KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- 지난 시간에 알파고(AlphaGo)를 배웠다. 이 시점에서 질문 하나!
  만약 알파고가 기존에 사람이 둔 수를 전혀 참고하지 않고 완전히
  강화학습만으로 학습을 한다면 어떨까? → **알파고 제로 (AlphaGo Zero)**
  - 알파고 제로는 알파고보다 더 적은 자원으로 훨씬 더 많은 능력을 갖게 되었다.
  - 알파고 제로는 기존 알파고보다 훨씬 단순하며 직접 만들어야 하는 특징 평면도 없다.
  - 사람이 둔 대국 기록도 없고, 몬테카를로 롤아웃도 없다.
  - 알파고에서는 두 신경망과 세 훈련 과정을 사용했지만,
    알파고 제로는 하나의 신경망과 하나의 훈련 과정만 사용한다.

들어가며

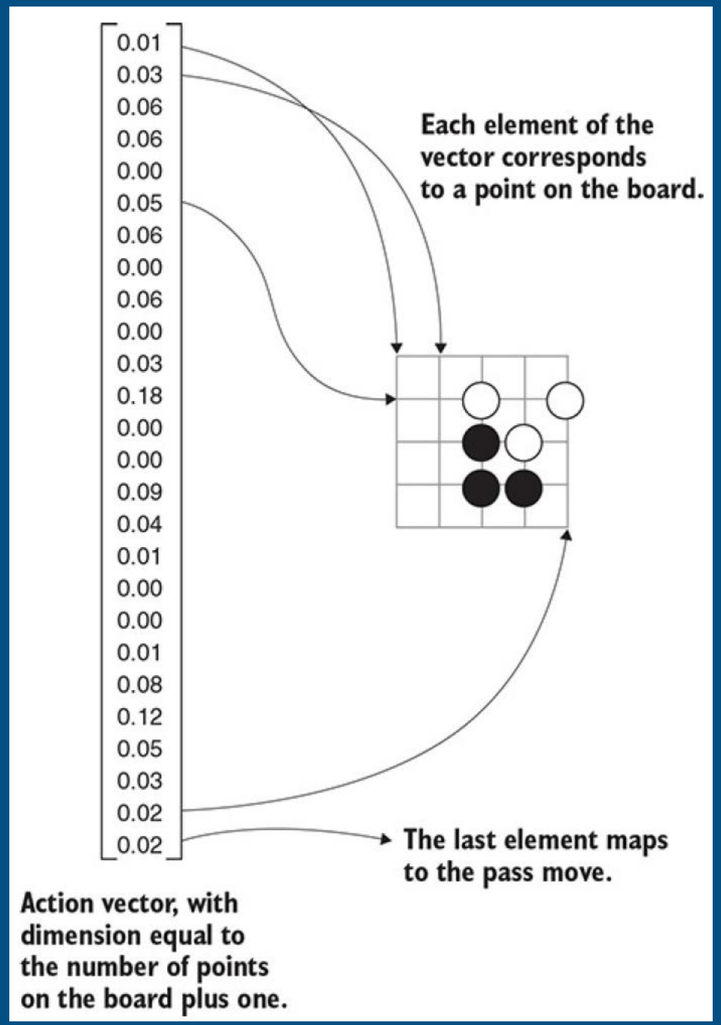
- 하지만 알파고 제로는 기존 알파고보다 강하다! 어떻게 이럴 수 있을까?
  - 알파고 제로는 정말 큰 신경망을 사용한다. 가장 강력한 알파고 제로 버전은 기존 알파고 신경망의 4배 이상 크기인 80개의 합성곱층을 사용하는 거대한 신경망을 사용한다.
  - 알파고 제로는 혁신적이고 새로운 강화학습 기법을 사용한다.
    기존 알파고는 정책 신경망을 단독으로 훈련한 후 이를 트리 탐색을 강화하는 데 사용했다.
    반면 알파고 제로는 트리 탐색과 강화학습을 처음부터 결합해서 사용한다.

트리 탐색용 신경망 만들기

- 알파고 제로는 입력값 하나와 출력값 둘을 갖는 단일 신경망을 사용한다.
  - 하나는 출력값의 확률 분포를 생성한다. 다른 하나는 이 경기가 흑/백 중 누가 이길 것 같은 지를 나타내는 단일 수치를 생성한다. (행위자-비평가 학습 방식과 같은 구조)
  - 하지만 이전에 다뤘던 신경망과 알파고 제로 신경망의 출력에는 작은 차이점이 하나 있다. 바로 차례를 넘기는 방법이다.
    - 이전에 자체 대국을 구현했을 때는 차례를 넘기는 방법을 직접 코드에 작성했다. `PolicyAgent`를 사용하는 자체 대국봇에는 눈을 채우지 못했을 때 본인의 돌을 치우는 자체 규칙이 들어 있었다. 유일한 가능한 수가 자충수이면 `PolicyAgent`는 차례를 넘긴다. 이를 통해 자체 대국이 합리적으로 이뤄진다.
    - 알파고 제로는 자체 대국 시 트리 탐색을 사용하므로 따로 규칙을 추가할 필요가 없다. 차례를 넘기는 것도 수를 두는 것과 마찬가지로 처리하면 되고, 봇이 언제 차례를 넘기는 것이 적절한 지 알아서 배우기를 기대하면 된다. 트리 탐색을 통해서 돌을 뒀을 때 질 것 같다고 판단하게 되면 차례를 넘기는 것을 선택하면 된다.
      이는 행동 후 바둑판의 모든 점에 대해 차례를 넘기는 것에 대한 확률을 구해야 한다는 걸 의미한다.

트리 탐색용 신경망 만들기

- 가능한 수를 벡터로 변환하는 과정



트리 탐색용 신경망 만들기

- 차례 넘기기를 추가해서 바둑판을 변환하도록 수정

```python
class ZeroEncoder(Encoder):
...
    def encode_move(self, move):
        if move.is_play:
            return (self.board_size * (move.point.row - 1) + (move.point.col - 1))
        elif move.is_pass:
            return self.board_size * self.board_size
        raise ValueError('Cannot encode resign move')

    def decode_move_index(self, index):
        if index == self.board_size * self.board_size:
            return Move.pass_turn()
        row = index
        col = index % self.board_size
        return Move.play(Point(row=row + 1, col=col + 1))

    def num_moves(self):
        return self.board_size * self.board_size + 1
```

신경망으로 트리 탐색 안내하기

- **정책(Policy)**은 에이전트에게 어떻게 결정할 지 알려주는 알고리즘이다.
  - 이전에 강화학습 예제에서 다뤘던 정책은 상대적으로 단순했다.
    - 정책 경사 학습과 행위자-비평가 학습에서는 신경망이 어떤 수를 직접 골라야 할 지 알려줬다.
    - Q-학습에서는 각 가능한 수에 대한 Q값을 구하는 데 관여했고, 그 중 가장 높은 Q값을 가진 수를 선택했다.
- 알파고 제로의 정책은 트리 탐색 형태를 포함한다.
  - 여전히 신경망을 사용하지만 직접 수를 고르거나 평가하는 목적이 아닌 트리 탐색을 안내하는 역할이다.
  - 자체 대국 중 트리 탐색을 포함함으로써 자체 대국을 더 실제적으로 두게 된다.

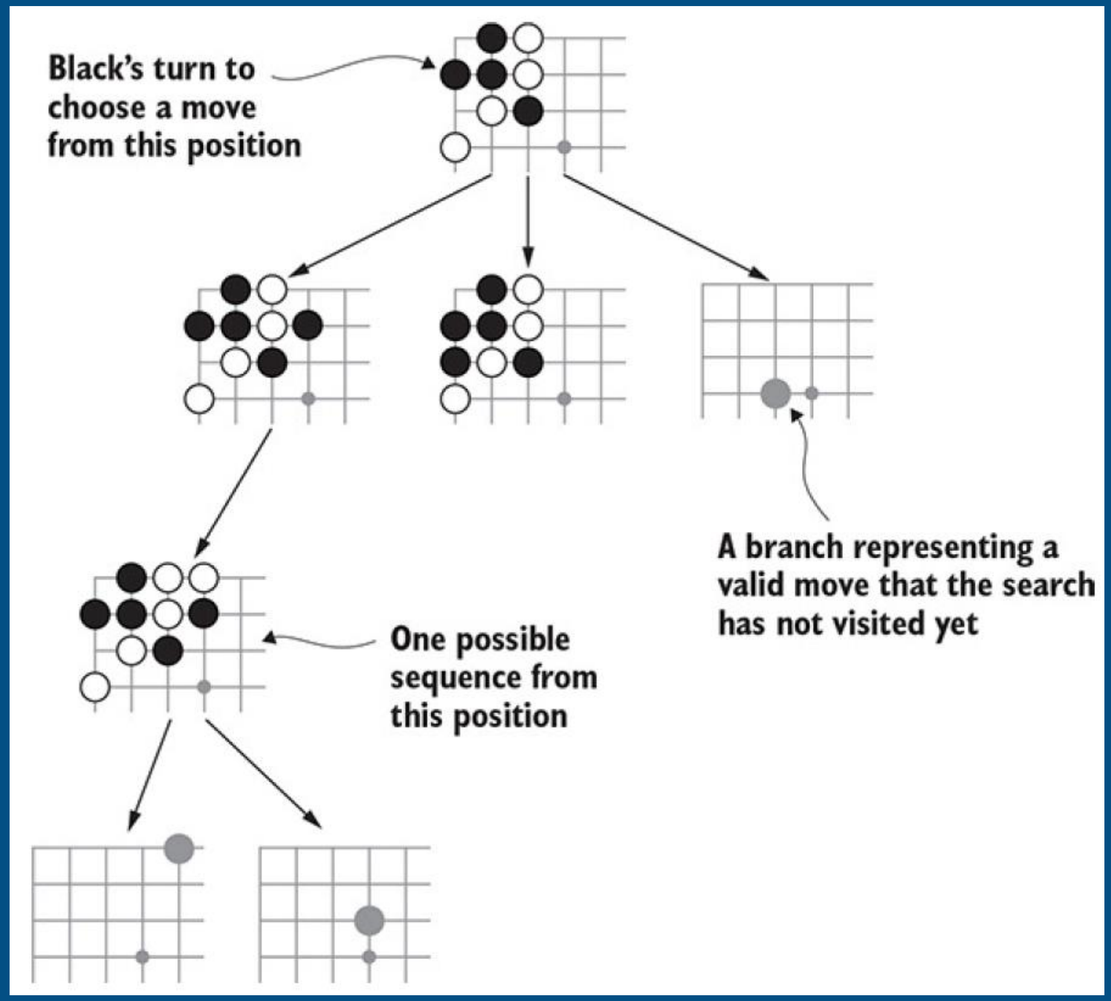
신경망으로 트리 탐색 안내하기

- 트리 탐색 알고리즘 비교

	MCTS	알파고	알파고 제로
가지 선택	UCT 점수	UCT 점수 + 정책 신경망의 사전 확률	UCT 점수 + 결합 신경망의 사전 확률
가지 평가	임의 경기 진행	가치 신경망 + 임의 경기 진행	결합 신경망의 가치

신경망으로 트리 탐색 안내하기

- 알파고 제로 방식 탐색 트리의 일부



신경망으로 트리 탐색 안내하기

- 게임 트리의 각 노드는 가능한 바둑판 위치를 나타낸다.
  - 이 위치에서 다음에 이어질 수가 가능한 수라는 것도 알고 있다.
  - 알고리즘은 이 중 몇 개의 다음 수를 이미 방문했지만 모든 수를 다 방문한 것은 아니다.
  - 각각 다음 수(방문한 것이든 아니든)에 대한 가지를 만든다.
    각 가지는 다음 값을 추적한다.
    - **수의 사전 확률** : 방문하기 전에 이 수가 얼마나 좋을지 나타낸다.
    - **방문수** : 트리 탐색 중 이 가지를 몇 번 방문했는지 나타낸다. 이 값은 0이 될 수도 있다.
    - **이 가지를 통과한 모든 방문에 대한 기댓값** : 이 트리의 모든 방문에 대한 평균값이다.
      이 평균값을 쉽게 갱신하려면 모든 값의 합을 저장한 후 평균을 구하기 위해 방문수로 나누면 된다.

신경망으로 트리 탐색 안내하기

- 가지 통계 추적용 구조

```python
class Branch:
    def __init__(self, prior):
        self.prior = prior
        self.visit_count = 0
        self.total_value = 0.0
```

신경망으로 트리 탐색 안내하기

- 알파고 제로 방식 탐색 트리의 노드

```python
class ZeroTreeNode:
    def __init__(self, state, value, priors, parent, last_move):
        self.state = state
        self.value = value
        self.parent = parent
        self.last_move = last_move
        self.total_visit_count = 1
        self.branches = {}

        for move, p in priors.items():
            if state.is_valid_move(move):
                self.branches[move] = Branch(p)
        self.children = {}

    def moves(self):
        return self.branches.keys()

    def add_child(self, move, child_node):
        self.children[move] = child_node

    def has_child(self, move):
        return move in self.children
```

신경망으로 트리 탐색 안내하기

- 트리 노드로부터 가지 정보를 읽어오는 헬퍼

```
class ZeroTreeNode:
...
    def expected_value(self, move):
        branch = self.branches[move]
        if branch.visit_count == 0:
            return 0.0
        return branch.total_value / branch.visit_count

    def prior(self, move):
        return self.branches[move].prior

    def visit_count(self, move):
        if move in self.branches:
            return self.branches[move].visit_count
        return 0
```

신경망으로 트리 탐색 안내하기

- 트리 따라 내려가기
  - 탐색할 때마다 우리는 트리를 따라 내려가기 시작한다.
    - 중요한 점은 어느 것이 다음에 가능한 바둑판 위치인지 알아야 한다는 거다. 그래야 좋은지 아닌지 판단할 수 있다.
    - 정확한 평가를 위해 여러분의 수에 대해 상대는 가능한 방법 중 가장 강한 방법으로 맞대응하고 있다고 가정하자.
      물론 가장 강한 대응법이 뭔지 아직 모른다. 무수한 수 중 어느 수가 좋은지 찾아내야 한다.
      여기서는 불확실성과 마주한 채로 강한 수를 선택하는 알고리즘을 설명한다.
  - 기댓값은 각 가능한 수가 얼마나 좋은지 추정한 값이다.
    - 하지만 추정 정확도가 항상 동일한 건 아니다.
      만약 특정 가지를 읽는데 더 많은 시간을 소비했다면 이 가지의 추정치는 다른 가지보다 더 정확할 것이다.
    - 가장 좋은 변화 중 하나를 더 자세히 읽을 수 있고, 이를 통해 더 정확하게 추정할 것이다.
      그래서 좀 더 적게 탐색한 가지를 더 읽어서 추정의 정확도를 높일 수 있다.
    - 처음에 생각했던 것보다 어떤 수가 더 좋을 수도 있다. 이를 증명할 방법은 이 수를 좀 더 확장해보는 것 뿐이다.
      다시 한 번 **탐험**과 **탐색**이 추구하는 방향이 서로 다른 것을 볼 수 있다.

신경망으로 트리 탐색 안내하기

- 트리 따라 내려가기
  - 원래의 MCTS 알고리즘은 UCT (Upper Confidence Bounds for Trees) 방식으로 이 두 목표의 균형을 맞춘다. UCT의 식은 다음 두 우선 순위에 대한 균형을 맞춘다.
    - 만약 가지를 여러 번 방문했다면 이 가지의 기댓값을 신뢰하는 것이다. 이 경우 추정치가 더 높은 가지를 선호한다.
    - 몇 번 방문하지 않은 가지의 경우 기댓값은 실제와 다를 것이다. 기댓값이 좋든 나쁘든 몇 번 더 방문해서 추정치의 정확도를 높이고 싶을 것이다.
  - 알파고 제로는 여기에 세번째 요소를 추가한다.
    - 거의 방문하지 않은 가지 중에서는 사전 확률이 높은 가지를 선택할 것이다. 이는 경기에 대해 정확히 살펴본 내용을 고려하기 전에 직관적으로 좋아 보이는 수다.

신경망으로 트리 탐색 안내하기

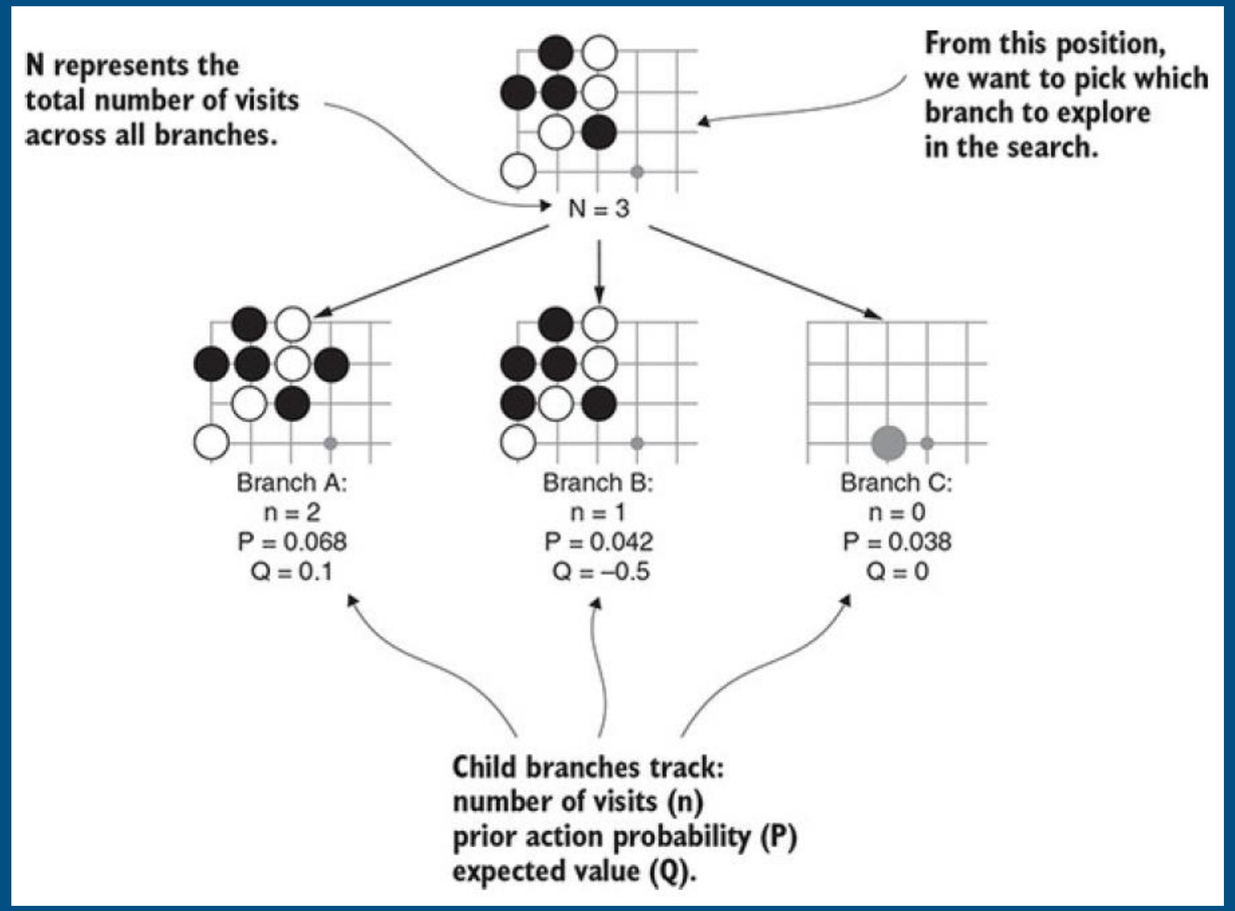
- 트리 따라 내려가기
  - 수학적으로 알파고 제로의 점수 함수는 다음과 같다.

$$Q + cP\frac{\sqrt{N}}{1+n}$$

  - Q는 가지별 모든 방문에 대한 평균 기댓값이다. (방문하지 않은 가지의 기댓값은 0이다.)
  - P는 후보 수의 사전 확률이다.
  - N은 부모 노드의 방문 횟수다.
  - n은 자식 노드의 방문 횟수다.
  - c는 탐험 대 탐색의 균형을 맞추는 요소다. (보통 이 값은 시행 착오를 거쳐 결정하게 된다.)

신경망으로 트리 탐색 안내하기

- 트리 따라 내려가기
  - 알파고 제로 트리 탐색 시 따라갈 가지 선택



	Q	n	N	P	$P\sqrt{N}/(n+1)$
가지 A	0.1	2	3	0.068	0.039
가지 B	−0.5	1	3	0.042	0.036
가지 C	0	0	3	0.038	0.065

신경망으로 트리 탐색 안내하기

- 트리 따라 내려가기
  - 자식 가지 선택하기

```
class ZeroAgent(Agent):
...
    def select_branch(self, node):
        total_n = node.total_visit_count

        def score_branch(move):
            q = node.expected_value(move)
            p = node.prior(move)
            n = node.visit_count(move)
            return q + self.c * p * np.sqrt(total_n) / (n + 1)

        return max(node.moves(), key=score_branch)
```

신경망으로 트리 탐색 안내하기

- 트리 따라 내려가기
  - 탐색 트리 따라 내려가기

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
        root = self.create_node(game_state)

        for i in range(self.num_rounds):
            node = root
            next_move = self.select_branch(node)
            while node.has_child(next_move):
                node = node.get_child(next_move)
                next_move = self.select_branch(node)
```

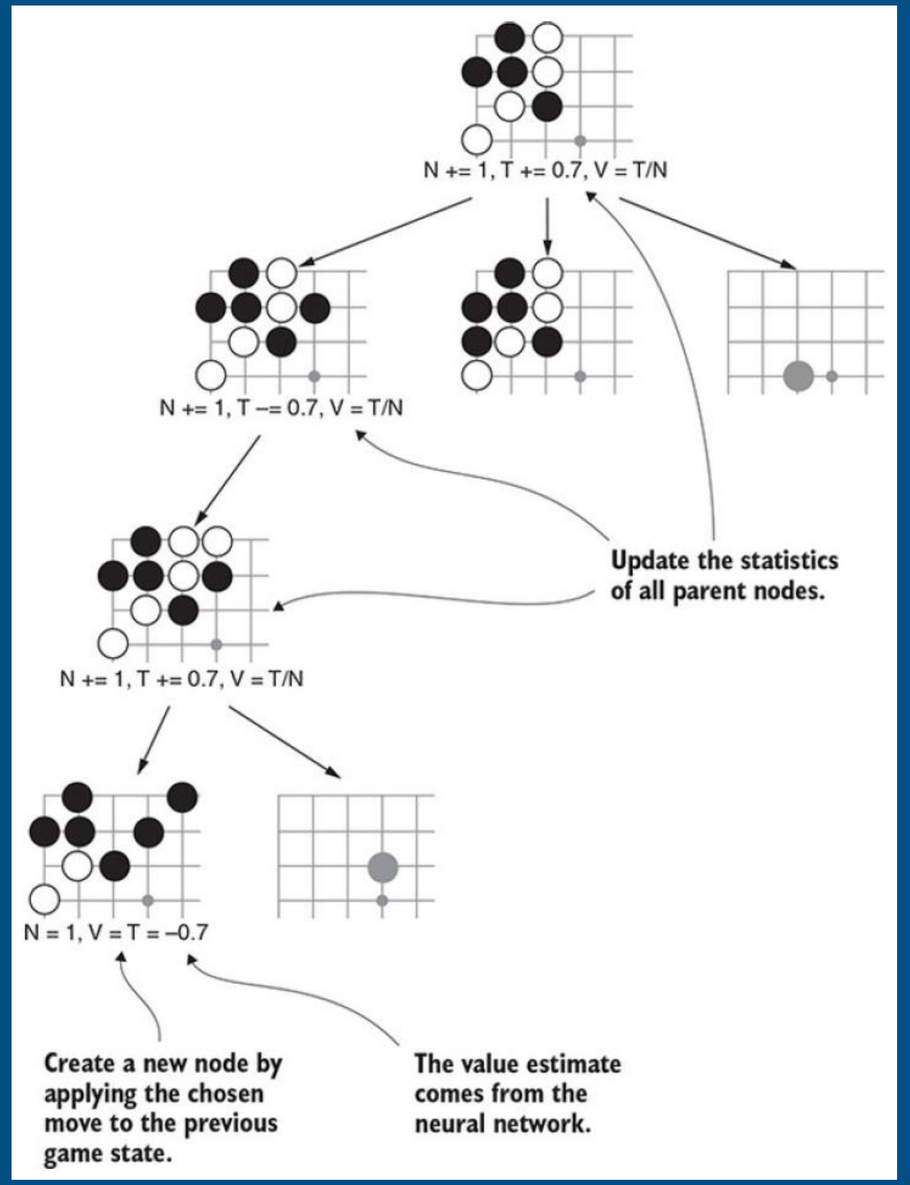
신경망으로 트리 탐색 안내하기

- 트리 확장
  - 탐색 트리에 새 노드 만들기

```
class ZeroAgent(Agent):
...
    def create_node(self, game_state, move=None, parent=None):
        state_tensor = self.encoder.encode(game_state)
        model_input = np.array([state_tensor])
        priors, values = self.model.predict(model_input)
        priors = priors[0]
        value = values[0][0]
        move_priors = {
            self.encoder.decode_move_index(idx): p
            for idx, p in enumerate(priors)
        }
        new_node = ZeroTreeNode(game_state, value, move_priors, parent, move)
        if parent is not None:
            parent.add_child(move, new_node)
    return new_node
```

신경망으로 트리 탐색 안내하기

- 트리 확장
  - 알파고 제로의 탐색 트리 확장
    - 우선 새 경기 상태를 구한다.
    - 경기 상태로부터 새 노드를 만들어서 트리에 더한다.
      그러면 신경망이 경기 상태의 추정치를 구할 것이다.
    - 마지막으로 새 노드의 부모의 값을 갱신한다.
      방문수 N에 1을 더하고 평균값 V를 갱신한다.
    - 여기서 T는 노드를 통과하는 모든 방문수다.
      이 변수는 평균 재계산을 쉽게 하기 위한 기록용이다.



신경망으로 트리 탐색 안내하기

- 트리 확장
  - 탐색 트리를 확장하고 모든 노드 통계량을 갱신

```
class ZeroTreeNode:
...
    def record_visit(self, move, value):
        self.total_visit_count += 1
        self.branches[move].visit_count += 1
        self.branches[move].total_value += value

class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
            new_state = node.state.apply_move(next_move)
            child_node = self.create_node(new_state, parent=node)

            move = next_move
            value = -1 * child_node.value
            while node is not None:
                node.record_visit(move, value)
                move = node.last_move
                node = node.parent
                value = -1 * value
```

신경망으로 트리 탐색 안내하기

- 수 선택
  - 수 선택의 가장 간단한 규칙은 가장 높은 방문수를 갖는 수를 선택하는 거다.
    그렇다면 왜 기댓값이 아닌 방문수일까?
    - 가장 많이 방문된 가지가 가장 큰 기댓값을 가질 거라고 예상할 수 있기 때문이다.
    - 이전 가지 선택 식을 떠올려보자. 가지의 방문 횟수가 늘어나면 $1/(n+1)$은 점점 줄어든다.
      따라서 가지 선택 함수는 Q값으로만 수를 선택한다. 높은 Q값을 가진 가지일수록 방문수가 크다.
    - 이제 몇 번 방문되지 않은 가지는 아무것도 할 수 없다. Q값이 작건 크건 방문수가 작으면 그 추정치를 믿을 수 없다.
      만약 Q가 큰 가지를 무작정 선택했다면 그 가지는 한 번 방문되었을 뿐이고 실젯값은 훨씬 더 작을 수도 있다.
    - 방문수를 기반으로 선택해야 하는 이유가 바로 여기에 있다.
      방문수를 기반으로 선택하면 높은 추정치와 신뢰할 만한 추정치를 갖는 가지를 선택할 수 있다.

신경망으로 트리 탐색 안내하기

- 수 선택
  - 방문수가 가장 큰 수 선택하기

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
        return max(root.moves(), key=root.visit_count)
```

신경망으로 트리 탐색 안내하기

- 수 선택
  - 자체 대국 시뮬레이션

```
def simulate_game(board_size, black_agent, black_collector, white_agent, white_collector):
    print('Starting the game!')
    game = GameState.new_game(board_size)
    agents = {
        Player.black: black_agent,
        Player.white: white_agent,
    }

    black_collector.begin_episode()
    white_collector.begin_episode()
    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)
    if game_result.winner == Player.black:
        black_collector.complete_episode(1)
        white_collector.complete_episode(-1)
    else:
        black_collector.complete_episode(-1)
        white_collector.complete_episode(1)
```

훈련

- 가치 출력의 훈련 목표는 에이전트가 이겼을 때 1, 졌을 때 -1이다.
  - 많은 경기 후 출력값의 평균을 구하면 이 극단 사이의 값을 통해 봇의 이길 확률이 얼마인지 알게 된다. (Q-학습 및 행위자-비평가 학습과 완전히 같은 설정이다.)
- 행동 출력은 다소 다르다.
  - 신경망은 정책 학습 및 행위자-비평가 학습과 마찬가지로 가능한 수 간의 확률 분포를 출력으로 만들어낸다.
    - 정책 학습에서는 에이전트가 선택한 수와 정확히 일치하도록 신경망을 훈련시켰다.
    - 알파고 제로는 다소 다르게 동작한다. 신경망이 트리 탐색 중 방문한 횟수와 일치하도록 신경망을 훈련시킨다.

훈련

- 이런 방식이 어떻게 경기 능력을 향상시킬까?
  MCTS 방식 탐색 알고리즘이 동작하는 방법을 생각해보자.
  - 그럭저럭 맞는 가치 함수를 갖고 있다고 가정하자. 이 함수는 엄청나게 정확하지는 않지만 대충 이기는 위치와 지는 위치를 구분할 정도는 된다.
  - 그리고 사전 확률은 신경 쓰지 말고 탐색 알고리즘을 실행한다.
    설계상 이 경우 가장 나은 가지를 고르는 데 더 많은 시간이 걸릴 것이다.
  - 왜냐하면 가지 선택 규칙 때문이다. UCT 식의 Q는 높은 값의 가지가 더 자주 선택되도록 되어 있다. 탐색할 시간이 무한정 있다면 어쨌든 결국 최선의 수로 수렴하게 되어 있다.

훈련

- MCTS 방식 탐색 알고리즘이 동작하는 방법을 생각해보자.
  - 트리 탐색을 충분히 많이 하면 방문수가 신뢰의 원천이라는 걸 깨닫게 될 것이다. 수를 두었을 때 어떤 일이 일어나는지 이미 확인한 후이므로 이 수가 좋은지 나쁜지 알 수 있는 것이다. 따라서 탐색 수는 사전 확률 함수 훈련의 목표치가 될 수 있다.
  - 사전 함수는 시간이 충분한 경우 트리 탐색이 어디에서 시간을 많이 소요할 지 예측한다. 이전 실행에서 훈련된 함수로 무장한 트리 탐색은 시간을 절약하고 보다 중요한 가지를 바로 찾도록 할 수 있다. 정확한 사전 함수를 사용하면 탐색 알고리즘은 몇 개의 롤아웃만 사용하지만 롤아웃을 더 많이 사용하는 느린 탐색과 비슷한 결과를 낼 것이다. 어떤 의미에서 신경망은 이전 탐색에서 무슨 일이 일어났고 건너뛰어도 되는 지식이 무엇인지 '기억한다'고 생각할 수도 있다.

훈련

- 알파고 제로의 학습에 특화된 경험 수집기

```python
class ZeroExperienceCollector:
    def __init__(self):
        self.states = []
        self.visit_counts = []
        self.rewards = []
        self._current_episode_states = []
        self._current_episode_visit_counts = []

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

    def record_decision(self, state, visit_counts):
        self._current_episode_states.append(state)
        self._current_episode_visit_counts.append(visit_counts)

    def complete_episode(self, reward):
        num_states = len(self._current_episode_states)
        self.states += self._current_episode_states
        self.visit_counts += self._current_episode_visit_counts
        self.rewards += [reward for _ in range(num_states)]

        self._current_episode_states = []
        self._current_episode_visit_counts = []
```

훈련

- 경험 수집기에 결정 전달

```
class ZeroAgent(Agent):
...
    def select_move(self, game_state):
...
            if self.collector is not None:
            root_state_tensor = self.encoder.encode(game_state)
            visit_counts = np.array([
                root.visit_count(self.encoder.decode_move_index(idx))
                for idx in range(self.encoder.num_moves())
            ])
            self.collector.record_decision(root_state_tensor, visit_counts)
```

훈련

- 결합 신경망 훈련

```python
class ZeroAgent(Agent):
...
    def train(self, experience, learning_rate, batch_size):
        num_examples = experience.states.shape[0]

        model_input = experience.states

        visit_sums = np.sum(experience.visit_counts, axis=1).reshape((num_examples, 1))
        action_target = experience.visit_counts / visit_sums

        value_target = experience.rewards

        self.model.compile(SGD(lr=learning_rate), loss=['categorical_crossentropy', 'mse'])
        self.model.fit(model_input, [action_target, value_target], batch_size=batch_size)
```

훈련

- 전체 강화학습 주기
  - 큰 자체 대국 배치를 생성한다.
  - 경험 데이터로 모델을 훈련한다.
  - 갱신된 모델과 이전 버전 간의 대국으로 테스트를 한다.
  - 새 버전이 더 강한 것으로 측정되면 새 버전으로 변경한다.
  - 아니라면 자체 대국을 더 진행한 후 다시 테스트한다.
  - 필요한 만큼 이 과정을 반복한다.

훈련

- 강화학습 과정의 단일 주기

```
# Initialize a zero agent
board_size = 9
encoder = zero.ZeroEncoder(board_size)

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

pb = board_input

# 4 conv layers with batch normalization
for i in range(4):
    pb = Conv2D(64, (3, 3), padding='same', data_format='channels_first')(pb)
    pb = BatchNormalization(axis=1)(pb)
    pb = Activation('relu')(pb)

# Policy output
policy_conv = Conv2D(2, (1, 1), data_format='channels_first')(pb)
policy_batch = BatchNormalization(axis=1)(policy_conv)
policy_relu = Activation('relu')(policy_batch)
policy_flat = Flatten()(policy_relu)
policy_output = Dense(encoder.num_moves(), activation='softmax')(policy_flat)
```

훈련

- 강화학습 과정의 단일 주기

```
# Value output
value_conv = Conv2D(1, (1, 1), data_format='channels_first')(pb)
value_batch = BatchNormalization(axis=1)(value_conv)
value_relu = Activation('relu')(value_batch)
value_flat = Flatten()(value_relu)
value_hidden = Dense(256, activation='relu')(value_flat)
value_output = Dense(1, activation='tanh')(value_hidden)

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

# Create two agents from the model and encoder.
# 10 is a very small value for rounds_per_move. To train a strong
# bot, you should run at least a few hundred rounds per move.
black_agent = zero.ZeroAgent(model, encoder, rounds_per_move=10, c=2.0)
white_agent = zero.ZeroAgent(model, encoder, rounds_per_move=10, c=2.0)
c1 = zero.ZeroExperienceCollector()
c2 = zero.ZeroExperienceCollector()
black_agent.set_collector(c1)
white_agent.set_collector(c2)
```

훈련

- 강화학습 과정의 단일 주기

```
# In real training, you should simulate thousands of games for each training batch.
for i in range(5):
    simulate_game(board_size, black_agent, c1, white_agent, c2)

exp = zero.combine_experience([c1, c2])
black_agent.train(exp, 0.01, 2048)
```

디리클레 노이즈를 사용한 탐색 향상

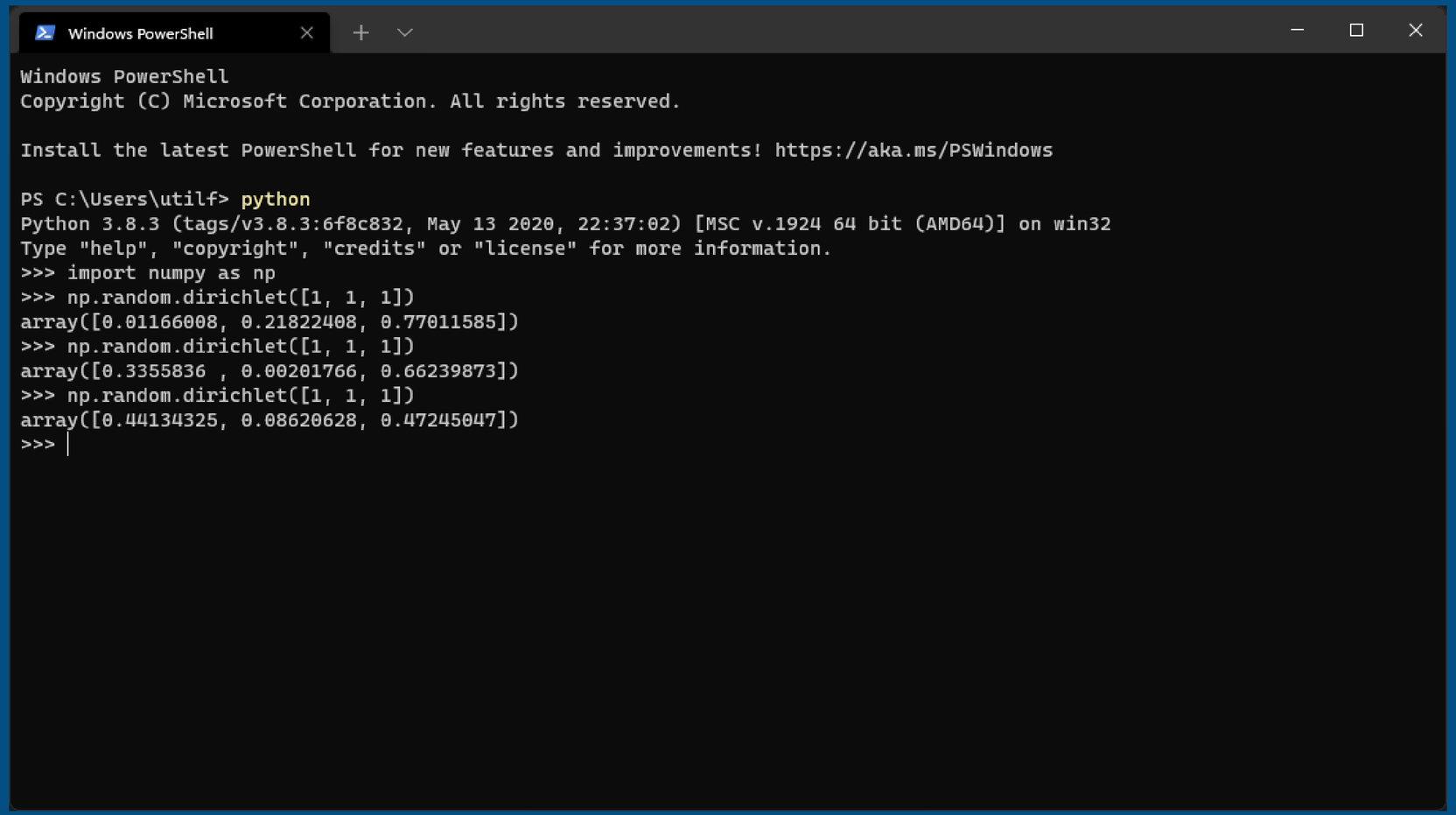
- 자체 대국 강화학습은 본질적으로 임의성을 갖는 과정이다.
  - 봇은 쉽게 이상한 방향으로 움직인다. 특히 훈련 초기에 심하다.
  - 봇이 이상한 곳에 갇혀 있는 걸 막으려면 약간의 임의성을 제공해야 한다.
    이 경우 봇이 정말 이상한 수에 갇혀 있더라도 좀 더 나은 수를 배울 가능성이 조금 생긴다.
  - 여기서는 알파고 제로가 탐험을 잘하기 위해 사용하는 방법 중 하나를 설명한다.
- 앞에서 봇의 선택에 다양성을 주는 몇 가지 기법을 사용했다.
  - 봇의 정책 결과에서 임의로 샘플링, ε-탐욕 알고리즘 등
  - 두 기법 모두 봇이 임의로 결정할 수 있는 시간을 제공한다.
  - 하지만 알파고 제로는 탐색 과정 중 다른 방식으로 임의성을 사용한다.

디리클레 노이즈를 사용한 탐색 향상

- 매 차례 1~2개의 수를 임의 선택해서 사전 확률을 높인다고 해보자.
  - 탐색 과정 초기에 사전 확률을 통해 어떤 가지를 고를지 정하기 때문에 이 수들은 추가로 방문을 받게 된다. 이 수가 나쁘다고 밝혀지면 탐색 시 빠르게 다른 가지로 이동할 것이기 때문에 문제될 게 없다.
  - 이 방식은 모든 수가 몇 회의 방문을 받게 되므로 탐색 시 사각지대가 발생하지 않는다.
- 알파고 제로는 노이즈(임의의 수 몇 개)를 탐색 트리 맨 위에 사전 확률로 추가하는 방식으로 유사한 효과를 얻는다.
  - **디리클레 분포**에서 노이즈를 만들어 앞에서 설명한 효과를 정확히 얻을 수 있다. 몇몇 수는 인공적으로 강화되고, 나머지 수는 그대로다.

디리클레 노이즈를 사용한 탐색 향상

- np.random.dirichlet()를 사용해 디리클레 분포 샘플링

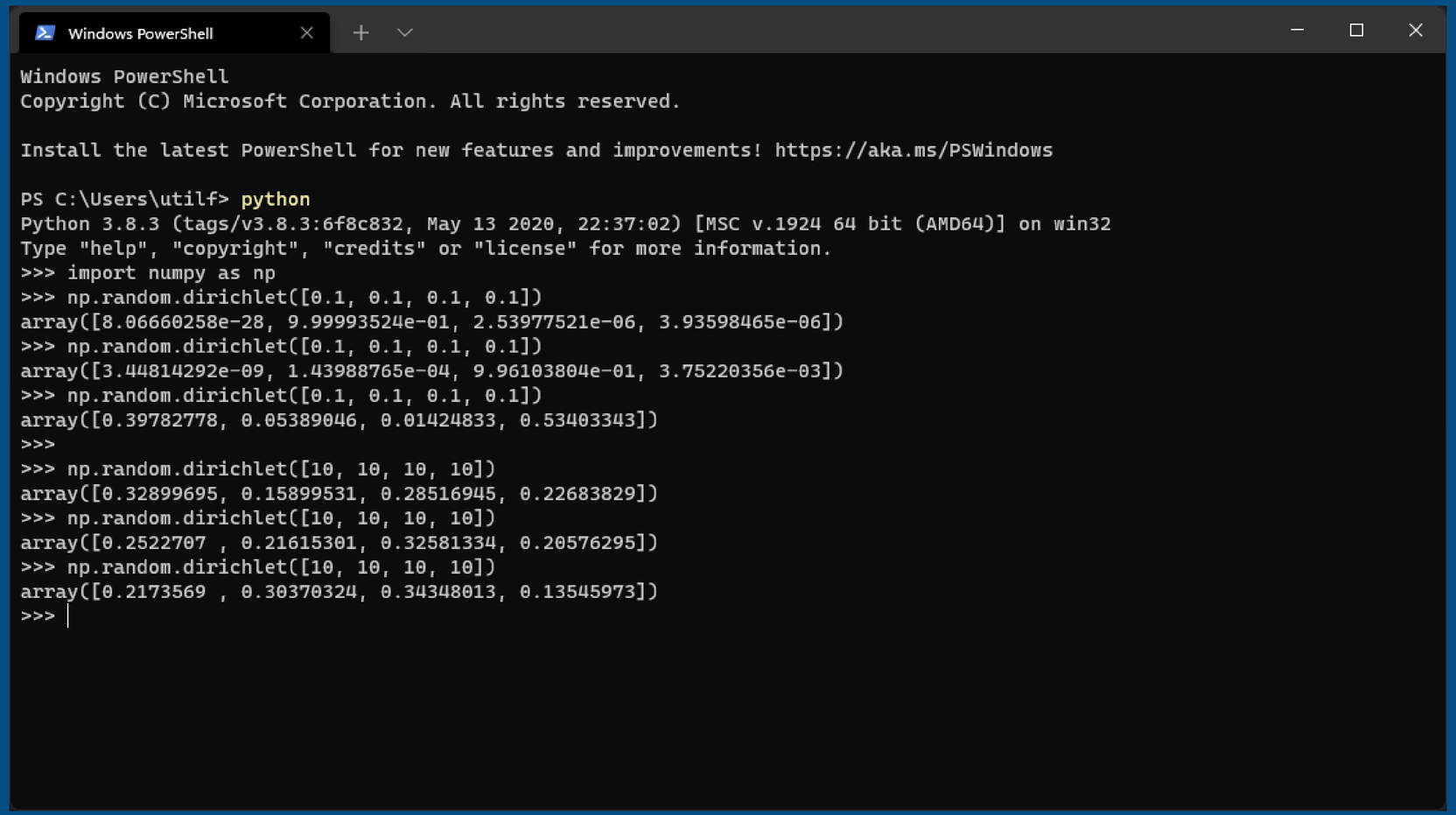


디리클레 노이즈를 사용한 탐색 향상

- 디리클레 분포 출력은 보통 α로 표기하는 **집중 파라미터 (Concentration Parameter)**를 사용해서 조절할 수 있다.
  - α가 0에 가까우면 디리클레 분포는 '울퉁불퉁한' 벡터를 만들 것이다.
    (대부분의 수가 0에 가깝고 몇 개의 큰 수만 남을 것이다.)
  - α가 크면 샘플은 '완만할' 것이다. (대부분의 값이 비슷할 것이다.)
  - 디리클레 분포를 사용해서 사전 분포를 수정할 수 있다.
    작은 α를 사용해 몇 개의 수에 높은 확률을 주고 나머지는 0에 가깝게 한다.
    그 후 실제 사전 확률에 디리클레 노이즈를 취해 가중치 평균을 낸 걸 사용하면 된다.
  - 알파고 제로는 집중 파라미터로 0.03을 사용했다.

디리클레 노이즈를 사용한 탐색 향상

- α가 0에 가까울 때의 디리클레 분포 예



더 깊은 신경망을 만드는 기법

- 배치 정규화
  - 심층 신경망은 각 계층이 점점 더 높은 수준의 원시 데이터 표현형을 학습할 수 있다는 개념으로 만들어졌다. 여기서 표현형이란 정확히 무엇인가?
    - 원시 데이터의 의미 있는 특징은 층의 특정 뉴런이 활성화됨에 따라 특정 숫자로 나타난다.
    - 하지만 실제 숫자 간의 연결은 완전히 제멋대로다.
      예를 들어, 특정 층의 모든 활성화된 값에 2를 곱해도 정보가 손실되지 않는다. 단지 크기가 달라질 뿐이다.
      원칙적으로 이런 변형은 신경망의 학습량에 영향을 미치지 않는다.

더 깊은 신경망을 만드는 기법

- 배치 정규화
  - 하지만 이런 활성화된 값의 절댓값은 실제 훈련 성능에 영향을 미칠 수 있다.
    배치 정규화(Batch Normalization)란 각 층의 활성화된 값 분포의 중심값을 0으로 하고 분산이 1이 되도록 값을 이동하는 기법을 말한다.
    - 훈련을 시작할 때는 활성화 함수가 어떻게 생겼는지 모른다.
    - 배치 정규화를 사용하면 훈련 중에 바로 데이터를 올바르게 조절할 수 있다.
      정규화 변환은 훈련 중에 입력을 바로 변경할 수 있다.

더 깊은 신경망을 만드는 기법

- 배치 정규화
  - 배치 정규화가 어떻게 훈련 성능을 높일 수 있을까? 여전히 연구가 진행중이다.
    - 기존 연구자들은 공변량이 변하는 걸 줄이기 위해 배치 정규화를 만들었다.
    - 어떤 층에서든 훈련 도중 활성화값이 움직이곤 한다.
      배치 정규화를 사용하면 이런 움직임을 수정해서 이후 층의 학습 부담을 덜어준다.
    - 하지만 최근의 연구에서는 공변량이 변하는 건 처음에 생각했던 것만큼 중요하지 않다고 하고 있다.
      대신 배치 정규화를 취한 값을 사용하면 손실 함수 값의 변동이 적어진다는 걸 알게 되었다.
    - 왜 배치 정규화가 도움이 되는지에 대해서는 여전히 연구 대상이지만 도움이 된다는 건 확실하다.
      TensorFlow 2에서는 신경망에 추가할 수 있는 `BatchNormalization` 층을 제공한다.

더 깊은 신경망을 만드는 기법

- 배치 정규화
  - TensorFlow 2 신경망에 배치 정규화층 추가

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, BatchNormalization, Conv2D

model = Sequential()
model.add(Conv2D(64, (3, 3), data_format='channels_first'))
model.add(BatchNormalization(axis=1))
model.add(Activation('relu'))
```

더 깊은 신경망을 만드는 기법

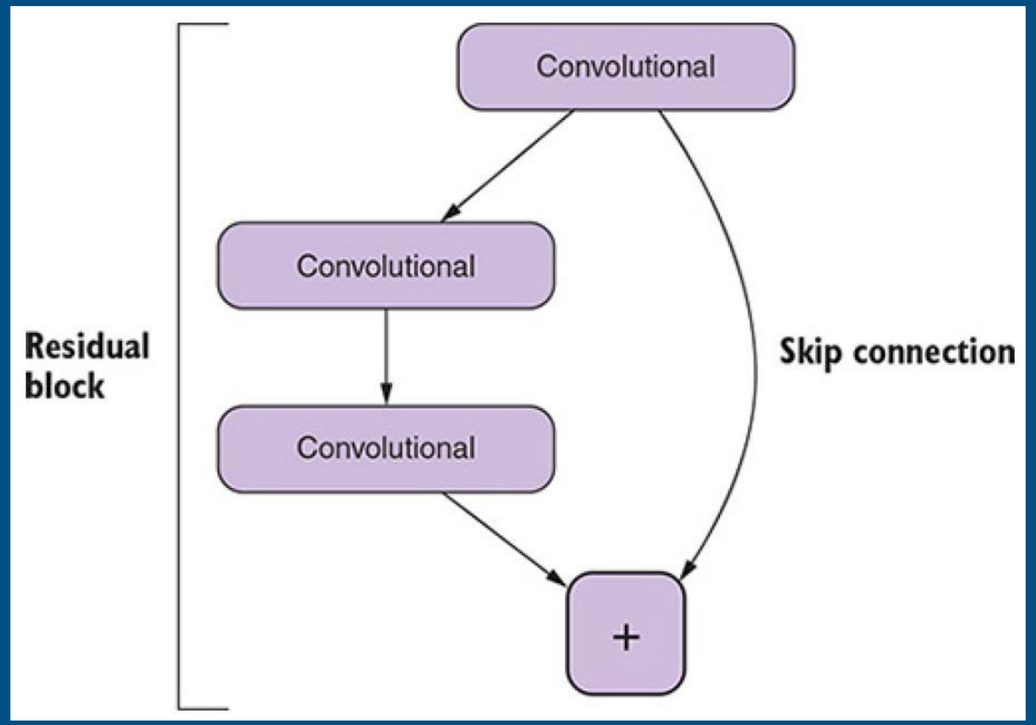
- 잔차 신경망
  - 세 은닉층이 가운데에 들어 있는 신경망을 잘 훈련시켰다고 해보자.
    여기에 4번째 층을 더하면 어떻게 될까?
    - 이론적으로는 신경망의 용량이 늘어날 것이다.
    - 하지만 최악의 경우 네 층의 신경망을 훈련할 때 앞의 세 층은 기존 3층 구조 신경망이었을 때 학습했던 걸 그대로 학습하고, 4번째 층은 입력값에 전혀 손을 대지 않은 채 넘기게 된다.
    - 이렇게 변경했을 때는 학습을 **더**하려는 것이지, **덜**하려는 건 아니었을 것이다.
      결국 더 깊은 신경망은 과적합 상태에 빠질 것이다.

더 깊은 신경망을 만드는 기법

- 잔차 신경망
  - 하지만 실제로는 이런 일이 항상 생기는 건 아니다.
    - 4층 신경망을 훈련할 때는 3층 신경망 때보다 데이터 나열 방법이 더 많다.
    - 가끔 복잡한 손실 함수 평면에서 확률적 경사하강법을 사용하다 골짜기에 빠질 수 있다.
      이런 경우 층을 더 더하면 과적합이 발생하지 않을 수 있다.
  - **잔차 신경망(Residual Network)**은 추가 층이 학습해야 할 걸 단순화하는 기법이다.
    - 세 개의 층으로 문제를 학습하는 데 무리가 없다면 4번째 층은 처음 세 층이 학습한 출력값과 목푯값 사이의 차이를 학습하는 데 초점을 맞추도록 할 수 있다. (이 차이를 **잔차(Residual)**라고 한다.)

더 깊은 신경망을 만드는 기법

- 잔차 신경망
  - 잔차 신경망 블록



스터디를 마치며

- 머신러닝을 사용한 최신 프로그램들을 보면 안에서 무슨 일이 일어나는지 이해할 수 있는 틀이 머릿속에 만들어질 거다. 다음을 생각해보자.
  - 모델은 무엇이고, 신경망 구조는 무엇일까?
  - 손실 함수와 훈련 목표는 무엇일까?
  - 훈련 과정은 어떻게 될까?
  - 입력값과 출력값은 어떻게 변환될까?
  - 기존 알고리즘이나 실제 소프트웨어 구현 단계에서 모델은 어떻게 적합화될까?

감사합니다!

스터디 듣느라 고생 많았습니다.