

Deep Reinforcement Learning for Multi-class Imbalanced Text Classification with Pre-trained Encoder

120220222 이지현

120220228 최예린

Github 주소 및 팀원 별 기여 사항

- Github 링크: https://github.com/FromMusicToStory/RL_project
- 팀원 별 기여 사항:
 - 이지현: Dataset class 및 Text classifier 작성, Value Agent와 Reward 설정, 모델 구현, 실험 및 보고서 작성
 - 최예린: Environment 세팅 및 Policy Agent 구성, 모델 구현, hydra를 이용한 학습 코드 작성, 실험 및 보고서 작성

➤ Introduction

프로젝트 목표:

불균형한 데이터셋 분류 문제를 해결하는 Paper Implementation & Extension

- "Deep reinforcement learning for imbalanced classification."
- Lin, Enlu, Qiong Chen, and Xiaoming Qi. Applied Intelligence 50.8 (2020): 2488-2502.
- Paper Link: <https://arxiv.org/pdf/1901.01379v1.pdf>

- 실 세계에서 skewed class distribution은 challenging한 문제임.
- 위 논문에서는 이러한 불균형한 클래스 문제를 해결하기 위해, classification task를 sequential decision-making process로 치환함.
- Deep Q-learning network를 적용함.
 - 매 time step에 agent는 '분류' action을 수행
 - Environment는 action을 평가하고 agent에게 reward를 부여. 이 때 minority class일 때 더 큰 reward를 부여해, agent가 minority class에 민감하게 학습할 수 있게 함.
 - 특별한 reward function을 통해, agent는 불균형 데이터셋에서 optimal classification policy를 찾을 수 있음.

- 위 논문의 결과를 재현하고, 수업 시간에 배운 내용을 기반으로 확장하고자 함.

➤ Introduction

프로젝트 주제 및 기존 논문과의 차별점

- **프로젝트 주제:** Deep Reinforcement Learning For Multi-Class Imbalanced Korean Text Classification with Pre-trained Network
 - Binary-Class Classification이 아니라, Multi-Class Classification 문제를 해결해보고자 함.
 - 2개 클래스 → 10개 클래스
 - Text Classification task에 집중하고자 함.
 - 특히 영어 데이터셋이 아닌, 한국어 데이터셋을 이용함.
 - Pre-trained Text Encoder로 기존 network를 대체함.
 - 최근, 자연어 처리 분야에서 성능을 보이고 있는 BERT 기반 모델을 사용할 예정.
 - 기존에 사용된 DQN뿐만 아니라 DQN 확장 모델들을 적용해보고, Value-based 방식 외에 Policy-based 방법론을 적용하고, 그 결과를 비교해보고자 함.
 - DQN
 - Double DQN
 - Dueling DQN
 - Policy Gradient
 - Keras-RL 패키지를 이용한 기존 코드를 Pytorch Lightning 패키지로 변환하여 구현함.

➤ Dataset

사용 데이터셋

- KLAID (Korean Legal Artificial Intelligence Datasets) 데이터셋

- 법률 판결 예측 (Legal Judgement Prediction; LJP) task를 위해 구축된 데이터셋
- 특히, 판결문 중 1심 형사사건들을 분석해 뽑아낸 ‘형사 범죄 분류’ 데이터셋임.
- 법률 도메인은 전문가의 지식이 필요한 분야이지만, AI 모델을 이용하면 작업 효율성을 높이고 일반인들의 이해를 도울 수 있음.
- 데이터 개수: 161,192건
- 데이터 형태: 아래와 같은 형식으로, 피고인의 범죄 사실 (fact)에 대해 위반한 법령 (laws_service), 해당 법령 번호 (laws_service_id)로 이루어져 있음.

laws_service_id (int64)	fact (string)	laws_service (string)
32	"피고인은 2018. 8. 9. 23:33경 술을 마신 상태로 경산시 사동에 있는 상호 불상의 식당에서부터 같은 동에 있는 부영5차 앞 삼거리까지 B 스타렉스 승용차를 운전한 다음 승용차 안에서 잠을 자던 중, 차량 운전자가 시동을 걸어 놓고 잠을 자고 있다는 112 신고를 받고 현장에 출동한 경산경찰서 C파출소 소속 경위 D으로부터 피고인의 입에서 술 냄새가 나고 보행이 비틀거리는 등 술에 취한 상태에서 운전하였다고 인정할 만한 상당한 이유가 있어 약 10분 동안 총 3회에 걸쳐 음주측정기에 입김을 불어 넣는 방법으로 음주측정에 응할 것을 요구받고도 정당한 사유 없이 이에 응하지 아니하였다."	"도로교통법 제148조의2 제2항, 도로교통법 제44조 제2항"

- 범죄 사실 (fact)이 입력으로 들어가면, 법령 번호 (law_service_id)를 출력하는 classification 문제를 풀고자 함.

➤ Dataset

데이터셋 전처리

- 데이터셋 추출

- 원래 데이터셋의 class는 총 177개, 전체 데이터셋 중 90%를 train set으로 사용 시 데이터 개수는 총 145,072개로 상당히 많은 class로 이루어져 있었음.
- 시간과 컴퓨팅 자원 문제로 인해 데이터 개수가 많은 순서대로 10개 class의 데이터를 추출해서 실험을 진행함. (train set: 71,159 / test set: 7,928)
- 경험적으로 multi-class text classification 문제는 약 5만 개 이상의 데이터를 사용할 경우 잘 학습된다고 판단했고, 아래와 같이 10개 클래스 안에서도 불균형도가 크게 나타났기 때문에 ‘불균형 데이터셋 학습’이라는 본 실험의 목적에도 부합한다고 판단됨.

class	8	9	4	1	7	34	20	16	0	5
num	11355	10798	9267	7684	7057	5510	5295	4939	4832	4422

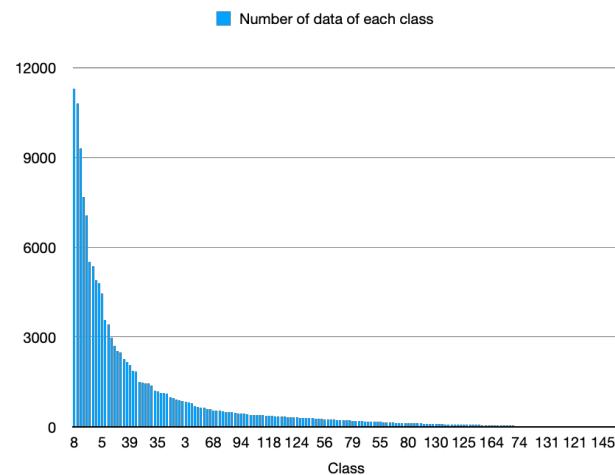
- 데이터셋 전처리

- Text Encoder의 입력으로 들어갈 수 있도록, BPE 기반 tokenizer를 통해 인코딩 처리

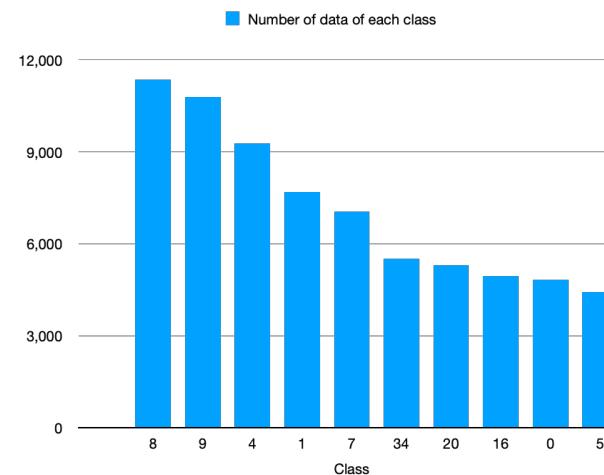
➤ Dataset

데이터셋 히스토그램

- 데이터셋 분포



전체 클래스별 데이터 분포



추출한 10개 클래스 데이터 분포

- 클래스 별 데이터 통계

- Sum : 전체 데이터 개수
- Max / min : 데이터가 가장 많은/적은 클래스의 데이터 개수
- Mean : 클래스 별 데이터 개수의 평균
- Median : 클래스 별 데이터 개수의 중간값

	All classes (177)	Top 10 classes
sum	145,072	71,159
max	11,355	11,355
min	10	4,422
mean	819.616	7,115.9
median	214	6283.5

➤ Environment Setting

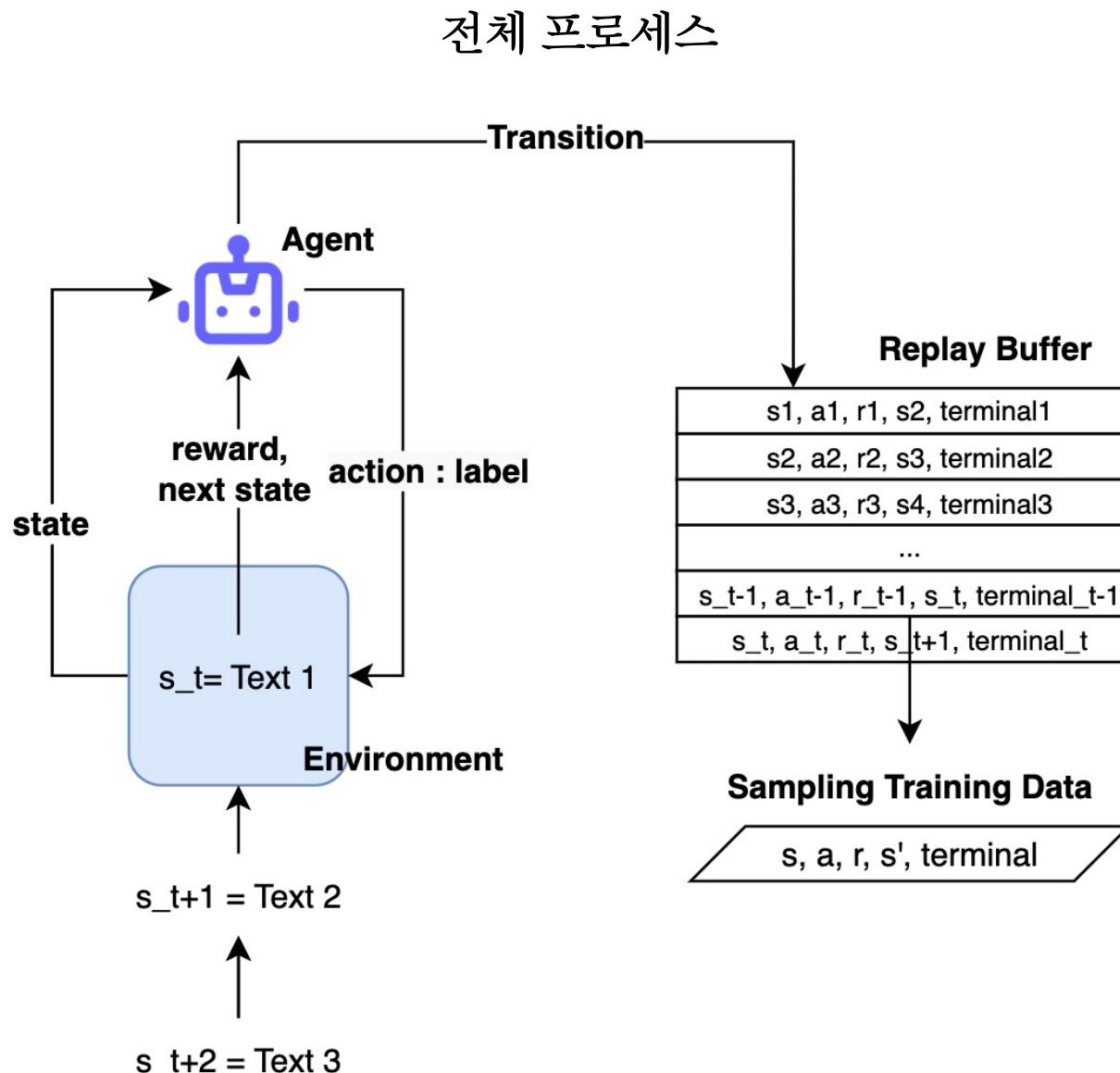
Problem & Environment

- Imbalanced Text Classification 문제로, 데이터셋 $D = (\mathbf{x}_1, l_1), (\mathbf{x}_2, l_2) \dots, (\mathbf{x}_t, l_t)$ 에서 \mathbf{x}_t 가 들어오면, 라벨 l_t 를 분류하는 문제라고 할 수 있음.
- 매 time step마다 \mathbf{x}_t 가 입력으로 들어오고, state s_t 는 ‘라벨 예측’이라는 action a_t 을 수행함.
- 이때 majority class의 라벨만큼 minority class의 라벨을 잘 예측하고자 함.
- 처음 Initializing 시에 train set을 shuffle하고, step 0부터 전체 데이터셋 길이만큼 전체 데이터가 다 분류될 때까지 수행해서 trajectory를 저장한 후에 에피소드가 종료됨.
- 혹은 agent가 minority class의 예측에 실패할 경우에도 에피소드가 종료됨.

State, Action, Reward, Policy & Transition probability

- **State S** : 1~ T 까지 매 time step마다 입력으로 들어오는 text sample. $s_t = x_t$ 라고 할 수 있음.
- **Action A** : agent가 state S (text sample)에 대한 label을 예측하는 행동, Multi-class classification이기 때문에 class가 10개라면, $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ 로 정의할 수 있음.
- **Reward R** : 환경에서 agent의 행동이 imbalanced dataset 분류 task를 성공했는지 실패했는지 feedback을 주기 위해, 특별히 디자인된 Reward function을 통해 majority class보다 minority class sample에 대해 더 큰 reward / punishment를 제공함. (Reward function은 p.11에 자세히 설명)
- **Policy π_θ** : parameter가 θ 인 text classifier를 의미. $\pi_\theta(x_t)$ 는 state s_t 에서 action a_t 를 수행하는 것을 나타냄. 이 때 학습을 통해 optimal classification policy $\pi^* : S \rightarrow A$ 를 찾는 것이 목표임.
- **Transition probability P** : agent가 state s_t 에서 다음 state s_{t+1} 로 이동할 때의 확률 $p(s_{t+1} | s_t, a_t)$ 은 training sample 순서에 따라 deterministic함.

➤ Methodology



* 논문에 있는 그림을 본 프로젝트의 데이터에 맞게 수정

➤ Methodology

Reward function for Imbalanced multi-class data classification

$$R(s_t, a_t, \underline{l}_t) = \begin{cases} +1, & a_t = l_t \text{ and } s_t \in D_P \\ -1, & a_t \neq l_t \text{ and } s_t \in D_P \\ \lambda, & a_t = l_t \text{ and } s_t \in D_N \\ -\lambda, & a_t \neq l_t \text{ and } s_t \in D_N \end{cases}$$

D_P : Minority class
 D_N : Majority class

- Majority class와 Minority class의 구분
 - 총 데이터 개수의 평균보다 데이터가 많을 경우 Majority class, 적을 경우에는 Minority class로 설정함.
- Minority class에 더 많은 가중치를 주기 위해, 다음과 같이 reward 설정
 - Minority class에 속하면서 agent가 정답을 맞췄을 때는 +1, 정답을 맞추지 못했을 때는 -1
 - Majority class에 속하면서 agent가 정답을 맞췄을 때는 $+\lambda$, 정답을 맞추지 못했을 때는 $-\lambda$
- λ 값은 $\frac{D_N}{D_N+D_p}$ 와 같이 전체 class 대비 해당 class의 비율로 설정하였고, Majority class의 클래스 별 λ 값은 아래와 같음.

Majority class	8	9	4	1
Lambda	0.1586	0.1521	0.1318	0.1073

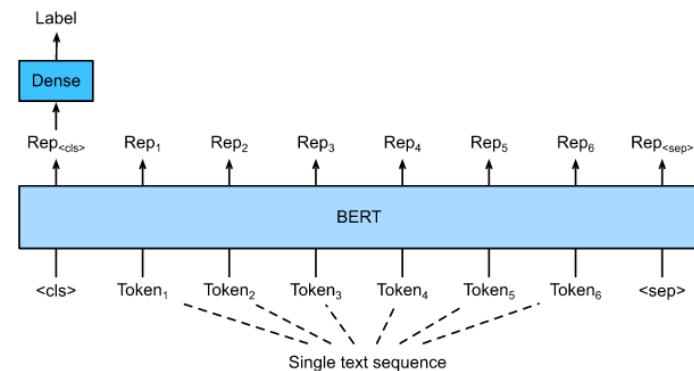
* 표의 λ 값은 소수점 다섯째 자리에서 반올림함.

➤ Methodology

DQN Algorithm

```
class Classifier(nn.Module):
    def __init__(self, model_name='klue/roberta-base', num_classes=10):
        super().__init__()
        self.model = AutoModel.from_pretrained(model_name)
        self.clf_layer = nn.Linear(self.model.config.hidden_size, num_classes)
        self.dropout = nn.Dropout(self.model.config.hidden_dropout_prob)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, input_ids, attention_mask):
        outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs[1]
        logits = self.clf_layer(pooled_output)
        return logits
```



- 위와 같은 구조의 network에서 state s_t 로 text encoding (input_ids, attention_mask) 값을 입력 받았을 때, class 10개 중 한 개를 선택하는 action a_t 를 취하게 됨.

➤ Methodology

DQN Algorithm

```

class ValueAgent(Agent):
    def __init__(self, env: ClassifyEnv, replay_buffer: ReplayBuffer):
        self.env = env
        self.reset()
        self.buffer = replay_buffer
        self.state = self.env.reset()

    def get_action(self, model: nn.Module, state: torch.Tensor, epsilon: float, device: str) -> int:
        if np.random.random() < epsilon:
            action = self.get_random_action()
        else:
            action = self.get_normal_action(model, state, device)
        return action

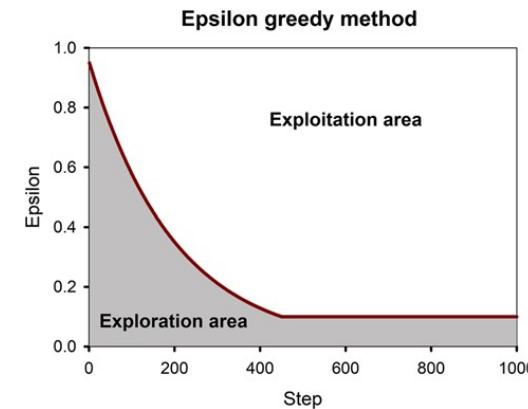
    def get_random_action(self) -> int:
        return randint(0, self.env.action_space.n - 1)

    def get_normal_action(self, model: nn.Module, state: torch.Tensor, device: str) -> int:
        input_id, attention_mask = state[0].unsqueeze(0), state[1].unsqueeze(0)

        if not isinstance(input_id, torch.Tensor):
            input_id = torch.Tensor(input_id).float()
            attention_mask = torch.Tensor(attention_mask).float()
        if device != 'cpu':
            input_id = input_id.to(device)
            attention_mask = attention_mask.to(device)

        logits = model(input_ids=input_id, attention_mask=attention_mask)      # classification model
        action = torch.argmax(logits, dim=1)
        return int(action)

```



$$a = \arg \max_a Q_\theta(s, a)$$

- Action a_t 를 선택할 때는 적절한 exploration과 exploitation의 조화를 위해 Epsilon-greedy policy에 따라서, epsilon보다 작을 경우 random action을 선택하고, epsilon보다 클 경우 argmax 값을 선택함.
- Epsilon 값은 학습이 진행될수록 작아지도록 설정함.

➤ Methodology

DQN Algorithm

Reward function

```

def step(self, prediction):
    # Input : model's prediction value
    # Output : data, reward, is_terminal, info
    self.y_preds.append(prediction)
    info = {}
    terminal = False
    answer_t = self.answer[self.step_id] # answer of this step
    if prediction == answer_t:
        if answer_t in self.minorities:
            reward = 1
        else:
            reward = 1.0 * self.imb_rate[answer_t]
    else:
        if answer_t in self.minorities:
            reward = -1
            if self.run_mode == 'train':
                terminal = True
            else:
                reward = -1.0 * self.imb_rate[answer_t]
        self.step_id += 1

    if self.step_id == self.game_len - 1:
        y_true_cur = self.answer[:self.step_id]
        info['fmeasure'] = self.get_metrics(self.y_preds, y_true_cur)

    terminal = True # end of step

    return (self.env_data[self.step_id][0], self.env_data[self.step_id][1]), reward, terminal, info

```

```

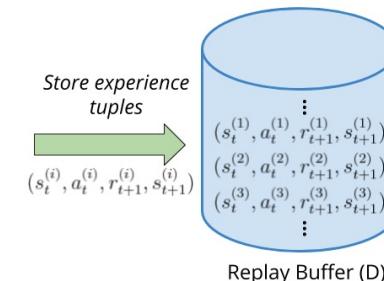
@torch.no_grad()
def step(self, model: nn.Module, epsilon: float, device: str = "cuda:0") -> Tuple[float, bool]:
    action = self.get_action(model, self.state, epsilon, device)
    new_state, reward, terminal, _ = self.env.step(action)
    trans = Transition(self.state, action, reward, new_state, terminal)

    self.buffer.append(trans)

    self.state = new_state
    if terminal:
        self.reset()
    return reward, terminal

def reset(self):
    self.state = self.env.reset()

```



- 선택된 action에 따라 위와 같이 Reward function에 따른 reward r_t 를 부여 받음.
- 매 step을 수행하면서 replay buffer에 transition (s_t, a_t, r_t, s_{t+1})을 저장함.

➤ Methodology

DQN Algorithm

```
class DQNLoss(object):
    def __init__(self, criterion, gamma):
        self.criterion = criterion
        self.gamma = gamma

    def __call__(self, batch, main_net, target_net):
        states, actions, rewards, next_states, terminals = batch
        state_action_values = main_net(input_ids=states[0], attention_mask=states[1]).gather(1, actions.unsqueeze(-1))
        with torch.no_grad():
            next_state_values = target_net(input_ids=states[0], attention_mask=states[1]).max(1)[0]
            next_state_values[terminals] = 0.0
            next_state_values = next_state_values.detach()
        expected_state_action_values = next_state_values * self.gamma + rewards
        return self.criterion(state_action_values, expected_state_action_values)
```

$$L(\theta) = \frac{1}{K} \sum_{i=1}^K (r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a') - Q_{\theta}(s_i, a_i))^2$$

↓ ↓
 Compute Compute
 using θ' using θ

$$Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a')$$

- 학습의 안정성을 위해 Target Q network θ' 를 freeze하고, 위와 같이 target value에 대한 Loss를 계산한 다음, main Q network θ 를 업데이트하는 방식을 취함.

➤ Methodology

Double DQN Algorithm

```
class DoubleDQNLoss(DQNLoss):
    def __init__(self, criterion, gamma):
        super(DoubleDQNLoss, self).__init__(criterion=criterion, gamma=gamma)

    def __call__(self, batch, main_net, target_net):
        states, actions, rewards, next_states, terminals = batch
        state_action_values = main_net(input_ids=states[0], attention_mask=states[1]).gather(1, actions.unsqueeze(-1)).
        with torch.no_grad():
            next_state_actions = main_net(input_ids=states[0], attention_mask=states[1]).max(1)[1]
            next_state_values = target_net(input_ids=states[0], attention_mask=states[1]).gather(1, next_state_actions.
            next_state_values[terminals] = 0.0
            next_state_values = next_state_values.detach()
        expected_state_action_values = next_state_values * self.gamma + rewards
        return self.criterion(state_action_values, expected_state_action_values)
```

$$y = r + \gamma Q_{\theta'}(s', \arg \max_{a'} Q_{\theta}(s', a'))$$
$$y = r + \gamma Q_{\theta'}(s', a')$$

- Double DQN은 DQN의 overestimation 문제를 해결하기 위해, main Q network θ 를 이용해 action을 선택하고 해당 action의 target Q network θ' 에서 가지는 Q-value를 활용하는 방식으로 진행됨.

Dueling DQN Algorithm

```

class DuelingClassifier(nn.Module):
    def __init__(self, model_name='klue/roberta-base', num_classes=10):
        super().__init__()
        self.model = AutoModel.from_pretrained(model_name)
        self.advantage_layer = nn.Linear(self.model.config.hidden_size, num_classes)
        self.value_layer = nn.Linear(self.model.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs[1]
        value = self.value_layer(pooled_output)
        advantage = self.advantage_layer(pooled_output)
        adv_average = torch.mean(advantage, dim=1, keepdim=True)
        return value + advantage - adv_average

```

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{\mathcal{A}} \sum_{a'} A(s, a') \right)$$

- Dueling DQN은 advantage function $A(s, a) = Q(s, a) - V(s)$ 을 이용하는 방식으로, state s 에서 상대적인 action a 의 가치를 고려할 수 있음.
- Identifiability issue를 고려하기 위해, 위와 같은 수식으로 A 의 normalize를 수행함.

Policy Gradient Algorithm

1. Initialize the policy network parameter θ and value network parameter ϕ
2. Generate N number of trajectories $\{\tau^i\}_{i=1}^N$ following the policy π_θ
3. Compute the return (reward-to-go) R_t
4. Compute the policy gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^N \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t | s_t) (R_t - V_\phi(s_t)) \right]$$

5. Update the policy network parameter θ using gradient ascent as $\theta = \theta + \alpha \nabla_\theta J(\theta)$
6. Compute the MSE of the value network:

$$J(\phi) = \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^{T-1} (R_t - V_\phi(s_t))^2$$

7. Compute gradients $\nabla_\phi J(\phi)$ and update the value network parameter ϕ using gradient descent as $\phi = \phi - \alpha \nabla_\phi J(\phi)$
8. Repeat steps 2 to 7 for several iterations

```

def calculate_returns(self, rewards):
    discounted_rewards = [math.pow(self.gamma,i) * r for i, r in enumerate(rewards)]
    return [sum(discounted_rewards[i:]) for i in range(len(discounted_rewards))]

returns = self.calculate_returns(self.episode_reward)

p_loss = [-1 * self.episode_prob[i] * returns[i] for i in range(len(self.episode_prob))]
p_loss = torch.mean(torch.stack(p_loss))

v_loss = self.v_criterion(torch.tensor(returns).to(device), predictions[:len(self.episode_prob)])

p_opt.zero_grad()
p_loss.requires_grad = True
self.manual_backward(p_loss)
p_opt.step()

v_opt.zero_grad()
v_loss.requires_grad = True
self.manual_backward(v_loss)
v_opt.zero_grad()

self.episode_steps = 0
self.episode_reward = []
self.episode_prob = []

```

➤ Policy Gradient는 위와 같은 알고리즘에 따라 return을 저장한 다음 policy gradient를 계산하고, gradient ascent에 따라 policy network를 업데이트하는 방식을 취함. 이후, value network로 Loss를 계산해 value network의 gradient descent update를 진행함.

➤ Experiment

Experiment Environment

- 학습 속도를 고려하여 2개의 서버를 사용하여 훈련 및 추론 진행

		서버 A	서버 B
SW	OS	Ubuntu 20.04.5 LTS	Ubuntu 18.0.4 LTS
	Pytorch	1.12.0	1.12.1
	CUDA	11.3	11.3
HW	CPU	AMD Ryzen Threadripper PRO 3975WX 총 64 core	Intel Xeon Processor (Cascadelake) 2개 총 16 core
	GPU	RTX A5000 24GB 4개	A100 40GB 2개

- 두 서버의 Pytorch, CUDA version은 동일하므로, 두 서버의 사양 차이가 훈련 및 테스트 성능에 미치는 영향은 미미할 것으로 판단하고, 실험 진행.
- 또한, 학습 프로세스가 batch processing을 하지 않으므로 GPU 차이로 인한 모델 간 훈련 시간의 차이도 없었음.
 - 단, GPU의 용량이 다르므로 한번에 돌릴 수 있는 모델의 개수가 달랐음.

Hyperparameter setting & Metrics

- Pretrained Language Model : Klue–Roberta Base ([Huggingface link](#))
 - KLUE : Korean Language Understanding Benchmark
 - 사전 학습 데이터 : Korean Wikipedia, News, NSMC etc.
 - RoBERTa : BERT에 dynamic masking 등의 방법론을 적용한 확장된 모델

- Metrics
 - F1-Score : 각 클래스 별로 F1-score를 계산한 후, unweighted mean을 취함.
 - Class imbalance를 고려하지 않은 Metric임.
 - Sklearn 패키지의 f1_score 함수를 이용.
 - 작동 예시 : `f1_score(trues, preds, average='macro')`
 - Weighted F1-Score : 각 클래스 별로 F1-score를 계산한 후, weighted mean을 구함.
 - Class imbalance를 고려한 Metric임.
 - 작동 예시 : `f1_score(trues, preds, average='weighted')`

➤ Experiment

Hyperparameter setting & Metrics

- Train Hyperparameter

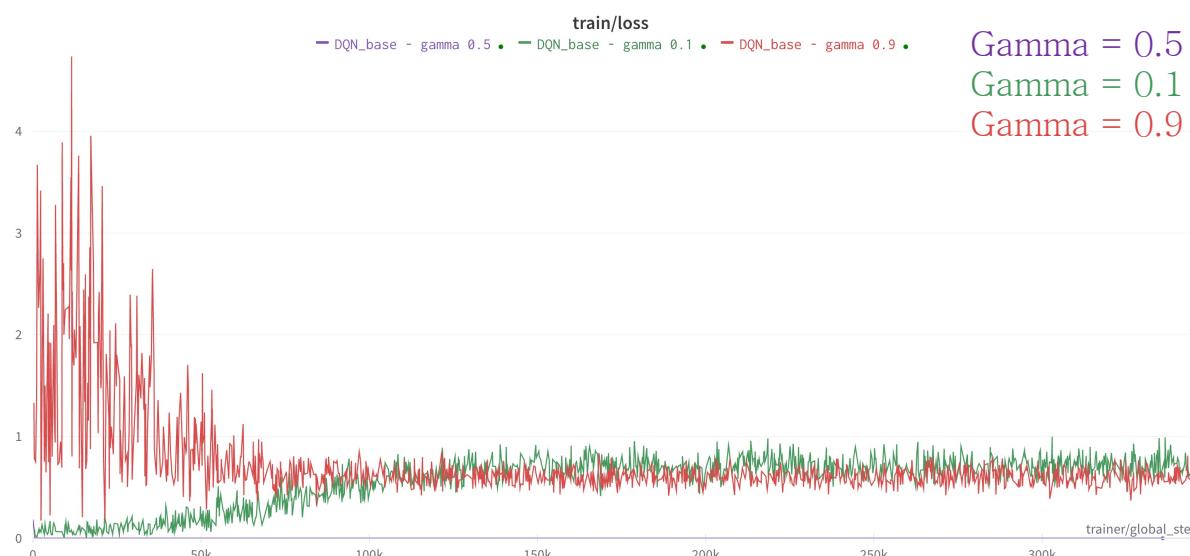
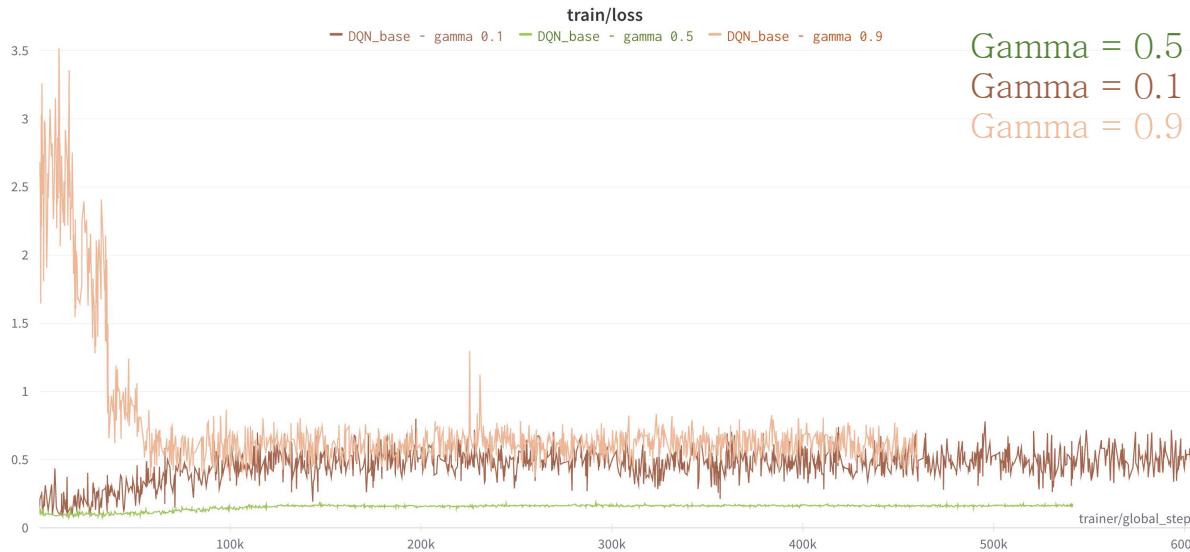
Hyperparameter	value
Optimizer	AdamW
Learning Rate	5e-5
Max Epoch	200
Accumulated grad batches	1
Sync rate	10
Epsilon start	1.0
Epsilon end	0.01
Initial epsilon for populating	1.0

- 이 Hyperparamter들은 이후 슬라이드에 나오는 hyperparameter 실험에서 모두 동일하게 설정하여 비교 실험이 가능하도록 함.
- Epsilon 값은 학습 초기에는 1.0, 학습이 진행됨에 따라 줄어들어 0.01까지 수렴하도록 설정함.

➤ Experiment

Experiment Result

- DQN – discount factor (gamma : 0.1 / 0.5 / 0.9)



➤ 위/아래 그래프는 각각 다른 random seed 하에서 학습된 DQN 모델의 train loss 그래프임.

- 위 : Random Seed: 42
- 아래 : Random Seed: 1111

➤ 두 그래프 모두 gamma가 0.5일 때 loss가 가장 아래에 수렴하고 있음을 확인할 수 있음.

➤ Gamma가 0.9일 때는 loss가 높게 시작했다가 떨어지는 양상을 보임.

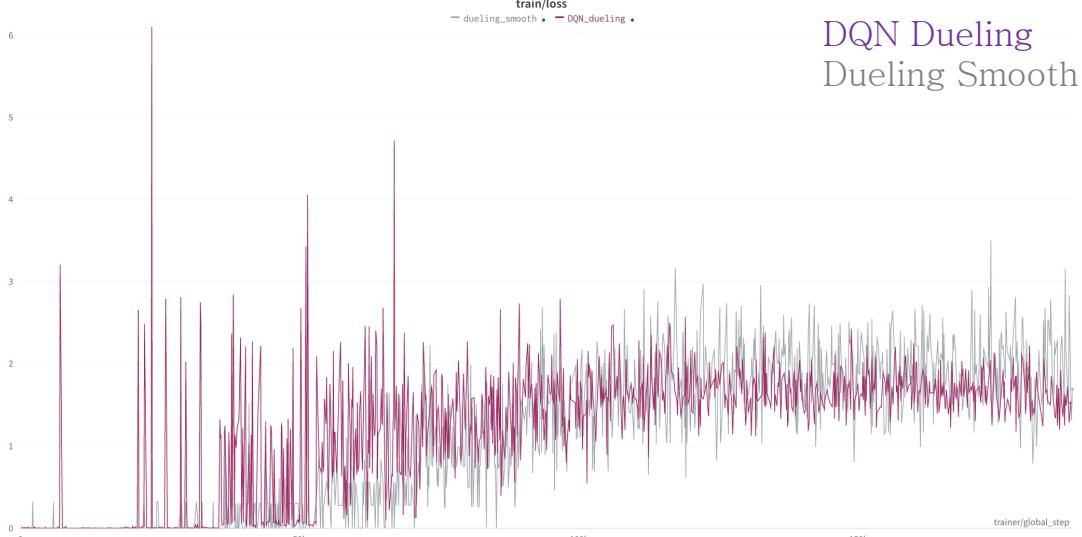
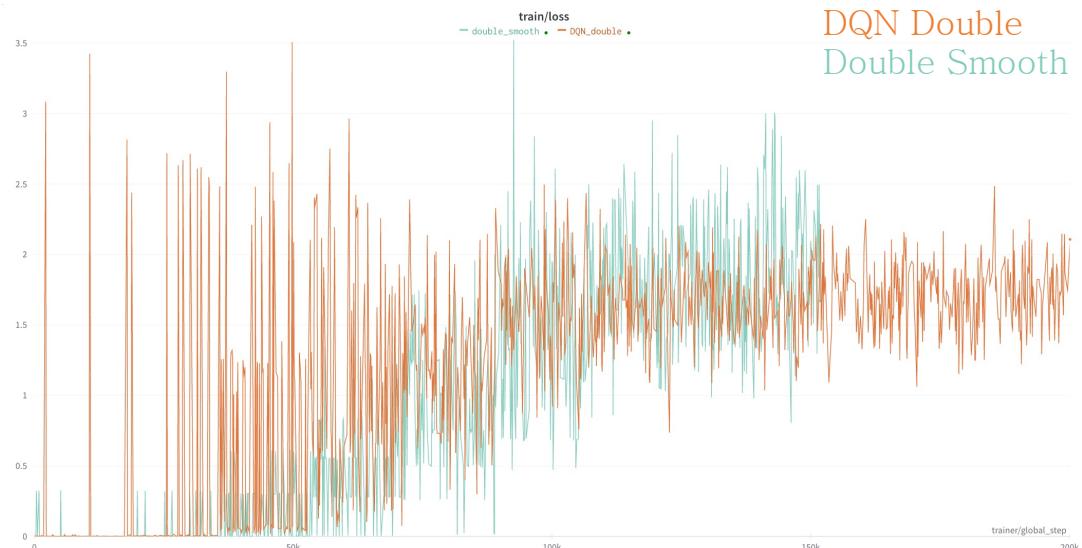
➤ Gamma가 0.1일 때는 학습이 진행됨에 따라 loss가 오히려 올라가는 모습을 보임.

➤ 미래를 고려하면서 학습할수록, 학습이 잘 되는 경향이 있는 것으로 보임.

➤ Experiment

Experiment Result

- DQN – loss function (MSE / Smooth L1 loss)

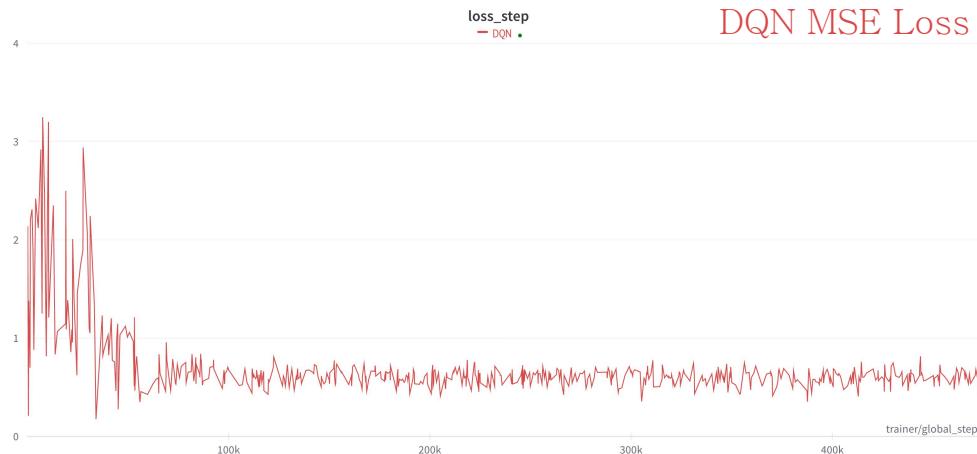


- 위/아래 그래프는 각각 다른 Loss 함수를 이용하여 학습된 Double DQN / Dueling DQN 모델의 train loss 그래프임.
- 50k step 이전에는 MSE loss가 Smooth L1 Loss에 비해 진폭이 커짐.
- 50k step 이후부터는 오히려 Smooth L1 Loss가 더 진폭이 커짐.
- 학습이 진행될 수록 Q 값이 정답에 가까워지고 이상치가 줄어들기 때문에 초반에는 이상치에 취약한 MSE loss의 진폭이 크지만 중반 이후부터는 Smooth L1 loss 보다 진폭이 작은 것으로 해석할 수 있음.

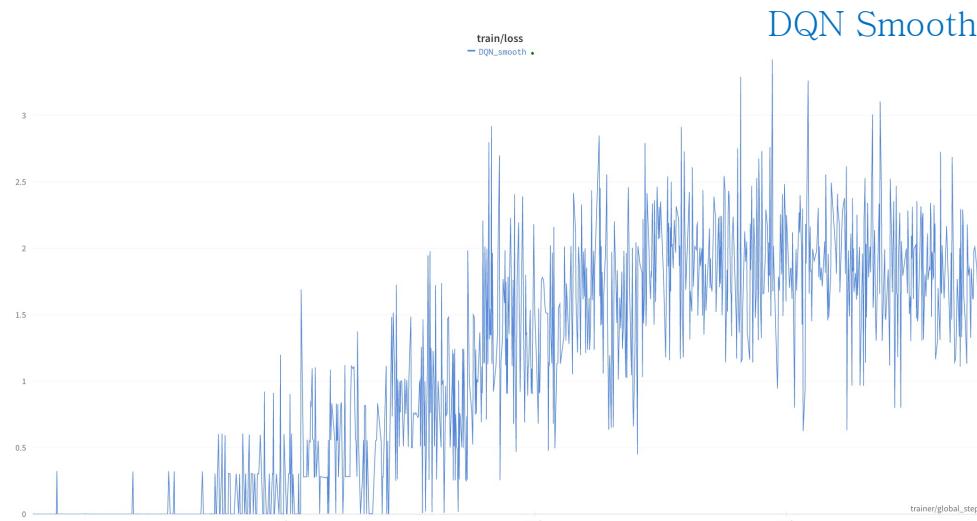
➤ Experiment

Experiment Result

- DQN – loss function (MSE / Smooth L1 loss)



➤ DQN 모델도 같은 양상을 보였음.

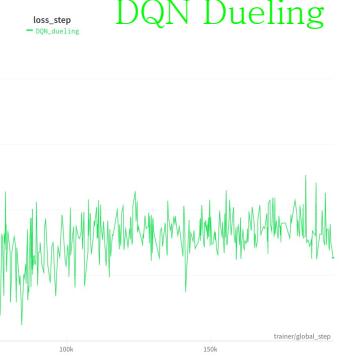
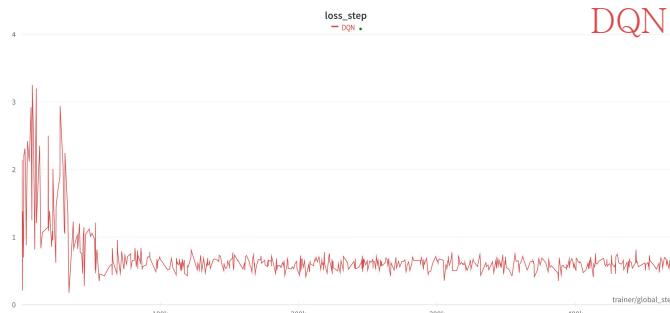


➤ Experiment

Experiment Result

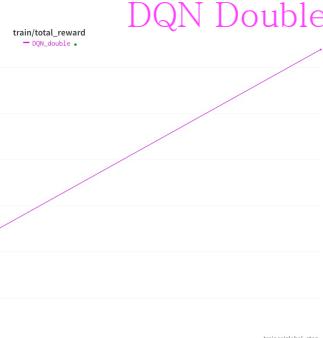
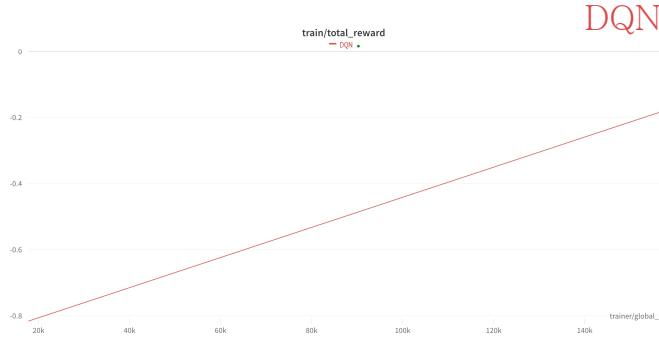
■ DQN Algorithm Variation – DQN / Double DQN / Dueling DQN

• Train Loss



➤ Train loss가 하향하는 양상은 같지만, loss가 가장 적은 것은 DQN algorithm임.

• Train reward



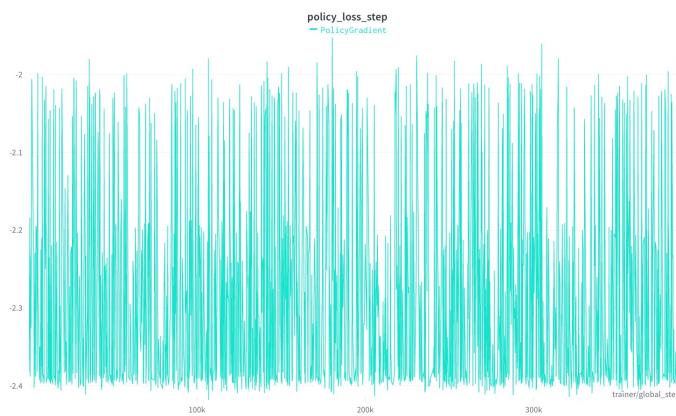
➤ 세 algorithm 모두 reward가 점차 증가하는 양상을 보였음. 그런데 DQN의 경우, 다른 두 알고리즘과 다르게 reward가 음수에 머물렀음. 동일 train step (100k) 기준 DQN은 -0.4 이하, Double DQN은 0.72, Dueling DQN은 0.9 였음. DQN은 다른 두 알고리즘에 비해 Reward의 증가 속도가 더딘 것을 알 수 있음.

➤ Experiment

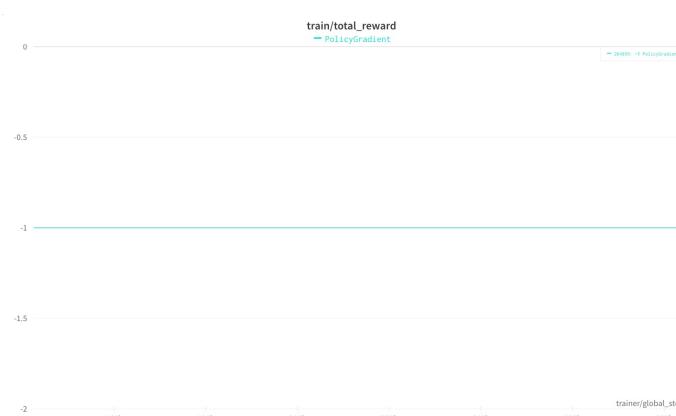
Experiment Result

■ Policy Gradient

■ Train loss



■ Train Total reward



➤ Gradient Oscillation 이 나타남을 확인함.

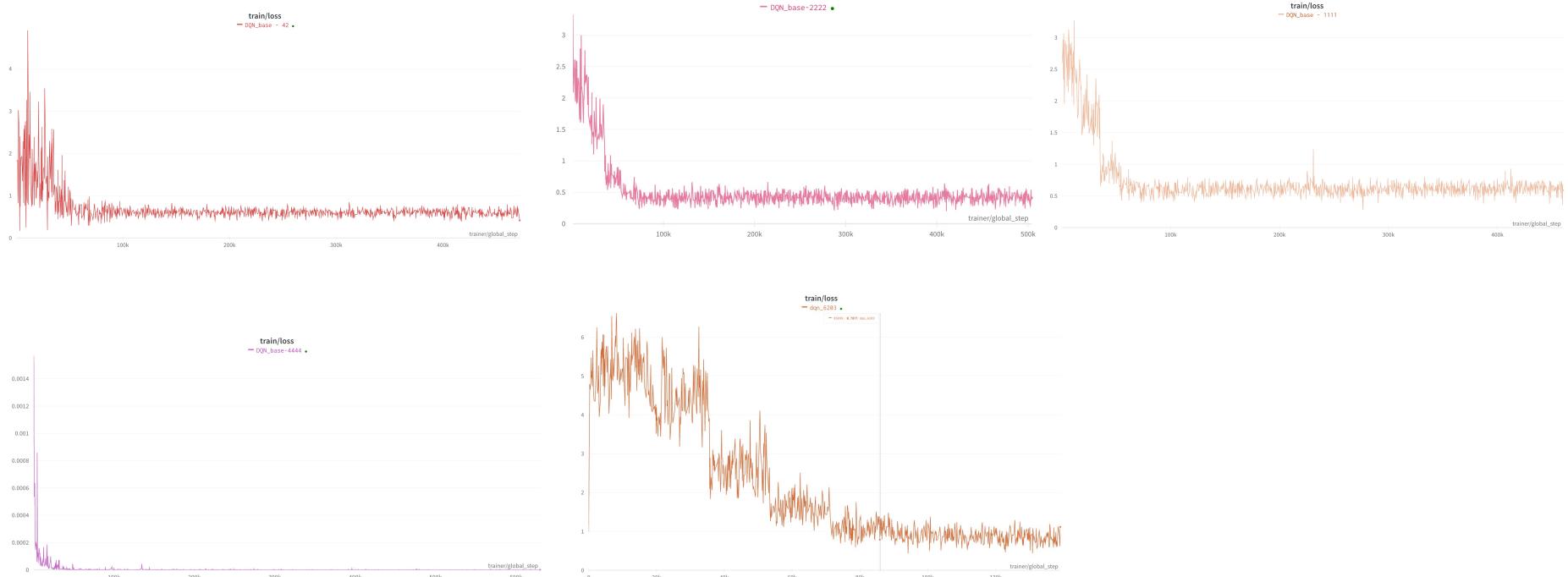
- 딥러닝 모델에서는 gradient가 너무 크면 학습이 잘 되지 않는 경향이 있음.
- Policy Gradient 방법론에서 파라미터 업데이트 시 gradient의 높은 분산으로 인해 학습이 잘 수렴되지 않는 문제가 발생하는 것으로 예상됨.
- 그래프 상으로는 진폭이 큰 것으로 보이지만 최댓값과 최솟값의 차이가 0.4로 작은 값임.
- 이는 대규모 데이터를 사용했기 때문으로 해석할 수 있음.
- Policy net이 1 episode를 수행한 이후 gradient를 update하는데, 이때 update interval이 너무 커졌던 것으로 보임.
- Learning rate를 줄이거나 variance reduction을 위한 여러 방법론들을 적용함으로써 학습이 잘 수렴되도록 하는 추가 실험이 필요할 것으로 보임.

➤ Experiment

Experiment Result

■ Random seed – DQN

- 실험한 Random seed numbers : 42, 1111, 2222, 3333, 4444, 5555, 666, 777, 3040, 6203, 6427 등



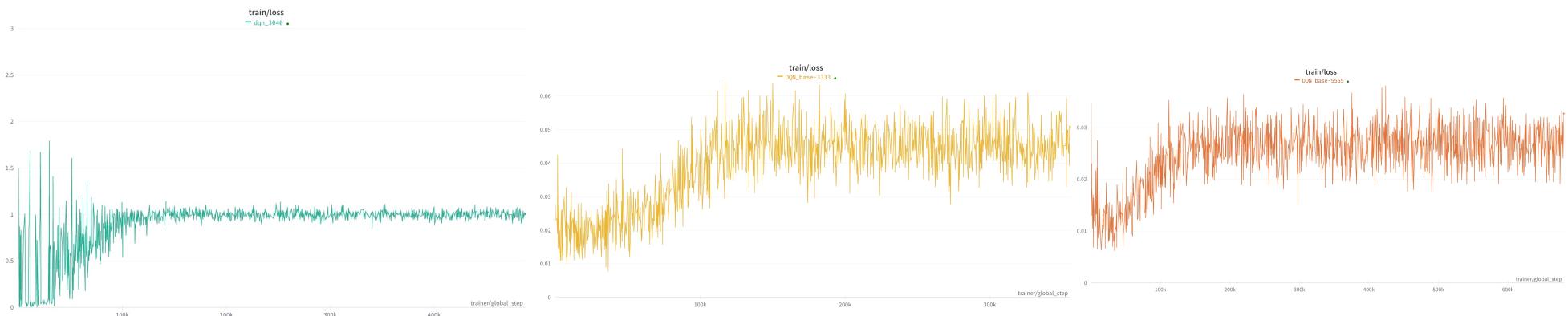
- 무작위로 여러 seed에서 DQN 모델을 학습시켰을 때 학습 양상을 보면, 대략 100k step부터 loss가 줄어들어 수렴하는 경향성을 보이는 그래프가 다수 있었음.

➤ Experiment

Experiment Result

■ Random seed – DQN

- 실험한 Random seed numbers : 42, 1111, 2222, 3333, 4444, 5555, 666, 777, 3040, 6203, 6427 등



- 그러나 몇몇 seed에서는 loss가 오히려 증가하는 형태를 보이기도 함.
- 또한, seed에 따라 수렴되는 값이 약 0.5 / 0.003 / 1로 다양하게 나타남.
- 즉, seed에 따른 학습에 영향력이 있는 것으로 보이며, 추가적인 실험 및 통계적 분석이 필요할 것으로 파악됨.

➤ Conclusion

Conclusion and Future Works

▪ Conclusion

- DQN의 확장 모델들 (Double DQN, Dueling DQN)과 DQN 간 뚜렷한 성능 차이가 나타나지는 않음.
 - 이는 학습을 위해 구성한 모델이 pretrained model 위에 linear layer를 DQN의 종류에 따라 1~2 개 붙인 구조인데, 사전 학습 모델에 비해 그 layer 개수가 적기 때문에 모델 학습 결과에 대한 영향이 미미했던 것으로 보임.
- 수업 시간에 이용했던 예제용 environment들과 달리 대규모 데이터셋과 state, 불확실성이 큰 policy (classifier)로 인해, 모델 학습 시 loss 수렴이 잘 되지 않았음.
 - 학습 시간이 매우 길었으며, 실제로 f1 score 측정 결과에서도 본 프로젝트의 목적인 imbalanced text classification의 성능이 좋지 못했음.

▪ Future Works

- Vanilla Classification 모델과 Deep Reinforcement Learning 모델의 성능 비교를 통해, Imbalanced Text classification task에서 강화학습의 유효성을 검증해볼 필요가 있음.
- 사전 학습 모델 위에 더 많은, 복잡한 Layer를 올려서 fine-tuning이 더 잘 되도록 학습해볼 수 있음.
- 추가적으로 Actor-Critic과 같은 다른 방법론들을 적용하고 평가할 수 있음.
- Reward function에 Multi-Class classification을 위한 특성을 반영하는 것이 필요. 현재 minority와 majority class를 binary 형태로 구분하지만, scale 형태로 확장할 수 있음.
- 매 time step마다 하나의 state에 대해 계산하는 대신, batch processing을 적용해 훈련 속도를 높일 수 있음.

References

- **Paper Link**
 - <https://arxiv.org/pdf/1901.01379v1.pdf>
- **Official Paper Code**
 - <https://github.com/linenus/DRL-For-imbalanced-Classification>
- **Dataset**
 - <https://huggingface.co/datasets/lawcompany/KLAID>
- **For Code Implementation (Pytorch-lightning tutorial):**
 - https://pytorch-lightning.readthedocs.io/en/stable/notebooks/lightning_examples/reinforce-learning-DQN.html

감사합니다
