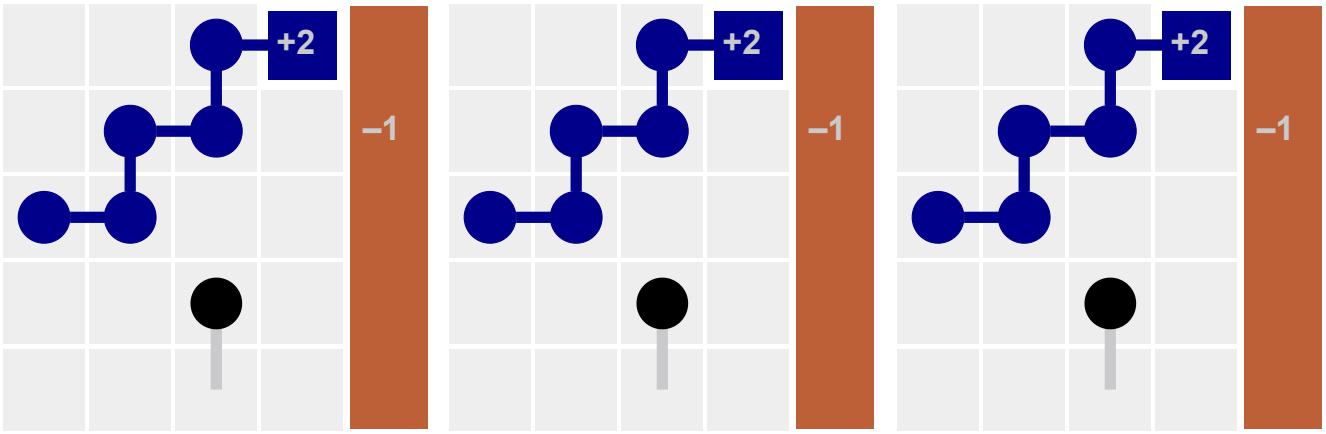
The Paths Perspective on Value Learning

A closer look at how Temporal Difference learning merges paths of experience for greater statistical efficiency.

PLAYING...

These value estimators behave differently where paths of experience intersect.



Monte Carlo

$V(s_t) \leftarrow R_t$

Temporal Difference

$V(s_t) \leftarrow r_t + \gamma V(s_{t+1})$

Q-Learning

$Q(s_t, a_t) \leftarrow r_t + \gamma V(s_{t+1})$

$V(s_{t+1}) = \max_a Q(s_{t+1}, a_{t+1})$

AUTHORS

Sam Greydanus

Chris Olah

AFFILIATIONS

Google Brain

OpenAI

PUBLISHED

Sept. 30, 2019

DOI

10.23915/distill.00020

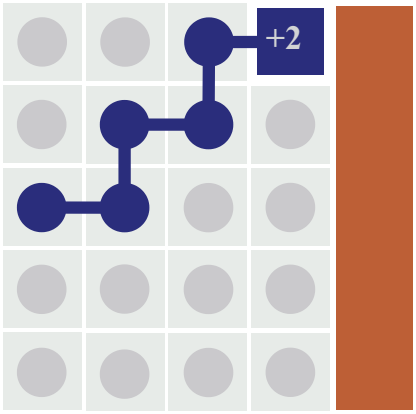
Introduction

In the last few years, reinforcement learning (RL) has made remarkable progress, including beating world-champion Go players, controlling robotic hands, and even painting pictures.

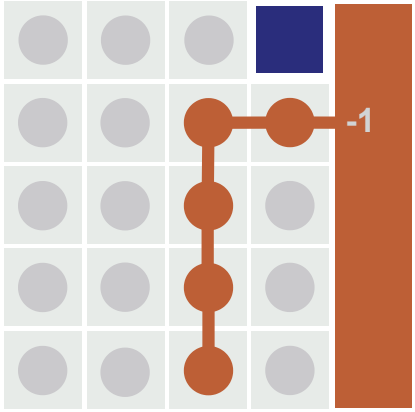
One of the key sub-problems of RL is value estimation – learning the long-term consequences of being in a state. This can be tricky because future returns are generally noisy, affected by many things other than the present state. The further we look into the future, the more this becomes true. But while difficult, estimating value is also essential to many approaches to RL. [1]

The natural way to estimate the value of a state is as the average return you observe from that state. We call this Monte Carlo value estimation.

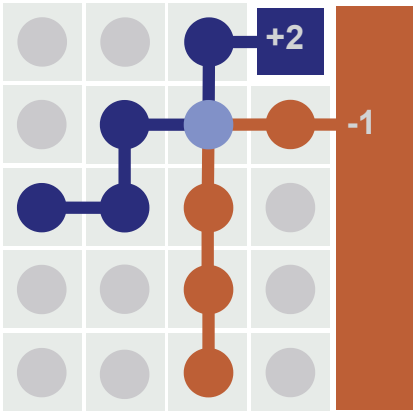
Cliff World [1] is a classic RL example, where the agent learns to walk along a cliff to reach a goal.



Sometimes the agent reaches its goal.



Other times it falls off the cliff.



Monte Carlo averages over trajectories where they intersect.

If a state is visited by only one episode, Monte Carlo says its value is the return of that episode. If multiple episodes visit a state, Monte Carlo estimates its value as the average over them.

Let's write Monte Carlo a bit more formally. In RL, we often describe algorithms with update rules, which tell us how estimates change with one more episode. We'll use an "updates toward" ($\hookleftarrow$) operator to keep equations simple. [2]

$$V(s_t) \quad \hookleftarrow \quad R_t$$

State value — Return

The term on the right is called the return and we use it to measure the amount of long-term reward an agent earns. The return is just a weighted sum of future rewards $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$ where γ is a discount factor which controls how much short term rewards are worth relative to long-term rewards. Estimating value by updating towards return makes a lot of sense. After all, the *definition* of value is expected return. It might be surprising that we can do better.

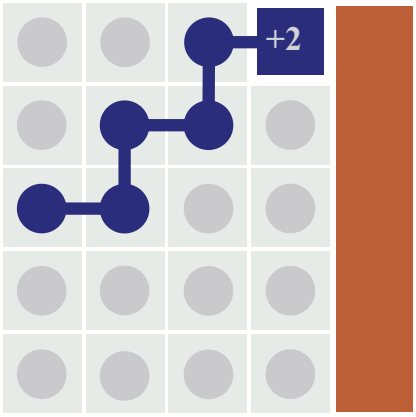
Beating Monte Carlo

But we can do better! The trick is to use a method called *Temporal Difference (TD) learning*, which bootstraps off of nearby states to make value updates.

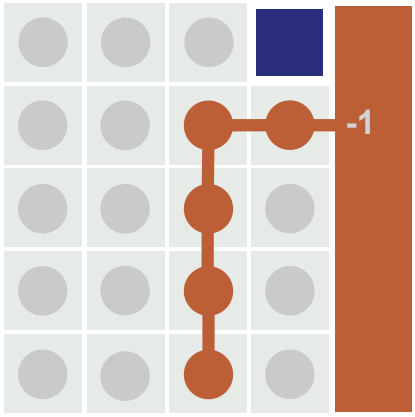
$$V(s_t) \quad \hookleftarrow \quad r_t \quad + \quad \gamma V(s_{t+1})$$

State value | Reward | Next state value

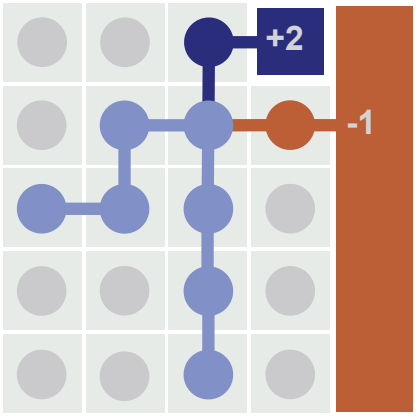
Intersections between two trajectories are handled differently under this update. Unlike Monte Carlo, TD updates merge intersections so that the return flows backwards to all preceding states.



Sometimes the agent reaches its goal.



Other times it falls off the cliff.



TD learning merges paths where they intersect.

What does it mean to "merge trajectories" in a more formal sense? Why might it be a good idea? One thing to notice is that $V(s_{t+1})$ can be written as the expectation over all of its TD updates:

$$\begin{aligned} V(s_{t+1}) \quad &\simeq \quad \mathbb{E}\left[r'_{t+1} \; + \; \gamma V(s'_{t+2})\right] \\ &\simeq \quad \mathbb{E}\left[r'_{t+1}\right] \; + \; \gamma \, \mathbb{E}\left[V(s'_{t+2})\right] \end{aligned}$$

Now we can use this equation to expand the TD update rule recursively:

$$\begin{aligned} V(s_t) \quad &\hookleftarrow \quad r_t \quad + \quad \gamma V(s_{t+1}) \\ &\hookleftarrow \quad r_t \quad + \quad \gamma \, \mathbb{E}\left[r'_{t+1}\right] \quad + \quad \gamma^2 \, \mathbb{E}\left[V(s''_{t+2})\right] \\ &\hookleftarrow \quad r_t \quad + \quad \gamma \, \mathbb{E}\left[r'_{t+1}\right] \quad + \quad \gamma^2 \, \mathbb{E}\mathbb{E}\left[r''_{t+2}\right] \quad + \quad \ldots \end{aligned}$$

This gives us a strange-looking sum of nested expectation values. At first glance, it's not clear how to compare them with the more simple-looking Monte Carlo update. More importantly, it's not clear that we *should* compare the two; the updates are so different that it feels a bit like comparing apples to oranges. Indeed, it's easy to think of Monte Carlo and TD learning as two entirely different approaches.

But they are not so different after all. Let's rewrite the Monte Carlo update in terms of reward and place it beside the expanded TD update.

MC update

$$V(s_t) \hookleftarrow \underbrace{r_t}_{\text{Reward from present path.}} + \underbrace{\gamma\, r_{t+1}}_{\text{Reward from present path.}} + \underbrace{\gamma^2\, r_{t+2}}_{\text{Reward from present path...}} + \dots$$

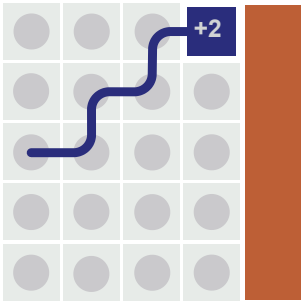
TD update

$$V(s_t) \hookleftarrow \underbrace{r_t}_{\text{Reward from present path.}} + \underbrace{\gamma\, \mathbb{E}\left[r'_{t+1}\right]}_{\text{Expectation over paths intersecting present path.}} + \underbrace{\gamma^2\, \mathbb{E}\mathbb{E}\left[r''_{t+2}\right]}_{\text{Expectation over paths intersecting } \textit{paths intersecting} \text{ present path...}} + \dots$$

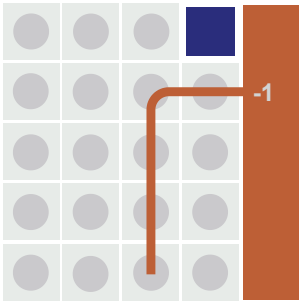
A pleasant correspondence has emerged. The difference between Monte Carlo and TD learning comes down to the nested expectation operators. It turns out that there is a nice visual interpretation for what they are doing. We call it the *paths perspective* on value learning.

The Paths Perspective

We often think about an agent's experience as a series of trajectories. The grouping is logical and easy to visualize.

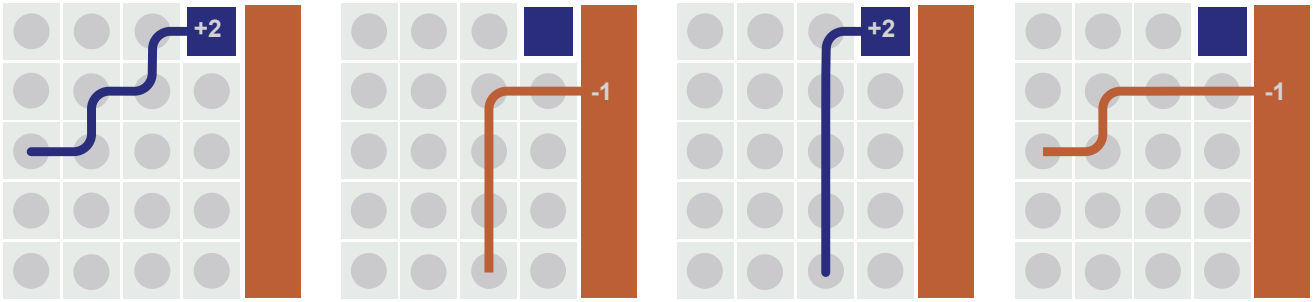


Trajectory 1



Trajectory 2

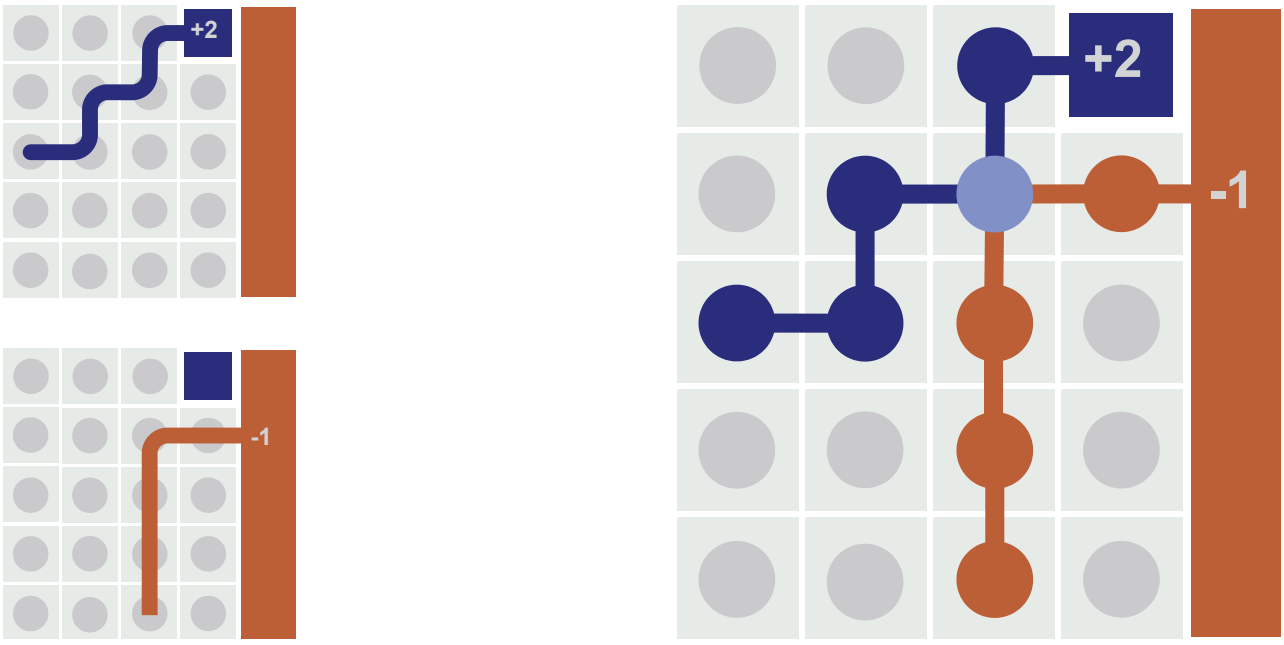
But this way of organizing experience de-emphasizes relationships *between* trajectories. Wherever two trajectories intersect, both outcomes are valid futures for the agent. So even if the agent has followed Trajectory 1 to the intersection, it could *in theory* follow Trajectory 2 from that point onward. We can dramatically expand the agent's experience using these simulated trajectories or "paths."



Path 1 Path 2 Path 3 Path 4

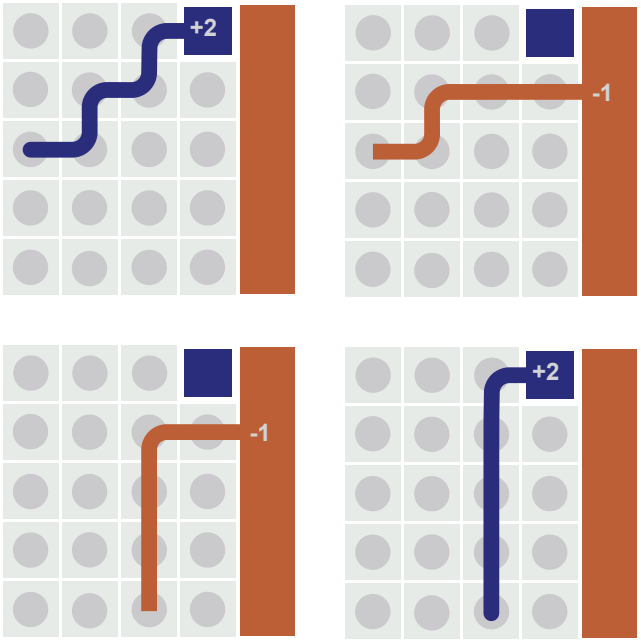
Estimating value. It turns out that Monte Carlo is averaging over real trajectories whereas TD learning is averaging over all possible paths. The nested expectation values we saw earlier correspond to the agent averaging across *all possible future paths*.

MONTE CARLO ESTIMATION

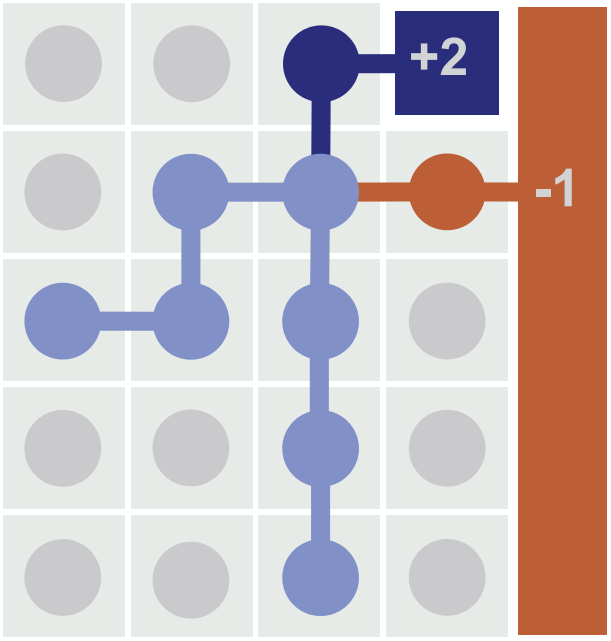


Averages over **real trajectories** Resulting MC estimate

TEMPORAL DIFFERENCE ESTIMATION



Averages over **possible paths**



Resulting TD estimate

Comparing the two. Generally speaking, the best value estimate is the one with the lowest variance. Since tabular TD and Monte Carlo are empirical averages, the method that gives the better estimate is th one that averages over more items. This raises a natural question: Which estimator averages over more items?

$$Var[V(s)] \quad \propto \quad \frac{1}{N}$$

Variance of estimate

Inverse of the number of items in the average

First off, TD learning never averages over fewer trajectories than Monte Carlo because there are never fewer simulated trajectories than real trajectories. On the other hand, when there are *more* simulated trajectories, TD learning has the chance to average over more of the agent's experience. This line of reasoning suggests that TD learning is the better estimator and helps explain why TD tends to outperfor Monte Carlo in tabular environments.

Introducing Q-functions

An alternative to the value function is the Q-function. Instead of estimating the value of a state, it estimates the value of a state and an action. The most obvious reason to use Q-functions is that they

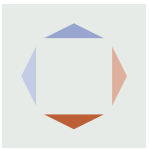
$\pi(s, a)$

Many times we'd like to compare the value of actions under a policy.

$V(s)$

It's hard to do this with a value function.

$Q(s, a)$



It's easier to use Q-functions, which estimate joint state-action values.

There are some other nice properties of Q-functions. In order to see them, let's write out the Monte Carlo and TD update rules.

Updating Q-functions. The Monte Carlo update rule looks nearly identical to the one we wrote down for $V(s)$:

$$\underbrace{Q(s_t, a_t)}_{\text{State-action value}} \hookleftarrow \underbrace{R_t}_{\text{Return}}$$

We still update towards the return. Instead of updating towards the return of being in some state, though, we update towards the return of being in some state *and* selecting some action.

Now let's try doing the same thing with the TD update:

$$\underbrace{Q(s_t, a_t)}_{\text{State-action value}} \hookleftarrow \underbrace{r_t}_{\text{Reward}} + \underbrace{\gamma Q(s_{t+1}, a_{t+1})}_{\text{Next state value}}$$

This version of the TD update rule requires a tuple of the form $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, so we call it the *Sarsa* algorithm. Sarsa may be the simplest way to write this TD update, but it's not the most efficient. The problem with Sarsa is that it uses $Q(s_{t+1}, a_{t+1})$ for the next state value when it really should be using $V(s_{t+1})$. What we need is a better estimate of $V(s_{t+1})$.

$$\underbrace{Q(s_t, a_t)}_{\text{State-action value}} \hookleftarrow \underbrace{r_t}_{\text{Reward}} + \underbrace{\gamma V(s_{t+1})}_{\text{Next state value}}$$

$$V(s_{t+1}) = ?$$

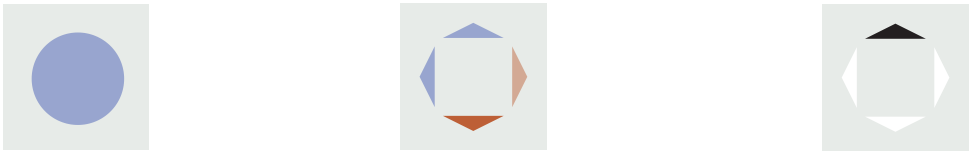
There are many ways to recover $V(s_{t+1})$ from Q-functions. In the next section, we'll take a close look at four of them.

Learning Q-functions with reweighted paths

Expected Sarsa. A better way of estimating the next state's value is with a weighted sum [3] over its Q-values. We call this approach Expected Sarsa:

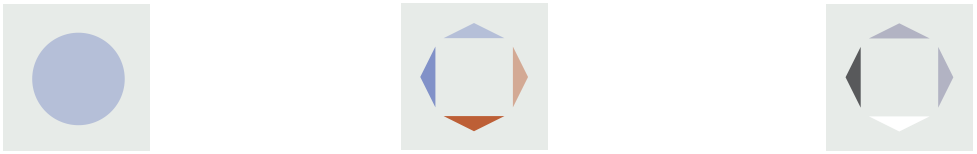
Sarsa uses the Q-value associated with a_{t+1} to estimate the next state's value.

$$V(s_{t+1}) = Q(s_{t+1}, a) \cdot a_{t+1}$$



Expected Sarsa uses an expectation over Q-values to estimate the next state's value.

$$V(s_{t+1}) = Q(s_{t+1}, a) \cdot \pi(s_{t+1}, a)$$

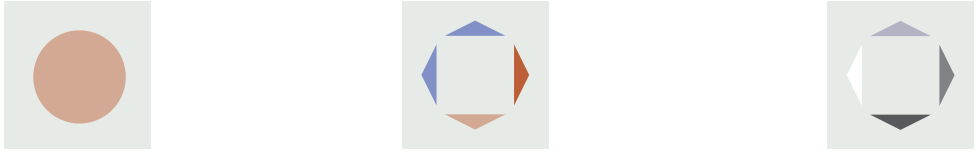


Here's a surprising fact about Expected Sarsa: the value estimate it gives is often *better* than a value estimate computed straight from the experience. This is because the expectation value weights the Q-values by the true policy distribution rather than the empirical policy distribution. In doing this, Expected Sarsa *corrects for the difference between the empirical policy distribution and the true policy distribution.*

Off-policy value learning. We can push this idea even further. Instead of weighting Q-values by the true policy distribution, we can weight them by an arbitrary policy, π^{off}:

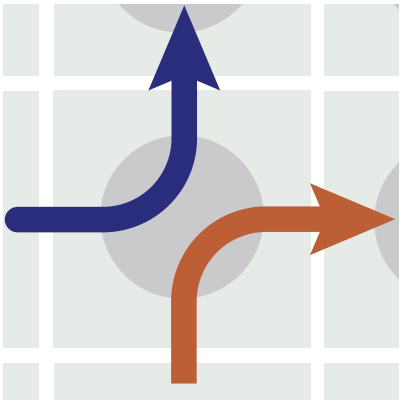
Off-policy value learning weights Q-values by an arbitrary policy.

$$V^{\pi^{off}}(s_{t+1}) = Q^{\pi^{off}}(s_{t+1}, a) \cdot \pi^{off}(s_{t+1}, a)$$

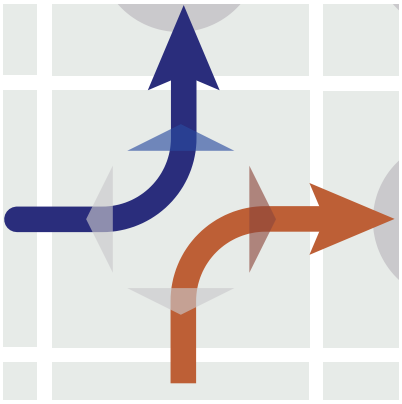


This slight modification lets us estimate value under any policy we like. It's interesting to think about Expected Sarsa as a special case of off-policy learning that's used for on-policy estimation.

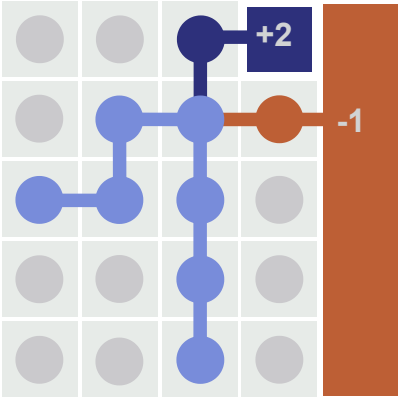
Re-weighting path intersections. What does the paths perspective say about off-policy learning? To answer this question, let's consider some state where multiple paths of experience intersect.



Multiple paths of experience intersect at this state.



Paths that exit the state through different actions get associated with different Q-values.



Weight of upward path: 0.50

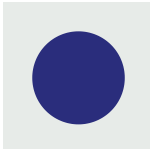
Whenever we re-weight Q-values, we also re-weight the paths that pass through them.

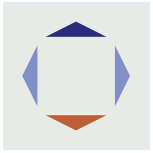
Wherever intersecting paths are re-weighted, the paths that are most representative of the off-policy distribution end up making larger contributions to the value estimate. Meanwhile, paths that have low probability make smaller contributions.

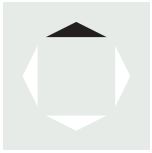
Q-learning. There are many cases where an agent needs to collect experience under a sub-optimal policy (e.g. to improve exploration) while estimating value under an optimal one. In these cases, we use a version of off-policy learning called Q-learning.

Q-learning estimates value under the optimal policy by choosing the max Q-value.

$$V^{\pi^*}(s_{t+1}) \quad = \quad Q^{\pi^*}(s_{t+1}, a) \quad \cdot \quad \underset{a}{\operatorname{argmax}}\, Q^{\pi^*}(s_{t+1}, a)$$







Q-learning prunes away all but the highest-valued paths. The paths that remain are the paths that the agent will follow at test time; they are the only ones it needs to pay attention to. This sort of value learning often leads to faster convergence than on-policy methods [4].

Double Q-Learning. The problem with Q-learning is that it gives biased value estimates. More specifically, it is over-optimistic in the presence of noisy rewards. Here's an example where Q-learning fails:

You go to a casino and play a hundred slot machines. It's your lucky day: you hit the jackpot on machine 43. Now, if you use Q-learning to estimate the value of being in the casino, you will choose the best outcome over the actions of playing slot machines. You'll end up thinking that the value of the casino is the value of the jackpot…and decide that the casino is a great place to be!

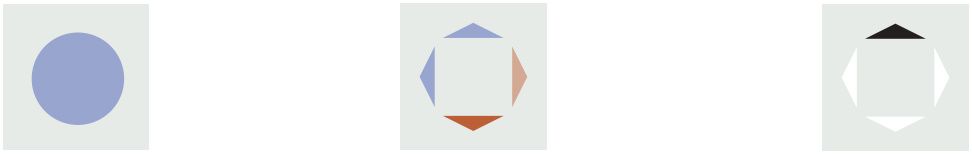
Sometimes the largest Q-value of a state is large *just by chance*; choosing it over others makes the value estimate biased. One way to reduce this bias is to have a friend visit the casino and play the same set of slot machines. Then, ask them what their winnings were at machine 43 and use their response as your value estimate. It's not likely that you both won the jackpot on the same machine, so this time you won't end up with an over-optimistic estimate. We call this approach *Double Q-learning*. [2]

Putting it together. It's easy to think of Sarsa, Expected Sarsa, Q-learning, and Double Q-learning as different algorithms. But as we've seen, they are simply different ways of estimating $V(s_{t+1})$ in a TD update.

ON-POLICY METHODS

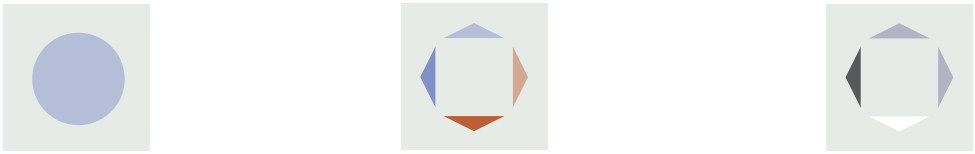
Sarsa uses the Q-value associated with a_{t+1} to estimate the next state's value.

$$V(s_{t+1}) = Q(s_{t+1}, a) \cdot a_{t+1}$$



Expected Sarsa uses an expectation over Q-values to estimate the next state's value.

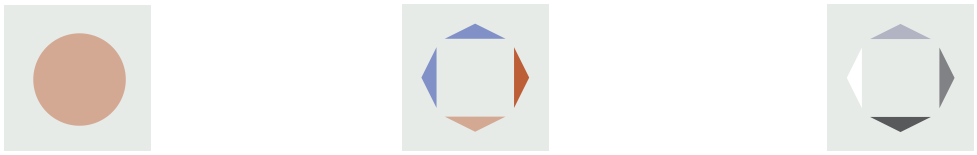
$$V(s_{t+1}) = Q(s_{t+1}, a) \cdot \pi(s_{t+1}, a)$$



OFF-POLICY METHODS

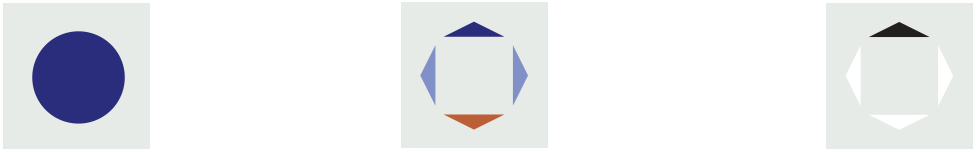
Off-policy value learning weights Q-values by an arbitrary policy.

$$V^{\pi^{off}}(s_{t+1}) = Q^{\pi^{off}}(s_{t+1}, a) \cdot \pi^{off}(s_{t+1}, a)$$



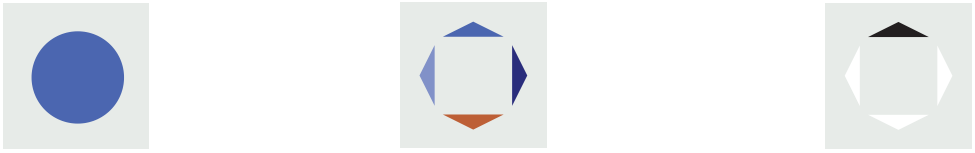
Q-learning estimates value under the optimal policy by choosing the max Q-value.

$$V^{\pi^*}(s_{t+1}) = Q^{\pi^*}(s_{t+1}, a) \cdot \underset{a}{\operatorname{argmax}}\, Q^{\pi^*}(s_{t+1}, a)$$



Double Q-learning selects the best action with Q_A and then estimates the value of that action with Q_B.

$$V_B^{\pi^*}(s_{t+1}) = Q_B^{\pi^*}(s_{t+1}, a) \cdot \underset{a}{\operatorname{argmax}}\, Q_A^{\pi^*}(s_{t+1}, a)$$

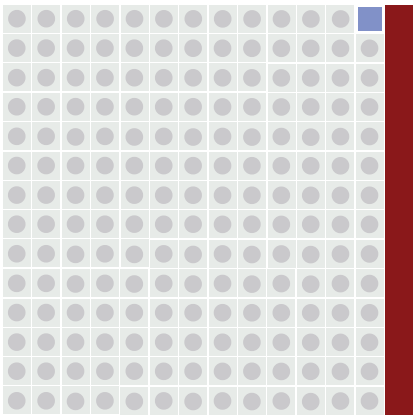


The intuition behind all of these approaches is that they re-weight path intersections.

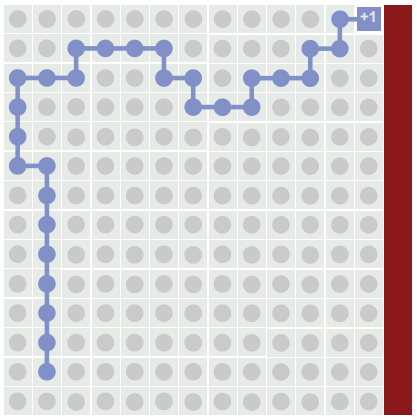
Re-weighting paths with Monte Carlo. At this point, a natural question is: Could we accomplish the same re-weighting effect with Monte Carlo? We could, but it would be messier and involve re-weighting all of the agent's experience. By working at intersections, TD learning re-weights individual transitions instead of episodes as a whole. This makes TD methods much more convenient for off-policy learning.

Merging Paths with Function Approximators

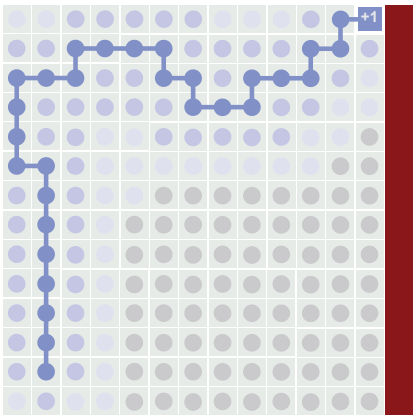
Up until now, we've learned one parameter — the value estimate — for every state or every state-action pair. This works well for the Cliff World example because it has a small number of states. But most interesting RL problems have a large or infinite number of states. This makes it hard to store value estimates for each state.



Large or infinite state spaces are a characteristic of many interesting RL problems. Value estimation in these spaces often requires function approximation.



Tabular value functions keep value estimates for each individual state. They consume a great deal of memory and don't generalize.

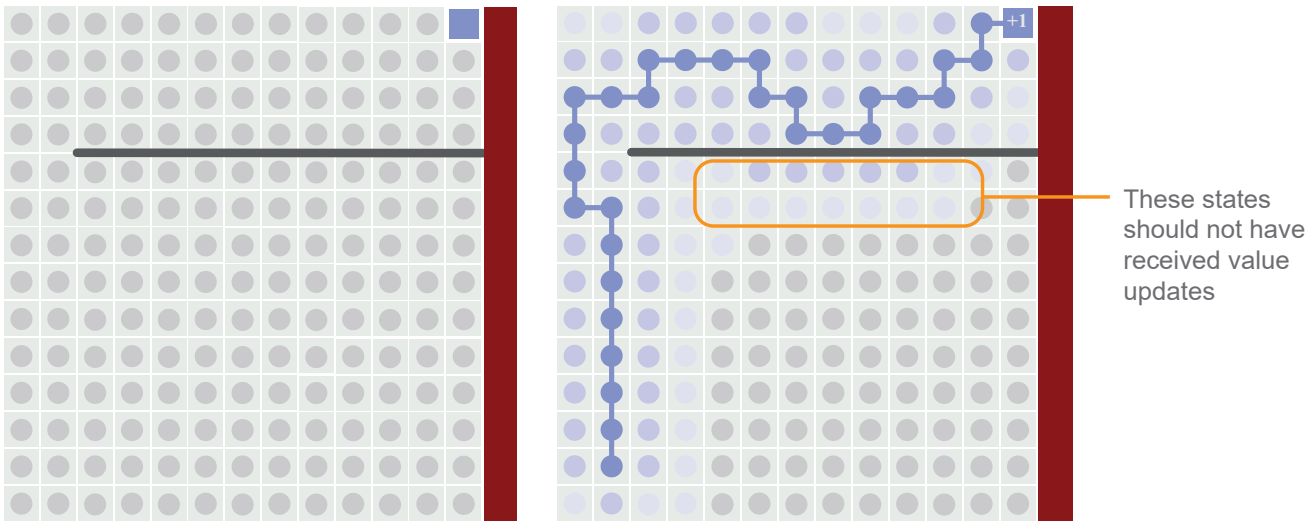


Euclidean averagers – which are a type of function approximator – save memory and let agents generalize to states they haven't visited yet.

Instead, we must force our value estimator to have fewer parameters than there are states. We can do this with machine learning methods such as linear regression, decision trees, or neural networks. All of these methods fall under the umbrella of function approximation.

Merging nearby paths. From the paths perspective, we can interpret function approximation as a way of merging nearby paths. But what do we mean by "nearby"? In the figure above, we made an implicit decision to measure "nearby" with Euclidean distance. This was a good idea because the Euclidean distance between two states is highly correlated with the probability that the agent will transition between them.

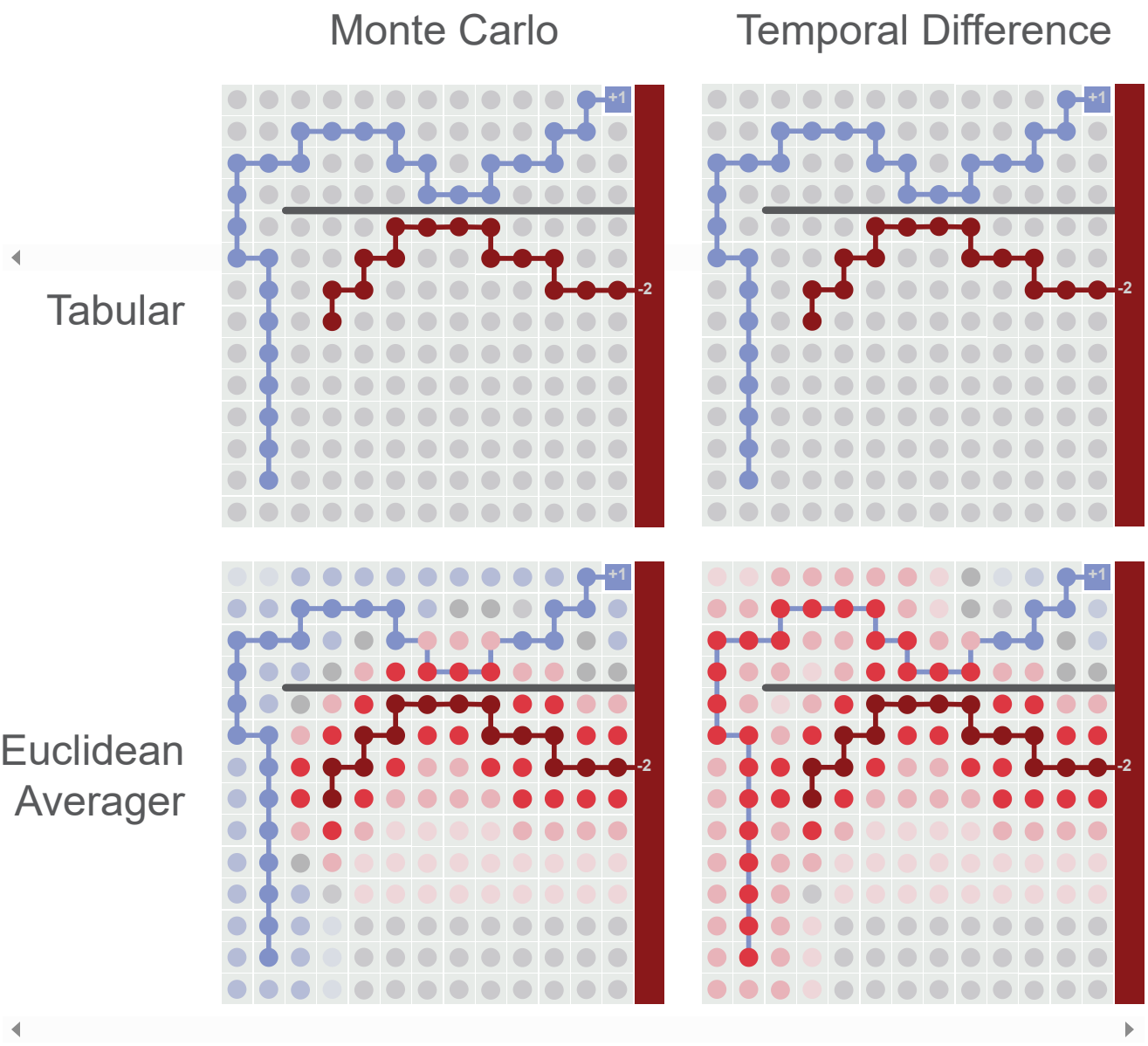
However, it's easy to imagine cases where this implicit assumption breaks down. By adding a single long barrier, we can construct a case where the Euclidean distance metric leads to bad generalization. The problem is that we have merged the wrong paths.



Imagine changing the Cliff World setup by adding a long barrier.

Now, using the Euclidean averager leads to bad value updates.

Merging the wrong paths. The diagram below shows the effects of merging the wrong paths a bit more explicitly. Since the Euclidean averager is to blame for poor generalization, both Monte Carlo and TD make bad value updates. However, TD learning amplifies these errors dramatically whereas Monte Carlo does not.



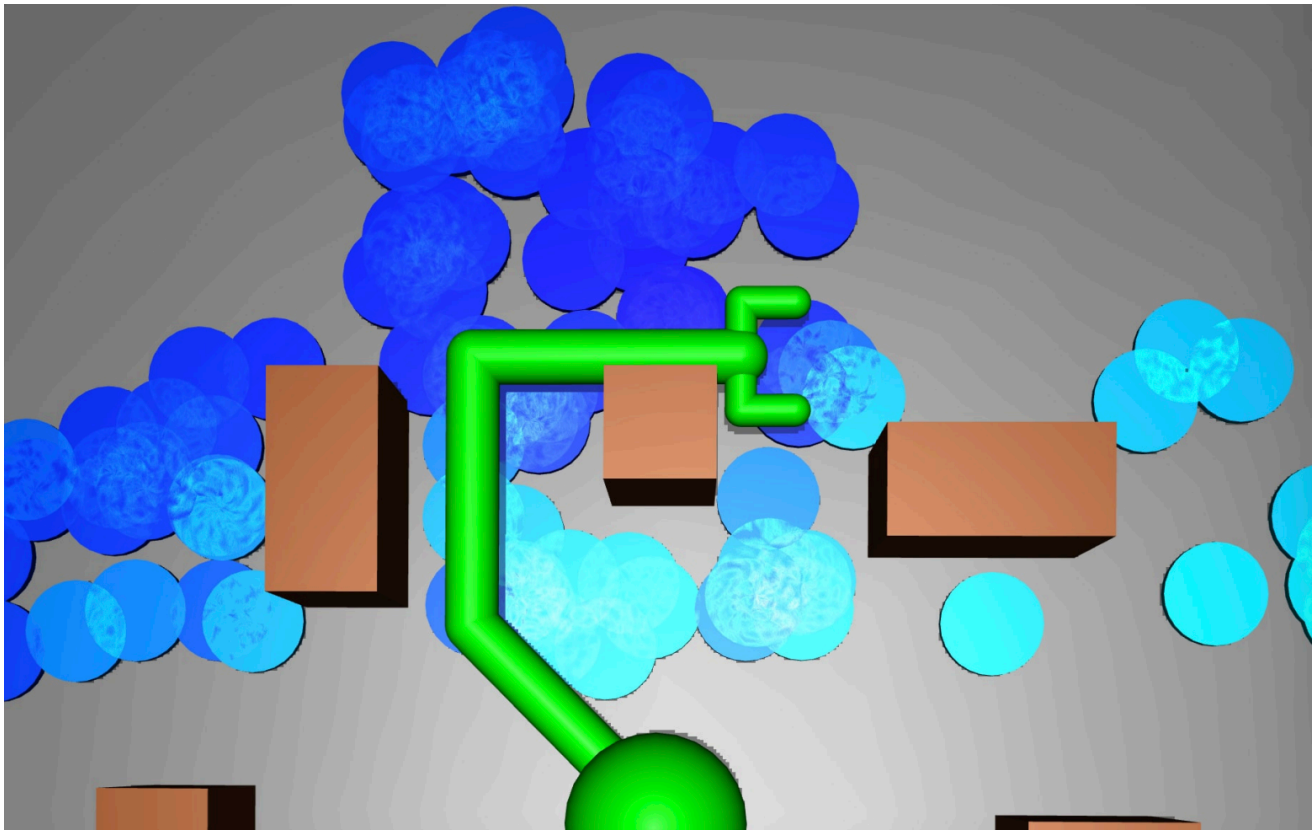
We've seen that TD learning makes more efficient value updates. The price we pay is that these updates end up being much more sensitive to bad generalization.

Implications for deep reinforcement learning

Neural networks. Deep neural networks are perhaps the most popular function approximators for reinforcement learning. These models are exciting for many reasons, but one particularly nice property is that they don't make implicit assumptions about which states are "nearby."

Early in training, neural networks, like averagers, tend to merge the wrong paths of experience. In the Cliff Walking example, an untrained neural network might make the same bad value updates as the Euclidean averager.

But as training progresses, neural networks can actually learn to overcome these errors. They learn which states are "nearby" from experience. In the Cliff World example, we might expect a fully-trained neural network to have learned that value updates to states *above* the barrier should never affect the values of states *below* the barrier. This isn't something that most other function approximators can do. It's one of the reasons deep RL is so interesting!



A distance metric learned by a neural network [3]. **Lighter blue → more distant**. The agent, which was trained to grasp objects using the robotic arm, takes into account obstacles and arm length when it measures the distance between two states.

TD or not TD? So far, we've seen how TD learning can outperform Monte Carlo by merging paths of experience where they intersect. We've also seen that merging paths is a double-edged sword: when function approximation causes bad value updates, TD can end up doing worse.

Over the last few decades, most work in RL has preferred TD learning to Monte Carlo. Indeed, many approaches to RL use TD-style value updates. With that being said, there are many other ways to use Monte Carlo for reinforcement learning. Our discussion centers around Monte Carlo for value estimation in this article, but it can also be used for policy selection as in Silver et al. [4]

Since Monte Carlo and TD learning both have desirable properties, why not try building a value estimator that is a mixture of the two? That's the reasoning behind TD(λ) learning. It's a technique that simply interpolates (using the coefficient λ) between Monte Carlo and TD updates [5]. Often, TD(λ) works better than either Monte Carlo or TD learning alone [6].

Conclusion

In this article we introduced a new way to think about TD learning. It helps us see why TD learning can be beneficial, why it can be effective for off-policy learning, and why there can be challenges in combining TD learning with function approximators.

We encourage you to use the playground below to build on these intuitions, or to try an experiment of your own.

GRIDWORLD PLAYGROUND

Learning Algorithm

Monte Carlo
Sarsa
Expected Sarsa
Q-Learning

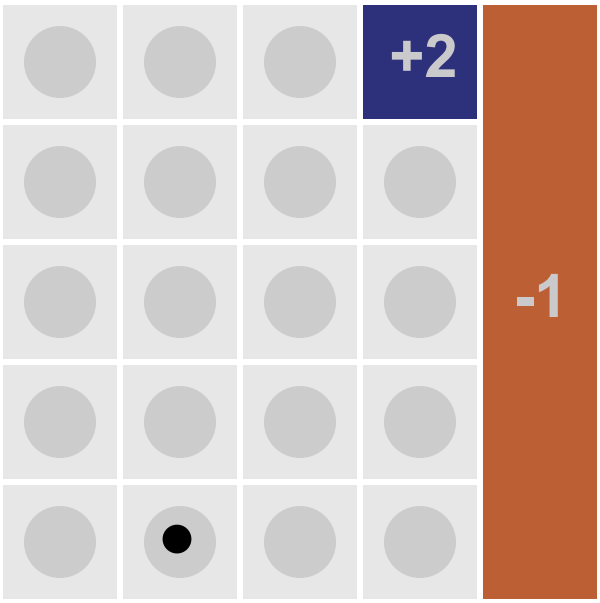
Visualization

Policy
Q(s,a)
V(s)

Epsilon-greedy policy

explore exploit

Add an agent



Acknowledgments

We are grateful to Ludwig Schubert, Justin Gilmer, Shan Carter, and John Schulman for feedback on early drafts of this article. Shan also gave invaluable design advice on diagrams and interactive components. Thanks to Cassandra Xia for helping develop the playground demo.

Sam would like to thank the Google AI Residency Program for offering continuous support and guidance as he explored this line of research.

Author Contributions

Concepts: Chris introduced the central concept and structure of this article. Working together, Sam and Chris fleshed out the finer points.

Writing & Diagrams: The text and figures were initially drafted by Sam and then refined under Chris's guidance. The code used for the hero and playground visualizations was written by Chris and Cassandra and modified by Sam to fit the Distill format.

Footnotes

1. For many approaches (policy-value iteration), estimating value essentially is the whole problem, while in other approaches (actor-critic models), value estimation is essential for reducing noise. [↩]

2. In tabular settings such as the Cliff World example, this "update towards" operator computes a running average. More specifically, the n^{th} Monte Carlo update is $V(s_t) = V(s_{t-1}) + \frac{1}{n}\left[R_n - V(s_t)\right]$ and we could just as easily use the "+=" notation. But when using parameteric function approximators such as neural networks, our "update towards" operator may represent a gradient step, which cannot be written in "+=" notation. In order to keep our notation clean and general, we chose to use the $\hookleftarrow$ operator throughout. [↩]

3. Also written as an expectation value, hence "Expected Sarsa". [↩]

4. Try using the Playground at the end of this article to compare between approaches. [↩]

5. In the limit $\lambda = 0$, we recover the TD update rule. Meanwhile, when $\lambda = 1$, we recover Monte Carlo. [↩]

6. Researchers often keep the λ coefficient constant as they train a deep RL model. However, if we think Monte Carlo learning is best early in training (before the agent has learned a good state representation) and TD learning is best later on (when it's easier to benefit from merging paths), maybe the best approach is to anneal λ over the course of training. [↩]

References

1. **Reinforcement Learning: An Introduction** [HTML]
   Sutton, R. and Barto, A., 2017. MIT Press.

2. **Double Q-learning** [PDF]
   Van Hasselt, H., 2010. Advances in Neural Information Processing Systems.

3. **Universal Planning Networks** [PDF]
   Srinivas, A., Jabri, A., Abbeel, P., Levine, S. and Finn, C., 2018. Proceedings of Machine Learning Research.

4. **Mastering the game of go without human knowledge**
   Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and others,, 2017. Nature, Vol 550(7676), pp. 354. Nature Publishing Group.

Updates and Corrections

If you see mistakes or want to suggest changes, please create an issue on GitHub.

Reuse

Citation

For attribution in academic contexts, please cite this work as

```
Greydanus & Olah, "The Paths Perspective on Value Learning", Distill, 2019.
```

BibTeX citation

```
@article{greydanus2019the,
  author = {Greydanus, Sam and Olah, Chris},
  title = {The Paths Perspective on Value Learning},
  journal = {Distill},
  year = {2019},
  note = {https://distill.pub/2019/paths-perspective-on-value-learning},
  doi = {10.23915/distill.00020}
}
```

Distill is dedicated to clear explanations of machine learning

About Submit Prize Archive RSS GitHub Twitter ISSN 2476-0757