Algorithms for Reinforcement Learning

Stanford, California

Contents

Acknowledgments

This work is taken from the lecture notes for the course *Reinforcement Learning* at Stanford University, CS 234 (`cs234.stanford.edu`). The contributors to the content of this work are Emma Brunskil and Luke Johnston—this collection is simply a typesetting of existing lecture notes with minor modifications and additions of working Julia implementations. We would like to thank the original authors for their contribution. In addition, we wish to thank Mykel Kochenderfer and Tim Wheeler for their contribution to the Tufte-Algorithms LaTeX template, based off of *Algorithms for Optimization*.[1]

[1] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

Robert J. Moss
Stanford, Calif.
February 18, 2021

Ancillary material is available on the template's webpage:
`https://github.com/sisl/textbook_template`

7 Deep Q-Learning and CNNs

From CS234 Winter 2021, Tian Tan and Emma Brunskill, Stanford University.

7.1 Value-Based Deep Reinforcement Learning

In this section, we introduce three popular value-based deep reinforcement learning (RL) algorithms: *Deep Q-Network* (DQN),[1] *Double DQN*,[2] and *Dueling DQN*.[3] All the three neural architectures are able to learn successful policies directly from *high-dimensional inputs*, e.g. preprocessed pixels from video games, by using *end-to-end* reinforcement learning, and they all achieved a level of performance that is comparable to a professional human games tester across a set of 49 names on Atari 2600.[4]

Convolutional Neural Networks (CNNS)[5] are used in these architectures for feature extraction from pixel inputs. Understanding the mechanisms behind feature extraction via CNNs can help better understand how DQN works. The Stanford CS231N course website contains wonderful examples and introduction to CNNs. Here, we direct the reader to the following link for more details on CNNs.[6] The remaining of this section will focus on generalization in RL and value-based deep RL algorithms.

7.1.1 Action-value function approximation

In the previous lecture, we use parameterized function approximators to represent the action-value function (i.e. Q-function). If we denote the set of parameters as $\mathbf{w}$, the Q-function in this *approximation setting* is represented as $\hat{q}(s, a, \mathbf{w})$.

Let's first assume we have access to an oracle $q(s, a)$, the approximate Q-function can be learned by minimizing the mean-squared error between the true action-value function $q(s, a)$ and its approximated estimates:

$$J(\mathbf{w}) = \mathbb{E}\left[\left(q(s, a) - \hat{q}(s, a, \mathbf{w})\right)^2\right] \tag{7.1}$$

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, vol. 30, 2016.

[3] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2016.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 25, pp. 1097–1105, 2012.

[6] `http://cs231n.github.io/convolutional-networks`

We can use *stochastic gradient descent* (SGD) to find a local minimum of J by sampling gradients w.r.t. parameters $\mathbf{w}$ and updating $\mathbf{w}$ as follows:

$$\Delta(\mathbf{w}) = -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha\mathbb{E}\Big[\big(q(s,a) - \hat{q}(s,a,\mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w})\Big] \tag{7.2}$$

where α is the learning rate. In general, the true action-value function $q(s,a)$ is unknown, so we substitute the $q(s,a)$ in equation (7.2) with an approximate *learning target*.

In Monte Carlo methods, we use an unbiased return G_t as the substitute target for episodic MDPs:

$$\Delta(\mathbf{w}) = \alpha(G_t - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w}) \tag{7.3}$$

For SARSA, we instead use *bootstrapping* and present a TD (biased) target $r + \gamma\hat{q}(s',a',\mathbf{w})$, which leverages the current function approximation value,

$$\Delta(\mathbf{w}) = \alpha(r + \gamma\hat{q}(s',a',\mathbf{w}) - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w}) \tag{7.4}$$

where a' is the action taken at the next state s' and γ is a discount factor. For Q-learning, we use a TD target $r + \gamma\max_{a'}\hat{q}(s',a',\mathbf{w})$ and update $\mathbf{w}$ as follows:

$$\Delta(w) = \alpha(r + \gamma\max_{a'}\hat{q}(s',a',\mathbf{w}) - \hat{q}(s,a,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w}) \tag{7.5}$$

In subsequent sections, we will introduce how to approximate $\hat{q}(s,a,\mathbf{w})$ by using a deep neural network and learn neural network parameters $\mathbf{w}$ via end-to-end training.

7.1.2 *Generalization: Deep Q-network (DQN)*

The performance of linear function approximators highly depends on the quality of features. In general, handcrafting an appropriate set of features can be diffcult and time-consuming. To scale up to making decisions in really *large domains* (e.g. huge state space) and enable automatic feature extraction, deep neural networks (DNNs) are used as function approximators.

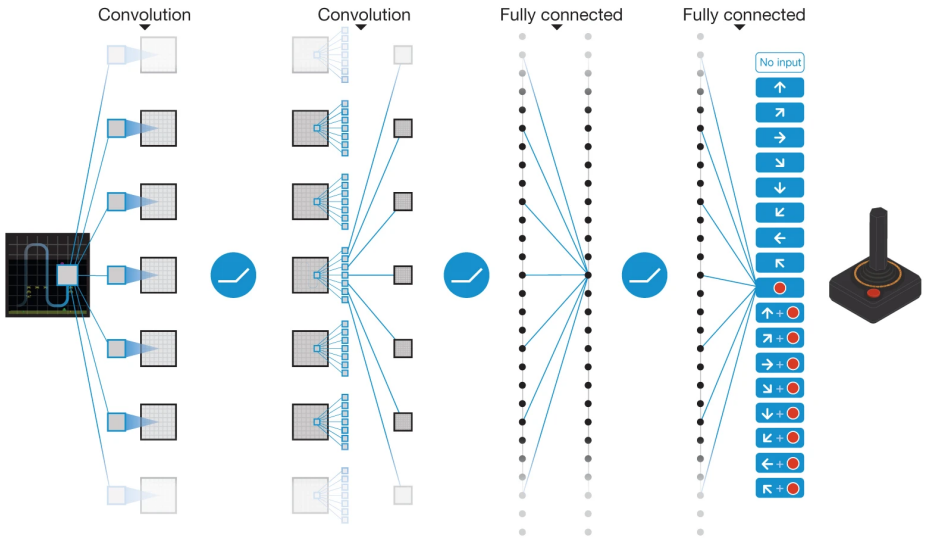

Figure 7.1. Illustration of the deep Q-network: the input to the network consists of an 84×84×4 preprocessed image, followed by three convolutional layers and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (ReLU).

DQN architecture. An illustration of the DQN architecture is shown in figure 7.1. The network takes preprocessed pixel image from Atari game environment (see section 7.1.3 for preprocessing) as inputs, and outputs a vector containing Q-values for each valid action. The preprocessed pixel input is a summary of the game state s, and a single output unit represents the $\hat{q}$ function for a single action a. Collectively, the $\hat{q}$ function can be denoted as $\hat{q}(s, \mathbf{w}) \in \mathbb{R}^{|A|}$. For simplicity, we will still use notation $\hat{q}(s, a, \mathbf{w})$ to represent the estimated action-value for a (s, a) pair in the following paragraphs.

Details of the architecture. The input consists of an $84 \times 84 \times 4$ image. The first convolutional layer has 32 filters of size 8×8 with stride 4 and convolves with the input image, followed by a rectifier nonlinearity (ReLU).[7] The second hidden layer convolves 64 filters of 4×4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that has 64 filters of 3×3 with stride 1, followed by a ReLU. The final hidden layer is a fully-connected layer with 512 ReLUs. The output layer is a fully-connected linear layer.

[7] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Icml*, 2010.

7.1.3 *Preprocessing raw pixels*

The raw Atari 2600 frames are of size $(210 \times 160 \times 3)$, where the last dimension is corresponding to the RGB channels. The preprocessing step adopted in Mnih et al. aims at reducing the input dimensionality and dealing with some artifacts of the game emulator. We summarize the preprocessing as follows:

- **Single frame encoding**: to encode a single frame, the maximum value for each pixel color value over the frame being encoded and the previous frame is returned. In other words, we return a pixel-wise max-pooling of the 2 consecutive raw pixel frames.
- **Dimensionality reduction**: extract the Y channel, also known as luminance, from the encoded RGB frame and rescale it to $(84 \times 84 \times 1)$.

The above preprocessing is applied to the 4 most recent raw RGB frames and the encoded frames are stacked together to produce the input (of shape $(84 \times 84 \times 4)$) to the Q-network. Stacking together the recent frames as game state is also a way to transform the game environment into a (almost) Markovian world.

7.1.4 *Training algorithm for DQN*

The use of large deep neural network function approximators for learning action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, and learning and training tend to be very unstable. In order to use large nonlinear function approximators and scale online Q-learning, DQN introduced two major changes: the use of *experience replay*, and a separate *target network*. The full algorithm is presented in algorithm 7.1. Essentially, the Q-network is learned by minimizing the following mean squared error

$$J(\mathbf{w}) = \mathbb{E}(s_t, a_t, r_t, s_{t+1}) \left[\left(y_t^{\text{DQN}} - \hat{q}(s_t, a_t, \mathbf{w}) \right)^2 \right], \tag{7.6}$$

where y_t^{DQN} is the one-step-ahead learning target

$$y_t^{\text{DQN}} = r_t + \gamma \max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}^-), \tag{7.7}$$

where $\mathbf{w}^-$ represents the parameters of the target network,[8] and the parameters $\mathbf{w}$ of the online network are updated by sampling gradients from minibatches of past transition tuples (s_t, a_t, r_t, s_{t+1}).

[8] Note, although the learning target is computed from the target network with $\mathbf{w}^-$, the targets y_t^{DQN} are considered to be fixed when making updates to $\mathbf{w}$.

Experience replay. The agent's experiences (or transitions) at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a fixed-sized dataset (or *replay buffer*) $\mathcal{D}_t = \{e_1, \ldots, e_t\}$. The replay buffer is used to store the most recent $k = 1$ million experiences (see table 7.1 for an illustration of replay buffer). The Q-network is updated by SGD with sampled gradients from minibatch data. Each transition sample in the minibatch is sampled uniformly at random from the pool of stored experiences, $(s, a, r, s') \sim \mathcal{U}(\mathcal{D})$. This approach has the following advantages over standard online Q-learning:

s_1, a_1, r_1, s_2
s_2, a_2, r_2, s_3
...
s_t, a_t, r_t, s_{t+1}

Table 7.1. Replay buffer: the transition (s, a, r, s') is uniformly sampled from the replay buffer for updating Q-network.

- **Greater data efficiency**: each step of experience can be potentially used for many updates, which improves data efficiency.
- **Remove sample correlations**: randomizing the transition experiences breaks the correlations between consecutive samples and therefore reduces the variance of updates and stabilizes the learning.
- **Avoiding oscillations or divergence**: the behavior distribution is averaged over many of its previous states and transitions, smoothing out learning and avoiding oscillations or divergence in the parameters. (Note that when using experience replay, it is required to use an off-policy method, e.g. Q-learning, because the current parameters are different from those used to generate the samples).

Limitation of experience replay. The replay buffer does not differentiate important transitions or informative transitions and it always overwrites with the recent transitions due to fixed buffer size. Similarly, the uniform sampling from the buffer gives equal importance to all stored experiences. A more sophisticated replay strategy, **prioritized replay**, has been proposed by Schaul et al., which replays important transitions more frequently, and therefore the agent learns more efficiently.

Target network. To further improve the stability of learning and deal with *non-stationary learning targets*, a separate target network is used for generating the targets y_j in the Q-learning update. More specifically, every C updates/steps the target network $\hat{q}(s, a, \mathbf{w}^-)$ is updated by copying the parameters' values $(\mathbf{w}^- = \mathbf{w})$ from the online network $\hat{q}(s, a, \mathbf{w})$, and the target network remains unchanged

and generates targets y_j for the following C updates. This modification makes the algorithm more stable compared to standard online Q-learning, and $C = 10000$ was used in the original DQN.

Initialize replay buffer $\mathcal{D}$ with a fixed capacity
Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
Initialize target action-value function $\hat{q}_{\text{target}}$ with random weights $\mathbf{w}^-$
for episode $m = 1, \ldots, M$ **do**
  Observe initial frame x_1 and preprocess frame to get state s_1
  for time step $t = 1, \ldots, T$ **do**
    Select action $a_t = \begin{cases} \text{random action} & \text{w/ probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}) & \text{otherwise} \end{cases}$
    Execute action a_t in simulator/emulator and observe reward r_t and image x_{t+1}
    Preprocess s_t, x_{t+1} to get s_{t+1} and store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample uniformly a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{if episode ends at step } j+1 \\ y_j = r_j + \gamma \max_{a'} \hat{q}_{\text{target}}(s_{j+1}, a', \mathbf{w}^-) & \text{otherwise} \end{cases}$
    Perform a SGD step on $J(\mathbf{w}) = \frac{1}{N} \sum_{j=1}^{N} \left(y_j = \hat{q}(s_j, a_j, \mathbf{w})\right)^2$ w.r.t. parameters $\mathbf{w}$
    Every C steps, reset $\mathbf{w}^- = \mathbf{w}$

Algorithm 7.1. Deep Q-learning

7.1.5 *Training details*

In the original DQN paper,[9] a different network (or agent) was trained on each game with the same architecture, learning algorithm and hyperparameters. The authors clipped all positive rewards from the game environment at $+1$ and all negative rewards at -1, which makes it possible to use the same learning rate across all different games. For games where there is a life counter (e.g. *Breakout*), the emulator also returns the number of lives left in the game, which was then used to mark the end of an episode during training by explicitly setting future rewards to zeros. They also used a simple frame-skipping technique (or *action repeat*): the agent selects actions on every 4-th frame instead of every frame, and its last action is repeated on skipped frames. This reduces the frequency of

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

decisions without impacting the performance too much and enables the agent to play roughly 4 times more games during training.

RMSProp[10] was used by Mnih et al. for training DQN with minibatches of size 32. During training, they applied ϵ-greedy policy with ϵ linearly annealed from 1.0 to 0.1 over the first million steps, and fixed at 0.1 afterwards. The replay buffer was used to store the most recent 1 million transitions. For evaluation at test time, they used ϵ-greedy policy with $\epsilon = 0.05$.

[10] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

7.2 Reducing bias: Double deep Q-network (DDQN)

The max operator in DQN, uses the same network values both to select and to evaluate an action. This setting makes it more likely to select overestimated values and resulting in overoptimistic target value estimates. Van Hasselt, Guez, and Silver also showed that the DQN algorithm suffers from substantial overestimations in some games in the Atari 2600. To prevent overestimation and reduce bias, we can *decouple* the *action selection* from *action evaluation*.

Recall in Double Q-learning, two action-value functions are maintained and learned by randomly assigning transitions to update one of the two functions, resulting in two different sets of function parameters, denoted here as $\mathbf{w}$ and $\mathbf{w}'$. For computing targets, one function is used to select the greedy action and the other to evaluate its value:

$$y_t^{\text{DoubleQ}} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}') \tag{7.8}$$

Note that the action selection (arg max) is due to the function parameters $\mathbf{w}$, while the action value is evaluated by the other set of parameters $\mathbf{w}'$.

The idea of reducing overestimations by decoupling action selection and action evaluation in computing targets can also be extended to deep Q-learning. The target network in DQN architecture provides a natural candidate for the second action-value function, without introducing additional networks. Similarly, the greedy action is generated according to the online network with parameters $\mathbf{w}$, but its value is estimated by the target network with parameters $\mathbf{w}^-$. The resulting algorithm is referred as Double DQN,[11] which just replaces the target computation of y_j in algorithm 7.1 by the following update target:

$$y_t^{\text{DoubleDQN}} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}^-) \tag{7.9}$$

[11] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, vol. 30, 2016.

The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network **w**. The rest of the DQN algorithm remains intact.

7.3 *Decoupling value and advantage: Dueling DQN*

This section introduces the dueling DQN algorithm.[12]

[12] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, ''Dueling Network Architectures for Deep Reinforcement Learning,'' in *International Conference on Machine Learning (ICML)*, 2016.

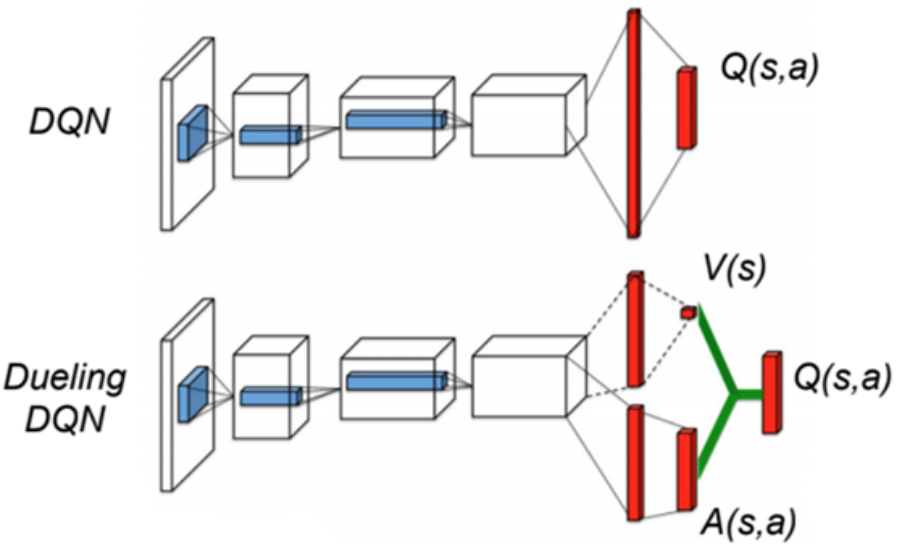

Figure 7.2. Single stream deep Q-network (top) and the dueling deep Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value $V(s)$ and the advantages $A(s,a)$ for each action; the green output module implements **??** to combine the two streams. Both networks output Q-values for each action.

7.3.1 *The dueling network architecture*

Before we delve into dueling architecture, let's first introduce an important quantity, the *advantage* function, which relates the value and Qofunctions (assume following a policy π):

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{7.10}$$

Recall $V\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^{\pi}(s,a)]$, thus we have $E_{a\sim\pi(s)}[A^{\pi}(s,a)] = 0$. Intuitively, the advantage function subtracts the value of the state from the Q function to get a relative measure of the importance of each action.

Like in DQN, the dueling network is also a DNN function approximator for learning the Q-function. Differently, it approximates the Q-function by *decoupling* the value function and the advantage function. Figure 7.2 illustrates the dueling network architecture and the DQN for comparison.

The lower layers of the dueling network are convolutional as in the DQN. However, instead of using a single stream of fully connected layers for Q-value estimates, the dueling network uses two streams of fully connected layers. One stream is used to provide value function estimate given a state, while the other stream is for estimating advantage function for each valid action. Finally, the two streams are combined in a way to produce and approximate the Q-function. As in DQN, the output of the network is a vector of Q-values, one for each action.

Note that since the inputs and the final outputs (combined two streams) of the dueling network are the same as that of the original DQN, the training algorithm (algorithm 7.1) introduced above for DQN and for Double DQN can also be applied here to train the dueling architecture. The separated two-stream design is based on the following observations or intuitions from the authors:

- For many states, it is unnecessary to estimate the value of each possible action choice. In some states, the action selection can be of great importance, but in many other states the choice of action has no repercussion on what happens next. On the other hand, the state value estimation is of significant importance for every state for a bootstrapping based algorithm like Q-learning.
- Features required to determine the value function may be different than those used to accurately estimate action benefits.

Combing the two streams of fully connected layers for Q-value estimate is not a trivial task. This aggregating module (shown in green lines in figure 7.2), in fact, requires very thoughtful design, which we will see in the next subsection.

7.3.2 *Q-value estimation*

From the definition of advantage function (7.10), we have $Q^\pi(s,a) = A^\pi(s,a) + V^\pi(s)$, and $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Furthermore, for a deterministic policy (commonly used in value-based deep RL), $a^* = \arg\max_{a'\in A} Q(s,a')$, it follows that $Q(s,a^*) = V(s)$ and hence $A(s,a^*) = 0$. The greedily selected action has zero advantage in this case.

Now consider the dueling network architecture in figure 7.2 for function approximation. Let's denote the scalar output value function from one stream of the fully-connected layers as $\hat{v}(s, \mathbf{w}, \mathbf{w_v})$, and denote the vector output advantage function from the other stream as $A(s, a, \mathbf{w}, \mathbf{w_A})$. We use $\mathbf{w}$ here to denote the shared parameters in the convolutional layers, and use $\mathbf{w_v}$ and $\mathbf{w_A}$ to represent parameters in the two different streams of fully-connected layers. Then, probably the most simple way to design the aggregating module is by following the definition:

$$\hat{q}(s, a, \mathbf{w}, \mathbf{w_A}, \mathbf{w_v}) = \hat{v}(s, \mathbf{w}, \mathbf{w_v}) + A(s, a, \mathbf{w}, \mathbf{w_A}) \tag{7.11}$$

The main problem with this simple design is that equation (7.11) is unidentifiable. Given $\hat{q}$, we cannot recover $\hat{v}$ and A uniquely, e.g. adding a constant to $\hat{v}$ and subtracting the same constant from A gives the same Q-value estimates. The unidentifiable issue is mirrored by poor performance in practice.

To make Q-function identifiable, recall in the deterministic policy case discussed above, we can force the advantage function to have zero estimate at the chosen action. Then, we have

$$\hat{q}(s, a, \mathbf{w}, \mathbf{w_A}, \mathbf{w_v}) = \hat{v}(s, \mathbf{w}, \mathbf{w_v}) + \left(A(s, a, \mathbf{w}, \mathbf{w_A}) - \max_{a' \in A} A(s, a', \mathbf{w}, \mathbf{w_A}) \right) \tag{7.12}$$

For a deterministic policy,

$$a^* = \arg\max_{a' \in A} \hat{q}(s, a', \mathbf{w}, \mathbf{w_A}, \mathbf{w_v}) = \arg\max_{a' \in A} A(s, a', \mathbf{w}, \mathbf{w_A}), \tag{7.13}$$

equation (7.12) gives $\hat{q}(s, a^*, \mathbf{w}, \mathbf{w_A}, \mathbf{w_v}) = \hat{v}(s, \mathbf{w}, \mathbf{w_v})$. Thus, the stream $\hat{v}$ provides an estimate of the value function, and the other stream A generates advantage estimates.

Wang et al. also proposed an alternative aggregating module that replaces the max with a mean operator:

$$\hat{q}(s, a, \mathbf{w}, \mathbf{w_A}, \mathbf{w_v}) = \hat{v}(s, \mathbf{w}, \mathbf{w_v}) + \left(A(s, a, \mathbf{w}, \mathbf{w_A}) - \frac{1}{|A|} \sum_{a'} A(s, a', \mathbf{w}, \mathbf{w_A}) \right) \tag{7.14}$$

Although this design in some sense loses the original semantics of $\hat{v}$ and A, the author argued that it improves the stability of learning: the advantages only need to change as fast as the mean, instead of having to compensate any change to the advantage of the optimal action. Therefore, the aggregating module in the dueling network[13] is implemented following equation (7.14). When acting, it suffices to evaluate the advantage stream to make decisions.

The advantage of the dueling network lies in its capability of approximating the value function efficiently. This advantage over single-stream Q networks grows when the number of actions is large, and the dueling network achieved state-of-the-art results on Atari games as of 2016.

[13] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2016.

8 Policy Gradient

From CS234 Winter 2021, Luke Johnston and Emma Brunskill, Stanford University.

8.1 Introduction to Policy Search

So far, in order to learn a policy, we have focused on **value-based** approaches where we find the optimal state value function or state-action value function with parameters θ,

$$V_\theta(s) \approx V^\pi(s) \tag{8.1}$$

$$Q_\theta(s,a) \approx Q^\pi(s,a) \tag{8.2}$$

and then use V_θ or Q_θ to extract a policy, e.g. with ϵ-greedy. However, we can also use a **policy-based** approach to directly parameterize the policy:

$$\pi_\theta(a \mid s) = \mathbb{P}[a \mid s; \theta] \tag{8.3}$$

In this setting, our goal is to directly find the policy with the highest value function V^π, rather than first finding the value-function of the optimal policy and then extracting the policy from it. Instead of the policy being a look-up table from states to actions, we will consider stochastic policies that are parameterized. Finding a good policy requires two parts:

1. **Good policy parameterization**: our function approximation and state/action representations must be expressive enough
2. **Effective search**: we must be able to find good parameters for our policy function approximation

Policy-based RL has a few advantages over value-based RL:

- Better convergence properties (see chapter 13.3 of Sutton and Barto)
- Effectiveness in high-dimensional or continuous action spaces, e.g. robotics. One method for continuous action spaces is covered in **??**.
- Ability to learn stochastic policies. See the following section.

The disadvantages of policy-based RL methods are:

- They typically converge to locally rather than globally optimal policies, since they rely on gradient descent.
- Evaluating a policy is typically data ineffcient and high variance.

8.2 Stochastic Polices

In this section, we will briefly go over two environments in which a stochastic policy is better than any deterministic policy.

Rock-paper-scissors. For a relatable example, in the popular zero-sum game of rock-paper-scissors, any policy that is not uniformly random

$$P(\texttt{rock} \mid s) = 1/3$$
$$P(\texttt{scissors} \mid s) = 1/3$$
$$P(\texttt{paper} \mid s) = 1/3$$

can be exploited.

Example 8.1. A deterministic rock-paper-scissors policy could be exploited.

8.2.1 Example: Aliased gridworld

In the gridworld environment in figure 8.1, suppose that the agent can move in the four cardinal directions, so its actions space is $\mathcal{A} = \{\texttt{N}, \texttt{S}, \texttt{E}, \texttt{W}\}$. However, suppose that it can only sense the walls around its current location. Specifically, it observes features of the following form for each direction:

$$\phi(s) = \begin{bmatrix} \mathbb{1}(\text{wall to } \texttt{N}) \\ \cdots \\ \mathbb{1}(\text{wall to } \texttt{W}) \end{bmatrix}$$

Note that its observations are not fully representative of the environment, as it cannot distinguish between the two gray squares. This also means that its domain is not Markov. Hence, a deterministic

policy must either learn to always go left in the gray squares, or always go right. Neither of these policies is optimal, since the agent can get stuck in one corner of the environment, seen in figure 8.2.

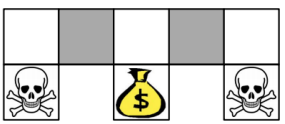

Figure 8.1. In this partially observable gridworld environment, the agent cannot distinguish between the gray states.

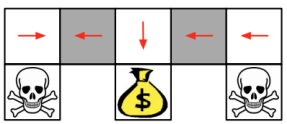

Figure 8.2. For this deterministic policy, the agent cannot ''escape'' from the upper-left two states.

However, a stochastic policy can learn to randomly select a direction in the gray states, guaranteeing that it will eventually reach the reward from any starting location. In general, stochastic policies can help overcome an adversarial or non-stationary domain and cases where the state-representation is not Markov.

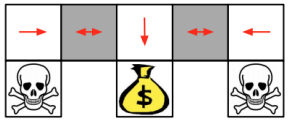

Figure 8.3. A stochastic policy which moves `E` or `W` with equal probability in the gray states will reach the goal in a few time steps with high probability.

8.3 Policy Optimization

In this section, we disuss methods to directly optimize the policy parameters.

8.3.1 Policy objective functions

Once we have defined a policy $\pi_\theta(a \mid s)$, we need to able to measure how it is performing in order to optimize it. In an episodic environment, a natural measurement is the **start value** of the policy, which is the expected value of the start state:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1] \tag{8.4}$$

In continuing environments we can use the **average value** of the policy, where $d^{\pi_\theta}(s)$ is the stationary distribution of π_θ:

$$J_{\text{avg-V}}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{8.5}$$

or alternatively we can use the **average reward** per time-step:

$$J_{\text{avg-R}}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s) R(s, a) \tag{8.6}$$

In these notes we discuss the episodic case, but all the results we derive can be easily extended to the non-episodic case. We will also focus on the case where the discount $\gamma = 1$, but again, the results are easily extended to general γ.

8.3.2 Optimization methods

With an objective function, we can treat our policy-based reinforcement learning problem as an optimization problem. In these notes, we focus on gradient descent, because recently that has been the most common optimization method for policy-based RL methods. However, it is worth considering some **gradient-free optimization methods**, including the following:

- Hill climbing
- Simplex / amoeba / Nelder Mead
- Genetic algorithms
- Cross-entropy method (CEM)
- Covariance Matrix Adaptation (CMA)
- Evolution strategies

These methods have the advantage over gradient-based methods in that they do not have to compute a gradient of the objective function. This allows the policy parameterization to be non-differentiable, and these methods are also often easy to parallelize. Gradient-free methods are often a useful baseline to try, and sometimes they can work embarrassingly well.[1] However, this methods are usually not very sample effcient because they ignore the temporal structure of the rewards—updates only take into account the total reward over the entire episode, and they do not break up the reward into different rewards for each state in the trajectory. (See section 8.6).

[1] https://openai.com/blog/evolution-strategies/

8.4 Policy Gradient

Let us define $V(\theta)$ to be the objective function we wish to maximize over θ. Policy gradient methods search for a local maximum in $V(\theta)$ by ascending the gradient of the policy w.r.t parameters θ

$$\Delta\theta = \alpha \nabla_\theta V(\theta) \tag{8.7}$$

where α is a step-size parameter and $\nabla_\theta V(\theta)$ is the policy gradient

$$\nabla_\theta V(\theta) = \begin{bmatrix} \frac{\partial V(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial V(\theta)}{\partial \theta_n} \end{bmatrix} \tag{8.8}$$

8.4.1 Computing the gradient

With this setup, all we have to do is compute the gradient of the objective function $V(\theta)$, and we can optimize it! The method of finite difference from calculus provides an approximation of the gradient:

$$\frac{\partial V(\theta)}{\partial \theta_k} \approx \frac{V(\theta + \epsilon u_k) - V(\theta)}{\epsilon} \tag{8.9}$$

where u_k is a unit vector with 1 in k-th component, 0 elsewhere. This method uses n evaluations to compute the policy gradient in n dimensions, so it is quite ineffcient, and it usually only provides a noisy approximation of the true policy gradient. However, it has the advantage that it works for non-differentiable policies. An example of a successful use of this method to train the AIBO robot gait.[2]

[2] N. Kohl and P. Stone, "Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, 2004.

Analytic gradients. Let us set the objective function $V(\theta)$ to be the expected rewards for an episode,

$$V(\theta) = \mathbb{E}_{(s_t, a_t) \sim \pi_\theta} \left[\sum_{t=0}^{T} R(s_t, a_t) \right] \tag{8.10}$$

$$= \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \right] \tag{8.11}$$

$$= \sum_{\tau} P(\tau; \theta) R(\tau) \tag{8.12}$$

where τ is a trajectory,

$$\tau = (s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T) \tag{8.13}$$

and $P(\tau; \theta)$ denotes the probability over trajectories when following policy π_θ, and $R(\tau)$ is the sum of rewards for a trajectory. Note that this objective function is the same as the start value $J_1(\theta)$ as mentioned in section 8.3.1 when the discount $\gamma = 1$.

If we can mathematically compute the policy gradient $\nabla_\theta \pi_\theta(a \mid s)$, then we can directly compute the gradient of this objective function with respect to θ:

$$\begin{aligned}
\nabla_\theta V(\theta) &= \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau) && (8.14)\\
&= \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau) && (8.15)\\
&= \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_\theta P(\tau;\theta) R(\tau) && (8.16)\\
&= \sum_\tau P(\tau;\theta) R(\tau) \underbrace{\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}}_{\substack{\text{likelihood}\\ \text{ratio}}} && (8.17)\\
&= \sum_\tau P(\tau;\theta) R(\tau) \nabla_\theta \log P(\tau;\theta) && (8.18)\\
&= \mathbb{E}_{\tau\sim\pi\theta}\left[R(\tau) \nabla_\theta \log P(\tau;\theta)\right] && (8.19)
\end{aligned}$$

The expression $\nabla_\theta P(\tau;\theta)/P(\tau;\theta)$ in equation (8.17) is known as the **likelihood ratio**. The tricks in equations (8.16) to (8.19) helps for two reasons. First, it helps us get the gradient into the form $\mathbb{E}_{\tau\sim\pi_\theta}[\ldots]$, which allows us to approximate the gradient by sampling trajectories $\tau^{(i)}$:

$$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \nabla_\theta \log P(\tau^{(i)};\theta) \tag{8.20}$$

Second, computing $\nabla_\theta \log P(\tau^{(i)};\theta)$ is easier than working with $P(\tau^{(i)};\theta)$ directly:

$$\begin{aligned}
\nabla_\theta \log P(\tau^{(i)};\theta) &= \nabla_\theta \log \left[\underbrace{\mu(s_0)}_{\substack{\text{initial}\\ \text{state}\\ \text{distribution}}} \prod_{t=0}^{T-1} \underbrace{\pi_\theta(a_t \mid s_t)}_{\text{policy}} \underbrace{P(s_{t+1} \mid s_t, a_t)}_{\text{dynamics model}} \right] && (8.21)\\
&= \nabla_\theta \left[\log \mu(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t \mid s_t) + \log P(s_{t+1} \mid s_t, a_t) \right] && (8.22)\\
&= \underbrace{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)}_{\text{no dynamics model required!}} && (8.23)
\end{aligned}$$

Working with $\log P(\tau^{(i)};\theta)$ instead of $P(\tau^{(i)};\theta)$ allows us to represent the gradient without reference to the initial state distribution, or even the environment dynamics model!

The expression $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ is known as the **score function**. Putting equation (8.20) and equation (8.23) together, we get

$$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) \tag{8.24}$$

which we can convert into a concrete algorithm for optimizing π_θ (**??**). But before that, we will mention the generalized version of this result and cover an optimization of the above derivation that takes advantage of decomposing $R(\tau^{(i)})$ into a sum of reward terms $r_t^{(i)}$ (section 8.6).

8.5 The Policy Gradient Theorem

Theorem 8.1. *For any differentiable policy $\pi_\theta(a \mid s)$ and for any of the policy objective functions $V(\theta) = J_1$, $J_{\text{avg-R}}$, or $\frac{1}{1-\gamma} J_{\text{avg-V}}$, the policy gradient is*

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta} \left[Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a \mid s) \right] \tag{8.25}$$

We will not go over the derivation of this more general theorem, but the same concepts discussed in this lecture apply to non-episodic (continuing) environments. In our discussion thus far, the total episode rewards $R(\tau)$ have been substituted in place of the Q values of this theorem, but in the following section we will use the temporal structure to get our result into a form that looks more like this theorem, where the future returns G_t (which are unbiased estimates of $Q(s_t, a_t)$) appear in place of $Q^{\pi_\theta}(s,a)$.

8.6 Temporal Structure

Equation (8.19) above can be written

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{8.26}$$

Notice that the rewards $R(\tau^{(i)})$ are treated as a single number which is a function of an entire trajectory $\tau^{(i)}$. We can break this down into the sum of all the rewards encountered in the trajectory,

$$R(\tau) = \sum_{t=0}^{T-1} R(s_t, a_t) \tag{8.27}$$

Using this knowledge, we can derive the gradient estimate for a single reward term r'_t in exactly the same way we derived equation (8.26):

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[r'_t] = \mathbb{E}_{\pi_\theta}\left[r'_t \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \tag{8.28}$$

Since $\sum_{t'=t}^{T-1} r_{t'}^{(i)}$ is the return $G_t^{(i)}$, we can sum this up over all time steps for a trajectory to get

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t'=0}^{T-1} r'_t \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \tag{8.29}$$

$$= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T-1} r'_t\right] \tag{8.30}$$

$$= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} G_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right] \tag{8.31}$$

Our final expression that we will use in the policy gradient algorithm in the next section is:

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \approx \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} G_t^{(i)} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) \tag{8.32}$$

Policy gradient example. Going from equation (8.29) to equation (8.30) may not be obvious, so let's go over a quick example. Say we have a trajectory that is three time steps long. Then equation (8.29) becomes

$$\begin{aligned}
&\nabla_\theta V(\theta) = \\
&\quad \mathbb{E}_{\pi_\theta}[r_0 \nabla_\theta \log \pi_\theta(a_0 \mid s_0) + \\
&\qquad r_1(\nabla_\theta \log \pi_\theta(a_0 \mid s_0) + \nabla_\theta \log \pi_\theta(a_1 \mid s_1)) + \\
&\qquad r_2(\nabla_\theta \log \pi_\theta(a_0 \mid s_0) + \nabla_\theta \log \pi_\theta(a_1 \mid s_1) + \nabla_\theta \log \pi_\theta(a_2 \mid s_2))]
\end{aligned}$$

Regrouping the terms, we get

$$\begin{aligned}\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}[&\nabla_\theta \log \pi_\theta(a_0 \mid s_0)(r_0 + r_1 + r_2) + \\ &\nabla_\theta \log \pi_\theta(a_1 \mid s_1)(r_1 + r_2) + \\ &\nabla_\theta \log \pi_\theta(a_2 \mid s_2)(r_2)]\end{aligned}$$

which equals equation (8.30) as expected. The main idea is that the policy's choice at a particular time step t only affects rewards received in later steps of the episode, and has no effect on rewards received in previous time steps. Our original expression in equation (8.26) did not take this into account.

8.7 REINFORCE: A Monte Carlo Policy Gradient Algorithm

We've done most of the work towards our first policy gradient algorithm in the sections above. The algorithm simply samples multiple trajectories following the policy π_θ while updating θ using the estimated gradient in equation (8.32).[3]

[3] REINFORCE leverages the likelihood ratio score function and temporal structure (using G_t), and thus reduces variance and improves our gradient estimate.

Algorithm 8.1. REINFOCE: Monte Carlo policy gradient algorithm.

function REINFORCE(α)
  Initialize policy parameters θ arbitrarily
  for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
  return θ

8.8 Differentiable Policy Classes

In this section, we introduce a few classes of policies which are differentiable.

8.8.1 *Discrete action space:* softmax *policy*

In discrete action spaces, the softmax function is commonly used to parameterize the policy:

$$\pi_\theta(a \mid s) = \frac{\exp\left(\phi(s,a)^\top \theta\right)}{\sum_{a'} \exp\left(\phi(s,a')^\top \theta\right)} \tag{8.33}$$

The score function for the softmax policy is then:

$$\nabla_\theta \log_\theta \pi(a \mid s) = \nabla_\theta \left[\phi(s,a)^\top \theta - \log \sum_{a'} \exp\left(\phi(s,a')^\top \theta\right) \right] \tag{8.34}$$

$$= \phi(s,a) - \frac{1}{\sum_{a'} \exp\left(\phi(s,a')^\top \theta\right)} \nabla_\theta \sum_{a'} \exp\left(\phi(s,a')^\top \theta\right) \tag{8.35}$$

$$= \phi(s,a) - \frac{1}{\sum_{a'} \exp\left(\phi(s,a')^\top \theta\right)} \sum_{a'} \phi(s,a') \exp\left(\phi(s,a')^\top \theta\right) \tag{8.36}$$

$$= \phi(s,a) - \sum_{a'} \phi(s,a') \frac{\exp\left(\phi(s,a')^\top \theta\right)}{\sum_{a''} \exp\left(\phi(s,a'')^\top \theta\right)} \tag{8.37}$$

$$= \phi(s,a) - \sum_{a'} \phi(s,a') \pi_\theta(a' \mid s) \tag{8.38}$$

$$= \phi(s,a) - \mathbb{E}_{a' \sim \pi_\theta(a'|s)} \left[\phi(s,a')\right] \tag{8.39}$$

8.8.2 *Continous action space: Gaussian policy*

For continuous action spaces, a common choice is a Gaussian policy $a \sim \mathcal{N}(\mu(s), \sigma^2)$.

- The mean action is a linear combination of state features: $\mu(s) = \phi(s)^\top \theta$
- The variance σ^2 can be fixed, or also parameterized

The score function is:

$$\nabla_\theta \log \pi_\theta(a \mid s) = \frac{(a - \mu(s))\phi(s)}{\sigma^2} \tag{8.40}$$

8.9 *Variance Reduction with a Baseline*

A weakness of Monte Carlo policy gradient algorithms is that the returns $G_t^{(i)}$ often have high variance across multiple episodes. One way to address this is to subtract a **baseline** $b(s)$ from each $G_t^{(i)}$. The baseline can be any function, as long as it does not vary with a.

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T-1} (G_t - b(s_t)) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{8.41}$$

First, why do we want to do this? Intuitively, we can think of $(G_t - b(s_t))$ as an estimate of how much better we did after time step t than is expected by the baseline $b(s_t)$. So, if the baseline is approximately equal to the expected return $b(s_t) \approx \mathbb{E}[r_t + r_{t+1} + \cdots + r_{T-1}]$, then we will be increasing the log-probability of action a_t proportionally to how much better the return G_t is than expected. Previously, we were increasing the log-probability proportionally to the magnitude of G_t, so even if the policy always achieved exactly its expected returns, we would still be applying gradient updates that could cause it to diverge. The quantity $(G_t - b(s_t))$ is usually called the **advantage**, A_t . We can estimate the true advantage from a sampled trajectory $\tau^{(i)}$ with

$$\hat{A}_t = (G_t^{(i)} - b(s_t)) \tag{8.42}$$

Secondly, why can we do this? It turns out that subtracting a baseline in this manner does not introduce any bias into the gradient calculation. $\mathbb{E}_\tau[b(s_t)\nabla_\theta \log \pi\theta(a_t \mid s_t)]$ evaluates to zero, and hence has no effect on the gradient update.

$$\begin{aligned}
&\mathbb{E}_{\tau\sim\pi_\theta}[b(s_t)\nabla_\theta \log \pi_\theta(a_t \mid s_t)] \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[\mathbb{E}_{s_{(t+1):T},a_{t:(T-1)}}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)b(s_t)]\right] && \text{(break up expectation)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{s_{(t+1):T},a_{t:(T-1)}}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)]\right] && \text{(pull baseline term out)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)]\right] && \text{(remove irrelevant variables)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\sum_{a_t}\pi_\theta(a_t \mid s_t)\frac{\nabla_\theta\pi_\theta(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}\right] && \text{(expand expectation, take derivative of log)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\sum_{a_t}\nabla_\theta\pi_\theta(a_t \mid s_t)\right] && \text{(cancel } \pi_\theta\text{)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\nabla_\theta\sum_{a_t}\pi_\theta(a_t \mid s_t)\right] && \text{(move gradient)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\nabla_\theta 1\right] && \text{(sum over } \pi_\theta(\cdot \mid s_t) = 1\text{)} \\
&\quad = \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\cdot 0\right] && \text{(derivative of constant is zero)} \\
&\quad = 0 && \text{(thus, no added bias)}
\end{aligned}$$

8.9.1 *Vanilla policy gradient*

Using the baseline as described above, we introduce the ''vanilla'' policy gradient algorithm. Suppose that the baseline function has parameters $\mathbf{w}$.

function POLICYGRADIENT(α)
  Initialize policy parameters θ and baseline values $b(s)$ for all s (e.g., to 0)
  for iteration $= 1, 2, \ldots$ **do**
    Collect a set of m trajectories by executing the current policy π_θ
    for each time step t of each trajectory $\tau^{(i)}$ **do**
      Compute the *return* $G_t^{(i)} = \sum_{t'=1}^{T-1} r_{t'}$
      Compute the *advantage estimate* $\hat{A}_t^{(i)} = G_t^{(i)} - b(s)$ ▷ $b(s)$ is often an estimate of $V(s)$
    Re-fit the baseline to the empirical returns by update $\mathbf{w}$ to minimize:

$$\sum_{i=1}^{m} \sum_{t=0}^{T-1} \| b(s_t) - G_t^{(i)} \|^2$$

    Update policy parameters θ using the policy gradient estimate $\hat{g}$:

$$\hat{g} = \sum_{i=1}^{m} \sum_{t=0}^{T-1} \hat{A}_t^{(i)} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})$$

    (using an optimizer like SGB ($\theta \leftarrow \theta + \alpha \hat{g}$) or Adam)
  return θ and baseline values $b(s)$

Algorithm 8.2. Vanilla policy gradient algorithm.

One natural choice for the baseline is the state value function, $b(s_t) = V(s_t)$. Under this formulation, we can define the advantage function as $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. However, since we do not know the true state values, we instead use an estimate $\hat{V}(s_t; \mathbf{w})$ for some weight vector $\mathbf{w}$. We can simultaneously learn the weight vector $\mathbf{w}$ for the baseline (state-value) function and policy parameters θ using the Monte Carlo trajectory samples.

Note that in the above algorithm, we usually do not compute the gradients $\sum_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ individually. Rather, we accumulate data from a batch into a loss function

$$L(\theta) = \sum_t \hat{A}_t \log \pi_\theta(a_t \mid s_t) \tag{8.43}$$

and then apply the gradients all at once by computing $\nabla_\theta L(\theta)$. We can also introduce a component into this loss to fit the baseline function:

$$L(\theta, \mathbf{w}) = \sum_t \left(\hat{A}_t \log \pi_\theta(a_t \mid s_t) - \|b(s_t) - G_t^{(i)}\|^2 \right) \tag{8.44}$$

We can then compute the gradients of $L(\theta, \mathbf{w})$ w.r.t. θ and $\mathbf{w}$ to perform SGD updates.

8.9.2 *N-step estimators*

In the above derivations, we have used the Monte Carlo estimates of the reward in the policy gradient approximation. However, if we have access to a value function (for example, the baseline), then we can also us TD methods for the policy gradient update, or any intermediate blend between TD and MC methods:

$$\begin{aligned}
\hat{G}_t^{(1)} &= r_t + \gamma V(s_{t+1}) \\
\hat{G}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
&\cdots \\
\hat{G}_t^{(\text{inf})} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) + \cdots
\end{aligned}$$

which we can also use to compute advantages:

$$\begin{aligned}
\hat{A}_t^{(1)} &= r_t + \gamma V(s_{t+1}) - V(s_t) \\
\hat{A}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) \\
&\cdots \\
\hat{A}_t^{(\text{inf})} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) + \cdots - V(s_t)
\end{aligned}$$

$\hat{A}_t^{(1)}$ is a purely TD estimate, and has low variance, but high bias. $\hat{A}_t^{(\text{inf})}$ is a purely MC estimate, and has zero bias, but high variance. If we choose an intermediate value of k for $A_t^{(k)}$, we can get an intermediate amount of bias and an intermediate amount of variance.

8.9.3 *Common template of policy gradient algorithms*

Many policy gradient algorithms follow a common template.

function POLICYGRADIENTTEMPLATE()
  for iteration $= 1, 2, \ldots$ **do**
    Run policy for T timesteps or N trajectories
    At each timestep in each trajectory, compute target $Q^\pi(s_t, a_t)$ and baseline $b(s_t)$
    Compute estimate policy gradient $\hat{g}$
    Update the policy using $\hat{g}$, potentially constrained to a local region
  return θ and baseline values $b(s)$

Algorithm 8.3. Common template of policy gradient algorithms.

References

1. M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, ''The Arcade Learning Environment: An Evaluation Platform for General Agents,'' *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013 (cit. on p. 1).
2. M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019 (cit. on p. iv).
3. N. Kohl and P. Stone, ''Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion,'' in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, 2004 (cit. on p. 16).
4. A. Krizhevsky, I. Sutskever, and G. E. Hinton, ''Imagenet Classification with Deep Convolutional Neural Networks,'' *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 25, pp. 1097–1105, 2012 (cit. on p. 1).
5. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., ''Human-Level Control Through Deep Reinforcement Learning,'' *Nature*, vol. 518, no. 7540, pp. 529–533, 2015 (cit. on pp. 1, 4, 6, 7).
6. V. Nair and G. E. Hinton, ''Rectified Linear Units Improve Restricted Boltzmann Machines,'' in *Icml*, 2010 (cit. on p. 3).
7. T. Schaul, J. Quan, I. Antonoglou, and D. Silver, ''Prioritized Experience Replay,'' *ArXiv Preprint ArXiv:1511.05952*, 2015 (cit. on p. 5).
8. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018 (cit. on p. 12).
9. H. Van Hasselt, A. Guez, and D. Silver, ''Deep Reinforcement Learning with Double Q-Learning,'' in *AAAI Conference on Artificial Intelligence (AAAI)*, vol. 30, 2016 (cit. on pp. 1, 7).
10. Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, ''Dueling Network Architectures for Deep Reinforcement Learning,'' in *International Conference on Machine Learning (ICML)*, 2016 (cit. on pp. 1, 8, 10, 11).