Computational Neuroscience Lab

Institute of Computer Science, University of Tartu

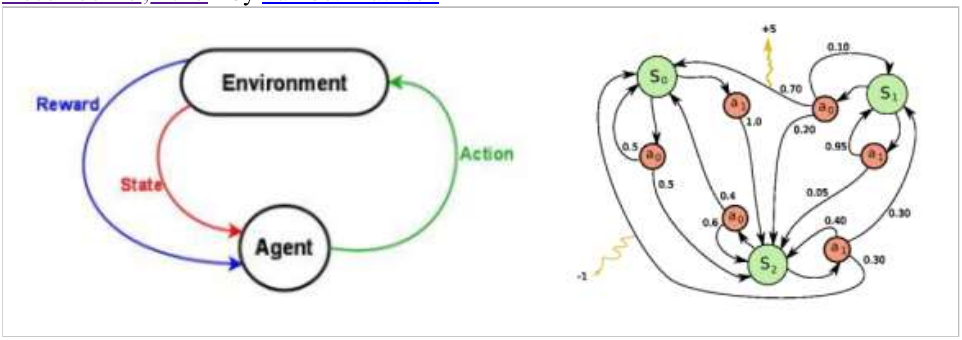
Main Menu

Skip to content

- Recent
- People
- Research
  - Thesis Proposals
- Publications
- Teaching
- Resources
- Contact

Demystifying Deep Reinforcement Learning

December 19, 2015 · by Tambet Matiisen ·



This is the part 1 of my series on deep reinforcement learning. See part 2 "Deep Reinforcement Learning with Neon" for an actual implementation with Neon deep learning toolkit.

Today, exactly two years ago, a small company in London called DeepMind uploaded their pioneering paper "Playing Atari with Deep Reinforcement Learning" to Arxiv. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased. The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human!

It has been hailed since then as the first step towards general artificial intelligence – an AI that can survive in a variety of environments, instead of being confined to strict realms such as playing chess. No wonder DeepMind was immediately bought by Google and has been on the forefront of deep learning research ever since. In February 2015 their paper "Human-level control through deep reinforcement learning" was featured on the cover of Nature, one of the most prestigious journals in science. In this paper they applied the same model to 49 different games and achieved superhuman performance in half of them.

Still, while deep models for supervised and unsupervised learning have seen widespread adoption in the community, deep reinforcement learning has remained a bit of a mystery. In this blog post I will be trying to demystify this technique and understand the rationale behind it. The intended audience is someone who already has background in machine learning and possibly in neural networks, but hasn't had time to delve into reinforcement learning yet.

The roadmap ahead:

1. **What are the main challenges in reinforcement learning?** We will cover the credit assignment problem and the exploration-exploitation dilemma here.
2. **How to formalize reinforcement learning in mathematical terms?** We will define Markov Decision Process and use it for reasoning about reinforcement learning.
3. **How do we form long-term strategies?** We define "discounted future reward", that forms the main basis for the algorithms in the next sections.
4. **How can we estimate or approximate the future reward?** Simple table-based Q-learning algorithm is defined and explained here.
5. **What if our state space is too big?** Here we see how Q-table can be replaced with a (deep) neural network.
6. **What do we need to make it actually work?** Experience replay technique will be discussed here, that stabilizes the learning with neural networks.
7. **Are we done yet?** Finally we will consider some simple solutions to the exploration-exploitation problem.

Reinforcement Learning

Consider the game Breakout. In this game you control a paddle at the bottom of the screen and have to bounce the ball back to clear all the bricks in the upper half of the screen. Each time you hit a brick, it disappears and your score increases – you get a reward.

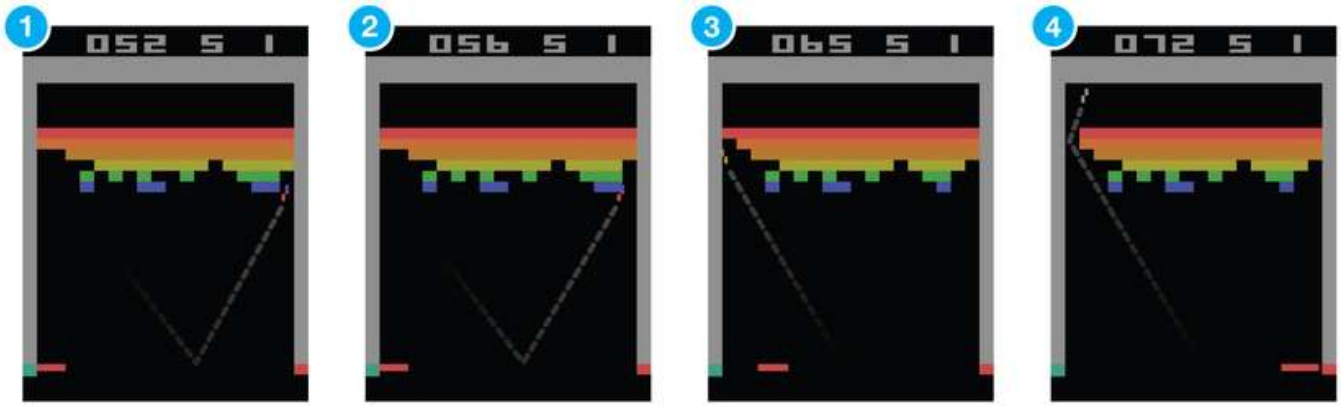


Figure 1: Atari Breakout game. Image credit: DeepMind.

Suppose you want to teach a neural network to play this game. Input to your network would be screen images, and output would be three actions: left, right or fire (to launch the ball). It would make sense to treat it as a classification problem – for each game screen you have to decide, whether you should move left, right or press fire. Sounds straightforward? Sure, but then you need training examples, and a lots of them. Of course you could go and record game sessions using expert players, but that's not really how we learn. We don't need somebody to tell us a million times which move to choose at each screen. We just need occasional feedback that we did the right thing and can then figure out everything else ourselves.

This is the task **reinforcement learning** tries to solve. Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.

While the idea is quite intuitive, in practice there are numerous challenges. For example when you hit a brick and score a reward in the Breakout game, it often has nothing to do with the actions (paddle movements) you did just before getting the reward. All the hard work was already done, when you positioned the paddle correctly and bounced the ball back. This is called the **credit assignment problem** – i.e., which of the preceding actions were responsible for getting the reward and to what extent.

Once you have figured out a strategy to collect a certain number of rewards, should you stick with it or experiment with something that could result in even bigger rewards? In the above Breakout game a simple strategy is to move to the left edge and wait there. When launched, the ball tends to fly left more often than right and you will easily score on about 10 points before you die. Will you be satisfied with this or do you want more? This is called the **explore-exploit dilemma** – should you exploit the known working strategy or explore other, possibly better strategies.

Reinforcement learning is an important model of how we (and all animals in general) learn. Praise from our parents, grades in school, salary at work – these are all examples of rewards. Credit assignment problems and exploration-exploitation dilemmas come up every day both in business and in relationships. That's why it is important to study this problem, and games form a wonderful sandbox for trying out new approaches.

Markov Decision Process

Now the question is, how do you formalize a reinforcement learning problem, so that you can reason about it? The most common method is to represent it as a Markov decision process.

Suppose you are an **agent**, situated in an **environment** (e.g. Breakout game). The environment is in a certain **state** (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on). The agent can perform certain **actions** in the environment (e.g. move the paddle to the left or to the right). These actions sometimes result in a **reward** (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called **policy**. The environment in general is stochastic, which means the next state may be somewhat random (e.g. when you lose a ball and launch a new one, it goes towards a random direction).

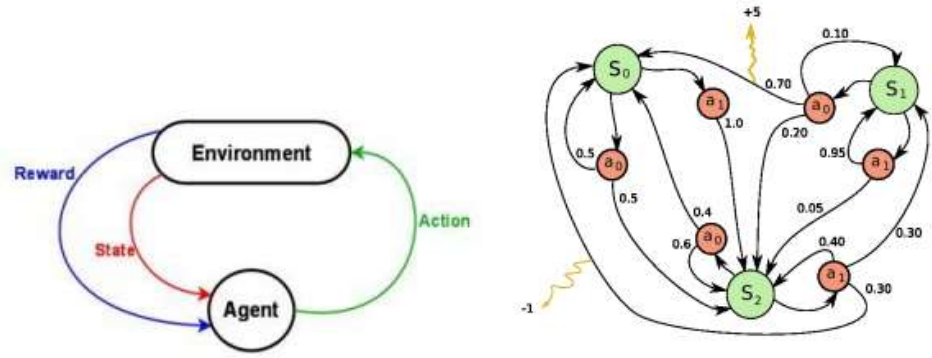


Figure 2: *Left:* reinforcement learning problem. *Right:* Markov decision process. Image credit: Wikipedia.

The set of states and actions, together with rules for transitioning from one state to another and for getting rewards, make up a Markov decision process. One episode of this process (e.g. one game) forms a finite sequence of states, actions and rewards:

$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$

Here s_i represents the state, a_i is the action and r_{i+1} is the reward after performing the action. The episode ends with terminal state s_n (e.g. "game over" screen). A Markov decision process relies on the Markov assumption, that the probability of the next state s_{i+1} depends only on current state s_i and performed action a_i, but not on preceding states or actions.

Discounted Future Reward

To perform well in long-term, we need to take into account not only the immediate rewards, but also the future awards we are going to get. How should we go about that?

Given one run of Markov decision process, we can easily calculate the **total reward** for one episode:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Given that, the **total future reward** from time point *t* onward can be expressed as:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use **discounted future reward** instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step *t* can be expressed in terms of the same thing at time step *t*+1:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor $\gamma = 0$, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like $\gamma = 0.9$. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor $\gamma = 1$.

A good strategy for an agent would be to **always choose an action, that maximizes the discounted future reward**.

Q-learning

In Q-learning we define a function $Q(s, a)$ representing the discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = max_\pi R_{t+1}$$

The way to think about $Q(s, a)$ is that it is "the best possible score at the end of game after performing action a in state s". It is called Q-function, because it represents the "quality" of certain action in given state.

This may sound quite a puzzling definition. How can we estimate the score at the end of game, if we know just current state and action, and not the actions and rewards coming after that? We really can't. But as a theoretical construct we can assume existence of such a function. Just close your eyes and repeat to yourself five times: "$Q(s, a)$ exists, $Q(s, a)$ exists, …". Feel it?

If you're still not convinced, then consider what the implications of having such a function would be. Suppose you are in state s and pondering whether you should take action a or b. You want to select the action, that results in the highest score at the end of game. Once you have the magical Q-function, the answer becomes really simple – pick the action with the highest Q-value!

$$\pi(s) = argmax_a Q(s,a)$$

Here π represents the policy, the rule how we choose an action in each state.

OK, how do we get that Q-function then? Let's focus on just one transition $< s, a, r, s' >$. Just like with discounted future rewards in previous section we can express Q-value of state s and action a in terms of Q-value of next state s'.

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

This is called the **Bellman equation**. If you think about it, it is quite logical – maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.

The main idea in Q-learning is that **we can iteratively approximate the Q-function using the Bellman equation**. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The gist of Q-learning algorithm is as simple as the following:

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

α in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when α=1, then two Q[s,a]-s cancel and the update is exactly the same as Bellman equation.

max_a, Q[s',a'] that we use to update Q[s,a] is only an estimation and in early stages of learning it may be completely wrong. However the estimations get more and more accurate with every iteration and it has been shown, that if we perform this update enough times, then the Q-function will converge and represent the true Q-value.

Deep Q Network

The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the existence of each individual brick. This intuitive representation is however game specific. Could we come up with something more universal, that would be suitable for all the games? Obvious choice is screen pixels – they implicitly contain all of the relevant information about the game situation, except for the speed and direction of the ball. Two consecutive screens would have these covered as well.

If we would apply the same preprocessing to game screens as in the DeepMind paper – take four last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels – we would have $256^{84\times84\times4} \approx 10^{67970}$ possible game states. This means 10^{67970} rows in our imaginary Q-table – that is more than the number of atoms in the known universe! One could argue that many pixel combinations and therefore states never occur – we could possibly represent it as a sparse table containing only visited states. Even so, most of the states are very rarely visited and it would take a lifetime of the universe for the Q-table to converge. Ideally we would also like to have a good guess for Q-values for states we have never seen before.

This is the point, where deep learning steps in. Neural networks are exceptionally good in coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions immediately available.

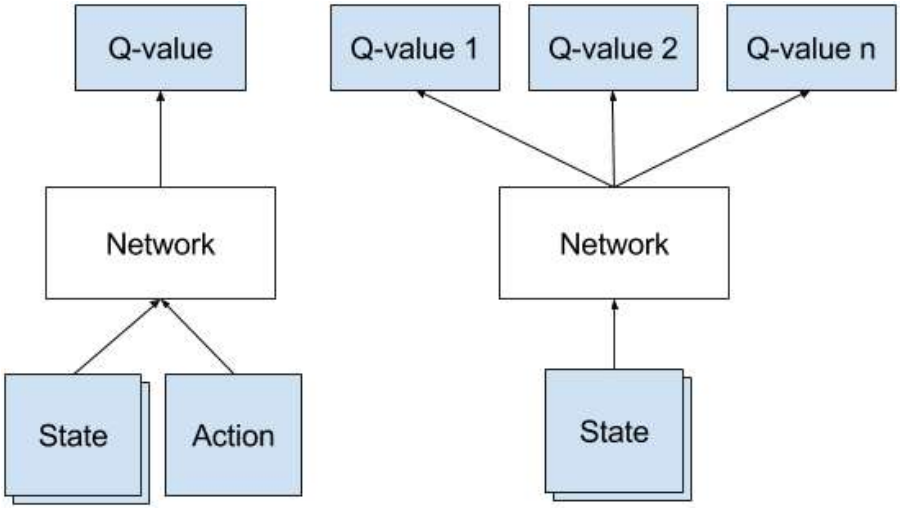


Figure 3: *Left:* Naive formulation of deep Q-network. *Right:* More optimal architecture of deep Q-network, used in DeepMind paper.

The network architecture that DeepMind used is as follows:

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8×8	4	32	ReLU	20x20x32
conv2	20x20x32	4×4	2	64	ReLU	9x9x64
conv3	9x9x64	3×3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

This is a classical convolutional neural network with three convolutional layers, followed by two fully connected layers. People familiar with object recognition networks may notice that there are no pooling layers. But if you really think about that, then pooling layers buy you a translation invariance – the network becomes insensitive to the location of an object in the image. That makes perfectly sense for a classification task like ImageNet, but for games the location of the ball is crucial in determining the potential reward and we wouldn't want to discard this information!

Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari). Q-values can be any real values, which makes it a regression task, that can be optimized with a simple squared error loss.

$$L = \frac{1}{2}[\underbrace{r + \gamma max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

Given a transition $< s, a, r, s' >$, the Q-table update rule in the previous algorithm must be replaced with the following:

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s' and calculate maximum over all network outputs $max_{a'} Q(s', a')$.
3. Set Q-value target for action a to $r + \gamma max_{a'} Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

Experience Replay

By now we have an idea how to estimate the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network. But it turns out that approximation of Q-values using non-linear functions is not very stable. There is a whole bag of tricks that you have to use to actually make it converge. And it takes a long time, almost a week on a single GPU.

The most important trick is **experience replay**. During gameplay all the experiences $< s, a, r, s' >$ are stored in a replay memory. When training the network, random samples from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and the train network on these.

Exploration-Exploitation

Q-learning attempts to solve the credit assignment problem – it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward. But we haven't touched the exploration-exploitation dilemma yet…

Firstly observe, that when a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is **ε-greedy exploration** – with probability ε choose a random action, otherwise go with the "greedy" action with the highest Q-value. In their system DeepMind actually decreases ε over time from 1 to 0.1 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

Deep Q-learning Algorithm

This gives us the final deep Q-learning algorithm with experience replay:

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability ε select a random action
        otherwise select a = argmaxa'Q(s,a')
    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D

    sample random transitions <ss, aa, rr, ss'> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then tt = rr
```

```
        otherwise tt = rr + γmaxa'Q(ss', aa')
    train the Q network using (tt - Q(ss, aa))^2 as loss

    s = s'
until terminated
```

There are many more tricks that DeepMind used to actually make it work – like target network, error clipping, reward clipping etc, but these are out of scope for this introduction.

The most amazing part of this algorithm is that it learns anything at all. Just think about it – because our Q-function is initialized randomly, it initially outputs complete garbage. And we are using this garbage (the maximum Q-value of the next state) as targets for the network, only occasionally folding in a tiny reward. That sounds insane, how could it learn anything meaningful at all? The fact is, that it does.

Final notes

Many improvements to deep Q-learning have been proposed since its first introduction – Double Q-learning, Prioritized Experience Replay, Dueling Network Architecture and extension to continuous action space to name a few. For latest advancements check out the NIPS 2015 deep reinforcement learning workshop and ICLR 2016 (search for "reinforcement" in title). But beware, that deep Q-learning has been patented by Google.

It is often said, that artificial intelligence is something we haven't figured out yet. Once we know how it works, it doesn't seem intelligent any more. But deep Q-networks still continue to amaze me. Watching them figure out a new game is like observing an animal in the wild – a rewarding experience by itself.

Credits

Thanks to Ardi Tampuu, Tanel Pärnamaa, Jaan Aru, Ilya Kuzovkin, Arjun Bansal and Urs Köster for comments and suggestions on the drafts of this post.

Links

- David Silver's lecture about deep reinforcement learning
- Slightly awkward but accessible illustration of Q-learning
- UC Berkley's course on deep reinforcement learning
- Rich Sutton's tutorial on reinforcement learning
- David Silver's reinforcement learning course
- Nando de Freitas' course on machine learning (two lectures about reinforcement learning in the end)
- Andrej Karpathy's course on convolutional neural networks

Tags: ai, deep learning, reinforcement learning

59 responses on "Demystifying Deep Reinforcement Learning"

1. Pingback: Deep Reinforcement Learning With Neon | Computational Neuroscience Lab·
2.  *antianticamper* December 26, 2015 at 1:58 pm · · Reply →

   Very nice explanation, especially subtle points like the update rule for all actions simultaneously. But even without pooling, parameter tying in the convolutional layers implements a form of translational invariance which, as you mention, does not seem appropriate in this situation. So why the use of shared parameters?

   -  *Tambet Matiisen* December 26, 2015 at 9:45 pm · · Reply →

     You are absolutely right – convolutional layers implement translation invariance too, but they don't throw away the location information the same way as pooling layers do. The feature map produced by convolutional filter still has the information where the filter matched. This opinion about pooling layers has been expressed several times by Geoffrey Hinton: https://www.quora.com/Why-does-Geoff-Hinton-say-that-its-surprising-that-pooling-in-CNNs-work

     Other than that, using convolutional layers makes sense here, because the same object (the ball for example) can occur anywhere on the screen. It would be wasteful to learn separate "ball detector" for each part of the screen. Weight sharing saves a lot of parameters in those layers and allows learning to progress much faster.

This is not the same for all use cases – for example if your task is face recognition and input to your network is pre-cropped face, then the nose doesn't appear anywhere on the image, it is predominantly in the center. That's why Facebook uses locally connected layers instead of convolutional layers in their DeepFace implementation.

3.  *stewie* January 7, 2016 at 4:02 pm · · Reply →

   Thank you so much for this article.

   But I'm still not clear how to get the immediate reward value by looking at the pixels.

   for example, in the end of a game, you observed the final score is 54, what does it mean?

   for most of the games, the higher is better, but for games like UNO, less is better.

   how can you determine the performance of an action by looking at pixels?

   This is even harder for the immediate reward as the score might not be changed.

   -  *mat kelcey* January 8, 2016 at 12:56 am · · Reply →

     In the deepmind result the score wasn't derived from the pixels, it was explicitly provided. reward = change in score. See section 2 of their "Playing Atari with Deep Reinforcement Learning" paper.

     -  *stewie* January 8, 2016 at 9:26 pm · · Reply →

       Thank you so much.

       I found this in that paper: In addition it receives a reward rt representing the change in game score.

       They need human labelled reward rt after all.

       -  *Tambet Matiisen* January 8, 2016 at 9:55 pm · · Reply →

         Arcade Learning Environment has "plugin" for each game, which knows where in memory the game keeps the score. Because Atari 2600 has only 128 bytes of RAM, figuring out the memory location and writing a plugin is not that hard. List of plugins for supported games in A.L.E. can be found here: https://github.com/mgbellemare/Arcade-Learning-Environment/tree/master/src/games/supported

4.  *Alex* January 25, 2016 at 1:57 pm · · Reply →

   Thank you for this valuable post.

   I have become interested in reinforcement learning and I would like to explore its potential applications in business and economic research. In economics we are interested in not only the performance of a model but also its interpretability. Could you please advise me as to whether it is possible in reinforcement learning to understand why a decision is taken or visualise a pattern of action or to interpret the overall result?

   Many thanks,
   Alex

   -  *Tambet Matiisen* January 27, 2016 at 9:02 pm · · Reply →

No, I don't think reinforcement learning helps you with interpretability. Yes, you can say that this particular action was chosen because it had the highest future reward. But how it estimated this future reward still depends on what function approximation technique you use. In case of deep neural networks the guided backpropagation technique (https://github.com/Lasagne/Recipes/blob/master/examples/Saliency%20Maps%20and%20Guided%20Backpropagation.ipynb) has been found useful by some.

5.  *Shi* February 24, 2016 at 12:47 am · · Reply →

Hi Tambet,

Thank you for this article. but I really had some hard-time understanding why the bellman equation iteration thing converges. Then I realized that one of the equations in your article seems to be wrong:

in this equation, you missed the γ (discount factor), without γ, it shouldn't always converge?
$Q(s,a)=r+\max_{a'}Q(s',a')$

as in wikipedia https://en.wikipedia.org/wiki/Bellman_equation#cite_note-NeuroDynProg-4

the discount factor "B" should be $0 < B < 1$

so you can't ignore it. otherwise it won't necessarily converge, am I right?

-  *Tambet Matiisen* February 25, 2016 at 6:23 am · · Reply →

  Yes Shi, you are right, it wouldn't converge without discount factor. I fixed the formula, thanks for spotting this!

  You may spot some other simplifications, for example the definition of Q-value should include expectation over all possible paths (because our environment might be stochastic). I deliberately chose to do this, to keep the post simple and notation minimal. Refer to original papers or links in the end for more formal treatment.

6.  *Shuang* March 10, 2016 at 11:49 pm · · Reply →

Thanks for the post, very clear. I have one question: is there a missing γ in the equation for squared error loss L?

-  *Tambet Matiisen* March 11, 2016 at 6:10 am · · Reply →

  Thanks Shuang, yes I was missing gamma there. Now fixed.

7.  *Pedro Marcal* April 22, 2016 at 5:28 pm · · Reply →

Very useful review of DQN. I think a lot of credit on Q learning should go to C. Watkins, whose thesis at Cambridge established the Q theory that was so useful to deep mind. In any case Watkins theory was an important backgrounder to understanding the delayed reward system in reinforcement learning. I am puzzled as to why Google was granted a patent on deep reinforcement learning given thE general theory formulated by Watkins.

-  *Tambet Matiisen* April 24, 2016 at 5:06 am · · Reply →

  Thanks Pedro for pointing that out. As this is blog post not academical paper, I will leave proper referencing to original papers. Both DeepMind's papers cite Watkin's Q-learning paper from 1992.

8.  *Serra* April 29, 2016 at 9:28 pm · · Reply →

Hi, very good article.
I just compared the network architecture from the DeepMind paper (http://arxiv.org/abs/1312.5602) with the one you listed and they seem to differ. In the paper they used only 3 hidden layers (2 conv + 1 fc). Do you have a different source or am I missing something?
Thanks.

-  *Tambet Matiisen* April 30, 2016 at 5:19 am · · Reply →

  Yes, I'm using network architecture from their Nature paper "Human-level control through deep reinforcement learning".
  https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf

9.  *Ricard Racinskij* June 12, 2016 at 3:55 pm · · Reply →

   Thank you for such a great article!
   I created a toy scene in Unity and adapted an available Torch implementation of your blog post for it (www.github.com/SeanNaren/TorchQLearningExample). It looks like the neural network tends to learn some set of Q values and then sticks with them, even if subsequent behaviour of the agent is obviously suboptimal. Which parameter (lr, epsilon, discount factor os something else) is the key to solving this issue? Does it make sense to use a recurrent net, such as LSTM, instead of feedforward network?

10.  *Andy T* June 20, 2016 at 8:04 pm · · Reply →

    Dear Tambet,

    Thank you for the excellent explanation!
    I have some beginner questions that I hope you could help with:

    The "Deep Q-learning Algorithm" section makes sense, but I am wondering if we actually store random transitions of sufficient length in our replay memory (i.e. four last screen images, the action taken on the last frame, the next reward, and the next state) rather than just one state per entry. Otherwise, how could we recover the frames?

    Next, once we retrieve these four last states, I notice we only choose the last action as the input to the neural network. If a critical step occurred before this time step, it would need to be randomly sampled to be properly captured (That's fine, since the same argument applies as we move into the future, which is why this works to begin with). It seems that this sampling actually provides a chance to ignore updates when there is too little of a difference between actions (i.e. none of the actions really made a big difference).

11. Pingback: Deep-Q – Mémoires·

12.  *Thijs* November 6, 2016 at 10:09 am · · Reply →

    I have 2 questions:
    1) When implementing this, what do you store in the experience replay buffer? The 512 inputs of fc5? That would be a compact state representation.
    2) (if so) then while training, the mapping function between the screenshots and the 512 size state representation stored in the buffer will change -it will learn more relevant representations-. That change will invalidate the state representations in the experience buffer. How is this handled? Is the experience buffer being refreshed like a FIFO buffer?

    -  *Tambet Matiisen* November 6, 2016 at 6:32 pm · · Reply →

      The states are stored in replay memory as raw images. Otherwise it wouldn't be clear how you learn the convolutional layers.

13. *Lorenzo* November 9, 2016 at 3:35 pm · · Reply →

    Thank you very much for your tutorial!
    Finally someone explained easily how backpropagation with multiple action output works.
    I have two questions:
    1) Normally in Q learning we have an alfa in order to decide how easily new information will be overwritten. How can we express that now?
    2) When we are backpropagating with the error on the chosen action and 0 error for all the other one, aren't we giving "false" information to the network? The other QValues might be very far from target value, but we are basically telling the network that they are perfect values (0 error). Is it ok since all Q values will eventually converge to Q*?

    - *Tambet Matiisen* November 10, 2016 at 3:22 pm · · Reply →

      1) The learning rate of network now plays the role of alfa.
      2) You are right. Because we only have information about the action we took, there is now good basis for changing the other Q-values. One way to overcome this is to use dueling architecture (https://arxiv.org/abs/1511.06581) that decomposes Q-value into state-value and advantage. This way the Q-values of other actions change as well, because you are also modifying the baseline state-value. I have basic version of dueling architecture implemented here: https://github.com/tambetm/gymexperiments. If you dig deeper, you can also find policy gradient and A3C implementations, but they are not mentioned in the README 😀.

      - *Naim* November 29, 2016 at 10:05 pm · · Reply →

        I'm trying to implement the dueling architecture, but I'm having trouble understanding how to do backpropagation after calculating the error.
        I can't understand how to calculate deltas of Value function and Advantage functions using the error I got from Q(s,a).
        It's difficult because we get from the previous duel streams to the final Q output without using weights.
        Any help is appreciated, and thanks a lot for the tutorial it has been very helpful. I wish there were tutorials like that for all other machine learning algorithms.

        - *Tambet Matiisen* December 12, 2016 at 12:54 am · · Reply →

          I'm not sure if I got you right, but if your Q-value Q(s,a) = A(s,a) + V(s), then gradient (delta) for A(s,a) is the same as for Q(s,a). The gradient for V(s) is sum of all gradients for Q(s,a)-s. But if you are using modern automatic differentiation library then you really shouldn't worry too much about that. For example here is one implementation of dueling architectures: https://github.com/tambetm/gymexperiments/blob/master/duel.py.

14. *Joshua Staker* November 18, 2016 at 6:25 pm · · Reply →

    I have a question regarding this paragraph:

    "This is a classical convolutional neural network with three convolutional layers, followed by two fully connected layers. People familiar with object recognition networks may notice that there are no pooling layers. But if you really think about that, then pooling layers buy you a translation invariance – the network becomes insensitive to the location of an object in the image. That makes perfectly sense for a classification task like ImageNet, but for games the location of the ball is crucial in determining the potential reward and we wouldn't want to discard this information!"

    But in their architecture, they are performing convolution with stride, so the data is still getting downsampled. My intuition with max-pooling is that pooling happens after the learned filters are applied to the data, so the network learns what information is useful to pass to the next layer, and if downsampling is to be performed, it might as well be pooling so that we also get the invariance

benefits at the same time. What am I missing? How does convolution with stride>1 preserve spatial information better that stride=1 with pooling?

-  *Tambet Matiisen* November 19, 2016 at 6:52 pm · · Reply →

  Joshua, you are right, convolution with stride >1 is basically equivalent to max pooling – see this paper https://arxiv.org/abs/1412.6806. So in the hindsight maybe that comment wasn't too well justified, but at least it makes a good story 🙂.

  -  *Joshua Staker* November 19, 2016 at 7:30 pm · · Reply →

    Gotcha. Thanks so much for the clarification.

15.  *Omer Korech* November 26, 2016 at 6:09 am · · Reply →

    Thanks a lot.
    A small typo: the first word in the following citation from your blog should be deleted
    "maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state."

    -  *Tambet Matiisen* December 12, 2016 at 12:47 am · · Reply →

      Thanks Omer! I guess I wanted to emphasize the case when reward function r(s,a) is deterministic and by adding reward and max discounted future reward from next state you actually get max discounted future reward for this state. But I must admit I was bit lax with definitions and expectations, for the sake of making this approachable to mere programmers as myself.

16.  *fazil* December 6, 2016 at 8:38 pm · · Reply →

    Thank you for the article it is very clear
    I have one question
    Is I can apply Q learning for two agents at the same time for example (avoid collision for two people face to face)

    -  *Tambet Matiisen* December 12, 2016 at 12:57 am · · Reply →

      Yes, you can apply Q-learning to two agents interacting with each other, see our paper on cooperative and competitive behaviour: https://arxiv.org/abs/1511.08779. Although there are no theoretical guarantees of convergence, in practice it tends to work fine.

      -  *fazil* January 16, 2017 at 8:08 pm · · Reply →

        Thank's
        I need an implementation model for Q learning if possible

17.  *Kerawit* January 30, 2017 at 10:26 am · · Reply →

    I love your article. Thank you so much. I'm trying to implement dqn with tensorflow to learn Breakout wih Gym. It is normal that in the beginning Qmax from my networks starts at a very high value (thousands, sometimes millions)?

    -  *Tambet Matiisen* January 31, 2017 at 8:45 am · · Reply →

      That must be a bug. Randomly initialized network produces Q-values around 0 and it should gradually go up.

-  *Kerawit* January 31, 2017 at 1:56 pm · · Reply →

  Thanks to you, I have managed to get the Qmax in the beginning down close to zero by setting stddev of the initial random weights to 0.01 instead of 0.1.

  Is there any particular reason that you chose not to do padding at the convolution layers?

  -  *Tambet Matiisen* February 5, 2017 at 5:20 pm · · Reply →

    No.

18. *Dora* March 1, 2017 at 4:45 am · · Reply →

    Very well explained.

    I'm trying to learn a policy for a game using RL. Sometimes there are invalid moves then the agent gets negative reward & the game state doesn't change. The agent is trained with these experiences.

    But the agent is making the same invalid move again and again. Also with this, the game state is not changing. It is stuck in infinite recursive loop. Any suggestions to improve?

19. *Adam* March 10, 2017 at 1:52 am · · Reply →

    Tambet,

    I'm a bit confused – I have a copy of the Atari paper that describes a different network structure:

    "We now describe the exact architecture used for all seven Atari games. The input to the neural network consists is an 84 × 84 × 4 image produced by φ. The first hidden layer convolves 16 8 × 8 filters with stride 4 with the input image and applies a rectifier nonlinearity [10, 18]. The second hidden layer convolves 32 4 × 4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units."

    This is different from the network you describe in the table above – the network you describe is a five layer network.

    -  *Tambet Matiisen* March 10, 2017 at 7:05 am · · Reply →

      I think you are referring to the earlier NIPS workshop paper. My post was based on the later Nature article: http://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf

20. *vobien* April 2, 2017 at 4:30 am · · Reply →

    Perfect! helpful for newbie about reinforcement learning.
    Thank you.

21. *vineet mehta* April 11, 2017 at 11:03 am · · Reply →

    Superb. I have been searching for this since months. I loved the way you explained each and everything. Please continue doing this. Really helpful!
    Thanks alot again.

22. *Digital Nomad* April 18, 2017 at 3:29 pm · · Reply →

Did you know Google has pattented DQN. Your comment on that?

-  *Tambet Matiisen* April 19, 2017 at 5:25 am · · Reply →

  You can check the discussion from Reddit:
  https://www.reddit.com/r/MachineLearning/comments/4z5oa4/is_google_patenting_dqn_really_justified/

  In short:
  1. it is arguable if the patent is justified.
  2. Google probably did it just to prevent others from patenting it.
  3. probably it will never be enforced, and by now there are many DQN variations + A3C and others, which makes it questionable if the patent applies to them.

23. Pingback: Design an artificial agent to play VikingMUD – Grensesnittet·

24.  *Dan* June 8, 2017 at 6:59 pm · · Reply →

    Hi.

    In the algorithm section:

    select an action a
    with probability ε select a random action
    otherwise select a = argmaxa'Q(s,a')
    carry out action a

    Should the >ss'< ?

    -  *Tambet Matiisen* June 9, 2017 at 8:33 pm · · Reply →

      No, this is the epsilon-exploration. a' might confuse you, but it is just placeholder for argmax.

25.  *Mark* June 23, 2017 at 10:25 am · · Reply →

    Hi Tambet,

    Really useful and helpful article – thank you! One question, is it possible to use deep Q-learning when we have continuous rather than discrete actions? For example, I understand how it works when the possible actions are discrete like 'Up', 'Down', 'Left' or 'Right', but what if say we are trying to train a car to drive autonomously by reinforcement learning and want the NN to give us a steering angle output. Obviously we would want the steering angle which maximises Q but is this achievable when the possible values of steering angle are continuous rather than discrete?

    Thanks for the help!

    -  *Tambet Matiisen* June 26, 2017 at 9:05 am · · Reply →

      Take a look on https://arxiv.org/abs/1509.02971.

26.  *Alexander Yau* July 5, 2017 at 11:28 am · · Reply →

    Hi, thank you, a good post, my question is why the loss function (tt – Q(ss, aa))^2 is right and works?

    -  *Tambet Matiisen* August 21, 2017 at 6:04 am · · Reply →

Squared error is the most common loss function for regression tasks.

27. *Xiaokang Wang* August 19, 2017 at 12:06 am · · Reply →

    Thanks Tambet for very informative and clear blog.
    I have a question about learning rate, alpha, which is involved in the following equation.
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] – Q[s,a])
    But how does the alpha play a role in the regression task as it is not involved in the mean square loss?

    - *Tambet Matiisen* August 21, 2017 at 6:02 am · · Reply →

      Alpha is included implicitly in the method that you use to optimize the loss function. For example if you use gradient descent for neural networks, learning rate plays the role of alpha.

      - *Xiaokang Wang* August 31, 2017 at 10:07 pm · · Reply →

        Hi Tambet,
        Thanks for your reply. The alpha just scales down or up the loss and also the gradient in α(r + γmaxa' Q[s',a'] – Q[s,a]), so it in fact serves as tuning the learning rate when updating the weights.

28. *Gordon Boateng* October 30, 2017 at 6:42 am · · Reply →

    Thanks, Tambet for your post. Very helpful. I am new to RL and I would like to ask whether you have other posts that link DQN to Cloud Resource Allocation for Delay Sensitive Industrial Networks

    - *Tambet Matiisen* October 30, 2017 at 12:56 pm · · Reply →

      I haven't heard much about deep reinforcement learning deployed in industry. There is DeepMind's blog post about improving cooling efficiency in Google datacenters, but it does not give out any technical details. I'd be happy to have this information myself.

29. *Mian Shah* December 8, 2017 at 7:42 pm · · Reply →

    Thank you Tambet. This was a great post. Very helpful.

30. *Tony* February 13, 2018 at 7:59 am · · Reply →

    Your improper grasp of comma usage was highly distracting! Other than that, it's a very good post. Thank you.

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Post Comment

Search Search … Search

Back **March 2018** Next

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Tags

ai bci computer science course data analysis deep learning eeg emotiv epoc neon opengl python reinforcement learning seminar source localization teaching