An introduction to Deep Reinforcement Learning

Imperial Machine Learning Society

Pierre H. Richemond

Imperial College London - Data Science Institute & Deep Learning Network

Table of contents

Introduction: DRL, what and why

First things first, the most important slide

Ask questions !

RL in one minute !

- Reinforcement learning = adaptive, extreme trial-and-error
- To quote Remi Munos (newly head of DeepMind Paris), 'a child doesn't learn to ride a bicycle by solving equations; they try, fall, learn a bit and try again'
- Deep RL = RL combined with deep neural networks
- Enabled by large amounts of collected data, combined with the processing power of GPUs

Domains connex to RL

- Computer Science
- Engineering
- Mathematics
- Psychology
- Neuroscience
- Economics

RL and machine learning

- Supervised learning
- Unsupervised learning
- Reinforcement learning : dealing with an evolving dataset

Examples of RL - Successes & use cases

- Flying stunt with a helicopter
- More generally, robotics control
- Potentially, self-driving cars
- Superhuman performance in board games (Backgammon, Chess, Go) just from rules
- Control of a power station with a view of reducing energy consumption
- Trade the financial markets
- Learn to play video games better than humans just from pixels and score

RL can be seen as a form of supervised, optimal control.

Frontiers of RL

- Hard games : partially observable ! You don’t necessarily know all of what’s going on.
- Poker
- The real frontier : Starcraft II (long term planning; many actions; hard for humans too !)

Chronology

- 1984 Rich Sutton's thesis
- 1989 Q-learning (Watkins)
- 1991 REINFORCE (Williams & Peng)
- 2006 Monte-Carlo Tree Search (Szepesvari)
- 2009 David Silver's Ph.D. thesis
- 2013 Deep Q-Networks
- 2015 DQN scales - *Nature* paper
- 2016 Actor critic + deep networks
- 2017 Soft Q-learning = soft actor-critic ; AlphaGo Zero

Key algorithms are 25+ years old

RL in the brain ?

- Reinforcement learning has revolutionized our understanding of learning in the brain in the last 20 years
- The most famous success story is dopamine and prediction errors via average reward
- Basal ganglia exhibit an actor-critic architecture
- Evolutionary methods also work in RL
- Yael Niv of Princeton has an excellent tutorial on the topic; Demis Hassabis, CEO of DeepMind, is also an authority
- Some of the *replay* concepts we will see that speed up RL convergence have eerie neuroscience connections.

The temples of reinforcement learning

- Google DeepMind, arguably
- OpenAI
- The University of Alberta - Canada
- Berkeley AI Research (BAIR) - US
- INRIA - France

Videos

- Atari: https://www.youtube.com/watch?v=V1eYniJ0Rnk
- Helicopters: https://www.youtube.com/watch?v=M-QUkgk3HyE
- Robots: https://www.youtube.com/watch?v=ZhsEKTo7V04
- Moar robots: https://www.youtube.com/watch?v=hx_bgoTF7bs&t=65s

Principles : some math (as little as possible)

Everything starts with basic concepts

- Environment
- Agent
- State
- Action
- Next state
- Reward

We will get to all these concepts.

The reward is the most important

- A reward R_t is a scalar feedback signal
- Indicates how well the agent is doing at step t
- The agent's job is ALWAYS to maximize cumulative reward
- The *reward hypothesis* posits *all goals* can be described by the maximisation of the *expected cumulative reward*
- A philosophical debate...

Examples of rewards

- Helicopter : + reward for going higher and following trajectory ; - reward for crashing
- Chess or Go : no reward for most moves, +1 for winning the game, -1 for losing
- Manage an investment portfolio : reward is the change in dollar market value of the portfolio
- Humanoid robot walk : +something for forward motion, -1 for falling over
- Atari : reward is the change in score in your emulator

Sometimes the reward is exogeneous (imposed by the world you're in) sometimes it's designed-engineered by the algorithm creator.

Sequential Decision Making

- Goal : select actions in order to maximise total future reward
- Actions may have long term consequences
- Rewards may be delayed
- It may be better to practice delayed gratification, and sacrifice immediate reward for the longer-term gain

Agent and environment

At each step t, the agent

- Executes action A_t
- Receives observation O_t
- Receives scalar reward R_t

The environment then:

- Receives action A_t
- Emits observation O_{t+1}
- Emits scalar reward R_{t+1}

Then, t is incremented (at environment step).

History, transitions, and state

- The history is the sequence of transitions.
- A transition is a triplet state, action, reward
- This represents the sensorimotor stream of a robot, or (embodied) agent
- The history can be written as $H_t = O_1, R_1, A_1, \cdots, A_{t-1}, O_t, R_t$
- State is the information used to determine what happens next
- State is a function of history : $S_t = f(H_t)$

Examples

- Chess
- Atari
- Humanoid robot walk

Markov state

Definition : a state is *Markov* iff

$$\mathbb{P}[S_{t+1}|(S_1, \cdots, S_t)] = \mathbb{P}[S_{t+1}|S_t]$$

- ie 'the future is independent of the past, given the present'
- ie 'all of the history is contained in the current state'
- ie 'the state is a sufficient statistic of the future'
- ie 'we are memory-less'

We can *markovianize* (sic) the state by adding more information to it - for instance, the last few frames in a video game so as to be able to infer speed and momentum.

To summarize till now

- Given a certain environment (game abstraction), an agent (player) selects between actions a available to her, and in doing so, collects a reward r and moves from state s to state s'.
- The agent's objective is to pick actions and build a trajectory so as to maximize the sum of their rewards.
- Rewards and states are not known in advance.

Undiscounted warning

- If you've ever encountered RL before, you'll notice I am not using γ's in this course
- γ is also known as a *discount factor* that controls how far in the future we want to look
- γ is here to make the math work; better to build first intuitions without
- In practice it's an important parameter that can be (carefully) tuned for performance.

Major components of an RL agent

An RL agent may include one or more of these components :

- Policy : agent's behaviour or function
- Value function : how good is each state of action
- Model : agent's representation of the environment

Policy

- The policy is the agent's behaviour.
- It is a map (function) from state to action, or a distribution over actions given states.
- This means:
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$

A policy fully defines the behaviour of an agent, and it only depends on the current state (not the history). Hence policies are *stationary* that is time-independent.

Value function

- A value function is a prediction of future reward.
- It is used to evaluate the goodness/badness of states
- And therefore to select between actions !
- Example:

$$V_\pi(s) = \mathbb{E}[r_{t+1} + r_{t+2} + \cdots | S_t = s]$$

Model

- A model predicts what the environment will do next. Can you say 'imagination' ?
- $\mathcal{P}$ predicts the next state
- $\mathcal{R}$ predicts the next (immediate) reward.

RL Agent Taxonomy : categorizing RL agents, 1

There are many possible ways (hence algorithms) to do RL !

- Value-based : value function ; no policy
- Policy-based : policy ; no value function
- Actor-critic : policy and value function, together.

In this course we will focus on value-based methods.

RL Agent Taxonomy : categorizing RL agents, 2

- Model-free : policy and/or value function; but no model
- Model based: policy and/or value function; model

In this course we will focus on model-free methods.

RL Agent Taxonomy : categorizing RL agents, 3

When we use neural networks to determine those, we say we train the concept in question.

Examples : 'I trained my policy network' 'I trained my model'

Atari Example

Our canonical example. The rules of the game are unknown, and we want to to learn directly from interactive game-play. We pick actions on joystick, see resulting pixels and scores.

- Environment = emulator
- States = pixels & score (!).
- Notion of transition between states : from one screenshot to another.
- Actions = joystick button presses
- Rewards = changes in score

Exploration vs Exploitation, 1

- RL is like trial-and-error.
- Recall that rewards and states are not known in advance.
- The agent should discover a good policy from experiencing the environment
- Doing so should be accomplished without sacrificing too much reward along the way !

Exploration vs Exploitation, 2

- Exploration finds more information about the environment
- Exploitation exploits known information to maximise reward
- It is important to do both, in a balanced way
- If always following the same policy, some states may remain never visited = high possibility of being stuck with a suboptimal policy in stochastic environments.

Exploration vs Exploitation, 3 - Examples

- Restaurant selection
- Online Banner Ads
- Natural Resources Mining
- Game Playing

Prediction vs Control

- Prediction = evaluate the future (given a policy)
- Control = optimise the future (find the best policy)

Some more concepts

- Markov Chains
- Markov Rewards Processes
- Markov Decision Processes (MDP)
- Partially Observable Markov Decision Processes (POMDP)

Markov Chains

- Markov Chains : you have n states labelled from 1 to n and perform a (random) walk on the chain.
- Each state of the way, you have a given *transition probability* (not necessarily known in advance) $p_{i,j}$ to move from state S_i to state s_j.
- Because we are memoryless, these numbers only depend on starting state s_i (and where we're going) but not on the history of the states.
- Example : a grad student oscillating between two states - S_1 'writing thesis' - stays in that state most of the time, but also goes to state S_2 'partying at the bar'
- The *state transition matrix* is written as $P = \begin{bmatrix} 0.8 & 0.2 \\ 0.9 & 0.1 \end{bmatrix}$.
- There is a rich and complex mathematical body of work on Markov chains, but this is all we will need here.

MRPs

- Markov chains are underpinning the transition dynamics in the states of our environment.
- Markov Rewards Processes are Markov chains with rewards attached !
- Think of it as collecting a string of rewards r_i associated with the states s_i taken via the (random) walk on the chain.
- Note that in RL the whole point is for the walk not to be random anymore.
- The n-step return $R_n = \sum_{i=1}^{n} r_i$ is just the sum of n rewards collected.
- It is a random quantity whose expectation (the *expected return*) we'd like to maximise ; mathematically, RL is cast as an optimisation problem.

Finally, MDPs

- MDPs are Markov Decision Processes
- They are Markov Rewards Processes where now actions are informing randomly the next state you're in, i.e., MRPs with decisions.
- So every state s_i becomes a couple of a state and action (s_i, a_i).
- Because of the introduction of those actions, the maximisation on previous slide becomes an active maximisation process
- That is, RL is also an control problem.

MDPs and POMDPs

- Partially Observable Markov Decision Processes (POMDP) : In a partially observable environment, the agent indirectly observes the environment, and some state information is hidden from her (example : Poker vs Chess)

Why all this ?

Because we are Markov, we will only ever be interested in two states: the one we are coming from (s) and the one we are going to (s'). We can discard anything that comes before s. There are only two indices here, so matrices (rather than more general d-dimensional tensors) will be all we need. Notation-wise :

- $T(s, a, s')$ will be the *transition dynamics*, or probability to go to state s' after having taken action a in state s.
- $R(s, a, s')$ will be the reward associated with going to from state s to s' when taking action a.

Markov Decision Processes are the typical formulation used for reinforcement learning problems in the literature.

Q-learning

Some other algorithm names that you may hear of

- SARSA
- TD(1) (Monte-Carlo)
- TD(λ)

In this course we will focus on *Q-learning*.

Q and V

- V numbers are known as the state-value functions.
- They depend on policy π, so we often denote them by $V_\pi(s)$.
- They indicate how good a given state is (the long-term value of a state s) for a player following the policy π:

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} r_{t+i} \quad | \quad S_t = s]$$

- ie the *expected cumulative return* from starting in state s, and following policy π thereafter.

Q and V

- Q (for quality !) numbers go one step beyond, and are action-value functions (shorthand for action-state-value functions).
- These numbers contain more information than the $V(s)$ above.
- They indicate how good it is, in state s, to first take action a before continuing to follow policy π for all other next actions.
- They are denoted by $Q_\pi(s, a)$:

$$Q_\pi(s, a) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} r_{t+i} \quad | \quad S_t = s, A_t = a]$$

- ie the *expected cumulative return* from starting in state s, taking first action a, and then following policy π thereafter.

Qs and Vs redux : * optimality

- In general these numbers depend on the policy we use to choose our actions. A given chess position is not as good if played by me (π_1) or by Kasparov (π_2) !
- The optimal Q and V numbers, i.e. the ones that would be attainable by a perfect controller or player π^*, are denoted by a star : $V^*(s) = V_{\pi^*}(s) = \max_{\pi} \quad V_\pi(s)$
- Similarly we define $Q^*(s,a) = Q_{\pi^*}(s,a) = \max_{\pi} \quad Q_\pi(s,a)$.

Qs and Vs redux : * optimality, 2

- If we knew the optimal V numbers perfectly (through a *critic* or *value function iteration*) and we had a model of our world, we could act greedily towards choosing which state to go to.
- If we knew the Q^* numbers perfectly for each action, we would be in even better shape (why ?).
- The goal of value-function based RL is to determine and act according to those Q^* numbers.

How we are going to do RL in practice !

- So we choose to do Q-learning : we will learn the optimal action-values $Q^*(s, a)$.
- To do so, we will keep track of a table of 5 potential numbers (one per Atari action) for each state/screenshot.
- We will initialize the Q's randomly and then interact with the environment by playing, collecting rewards and observations, and placing them into a memory : an *experience replay* buffer.
- Then we will use the Bellman equation to get our network to output better and better estimates for the Q^*.
- As we do this we will also continue playing to get better experiential data, by following the (implicit, greedy) policy given by those new Q values.

The Bellman Equation

- The Bellman equation is possibly the most important equation in all of reinforcement learning.
- It's a time consistency property for optimal value functions. It applies to either Q or V.
- At optimality we get (easy to prove just by expanding the expectation sum one step forward) :

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \max_{a'} \quad Q^*(s',a') \right]$$

- It's important by itself, but most importantly, it will tell us how to do our neural network training.
- Notice we have $Q^*(s,\cdot)$ on the left, but $Q^*(s',\cdot)$ on the right !

Bellman equation : intuition

- In the case where we are looking back at past, realized transitions, we can simply set $T(s, a, s') = 1$ for the only one transition that has actually happened, and 0 elsewhere.
- This allows us to give an ex-post, intuitive decomposition of the Bellman equation in two parts:
- A horizontal (time axis) part: for the actual action taken a_{actual},

$$Q^*(s, a_{actual}) = r(s, a_{actual}) + V^*(s')$$

- A vertical (optimal action) part:

$$V^*(s') = \max_a \quad Q^*(s', a)$$

ϵ-greedy : exploitation vs exploration, redux

Making sure we are not stuck in a suboptimal policy. In general (99% of the times) we will use $\arg\max Q(s, a)$ as an implicit policy.
However:

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- We pick a small number ϵ before running the algorithm.
- With probability $1 - \epsilon$, choose the greedy action
- With probability ϵ, choose an action at random.
- $\pi(a|s) = \frac{\epsilon}{m} + (1 - \epsilon)\delta_{a=argmaxQ(s,a^*)}$

We can show theoretically that this leads to policy improvement.

Theoretical Q-learning in 1 slide.

Q learning is learning the numbers that enable us to act.

- The target policy π is greedy.
- If now we are being ϵ-greedy, everything simplifies and we get the Q-learning target $R_{t+1} + \max_{a'} \quad Q(s_{t+1}, a')$
- As such, we update $Q(s, a)$ in the *direction* of the target :

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \max_{a'} \quad Q(s_{t+1}, a'))$$

- Theorem : As we take more iterations, Q-learning control converges to the optimal action-value function, $Q(s, a) \rightarrow Q^*(s, a)$.

Description of the theoretical, tabular algorithm

We will train in *episodes* as follows.

- Initialize $Q(s, a)$ for each s, a arbitrarily, and $Q(terminalstate, \cdot) = 0$.
- For each episode, repeat :
- Initialize S
- Repeat (for each step of episode:)
- Choose A from S using ϵ-greedy policy derived from Q
- Take action A, observe R, S'
- $Q_{t+1}(S, A) \leftarrow Q_t(S, A) + \alpha(R + \max_a \quad Q_t(S', a))$
- $S \leftarrow S'$

Problems in practice

First problem : there is a potential infinity of screenshots s. We will hence not be able to implement $Q(s, a)$ as a lookup table, but rather, will resort to using a neural network.

Tabular vs function approximation

Stop me if you think that you've heard this one before...

- Tabular : nice theoretical guarantees and regret bounds (converge speed) but intractable...
- Function approximation : may or may not converge... but heh, it works - at scale !
  In general we use neural networks for function approximation and we stabilize learning using a various set of RL-specific tricks (not just NN tricks).

Convolutional neural networks, 1

- You can think of a neural network as a black box.
- Convolutional neural networks are now the weapon of choice when it comes to computer vision.
- They are interesting in our RL context because of their ability to generalize.

Convolutional neural networks, 2

- Once trained, the neural network can be thought of as a big composite function.
- The exact operations it performs are not so relevant for our purpose ; they are mostly matrix multiplications, taking out negative values, and a mathematical operation named convolution, which is intuitively the same as filtering of an (image) signal.
- The input of the convolutional network is the pixels of the image, and in general, we have n scalar outputs (a vector), where n is the number of output classes.

Convolutional neural networks, 3

- Training a convolutional neural network for classification (say binary : of cats and dogs) consists in showing it many images (batches in a dataset) of cats and dogs, and using a training algorithm to have it output either a 1 when shown a cat, or a 2 when shown a dog.
- The ImageNet computer vision contest was held every year till recently, and consisted in having a program try to assign (recognize) one of 1000 classes to a 224x224x3 image picked in a dataset of 1 million. The last five years' entries were all won by convnets.
- Neural networks, and convnets here, are interesting in our RL context because of their ability to process visual (pixel-space) data, and create hierarchical representations of images.

Convolutional neural networks, 4

- You will learn more about the details of convnets in an upcoming course ; they are an incredibly important and powerful building block of deep learning and litterally power all of modern computer vision (see Stanford CS231n: Convolutional Neural Networks for Visual Recognition).
- In our Atari context, we will use convnets to take as an input a screenshot of the game, and output five scalar numbers : one for each possible action to take.
- In mathematical terms, convnets will perform function approximation of the optimal state-value functions $Q^*(s, a)$.
- The question of course now is how do we train them to produce the right answer ! (You can't just set them to a value like a lookup table - you need to change their weights).

Bootstrapping

- Bootstrapping = wishful thinking
- 'What if we were already at optimality ?'
- The Bellman equation is not valid before we reached a perfect policy ; we are going to pretend it is.
- So instead of pretending the difference between the left side and the right side of the equation (the *Bellman residual*) is zero, we will iteratively minimize it.

Bootstrapping : action-vector view

- In this case, we can repeatedly *pick a realized past transition* (s, a, r, s') and do the following :
- If we pass the destination screenshot s' to the network, we will get an estimate for the value of the state by just computing the action-vector $Q^*(s', a)$ and taking its maximum. We add the effective reward we got on our way there, and make a note of this total (a good proxy estimate of $V^*(s') + r(s, a)$).
- The Bellman equation says that when applying our network to the original screenshot, picking only the value in the action-vector that matches the actual chosen action, should return that very number.
- Hence, we can enforce that by now training the network (using standard supervised learning methods) *to return that target value*.
- We will do that, and then move to another transition (s, a, r, s') and repeat in batches.

Bootstrapping : action-vector view

- Our RL setting and specifically the Bellman equation enabled us to determine target values our neural network should aim to return.
- Viewed in this way, RL is a second-level algorithm or meta-algorithm that sits on top of supervised learning.
- You can see that this principle of Q-learning will work with different types of neural networks.
- There will be interplay between neural architecture (deep) and reinforcement algorithm (RL).

Two stabilizing tricks

- First stabilizing trick is to shuffle transitions (randomize their order in time) in our replay buffer;
- otherwise they are too correlated in time, and the network saturates.

Is this what dreams are made of ?

Two stabilizing tricks

- Second trick is to use two networks : one for $Q(s')$ prediction, another one for $Q(s)$ training. This also stabilizes training by preventing a negative feedback loop.
- These two methods are called *experience replay* and *target network freezing* by Google DeepMind and gave them their first success training networks to superhuman performance on the Atari domain in 2015.

Description of the practical Q-learning algorithm

Initialize replay memory D to capacity N
Initialize action-value network Q with random weights θ
Initialize target action-value network $\hat{Q}$ with weights $\theta^- = \theta$
for episode 1, M **do**
    Initialize sequence with s_1
    for $t = 1, T$ **do**
        With probability ϵ, select a random action a_t
        Otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in the emulator
        Observe reward r_t and image s_{t+1}
        Store experience (s_t, a_t, r_t, s_{t+1}) in D
        Sample random minibatch of (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = \begin{cases} r_j & \text{if episode terminates step } j+1 \\ r_j + \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
        with respect to network weights θ (ADAM)
        Every C steps, reset $\hat{Q} = Q$

DQN with PyTorch in practice

PyTorch itself

- PyTorch is a Python tensor calculation package developed by Soumith Chintala at FAIR.
- It is suitable both for DL and RL and provides the 'right' abstraction-productivity tradeoff.
- It provides dynamic graph construction, which makes debugging easier than with static graphing oriented tensors.
- TensorFlow is another solution that recently added dynamic capabilities - whether you pick one or the other is a matter of taste

OpenAI Gym - Atari subdomain

- OpenAI Gym - *the classic*
- OpenAI Universe
- OpenAI RoboSchool - an open-source robotic control suite - to replace *MuJoCo*
- OpenAI Baselines

Practical session

See code in Jupyter notebook at

`https://github.com/higgsfield/RL-Adventure/blob/master/1.dqn.ipynb`

How to go further

Books and articles

- Sutton & Barto's *An introduction to Reinforcement Learning*
  `http://incompleteideas.net/book/the-book-2nd.html`
- My reading list @ KloudStrife
  `https://kloudstrifeblog.wordpress.com/`

Videos - Youtube

- David Silver UCL Lectures `https://www.youtube.com/watch?v=2pWv7GOvuf0`
- OpenAI Deep RL bootcamp `https://sites.google.com/view/deep-rl-bootcamp/lectures`
- Sergey Levine's Berkeley CS294 DeepRL course `http://rll.berkeley.edu/deeprlcourse/`

Great Ph.D. Theses

- David Silver
- Brian Ziebart
- John Schulman
- More theoretical : Mo Azar, Ian Osband

Key PyTorch code repos

Both of these extremely high quality and recommended.

- Kai Arulkumaran's RAINBOW repo :
  `https://github.com/Kaixhin/Rainbow`
- Higgsfield's RAINBOW repo :
  `https://github.com/higgsfield/RL-Adventure`

Key TensorFlow code repos

- As close to official as possible :
  `https://github.com/tensorflow/models/tree/master/research/pcl_rl`
  by Ofir Nachum the author of PCL, soon to be integrated in TensorFlow
- RLlib: A Composable and Scalable Reinforcement Learning Library
  `https://github.com/ray-project/ray/tree/master/python/ray/rllib`
- Imperial's Brendan Maginnis Atari-RL
  `https://github.com/brendanator/atari-rl`
- OpenAI's *baselines*
  `https://github.com/openai/baselines`

What to replicate

- DQN : Human-level control through deep RL (Mnih et al.)
- A3C : Asynchronous methods for Deep RL (Mnih et al.)
- TRPO : Trust Region Policy Optimization (Schulman et al.)

What to replicate

RAINBOW : Combining improvements in reinforcement learning

- DQN
- Double-DQN
- Prioritized Experience Replay
- Dueling DQN
- A3C (Asynchronous, Advantage Actor-Critic)
- Noisy DQN
- Distributional DQN

A word on AlphaGo

Key concept = fictitious self-play

To learn as much as possible, you wouldn't play against an infant, and you (probably) wouldn't play against Kasparov either... the maximum signal would be from playing against yourself, it it were possible.

An idea whose time has come

- RL lets you do active machine learning on a constantly increasing dataset
- It has its issues, but RL is the future

DRL is hard... DRL that matters

- Encouraging reproducibility in reinforcement learning
- 'RL algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results.'
- False Discoveries everywhere ? Dependency on the random seed
- Multiple runs are necessary to give an idea of the variability. Reporting the best returns is not enough - at the very least, report top and bottom quantiles
- Don't get discouraged !

Don't be an alchemist

Suggestions for experiment design

Factors influencing training, by order of importance:

- Fix your random seed for reproducibility !
- *Reward scaling* (and more generally reward engineering) help a lot
- *Number of steps* in the calculation of returns
- *Discount factor*, if you have one, also an issue

Clip gradients to avoid NaNs, visualize them with TensorBoard (even in PyTorch !), and finally be patient waiting for convergence...

The #1 one challenge in RL

Sample efficiency - how do we design RL methods that are possibly re-usable (*meta-learning*), and less data-hungry ?

More questions from the audience ?

Announcements

- One final note : Josh Gordon of *Google TensorFlow* at Imperial on Thursday, 4pm. A good opportunity to learn more about the abilities of TF for RL !
- Deep Learning Reading Group every Tuesday evening
- KDD is coming to London ! 19-23 Aug 2018. I will personally organize the Deep Learning Day with numerous exciting speakers, including a PyTorch special guest.

Thanks for your time !

Email pierre.richemond@gmail.com

Twitter @KloudStrife