Lecture 11: Deep Reinforcement Learning

Emile van Krieken
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

THIS LECTURE

1. (Deep) Q-Learning
2. Actor-Critic methods
3. Model-based RL (World Models)

VU

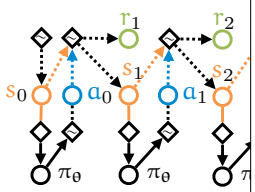
RECAP

REINFORCE:

1. $\boldsymbol{\tau} \sim p(\boldsymbol{\tau}|\theta)$
2. $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$

Discounted reward to-go

$$G_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$$



VU

NOTATION UPDATE

- Declutter notation:
  - Current timestep t:
    $r_t = r, a_t = a, s_t = s, G_t = G$
  - Next timestep $t + 1$:
    $r_{t+1} = r', a_{t+1} = a', s_{t+1} = s', G_{t+1} = G'$

A problem with RL is that the notation can become very cluttered and hard to understand. To prevent information overflow in the next slides, we will redefine some symbols.

For variables like the reward, action, state and expected reward to-go, we just use the symbol without the timestep to refer to the variable at timestep t.

We do the same for the 'next' timestep, t+1, but adding the prime character (') after the symbol to distinguish it from the current state.

Q-FUNCTION

Discounted reward to-go

$$G = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$$

Q-function (state-action value function):

$$Q^{\pi}(s, a) = \mathbb{E}_{p(\tau|\pi,s,a)}[G]$$

First, we will introduce two very important notions in RL: The Q-function. These are defined in terms of the discounted reward to-go that we not-coincidentally used in REINFORCE.

We will start by the harder one: The Q-function. It is also known as the state-action value function, for good reasons! The Q-function is the *expected* discounted reward to-go, after executing action a in state s, **and** afterwards following the policy \pi (!!!!!). I cannot stress this enough, so read it again: After the action is executed, the Q-function expects the agent to follow the policy \pi. It will come as no surprise that we will learn Q-functions in deep Q-learning :)

OPTIMALITY

Optimal Q function: For all states s and actions a:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

Given $Q^*(s, a)$, the **optimal policy** is

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q^*(s, a') \\ 0 & \text{otherwise} \end{cases}$$

Optimal policy is deterministic!

Next, we will introduce what the **optimal policy** in Reinforcement Learning is. This is the best policy that we can perform! It is a policy such that, for all states, the value function is maximized.

What does this policy look like? By expanding the value function, we see that it is an expectation of the Q-function over the optimal policy.

Now, we note that the optimal policy is necessarily deterministic. Why does it need to be deterministic? Note that the probability distribution of an expectation needs to sum to 1. Clearly, it is best to put all probability mass on the action that maximizes the Q-function. That is a deterministic policy!

In other words, the optimal policy is the policy that has a probability of 1 for (or all probability mass on) the action maximizing the Q-function.

RECURSIVE REWARD TO GO

$$
\begin{aligned}
G &= r' + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots + \gamma^{T-2-t} r_T) \\
&= r' + \gamma \sum_{t'=t+1}^{T-1} \gamma^{t'-t} r_{t'+1} \\
&= r' + \gamma G'
\end{aligned}
$$

Next, we rewrite the discounted reward to go. Simply write out the sum over discounted rewards. Leave one discount factor out. The sum inside the braces is exactly equal to G_{t+1}!

OPTIMAL Q-FUNCTION

$$Q^*(s,a) = \max_\pi \mathbb{E}_{p(\tau|\pi,s,a)}[G] = \mathbb{E}_{p(\tau|\pi^*,s,a)}[r' + \gamma G'] = Q^{\pi^*}(s,a)$$

Expand expectation:

$$= \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \mathbb{E}_{p(\tau|\pi^*,s')}[G']]$$

$$= \mathbb{E}_{p(s',r'|s,a)}\left[r' + \gamma \mathbb{E}_{\pi^*(a'|s')}\left[\mathbb{E}_{p(\tau|\pi^*,s',a')}[G']\right]\right]$$

That's the Q-function!

$$= \mathbb{E}_{p(s',r'|s,a)}\left[r' + \gamma \mathbb{E}_{\pi^*(a'|s')}[Q^*(s',a')]\right]$$

We will now look how to make the definition of the Q-function recursive, that is, it can be computed using itself.

We just found a recursive definition for G, so plug that in. The resulting expectation is over a sum over the next reward r' and the rest of the discounted reward G'. Also, we replace the maximum over policies by the optimal policy. After all, we cannot do better than that! Conveniently, the resulting expression is the definition of the Q-function of the optimal policy: The Q-function of the optimal policy IS the optimal Q-function :)

Next, note that the next reward is only dependent on the current state s and action a, and not on the rest of the trajectory. So, we can split the expectation in two: First we sample the next state and reward, and then the rest of the trajectory.

The first thing that happens in the trajectory then is sampling the next action a'. Of course, this happens from the optimal policy \pi^*. So, expand this expectation.

Now, the most inner expectation is exactly the Q function at the next timestep! So let's plug that in.

We now have a recursive definition of the Q-function! But we can simplify it even more.

OPTIMAL Q-FUNCTION

$$Q^*(s,a) = \mathbb{E}_{p(s',r'|s,a)}\left[r' + \gamma \mathbb{E}_{\pi^*(a'|s')}[Q^*(s',a')]\right]$$

recall:

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q^*(s,a') \\ 0 & \text{otherwise} \end{cases}$$

so:

$$Q^*(s,a) = \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \max_{a'} Q^*(s',a')]$$

No explicit policy in this equation!

Back to that recursive formulation of the optimal Q-function. Recall the optimal policy \pi^*. This is a deterministic policy, that simply chooses the action that maximizes the optimal Q-function!

This means we can simplify the recursive formulation even further: It chooses the maximizing action a', so we don't need that expectation over actions. Simply take the maximum of the optimal Q-function here!

This gives the so-called **Bellman optimality equation**: A way to define the optimal Q-function without any reference to an external policy.

OPTIMAL Q-FUNCTION

The **optimal Q-function**:

$$Q^*(s,a) = \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \max_{a'} Q^*(s',a')]$$

Idea: Estimate $Q^*(s,a)$ with NN $Q_\theta(s,a)$!

Policy:

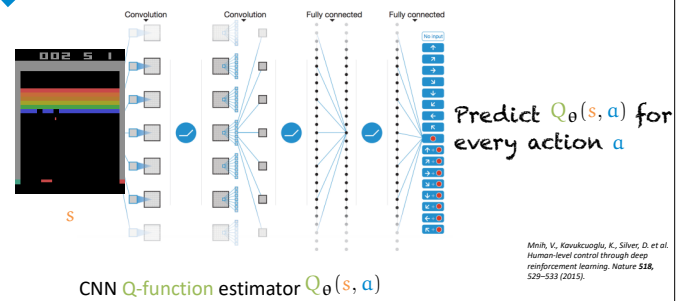
$$\pi_\theta(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q_\theta(s,a') \\ 0 & \text{otherwise} \end{cases}$$

VU

We found that the optimal Q-function satisfies this very nice recursion. So what exactly does it say? The optimal Q-function has that it is equal to the expected next reward, plus the discounted optimal Q-function in the next state.

We want to find a Q-function for which this equation holds. Why? From this Q-function we can extract the optimal policy, which takes the action that maximizes the Q-function.

ATARI Q-FUNCTION ESTIMATOR



Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015).

CNN Q-function estimator $Q_\theta(s,a)$

VU

What does a neural network that estimates Q-functions look like? Just like for policies, the neural network usually actually only receives the state, and not the action.

The output layer is important here: It is simply linear! For each action, it outputs a scalar value, which predicts the Q-function. This way, we don't need to run the neural network for each action: We run it once and immediately get Q-function estimations for each action.

BOOTSTRAP ERROR

$$Q^*(s,a) = \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \max_{a'} Q^*(s',a')]$$

So:

$$Q^*(s,a) - \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \max_{a'} Q^*(s',a')] = 0$$

Minimize **bootstrapped** error using regression:

$$\arg\min_\theta \left(Q_\theta(s,a) - \mathbb{E}_{p(s',r'|s,a)}[r' + \gamma \max_{a'} Q_\theta(s',a')] \right)^2$$

prediction — target

VU

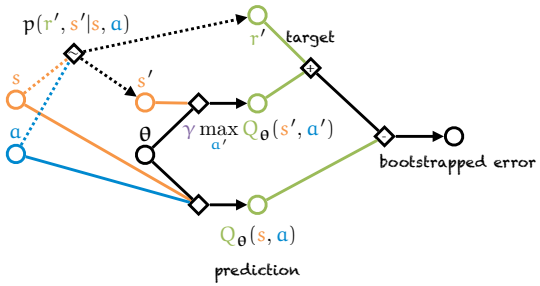
Again, recall the recursive definition of the optimal Q-function. Clearly, subtracting the right-hand side, this definition can be seen as the fact that the difference between the Q-function at the current state minus the second term (expected next reward + discounted Q-function) should be 0.

We want our Q-function estimate Q_\theta to approximate the optimal Q-function. So, it should also make sure that this equality holds!

The **bootstrapped** error is the difference between our estimate of the Q-function at the current state minus the expected next reward and the maximum of the *estimated* discounted Q-function. Why is it called bootstrapped? Because the error uses the Q-function estimator *itself* as a target!

In Deep-Q learning, we minimize the squared bootstrapped error, and use regression techniques to optimize this.

Q-LEARNING



VU

Here we see the computation graph of Q-learning. Compare it closely with the equation on the previous slide to see if you can follow the computation. Note that the same set of parameters is used both to compute the prediction and the target.

DEEP Q-LEARNING

Start in state s

Deep Q-Learning (v1):

1. $a \leftarrow \arg\max_{a'} Q_\theta(s, a')$ — Choose action
2. $s', r' \sim p(s', r'|s, a)$ — Execute action
3. $\theta \leftarrow \theta - \alpha \nabla_\theta \left(Q_\theta(s, a) - r' - \gamma \max_{a'} Q_\theta(s', a') \right)^2$ — Update parameters
4. $s \leftarrow s'$

VU

Let's look at our first version of the Deep Q-learning algorithm. This is an **online** algorithm, which means that we update our policy (in this case, Q-value parameters) immediately after executing an action, instead of waiting until the trajectory is over.

The algorithm is pretty simple:

1. Choose an action. This happens by taking the action that maximizes the estimated Q-value.
2. Execute the action in the environment. We transition to the next state, and receive a reward, just like before.
3. Update the parameters using the bootstrapped error equation. This is a simple gradient descent step.

Repeat this!

EPSILON GREEDY

Deterministic action selection

$a \leftarrow \arg\max_{a'} Q_\theta(s, a')$

No **exploration**, only **exploitation**!

- Always go to same restaurant
- Never find anything better!

Solution: **epsilon greedy**

Take random action with ϵ probability

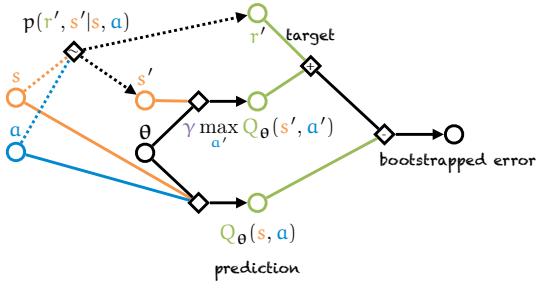
VU

Unfortunately, this is a very, very poor algorithm. One of the reasons is that our action selection algorithm is deterministic: We always execute the same action in a state (namely, the one maximizing the Q-value).

Analogy: Say we want to figure out what the best restaurant in town is. With this way of action selection, we'd take the first restaurant that seems nice, **exploit** that knowledge, and then always visit that restaurant without ever **exploring** any alternatives. Obviously, this leads to sub-optimal behaviour as there's probably a much better restaurant in town! This problem is called the **exploration - exploitation tradeoff.**

The most common solution to this problem is **epsilon greedy**. Instead of always taking the best action, we simply take a random action with epsilon probability.

MOVING TARGET

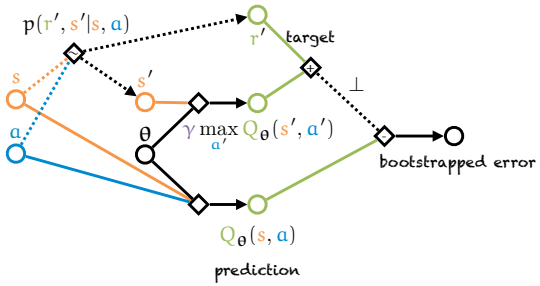


VU

Another issue is the following. Let's see what happens in our backwards pass. How are the parameters theta updated? As you can see, it happens both through the prediction, and *also* through the target Q-value prediction! This means that by minimizing the bootstrapped error equation, we change both the prediction and the target. If that seems wrong, from a machine learning perspective, then you're most certainly right.

In fact, if you'd try this out, it'd be very hard to get it to work, as there is no clearly defined target for the regression problem.

SIMPLE SOLUTION



VU

A very simple, though somewhat hacky, solution, is to just remove the gradient from the target, again using that weird bot symbol we saw before during the first RL lecture. This already helps quite a bit.

DEEP Q-LEARNING

Deep Q-Learning (v2):

1. $a = \begin{cases} \text{random action} & \text{with } \epsilon \text{ probability} \\ \arg\max_{a'} Q_\theta(s, a') & \text{otherwise.} \end{cases}$ Epsilon-greedy
2. $s', r' \sim p(s', r'|s, a)$
3. $\theta \leftarrow \theta - \alpha \nabla_\theta \left(Q_\theta(s, a) - \boxed{\perp}(r' + \gamma \max_{a'} Q_\theta(s', a')) \right)^2$
4. $s \leftarrow s'$

VU

Time to look at the second version of our Deep Q-learning algorithm. Not much has changed, though.

In the first line, we change how we select actions, from the "greedy" always take the best action, to epsilon-greedy, that takes a random action with epsilon probability.

We also added a bot symbol to show we don't put gradients throughout the target of the bellman error.

DEEP Q-LEARNING

Deep Q-learning requires many tricks to get working

- Experience replay
- Target networks
- Double Q-learning
- Multi-step returns
- Gradient clipping
- …

In assignment 5c, you can explore some of these for bonus points!

VU

Unfortunately, Deep Q-learning requires quite a lot of tricks to get working.

The most important one is experience replay. It solves two problems: states are sequentially corelated, and so do not satisfy the identically and independently distributed (iid) assumption that machine learning requires! Experience replay instead saves some states to 'replay' later to train the Q-function. It separates data collection and training the Q-function on it.

There are much more tricks to explore, but we won't get into that here. In assignment 5c, you can play with these!

SUMMARY: DEEP Q-LEARNING

Deep Q-learning

- Estimate optimal Q-function
- Minimize bootstrapped error

Requires many tricks to get working

Next: Combine ideas in Actor-critic

VU

ACTOR-CRITIC

VU

RECAP

Policy gradient:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

REINFORCE:

Estimate with Monte Carlo rollout:

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

High variance :(

VU

Last lecture, we talked about policy gradient methods, in particular about REINFORCE. The problem with this algorithm is that it has very high variance. This means that the method is very sample inefficient: It will take a very long time to converge, and requires a lot of data. In this part of the lecture, we will use methods inspired by Deep Q-learning to reduce the variance of policy gradient methods.

BASELINES

REINFORCE:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Reduce variance with **baseline** b_t:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t (G_t - b_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

baseline

VU

A simple important modification is baselines. A baseline is a value that is subtracted from the reward. This is still the unbiased policy gradient! In the next slide, we'll see why.

BASELINES

$$\mathbb{E}_{\pi_\theta(a_t|s_t)}[b_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$
$$= \sum_{a_t} b_t \cancel{\pi_\theta(a_t|s_t)} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\cancel{\pi_\theta(a_t|s_t)}}$$

VU

So why is subtracting that baseline unbiased? Consider the expectation of the baseline multiplied by the grad-log of the policy.

Write out the expectation and the derivative, just like in the last lecture when we derived the score function.

BASELINES

$$\mathbb{E}_{\pi_\theta(a_t|s_t)}[b_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$
$$= \sum_{a_t} b_t \cancel{\pi_\theta(a_t|s_t)} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\cancel{\pi_\theta(a_t|s_t)}}$$
$$= \sum_{a_t} b_t \nabla_\theta \pi_\theta(a_t|s_t) = b_t \nabla_\theta \sum_{a_t} \pi_\theta(a_t|s_t)$$
$$= b_t \nabla_\theta 1 = 0$$

VU

Since the baseline doesn't depend on the action, we can move it out of the summation. (and also the gradient, because it's linear).

Now, we have a sum over probabilities with all possible actions. A probability distribution should sum to 1! And the gradient of 1 is 0.

So, the expected value of the baseline multiplied with the grad-log policy is 0, so we can safely subtract it from the Q-function.

WHAT BASELINE?

REINFORCE with **baseline**:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t (G_t - b_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

baseline ?

Value function:

$$V^\pi(s_t) = \mathbb{E}_{p(\tau|\pi,s_t)}[G_t] = \mathbb{E}_{\pi(a_t|s_t)}[Q^\pi(s_t, a_t)]$$

Value function baseline:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t (G_t - V^\pi(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

VU

So what value should this baseline be? Usually, we take the **value function**, or **state value function** (compare with the Q-function, which is also called the **state-action value function**).

The value function is the expected reward-to-go from a state when following the policy. It is related to the Q-function as follows: It's the expected Q-function when sampling an action from the policy.

REINFORCE VS ACTOR-CRITIC

Act, receive reward.

How to reinforce?

$V^\pi(s_t) = 0.63$

REINFORCE:

I won!

Random reward

REINFORCE + baseline:

I won. That result is 37% better than expected!

Increase of random reward wrt expected reward

VU

So this reduces the variance, but why?

Consider an agent trying to learn chess. Training chess with REINFORCE would do the following: Perform an action, then see if in the end we win the game. If so, update based on that: It was a good action! Otherwise, if we lost, we say it was a bad action. But many things could have happened after taking that action, and it could also be very likely that we would lose after taking it.

With that baseline, it's a bit different: again, we reinforce actions that lead to winning. But this time, we subtract that by our current believe how likely it is we would win from that state. If we did not expect to win, then it turns out that the sequence of actions lead to an unlikely victory! These actions should be reinforced much more than if we were in a state that was already looking like it was in a winning position, and then indeed winning.

TRAINING VALUE FUNCTION

Like Deep Q-Learning, train neural network V_ϕ with regression.

1. Use rollouts:

$$\phi \leftarrow \phi - \alpha \sum_{t=0}^{T-1} (V_\phi(s_t) - G_t)^2$$

target: reward-to-go

2. Use bootstrapping (lower variance, biased):

$$\phi \leftarrow \phi - \alpha \sum_{t=0}^{T-1} (V_\phi(s_t) - \bot(r_{t+1} + \gamma V_\phi(s_{t+1})))^2$$

target: bootstrapped expected reward-to-go

VU

Cool! But we still need to be able to compute that baseline. We'll do this just like in Deep Q-Learning: We'll use a neural network that's going to estimate the value function.

There are two options: We can use the rewards-to-go from that state, just like we'd use for REINFORCE. That's a high-variance target though.

Instead, we usually use bootstrapping errors, just like in Deep Q-Learning: The target is the next reward plus the value function in the next state.

RECAP

REINFORCE with baseline:

1. $\tau \sim p(\tau|\theta)$
2. $\phi \leftarrow \phi - \alpha_c \sum_{t=0}^{T-1} (V_\phi(s_t) - \bot(r_{t+1} + \gamma V_\phi(s_t)))^2$
3. $\theta \leftarrow \theta + \alpha_a \sum_{t=0}^{T-1} \gamma^t (G_t - V_\phi(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$

VU

Q-FUNCTIONS IN POLICY GRADIENTS

Recall $Q^\pi(s, a) = \mathbb{E}_{p(\tau|\pi,s,a)}[G]$

Policy gradient:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

$$= \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t Q^\pi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

critic actor

Much lower variance!

VU

Recall how we defined Q-functions: The *expected* reward-to-go G_t from a state after performing an action (and then continuing to use the corresponding policy).

The reward-to-go G_t is the reward that is used to update our policy in REINFORCE… Within an expectation too!

Therefore, we can actually replace G_t by the Q-function. This gives us an **actor-critic** algorithm, where the policy is an actor, and the Q-function is the critic: It 'criticizes' taking an action.

This has much lower variance! Why? (next slide!)

REINFORCE VS ACTOR-CRITIC

Act, receive reward.

How to reinforce?

REINFORCE:

I won!

Random reward

Actor-critic:

I think I'll win with 0.63 probability!

Expected reward

VU

What's the motivation behind actor-critic?

Consider an agent trying to learn chess. Training chess with REINFORCE would do the following: Perform an action, then see if in the end we win the game. If so, update based on that: It was a good action! Otherwise, if we lost, we say it was a bad action. But many things could have happened after taking that action, and it could also be very likely that we would lose after taking it.

With actor-critic, we instead update an action based on the probability that we will win after taking that action is high. This probability isn't like in REINFORCE (randomly winning or losing), so it is no longer a source of variance!

BASELINES

Actor-critic:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t Q^\pi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Reduce variance even more with value function **baseline**:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t (\underbrace{Q^\pi(s_t, a_t) - V^\pi(s_t)}_{\text{advantage}}) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

actor

Advantage actor-critic

VU

So that's already a very useful modification to the policy gradient (sidenote: Both REINFORCE and actor-critic are called policy-gradient algorithms because (recall last lecture) they estimate the gradient of the expected reward!)

A second important modification is the addition of baselines. A baseline is a value that is subtracted from the critic. This is still the unbiased policy gradient! In the next slide, we'll see why.

ADVANTAGE VS Q-FUNCTION

Act, receive reward.

How to reinforce?

Actor-critic:

I think I'll win with 63% probability!

Expected reward

Advantage actor-critic:

I think I'll be 3% *more likely* to win.

Expected increase in reward

VU

Now compare advantage actor-critic with normal actor-critic. In advantage actor-critic, we don't reinforce based on expected reward, but based on something a bit different:

The difference between expected reward in the state and the expected reward after taking an action! For example, in chess, we reinforce actions of which we think it'll make us 3% *more* likely to win.

COMPUTING THE ADVANTAGE

Advantage actor-critic:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t (Q^\pi(s_t, a_t) - V^\pi(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

advantage A^π

Estimate V^π with V_ϕ (biased)

Estimate $Q^\pi(s_t, a_t)$ with $r_{t+1} + \gamma V_\phi(s_{t+1})$

$$A_\phi(s_t, r_{t+1}, s_{t+1}) = r_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

VU

So how do we compute the advantage in practice? We can use the value function estimation that we introduced when discussing baselines.

We can actually use this estimation to estimate the Value function baseline and the Q-function, by expanding the Q-function. To estimate the Q-function, we look what the next state is and add the reward in the next state to the discounted value function estimate of that state.

It should be noted that this gives biased estimates. This is because we only *estimate* the value function, and can definitely make mistakes in its predictions. The policy gradient is only equal to actor critic if Q is exactly the Q-function of the policy, not for other functions that estimate it! In that sense, actor-critic trades off bias for variance. This is different from REINFORCE with value function baselines, which is always unbiased!

In estimating the Q-function, there's additional bias by using the value function in this way, as it doesn't take the expectation over actions.

SUMMARY

Advantage actor-critic:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] \approx \sum_{t=0}^{T-1} \gamma^t A_\phi(s_t, r_{t+1}, s_{t+1}) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

where

$$A_\phi(s_t, r_t, s_{t+1}) = r_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

critic baseline

Trade off variance (=sample efficiency) for bias

VU

BATCH-MODE ALGORITHM

Advantage actor-critic:

1. $\tau \sim p(\tau|\theta)$
2. $\phi \leftarrow \phi - \alpha_c \sum_{t=0}^{T-1} (V_\phi(s_t) - \perp(r_{t+1} + \gamma V_\phi(s_t)))^2$ Update critic
3. $\theta \leftarrow \theta + \alpha_a \sum_{t=0}^{T-1} \gamma^t A_\phi(s_t, r_{t+1}, s_{t+1}) \nabla_\theta \log \pi_\theta(a_t|s_t)$ Update actor

VU

We will see versions of advantage actor-critic algorithm, starting with the batch-mode algorithm.

Here, in a loop, we sample a trajectory from the MDP, then update the parameters.

First, we update the critic. We just have to train the value function here, as the Q-function is expressed in terms of the value function! Training it happens just like in REINFORCE with a baseline using bootstrapping.

Then we update the actor based on the advantage function.

ONLINE ACTOR-CRITIC

Online Actor-Critic:

1. $a \sim \pi_\theta(a|s)$ — Select actions according to policy
2. $s', r' \sim p(s', r'|s, a)$
3. $\Phi \leftarrow \Phi - \alpha_c(V_\Phi(s) - \bot(r' + \gamma V_\Phi(s')))^2$ — Update critic
4. $\theta \leftarrow \theta + \alpha_a A_\Phi(s, r', s')\nabla_\theta \log \pi_\theta(a|s)$ — Update actor
5. $s \leftarrow s'$

VU

This algorithm has a second version, an 'online-mode' version. This happens just like in the algorithm of Deep Q-learning. Namely, that don't sample a complete trajectory, but update each time after taking an action.

It is not much harder: In a loop, we sample an action from the policy and execute it in the environment. Note that we don't need to do something like epsilon-greedy here because actor-critic already uses stochastic policies!

Now, we simply use the resulting next state and reward to compute an update for a single transition. Again, first the critic, then the actor. Then we assign the next state to the current state and continue looping.

It's a detail, but the actor update in this algorithm introduces bias compared to the previous algorithm. Can you spot why? (This is not important for the exam or anything, just interesting :))

A2C

Uses only a single sample

And batching over time would give correlated minibatches

A2C: Multiple online agents synchronously act

Collect experiences at each step for minibatch.

Efficient method!

VU

VARIANCE REDUCTION

Techniques to reduce variance:

- Baselines:
  - Reinforce difference with expected value
  - Unbiased, fairly low variance
- Actor-critic:
  - Reinforce expected value
  - Biased, fairly low variance
- Advantage actor-critic:
  - Reinforce increase in expected value
  - Biased, low variance

VU

WORLD MODELS

VU

ACTOR-CRITIC

Actor-critic methods

- Model actor: Policy NN
- Model rewards: Value function NN

What about 3rd RL component: **environment**?

VU

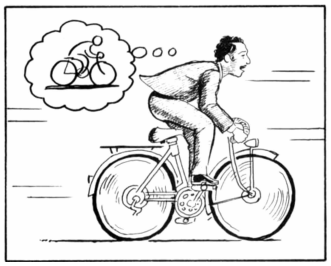
Actor-critic methods model two out of three RL components. The actor models actions through policy Networks, while rewards are modelled through the critic, in our case by estimating the value function.

So what about the 3rd RL component: states from the environment? Can those also be modelled?

WORLD MODEL

World Models

- Model **environment** using neural networks!
- Find good state representations



McCloud, Scott. Understanding Comics:The Invisible Art.Tundra Publishing, 1993.
*Ha, David, and Jürgen Schmidhuber. "World models." *

VU

The answer is yes! Modelling the environment using neural networks is often called **world models**. The goal of world models is to find representations that describe the current state well. This can be in multiple ways: It can compress and abstract information, or it can be used to represent knowledge about the world that isn't directly in the current state.

For example, if you have a robot that can look around, it only sees a part of the world. You'd also want to model what is outside of what it can see!

WORLD MODEL

World Models

- Model **environment!**

Model-based RL using Neural Networks

Use **generative model** to learn

1. how to represent states
2. to what state we transition after taking action
3. what reward we receive in state

World Models fall within the field of model-based reinforcement learning, by using neural networks to model the environment.

We use generative modelling (techniques from the last weeks lectures) to create world models.

We have multiple models:

1. A model that learns how to represent individual states
2. A model that learns how to transition from one state to another after taking an action (=transition probability)
3. A model that models the reward we receive in a state

WORLD MODEL

World Models combine many ideas from this course:

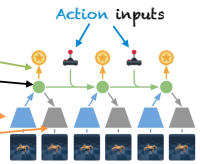
- Deep generative modelling
- Variational Auto-Encoders
- Gradient estimation
- Actor-critic

World models are also interesting because they combine a lot of ideas of this course and show well how they fit in the deep learning toolkit!

WORLD MODEL COMPONENTS

Components of our World Model:

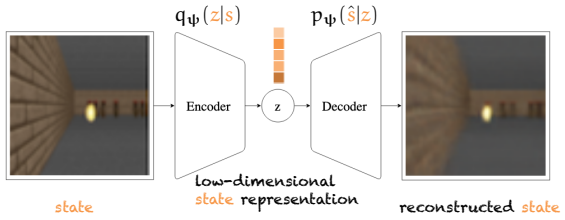
- Reward model $p_\psi(r|z)$
- Transition model $p_\psi(z'|z, a)$
- State encoder $q_\psi(z|s)$
- State decoder $p_\psi(\hat{s}|z)$



Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

Here, we summarize all components of the world model: A model for rewards, a model for state transitions, and a model for state representations (both encoding and decoding the state representations). We will start with that last one first. We use a VAE for this.

VARIATIONAL AUTO-ENCODER



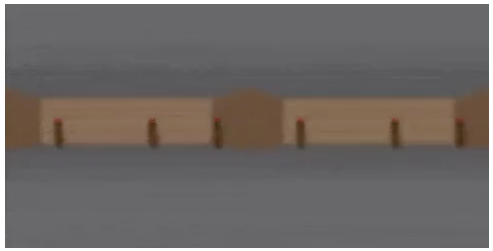
Ha, David, and Jürgen Schmidhuber. "World models."

VU

We will use a variational autoencoder (VAE) to represent states. For this, we use the latent space of VAEs. This latent space is useful, because the state representations allow reconstructing the original state. Therefore, they are a low-dimensional representation of the original state!

In this case, we'll be looking at states represented as images.

VARIATIONAL AUTO-ENCODER



real states reconstructed states

Ha, David, and Jürgen Schmidhuber. "World models." 2018

VU

Since we have a VAE, we can encode a state (=image), then decode it again to get a reconstruction. This gives a visual idea about how well we have learned to represent the states.

This simple example shows this can work pretty well!

TRAIN VISION MODEL

Training VAE to represent states:

1. Collect experience by acting in environment
2. Train VAE to reconstruct states

Goal: Learn useful representation z_t of states s_t

Compress s_t to what's essential for acting

VU

So how do we train a VAE to represent states in the environment? It's not so hard: We collect our 'training data' by just acting in the **real** environment! Then we use this training data to teach the VAE how to reconstruct states using the ELBO loss as explained in the VAE lectures.

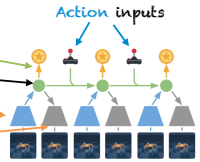
The goal is to learn useful representations of states. But how is useful defined? In the end, all we need to do is to be able to use the representation to act well in the environment.

An interesting question is whether the VAE representation is useful. It has to save a lot of information: Namely, everything needed to properly reconstruct an image! But our agent ideally gets just a representation that retrieves from the state was is essential for the agent to act. A common discussion point against using VAEs here is that the representation needs to do too much: It should both reconstruct images, and be easy to act with!

WORLD MODEL COMPONENTS

Components of our World Model:

- Reward model $p_\psi(r|z)$
- Transition model $p_\psi(z'|z, a)$
- State encoder $q_\psi(z|s)$
- State decoder $p_\psi(\hat{s}|z)$



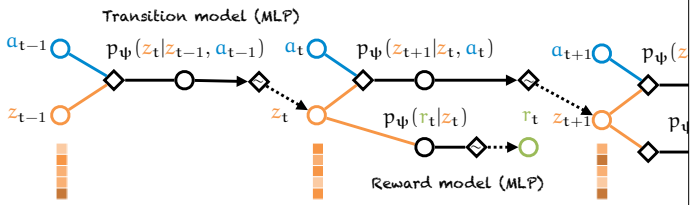
Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

We have seen how to encode and decode states to get a state representation. Now let's look at the other two models and how they are trained. It should be noted that the reward model and transition model, together with action inputs, create a markov decision process! We will see why next.

The reward model is a univariate Gaussian distribution, while the transition model is a multivariate Gaussian distribution.

TRANSITION DYNAMICS



VU

Let's look at the detailed computation graph formed using the world model. We start with some state representation in timestep t-1, in which we take an action. We use the transition model to determine the distribution over possible next states. From this distribution, we sample the next state! Note that this is the state *representation*, not the original state (for example an image). Note that the transition model is a neural network, in this case an MLP!

From this newly generated state representation, we use the reward model, also an MLP, to predict the reward we get in this state.

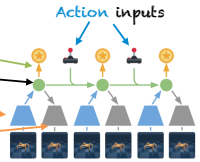
We also use the state representation, together with an action, to determine the next state transition probability. Note that here, we make no assumptions on what actions to take, or how we choose these actions!

This loop keeps repeating, just like in MDPs. In general, be sure to compare this to MDPs: World Models use almost the same probability distributions as in MDPs except that they model them using neural networks instead of using the environment itself. The only difference is that the environment generates the next state and reward together, instead of it being a separate model like in world models.

WORLD MODEL COMPONENTS

Components of our World Model:

- Reward model $p_\psi(r|z)$
- Transition model $p_\psi(z'|z, a)$
- State encoder $q_\psi(z|s)$
- State decoder $p_\psi(\hat{s}|z)$

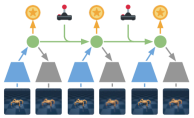


Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

TRAIN DYNAMICS MODEL

1. Collect experience by acting in environment
2. **foreach** (s, a, r', s')
   1. $z \sim q_\Psi(z|s)$ — VAE Encoder
   2. $z' \sim q_\Psi(z'|s')$
   3. Maximize $\log p_\Psi(z'|z, a)$ — Transition model
   4. Maximize $\log p_\Psi(r'|z')$ — Reward model



Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

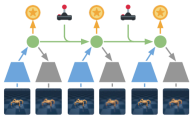
Next goal: How do we train the dynamics of the world model? This too can be done by collecting experiences in the real environment. Simply have an agent act in the real environment to collect the training data.

Experiences can be seen as a list of transitions from a state, where we take an action to get to a new state with a reward.

To train the world model, we first have to encode the states to the latent state representations z. Training the transition model is then easy: simply make sure that the transition model predicts the new state representation z' from the current state representation z. Similarly, the reward model is trained to predict the real reward from the new state representation.

DREAMING

World models can be used to **dream** trajectories
(more formally, **latent imagination**)
Use to train RL agents $\pi_\theta(a|z)$ *without interaction with environment*



Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

Now let's get to the exciting part: Dreaming! Because World Models are deep generative models, we cannot just train them, we can also generate new data! Within World Models, this is informally called 'dreaming', and more formally 'latent imagination'. Dreaming basically means imagining trajectories of states.

Why is this so useful? It allows us to train policies **without** any interaction with the environment at all. The world models create the environment to train on, which means we need way fewer interactions with the environment.

We would often like to minimize how often we interact with the environment: This might be because the environment takes a long time to run, or because it is expensive in some way to interact with the environment (for example, consider robots!).

The agents we train on the world model are trained on the state representations z instead of the states s themselves.

DREAMING

Ha, David, and Jürgen Schmidhuber. "World models."

VU

We can dream up complete segments of a video game. Here, we start off from some real state, then keep using this latent imagination to dream up a sequence. We can even use our own input here! To play the game 'in the dream', visit https://worldmodels.github.io/

TRAIN AGENTS USING WORLD MODELS

1. Choose initial state s
2. $z \sim q_\phi(z|s)$ — Encode initial state
3. **for T steps:**
   1. $a \sim \pi_\theta(a|z)$
   2. $z' \sim p_\psi(z'|z, a)$ — Transition
   3. $r' \sim p_\psi(r'|z')$ — Reward
   4. $\phi \leftarrow \phi - \alpha_c(V_\phi(z) - \bot(r' + \gamma V_\phi(z')))^2$ — Critic update
   5. $\theta \leftarrow \theta + \alpha_a A_\phi(z, r', z')\nabla_\theta \log \pi_\theta(a|z)$ — Actor update
   6. $z \leftarrow z'$

Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

We will adopt an actor-critic online-mode algorithm to train on world models. Assume we have a trained world model, how do we train the agent?

We start at an initial state, which we encode to the latent state representation z. Then we loop: use the policy to choose an action based on z.

Use the action and z to transition to the next state representation z'. Also predict the reward in that state.

Then we do an update. These are just like in normal (online-mode) advantage actor-critic, except that it use the state representations instead of the state itself.

Then we loop: Set the next state representation to the current state representation z.

COMPLETE ALGORITHM

Complete training loop:

1. Collect new experiences by acting in environment using current policy
2. Train VAE to reconstruct states
3. Train transition model
4. Train actor-critic using dreaming
5. Repeat

VU

We can now train the VAE and the transition model from experiences in the real environment, and train an agent from the resulting world models.

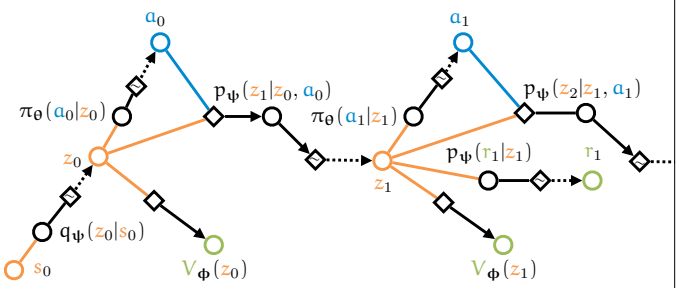
Now how does the whole training algorithm look like? From some initialization, we use the current policy to act in the environment. Note that, as the policy use state representations instead of the state itself, so we first have to encode states using the VAE encoder.

Next, we train the VAE to reconstruct states and the transition model to train the transition dynamics using the new experiences.

Finally, with the resulting world model, we train the actor-critic agent.

Importantly, we repeat this multiple times! Why? Initially, our agent is probably pretty bad, and will not be able to explore states that are hard to reach. After the agent has trained a bit, it's able to reach those new states. But now the world model has no knowledge of such states yet, and has to learn how to represent these new states!

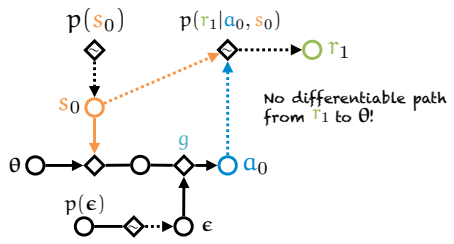
TRAINING BY DREAMING



VU

Here, we walk through the computation path when training an actor-critic agent inside a dream.

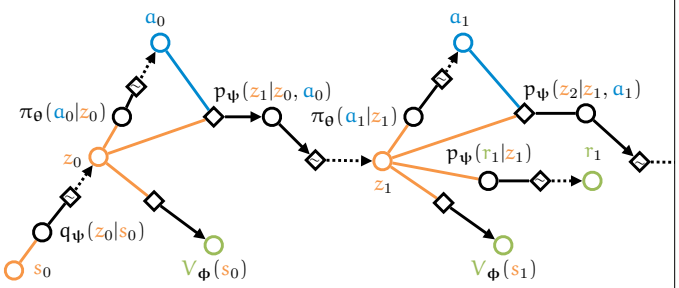
RECALL: REPARAMETERIZATION IN RL



VU

Recall the discussion on reparameterization from the last part of the previous lecture. We mentioned that we cannot use the low-variance gradient estimation trick reparameterization in RL because the environment is not differentiable. So even if we can make sampling differentiable through reparameterization, this doesn't solve the non-differentiability of the environment.

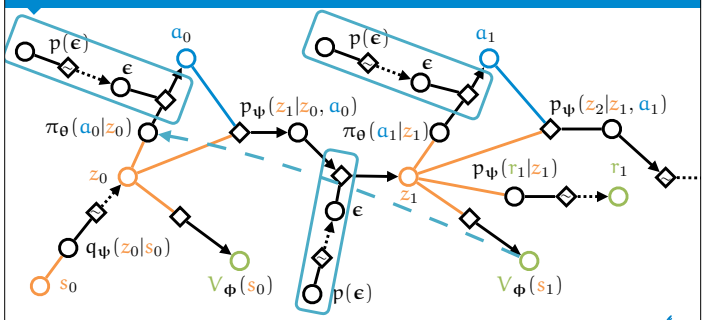
TRAINING BY DREAMING



VU

Now let's consider the computation graph when dreaming instead of running the true MDP. Notice how relatively few dotted lines there are? This is because we made the environment differentiable by modelling it with neural networks! The only non-differentiable steps are sampling, and we know just the trick to make this differentiable :)

TRAINING BY BACKPROPAGATING THROUGH DREAM



VU

Here, we modified the computation graph to use reparameterization at all sampling steps. Importantly, we can now follow the graph backward from the value functions to the parameters theta of the policy, because everything is differentiable! This means we cannot just dream our state, but also 'backpropagate' through the dream to efficiently optimize in it.

TRAIN AGENTS BY BACKPROPAGATING THROUGH WORLD MODELS

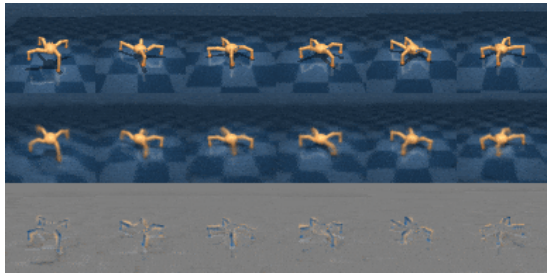
1. Choose initial state s
2. $z \sim q_\phi(z|s)$
3. **for** T **steps:**
   1. $a \sim \pi_\theta(a|z)$ — Reparameterize sample
   2. $z' \sim p_\psi(z'|z, a)$ — Reparameterize sample
   3. $r' \sim p_\psi(r'|z')$
   4. $\Phi \leftarrow \Phi - \alpha_c(V_\phi(z) - \bot(r' + \gamma V_\phi(z')))^2$
   5. $\theta \leftarrow \theta + \boxed{\alpha_a \nabla_\theta V_\phi(z')}$ — Actor backprop
   6. $z \leftarrow z'$

Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

By just ensuring we reparameterize the samples of the action and the state transitions, we can do backpropagation through the value function to train the agent! This means that the agent gets very accurate and informative gradient information, instead of the much less informative policy-gradient based methods.

DREAMER



Hafner, Danijar, et al. "Dream to Control: Learning Behaviors by Latent Imagination." International Conference on Learning Representations. 2019.

VU

WORLD MODELS

World Models

- Keep a model of environment
  - VAE represents states
  - Transition model represents dynamics
- Train agent using world models
  - "Dream" trajectories
    - No interaction with environment!
  - Backpropagate through dreams

VU