A Complete Reinforcement Learning System (Capstone)

Miguel A. Saavedra-Ruiz

March 2020

1 Milestone 1: Formalize Word Problem as MDP

Formalizing the Problem

The lunar lander is a spaceship whose task is to land in the moon Fig. 1. We will train an agent in a simulator to find a robust and efficient landing policy. Then, we could deploy the agent on the moon and it allow it to continue learning adjusting its value function and policy to account for the differences between our simulator and reality.

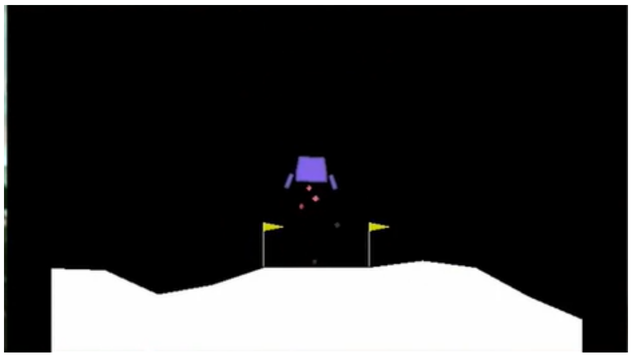


Figure 1: Lunar lander

Our goal is to land the lunar module in the landing zone located between the two yellow flags. The Landing zone is always in the same location, but the shape of the ground around it may change. We can fire the main thruster or either of the side thrusters to orient the module and slow its descent. **The state is composed of eight variables**. It includes the $X and Y$ **position** and **velocity** of the module as well as its **angle** and **angular velocity** with respect to the ground. We also have a sensor for each leg that determines if it is touching the ground Fig. 2

Environment

The environment inputs the action that actually is given to the dynamics function along with the current state to produce a next state. The next state in action will then be passed to reward function that encodes the desired behavior. Finally, the environment will omit the next state and the reward Fig. 3

Actions

There are four actions that the agent can take and these are listed below.

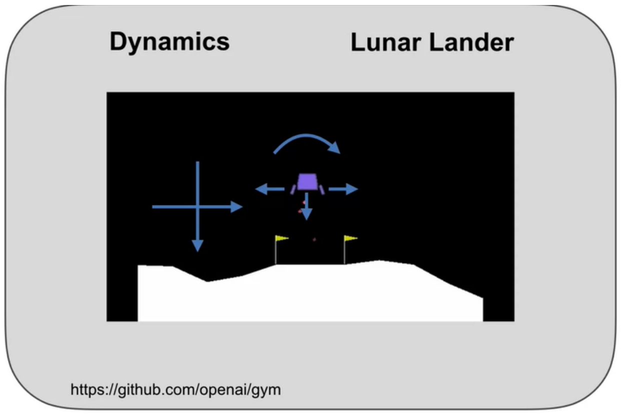
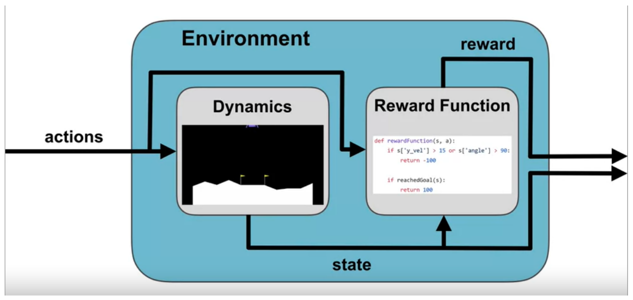


Figure 2: Lunar lander agent



```
def rewardFunction(s, a):
    if s['y_vel'] > 15 or s['angle'] > 90:
        return -100

    if reachedGoal(s):
        return 100
```

Figure 3: Lunar lander environment

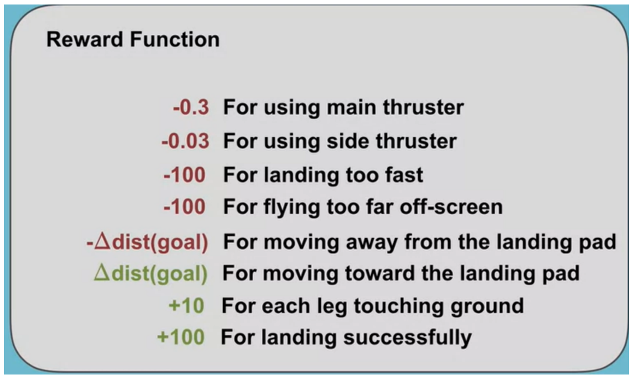
- The agent can fire the main thruster.
- Fire the left thruster.
- Fire the right thruster.
- Do nothing at all on this time step.

The first goal is to implement the reward function for this environment. Fuel is expensive and the main thruster uses a lot of it. We want to discourage the agent from using the main thruster more than necessary. The side thrusters use less fuel, so it is less bad for the agent to use those frequently. We want to encourage the agent to move towards the goal. So it will lose some reward based on how far it moved from the goal since the last time step. Let's try to discourage the agent from learning to pile the module to the surface in ways that might damage the equipment and will also discourage flying off into outer space or a distant creator never to be seen again. The agent will be rewarded for each leg that it manages to get touching the ground. And the agent will receive a large reward for successfully landing in the landing pad at an appropriate velocity Fig. 4.

2 Milestone 2: Choosing The Right Algorithm

Choosing the Learning Algorithm

First step, can we represent the value function using only a table? Let's recall the state space of the lunar lander problem. The agent observes the position, orientation, velocity and contact sensors of the lunar



Reward Function

-0.3	For using main thruster
-0.03	For using side thruster
-100	For landing too fast
-100	For flying too far off-screen
-Δdist(goal)	For moving away from the landing pad
Δdist(goal)	For moving toward the landing pad
+10	For each leg touching ground
+100	For landing successfully

Figure 4: Reward function of the lunar lander

module Fig. 2. Six of the eight state variables are continuous, which means that **we cannot represent them with a table**. And in any case we'd like to take advantage of **generalization** to learn faster.

Would this be well formulated as an average word problem? Think about the dynamics of this problem. The lunar module starts in low orbit and descends until it comes to rest on the surface of the moon. This process then repeats with each new attempt at landing beginning independently of how the previous one ended. **This is exactly our definition of an episodic task**. We use the **average reward formulation for continuing tasks, so that is not the best choice here.**

Let's think about if it's possible and beneficial to update the policy and value function on every time step, we can use Monte Carlo or TD. But think about landing your module on the moon. If any of our sensors becomes damaged during the episode, **we want to be able to update the policy before the end of the episode.** It's like what we discussed in the driving home example, we expect the **TD method to do better in this kind of problem**.

Finally, let's not lose sight of the objective here. **We want to learn a safe and a robust policy in our simulator so that we can use it on the moon**. We want to learn a **policy that maximizes reward**, and so **this is a control task**.

This leaves us with three algorithms, SARSA, expected SARSA and Q-learning. Since we are using f**unction approximation, learning and epsilon soft policy will be more robust than learning a deterministic policy**. Expected SARSA and SARSA, both allow us to learn an optimal epsilon soft-policy, but Q-learning does not. Now we need to choose between expected SARSA and SARSA. We mentioned in an earlier video that expected SARSA usually performs better than SARSA. So, let's eliminate SARSA.

The chosen algorithm is Expected SARSA.

3 Milestone 3: Identify Key Performance Parameters

Overview of Design Choices

The general diagram of the lunar lander as an MDP looks like this Fig. 5.

Now we need to decide on the function approximator,the optimizer for updating the action values, and how to do exploration. Always start simple. For a function approximation, that would mean using a fixed basis like tile coding. Unfortunately, **that might not be the best choice for this problem without carefully designing the tile coder.** If you tile all the inputs together, the number of features grows exponentially with the input dimension. For example, if you want to use ten tiles per dimension for this eight-dimensional problem, you could easily end up with 100 million features Fig. 6.

Instead, **we will be using a neural network**. One hidden layer should be sufficiently powerful to represent

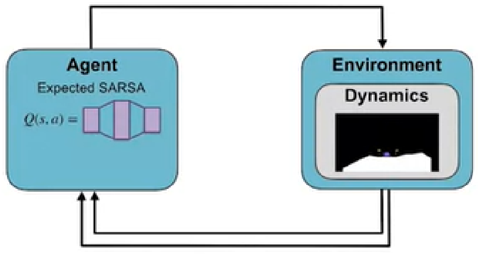


Figure 5: Lunas lander MDP

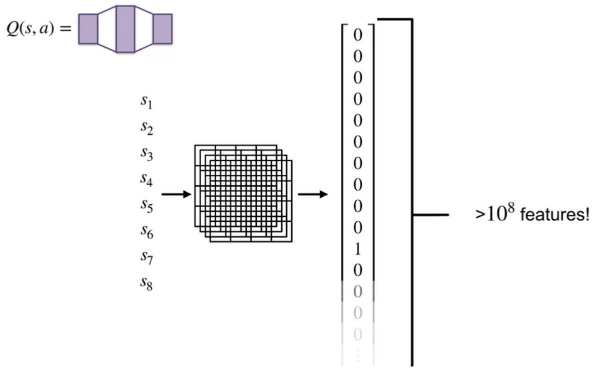


Figure 6: Tile coder for the lunar lander

the value function for lunar lander, and it will be a bit easier for you to implement. We need to decide the number of hidden units in that layer. Remember that you get to choose the size of the hidden layer of a neural network Fig. 7.

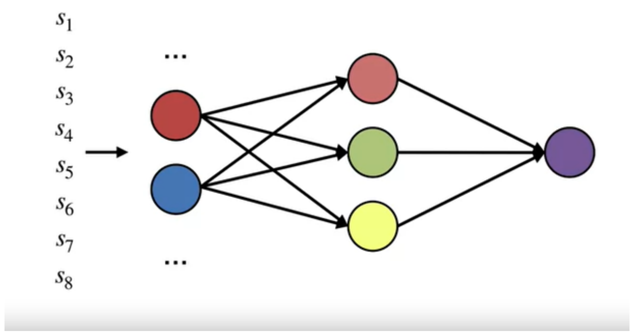


Figure 7: Neural Network for the lunar lander

The activation functions for this problem will be **rectified linear units or ReLUs**.

To train the Neural Network, Using baseline stochastic gradient descent will likely be too slow for this project. We could try this algorithm called adagrad. The downside to this is that adagrad decays the step sizes towards zero, which can be problematic for non-stationary learning. We could try RMSProp, which uses information about the curvature of the loss to improve the descent step. However, we'd like to also incorporate momentum to speed up learning. **A good choice can be the ADAM optimizer**. This combines the curvature information from RMSProp and momentum.

The last thing to discuss is which exploration method will be used. Optimistic initial values would be a reasonable choice if we were using a linear function approximator with non-negative features. But since we are using a neural network, it is difficult to maintain optimistic values and so is unlikely to be effective Fig. 8.

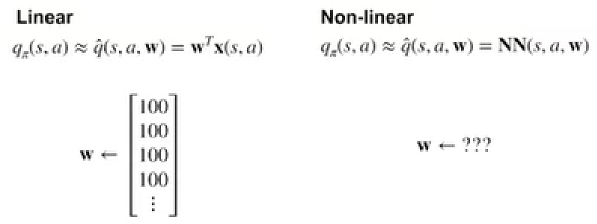

Linear
$q_\pi(s,a) \approx \hat{q}(s,a,\mathbf{w}) = \mathbf{w}^T\mathbf{x}(s,a)$
Non-linear
$q_\pi(s,a) \approx \hat{q}(s,a,\mathbf{w}) = \mathbf{NN}(s,a,\mathbf{w})$
$\mathbf{w} \leftarrow \begin{bmatrix}100\\100\\100\\100\\\vdots\end{bmatrix}$
$\mathbf{w} \leftarrow ???$

Figure 8: Exploring with linear methods for the lunar lander

We can also consider Epsilon greedy, this is very straight forward to implement. The downside though is that it's exploration completely ignores whatever information the action values might have. It is equally likely to explore an action with really negative value as an action with moderate value. **The choice of a Softmax Policy** could be better because the probability of selecting an action is proportional to the value of that action. This way we are less likely to explore actions that we think are really bad.

There are few things to consider when using a Softmax on the action values. First, let's think about how it affects the expected SARSA update. Remember that we need to compute the expectation over action values for the next state. This means we'll need to compute the probabilities for all the actions first under the Softmax function Fig. 9.

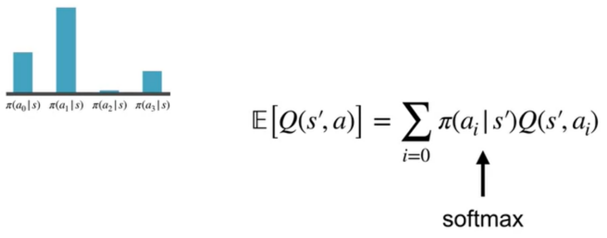

$\pi(a_0|s)$
$\pi(a_1|s)$
$\pi(a_2|s)$
$\pi(a_3|s)$
$\mathbb{E}\left[Q(s',a)\right] = \sum_{i=0} \pi(a_i|s')Q(s',a_i)$
softmax

Figure 9: Softmas policy with Expected Sarsa

Next, we also need to consider how much the agent focuses on the highest value actions. We control those with a temperature parameter called τ. If τ is large, then the agent is more stochastic and selects more of the actions Fig. 10.

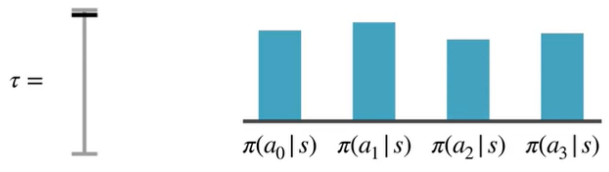


Figure 10: Large τ

For very large τ, the agent behaves nearly like a uniform random policy. For very small τ, the agent mostly selects the greedy action Fig. 11.

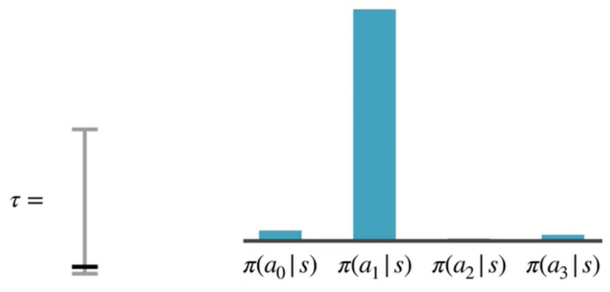


Figure 11: Small τ

Finally, we need to consider an additional trick to avoid overflow issues when computing the Softmax. Imagine that the action values are large. Exponentiating those values can get very large. Instead, we can use the fact that subtracting a constant from the action values when computing the probabilities has no effect. For example, we can subtract the maximum action value divided by the temperature. Then, all the exponents are negative and we avoid taking the exponent of large positive numbers Fig. 12.

$$\text{softmax}\left(Q(s,a) - \frac{\max_a Q(s,a)}{\tau} \right)$$

Figure 12: Correcting the overflow issue

Altogether, we now have a reasonable strategy to learn an optimal soft policy that also explores a bit more intelligently than Epsilon greedy. The agent takes actions according to its current Softmax policy and uses expected SARSA updates.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t))$$

4 Milestone 4: Implement Your Agent

Getting the Agent Details Right

Recall that we decided to use a neural network for the action values. For instance, in this case, the state would be composed of things like the position and velocity of the lander. We'll then use the resultant representation to estimate the value of each action. We do this by building a network that has one output node for each action Fig. 13.

To train this neural network to approximate the action value function, we will use the TD error to train the network. More precisely, we will modify the weights to reduce the TD error on each time step Fig. 14.

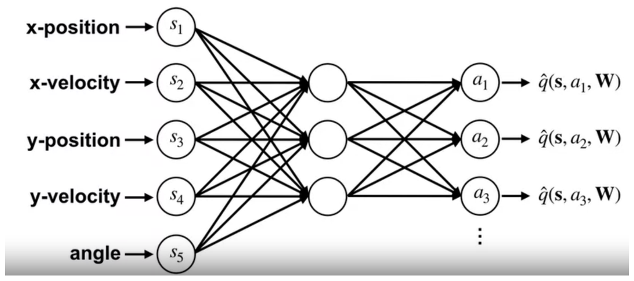

x-position
x-velocity
y-position
y-velocity
angle
s_1
s_2
s_3
s_4
s_5
a_1
a_2
a_3
$\hat{q}(\mathbf{s}, a_1, \mathbf{W})$
$\hat{q}(\mathbf{s}, a_2, \mathbf{W})$
$\hat{q}(\mathbf{s}, a_3, \mathbf{W})$

Figure 13: Neural network with States and actions

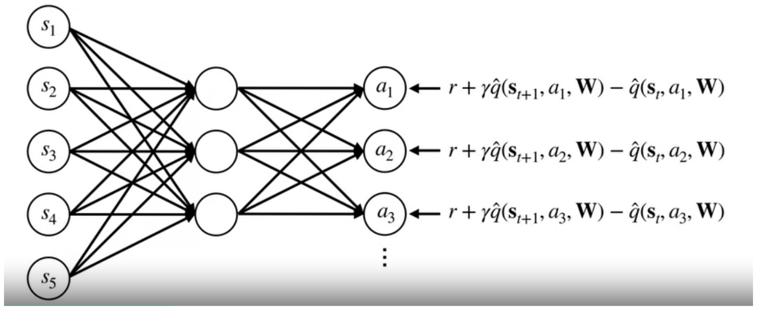

s_1
s_2
s_3
s_4
s_5
a_1
a_2
a_3
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_1, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_1, \mathbf{W})$
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_2, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_2, \mathbf{W})$
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_3, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_3, \mathbf{W})$

Figure 14: Training the Neural network with TD error

We will **only update the weights for the output corresponding to the action that was selected**. We simply do not update the weights in the last layer for the actions 2 and 3. For linear function approximation, we also maintain separate weights for each action value. We only updated weights for the action that was taken Fig. 15.

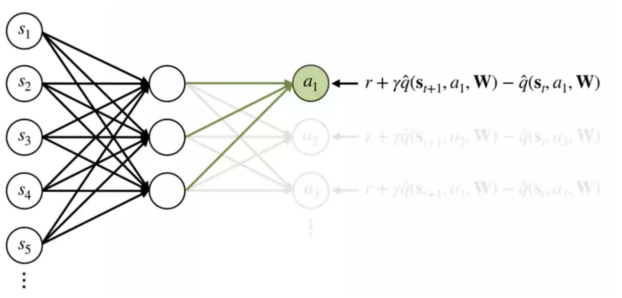

s_1
s_2
s_3
s_4
s_5
a_1
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_1, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_1, \mathbf{W})$

Figure 15: Updating the weights of the selected action

Other decision made was to use the ADAM algorithm. This algorithm combines both vector step-sizes and a form of momentum. Previously, we discussed vector step-sizes. Each way to the network has its own step-size adapted based on the statistics of the learning process. This means we can make larger updates to some weights, and smaller updates to the others Fig. 16.

This might be useful if the loss is flatter in some dimensions. Alternatively, we can take smaller steps in other dimensions where the loss changes more sharply.

We also discuss how to use momentum to accelerate our learning, especially if we find ourselves in a flat region of our loss. Remember that taking repeated steps in the same direction builds momentum. While taking steps in a different direction will kill the momentum.

The ADAM algorithm combines both of these ideas. It keeps a moving average of the gradients to compute the momentum. The β_m parameter is a meta-parameter that controls the amount of momentum. ADAM

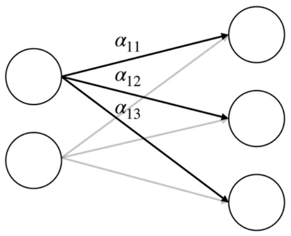
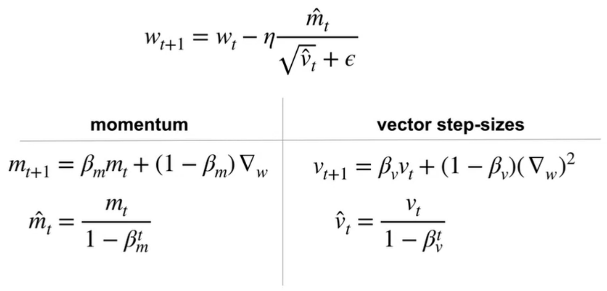


Figure 16: Multiple step-sizes

also keeps a moving average of the square of the gradient, this gives us a vector of step-sizes. This update typically results in more data-efficient learning because each update is more effective Fig. 17.

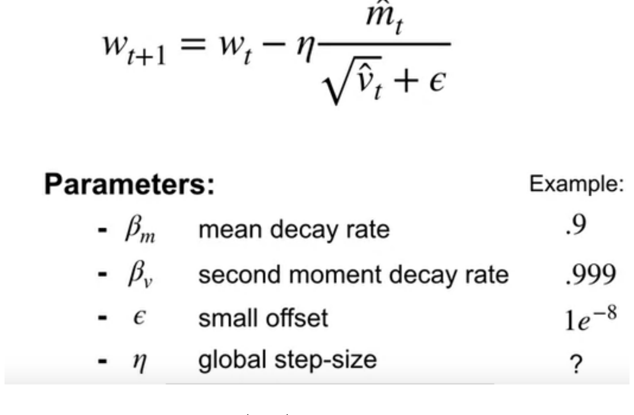


$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

momentum	vector step-sizes
$m_{t+1} = \beta_m m_t + (1-\beta_m)\nabla_w$	$v_{t+1} = \beta_v v_t + (1-\beta_v)(\nabla_w)^2$
$\hat{m}_t = \frac{m_t}{1-\beta_m^t}$	$\hat{v}_t = \frac{v_t}{1-\beta_v^t}$

Figure 17: ADAM algorithm

Now we have introduced several new meta parameters that we will need to set. We have the two decay rates, the size of the small offset in the denominator and a global step-size. So we haven't achieved meta-parameter free learning here. In fact, we have introduced four meta-parameters in place of one. Fortunately, it is typically not too difficult to find good settings for these meta-parameters using rules of thumb. But better performance can usually be achieved by tuning them individually Fig. 18.



$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Parameters:

		Example:
- β_m	mean decay rate	.9
- β_v	second moment decay rate	.999
- ϵ	small offset	$1e^{-8}$
- η	global step-size	?

Figure 18: ADAM meta-parameters

The task is to investigate the impact of different choices of the global step-size.

In-depth on Experience Replay

We can extend model based RL to the continuous RL case. For instance, experience replay is a simple method for trying to get the advantages of Dyna. The basic idea is to save a buffer of experience and let the data be the model. We sample experience from this buffer and update the value function with those samples similarly to how we sample from the model and update the values in Dyna.

Let's first take a look at how we store the experiences. For simplicity, let's go back to the tabular grid world. As you interact with the world, we observe a state, action, reward, next state tupple. We add these experiences to a buffer that we'll call the experience replay buffer. As we continue interacting with the world, we add more samples to this buffer until we eventually fill it up. When this happens, we can pick an older experienced to delete and replace with a new experience. We also need to consider the size of our buffer. By allowing the buffer to be really large, we can remember potentially useful transitions from many times steps ago. However, the agent does have practical limitations. We will need to consider the amount of memory used by large buffers and any computational impacts from storing and accessing large buffers Fig. 19.

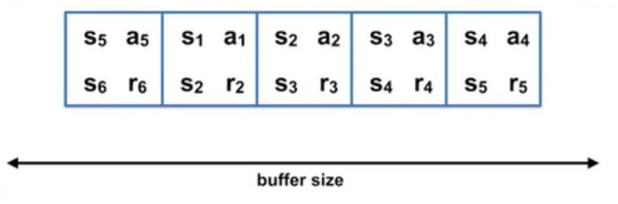
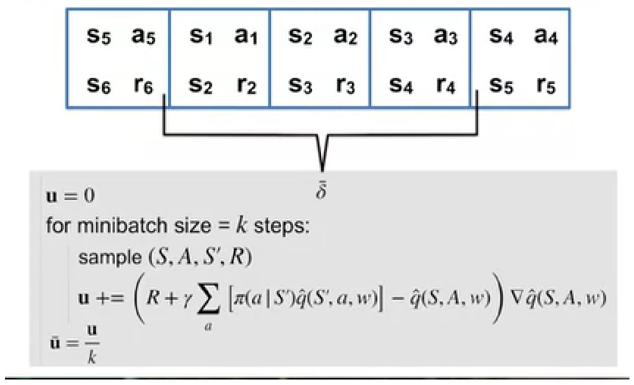


Figure 19: Buffer

Using a single sample from the experience replay buffer can produce a noisy update. Instead, we can use several samples from the buffer to create an average update to reduce that noise. The small collection of samples from the larger buffer is called a mini batch. The mini batch updates simply involves averaging the updates across the K random samples in the mini batch Fig. 20. Here's an example of computing this average update for q-learning. Restore the average update in a variable, let's call it U bar, then we grab several samples from the buffer and compute the q-learning update for each sample. Finally, we average those updates to get the mini batch update.



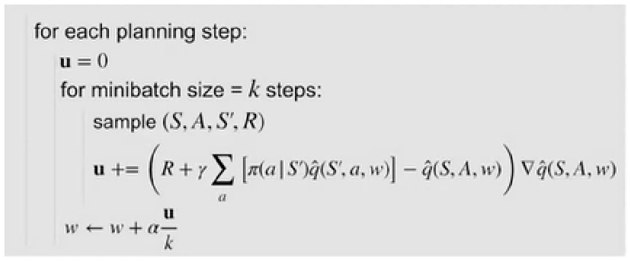
```
$\mathbf{u} = 0$
for minibatch size = $k$ steps:
    sample $(S, A, S', R)$
    $\mathbf{u} \mathrel{+}= \left(R + \gamma \sum_a \left[\pi(a \mid S')\hat{q}(S', a, w)\right] - \hat{q}(S, A, w)\right) \nabla \hat{q}(S, A, w)$
$\bar{\mathbf{u}} = \frac{\mathbf{u}}{k}$
```

Figure 20: Buffer with mini-batch

Putting it all together, we get the following pseudocode for experience replay. As in Dyna, we can think of the planning steps occurring in the background. The agent can perform several of these planning updates, which are the replay updates per each real step in the environment. More updates means we extract more from the data we have observed. Each of these replay updates consists of a mini batch update to reduce the noise of that update Fig. 21.

References

[1] University of Alberta. *A Complete Reinforcement Learning System (Capstone)*. 2019. URL: `https://www.coursera.org/learn/complete-reinforcement-learning-system` (visited on 03/19/2020).



```
for each planning step:
    $\mathbf{u} = 0$
    for minibatch size = $k$ steps:
        sample $(S, A, S', R)$
        $\mathbf{u} \mathrel{+}= \left(R + \gamma \sum_a \left[\pi(a \mid S')\hat{q}(S', a, w)\right] - \hat{q}(S, A, w)\right) \nabla \hat{q}(S, A, w)$
    $w \leftarrow w + \alpha \frac{\mathbf{u}}{k}$
```

Figure 21: Experience replay algorithm

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.