Model Predictive Control and Reinforcement Learning
– On-Policy Control with Function Approximation –

Joschka Boedecker and Moritz Diehl

University Freiburg

October 6, 2022
Lecture Overview

1. Function Approximation in Reinforcement Learning
2. Linear Methods
3. On-policy Control with Function Approximation
4. Off-policy Learning
5. Problems of Off-policy Learning with Function Approximation
6. Deep Q-learning
7. DDPG
8. TD3
Acknowledgement

Slide contents are partially based on *Reinforcement Learning: An Introduction* by Sutton and Barto and the Reinforcement Learning lecture by David Silver.
Function Approximation in Reinforcement Learning

- Up to this point, we represented all elements of our RL systems by tables (value functions, models and policies)
- If the state and action spaces are very large or infinite, this is not a feasible solution
- We can apply function approximation to find a more compact representation of RL components and to generalize over states and actions
- Reinforcement Learning with function approximation comes with new issues that do not arise in Supervised Learning – such as non-stationarity, bootstrapping and delayed targets
Here: we estimate value-functions $v_{\pi}(\cdot)$ and $q_{\pi}(\cdot, \cdot)$ by function approximators $\hat{v}(\cdot, w)$ and $\hat{q}(\cdot, \cdot, w)$, parameterized by weights $w$

But we can also represent models or policies
We can use different types of function approximators:

- Linear combinations of features
- Neural networks
- Decision trees
- Gaussian processes
- Nearest neighbor methods
- ...

Here: We focus on differentiable FAs and update the weights via gradient descent.
We want to update our weights w.r.t. the *Mean Squared Value Error* of our prediction:

\[
\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2} \alpha \nabla \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\
= \mathbf{w}_t + \alpha \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)
\]

However, we don’t have \( v_\pi(S_t) \).
### Function Approximation in Reinforcement Learning

#### Gradient MC

\[
\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})
\]

#### Semi-gradient TD(0)

\[
\mathbf{w} \leftarrow \mathbf{w} + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})
\]

Why are bootstrapping methods, defined this way, called *semi-gradient methods*?
Function Approximation in Reinforcement Learning

Gradient MC

\[ w \leftarrow w + \alpha [G_t - \hat{v}(S_t, w)] \nabla \hat{v}(S_t, w) \]

Semi-gradient TD(0)

\[ w \leftarrow w + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)] \nabla \hat{v}(S_t, w) \]

Why are bootstrapping methods, defined this way, called *semi-gradient methods*? They take into account the effects of changing \( w \) w.r.t. the prediction, but not w.r.t. the target!
Linear Methods

- Represent state $s$ by feature vector $x(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$
- These features can also be non-linear functions/combinations of state dimensions
- Linear methods approximate the value function by a linear combination of these features

$$\hat{v}(s, w) = w^\top x(s) = \sum_{i=1}^{d} w^i x^i(s)$$

- Therefore, $\nabla_w \hat{v}(s, w) = x(s)$
- Gradient MC prediction converges under linear FA
- On-policy linear semi-gradient TD(0) is stable
- Unfortunately, this does not hold for non-linear FA
The update at each time step $t$ is:

$$
\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left( R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t \right) \mathbf{x}_t
$$

$$
= \mathbf{w}_t + \alpha \left( R_{t+1}\mathbf{x}_t - \mathbf{x}_t \left( \mathbf{x}_t - \gamma \mathbf{x}_{t+1} \right)^\top \mathbf{w}_t \right)
$$

The expected next weight vector can thus be written:

$$
\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha (\mathbf{b} - \mathbf{A}\mathbf{w}_t),
$$

where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathbf{A} = \mathbb{E}[\mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]$.

If the system converges, it has to converge to the fixed point:

$$
\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}
$$
Coarse Coding

Divide the state space in circles/tiles/shapes and check in which some state is inside. This is a binary representation of the location of a state and leads to generalization.
Again, up to this point we discussed Policy Evaluation based on state value functions.

In order to apply FA in control, we parameterize the action-value function using Semi-gradient SARSA:

\[
\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}(S_t, A_t; \mathbf{w}) \right] \nabla \hat{q}(S_t, A_t; \mathbf{w})
\]
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : S \times A \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$

Initialize value-function weights $w \in \mathbb{R}^d$ arbitrarily (e.g., $w = 0$)

Loop for each episode:

- $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)

Loop for each step of episode:

- Take action $A$, observe $R, S'$
- If $S'$ is terminal:
  - $w \leftarrow w + \alpha \left[ R - \hat{q}(S, A, w) \right] \nabla \hat{q}(S, A, w)$
  - Go to next episode
- Choose $A'$ as a function of $\hat{q}(S', \cdot, w)$ (e.g., $\varepsilon$-greedy)
  - $w \leftarrow w + \alpha \left[ R + \gamma \hat{q}(S', A', w) - \hat{q}(S, A, w) \right] \nabla \hat{q}(S, A, w)$
  - $S \leftarrow S'$
  - $A \leftarrow A'$
Off-policy Learning

- We want to learn the optimal policy, but we have to account for the problem of maintaining exploration.
- We call the (optimal) policy to be learned the target policy $\pi$ and the policy used to generate behaviour the behaviour policy $b$.
- We say that learning is from data off the target policy – thus, those methods are referred to as off-policy learning.
Importance Sampling

- Weight returns according to the relative probability of target and behaviour policy
- Define state-transition probabilities $p(s'|s, a)$ as
  \[ p(s'|s, a) = \Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a) \]
- The probability of the subsequent trajectory under any policy $\pi$, starting in $S_t$, then is:
  \[
  \Pr\{A_t, S_{t+1}, A_{t+1}, \ldots S_T|S_t, A_{t:T-1} \sim \pi\} = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1})
  \]
  \[
  = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)
  \]
Importance Sampling

The relative probability therefore is:

**Definition: Importance Sampling Ratio**

\[
\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)}
\]

The expectation of the returns by \( b \) is \( \mathbb{E}[G_t|S_t = s] = v_b(s) \). However, we want to estimate the expectation under \( \pi \). Given the importance sampling ratio, we can transform the MC returns by \( b \) to yield the expectation under \( \pi \):

\[
\mathbb{E}[\rho_{t:T-1} G_t|S_t = s] = v_\pi(s).
\]

Importance Sampling can come with a vast increase in variance.
To use importance sampling with function approximation, replace the update to an array to an update to weight vector $w$, and correct it with the importance sampling weight.

### Off-policy MC Prediction

$$w \leftarrow w + \alpha \rho_{t:T-1}[G_t - \hat{v}(S_t, w)] \nabla \hat{v}(S_t, w)$$

### Semi-gradient Off-policy TD(0)

$$w \leftarrow w + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, w)$$

where $\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$
Baird’s Counterexample

The reward is 0 for all transitions, hence \( v_{\pi}(s) = 0 \). This could be exactly approximated by \( w = 0 \).
Baird’s Counterexample

Semi-gradient DP

\[ \mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{|S|} \sum_{s \in S} \left( \mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) | S_t = s] - \hat{v}(s, \mathbf{w}) \right) \nabla \hat{v}(s, \mathbf{w}) \]
The combination of

- Function Approximation,
- Bootstrapping and
- Off-policy Learning

is known as the *Deadly Triad*, since it can lead to stability issues and divergence.
Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

- Model-free off-policy RL algorithm that works on continuous state and discrete action spaces
- Q-function is represented by a multi-layer perceptron
- One of the first approaches that combined RL with ANNs, predecessor of DQN
Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

\begin{algorithm}
\begin{algorithmic}
\For{iteration $i = 1, \ldots, N$}
\State sample trajectory with $\epsilon$-greedy exploration and add to memory $D$
\State initialize network weights randomly
\State generate pattern set $P = \{(x_j, y_j) | j = 1..|D|\}$ with\end{algorithmic}
\begin{align*}
x_j &= (s_j, a_j) \quad \text{and} \quad y_j = \begin{cases} r_j & \text{if } s_j \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', w_i) & \text{else} \end{cases}
\end{align*}
\For{iteration $k = 1, \ldots, K$}
\State Fit weights according to:
\begin{align*}
L(w_i) &= \frac{1}{|D|} \sum_{j=1}^{|D|} (y_j - Q(x_j, w_i))^2
\end{align*}
\EndFor
\EndFor
\end{algorithm}

\textbf{Algorithm 1:} NFQ
Deep Q-Networks (DQN)

DQN provides a stable solution to deep RL:
- Use experience replay (as in NFQ)
- Sample minibatches (as opposed to Full Batch in NFQ)
- Freeze target Q-networks (no target networks in NFQ)
- Optional: Clip rewards or normalize network adaptively to sensible range
Deep Q-Networks: Experience Replay

To remove correlations, build data set from agent’s own experience

- Take action $a_t$ according to $\epsilon$-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $D$
- Sample random mini-batch of transitions $(s, a, r, s')$ from $D$
- Optimize MSE between Q-network and Q-learning targets, e.g.

$$L(w) = \mathbb{E}_{s,a,r,s' \sim D} [(r + \gamma \max_{a'} Q(s', a', w) - Q(s,a,w))^2]$$
Deep Q-Networks: Target Networks

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters $w^-$
  \[
  r + \gamma \arg \max_{a'} Q(s', a', w^-)
  \]

- Optimize MSE between Q-network and Q-learning targets
  \[
  L(w) = \mathbb{E}_{s, a, r, s' \sim D} [(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w))^2]
  \]

- Periodically update fixed parameters $w^- \leftarrow w$
  - hard update: update target network every $N$ steps
  - slow update: slowly update weights of target network every step by
    \[
    w^- \leftarrow (1 - \tau)w^- + \tau w
    \]
Deep Q-Networks (DQN)

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights

for episode $i = 1, \ldots, M$ do
    for $t = 1, \ldots, T$ do
        select action $a_t$ $\epsilon$-greedily
        Store transition $(s_t, a_t, s_{t+1}, r_t)$ in $D$
        Sample minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', w^-) & \text{else} \end{cases}$
        Update the parameters of $Q$ according to:
        $$\nabla w_i L_i(w_i) = \mathbb{E}_{s, a, s', r \sim D} [(r + \gamma \max_{a'} Q(s', a', w_i) - Q(s, a, w_i)) \nabla w_i Q(s, a, w_i)]$$
    end
end

Update target network
Deep Q-Networks: Reinforcement Learning in Atari
Deep Q-Networks: Reinforcement Learning in Atari

- End-to-end learning of values $Q(s, a)$ from pixels $s$
- Input state $s$ is a stack of raw pixels from the last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step
How much does DQN help?

<table>
<thead>
<tr>
<th>Game</th>
<th>Q-Learning</th>
<th>Q-Learning</th>
<th>Q-Learning</th>
<th>DQN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Target Q</td>
<td>+ Replay</td>
<td>+ Target Q</td>
<td></td>
</tr>
<tr>
<td>Breakout</td>
<td>3</td>
<td>10</td>
<td>241</td>
<td>317</td>
</tr>
<tr>
<td>Enduro</td>
<td>29</td>
<td>142</td>
<td>831</td>
<td>1006</td>
</tr>
<tr>
<td>River Raid</td>
<td>1453</td>
<td>2868</td>
<td>4103</td>
<td>7447</td>
</tr>
<tr>
<td>Seaquest</td>
<td>276</td>
<td>1003</td>
<td>831</td>
<td>2894</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>302</td>
<td>373</td>
<td>826</td>
<td>1089</td>
</tr>
</tbody>
</table>
Deep Deterministic Policy Gradient

- DDPG is an actor-critic method (*Continuous DQN*)
- Recall the DQN-target: $y_j = r_j + \gamma \max_a Q(s_{j+1}, a, w^-)$
- In case of continuous actions, the maximization step is not trivial
- Therefore, we approximate deterministic actor $\mu$ representing the $\arg \max_a Q(s_{j+1}, a, w)$ by a neural network and update its parameters following the *Deterministic Policy Gradient Theorem*:

$$\nabla_\theta J \approx \frac{1}{N} \sum_j \nabla_a Q(s_j, a, w)|_{a=\mu(s_j)} \nabla_\theta \mu(s_j, \theta)$$

- Exploration by adding Gaussian noise to the output of $\mu$
The Q-function is fitted to the adapted TD-target:

\[ y_j = r_j + \gamma Q(s_{j+1}, \mu(s_{j+1}, \theta^\prime), w^-) \]

The parameters of target networks \( \mu(\cdot, \theta^-) \) and \( Q(\cdot, \cdot, w^-) \) are then adjusted with a soft update:

\[ w^- \leftarrow (1 - \tau)w^- + \tau w \quad \text{and} \quad \theta^- \leftarrow (1 - \tau)\theta^- + \tau \theta \]

with \( \tau \in [0, 1] \)

DDPG is very popular and builds the basis for more SOTA actor-critic algorithms

However, it can be quite unstable and sensitive to its hyperparameters
Deep Deterministic Policy Gradient

Initialize replay memory \( D \) to capacity \( N \)
Initialize critic \( Q \) and actor \( \mu \) with random weights

for episode \( i = 1, \ldots, M \) do

  for \( t = 1, \ldots, T \) do

    select action \( a_t = \mu(s_t, \theta) + \epsilon \), where \( \epsilon \sim \mathcal{N}(0, \sigma) \)
    Store transition \( (s_t, a_t, s_{t+1}, r_t) \) in \( D \)
    Sample minibatch of transitions \( (s_j, a_j, r_j, s_{j+1}) \) from \( D \)
    Set \( y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma Q(s_{j+1}, \mu(s_{j+1}, \theta^-), w^-) & \text{else} \end{cases} \)
    Update the parameters of \( Q \) according to the TD-error
    Update the parameters of \( \mu \) according to:
    \[
    \nabla_\theta J \approx \frac{1}{N} \sum_j \nabla_a Q(s_j, a, w)|_{a = \mu(s_j)} \nabla_\theta \mu(s_j, \theta)
    \]

  end

end

Adjust the parameters of the target networks via a soft update
In all control algorithms so far, the target policy is created by the maximization of a value-function.

We thus consider the maximum over estimated values as an estimate of the maximum value.

This can lead to the so-called overestimation bias.
Recall the Q-learning target: \( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \)

Imagine two random variables \( X_1 \) and \( X_2 \):

\[
\mathbb{E} \left[ \max(X_1, X_2) \right] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])
\]

\( Q(S_{t+1}, a) \) is not perfect – it can be noisy:

\[
\max_a Q(S_{t+1}, a) = Q(S_{t+1}, \arg \max_a Q(S_{t+1}, a))
\]

If the noise in these is decorrelated, the problem goes away!
Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in S^+, a \in A(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
    Take action $A$, observe $R$, $S'$
    With 0.5 probability:
    $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left( R + \gamma Q_2(S', \text{argmax}_a Q_1(S', a)) - Q_1(S, A) \right)$
    else:
    $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left( R + \gamma Q_1(S', \text{argmax}_a Q_2(S', a)) - Q_2(S, A) \right)$
  $S \leftarrow S'$
  until $S$ is terminal
Double Q-learning

% left actions from A

Episodes

Double Q-learning

Q-learning

$\mathcal{N}(-0.1, 1)$
TD3 adds three adjustments to vanilla DDPG

- Clipped Double Q-Learning
- Delayed Policy Updates
- Target-policy smoothing
TD3: Clipped Double Q-Learning

- In order to alleviate the overestimation bias (which is also present in actor-critic methods), TD3 learns two approximations of the action-value function.
- It takes the minimum of both predictions as the second part of the TD-target:

\[ y_j = r_j + \gamma \min_{i \in \{1, 2\}} Q(s_{j+1}, \mu(s_{j+1}), w_i^-) \]
TD3: Delayed Policy Updates

Due to the mutual dependency between actor and critic updates...

- values can diverge when the policy leads to overestimation and
- the policy will lead to bad regions of the state-action space when the value estimates lack in (relative) accuracy

Therefore, policy updates on states where the value-function has a high prediction error can cause divergent behaviour

- We already know how to compensate for that: target networks
- Freeze target and policy networks between $d$ updates of the value function
- This is called a *Delayed Policy Update*
Target-policy Smoothing adds Gaussian noise to the next action in target calculation.

It transforms the Q-update towards an Expected SARSA update fitting the value of a small area around the target-action:

\[ y_j = r_j + \gamma \min_{i \in \{1,2\}} Q(s_j+1, \mu(s_j+1) + \text{clip}(\epsilon, -c, c), w_i^-), \]

where \( \epsilon \sim N(0, \sigma) \).
Table 2. Average return over the last 10 evaluations over 10 trials of 1 million time steps, comparing ablation over delayed policy updates (DP), target policy smoothing (TPS), Clipped Double Q-learning (CDQ) and our architecture, hyper-parameters and exploration (AHE). Maximum value for each task is bolded.

<table>
<thead>
<tr>
<th>Method</th>
<th>H-Cheetah</th>
<th>Hopper</th>
<th>Walker2d</th>
<th>Ant</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD3</td>
<td>9532.99</td>
<td><strong>3304.75</strong></td>
<td><strong>4565.24</strong></td>
<td>4185.06</td>
</tr>
<tr>
<td>DDPG</td>
<td>3162.50</td>
<td>1731.94</td>
<td>1520.90</td>
<td>816.35</td>
</tr>
<tr>
<td>AHE</td>
<td>8401.02</td>
<td>1061.77</td>
<td>2362.13</td>
<td>564.07</td>
</tr>
<tr>
<td>AHE + DP</td>
<td>7588.64</td>
<td>1465.11</td>
<td>2459.53</td>
<td>896.13</td>
</tr>
<tr>
<td>AHE + TPS</td>
<td>9023.40</td>
<td>907.56</td>
<td>2961.36</td>
<td>872.17</td>
</tr>
<tr>
<td>AHE + CDQ</td>
<td>6470.20</td>
<td>1134.14</td>
<td>3979.21</td>
<td>3818.71</td>
</tr>
<tr>
<td>TD3 - DP</td>
<td>9590.65</td>
<td>2407.42</td>
<td><strong>4695.50</strong></td>
<td>3754.26</td>
</tr>
<tr>
<td>TD3 - TPS</td>
<td>8987.69</td>
<td>2392.59</td>
<td>4033.67</td>
<td><strong>4155.24</strong></td>
</tr>
<tr>
<td>TD3 - CDQ</td>
<td>9792.80</td>
<td>1837.32</td>
<td>2579.39</td>
<td>849.75</td>
</tr>
<tr>
<td>DQ-AC</td>
<td>9433.87</td>
<td>1773.71</td>
<td>3100.45</td>
<td>2445.97</td>
</tr>
<tr>
<td>DDQN-AC</td>
<td><strong>10306.90</strong></td>
<td>2155.75</td>
<td>3116.81</td>
<td>1092.18</td>
</tr>
</tbody>
</table>