Neural Network – Deep Q-Learning

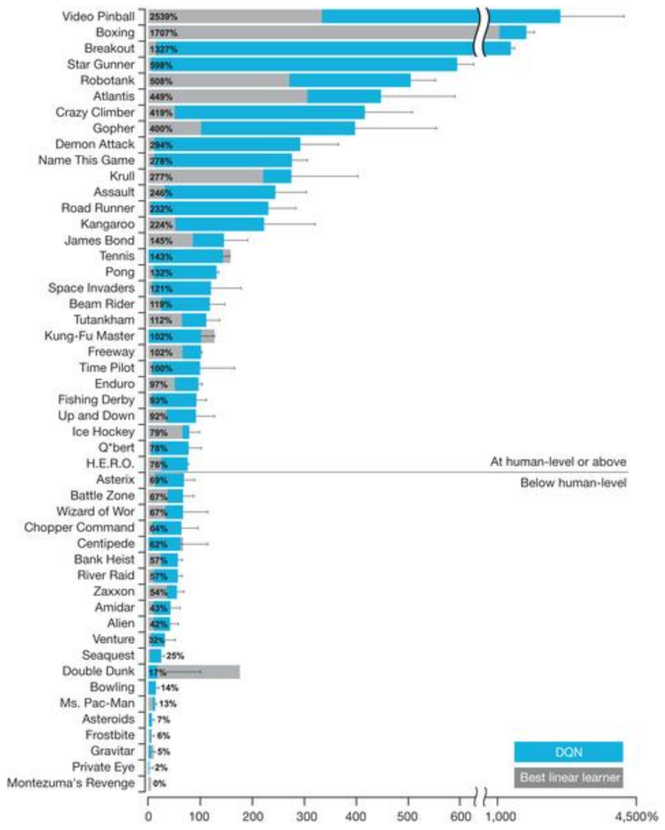
Marco Capotondi, Giulio Turrisi
Course: IA & Robotics

Reinforcement Learning is a technic based on a trial-and error paradigm, which aim is to develop an intelligent agent without the necessity to model it and only using the reward as a feedback to maximize it's performance.

One of the most famous form of it is The Q-Learning algorithm. The agent tries to choose its action to maximize its utility, and to do that usually programs maintain a large table which entry are the Q-Values, an index that link action-state and future reward.

This method is not suitable for a lot of real problem, where the number of possible states is too huge for the memory space available. One way to overcome it is to approximate its form to a function, and using Neural Network can make it possible. This non-linear approximator tend to diverge in this field(one of the causes is the correlations between the observation for example) , but using some trick it's possible to make it converge, obtaining a very good result as we'll show.

This project is based on the paper published in 2015 by DeepMind, called "Human-level control through deep reinforcement learning", which achieved an extremely good result in the converge of the algorithm and in the final score of their agent. Tested in about 49 games, the score can be considered similar and in some case higher to the performance of an human player, In fact, in about half of them the agent learns to get more than 75% of the human total results.



The purpose of this work is to reproduce the result obtained from their work(using Python and Tensorflow), and test it in different scenarios, from a simple GridWorld to a more complicated Atari Games.

In the following part we will introduce some basic concept of the algorithm, the architecture of the neural net used and so on.

1. Q-Learning

In the reinforcement learning literature, Q-learning is one of the most famous algorithm. The problems are modelled as a Markov decision ones, in which we can forget the history of all the past states if we consider only the last one.

The tuple used in this algorithm are <state,action,reward,new_state>. The agent is in a particular state X of the world, performs an action A and goes in a new state X', collecting a reward R from it.

One of the crucial step is how to assign a reward, to allow an agent to reconstruct a good policy to maximize their sum. In our case, the platform used(Gym) is the responsible of their assignment. The way how it works will be described in a dedicated chapter.

The basic element, the Q-Values, is calculated as:

$$Q(\mathbf{x}_t, a_t) = r(\mathbf{x}_t, a_t) + \gamma \max_{a' \in \mathbf{A}} Q(\mathbf{x}_{t+1}, a')$$

This is the sum of the immediate reward gained doing the action a in the state x, and continuing choosing the next action that will maximize the future rewards.

Instead, the table-implementation step, to update the single Q value is:

$$\hat{Q}(\mathbf{x}, a) \leftarrow \bar{r} + \gamma \max_{a'} \hat{Q}(\mathbf{x}', a')$$

So the agent in the first time will begin to explore the world to collect the rewards and updating this values, acting randomly or in a specific way trying maximize them(exploitation step).

The explore-exploitation strategy need to be carefully balanced. In the beginning will be convenience to choose a random action to obtain a lot of different information about the different states, but little by little it will necessary to move in order to achieve the best sum of future rewards, examining more the actual best path founded.

One of the best way to weigh this choice is the e-greedy implementation. Starting with a high probability to act randomly, step by step it will be reduced making the way to acting carefully.

In our case we choose this e-greedy implementation, and we'll approximate this Q-values table with our neural net that it's describe in the following chapter.

2. Neural Nets architecture

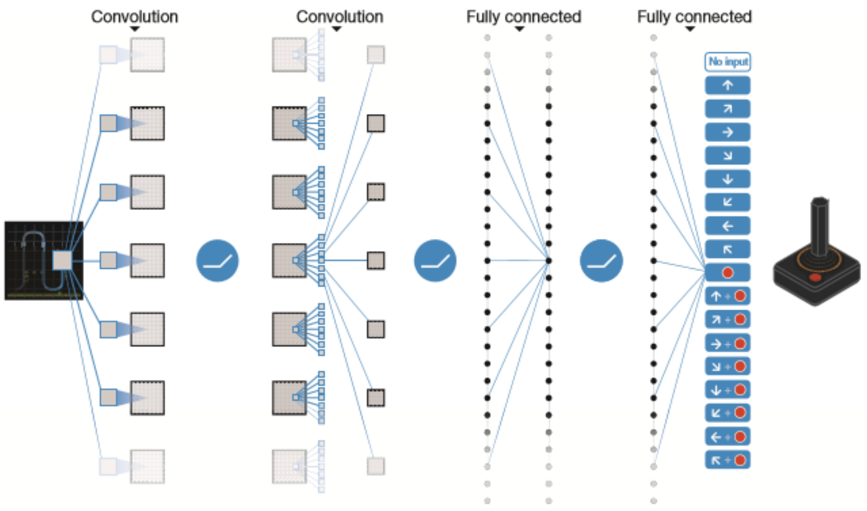
One way to parametrize our Q-values function is to create a Neural Network which consist in an input composed by the state of the world, and different outputs that represent directly the Q-Value of our different actions(one for each of them). To obtain this structure, considering that the state information in our games is carried by an image, we will use a Convolutional Neural Network, followed by two fully connected layer.

Due to the high resolution of the image, it will be necessary to preprocess it decreasing the complexity of our operation. It will be scaled to a 84x84 format, in a grey-scale color. In the case of **Breakout**(an Atari games) our input will take three of this images each time(three sequential and different state), because the Net need to know how the ball moves to act consciously! Instead in the **Gridworld** game, due to the fact that it's not dynamic, there is no reason to perform this step: just one frame is all we will need.

The first layer of our Conv Net, will convolve 32 filters of 8x8 with stride 4, applying then a relu. The second ones will convolve 64 filters of 4x4 with stride 2 and the last ones 64 filters of 3x3 with stride 1, each one followed by the same rectifier.

In the end, the output of this first conv part will enter in two sequential fully connected layers, which outputs will be as said before, in the same number of our game actions.

- Breakout: 6 actions
- Gridworld: 4 actions
- CartPole: 2 actions



Note that, we tried first our algorithm without the conv part, due to a lower complexity. In fact the third game, **CartPole,** uses as a state not an image but a little array of numbers that describes the actual position - angle of the pendulum. After testing it, we moved to the most complex environment provided by the other two games, using the Conv Part as explained before.

But how the backpropagation is performed? What is it the loss used? Before explain that, we need to describe two additional parts which permit Deep Mind to achieve a very good success. They are Experience Replay, and the use of a second network, called Q-Target.

Experience Replay:

As written in the beginning, one problem that causes instability of using this algorithm with a neural net, is the correlation between sequential states. One easy way to remove it, is to train our network not with the actual state, but with a randomly sampled one obtained in the past.
So, every time we performs a step, we put the new state in a buffer(called in our case replay_buffer), and performing the backpropagation sampling randomly from it. This addition requires to populate it a little bit before starting training our network, and so in the beginning there are a fixed number of step where the backprop. step is not performed. (Actually, we put the tuple <state,action,reward,new_state>).

Target Networks:

An additional implementation is the creation of a second neural network(identical to the first one), that at the beginning has the same weight/bias values.

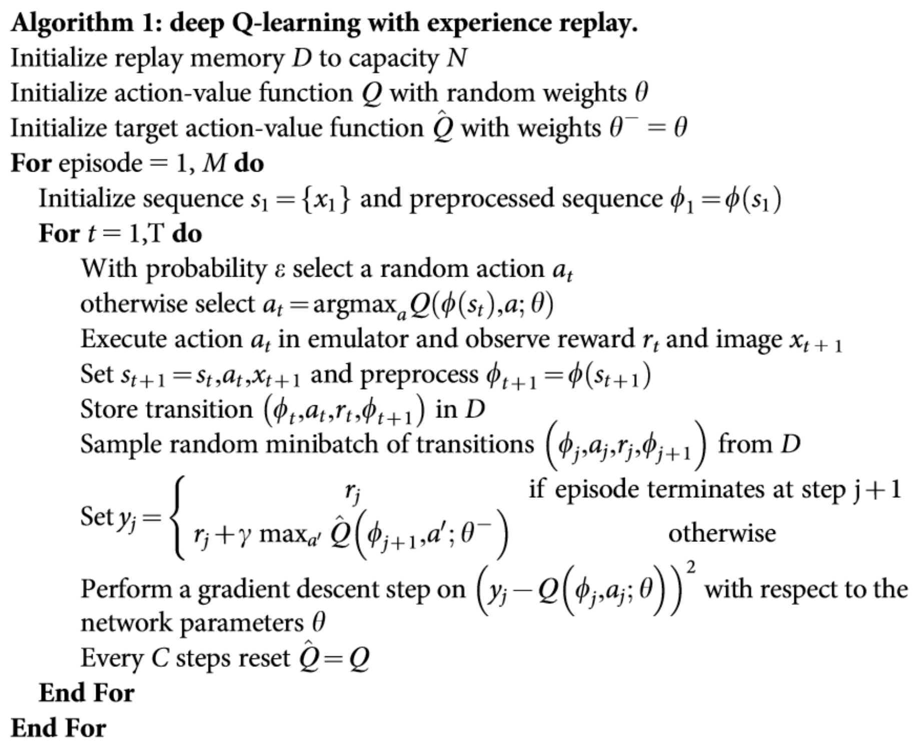
It will be used to calculate the loss to perform the backprop. step(done on the Q-Main).

$$\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$$

Yj will be calculated using the target network(sampling in the replay memory, taking the next-state and putting it as input in the Q-Target). Instead the second element using the Q-Main net, from the same sampling but using the old state.

The reason why DeepMind choose to implement it, is to increase further the stability of the algorithm. In fact during all the step, our Q-Main network values shift constantly, and if we use a constantly shifting values to adjust our network values, then it can give a large instability problem.
So we maintain the weight-bias of the Q-Target constant, and after some number of steps we update their values copying them from the Q-Main ones.

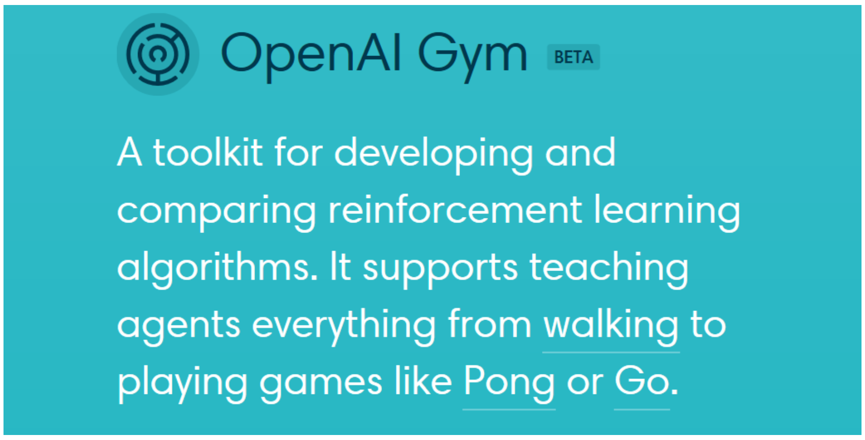
The complete pseudo-code of the principal algorithm is explained before(taken from the paper):



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

3. Gym – an openAi Environment

For testing our implementation, we used a fantastic tool provided by “OpenAI”, a non profit-organization with the aim to enhance the artificial intelligence research. They created Gym, a toolkit for fast prototyping algorithm, due to the easy access to a large number of possible environment where test it. There are present a lot of different games, with a large range of difficulties, where you can find everything from a simple game to Doom.



OpenAI Gym BETA

A toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

We used it to test our algorithm on the environments “CartPole” and “Breakout”, but in the beginning we used a more simple game(“GridWorld”) found on the net.

To interface with this tool, there are only few lines of code needed, easily explained in the table below.

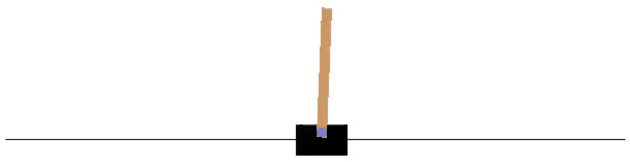
```
#importing the libraries
import gym
#create the environment
env = gym.make('Breakout-v0')
#resetting it
state = env.reset()
#after choosing an action, performing it and obtain reward and next state
obs, reward, done, _ = env.step(action)
```

How said before, it is Gym who takes care of reward too, giving the possibility to concentrate on the algorithm rather than modelling the world.

Now we’ll describe the different type of games used(starting from “CartPole”), due to the fact that it is the most simple because don’t need a convolutional net.

4. CartPole

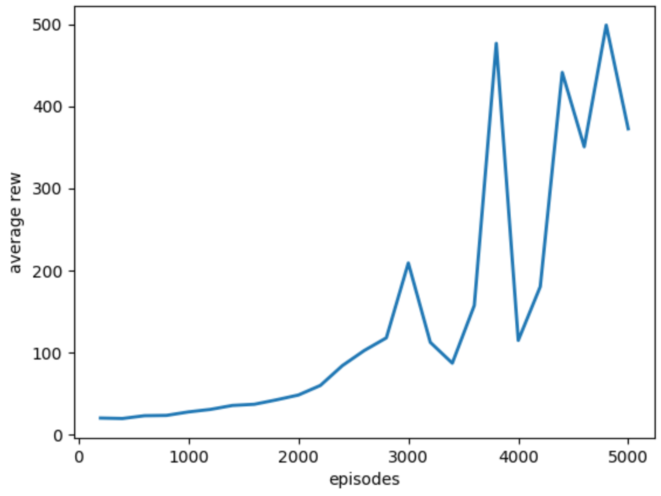
CartPole can be considered a simple task, due to the fact that we have only two action available and that the state is a set of number(not an image!) which contain information about the position of the link to stabilize.



From the Gym documentation:
“A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright.”

The learning part was performed over **5000 episodes**, each ones formed by **1000 max_epLenght**, which takes about 5 hours to end. The **replay_memory_size** was settled to **10000** elements, the **annealing_steps**(the rate with which reduce act from random action to the most useful ones) to **100000**, , **1000 target_freq**(after this number of steps, the Q-Target weights are copied from the Q-Main), and the **pretrain_steps**(with which we can populate the replay buffer before starting training) with **6000**.

(Remember, in the final episodes we continue to permit to choose some random action to explore)



After that, the test was done with the same setup, so the maximum score achievable during a single episode was 1000 points(one for each steps).

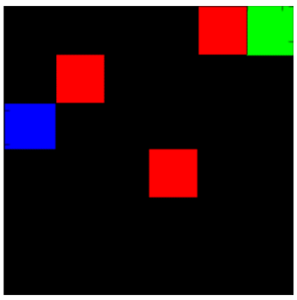
The result obtained was very good. Performing the test over 200 episodes, the average point obtained was 864, with a percentage of fully completed-rewarded episode of about 90%.

Due to the good result obtained, we moved to the next step: inserting the Conv. Part and passing to a games with an image as a state.

N.B the values used from setting the parameter in bold, were just found after some experiment. Due to the difficulties of our machines to test the different setups(too much time needed!), we provided only the best one founded, without any clear comparison that it's beyond the aim of this project.

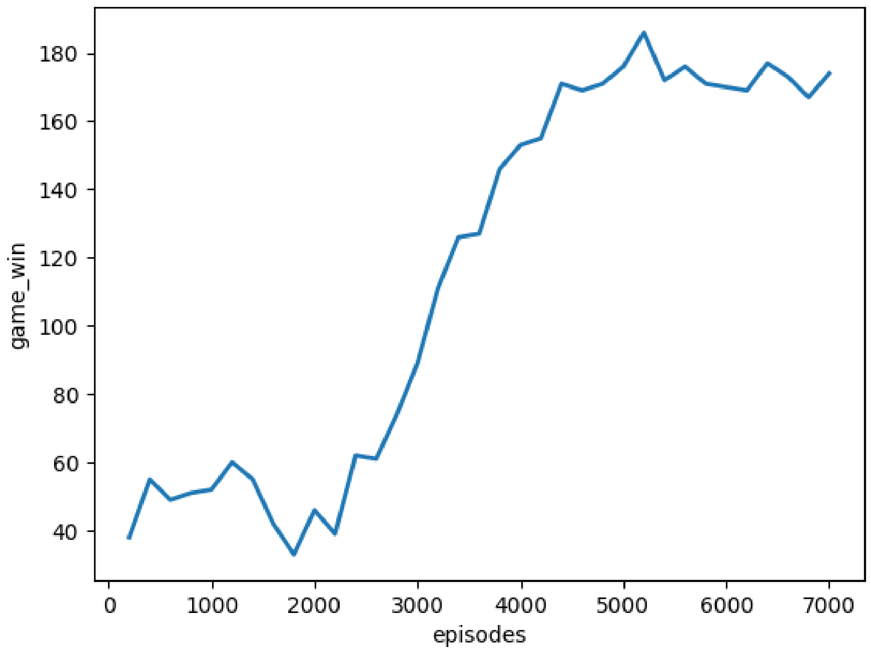
5. Simple Gridworld

The aim of this game, is to conduce a little blue square(our agent) to a green square goal, without falling into the red ones(three holes). We have here 4 actions, each one for the four possible direction(up,down..).



The setup used was: **7000 num_episodes**, **60 max_epLenght, 20000 replay_memory_size, 100000 anneling_steps, 15000 pre_train_steps** and **1000 target_freq**.

The number of step of one episode is smaller than the previous case because here when we reached the goal, we can end the episode easily.
The size of the net is equal with that described in Chapter 2. Here we don't rescale our image, and we put it in input with the full rgb color.

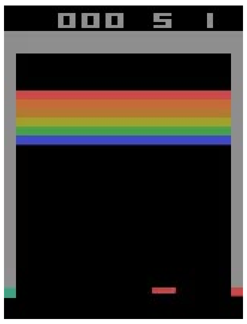


After this fixed training, that was more demanding due to the use of image as a state and the more operation performed from the Net, the result obtained was good enough to stop it and perform a test. The number of winning episode(over 200), was 173.

In this example is clear how good is our implementation. It performs very well, and the agent learn a good policy only seeing an image!

6. Breakout

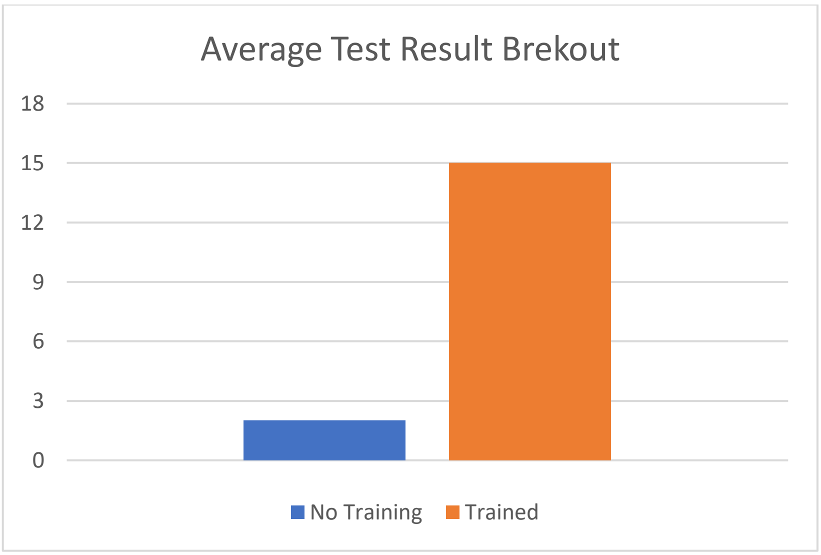
Breakout is clearly the most demanding environment used for this project. It's a famous Atari game where a little spacebar need to hit with the ball some brick present in the world, without missing him. The agent has 5 lives, and each colored bricks have a different rewards.



Here the state is an RGB image of the screen, which is an array of shape (210, 160, 3). So here we needed to preprocess the image(to low down the hardware requirements), to a 84x84 grey-colored image. As said before, due to the fact that the net need to learn how is the direction of the little ball, we stack 3 sequential frame together in input. We didn't change the architecture of the net, using the same values used for the GridWorld game(of course we re-train our network separately for each games).

Here, the parameters used were: **30000 num_episodes**, **1000 max_epLenght, 20000 replay_memory_size, 1000000 anneling_steps, 15000 pre_train_steps** and **100 target_freq**.

Due to the more complex environment and the lack of dedicated GPU for computing, the training was very slow. It took days to complete, and in the end we reached a poorer result compared to the one obtained by DeepMind. The average points performed during an episode was about **15**,(with a maximum of 31) but the training episodes usually used in literature to train are much larger than ours! It's quite impossible to reach such level of training(many people used a setup with a sli-configurations of dedicated GPU, training for days), and we can't simply do that. **But the most important part is that in the testing the results showed us a good improvement, that cannot be casual of course(the beginning score was only 1-2 point at episode).**



7. Final Considerations

As showed from the previous chapter, the resulting obtained is really positive in the first two simple games, and quite encouraging in the more complex Atari environment. It's amazing how we can just use an image to train our agent, obtaining a good result in a lot of different environment without changing our structure. DeepMind trained the same model over all the different games used, and it adapted itself automatically to every ones. Same thing here, the code used for the three games it's essentially the same, with a little tuning of the parameters of course.

One interesting thing is that the unification of Reinforcement Learning and Neural network is not only suitable for discrete action. It is possible to extend it also to the continuous realm, where the output of our Net is, for example, a torque for a joint choosen from a bounded interval. (Deep Deterministic Policy Gradients).

- https://gym.openai.com/
- http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html