KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- **정책 경사 학습(Policy Gradient Learning)**
  - 에이전트가 주어진 일을 더 잘하도록 가중치를 어떻게 변경할지 방향을 추정하는 방안을 제공한다. 각 가중치를 임의로 움직이는 대신 경험 데이터를 분석해서 특정 가중치를 늘리거나 줄이는 게 나을지 추측할 수 있다.
  - 여전히 무작위성이 사용되지만 정책 경하 학습은 이로 인한 결과를 좀 더 낫게 만들어준다.
- 정책 경사 체계 요약
  - **에이전트가 이긴 경우** : 이 경기에서 사용한 수에 대한 확률을 높인다.
  - **에이전트가 진 경우** : 이 경기에서 사용한 수에 대한 확률을 낮춘다.

좋은 결정을 정의하는 방법

- 다 더하기 게임 (Add It Up)
  - 게임 방식
    - 각 차례에서 각 선수는 1부터 5 중 하나의 숫자를 선택한다.
    - 100번째 차례가 지난 후 각 선수는 본인이 선택한 숫자를 모두 더한다.
    - 총 합이 가장 높은 선수가 이긴다.
  - 최적의 전략은 매 차례에 5를 고르는 것이다. 여기서는 게임 결과에 따라 확률적 정책을 점진적으로 향상시키는 **정책 학습**을 나타내는 데 사용할 것이다. 모두 이 게임의 최적 전략을 알고 있으므로 정책 학습이 완벽한 대국을 치르는 모습을 보게 될 것이다.

좋은 결정을 정의하는 방법

- 다 더하기 게임 (Add It Up)
  - 단순한 게임이지만 바둑과 같이 훨씬 더 진지한 게임에 적용해볼 수 있다.
    다 더하기 게임은 바둑과 마찬가지로 끝날 때까지 오래 걸리고, 한 게임 내에서도
    선수에게는 게임을 흥미진진하게 풀거나 망칠 기회가 여러 번 주어진다.
  - 경기 결과를 사용해서 정책을 갱신하려면 어떤 수가 점수를 얻을 것인지 라든가
    경기 승패에 영향을 미쳤는지 정의할 수 있어야 한다.
    → 이를 **점수 할당 문제(Credit Assignment Problem)**라고 한다.
  - 여기서는 각 결정에 점수를 부여해 수많은 경기 결과의 평균을 구하는 방법을 설명한다.
  - 이제 동일한 확률로 5개의 숫자 중 아무거나 하나를 선택하는 완전한 임의의 정책을
    사용하는 것부터 시작해보자.

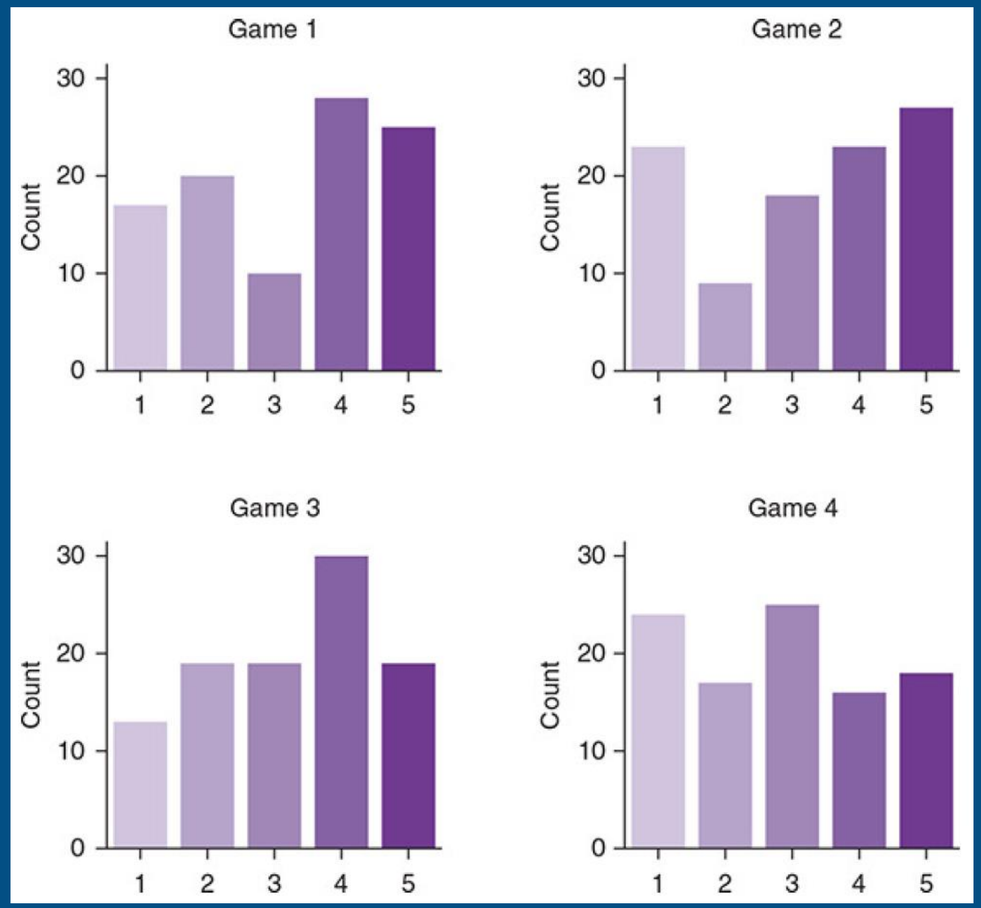
좋은 결정을 정의하는 방법

- 1부터 5 사이의 숫자를 임의로 선택하기

```python
import numpy as np

counts = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}

for i in range(100):
  choice = np.random.choice([1, 2, 3, 4, 5],
                            p=[0.2, 0.2, 0.2, 0.2, 0.2])
  counts[choice] += 1

print(counts)
```

좋은 결정을 정의하는 방법

- 1부터 5 사이의 숫자를 임의로 선택하기



좋은 결정을 정의하는 방법

- 다 더하기 게임 시뮬레이션

```python
def simulate_game(policy):
  player_1_choices = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
  player_1_total = 0
  player_2_choices = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
  player_2_total = 0

  for i in range(100):
    player_1_choice = np.random.choice([1, 2, 3, 4, 5],
                                       p=policy)
    player_1_choices[player_1_choice] += 1
    player_1_total += player_1_choice
    player_2_choice = np.random.choice([1, 2, 3, 4, 5],
                                       p=policy)
    player_2_choices[player_2_choice] += 1
    player_2_total += player_2_choice

  if player_1_total > player_2_total:
    winner_choices = player_1_choices
    loser_choices = player_2_choices
  else:
    winner_choices = player_2_choices
    loser_choices = player_1_choices
  return (winner_choices, loser_choices)
```

좋은 결정을 정의하는 방법

- 다 더하기 게임 시뮬레이션을 통한 샘플 출력

```
>>> policy = [0.2, 0.2, 0.2, 0.2, 0.2]
>>> simulate_game(policy)
({1: 22, 2: 17, 3: 16, 4: 27, 5: 18}, {1: 23, 2: 24, 3: 21, 4: 13, 5: 19})
>>> simulate_game(policy)
({1: 14, 2: 25, 3: 21, 4: 19, 5: 21}, {1: 19, 2: 24, 3: 17, 4: 18, 5: 22})
>>> simulate_game(policy)
({1: 13, 2: 20, 3: 21, 4: 25, 5: 21}, {1: 18, 2: 24, 3: 18, 4: 24, 5: 16})
>>> simulate_game(policy)
({1: 17, 2: 20, 3: 21, 4: 20, 5: 22}, {1: 19, 2: 25, 3: 17, 4: 17, 5: 22})
>>> simulate_game(policy)
({1: 21, 2: 21, 3: 21, 4: 18, 5: 19}, {1: 25, 2: 14, 3: 24, 4: 25, 5: 12})
>>>
```

좋은 결정을 정의하는 방법

- 다 더하기 게임 (Add It Up)
  - 샘플 출력에서 이긴 에이전트가 선택한 수와 패한 에이전트가 선택한 수의 개수를 비교해 더 나은 수를 알 수 있다. 이런 차이를 확률적으로 반영하면 정책을 향상시킬 수 있다.
  - 이런 경우 좋은 수는 어느 정도 이상 항상 나오도록 하고, 나쁜 수는 어느 정도 이상 나오지 않도록 설정할 수 있다. 그러면 확률분포가 좀 더 좋은 수가 많이 나오는 쪽으로 천천히 이동할 것이다.

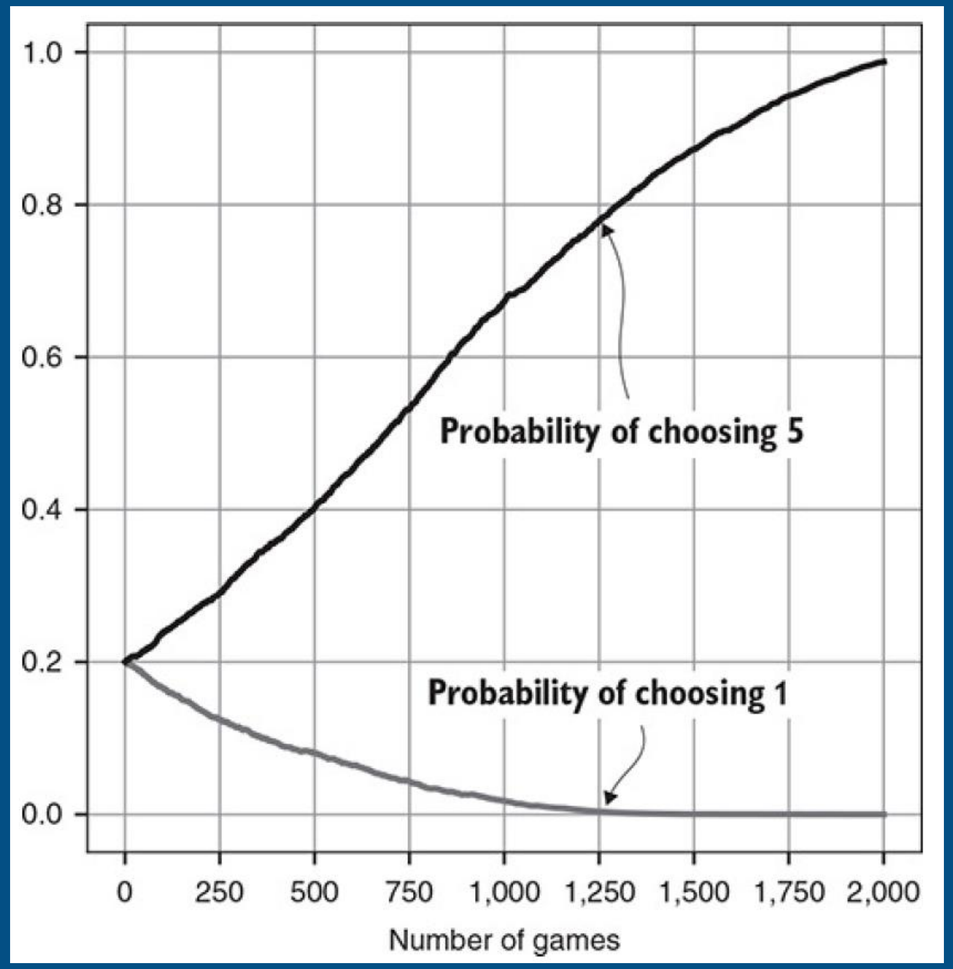
좋은 결정을 정의하는 방법

- 다 더하기 게임의 정책 학습 구현

```
def normalize(policy):
  policy = np.clip(policy, 0, 1)
  return policy / np.sum(policy)

choices = [1, 2, 3, 4, 5]
policy = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
learning_rate = 0.0001

for i in range(num_games):
  win_counts, lose_counts = simulate_game(policy)
  for i, choice in enumerate(choices):
    net_wins = win_counts[choice] - lose_counts[choice]
    policy[i] += learning_rate * net_wins
  policy = normalize(policy)
  print('%d: %s' % (i, policy))
```

좋은 결정을 정의하는 방법

- 다 더하기 게임의 정책 학습 구현



경사하강법을 사용해 정책 수정하기

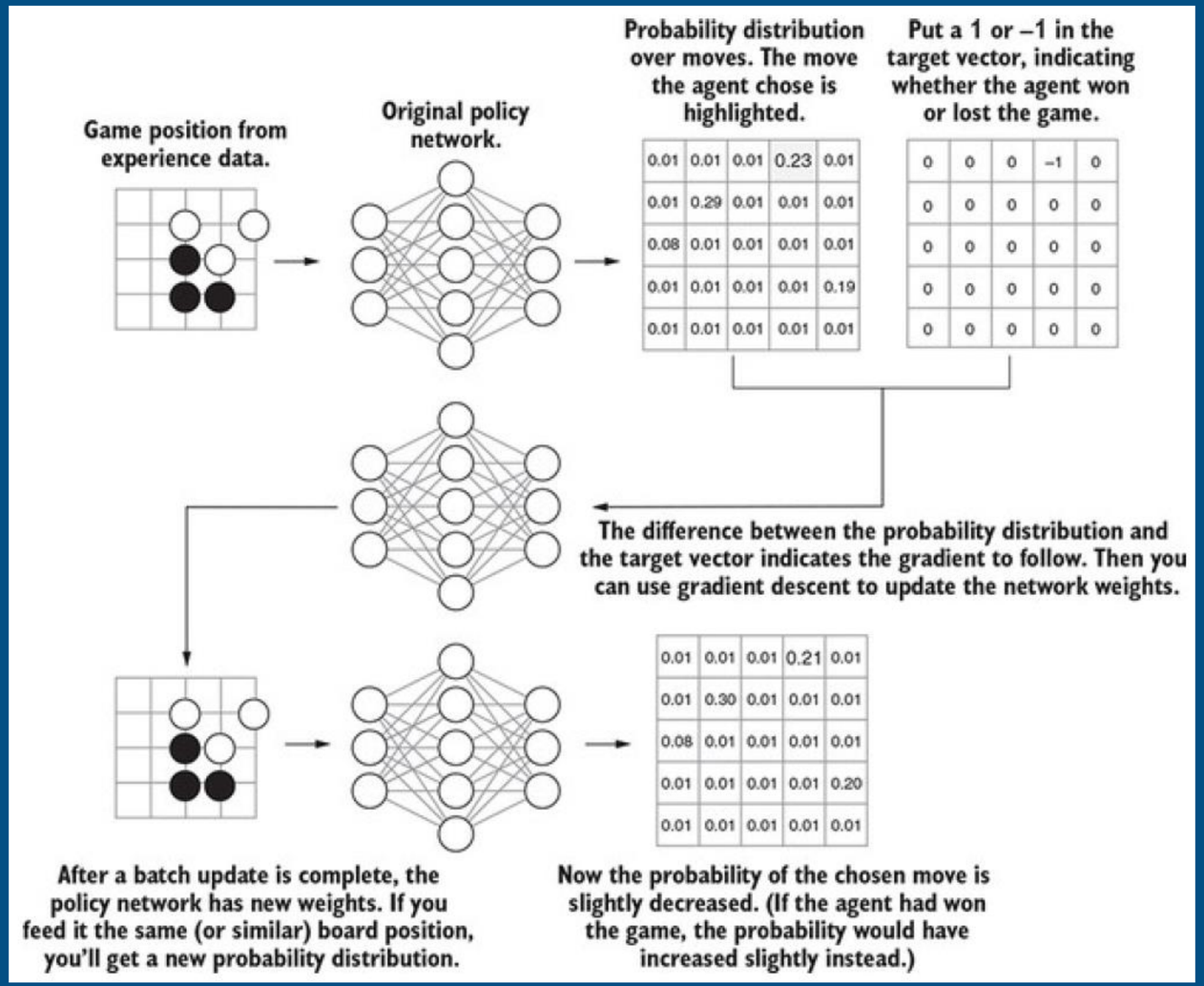
- 다 더하기 경기 학습과 바둑 학습 간에는 한 가지 확연한 차이가 있다.
  - 다 더하기 예제에서 사용했던 정책은 현재 경기 상태와 상관없다는 것이다.
    5는 언제 선택해도 좋은 수고, 1은 언제나 나쁜 수다.
  - 바둑에서 특정 수의 확률을 높인다는 것은 실제로는 **어떤 부류의 상황**에 대해
    특정 수의 확률을 높이는 것이다. 하지만 **어떤 부류의 상황**이라는 것은 매우 모호하다.
    따라서 '**어떤 부류의 상황**'이 무엇을 의미하는지 구분하는 것은 신경망에게 맡길 것이다.

경사하강법을 사용해 정책 수정하기

- 지난 스터디에서 신경망 정책을 만들었을 때 바둑판의 상태를 입력으로 사용해서 수의 확률분포를 출력으로 하는 함수를 만들었다.
  - 실험 데이터의 각 바둑판 상태에 따라 특정 수의 확률이 늘거나 줄었다.
  - 하지만 지난 스터디처럼 정책의 확률을 강제로 수정할 수는 없다.
    대신 원하는 결과가 나오도록 신경망의 가중치를 조절해야 한다. → 경사하강법!
  - 경사하강법으로 정책을 수정하는 것을 **정책 경사 학습(Policy Gradient Learning)** 이라고 한다. 여기서 사용할 학습 알고리즘은 **몬테카를로 정책 경사(Monte-Carlo Policy Gradient)** 또는 **REINFORCE** 방법이라고 한다.

경사하강법을 사용해 정책 수정하기

- 정책 경사 학습 순서도



경사하강법을 사용해 정책 수정하기

- 경험 데이터가 3개의 병렬 배열로 구성되었다는 것을 기억해보자.
  - `states[i]` : 에이전트가 자체 대국 중에 접한 특정 바둑판 위치를 나타낸다.
  - `actions[i]` : 해당 위치에서 에이전트가 선택한 수를 나타낸다.
  - `rewards[i]` : 에이전트가 이겼을 때 1, 졌을 때 -1이다.

경사하강법을 사용해 정책 수정하기

- 경험 데이터를 타깃 벡터로 변환하기

```python
def prepare_experience_data(experience, board_width, board_height):
  experience_size = experience.actions.shape[0]
  target_vectors = np.zeros((experience_size, board_width * board_height))
  for i in range(experience_size):
    action = experience.actions[i]
    reward = experience.rewards[i]
    target_vectors[i][action] = reward
  return target_vectors
```

경사하강법을 사용해 정책 수정하기

- 정책 경사 학습을 사용해서 경험 데이터로 에이전트 훈련하기

```
class PolicyAgent(Agent):
...
  def train(self, experience, lr, clipnorm, batch_size):
    self._model.compile(
      loss='categorical_crossentropy',
      optimizer=SGD(lr=lr, clipnorm=clipnorm))

    target_vectors = prepare_experience_data(
      experience,
      self._encoder.board_width,
      self._encoder.board_height)

    self._model.fit(
      experience.states, target_vectors,
      batch_size=batch_size,
      epochs=1)
```

경사하강법을 사용해 정책 수정하기

- `train()` 함수는 최적화기의 동작을 정하는 3가지 파라미터를 사용한다.
  - `lr` : **학습률**이다. 각 단계에서 가중치를 얼마나 조절할 수 있는지 조정한다.
  - `clipnorm` : 각 단계에서 가중치를 얼마나 조절할 수 있는지 최대치를 제한한다.
  - `batch_size` : 경험 데이터에서 한 번의 가중치 갱신으로 변화할 수 있는 수의 개수를 조절한다.

경사하강법을 사용해 정책 수정하기

- 기존에 저장한 경험 데이터로 훈련하기

```
learning_agent = agent.load_policy_agent(h5py.File(learning_agent_filename))
for exp_filename in experience_files:
  exp_buffer = rl.load_experience(h5py.File(exp_filename))
  learning_agent.train(
    exp_buffer,
    lr=learning_rate,
    clipnorm=clipnorm,
    batch_size=batch_size)

with h5py.File(updated_agent_filename, 'w') as updated_agent_outf:
  learning_agent.serialize(updated_agent_outf)
```

성능 향상 평가하기

- 두 에이전트의 성능 비교 스크립트

```
wins = 0
losses = 0
color1 = Player.black
for i in range(args.num_games):
  print('Simulating game %d/%d...' % (i + 1, args.num_games))
  if color1 == Player.black:
    black_player, white_player = agent1, agent2
  else:
    white_player, black_player = agent1, agent2
  game_record = simulate_game(black_player, white_player)
  if game_record.winner == color1:
    wins += 1
  else:
    losses += 1
  color1 = color1.other
print('Agent 1 record: %d/%d' % (wins, wins + losses))
```

감사합니다!

스터디 듣느라 고생 많았습니다.