KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- 알파고의 구성 요소
  - 프로의 바둑 대국 기록을 사용하는 지도학습 기반 딥러닝
  - 자체 대국을 두는 심층 강화학습
  - 심층 신경망으로 성능을 높이는 트리 탐색

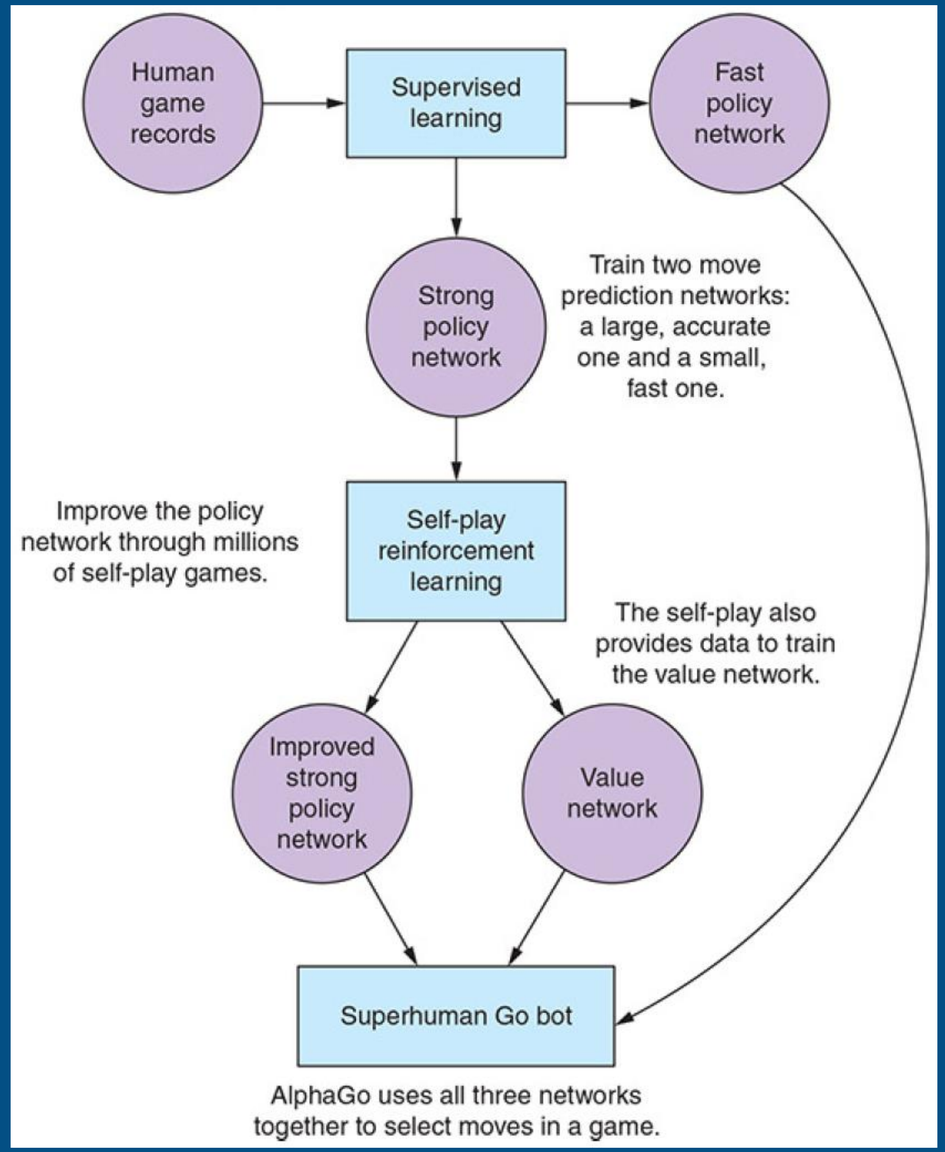
들어가며

- 알파고 시스템
  - 수 예측을 위해 두 심층 합성곱 신경망(**정책 신경망; Policy Network**)을 훈련시킨다.
    이 신경망 구조에서 하나는 좀 더 복잡해 **보다 정확한 결과**를 내고, 다른 하나는 더 작으며 **빠르게 결과**를 낸다. 그래서 이를 각각 **강한** 정책 신경망과 **빠른** 정책 신경망이라고 한다.
  - 강하고 빠른 정책 신경망은 48개의 특징 평면과 다소 복잡한 바둑판 변환기를 사용한다.
  - 정책 신경망의 훈련 단계를 마치면 강한 정책 신경망을 사용해서 자체 대국을 둔다.
    컴퓨팅 자원을 많이 사용할수록 봇 성능도 엄청나게 향상될 것이다.
  - 강한 자체 대국 신경망으로부터 **가치 신경망(Value Network)**를 만들어낼 것이다.
    이로서 신경망 훈련이 끝나고, 여기서부터는 더 이상 딥러닝을 사용하지 않는다.

들어가며

- 알파고 시스템
  - 바둑을 둘 때는 기본적으로 트리 탐색을 사용하지만 몬테카를로 롤아웃 대신 다음 단계를 알려주는 빠른 정책 신경망을 사용할 것이다.
    또한 가치 함수의 결과에 따라 트리 탐색 알고리즘 결과의 균형을 맞출 것이다.
  - 훈련 정책의 전체 과정을 수행하고, 자체 대국을 치르고, 인간을 뛰어넘는 실력으로 대국을 치르는 데까지는 어마어마한 컴퓨터 자원과 시간이 필요하다.

들어가며


Human game records
Supervised learning
Fast policy network
Strong policy network
Train two move prediction networks: a large, accurate one and a small, fast one.
Improve the policy network through millions of self-play games.
Self-play reinforcement learning
The self-play also provides data to train the value network.
Improved strong policy network
Value network
Superhuman Go bot
AlphaGo uses all three networks together to select moves in a game.

알파고의 신경망 훈련

- 알파고 시스템에서 각 신경망의 역할
  - **빠른 정책 신경망(Fast Policy Network)** : 이 신경망의 목적은 가장 정확한 수 예측기가 아니라 수를 매우 빠르게 예측할 수 있는 좋은 예측기다. 트리 탐색 폴아웃에서 사용된다.
  - **강한 정책 신경망(Strong Policy Network)** : 이 바둑 수 예측 신경망은 속도가 아닌 정확도에 초점이 맞춰져 있다. 이 신경망은 빠른 신경망보다 더 단계가 많고 바둑 수를 두 배 이상 잘 예측한다. 이 훈련 단계가 끝난 뒤 강화학습 기법을 활용해 강한 정책 신경망을 자체 대국의 출발점으로 삼는다. 이 단계에서 이 정책 신경망은 더 강해질 것이다.
  - **가치 신경망(Value Network)** : 강한 정책 신경망이 둔 자체 대국에서 만들어지는 새 데이터셋은 가치 신경망을 훈련하는 데 사용된다.

알파고의 신경망 훈련

- 알파고의 신경망 구조
  - 강한 정책 신경망은 13개 층으로 구성된 합성곱 신경망이다.
    각 층에는 19×19 필터가 있다.
    전체 신경망에 걸쳐 원래의 바둑판 크기를 일관되게 유지한다.
    이를 위해 입력값 크기도 조절해줘야 한다.
    첫번째 합성곱층의 커널 크기는 5고, 나머지 층의 커널 크기는 3이다.
    마지막 층은 소프트맥스 활성화 함수를 사용하고 출력 필터 하나를 가지며,
    이전 12개 층은 ReLU 활성화 함수를 사용하고 각각 출력 필터 192개를 가진다.

알파고의 신경망 훈련

- 알파고의 신경망 구조
  - 가치 신경망은 16개 층으로 구성된 합성곱 신경망으로,
    처음 12개 층은 강한 정책 신경망과 완전히 같다.
    13번째 층은 합성곱층이 하나 추가된 것으로 2~12번째 층과 같은 구조다.
    14번째 층은 커널 크기 1에 출력 필터를 갖는 합성곱 층이다.
    신경망은 두 밀집층으로 덮여 있다.
    밀집층 하나에 256개의 출력값과 ReLU 활성화 함수를 가지고,
    마지막 층은 출력값 하나와 tanh 활성화 함수를 사용한다.

알파고의 신경망 훈련

- 알파고의 신경망 구조
  - 알파고의 정책 신경망과 가치 신경망의 신경망 부분 초기화

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D


def alphago_model(input_shape, is_policy_net=False, num_filters=192,
                  first_kernel_size=5, other_kernel_size=3):
    model = Sequential()
    model.add(Conv2D(num_filters, first_kernel_size, input_shape=input_shape,
                     padding='same', data_format='channels_first', activation='relu'))

    for i in range(2, 12):
        model.add(Conv2D(num_filters, other_kernel_size, padding='same',
                         data_format='channels_first', activation='relu'))
```

알파고의 신경망 훈련

- 알파고의 신경망 구조
  - TensorFlow 2로 알파고의 강한 정책 신경망 만들기

```
        if is_policy_net:
            model.add(Conv2D(filters=1, kernel_size=1, padding='same',
                             data_format='channels_first', activation='softmax'))
            model.add(Flatten())
            return model
```

알파고의 신경망 훈련

- 알파고의 신경망 구조
  - TensorFlow 2로 알파고 가치 신경망 구축하기

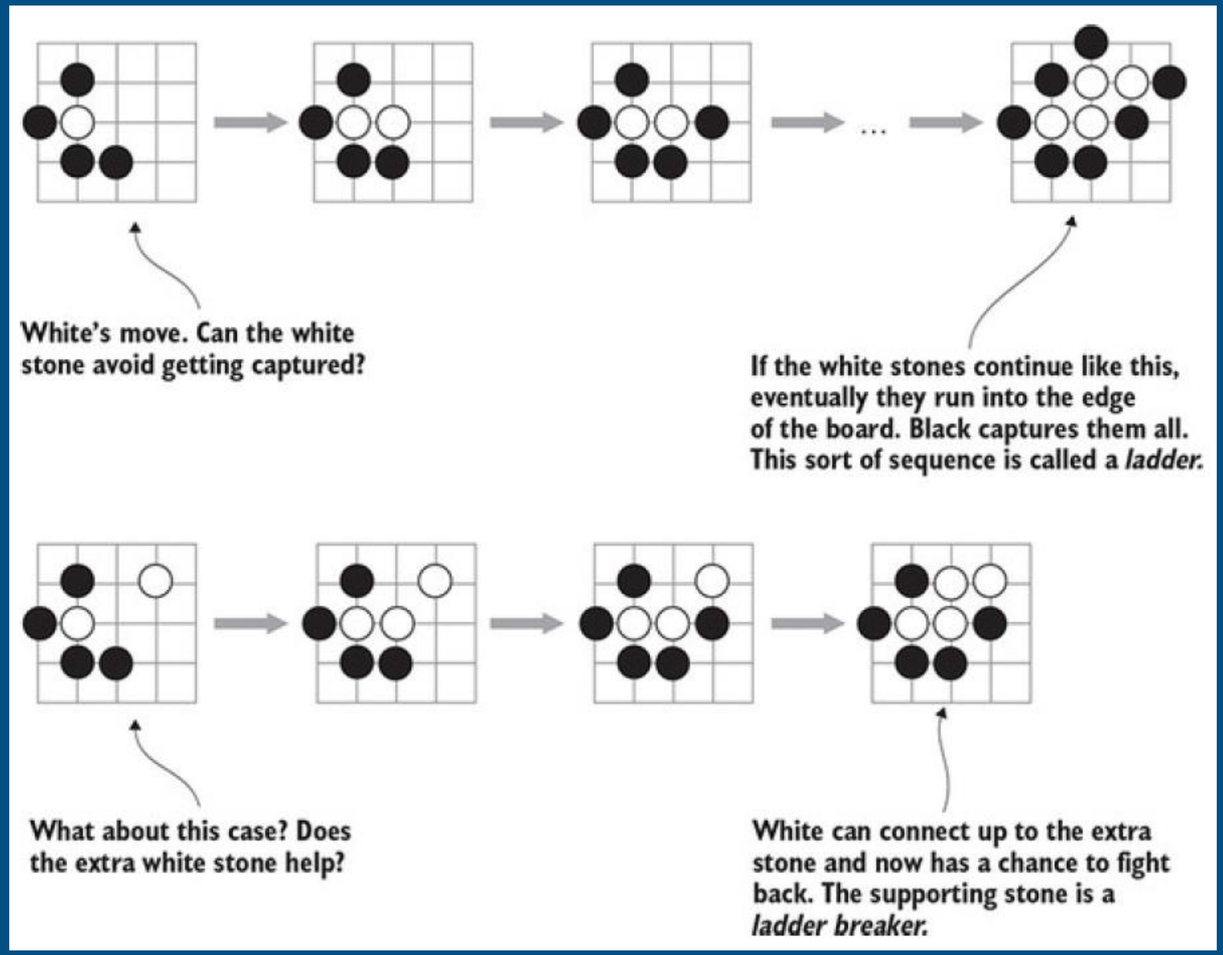
```
        else:
            model.add(Conv2D(num_filters, other_kernel_size, padding='same',
                             data_format='channels_first', activation='relu'))
            model.add(Conv2D(filters=1, kernel_size=1, padding='same',
                             data_format='channels_first', activation='relu'))
            model.add(Flatten())
            model.add(Dense(256, activation='relu'))
            model.add(Dense(1, activation='tanh'))
            return model
```

알파고의 신경망 훈련

- 알파고 바둑판 변환기
  - 알파고의 정책 신경망용 바둑판 변환기는 48개의 특징 평면을 가진다.
    가치 신경망은 평면 하나를 추가해서 이 특징을 강화한다.
  - 이 48개 평면은 11개 개념으로 구성되어 있다. 그 중 일부는 전에 사용했었다.
    알파고는 지금까지의 바둑판 변환기 예제보다 좀 더 바둑에 대한 전술적 상황을 반영한다.
    대표적인 예가 **축 캡처 (Ladder Capture)**와 **활로**의 개념을 만드는 것이다.

알파고의 신경망 훈련

- 알파고 바둑판 변환기



알파고의 신경망 훈련

- 알파고 바둑판 변환기
  - 알파고의 바둑판 변환기에서 꾸준히 사용한 기술은 **이항 특징 (Binary Feature)**이다.
  - 예를 들어, 활로(인접한 빈 점)를 막은 경우 바둑판의 각 돌에 대한 활로 수를 셀 때 하나의 특징 평면만 사용하는 것이 아니라 돌이 1, 2, 3, 혹은 그 이상의 활로를 가지고 있는지 명시하기 위한 평면을 이항적으로 나타내야 할 수 있다.
  - 알파고에서도 동일한 개념을 사용하지만 8개 평면을 사용해 숫자를 이진법으로 나타낸다. 예를 들어, 돌이 1, 2, 3, 4, 5, 6, 7, 8개 활로를 갖는다면 평면 8개가 필요하다는 뜻이다.

알파고의 신경망 훈련

- 알파고의 특징 평면

특징 이름	평면 수	상세
돌 색	3	3가지 특징 평면은 돌 색을 나타낸다. 현재 선수, 상대, 바둑판의 빈 점에 각각 하나씩 사용된다.
1	1	1로 꽉 찬 특징 평면
0	1	0으로 꽉 찬 특징 평면
적합한 수	1	가능한 수이면서 현재 선수의 눈에 채우지 않으면 1이고, 그렇지 않으면 0이다.
몇 회 전의 수	8	8개의 이항 평면으로 어떤 수가 놓인 후 몇 수가 지났는지를 센다.
활로	8	이 수와 관련된 이음의 활로를 특징 평면 8개에 나눠서 나타낸다.
수 이후의 활로	8	이 수를 두면 활로가 얼마나 생길까?
잡은 돌의 수	8	이 수로 주변의 돌이 몇 개나 잡힐까?
자충수	8	이 수를 두면 본인의 돌이 몇 개나 단수 상태에 빠지고, 다음 동작에서 상대에게 얼마나 잡힐까?
축잡기	1	이 돌이 축으로 잡힐까?
축에서 벗어남	1	가능한 축에서 모두 빠져나갈 수 있을까?
현재 돌 색	1	현재 플레이어가 흑이면 1, 백이면 0으로 채워진 평면이다.

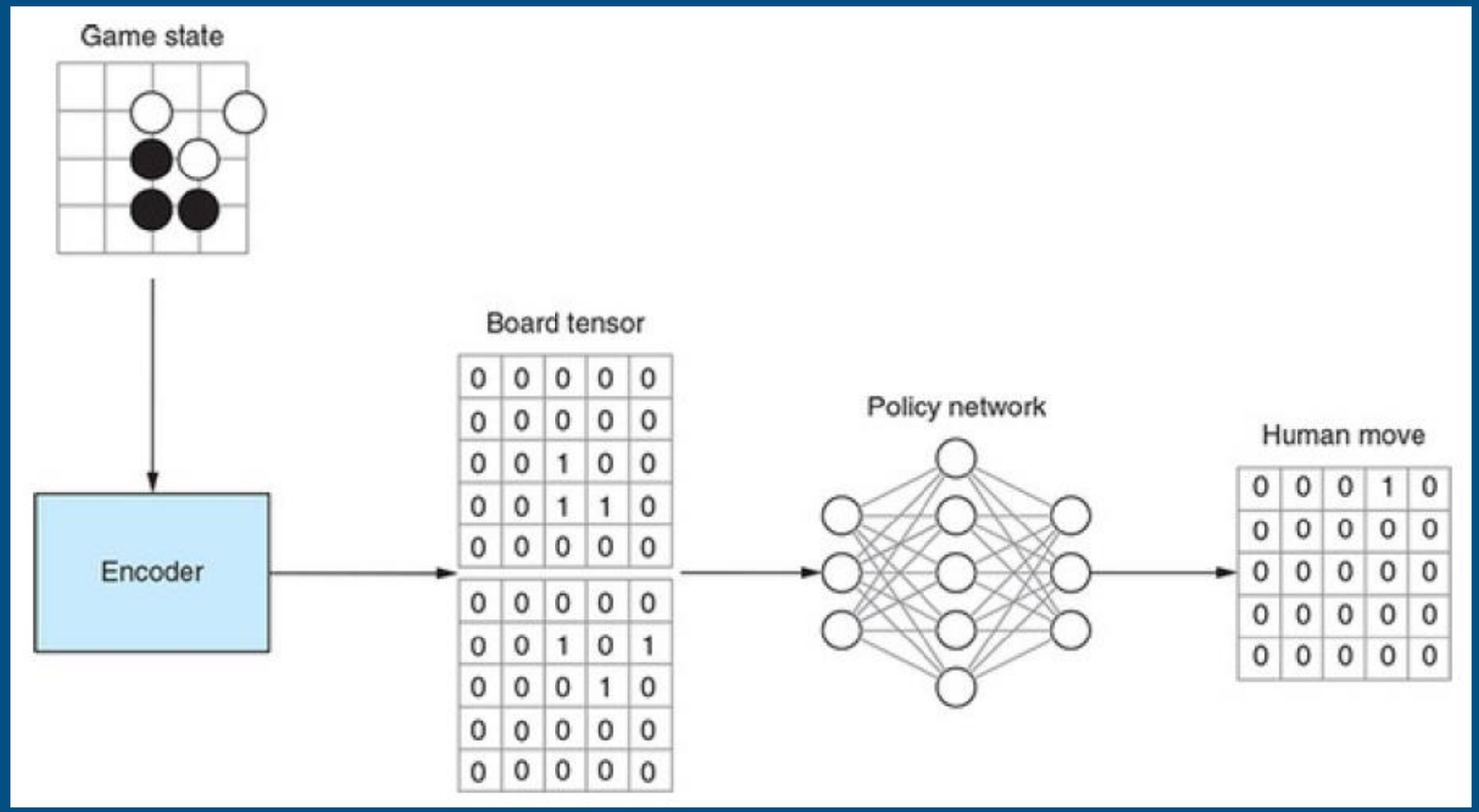
알파고의 신경망 훈련

- 알파고 바둑판 변환기
  - 알파고 바둑판 변환기를 정의하고 초기화하기

```
class AlphaGoEncoder(Encoder):
    def __init__(self, board_size, use_player_plane=False):
        self.board_width, self.board_height = board_size
        self.use_player_plane = use_player_plane
        self.num_planes = 48 + use_player_plane
```

알파고의 신경망 훈련

- 알파고 스타일의 정책 신경망 훈련하기
  - 알파고 정책 신경망의 지도 훈련 과정



알파고의 신경망 훈련

- 알파고 스타일의 정책 신경망 훈련하기
  - 알파고 정책 신경망 훈련의 첫번째 단계에 사용할 데이터 불러오기

```
from dlgo.data.parallel_processor import GoDataProcessor
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.agent.predict import DeepLearningAgent
from dlgo.networks.alphago import alphago_model

from tensorflow.keras.callbacks import ModelCheckpoint
import h5py

rows, cols = 19, 19
num_classes = rows * cols
num_games = 10000

encoder = AlphaGoEncoder()
processor = GoDataProcessor(encoder=encoder.name())
generator = processor.load_go_data('train', num_games, use_generator=True)
test_generator = processor.load_go_data('test', num_games, use_generator=True)
```

알파고의 신경망 훈련

- 알파고 스타일의 정책 신경망 훈련하기
  - TensorFlow 2를 사용해서 알파고 정책 신경망 만들기

```
input_shape = (encoder.num_planes, rows, cols)
alphago_sl_policy = alphago_model(input_shape, is_policy_net=True)

alphago_sl_policy.compile('sgd', 'categorical_crossentropy', metrics=['accuracy'])
```

알파고의 신경망 훈련

- 알파고 스타일의 정책 신경망 훈련하기
  - 정책 신경망 훈련 및 유지

```
epochs = 200
batch_size = 128
alphago_sl_policy.fit_generator(
    generator=generator.generate(batch_size, num_classes),
    epochs=epochs,
    steps_per_epoch=generator.get_num_samples() / batch_size,
    validation_data=test_generator.generate(batch_size, num_classes),
    validation_steps=test_generator.get_num_samples() / batch_size,
    callbacks=[ModelCheckpoint('alphago_sl_policy_{epoch}.h5')]
)

alphago_sl_agent = DeepLearningAgent(alphago_sl_policy, encoder)

with h5py.File('alphago_sl_policy.h5', 'w') as sl_agent_out:
    alphago_sl_agent.serialize(sl_agent_out)
```

자체 대국 부트스트래핑

- 우리는…
  - `alphago_sl_agent`를 통해 상대적으로 강한 정책 에이전트를 훈련시켰으므로 정책 경사 알고리즘을 사용해서 자체 대국을 두도록 할 수 있다.
- 딥마인드의 알파고는…
  - 현재 가장 강한 버전과 **서로 다른 강한 정책 신경망**을 맞붙인다.
  - 이 방식을 사용하면 과적합을 방지하고 전반적으로 성능이 더 나아지지만 `alphago_sl_agent`가 자체 대국을 두게 하는 단순한 접근 방식은 정책 에이전트를 더 강하게 만들기 위해 자기 플레이를 사용하자는 일반적인 생각을 전달한다.

자체 대국 부트스트래핑

- 다음 훈련 단계에서는…
  - 지도학습 정책 신경망 `alphago_sl_agent`를 두 번 불러온다. 첫번째 불러온 것은 `alphago_rl_agent`라고 하는 새로운 강화학습 에이전트로 사용할 것이고, 두번째는 이 에이전트의 상대 역할을 할 것이다.

자체 대국 부트스트래핑

- 두 자체 대국 상대를 만들기 위해 훈련 정책 신경망을 두 번 불러오기

```
from dlgo.agent.pg import PolicyAgent
from dlgo.agent.predict import load_prediction_agent
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.rl.simulate import experience_simulation
import h5py

encoder = AlphaGoEncoder()

sl_agent = load_prediction_agent(h5py.File('alphago_sl_policy.h5'))
sl_opponent = load_prediction_agent(h5py.File('alphago_sl_policy.h5'))

alphago_rl_agent = PolicyAgent(sl_agent.model, encoder)
opponent = PolicyAgent(sl_opponent.model, encoder)
```

자체 대국 부트스트래핑

- PolicyAgent 학습에 사용할 자체 대국 데이터 생성

```
num_games = 1000
experience = experience_simulation(num_games, alphago_rl_agent, opponent)

alphago_rl_agent.train(experience)

with h5py.File('alphago_rl_policy.h5', 'w') as rl_agent_out:
    alphago_rl_agent.serialize(rl_agent_out)

with h5py.File('alphago_rl_experience.h5', 'w') as exp_out:
    experience.serialize(exp_out)
```

가치 신경망 도출하기

- 알파고의 신경망 훈련 중 마지막 단계는 `alphago_rl_agent`에 사용했던 것과 **동일한 자체 대국 경험 데이터**를 사용해서 가치 신경망을 훈련하는 것이다.
- 이 단계는 구조적으로 이전 단계와 비슷한다.

가치 신경망 도출하기

- 알파고 가치 신경망 초기화

```
from dlgo.networks.alphago import alphago_model
from dlgo.encoders.alphago import AlphaGoEncoder
from dlgo.rl import ValueAgent, load_experience
import h5py

rows, cols = 19, 19
encoder = AlphaGoEncoder()
input_shape = (encoder.num_planes, rows, cols)
alphago_value_network = alphago_model(input_shape)

alphago_value = ValueAgent(alphago_value_network, encoder)
```

가치 신경망 도출하기

- 경험 데이터로 가치 신경망 훈련하기

```
experience = load_experience(h5py.File('alphago_rl_experience.h5', 'r'))

alphago_value.train(experience)

with h5py.File('alphago_value.h5', 'w') as value_agent_out:
    alphago_value.serialize(value_agent_out)
```

신경망을 사용한 탐색 개선

- 몬테카를로 탐색 트리 단계
  - **선택** : **하위 노드** 중에서 임의로 선택하는 방식으로 게임 트리를 횡단
  - **확장** : 트리에 **새 노드**(새 게임 상태) 추가
  - **평가** : **단말 노드**가 추가된 상태에서 경기를 임의로 완전히 종료할 때까지 시뮬레이션
  - **갱신** : 시뮬레이션이 완료되면 결과에 따라 트리 상태를 갱신
- 알파고 시스템은 보다 정교한 트리 탐색 알고리즘을 사용한다.

신경망을 사용한 탐색 개선

- 신경망으로 몬테카를로 롤아웃 개선하기
  - 빠른 정책 신경망을 사용해서 롤아웃 실행하기
    (롤아웃은 많은 것을 빠르게 해결해야 한다 → 빠른 정책 신경망이 제격)

```
def policy_rollout(game_state, fast_policy):
  next_player = game_state.next_player()
  while not game_state.is_over():
    move_probabilities = fast_policy.predict(game_state)
    greedy_move = max(move_probabilities)
    game_state = game_state.apply_move(greedy_move)

  winner = game_state.winner()
  return 1 if winner == next_player else -1
```

신경망을 사용한 탐색 개선

- 신경망으로 몬테카를로 롤아웃 개선하기
  - 정책 신경망은 기본적으로 동전 던지기보다 수 선택을 더 잘하므로 롤아웃 정책을 사용하는 것이 일단 더 낫긴 하다. 하지만 여전히 개선할 부분이 많다.
  - 예를 들어, 트리의 단말 노드에 있고 이를 확장해야 하는 경우 임의로 확장할 새 노드를 선택하지 말고 **좋은 수를 강한 정책 신경망에 물어볼 수 있다.**
    정책 신경망은 모든 다음 수에 대한 확률 분포를 제공하고, 각 노드는 이 확률을 따르므로 (정책에 따랐을 때의) 강한 수가 다른 수보다 선택될 확률이 더 높다. 이 노드의 확률은 다른 트리 탐색 전에 수가 얼마나 강한지에 대한 사전 지식을 제공하므로 이를 **사전 확률 (Prior Probability)**이라고 한다.

신경망을 사용한 탐색 개선

- 신경망으로 몬테카를로 롤아웃 개선하기
  - 마지막으로 가치 신경망이 경기에 어떻게 사용될 수 있는지 살펴 보자. 이미 임의로 추측하는 부분을 정책 신경망으로 대체함으로써 롤아웃의 성능을 향상시켰다. 하지만 여전히 각 노드는 단일 경기의 결과만 계산해서 이 노드가 얼마나 가치 있는지 구한다. 위치의 가치 추정은 가치 네트워크가 어떤 것이 좋은 것이라고 판단하는지 훈련한 것이므로 이에 대해서 이미 정교하게 추정하고 있을 것이다.
  - 알파고가 하는 일은 가치 신경망의 출력에 따라 롤아웃의 결과에 **가중치를 매기는** 것이다. 사람이 게임을 할 때 결정을 내리는 방식과 유사한데 실제로 가능한 많은 수를 내다보고, 이전에 게임을 할 때 어땠는 지도 고려한다. 좋은 것 같은 수 진행을 읽었다고 하면 이 위치가 좋지 않을 것 같다고 해도 수를 놓게 되고, 그 반대도 마찬가지다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 알파고 탐색 트리의 각 노드에는 **Q값**을 저장한다.
  - Q값 리마인드 : 현재 바둑판 상태 s와 가능한 다음 수 a가 있을 때,
    행동값 $Q(s,a)$는 현재 수 a가 상태 s에서 얼마나 좋은 수인지 추정한다.
  - 또한 각 노드는 **방문수**를 추적하는데, 탐색에 의해 이 노드가 얼마나 자주 통과되었는지,
    사전 확률 $\boldsymbol{P(s,a)}$나 강한 정책 신경망이 s에서 행동 a가 얼마나 가치 있는지
    파악하는 데 쓰인다.
  - 참고로 트리의 각 노드는 정확히 하나의 부모 노드를 가지지만,
    잠재적으로 여러 개의 **자식 노드**를 가질 수 있다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 알파고 트리의 노드 간단히 살펴 보기

```python
class AlphaGoNode:
    def __init__(self, parent, probability):
        self.parent = parent
        self.children = {}

        self.visit_count = 0
        self.q_value = 0
        self.prior_value = probability
```

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 현재 진행 중인 경기를 살펴 본다고 해보자.
    이미 큰 트리가 만들어져 있고, 방문수와 행동값의 추정치도 있다.
  - 이제 필요한 건 많은 경기를 시뮬레이션하고 경기 통계량을 얻어서
    시뮬레이션이 끝난 후 가장 좋은 수를 선택하는 것이다.
    경기 시뮬레이션을 하려면 트리를 어떻게 지나가야 할까?

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 현재의 경기 상태가 s고 방문수가 $N(s)$라고 한다면, 다음과 같이 행동을 선택할 수 있다.

$$a' = \text{argmax}_a \, Q(s,a) + \frac{P(s,a)}{1 + N(s,a)}$$

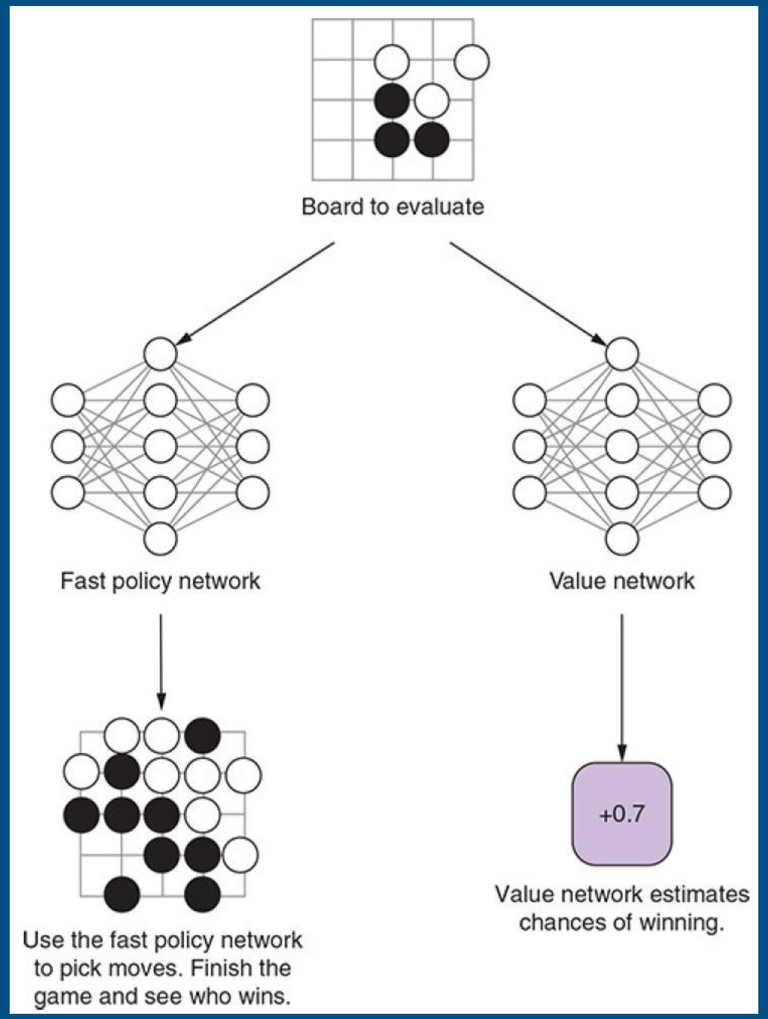
  - argmax 표기는 공식 $Q(s,a) + P(s,a)/\big(1 + N(s,a)\big)$를 최대로 만들 수 있는 파라미터 a를 구하는 것이다.
  - 최대로 만들 항목은 Q값과 방문수로 **정규화**된 사전 확률로 구성되어 있다.
  - 처음에 방문수는 0으로, 동일한 가중치를 가진 $Q(s,a) + P(s,a)$를 최대로 만들게 된다.
  - 방문수가 매우 크면 $P(s,a)/\big(1 + N(s,a)\big)$는 무시해도 될 수준으로 작아질 것이므로 $Q(s,a)$에만 신경쓰는 것이 효과적이다.
  - 이 유틸리티 함수를 $u(s,a) = P(s,a)/\big(1 + N(s,a)\big)$라고 하자. 이후 살짝 고치겠지만 이 버전에서도 필요한 내용은 다 가지고 있다. 이 표기법을 사용하면 수를 선택할 때 $a' = \text{argmax}_a \, Q(s,a) + u(s,a)$라고 쓸 수 있다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 요약하면 Q값에 따라 사전 확률에 가중치를 준 값으로 행동을 선택하는 것이다.
    트리를 가로지르면서 방문수를 요약하고, Q에 대해 좀 더 나은 추정을 하면
    천천히 **사전 추정치 (Prior Estimation)** 보다 Q값을 더 신뢰할 수 있게 될 것이다.
  - 지금까지 설명한 내용이 알파고가 기존의 트리에서 수를 **선택**하는 방식이다.
    하지만 노드 l에서 트리를 **확장**할 때는 어떻게 할까?

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색



신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 우선 강한 정책 신경망 $P(l)$의 예측값을 구하고 이를 l의 각 자식의 사전 확률로 저장한다. 그리고 정책 롤아웃과 가치 신경망을 결합해 다음과 같이 노드를 평가한다.

$$V(l) = \lambda \cdot \text{value}(l) + (1 - \lambda) \cdot \text{rollout}(l)$$

  - 이 식에서 $\text{value}(l)$은 l의 가치 신경망 값이고, $\text{rollout}(l)$은 l에서의 빠른 정책 신경망 롤아웃의 경기 결과를 나타내며, λ는 0과 1 사이의 값으로 기본값은 0.5다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 한발짝 물러서서 트리 탐색으로 총 n경기를 시뮬레이션하는 이유는 결국 수 선택이다. 수 선택이 제대로 이뤄지려면 시뮬레이션이 끝나고 방문수와 Q값을 갱신해야 한다.
    - 방문수 갱신은 쉽다. 탐색이 걸린 노드에 방문수를 1 늘려주면 된다.
    - Q값을 갱신하려면 모든 방문된 노드에 대해 $V(l)$을 더하고 이를 방문수로 나눈다.

$$Q(s,a) = \sum_{i=1}^{n} \frac{V(l_i)}{N(s,a)}$$

  - n번의 시뮬레이션에 대한 모든 값을 더하고 i번째 시뮬레이션에서 (s,a)에 해당하는 노드를 탐색한 경우 이 노드값도 더한다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 전체 과정을 요약하면 다음과 같다.
    - **선택** : $Q(s,a)+u(s,a)$를 최대로 하는 행동을 선택하며 게임 트리를 돌아다닌다.
    - **확장** : 새 노드를 확장할 때 강한 신경망에 사전 확률을 각 자식 노드에 저장하도록 요청한다.
    - **평가** : 시뮬레이션이 끝나면 빠른 정책 신경망을 사용한 롤아웃의 결과와 가치 신경망의 결과의 평균을 구해서 단말 노드를 평가한다.
    - **갱신** : 모든 시뮬레이션이 끝난 뒤 시뮬레이션에서 탐색한 노드의 방문수와 Q값을 갱신한다.

신경망을 사용한 탐색 개선

- 결합 가치 함수를 사용한 트리 탐색
  - 마지막으로 시뮬레이션이 끝난 후 **경기**에서 수를 어떻게 선택할까?
    간단하다. 가장 많이 방문한 노드를 선택한다.
  - 노드의 Q값이 높아질수록 해당 노드의 방문수도 더 많아질 것이다.
    시뮬레이션 후 노드의 방문수는 수의 상대적 가치를 알려주는 훌륭한 이정표가 된다.

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - Python으로 알파고 트리 노드 정의하기

```python
import numpy as np
from dlgo.agent.base import Agent
from dlgo.goboard_fast import Move
from dlgo import kerasutil
import operator

class AlphaGoNode:
    def __init__(self, parent=None, probability=1.0):
        self.parent = parent
        self.children = {}

        self.visit_count = 0
        self.q_value = 0
        self.prior_value = probability
        self.u_value = probability
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 노드는 트리 탐색 알고리즘의 세 부분에서 사용된다.
    - `select_child()` : 시뮬레이션에서 트리를 돌아다니면서 $\text{argmax}_a\, Q(s,a) + u(s,a)$에 따라 노드의 자식 노드를 선택한다. 이때 Q값과 유틸리티 함수의 합을 최대로 하는 작업을 선택한다.
    - `expand_children()` : 단말 노드에서는 이 위치에서 가능한 모든 수를 평가하는 강한 정책 신경망을 호출하고 각각에 `AlphaGoNode` 인스턴스를 추가한다.
    - `update_values()` : 모든 시뮬레이션을 끝낸 후 결과에 따라 `visit_count`, `q_value`, `u_value`를 갱신한다.

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - Q값을 최대화하는 알파고 자식 노드 선택

```
class AlphaGoNode:
...
    def select_child(self):
        return max(self.children.items(),
                   key=lambda child: child[1].q_value + \
                   child[1].u_value)

    def expand_children(self, moves, probabilities):
        for move, prob in zip(moves, probabilities):
            if move not in self.children:
                self.children[move] = AlphaGoNode(probability=prob)
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 알파고 노드의 요약 통계량을 갱신하는 메소드 `update_values()`는 조금 복잡하다.
    좀 더 복잡한 버전의 유틸리티 함수를 사용한다.

$$u(s,a) = c_u \sqrt{N_p(s,a)} \frac{P(s,a)}{1 + N(s,a)}$$

  - 이 유틸리티 함수에는 앞에서 소개한 버전보다 2개의 항이 더 있다.
  - 하나는 c_u로, 코드에서는 `c_u`로 표현했다. 이 항은 모든 노드를 고정 상수배로 크게 하며, 기본값은 5다.
  - 다음 항은 부모의 방문수에 제곱근을 취하는 유틸리티 함수다. (해당 노드의 부모는 N_p로 나타낸다)
    이 항은 더 많이 방문된 부모 노드를 가진 노드가 더 많이 활용되도록 한다.

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 방문수, Q값, 알파고 노드의 유틸리티 함수 갱신

```
class AlphaGoNode:
...
    def update_values(self, leaf_value):
        if self.parent is not None:
            self.parent.update_values(leaf_value)

        self.visit_count += 1

        self.q_value += leaf_value / self.visit_count

        if self.parent is not None:
            c_u = 5
            self.u_value = c_u * np.sqrt(self.parent.visit_count) \
                * self.prior_value / (1 + self.visit_count)
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - `AlphaGoNode` 정의를 통해 알파고의 탐색 알고리즘 트리 구조를 사용할 수 있게 되었다.
  - 이제 구현할 `AlphaGoMCTS` 클래스는 여러 인수를 사용해서 초기화하는 에이전트다.
    우선 이 에이전트에 빠른 정책 신경망, 강한 정책 신경망, 가치 신경망을 넣어주어야 한다.
    다음으로 롤아웃, 평가와 관련해 알파고에서만 사용하는 다음 파라미터를 정의해야 한다.
    - `lambda_value` : 롤아웃과 가치 함수 각각에 가중치를 추가할 때 사용하는 값이다.

$$V(l) = \lambda \cdot \text{value}(l) + (1 - \lambda) \cdot \text{rollout}(l)$$

    - `num_simulations` : 이 값은 수 선택 과정 중 얼마나 많은 시뮬레이션을 사용할 지 명시한다.
    - `depth` : 알고리즘에서 시뮬레이션별로 얼마나 많은 수를 내다볼지 정한다. (탐색 깊이)
    - `rollout_limit` : 단말값을 결정할 때 정책 롤아웃 $\text{rollout}(l)$을 실행한다.
      결과를 판단하기 전에 `rollout_limit` 파라미터를 사용해 롤아웃할 수를 알파고에 알려준다.

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 바둑 에이전트 AlphaGoMCTS 초기화

```
class AlphaGoMCTS(Agent):
    def __init__(self, policy_agent, fast_policy_agent, value_agent,
                 lambda_value=0.5, num_simulations=1000,
                 depth=50, rollout_limit=100):
        self.policy = policy_agent
        self.rollout_policy = fast_policy_agent
        self.value = value_agent

        self.lambda_value = lambda_value
        self.num_simulations = num_simulations
        self.depth = depth
        self.rollout_limit = rollout_limit
        self.root = AlphaGoNode()
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 알파고의 트리 탐색 과정 리마인드
    - 수를 둘 때 해야 할 첫번째 일은 게임 트리 시뮬레이션을 `num_simulations`회 실행하는 것이다.
    - 각 시뮬레이션에서 지정된 깊이(`depth`)까지 내다보는 탐색을 실행한다.
    - 노드에 자식 노드가 없다면 가능한 각 수에 새 AlphaGoNode를 추가하고,
      사전 확률을 구하는 강한 정책 신경망을 사용해서 트리를 **확장**한다.
    - 노드에 자식 노드가 있다면 Q값과 유틸리티를 최대로 만드는 수를 선택해 노드를 **선택**한다.
    - 이 시뮬레이션에서 사용한 수를 바둑판에 놓는다.
    - 지정된 깊이에 도달하면 가치 신경망에 정책 롤아웃이 결합된 가치 함수의 값을 구해 단말 노드를 **평가**한다.
    - 모든 알파고의 노드를 시뮬레이션에서의 단말값을 사용해서 갱신한다.

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 알파고의 트리 탐색 과정의 핵심 메소드

```
class AlphaGoMCTS(Agent):
...
    def select_move(self, game_state):
        for simulation in range(self.num_simulations):
            current_state = game_state
            node = self.root
            for depth in range(self.depth):
                if not node.children:
                    if current_state.is_over():
                        break
                    moves, probabilities = self.policy_probabilities(current_state)
                    node.expand_children(moves, probabilities)

                move, node = node.select_child()
                current_state = current_state.apply_move(move)

            value = self.value.predict(current_state)
            rollout = self.policy_rollout(current_state)

            weighted_value = (1 - self.lambda_value) * value + self.lambda_value * rollout

            node.update_values(weighted_value)
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 가장 많이 방문된 노드를 선택하고 트리의 시작 노드를 갱신

```
class AlphaGoMCTS(Agent):
...
    def select_move(self, game_state):
        ...
              move = max(self.root.children, key=lambda move:
                   self.root.children.get(move).visit_count)

        self.root = AlphaGoNode()
        if move in self.root.children:
            self.root = self.root.children[move]
            self.root.parent = None

        return move
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 바둑판에서 가능한 수에 대해 정규화된 강한 정책값 구하기

```
class AlphaGoMCTS(Agent):
...
    def policy_probabilities(self, game_state):
        encoder = self.policy._encoder
        outputs = self.policy.predict(game_state)
        legal_moves = game_state.legal_moves()
        if not legal_moves:
            return [], []
        encoded_points = [encoder.encode_point(move.point) for move in legal_moves if move.point]
        legal_outputs = outputs[encoded_points]
        normalized_outputs = legal_outputs / np.sum(legal_outputs)
        return legal_moves, normalized_outputs
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - `rollout_limit`에 도달할 때까지 경기 진행하기

```
class AlphaGoMCTS(Agent):
...
    def policy_rollout(self, game_state):
        for step in range(self.rollout_limit):
            if game_state.is_over():
                break
            move_probabilities = self.rollout_policy.predict(game_state)
            encoder = self.rollout_policy.encoder
            valid_moves = [m for idx, m in enumerate(move_probabilities)
                           if Move(encoder.decode_point_index(idx)) in game_state.legal_moves()]
            max_index, max_value = max(enumerate(valid_moves), key=operator.itemgetter(1))
            max_point = encoder.decode_point_index(max_index)
            greedy_move = Move(max_point)
            if greedy_move in game_state.legal_moves():
                game_state = game_state.apply_move(greedy_move)

        next_player = game_state.next_player
        winner = game_state.winner()
        if winner is not None:
            return 1 if winner == next_player else -1
        else:
            return 0
```

신경망을 사용한 탐색 개선

- 알파고의 탐색 알고리즘 구현
  - 세 심층 신경망을 사용하는 알파고 에이전트 초기화

```
from dlgo.agent import load_prediction_agent, load_policy_agent, AlphaGoMCTS
from dlgo.rl import load_value_agent
import h5py

fast_policy = load_prediction_agent(h5py.File('alphago_sl_policy.h5', 'r'))
strong_policy = load_policy_agent(h5py.File('alphago_rl_policy.h5', 'r'))
value = load_value_agent(h5py.File('alphago_value.h5', 'r'))

alphago = AlphaGoMCTS(strong_policy, fast_policy, value)
```

훈련할 때 고민해야 할 부분

- 알파고를 최대한 잘 사용하기 위해 짚고 넘어가야 할 점
  - 훈련의 첫번째 단계인 정책 신경망 지도학습은 KGS의 160,000 경기 데이터에서 3천만 개의 상태를 변환해서 실행했다. 딥마인드의 알파고팀은 총 3억 4천만 개의 훈련 단계를 계산했다.
  - 좋은 소식이라면 여러분도 동일한 데이터셋을 사용할 수 있다는 거다. 딥마인드에서 사용한 데이터셋은 앞에서 소개했던 KGS 훈련 데이터셋이다. 기본적으로 동일한 훈련 단계를 실행한다고 해도 아무런 문제가 없다. 하지만 나쁜 소식도 있다. 최신 GPU를 가지고 있더라도 훈련 과정이 몇 년은 걸리지 않을 수 있지만 몇 달은 걸린다는 것이다.

훈련할 때 고민해야 할 부분

- 알파고를 최대한 잘 사용하기 위해 짚고 넘어가야 할 점
  - 알파고 팀은 이 문제를 해결하기 위해 50개 GPU로 훈련 과정을 **분산**시켰고, 훈련에 3주가 걸렸다. 여기서는 심층 신경망을 분산 환경에서 실행하는 방법은 생략한다.
  - 따라서 만족스런 결과를 얻으려면 각 연산 부분을 간단히 만들어야 한다. 이전에 설명한 바둑판 변환기 중 하나를 사용하고 오늘 설명한 알파고의 정책 신경망과 가치 신경망보다 훨씬 작은 신경망을 사용하자. 또한 일단 작은 훈련 데이터셋으로 시작하면 훈련 과정이 진행되는 것을 볼 수 있을 것이다.
  - 자체 대국의 경우 딥마인드에서는 3천만 개의 서로 다른 수를 생성했다. 이는 실제로 만들 수 있을 거라 생각하는 수의 개수보다 엄청나게 많은 숫자다. 일단 지도학습으로 사람이 두는 수만큼 많은 자체 대국 수를 만들어 보자.

훈련할 때 고민해야 할 부분

- 알파고를 최대한 잘 사용하기 위해 짚고 넘어가야 할 점
  - 오늘 나왔던 큰 신경망을 그대로 가져다 매우 적은 데이터로 훈련하면 더 많은 데이터로 작은 신경망을 훈련하는 것보다 더 안좋은 결과가 나올 것이다.
  - 빠른 정책 신경망은 롤아웃에서 빈번하게 사용되므로 트리 탐색 속도를 빠르게 하려면 빠른 정책 신경망이 초반에는 정말 작은 크기여야 한다.
  - 앞에서 구현한 트리 탐색 알고리즘은 시뮬레이션 결과를 **순차적으로** 구한다. 이 과정을 빠르게 하기 위해 딥마인드에서는 탐색 스레드를 총 40개 사용해서 탐색 과정을 **병렬**로 처리했다. 병렬 버전에서는 여러 GPU를 사용해서 병렬로 심층 신경망을 평가하고, 여러 CPU를 사용해서 트리 탐색의 다른 부분을 실행했다.

훈련할 때 고민해야 할 부분

- 알파고를 최대한 잘 사용하기 위해 짚고 넘어가야 할 점
  - 여러 CPU를 사용해서 트리 탐색을 실행하는 것은 가능하지만 여기서는 다루지 않았다.
  - 대국 경험 향상을 위해서는 정확도와 성능의 트레이드오프가 필요하고, 이를 위해 시뮬레이션 수와 탐색 깊이를 줄일 수 있다. 이렇게 하면 초인적 성능은 안나오겠지만 컴퓨터와 경기를 진행할 정도는 될 것이다.

감사합니다!

스터디 듣느라 고생 많았습니다.