Introduction to Reinforcement Learning

or how to track cell movements in an automated way

Bogdan Mazoure

McGill University / MILA / Microsoft

February 23 2021

About

About me:
B.Sc. in Statistics + Computer Science, M.Sc. in Mathematics, now Ph.D. in Computer Science at McGill / MILA.
About MILA:
Inter-university association (McGill, UdeM, HEC, Polytechnique), formerly only UdeM.
10+ core professors and 100+ graduate students working on pure and applied deep learning methods.
We take Master's and Ph.D. students with strong mathematical and programming background, as well as year-long interns.

Simple example

Imagine a robot that must navigate a maze to some point (mouse-cheese here)

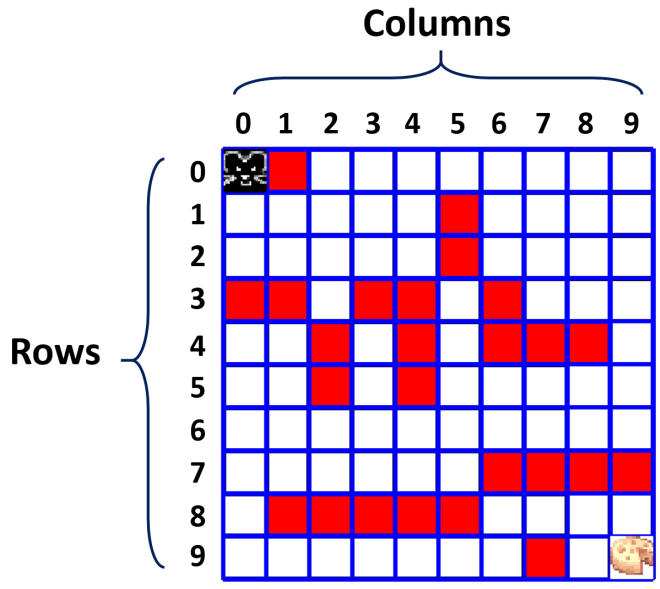


Figure 1: Simple navigation task

Suppose it can only move along the XY axes, one block at the time.

Markov Chains

Suppose we have some discrete space $\mathcal{X}$, say $\mathbb{Z}$. Every timestep, the system transitions to some state $x_t, 1 \leq t \leq T$. All of subsequent theory relies on the Markov property:

$$p(x_t|x_{1:t-1}) = p(x_t|x_{t-1}). \tag{1}$$

Typically, if we have n "states", then the "transition" probability can be stored in a matrix form as

$$\mathbf{P}_{xx'} = p(x'|x), \tag{2}$$

of size $n \times n$.

A word about stationarity

Markov chains can be loosely split into two parts: those with a "nice" long-term behaviour, and those with a more sporadic pattern.
The nice long-term behaviour can be characterized by stationarity, that is,

$$p(s_{t+k}|s_{t+k-1}) = p(s_t|s_{t-1}), \tag{3}$$

meaning that transition probability depends on the duration between events and not on the time itself.
How to compute the stationary distribution? Find a distribution which does not change over time, i.e. the eigenfunction of $\mathbf{P}$:

$$\rho\mathbf{P} = \rho \tag{4}$$

.

Markov Chain weather model

Very simple case:
3 states: sunny, rainy, cloudy. Suppose the weather today depends only on yesterday's weather with following probabilities:

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.4 & 0.6 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5}$$

.
Then,

$$p(\text{sunny at time } 5 | \text{sunny at time } 0) = (\mathbf{P}^5)_{00} = 0.54. \tag{6}$$

Example (continued)

Solving for ρ in Python through
`rho = numpy.linalg.eig(P.T)[1][:,0]` gives

$$\rho = (0.34, 0.37, 0.29), \tag{7}$$

the relative proportion of time spent in each of the 3 states after infinitely many steps, *regardless* of the initial probabilities.
Why do we care? Because classical statistical methods perform well on stationary data, which unfortunately happens very rarely in real datasets. RL tends to work well on non-stationary processes.

Simple example (continued)

Only running a Markov chain by itself is not the focus of this talk. You can check about sampling from MC using Monte Carlo Markov chains in the literature.
To solve decision problems, we introduce the notion of "action" into the Markov chain.

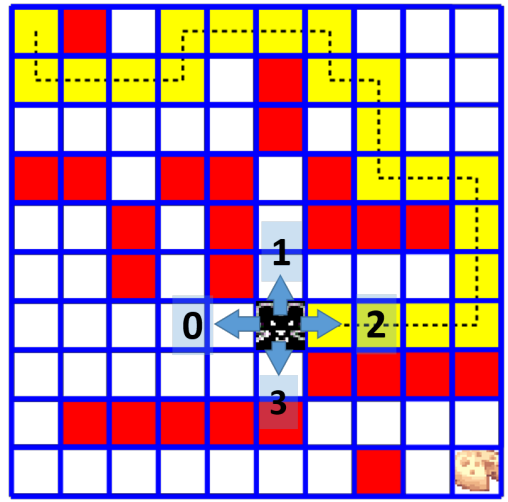


Figure 2: Which action should the agent pick in this state to get closer to the goal?

Markov Decision Processes

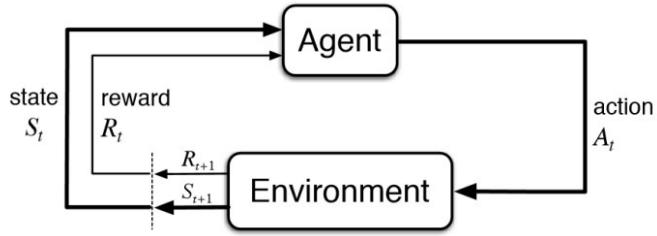


Figure 3: Agent-environment interaction

We can fully describe any Markovian decision system (e.g. video games) with the following quantities:

1. A set of states $\mathcal{X}$;
2. A set of actions $\mathcal{A}$;
3. A transition probability $p(x_t|a_t, x_{t-1})$;
4. A reward function $r(x_t, a_t)$.

What does RL data look like?

Typical data for reinforcement learning is organized into **trajectories:**

$$x_0, a_1, r_1, x_1, a_2, r_2, ..., x_{T-1}, a_T, r_T, x_T, \tag{8}$$

where x_0 is the initial state and x_T is the final state.
Re-using the maze example above, we can describe the yellow trajectory as:

$$(9,0), \text{"}\textit{down}\text{"}\,[3], +0, (8,0), \text{"}\textit{right}\text{"}\,[2], +0, ... \tag{9}$$

There is no X/y modelling as in supervised learning, neither just an X as in unsupervised learning (although we can transform trajectories into a regression task).

Policy

So far, we know how to describe a Markov decision process, but how do we find the solution (i.e. reach the goal state)?
Introduce the **policy**: $\pi(a_t|s_{t-1}) = p(a_t|s_{t-1})$. Conditional density function telling which action to take at every step.

- It can be deterministic: $p(a_1 = \text{"}down\text{"}|s_0 = (9, 0))$;
- It can also be continuous over $\mathbb{R}$: $p(a|s) = \mathcal{N}(0, 0.1)$.

Challenge: How to find π for a specific task?
Very vague answer: Take the gradient wrt the rewards cumulated in a trajectory via the *value function*.

Value function

We can use the sum of rewards collected in a trajectory as a metric for how well the agent is performing.

$$G_t = r_t + r_{t+1} + r_{t+2}... \tag{10}$$

Since T can be infinite, we discount cumulative rewards by a radius $\gamma \in (0, 1)$.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \leq \frac{r_{\max}}{1-\gamma}, \tag{11}$$

where $r_{\max} = \max(r_1, r_2, ...)$.
The value function is just the conditional expectation of G_t given s_t wrt to π!

$$\begin{aligned} V(s_t) &= \mathbb{E}_\pi[G_t|s_t], \\ Q(s_t, a_t) &= \mathbb{E}_\pi[G_t|s_t, a_t]. \end{aligned} \tag{12}$$

The operator $\mathbb{E}_\pi[\cdot]$ simply means: collect trajectories using π, then average the returns over all trajectories.

Learning π

Since ML is about **learning**, let's derive an easy learning rule for the Q function using the *Bellman* equation:

$$Q(s_t, a_t) = \max_{a \in \mathcal{A}} \left(r_t + \gamma Q(s_{t+1}, a) \right) \tag{13}$$

In a perfect setting, both sides are equal. In practice, however, they tend not to be, which we use to minimize the Bellman error:

$$\delta_t = Q(s_t, a_t) - \max_{a \in \mathcal{A}} \left(r_t + \gamma Q(s_{t+1}, a) \right) \tag{14}$$

The (deep) RL optimization problem now boils down to find parameters θ of Q_θ which solve

$$\min_\theta \frac{1}{2} \delta_\theta^2. \tag{15}$$

Once we found Q_θ, we just take $a_t | s_t = \max_{a \in \mathcal{A}} Q(s_t, a)$ (known as the *greedy* policy).

Deep reinforcement learning

To make it simple, the simplest deep RL algorithm is the deep Q-network, which wants to estimate a function $Q_\theta(s, a)$ parametrized by a neural network (fully connected, CNN, LSTM, etc).
Just like in supervised deep learning, we need:

1. Clearly defined input and output spaces;
2. A dataset;
3. A loss function;
4. An optimizer;

Deep Q-network (DQN)

The first and simplest proposed algorithm in 2015 was DQN. Relies on minimizing $\frac{1}{2}\delta^2$.
DQN was tested on Atari games, showing much better performance than an expert human player.

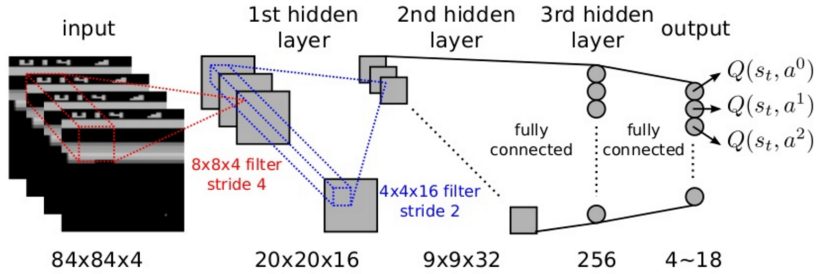


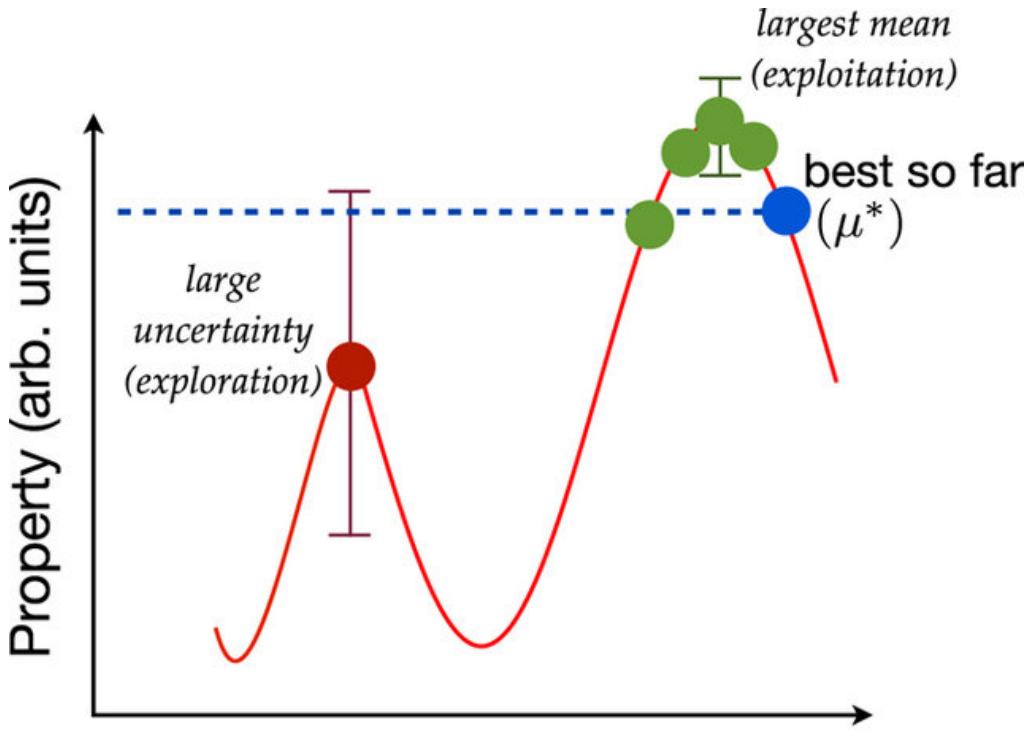
Figure 4: Neural network architecture of DQN

The input to the network is a tensor $84 \times 84 \times 4$. Each RGB image is converted to grayscale, then 4 consecutive images are stacked together (so that the network has an idea of velocity of the objects). The output is a vector of size $|\mathcal{A}|$, one value for every action. We take the action with the highest value.
`https://colab.research.google.com/github/tensorflow/agents/blob/master/docs/tutorials/1_dqn_tutorial.ipynb`

Exploration-exploitation trade-off

One might imagine an environment in which taking the immediate best action will put the agent into a local maximum. For example, moving a car back and forth in order to go on top of a hill will have a sinusoidal pattern.

Balancing optimal and suboptimal actions is known as the exploration-exploitation trade-off. The simplest strategy is, when asked to take an action, to act randomly $100\varepsilon\%$ of the time, and act greedy in the other cases.



Experience replay

So far, we still have not addressed the exact dataset structure. Data in deep RL is stored (for most algorithms) in an array known as **experience or buffer replay**.
A replay buffer of size K looks something like this:

$$\mathcal{B} = \begin{bmatrix} s_{10}^{1} & a_{11}^{1} & r_{11}^{1} & s_{11}^{1} & a_{12}^{1} \\ s_{98}^{2} & a_{99}^{2} & r_{99}^{2} & s_{99}^{2} & a_{100}^{2} \\ \vdots & \vdots & \vdots & & \\ s_{0}^{K} & a_{1}^{K} & r_{1}^{K} & s_{1}^{K} & a_{2}^{K} \end{bmatrix} \tag{16}$$

Just as like in deep learning, the neural network is trained on a randomly sampled batch from the buffer (think of randomly selecting rows in $\mathcal{B}$). Plug these quantities into Eq.9 and use any optimizer to find the first order gradient.

Replay buffer

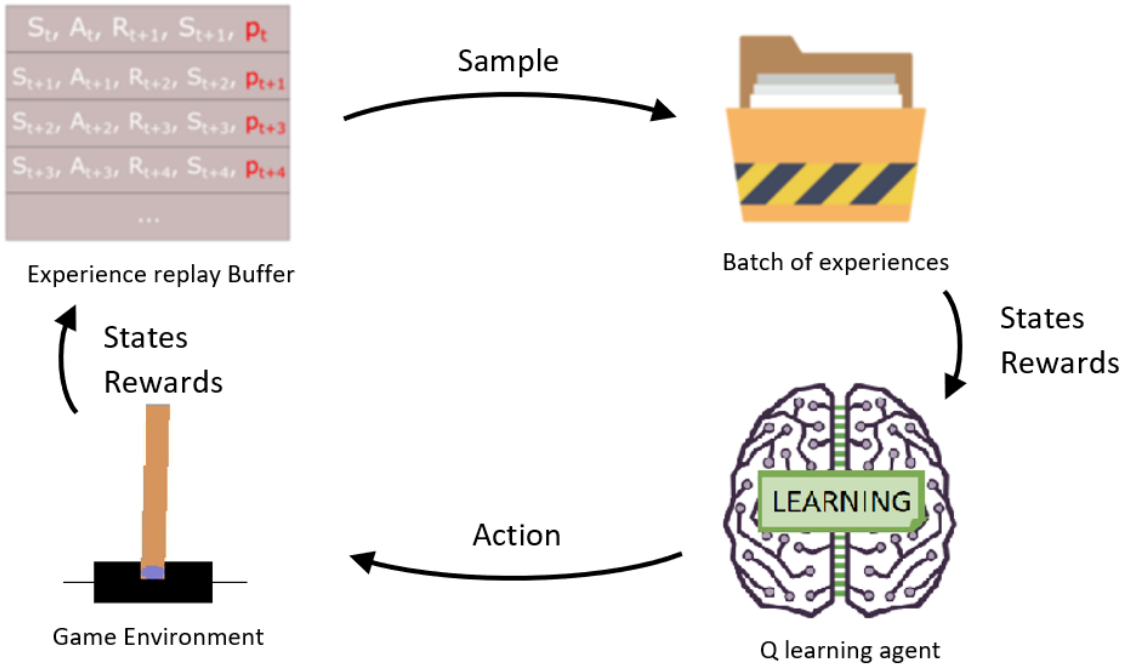


Figure 6: All game screens are added to the replay buffer. The neural network is then trained on random samples from the buffer.

A word about the optimizer

Picking the correct optimization rule is crucial to convergence rate in deep RL. Common optimizers:

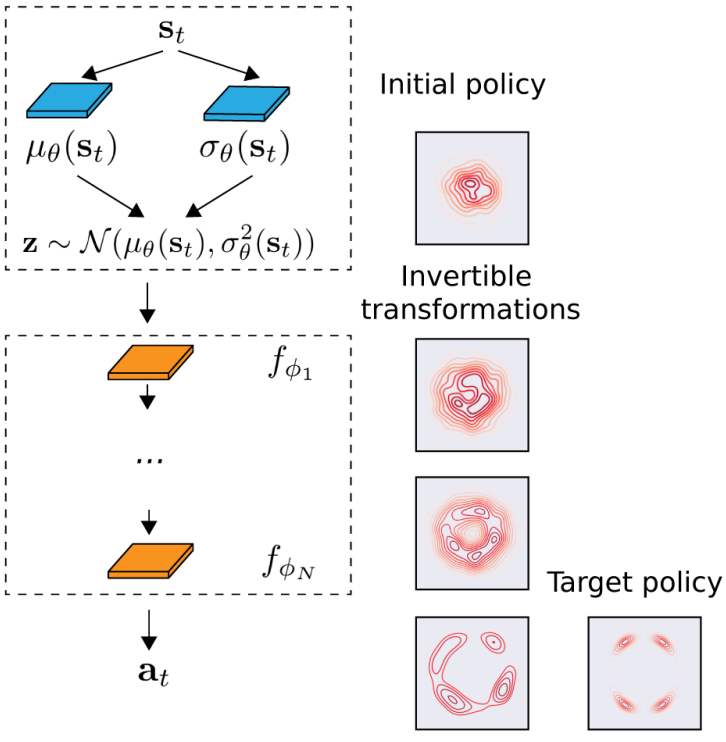
1. Adam: State-of-the-art, very efficient but sometimes unstable;
2. RMSProp: Quite good, useful for problems with a lot of moving parts;
3. Vanilla SGD;

Typically we use Adam with a learning rate of the order of 10^{-4}. Reminder: the general optimization rule for parameters θ with learning rate α, loss L and data batch x:

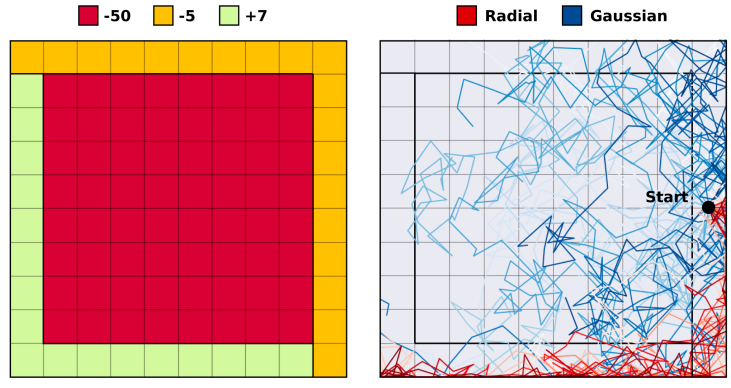
$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \alpha \nabla_\theta L_{\theta^{(t)}}(x) \tag{17}$$

My research interests

Learning meaningful representations in RL through density estimation, mathematical statistics, kernels embeddings and self-supervised learning.
For example, density estimation as robotic control:



(a) Transforming a Gaussian density into a complicated 4-modes shape.



(b) Example of robotic task where our approach saw huge improvements.

Applications

We will now go over a real-life application of RL: tracking cell movement.

Imagine you are a micro-biologist and would like to model the dynamics of cellular evolution from a set of pictures.

Raw images can be too uninformative about evolution *dynamics*, which is why the authors of the method pre-process all observations to extract positions of all cells on a given plate into a fixed length vector using the automated cell lineage tracing technology.

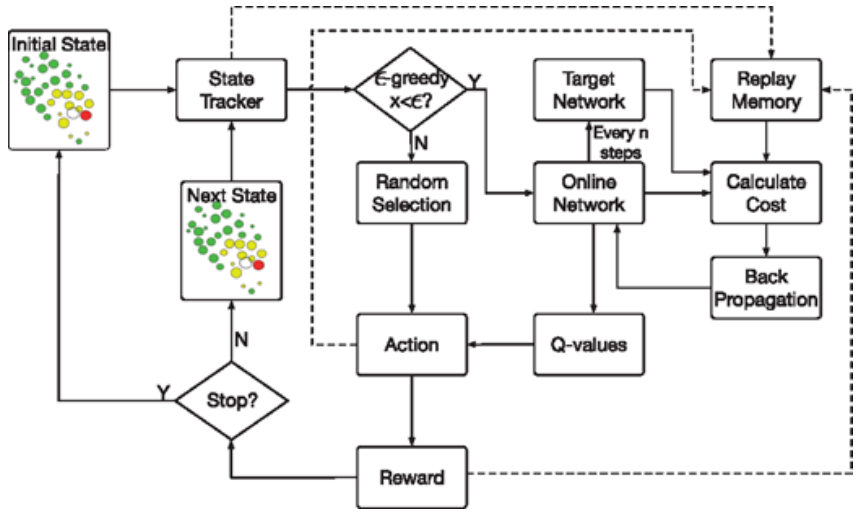


Figure 8: Interaction diagram of DQN for cell tracking

Cell state representation

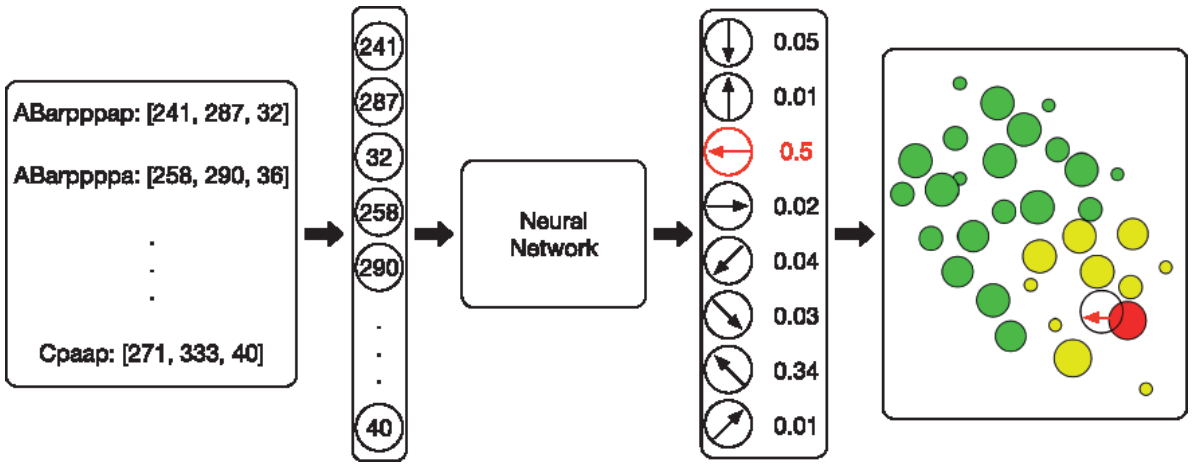


Figure 9: Encoding of the representation of a cell system extracted through a pre-processing method.

Cell movement model

Actions are represented by 8 angles of cardinal directions equally distant (at 45 degrees). The magnitude of the vector (i.e. speed) is fixed a priori.

Rewards are assigned based on a combination of collision and shortest path to destination.

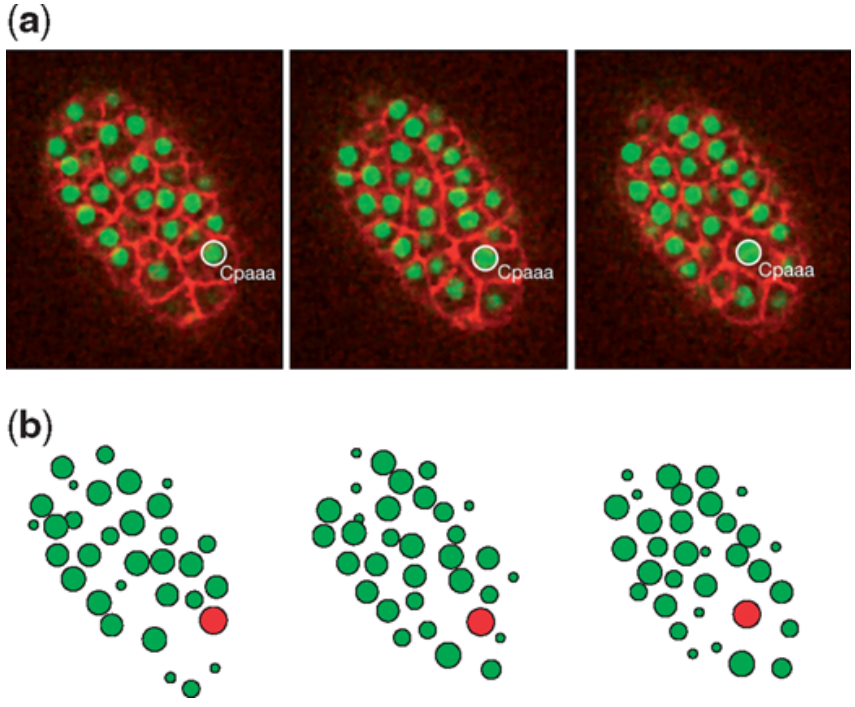


Figure 10: a) Ground truth cell colony labelled using cell lineage tracing and b) predictions made by the Q network

Results

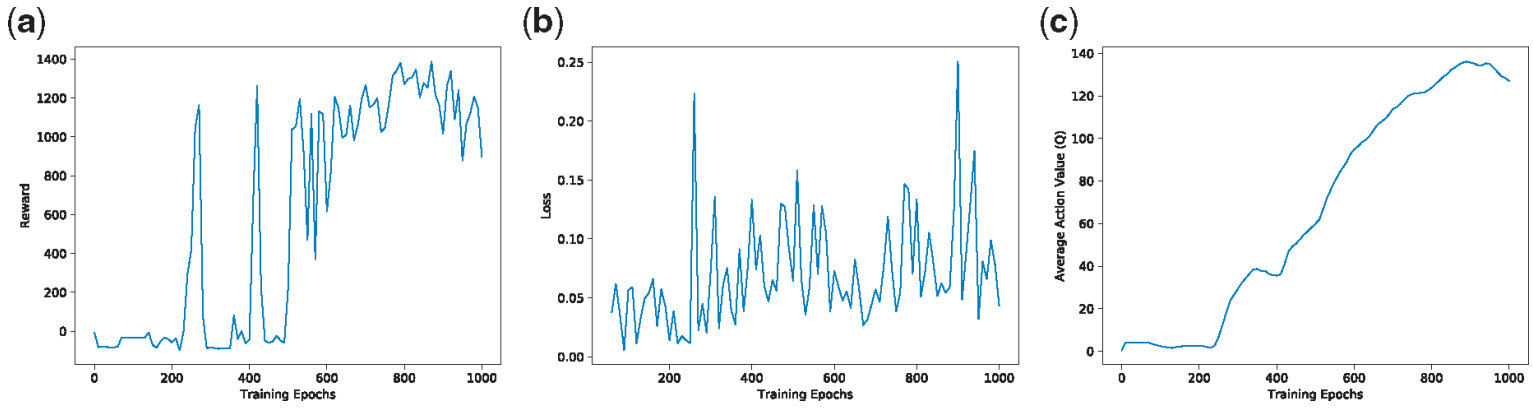


Figure 11: Training reward, loss and action values for the cell tracking problem.

Results

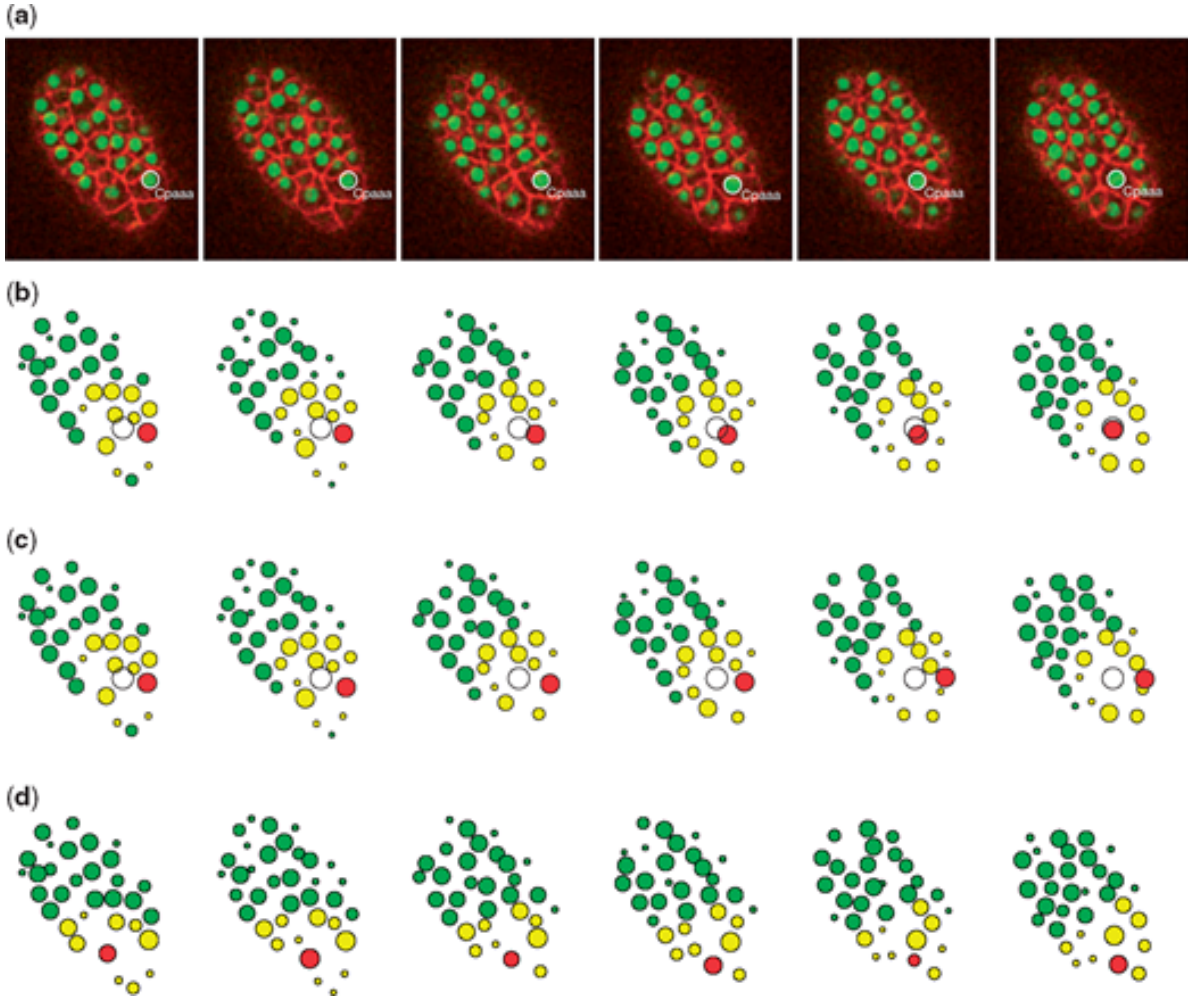


Figure 12: Ground truth vs learned dynamics model of cell movement with and without the distance rule. Distance rule makes cells go to a pre-determined location.

Conclusion

Reinforcement learning is currently a popular suite of tools to solve difficult time-dependent data problems, such as non-stationary data distribution and large state spaces.

Link to the cell movement tracking paper: `https://academic.oup.com/bioinformatics/article/34/18/3169/4986416`.

Currently, almost all applications of RL are in video games and robotics - now is the time to apply RL to bioinformatics, econometrics, psychology and other fields!
If you're interested to discuss the topic further, send me an e-mail at `bogdan.mazoure@mail.mcgill.ca` .