CHAPTER 1

DEEP REINFORCEMENT LEARNING

1 Neural Networks

...

2 Distributional Reinforcement Learning

Remember that we defined $\mathbb{P}^\pi_{s,a} := \mathbb{P}^\pi \otimes \delta_{S_0}(s) \otimes \delta_{A_0}(a)$ as the probability measure of the Markov reward process (S, A, R) started in (s, a). We define the random variable of the return under policy π as

$$Z^\pi := \sum_{t=0}^{\infty} \gamma^t R_t, \quad \gamma \in (0, 1).$$

Unlike the methods before, where we were interested in the expected reward $Q^\pi(s, a) = \mathbb{E}^\pi_{s,a}[Z^\pi]$, we are now interested in the distribution of these cumulative rewards. For that define

$$\eta^\pi_{s,a}(B) := \mathbb{P}^\pi_{s,a}(Z^\pi \in B), \quad B \in \mathcal{B}(\mathbb{R}).$$

In analogy to weak solutions for PDEs, i.e. in sense of distribution, the return distribution $\eta^\pi_{s,a}$ satisfies the Bellman equation in sense of distribution:

$$\forall \phi \in C_b(\mathbb{R}) : \quad \int_{\mathbb{R}} \phi(z) d\eta^\pi_{s,a}(z) = \mathbb{E}^\pi_{s,a}[\int_{\mathbb{R}} \phi(R + \gamma z) d\eta_{S',A'}(z)].$$

We define $f_{r,\gamma}(z) := r + \gamma z$ and the push forward

$$((\eta^\pi_{s,a})_{f_{r,\gamma}})(B) := \eta^\pi_{s,a}(f^{-1}_{r,\gamma}(B)), \quad B \in \mathcal{B}(\mathbb{R}).$$

then, the above can be written as

$$\begin{aligned}
\forall \phi \in C_b(\mathbb{R}) : \quad & \int_{\mathbb{R}} \phi(z) d\eta^\pi_{s,a}(z) = \mathbb{E}^\pi_{s,a}[\int_{\mathbb{R}} \phi(R + \gamma z) d\eta_{S',A'}(z)] = \mathbb{E}^\pi_{s,a}[\int_{\mathbb{R}} \phi(z) d(\eta_{S',A'})_{f_{R,\gamma}}(z)] \\
\iff \forall \phi \in C_b(\mathbb{R}) : \quad & \langle \phi, \eta^\pi_{s,a} \rangle = \mathbb{E}^\pi_{s,a}[\langle \phi, (\eta^\pi_{S',A'})_{f_{R,\gamma}} \rangle].
\end{aligned}$$

For any $\phi \in C_b(\mathbb{R})$ and any measure ν we have

$$\begin{aligned}
(\nu * \delta_r)(\phi) &= \int_{\mathbb{R}} \phi(z) d(\nu * \delta_r)(z) = \int_{\mathbb{R}} \int_{\mathbb{R}} \phi(x + y) d\nu(x) d\delta_r(y) \\
&= \int_{\mathbb{R}} \phi(x + r) d\nu(x) = (\nu(\cdot - r))(\phi).
\end{aligned}$$

Next, define $D_\gamma(x) := \gamma x$, then if $Z \sim \eta^\pi(x,a)$ then $\gamma Z \sim (\eta^\pi(s,a))_{D_\gamma}$. In total

$$Z\gamma + r \sim (\eta^\pi(s,a))_{D_\gamma}(\cdot - r) = ((\eta^\pi(s,a))_{D_\gamma} * \delta_r)$$

holds in sense of distribution. Now define the distributional Bellman operator in sense of distribution as

$$\mathcal{T}^\pi Z^\pi(s,a) = R(s,a) + \gamma Z^\pi(S', A'),$$

i.e. the distribution

$$\mathcal{T}^\pi : (\mathcal{P}(\mathbb{R}))^{\mathcal{S}\times\mathcal{A}} \to (\mathcal{P}(\mathbb{R}))^{\mathcal{S}\times\mathcal{A}}, (\nu_{s,a})_{(s,a)\in\mathcal{S}\times\mathcal{A}} \mapsto ((\mathcal{T}^\pi\nu)_{s,a})_{(s,a)\in\mathcal{S}\times\mathcal{A}}$$

is point wise defined for all $\phi \in C_b(\mathbb{R})$ as

$$(\mathcal{T}^\pi\nu)_{s,a}(\phi) := \mathbb{E}^\pi_{s,a}[((\nu(S',A'))_{D_\gamma} * \delta_R)(\phi)] = \sum_{s',a',r} \pi(a';s')p(s',r;s,a)((\nu_{s',a'})_{D_\gamma} * \delta_r)(\phi)$$

3 Deep Q-Networks

Deep Q-Networks (DQN) is a Q-learning variant that utilizes deep neural networks to approximate the Q-value. This algorithm addresses how to deal with large, continuous state action spaces $(\mathcal{S}, \mathcal{A})$. The idea is to reduce dimensionality by plugging in the state space $\mathcal{S}$ (e.g. pixels in an Atari game) into a convolutional neural layer, which is the fed into a feed forward neural network that outputs the Q-value for each action $a \in \mathcal{A}$. Problems with simple neural networks are

- They can forget
- Are unstable: small changes in Q-values can lead to big changes in the policy, i.e. the action distribution
- Correlation: In a trajectory τ_i of a rollout, the consecutive states s_t, s_{t+1} are highly correlated, which violates the i.i.d. assumption for estimating the expectation.
- The loss is non-stationary, since the target values depend on the parameters of the network itself, meaning
$$\mathcal{L}(\theta) := \mathbb{E}[(\mathbb{E}[R + \gamma \max_{a'} Q^\theta(S',a')] - Q^\theta(S,A))^2],$$
where both Q terms depend on θ.

In DQN these issues where solved by the following

- Experience Replay: A behavoiral policy π^b samples Rollouts, that are stored in a buffer $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}_{i=1}^N$. In order to break the correlation between consecutive states, mini batches are sampled uniformly from this buffer to update the Q-network.
- Target Network: A second network with parameters θ^- is introduced, which is bootstrapped, i.e. every C steps the parameters of the Q-network are copied to the target network $\theta^- \leftarrow \theta$. This delay stabilizes the training.

Tabular Q-learning can be interpreted in the following two ways:

- ML view: Fit a target Q_n to the data Q^{π^*}. Define $Y_n := R + \gamma \max_{a'\in\mathcal{A}} Q_n(S',a')$ and
$$\tilde{L}_n^Q := \mathbb{E}[(\mathbb{E}_{S,A}[Y_n] - Q_n(S,A))^2] = \mathbb{E}[(Y_n - Q_n(S,A))^2] + \mathbb{E}[\mathbb{V}_{S,A}(Y_n)] =: L_n^Q + \mathbb{E}[\mathbb{V}_{S,A}(Y_n)].$$

- RL view: We want to approximate the Bellman optimality operator

$$T^*Q(s,a) := \mathbb{E}_{s,a}[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')].$$

  Because it is a contraction in $\| \cdot \|_\infty$, the fixed point iteration converges.

Because the expectation is the minimizer of the L^2 error (variance), it holds that $\mathbb{E}[X] = \arg\min_\theta \mathbb{E}[(X-\theta)^2]$ which leads to

$$T^*Q(s,a) = \mathbb{E}_{s,a}[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')] = \arg\min_\theta \mathbb{E}_{s,a}[(R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') - \theta)^2].$$

Thus doing one step SGD (with step size $\frac{\alpha}{2}$???) on L_n^Q or doing the iteration for Q-learning yields the same scheme:

$$Q_{n+1}(s,a) = Q_n(s,a) + \alpha(y - Q_n(s,a)).$$

Remark 3.1

When doing Q-learning with neural networks, we no longer do the gradient on $(s,a) \in \mathcal{S} \times \mathcal{A}$, but instead on a smaller parameter space $\Theta \subset \mathcal{S} \times \mathcal{A}$. Additionally, parameterizing the target with a different set of parameters θ^- the gradient of the loss becomes

$$\nabla_\theta L_n^Q = \mathbb{E}[\nabla_\theta (Y_n^{\theta^-} - Q_n^\theta(S,A))^2] = -\mathbb{E}[2(Y_n^{\theta^-} - Q_n^\theta(S,A))\nabla_\theta Q_n^\theta(S,A)].???-$$

So the gradient step for SGD becomes

$$Q_{n+1}^\theta(s,a) = Q_n^\theta(s,a) + \alpha(y_n^{\theta^-} - Q_n^\theta(s,a))\nabla_\theta Q_n^\theta(s,a).$$

Remark 3.2

In a convolutional neural network a convolutional layer is given by

$$(f_k * h)(i) = \sum_{j=-n}^{n} f_k(i-j)h(i),$$

where f_i is a filter $k = 1, ..., K$ and h is some input vector. These K filters operator simultaninously to and produce $(f_k * h)(i), k = 1, ..., K$ as output. K is the number of Filters. In this setting the step size is called stride and was in the above 1. In general it is

$$(f_k * h)(i) = \sum_{j=-n}^{n} f_k(i+s-j)h(i), \quad s \in \mathbb{N}.$$

Remark 3.3

For Atari games the Architecture is as follows:

- Preprocessing: The environment 84×84 pixels is converted into grayscale. Let $x_t \in \mathbb{R}^{H \times W}$ be a greyscale image at time t, where $H = W = 84$. To create a sense of motion four consecutive images are stacked: $s_t := (x_t, ..., x_{t-3}) \in \mathbb{R}^{H \times W \times 4}$. Together with the scores, this is fed into the neural network.

- Convolutional Layers: There are three convolutional layers used. Each convolutional layers is followed by a ReLU activation function. The first layer has 32 filters of size 8×8 with stride 4. The second layer has 64 filters of size 4×4 with stride 2. The third layer has 64 filters of size 3×3 with stride 1.
- Feed Forward Layers: The output of the last convolutional layer is flattened and fed into a fully connected layer with 512 units, followed by a ReLU activation function.
- Output: The final layer in the network is a fully connected layer that outputs a vector of Q-values, one for each possible action in the Atari game.

Remark 3.4

- Experience Replay: This solves two problems: From the buffer $\mathcal{D}$ mini batches are chosen following an ϵ-greedy policy, which solves the problem of the correlation between rollouts and the policy. Further, it also solves the problem, that neural networks are prone to forget old data.
  If the buffer is full, the oldest transitions are removed first.
- Target Networks:
  - Different Parametrization: The target distribution is parametrized with θ^- Every N steps it is updated with a soft update rule
    $$\theta^- \leftarrow \tau\theta^- + (1-\tau)\theta.$$
    This stabilizes training:
    $$\tilde{L}_n(\theta) := \mathbb{E}[\Big(\mathbb{E}_{S,A}[Y_n^{\theta^-}] - Q_n^{\theta}(s,a)\Big)^2] = L_n(\theta) + \mathbb{E}[\mathbb{V}_{S,A}[Y_n^{\theta^-}]],$$
    because $\tilde{L}_n(\theta)$ no longer depends on the expected variance.
  - Update every N steps: With the second point of view of Q learning, updating every N steps is actually N step SGD of
    $$T^*Q^{\theta^-}(s,a) = \mathbb{E}_{s,a}[R + \gamma Q^{\theta^-}(S',a')] = \arg\min_{\theta} \mathbb{E}[(Y_n^{\theta^-} - \theta)^2].$$

Algorithm 1: The Deep Q-Network (DQN) algorithm.

```
Data: Replay buffer capacity $|D|$, discount factor $\gamma$, exploration rate $\varepsilon$, target update frequency $N$
Result: Trained Q-network approximating $Q^*$
Initialise replay buffer $D$
Initialise Q-network with random weights $\theta$
Copy weights to target network: $\theta^- \leftarrow \theta$
for each episode do
    Pre-process the initial observation $s_1$ to obtain state representation $\phi_1$
    for each time-step $t$ do
        Select action $a_t$ using $\varepsilon$-greedy policy based on $Q(\phi_t, a; \theta)$
        Execute action $a_t$, observe reward $r_t$ and next observation $s_{t+1}$
        Pre-process $s_{t+1}$ to obtain $\phi_{t+1}$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in replay buffer $D$
        Sample mini-batch $B$ of transitions from $D$
        for each transition $i \in B$ do
            Compute target:
            $Y_i = \begin{cases} r_i, & \text{if terminal transition,} \\ r_i + \gamma \max_{a'} Q(\phi_i', a'; \theta^-), & \text{otherwise.} \end{cases}$
        end
        Perform gradient descent on loss:
            $L_B(\theta) = \frac{1}{|B|} \sum_{i \in B} (Y_i - Q(\phi_i, a_i; \theta))^2$
          Update Q-network weights $\theta$
        if every $N$ steps then
            Update target network: $\theta^- \leftarrow \theta$
        end
        However, target networks increase memory, as it another neural network needs to be stored
          and updating the weights increases computational costs.
    end
end
```

Remark 3.5

- Double Q-learning and Double DQN: Q-learning over estimates the action values due to the max. operator, In Double Q-learning we maintain two independent networks Q^A and Q^B to estimate Q^{π^*}:

$$
\begin{aligned}
Q^A(s,a) &\leftarrow (1-\alpha)Q^A(s,a) + \alpha\left(r + \gamma Q^B(s', \arg\max_{a' \in \mathcal{A}} Q^A(s',a'))\right) \\
&= (1-\alpha)Q^A(s,a) + \alpha\left(r + \gamma Q^A(s', \arg\max_{a' \in \mathcal{A}} Q^A(s',a'))\right) \\
&+ \gamma\left(Q^B(s', \arg\max_{a' \in \mathcal{A}}(s',a')) - Q^A(s', \arg\max_{a' \in \mathcal{A}} Q^A(s',a'))\right).
\end{aligned}
$$

In Double Q-learning the Networks are updated in every step and independendly. In Double DQN we give one network the role of the target network and the other the Q-network. The Q-network is updated every step. The target network is updated only every N step with the soft update rule, which depends on the Q-network. So indepence and simultaninous updates do not happen.

- Clipped Double Q/TD3: Another approach is to clip the bias term (third summand in the above) by taking its negative part: $b^- := \min\{b, 0\}$. The deep variant of this method is called Twin delayed double Deep Q Networks.

4 Proximal Policy Gradient

Consider a finite time MDP and the respective Value/Q-functions:

$$V^\pi(s) := \mathbb{E}_s^\pi[\sum_{t=0}^{T-1} \gamma^t R_t], \quad V_t^\pi[\sum_{t'=t}^{T-1} \gamma^{t'} R_{t'}], \quad Q_t^\pi(s,a) = \mathbb{E}_{s,a}^\pi[\sum_{t'=t}^{T_1} \gamma^{t'} R_{t'}],$$

where $(\pi_\theta)^{\theta\in\Theta}$ is a family of differentiable policies. The gradient is then given by

$$\nabla_\theta V^{\pi_\theta}(s) = \sum_{t=0}^{T-1} \gamma^t \mathbb{E}^{\pi_\theta}[\nabla_\theta \log(\pi_\theta(A_t; S_t)) R_t^T], \quad R_t^T := \sum_{i=t}^{T-1} \gamma^{i-t} R_i.$$

The idea is now threefold. Because the gradient is an expectation, we need to sample from it. We want to reuse as many samples as possible, do as few updates of the parameter θ as possible to reduce computational time and reduce the variance of the expectation:

- Reusing samples: We think of PPO as a rollout based stochastic gradient approximation. We reuse old parameters for gradient steps (surragate gradient steps) and show that these biased gradient steps have an advantage. Let D^{π_θ} be the set of Rollouts created by π_θ. Randomly reshuffle this set and decompose it into mini-batches. We pass through this dataset, by doing a gradient step for each mini batch. After we have passed through the entire dataset (one epoch) we set $\theta := \theta_{old}$ and start a new cycle.

- Reduce updates:

- Reduce variance: Subtracting a Basline can reduce variance. We choose the Baseline as the Value function, i.e.

  $$\nabla_\theta V^{\pi_\theta}(s) = \sum_{t=0}^{T-1} \gamma^t \mathbb{E}^{\pi_\theta}[\nabla_\theta \log(\pi_\theta(A_t; S_t)) \underbrace{(Q_t^{\pi_\theta} - V_t^{\pi_\theta})}_{=:A_t^{\pi_\theta}}],$$

  where $A_t^{\pi_\theta}$ is called advantage function.
  Another way is to decouple sampling and updating, using importance sampling:

  $$\nabla_\theta V^{\pi_\theta}(s) = \sum_{t=0}^{T-1} \gamma^t \mathbb{E}^{\pi_{\theta^{old}}} \left[\sum_{i=0}^{t} \frac{\pi_\theta(A_i; S_i)}{\pi_{\theta^{old}}(A_i; S_i)} \nabla_\theta \log(\pi_\theta(A_t; S_t)) A_t^{\pi_\theta}(S_t, A_t) \right].$$

Thus the current policy needs to be evaluted and $A_t^{\pi_\theta}$ needs to be computed. The alternation between gradient steps and policy evalutaiton is called actor critic. A2C does this by using neural networks to parameterize

the Advantage function and uses GAE for advantage estimation.
GAE is used for infinite time MDPs. Also used for finite time by truncating at T. Usually γ is ignored, but this is required???

Remark 4.1

Assumptions:

- Bounded rewards: $|R_t| \leq R_*$
- Bounded score function: $\forall \theta \in \mathbb{R}^d, s \in \mathcal{S}, a \in \mathcal{A}: \quad |\nabla_\theta \log(\pi_\theta(a;s))| \leq \Pi_*$
- Lipschitz score function:

$$\forall \theta \in \mathbb{R}^d, s \in \mathcal{S}, a \in \mathcal{A}: \quad |\nabla_\theta \log(\pi_\theta(a;s)) - \nabla_{\theta'} \log(\pi_{\theta'}(a;s))| \leq L_s|\theta - \theta'|.$$

- Access to $\hat{A}_t$ which is bounded by A_*.

5 Advantage Estimation

6 Variants to PPO

7 Stable Baselines3

8 Hyperparameter Tuning