Function Approximation, Deep Q Network

Milan Straka

November 11, 2019
Where Are We

- Until now, we have solved the tasks by explicitly calculating expected return, either as $v(s)$ or as $q(s, a)$.
  - Finite number of states and actions.
  - We do not share information between different states or actions.
  - We use $q(s, a)$ if we do not have the environment model (a \textit{model-free} method); if we do, it is usually better to estimate $v(s)$ and choose actions as $\arg \max_a \mathbb{E}R + v(s')$.

- The methods we know differ in several aspects:
  - Whether they compute return by simulating whole episode (Monte Carlo methods), or by using bootstrapping (temporal difference, i.e., $G_t \approx R_t + v(S_t)$, possibly $n$-step)
    - TD methods more noisy and unstable, but can learn immediately and explicitly assume Markovian property of value function
  - Whether they estimate the value function of the same policy they use to generate episodes (on-policy) of not (off-policy)
    - off-policy methods again more noisy and unstable, but more flexible
We will approximate value function \( v \) and/or state-value function \( q \), choosing from a family of functions parametrized by a weight vector \( \mathbf{w} \in \mathbb{R}^d \).

We will denote the approximations as

\[
\hat{v}(s, \mathbf{w}), \\
\hat{q}(s, a, \mathbf{w}).
\]

Weights are usually shared among states. Therefore, we need to define state distribution \( \mu(s) \) to allow an objective for finding the best function approximation.

The state distribution \( \mu(s) \) gives rise to a natural objective function called *Mean Squared Value Error*, denoted \( \overline{VE} \):

\[
\overline{VE}(\mathbf{w}) \stackrel{\text{def}}{=} \sum_{s \in S} \mu(s) [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2.
\]
Function Approximation

For on-policy algorithms, $\mu$ is usually on-policy distribution. That is the stationary distribution under $\pi$ for continuous tasks, and for the episodic case it is defined as

$$\eta(s) = h(s) + \sum_{s'} \eta(s') \sum_a \pi(a|s') p(s'|s', a),$$

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} ,$$

where $h(s)$ is a probability that an episode starts in state $s$. 
The functional approximation (i.e., the weight vector $\mathbf{w}$) is usually optimized using gradient methods, for example as

$$
\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{1}{2} \alpha \nabla_{\mathbf{w}_t} [v_{\pi}(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2
$$

$$
\leftarrow \mathbf{w}_t + \alpha [v_{\pi}(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla_{\mathbf{w}_t} \hat{v}(S_t, \mathbf{w}_t).
$$

As usual, the $v_{\pi}(S_t)$ is estimated by a suitable sample. For example in Monte Carlo methods, we use episodic return $G_t$, and in temporal difference methods, we employ bootstrapping and use $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$. 
**Monte Carlo Gradient Policy Evaluation**

---

**Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$**

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v}: S \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $w \in \mathbb{R}^d$ arbitrarily (e.g., $w = 0$)

Loop forever (for each episode):
- Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using $\pi$
- Loop for each step of episode, $t = 0, 1, \ldots, T - 1$:
  $$ w \leftarrow w + \alpha \left[ G_t - \hat{v}(S_t, w) \right] \nabla \hat{v}(S_t, w) $$

*Algorithm 9.3 of "Reinforcement Learning: An Introduction, Second Edition".*
Linear Methods

A simple special case of function approximation are linear methods, where

\[
\hat{v}(\mathbf{x}(s), \mathbf{w}) \overset{\text{def}}{=} \mathbf{x}(s)^T \mathbf{w} = \sum x(s)_i w_i.
\]

The \(\mathbf{x}(s)\) is a representation of state \(s\), which is a vector of the same size as \(\mathbf{w}\). It is sometimes called a feature vector.

The SGD update rule then becomes

\[
\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha [v_\pi(S_t) - \hat{v}(\mathbf{x}(S_t), \mathbf{w}_t)] \mathbf{x}(S_t).
\]
State Aggregation

Simple way of generating a feature vector is *state aggregation*, where several neighboring states are grouped together.

For example, consider a 1000-state random walk, where transitions go uniformly randomly to any of 100 neighboring states on the left or on the right. Using state aggregation, we can partition the 1000 states into 10 groups of 100 states. Monte Carlo policy evaluation then computes the following:

![Diagram](Figure 9.1 of “Reinforcement Learning: An Introduction, Second Edition”.)
Many methods developed in the past:

- polynomials
- Fourier basis
- tile coding
- radial basis functions

But of course, nowadays we use deep neural networks which construct a suitable feature vector automatically as a latent variable (the last hidden layer).
If $t$ overlapping tiles are used, the learning rate is usually normalized as $\alpha/t$. 

*Figure 9.9 of “Reinforcement Learning: An Introduction, Second Edition”.*
Tile Coding

For example, on the 1000-state random walk example, the performance of tile coding surpasses state aggregation:

\[ \sqrt{VE} \]

averaged over 30 runs

Figure 9.10 of "Reinforcement Learning: An Introduction, Second Edition".
Asymmetrical Tile Coding

In higher dimensions, the tiles should have asymmetrical offsets, with a sequence of \((1, 3, 5, \ldots, 2d - 1)\) being a good choice.

Figure 9.11 of “Reinforcement Learning: An Introduction, Second Edition”.

In TD methods, we again use bootstrapping to estimate \( v_\pi(S_t) \) as \( R_{t+1} + \gamma \hat{v}(S_{t+1}, w) \).

<table>
<thead>
<tr>
<th>Semi-gradient TD(0) for estimating ( \hat{v} \approx v_\pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> the policy ( \pi ) to be evaluated</td>
</tr>
<tr>
<td><strong>Input:</strong> a differentiable function ( \hat{v} : S^+ \times \mathbb{R}^d \rightarrow \mathbb{R} ) such that ( \hat{v}(\text{terminal}, \cdot) = 0 )</td>
</tr>
<tr>
<td><strong>Algorithm parameter:</strong> step size ( \alpha &gt; 0 )</td>
</tr>
<tr>
<td><strong>Initialize value-function weights</strong> ( w \in \mathbb{R}^d ) arbitrarily (e.g., ( w = 0 ))</td>
</tr>
</tbody>
</table>

Loop for each episode:
- Initialize \( S \)
  
  Loop for each step of episode:
  - Choose \( A \sim \pi(\cdot | S) \)
  - Take action \( A \), observe \( R, S' \)
  - \( w \leftarrow w + \alpha [R + \gamma \hat{v}(S', w) - \hat{v}(S, w)] \nabla \hat{v}(S, w) \)
  - \( S \leftarrow S' \)
  - until \( S \) is terminal

Note that such algorithm is called *semi-gradient*, because it does not backpropagate through \( \hat{v}(S', w) \).
An important fact is that linear semi-gradient TD methods do not converge to $\overline{VE}$. Instead, they converge to a different TD fixed point $w_{TD}$.

It can be proven that

$$\overline{VE}(w_{TD}) \leq \frac{1}{1 - \gamma} \min_w \overline{VE}(w).$$

However, when $\gamma$ is close to one, the multiplication factor in the above bound is quite large.
As before, we can utilize $n$-step TD methods.

### $n$-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

- **Input:** the policy $\pi$ to be evaluated
- **Input:** a differentiable function $\hat{v}: S^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$
- **Algorithm parameters:** step size $\alpha > 0$, a positive integer $n$
- **Initialize value-function weights** $w$ arbitrarily (e.g., $w = 0$)
- All store and access operations ($S_t$ and $R_t$) can take their index $\text{mod } n + 1$

Loop for each episode:

1. Initialize and store $S_0 \neq \text{terminal}$
2. $T \leftarrow \infty$
3. Loop for $t = 0, 1, 2, \ldots$:
   - If $t < T$, then:
     - Take an action according to $\pi(\cdot | S_t)$
     - Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
     - If $S_{t+1}$ is terminal, then $T \leftarrow t + 1$
     - $\tau \leftarrow t - n + 1$ ($\tau$ is the time whose state's estimate is being updated)
   - If $\tau \geq 0$:
     - $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
     - If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, w)$ ($G_{\tau:T+n}$)
     - $w \leftarrow w + \alpha [G - \hat{v}(S_{\tau}, w)] \nabla \hat{v}(S_{\tau}, w)$
4. Until $\tau = T - 1$

*Algorithm 9.5 of "Reinforcement Learning: An Introduction, Second Edition".*
Temporal Difference Semi-Gradient Policy Evaluation

Figure 9.2 of “Reinforcement Learning: An Introduction, Second Edition”.

Average RMS error over 1000 states and first 10 episodes
Until now, we talked only about policy evaluation. Naturally, we can extend it to a full Sarsa algorithm:

### 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:
  - $w \leftarrow w + \alpha [R - \hat{q}(S, A, w)] \nabla \hat{q}(S, A, w)$
  - Go to next episode
- Choose $A'$ as a function of $\hat{q}(S', \cdot, w)$ (e.g., $\varepsilon$-greedy)
- $w \leftarrow w + \alpha [R + \gamma \hat{q}(S', A', w) - \hat{q}(S, A, w)] \nabla \hat{q}(S, A, w)$
- $S \leftarrow S'$
- $A \leftarrow A'$
Additionally, we can incorporate \( n \)-step returns:

<table>
<thead>
<tr>
<th>Episodic semi-gradient ( n )-step Sarsa for estimating ( \hat{q} \approx q_* ) or ( q_\pi )</th>
</tr>
</thead>
</table>
| **Input**: a differentiable action-value function parameterization \( \hat{q} : S \times A \times \mathbb{R}^d \to \mathbb{R} \)  
| **Input**: a policy \( \pi \) (if estimating \( q_\pi \))  
| **Algorithm parameters**: step size \( \alpha > 0 \), small \( \varepsilon > 0 \), a positive integer \( n \)  
| **Initialize** value-function weights \( w \in \mathbb{R}^d \) arbitrarily (e.g., \( w = 0 \))  
| All store and access operations \( (S_t, A_t, \text{and } R_t) \) can take their index mod \( n + 1 \)  
| **Loop for each episode**:  
| Initialize and store \( S_0 \neq \text{terminal} \)  
| Select and store an action \( A_0 \sim \pi(\cdot|S_0) \) or \( \varepsilon \)-greedy wrt \( \hat{q}(S_0, \cdot, w) \)  
| \( T \leftarrow \infty \)  
| **Loop for** \( t = 0, 1, 2, \ldots : \)  
| If \( t < T \), then:  
| Take action \( A_t \)  
| Observe and store the next reward as \( R_{t+1} \) and the next state as \( S_{t+1} \)  
| If \( S_{t+1} \) is terminal, then:  
| \( T \leftarrow t + 1 \)  
| else:  
| Select and store \( A_{t+1} \sim \pi(\cdot|S_{t+1}) \) or \( \varepsilon \)-greedy wrt \( \hat{q}(S_{t+1}, \cdot, w) \)  
| \( \tau \leftarrow t - n + 1 \) (\( \tau \) is the time whose estimate is being updated)  
| If \( \tau \geq 0 \):  
| \( G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i \)  
| If \( \tau + n < T \), then \( G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, w) \)  
| \( w \leftarrow w + \alpha [G - \hat{q}(S_\tau, A_\tau, w)] \nabla \hat{q}(S_\tau, A_\tau, w) \)  
| Until \( \tau = T - 1 \)  

The performances are for semi-gradient Sarsa(\(\lambda\)) algorithm (which we did not talked about yet) with tile coding of 8 overlapping tiles covering position and velocity, with offsets of \((1, 3)\).
Mountain Car Example

Figure 10.3 of “Reinforcement Learning: An Introduction, Second Edition”.

Figure 10.4 of “Reinforcement Learning: An Introduction, Second Edition”.
Consider a deterministic transition between two states whose values are computed using the same weight:

\[ w \rightarrow 2w \]

*Figure from Section 11.2 of "Reinforcement Learning: An Introduction, Second Edition".*

- If initially \( w = 10 \), TD error will be also 10 (or nearly 10 if \( \gamma < 1 \)).
- If for example \( \alpha = 0.1 \), \( w \) will be increased to 1 (by 10%).
- This process can continue indefinitely.

However, the problem arises only in off-policy setting, where we do not decrease value of the second state from further observation.
The previous idea can be realized for instance by the following *Baird's counterexample*:

\[
\begin{align*}
2w_1 + w_8 & \\
2w_2 + w_8 & \\
2w_3 + w_8 & \\
2w_4 + w_8 & \\
2w_5 + w_8 & \\
2w_6 + w_8 & \\
\end{align*}
\]

- \( \pi(\text{solid}|\cdot) = 1 \)
- \( b(\text{dashed}|\cdot) = 6/7 \)
- \( b(\text{solid}|\cdot) = 1/7 \)
- \( \gamma = 0.99 \)

The rewards are zero everywhere, so the value function is also zero everywhere.

*Figure 11.1 of "Reinforcement Learning: An Introduction, Second Edition".*
Off-policy Divergence With Function Approximation

However, for off-policy semi-gradient Sarsa, or even for off-policy dynamic-programming update, where we compute expectation over all following states and actions, the weights diverge to $\infty$.

$$w_{t+1} \leftarrow w_t + \frac{\alpha}{|S|} \sum_s \left( \mathbb{E}_\pi \left[ R_{t+1} + \gamma \hat{v}(S_{t+1}, w_k) | S_t = s \right] - \hat{v}(s, w_k) \right) \nabla \hat{v}(s, w_k)$$
Volodymyr Mnih et al.: *Playing Atari with Deep Reinforcement Learning* (Dec 2013 on arXiv). In 2015 accepted in Nature, as *Human-level control through deep reinforcement learning*. Off-policy Q-learning algorithm with a convolutional neural network function approximation of action-value function. Training can be extremely brittle (and can even diverge as shown earlier).
Figure 1 of the paper “Human-level control through deep reinforcement learning” by Volodymyr Mnih et al.
Figure 3 of the paper “Human-level control through deep reinforcement learning” by Volodymyr Mnih et al.
Deep Q Network

Extended Data Figure 2a of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.
Extended Data Figure 2b of the paper “Human-level control through deep reinforcement learning” by Volodymyr Mnih et al.
Deep Q Networks

- Preprocessing: 210 × 160 128-color images are converted to grayscale and then resized to 84 × 84.
- Frame skipping technique is used, i.e., only every 4th frame (out of 60 per second) is considered, and the selected action is repeated on the other frames.
- Input to the network are last 4 frames (considering only the frames kept by frame skipping), i.e., an image with 4 channels.
- The network is fairly standard, performing
  - 32 filters of size 8 × 8 with stride 4 and ReLU,
  - 64 filters of size 4 × 4 with stride 2 and ReLU,
  - 64 filters of size 3 × 3 with stride 1 and ReLU,
  - fully connected layer with 512 units and ReLU,
  - output layer with 18 output units (one for each action)
Deep Q Networks

- Network is trained with RMSProp to minimize the following loss:

\[
\mathcal{L} = \mathbb{E}_{(s,a,r,s') \sim data} \left[ (r + [s' \text{ not terminal}] \cdot \gamma \max_{a'} Q(s', a'; \tilde{\theta}) - Q(s, a; \theta))^2 \right].
\]

- An $\varepsilon$-greedy behavior policy is utilized.

Important improvements:

- experience replay: the generated episodes are stored in a buffer as $(s, a, r, s')$ quadruples, and for training a transition is sampled uniformly;
- separate target network $\tilde{\theta}$: to prevent instabilities, a separate target network is used to estimate state-value function. The weights are not trained, but copied from the trained network once in a while;
- reward clipping: because rewards have wildly different scale in different games, all positive rewards are replaced by $+1$ and negative by $-1$
  - furthermore, $(r + [s' \text{ not terminal}] \cdot \gamma \max_{a'} Q(s', a'; \tilde{\theta}) - Q(s, a; \theta))$ is also clipped to $[-1, 1]$ (i.e., a smooth $L_1$ loss or Huber loss).
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1, T$ do
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \left\{ \begin{array}{ll} r_j & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{array} \right.$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For

Algorithm 1 of the paper "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.
## Deep Q Networks Hyperparameters

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>minibatch size</td>
<td>32</td>
</tr>
<tr>
<td>replay buffer size</td>
<td>1M</td>
</tr>
<tr>
<td>target network update frequency</td>
<td>10k</td>
</tr>
<tr>
<td>discount factor</td>
<td>0.99</td>
</tr>
<tr>
<td>training frames</td>
<td>50M</td>
</tr>
<tr>
<td>RMSProp learning rate and momentum</td>
<td>0.00025, 0.95</td>
</tr>
<tr>
<td>initial $\epsilon$, final $\epsilon$ (linear decay) and frame of final $\epsilon$</td>
<td>1.0, 0.1, 1M</td>
</tr>
<tr>
<td>replay start size</td>
<td>50k</td>
</tr>
<tr>
<td>no-op max</td>
<td>30</td>
</tr>
</tbody>
</table>