2019 OSS 개발자 포럼 겨울 캠프

AlphaZero - Day 2

옥찬호

Nexon Korea, Microsoft MVP

utilForever@gmail.com

오늘 다룰 내용

2019 OSS Winter
AlphaZero - Day 2

- 신경망 구조 설계
  - 입력 데이터 전처리
  - 신경망 구조
- MCTS 변형
- 학습 시키기
  - 학습 데이터 만들기
  - 학습

PyTorch 설치

https://pytorch.org/get-started/locally/

PyTorch Build	**Stable (1.3)**		Preview (Nightly)		
Your OS	Linux	Mac	**Windows**		
Package	Conda	**Pip**	LibTorch	Source	
Language	Python 2.7	Python 3.5	**Python 3.6**	Python 3.7	C++
CUDA	9.2	10.1	**None**		
Run this Command:	`pip3 install torch==1.3.1+cpu torchvision==0.4.2+cpu -f https://download.pytorch.org/whl/torch_stable.html`				

Tensor 자료형

torch.Tensor

- 다차원의 행렬을 표현할 때 사용되는 numpy의 ndarray와 비슷한 자료형.
- GPU를 효율적으로 사용할 수 있다.

Tensor 자료형

torch.empty, torch.zeros 등을 이용해서 tensor를 만들 수 있다.

```
x = torch.empty((1, 2))
y = torch.zeros((1, 2))
print(x)
print(y)
```

```
tensor([[4.2141e-36, 0.0000e+00]])
tensor([[0., 0.]])
```

Tensor 자료형

torch.tensor를 이용해 파이썬 변수를 Tensor로 변환할 수 있다.

```
x = torch.tensor(1.)
y = torch.tensor([0., 1., 2., 3.])
print(x, y)
```

```
tensor(1.) tensor([0., 1., 2., 3.])
```

Tensor 자료형

torch.cuda.is_available()가 True일 경우, GPU로 Tensor를 옮겨 연산 속도를 높일 수 있다.

방법 1

```
x = torch.zeros((1, 2))
x = x.cuda()
print(x)
```

```
tensor([[0., 0.]], device='cuda:0')
```

방법 2

```
x = torch.zeros((1, 2))
device = torch.device('cuda')
x = x.to(device)
print(x)
```

```
tensor([[0., 0.]], device='cuda:0')
```

Tensor 자료형

사칙연산, 내적 등의 연산을 할 수 있다.

```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[5, 6], [7, 8]])
print(x + y)               # 방법 1
print(torch.add(x, y))     # 방법 2
```

```
tensor([[ 6,  8],
        [10, 12]])
tensor([[ 6,  8],
        [10, 12]])
```

```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[5, 6], [7, 8]])
print(x * y)               # 방법 1
print(torch.mul(x, y))     # 방법 2
```

```
tensor([[ 5, 12],
        [21, 32]])
tensor([[ 5, 12],
        [21, 32]])
```

Tensor 자료형

값이 하나뿐이라면, item을 이용해 파이썬 정수로 바꿀 수 있다.

```
x = torch.tensor([[1.]])
n = x.item()
print(n)
```

```
1.0
```

Tensor 자료형

view를 이용해 Tensor의 shape를 바꿀 수 있다.

```
x = torch.tensor([1., 2., 3., 4.])
y = x.view(-1, 2)
print(x)
print(y)
```

```
tensor([1., 2., 3., 4.])
tensor([[1., 2.],
        [3., 4.]])
```

Tensor 자료형

Tensor의 backward로, requires_grad=True인 Tensor들이 포함된 연산들에 대해 Backpropagation을 할 수 있다.

```
w = torch.tensor([1., 2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
x = torch.tensor([[1., 1.]])
p = w * x + b
p = p.mean()
p.backward()
print(p.grad_fn, w.grad)
```

```
<MeanBackward0 object at 0x7f47478c30f0> tensor([0.5000, 0.5000])
```

torch.nn.Module

torch.nn에 있는 Module 클래스는 복잡한 신경망 구현을 효율적으로 할 수 있게 도와준다.

이후 슬라이드에서 다룰 클래스들은 모두 Module의 서브클래스이다.

```
from torch import nn
```

torch.nn.Linear

torch.nn.Linear(in_features, out_features, bias=True)

$$\text{Linear}(x) = W^{\mathrm{T}}x + b$$

```
m = nn.Linear(2, 1)
input = torch.tensor([[1.0, 1.0]])
output = m(input)
print(m.weight)
print(m.bias)
print(output)
```

```
Parameter containing:
tensor([[-0.6183, -0.1833]]
Parameter containing:
tensor([-0.3479], requires_
tensor([[-1.1494]], grad_fn
```

torch.nn.ReLU

torch.nn.ReLU(inplace=False)

$$\mathrm{ReLU}(x) = \max(0, x)$$

inplace=True일 경우, input으로 들어온 Tensor를 수정한다.

```
m = nn.ReLU()
input = torch.tensor([1., -1.])
output = m(input)
print(output)


tensor([1., 0.])
```

```
m = nn.ReLU(inplace=True)
input = torch.tensor([1., -1.])
m(input)
print(input)


tensor([1., 0.])
```

torch.nn.Tanh

torch.nn.Tanh()

$$\mathrm{Tanh}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

```
m = nn.Tanh()
input = torch.tensor([1., -1.])
output = m(input)
print(output)
```

```
tensor([ 0.7616, -0.7616])
```

torch.nn.Conv2d

torch.nn.Conv2d(in_channels, out_channels, kernel_size, padding=1)

```
m=nn.Conv2d(33, 12, 4, padding=2)
input = torch.randn(5, 33, 15, 15)
output = m(input)
print(input.shape, output.shape)
```

```
torch.Size([5, 33, 15, 15]) torch.Size([5, 12, 16, 16])
```

torch.nn.Sequential

torch.nn.Sequential(*args)

args에 전달된 순서대로 Module을 연결해 새로운 Module을 만든다

```
model = nn.Sequential(
      nn.Conv2d(1, 20, 5),
      nn.ReLU(),
      nn.Conv2d(20, 64, 5),
      nn.ReLU()
)
```

```
input = torch.randn(5, 1, 15, 15)
output = model(input)
print(input.shape)
print(output.shape)
```

```
torch.Size([5, 1, 15, 15])
torch.Size([5, 64, 7, 7])
```

사용자 정의 Module

이전 슬라이드에서 다루었던 Module들 보다 더 복잡한 모델을 원한다면, Module의 서브클래스를 정의하면 된다.

```
class Network(nn.Module):
  def __init__(self, input_dim):
    super(Netword, self).__init__()
    self.linear = nn.Linear(input_dim, 128)
    self.relu = nn.ReLU()
  def forward(self, x):
    return self.relu(self.linear(x))
```

parameters

Module의 parameters를 이용하면, Module에 있는 Tensor중에서 학습시킬 수 있는(requires_grad=True) Tensor를 추출할 수 있다.

```
m = nn.Linear(2, 1)
for p in m.parameters():
  print(p)
```

```
Parameter containing:
tensor([[-0.5406, -0.2133]], requires_grad=True)
Parameter containing:
tensor([0.4808], requires_grad=True)
```

torch.nn.functional

softmax, mse_loss 등 필요한 함수들이 구현되어 있는 패키지

보통 F라는 이름으로 import해서 사용한다.

```
import torch.nn.functional as F
```

F.log_softmax

F.log_softmax(input, dim=None)

$$\text{log\_softmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

```
input = torch.randn(2, 3)
pi = F.log_softmax(input, dim=1)
print(pi)
```

```
tensor([[-2.7289, -0.1215, -3.0138],
        [-2.5390, -0.3097, -1.6746]])
```

F.mse_loss

F.mse_loss(input, target)

$$\text{mse\_loss}(x, y) = \frac{\sum_N (x_i + y_i)^2}{N}$$

```
input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
loss = F.mse_loss(input, target)
loss.backward()
```

torch.optim

다양한 최적화 알고리즘이 구현되어 있는 패키지

1. optimizer.zero_grad()로 각 Tensor의 grad를 초기화한다.
2. loss.backward()로 grad를 업데이트한다.
3. optimizer.step()으로 grad를 이용해 Tensor를 업데이트한다.

torch.optim.SGD

torch.optim.SGD(params, lr, momentum=0, weight_decay=0)

stochastic gradient descent가 구현되어 있는 클래스이다.

옵션으로 momentum, weight_decay를 추가할 수 있다.

torch.optim.SGD


```
model = nn.Linear(8, 16)
opt = optim.SGD(model.parameters, lr=0.1, momentum=0.9)
opt.zero_grad()
input = torch.randn(8)
output = model(input)
target = torch.randn(16)
loss = F.mse_loss(output, target)
loss.backward()
opt.step()
```

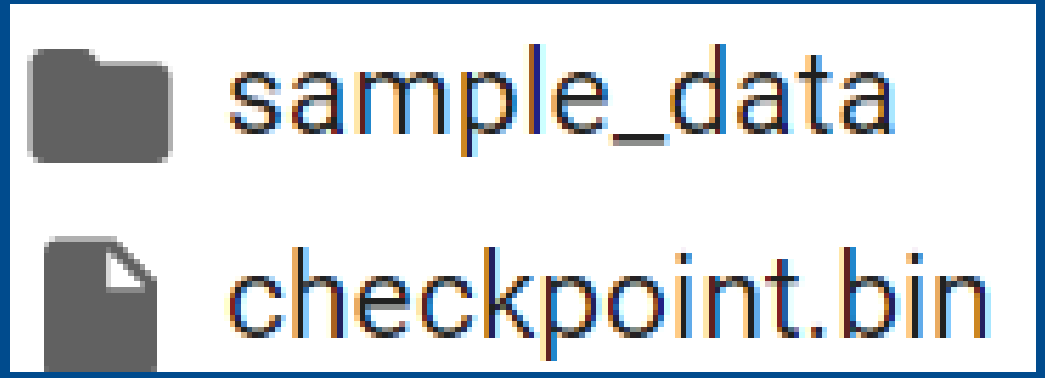
torch.save

torch.save(obj, f)

obj를 직렬화하여 f라는 이름으로 저장한다.

model을 저장하려면 obj에 model.state_dict()를 넣어주면 된다.

```
torch.save(model.state_dict(), 'checkpoint.bin')
```



torch.load

torch.load(f)

f라는 이름의 파일을 불러와 역직렬화한다.

model.state_dict()를 저장했으면, 불러올 때는
model.load_state_dict를 이용하면 된다.

```
model.load_state_dict(torch.load('checkpoint.bin'))
```

```
<All keys matched successfully>
```

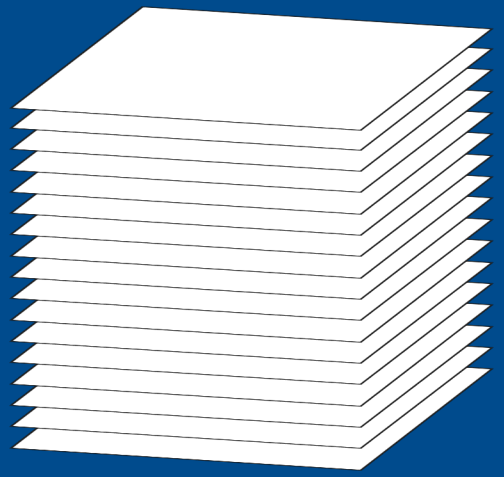
입력 데이터 전처리

오목판의 상태를 신경망에 넣어 주기 위한 처리를 해야 한다.
→ 이를 전처리(Preprocess)라 부른다.

입력 데이터의 모양은 $15 \times 15 \times 17$ 이다.

- X_t : 나의 돌 위치 정보
- Y_t : 상대의 돌 위치 정보
- C : 나의 돌 색상 정보 (흑이면 전부 1, 백이면 전부 0)

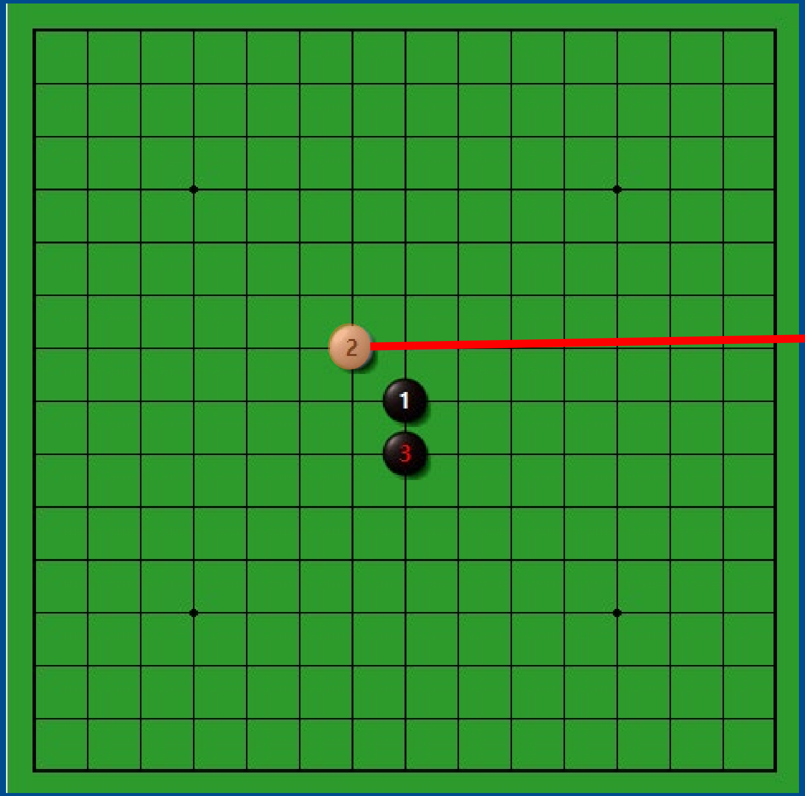
→ 입력 데이터 $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \cdots, X_{t-7}, Y_{t-7}, C]$



입력 데이터 전처리

2019 OSS Winter
AlphaZero - Day 2

돌의 위치 정보



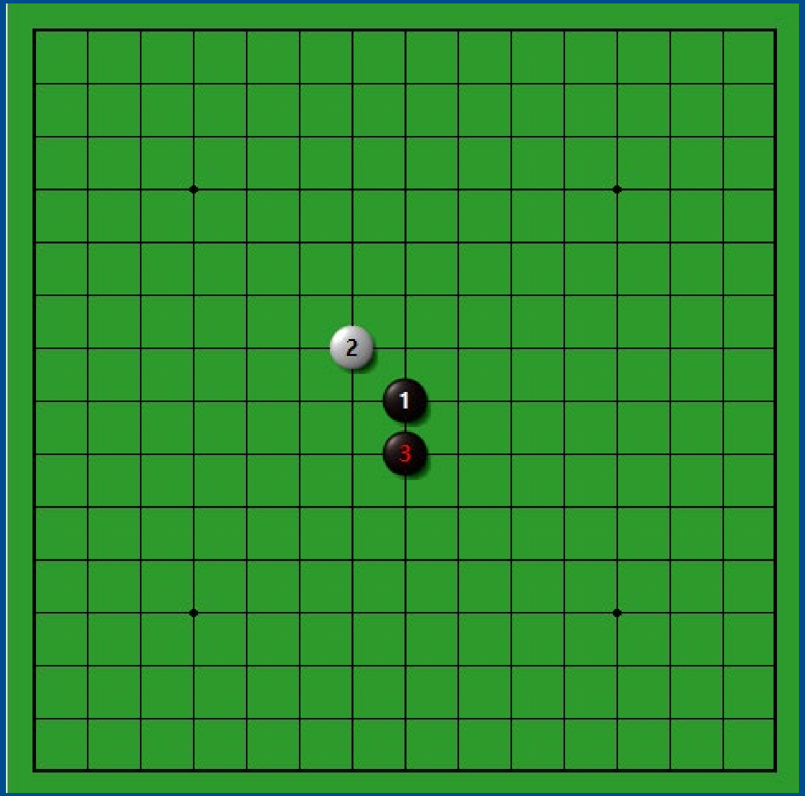
백이 둘 차례

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0번 채널

입력 데이터 전처리

돌의 위치 정보



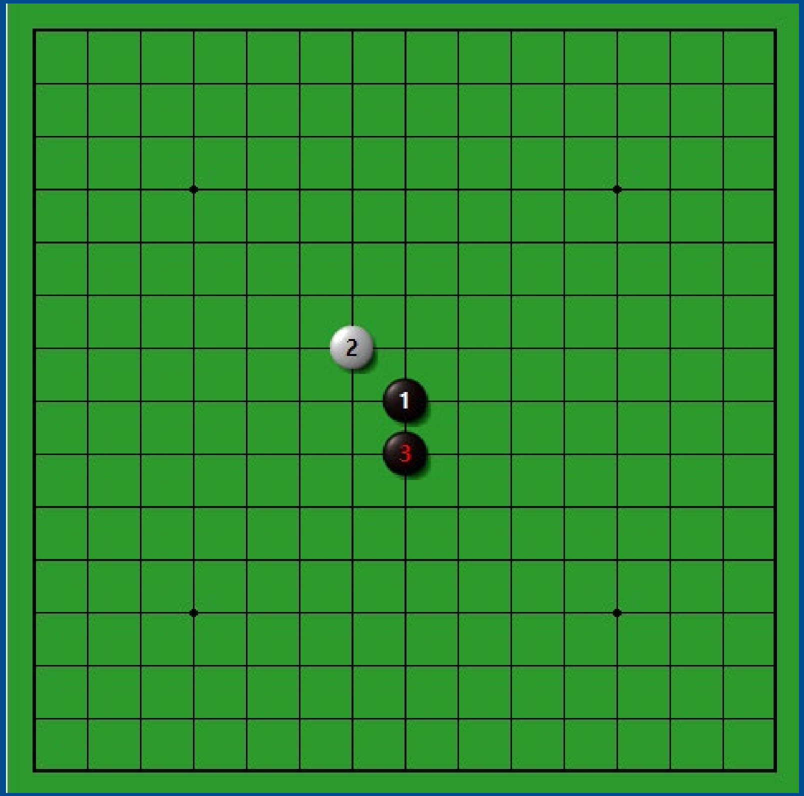
백이 둘 차례

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1번 채널

입력 데이터 전처리

돌의 위치 정보



백이 둘 차례

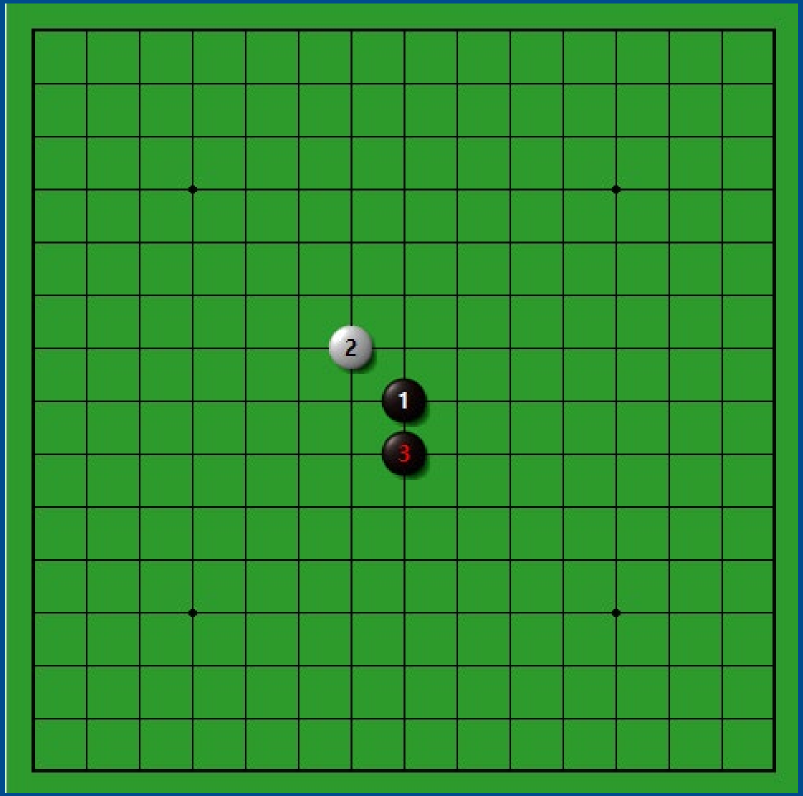
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3번 채널

입력 데이터 전처리

2019 OSS Winter
AlphaZero - Day 2

돌의 위치 정보



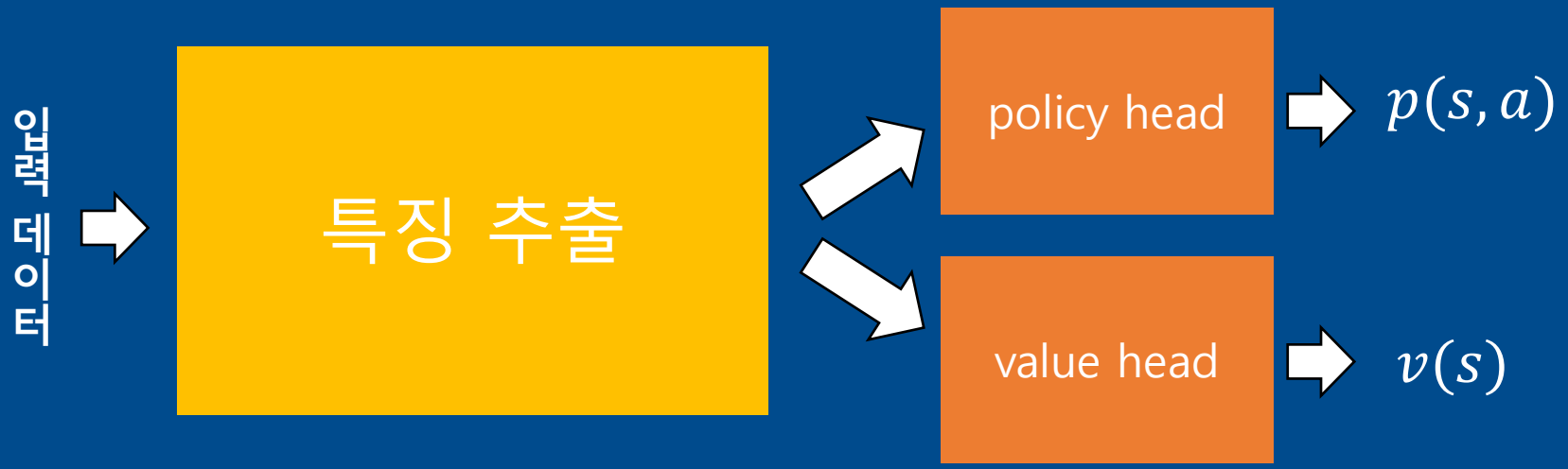
백이 둘 차례

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4번 채널

신경망 구조 설계

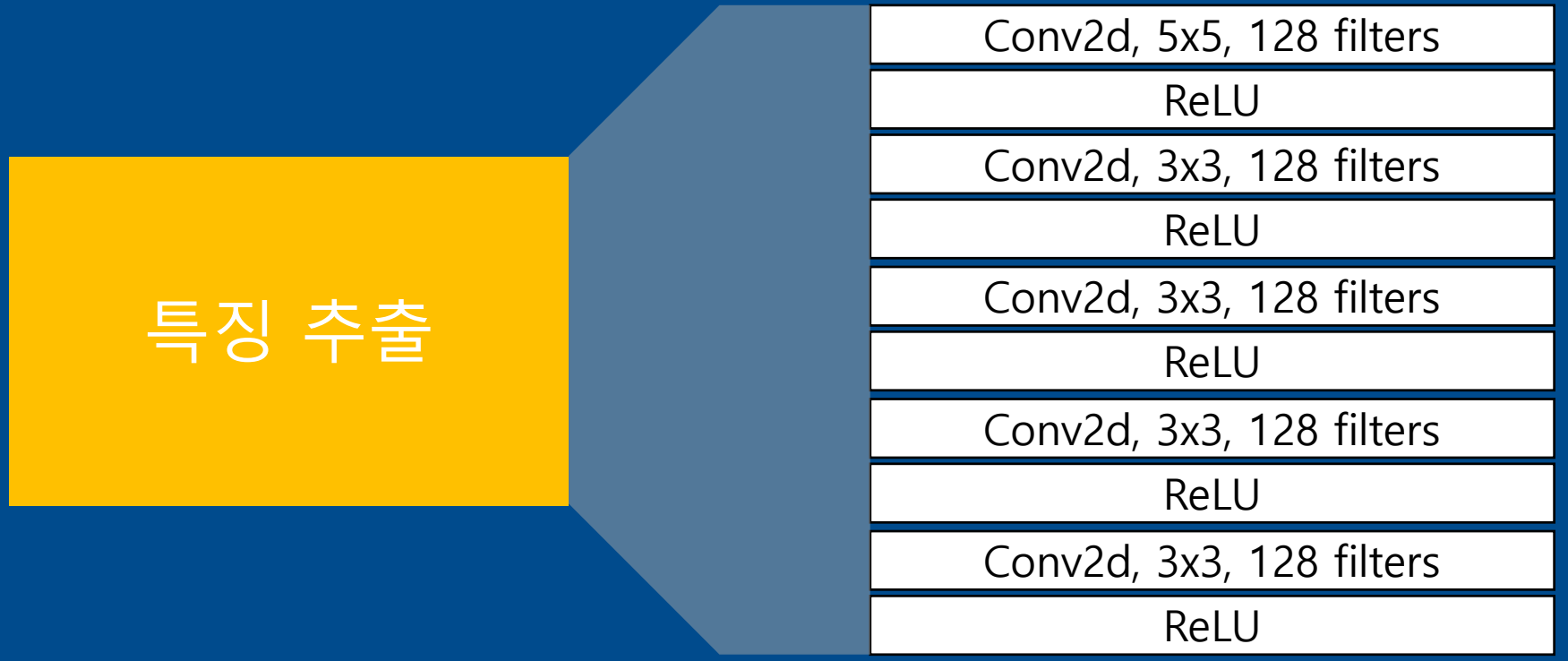
전처리한 입력 데이터를 받아 정책(policy)과 가치(value)를 추정하는 신경망을 만들어야 한다.



신경망 구조 설계

특징 추출

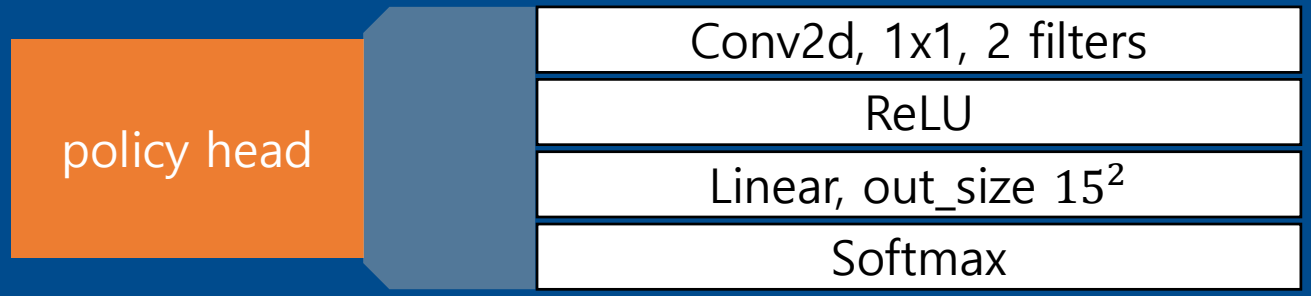
- 오목 판은 가로 세로 15픽셀의 이미지로 볼 수 있다.
- 이미지에서 특징을 추출할 때 사용하는 Convolutional Neural Network(CNN)를 사용.
- 최종적으로 $15 \times 15 \times 128$ 크기의 특징맵(feature map)이 나온다.



신경망 구조 설계

Policy Head

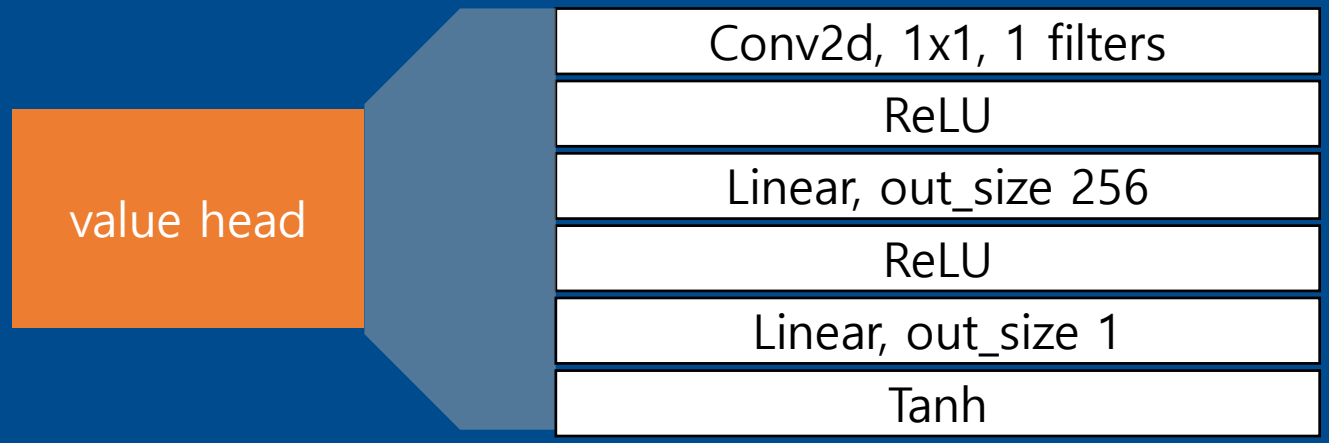
- 추출된 특징맵에서 정책을 구하는 부분.
- 판 위의 모든 점에 대한 정책과 패스할 정책을 합쳐 크기가 256인 벡터가 나온다.



신경망 구조 설계

Value Head

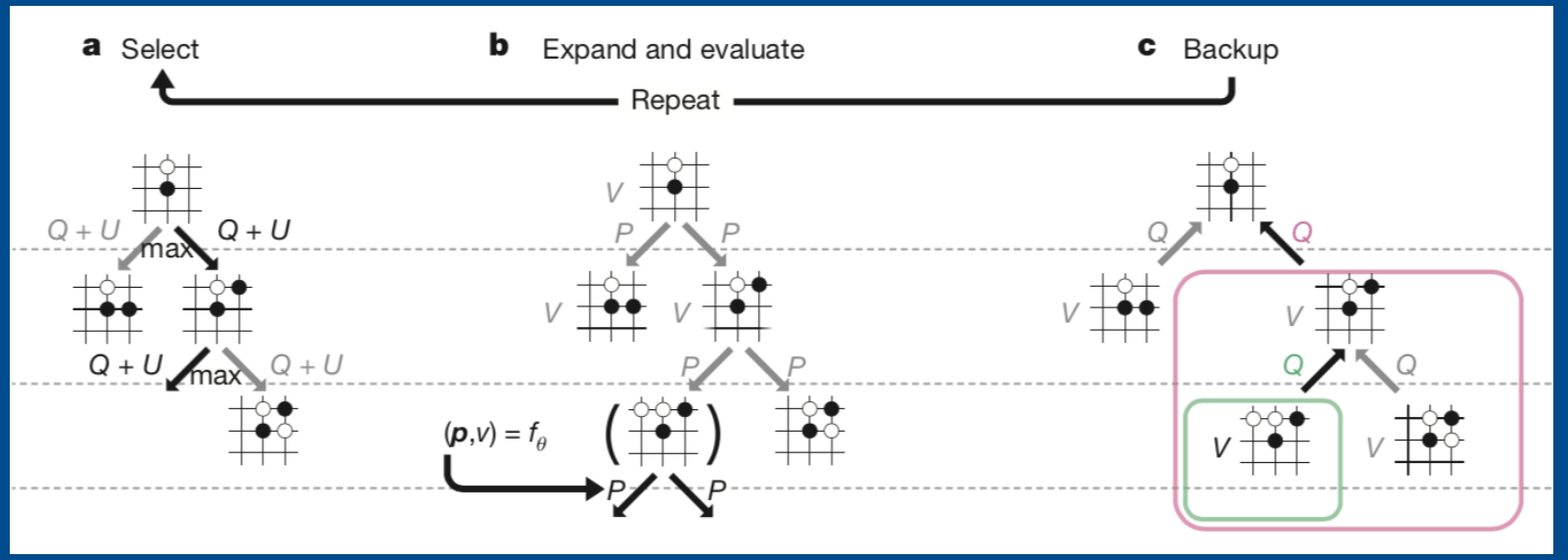
- 추출된 특징맵에서 가치를 구하는 부분.
- 크기가 1인 승률이 나온다.



MCTS 변형

AlphaZero는 변형된 MCTS를 사용한다.

- PUCT(Polynomial Upper Confidence Trees) 알고리즘 사용
- Rollout 제거



MCTS 변형

PUCT 알고리즘

탐색할 수는 다음 수식에 의해 결정한다.

$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$

이때 $Q(s_t, a)$는 평균 승률로 다음과 같이 정의된다.

$$Q(s_t, a) = \frac{W(s_t, a)}{N(s_t, a)}$$

MCTS 변형

PUCT 알고리즘

$U(s_t, a)$는 다음과 같이 정의된다.

$$U(s_t, a) = C(s_t)P(s_t, a)\frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)}$$

그리고 이전엔 상수였던 temperature가 다음과 같은 함수 $C(s_t)$로 바뀌었다.

$$C(s_t) = \log_e\left(\frac{1 + N(s_t, a) + c_{base}}{c_{base}}\right) + c_{init}$$

MCTS 변형

Alpha Zero에선 Rollout을 제거하고, 대신 승률 추정만 한다.

따라서 Backup 단계에서 각 노드의 값은 다음과 같이 변한다.

$$N(s_t, a) = N(s_t, a) + 1$$
$$W(s_t, a) = W(s_t, a) + v$$

학습 데이터 만들기

학습 데이터는 상태(s), 정책(π), 게임 결과(z)의 쌍으로 구성된다.

정책은 MCTS에서 선택한 수를 최적의 수로 가정하여 학습한다.

→ 정책 $\pi(s,a) = \frac{N(s,a)}{\sum_b N(s,b)}$

가치는 게임의 결과를 예측하도록 학습한다.

→ $z = \begin{cases} 1 & (if\ win) \\ 0 & (if\ draw) \\ -1 & (if\ lose) \end{cases}$

학습 데이터 만들기

강화학습에서 탐험은 매우 중요하고, 현재 모델에 과적합(overfitting)되는 걸 막기 위해 다음과 같이 noise를 섞는다.

$$P(s,a) = (1-\epsilon)p_a + \epsilon\eta_a \text{ where } \eta \sim \mathrm{Dir}(\alpha)$$

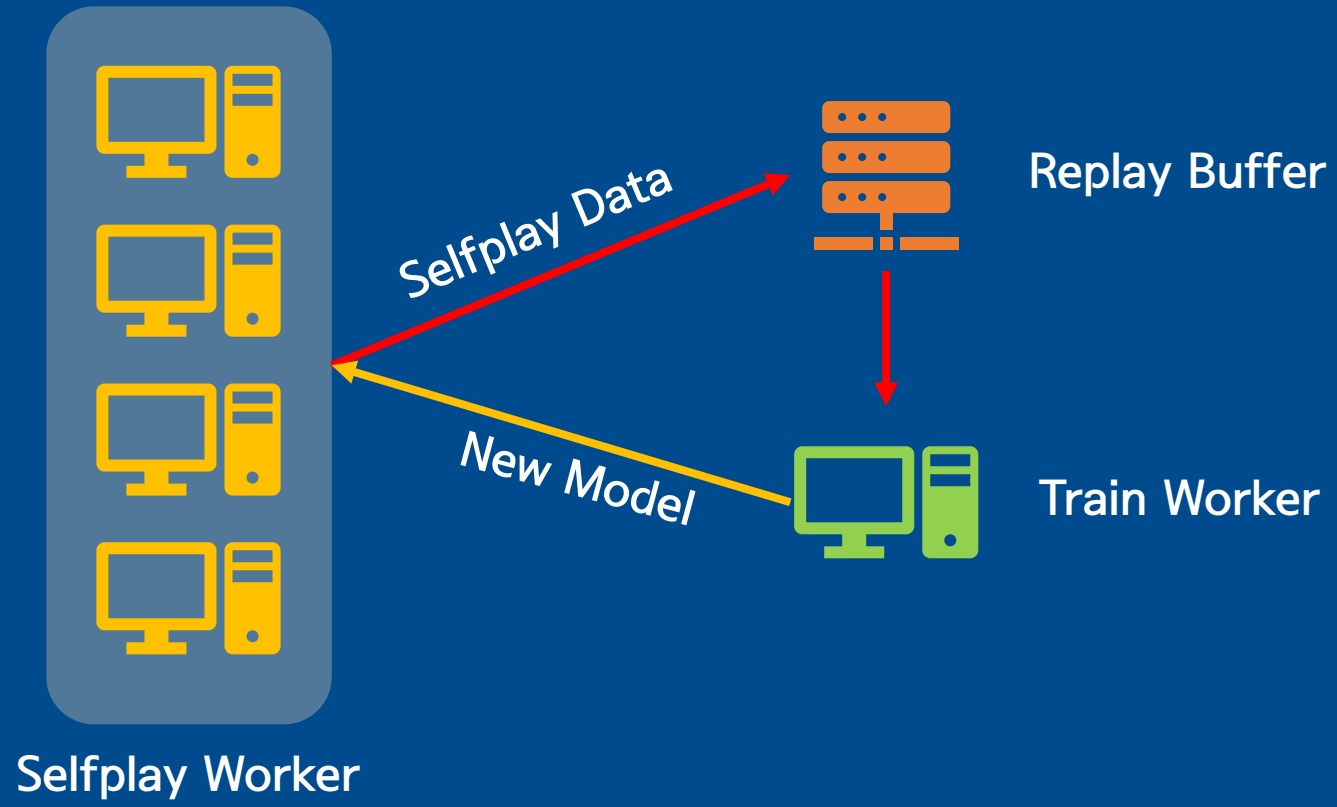
$\mathrm{Dir}(\alpha)$는 python에서 다음과 같이 구할 수 있다.

```
import numpy

alpha = 0.03
noise = numpy.random.dirichlet([alpha] * 256)
```

학습 시키기

만들어진 학습 데이터는 Replay Buffer에 저장된다.



학습 시키기

- Selfplay Worker
  - 학습데이터를 만든다.
  - Train Worker에서 학습된 모델을 받아 학습 데이터를 만든다.

- Replay Buffer
  - Selfplay Worker에서 만들어진 데이터를 저장하는 곳이다.

- Train Worker
  - Replay Buffer에서 데이터를 가져와 새로운 신경망을 학습한다.

학습 시키기

신경망(θ)에선 다음의 출력이 나온다.

$$f_\theta(s) = (\boldsymbol{p}, v)$$

위 신경망을 다음의 loss function으로 최적화한다.

$$L = (z - v)^2 - \boldsymbol{\pi}^\top \log \boldsymbol{p} + c\|\theta\|^2$$

감사합니다

http://github.com/utilForever

질문 환영합니다!