, l_{t-1}, a_{t-1}, x_t$ 에 대한 stochastic policy $\pi_\theta((a_t, l_t) | s_{1:t})$ 를 학습하기 위해 모든 시간에 대한 정보를 모아야 한다.

RAM 의 경우는 π_θ 가 LSTM 에 의해 결정되며 trajectory $s_{1:t}$ 는 hidden state h_t 에 압축되어 있다. 따라서 우리는 return 을 최대화하기 위해 agent parameter $\{\theta_g, \theta_h, \theta_a\}$ 만을 학습하면 된다. 강화학습의 목표는 return $R = \sum_{t=1}^T r_t$ ($\gamma = 1$) 을 최대화 하는 것이며, return 은 그 경로 $s_{1:t}$ 에 의해 결정된다. 또한 경로를 결정하는 것은 policy 이므로, objective function 은 다음과 같이 주어진다.

$$J(\theta) = \mathbb{E}_{p(s_{1:T}; \theta)}[R]$$

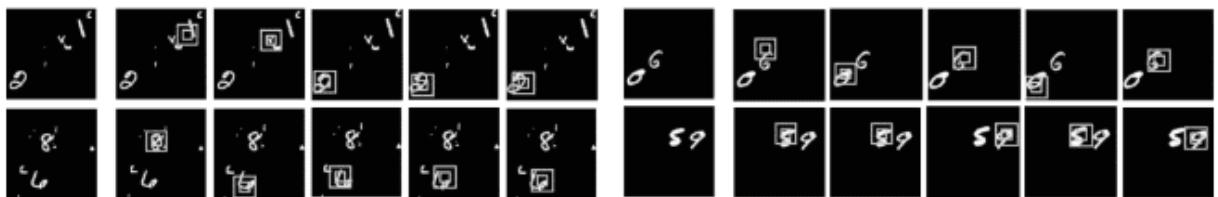
여기서 우리는 REINFORCE with baseline 방법을 사용한다. 단, 직접 모든 경우를 계산할 방법은 없으므로 M 회의 시행을 통해 sample approximation 을 시행한다.

$$\begin{aligned}\nabla_{\theta} J &= \mathbb{E}_{p(s_{1:T}; \theta)} \left[\sum_{t=1}^T (R_t - b_t) \nabla_{\theta} \log \pi_{\theta}(a_t | s_{1:t}) \right] \\ &= \frac{1}{M} \sum_{i=1}^M \sum_{t=1}^T (R_t^i - b_t) \nabla_{\theta} \log \pi_{\theta}(a_t^i | s_{1:t}^i)\end{aligned}$$

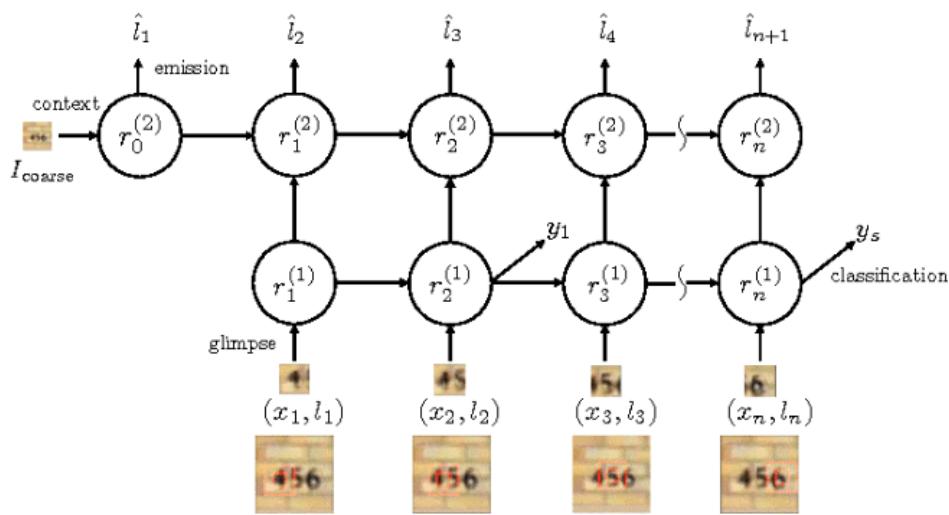
이 때 $R_t = \sum_{k=t}^T r_k$ 이며 i 는 각 시행을 나타낸다. $\nabla_{\theta} \log \pi_{\theta}(a_t^i | s_{1:t}^i)$ 는 LSTM 의 gradient 로서 Backpropagation 을 이용해 구할 수 있다.

RAM 은 전체를 보지 않고 부분만을 보기 때문에 배경과 같은 필요없는 부분들을 무시할 수 있는데, 이는 사람이 사물을 인식하는 과정과 유사하다. 또한 patch 는 정해진 size $s \times s$ 에 따라 만들어지기 때문에 input image 의 크기에 관련이 없다. 즉, 그림이 커져도 연산량이 크게 늘지는 않는다. (물론 너무 커지면 location 찾기가 힘들다)

9.4 Deep Recurrent Visual Attention Model (DRAM)



DRAM 은 RAM 의 발전된 모형으로 하나의 이미지에서 여러가지 물체의 위치를 인식하고 무엇인지 알아낸다. deep RNN (two stacked LSTM) 을, input image 에서 물체가 있는 위치를 찾아내는 RL 방식으로 학습한다. 앞서 CNN 을 사용한 것들 보다 더 적은 연산으로 높은 정확성을 보인다.



DRAM 은 다음과 같은 network 구조를 가진다 :

1. **Recurrent Network** : two stacked LSTM 을 가지며, $r_t^{(1)}$ 은 물체의 classification 을, $r_t^{(2)}$ 는 물체의 location 을 담당한다 :

$$\begin{aligned} r_t^{(1)} &= \text{LSTM}(g_t, r_{t-1}^{(1)}) \\ r_t^{(2)} &= \text{LSTM}(r_t^{(1)}, r_{t-1}^{(2)}) \end{aligned}$$

2. **Glimpse Network** : RAM에서 살펴본 것과 같이, glimpse sensor에서 나온 $\rho(x_t, l_t)$ 와 l_t 가 network를 거쳐 glimpse feature vector g_t 를 생성한다. 다만 기존에는 $\rho(x_t, l_t)$ 를 바로 FC로 보냈다면, 여기서는 $\rho(x_t, l_t)$ 를 Convolution 3번 하고 마지막 FC를 거친다. l_t 는 따로 FC를 거친다. 그리고 이들을 성분별로 곱해 glimpse feature vector g_t 를 구한다.
3. **Emission Network** : RAM에서 location network 와 유사하다. 이 경우는 FC로 되어있으며, hidden state $r_t^{(2)}$ 를 input으로 받아 다음 glimpse location l_{t+1} 을 예측한다. REINFORCE (policy gradient method) 방법을 이용해 학습한다.
4. **Classification Network** : $r_t^{(1)}$ 을 input으로 하고 FC와 softmax를 이용해 class label y_i 를 예측한다. 문장단위의 label이 있을 경우 순서대로 $\{y_1, y_2, \dots, y_s\}$ 의 sequence로 처리한다.
5. **Context Network** : input image에서 down sampling을 통해 resolution을 낮춘 coarse image 가 3개의 convolution을 거쳐 feature vector $r_0^{(2)}$ 를 생성한다. 즉, 2단계 LSTM을 initialize 한다.

training input image I , class label y 를 알고 있다면 supervised classification 학습에 사용되는 cross entropy objective function은 다음과 같다 :

$$\log p(y|I, W) = \log \sum_l p(l|I, W)p(y|l, I, W) \geq \sum_l p(l|I, W) \log p(y|l, I, W) = \mathcal{F}$$

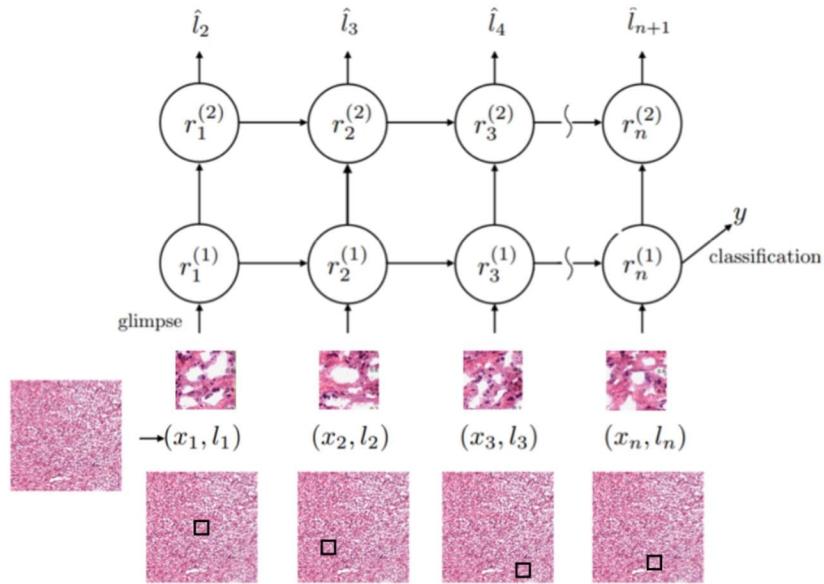
정의대로는 미분이 어렵기 때문에 \mathcal{F} 를 최소화하도록 한다. 역시 모든 경우에 대해 계산할 수는 없기 때문에 M 번의 시행 후 산술평균을 구하며, 이에 따른 gradient는 다음과 같다 :

$$\begin{aligned} \frac{\partial \mathcal{F}}{\partial W} &= \sum_l \left[p(l|I, W) \frac{\partial \log p(y|l, I, W)}{\partial W} + \frac{\partial p(l|I, W)}{\partial W} \log p(y|l, I, W) \right] \\ &= \sum_l p(l|I, W) \left[\frac{\partial \log p(y|l, I, W)}{\partial W} + \frac{\partial \log p(l|I, W)}{\partial W} \log p(y|l, I, W) \right] \\ &\approx \frac{1}{M} \sum_{m=1}^M p(l|I, W) \left[\frac{\partial \log p(y|l^m, I, W)}{\partial W} + \log p(y|l^m, I, W) \frac{\partial \log p(l^m|I, W)}{\partial W} \right] \end{aligned}$$

만약 class label y 가 sequential $\{y_1, y_2, \dots, y_s\}$ 라면 식이 조금 달라진다. 우선 y_e 를 ‘end of sequence’로 정의하고, 이것이 나오면 machine을 멈추는 것으로 한다. 학습된 machine은 적당한 문장을 생성하고 마지막으로 y_e 를 생성해 sequence를 멈춘다. 다음과 같은 objective function을 사용한다 :

$$\log p(y_1, y_2, \dots, y_s | I, W) = \sum_{s=1}^S \log \sum_{l_s} p(l_s | I, W) p(y_s | l_s, I, W)$$

조직 병리학 (histopathological)에 사용되는 DRAM의 예시이다 :

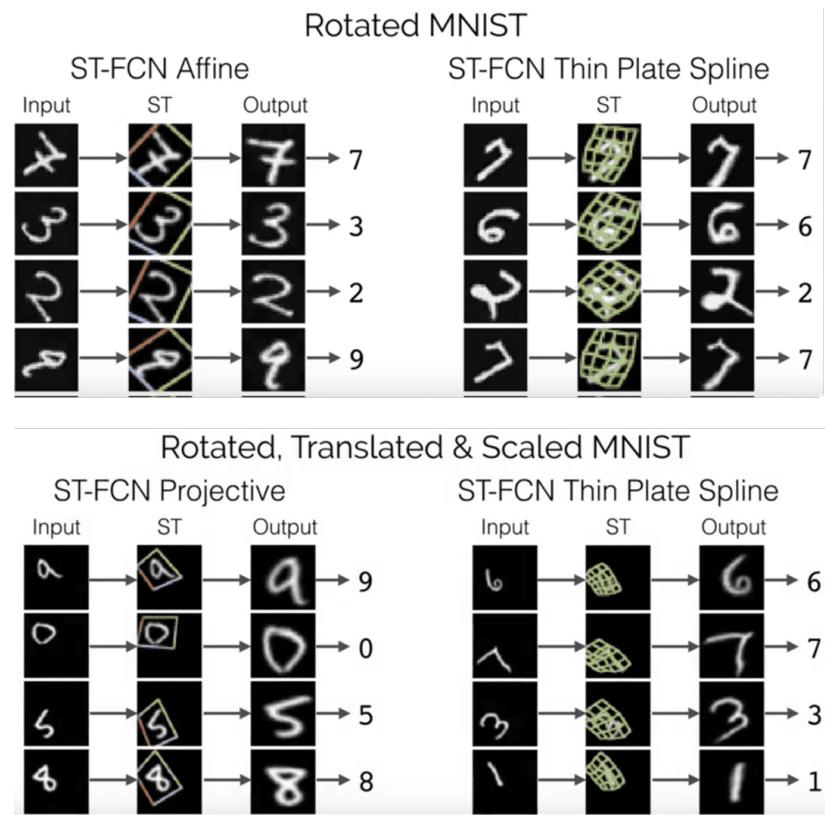


세포 조직과 관련된 이미지가 있을 때 문제가 되는 위치를 찾고 그것이 정말 문제인지 classification 하는 machine 이다. 기존의 CNN-LSTM 모델도 성능이 좋기는 하지만 DRAM 방식은 일부분만 보기 때문에 계산이 빠르고 보다 좋은 성능을 낼 수 있다.

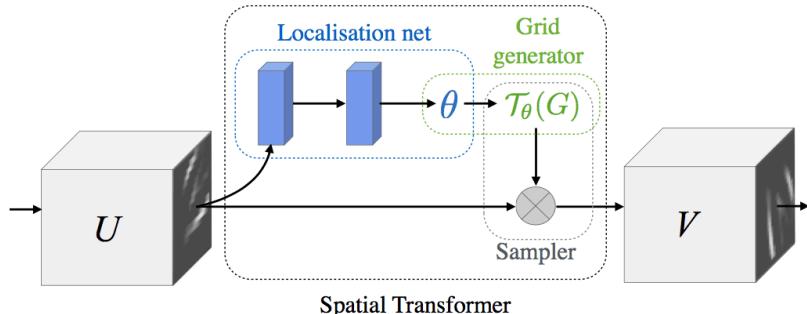
9.5 Enriched Deep Recurrent Visual Attention Model (EDRAM)

9.5.1 Spatial Transformation Network (STN)

이전에 사용된 방법들의 경우 Attention 을 이용해 그림의 어느 부분을 관찰할지 결정하고 주로 CNN 을 이용하여 사진을 관찰했다. 하지만 CNN 은 **spatially invariant** 하기 때문에 (즉, pixel 의 상대적 위치가 고정되어있기 때문에) 변형된 이미지에 대한 인식이 좋지 못했다. STN 은 **affine transformation** 을 사용하여 cropping, translation, rotation, scaling, skewing 된 것도 (미분 가능한 방법으로) 인식할 수 있다.



이전까지 우리는 CNN 을 이용해 사진을 인식했다. Spatial Transformer은 localization network, grid generator, sampler 로 이루어진, CNN 의 역할을 대체하는 network 이며 일종의 module 이다.



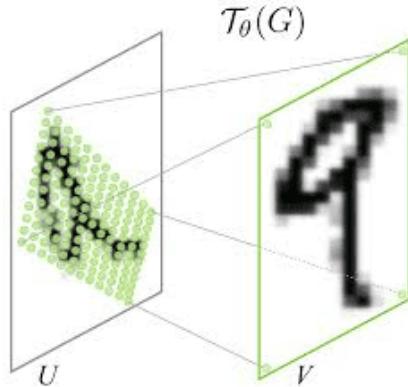
이전에 사용된 CNN 은 spatially invariant 하기 때문에 그 한계가 있었다면, spatial transformer는 feature map 을 spatially transform 하여 translation, scale, rotation, generic warping 에 대해 학습한다. 3개의 network 는 미분 가능하기 때문에 end-to-end 방식으로 각 network 의 parameter 를 backpropagation 하면 된다.

Localization Network 그림의 어느 부분을 잘라낼지 찾는다. input feature map $U_{W \times H}$ 를 받아 transformation parameter θ (혹은, matrix A_θ) 를 grid generator 로 보내준다. FC, convolution 으로 이루어져 있다.

Grid Generator 받은 θ 를 이용해 sampling grid $T_\theta(G) = \{(x_i^s, y_i^s)\} \subset U$ 를 생성한다. 즉, input image 에서 뽑을 부분에 점을 찍어 후에 sampler 가 transformed output feature map $V_{W' \times H'}$ 을

생성하도록 한다. 이 때 V 의 i -th pixel 좌표가 (x_i^t, y_i^t) , 이에 대응되는 U 의 i -th pixel 좌표가 (x_i^s, y_i^s) 이다.

계산은 약간 헷갈린다. target $V_{W' \times H'}$ 가 이미 있고 이에 대응되는 $U_{W \times H}$ 가 있다고 생각하면 다음과 같은 대응이 진행된다. 이는 원하는 image에 해당하는 영역이 원래 image에서 어디에 해당하는 것인지 찾는 것이다.



이를테면, θ 의 크기가 6인 경우의 예시는 다음과 같이 계산된다 :

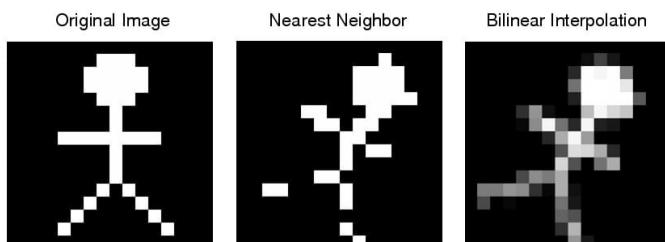
$$\begin{pmatrix} x_i^s \\ y_i^s \\ 1 \end{pmatrix} = T_\theta(G_i) = A_\theta \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix}$$

이 때 (x_i^t, y_i^t) 에 1 이 추가된 것은 bias (상수항) 의 목적이다.

Sampler input feature map U 와 sampling grid $T_\theta(G)$ 를 받아 transformed image V 를 만든다. V 는 input feature map 의 ‘rectified and resized’ version 이다. 그런데, sampling grid 는 실수값을 사용하기 때문에 grid 좌표 (x_i^s, y_i^s) 는 실수이다. 반면 U 의 pixel 들의 좌표는 정수인데, 여기서 문제가 생긴다.

Sampler 의 궁극적인 목표는 V 의 i -th pixel 좌표 (x_i^t, y_i^t) 에 해당하는 pixel value V_i 를 찾는 것이다. (x_i^t, y_i^t) 에 해당하는 grid 는 (x_i^s, y_i^s) 이기 때문에 여기에 해당하는 U 에서의 pixel 값을 가져오면 될 것 같지만 (x_i^s, y_i^s) 가 실수이기 때문에 값을 그대로 가져올 수가 없는 것이다.

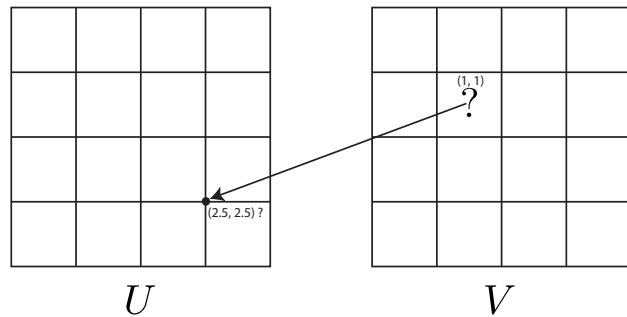
때문에 Sampler 는 nearest integer sampling 이나 bilinear interpolation 방법을 사용한다. 아래는 회전하는 STN machine에서 각각의 방법을 사용한 결과이다.



주로 bilinear interpolation 을 사용하기 때문에, 어떤 방식으로 이루어지는지 알아보자. 우선 $V(1, 1)$ 에 해당하는 U 의 grid 를 Grid Generator에서 다음과 같이 넘겨줬다고 하자.

$$\begin{pmatrix} 2.5 \\ 2.5 \end{pmatrix} = \begin{bmatrix} 1 & 0.5 & 1 \\ 0.5 & 1 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

그런데 $(2.5, 2.5)$ 라는 좌표는 U 에서 존재하지 않는다.



bilinear interpolation에 따른 V_i 는 다음과 같이 표현된다 :

$$V_i = \sum_{n=0}^H \sum_{m=0}^W U_{mn} \text{ReLU}(1 - |x_i^s - m|) \text{ReLU}(1 - |y_i^s - n|)$$

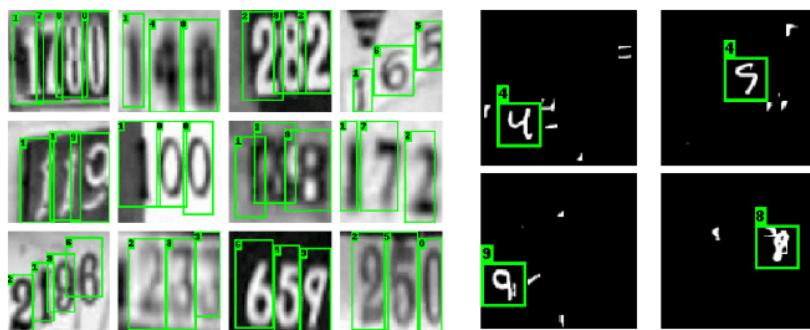
STN에서 input은 두 갈래로 나뉘어 가기 때문에 backpropagation 또한 두 갈래가 있다. V 에서 U 로 바로 가서 전체 machine(즉, STN을 제외한 다른 networks)을 학습하는 길이 있고, V 에서 x_i^s 를 통해 Spatial Transformer를 학습하는 길이 있다. 각각의 gradient 값은 다음과 같다 :

$$\begin{aligned} \frac{\partial V_i}{\partial U_{mn}} &= \sum_{n=0}^H \sum_{m=0}^W \text{ReLU}(1 - |x_i^s - m|) \text{ReLU}(1 - |y_i^s - n|) \\ &= \sum_{n=0}^H \sum_{m=0}^W \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - n|) \\ \frac{\partial V_i}{\partial x_i^s} &= \sum_{n=0}^H \sum_{m=0}^W U_{mn} \max(0, 1 - |y_i^s - n|) \frac{\partial \max(0, 1 - |x_i^s - m|)}{\partial x_i^s} \\ &= \sum_{n=0}^H \sum_{m=0}^W U_{mn} \max(0, 1 - |y_i^s - n|) \times \begin{cases} 0 & \text{if } |m - x_i^s| \geq 1 \\ 1 & \text{if } m \geq x_i^s > m - 1 \\ -1 & \text{if } m < x_i^s < m + 1 \end{cases} \end{aligned}$$

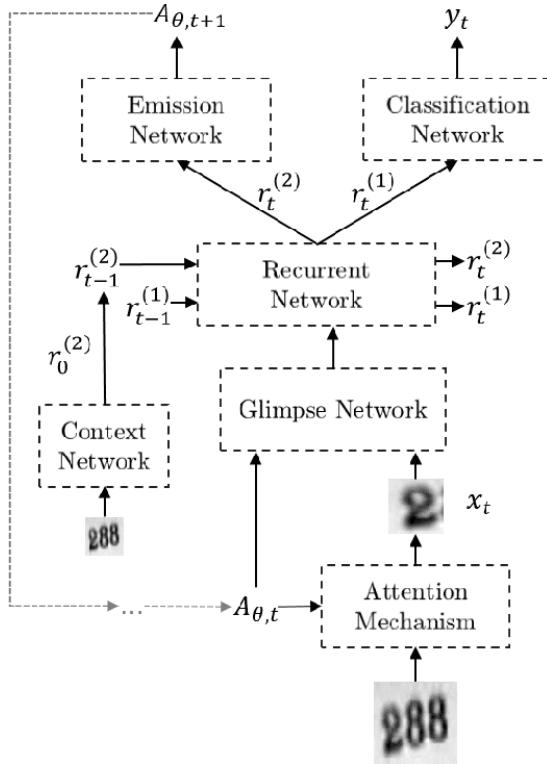
STN에서 Spatial transformer은 하나의 모듈로서, CNN 구조에 삽입하는 것도 가능하며, dimension이 줄어들기 때문에 연산 속도가 빨라지기도 한다. 또한 여러 물체를 인식하기 위해 spatial transformer를 병렬로 연결하여 사용할 수도 있다.

9.5.2 EDRAM

DRAM까지는 강화학습을 이용해 물체의 위치를 잡아냈었지만, 여기서는 Spatial Transformer를 사용하여 강화학습 부분을 없앴다.



EDRAM 기본적인 구조는 DRAM에 Spatial Transformer를 합친 모습이다. 즉, DRAM에 있던 Recurrent Network, Glimpse Network, Emission Network, Classification Network, Context Network는 기본적으로 존재한다. 논문에 소개된 EDRAM의 목표는 이전의 DRAM을 total differentiable하게 만들어 SGD를 시행하는 것이다.

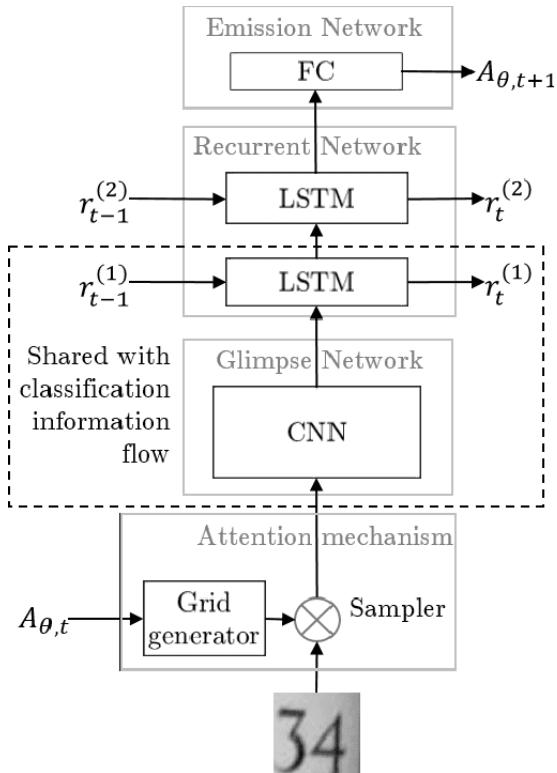


input image는 Attention Mechanism (Spatial Transformer)에 들어가 적절히 조절된 그림이 Glimpse Network로 들어간다. 기존 DRAM에서 Emission Network는 location l_t 를 생성했으나, 여기서는 Spatial Transformer에서 사용되는 parameter $A_{\theta,t}$ 를 Attention Mechanism (Spatial Transformer)으로 보내준다. parameter $A_{\theta,t}$ 는 transformed image x_t 와 함께 Glimpse Network로 들어간다. Glimpse Network로 들어간 transformed image x_t 는 convolution layer와 FC를 거치며, $A_{\theta,t}$ 는 FC 하나를 거친다.

기존의 STN을 사실 두 가지 문제가 있다 :

- Transformation parameter는 localization 과정에서 생성되는데 이 때문에 parameter가 많아졌다. 이는 계산량의 증가를 말한다.
- 기본적으로 module 형식이기 때문에 이전 정보와 상관없이 module에 들어오는 image만을 이용해 output(transformed image)을 생성했다. 만약 STN이 이전 정보를 이용할 수 있다면 더 좋은 정확도를 보일 수 있을 것이다.

EDRAM에서 사용되는 STN은 attention mechanism과 결합되어 있다. 즉, transformation parameter를 그림에서 바로 찾는 module 형식이 아니라, Emission Network에 의존해서 얻어낸다. 논문에서는 Glimpse Network는 이전과 같이 classification LSTM에 필요한 glimpse feature vector g_t 를 생성하는 역할도 하지만, 더불어 STN의 Localization Network의 역할도 수행하게 하여 문제를 해결한다. 즉, classification과 transformation information flow에 영향을 준다. 이는 Glimpse Network를 통한 classification error가 Emission Network의 localization error에 영향을 주며 그 반대도 성립한다는 것이다. 또한 이렇게 역할을 통합하면 parameter가 매우 줄어든다.



기본적으로 논문에 소개된 transformation parameter 의 개수는 6개 이다 :

$$A_{\theta,t} = \begin{bmatrix} \theta_{1,t} & \theta_{2,t} & \theta_{3,t} \\ \theta_{4,t} & \theta_{5,t} & \theta_{6,t} \end{bmatrix}$$

STN 은 이를 이용해 mesh grid 를 찾아야 한다.

$$\begin{pmatrix} x_i^s \\ y_i^s \end{pmatrix} = A_{\theta,t} \begin{pmatrix} x_i^g \\ y_i^g \\ 1 \end{pmatrix} = \begin{bmatrix} \theta_{1,t} & \theta_{2,t} & \theta_{3,t} \\ \theta_{4,t} & \theta_{5,t} & \theta_{6,t} \end{bmatrix} \begin{pmatrix} x_i^g \\ y_i^g \\ 1 \end{pmatrix}$$

이 때 $\theta_{1,t}, \theta_{5,t}$ 는 zoom, $\theta_{2,t}, \theta_{4,t}$ 는 skewness, $\theta_{3,t}, \theta_{6,t}$ 는 center position 을 결정한다. pixel value 를 찾는 과정은 이전에 언급한 bilinear interpolation 방법을 그대로 사용한다.

RAM 의 경우 강화학습이 필요했기에 그 과정이 복잡했으나, EDRAM 에서는 attention 을 이용해 이를 없앴다. loss function 은 다음과 같이 주어진다 :

$$L = \frac{1}{N} \sum_{i=1}^S \sum_{j=1}^N (\alpha_1 L_{i,j}^y + \alpha_2 L_{i,j}^{A_\theta})$$

이 때 N 은 하나의 target (object) 에 대한 시행 회수이며, S 는 하나의 image 안에 있는 target (object) 의 개수이다. α_1, α_2 는 $L_{i,j}^y$ 와 $L_{i,j}^{A_\theta}$ 의 영향력을 조절해준다. 각각은 무엇을 의미하는지 살펴보자.

$$L_{i,j}^y = -\log p_{i,j,y_{gt}}$$

$$L_{i,j}^{A_\theta} = \sum_{k=1}^6 \beta_k (\theta_{k,i,j} - \theta_{k,i,j^{gt}})^2$$

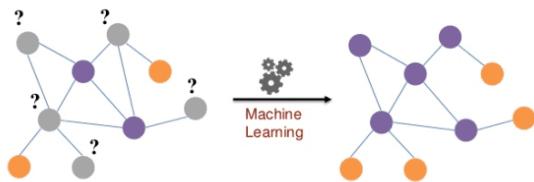
즉, $L_{i,j}^y$ 는 classification 의 cross entropy 이다. 이 때 y_{gt} 는 ground truth position 이다. $L_{i,j}^{A_\theta}$ 는 localization error 이며 $\theta_{k,i,j^{gt}}$ 는 ground truth value 이다. β_k 는 중요한 parameter (이를테면 width, height) 에 대해 더 많은 attention 을 취하기 위한 값이다. 물론 ground truth value 를 얻는 과정은 힘들 것 이다.

10 Lecture 10 : Graph Representation Learning

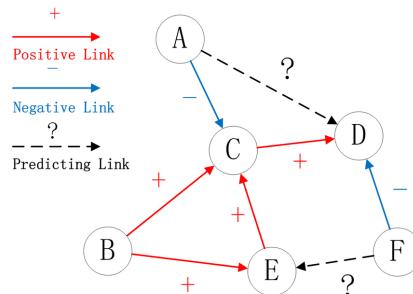
10.1 Graph Representation Learning (GRL)

GRL은 빠르게 성장하고 있는 ML의 한 분야로 graph 구조를 가진 데이터를 학습하는 기술이다. 이는 약물 설계나 social network에서 친구 추천 등 다양한 응용 분야가 있는 기술이다. 이를테면 초기의 아마존의 경우 잘 팔리는 물건을 기준으로 모든 사람들에게 (취향에 상관없이) 추천했다. 그러던 어느 날 한 통계학자가 graph 형식으로 고객들의 취향을 분석하는 것을 제안했고, 그 해 매출이 30% 상승하고 기존 직원들은 많이 해고되었다고 한다. graph에 대한 ML의 과제는 다음과 같다 :

- Node classification : 주어진 node의 type을 결정한다.

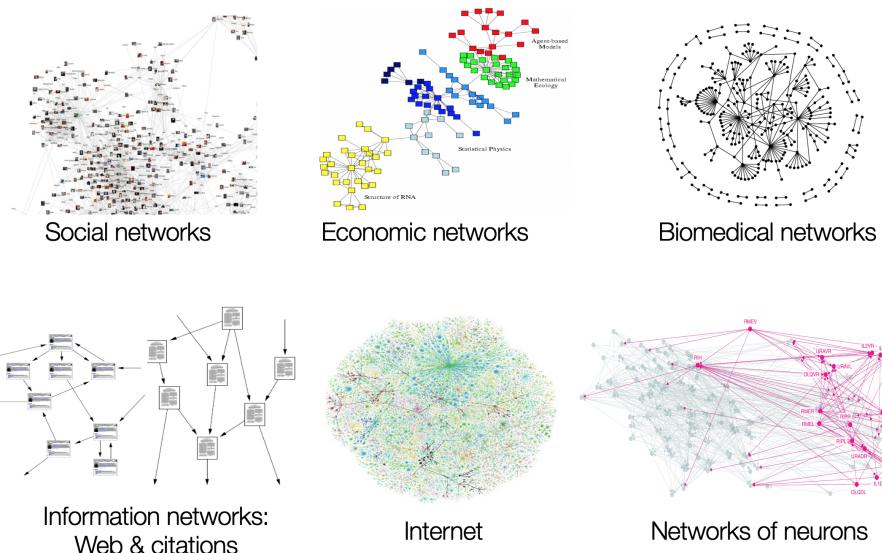


- Link prediction : 두 node가 연결되었는지 결정한다.

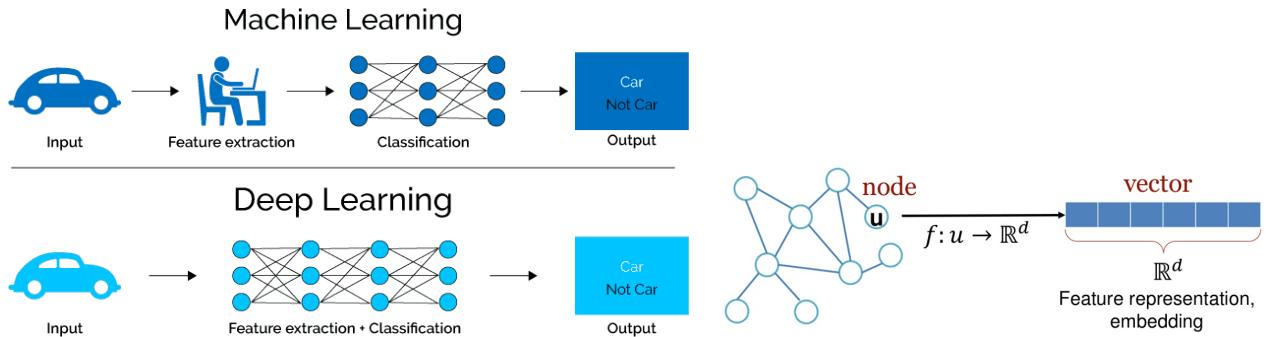


- Community detection : node가 어떻게 cluster되어 있는지 결정한다.
- Network similarity : 두 network가 얼마나 유사한지 결정한다.

다음은 graph 형식으로 되어있는 다양한 data의 예시이다.

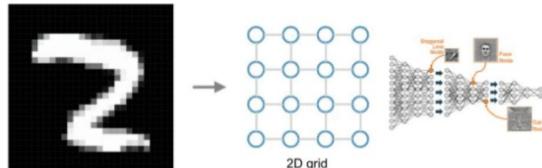


Machine Learning 의 경우 사람이 직접 데이터의 모든 feature 를 알아서 feature extraction 해야 했다. 하지만 이는 매우 불편하고 시간이 많이 소요되며 feature 를 모르면 작업할 수 없다는 단점이 있었다. graph 의 경우는 feature extraction 하는 것이 거의 불가능하다. 이를테면 문자구조를 사람이 직접 feature extraction 하는 방법은 아직까지 없다.



Deep Learning 에 들어 데이터를 machine 에 넣으면 알아서 feature 를 찾아 output 을 만들 수 있게 되었다. graph data 는 각 vertex (node) 가 나타내는 data 가 있고, node 간의 관계를 나타내는 data 가 있다. 우측 그림은 이 모두를 embedding 하는 모형이다. 여기서는 graph data 를 벡터로 변환해 Deep Learning 하는 방법을 알아볼 것 이다. 우선 GRL 이 어려운 이유를 잠시 살펴보자.

- 기존의 DL 기법들은 간단한 grid 나 sequence 에 대한 것 이었다.
 - CNN : 고정된 크기의 image 나 grid 에 대한 것 이었다.



- RNN : sequence 에 대한 것 이었다.



- 반면 graph data 는 많이 복잡하다. 복잡한 위상학적 구조를 가지고 있기에 (원래 graph 의 정의는 vertex 와 각 vertex 간의 연결이다.) grid 처럼 spatial locality 를 가지고 있지도 않고, 고정된 node 순서나 기준이 되는 reference point 가 존재하지 않아 isomorphism problem (똑같은 graph 를 표현하는 서로 다른 그림이 많다.) 이 생기기도 한다. 또한 node 나 link 는 유일하지 않고 다양한 종류의 node (사람, 동물, 기계 등) 나 link (정방향, 양방향, 굽기 등) 가 존재할 수 있다.

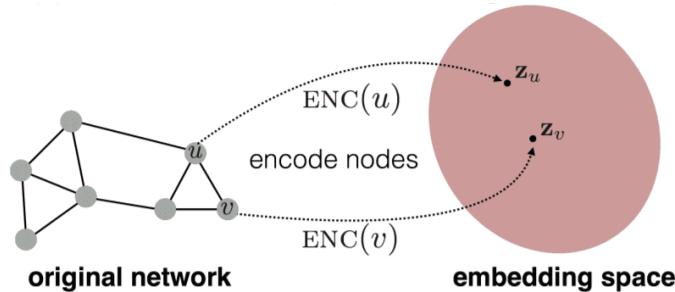
우리는 다음과 같은 틀을 가지고 GRL 을 살펴볼 것 이다.

- **Node Embedding** (shallow) : 각각의 node 를 저차원 벡터로 사상한다.
 - Matrix factorization based methods : Laplacian Eigenmaps, Graph Factorization, GraRep, HOPE
 - Random walk based algorithms : DeepWalk, node2vec
 - LINE, HARP, ...
- **Graph Neural Network** : node embedding 된 것을 가지고 graph 전반에 대한 DL 기법으로 feature 를 학습한다.

- Graph Convolutional Network (GCN)
 - GraphSAGE
 - Gated Graph Neural Network (GGNN)
 - Graph Attention Network (GAT)
 - Subgraph embeddings
- Generative graph model : 사실같은 graph data 를 만드는 방법을 학습한다. (신약 개발과 같은 경우 GAN을 결합하여 사용하나, 여기서는 언급하지 않는다.)

10.2 Node Embedding (shallow)

기본적인 idea 는 node (vertex) 가 graph 에서 갖는 위치와, 그가 인접한 node 간의 구조 정보를 담고 있는 저차원 벡터를 생성하는 것 이다.

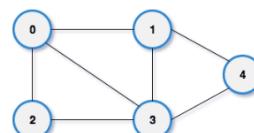


이것은 DNN 은 아니고 직접 feature extraction 하기 때문에 ML 의 고전적인 방법이다. 직관적으로 생각해 봤을 때 word2vec 에서 그랬듯이 비슷한 구조를 갖는 node 는 생성된 저차원벡터가 비슷한 위치에 있을 것 이다. 이렇게 node embedding 을 한 결과를 ML 에 넣으면 GRL 의 목적을 달성할 수 있을 것 이다.

다음은 graph 에 대한 adjacency matrix 예시이다 (이를테면 B 는 C 보다는 A 와 유사성이 높을 것 이다.) :

$$\begin{array}{c} \text{graph: } \\ \begin{array}{ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array} \end{array}$$

다음과 같은 graph 를 고려해보자. 각각의 node 는 자기 자신으로 순환하는 link 를 가지고 있다고 가정하자.



adjacency matrix 는 graph로부터 바로 읽어낼 수 있으며, node embedding 작업을 통해 graph의 정보를 담은 **embedding lookup** 을 우측과 같이 얻었다고 가정하자.

$$\text{adjacency matrix : } A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad \text{embedding lookup : } Z = \begin{bmatrix} 0.5 & 0.6 & 0.0 & 0.0 & 0.2 \\ 0.1 & 0.2 & 0.0 & 0.0 & 0.3 \\ 0.1 & 0.1 & 0.2 & 0.0 & 0.1 \\ 0.3 & 0.1 & 0.8 & 1.0 & 0.4 \end{bmatrix}$$

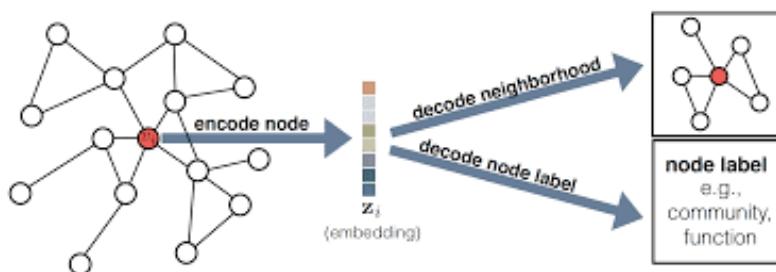
이 때 node '1'에 대한 node embedding 은 다음과 같다.

$$\mathbf{z}_1 = \begin{bmatrix} 0.6 \\ 0.2 \\ 0.1 \\ 0.1 \end{bmatrix}$$

즉, graph $G = (V, E)$ (V : node space, E : 쌍을 이룬 node 들의 집합) 라고 할 때, 다음과 같은 정의들이 있다.

- **node feature matrix** : $X \in \mathbb{R}^{F \times |V|}$ where F : input feature dimension per node
⇒ 각 노드가 갖는 feature 를 나타내는 matrix 이다. 각 column 은 각 node feature 에 대한 vector 이다. 해당 subsection 에서는 다루지 않으며, 다음 subsection 에서 다룬다.
- **node feature matrix** : $A \in \mathbb{R}^{|V| \times |V|}$
⇒ 고전적으로 사용되는 matrix 이다. 각 node 가 연결되어있는가 아닌가를 나타내는 binary matrix 이다. 물론 link 가 이를테면 '친밀도' 일 경우 그 정도에 따라 값을 실수로 조정해야 한다.
- **node embedding** : $\mathbf{z} \in \mathbb{R}^N$ where $N \ll |V|$. (**embedding lookup** : Z)
⇒ 이것을 찾는 것이 node embedding 이다. 위 예시의 경우 1은 2 보다는 0과 내적값이 높다.

Encoder-Decoder approach : Encoder 는 각각의 노드 v 가 graph에서 갖는 위치, v 와 인접한 node 의 구조, v 의 feature 의 정보 등을 담고 있는 node embedding \mathbf{z}_v 를 찾는다. Decoder 는 \mathbf{z}_v 로부터 우리가 원하는 정보를 추출한다. 이를테면 v 와 인접한 node 들을 찾거나, v 에 해당하는 label 을 classification 한다. 이렇게 Encoder 와 Decoder 를 함께 학습시키면 더욱 좋은 node embedding 을 찾아낼 것이다.



다음은 학습 과정의 요약이다.

1. Encoder ENC : $V \rightarrow \mathbb{R}^N$

graph에서 node embedding 을 계산한다. learnable parameter 를 사용한다.

2. Decoder DEC : $\mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}^+$

encoder 는 graph에서 직접 vector를 만들었지만, decoder의 경우 vector를 이용해서 바로 graph를 만든다면 원래 graph와 만들어진 graph를 비교할 수가 없다. decoder는 vector를 이용해서 ‘node의 유사성’을 계산한다. 이 ‘유사성’은 정하기 나름이기 때문에, learnable parameter가 아니다.

3. Similarity $s_G : V \times V \rightarrow \mathbb{R}^+$

원래 graph에서 node 간에 ‘유사하다’라는 것을 미리 정의해놓고 계산한 값이다. Decoder에서 계산한 값과 원래 graph에서 계산한 값이 비슷하기를 바라게 될 것이다.

4. Loss : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \Rightarrow$ decoded similarity value $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$ 와 true similarity value $s_G(v_i, v_j)$ 간의 차이를 계산한다.

5. Objective :

$$\mathcal{L} = \sum_{(v_i, v_j) \in V \times V} \text{Loss}[\text{DEC}(\mathbf{z}_i, \mathbf{z}_j), s_G(v_i, v_j)]$$

모든 pair에 대해서 loss를 합한 것이 최소화 되도록 학습을 진행한다. 여기서 loss, dec, similarity를 무엇으로 사용하는지에 따라 종류가 나뉜다.

Encoder-Decoder approach는 학습 방식에 따라 Matrix factorization과 Random walk 방식이 있으며, 이에 대한 요약이 다음과 같다.

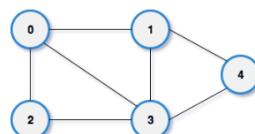
Type	Method	Decoder	Proximity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps [4]	$\ \mathbf{z}_i - \mathbf{z}_j\ _2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_G(v_i, v_j)$
	Graph Factorization [1]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	GraRep [9]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, \dots, \mathbf{A}_{i,j}^k$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	HOPE [44]	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
Random walk	DeepWalk [46]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec [27]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$ (biased)	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

10.2.1 Matrix factorization

graph의 모든 정보를 알고 있어야 계산이 가능하다. 때문에 많은 계산이 필요하다. $Z = [\mathbf{z}_i]$, $S = [s_G(v_i, v_j)]$ 라고 하면 Laplacian Eigenmaps를 제외한 다른 경우 다음과 같이 matrix 형태로 표시되기 때문에 그 이름이 붙었다.

$$\mathcal{L} = \|Z^T Z - S\|_2^2$$

이전에 봤던 graph를 고려해보자. 각각의 node는 자기 자신으로 순환하는 link를 가지고 있다고 가정하자.



- **Laplacian Eigenmaps** 다음과 같은 embedding lookup 을 고려하자. node 는 column 순서대로 0, 1, 2, 3, 4 라고 하자.

$$Z = \begin{bmatrix} 0.5 & 0.6 & 0.0 & 0.0 & 0.2 \\ 0.1 & 0.2 & 0.0 & 0.0 & 0.3 \\ 0.1 & 0.1 & 0.2 & 0.0 & 0.1 \\ 0.3 & 0.1 & 0.8 & 1.0 & 0.4 \end{bmatrix}$$

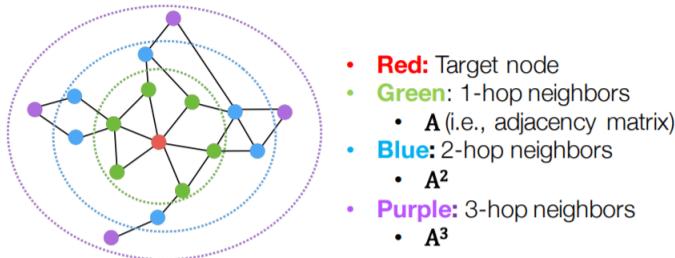
이 때 $D_{01} = 0.06$, $D_{12} = 0.50$ 이기 때문에 1와 2의 거리가 0과 1의 거리보다 멀다는 것을 알 수 있다.

- **Graph Factorization (GF)** 위의 예시를 사용했을 때 $D_{01} = 0.36$, $D_{12} = 0.1$ 이기 때문에 0 과 1의 유사성이 1와 2의 유사성보다 높다는 것을 알 수 있다. 만약 adjacency matrix 가 다음과 같다고 가정하자.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

이 때 similarity 는 $s_{01} = 1$, $s_{12} = 0$ 이다.

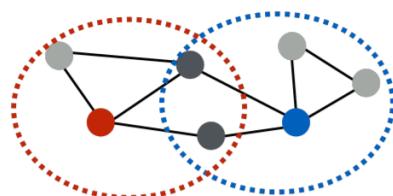
- **GraRep** GF 에서는 target node 와 인접한 것, 즉 adjacency matrix $A_{i,j}$ 만을 고려했다. 즉, $S_{ij} = s_G(v_i, v_j) = A_{i,j}$ 를 사용했다. GraRep 은 여기서 나아가 target node 에서 k 번째 떨어진 곳 (k -hop neighbor) 까지 고려한다.



또한 similarity 는 log transformed probabilistic adjacency matrix 를 사용한다.

$$S_{ij} = s_G(v_i, v_j) = \tilde{A}_{i,j}^k = \max \left[\log \left(\frac{\frac{A_{i,j}}{d_i}}{\sum_{l \in V} \frac{A_{l,j}}{d_l}} \right)^k - \alpha, 0 \right]$$

- **HOPE** node neighborhood 간의 겹치는 정도를 측정하여 similarity 로 사용한다. 즉, v_i 와 v_j 의 similarity S_{ij} 는 v_i 에서 한 칸 떨어진 node 들의 집합과 v_j 에서 한 칸 떨어진 node 들의 집합 중 겹치는 node 의 수이다. 아래 예시의 경우 빨간 점과 파란 점의 similarity 는 2 이다.



10.2.2 Random walk

graph에서 짧은 random walk를 시행했을 때 자주 같이 나오는 node들은 유사하다고 가정하는 것이 random walk embedding 방식이다. matrix representation에서는 결정된 구조 값으로 node 간의 유사성을 판단했지만, random walk 방식의 경우 **flexible stochastic similarity measure** $p_G(v_j|v_i)$ 를 사용한다. $p_G(v_j|v_i)$ 는 node v_i 에서 시작해 특정 random walk 방식을 사용했을 때 v_j 를 거칠 확률이다.

$$S_{ij} = s_G(v_i, v_j) = p_G(v_j|v_i)$$

decoder 식을 살펴보면, matrix factorization에서 사용한 값에 softmax를 적용한 것을 알 수 있다. 이는 $p_G(v_j|v_i)$ 와 같이, decoder 값을 확률과 같이 사용하기 위함이다.

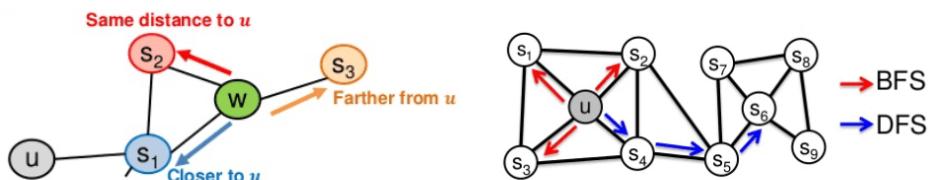
$$D_{ij} = \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) = \frac{e^{\mathbf{z}_i^T \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^T \mathbf{z}_k}}$$

loss를 정의하기 위해서는 우선 각 node에서 시작해 random walk를 시행한 후 그 중 몇 개를 sampling한 training set D 를 만들어야 한다. 하나의 node v_i 에 대해서는 확률 $(v_i, v_j) \sim p_G(v_j|v_i)$ 에 따라 N 개의 쌍을 뽑는다. 그러면 objective function은 다음과 같다.

$$\mathcal{L} = - \sum_{(v_i, v_j) \in D} \log [\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)]$$

matrix factorization의 경우 모든 node에 대해서 계산했기 때문에 연산량이 많았다. 여기서는 연산량을 줄이기 위해 두 가지 방법을 사용한다.

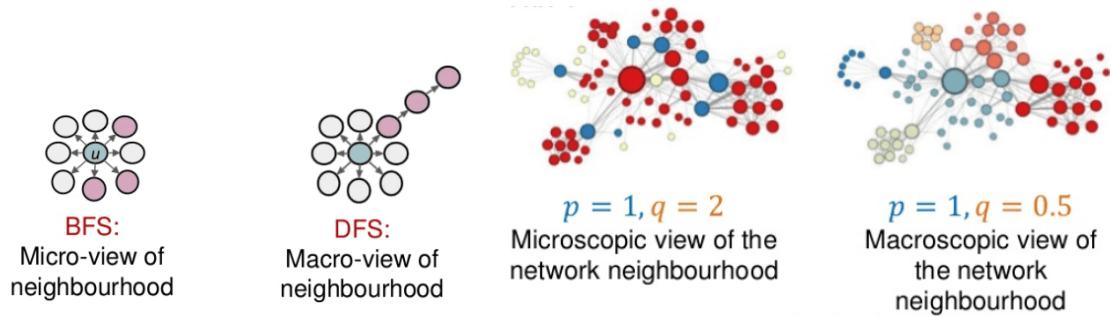
- DeepWalk의 경우에는 Decoder에서 사용되는 normalization factor $\sum_{v_k \in V} e^{\mathbf{z}_i^T \mathbf{z}_k}$ 를 계산하기 위해 **hierarchical softmax** (binary tree)를 사용한다. 연산량은 log 값으로 줄어들 것을 기대할 수 있을 것이다.
- node2vec의 경우에는 normalization factor를 **negative sampling** 방식을 이용해 계산한다. 연산량은 negative sample의 크기에 고정될 것을 기대할 수 있을 것이다. 추가로 hyperparameter p 와 q 를 정의하는데, 각각은 walk 후 u 를 기준으로 가까이 올 likelihood와 멀리 갈 likelihood이다.



이 때 W 에서 S_2 로 가는 parameter 1, S_1 으로 가는 parameter $\frac{1}{p}$, S_3 로 가는 parameter $\frac{1}{q}$ 이다. 이들을 normalize하여 확률로서 사용한다 :

$$P(S_3|W) = \frac{\frac{1}{q}}{1 + \frac{1}{p} + \frac{1}{q}}$$

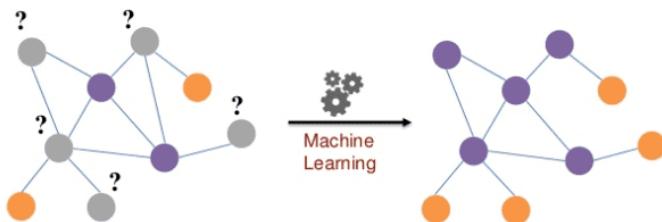
p 와 q 를 조절해 BFS (breath-first search, low p)와 DFS (depth-first search, low q) 성격을 조절하게 된다. BFS의 경우 local 구조를 강조하며 DFS는 global 구조의 역할을 강조한다.



해당 subsection에서 다른 node embedding은 ‘shallow’한 ML에 국한되었다. $O(|V|)$ 정도의 parameter가 필요했으며, 각각의 node는 유일한 embedding vector를 가졌다. 또한 transductive하여 원래 input graph에서 없던 node를 추가시키면 machine 자체를 수정하고 다시 학습해야만 했다. 또한 우리는 이제까지 node feature를 다루지 않았는데, 실제 graph는 사람/사물/장소 등 많은 feature의 node가 가능하다. 이제 우리는 encoder를 ‘deep’하게 만들어 사용하고자 한다.

10.3 Graph Neural Network (GNN)

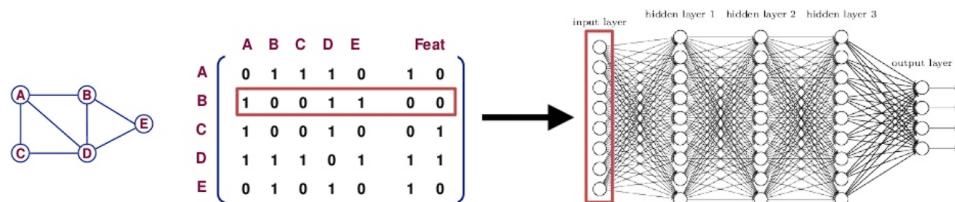
Predicting Node Properties Problem 주어진 graph에서 몇 개의 node feature는 알고 있고 나머지는 모를 때, 모르는 node들의 feature를 맞추는 semi-supervised 문제이다. 이런 문제는 어떻게 풀 수 있을까?



우리는 이 문제를 풀기 위해 다음과 같은 방법들을 고려한다.

- Graph Convolutional Network (GCN)
- GraphSAGE
- Gated Graph Neural Network (GGNN)
- Graph Attention Network (GAT)

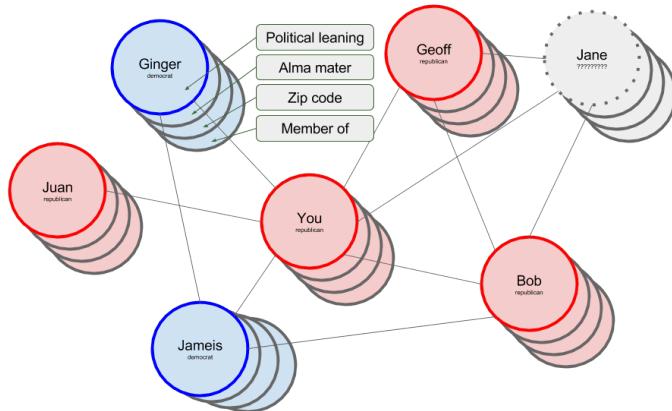
Naive Approach 다음의 예시에서는 자기순환이 없으며, matrix 첫 번째 줄은 node A에 대한 연결상태 (adjacency matrix A)와 node feature (feature matrix X)을 나타낸 것이다. 두 번째 줄은 node B에 대한 연결상태와 node feature이다. 이런 각각의 줄이 NN에 들어가는 input data가 된다.



그러나 이 방법은 node 수가 늘어날 수록 parameter 가 너무 많이 늘어난다. 또한 matrix 순서를 바꾸거나, graph 에 노드를 하나만 추가해도 machine 자체가 바뀌고 다시 학습을 해야 한다. 이상적인 GNN 은 이런 단점들을 없애주기를 바라며, 연산량을 줄이기 위해 node 의 neighbor 정도에만 계산하는 locality 특성을 갖기 바란다.

10.3.1 Graph Convolutional Network (GCN)

Graph Convolutional Network (GCN) adjacency matrix, node feature 를 input 으로 받아 node embedding 하는 NN 이다.



우선 다음과 같은 graph $G = (V, E)$ 가 주어졌다고 가정하자.

- feature matrix $X \in \mathbb{R}^{|V| \times F}$
- adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$

우리의 목적은 몇 개의 layer function f 를 통해 node embedding $f(X, A) = Z \in \mathbb{R}^{|V| \times F'}$ 를 만드는 것이다. 이 때 $F \gg F'$ 으로 차원이 줄어야 한다.

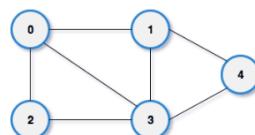
l 번째 layer 의 layer vector H^l 을 다음과 같이 정의하자. 우리는 layer 를 쌓아 layer vector H^l 를 X 로 시작해 Z 로 만들 것이다.

$$\begin{aligned} H^{(0)} &= X \\ H^{(l+1)} &= f(H^{(l)}, A) \\ H^{(L)} &= Z \end{aligned}$$

우선 가장 단순하게 생각해 보자. f 를 다음과 같이 정의해보자.

$$f(H^{(l)}, A) = \sigma(AH^lW^l)$$

A 의 차원은 $|V| \times |V|$ 이므로 H^l 의 차원은 $|V| \times \cdot$ 이어야 한다. 또한, $H^{(0)} = X$ 의 차원은 $|V| \times F$ 이며 여기서 시작해 $H^{(L)} = Z$ 의 차원 $|V| \times F'$ 으로 끝나야 한다. 이를테면 다음의 graph 를 보자.



adjacency matrix 와 layer vector 를 이용한 계산은 다음과 같다.

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \quad H^{(l)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \quad AH^{(l)}W^{(l)} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 1 & 1 \end{bmatrix} W^{(l)}$$

여기서 문제가 두가지 발생한다. 첫째로, 모든 node 는 자기 자신의 정보를 버리게 한다. node ‘1’ 은 node ‘0’, ‘3’, ‘4’ 에 연결되어 있으며 이는 A 의 두번째 row 에 나타나 있다. 이것이 $H^{(l)}$ 과 곱해지면서 $H^{(l)}$ 의 1, 4, 5 번째 row 의 정보를 가져간다. 즉, node ‘0’, ‘3’, ‘4’ 의 정보만 가져가고 node ‘1’, ‘2’ 의 정보는 버린다. 즉, A 와 $H^{(l)}$ 가 곱해지는 과정에서 각 node 는 자기 자신의 정보를 버린다. 때문에 우리는 graph 에 **자기순환**을 추가한다.

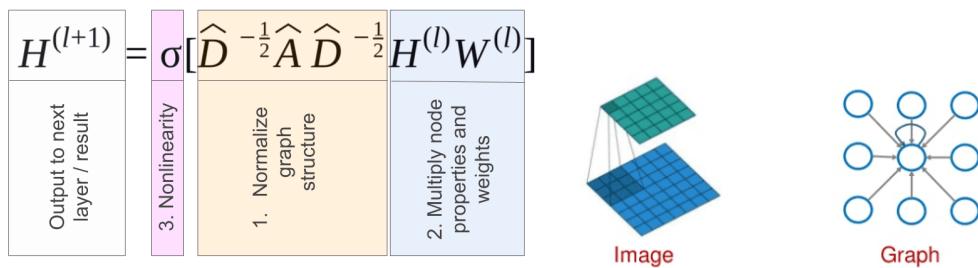
$$\tilde{A} = A + I$$

두번째 문제는, $AH^{(l)}$ 를 계산하면 연결이 많은 node 에 해당하는 row 의 합이 큰 것을 확인할 수 있다. 하지만 우리는 ‘분포’ 만 원한다. 때문에 **degree matrix** D 를 다음과 같이 정의한다. diagonal component 는 각 node 가 갖고 있는 (자기순환을 포함한) 연결 회수이다.

$$\tilde{D} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix}$$

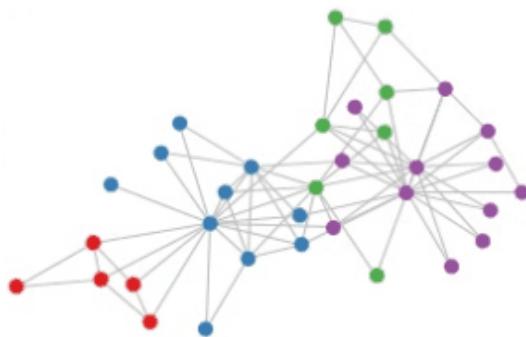
이제 $\tilde{D}^{-1}\tilde{A}$ 를 이용하면 될 것 같지만, symmetric 하게 만들어주는 것이 더 좋다:

$$A \rightarrow \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{2\sqrt{3}} & \frac{1}{2\sqrt{5}} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{2\sqrt{5}} & \frac{1}{2\sqrt{3}} \\ \frac{1}{2\sqrt{3}} & 0 & \frac{1}{3} & \frac{1}{\sqrt{15}} & 0 \\ \frac{1}{2\sqrt{5}} & \frac{1}{2\sqrt{5}} & \frac{1}{\sqrt{15}} & \frac{1}{5} & \frac{1}{\sqrt{15}} \\ 0 & \frac{1}{2\sqrt{3}} & 0 & \frac{1}{\sqrt{15}} & \frac{1}{3} \end{bmatrix}$$



이 과정은 convolution 과 많이 닮아있다. 1.에 해당하는 $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ 는 처음 한 번만 계산한다. 그리고 $H^{(l)}$ 에 대한 계산을 진행할 때 $W^{(l)}$ 은 모든 node 에 대해 같다. 즉, 하나의 layer 에서 $W^{(l)}$ 은 마치 convolution 에서의 filter 와 같이 작용한다.

다음은 Zachary’s Karate club graph 예시이다. 34명의 멤버들은 각자가 친한 community 와 모여 있고 이들은 같은 색으로 나타냈다. 만약 색이 주어지지 않았다면 어떻게 해야 할까? GCN 에서 3개의 layer에 대해 random initialized weight $W^{(l)}$ 을 가지고, feature 없이 진행했을 때 300회의 iteration 후 잘 작동하는 것을 알 수 있었다.



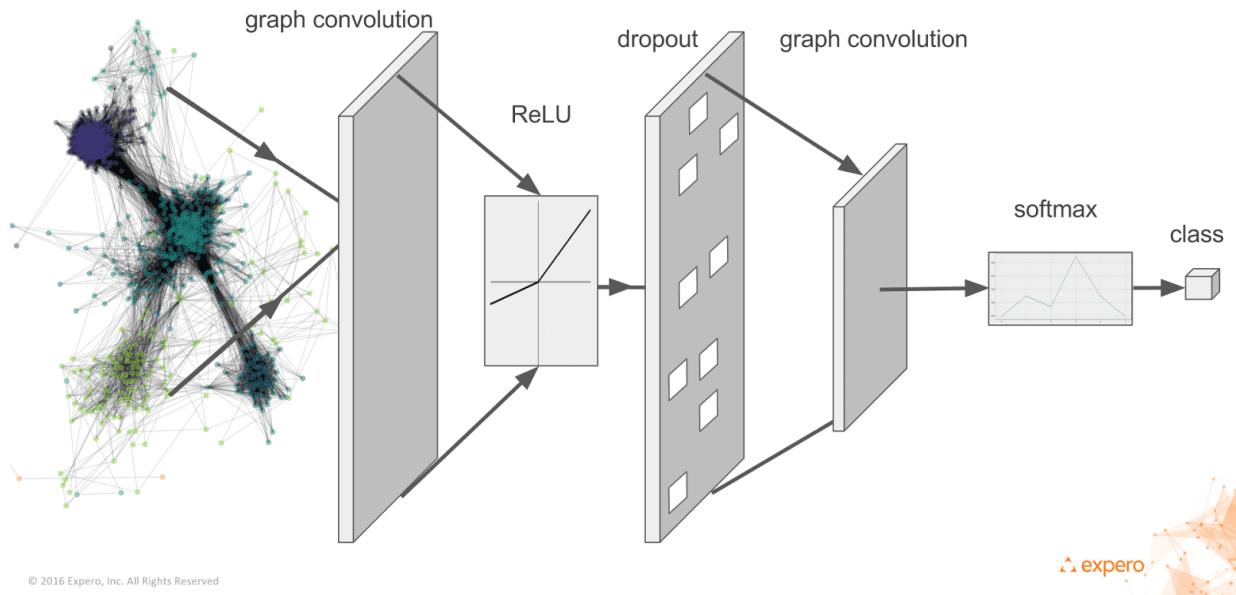
다음은 Facebook friendship network 를 Semi-supervised node classification 하는 예시이다. 두개의 layer 가 있으며 그 식을 이어보면 다음과 같다.

$$Z = f(X, A) = \text{softmax}(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \text{ReLU}(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X W^0) W^{(1)})$$

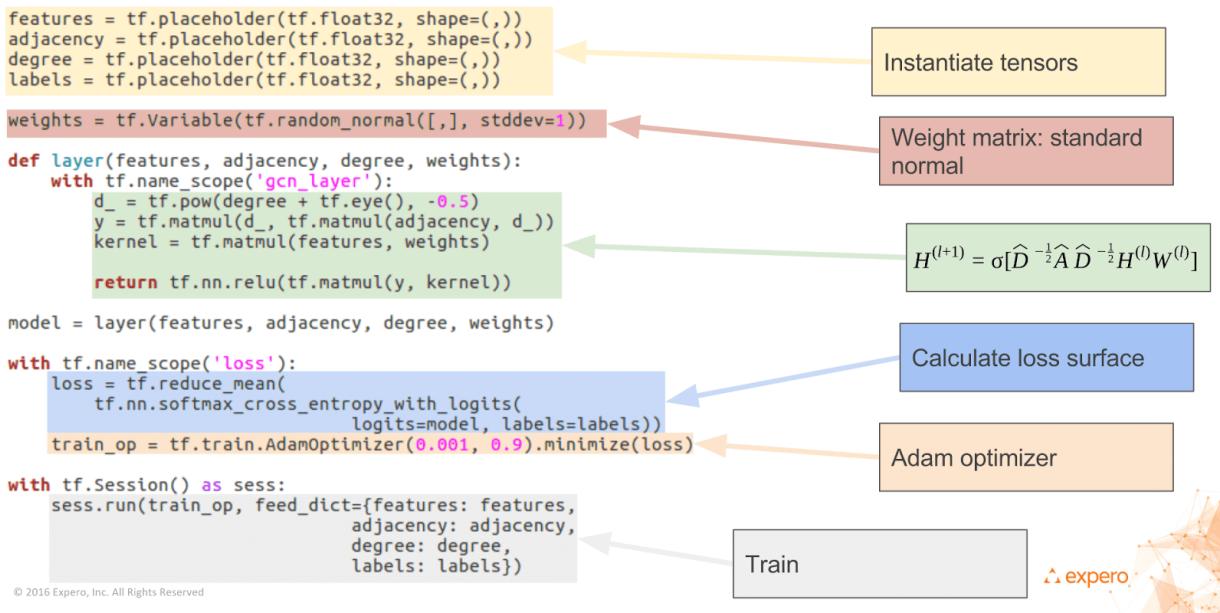
labeled 되어있는 training data 에 대해 마지막에 classification 해야 하므로 loss function 으로서 cross-entropy 를 사용한다.

$$\mathcal{L} = - \sum_{l \in I_L} \sum_{f=1}^F Y_{lf} \ln Z_{lf}$$

이 때 I_L 은 labeled node index set 이며 Y_l 은 index l 에 해당되는 label 이다.

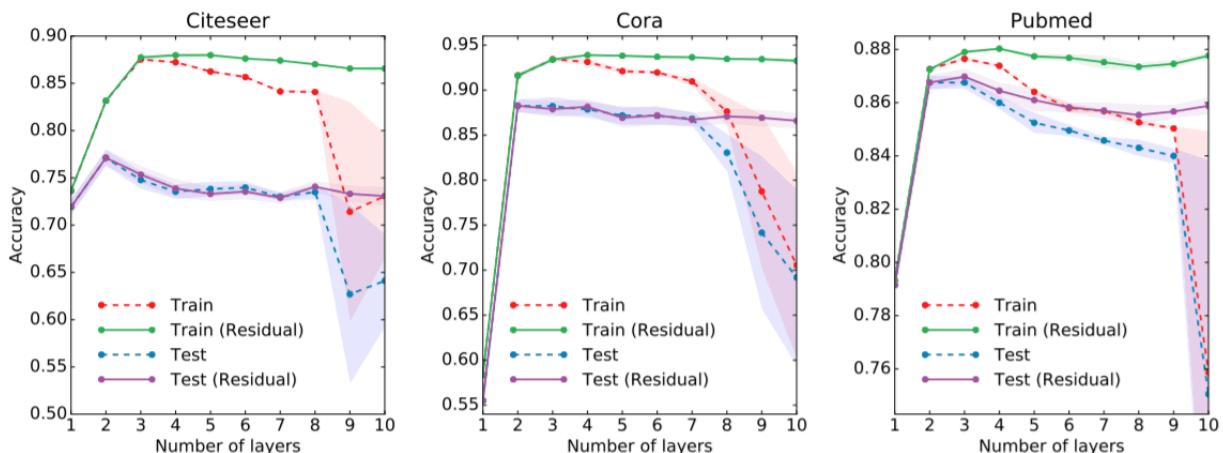


이에 대한 tensorflow code 이다.



model depth 다음은 위에서 설명한 GCN 과 residual GCN 을 다양한 데이터에 적용하여 비교한 그래프이다. 400 epoch, Adam($\gamma = 0.01$) 등의 옵션을 적용했다. **residual GCN** 은 GCN 에 residual 방식을 적용한 것이다. CNN 에서 다뤘듯이, residual 방식은 layer를 적용하면서 이전 state 를 identity 로 가져온다.

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) + H^{(l)}$$



layer 개수가 늘어날 수록 일반적인 GCN 은 그 성능이 떨어지는 것을 볼 수 있으며, 2 - 3 개 정도의 layer 가 적당한 것을 볼 수 있다. layer 가 7개를 넘어가면 그만큼 target 에서 먼 node 까지 고려한다는 의미이므로 그 성능이 떨어지며 overfitting 된다.

Relation to Weisfeiler-Lehman Algorithm 고전적인 algorithm 으로, GNN 에서 새로운 방법이 나오면 자주 이것과 비교하고는 한다. hash function 으로 dimension 을 줄인다는 특징이 있다.

Algorithm 1: WL-1 algorithm (Weisfeiler & Lehmann, 1968)

Input: Initial node coloring $(h_1^{(0)}, h_2^{(0)}, \dots, h_N^{(0)})$
Output: Final node coloring $(h_1^{(T)}, h_2^{(T)}, \dots, h_N^{(T)})$
 $t \leftarrow 0;$
repeat
 for $v_i \in \mathcal{V}$ **do**
 $h_i^{(t+1)} \leftarrow \text{hash} \left(\sum_{j \in \mathcal{N}_i} h_j^{(t)} \right);$
 $t \leftarrow t + 1;$
 until stable node coloring is reached;

이 때 $h_i^{(t)}$ 는 node v_i 의 (iteration t 에서) label 이며 \mathcal{N}_i 는 v_i 의 neighborhood node 의 index 들이다. 이 알고리즘을 NN 처럼 학습 가능하도록 미분 가능하게 만들고자 하면 hash function 을 다음과 같이 바꿀 수도 있다.

$$\mathbf{h}_i^{(l+1)} = \sigma \left(W^{(l)} \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)} \right)$$

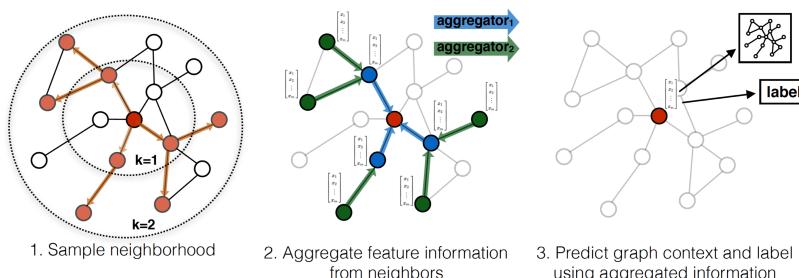
이 때 c_{ij} 는 (v_i, v_j) 에 대해 적절히 고른 정규화 상수이다. 또한 이제 $h_i^{(t)}$ 는 layer l 에서 node i 에 대한 activation vector 가 된다. $W^{(l)}$ 는 layer l 에서의 weight matrix이며 σ 는 미분 가능한 비선형 함수이다. 여기서 $c_{ij} = \sqrt{d_i d_j}$ ($d_i = |\mathcal{N}_i|$) 를 고르면 GCN 의 vector form 을 얻는다.

$$\mathbf{h}_i^{(l+1)} = \sigma \left(W^{(l)} \sum_{j \in \mathcal{N}_i} \frac{\mathbf{h}_j^{(l)}}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}} \right)$$

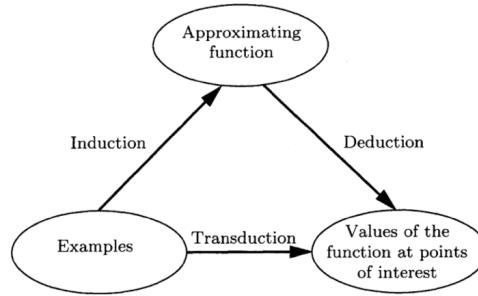
즉, 넓은 의미에서 GCN 은 Weisfeiler-Lehman Algorithm 을 미분 가능하고 learnable 하게 바꾼 일반화이다.

10.3.2 GraphSAGE

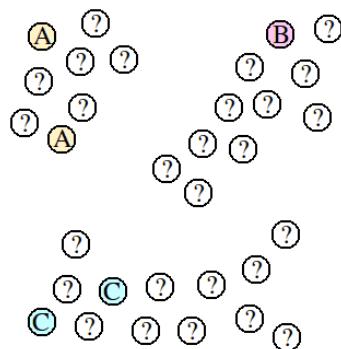
SAmpLE + aggREGate 이전까지 다른 GCN 은 graph 크기가 바뀌면 모델이 계속해서 바뀌어야 했다. 지금부터는 graph 크기에 관계 없이 같은 모델을 사용할 수 있는 방법에 대해 다룬다. GraphSAGE 는 각 노드에 대해 고정된 크기의 neighborhood 들을 sampling 하고 여기에 aggregator (mean or LSTM) 를 적용한다. sampling 을 적용하는 inductive method 이기 때문에 graph size 에 독립적이다.



Transductive vs Inductive



- **Induction** : data 몇개가 주어졌을 때 이를 설명하는 function 을 찾는 것 이다. 지금까지 대부분의 DNN 이 이 방식이었다.
- **Deduction** : 함수가 주어졌을 때 값을 대입하여 결과를 얻는 것 이다.
- **Transduction** : data 몇개를 알 때 function 없이 특정한 점에 대한 값을 알아내는 방법이다.

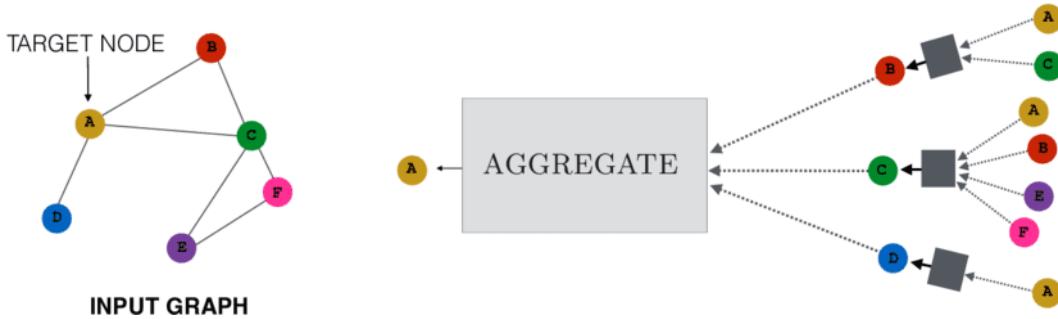


위 예시의 경우, 직관적으로 봤을 때 clustering 을 할 수도 있을 것이다. 그런데 이 과정에서 우리는 모든 데이터를 봤고, unlabeled data 가 뭉쳐있다는 것을 이용해, 즉 transductive 방법을 통해 clustering 한 것 이다. 반면 inductive 는 우리가 아는 label 을 통해 규칙을 만들고, 그 규칙에 따라 모르는 label 을 분류하게 된다.

transductive 방법은 unlabeled 된 data 가 많아도 학습이 잘 진행된다. 하지만 새로운 데이터가 추가될 경우 다시 학습을 진행해야 한다는 단점이 있다. 따라서 inductive 를 사용해야 할 것 같지만, inductive 는 많이 어렵다. 이미 알고 있는, label data 가 많아야 하는데 이런 경우는 드물기 때문에 학습이 어렵다. graphSAGE 는 GCN (semi-supervised, transductive) 방식을 확장해서 inductive unsupervised 방식으로 바꾼다.

Neighborhood Aggregation 특정 노드의 근처에 있는 것들만 모아 node embedding 을 생성한다. (기존 node embedding 은 전체 구조가 필요했으며 feature 를 사용하기 어려웠다.) 때문에 연산이 빠르고 feature 도 자유롭게 사용할 수 있다.

다음과 같은 예시의 경우 target node 가 A 일 때, 이와 인접한 B, C, D 에 대한 정보를 aggregate 한다. (depth-1) 이것이 하나의 layer 이다. 물론 각각의 B, C, D 는 그것들과 인접한 node 의 정보를 aggregate 한 것 이다. (depth-2) layer 가 많아질 수록 더 먼 곳의 정보까지 합해질 것으로 예상할 수 있다.



또한 depth 가 2 이상이 되면 B와 C 가 연결되지 않았다고 가정했을 때 A에 들어가는 B 와 C 의 정보는 더 적을 것 이다. target 에서 직접 연결된 것만 중요한 것이 아니라, target 의 neighborhood 가 서로 어떻게 연결되었는지도 similarity 에 영향을 주게 되는 것 이다. 다음은 GraphSAGE 에서 aggregation 하여 node embedding 을 찾는 algorithm 이다.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $G(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K ; weight matrices $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$; non-linearity σ ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations \mathbf{z}_v for all $v \in \mathcal{V}$

```

1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
2 for  $k = 1 \dots K$  do
3   for  $v \in \mathcal{V}$  do
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ ;
5      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
6   end
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
8 end
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 

```

이때 K 는 depth, AGGREGATE_k 는 iteration k 에서의 (learnable) differentiable aggregator, \mathbf{W}^k 는 iteration k 에서의 (learnable) weight function 이다. 알고리즘을 보면 우선 $\mathbf{h}_v^0 = \mathbf{x}_v$ =input feature 로 초기화 한다.

iteration k 에서는 모든 node 들에 대해 알고리즘을 적용하는데 우선 하나의 target v 가 정해졌을 때를 고려해 보자. 우선 target v 의 neighborhood set $\mathcal{N}(v)$ 안에 있는 모든 node 에 대한 이전 단계 node vector \mathbf{h}_u^{k-1} 에 대하여 ‘aggregate’ 를 적용하여 neighbor vector $\mathbf{h}_{\mathcal{N}(v)}^k$ 를 만든다. 이제 $\mathbf{h}_{\mathcal{N}(v)}^k$ 와 \mathbf{h}_v^{k-1} 를 concatenate 하고 적절한 과정을 거쳐 \mathbf{h}_v^k 를 생성한다. concatenate 는 보통 벡터를 붙이는 것을 말하지만, 사실은 concatenate 함수를 그때그때 정의하기 나름이다.

depth K 만큼의 계산이 끝나면 결과적으로 node embedding \mathbf{z}_v 를 얻는다. 이를 목적에 맞는 loss function 을 적용하고 SGD 를 적절히 사용하면 AGGREGATE_k 와 \mathbf{W}^k 를 학습할 수 있을 것 이다.

aggregation 과정에서는 node 의 순서가 고려되지 않으며, 고려되어서도 안된다. graph $G = (\mathcal{V}, \mathcal{E})$ 안에 node 순서에 대한 정보는 원래 없었기 때문이다. 따라서 Aggregator 는 symmetric 하게 만들어야 한다. 하지만 이는 어렵기 때문에 간혹 비대칭을 허용하기도 한다. 다음은 aggregator 의 종류이다.

- Mean aggregator : 가장 단순하게, 평균을 내는 방법이다. 학습이 필요 없다.

$$\begin{aligned}
 \mathbf{h}_{\mathcal{N}(v)}^k &= \sum_{u \in \mathcal{N}(v)} \frac{\mathbf{h}_u^{k-1}}{|\mathcal{N}(v)|} \\
 \mathbf{h}_v^k &= \sigma[\mathbf{W}_k \mathbf{h}_{\mathcal{N}(v)}^k + \mathbf{B}_k \mathbf{h}_v^{k-1}] \\
 &= \sigma \left[\mathbf{W}_k \sum_{u \in \mathcal{N}(v)} \frac{\mathbf{h}_u^{k-1}}{|\mathcal{N}(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1} \right]
 \end{aligned}$$

혹은 다음과 같이 neighbor 는 물론 자기 자신까지 평균내는 방법이 있다. 물론 학습은 필요가 없다.

$$\mathbf{h}_v^k = \sigma \left[\mathbf{W}_k \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{\mathbf{h}_u^{k-1}}{|\mathcal{N}(v)| + 1} \right]$$

식을 보면 GCN 과 아주 유사하다는 것을 알 수 있다.

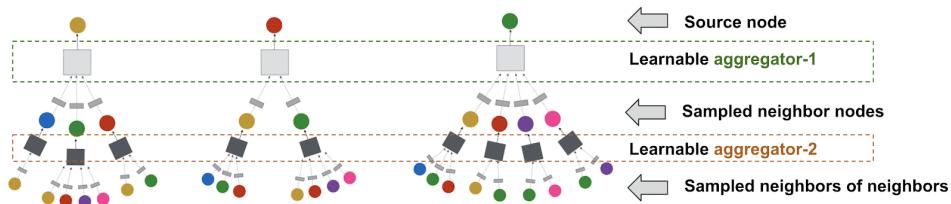
- LSTM aggregator : link 가 너무 많은 경우 단순히 평균을 내면 정보가 사라지기 때문에 node 에 임의로 순서를 부여해 LSTM 으로 학습하는 방법이다. symmetric 이 아니어도 되는 경우에 사용한다.

$$\text{AGGREGATE}_k^{\text{LSTM}} = \text{LSTM}(\mathbf{h}_{u_i}^{k-1}) \quad \text{where } u_i \in \mathcal{N}(v)$$

- Pooling aggregator : neighbor 들에 대한 node vector 를 모두 대입한 뒤 maximum 을 잡는다. symmetric 한 방법이다.

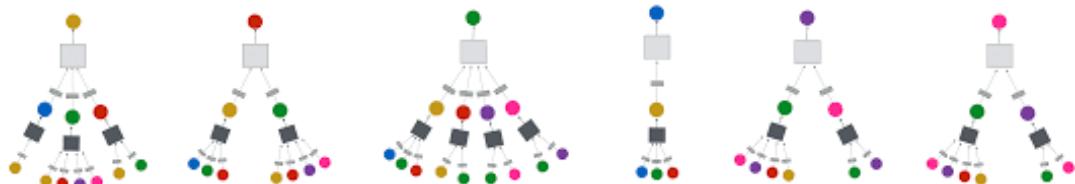
$$\text{AGGREGATE}_k^{\text{pool}} = \max \left\{ \sigma(\mathbf{W}_{\text{pool}} \mathbf{h}_u^{k-1} + \mathbf{b}) \mid u \in \mathcal{N}(v) \right\}$$

Inductivity 어째서 GraphSAGE 가 inductive 라는 것일까? inductive 하기 위해서는 node 가 추가되어도 모델이 바뀌면 안된다. depth-2 인 다음과 같은 모델을 고려해 보자.

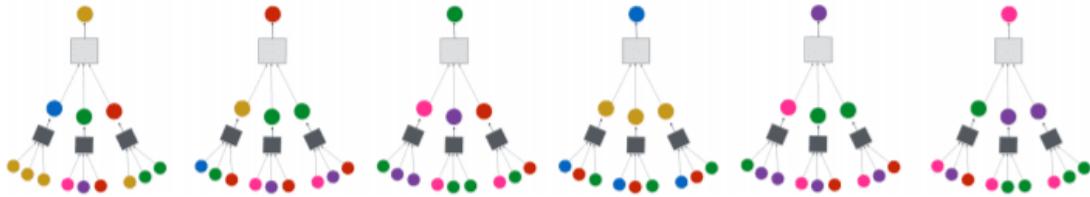


GraphSAGE는 같은 layer 에서 같은 weight 를 사용한다. (weight sharing) 즉, aggregator-1 이라고 표시된 부분의 weight 는 모두 같고, aggregator-2 라고 표시된 부분의 weight 는 모두 같다. 때문에 node 가 추가되어도 같은 모델을 사용하게 되고, 계산량도 많이 줄어들게 된다. 만약 node 가 추가되고 graph 모양이 매우 커져도 각 node 에 대한 학습 과정은 유지된다. 때문에 inductive 하다. 이런 특징을 이용하면 이전에 없던 새로운 graph 를 생성할 수도 있다.

algorithm GraphSAGE 는 SGD를 minibatch 방식으로 진행하기 때문에 하나의 graph 안에서 데이터를 나눠줘야 한다. 하지만 node 에 따라 연결되는 link 수가 다르기 때문에 학습하는 batch 개수를 정할 수 없을 것으로 예상된다. 이를테면 다음과 같은 경우 각 node 에 연결되는 link 수가 제각기 다르다.



대안으로 우리는 link 수를 고정하여 학습을 진행한다. (때문에 ‘SAmple’ 이라는 말이 들어간다.) 정해놓은 link 수 (random 으로 정한다) 보다 적을 경우 중복이 발생하는 것을 볼 수 있다.



다음은 GraphSAGE 의 minibatch forward propagation algorithm 이다. neighborhood 를 뽑는 함수 \mathcal{N}_k 는 미리 정해서 모델링 해야 한다.

Algorithm 2: GraphSAGE minibatch forward propagation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
 input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
 depth K ; weight matrices $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$;
 non-linearity σ ;
 differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$;
 neighborhood sampling functions, $\mathcal{N}_k : v \rightarrow 2^{\mathcal{V}}, \forall k \in \{1, \dots, K\}$

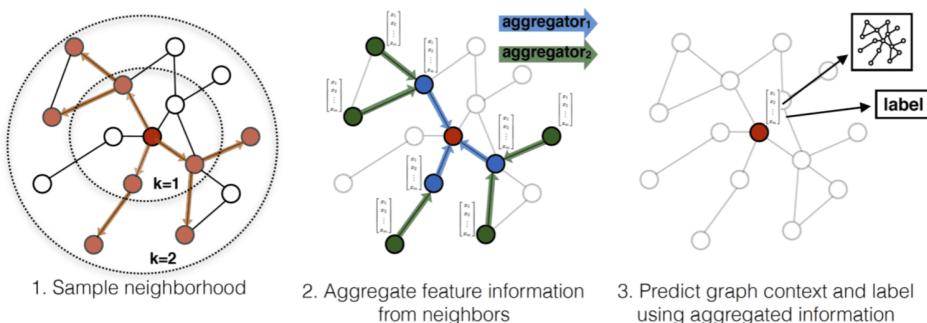
Output : Vector representations \mathbf{z}_v for all $v \in \mathcal{B}$

```

1  $\mathcal{B}^K \leftarrow \mathcal{B}$ ;
2 for  $k = K \dots 1$  do
3    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$  ;
4   for  $u \in \mathcal{B}^k$  do
5      $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$ ;
6   end
7 end
8  $\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$  ;
9 for  $k = 1 \dots K$  do
10  for  $u \in \mathcal{B}^k$  do
11     $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$ ;
12     $\mathbf{h}_u^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k))$ ;
13     $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$ ;
14  end
15 end
16  $\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$ 

```

줄 2 - 7 는 각 node 에 대한 aggregate 계산을 하기 위해 필요한 neighbor node 를 sampling 하는 과정이다. sampling 과정은 minibatch \mathcal{B} 에서 depth K 가 되는 \mathcal{B}^0 까지 진행된다. 그 다음은 aggregation 과정이다. 이를 요약하자면 다음과 같다.



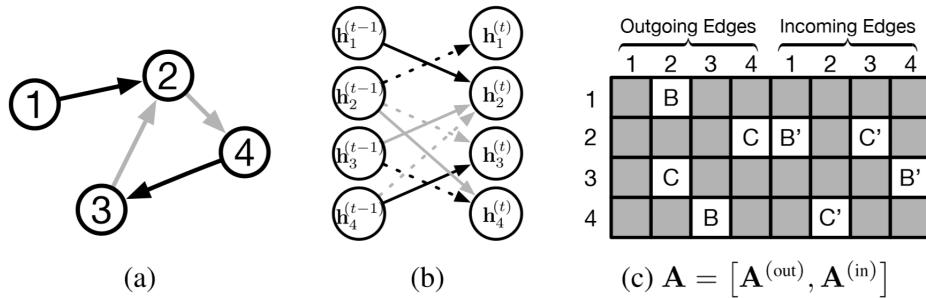
Training Unsupervised learning 이기 때문에 적절한 target 이 없다. 따라서 node embedding 의 유사성을 기준으로 보는데, Objective function 은 다음과 같다.

$$J_G(\mathbf{z}_v) = -\log (\sigma(\mathbf{z}_v^T \mathbf{z}_u)) - Q \cdot \mathbb{E}_{u_n \sim P_n(u)} \left[\log [\sigma(-\mathbf{z}_v^T \mathbf{z}_{u_n})] \right]$$

우선 첫번째 행의 \mathbf{z}_u 는 v 의 neighbor 중 하나의 node embedding 이다. 이 행은 인접한 node 의 embedding 곱이 커지도록 한다. 두번째 행은 negative sampling 을 사용한다. Q 는 negative sample 의 개수이며 v_n 은 negative sample distribution $P_n(v)$ 에 따라 sampling 되었다. 또한 이 때 negative sample 은 v 의 neighbor 가 아닌 것 만을 고른다. 즉, v 와 멀리있는 것과의 embedding 곱이 작아지도록 한다.

10.3.3 Gated Graph Neural Network (GGNN)

GraphSAGE 에서 propagation 을 없애고 recurrent model 인 GRU 를 사용한다. GraphSAGE 의 경우 2 - 3 개 정도의 layer 까지만 사용 가능했고 간단한 graph 는 이것으로 충분했다. 하지만 graph 가 복잡해지면 더 먼 node 끼리의 관계도 알 필요가 있고, GGNN은 20 정도의 면 node 간의 관계도 알아낼 수 있다. 적절한 timestep T 를 정해 recurrent layer 를 unroll 하여 사용한다.



가장 좌측에 주어진 graph $G = V, E$ 를 고려해 보자. node ‘1’에서 ‘2’로 가는 검정색 선은 type ‘B’ link 이다. node ‘2’ 입장에서 보면 ‘1’에서 ‘2’로 들어오는 type ‘B’ link 이다. 각 node 별로는 hidden state h 가 있는데, timestep t 일때 node ‘1’, ‘2’, ‘3’, ‘4’ 순서대로 $h_1^{(t)}, h_2^{(t)}, h_3^{(t)}, h_4^{(t)}$ 로 나타내어 진다.

가운데 그림은 이 hidden state 들이 timestep 에 따라 어떻게 영향을 주는지 나타내고 있다. node ‘1’에서 ‘2’로 가는 type ‘B’ link 가 있기에 이는 $h_1^{(t-1)}$ 에서 $h_2^{(t)}$ 로 가는 검은색 실선으로 나타내어져 있다. 반면 node ‘2’ 입장에서 보면 type ‘B’ link 가 거꾸로 연결되어있는 것이기 때문에 $h_2^{(t-1)}$ 에서 $h_1^{(t)}$ 로 가는 검은색 점선으로 나타내어져 있다. 회색 선은 type ‘C’ 라고 하자.

가장 우측 그림은 가운데 그림을 matrix 형태로 나타낸 것 이다. 각 row 는 어떤 node 를 기준으로 하는가 이고 column 은 어디로 나가는 것 인지 (outgoing), 혹은 어디서 들어오는 것 인지 (incoming) 를 나타낸 것 이다. 가운데 그림을 기준에서 나타난 점선은 incoming 으로 matrix 상에서 prime 표시한다. 이 matrix 를 $A \in \mathbb{R}^{D|V| \times 2D|V|}$ 라고 하자. 우리의 예시에서는 node feature 를 고려하지 않기 때문에 $D = 1$ 이다.

특정 node v 의 initial hidden state $h_v^{(1)}$ 는 일반적으로 node feature \mathbf{x}_v 을 사용하나, dimension 이 부족할 경우 그 목적에 따라 0 - padding 을 적절히 추가하여 만든다.

$$h_v^{(1)} = \begin{bmatrix} \mathbf{x}_v \\ 0 \end{bmatrix}$$

그리고 이를 이용해 GRU 에 넣을 input a_t^v 를 만든다. 각 node 에 대한 정보가 aggregation 되는 과정이다.

$$a_t^v = A_v^T \begin{bmatrix} h_1^{(t-1)} \\ \vdots \\ h_{|V|}^{(t-1)} \end{bmatrix} + \mathbf{b}$$

이 때 \mathbf{b} 는 bias vector 이며 dimension $2D$ 이다. 또한 A_v 는 어떤 node 에 대한 outgoing 과 incoming

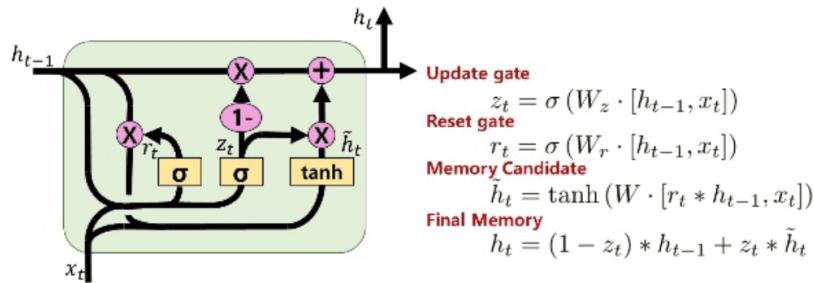
정보를 합친 matrix 이다. 이를테면 node ‘2’에 대한 A_2 는 다음과 같다.

$$A_2 = \begin{bmatrix} B & 0 \\ 0 & 0 \\ C & 0 \\ 0 & C' \end{bmatrix}$$

이제 a_t^v 를 GRU 에 넣는다.

$$\begin{aligned} z_t^v &= \sigma(W_z a_t^v + U_z h_v^{(t-1)}) \\ r_t^v &= \sigma(W_r a_t^v + U_r h_v^{(t-1)}) \\ \tilde{h}_v^{(t)} &= \tanh(W_h a_t^v + U_h(r_t^v \odot h_v^{(t-1)})) \\ h_v^{(t)} &= (1 - z_t^v) \odot h_1^{(t-1)} + z_t^v \odot \tilde{h}_v^{(t)} \end{aligned}$$

이 때 σ 는 sigmoid function 이며 \odot 은 elementwise product 이다. 원래 GRU 와 완전히 같은 식임을 알 수 있다.



10.3.4 Graph Attention Network (GAT)

GraphSAGE 에 **masked self-attention** 을 추가한 방법이다. 즉, graph 의 전체적인 구조를 모르는 상태로 aggregation 하는 과정에서 각각의 neighbor node feature 에 서로 다른 중요도를 부여하는 방식이다.

GraphSAGE 에서 그랬던 것처럼, 하나의 layer 만을 고려해 보자. 우리의 목적은 layer input 으로 node feature vectors $\{h_1, h_2, \dots, h_{|V|}\}$ ($h_i \in \mathbf{R}^F$) 를 받아 layer output 으로 새로운 node feature vectors $\{h'_1, h'_2, \dots, h'_{|V|}\}$ ($h'_i \in \mathbf{R}^{F'}$) 을 만드는 것이다. 이 때 F 는 기존 number of features 이며 $F' < F$ 는 number of higher level features 이다.

벡터의 차원을 바꾸기 위해서는 우선 선형변환이 있어야 할 것 같다. 때문에 하나의 layer에 들어온 모든 node 에 대해 같은 weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ 를 적용한다 :

$$h_i \rightarrow \mathbf{W}h_i$$

그리고 node i 의 모든 neighborhoods $j \in \mathcal{N}_i$ 들에 대하여 적절한 shared attentional mechanism a 를 통해 self attention 을 적용한다. attention coefficient 는 다음과 같다.

$$\begin{aligned} a &: \mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R} \\ e_{ij} &= a(\mathbf{W}h_i, \mathbf{W}h_j) \end{aligned}$$

e_{ij} 는 node ‘j’ 의 feature h_j 가 node ‘i’ 에게 얼마나 중요한지를 나타내는 상수이다. attention mechanism a 는 후에 설명한다. e_{ij} 를 계산할 때는 node ‘i’ 를 기준으로 neighbor 에 대해서만 계산하기 때문에 ‘masked’ 라는 말이 붙었다. GraphSAGE 를 기준으로 본다면 depth-1 이라 할 수 있을 것이다.

attention coefficient 는 그 기준 node 에 따라 상대적 크기가 다르다. 때문에 크기를 맞춰줄 적절한 변형이 필요하다. 논문에서는 softmax 를 사용해 attention coefficient 를 attention value 로 만들었다.

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{e^{e_{ij}}}{\sum_{k \in \mathcal{N}_i} e^{e_{ik}}}$$

위의 attention value 를 사용하여 구한 새로운 feature vector 는 다음과 같다.

$$h'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} h_j \right)$$

이를 이용하여 여러가지 실험을 해본 결과, 논문에서는 attention mechanism a 를 single-layer feedforward NN 으로 제안했다.

$$e_{ij} = a(\mathbf{W} h_i, \mathbf{W} h_j) = \text{LeakyReLU}[\mathbf{a}^T (\mathbf{W} h_i || \mathbf{W} h_j)]$$

이 때 $\mathbf{a} \in \mathbb{R}^{2F'}$ 는 적절한 weight vector 이며 LeakyReLU 는 negative slope 0.2 를 사용했다. $||$ 는 concatenate 이다. 즉, 사실은 다음과 같이 쓸 수도 있다.

$$e_{ij} = \text{LeakyReLU}[\mathbf{a}_1^T \mathbf{W} h_i + \mathbf{a}_2^T \mathbf{W} h_j] \quad \text{where } \mathbf{a}_1, \mathbf{a}_2 \in \mathbb{R}^{F'}$$

이를 이용하면 attention value 는 다음과 같다.

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp \left[\text{LeakyReLU}[\mathbf{a}^T (\mathbf{W} h_i || \mathbf{W} h_j)] \right]}{\sum_{k \in \mathcal{N}_i} \exp \left[\text{LeakyReLU}[\mathbf{a}^T (\mathbf{W} h_i || \mathbf{W} h_k)] \right]}$$

또한 학습과정을 안정화하기 위해서는 Multi-head attention 을 사용한다. K 개의 독립적인 attention mechanism 은 서로 다른 weight 를 사용하여 다음과 같이 concatenate 된다.

$$h'_i = \left\| \sum_{k=1}^K \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k h_j \right) \right\|$$

요약하면 다음과 같다.

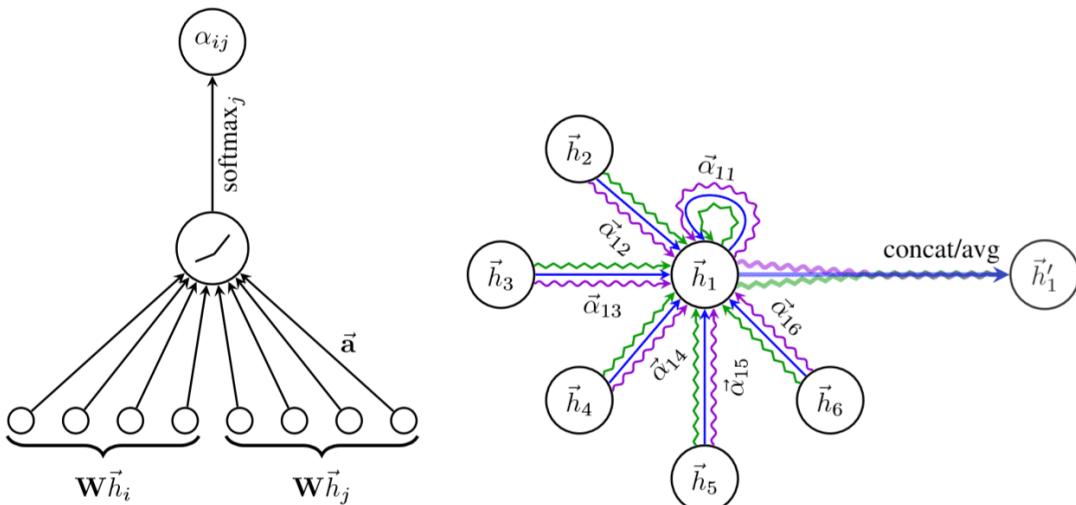


Figure 1: **Left:** The attention mechanism $a(\mathbf{W} \vec{h}_i, \mathbf{W} \vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{a} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain \vec{h}'_1 .

다음은 pre-trained GAT 의 첫번째 hidden layer 를 Cora (Coriolis Ocean database ReAnalysis) dataset 에 적용한 것 이다. 8-head attention 을 사용했다.

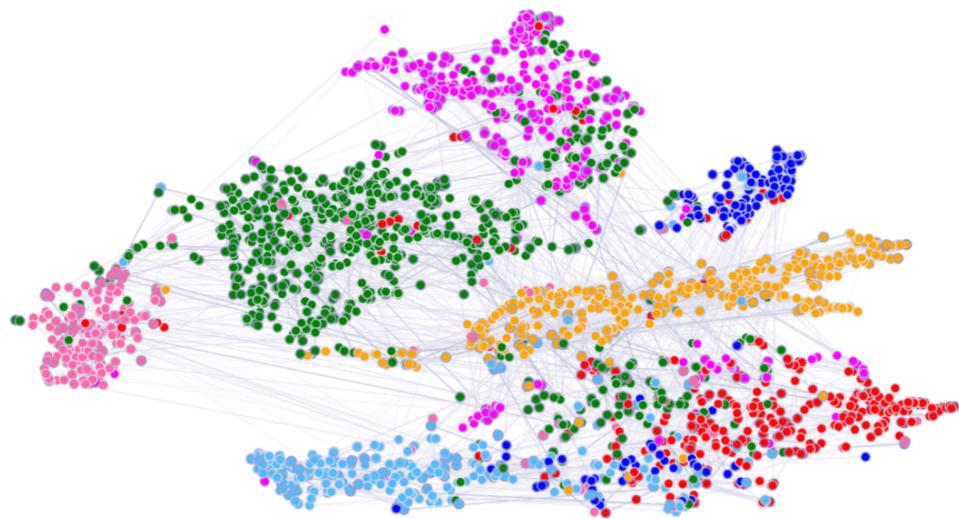


Figure 2: A t-SNE plot of the computed feature representations of a pre-trained GAT model's first hidden layer on the Cora dataset. Node colors denote classes. Edge thickness indicates aggregated normalized attention coefficients between nodes i and j , across all eight attention heads ($\sum_{k=1}^K \alpha_{ij}^k + \alpha_{ji}^k$).