Wouter van Heeswijk, PhD Follow

Dec 9, 2021 · 5 min read · · Listen

Implement Value Iteration in Python — A Minimal Working Example

Master the simple and classical Dynamic Programming algorithm to find optimal solutions for Markov Decision Process models

In the age of artificial intelligence, exact algorithms are not exactly *hot*. If a machine cannot learn it by itself, what's the point? Why bother solving **Markov Decision Process** models with solutions that do not scale anyway? Why not dive straight into reinforcement learning algorithms?

It still pays off to learn the classical **dynamic programming** techniques. First of all, they are still widely used in practice. Many software developer jobs incorporate dynamic programming as part of their interview process. Although there are only so many states and actions you can enumerate, you may be surprised of the real-world problems that can still be resolved to optimality.

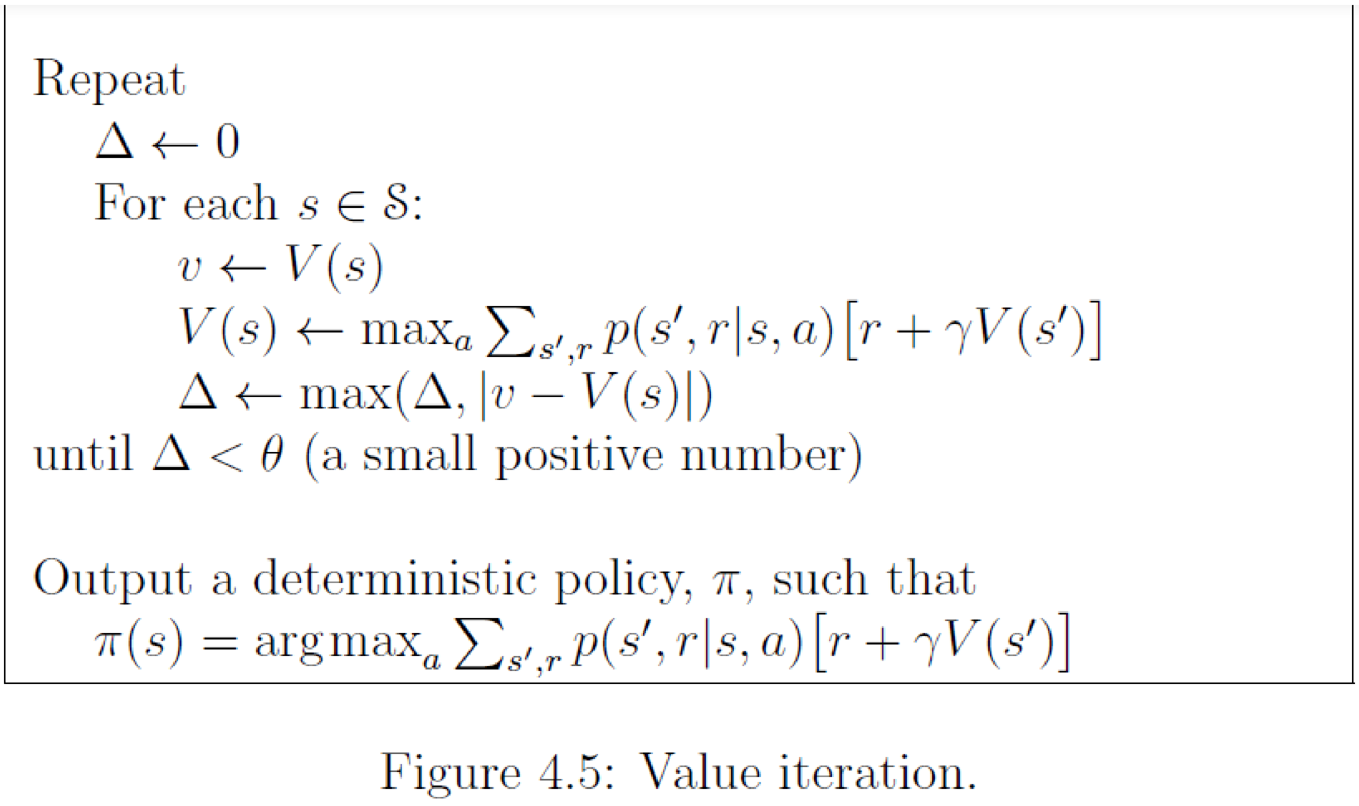
Second, even if only interested in reinforcement earning, many algorithms in that domain are firmly rooted in dynamic programming. Four policy classes may be distinguished in reinforcement learning, one of them being **value function approximation**. Before moving to such approaches, having an understanding of the classical value iteration algorithm is essential. Fortunately, it happens to be outlined in this very article.

The Four Policy Classes of Reinforcement Learning

towardsdatascience.com

Value iteration

The elegance of the value iteration algorithm is something to be admired. It really just takes a few lines of mathematical expressions, and not many more lines of code. Let's check the seminal denotation by Sutton & Barto:



```
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Figure 4.5: Value iteration.

Value iteration algorithm [source: Sutton & Barto (publicly available), 2019]

The intuition is fairly straightforward. First, you **initialize** a value for each state, for instance at 0.

Then, for every state you **compute the value** `V(s)` , by multiplying the reward for each action `a` (direct reward `r` + downstream value `V(s')`) with the transition probability `p` .

Suppose we indeed initialized at 0 and direct rewards `r` are non-zero. The discrepancies will directly become visible in the **difference** expression `|v-V(s)|` , where `v` is the old estimate and `V(s)` the new one. So, the error `Δ` will exceed the threshold `θ` (a small value) and a new iteration follows.

By performing sufficiently many iterations, the algorithm will **converge** to a point where `|v-V(s)|<θ` for every state. You can then resolve the `argmax` to find the optimal action for each state; knowing the true value functions `V(s)` equates having the it is good to keep in mind the optimality conditions.

A minimal working example

With the mathematics out of the way, let's continue with the coding example. To keep the focus on the algorithm, the problem is extremely simple.

The problem

Consider a one-dimensional world (a row of tiles), with a single terminating state. Hitting the terminating state yields a reward of +10, every other action costs -1. The agent can move left or right, but — to not make it *too* trivial —the agent moves into the wrong direction 10% of the time. It's a very simple problem, but it has (i) a direct reward, (ii) an expected downstream reward, and (iii) transition probabilities.

The algorithm

The Python algorithm stays pretty close to the mathematical outline provided by Sutton & Barto, so no need to expand too much. The full code fits in a single Gist:

```python
# Initialize Markov Decision Process model
actions = (0, 1)  # actions (0=left, 1=right)
states = (0, 1, 2, 3, 4)  # states (tiles)
rewards = [-1, -1, 10, -1, -1]  # Direct rewards per state
gamma = 0.9  # discount factor
# Transition probabilities per state-action pair
probs = [
    [[0.9, 0.1], [0.1, 0.9], [0, 0], [0, 0], [0, 0]],
    [[0.9, 0.1], [0, 0], [0.1, 0.9], [0, 0], [0, 0]],
    [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]],  # Terminating state (all probs 0)
    [[0, 0], [0, 0], [0.9, 0.1], [0, 0], [0.1, 0.9]],
    [[0, 0], [0, 0], [0, 0], [0.9, 0.1], [0.1, 0.9]],
]

# Set value iteration parameters
max_iter = 10000  # Maximum number of iterations
delta = 1e-400  # Error tolerance
V = [0, 0, 0, 0, 0]  # Initialize values
```

```
        max_diff = 0  # Initialize max difference
        V_new = [0, 0, 0, 0, 0]  # Initialize values
        for s in states:
            max_val = 0
            for a in actions:

                # Compute state value
                val = rewards[s]  # Get direct reward
                for s_next in states:
                    val += probs[s][s_next][a] * (
                        gamma * V[s_next]
                    )  # Add discounted downstream values

                # Store value best action so far
                max_val = max(max_val, val)

                # Update best policy
                if V[s] < val:
                    pi[s] = actions[a]  # Store action with highest value

            V_new[s] = max_val  # Update value with highest value

            # Update maximum difference
            max_diff = max(max_diff, abs(V[s] - V_new[s]))

        # Update value functions
        V = V_new

        # If diff smaller than threshold delta for all states, algorithm terminates
        if max_diff < delta:
            break
```

value_iteration.py hosted with ❤ by GitHub view raw

Some experiments

Ok, some experiments then. We start by showing two iterations of the algorithm in full detail.

$$V(0) = \max_{a \in A} \sum_{s',r} p(s',r \mid s,a)[r(s,a) + V(s')]$$

```
=max((0.9*[-1+0]+          # State 1 (action 'Left')
0.1*[-1+0]+                # State 2 (action 'Left')
0.0*[-1+0]+                # State 3 (action 'Left')
0.0*[-1+0]+                # State 4 (action 'Left')
0.0*[-1+0];                # State 5 (action 'Left')
0.1*[-1+0]+                # State 1 (action 'Right')
0.9*[-1+0]+                # State 2 (action 'Right')
0.0*[-1+0]+                # State 3 (action 'Right')
0.0*[-1+0]+                # State 4 (action 'Right')
0.0*[-1+0])                # State 5 (action 'Right')
= max(-1;-1) = -1.         Δ = |−1 − 0| = 1
```

Value iteration step 1, state 0 [image by author]

That seems like a lot of computational effort for such a small problem. Indeed, it's easy to see why dynamic programming does not scale well. In this case, all values `V(s)` are still at 0 — as we just started — so the estimated state value `V(0)` is simply the direct reward -1.

Let's try one more, somewhat further along the line. Same computation, but now the values `V(s)` have been updated throughout several iterations. We now have `V=[5.17859, 7.52759, 10.0, 7.52759, 5.17859]`. Again, we plug in the values:

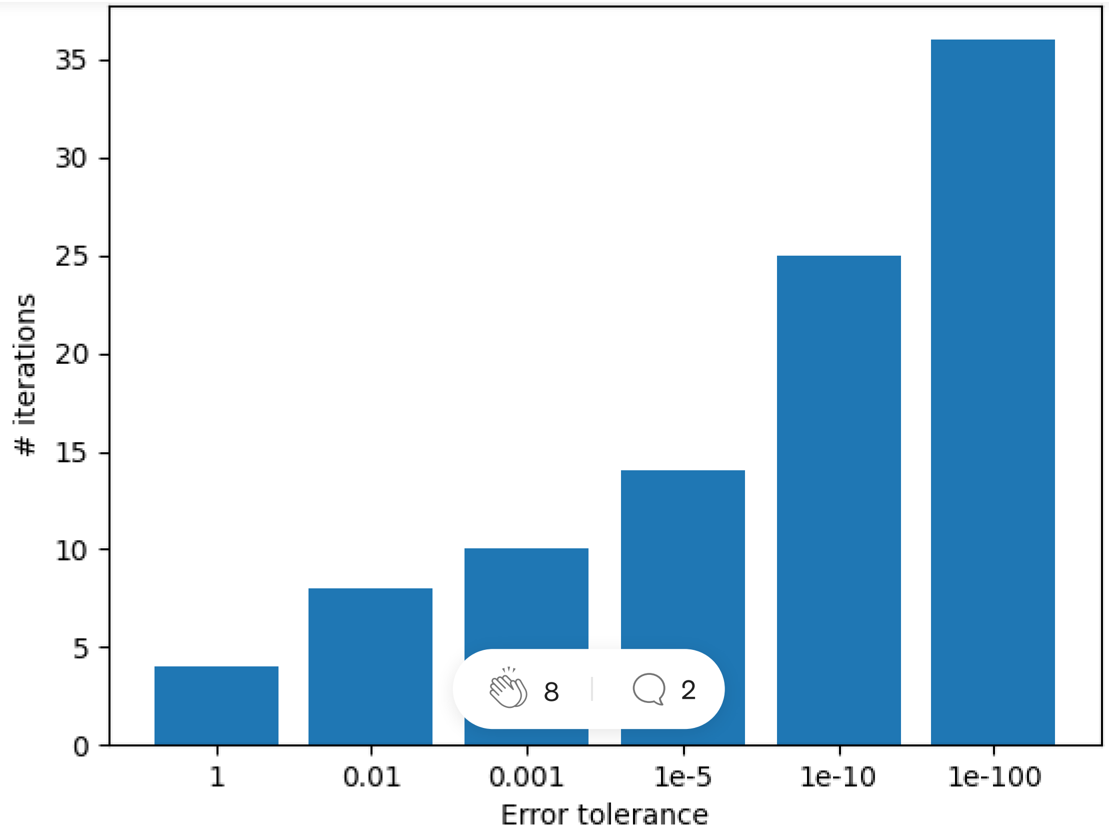
$$a \in A \sum_{s',r}$$

$$\begin{array}{ll}
=\max((0.9*[-1+5.18]+ & \text{# State 1 (action 'Left')} \\
0.1*[-1+7.53]+ & \text{# State 2 (action 'Left')} \\
0.0*[-1+10.0]+ & \text{# State 3 (action 'Left')} \\
0.0*[-1+7.53]+ & \text{# State 4 (action 'Left')} \\
0.0*[-1+5.18]; & \text{# State 5 (action 'Left')} \\
0.1*[-1+5.18]+ & \text{# State 1 (action 'Right')} \\
0.9*[-1+7.53]+ & \text{# State 2 (action 'Right')} \\
0.0*[-1+10.0]+ & \text{# State 3 (action 'Right')} \\
0.0*[-1+7.53]+ & \text{# State 4 (action 'Right')} \\
0.0*[-1+5.18]) & \text{# State 5 (action 'Right')} \\
= \max(3.87;5.56) = 5.56. & \Delta = |5.56 - 5.18| = 0.38
\end{array}$$

Value iteration step 4, state 0 [image by author]

Thus, we update `V(0)` from 5.18 to 5.56. The error would be `Δ=(5.56-5.18)=0.38`. In turn, this will affect the updates of the other states, and this process continues until `Δ<θ` for all states. For state 0, the optimal value is 5.68, hit within 10 iterations.

The most interesting parameter to test here is the error tolerance `θ`, which influences the numbers of iterations before convergence.



Number of iterations required for various error tolerances θ. The lower the tolerance, the more iterations are needed before the algorithm converges.

Final words

Value iteration is one of the cornerstones for Reinforcement Learning. It is easy to implement and understand. Before moving towards more advanced implementations, make sure to grasp this fundamental algorithm.

Some other minimal working examples you might be interested in:

Implement Policy Iteration in Python — A Minimal Working Example

Learn about this classical Dynamic Programming algorithm to optimally solve Markov Decision Process models

A Minimal Working Example for Deep Q-Learning in TensorFlow 2.0

A multi-armed bandit example to train a Q-network. The update procedure takes just a few lines of code using TensorFlow

towardsdatascience.com

A Minimal Working Example for Continuous Policy Gradients in TensorFlow 2.0

A simple example for training Gaussian actor networks. Defining a custom loss function and applying the GradientTape…

towardsdatascience.com

A Minimal Working Example for Discrete Policy Gradients in TensorFlow 2.0

A multi-armed bandit example for training discrete actor networks. With the aid of the GradientTape functionality, the…

towardsdatascience.com

References

Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.