

QuantGANs & DDPG

Hyelin Choi

Department of Mathematics

Sungkyunkwan University

May 21, 2024

Notation

Adjusted closing price 조정 종가



The closing price of i^{th} asset after period t is $v_{i,t}^{close}$.

종가

The closing price of all assets comprise the price vector for period t as v_t^{close} .

Notation

$$y_t = \frac{\mathbf{v}_t}{\mathbf{v}_{t-1}} = \left(1, \frac{v_{1,t}}{v_{1,t-1}}, \dots, \frac{v_{m,t}}{v_{m,t-1}}\right)^T$$

is the price fluctuating vector

$$\mathbf{w}_{t-1} = (w_{0,t-1}, w_{1,t-1}, \dots, w_{m,t-1})^T$$

represents the reallocated weight at the end of time $t - 1$
with constraint $\sum_i w_{i,t-1} = 1$
실제 배분 비율 (자산 보유 비율)

$$a_{t-1} = (a_{0,t-1}, a_{1,t-1}, \dots, a_{m,t-1})^T$$

represents the desired allocating weight at the end of time $t - 1$
with constraint $\sum_{i=0}^n a_{i,t-1} = 1$
목표 배분 비율

- Reward(r): the naive fluctuation of wealth minus transaction cost. The fluctuation of wealth is $a_{t-1}^T \cdot y_{t-1}$. In the meanwhile, transaction cost should be subtracted from that, which equals $\mu \sum_{i=1}^m |a_{i,t-1} - w_{i,t-1}|$. The equation above suggests that only transactions in stocks occur transaction cost. Specifically, we set $\mu = 0.25\%$. In conclusion, the immediate reward at time $t-1$ as:

$$r_t(s_{t-1}, a_{t-1}) = \log(\underbrace{\mathbf{a}_{t-1} \cdot \mathbf{y}_{t-1}}_{\text{fluctuation of wealth}} - \underbrace{\mu \sum_{i=1}^m |a_{i,t-1} - w_{i,t-1}|}_{\text{transaction cost}}) + \text{epsilon } (=10\text{e-}8)$$

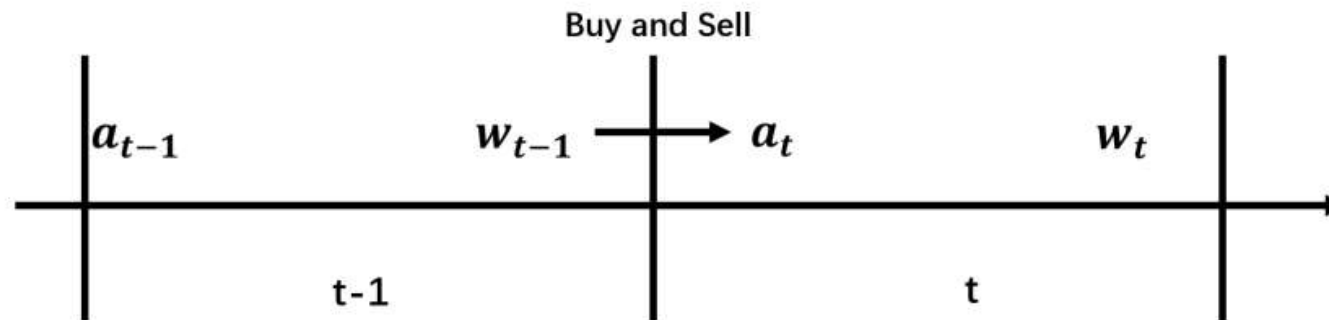
- State(s): one state includes previous open, closing, high, low price, volume or some other financial indexes in a fixed window.

Adjusted close

[*****100%*****] 1 of 1 completed					
	Open	High	Low	Close	Adj Close \
Date					
2009-01-02	902.989990	934.729980	899.349976	931.799988	931.799988
2009-01-05	929.169983	936.630005	919.530029	927.450012	927.450012
2009-01-06	931.169983	943.849976	927.280029	934.700012	934.700012
2009-01-07	927.450012	927.450012	902.369995	906.650024	906.650024
2009-01-08	905.729980	910.000000	896.809998	909.729980	909.729980
...	...	...	...	...	...
2019-12-23	3226.050049	3227.780029	3222.300049	3224.010010	3224.010010
2019-12-24	3225.449951	3226.429932	3220.510010	3223.379883	3223.379883
2019-12-26	3227.199951	3240.080078	3227.199951	3239.909912	3239.909912
2019-12-27	3247.229980	3247.929932	3234.370117	3240.020020	3240.020020
2019-12-30	3240.090088	3240.919922	3216.570068	3221.290039	3221.290039
Volume					
Date					
2009-01-02	4048270000				
2009-01-05	5413910000				
2009-01-06	5392620000				
2009-01-07	4704940000				
2009-01-08	4991550000				
...	...				
2019-12-23	3064530000				
2019-12-24	1296530000				
2019-12-26	2164540000				
2019-12-27	2429150000				
2019-12-30	3021720000				
[2767 rows x 6 columns]					

- Action(a): the desired allocating weights, $a_{t-1} = (a_{0,t-1}, a_{1,t-1}, \dots, a_{m,t-1})^T$ is the allocating vector at period $t-1$, subject to the constraint $\sum_{i=0}^n a_{i,t-1} = 1$. Due to the price movement in a day, the weights vector a_{t-1} at the beginning of the day would evolve into w_{t-1} at the end of the day:

$$w_{t-1} = \frac{y_{t-1} \odot a_{t-1}}{y_{t-1} \cdot a_{t-1}}$$



Ornstein-Uhlenbeck Process

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for episode = 1, M **do**

 Initialize a random process $\mathcal{N}$ for action exploration

 Receive initial observation state s_1

for t = 1, T **do**

 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

 Execute action a_t and observe reward r_t and observe new state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in R

 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

 Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

end for

end for

Ornstein-Uhlenbeck Process

The Ornstein-Uhlenbeck process x_t is defined by the following stochastic differential equation:

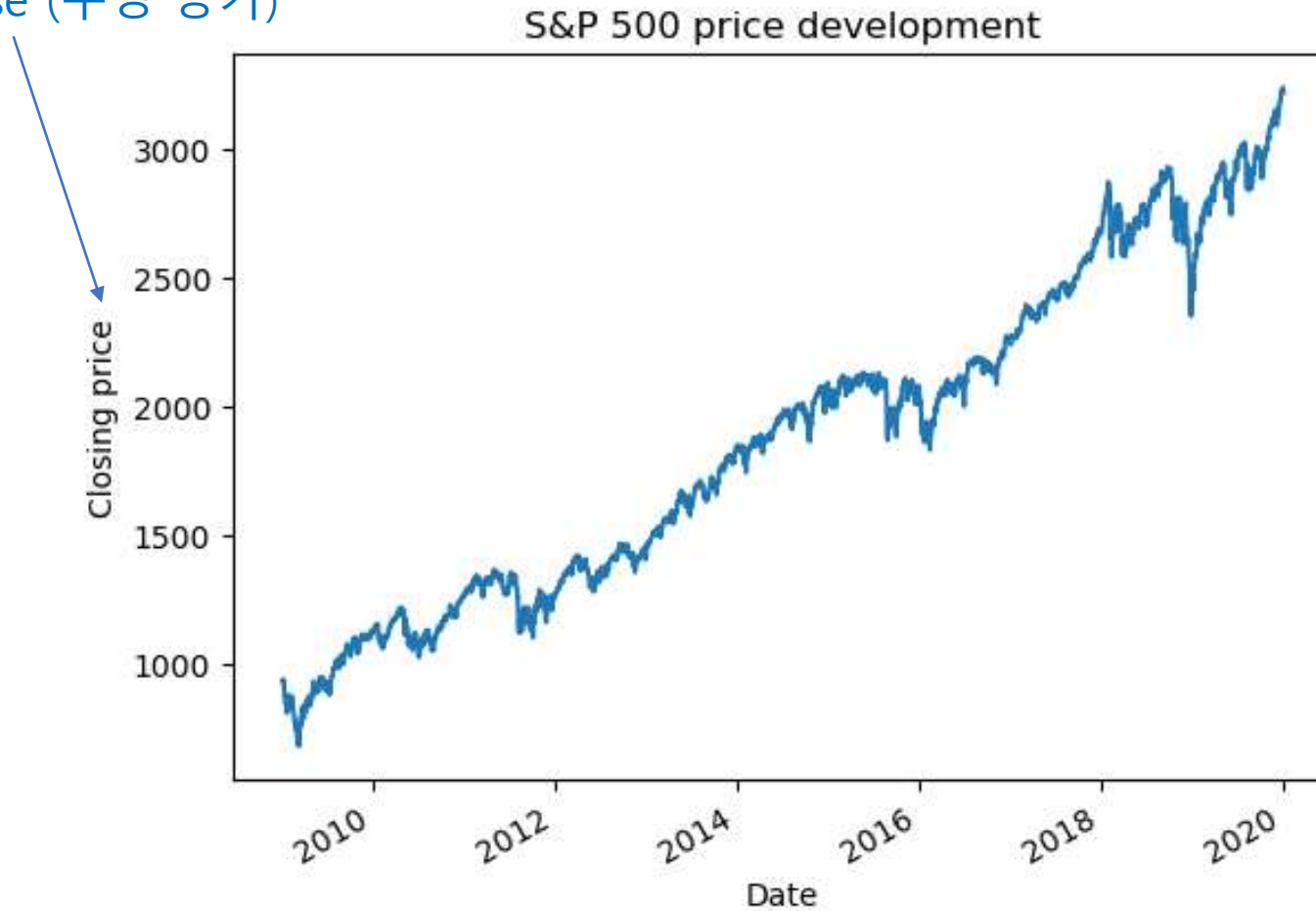
$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

where $\theta > 0$ and $\sigma > 0$ are parameters, μ is drift term and W_t denotes the Brownian motion.

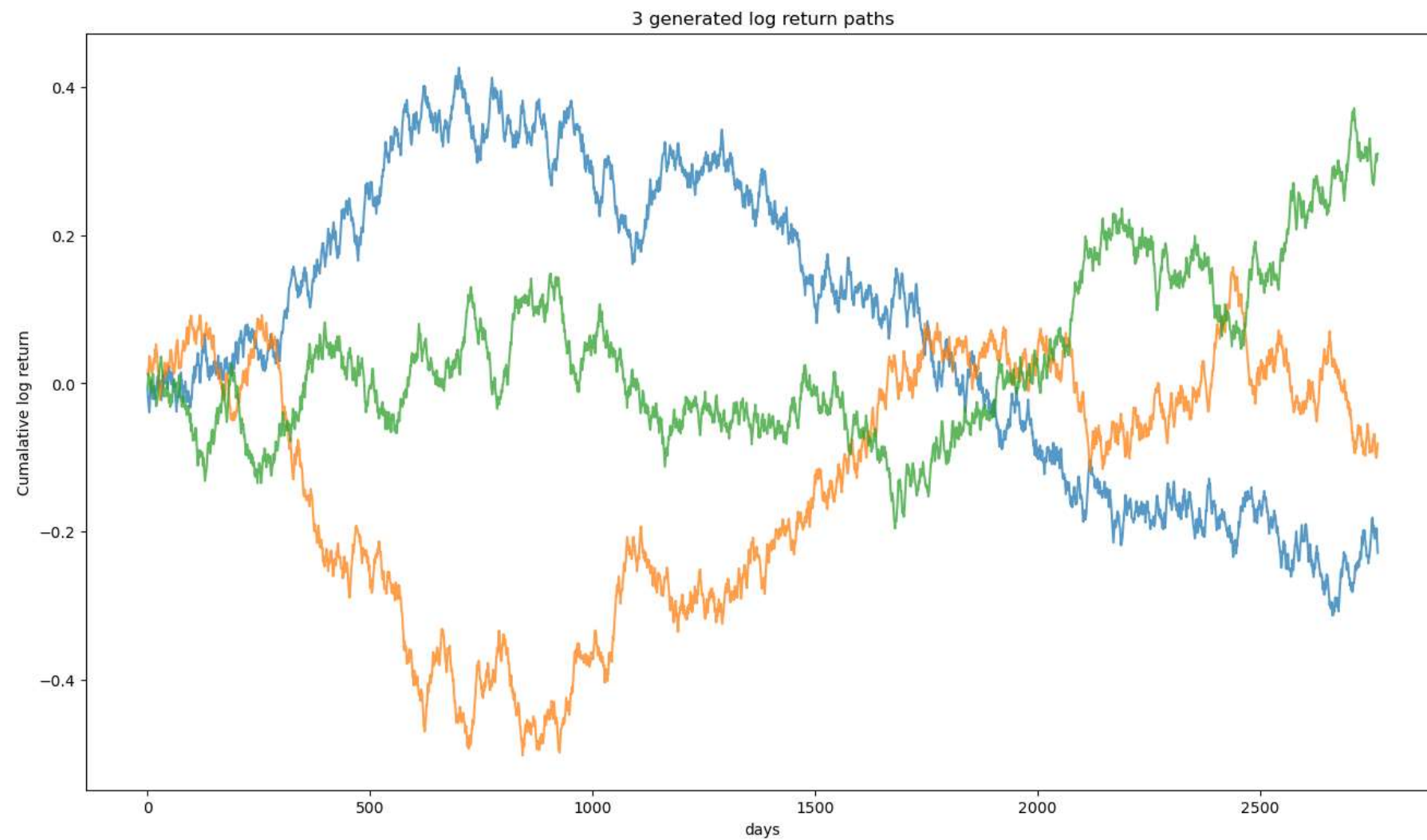
$$\mu=0.0, \theta=0.15, \sigma=0.2, dt=1e-2$$

N: number of features (**N=1**)

Adjusted close (수정 증가)



M, L



Actor

```
class DDPGActor(nn.Module):
    def __init__(self, M, L, N): # pass number of assets (M) for defining the network
        super(DDPGActor, self).__init__()
        self.M = M
        self.L = L
        self.N = N
        self.basic = cnnNetwork(self.M, self.L, self.N)
        self.fcl = nn.Linear(self.M, self.M)
        self.softmax = nn.Softmax()

    def forward(self, input):
        x, w = input
        # x = x.permute(0, 3, 1, 2)
        y = self.basic(x)
        y = self.fcl(torch.add(y, w))
        y = self.softmax(y)
        return y
```

x: state, w: action

$g(f(\text{state}) + \text{action})$

cnnNetwork

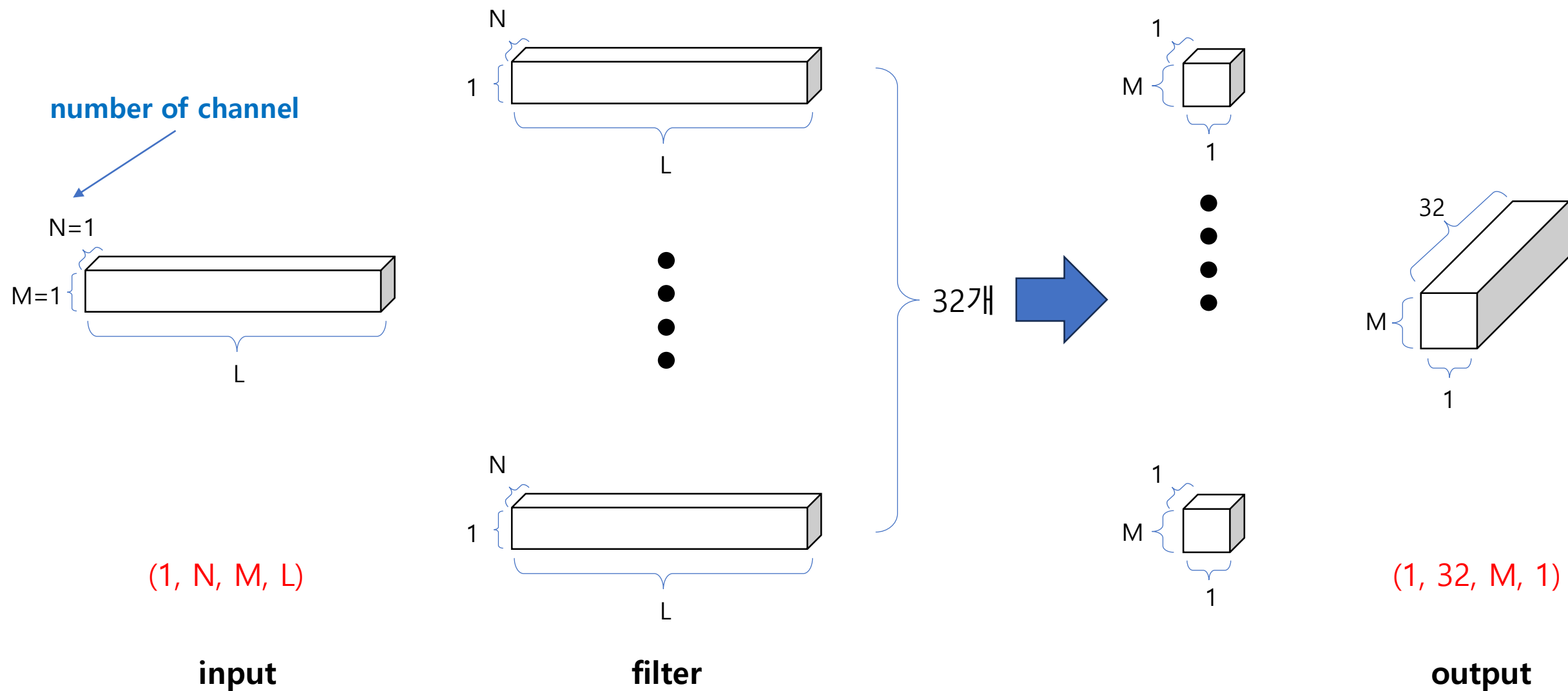
```
class cnnNetwork(nn.Module):
    def __init__(self, M, L, N):
        super(cnnNetwork, self).__init__()
        self.M = M
        self.L = L
        self.N = N

        self.conv2d1 = nn.Conv2d(in_channels=self.N, out_channels=32,
                                   kernel_size = (1,self.L), stride = 1, padding = "valid", bias=False)
        self.batchnorm1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU()

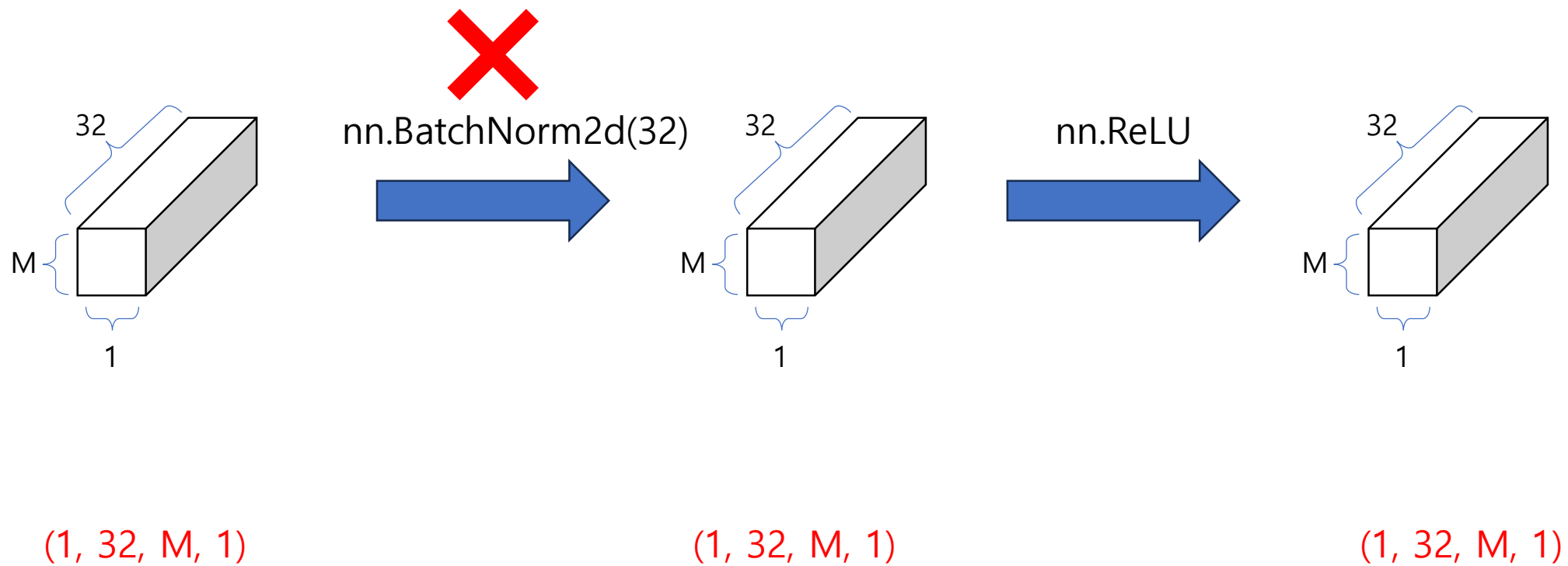
        self.conv2d2 = nn.Conv2d(in_channels=32, out_channels=1,
                                   kernel_size = (1,1), stride = 1, padding = "valid", bias=False)
        self.batchnorm2 = nn.BatchNorm2d(1)
        self.flatten = nn.Flatten()

    def forward(self, x):
        y = self.conv2d1(x)
        y = self.relu(self.batchnorm1(y))
        y = self.conv2d2(y)
        y = self.relu(self.batchnorm2(y))
        y = self.flatten(y)
        return y
```

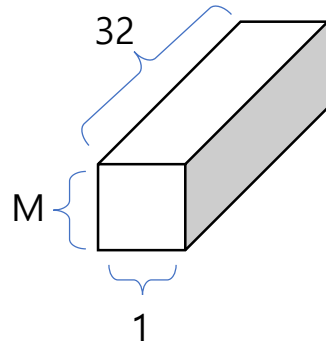
```
self.conv2d1 = nn.Conv2d(in_channels=self.N, out_channels=32, kernel_size = (1,self.L), stride = 1, padding = "valid", bias=False)
```



```
self.batchnorm1 = nn.BatchNorm2d(32)  
self.relu = nn.ReLU()
```

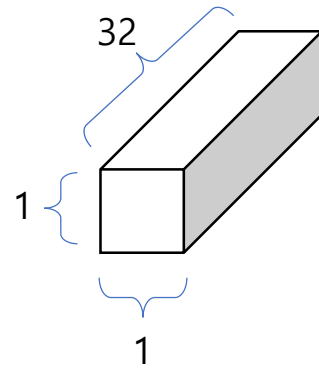


```
self.conv2d2 = nn.Conv2d(in_channels=32, out_channels=1, kernel_size = (1,1), stride = 1, padding = "valid", bias=False)
```

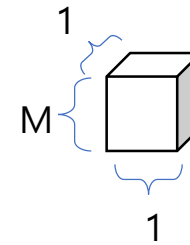


(1, 32, M, 1)

input



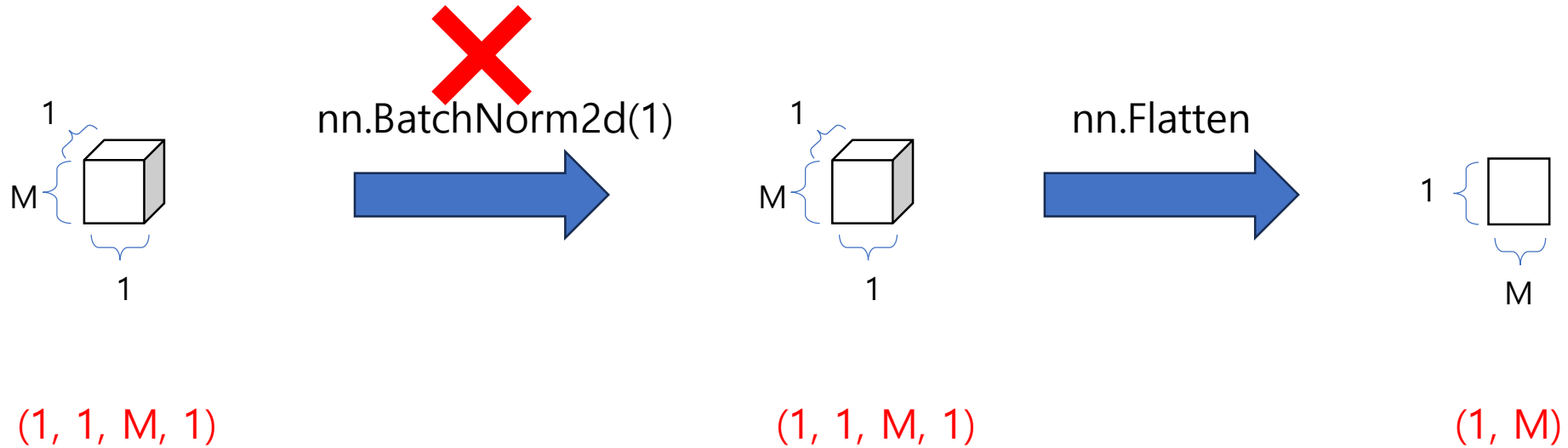
filter



(1, 1, M, 1)

output

```
self.batchnorm2 = nn.BatchNorm2d(1)
self.flatten = nn.Flatten()
```

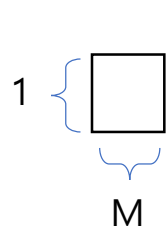


Actor

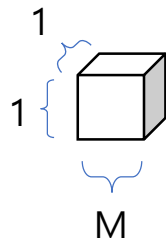
```
class DDPGActor(nn.Module):
    def __init__(self, M, L, N): # pass number of assets (M) for defining the network
        super(DDPGActor, self).__init__()
        self.M = M
        self.L = L
        self.N = N
        self.basic = cnnNetwork(self.M, self.L, self.N)
        self.fcl = nn.Linear(self.M, self.M)
        self.softmax = nn.Softmax()

    def forward(self, input):
        x, w = input
        # x = x.permute(0, 3, 1, 2)
        y = self.basic(x)
        y = self.fcl(torch.add(y, w))
        y = self.softmax(y)
        return y
```

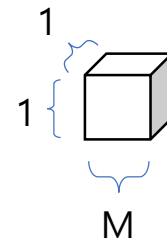
`torch.add(y, w)`



$(1, M)$



$(1, 1, 1, M)$

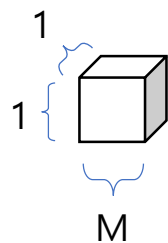


$(1, 1, 1, M)$

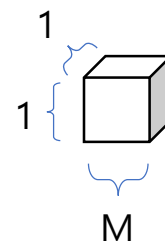
action (w)

```
self.fc1 = nn.Linear(self.M, self.M)
```

```
y = self.fc1(torch.add(y,w))
```

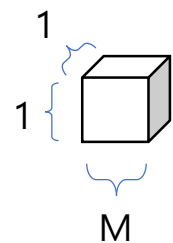


(1, 1, 1 ,M)



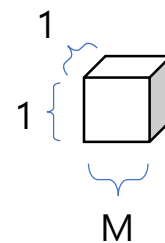
(1, 1, 1 ,M)

`y = self.softmax(y)`



(1, 1, 1 ,M)

softmax
→



(1, 1, 1 ,M)

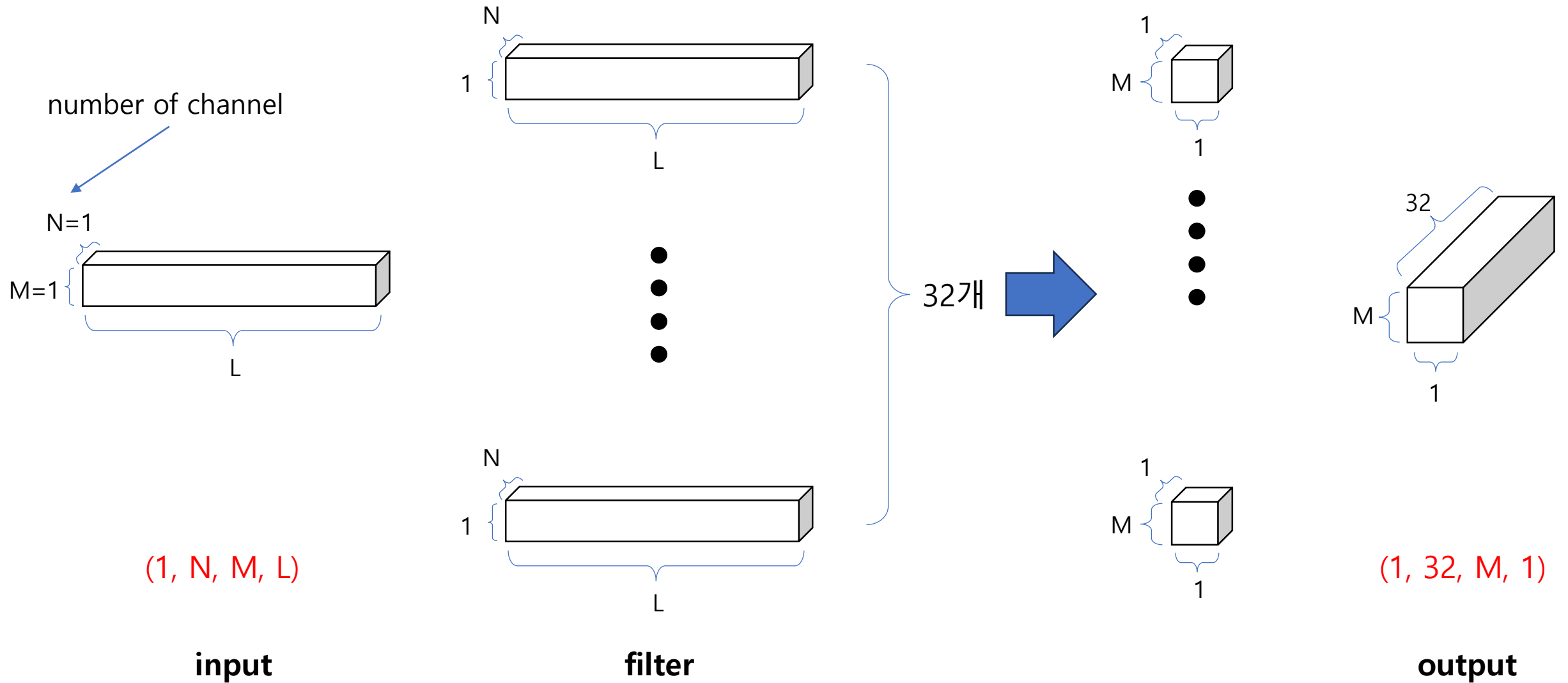
Critic

```
class DDPGCritic(nn.Module):
    def __init__(self, M, L, N):
        super(DDPGCritic, self).__init__()
        self.M = M
        self.L = L
        self.N = N
        self.basic = cnnNetwork(self.M, self.L, self.N)
        self.fc1 = nn.Linear(self.M, 1)

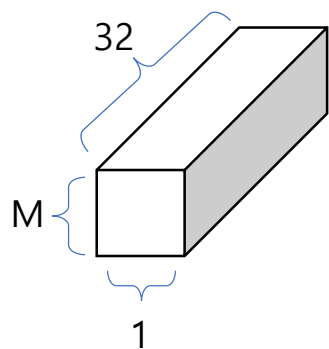
    def forward(self, input):
        x, w, action = input      x: state, w: previous_action, action: action
        # x = x.permute(0, 3, 1, 2)
        y = self.basic(x)
        y = torch.add(y, action)
        y = torch.add(y, w)
        y = self.fc1(y)
        return y
```

$g(f(\text{state}) + \text{action} + \text{previous_action})$

```
self.conv2d1 = nn.Conv2d(in_channels=self.N, out_channels=32, kernel_size = (1,self.L), stride = 1, padding = "valid", bias=False)
```

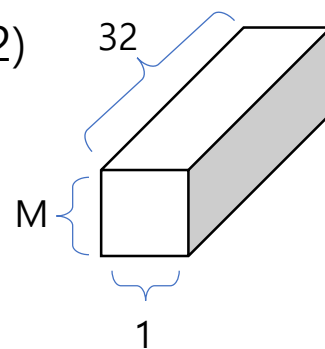


```
self.batchnorm1 = nn.BatchNorm2d(32)  
self.relu = nn.ReLU()
```



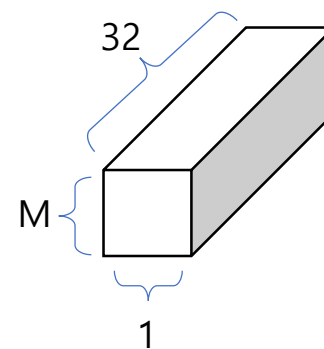
(1, 32, M, 1)

nn.BatchNorm2d(32)



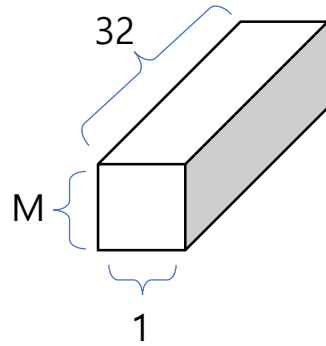
(1, 32, M, 1)

nn.ReLU



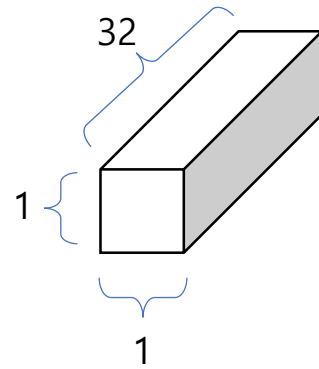
(1, 32, M, 1)

```
self.conv2d2 = nn.Conv2d(in_channels=32, out_channels=1, kernel_size = (1,1), stride = 1, padding = "valid", bias=False)
```

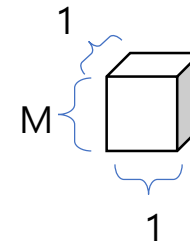


(1, 32, M, 1)

input



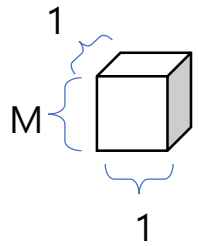
filter



(1, 1, M, 1)

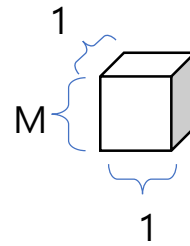
output


```
self.batchnorm2 = nn.BatchNorm2d(1)
self.flatten = nn.Flatten()
```



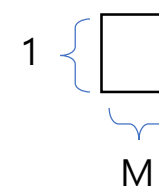
(1, 1, M, 1)

nn.BatchNorm2d(1)



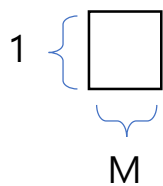
(1, 1, M, 1)

nn.Flatten

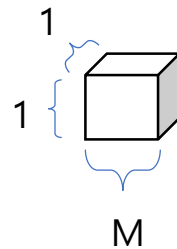


(1, M)

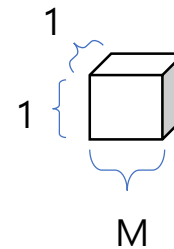
`torch.add(y, action)`



$(1, M)$



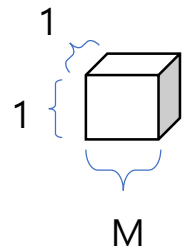
$(1, 1, 1, M)$



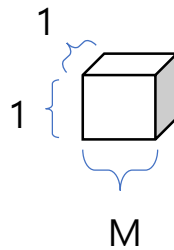
$(1, 1, 1, M)$

action

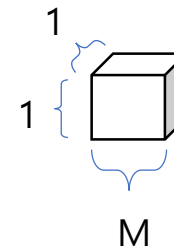
`torch.add(y, w)`



$(1, 1, 1, M)$



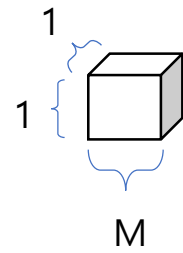
$(1, 1, 1, M)$



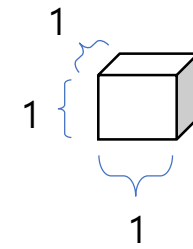
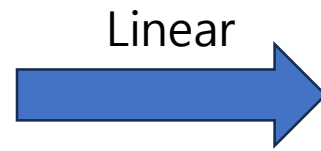
$(1, 1, 1, M)$

previous_action

```
self.fc1 = nn.Linear(self.M, 1)
```



(1, 1, 1 ,M)



(1, 1, 1 ,1)