Reinforcement Learning for Natural Language Processing

Richard S. Sutton – Image from Web # Reinforcement Learning for Natural Language Processing

Discriminative learning vs Generative learning

2012년 이미지넷 대회(ImageNet competition)에서 딥러닝을 활용한 AlexNet이 우승을 차지한 이래로 컴퓨터 비전(Computer Vision), 음성인식(Speech Recog-

nition), 자연어처리(Natural Language Processing) 등이 차례로 딥러닝에 의해 정복당해 왔습니다. 딥러닝이 뛰어난 능력을 보인 분야는 특히 분류(classification) 분야였습니다. 기존의 전통적인 방식과 달리 패턴 인식(pattern recognition) 분야에서는 압도적인 성능을 보여주었습니다. 이러한 분류(classification) 문제는 보통 Discriminative Learning에 속하는데 이를 일반화 하면 다음과 같습니다.

$$\hat{\theta} = argmaxP(y|x;\theta)$$

주어진 x에 대해서 최대의 y 값을 갖도록 하는 파라미터 θ를 찾아내는 것 입니다. 이러한 조건부 확률 $P(y|x)$ 분포를 학습하는 것을 discriminative learning이라고 부릅니다. 하지만 이에 반해 Generative Learning은 확률분포 $P(x)$를 학습하는 것을 이릅니다. 따라서 Generative learning이 훨씬 더 학습하기 어렵습니다. 예를 들어

1. 사람의 생김새(x)가 주어졌을 때 성별($y|x$)의 확률 분포를 배우는 것
2. 사람의 생김새(x)와 성별(y) 자체의 확률 분포를 배우는 것

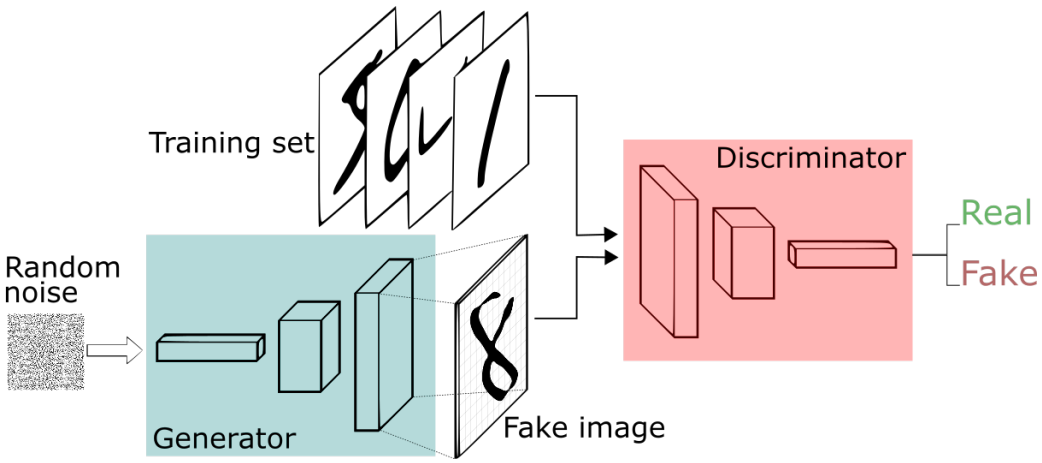
두가지 경우를 비교하면 2번째가 훨씬 더 어려움을 알 수 있습니다. Discriminative learning은 y와 x와의 관계를 배우는 것이지만, generative learning은 x $(and$ $y)$ 자체를 배우는 것이기 때문입니다. 그리고 이것을 수식으로 일반화 하면 아래와 같습니다.

$$\hat{\theta} = argmaxP(x,y;\theta)$$

사실 이제는 패턴인식과 같은 discriminative learning은 이제 딥러닝으로 너무나도 당연하게 잘 해결되기 때문에, 사람들의 관심과 연구 트렌드는 위와 같은 generative learning으로 집중되고 있습니다.

Generative Adversarial Network (GAN)

2016년부터 주목받기 시작하여 2017년에 가장 큰 화제였던 분야는 단연 GAN이라고 말할 수 있습니다. Variational Auto Encoder(VAE)와 함께 Generative learning을 대표하는 방법 중에 하나입니다. GAN을 통해서 우리는 사실같은 이미지를 생성해내고 합성해내는 일들을 딥러닝을 통해 할 수 있게 되었습니다. 이러한 합성/생성 된 이미지들을 통해, 자율주행과 같은 실생활에 중요하고 어렵지만 훈련 데이터셋을 얻기 힘든 문제들을 해결 하는데 큰 도움을 얻을 수 있으리라고 예상됩니다. (실제로 GTA게임을 통해 자율주행을 훈련하려는 시도는 이미 유명합니다.)



Generative Adversarial Network overview – Image from web

위와 같이 Generator(G)와 Discriminator(D) 2개의 모델을 각기 다른 목표를 가지고 동시에 훈련시키는 것입니다. D는 임의의 이미지를 입력으로 받아 이것이 실제 존재하는 이미지인지, 아니면 합성된 이미지인지 탐지 해 내는 역할을 합니다. G는 어떤 이미지를 생성 해 내되, D를 속이는 이미지를 만들어 내는 것이 목표입니다. 이렇게 두 모델이 잘 균형을 이루며 min/max 게임을 펼치게 되면, G는 결국 훌륭한 이미지를 합성 해 내는 Generator가 됩니다.

Why GAN is important?

마찬가지의 이유로 GAN또한 주목받게 됩니다. 예를 들어, 생성된 이미지와 정답 이미지 간의 차이를 비교하는데 MSE(Mean Square Error)방식을 사용하게 되면, 결국 이미지는 MSE를 최소화 하기 위해서 자신의 학습했던 확률 분포의 중간으로 출력을 낼 수 밖에 없습니다. 예를 들어 사람의 얼굴을 일부 가리고 가려진 부분을 채워 넣도록 훈련한다면, MSE 손실함수(loss function) 아래에서는 각 픽셀마다 가능한 확률 분포의 평균값으로 채워 질 겁니다. 이것이 MSE를 최소화 하는 길이기 때문입니다. 하지만 우리는 그런 흐리멍텅한 이미지를 잘 생성된(채워진) 이미지라고 하지 않습니다. 따라서 사실적인 표현을 위해서는 MSE보다 정교한 목적함수(objective function)를 쓸 수 밖에 없습니다. GAN에서는 그러한 복잡한 함수를 D가 근사하여 해결한 것 입니다.

GAN과 NLP

위와 같이 GAN은 컴퓨터 비전(CV)분야에서 대성공을 이루었지만 자연어처리(NLP)에서는 적용이 어려웠습니다. 그 이유는 Natural Language 자체의 특성에 있습니다. 이미지라는 것은 어떠한 continuous한 값들로 채워진 2차원의 matrix입니다. 하지만 이와 달리 단어라는 것은 discrete한 symbol로써, 결국 언어라는 것은 어떠한 descrete한 값들의 순차적인 배열 입니다. 비록 우리는 NNLM이나 NMT Decoder를 통해서 latent variable로써 언어의 확률을 모델링 $P(w_1, w_2, \cdots, w_n)$ 하고 있지만, 결국 언어를 나타내기 위해서는 해당 확률 모델에서 sampling(또는 argmax)을 하는 과정을 거쳐야 하고 이 과정은 미분이 불가능하거나 미분이 되더라도 gradient가 0이 되어 back-propagation이 불가 합니다.

$$\hat{w}_t = argmax P(w_t | w_1, \cdots, w_{t-1})$$

이러한 이유 때문에 Discriminator의 loss를 Generator에 전달 할 수가 업고, 따라서 GAN을 NLP에는 적용할 수 없는 인식이 지배적이었습니다. 하지만 강화학습을 사용함으로써 Adversarial learning을 NLP에도 우회적으로 사용할 수 있게 되었습니다.

참고로 Reparameterization Trick을 이용해 이 문제를 해결하려는 시도들도 있습니다. Gumbel Softmax [Jang at el.2016]가 대표적입니다. 이를 활용하면 gradient를 전달 할 수 있기 때문에, policy gradient 없이 문제를 해결 할 수도 있습니다.

Why we use RL?

위와 같이 GAN을 사용하기 위함 뿐만이 아니라, 강화학습은 매우 중요합니다. 어떠한 문제를 해결함에 있어서 cross entropy를 쓸 수 있는 분류(classification) 문제나, tensor간의 오류(error)를 구할 수 있는 MSE 등으로 정의 할 수 없는 복잡한 목적함수(objective function)들이 많이 존재하기 때문입니다. (비록 그동안 그러한 objective function으로 문제를 해결하였더라도 문제를 단순화 하여 접근한 것일 수도 있습니다.) 우리는 이러한 문제들을 강화학습을 통해 해결하거나 성능을 더 극대화 할 수 있습니다. 이를 위해서 잘 설계된 reward를 통해서 보다 복잡하고 정교한 task의 문제를 해결 할 수 있습니다.

이제 강화학습에 대해 간단히 다루고, 이를 NLP와 NMT에 어떻게 적용시키는지 다루어 보도록 하겠습니다. # Expectation

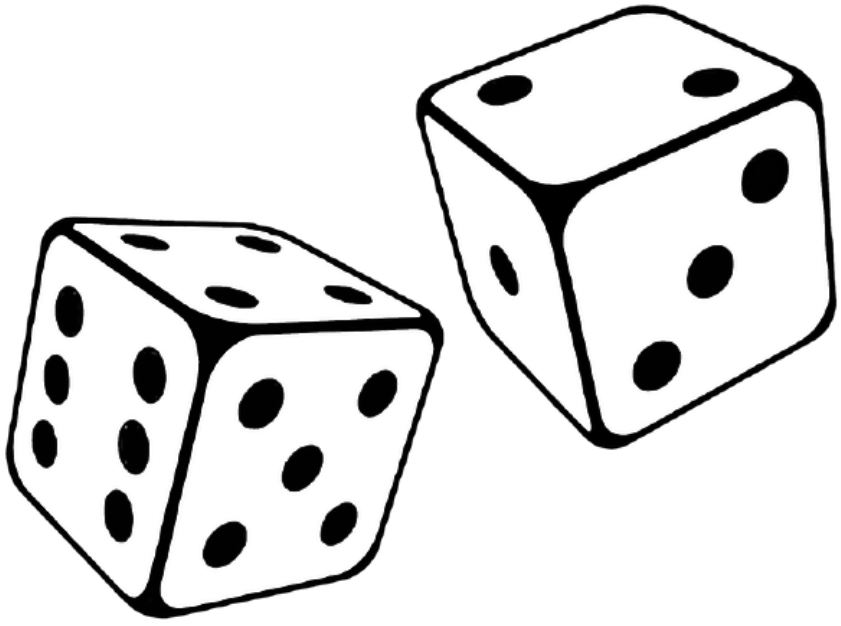


Figure 1:

기대값(expectation)은 보상(reward)과 그 보상을 받을 확률을 곱한 값의 총 합을 통해 얻을 수 있습니다. 즉, reward에 대한 가중합(weighted sum)라고 볼 수 있습니다. 주사위의 경우에는 reward의 경우에는 1부터 6까지 받을 수 있지만, 각 reward에 대한 확률은 1/6로 동일합니다.

$$expected\ reward\ from\ dice = \sum_{x=1}^{6} P(X = x) \times reward(x)$$
$$where\ P(x) = \frac{1}{6}, \forall x\ \ and\ \ reward(x) = x.$$

따라서 실제 주사위의 기대값은 아래와 같이 3.5가 됩니다.

$$\frac{1}{6} \times (1 + 2 + 3 + 4 + 5 + 6) = 3.5$$

또한, 위의 수식은 아래와 같이 표현 할 수 있습니다.

$$\mathbb{E}_{X \sim P}[reward(x)] = \sum_{x=1}^{6} P(X = x) \times reward(x) = 3.5$$

주사위의 경우에는 discrete variable을 다루는 확률 분포이고, continuous variable의 경우에는 적분을 통해 우리는 기대값을 구할 수 있습니다.

Monte Carlo Sampling

Monte Carlo Sampling은 난수를 이용하여 임의의 함수를 근사하는 방법입니다. 예를 들어 임의의 함수 f가 있을 때, 사실은 해당 함수가 Gaussian distribution을 따르고 있고, 충분히 많은 수의 random number x를 생성하여, $f(x)$를 구한다면, $f(x)$의 분포는 역시 gaussian distribution을 따르고 있을 것 입니다. 이와 같이 임의의 함수에 대해서 Monte Carlo 방식을 통해 해당 함수를 근사할 수 있습니다.

따라서 Monte Carlo sampling을 사용하면 기대값(expectation) 내의 표현을 밖으로 꺼낼 수 있습니다. 즉, 주사위의 reward에 대한 기대값을 아래와 같이 간단히(simplify) 표현할 수 있습니다.

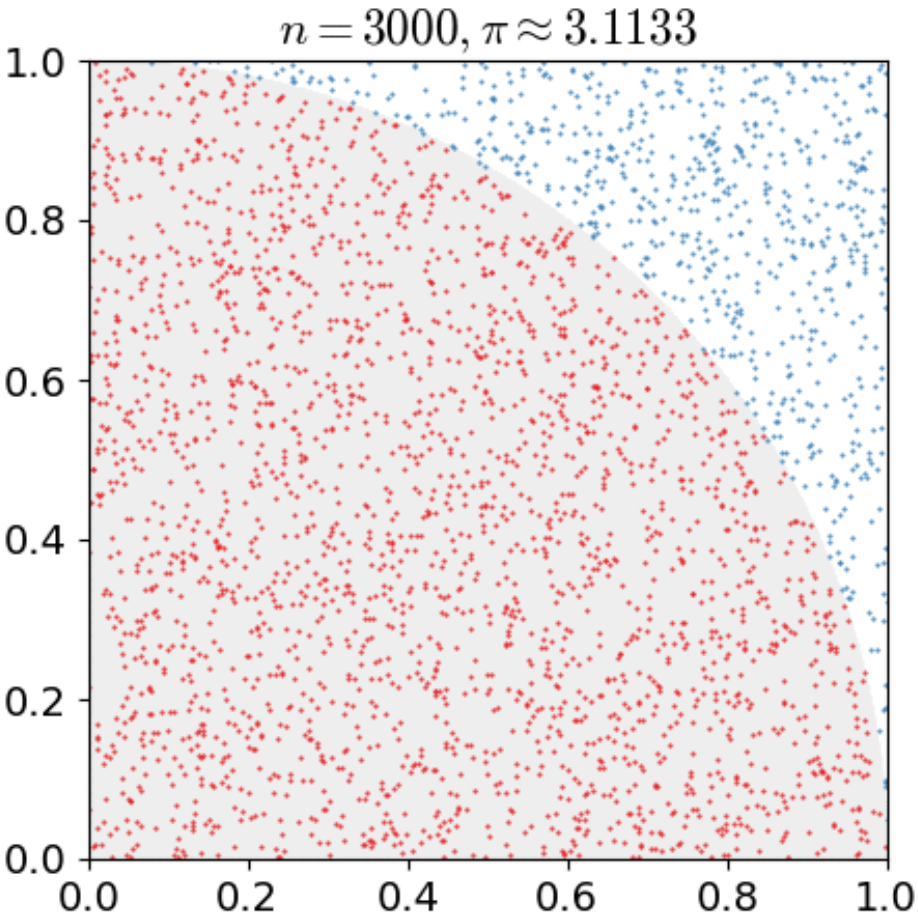


Figure 2: approximation of pi using Monte Carlo

$$\mathbb{E}_{X\sim P}[reward(x)] \approx \frac{1}{N}\sum_{i=1}^{N} reward(x_i)$$

주사위 reward의 기대값은 N번 sampling한 주사위 값의 평균이라고 할 수 있습니다. 실제로 N이 무한대에 가까워질 수록 (커질 수록) 해당 값은 실제 기대값 3.5에 가까워질 것 입니다. 따라서 우리는 경우에 따라서 $N = 1$인 경우도 가정 해 볼 수 있습니다. 즉, 아래와 같은 수식이 될 수도 있습니다.

$$\mathbb{E}_{X\sim P}[reward(x)] \approx reward(x) = x$$

위와 같은 가정을 가지고 수식을 간단히 표현할 수 있게 되면, 이후 gradient를 구한다거나 할 때에 수식이 간단해져 매우 편리합니다.

Brief About Reinforcement Learning

강화학습은 매우 방대하고 유서 깊은 학문입니다. 이 챕터에서 모든 내용을 상세하게 다루기엔 무리가 있습니다. 따라서 우리가 다룰 policy gradient를 알기 위해서 필요한 정도로 가볍게 짚고 넘어가고자 합니다. 더 자세한 내용은 Sutton 교수의 강화학습 책 Reinforcement Learning: An Introduction [Sutton et al.2017]을 참고하면 좋습니다.

Universe

먼저, 강화학습은 어떤 형태로 구성이 되어 있는지 이야기 해 보겠습니다. 강화학습은 어떠한 객체가 주어진 환경에서 상황에 따라 어떻게 행동해야 할지 학습하는 방법에 대한 학문입니다. 그러므로 강화학습은 아래와 같은 요소들로 구성되어 동작 합니다.

처음 상황(state) S_t $(t = 0)$을 받아서 agent는 자신의 정책에 따라 행동(action) A_t를 선택합니다. 그럼 environment는 agent로부터 action A_t를 받아 보상(reward) R_{t+1}과 새롭게 바뀐 상황(state) S_{t+1}을 반환합니다. 그럼 agent는 다시 그것을 받아 action을 선택하게 됩니다. 따라서 아래와 같이 state, action reward가 시퀀스(sequence)로 주어지게 됩니다.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, \cdots$$

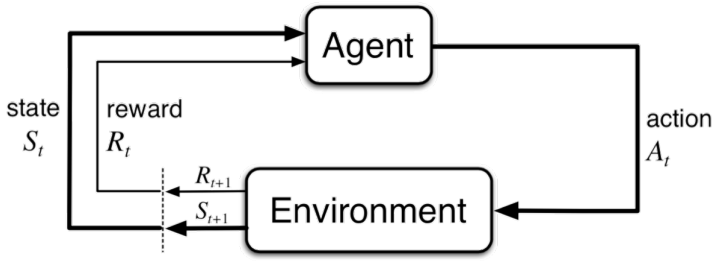


Figure 3:

특정 조건이 만족되면 environment는 이 시퀀스를 종료하고, 이를 하나의 에피소드(episode)라고 합니다. 반복되는 에피소드 하에서 agent를 강화학습을 통해 적절한 행동(보상을 최대로)을 하도록 훈련하는 것이 우리의 목표 입니다.

Markov Decision Process

그리고 여기에 더해서 Markov Decision Process (MDP)라고 하는 성질을 도입합니다.

우리는 온세상의 현재(present) $T = t$ 이 순간을 하나의 상황(state)으로 정의하게 됩니다. 그럼 현재상황(present state)이 주어졌을 때, 미래 $T > t$는 과거 $T < t$로부터 독립(independent)이라고 가정 합니다. 그럼 이제 우리 세상은 Markov process(마코프 프로세스)상에서 움직인다고 할 수 있습니다. 이제 우리는 현재 상황에서 미래 상황으로 바뀔 확률을 아래와 같이 수식으로 표현할 수 있습니다.

$$P(S'|S)$$

여기에 Markov decision process(마코프 결정 프로세스)는 결정을 내리는 과정, 즉 행동을 선택하는 과정이 추가 된 것 입니다. 풀어 설명하면, 현재 상황에서 어떤 행동을 선택 하였을 때 미래 상황으로 바뀔 확률이라고 할 수 있습니다. 따라서 그 수식은 아래와 같이 표현 할 수 있습니다.

$$P(S'|S, A)$$

이제 우리는 MDP 아래에서 environment(환경)와 agent(에이전트)가 state와 reward, action을 주고 받으며 나아가는 과정을 표현 할 수 있습니다.

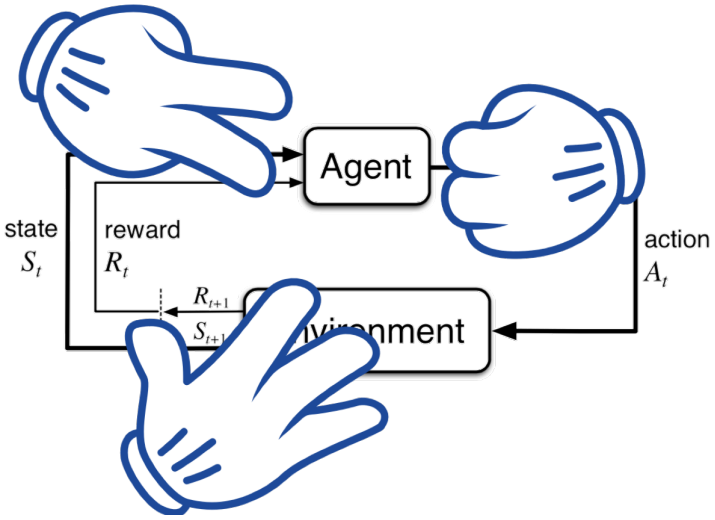


Figure 4:

Reward

앞서, agent가 어떤 행동을 선택 하였을 때, 환경(environment)으로부터 보상(reward)을 받는다고 하였습니다. 이때 우리는 G_t를 어떤 시점으로부터 받은 보상의 총 합이라고 정의 합니다. 따라서 G_t는 아래와 같이 정의 됩니다.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

이때 우리는 discount factor γ를 도입하여 수식을 다르게 표현 할 수도 있습니다. γ는 0과 1 사이의 값으로, discount factor가 도입됨에 따라서 우리는 먼 미래의 보상보다 가까운 미래의 보상을 좀 더 중시해서 다룰 수 있게 됩니다.

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

Policy

Agent는 주어진 상황(state)에서 앞으로 받을 보상(reward)의 총 합을 최대로 하도록 행동해야 합니다. 마치 우리가 눈 앞의 즐거움을 참고 미래를 위해서 시험공부를

하듯이, 눈 앞의 작은 손해보다 먼 미래까지의 보상의 총 합이 최대가 되는 것이 중요합니다. 따라서 agent는 어떠한 상황에서 어떻게 행동을 해야겠다라는 기준이 있어야 합니다.

사람도 상황에 따라서 즉흥적으로 임의의 행동을 선택하기보단 자신의 머릿속에 훈련되어 있는대로 행동하기 마련입니다. 무릎에 작은 충격이 왔을 때 다리를 들어올리는 '무릎반사'와 같은 무의식적인 행동에서부터, 파란불이 들어왔을 때 길을 건너는 행동, 어려운 수학 문제가 주어지면 답을 얻는 과정까지, 모두 주어진 상황에 대해서 행동해야 하는 기준이 있습니다.

정책(policy)은 이렇게 agent가 상황에 따라서 어떻게 행동을 해야 할 지 확률적으로 나타낸 기준 입니다. 같은 상황이 주어졌을 때 항상 같은 행동만 반복하는 것이 아니라 확률적으로 행동을 선택한다고 할 수 있습니다. 물론 확률을 1로 표현하면 같은 행동만 반복하게 될 겁니다.

정책은 상황에 따른 가능한 행동에 대한 확률의 맵핑(mapping) 함수라고 할 수 있습니다. 수식으로는 아래와 같이 표현 합니다.

$$\pi(a|s) = P(A_t = a|S_t = s)$$

따라서 우리는 마음속의 정책에 따라 비가 오는 상황(state)에서 자장면과 짬뽕 중에 어떤 음식을 먹을지 확률적으로 선택 할 수 있고, 맑은 날에도 다른 확률 분포 중에서 선택 할 수 있습니다.

Value Function

가치함수(value function)는 주어진 정책(policy) π 아래에서 상황(state) s에서부터 앞으로 얻을 수 있는 보상(reward) 총 합의 기대값을 표현합니다. $v_\pi(s)$라고 표기하며, 아래와 같이 수식으로 표현 될 수 있습니다.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \forall s \in \mathcal{S}$$

앞으로 얻을 수 있는 보상의 총 합의 기대값은 기대누적보상(expected cumulative reward)라고 표현하기도 합니다.

Figure 5:

Action-Value Function (Q-Function)

행동가치함수(action-value function)은 큐함수(Q-function)라고 불리기도 하며, 주어진 정책 π 아래 상황(state) s에서 행동(action) a를 선택 하였을 때 앞으로 얻을 수 있는 보상(reward)의 총 합의 기대값(expected cumulative reward, 기대누적보상)을 표현합니다. 가치함수는 어떤 상황 s에서 어떤 행동을 선택하는 것에 관계 없이 얻을 수 있는 누적 보상의 기대값이라고 한다면, 행동가치함수는 어떤 행동을 선택하는가에 대한 개념이 추가 된 것 입니다.

상황과 행동에 따른 기대누적보상을 나타내는 행동가치함수의 수식은 아래와 같습니다.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t=s,A_t=a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t=s,A_t=a\right]$$

Q-learning

우리는 올바른 행동가치함수를 알고 있다면, 어떠한 상황이 닥쳐도 항상 기대누적보상을 최대화(maximize)하는 행복한 선택을 할 수 있을 것 입니다.

따라서 행동가치함수를 잘 학습하는 것을 Q-learning(큐러닝)이라고 합니다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[\overbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}^{\text{Target}} - \overbrace{Q(S_t, A_t)}^{\text{Current}}\Big]$$

위 수식처럼 target과 current 사이의 차이를 줄이면, 결국 올바른 큐함수를 학습하게 될 것 입니다.

Deep Q-learning (DQN)

큐함수를 배울 때 상황(state) 공간과 행동(action) 공간이 너무 커서 상황과 행동이 희소한(sparse) 경우에는 문제가 생깁니다. 훈련 과정에서 희소성(sparseness)으로 인해 잘 볼 수 없기 때문 입니다. 따라서 우리는 상황과 행동을 discrete한 별개의 값으로 다루되, 큐함수를 근사(approximation)함으로써 문제를 해결할 수 있습니다.

생각 해 보면, 아까 비가 올 때 자장면과 짬뽕을 선택하는 문제도, 비가 5mm가 오는것과 10mm가 오는 것은 비슷한 상황이며, 100mm 오는 것과는 상대적으로 다른 상황이라고 할 수 있습니다. 하지만 해가 짱짱한 맑은 날에 비해서 비가 5mm 오는 거소가 100mm 오는 것은 비슷한 상황이라고도 할 수 있습니다.

이처럼 상황과 행동을 근사하여 문제를 해결한다고 할 때, 신경망(neural network)은 매우 훌륭한 해결 방법이 될 수 있습니다. 딥마인드(DeepMind)는 신경망을 사용하여 근사한 큐러닝을 통해 아타리(Atari) 게임을 훌륭하게 플레이하는 강화학습 방법을 제시하였고, 이를 deep Q-learning (or DQN)이라고 이름 붙였습니다.

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

위의 수식처럼 큐함수 부분을 신경망을 통해 근사(approximate)함으로써 희소성(sparseness)문제를 해결하였습니다.

Policy Gradients

Policy Gradients는 정책기반 강화학습(Policy based Reinforcement Learning) 방식에 속합니다. 알파고를 개발했던 DeepMind에 의해서 유명해진 Deep Q-

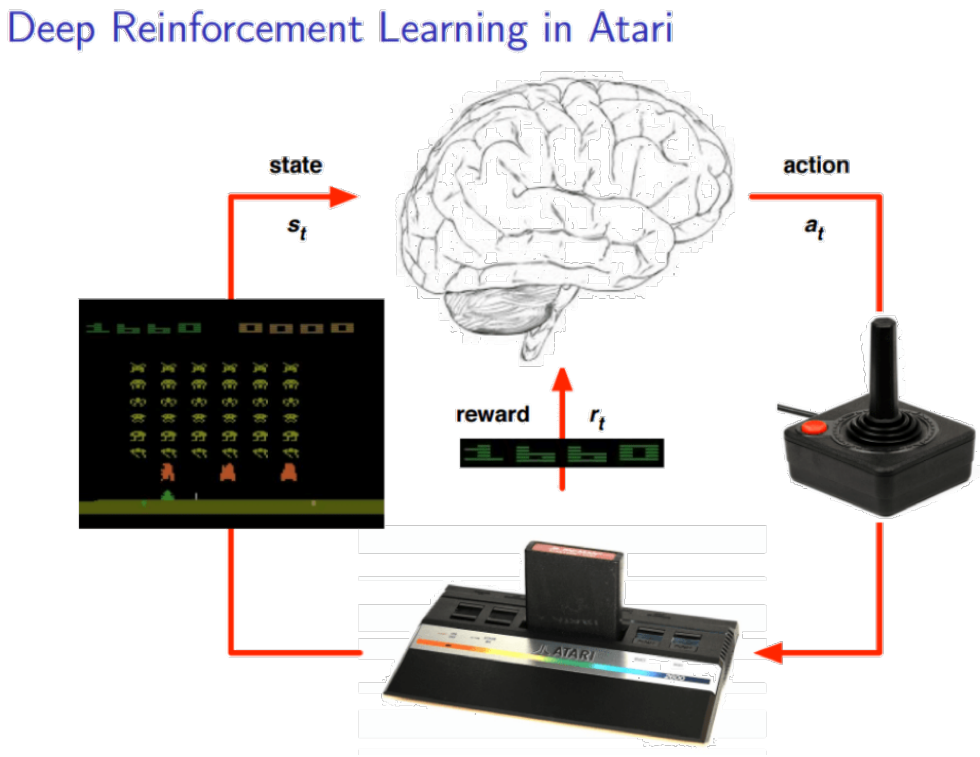

Deep Reinforcement Learning in Atari
state
s_t
action
a_t
reward
r_t

Figure 6:

Learning은 가치기반 강화학습(Value based Reinforcement Learning) 방식에 속합니다. 실제 딥러닝을 사용하여 두 방식을 사용 할 때에 가장 큰 차이점은, Value based방식은 인공신경망을 사용하여 어떤 action을 하였을 때에 얻을 수 있는 보상을 예측 하도록 훈련하는 것과 달리, policy based 방식은 인공신경망은 어떤 action을 할지 훈련되고 해당 action에 대한 보상(reward)를 back-propagation 할 때에 gradient를 통해서 전달해 주는 것이 가장 큰 차이점 입니다. 따라서 어떤 Deep Q-learning의 경우에는 action을 선택하는 것이 deterministic한 것에 비해서, Policy Gradient 방식은 action을 선택 할 때에 stochastic한 process를 거치게 됩니다. Policy Gradient에 대한 수식은 아래와 같습니다.

$$\pi_\theta(a|s) = P_\theta(a|s) = P(a|s;\theta)$$

위의 π는 정책(policy)을 의미합니다. 즉, 신경망 θ는 현재 상황(state) s가 주어졌을 때, 어떤 선택(action) a를 해야할 지 확률을 반환 합니다.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] = v_\theta(s_0) \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\mathcal{R}_{s,a} \end{aligned}$$

우리의 목표(objective)는 최초 상황(initial state)에서의 기대누적보상(expected cumulative reward)을 최대(maximize)로 하도록 하는 policy(θ)를 찾는 것 입니다. 최소화 하여야 하는 손실(loss)와 달리 보상(reward)는 최대화 하여야 하므로 기존의 gradient descent 대신에 gradient ascent를 사용하여 최적화(optimization)을 수행 합니다.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta)$$

Gradient ascent에 따라서, $\nabla_\theta J(\theta)$를 구하여 θ를 업데이트 해야 합니다. 여기서 $d(s)$는 markov chain의 stationary distribution으로써 시작점에 상관없이 전체의 trajecotry에서 s에 머무르는 시간의 proportion을 의미합니다.

$$\begin{aligned} \nabla_\theta \pi_\theta(s,a) &= \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a) \end{aligned}$$

이때, 위의 로그 미분의 성질을 이용하여 아래와 같이 $\nabla_\theta J(\theta)$를 구할 수 있습니다. 이 수식을 해석하면 매 time-step 별 상황(s)이 주어졌을 때 선택(a)할 로그 확률의 gradient에, 그에 따른 보상(reward)을 곱한 값의 기대값이 됩니다.

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(s,a)\mathcal{R}_{s,a} \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\nabla_\theta \log \pi_\theta(s,a)\mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) r]\end{aligned}$$

_Policy Gradient Theorem_에 따르면, 여기서 해당 time-step에 대한 즉각적인 reward(r) 대신에 episode의 종료까지의 총 reward, 즉 Q function을 사용할 수 있습니다.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)]$$

여기서 바로 Policy Gradients의 진가가 드러납니다. 우리는 policy network에 대해서 gradient를 구하지만, Q-function에 대해서는 gradient를 구할 필요가 없습니다. 즉, 미분의 가능 여부를 떠나서 임의의 어떠한 함수라도 보상 함수(reward function)로 사용할 수 있는 것입니다. 이렇게 어떠한 함수도 reward로 사용할 수 있게 됨에 따라, 기존의 단순히 cross entropy와 같은 손실 함수(loss function)에 학습(fitting) 시키는 대신에 좀 더 실제 문제에 부합하는 함수(번역의 경우에는 BLEU)를 사용하여 θ를 훈련시킬 수 있게 되었습니다. 위의 수식에서 기대값 수식을 Monte Carlo sampling으로 대체하면 아래와 같이 parameter update를 수행 할 수 있습니다.

$$\theta \leftarrow \theta + \alpha Q^{\pi_\theta}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

위의 수식을 좀 더 쉽게 설명 해 보면, Monte Carlo 방식을 통해 sampling 된 action들에 대해서 gradient를 구하고, 그 gradient에 reward를 곱하여 주는 형태입니다. 만약 샘플링 된 해당 action들이 좋은 (큰 양수) reward를 받았다면 learning rate α에 추가적인 곱셈을 통해서 더 큰 step으로 gradient ascending을 할 수 있을 겁니다. 하지만 negative reward를 받게 된다면, gradient의 반대방향으로 step을 갖도록 값이 곱해지게 될 겁니다. 따라서 해당 샘플링 된 action들이 앞으로는 잘 나오지 않도록 parameter θ가 update 됩니다.

따라서 실제 gradient에 따른 local minima(지역최소점)를 찾는 것이 아닌, 아래 그림과 같이 실제 reward-objective function에 따른 최적을 값을 찾게 됩니다.

하지만, 기존의 gradient는 방향과 크기를 나타낼 수 있었던 것에 비해서, policy gradients는 기존의 gradient의 방향에 크기(scalar)값을 곱해줌으로써 방향을 직접 지정해 줄 수는 없습니다. 따라서 실제 목적함수(objective function)에 따른 최적의 방향을 스스로 찾아갈 수는 없습니다. 그러므로 사실 훈련이 어렵고 비효율적인 단점을 갖고 있습니다.

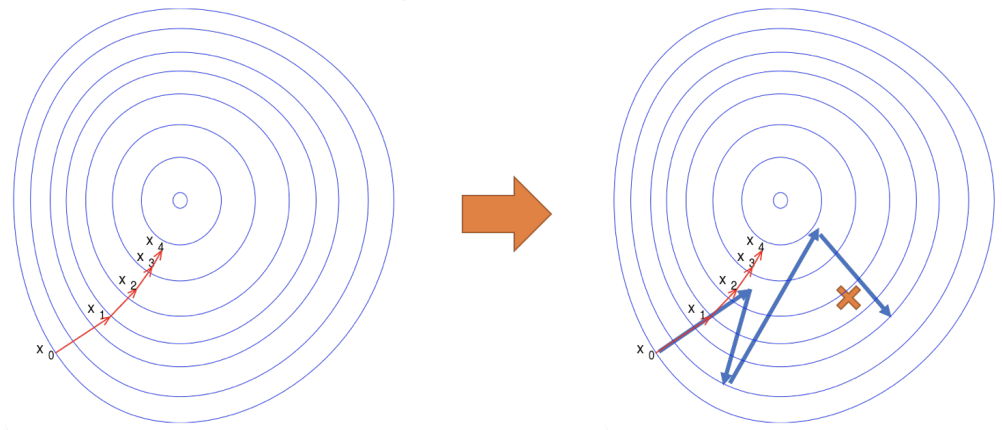


Figure 7:

Policy Gradient에 대한 자세한 설명은 원 논문인 [Sutton at el.1999], 또는 해당 저자가 쓴 텍스트북 Reinforcement Learning: An Introduction[Sutton et al.2017]을 참고하거나, DeepMind David Silver의 YouTube 강의를 참고하면 좋습니다.

MLE vs RL(Policy Gradients)

여기서 reward의 역할을 좀 더 직관적으로 설명하고 넘어가도록 하겠습니다.

우리에게 n개의 시퀀스로 이루어진 입력을 받아, m개의 시퀀스로 이루어진 출력을 하는 함수를 근사하는 것이 목표로 주어진다고 가정 해 봅니다. 그렇다면 시퀀스 X와 Y는 $\mathcal{B}$라는 dataset에 존재 합니다.

$$\mathcal{B} = \{(X_i, Y_i)\}_{i=1}^{N}$$
$$X = \{x_1, x_2, \cdots, x_n\}$$
$$Y = \{y_0, y_1, \cdots, y_m\}$$

근사하여 얻은 함수는 임의의 시퀀스 X가 주어졌을 때, $\hat{Y}$를 반환하도록 잘 학습되어 있을 겁니다.

$$\hat{Y} = argmax_Y P(Y|X)$$

그럼 해당 함수를 근사하기 위해서 우리는 parameter θ를 학습해야 합니다. θ는 아래와 같이 Maximum Likelihood Estimation(MLE)를 통해서 얻어질 수 있습니다.

$$\begin{aligned}\hat{\theta} &= argmax_\theta P(\theta|X, Y) \\ &= argmax_\theta P(Y|X;\theta)P(\theta)\end{aligned}$$

θ에 대해 MLE를 수행하기 위해서 우리는 목적함수(objective function)을 아래와 같이 정의 합니다. 아래는 cross entropy loss를 목적함수로 정의 한 것 입니다. 우리의 목표는 목적함수를 최소화(minimize)하는 것 입니다.

$$\begin{aligned}J(\theta) &= -\frac{1}{N}\sum_{(X,Y)\in\mathcal{B}} P(Y|X)\log P(Y|X;\theta) \\ &= -\frac{1}{N}\sum_{(X,Y)\in\mathcal{B}}\sum_{i=0}^{m}\log P(y_i|X, y_{<i};\theta)\end{aligned}$$

위의 수식에서 $P(Y|X)$는 훈련 데이터셋에 존재하므로 보통 1이라고 할 수 있습니다. 따라서 수식에서 생략 할 수 있습니다. 위에서 정의한 목적함수를 최소화 하여야 하기 때문에, gradient descent를 통해 지역최소점(local minima)를 찾아내어 전역최소점(global minima)에 근사(approximation)할 수 있습니다. 해당 수식은 아래와 같습니다.

$$\begin{aligned}&\theta \leftarrow \theta - \gamma\nabla J(\theta) \\ &\theta \leftarrow \theta + \gamma\frac{1}{N}\sum_{(X,Y)\in\mathcal{B}}\sum_{i=0}^{m}\nabla_\theta \log P(y_i|X, y_i;\theta)\end{aligned}$$

우리는 위의 수식에서 learning rate γ를 통해 update step size를 조절 하는 것을 확인할 수 있습니다. 아래는 policy gradients에 기반하여 expected cumulative reward를 최대로 하는 gradient ascent 수식 입니다. Reward를 최대화(maximization)해야 하기 때문에 gradient acsent를 사용하는 것을 볼 수 있습니다.

$$\theta \leftarrow \theta + \alpha \nabla J(\theta)$$
$$\theta \leftarrow \theta + \alpha Q^{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

위의 수식에서도 이전 MLE의 gradient descent 수식과 마찬가지로, α와 $Q^{\pi_\theta}(s_t, a_t)$가 gradient 앞에 붙어서 learning rate역할을 하는 것을 볼 수 있습니다. 따라서 reward에 따라서 해당 action들로부터 배우는 것을 더욱 강화하거나 반대방향으로 부정할 수 있는 것 입니다. 마치 좀 더 쉽게 비약적으로 설명하면 결과에 따라서 동적으로 learning rate를 알맞게 조절해 주는 것이라고 이해할 수 있습니다.

REINFORCE with baseline

만약 위의 policy gradient를 수행 할 때, 보상이 항상 양수인 경우는 어떻게 동작할까요?

예를 들어 우리가 학교에서 100점 만점의 시험을 보았다고 가정해 보겠습니다. 시험 점수는 0점에서부터 100점까지 분포가 되어 평균 점수 근처에 있을 것입니다. 따라서 대부분의 학생들은 양의 보상을 받게 됩니다. 그럼 위의 기존 policy gradient는 항상 양의 보상을 받아 학생에게 박수쳐주며 해당 정책(policy)를 더욱 독려 할 것 입니다. 하지만 알고보면 평균점수 50점 일 때, 시험점수 10점은 매우 나쁜 점수라고 할 수 있습니다. 따라서 받수 받기보단 혼나서, 기존 정책(policy)의 반대방향으로 학습해야 합니다. 하지만 평균점수 50점일 때 시험점수 70점은 여전히 좋은 점수이고 박수 받아 마땅 합니다. 마찬가지로 평균 50점일 때 시험점수 90점은 70점보다 더 훌륭한 점수이고 박수갈채를 받아야 합니다.

주어진 상황에서 받아 마땅한 누적보상이 있기 때문에, 우리는 이를 바탕으로 현재 정책이 얼마나 훌륭한지 평가 할 수 있습니다. 이를 아래와 같이 policy gradient 수식으로 표현할 수 있습니다.

$$\theta \leftarrow \theta + \alpha \Big(G_t - b(S_t) \Big) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Reinforcement Learning on Natural Language Generation

How to Apply

강화학습은 Markov Decision Process (MDP)상에서 정의되고 동작합니다. 따라서 여러 선택(action)들을 통해서 여러 상황(state)들을 옮겨다니며(transition) 에피소드가 구성되고, action과 state에 따라서 보상(reward)가 주어지며 누적(cumulated)되어, episode가 종료되면 누적보상(cumulative reward)를 얻을 수 있습니다.

따라서 이를 자연어처리에 적용하게 되면 텍스트 분류(text classification)와 같은 문제에 적용되기 보단 자연어생성(natural language generation)에 적용되게 됩니다. 이제까지 생성된 단어들의 시퀀스가 현재의 상황(state)이 될 것이며, 이제까지 생성된 단어를 기반으로 새롭게 선택하는 단어가 action이 될 것 입니다. 이렇게 문장의 첫 단어(BOS, beginning of sentence)부터 문장의 끝 단어(EOS, end of sentence)까지 선택하는 과정(한 문장을 생성하는 과정)이 하나의 에피소드(episode)가 됩니다. 우리는 훈련 corpus에 대해서 문장을 생성하는 방법을 배워(즉, episode를 반복하여) 기대누적보상(expected cumulative reward)을 최대화(maximize)할 수 있도록 번역 신경망(강화학습에서는 정책망) θ를 훈련 하는 것 입니다.

다시한번 정리를 위해, 기계번역에 강화학습을 대입시켜 보면, 현재의 상황(state)은 주어진 source 문장과 이전까지 생성(번역)된 단어들의 시퀀스가 될 것이고, action은 현재 state에 기반하여 새로운 단어를 선택 하는 것이 될 것 입니다. 그리고 action을 선택하게 되면 다음 state는 deterministic하게 정해지게 됩니다. action을 선택한 후에 환경(environment)으로부터 즉각적인(immediate) 보상(reward)를 받지는 않으며, 모든 단어의 선택이 끝나고(EOS를 선택) 에피소드(episode)가 종료되면 BLEU 점수를 통해 보상을 받을 수 있습니다. 즉, 종료 시에 받는 보상값은 에피소드 누적보상값(cumulative reward)과 일치 합니다.

강화학습을 통해 모델을 훈련할 때, 훈련의 처음부터 강화학습만 적용하기에는 그 훈련방식이 비효율적이고 어려움이 크기 때문에, 보통 기존의 maximum likelihood estimation (MLE)를 통해 어느정도 학습이 된 신경망 θ에 강화학습을 적용 합니다. – 강화학습은 탐험(exploration)을 통해 더 나은 정책의 가능성을 찾고, exploitation을 통해 그 정책을 발전시켜 나갑니다.

Characteristics

우리는 이제까지 강화학습 중에서도 정책기반 학습 방식인 policy gradients에 대해서 간단히 다루어 보았습니다. 사실, policy gradients의 경우에도 소개한 방법 이외에도 발전된 방법들이 많이 있습니다. 예를 들어 Actor Critic의 경우에는 정책망(θ) 이외에도 가치 네트워크(W)를 따로 두어, episode의 종료까지 기다리지 않고 online으로 학습이 가능합니다. 여기에서 더욱 발전하여 기존의 단점을 보완한 A3C와 같은 다양한 방법들이 존재 합니다. - Reinforcement Learning: An Introduction[Sutton et al.2017]을 참고하세요.

하지만, 자연어처리에서의 강화학습은 이런 다양한 방법들을 굳이 사용하기보다는 간단한 REINFORCE with baseline를 사용하더라도 큰 문제가 없습니다. 이것은 자연어처리 분야의 특성에서 기인합니다. 강화학습을 자연어처리에 적용할 때는 아래와 같은 특성들이 있습니다.

1. 매우 많은 action들이 존재 합니다. 보통 다음 단어를 선택하는 것이 action이 되기 때문에, 선택 가능한 action set의 크기는 어휘(vocabularty) 사전의 크기와 같다고 볼 수 있습니다. 따라서 action set의 사이즈는 보통 몇만개가 되기 마련입니다.
2. 매우 많은 state들이 존재 합니다. 단어를 선택하는 것이 action이었다면, 이제까지 선택된 단어들의 시퀀스는 state가 되기 때문에, 여러 time-step을 거쳐 수많은 action(단어)들이 선택되었다면, 가능한 state의 경우의 수는 매우 커질 것 입니다. - $|state| = |action|^n$
3. 매우 많은 action들과 매우 많은 state들을 훈련 과정에서 모두 겪어보고 만나보는 것은 거의 불가능하다고 볼 수 있습니다. 또한, 추론(inference)과정에서 보지 못한(unseen) 샘플을 만나는 것은 매우 당연한 일이 될 것 입니다. 따라서 이러한 희소성(sparseness)문제는 큰 골칫거리가 될 수 있습니다. 하지만 우리는 신경망(neural network)과 딥러닝을 통해서 이 문제를 해결 할 수 있습니다. - 이것은 Deep Reinforcement Learning이 큰 성공을 거두고 있는 이유이기도 합니다.
4. 강화학습을 자연어처리에 적용할 때에 쉬운 점도 있습니다. 대부분 하나의 문장을 생성하는 것이 하나의 에피소드(episode)가 되는데, 보통 문장의 길이는 길어봤자 80단어도 되기 힘들다는 것 입니다. 따라서 다른 분야의 강화학습보다 훨씬 쉬운 이점을 가지게 됩니다. 예를 들어 딥마인드(DeepMind)의 바둑(AlphaGo)이나 스타크래프트의 경우에는 하나의 에피소드가 끝나기까지 매우 긴 시간이 흐르게 됩니다. 따라서 에피소드 내에서 선택되었던 action들이 update되기 위해서는 매우 긴 에피소드가 끝나기를 기다려야 할 뿐만 아니라, 10분 전에 선택 했던 action이

이 게임(game)의 승패에 얼마나 큰 영향을 미쳤는지 알아내는 것은 매우 어려운 일이 될 것 입니다. 따라서, 자연어처리 분야가 다른 분야에 비해서 에피소드가 짧은 것은 매우 큰 이점으로 작용하여 정책망(policy network)을 훨씬 더 쉽게 훈련(training)시킬 수 있게 됩니다.

5. 대신에 문장 단위의 에피소드를 가진 강화학습에서는 보통 에피소드 중간에 reward를 얻기는 힘듭니다. 예를 들어 번역의 경우에는 각 time-step 마다 단어를 선택할 때 즉각(immediate) reward를 얻지 못하고, 번역이 모두 끝난 이후 완성된 문장과 정답(reference) 문장을 비교하여 BLEU 점수를 누적 reward로 사용하게 됩니다. 마찬가지로 에피소드가 매우 길다면 이것은 매우 큰 문제가 되었겠지만, 다행히도 문장 단위의 에피소드 상에서는 큰 문제가 되지 않습니다.

이제 우리는 위의 특성들을 활용하여 강화학습을 성공적으로 기계번역에 적용한 방법들과 사례들을 살펴보고 실습 해 보도록 하겠습니다. # Supervised NMT

Cross-entropy vs BLEU

$$L = -\frac{1}{|Y|}\sum_{y \in Y} P(y) \log P_\theta(y)$$

Cross entropy는 훌륭한 분류(classification) 문제에서 이미 훌륭한 손실함수(loss function)이지만 약간의 문제점을 가지고 있습니다. 자연어생성(NLG)을 위한 sequence-to-sequence의 훈련 과정에 적용하게 되면, 그 자체의 특성으로 인해서 우리가 평가하는 BLEU와의 괴리(discrepancy)가 생기게 됩니다. (자세한 내용은 이전 챕터 내용 참조 바랍니다.) 따라서 어찌보면 우리가 원하는 실제 기계번역의 목표(objective)와 다름으로 인해서 cross-entropy 자체에 오버피팅(over-fitting)되는 효과가 생길 수 있습니다. 일반적으로 BLEU는 human evluation과 좋은 상관관계에 있다고 알려져 있지만, cross entropy는 이에 비해 낮은 상관관계를 가지기 때문입니다. 따라서 차라리 BLEU를 훈련 과정의 목적함수(objective function)로 사용하게 된다면 더 좋은 결과를 얻을 수 있을 것 입니다. 마찬가지로 다른 NLG 문제(요약 및 챗봇 등)에 대해서도 비슷한 접근을 생각 할 수 있습니다.

Minimum Risk Training

위의 아이디어에서 출발한 논문[Shen at el.2015]이 Minimum Risk Training이라는 방법을 제안하였습니다. 이때에는 Policy Gradient를 직접적으로 사용하진 않았지만,

거의 비슷한 수식이 유도 되었다는 점이 매우 인상적입니다.

$$\hat{\theta}_{MLE} = argmin_{\theta}(\mathcal{L}(\theta))$$
$$where\ \mathcal{L}(\theta) = -\sum_{s=1}^{S} \log P(y^{(s)}|x^{(s)};\theta)$$

기존의 Maximum Likelihood Estimation (MLE)방식은 위와 같은 손실 함수(Loss function)를 사용하여 $|S|$개의 입력과 출력에 대해서 손실(loss)값을 구하고, 이를 최소화 하는 θ를 찾는 것이 목표(objective)였습니다. 하지만 이 논문에서는 Risk를 아래와 같이 정의하고, 이를 최소화 하는 학습 방식을 Minimum Risk Training (MRT)라고 하였습니다.

$$\begin{aligned}\mathcal{R}(\theta) &= \sum_{s=1}^{S} E_{y|x^{(s)};\theta}[\triangle(y, y^{(s)})] \\ &= \sum_{s=1}^{S} \sum_{y \in \mathcal{Y}(\S^{(J)})} P(y|x^{(s)};\theta)\triangle(y, y^{(s)})\end{aligned}$$

위의 수식에서 $\mathcal{Y}(x^{(s)})$는 full search space로써, s번째 입력 $x^{(s)}$가 주어졌을 때, 가능한 정답의 집합을 의미합니다. 또한 $\triangle(y, y^{(s)})$는 입력과 파라미터(θ)가 주어졌을 때, sampling한 y와 실제 정답 $y^{(s)}$의 차이(error)값을 나타냅니다. 즉, 위 수식에 따르면 risk $\mathcal{R}$은 주어진 입력과 현재 파라미터 상에서 얻은 y를 통해 현재 모델(함수)을 구하고, 동시에 이를 사용하여 risk의 기대값을 구한다고 볼 수 있습니다.

$$\hat{\theta}_{MRT} = argmin_{\theta}(\mathcal{R}(\theta))$$

이렇게 정의된 risk를 최소화(minimize) 하도록 하는 것이 목표(objective)입니다. 사실 risk 대신에 reward로 생각하면, reward를 최대화(maximize) 하는 것이 목표가 됩니다. 결국은 risk를 최소화 할 때에는 gradient descent, reward를 최대화 할 때는 gradient ascent를 사용하게 되므로, 수식을 풀어보면 결국 완벽하게 같은 이야기라고 볼 수 있습니다. 따라서 실제 구현에 있어서는 $\triangle(y, y^{(s)})$ 사용을 위해서 BLEU 점수에 -1을 곱하여 사용 합니다.

$$\tilde{\mathcal{R}}(\theta) = \sum_{s=1}^{S} E_{y|x^{(s)};\theta,\alpha}[\triangle(y, y^{(s)})]$$
$$= \sum_{s=1}^{S} \sum_{y \in \mathcal{S}(x^{(s)})} Q(y|x^{(s)};\theta,\alpha)\triangle(y, y^{(s)})$$

where $\mathcal{S}(x^{(s)})$ *is a sampled subset of the full search space* $\dagger(x^{(s)})$
and $Q(y|x^{(s)};\theta,\alpha)$ *is a distribution defined on the subspace* $S(x^{(s)})$:

$$Q(y|x^{(s)};\theta,\alpha) = \frac{P(y|x^{(s)};\theta)^{\alpha}}{\sum_{y' \in S(x^{(s)})} P(y'|x^{(s)};\theta)^{\alpha}}$$

하지만 주어진 입력에 대한 가능한 정답에 대한 전체 space를 탐색(search)할 수는 없기 때문에, Monte Carlo를 사용하여 서브스페이스(sub-space)를 샘플링(sampling) 하는 것을 택합니다. 그리고 위의 수식에서 θ에 대해서 미분을 수행합니다. 미분을 하여 얻은 수식은 아래와 같습니다.

$$\frac{\partial \tilde{R}(\theta)}{\partial \theta_i} = \alpha \sum_{s=1}^{S} \mathbb{E}_{y|x^{(s)};\theta,\alpha}[\frac{\partial P(y|x^{(s)};\theta)}{\partial \theta_i P(y|x^{(s)};\theta)} \times (\triangle(y, y^{(s)}) - \mathbb{E}_{y'|x^{(s)};\theta,\alpha}[\triangle(y', y^{(s)})])]$$
$$= \alpha \sum_{s=1}^{S} \mathbb{E}_{y|x^{(s)};\theta,\alpha}[\frac{\partial \log P(y|x^{(s)};\theta)}{\partial \theta_i} \times (\triangle(y, y^{(s)}) - \mathbb{E}_{y'|x^{(s)};\theta,\alpha}[\triangle(y', y^{(s)})])]$$
$$\approx \alpha \sum_{s=1}^{S} \frac{\partial \log P(y|x^{(s)};\theta)}{\partial \theta_i} \times (\triangle(y, y^{(s)}) - \frac{1}{K}\sum_{k=1}^{K} \triangle(y^{(k)}, y^{(s)}))$$

$$\theta_{i+1} \leftarrow \theta_i - \frac{\partial \tilde{R}(\theta)}{\partial \theta_i}$$

이제 미분을 통해 얻은 MRT의 최종 수식을 해석 해 보겠습니다. 이해가 어렵다면 아래의 policy gradients 수식과 비교하며 따라가면 좀 더 이해가 수월할 수 있습니다.

- s번째 입력 $x^{(s)}$를 신경망 θ에 넣어 얻은 로그확률 $\log P(y|x^{(s)};\theta)$을 미분하여 gradient를 얻습니다.
- 그리고 θ로부터 샘플링(samping) 한 y와 실제 정답 $y^{(s)}$와의 차이(여기서는 주로 BLEU에 -1을 곱하여 사용)값에서

- 또 다시 θ로부터 샘플링하여 얻은 y'와 실제 정답 $y^{(s)}$와의 차이(마찬가지로 -BLEU)의 기대값을
- 빼 준 값을 risk로써 로그확률의 gradient에 곱해 줍니다.
- 이 과정을 전체 데이터셋(실제로는 mini-batch) S에 대해서 수행한 후 합(summation)을 구하고 learning rate α를 곱 합니다.

최종적으로는 기대값 수식을 monte carlo sampling을 통해 제거할 수 있습니다.

아래는 policy gradients 수식 입니다.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \times Q^{\pi_\theta}(s,a)]$$
$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

MRT는 risk에 대해 minimize 해야 하기 때문에 gradient descent를 해 주는 것을 제외하면 똑같은 수식이 나오는 것을 알 수 있습니다.

System	Architecture	Training	Vocab	BLEU
Existing end-to-end NMT systems				
Bahdanau et al. (2015)	gated RNN with search	MLE	30K	28.45
Jean et al. (2015)	gated RNN with search		30K	29.97
Jean et al. (2015)	gated RNN with search + PosUnk		30K	33.08
Luong et al. (2015b)	LSTM with 4 layers		40K	29.50
Luong et al. (2015b)	LSTM with 4 layers + PosUnk		40K	31.80
Luong et al. (2015b)	LSTM with 6 layers		40K	30.40
Luong et al. (2015b)	LSTM with 6 layers + PosUnk		40K	32.70
Sutskever et al. (2014)	LSTM with 4 layers		80K	30.59
Our end-to-end NMT systems				
this work	gated RNN with search	MLE	30K	29.88
	gated RNN with search	MRT	30K	31.30
	gated RNN with search + PosUnk	MRT	30K	34.23

Table 7: Comparison with previous work on English-French translation. The BLEU scores are case-sensitive. "PosUnk" denotes Luong et al. (2015b)'s technique of handling rare words.

Figure 8:

위와 같이 훈련한 MRT에 대한 성능을 실험한 결과 입니다. 기존의 MLE 방식에 비해서 BLEU가 1.5가량 상승한 것을 확인할 수 있습니다. 이처럼 MRT는 강화학습으로써의 접근을 전혀 하지 않고도, 수식적으로 policy gradients의 일종인 REINFORCE with baseline 수식을 이끌어내고 성능을 끌어올리는 방법을 제시한 점이 인상깊습니다.

Implementation

우리는 아래의 방법을 통해 Minimum Risk Training을 PyTorch로 구현 할 겁니다.

1. 먼저 BLEU를 통해 얻은 reward에 -1을 곱해주어 risk로 변환 합니다.
2. 그리고 로그 확률에 risk를 곱해주고, 기존에 Negative Log Likelihodd Loss (NLLLoss)를 사용했으므로 NLLLoss 값에 -1을 곱해주어 sum of positive log probability를 구합니다.
3. Summation 결과물에 대해서 θ에 대해 미분을 수행하면, back-propagation을 통해서 신경망 θ 전체에 gradient가 구해집니다.
4. 이 gradient를 사용하여 gradient descent를 통해 최적화(optimize) 하도록 할 겁니다.

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s=1}^{S} \left(\log P(y|x^{(s)};\theta) \times \left(\triangle(y, y^{(s)}) - \frac{1}{K} \sum_{k=1}^{K} \triangle(y^{(k)}, y^{(s)}) \right) \right)$$

$$where\ \triangle(\hat{y}, y) = -BLEU(\hat{y}, y)$$

$$\theta \leftarrow \theta - \lambda \nabla_\theta J(\theta)$$

우리는 실험을 통해서 심지어 $K = 1$일 때도, MRT가 잘 동작함을 확인할 수 있습니다.

Code

MRT(or RL)을 PyTorch를 사용하여 구현 해 보도록 하겠습니다. 자세한 전체 코드는 이전의 NMT PyTorch 실습 코드의 git repository에서 다운로드 할 수 있습니다.

- git repo url: https://github.com/kh-kim/simple-nmt

train.py

```
import argparse, sys

import torch
```

```
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer
import simple_nmt.rl_trainer as rl_trainer

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to save. Ado
    p.add_argument('-train', required = True, help = 'Training set file name excep
    p.add_argument('-valid', required = True, help = 'Validation set file name exc
    p.add_argument('-lang', required = True, help = 'Set of extention represents l
    p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to train. C

    p.add_argument('-batch_size', type = int, default = 32, help = 'Mini batch siz
    p.add_argument('-n_epochs', type = int, default = 18, help = 'Number of epochs
    p.add_argument('-print_every', type = int, default = 1000, help = 'Number of g
    p.add_argument('-early_stop', type = int, default = -1, help = 'The training w

    p.add_argument('-max_length', type = int, default = 80, help = 'Maximum length
    p.add_argument('-dropout', type = float, default = .2, help = 'Dropout rate. D
    p.add_argument('-word_vec_dim', type = int, default = 512, help = 'Word embedd
    p.add_argument('-hidden_size', type = int, default = 768, help = 'Hidden size
    p.add_argument('-n_layers', type = int, default = 4, help = 'Number of layers

    p.add_argument('-max_grad_norm', type = float, default = 5., help = 'Threshold
    p.add_argument('-adam', action = 'store_true', help = 'Use Adam instead of usi
    p.add_argument('-lr', type = float, default = 1., help = 'Initial learning rat
    p.add_argument('-min_lr', type = float, default = .000001, help = 'Minimum lea
    p.add_argument('-lr_decay_start_at', type = int, default = 10, help = 'Start l
    p.add_argument('-lr_slow_decay', action = 'store_true', help = 'Decay learning
    p.add_argument('-lr_decay_rate', type = float, default = .5, help = 'Learning

    p.add_argument('-rl_lr', type = float, default = .01, help = 'Learning rate fo
    p.add_argument('-n_samples', type = int, default = 1, help = 'Number of sample
```

```
    p.add_argument('-rl_n_epochs', type = int, default = 10, help = 'Number of epo
    p.add_argument('-rl_n_gram', type = int, default = 6, help = 'Maximum number o

    config = p.parse_args()

    return config

if __name__ == "__main__":
    config = define_argparser()

    import os.path
    # If the model exists, load model and configuration to continue the training
    if os.path.isfile(config.model):
        saved_data = torch.load(config.model)

        prev_config = saved_data['config']
        config = overwrite_config(config, prev_config)
        config.lr = saved_data['current_lr']
    else:
        saved_data = None

    # Load training and validation data set.
    loader = DataLoader(config.train,
                        config.valid,
                        (config.lang[:2], config.lang[-2:]),
                        batch_size = config.batch_size,
                        device = config.gpu_id,
                        max_length = config.max_length
                        )

    input_size = len(loader.src.vocab) # Encoder's embedding layer input size
    output_size = len(loader.tgt.vocab) # Decoder's embedding layer input size an
    # Declare the model
    model = Seq2Seq(input_size,
                    config.word_vec_dim, # Word embedding vector size
                    config.hidden_size, # LSTM's hidden vector size
                    output_size,
                    n_layers = config.n_layers, # number of layers in LSTM
```

```
                dropout_p = config.dropout # dropout-rate in LSTM
                )

# Default weight for loss equals to 1, but we don't need to get loss for PAD
# Thus, set a weight for PAD to zero.
loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0.
# Instead of using Cross-Entropy loss, we can use Negative Log-Likelihood(NL
criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

print(model)
print(criterion)

# Pass models to GPU device if it is necessary.
if config.gpu_id >= 0:
    model.cuda(config.gpu_id)
    criterion.cuda(config.gpu_id)

# If we have loaded model weight parameters, use that weights for declared m
if saved_data is not None:
    model.load_state_dict(saved_data['model'])

# Start training. This function maybe equivalant to 'fit' function in Keras.
trainer.train_epoch(model,
                    criterion,
                    loader.train_iter,
                    loader.valid_iter,
                    config,
                    start_epoch = saved_data['epoch'] if saved_data is not No
                    others_to_save = {'src_vocab': loader.src.vocab, 'tgt_voca
                    )

# Start reinforcement learning.
if config.rl_n_epochs > 0:
    rl_trainer.train_epoch(model,
                           criterion, # Although it does not use cross-entropy l
                           loader.train_iter,
                           loader.valid_iter,
```

```
                                config,
                                start_epoch = (saved_data['epoch'] - config.n_epochs)
                                others_to_save = {'src_vocab': loader.src.vocab, 'tgt_
                                )
```

simple_nmt/rl_trainer.py

```
import time
import numpy as np
#from nltk.translate.bleu_score import sentence_bleu as score_func
from nltk.translate.gleu_score import sentence_gleu as score_func

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

import utils
import data_loader

def get_reward(y, y_hat, n_gram = 6):
    # This method gets the reward based on the sampling result and reference sen
    # For now, we uses GLEU in NLTK, but you can used your own well-defined rewa
    # In addition, GLEU is variation of BLEU, and it is more fit to reinforcemen

    # Since we don't calculate reward score exactly as same as multi-bleu.perl,
    # (especialy we do have different tokenization,) I recommend to set n_gram t

    # |y| = (batch_size, length1)
    # |y_hat| = (batch_size, length2)

    scores = []

    # Actually, below is really far from parallized operations.
    # Thus, it may cause slow training.
    for b in range(y.size(0)):
        ref = []
        hyp = []
```

```
            for t in range(y.size(1)):
                ref += [str(int(y[b, t]))]
                if y[b, t] == data_loader.EOS:
                    break

            for t in range(y_hat.size(1)):
                hyp += [str(int(y_hat[b, t]))]
                if y_hat[b, t] == data_loader.EOS:
                    break

            # for nltk.bleu & nltk.gleu
            scores += [score_func([ref], hyp, max_len = n_gram) * 100.]

            # for utils.score_sentence
            #scores += [score_func(ref, hyp, 4, smooth = 1)[-1] * 100.]
        scores = torch.FloatTensor(scores).to(y.device)
        # |scores| = (batch_size)

        return scores

    def get_gradient(y, y_hat, criterion, reward = 1):
        # |y| = (batch_size, length)
        # |y_hat| = (batch_size, length, output_size)
        # |reward| = (batch_size)
        batch_size = y.size(0)

        # Before we get the gradient, multiply -reward for each sample and each time
        y_hat = y_hat * -reward.view(-1, 1, 1).expand(*y_hat.size())

        # Again, multiply -1 because criterion is NLLLoss.
        log_prob = -criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguou
        log_prob.div(batch_size).backward()

        return log_prob

    def train_epoch(model, criterion, train_iter, valid_iter, config, start_epoch = 1
        current_lr = config.rl_lr

        highest_valid_bleu = -np.inf
```

```
no_improve_cnt = 0

# Print initial valid BLEU before we start RL.
model.eval()
total_reward, sample_cnt = 0, 0
for batch_index, batch in enumerate(valid_iter):
    current_batch_word_cnt = torch.sum(batch.tgt[1])
    x = batch.src
    y = batch.tgt[0][:, 1:]
    batch_size = y.size(0)
    # |x| = (batch_size, length)
    # |y| = (batch_size, length)

    # feed-forward
    y_hat, indice = model.search(x, is_greedy = True, max_length = config.max_
    # |y_hat| = (batch_size, length, output_size)
    # |indice| = (batch_size, length)

    reward = get_reward(y, indice, n_gram = config.rl_n_gram)

    total_reward += float(reward.sum())
    sample_cnt += batch_size
    if sample_cnt >= len(valid_iter.dataset.examples):
        break
avg_bleu = total_reward / sample_cnt
print("initial valid BLEU: %.4f" % avg_bleu) # You can figure-out improvement
model.train() # Now, begin training.

# Start RL
for epoch in range(start_epoch, config.rl_n_epochs + 1):
    #optimizer = optim.Adam(model.parameters(), lr = current_lr)
    optimizer = optim.SGD(model.parameters(), lr = current_lr) # Default hype
    print("current learning rate: %f" % current_lr)
    print(optimizer)

    sample_cnt = 0
    total_loss, total_bleu, total_sample_count, total_word_count, total_parame
    start_time = time.time()
```

```
train_bleu = np.inf

for batch_index, batch in enumerate(train_iter):
    optimizer.zero_grad()

    current_batch_word_cnt = torch.sum(batch.tgt[1])
    x = batch.src
    y = batch.tgt[0][:, 1:]
    batch_size = y.size(0)
    # |x| = (batch_size, length)
    # |y| = (batch_size, length)

    # Take sampling process because set False for is_greedy.
    y_hat, indice = model.search(x, is_greedy = False, max_length = config
    # Based on the result of sampling, get reward.
    q_actor = get_reward(y, indice, n_gram = config.rl_n_gram)
    # |y_hat| = (batch_size, length, output_size)
    # |indice| = (batch_size, length)
    # |q_actor| = (batch_size)

    # Take samples as many as n_samples, and get average rewards for the
    # I figured out that n_samples = 1 would be enough.
    baseline = []
    with torch.no_grad():
        for i in range(config.n_samples):
            _, sampled_indice = model.search(x, is_greedy = False, max_len
            baseline += [get_reward(y, sampled_indice, n_gram = config.rl_
        baseline = torch.stack(baseline).sum(dim = 0).div(config.n_samples
        # |baseline| = (n_samples, batch_size) --> (batch_size)

    # Now, we have relatively expected cumulative reward.
    # Which score can be drawn from q_actor subtracted by baseline.
    tmp_reward = q_actor - baseline
    # |tmp_reward| = (batch_size)
    # calcuate gradients with back-propagation
    get_gradient(indice, y_hat, criterion, reward = tmp_reward)

    # simple math to show stats
```

```
total_loss += float(tmp_reward.sum())
total_bleu += float(q_actor.sum())
total_sample_count += batch_size
total_word_count += int(current_batch_word_cnt)
total_parameter_norm += float(utils.get_parameter_norm(model.parameter
total_grad_norm += float(utils.get_grad_norm(model.parameters()))

if (batch_index + 1) % config.print_every == 0:
    avg_loss = total_loss / total_sample_count
    avg_bleu = total_bleu / total_sample_count
    avg_parameter_norm = total_parameter_norm / config.print_every
    avg_grad_norm = total_grad_norm / config.print_every
    elapsed_time = time.time() - start_time

    print("epoch: %d batch: %d/%d\t|param|: %.2f\t|g_param|: %.2f\trwd
```

```
    total_loss, total_bleu, total_sample_count, total_word_count, tota
    start_time = time.time()

    train_bleu = avg_bleu

# In orther to avoid gradient exploding, we apply gradient clipping.
torch_utils.clip_grad_norm_(model.parameters(), config.max_grad_norm)
# Take a step of gradient descent.
optimizer.step()

sample_cnt += batch_size
if sample_cnt >= len(train_iter.dataset.examples):
    break
```

```
sample_cnt = 0
total_reward = 0

# Start validation
with torch.no_grad():
    model.eval() # Turn-off drop-out

    for batch_index, batch in enumerate(valid_iter):
        current_batch_word_cnt = torch.sum(batch.tgt[1])
        x = batch.src
        y = batch.tgt[0][:, 1:]
        batch_size = y.size(0)
        # |x| = (batch_size, length)
        # |y| = (batch_size, length)

        # feed-forward
        y_hat, indice = model.search(x, is_greedy = True, max_length = cor
        # |y_hat| = (batch_size, length, output_size)
        # |indice| = (batch_size, length)

        reward = get_reward(y, indice, n_gram = config.rl_n_gram)

        total_reward += float(reward.sum())
        sample_cnt += batch_size
        if sample_cnt >= len(valid_iter.dataset.examples):
            break

    avg_bleu = total_reward / sample_cnt
    print("valid BLEU: %.4f" % avg_bleu)

    if highest_valid_bleu < avg_bleu:
        highest_valid_bleu = avg_bleu
        no_improve_cnt = 0
    else:
        no_improve_cnt += 1

    model.train()
```

```
model_fn = config.model.split(".")
model_fn = model_fn[:-1] + ["%02d" % (config.n_epochs + epoch), "%.2f-%.4f
# PyTorch provides efficient method for save and load model, which uses
to_save = {"model": model.state_dict(),
           "config": config,
           "epoch": config.n_epochs + epoch + 1,
           "current_lr": current_lr
           }
if others_to_save is not None:
    for k, v in others_to_save.items():
        to_save[k] = v
torch.save(to_save, '.'.join(model_fn))

if config.early_stop > 0 and no_improve_cnt > config.early_stop:
    break
```

Unsupervised NMT

Supervised learning 방식은 높은 정확도를 자랑하지만 labeling 데이터가 필요하기 때문에 데이터 확보, 모델 및 시스템을 구축하는데 높은 비용과 시간이 소요됩니다. 하지만 Unsupervised Learning의 경우에는 데이터 확보에 있어서 훨씬 비용과 시간을 절감할 수 있기 때문에 좋은 대안이 될 수 있습니다.

Parallel corpus vs Monolingual corpus

그러한 의미에서 parallel corpus에 비해서 확보하기 쉬운 monolingual corpus는 좋은 대안이 될 수 있습니다. 소량의 parallel corpus와 다량의 monolingual corpus를 결합하여 더 나은 성능을 확보할 수도 있을 것입니다. 이전 챕터에 다루었던 Back translation과 Copied translation에서 이와 관련하여 NMT의 성능을 고도화하는 방법을 보여주었습니다. 강화학습에서도 마찬가지로 unsupervised 방식을 적용하려는 시도들이 많이 보이고 있습니다. 다만, 대부분의 방식들은 아직 실제 field에서 적용하기에는 다소 효율성이 떨어집니다.

Unsupervised NMT

위의 Dual Learning 논문과 달리 이 논문[Lample at el.2017]은 오직 Monolingual Corpus만을 사용하여 번역기를 제작하는 방법을 제안하였습니다. 따라서 Unsupervised NMT라고 할 수 있습니다.

이 논문의 핵심 아이디어는 아래와 같습니다. 제일 중요한 것은 encoder가 언어에 상관 없이 같은 내용일 경우에 같은 vector로 encoding할 수 있도록 훈련하는 것입니다. 이러한 encoder를 만들기 위해서 GAN이 도입되었습니다.

GAN을 NLP에 쓰지 못한다고 해 놓고 GAN을 썼다니 이게 무슨 소리인가 싶겠지만, encoder의 출력값인 vector에 대해서 GAN을 적용한 것이라 discrete한 값이 아닌 continuous한 값이기 때문에 가능한 것 입니다.

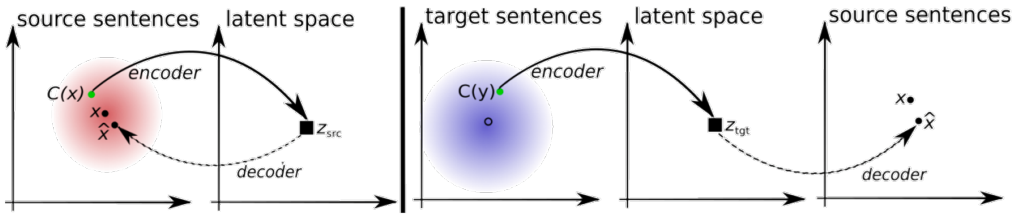


Figure 1: Toy illustration of the principles guiding the design of our objective function. Left (auto-encoding): the model is trained to reconstruct a sentence from a noisy version of it. x is the target, $C(x)$ is the noisy input, $\hat{x}$ is the reconstruction. Right (translation): the model is trained to translate a sentence in the other domain. The input is a noisy translation (in this case, from source-to-target) produced by the model itself, M, at the previous iteration (t), $y = M^{(t)}(x)$. The model is symmetric, and we repeat the same process in the other language. See text for more details.

Figure 9:

이렇게 다른 언어일지라도 동일한 내용에 대해서는 같은 vector로 encoding하도록 훈련 된 encoder의 출력값을 가지고 decoder로 원래의 문장으로 잘 돌아오도록 해 주는 것이 이 논문의 핵심 내용입니다.

특기 할 만한 점은 이 논문에서는 언어에 따라서 encoder와 decoder를 다르게 사용한 것이 아니라 언어에 상관없이 1개씩의 encoder와 decoder를 사용하였습니다. 또한 이 논문[Conneau at el.,2017]에서 제안한 word by word translation 방식으로 pre-training 한 모델을 사용합니다.

이 논문의 훈련은 3가지 관점에서 수행됩니다.

Denoising Autoencoder

이전 챕터에서 다루었듯이 Seq2seq 모델도 결국 Autoencoder의 일종이라고 볼 수 있습니다. 그러한 관점에서 autoencoder(AE)로써 단순 복사(copy) task는 굉장히 쉬운 task에 속합니다. 그러므로 단순히 encoding 한 source sentence를 같은 언어의 문장으로 decoding 하는 것은 매우 쉬운 일이 될 것입니다. 따라서 AE에게 단순히 복사 작업을 지시하는 것이 아닌 noise를 섞어 준 source sentence에서 denoising을 하면서 reconstruction(복원)을 할 수 있도록 훈련해야 합니다. 따라서 이 task의 objective는 아래와 같습니다.

$$\mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell) = \mathbb{E}_{x \sim \mathcal{D}_\ell, \hat{x} \sim d(e(C(x),\ell),\ell)}[\triangle(\hat{x}, x)]$$

$\hat{x} \sim d(e(C(x), \ell), \ell)$는 source sentence x를 C를 통해 noise를 추가하고, 같은 언어 ℓ로 encoding과 decoding을 수행한 것을 의미합니다. $\triangle(\hat{x}, x)$는 원문과 복원된 문장과의 차이(error)를 나타냅니다.

Noise Model

Noise Model $C(x)$는 임의로 문장 내 단어들을 drop하거나, 순서를 섞어주는 일을 합니다. drop rate는 보통 0.1, 순서를 섞어주는 단어사이의 거리는 3정도가 적당한 것으로 설명 합니다.

Cross Domain Training (Translation)

이번엔 이전 iteration의 모델 M에서 언어(ℓ_2)의 noisy translated된 문장(y)을 다시 언어(ℓ_1) source sentence로 원상복구 하는 task에 대한 objective 입니다.

$$y = M(x)$$
$$\mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_1, \ell_2) = \mathbb{E}_{x \sim \mathcal{D}_{\ell_1}, \hat{x} \sim d(e(C(y),\ell_2),\ell_1)}[\triangle(\hat{x}, x)]$$

Adversarial Training

Encoder가 언어와 상관없이 항상 같은 분포로 hyper plane에 projection하는지 검사하기 위한 discriminator가 추가되어 Adversarial Training을 진행합니다.

Discriminator는 latent variable z의 언어를 예측하여 아래의 cross-entropy loss를 minimize하도록 훈련됩니다. x_i, ℓ_i는 같은 언어(language pair)를 의미합니다.

$$\mathcal{L}_D(\theta_D|\theta, \mathcal{Z}) = -\mathbb{E}_{(x_i,\ell_i)}[\log p_D(\ell_i|e(x_i, \ell_i))]$$

따라서 encoder는 discriminator를 속일 수 있도록(fool) 훈련 되야 합니다.

$$\mathcal{L}_{adv}(\theta_{enc}, \mathcal{Z}|\theta_D) = -\mathbb{E}_{(x_i,\ell_i)}[\log p_D(\ell_j|e(x_i, \ell_i))]$$
$$where\ j = -(i-1)$$

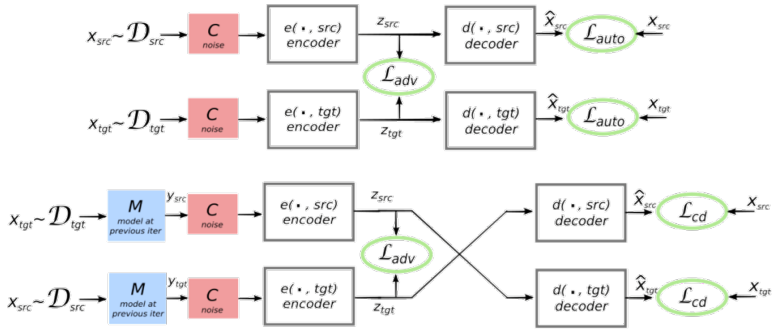


Figure 2: Illustration of the proposed architecture and training objectives. The architecture is a sequence to sequence model, with both encoder and decoder operating on two languages depending on an input language identifier that swaps lookup tables. Top (auto-encoding): the model learns to denoise sentences in each domain. Bottom (translation): like before, except that we encode from another language, using as input the translation produced by the model at the previous iteration (light blue box). The green ellipses indicate terms in the loss function.

Figure 10:

위의 3가지 objective를 결합하면 Final Objective Function을 얻을 수 있습니다.

$$\begin{aligned}\mathcal{L}(\theta_{enc}, \theta dec, \mathcal{Z}) =& \lambda_{auto}[\mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{src}) + \mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{tgt})] \\ &+ \lambda_{cd}[\mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{src}, \ell_{tgt}) + \mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{tgt}, \ell_{src})] \\ &+ \lambda_{adv}\mathcal{L}_{adv}(\theta_{enc}, \mathcal{Z}|\theta_D)\end{aligned}$$

λ를 통해서 선형결합(linear combination)을 취하여 기존의 손실함수에 추가 합니다. 이 과정을 pseudo code로 나타내면 아래와 같습니다.

Algorithm 1 Unsupervised Training for Machine Translation

procedure TRAINING($\mathcal{D}_{src}, \mathcal{D}_{tgt}, T$)
  Infer bilingual dictionary using monolingual data (Conneau et al., 2017)
  $M^{(1)} \leftarrow$ unsupervised word-by-word translation model using the inferred dictionary
  for $t = 1, T$ **do**
    using $M^{(t)}$, translate each monolingual dataset
    // discriminator training & model trainingl as in eq. 4
    $\theta_{discr} \leftarrow \arg\min \mathcal{L}_D, \quad \theta_{enc}, \theta_{dec}, \mathcal{Z} \leftarrow \arg\min \mathcal{L}$
    $M^{(t+1)} \leftarrow e^{(t)} \circ d^{(t)}$ // update MT model
  end for
  return $M^{(t+1)}$
end procedure

Figure 11:

이 논문에서 제안한 방식은 오직 단방향(monolingual) corpus만 존재할 때에 번역기를 만드는 방법에 대해서 다룹니다. 병렬(parallel) corpus가 없는 상황에서도 번역기를 만들 수 있다는 점은 매우 고무적이지만, 이 방법 자체만으로는 실제 필드에서 사용될 가능성은 적어 보입니다. 보통의 경우 필드에서 번역기를 구축하는 경우에는 병렬 corpus가 없는 경우는 드물고, 없다고 하더라도 단방향 corpus만으로 번역기를 구축하여 낮은 성능의 번역기를 확보하기보다는 비용을 들여 병렬 corpus를 직접 구축 한 후에, 병렬 corpus와 다수의 단방향 corpus를 합쳐 번역기를 구축하는 방향으로 나아갈 것이기 때문입니다.