Approximate Methods

July 13, 2018

1 Linear Approximate Methods for Reinforcement Learning

For many realistic environments, the state space is simply too complex for tabular methods in practice (finite time and memory). This can be either because the number of state become to large or because of continuous states (e.g. the **curse of dimensionality**). We therefore need a method that can *generalize* from past experience to new situations. The type of generalization we require is often called **function approximation**. This notebook gives a brief overview on approximate methods for Reinforcement Learning, and shows how to extend tabular SARSA from the previous notebook to use function approximation.

In this notebook we focus on linear approximations. Although linear approximations are less powerfull than their non-linear counter part (like deep neural networks) they provide much stronger theoretical foundations. Linear approximate methods are therefore a good place to start learning about approximate reinforcement learning.

1.1 (State / State-Action) Value Function Approximation

As in the tabular case we focus on methods that estimate a value function. The goal is to approximate the value function from the experience generated using a known policy π. For simplicity we will use describe the state value function v_π, but these result can easily be extended to the state-action value function q_π. The key difference from tabular methods is that the value function is now represented as a *functional form parametrized by a weight vector* $\mathbf{w} \in \mathbb{R}^d$, where d is the number of parameters. We write the approximation of the state value as $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$.

It is important to note that we will have less weights than number of states, which implies that with function approximation updates on states are **no longer decoupled**. I.e. updating one weight will change the value of several states. It is therefore not possible to perfectly estimate all the state values simultaneously. We must therefore, implicitly or explicitly, determine how much we care about accurate value estimates for each state, s. A common solutions is to use the fraction of time spent in s by the agent during training. We call this distribution the **on-policy distribution**, and dennote it by $\mu(s)$.

This is important to notice when we talk about the **error on a state** s: this error will be understood by the squared difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$, weighted by a specified **state distribution**, for instance $\mu(s)$:

$$\overline{VE} = \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi - \hat{v}(s, \mathbf{w}) \right]^2 .$$

we will call this quantity the *Mean Squared Value Error*. Having specified this error function, the ideal goal will then be to find an optimal weight vector $\mathbf{w}^*$ for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for

all possible $\mathbf{w}$. In general this is not realistically achievable, so the goal of function approximation is to converge to a **local optimum**, in which the aforementioned relation holds but only in a *neighborhood* of $\mathbf{w}^*$.

1.2 Stochastic Gradient Descent

We want to minimize the $\overline{VE}$ error taking into consideration the observed experience. *Stochastic Gradient Descent* (SGD) methods will allow us to do so by adjusting the weight vector after each example is observed from experience, moving it in the opposite direction of the **gradient** of the error function, which is the direction that reduces the error evaluated on the considered experience. For a single sample this results in the following update:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \frac{1}{2}\alpha\nabla\overline{VE} \\ &= \mathbf{w}_t + \frac{1}{2}\alpha\nabla\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]^2 \\ &= \mathbf{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla\hat{v}(S_t, \mathbf{w}_t).\end{aligned}$$

Where α is the learning rate. By using the on-policy distribution we can completely ignore μ in the expression as the fraction of times s is experienced exactly follows this distribution.

Obviously we will not have access to the true value $v_\pi(S_t)$ available, so we need to replace the target of the update with an *approximation* of it. The choice of how we decide to approximate v_π gives rise to two families of algorithms: *true gradient methods* and *semi-gradient methods*. If we call U_t the chosen approximation, the SGD weight update becomes

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla\hat{v}(S_t, \mathbf{w}_t).$$

We can now see that if we choose U_t to be an *unbiased estimate* of v_π (so that $\mathbb{E}\left[U_t|S_t = s\right] = \mathbb{E}v_\pi(S_t)$ for each t), then $\mathbf{w}_{t+1}$ is guaranteed to converge to a local optimum. This is the case if we choose as a target the *Monte Carlo target*,

$$U_t = \sum_{k=t}^{T}\gamma^{k-t}R_{k+1} = G_t$$

This means that we need to run an episode until it terminates before making any updates.

Instead we might wish to use **bootsrapped targets**, i.e. using current estimates of the value funciton to update the value function. In this case the target is on the form

$$U_t = \sum_{k=t}^{t+n-1}\gamma^{k-t}R_{k+1} + \gamma^n v(S_{t+n})$$

When computing the gradients we ignore the dependence of U_t on the weight parameter $\mathbf{w}_t$, and treat it as a constant. These methods are therefore known as **semi-gradient methods**. Bootstrapped target are *biased*, but tend to lead to much faster convergence in practice (see Sutton and Barto [1] for details).

1.3 Linear Approximation

One of the simplest but nevertheless important function approximator is is the **linear function approximator**, where $\hat{v}(\cdot, \mathbf{w})$ is simply a linear transformation, parameterized by the weight vector $\mathbf{w}$.

Corresponding to each state s, thereis a real-valued vector $\mathbf{x}(s)$ (with the same number of components as $\mathbf{w}$) which we will call **feature vector** represeting the state s. Linear methods approximate the state-value function as the inner product between $\mathbf{x}(s)$ and $\mathbf{w}$:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w} \cdot \mathbf{x}(s) = \sum_{i=1}^{d} w_i \cdot x_i(s).$$

The gradient of the approximate function w.r.t. the parameter vector $\mathbf{w}$ in this case is simply $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$, so that the SGD update becomes

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \mathbf{x}(S_t)$$

Linear methods are useful for studying the convergence properties of approximate methods. In particular it is important to notice how linear methods assure the existance of *only one optimum*, so that any method that is able to converge to a local optimum is guaranteed to converge to the global optimum.

2 Algorithms for on-policy control with approximation

The results regarding approximate methods for state-value estimation presented above are easily extended to **controlling tasks**: all that is required is to replace in the formuale the state-value function $v_\pi(s)$ with the *action-value function* $q_\pi(s, a)$. We will now present the *n-step SARSA algorithm* (as alread presented in the tabular case) adapted to function approximation for solving episodic tasks. We will then briefly go over the effects of introducing *off-policy methods* to function approximations.

2.1 Episodic semi-gradient SARSA

According to what we said above, it should appear clear that the only thing left to specify in our update is the *target* U_t. If we choose it to be the **n-step return** as specified by Sutton and Barto [1]:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} R_{k+1} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_t)$$

then the n-step update equation becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t) = \mathbf{w}_t + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \mathbf{x}(S_t, A_t)$$

For our linear model.

Many counterexamples have been devised to show this behaviour.

2.2 The Deadly Triad

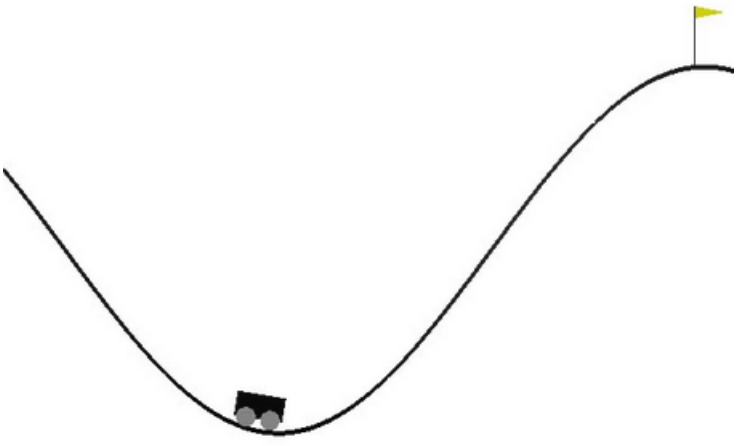
In the the notebook on tabular methods we also covered Q-learning, an off-policy method. It can be shown that off-policy approximate methods suffer from severe *instability* issues when combined with bootstrapping When these three components are present at the same time it is called **the deadly triad**. Each of these components provide desirable traits:

- **Function Approximation**: it offers a scalable way of *generalizing state spaces* that are much larger than the available memory and computational resources. Clearly, this cannot be given away;
- **Boostrapping**: even if not strictly necessary, it offers boosts in *computational and data efficiency* which are too "tempting" to be given away;
- **Off-policy learning**: it is *essential* for certain use cases and play an important role in the creation of agents with many desirable (human like) intelligence traits.

There are techniques that enable combining all three componetns, but at the cost of convergence guarantees. Since we are interested in cases with sound theoretical foundations we will leave off-policy methods for another discussion (see Sutton and Barto [1] for details).

3 Environment: OpenAI gym

In order to demonstrate linear function approximation we will in the following implement and validate *n-step SARSA* with linear function approximation on the **Mountain Car environment**, offered in the **OpenAI gym** toolkit. In this environment we are faced with the task of driving a car on a one-dimensional track. The car is positioned between two "mountains", and the goal is placed upon the rightmost mountain. The car's engine is underpowerd, and cannot push the car over the slope directly, so to reach the goal is necessary to build up momentup by oscillating back and forth between the two mountains first.



The `MountainCar-v0` environment implemented by OpenAI

The agent receives a reward of -1 on all timesteps until the car moves over the goal, which terminates the episode. The episode also terminates after 200 timesteps. There are three possible actions: accelerate backwards, accelerate forwards and do nothing. The car moves according to

a simplified physics model. Each state s is a two-dimensional real vector where the first coordinate represents the position of the car on the one-dimensional track, whilst the second coordinate represents the car's velocity.

3.1 Tile Coding

We specified both the target and the approximator used in our algorithm, but we also need to specify the **features** that we want to use. Since we are using a simple linear function approximator it isn't possible for the model to sufficiently distinguish between different states. The possible solutions are 1) use a more powerful function approximator (e.g. neural networks) or 2) feature engineering. In this notebook we will use the feature engineering approach, specifically **tile coding**, which maps real values a higher number of binary features.

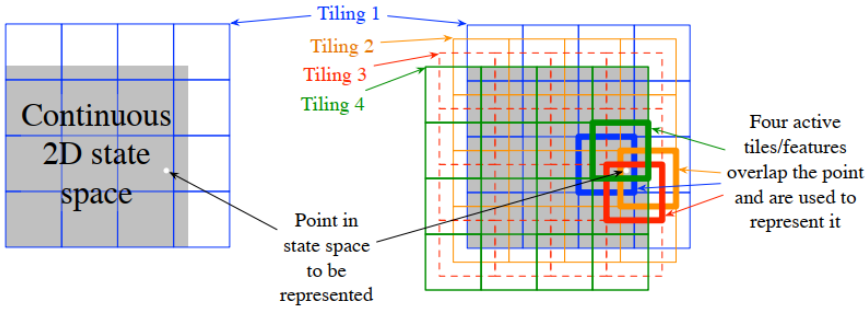


Figure 9.9 from Sutton Barto

Let's consider a two-dimensional state space, as in the image above. Tile coding divides each dimension of the state space into n *bins*, which are then used to build a $n \times n$-dimensional binary feature matrix $\mathbf{x}$. Each component of the feature matrix, $x_{i,j}$, will be either 1 or 0 depending on whether the state is contained in the $\{i, j\}$-th bin or not. One of said binning of the state space is called a **tiling**. If we limit ourselves to only one tiling, we are simply aggregating together states and encoding them in a binary vector. A more strong encoding can be achieved by considering *overlapping tilings*: these are build by considering multiple tilings, each and one of them obtained by adding an *offset* to the original tiling. Say that we decide to use m tilings to encode our state space, we will end up with a $m \times (n \times n)$ binary matrix representing a state s. Tile coding allows us to have a sparse representation of a continuous state space, and results in a flexible and computationally flexible feature transformation.

```
In [1]: ## Useful Jupyter setup commands
        %load_ext autoreload
        %autoreload 2
        get_ipython().run_line_magic('matplotlib', 'inline')

In [2]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        from IPython.display import clear_output
        import numpy as np
        import matplotlib.pyplot as plt
```

```
import gym

import utils
from tileEncoder import TileEncoder
from agents import ApproximateNStepSARSA, RandomAgent
```

4 Experiments

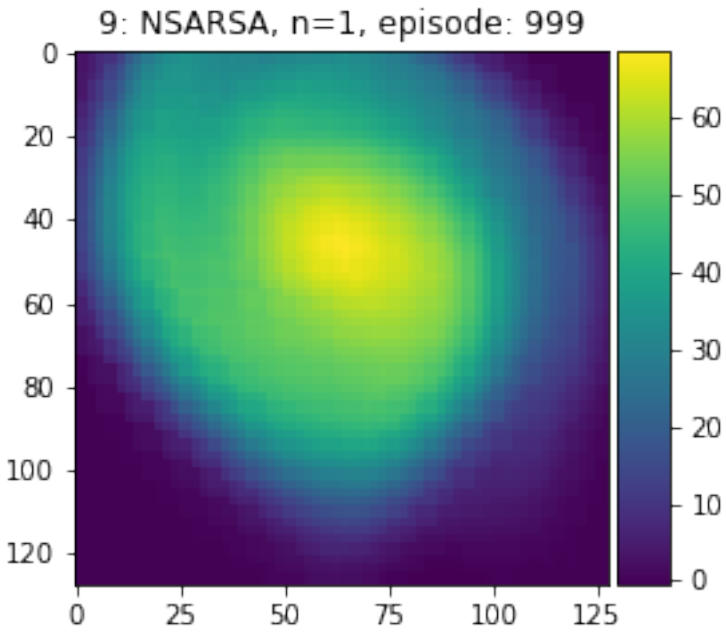
Below we train approximate n-step SARSA the Mountain Car environment (for clarity we simply use $n = 1$). In this notebook we will use an ϵ-greedy policy, with $\epsilon = 0.1$ held constant, unless noted otherwise. We encode the state space using tile coding, using $k = 8$ tilings and $l = 8$ bins for each tiling.

During training we monitor - The cost-to-go function, intended as $-\max_a \hat{q}(s, a, \mathbf{w})$

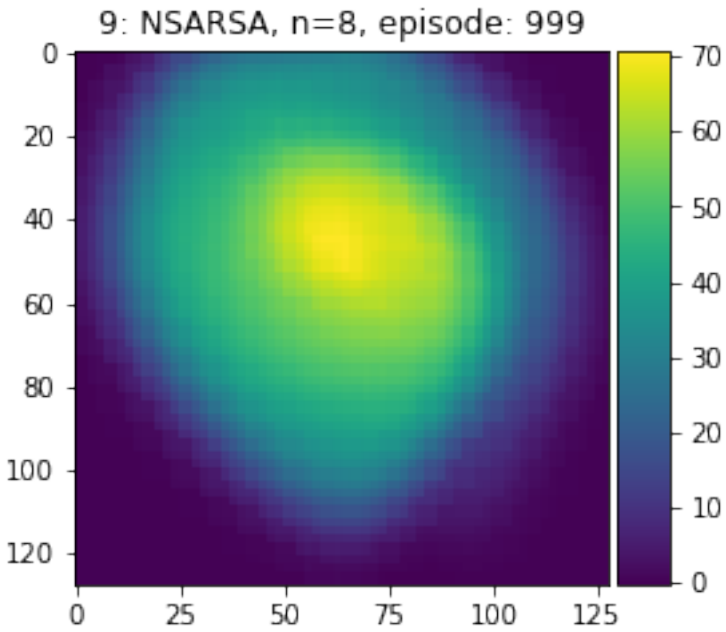
In addition, we render a full episode until termination after every 100 episodes.

In [3]:
```
## Run settings
num_runs = 10  # Number of runs to average rewards over
eps_per_run = 1000  # Number of episodes (terminations) per run

alpha = 0.01

nbins = 8
ntiles = 8

# n parameter in n-step Bootstrapping
n1 = 1  # agent 1
n2 = 8  # agent 2
```

In [4]:
```
ApproxNSARSALearning_rewards_n1 = []
env = TileEncoder(gym.make('MountainCar-v0'),nbins=nbins,ntiles=ntiles)
for i in range(num_runs):
    NSARSA_Learning = ApproximateNStepSARSA(env.obspace_shape(), env.nactions(), n=n1,
    _, rewards = utils.approx_run_loop(env, NSARSA_Learning, str(i)+': NSARSA, n='+str
    ApproxNSARSALearning_rewards_n1.append(rewards)
env.close()

ApproxNSARSALearning_rewards_n1  = np.array(ApproxNSARSALearning_rewards_n1)
```



In [5]:
```
ApproxNSARSALearning_rewards_n2 = []
env = TileEncoder(gym.make('MountainCar-v0'),nbins=nbins,ntiles=ntiles)
for i in range(num_runs):
    NSARSA_Learning = ApproximateNStepSARSA(env.obspace_shape(), env.nactions(), n=n2,
#     NSARSA_Learning = RandomAgent(num_actions=env.nactions())
    _, rewards = utils.approx_run_loop(env, NSARSA_Learning, str(i)+': NSARSA, n='+str
    ApproxNSARSALearning_rewards_n2.append(rewards)
env.close()

ApproxNSARSALearning_rewards_n2 = np.array(ApproxNSARSALearning_rewards_n2)
```

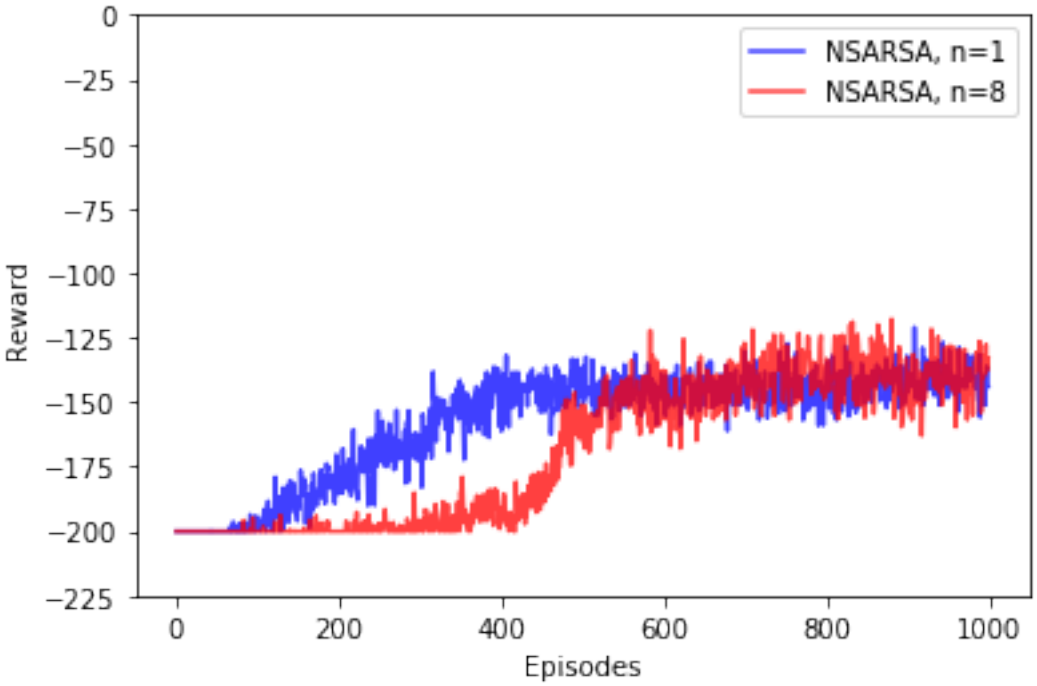


5 Discussion

As we can see by the cost-to-go function, the agent learns to oscillate between the two mountains. This can be seen by noticing the circular trajectories followed by the agent in the state space. Note that to states which are explored most frequently are associated the worst action-values.

Below we plot the (smoothened) average reward obtained by the approximate n-step SARSA as a function of the number of simulated episodes.

```
In [10]: plt.figure()
         utils.reward_plotter(ApproxNSARSALearning_rewards_n1, 'NSARSA, n='+str(n1), col='b', s
         utils.reward_plotter(ApproxNSARSALearning_rewards_n2, 'NSARSA, n='+str(n2), col='r', s
         axes = plt.gca()
         axes.set_ylim([-225, 0])
         plt.show()
```



6 Bibliographic Notes

[1] Richard S. Sutton and Andrew G. Barto. 1998. Introduction to Reinforcement Learning (1st ed.). MIT Press, Cambridge, MA, USA.