KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- **행위자-비평가 학습(Actor-Critic Learning)**
  - (11주차에서 배운) 정책 학습 + (오늘 배운) 가치 학습
  - 정책 학습은 **행위자(Actor)** 역할을 한다. (즉, 어떤 수를 둘 지 정한다.)
  - 가치 함수는 **비평가(Critic)** 역할을 한다. (즉, 에이전트가 경기에서 우세인지 열세인지 살펴본다. 경기를 복기해 학습을 하는 것처럼 이 피드백을 훈련 과정에 적용한다.)
- **어드밴티지(Advantage)**
  - 실제 경기 결과와 예상 경기 결과의 차잇값

어드밴티지

- 어드밴티지란 무엇인가
  - 농구 경기를 보고 있다고 하자. 4쿼터가 끝나가는 시점에서 가장 좋아하는 선수가 3점 슛을 넣었다. 얼마나 흥분되는 일인가? 이는 경기 상태에 따라 달라진다.
  - 만약 현재 점수가 80 대 78이면 놀라서 나도 모르게 자리에서 벌떡 일어날 것이다. 하지만 점수가 110 대 80이라면 큰 감흥이 없을 것이다. 여기에는 어떤 차이가 있을까?
  - 접전을 치르는 경기라면 3점은 경기 결과에 매우 큰 변화를 불러올 수 있다. 하지만 경기가 어느 한쪽이 압도적으로 이기고 있다면 한 번 득점한 것으로는 결과에 영향을 미치지 못한다. 가장 중요한 승부는 결과가 팽팽할 때 생기는 법이다.
  - 강화학습에서 어드밴티지는 이 개념을 수량화한 식이다.

어드밴티지

- 어드밴티지란 무엇인가
  - 어드밴티지를 구하려면 가장 먼저 추정 상탯값 $V(s)$가 필요하다.
    에이전트가 보게 될 예상값으로, 특정 상태 s에 따라서 주어진다.
  - 바둑의 경우 $V(s)$는 바둑판 상태가 흑 또는 백 어느 쪽에 유리한지 나타낸다고 보면 된다.
  - $V(s)$가 1에 가까운 경우 에이전트가 앞서고 있다는 뜻이고,
    -1에 가까운 경우 에이전트가 지고 있다는 뜻이다.

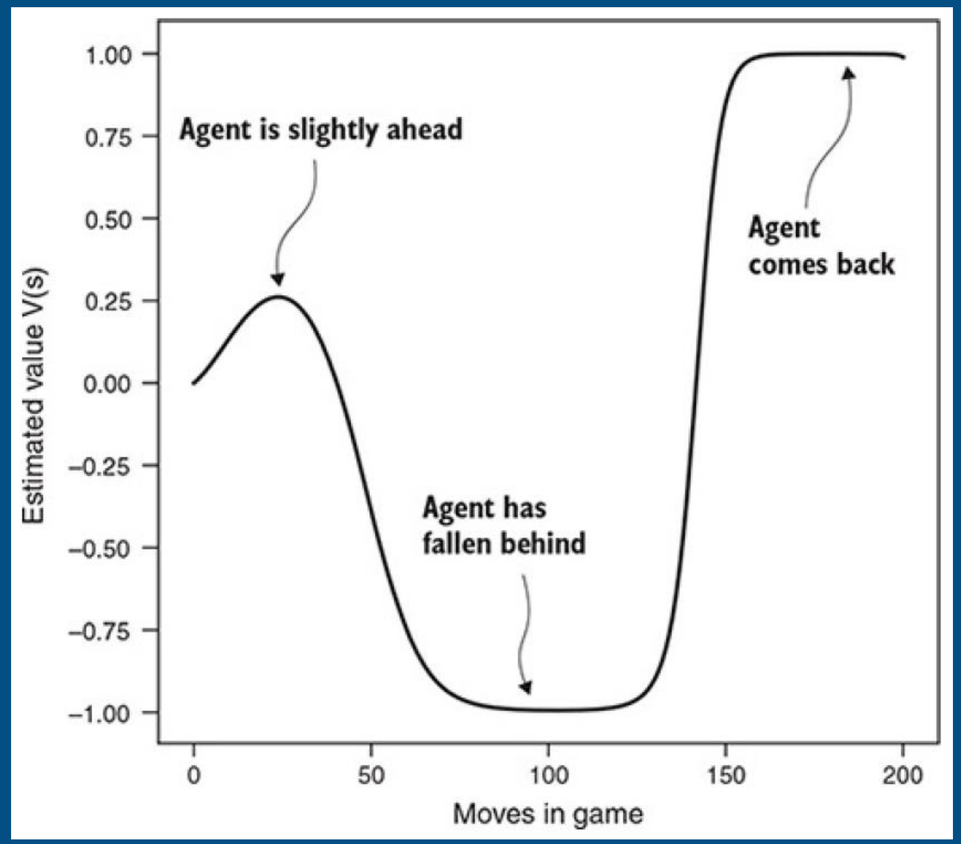
어드밴티지

- 어드밴티지란 무엇인가
  - $V(s)$는 앞에서 배운 행동-가치 함수 $Q(s,a)$의 개념과 유사하다.
    차이점은 $V(s)$는 수를 선택하기 **전**에 현재 바둑판의 상태가 얼마나 유리한지 나타내고,
    $\mathrm{Q}(s,a)$는 수를 선택한 **후** 바둑판 상태가 얼마나 유리하게 되었는지 나타낸다.
  - 어드밴티지의 정의 : $A = Q(s,a) - V(s)$
    - 예를 들어, 지금 상태가 좋은데($V(s)$가 높음) 수를 잘못 선택하면($\mathrm{Q}(s,a)$가 낮음) 어드밴티지를 포기해야 한다.
  - 문제는 $Q(s,a)$를 어떻게 구하는지 모른다는 거다. 하지만 대국 종료 시 받는 승점을
    실제 Q의 비편향 추정치로 사용할 수 있다. 그러면 승점 R을 받을 때까지 기다린 후
    다음과 같이 어드밴티지를 구할 수 있다.

$$A = R - V(s)$$

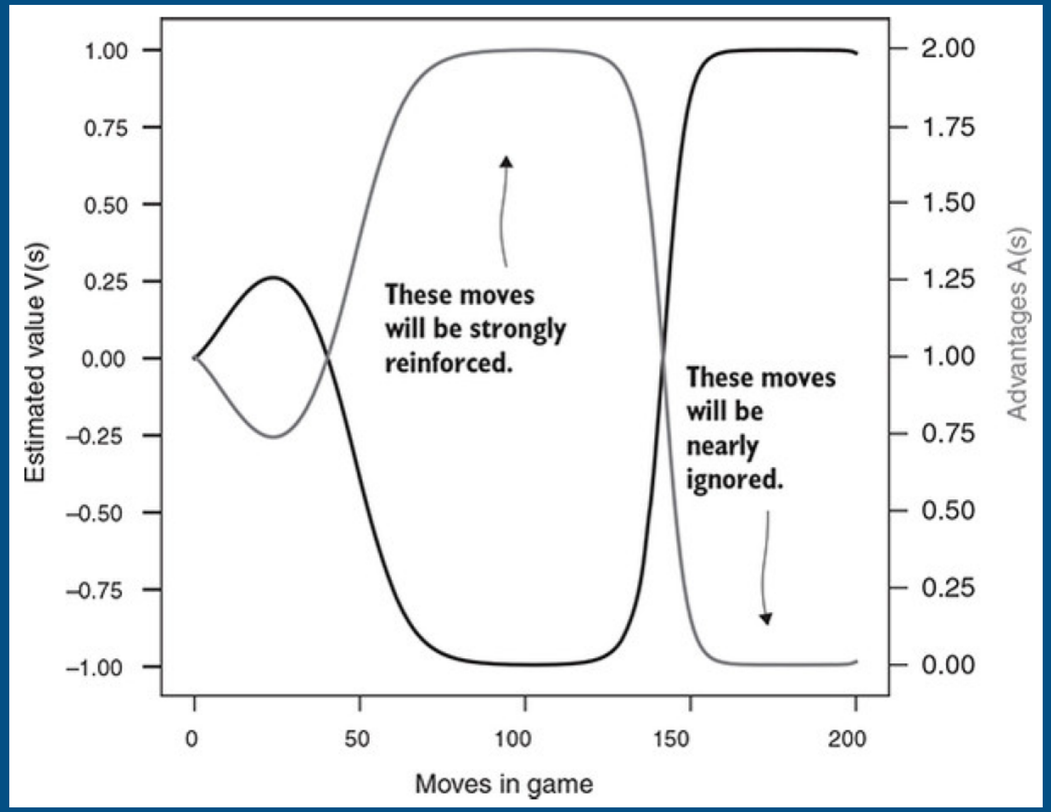
어드밴티지

- 어드밴티지란 무엇인가
  - 가상의 경기에서의 예상 가치



어드밴티지

- 어드밴티지란 무엇인가
  - 가상의 경기에서의 각 수의 우셋값



어드밴티지

- 자체 대국 중에 어드밴티지 구하기
  - 어드밴티지 기록을 위한 `ExperienceCollector` 갱신

```
class ExperienceCollector(object):
  def __init__(self):
    self.states = []
    self.actions = []
    self.rewards = []
    self.advantages = []
    self._current_episode_states = []
    self._current_episode_actions = []
    self._current_episode_estimated_values = []
```

어드밴티지

- 자체 대국 중에 어드밴티지 구하기
  - 추정치를 저장하도록 `ExperienceCollector` 갱신

```python
class ExperienceCollector(object):
  def record_decision(self, state, action, estimated_value=0):
    self._current_episode_states.append(state)
    self._current_episode_actions.append(action)
    self._current_episode_estimated_values.append(estimated_value)
```

어드밴티지

- 자체 대국 중에 어드밴티지 구하기
  - 에피소드 끝에서 어드밴티지 구하기

```
class ExperienceCollector(object):
  def complete_episode(self, reward):
    num_states = len(self._current_episode_states)
    self.states += self._current_episode_states
    self.actions += self._current_episode_actions
    self.rewards += [reward for _ in range(num_states)]

    for i in range(num_states):
      advantage = reward - \
        self._current_episode_estimated_values[i]
      self.advantages.append(advantage)

    self._current_episode_states = []
    self._current_episode_actions = []
    self._current_episode_estimated_values = []
```

어드밴티지

- 자체 대국 중에 어드밴티지 구하기
  - `ExperienceBuffer` 구조에 어드밴티지 추가

```
class ExperienceBuffer(object):
  def __init__(self, states, actions, rewards, advantages):
    self.states = states
    self.actions = actions
    self.rewards = rewards
    self.advantages = advantages

  def serialize(self, h5file):
    h5file.create_group('experience')
    h5file['experience'].create_dataset('states', data=self.states)
    h5file['experience'].create_dataset('actions', data=self.actions)
    h5file['experience'].create_dataset('rewards', data=self.rewards)
    h5file['experience'].create_dataset('advantages', data=self.advantages)
```

어드밴티지

- 자체 대국 중에 어드밴티지 구하기
  - `ExperienceBuffer` 구조에 어드밴티지 추가

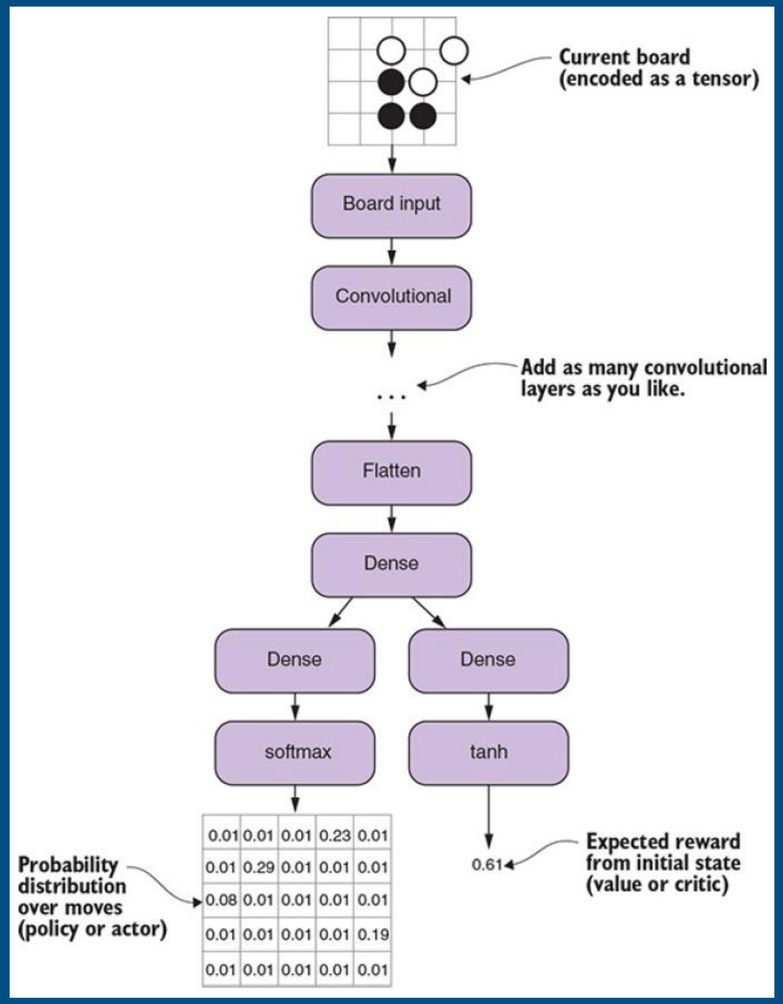
```
class ExperienceBuffer(object):
  def combine_experience(collectors):
    combined_states = np.concatenate(
      [np.array(c.states) for c in collectors])
    combined_actions = np.concatenate(
      [np.array(c.actions) for c in collectors])
    combined_rewards = np.concatenate(
      [np.array(c.rewards) for c in collectors])
    combined_advantages = np.concatenate([
      np.array(c.advantages) for c in collectors])

    return ExperienceBuffer(
      combined_states,
      combined_actions,
      combined_rewards,
      combined_advantages)
```

행위자-비평가 신경망 설계

- Q-학습 신경망 : 입력이 둘인 신경망
  - 바둑판에 대한 입력값 하나
  - 제안된 수에 대한 다른 입력값 하나
- 행위자-비평가 신경망 : 입력값 하나와 출력값 두 개를 갖는 신경망
  - 입력값 : 바둑판 상태를 표현
  - 출력값 1 : 수(행위자)의 확률분포
  - 출력값 2 : 현재 위치의 예상 반환값(비평)의 확률분포

행위자-비평가 신경망 설계

- 바둑용 행위자-비평가 학습 신경망



행위자-비평가 신경망 설계

- 정책과 가치 출력값을 출력하는 신경망

```
from tensorflow.keras.layers import Conv2D, Dense, Flatten, Input
from tensorflow.keras.models import Model

board_input = Input(shape=encoder.shape(), name='board_input')

conv1 = Conv2D(64, (3, 3), padding='same', activation='relu')(board_input)
conv2 = Conv2D(64, (3, 3), padding='same', activation='relu')(conv1)
conv3 = Conv2D(64, (3, 3), padding='same', activation='relu')(conv2)

flat = Flatten()(conv3)
processed_board = Dense(512)(flat)

policy_hidden_layer = Dense(512, activation='relu')(processed_board)
policy_output = Dense(encoder.num_points(), activation='softmax')(policy_hidden_layer)

value_hidden_layer = Dense(512, activation='relu')(processed_board)
value_output = Dense(1, activation='tanh')(value_hidden_layer)

model = Model(inputs=board_input, outputs=[policy_output, value_output])
```

행위자-비평가를 사용한 대국

- 행위자-비평가 에이전트로 수 선택하기

```python
class ACAgent(Agent):
  def select_move(self, game_state):
    num_moves = self.encoder.board_width * self.encoder.board_height

    board_tensor = self.encoder.encode(game_state)
    X = np.array([board_tensor])

    actions, values = self.model.predict(X)
    move_probs = actions[0]
    estimated_value = values[0][0]

    eps = 1e-6
    move_probs = np.clip(move_probs, eps, 1 - eps)
    move_probs = move_probs / np.sum(move_probs)
```

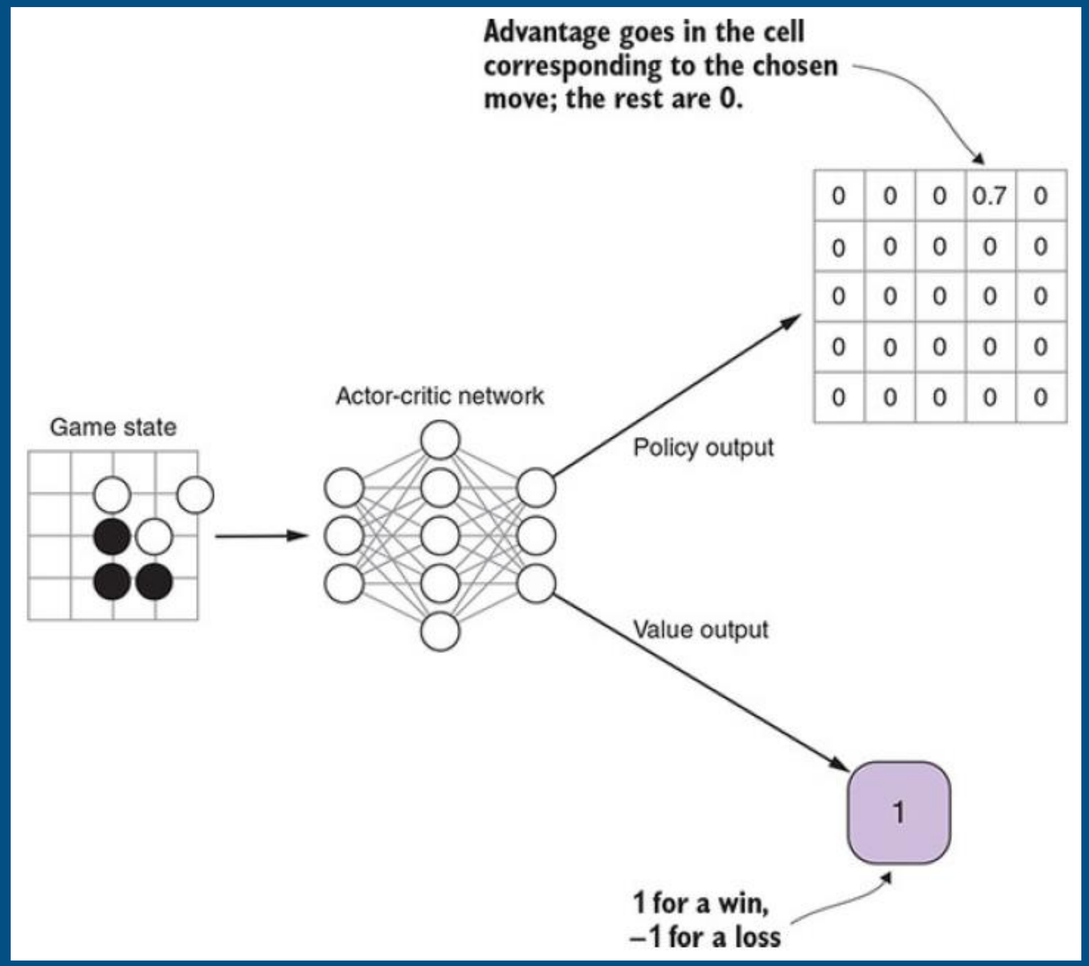
행위자-비평가를 사용한 대국

- 행위자-비평가 에이전트로 수 선택하기

```
    candidates = np.arange(num_moves)
    ranked_moves = np.random.choice(
      candidates, num_moves, replace=False, p=move_probs)
    for point_idx in ranked_moves:
      point = self.encoder.decode_point_index(point_idx)
      move = goboard.Move.play(point)
      move_is_valid = game_state.is_valid_move(move)
      fills_own_eye = is_point_an_eye(
        game_state.board, point,
        game_state.next_player)
      if move_is_valid and (not fills_own_eye):
        if self.collector is not None:
            self.collector.record_decision(
              state=board_tensor,
              action=point_idx,
              estimated_value=estimated_value
            )
        return goboard.Move.play(point)
    return goboard.Move.pass_turn()
```

행위자-비평가 에이전트 훈련

- 행위자-비평가 학습에 대한 훈련 과정



행위자-비평가 에이전트 훈련

- 행위자-비평가 에이전트로 수 선택하기

```
class ACAgent(Agent):
  def train(self, experience, lr=0.1, batch_size=128):
    opt = SGD(lr=lr)
    self.model.compile(
      optimizer=opt,
      loss=['categorical_crossentropy', 'mse'],
      loss_weights=[1.0, 0.5])

    n = experience.states.shape[0]
    num_moves = self.encoder.num_points()
    policy_target = np.zeros((n, num_moves))
    value_target = np.zeros((n,))
    for i in range(n):
      action = experience.actions[i]
      policy_target[i][action] = experience.advantages[i]
      reward = experience.rewards[i]
      value_target[i] = reward

    self.model.fit(
      experience.states,
      [policy_target, value_target],
      batch_size=batch_size,
      epochs=1)
```

감사합니다!

스터디 듣느라 고생 많았습니다.