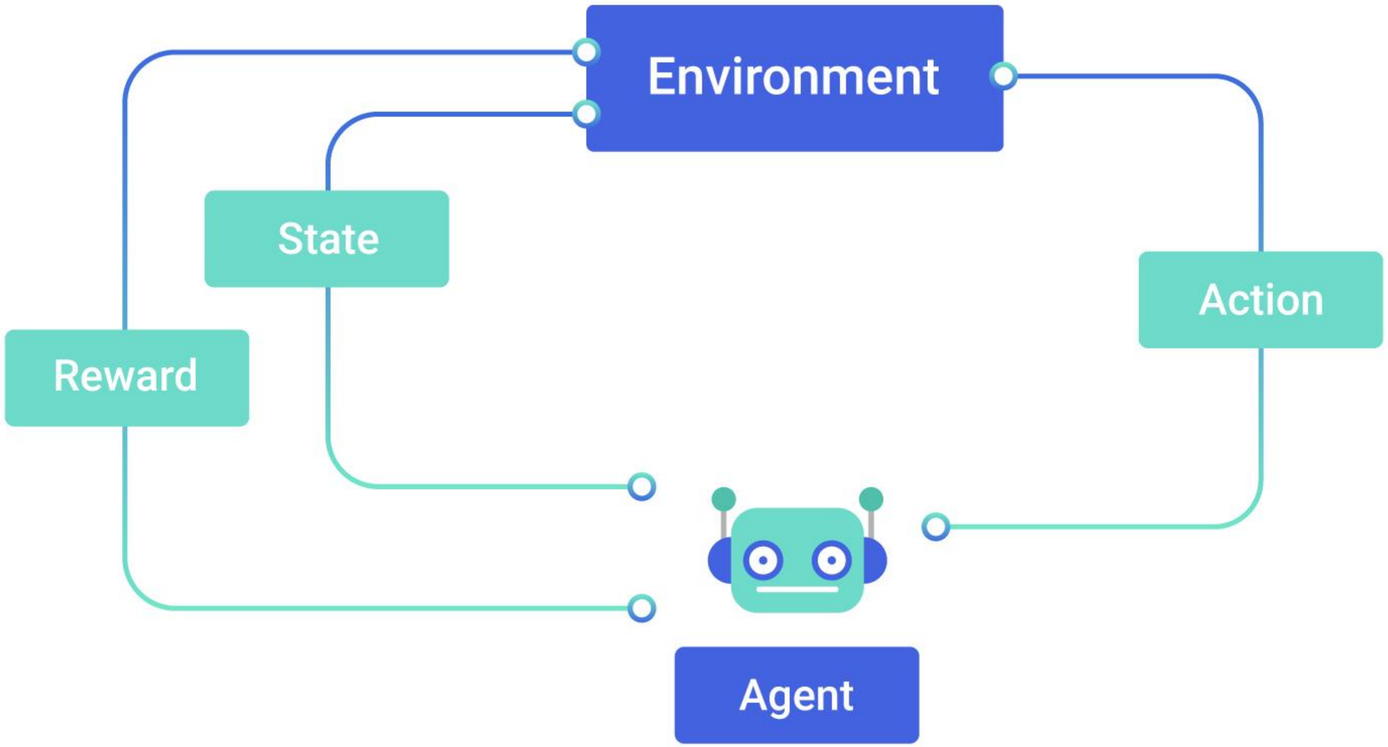
KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- 강화학습이란



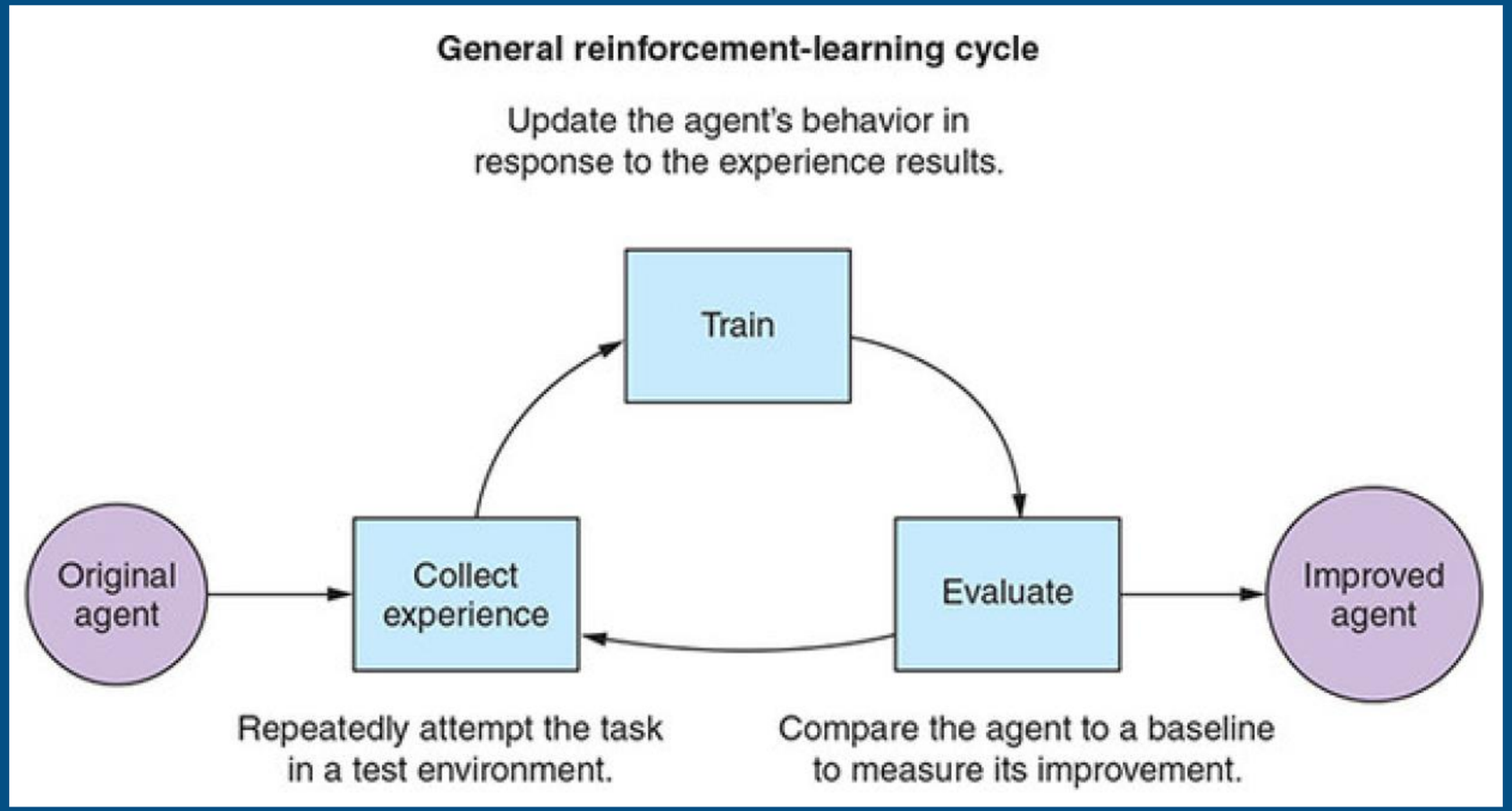
https://perfectial.com/blog/reinforcement-learning-applications/

강화학습 주기

- 바둑봇 : 작업을 수행하는 결정을 내리는 프로그램 → **에이전트(Agent)**
- 강화학습의 목표 : 에이전트의 성능을 최대한 좋게 만드는 것!
  - 즉, 에이전트가 바둑에서 이기게 하는 것이다. 우선 바둑봇이 자체 대국을 수회 두게 한다. 이때 각 대국의 모든 차례와 결과를 기록해둬야 한다. → **경험(Experience)**
  - 그리고 자체 대국에서 일어난 결과에 따라 행동을 갱신하도록 봇을 훈련시킨다. 여기서 핵심 개념은 봇이 이긴 경기에서 했던 결정은 반복하고, 진 경기에서 했던 결정은 하지 않도록 하는 것이다. → 훈련 알고리즘
    - 정책 경사 알고리즘(Policy Gradient Algorithm)
    - 큐러닝(Q-Learning)
    - 행위자-비평가 알고리즘 (Actor-Critic Algorithm)

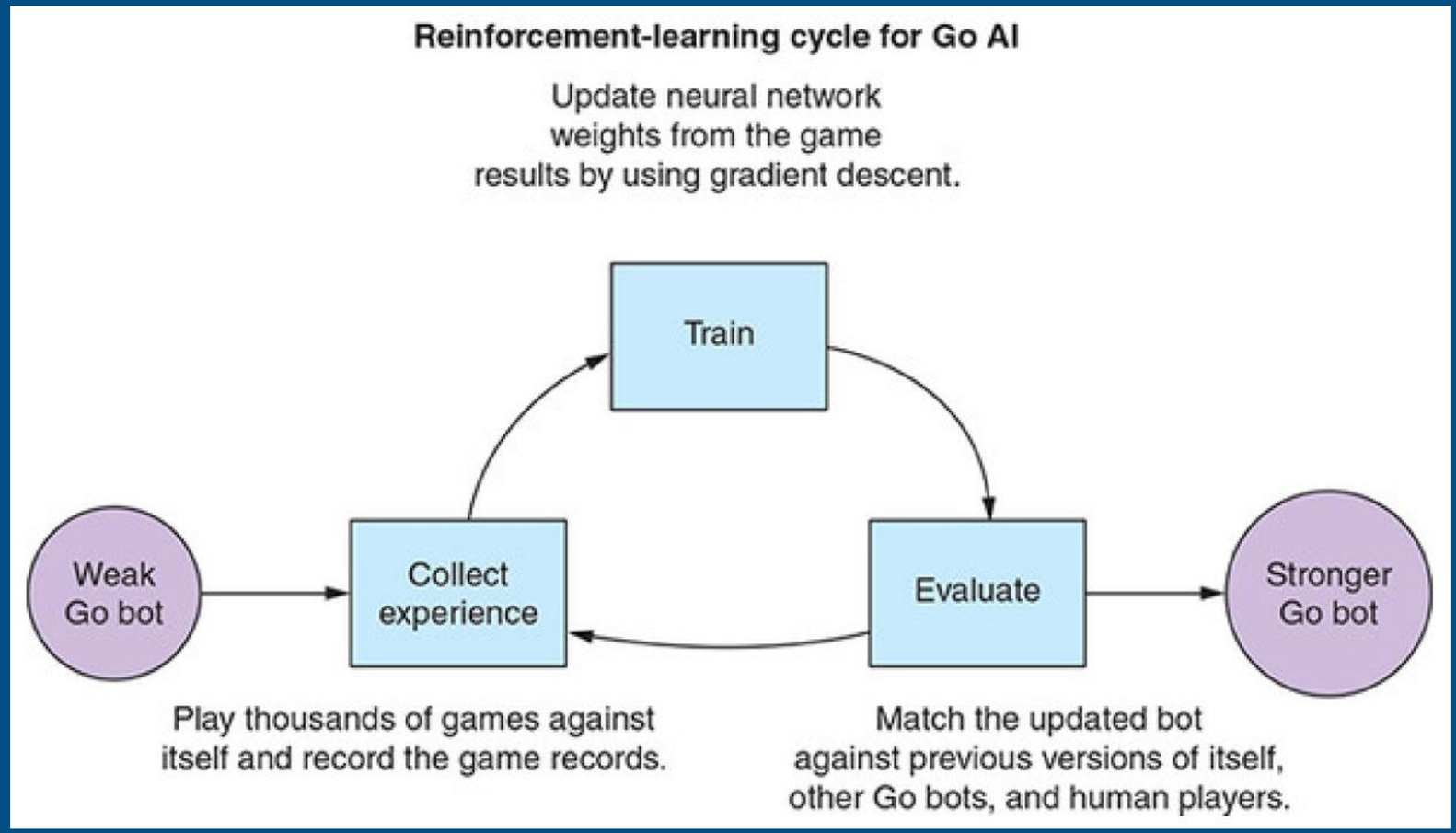
강화학습 주기

- 일반적인 강화학습 주기



강화학습 주기

- 바둑 AI의 강화학습 주기



경험을 통해 어떻게 달라질까

- 게임에서는 경험치를 개별 게임이나 **에피소드(Episode)**로 나눌 수 있다.
  - 에피소드는 명확히 끝이 있고, 한 에피소드 내에서 내린 결정은
    그 다음 에피소드에 영향을 미치지 않는다.
  - 에피소드가 있는 경우 에이전트는 각 환경에 대한 **상태(State)**를 처리할 수 있다.
  - 에이전트는 현재 상태 기반으로 다음 **행동(Action)**을 결정한다.
  - 다음 상태는 선택한 행동과 환경 변화 모두에 영향을 받는다.
  - 바둑의 경우 AI는 바둑판 현황(상태)을 보고, 맞는 수(행동)를 선택한다.
    그러면 다음 차례에 AI가 새로 바뀐 바둑판 현황(다음 상태)을 접하게 될 것이다.

경험을 통해 어떻게 달라질까

- 게임에서는 경험치를 개별 게임이나 **에피소드(Episode)**로 나눌 수 있다.
  - 에이전트가 다음 행동을 선택했을 때 다음 상탯값에는 상대방 수까지 포함된다는 걸 기억해야 한다. 현재 상태와 행동으로 다음 값을 선택할 수 없으므로 상대방 수가 나올 때까지 기다려야 한다. 상대방의 행동은 에이전트가 탐색하기 위해 학습해야 하는 **환경(Environment)** 요인 중 하나다.
  - 탐색 능력을 향상시키려면 에이전트가 목표를 달성했는지 피드백을 주어야 한다. 목표를 달성했는지 여부에 따라 계산한 점수로 피드백하는 **보상(Reward)**을 사용할 수 있다. 이 바둑의 목표는 경기에서 이기는 것이므로 매번 이길 때마다 1점을 주고, 질 때마다 -1점을 주는 식으로 진행한다.

경험을 통해 어떻게 달라질까

- 게임에서는 경험치를 개별 게임이나 **에피소드(Episode)**로 나눌 수 있다.
  - 보상은 게임 종료 후에 한 번에 받는 경우도 있고, 행동에 따라 조금씩 받는 경우도 있다.
    예를 들어, 바둑과 오델로를 비교해 보자.
  - 강화학습의 핵심 개념은 행동이 한참 후에 받을 보상에도 영향을 미친다는 점이다.
    바둑에서 35번째 강수를 두어서 200수 후에 이기게 되는 경우를 생각해 보자.
    초기에 잘 둔 수가 경기 승리 후 보상으로 나타날 것이다.
    또한 실수로 전체 경기의 보상을 날려버릴 수도 있다.
    에이전트가 행동 후에 받게 되는 보상을 행동의 **반환값(Return)**이라고 한다.

경험을 통해 어떻게 달라질까

- 행동의 반환값 구하기

```
for exp_idx in range(exp_length):
  total_return[exp_idx] = reward[exp_idx]
  for future_reward_idx in range(exp_idx + 1, exp_length):
    total_return[exp_idx] += reward[future_reward_idx]
```

경험을 통해 어떻게 달라질까

- 행동의 반환값에 대해 다시 생각해 보자.
  - 오델로 예제에서 첫번째 차례에 결정한 것은 세번째 차례의 점수에 많은 영향을 미칠 것이고, 혹은 어느 정도까지는 그럭저럭 영향을 미칠 수도 있다.
  - 하지만 세번째 차례의 결정이 20번째 결정에 얼마나 영향을 미칠지는 파악하기 어렵다.
  - 이를 반환값 계산의 개념으로 나타내려면 매 행동에 대한 보상의 가중합을 구해야 한다. 가중치는 행동이 멀어질수록 작아져야 먼 미래의 보상이 직전의 보상보다 적게 영향을 미친다. 이를 **보상 감소(Discounting)**라고 한다.

경험을 통해 어떻게 달라질까

- 보상 감소 구하기

```
for exp_idx in range(exp_length):
  discounted_return[exp_idx] = reward[exp_idx]
  discount_amount = 0.75
  for future_reward_idx in range(exp_idx + 1, exp_length):
    discounted_return[exp_idx] +=
      discount_amount * reward[future_reward_idx]
    discount_amount *= 0.75
```

학습 가능한 에이전트 만들기

- **정책(Policy)**은 주어진 상태에서 행동을 선택하는 함수다.
  - 이전에 `select_move()` 함수를 사용하는 Agent 클래스를 여러 가지로 구현했다.
    여기서 각 `select_move()` 함수가 정책이다. 경기 상태를 입력받고 수를 출력한다.
  - 강화학습을 사용하는 경우에는 다른 컴퓨터 프로그램을 사용해서 정책을 자동으로
    갱신해야 한다. 이전에 합성곱 신경망을 배우면서 이를 배웠었다.
  - 앞에서 만들었던 수 예측 신경망은 바둑판의 각 점에 대한 값을 벡터로 출력한다.
    이 값은 다음 수에 대한 신경망의 신뢰도를 나타낸다. 이 결과로 어떻게 정책을 구성할까?

학습 가능한 에이전트 만들기

- **정책(Policy)**은 주어진 상태에서 행동을 선택하는 함수다.
  - 한 가지 방법은 단순히 가장 높은 값의 수를 선택하는 것이다.
    네트워크가 좋은 수를 선택하도록 잘 훈련되어 있다면 좋은 결과를 낼 것이다.
    하지만 어떤 바둑판 상태가 주어지든 동일한 수를 선택한다.
  - 강화학습을 통해 이런 문제를 강화하고 싶다면 다양한 수를 선택해야 한다.
    어떤 경우에는 더 나아질 것이고, 어떤 경우에는 더 엉망이 될 것이다.
    이때 나오는 결과를 보고 어떤 수가 좋은 수인지 알아챌 수 있다.
    하지만 성능 향상을 위해서는 다양성이 필요하다.

학습 가능한 에이전트 만들기

- **정책(Policy)**은 주어진 상태에서 행동을 선택하는 함수다.
  - 항상 가장 높은 수치의 수를 선택하는 대신 확률적 정책을 만들어보자.
    여기서 '**확률적(Stochastic)**'은 동일한 바둑판 상태가 두 번 주어졌을 때
    에이전트가 다른 수를 선택하도록 하는 것을 의미한다.
    여기에는 임의성이 포함되지만 전에 구현했던 `RandomAgent` 같은 방식은 아니다.
  - `RandomAgent`는 경기 중에 어떤 일이 일어났는지 전혀 고려하지 않고 수를 선택한다.
    확률적 정책이란 수 선택은 바둑판 상태에 기반하지만 100% 신뢰할 수는 없다는 것이다.

학습 가능한 에이전트 만들기

- 확률분포에 따른 샘플링
  - 예를 들어 가위바위보를 한다고 했을 때 경기 중 50%는 바위를 내고,
    30%는 보를 내고, 20%는 가위를 내는 정책을 따를 수 있다.
    이는 세 선택지에 대한 **확률분포(Probability Distribution)**다.
  - 이때 확률값의 합은 100%여야 한다. 그래야 정책이 항상 선택지 중 하나를 고를 수 있다.
    선택지를 임의로 선택하는 과정을 확률분포에 따른 **샘플링(Sampling)**이라고 한다.

학습 가능한 에이전트 만들기

- 확률분포에 따른 샘플링

```python
import random

def rps():
  randval = random.random()
  if 0.0 <= randval < 0.5:
    return 'rock'
  elif 0.5 <= randval < 0.8:
    return 'paper'
  else:
    return 'scissors'
```

학습 가능한 에이전트 만들기

- 확률분포에 따른 샘플링
  - numpy로 구현한 확률분포 샘플링

```python
import numpy as np

def rps():
  return np.random.choice(
    ['rock', 'paper', 'scissors'],
    p=[0.5, 0.3, 0.2])
```

학습 가능한 에이전트 만들기

- 확률분포에 따른 샘플링
  - numpy를 사용한 확률분포 반복 샘플링

```python
import numpy as np

def repeated_rps():
  return np.random.choice(
    ['rock', 'paper', 'scissors'],
    size=3,
    replace=False,
    p=[0.5, 0.3, 0.2])
```

학습 가능한 에이전트 만들기

- 확률분포 제한
  - 강화학습 과정은 초반에는 꽤 불안정하다. 에이전트는 일부 승리에 대해 과하게 반응하고, 그다지 좋지 않은 수에 임의로 높은 확률을 부여하기도 한다. (초보 선수도 마찬가지)
    확률이 특정 수에 1을 몰아주기도 한다. 이런 경우는 다소 문제가 될 수 있다.
    에이전트는 항상 동일한 수를 선택할 것이고, 다시 돌아갈 방법도 없기 때문이다.
  - 이를 방지하려면 확률 분포를 제한해 확률이 0 또는 1이 되지 않도록 해야 한다.
    여기서는 numpy의 `np.clip()` 함수를 사용하면 된다.

학습 가능한 에이전트 만들기

- 확률분포 제한

```python
def clip_probs(original_probs):
  min_p = 1e-5
  max_p = 1 - min_p
  clipped_probs = np.clip(original_probs, min_p, max_p)
  clipped_probs = clipped_probs / np.sum(clipped_probs)
  return clipped_probs
```

학습 가능한 에이전트 만들기

- 에이전트 초기화
  - 확률 정책에 따라 수를 선택하고 경험 데이터를 통해 학습하는 새로운 형태의 에이전트인 `PolicyAgent`를 만들어보자.

```python
class PolicyAgent(Agent):
  def __init__(self, model, encoder):
    self._model = model
    self._encoder = encoder
```

학습 가능한 에이전트 만들기

- 에이전트 초기화
  - 새 학습 에이전트 생성하기

```
encoder = encoders.simple.SimpleEncoder((board_size, board_size))
model = Sequential()
for layer in dlgo.networks.large.layers(encoder.shape()):
  model.add(layer)
model.add(Dense(encoder.num_points()))
model.add(Activation('softmax'))
new_agent = agent.PolicyAgent(model, encoder)
```

학습 가능한 에이전트 만들기

- 물리 장치로부터 에이전트 불러오고 저장하기
  - PolicyAgent를 디스크에 기록하기

```python
def serialize(self, h5file):
  h5file.create_group('encoder')
  h5file['encoder'].attrs['name'] = self._encoder.name()
  h5file['encoder'].attrs['board_width'] = self._encoder.board_width
  h5file['encoder'].attrs['board_height'] = self._encoder.board_height
  h5file.create_group('model')
  kerasutil.save_model_to_hdf5_group(self._model, h5file['model'])
```

학습 가능한 에이전트 만들기

- 물리 장치로부터 에이전트 불러오고 저장하기
  - `serialize()` 함수 용례

```python
import h5py
with h5py.File(output_file, 'w') as outf:
  agent.serialize(outf)
```

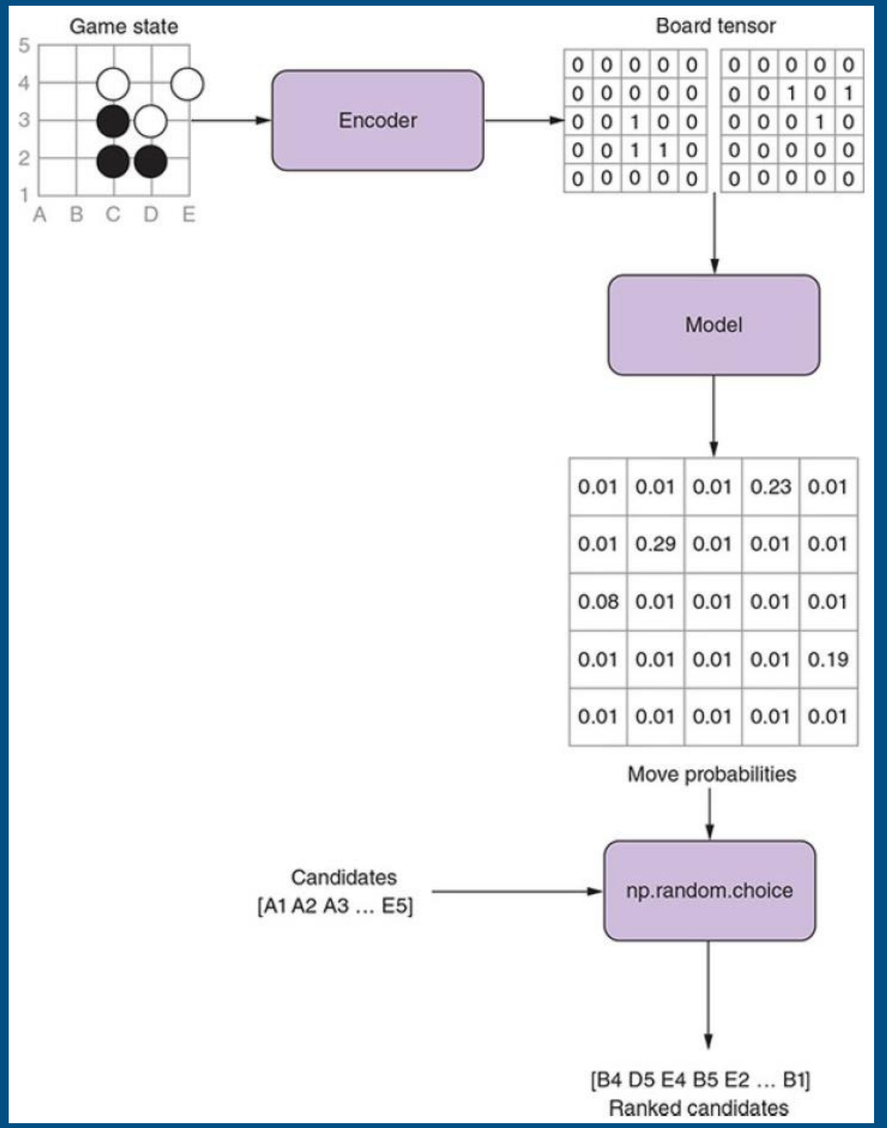
학습 가능한 에이전트 만들기

- 물리 장치로부터 에이전트 불러오고 저장하기
  - 파일에서 정책 에이전트 불러오기

```python
def load_policy_agent(h5file):
  model = kerasutil.load_model_from_hdf5_group(h5file['model'])
  encoder_name = h5file['encoder'].attrs['name']
  board_width = h5file['encoder'].attrs['board_width']
  board_height = h5file['encoder'].attrs['board_height']
  encoder = encoders.get_encoder_by_name(
    encoder_name,
    (board_width, board_height))
  return PolicyAgent(model, encoder)
```

학습 가능한 에이전트 만들기

- 수 선택 구현
  - 수 선택 과정
    - 경기 상태를 수치 텐서로 변환한다.
    - 텐서를 모델에 넣어 수 확률을 구한다.
    - 바둑판의 모든 점으로부터 다음에 시도할 수의 확률에 대해 순서를 매긴 후 상위권을 가져온다.



학습 가능한 에이전트 만들기

- 수 선택 구현
  - 신경망을 사용해서 수 선택하기

```python
def select_move(self, game_state):
  board_tensor = self._encoder.encode(game_state)
  X = np.array([board_tensor])
  move_probs = self._model.predict(X)[0]
  move_probs = clip_probs(move_probs)
  num_moves = self._encoder.board_width * self._encoder.board_height
  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 = gobard.Move.play(point)
    is_valid = game_state.is_valid_move(move)
    is_an_eye = is_point_an_eye(game_state.board, point, game_state.next_player)
    if is_valid and (not is_an_eye):
      return goboard.Move.pass_turn()
  return goboard.Move.pass_turn()
```

자체 대국

- 경험 데이터 나타내기
  - 경험 버퍼 생성자

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

자체 대국

- 경험 데이터 나타내기
  - 경험 버퍼를 디스크에 저장하기

```
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)
```

자체 대국

- 경험 데이터 나타내기
  - HDF5 파일에서 `ExperienceBuffer` 꺼내오기

```
def load_experience(h5file):
  return ExperienceBuffer(
    states=np.array(h5file['experience']['states']),
    actions=np.array(h5file['experience']['actions']),
    rewards=np.array(h5file['experience']['rewards']))
```

자체 대국

- 경험 데이터 나타내기
  - 단일 에피소드 상의 결정을 추적하는 객체

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

  def begin_episode(self):
    self._current_episode_states = []
    self._current_episode_actions = []

  def record_decision(self, state, action):
    self._current_episode_states.append(state)
    self._current_episode_actions.append(action)
```

자체 대국

- 경험 데이터 나타내기
  - 단일 에피소드 상의 결정을 추적하는 객체

```
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)]

    self._current_episode_states = []
    self._current_episode_actions = []

  def to_buffer(self):
    return ExperiencedBuffer(
      states=np.array(self.states),
      actions=np.array(self.actions),
      rewards=np.array(self.rewards))
```

자체 대국

- 경험 데이터 나타내기
  - `ExperienceCollector`의 4가지 메소드
    - `begin_episode()`와 `complete_episode()` : 단일 경기의 시작과 끝을 나타내는 자체 대국 드라이버다.
    - `record_decision()` : 에이전트가 선택한 행동을 나타내기 위해 호출한다.
    - `to_buffer()` : `ExperienceCollector`가 기록한 모든 내용을 묶어서 `ExperienceBuffer`로 반환해준다. 자체 대국 드라이버는 자체 대국 세션 끝에서 이 메소드를 호출한다.

자체 대국

- 경험 데이터 나타내기
  - ExperienceCollector와 PolicyAgent 결합하기

```
class PolicyAgent(Agent):
...
  def set_collector(self, collector):
    self._collector = collector
...
  def select_move(self, game_state):
...
    if self.collector is not None:
      self.collector.record_decision(
        state=board_tensor,
        action=point_idx)
    return gobard.Move.play(point)
```

자체 대국

- 대국 시뮬레이션
  - 두 에이전트 간의 대국 시뮬레이션

```
def simulate_game(black_player, white_player):
  game = GameState.new_game(BOARD_SIZE)
  agents = {
    Player.black: black_player,
    Player.white: white_player,
  }
  while not game.is_over():
    next_move = agents[game.next_player].select_move(game)
    game = game.apply_move(next_move)
  game_result = scoring.compute_game_result(game)
  return game_result.winner
```

자체 대국

- 대국 시뮬레이션
  - 경험치 배치 생성 초기화

```
agent1 = agent.load_policy_agent(h5py.File(agent_filename))
agent2 = agent.load_policy_agent(h5py.File(agent_filename))
collector1 = rl.ExperienceCollector()
collector2 = rl.ExperienceCollector()
agent1.set_collector(collector1)
agent2.set_collector(collector2)
```

자체 대국

- 대국 시뮬레이션
  - 대국 배치 실행

```python
for i in range(args.num_games):
  collector1.begin_episode()
  collector2.begin_episode()

  game_record = simulate_game(agent1, agent2)
  if game_record.winner == Player.black:
    collector1.complete_episode(reward=1)
    collector2.complete_episode(reward=-1)
  else:
    collector2.complete_episode(reward=1)
    collector1.complete_episode(reward=-1)
```

자체 대국

- 대국 시뮬레이션
  - 경험 데이터 배치 저장

```
experience = rl.combine_experience([collector1, collector2])
with h5py.File(experience_filename, 'w') as experience_outf:
  experience.serialize(experience_outf)
```

감사합니다!

스터디 듣느라 고생 많았습니다.