Lecture 12: Deep Reinforcement Learning

Matteo Hessel

UCL 2020

Looking back:

In the past lectures we have discussed a range of important topics:

1. Trading off exploration / exploitation in Bandits: Greedy, Policy gradients, UCB;
2. Sequential decision making in MDPs: Policy iteration, Value iteration;
3. Model-free prediction and control: Monte Carlo, TD, Q-learning, Sarsa;
4. Planning with learned models; expectation vs stochastic models, Dyna, search;
5. Off-policy prediction and control: importance sampling, vtrace.
6. Policy gradients REINFORCE, actor-critics;

Recap: Value function approximation

- Tabular RL does not scale to large complex problems:
  1. Too many states to store in memory
  2. Too slow to learn the values of each state separately,
- We need to generalise what we learn across states.

Recap: Value function approximation

- Estimate values (or policies) in an approximate way:
  1. Map states s onto a suitable "feature" representation $\phi(s)$.
  2. Map features to values through a parametrised function $v_\theta(\phi)$
  3. Update parameters θ so that $v_\pi(s) \sim v_\theta(\phi(s))$
- In past lectures, the feature representation was typically "fixed"
- The parametrised function v_θ was just a linear mapping
- Today, we will consider more complicated non-linear mappings v_θ

Recap: Value function approximation

- Goal: find θ that minimises the difference between v_π and v_θ

$$L(\theta) = E_{S \sim d}[(v_\pi(S) - v_\theta(S))^2]$$

  Where d is the state visitation distribution induced by π and the dynamics p.
- Solution: use gradient descent to iteratively minimise this objective

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta L(\theta) = \alpha E_{S \sim d}[(v_\pi(S) - v_\theta(S)\nabla_\theta v_\theta(S)]$$

Recap: Value function approximation

- Problem: evaluating the expectation is going to be hard in general,
- Solution: use stochastic gradient descent, i.e. sample the gradient update,

$$\Delta\theta = \alpha(G_t - v_\theta(S_t))\nabla_\theta v_\theta(S_t)$$

- where G_t is a suitable sampled estimate of the return,
- Monte Carlo Prediction $\rightarrow G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots$
- TD Prediction $\rightarrow G_t = R_t + \gamma v_\theta(S_{t+1})$

Looking forward:

In the next two lectures we will focus on RL with deep function approximation:

- how do ideas from the previous lectures apply in this setting?
- how can we make RL algorithms more compatible with deep learning?
- how can we make deep learning models more suitable for RL?

Deep value function approximation

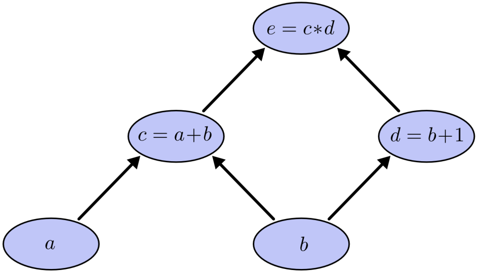
- Parametrise v_θ using a deep neural network.
- For instance as a multilayer perceptron:

$$v_\theta(S) = W_2 \; tanh(W_1 * S + b_1) + b_2$$

- where $\theta = \{W_1, b_1, W_2, b_2\}$
- when v_θ was linear ∇v_θ was trivial to compute
- how do we compute such gradient if v_θ is parameterised by a deep neural net?

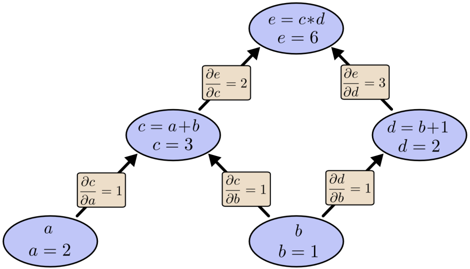
Computational graphs

- We can represent computation via direct acyclic graphs
- specifying the sequence of operations to compute some quantity
- e.g. we can represent the sequence of operations in a neural network,



Automatic differentiation

- If we know how to compute gradients for individual nodes wrt their inputs,
- we can compute gradients of any node wrt to any other, in one backward sweep,
- Accumulate the gradient products along paths, sum gradients when paths merge.



JAX

- There are many autodiff frameworks to compute gradients in deep networks
- In this course we will be using JAX:

$$\text{JAX} = \text{Numpy} + \text{Autodiff} + \text{Accelerators}$$

- Numpy: the canonical Python library for defining matrices and matrix operations,
- Autodiff: implemented via tracing by `jax.grad`,
- Accelerators (GPU/TPU): supported via just in time compilation with `jax.jit`.

JAX - ecosystem

There a growing ecosystem built around JAX to support

- Neural network definition: Haiku
- Optimisation: Optix
- Reinforcement learning: Rlax
- Many more ...

Deep Q-learning

- Use a neural network to approximate q_θ: $O_t \mapsto \mathbb{R}^m$ for m actions
- Update parameters θ through the stochastic update:

$$\Delta\theta = \alpha(G_t - q_\theta(S_t, A_t))\nabla_\theta q_\theta(S_t, A_t), \quad G_t = R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a)$$

- For consistency with DL notation you may write this as gradient of a pseudo-loss:

$$L(\theta) = \frac{1}{2}\left(R_{t+1} + \gamma [\![\max_a q_\theta(S_{t+1}, a)]\!] - q_\theta(S_t, A_t)\right)^2$$

- Note: we ignore the dependency of the bootstrap target on θ,
- Note: this is not a true loss!

Deep Q-learning in JAX

- First, we define the neural network q_θ using Haiku:

```
import haiku as hk

def forward_pass(obs):
  network = hk.Sequential([
      lambda x: jnp.reshape(x, (-1,)),
      hk.Linear(50),
      jax.nn.relu,
      hk.Linear(3)
  ])
  return network(obs)

network_init, network_apply = hk.transform(forward_pass)
```

Deep Q-learning in JAX

- Next, we define the update to parameters θ:

```
@jax.jit
def loss_fn(theta, obs_tm1, a_tm1, r_t, d_t, obs_t):
  q_tm1 = network_apply(theta, obs_tm1)
  q_t = network_apply(theta, obs_t)
  target_tm1 = r_t + d_t * jnp.max(q_t)
  td_error = jax.lax.stop_gradient(target_tm1) - q_tm1[a_tm1]
  return 0.5 * (td_error)**2

@jax.jit
def update(theta, obs_tm1, a_tm1, r_t, d_t, obs_t):
  dl_dtheta = jax.grad(loss_fn)(theta, obs_tm1, a_tm1, r_t, d_t, obs_t)
  return tree_multimap(lambda p, g: p-alpha*g, theta, dl_dtheta)
```

Deep learning aware RL

We know from deep learning literature that

- Using mini-batches instead of single samples is typically better,
- Stochastic gradient descent assumes gradients are sampled i.i.d.

However in online reinforcement learning algorithm:

- We perform an update on every new update,
- Consecutive updates are strongly correlated.

Deep learning aware RL - planning

Can we make RL more deep learning friendly?

- In the planning lectures we discussed Dyna-Q and Experience Replay,
- these mix online updates with updates on data sampled from
  1. a buffer of past experience
  2. a learned model of the environment
- Both approaches can
  1. reduce correlation between consecutive updates,
  2. support mini-batch updates instead of vanilla SGD.

Deep learning aware RL - other approaches

Experience replay / planning with learned models are not the only ways to address this:

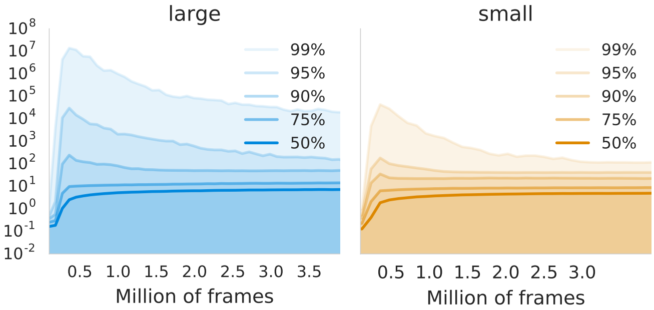
- better online algorithms: e.g. eligibility traces,
- better optimisers: e.g. momentum
- change the problem setting itself: e.g. parallel environments

The deadly triad

- If we use Dyna-Q or experience replay (DQN), we are combining:
  1. Function approximation: we are using a neural network to fit action values,
  2. Bootstrapping: we bootstrap on $max_a Q_\theta(s, a)$ to construct the target,
  3. Off-policy learning: the replay hold data from a mixture of past policies.
- What about the deadly triad?
- Is this a sane thing to do?

The deadly triad in deep RL (van Hasselt et al. 2018)

- Empirically we actually find that unbounded divergence is rare,
- More common are value explosions that recover after an initial phase,



- This phenomenon is also referred to as "soft-divergence".

Target networks

- Soft divergence still cause value estimates to be quite poor for extended periods.
- We can address this in our deep RL agents using a separate target network:
  1. Hold fixed the parameters used to compute the bootstrap targets $max_a Q_\theta(s, a)$,
  2. Only update them periodically (every few hundreds or thousands of updates).
- This breaks the feedback loop that sits at the heart of the deadly triad.

Target networks in JAX

- Target network switching in JAX is trivial
- Thanks to the functional programming style

```
q = network_apply(params, obs_t)
target_q = network_apply(target_params, obs_t)
if i % target_refresh_period == 0:
  target_params = jax.tree_map(lambda t: t.copy(), params)
```

Deep double Q-learning (van Hasselt et al. 2016)

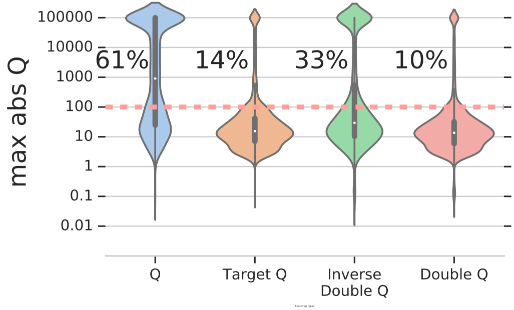
- Q-learning has an overestimation bias, that can be corrected by double Q-learning

$$L(\theta) = \frac{1}{2}\left(R_{i+1} + \gamma[\![q_{\theta^-}(S_{i+1}, \operatorname*{argmax}_a q_\theta(S_{i+1}, a))]\!] - q_\theta(S_i, A_i)\right)^2$$

- Great combination with target networks: we can use the frozen params as θ^-.
- What is the effect of double Q-learning on the likelihood of soft divergence?

The deadly triad in deep RL - estimators

- The form of the statistical estimator of the return matters for divergence!

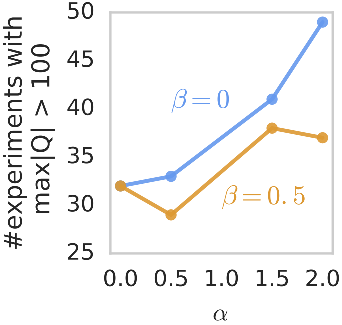


Prioritized replay (Schaul et al. 2016)

- DQN samples uniformly from replay
- Idea: prioritize transitions on which we can learn much
- Basic implementation:
$$\text{priority of sample } i = |\delta_i|\,,$$
where δ_i was the TD error on the last this transition was sampled
- Sample according to priority
- Typically involves some additional design choices

The deadly triad in deep RL - state distribution

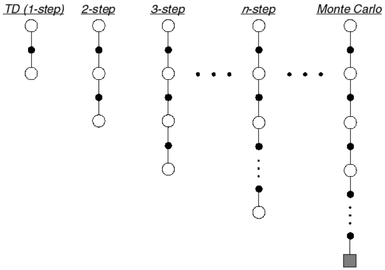
- We bias sampled states away from the state visitation under the agent policy,
- Our updates are going to be even more off-policy!



- We can use importance sampling to correct at least partially.

Multi-step updates (Sutton 1988)

- Today we always considered targets that bootstraps after a single step,
- e.g. $G_t = R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a)$
- In general, targets may look n steps into the future



Multi-step prediction

- Define the n-step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n v_\theta(S_{t+n})$$

- For $n = 1, 2, \infty$, we interpolate between 1-step TD and MC:

$$\begin{array}{lll}
n = 1 & (TD) & G_t^{(1)} = R_{t+1} + \gamma v_\theta(S_{t+1}) \\
n = 2 & & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\theta(S_{t+2}) \\
\vdots & & \vdots \\
n = \infty & (MC) & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T
\end{array}$$

- n-step deep temporal-difference learning

$$\Delta\theta = \alpha(G_t^{(n)} - v_\theta(S_t))\nabla_\theta v_\theta(S_t)$$

Multi-step control

- Define the *n*-step Q-learning target

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \underbrace{q_{\theta^-}(S_{i+1}, \underset{a}{\text{argmax}}\ q_\theta(S_{i+1}, a))}_{\text{Double Q bootstrap target}}$$

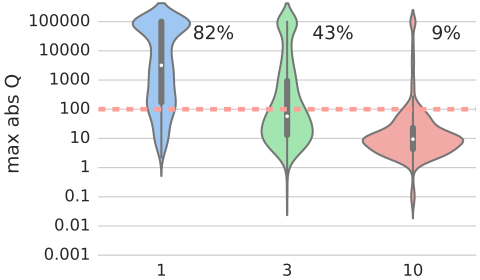
- Multi-step deep Q-learning

$$\Delta\theta = \alpha(G_t^{(n)} - q_\theta(S_t, A_t))\nabla_\theta q_\theta(S_t, A_t)$$

- Return is partially on-policy, bootstrap is off-policy
- A well-defined target: *"On-policy for n steps, and then act greedy"*
- That's okay — less greedy, but still a policy improvement.

The deadly triad in deep RL - multi step targets

- Multi-step targets allow to trade-off bias and variance,
- They also reduce our reliance on bootstrapping,
- As a result they also reduce the likelihood of divergence.

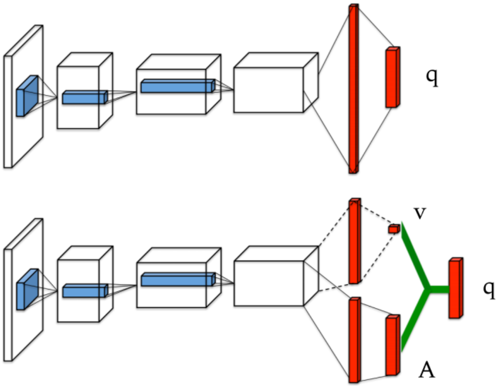


RL aware deep learning: architectures

- Much of the successes of deep learning have come from encoding the right inductive bias in the network structure:
  - Translational invariance in image recognition $\rightarrow$ convolutional nets,
  - Long term memory $\rightarrow$ gating in LSTMs,
- We shouldn't just copy architectures designed for supervised problems,
- What are the right architectures to encode inductive biases that are good for RL?

Dueling networks (Wang et al. 2016)

- We can decompose $q_\theta(s, a) = v_\xi(s) + A_\chi(s, a)$, where $\theta = \xi \cup \chi$
- Here $A_\chi(s, a)$ is the advantage for taking action a



Dueling networks

Dueling networks in JAX

- We can easily build networks with arbitrary topology

```
import haiku as hk

def forward_pass(obs):
  flatten_fn = lambda x: jnp.reshape(x, (-1,))
  h = hk.Sequential([flatten_fn, hk.Linear(20)])(obs)
  a = hk.Sequential([jax.nn.relu, hk.Linear(3)])(h)
  v = hk.Sequential([jax.nn.relu, hk.Linear(1)])(h)
  return a + v

network_init, network_apply = hk.transform(forward_pass)
```

RL aware deep learning: capacity

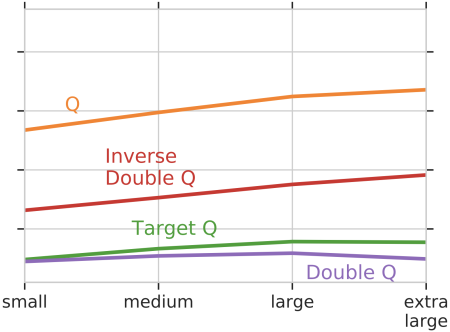
- In supervised deep learning we often find that:

$$\text{More Data} + \text{More capacity} = \text{Better performance}$$

- The loss is easier to optimise, there is less interference, etc ...
- How does network capacity affect value function approximation?

The deadly triad in deep RL - network capacity

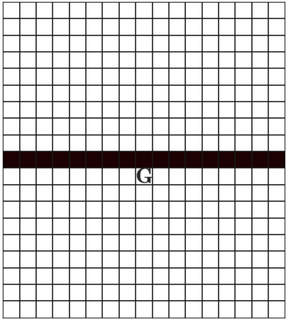
- Larger networks do typically perform better overall,
- But... they are however more susceptible to the deadly triad,



(b) Network size

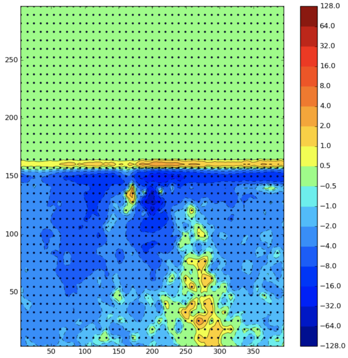
RL aware deep learning: generalisation

- The deadly triad shows that generalization in RL can be tricky
- Consider the problem of value learning in presence of sharp discontinuities of v_π

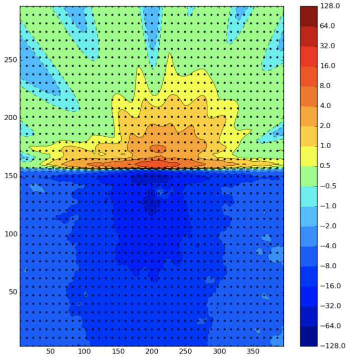


RL aware deep learning: generalisation

- TD learning with deep function approximation leads to "leakage propagation"



(c) MC prediction error heatmap



(d) TD prediction error heatmap

Learning about many things

- Behind "deadly triad" / "leakage propagation" is inappropriate generalisation,
- Better representations can help with these issues,
- E.g. we can share the state representation across many tasks
    1. Predict the value of different policies,
    2. Predict future observations,
    3. Predict/control other "cumulants", different from the main task reward.