ECE 1508: Reinforcement Learning

Chapter 6: Actor Critic Methods

Ali Bereyhi

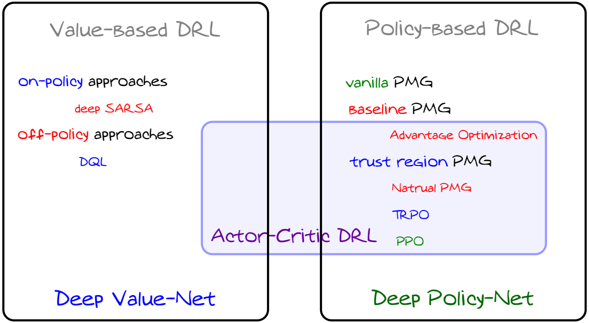
ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Summer 2024

Deep RL: *Sort of Division*



Deep RL: *Sort of Division*

In actor-critic approaches we have both networks

- *an actor has a policy network*
  ↳ *This network enables it to act at each particular state*
- *a critic has a value network*
  ↳ *This network enables it to evaluate its policy*
  ↳ *The evaluation will help improving the policy policy*

Deep RL: *Sort of Division*

Attention

For many people actor-critic $\equiv$ PGM: they usually argue that

- *to implement a PGM we need to estimate values*
- *we should do it by a value network*

So, any PGM is at the end actor-critic

That's practically true; however, in principle, we can

implement PGMs via basic Monte Carlo

So, we could also have a pure PGM, e.g., REINFORCE!

Implementing PGMs

Let's get back to PGMs: say we want to implement a PGM

- *We usually use sample advantages, i.e.,*

$$U_t = R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}} \left(S_{t+1} \right) - v_{\pi_{\boldsymbol{\theta}}} \left(S_t \right)$$

So, we need to know the value function $v_{\pi_{\boldsymbol{\theta}}} \left(\cdot \right)$ of our policy $\pi_{\boldsymbol{\theta}}$

\+ *Well, why don't we evaluate it once and use it forever?*

– *Attention! We need this evaluation each time we update policy $\pi_{\boldsymbol{\theta}}$!*

\+ *How exactly we do it then? You promised to tell us!*

– *Sure! Let's use what we have learned up to now*

Advantage PGM: *Implementing*

Let's look at the classic advantage optimization PGM

`AdvantagePGM():`

Initiate with $\boldsymbol{\theta}$ and learning rate α
while interacting **do**
    Set $\hat{\nabla} = \mathbf{0}$
    for mini-batch $b = 1 : B$ **do**
        Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ with policy $\pi_{\boldsymbol{\theta}}$
        for $t = 0 : T-1$ **do**
            Compute sample advantage $U_t = R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) - v_{\pi_{\boldsymbol{\theta}}}(S_t)$
            Compute sample gradient $\hat{\nabla} \leftarrow \hat{\nabla} + U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t | S_t) / B$
        end for
    end for
    Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \hat{\nabla}$
end while

To implement, we need to estimate $v_{\pi_{\boldsymbol{\theta}}}(S_t)$ for all trajectories in mini-batch

Estimating Values: *Monte-Carlo*

Say we are looking into one trajectory τ

$$\tau : S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We know how to use this trajectory to compute value estimates: for each t

$$\hat{V}_t = \text{estimate of value for } S_t = G_t = \sum_{i=t}^{T} \gamma^i R_{i+1}$$

If we the same state happens multiple times in the trajectory: we count the number of times $S_t = S$ *appears in the trajectory and average estimates, i.e.,*

$$\hat{v}_{\pi_{\boldsymbol{\theta}}}(S) = \frac{1}{\mathcal{N}(S \in \tau)} \sum_{t=0}^{T-1} \mathbf{1}\{S_t = S\} \hat{V}_t$$

where $\mathcal{N}(S \in \tau)$ *is the number of times* S *has appeared in* τ

Estimating Values: *Monte-Carlo*

If we have a mini-batch $\mathbb{B}$ of trajectories

$$\tau : S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We use the same approach

$$\hat{V}_t [\tau] = G_t [\tau] = \sum_{i=t}^{T} \gamma^i R_{i+1} [\tau]$$

We count the number of times $S_t = S$ appears in all trajectories and average the sample estimates, i.e.,

$$\hat{v}_{\pi_{\boldsymbol{\theta}}} (S) = \frac{1}{\mathcal{N} (S \in \mathbb{B})} \sum_{\tau \in \mathbb{B}} \sum_{t=0}^{T-1} \mathbf{1} \{S_t [\tau] = S\} \hat{V}_t [\tau]$$

where $\mathcal{N} (S \in \mathbb{B})$ the number of times S has appeared in $\mathbb{B}$

Advantage PGM: *With Value Estimates*

`EstAdvantagePGM():`

Initiate with $\boldsymbol{\theta}$ and learning rate α
while interacting ***do***
  for mini-batch $b = 1 : B$ ***do***
    Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ *with policy* $\pi_{\boldsymbol{\theta}}$
  end for
  Estimate value of all observed states in the mini-batch as $\hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t)$
  Set $\hat{\nabla} = \mathbf{0}$
  for $b = 1 : B$ ***do***
    for $t = 0 : T-1$ ***do***
      Compute sample advantages $U_t = R_{t+1} + \gamma \hat{v}_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t)$
      Update sample gradient $\hat{\nabla} \leftarrow \hat{\nabla} + U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t|S_t)/B$
    end for
  end for
  Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \hat{\nabla}$
end while

Advantage PGM: *With Value Estimates*

We could guess that this algorithm is not going to perform very impressive!

+ *And why is that?!*
– *For the exact same reasons we said at the beginning of Chapter 4*
    ↳ *We have lots of states*
    ↳ *Many of them are rarely observed in a small mini-batch*
    ↳ *The estimates can hence be very high variance*
    ↳ *Also we need to wait for the whole mini-batch to be ready*
    ↳ . . .
+ *So, what is the solution?*
– *You tell me!*
+ *We go for function approximation via value networks!*
– *You got it right!*

Recall: *Value Network*

Let's keep our trajectories here

$$\tau : S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

What we need is a simple v-network, as we only need the state values

$$\mathbf{x}(s) \longrightarrow \boxed{v_{\mathbf{w}}(\cdot)} \longrightarrow \hat{v}_\pi(s)$$

In Chapter 4, we saw that we could train it via sample returns, i.e.,

$$\textit{Dataset} = \left\{ \left(S_t[\tau], \hat{V}_t[\tau] \right) : \forall t \text{ and } \tau \right\}$$

and we train the network by minimizing the least-square loss

Value Network: *Training*

Let's keep our trajectories here

$$\tau : S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

This means that we compute the loss function

$$\mathcal{L}^v \left(\mathbf{w} \right) = \sum_{\tau} \sum_{t} \left(v_{\mathbf{w}} \left(S_t \left[\tau \right] \right) - \hat{V}_t \left[\tau \right] \right)^2$$

and update the wights of the v-network as

$$\mathbf{w} \leftarrow \operatorname*{argmin}_{\mathbf{w}} \mathcal{L}^v \left(\mathbf{w} \right)$$

which we approximately solve using gradient descent

Basic Actor-Critic

This is going to end us with a basic actor-critic algorithm:

`AC_v1():`

Initiate with $\boldsymbol{\theta}$ *and* $\mathbf{w}$*, as well as a learning rate* α
while *interacting* **do**
    for *mini-batch* $b = 1 : B$ **do**
        Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ *with policy* $\pi_{\boldsymbol{\theta}}$
        for $t = 0 : T-1$ **do**
            Compute value estimate $\hat{V}_t$
            Compute sample advantages $U_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_{t+1})$
            Update sample gradient $\hat{\nabla} \leftarrow \hat{\nabla} + U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t | S_t) / B$
        end for
    end for
    Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \hat{\nabla}$
    Update $\mathbf{w}$ *by SGD using value estimates* $\hat{V}_t$
end while

Training Value Network: *TD Estimates*

$$\tau : S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

But now that we have a value network, we could also use TD: at step t, we set

$$\hat{V}_t = \text{estimate of value for } S_t = R_{t+1} + \gamma v_{\mathbf{w}} \left(S_{t+1} \right)$$

- *We can estimate the advantage using the current value network*

$$U_t = R_{t+1} + \gamma v_{\mathbf{w}} \left(S_{t+1} \right) - v_{\mathbf{w}} \left(S_{t+1} \right)$$

- *We then use least-squares update value network by TD sample estimates*

$$\mathcal{L}^v \left(\mathbf{w} \right) = \sum_{\tau} \sum_{t=0}^{T-1} \left(v_{\mathbf{w}} \left(S_t \left[\tau \right] \right) - \hat{V}_t \left[\tau \right] \right)^2$$

Training Value Network: *TD Estimates*

Let's write the update rule of the value network: we compute the gradient of loss and move in that direction

$$\nabla \mathcal{L}^{v}\left(\mathbf{w}\right) = 2 \sum_{t=0}^{T-1} \underbrace{\left(v_{\mathbf{w}}\left(S_t\right) - \hat{V}_t\right)}_{-\Delta_t} \nabla v_{\mathbf{w}}\left(S_t\right)$$

We used to call Δ_t the TD error, and set the learning rate to some $\beta/2$ to get

$$\mathbf{w} \leftarrow \mathbf{w} + \beta \sum_{t=0}^{T-1} \Delta_t \nabla v_{\mathbf{w}}\left(S_t\right)$$

Training Value Network: *TD Estimates*

Let's look at TD error: recall that our labels, i.e., sample estimates of values, are

$$\hat{V}_t = R_{t+1} + \gamma v_{\mathbf{w}} \left(S_{t+1} \right)$$

So the TD error is given by

$$\begin{aligned}\Delta_t &= \hat{V}_t - v_{\mathbf{w}} \left(S_t \right) \\ &= R_{t+1} + \gamma v_{\mathbf{w}} \left(S_{t+1} \right) - v_{\mathbf{w}} \left(S_t \right) \\ &= U_t\end{aligned}$$

This leads us to what we observed in Chapter 5

Recall: Advantage vs TD Error

Advantage is an estimator of TD error

A2C: *Basic Version*

A2C():

Initiate with $\boldsymbol{\theta}$ *and* $\mathbf{w}$, *as well as learning rates* α *and* β

while *interacting* **do**

  Start with zero gradients $\hat{\nabla}_{\mathbf{w}} = \hat{\nabla}_{\boldsymbol{\theta}} = \mathbf{0}$

  Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ *with policy* $\pi_{\boldsymbol{\theta}}$

  Compute sample advantages $U_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$

  for $t = 0 : T-1$ **do**

    Compute sample policy gradient $\hat{\nabla}_{\boldsymbol{\theta}} \leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t | S_t)$

    Compute sample value gradient $\hat{\nabla}_{\mathbf{w}} \leftarrow \hat{\nabla}_{\mathbf{w}} + U_t \nabla v_{\mathbf{w}}(S_t)$

  end for

  Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \hat{\nabla}_{\boldsymbol{\theta}}$

  Update value network $\mathbf{w} \leftarrow \mathbf{w} + \beta \hat{\nabla}_{\mathbf{w}}$

end while

This is the single-trajectory form of

Advantage Actor Critic $\equiv$ *A2C*

A2C: *Online Version*

Since we use TD, we can also update online, i.e., in each time step

`OnlineA2C():`

Initiate with $\boldsymbol{\theta}$ and $\mathbf{w}$, a random state S_0, $t = 0$ and learning rates α and β
while** interacting **do
  Sample A_t from $\pi_{\boldsymbol{\theta}}(\cdot | S_t)$
  Sample single step $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$ from environment
  Compute sample advantage $U_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$
  Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t | S_t)$
  Update value network $\mathbf{w} \leftarrow \mathbf{w} + \beta U_t \nabla v_{\mathbf{w}}(S_t)$
  ***if** S_{t+1} is terminal **then** draw a random S_{t+1}*
  Set $t \leftarrow t + 1$
end while

But, that would be too noisy and hence quite unstable

A2C: *Mini-Batch Version*

We can further extend to mini-batch learning

`miniBatchA2C():`

Initiate with $\boldsymbol{\theta}$ *and* $\mathbf{w}$, *as well as learning rates* α *and* β

while *interacting* **do**

  Start with zero gradients $\hat{\nabla}_{\mathbf{w}} = \hat{\nabla}_{\boldsymbol{\theta}} = \mathbf{0}$

  for *mini-batch* $b = 1 : B$ **do**

    Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ *with policy* $\pi_{\boldsymbol{\theta}}$

    Compute sample advantages $U_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$

    for $t = 0 : T - 1$ **do**

      Compute sample policy gradient $\hat{\nabla}_{\boldsymbol{\theta}} \leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + U_t \nabla \log \pi_{\boldsymbol{\theta}}(A_t | S_t)$

      Compute sample value gradient $\hat{\nabla}_{\mathbf{w}} \leftarrow \hat{\nabla}_{\mathbf{w}} + U_t \nabla v_{\mathbf{w}}(S_t)$

    end for

  end for

  Update policy network $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \hat{\nabla}_{\boldsymbol{\theta}}$

  Update value network $\mathbf{w} \leftarrow \mathbf{w} + \beta \hat{\nabla}_{\mathbf{w}}$

end while

Actor-Critic via Shared-Network

There is one extra obvious fact: *the policy and values that we learn are very much mutually related!*

+ *So, why don't we learn them together?!*
- *Actually we can!*

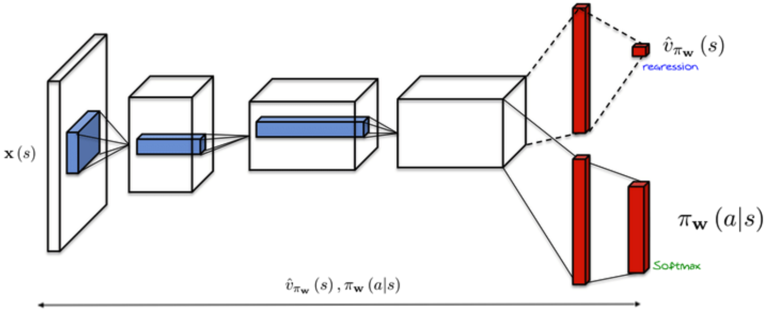
We can consider an actor-critic model, i.e.,

$$\mathbf{x}(s,a) \longrightarrow \boxed{(\pi, v)_{\mathbf{w}}(\cdot)} \longrightarrow \hat{v}_{\pi_{\mathbf{w}}}(s), \pi_{\mathbf{w}}(a|s)$$

and train it all together!

- *This model can be simply a DNN*
- *The DNN's output contains both policy and value*

Actor-Critic via Shared-Network: *Visualization*



Here, value and policy share same layers except the few last layers

Actor-Critic via Shared-Network: *Loss*

+ *But how can we train the loss in this network?*
- *We could let it to be proportional to sum of our both objectives*

We could define a new loss as

$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= -\mathcal{J}(\pi_{\mathbf{w}}) + \xi \mathcal{L}^{v}(\mathbf{w}) \\ &= \sum_{\tau} \sum_{t=0}^{T-1} -U_t[\tau] \log \pi_{\mathbf{w}}(A_t[\tau] | S_t[\tau]) + \xi \left(v_{\mathbf{w}}(S_t[\tau]) - \hat{V}_t[\tau] \right)^2\end{aligned}$$

for some hyperparameter ξ: it's easy to see that in this case

$$\nabla \mathcal{L}(\mathbf{w}) = -\sum_{\tau} \sum_{t=0}^{T-1} U_t[\tau] \left[\nabla \log \pi_{\mathbf{w}}(A_t[\tau] | S_t[\tau]) + \xi \nabla v_{\mathbf{w}}(S_t[\tau]) \right]$$

A2C: *Shared-Network Version*

sharedNetA2C():

Initiate shared network $(\pi_{\mathbf{w}}, v_{\mathbf{w}})$ *with* $\mathbf{w}$

Choose potentially scheduled value-weight ξ *and learning rate* α

while *interacting* ***do***

    Start with zero gradients $\hat{\nabla}_{\mathbf{w}} = \mathbf{0}$

    for *mini-batch* $b = 1 : B$ ***do***

        Sample $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ *with policy* $\pi_{\mathbf{w}}$

        Compute sample advantages $U_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$

        for $t = 0 : T-1$ ***do***

            Compute sample gradient $\hat{\nabla} \leftarrow \hat{\nabla} + U_t \left[\nabla \log \pi_{\mathbf{w}}(A_t | S_t) + \xi \nabla v_{\mathbf{w}}(S_t)\right]$

        end for

    end for

    Update shared network $\mathbf{w} \leftarrow \mathbf{w} + \alpha \hat{\nabla}$

end while

Extension to Other PGMs

In practice, all PGMs we studies in Chapter 5 can be implemented via the actor-critic idea through the following general framework

Loop over the following three steps

1. *Specify policy and value networks*
   - ↳ *They could be either separate DNNs or DNNs with shared layers*
2. *Set the policy and sample a batch of trajectories*
3. *Go over the batch for multiple epochs*
   - ↳ *After each mini-batch estimate the policy and value gradient*
   - ↳ *Update both policy and value networks after each mini-batch*

Let's now look at the TRPO and PPO algorithms in actor-critic framework

TRPO: *Actor-Critic*

```
TRPO():
```

Initiate with $\boldsymbol{\theta}$ and $\mathbf{w}$, as well as factor $\alpha < 1$ and learning rate β

while** interacting **do

    Sample a batch of trajectories $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_T} S_T$ by policy $\pi_{\boldsymbol{\theta}}$

    for** multiple epochs **do

        for** samples in each mini-batch **do

            Compute sample advantage using $v_{\mathbf{w}}(\cdot)$

            Update policy gradient $\hat{\nabla}_{\boldsymbol{\theta}}$ and value gradient $\hat{\nabla}_{\mathbf{w}}$

        end for

        Compute a Hessian estimator $\hat{\mathbf{H}}$ and solve $\hat{\mathbf{H}}\mathbf{y} = \hat{\nabla}$ for $\mathbf{y}$

        Backtrack on a line to find minimum i satisfying $\bar{D}_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}'} \| \pi_{\boldsymbol{\theta}}) \leqslant d_{\max}$

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} + \alpha^i \sqrt{\frac{2 d_{\max}}{\mathbf{y}^\mathsf{T} \hat{\mathbf{H}} \mathbf{y}}} \mathbf{y}$$

        Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}'$ and $\mathbf{w} \leftarrow \mathbf{w} + \beta \hat{\nabla}_{\mathbf{w}}$

    end for

end while

Recall: *Use Importance Sampling*

Attention: *Importance Sampling*

It is important to remember that in each iteration of PGM

we estimate the policy gradient via importance sampling

Let's denote the policy of current mini-batch with $\pi_{\boldsymbol{\theta}}$: in next mini-batch we compute the policy gradient as

$$\hat{\nabla}_{\boldsymbol{\theta}} \leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + \sum_{t=0}^{T} U_t \frac{\nabla \pi_{\mathbf{x}} \left(A_t | S_t \right) |_{\mathbf{x}=\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}} \left(A_t | S_t \right)}$$

with U_t being the sample advantage of policy $\pi_{\boldsymbol{\theta}}$

Recall: *Use Importance Sampling*

+ *You mentioned this before! What is new about this?!*
– *Well! We should also consider it in our value estimation!*

Denote the policy that we sampled with in line 3 with $\pi_{\boldsymbol{\theta}_{\text{old}}}$: after multiple mini-batches we have an updated policy gradient $\pi_{\boldsymbol{\theta}}$

⚠ *To update the value network in this mini-batch, we consider the Bellman equation which says*

$$v_{\pi_{\boldsymbol{\theta}}}(S_t) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1})\right\}$$

and set the labels for value network training as

$$\hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t) = R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1})$$

But this is only valid if we had sampled the trajectory by $\pi_{\boldsymbol{\theta}}$!

Recall: *Use Importance Sampling*

+ *Shall we use importance sampling here as well?!*
- *Sure!*

By importance sampling we could say

$$
\begin{aligned}
v_{\pi_{\boldsymbol{\theta}}}(S_t) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1})\right\} \\
&= \mathbb{E}_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}}\left\{\left(R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1})\right)\frac{\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(A_t|S_t)}\right\}
\end{aligned}
$$

and now we can compute value estimators from our sample trajectories as

$$
\hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t) = \left(R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}}(S_{t+1})\right)\frac{\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(A_t|S_t)}
$$

Recall: *Use Importance Sampling*

This means that the advantage should be computed as

$$
\begin{aligned}
U_t &= \underbrace{\left(R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}_{\text{old}}}}\left(S_{t+1}\right)\right)}_{\text{samples in batch}} \underbrace{\frac{\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}_{\text{old}}}\left(A_t|S_t\right)}}_{\text{importance sampling}} - v_{\mathbf{w}}\left(S_t\right) \\
&\approx \left(R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}_{\text{old}}}}\left(S_{t+1}\right) - v_{\pi_{\boldsymbol{\theta}_{\text{old}}}}\left(S_t\right)\right) \frac{\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}_{\text{old}}}\left(A_t|S_t\right)} \\
&= U_t^{\text{old}} \frac{\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}_{\text{old}}}\left(A_t|S_t\right)}
\end{aligned}
$$

This is the correct way of estimating advantage: other implementations that ignore this will have bias especially with too much epochs

Recall: *Use Importance Sampling*

Consequently, the value gradient is updated as

$$\hat{\nabla}_{\mathbf{w}} \leftarrow \hat{\nabla}_{\mathbf{w}} + \frac{1}{T} \sum_{t=0}^{T-1} U_t \nabla v_{\mathbf{w}} \left(S_t \right)$$

$$\leftarrow \hat{\nabla}_{\mathbf{w}} + U_t^{\text{old}} \frac{\pi_{\boldsymbol{\theta}} \left(A_t | S_t \right)}{\pi_{\boldsymbol{\theta}_{\text{old}}} \left(A_t | S_t \right)} \nabla v_{\mathbf{w}} \left(S_t \right)$$

at the end of each trajectory

+ *Shall we also consider this when we update the policy?*
– *Of course!*

Recall: *Use Importance Sampling*

In a given mini-batch we update the policy gradient as

$$
\begin{aligned}
\hat{\nabla}_{\boldsymbol{\theta}} &\leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + \sum_{t=0}^{T} U_t \frac{\nabla \pi_{\mathbf{x}} \left(A_t | S_t \right) |_{\mathbf{x}=\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}} \left(A_t | S_t \right)} \\
&\leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + \sum_{t=0}^{T} U_t^{\mathrm{old}} \frac{\pi_{\boldsymbol{\theta}} \left(A_t | S_t \right)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}} \left(A_t | S_t \right)} \frac{\nabla \pi_{\mathbf{x}} \left(A_t | S_t \right) |_{\mathbf{x}=\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}} \left(A_t | S_t \right)} \\
&\leftarrow \hat{\nabla}_{\boldsymbol{\theta}} + \sum_{t=0}^{T} U_t^{\mathrm{old}} \frac{\nabla \pi_{\mathbf{x}} \left(A_t | S_t \right) |_{\mathbf{x}=\boldsymbol{\theta}}}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}} \left(A_t | S_t \right)}
\end{aligned}
$$

which is now consistent with importance sampling

TRPO: *Actor-Critic*

```
TRPO_AC():
```

1: *Initiate with* $\boldsymbol{\theta} = \boldsymbol{\theta}_{\text{old}}$ *and* $\mathbf{w}$, *as well as factor* $\alpha < 1$ *and learning rate* β

2: ***while*** *interacting* ***do***

3: *Sample a batch of trajectories* $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_T} S_T$ *by policy* $\pi_{\boldsymbol{\theta}_{\text{old}}}$

4: *Compute sample advantage* U_t^{old} *using* $v_{\mathbf{w}}(\cdot)$

5: ***for*** *multiple epochs in each mini-batch* ***do***

6: *Compute gradient estimators* $(\hat{\nabla}_{\boldsymbol{\theta}}, \hat{\nabla}_{\mathbf{w}})$ *from* U_t^{old} *via importance sampling*

7: *Compute a Hessian estimator* $\hat{\mathbf{H}}$ *and solve* $\hat{\mathbf{H}}\mathbf{y} = \hat{\nabla}_{\boldsymbol{\theta}}$ *for* $\mathbf{y}$

8: *Backtrack on a line to find minimum* i *satisfying* $\bar{D}_{\text{KL}}(\pi_{\boldsymbol{\theta}'} \| \pi_{\boldsymbol{\theta}}) \leqslant d_{\max}$

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} + \alpha^i \sqrt{\frac{2 d_{\max}}{\mathbf{y}^\mathsf{T} \hat{\mathbf{H}} \mathbf{y}}}\, \mathbf{y}$$

9: *Update* $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}'$ *and* $\mathbf{w} \leftarrow \mathbf{w} + \beta \hat{\nabla}_{\mathbf{w}}$

10: ***end for***

11: *Update* $\pi_{\boldsymbol{\theta}} \leftarrow \pi_{\boldsymbol{\theta}_{\text{old}}}$

12: ***end while***

PPO: *Actor-Critic*

In PPO, we maximize in each iteration the restricted surrogate

$$\tilde{\mathcal{L}}\left(\pi_{\mathbf{x}}\right) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left\{\min\left\{U_t \frac{\pi_{\mathbf{x}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right)}, \ell_{\varepsilon}\left(U_t\right)\right\}\right\}$$

Similar to TRPO, we can estimate restricted surrogate via importance sampling

$$\begin{aligned}\tilde{\mathcal{L}}\left(\pi_{\mathbf{x}}\right) &= \mathtt{mean}\left[\sum_t \min\left\{U_t \frac{\pi_{\mathbf{x}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}}\left(A_t|S_t\right)}, \ell_{\varepsilon}\left(U_t\right)\right\}\right] \\ &= \mathtt{mean}\left[\sum_t \min\left\{U_t^{\mathrm{old}} \frac{\pi_{\mathbf{x}}\left(A_t|S_t\right)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}\left(A_t|S_t\right)}, \ell_{\varepsilon}\left(U_t\right)\right\}\right]\end{aligned}$$

where the mean is computed over the sample trajectories of a mini-batch

PPO Algorithm: *Actor-Critic*

`PPO_AC():`

Initiate with $\boldsymbol{\theta} = \boldsymbol{\theta}_{\mathrm{old}}$ *and* $\mathbf{w}$, *as well as learning rates* α *and* β

while *interacting* ***do***

    Sample a batch of trajectories $S_0, A_0 \xrightarrow{R_1} \cdots \xrightarrow{R_T} S_T$ *by policy* $\pi_{\boldsymbol{\theta}_{\mathrm{old}}}$

    Compute sample advantage U_t^{old} *using* $v_{\mathbf{w}}(\cdot)$

    for *multiple epochs in each mini-batch* ***do***

        Compute value gradient estimator $\hat{\nabla}_{\mathbf{w}}$ *from* U_t^{old} *via importance sampling*

        Compute the restricted surrogate

$$\tilde{\mathcal{L}}(\pi_{\mathbf{x}}) = \texttt{mean}\left[\sum_t \min\left\{U_t^{\mathrm{old}} \frac{\pi_{\mathbf{x}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(A_t|S_t)}, \ell_\varepsilon(U_t)\right\}\right]$$

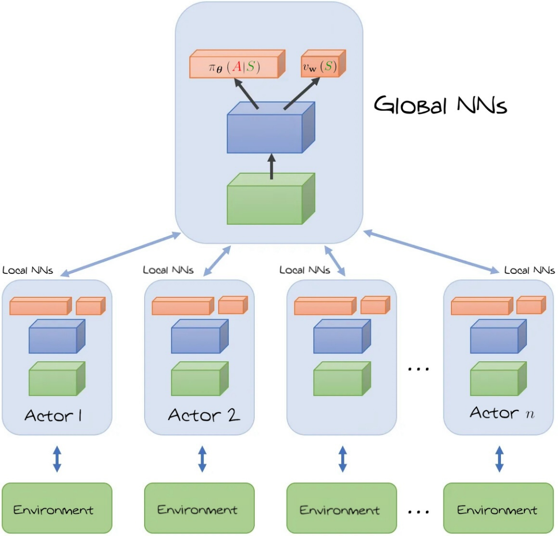
        Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla \tilde{\mathcal{L}}(\pi_{\mathbf{x}})\,|_{\mathbf{x}=\boldsymbol{\theta}}$ *and* $\mathbf{w} \leftarrow \mathbf{w} + \beta \hat{\nabla}_{\mathbf{w}}$

    end for

    Update $\pi_{\boldsymbol{\theta}} \leftarrow \pi_{\boldsymbol{\theta}_{\mathrm{old}}}$

end while

Distributed Actor-Critic



Distributed Setting: *Asynchronous vs Synchronous*

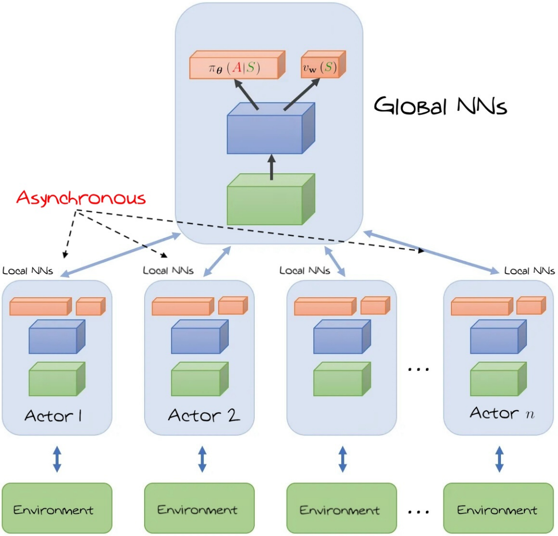
We can implement actor-critic approaches in a distributed fashion

- *Multiple actors few samples with local policy and value network*
  - ↳ *Each actor focuses on a specific part of environment*
- *They share their gradient estimators with the server*
- *Server treats the collected estimators as of a large mini-batch*
  - ↳ *It updates its networks on this large mini-batch*
- *All actors update their local networks every time server shares its networks*

We can implement this setting

- *Synchronous*
  - ↳ *This is basically the same as what we do in A2C, TRPO or PPO*
- *Asynchronous*
  - ↳ *Server does not wait for all actors to send their estimators*
  - ↳ *It uses what it has every couple of rounds and remaining in next rounds*

A3C: *Asynchronous A2C*



Some Final Remarks

It turns out that asynchronous update can negatively impact convergence

- *A3C is hence not really extended to other PGMs*
- *In practice we usually implement actor-critic approaches in synchronous distributed form*

\+ *Is that it? Are we free to go now?!*

– *Pretty much Yes! Just we may further take a look at deterministic policy gradient approaches as well*

\+ *Is it a new set of approaches?!*

– *No! It's a specific form of actor-critic methods that are better compatible with continuous actions*

Learning *Deterministic* Policy

From model-based RL we know that: *the optimal policy can be deterministic*

Why don't we train a policy network that learns a deterministic policy?

+ *You are contradicting yourself! You said that stochastic policy is a general case that includes deterministic policies as well! Now you want to get back to a deterministic policy?!*
– *Well! You're right! But there will be no harm in learning a deterministic policy! It might only be less effective!*
+ *Why we should do it then?*
– *It could give us some benefits, especially when we have continuous action-space*

Learning *Deterministic* Policy

With continuous action-space, policy is a density function

- *With discrete action-space, we can show policy by a finite vector*[1]

$$\mathbf{x}(s) \longrightarrow \boxed{\pi_{\boldsymbol{\theta}}(\cdot|s)} \longrightarrow \begin{bmatrix} \pi_{\boldsymbol{\theta}}(a^1|s) \\ \vdots \\ \pi_{\boldsymbol{\theta}}(a^M|s) \end{bmatrix}$$

- *Unlike discrete action-space, we cannot do this with continuous actions*
  - ↳ *We should learn a function from state feature, e.g.,*

$$\pi_{\boldsymbol{\theta}}(a|s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(a - \mathrm{DNN}(\mathbf{x}(s)|\boldsymbol{\theta}))}{2\sigma^2}\right\}$$

  - ↳ *We then sample from this learned density function, e.g.,*
    - ↳ *Draw a sample from Gaussian distribution with mean* $\mathrm{DNN}(\mathbf{x}(s)|\boldsymbol{\theta})$ *and variance* σ^2

[1] *We have seen this in Assignment 3*

Learning *Deterministic* Policy

There are several things that could go wrong

- *What if the generated sample is out of accepted range?*
  - ↳ *Our sample from Gaussian distribution is extremely large*
  - ↳ *In sensitive control settings, this could harm the system*
- *What if we only try a few samples?*
  - ↳ *We don't see the probabilities as opposed to continuous actions*
  - ↳ *We then cannot really reject too many samples*

With continuous actions, we usually prefer to learn a deterministic policy: its main feature is that it can be represented by a single action $a^\star$

$$\pi_{\boldsymbol{\theta}}\left(a|s\right) = \begin{cases} 1 & a = a^\star \\ 0 & a \neq a^\star \end{cases}$$

Deterministic Policy Network

Considering a deterministic policy, we only need to learn an estimate of optimal action in each state: we can revise our policy network into a deterministic policy network

$$\mathbf{x}\left(s\right) \longrightarrow \boxed{\mu_{\boldsymbol{\theta}}\left(\cdot\right)} \longrightarrow \hat{a}^\star\left(s\right)$$

Deterministic Policy Network

Deterministic policy network maps a state-features into a single action and can be realized by a DNN with input being the state feature representation and a single output

Deterministic Policy Network: *Training*

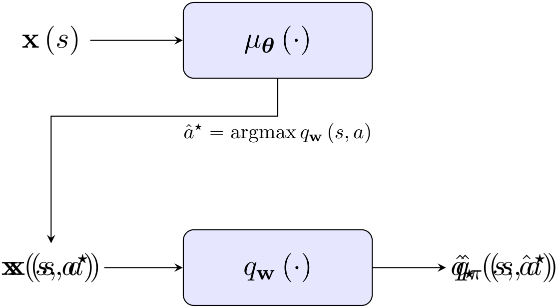
+ *How should we train these networks? Similar to other policy networks?*
- *We can look at them as a special form of policy networks and do the same thing, yes! However, it turns out that in this special case, there are better ways to do it! Especially as we always implement them actor-critic*
+ *So, we should start all over again?!*
- *Not really! We should basically use what we learned for DQL*

Deterministic Policy Network: *Training*

Recall the property of optimal policy: it gives maximum value and action-value

---

If we have a Q-network that estimates optimal action-value, we can say



The output satisfies

$$\hat{q}_{\star}(s, \hat{a}^{\star}) = \max_{a} \hat{q}_{\star}(s, a)$$

Deterministic Policy Network: *Training*

So, in actor-critic form with a Q-network, we could train the deterministic policy network as

$$\boldsymbol{\theta}^\star = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, Q_{\mathbf{w}}\left(s, \mu_{\boldsymbol{\theta}}\left(s\right)\right)$$

which we can solve using gradient ascent by updating as

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} Q_{\mathbf{w}}\left(s, \mu_{\boldsymbol{\theta}}\left(s\right)\right) \\ &\leftarrow \boldsymbol{\theta} + \alpha \underbrace{\frac{\partial}{\partial a} Q_{\mathbf{w}}\left(s, a\right)\big|_{a=\mu_{\boldsymbol{\theta}}(s)}}_{\text{Backprop over Q-Net}} \quad \underbrace{\nabla \mu_{\boldsymbol{\theta}}\left(s\right)}_{\text{Backprop over policy}}\end{aligned}$$

Moral of Story

As long as we have an estimator of optimal action-value function, we can train deterministic policy network very easily!

Deterministic Policy Gradient

- \+ *But how can we can find such an estimator?*
- – *Well! We have done this before!*
- \+ *You mean in DQL?!*
- – *Exactly! In Q-learning we use Bellman's optimality equation to estimate optimal action-value function: we can do the same here*

Recall that Bellman's optimality equation indicate that

$$q_\star(S_t, A_t) = R_{t+1} + \gamma \mathbb{E}_{S_{t+1} \sim p(\cdot|S_t, A_t)} \left\{ \max_a q_\star(S_{t+1}, a) \right\}$$

and if we know the action $a^\star = \operatorname{argmax}_a q_\star(S_{t+1}, a)$, *we could write*

$$q_\star(S_t, A_t) = R_{t+1} + \gamma \mathbb{E}_{S_{t+1} \sim p(\cdot|S_t, A_t)} \{ q_\star(S_{t+1}, a^\star) \}$$

Deterministic Policy Gradient

If we use our deterministic policy network in one time step we sample

$$S_t, \mu_{\boldsymbol{\theta}} \left(S_{t+1} \right) \xrightarrow{R_{t+1}} S_{t+1}$$

We can then sample an estimator of optimal action-value at $a = \mu_{\boldsymbol{\theta}} \left(S_{t+1} \right)$

$$\hat{Q}_t = R_{t+1} + \gamma Q_{\mathbf{w}} \left(S_{t+1}, \mu_{\boldsymbol{\theta}} \left(S_{t+1} \right) \right)$$

Once we are over with sample trajectory: we update Q-network to minimize loss

$$\mathcal{L} \left(\mathbf{w} \right) = \frac{1}{T} \sum_{t=0}^{T-1} \left(Q_{\mathbf{w}} \left(S_t, \mu_{\boldsymbol{\theta}} \left(S_t \right) \right) - \hat{Q}_t \right)^2$$

And the life is much easier as compared to TRPO and PPO ☺

Deterministic Policy Gradient

We do very well know how to do this

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \beta \nabla \mathcal{L}(\mathbf{w}) \\ &\leftarrow \mathbf{w} + \beta \mathtt{mean}\left[\left(\hat{Q}_t - Q_{\mathbf{w}}\left(S_t, \mu_{\boldsymbol{\theta}}(S_t)\right)\right) \nabla Q_{\mathbf{w}}\left(S_t, \mu_{\boldsymbol{\theta}}(S_t)\right)\right] \\ &\leftarrow \mathbf{w} + \beta \mathtt{mean}\left[\Delta_t \nabla Q_{\mathbf{w}}\left(S_t, \mu_{\boldsymbol{\theta}}(S_t)\right)\right]\end{aligned}$$

where $\Delta_t = \hat{Q}_t - Q_{\mathbf{w}}\left(S_t, \mu_{\boldsymbol{\theta}}(S_t)\right)$ *is the TD error*

Alternating between the two update rules, we end up with a

Deterministic Policy Gradient $\equiv$ *DPG*

algorithm: there are various DPG algorithms; we take a look into the famous one

A Basic DPG Algorithm

We can use these updates to write a simple online DPG algorithm

DPG_v1():

Initiate with $\boldsymbol{\theta}$ *and* $\mathbf{w}$, *as well as factor* $\alpha < 1$ *and learning rate* β
Initiate some initial state S_0 *and draw action* A_0 *as* $A_0 \leftarrow \mu_{\boldsymbol{\theta}}(S_0)$
while** interacting **do
    Sample a time step $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
    Draw the next optimal action as $A_{t+1} \leftarrow \mu_{\boldsymbol{\theta}}(S_{t+1})$
    Compute $\Delta = R_{t+1} + \gamma Q_{\mathbf{w}}(S_{t+1}, A_{t+1}) - Q_{\mathbf{w}}(S_t, A_t)$
    Update value network as $\mathbf{w} \leftarrow \mathbf{w} + \beta \Delta \nabla Q_{\mathbf{w}}(S_t, A_t)$
    Update policy network as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{\partial}{\partial a} Q_{\mathbf{w}}(S_t, a)\big|_{a=A_t} \nabla \mu_{\boldsymbol{\theta}}(S_t)$
    Go for next state $S_t \leftarrow S_{t+1}$
    if S_t *is terminal **then***
        Draw a new random state S_0 *and* $A_0 \leftarrow \mu_{\boldsymbol{\theta}}(S_0)$
    end if
end while

Basic DPG: *Practical Challenges*

Using our knowledge, we can easily detect challenges of this basic algorithm

- *Lack of exploration* $\rightarrow$ ϵ*-greedy improvement*
  - ↳ *We follow blindly the deterministic policy network*
  - ↳ *We do not give any chance for exploration*
  - ↳ *This can quickly stick us to a bad locally-optimal deterministic policy*
- *High-variance gradient estimators* $\rightarrow$ *experience replay*
  - ↳ *It's online and hence update the networks with single time step samples*
  - ↳ *We need more samples to compute better estimators*
  - ↳ *We would like to have independent samples*
- *Variation of training labels* $\rightarrow$ *target network*
  - ↳ *Each time we update, we change the label in the training batch*
  - ↳ *This can severely deteriorate the convergence of algorithm*

DPG: ϵ-Greedy Improvement

To have sufficient exploration of environment: we can follow ϵ-greedy approach

- \+ *But how does it work here? You said we have continuous actions!*
- – *Well! We can add continuous randomness to our policy*

Say we get $A_t \leftarrow \mu_{\boldsymbol{\theta}}(S_t)$ at time t: then we replace our action with

$$A_t \leftarrow A_t + \sqrt{\epsilon} Z_t$$

where Z_t is random noise with mean zero and variance one

> *Classical choice of Z_t is zero-mean unit-variance Gaussian variable, i.e.,*
>
> $$Z_t \sim \mathcal{N}(0, 1)$$

Note that the noise term $\sqrt{\epsilon} Z_t$ is then zero-mean with variance ϵ

DPG: ϵ-Greedy Improvement

+ *But what if after adding $\sqrt{\epsilon} Z_t$, the action gets out of its allowed range? For instance, we get $A_t = 5$ and $\sqrt{\epsilon} Z_t = 3$, but we should have all actions between 2 and 6*
– *That's a valid question! We usually clip the action in this case*

To avoid out-of-range actions, we replace apply ϵ-greedy approach as

$$A_t \leftarrow \mathrm{Clip}\left(\mu_{\boldsymbol{\theta}}\left(S_t\right) + \sqrt{\epsilon} Z_t, a_{\min}, a_{\max}\right)$$

where $a_{\min}$ and $a_{\max}$ are minimum and maximum allowed actions and

$$\mathrm{Clip}\left(x, a_{\min}, a_{\max}\right) = \begin{cases} a_{\min} & x < a_{\min} \\ x & a_{\min} \leqslant x \leqslant a_{\max} \\ a_{\max} & x > a_{\max} \end{cases}$$

DPG: *Replay Buffer*

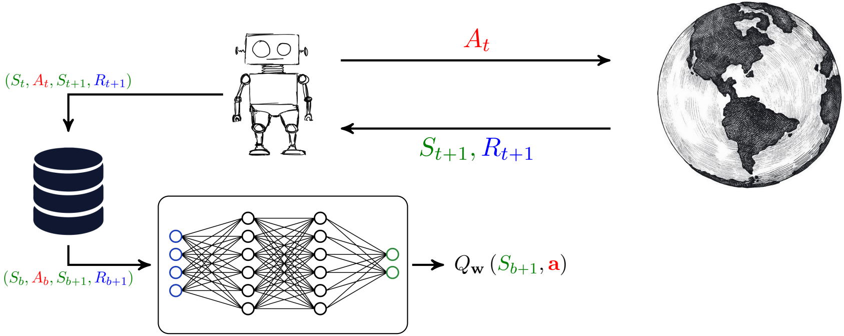
To reduce estimator's variance and enhance sample-efficiency, we can use experience replay as in DQL

+ *Wait a moment! But we talked about the fact that experience replay can increase estimator's variance in PGMs due to importance sampling argument! Now, we just ignore all those discussions?!*
– *With stochastic policy yes! But here we have a deterministic policy*

Deterministic policy returns only one action

- *For each choice of $\boldsymbol{\theta}$, our policy chooses only one action*
  - ↳ *If we change $\boldsymbol{\theta}$ we only change this action*
- *Policy update does not change the probability of all actions*
  - ↳ *This is in fact why Q-learning does not suffer from high estimate variance*

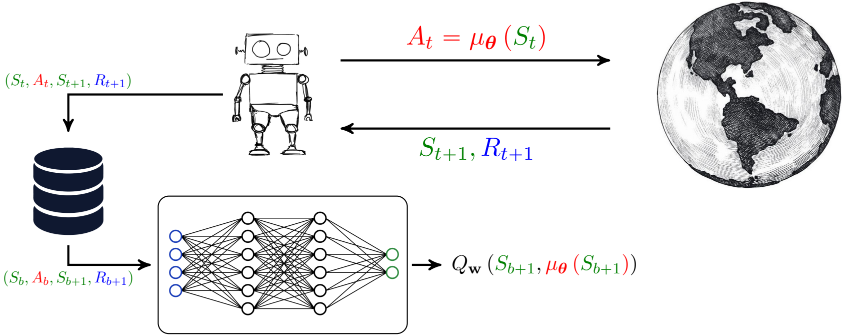
Recall: *Experience Replay in DQL*



We update DQN by randomly sampled mini-batches using TD error

$$\Delta_b \leftarrow R_{b+1} + \gamma \max_m Q_{\mathbf{w}}\left(S_{b+1}, a^m\right) - Q_{\mathbf{w}}\left(S_b, A_b\right)$$

DPG: *Experience Replay*



We now use the deterministic policy network to compute TD error

$$\Delta_b \leftarrow R_{b+1} + \gamma Q_{\mathbf{w}}\left(S_{b+1}, \mu_{\boldsymbol{\theta}}\left(S_{b+1}\right)\right) - Q_{\mathbf{w}}\left(S_b, A_b\right)$$

DPG: *Target Network*

The last thing to handle is to keep training dataset fixed for a while

- *After each mini-batch, we change both policy and Q-network*
  - ↳ *We update* $\mathbf{w}$ *and* $\boldsymbol{\theta}$
- *If we use the same networks to compute the estimate*

$$\hat{Q}_b = R_{b+1} + \gamma Q_{\mathbf{w}} \left(S_{b+1}, \mu_{\boldsymbol{\theta}} \left(S_{b+1} \right) \right)$$

then our next iteration runs over a different dataset

  - ↳ *This can cause or training loop to diverge*

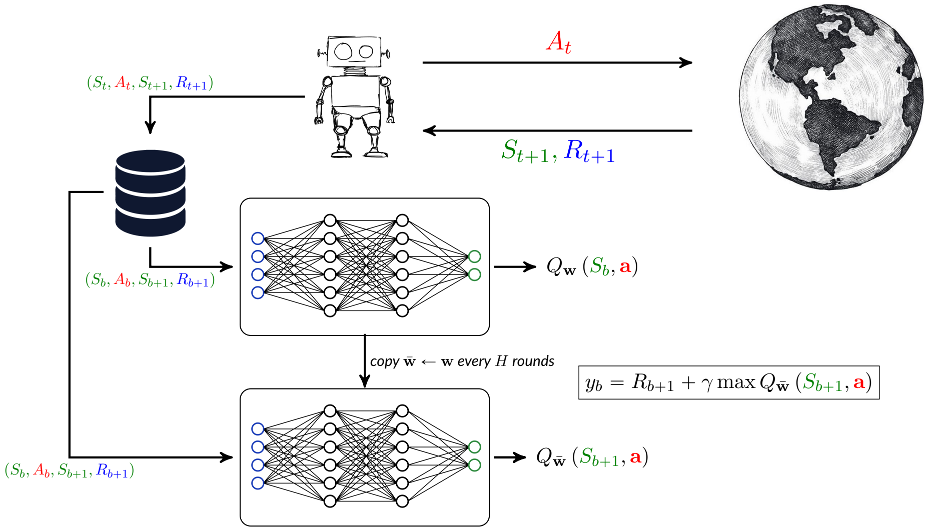
We have dealt with this in DQL using target network

Target Network in DPG

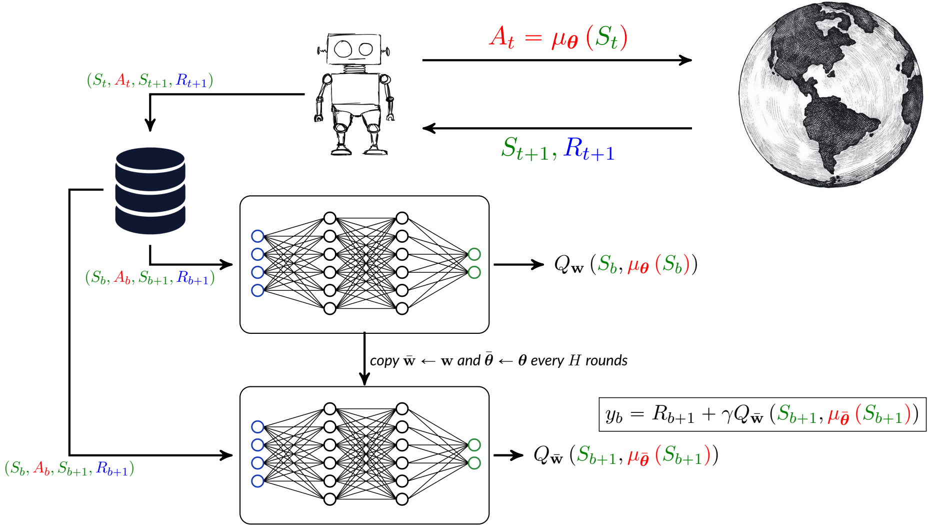
Copy DQN and policy network into the exactly same target networks

- *Use these target networks to compute the estimates*
- *Update them every multiple iterations by new copies of online networks*

Recall: *Target Network in DQL*



DPG: *Target Networks*



DDPG: *Deep DPG Algorithm*

```
DDPG():
```

Initiate with $\boldsymbol{\theta} = \bar{\boldsymbol{\theta}}$ *and* $\mathbf{w} = \bar{\mathbf{w}}$, *as well as factor* $\alpha < 1$ *and learning rate* β

Initiate state S_0 *and draw* $A_0 \leftarrow \mathrm{Clip}\left(\mu_{\boldsymbol{\theta}}\left(S_0\right) + \sqrt{\epsilon}\mathcal{N}\left(0,1\right)\right)$

while *interacting* ***do***

    Sample a time step $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$ *and save in replay buffer*

    for *multiple iterations* ***do***

        Sample $S_b, A_b \xrightarrow{R_{b+1}} S_{b+1}$ *from replay buffer*

        Draw $A_{b+1} \leftarrow \mathrm{Clip}\left(\mu_{\bar{\boldsymbol{\theta}}}\left(S_{b+1}\right) + \sqrt{\epsilon}\mathcal{N}\left(0,1\right)\right)$

        Compute $\Delta = R_{b+1} + \gamma Q_{\bar{\mathbf{w}}}\left(S_{b+1}, A_{b+1}\right) - Q_{\mathbf{w}}\left(S_b, A_b\right)$

        Update value network as $\mathbf{w} \leftarrow \mathbf{w} + \beta \Delta \nabla Q_{\mathbf{w}}\left(S_t, A_t\right)$

        Update policy network as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{\partial}{\partial a} Q_{\mathbf{w}}\left(S_t, a\right)\big|_{a=A_t} \nabla \mu_{\boldsymbol{\theta}}\left(S_t\right)$

        if H *iterations passed* ***then***

            Copy $\bar{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}$ *and* $\bar{\mathbf{w}} \leftarrow \mathbf{w}$

        end if

    end for

end while

DDPG: *Overestimate Issue*

DDPG has been the key DPG algorithm and widely used for

- *Dealing with continuous action spaces*
- *Enabling off-policy learning with a deterministic version of PGM*

DDPG has shown a key issue; namely, overestimate of values

Value Overestimate of DDPG

Action-values estimated by DDPG can have significant biases

\+ *We had it also in DQL and you said "it's not a big deal in general"! So, do we care about it here?!*

– *Well! Here is more important! Because, we use those estimates to update the policy and large biases would explode TD error!*

\+ *So, shall we do double DQL then?!*

– *Pretty much yes!*

TD3: *Twin Delayed DDPG*

Value overestimate has been addressed in the extended version of DDPG

Twin Delayed DDPG $\equiv$ *TD3*

In TD3, we add three extra tricks to DDPG

1. *We use double DQN to suppress undesired bias*
   - ↳ *Remember that bias was mainly coming out of* $\max$ *operator*
   - ↳ *We came with a remedy called double Q-learning*
2. *We delay the policy update, i.e., update the policy less frequent*
   - ↳ *We update DQN after each mini-batch, but*
   - ↳ *We update policy network once every couple of mini-batches*
3. *We add extra noise to actions when we use them in target networks*
   - ↳ *This way we make the estimation error somehow independent*
   - ↳ *This leads to less bias*

From DPG to PGM

Both ideas of learning deterministic or stochastic policy have pros and cons

- *For deterministic policy we could say*
  - ↳ *We can efficiently use off-policy learning*
  - ↳ *We can get better sample efficiency*
  - ↳ *But we get less chance of being optimal*
    - ↳ *We do not search among possible random optimal policies*
- *For stochastic policy we could say*
  - ↳ *We search among much larger set of policies*
  - ↳ *We can converge to a better policy*
  - ↳ *But we have troubles with sample efficiency*
    - ↳ *We cannot easily learn off-policy due to limits of importance sampling*

\+ *Is there any way to get good things of both worlds?*

– *Soft actor-critic approaches actually do this*

Recall: *Information Content and Entropy*

To understand the idea behind soft actor-critic, let's recap some definitions

Information Content

The information content of random variable $X \sim p(x)$ *is*

$$i(X) = \log \frac{1}{p(X)}$$

The information contents have some interesting properties

- *It's always non-negative, since* $0 \leqslant p(x) \leqslant 1$
- *The less likely outcome* $X = x$ *is, the more will be its information content*
  - ↳ *Think about it! You will find it very intuitive*

Recall: *Information Content and Entropy*

Entropy

For random variable $X \sim p(x)$, entropy is its average information content, i.e.,

$$H_p(X) = \mathbb{E}_p\left\{i(X)\right\} = \mathbb{E}_p\left\{\log\frac{1}{p(X)}\right\} = \int_x p(x)\log\frac{1}{p(x)}$$

Entropy quantifies how much confusion we have about X

- *If X is highly random, e.g., uniformly or Gaussian distributed,*
  - ↳ *Then $H_p(X)$ is very large*
- *If X is deterministic*
  - ↳ *Then $H_p(X) = 0$*

Redefining Value Function

After dealing with both deterministic and stochastic policies we might formulate the best policy as follows

- *It's globally deterministic*
  - ↳ *If one action gives better reward, it should go for it*
- *It's locally stochastic*
  - ↳ *Among actions with same rewards, it chooses one at random*

We could capture both these behaviors by looking into a new metric

Say we play with policy π: at time t, we are interested in

$$\tilde{R}_{t+1} = R_{t+1} + \xi H_{\pi}\left(A_t | S_t\right)$$

for some ξ, where $H_{\pi}\left(A_t | S_t\right)$ is entropy of action $A_t \sim \pi\left(\cdot | S_t\right)$

Redefining Value Function

Say we play with policy π: at time t, we are interested in

$$\tilde{R}_{t+1} = R_{t+1} + \xi H_\pi \left(A_t | S_t \right)$$

for some ξ, where $H_\pi \left(A_t | S_t \right)$ is entropy of action $A_t \sim \pi \left(\cdot | S_t \right)$

This new modified reward incorporates both desires

- *Being globally deterministic*
  - ↳ *For actions with larger R_{t+1}, the modified $\tilde{R}_{t+1}$ is also larger*
- *Being locally stochastic*
  - ↳ *For actions with same R_{t+1}, policy with higher randomness has larger $\tilde{R}_{t+1}$*

Well! This might be a better reward!

SAC: *Soft Actor-Critic*

We can use either DPG or PGM to develop an actor-critic method for this new reward, i.e., we could define

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left\{\sum_{i=0}^{\infty} \gamma^i \tilde{R}_{t+i+1} | S_t = s\right\} \\ &= \mathbb{E}_\pi\left\{\sum_{i=0}^{\infty} \gamma^i \left[R_{t+i+1} + \xi H_\pi\left(A_{t+i} | S_{t+i}\right)\right] | S_t = s\right\} \end{aligned}$$

Interestingly, we end up in both cases with the same policy and value gradients!

The derived actor critic method is referred to as

Soft Actor-Critic $\equiv$ *SAC*

DRL Algorithms

Most DRL algorithms used in practice are actor-critic

We already discussed all main classes of actor-critic approaches

To each class, there are various extensions

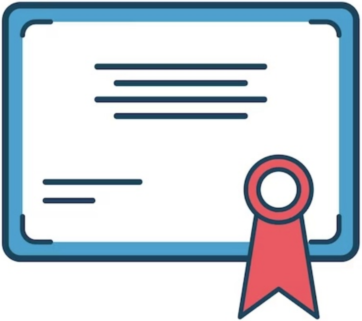
- *You are able now to follow all those extensions*
- *If necessary, you could come up with your own particular extension!*

A rule-of-thumb is

- *If you deal with discrete actions and have no concern on sample efficiency*
  ↳ *Use stochastic-policy actor-critic approaches*
- *If you deal with continuous actions and/or need sample efficiency*
  ↳ *Use DPG-like actor-critic approaches*

OpenAI: *Spinning Up in DRL*

Congratulations! You are now Deep RL experts!



Looking for some mini-projects for further practices? Take a look at OpenAI page