Reinforcement Learning

Aldo Faisal with contributions
by Ed Johns and Paul Bilokon

Imperial College London

October 2022 – Version 8.3

Section overview

Actor-Critic Methods

Learn, Plan, Act

Two uses of experience:

- model learning: to improve the model
- direct RL: directly improve the value function and policy

Improving value function and/or policy via a model is sometimes called indirect RL or model-based RL (or just "planning")
Two uses of experience:

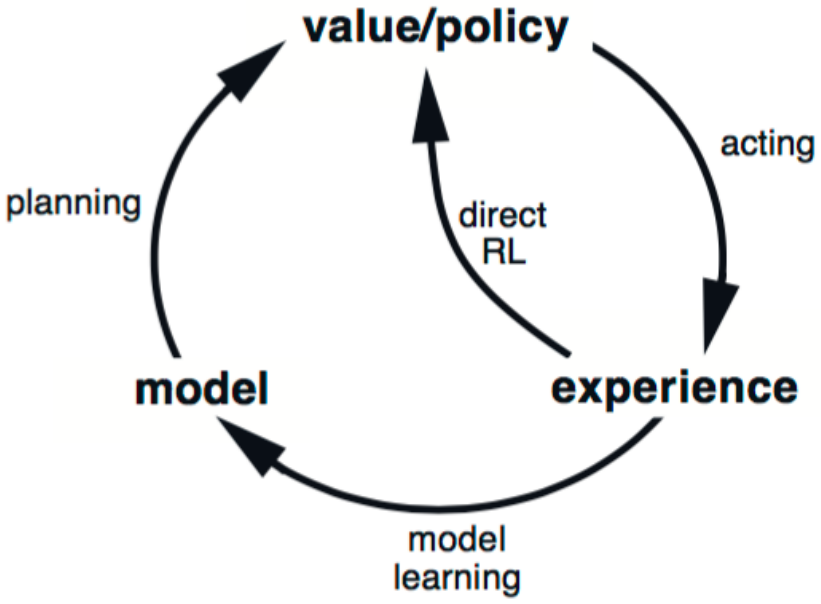
- Indirect (model-based) methods: get "better" policy with fewer interactions
- direct methods: simpler to implement and not affected by models

These two flavours can be meaningfully combined and can occur simultaneously and in parallel.

Actors & Critics

- Policy-Based methods (actors)
  - Find the optimal policy directly without the Q/V-function.
  - Policy-based are considered to have a faster convergence and are better for continuous and stochastic environments.
  - Examples: Policy-Based algorithms like Policy Gradients and REINFORCE
- Value Based methods (critics)
  - Find/approximate optimal value function (mapping action to value).
  - These are considered more sample efficient and steady.
  - Examples: Q Learning, Deep Q Networks, Double Dueling Q Networks, etc

Actor-Critic Methods I



- The principal idea is to split the RL model in two: one for computing an action based on a state and another one to produce the Q values of the action.

Actor-Critic Methods II

- The actor takes as input state and outputs action. It 'acts out', i.e. controls how the agent behaves by learning the optimal policy (policy-based learning).
- The critic evaluates the action by computing the value function (value-based learning).
- Those two models compete and both get better in their own role.
- The result is that the combined architecture learns more efficiently than the two methods separately.

From policy gradient to Actor-Critic Method I

Why combine methods?

- Because agents might take a lot of actions over the course of an episode, it is hard to assign credit to actions, which means that these updates have a high variance
- Therefore, it may take a lot of updates for your policy to converge.
- Moreover, PGs method only works in episodic regimes. If your agent is not acting in an episodic environment, it will never get an update.

From policy gradient to Actor-Critic Method II

Plan: We use advantages of TD Learning and define a Critic. Critic predicts the long term value of a state s, or a state-action pair (s, a). We can use this value $Q(s, a)$ instead of the empirical returns.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r_t(s, a) + \max_{a'} \gamma Q(s', a') - Q(s, a)] \quad (67)$$

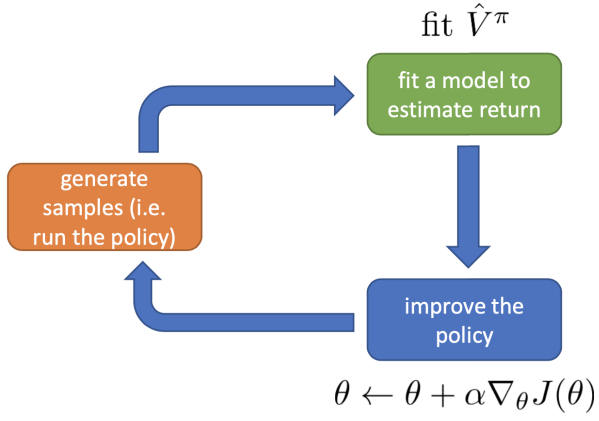
- Low variance, because we update our parameters at each step using the Q values.
- Faster convergence of the policy in an episode
- Moreover, we are bootstrapping off of our own predictions of Q-values.
- Moreover, we can learn in non-episodic domains.

Advantage Actor-Critic (A2C) I

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}^\pi_\phi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}^\pi_\phi(\mathbf{s}'_i) - \hat{V}^\pi_\phi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

$$y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}^\pi_\phi(\mathbf{s}_{i,t+1})$$

$$\mathcal{L}(\phi) = \frac{1}{2}\sum_i \left\|\hat{V}^\pi_\phi(\mathbf{s}_i) - y_i\right\|^2$$



- Key element is the Advantage element.
- Q functions are decomposed into 2 pieces: the state Value function $V(s)$ and the advantage value $A(s, a)$:

$$Q(s, a) = V(s) + A(s, a) \tag{68}$$

Advantage Actor-Critic (A2C) II

- The advantage function captures how much better an action a is compared to the others at a given state s, while as we know the value function captures how good it is to be at this state.

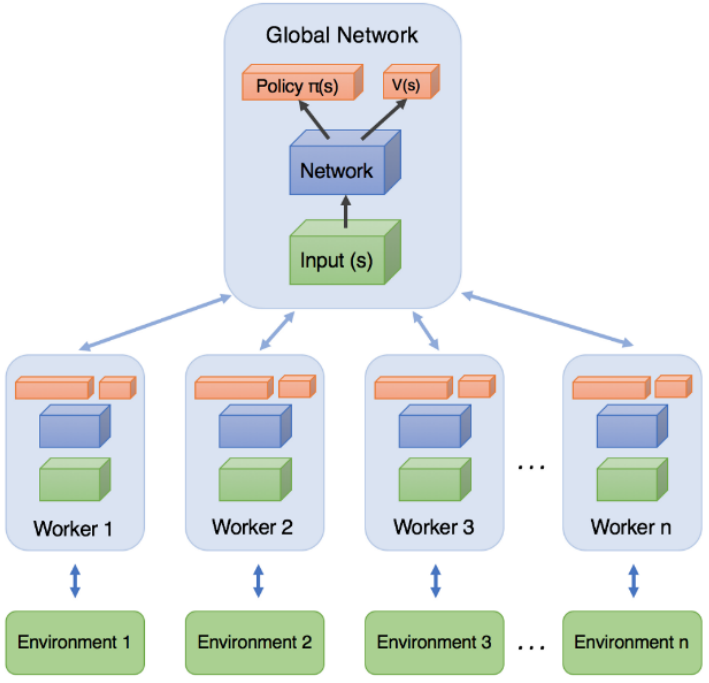
$$A(s,a) = Q(s,a) - V(s) \tag{69}$$
$$A(s,a) = r + \gamma V(\hat{s}) - V(s) \tag{70}$$

- The agent learns Advantage values instead of Q values.
- That way the evaluation of an action is based not only on how good the action is, but also how much better it can be.
- This reduces high variance of policy networks and stabilises the model during training.

Asynchronous Advantage Actor-Critic (A3C) I

- Key difference to A2C is the Asynchronous element.
- A3C consists of multiple independent agents(networks) with their own weights, who interact with a different copy of the environment in parallel.
- Thus, they can explore a bigger part of the state-action space in much less time.
- The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters.

Asynchronous Advantage Actor-Critic (A3C) II



The updates are not happening simultaneously and thatâs where the asynchronous comes from.

Asynchronous Advantage Actor-Critic (A3C) III

After each update, the agents resets their parameters to those of the global network and continue their independent exploration and training for n steps until they update themselves again.

Information flows from the agents to the global network but also between agents as each agent resets his weights by the global network, which has the information of all the other agents.

Section overview

Tricks of the Trade

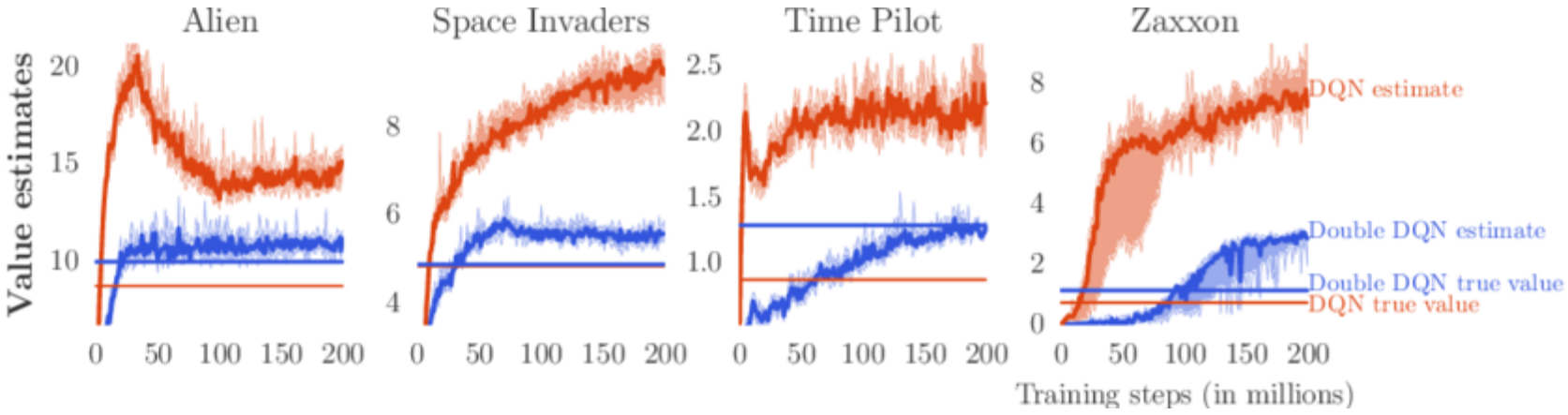
Training Deep RL systems is a skill, requiring insight & experience I

Deep RL systems are complex, dynamic and expensive to train. They will often not train out of the box and require considerable amount of hand-tuning. Both our lab assignments and coursework are meant to help you gain that "feeling" or "intuition" for this "craft".
However, as with all craft, there is underpinning science that when we understand and apply can greatly ease the "craft" part. This is what makes experts in reinforcement learning so valueable to industry and society. In simplistic terms, a builder can assemble a building, but an architect knows if it will collapse. This is what sets you as future machine learners appart from coders who download and simply run any of the hundred DQN implementations available online.

Training Deep RL systems is a skill, requiring insight & experience II

The following slides on the "tricks of the trade" are thus less classical lecture material, and more guidelines to help you crack real-world problems. Nevertheless, careful thinking should allow you to at least intuitively relate concepts of RL theory with the "educated hacks" described in the following.

Training Deep RL systems is a skill - Dealing with Bias I

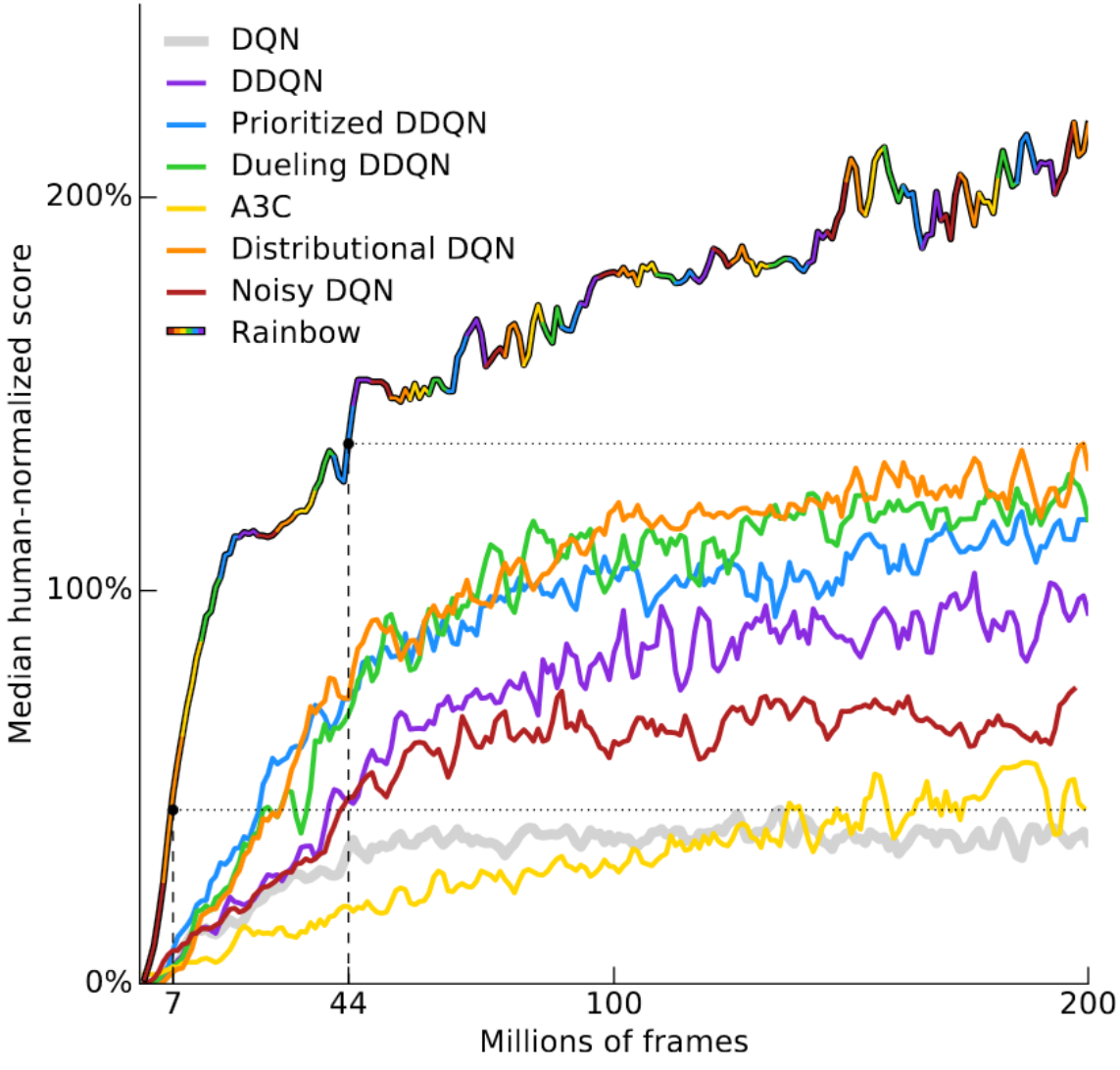


The plot above shows value estimates by DQN (orange) and Double DQN (blue) on four Atari games - obtained by running DQN and DDQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded (end of

Training Deep RL systems is a skill - Dealing with Bias II

training), and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias during learning, we thus see that learning with DDQN is much more stable and less bias prone. After Van Hasselt et al (2015).

Training Deep RL systems is a skill - Sample Efficiency I



Hessel et al (2017).

Tricks of the trade: DQN as example I

In general in Deep RL system we need to consider how we structure the problem (how we frame the MDP). E.g. DQNs are clever because they deal with two aspects of training hurdles:

- Format the input before feeding it into the function approximator
- Control for training variability & stability

Pause here and consider which of the elements of DQNs help us to deal with the two core aspects before continuing.

Tricks of the trade: Formatting inputs I

The following tricks of the Deep RL trade craft are useful, but should be considered carefully if they will behave well for your specific task:

Frame Stacking

In dynamic environments we concatenate multiple time frames of input data (e.g. screen shots) so that the state description contains a description of "position", "velocity" and "acceleration".

Input normalisation

Reducing the complexity of the input by discretising or limiting its range (e.g. making input images black and white only)

Tricks of the trade: Formatting inputs II

Frames of reference/Coordinate Systems

Setting the state up in such a way that the problem becomes simpler to generalise across observations (e.g. it may make sense to setup a gridworld in a coordinate system where the constant goal state is in the origin, or even where the agent is always in the centre of the coordinate system, a so called body-centered coordinate frame). This is often relevant in robotic Deep RL, but also elsewhere, where we know the inherent symmetry of tasks or can choose a description to encode an invariance, e.g. to translations or rotations.

Tricks of the trade: Control of training I

Some of the following points should be really explore in more depth in a Deep Learning course, so we will point you to them but not go into detail.

Bootstrapping

We use estimates to update estimates, this reduces variance during training as we are less reliant on the samples, on the other hand it introduces bias from our initialisation choices.

Tricks of the trade: Control of training II

Reward normalisation or Reward standardisation

Normalising, rescaling or clipping rewards is useful for Deep RL, because having individual rewards that vary greatly can create an unstable learning process where minor events with too large payoffs can confuse the learning. Reward normalisation or Reward standardisation reduces this instability by simply confining the size of the possible unwanted variations.

Tricks of the trade: Control of training III

Target networks

Target networks can be used to reduce the training instability introduced by bootstrapping. Target networks slow down and decouple the dynamics of updating estimates of the value functions, either by making target network to be held constant and only periodically updated to match their reference network (**hard updates**) or gradually updated to slowly converge towards the reference network weights (**soft updates**).

Tricks of the trade: Control of training IV

Weight initialisation

Selection of initial parameter values for gradient-based optimization of deep neural networks is one of the most impactful hyperparameter choices in ay deep learning system, affecting both convergence times and model performance/generalisation capabilities. A "good" initialisation of weights is important to ensure that the initial starting point is "flexible" enough so that learning can immediately change weights in an effective manner, e.g. by avoiding vanishing gradients where learning is very slow or having too large weight values that may take a long time to reign in. In general, how to choose these initiationlised is not well understood, although some theory work (e.g. Saxe et

Tricks of the trade: Control of training V

al, 2014, ICML; Hu et al, 2020, ICML) point to orthogonal weight intiatlisation as theroetically motivated approach (that is already implmented in PyTorch and TensorFlow).

Gradient clipping

Gradient clipping rescales the deep neural networks parameter's gradients so that the (Euclidean) norm of the vector containing all of the model gradients gets clamped under some threshold. The trick here is that while performing gradient clipping results in an upper bound for this norm, it is a rescaling and not a clipping operation that is performed on the individual gradients. Applying gradient clipping is useful when you are experiencing large or exploding gradients, e.g.

Tricks of the trade: Control of training VI

when training task presents large numerical range for its returns. Therefore, gradient clipping can be thought of like acting a bit like Reward standardisation, but may be applicable in setting where the reward standardisation is impractical to implement. However, there may be also other reasons why gradients explode, such as log probability of an almost 0 probability action that was sampled by the policyIn all these cases gradient clipping may be useful to deal with exploding gradients, irrespective of underlying causes, hence most Deep Learning frameworks such as PyTorch support it appropriately.

Tricks of the trade: Control of training VII

Considering (mini-)batch sizes In general (mini-)batch training is useful for decorrelating traces of state-action-reward experiences, reuse observed transitions and reduce variance during training. It is worth here just hinting at a few interesting interactions: A small batch size will create higher variance gradients, which if you use gradient clipping, are more likely to get clipped – thus it has the benefit of reducing the variance of gradients, but it also means that we introduce a bias in our gradient estimates (variance-bias dilemma). In contrast, increasing the batch size is a bias-free method of reducing variance, at the cost of higher computational costs. The choice of batch size and gradient clipping values therefore reduces itself to the

Tricks of the trade: Control of training VIII

ubiquitous machine learning trade-off between variance, bias and computational complexity.