ORC Project: Deep Q Learning on two joints manipulator

Giovanni Lorenzini
223715
giovanni.lorenzini@studenti.unitn.it

Simone Luchetta
223716
simone.luchetta@studenti.unitn.it

Abstract

We review the usage of a Neural Network in order to perform non-linear function approximation for accomplishing the Deep Q-Learning technique. The goal is to let a robot composed of a number of arbitrary joints learn a proper swing up manoeuvre from its experience, that is collected by running hundreds of simulations over.

Several experiments involving parameter tuning and architectures breakdown are discussed.

The code is available at the following Github repository: https://github.com/lorenzinigiovanni/orc-project.

1. Deep Q-Learning

Tabular methods such as Monte Carlo and Temporal Difference are used to solve problems for which states and actions can be discretized and represented as arrays and tables, along with costs associated to those states and the outcome of each action taken. The main idea is to solve Bellman equations by leveraging on iterative methods and dynamic programming so that an agent can deal with the given problem by learning an optimal policy.

The drawback of this approach is that actions and states might explode in space, leading to the impossibility of storing the vastity of entries in the table. Thus, it is needless to say that any continuous or large-enough discrete state space makes the computation of the Q-values unfeasible.

To render this idea, let's give an example: the chess game scenario. It is estimated that chess has around 10^{120} states, which would imply storing a table that is far beyond technologies possibilities.

For this reason, non-linear function approximators such as Neural Networks are deployed. This approach addresses the problem by estimating the Q-value of a state by learning the features of the state space set. Using this framework, it is possible to tackle the problem by easing computations and memory constraints.

Thus, the problem is re-arranged as in the following equation:

$$\phi(x_t, \theta)[u] \approx Q(x_t, u) \tag{1}$$

Where the approximated Q-value for state x_t taking action u is the result of a forward pass to the Neural Network $\phi(x_t, \theta)$ of the state x_t, using parameters θ, indexing action u. The setup of our function approximation model is depicted in Fig. 1.

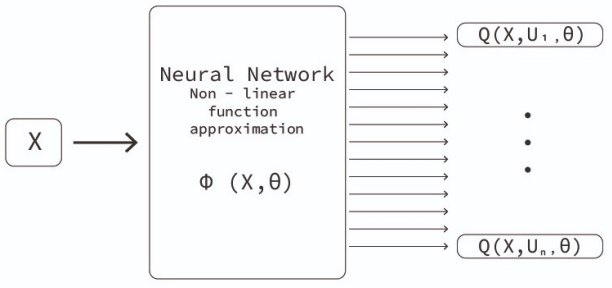

Figure 1. Function approximation scheme.

Given a state x_t, the agent will perform an action $u^\star$ by choosing the control that minimizes the associated approximation of the Q-value $Q(x_t, u_i, \theta)$.

The training procedure consists in collecting data from numerous simulations, called *episodes*, whose lengths span up to a defined number of simulation steps. The agent follows an ϵ greedy strategy in order to favour both exploitation (following what he knows) and exploration (reach out for something new). The trade-off between the two is determined by a parameter that decays exponentially as the growth of the episodes, giving more importance to explore new states in the first part of the training run. The mathematical trade-off is expressed as follows:

$$ExplorationProbability = \max(0.001, e^{0.01 * ep}) \tag{2}$$

Where ep is the counter of episodes, 0.001 is the minimum exploration probability and 0.01 is the exploration decay.

Training data are stored in an experience replay that contains a tuple formed by:

- X: current agent's state;
- U: control chosen by the agent;
- $Cost$: cost of the step starting from the current state X choosing action U;
- X_{next}: state reached after being in X, performing a step with control U.

The usage of this experience replay is as follows: samples of entry tuples are collected from the data structure, and are used in order to update the Network by following the update formulae, as in Eq. 3, 4, 5.

$$target = cost + \gamma \; min_u \; Q_{target}(X_{next}) \tag{3}$$

$$current = min_u \; Q_{main}(X) \tag{4}$$

$$loss = SmoothL1Loss(current, target) \tag{5}$$

2. Our Networks

The code is developed in Python, using Pytorch [2] for embedding the Network architectures.

In order to keep the training procedure more stable, there is a trick for which two Networks structured with the same architecture are employed. One is referred to as the *main* Network that takes advantage of the other one, named *target*. The latter is just a copy that is periodically synchronized with the weights of the former once every 100 simulation steps. The *target* Network is used to compute the Q-values of the next states in the Bellman equation for the *main* Network, and its purpose is to assure a stable reference during the learning process. The idea is documented in [1].

The loss function for updating the *main* Network's weights is the one described in Eq. 5, that is also suggested in [1]. This is referred to as Huber loss and it penalizes quadratically around an interval close to zero, linearly elsewhere. A representation of this function is depicted in Fig. 2.

Hyper-parameters used to train the *main* Network for accomplishing the swing-up manoeuvre for a 2-jointed robot can be found in Tab. 1.

Notice that a comparison between two methods of storing model-weights using the same architecture has been conducted. For the first one we thought of saving the Network's parameters every-time that the robot performed a swing up manoeuvre obtaining the lowest cost possible starting from the relaxed-down position. Instead, the second one saved the weights at the end of the whole training process. Overall we think that both these methods achieve

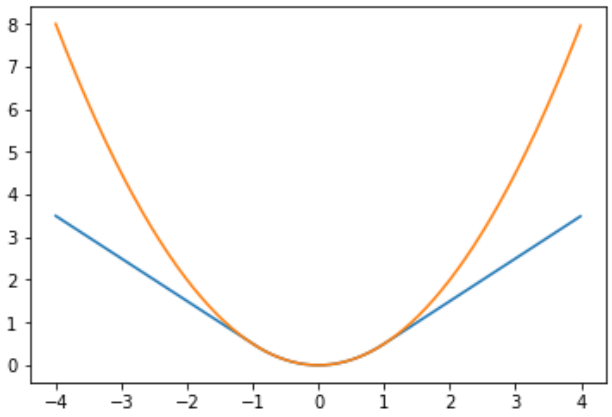

Figure 2. Huber Loss (in blue) compared to squared error (orange).

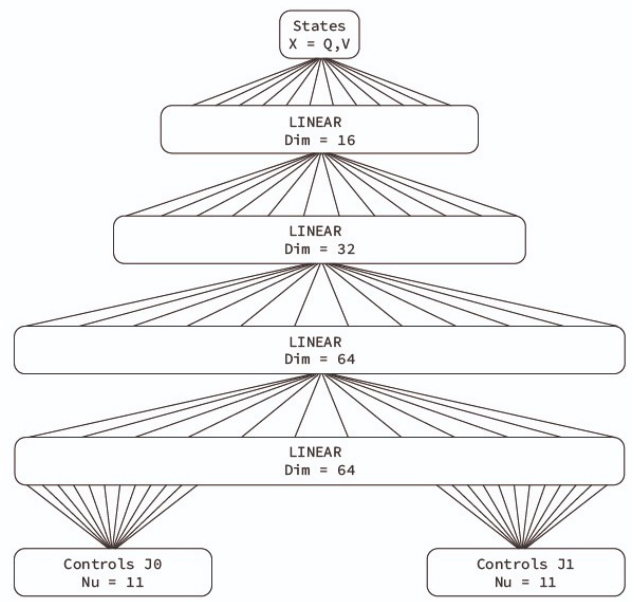

Figure 3. Neural Network's architecture. Two heads are appended after the last linear layer, out of which controls for governing a two jointed robot are extracted.

the goal, but the former is more expensive as we run a simulation at every end of an episode, and can learn far less as subsequent runs are discarded and the weights of the model are updated no more.

This architecture works fine if the problem is relatively small. Hereby we suggest a few tweaks that could help if the complexity of the problem enlarges:

- Bigger neural network: a neural network with more layers and weights can be more expressive and approximate the Q function better. In any case we need to avoid over-fitting.

Learning rate	10^{-3}	Learning rate of the Neural Network
Update steps	100	Update the target Network once every update steps
Cycle	4	Compute loss for the main Network once every cycle steps
Min steps	1024	Minimum number of steps to train the model
Memory	10^6	Memory of the replay buffer
Batch size	64	Size of the batches passed in the forward pass
N episodes	2500	Number of training episodes to fully accomplish the goal for 2-jointed robot
Episode length	500	Length of a training episode
Discount	0.9	Gamma discount factor
Initial exploration probability	1	Set exploration parameter
Exploration decay	10^{-2}	Influence how much exploration decreases exponentially over the epochs
Minimum exploration probability	10^{-3}	Minimum exploration threshold
Nu	11	Discretizations of the controls u
U max	2	Maximum control torque

Table 1. Hyper-parameters for controlling a 2-armed robot.

- Lower δt: we found that the time sampling parameter is crucial for getting smoother episode runs (the lower the better, but at the expense of an increase in compute time).

3. Hyper-parameter tuning

During debugging sessions, we found out that setting up the exponential decay parameter is of crucial concern. Empirical result show that a good value is one that makes reach the exploration probability around zero at the number of training episodes.

We chose Adam as optimizer as it is seemingly the one that converges faster to a solution and it is an evolution over the RMSProp that is also used in [3] to cover the Deep Q-Learning technique.

Regarding the simulation environment, setting the maximum torque as default ($\tau = 2$) implies that the robot needs to be trained for a very high number of epochs, otherwise the Network learns to minimize the costs by oscillating the joints as if they were clock hands, either clockwise or anticlockwise. To us, it seemed that there might be a lack of power, so we thought about increasing the maximum torque parameter $\tau = 5$. Results show that this trick is effective, as the robot can learn to balance itself in the swung up position in a considerable fraction of the training epochs with respect to the under-powered runs.

Parameters used during training are reported in Tab. 1.

4. Results

4.1. One joint robot

The robot with a single joint take only few epochs to train (in less than 50 epochs can perform the swing up manoeuvre) and work quite well. The behavior over time for the single jointed robot is shown in Fig. 4. The joint angles and the action applied to the joint are represented in the upper and lower charts respectively. The x axis is the step of the simulation. In Fig. 5 there are the value and policy tables. From the value table it is possible to observe the fact that the minor cost is found along a diagonal, where either the robot is oriented upwards or has the speed to reach that position. Instead, looking at the policy table we can see that:

- Along the main diagonal, controls permits the robot to keep itself around a balancing position.
- When the robot is oriented downward (for joint angles $q \in [-3.14, -1.6]$ or $[1.6, 3.14]$ radians circa), the applied control is flipped when the speed changes sign. This is due to the need of the robot to gain sufficient inertia in the next swing.
- Around -1.5 radians with negative speed and around 1.5 radians with positive speed, there is a zone where the robot is uncertain on what to do: on a side it has not enough inertia to perform the complete swing up manoeuvre and so it applies an action that speeds the robot up so it can try to complete the manoeuvre in the next steps; on the other side it has probably overshot the vertical point and applied negative torque to return to the erect position.

4.2. Two joints robot

The robot has initially been tested with the torque limited to 2. It takes longer to train to achieve a nice result and finds a manoeuvre that enables it to stay steady upwards. We show evidence of this in Fig. 8: with only 500 epochs, it is possible to see that it seeks to swing and rotate around its base like a clock, as it seems that it is the only strategy to minimize the cost. Instead the network trained for 2500 epochs makes the robot properly execute an appropriate swing up manoeuvre, as can be seen in Fig. 9.

Setting the torques limit to 5 simplifies the manoeuvres that the robot has to take and so it can perform the movement with a training of only 500 epochs as in Fig. 6. On the other side, with a training of 2500 epochs it goes to the

erect position faster and stay more centered, as can be seen in Fig. 7.

Some papers suggest also to include a *termination-state* flag to boost up training performances, that allows the Network to take directly the cost associated to a terminal state without delving deeper at the discounted horizon. This *termination-state* truncates the updates of the target *Q-Values* in Eq. 3, so to include only the term associated to the cost of the current state-action pair; it is set to true whenever an episode ends. We conducted some experiments training the Network with and without the *termination-state* stored in the replay buffer. Using same hyper-parameters on different runs, we observed that for our scenario it is meaningless to keep this flag stored as employing the *termination-state* flag only worsened the training experience. Comparisons of this behaviors are presented in Fig. 6 and Fig. 10 respectively. For each run, Networks are trained for 500 epochs and the maximum torque parameter for the robot is set equal to 5 for achieving faster convergence.

Other papers suggest to save the network weights when the network performs better. In order to verify this, we put the Network in evaluation mode at the end of each epoch, and placed the initial joint configurations to start from the relaxed down position. Each time the succession of steps retrieved a cost better than the current best, Network's weights had been updated. Unfortunately, this technique has not worked for us as the network learn less and the robot cannot perform the swing up manoeuvre. Proof of this behavior is reported in Fig. 11.

The use of 1-dimensional convolution layers is also investigated. Compared to a pure run that uses the multi-layer feed-forward architecture, and under the use of the same hyper-parameters, we noticed that the hybrid-convolutional Network reached the equilibrium point performing severally less swings, but arriving decidedly more crooked.

4.3. Three joints robot

The Network has been also trained for a three-jointed robot. Our guess is that the Network used for the other two runs did not contain enough parameters to favour a good behavior for the robot, hence we appended another larger feed-forward layer. Still, the hyper-parameters employed remained the same ones inserted in Tab. 1, with exception of the maximum torque (*U max*) which was set to 10 and for the number of episodes (*N episodes*) which were raised to 10000.

In Fig. 12 it is possible to observe the behavior of the robot after a training run of 10000 episodes is completed.

5. Conclusions

The robot learns to perform the swing up manoeuvre successfully. Many tests have been conducted in order to investigate the goodness of our Neural Network.

The training process is quite fast (circa 5 episodes per second), and faster with respect to the Q-Learning exercise we have done in class, even despite the fact that the Network has to deal with continuous instead of discretized states which should be more costly in terms of compute. We think our method is highly optimized as we update the network weights using large batches of data and using fast matrix operations.

We observed that if training episodes are kept low, the agent learns a rough $V(x, \theta)$ function and a policy that is not able to perform the swing up manoeuvre. However, by increasing the number of episodes we noticed that the agent can recover a fairly good policy that permits to perform the manoeuvres needed to find the equilibrium point, but the robot might be have its joints slightly crooked. With an even higher number of episodes the accuracy of the $V(x, \theta)$ function and of the policy $\Pi(x, \theta)$ increases, leading to smoother movements that make the robot stay within the desired pose constraints more precisely.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. 2

[2] PyTorch. Pytorch source code. https://github.com/pytorch/pytorch. 2

[3] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network, 2017. 3

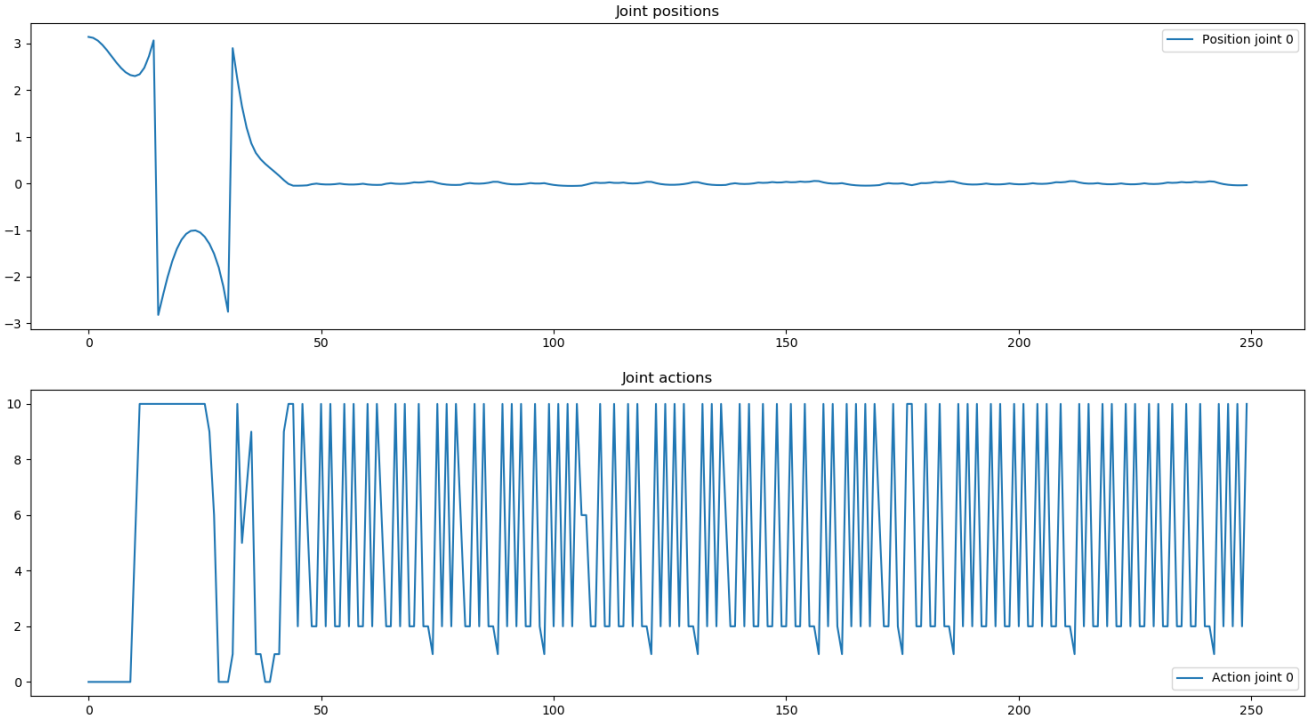

Figure 4. Joints: 1, episodes: 500, torque: 2. Joint angles (above) and joint torques (below) are illustrated.

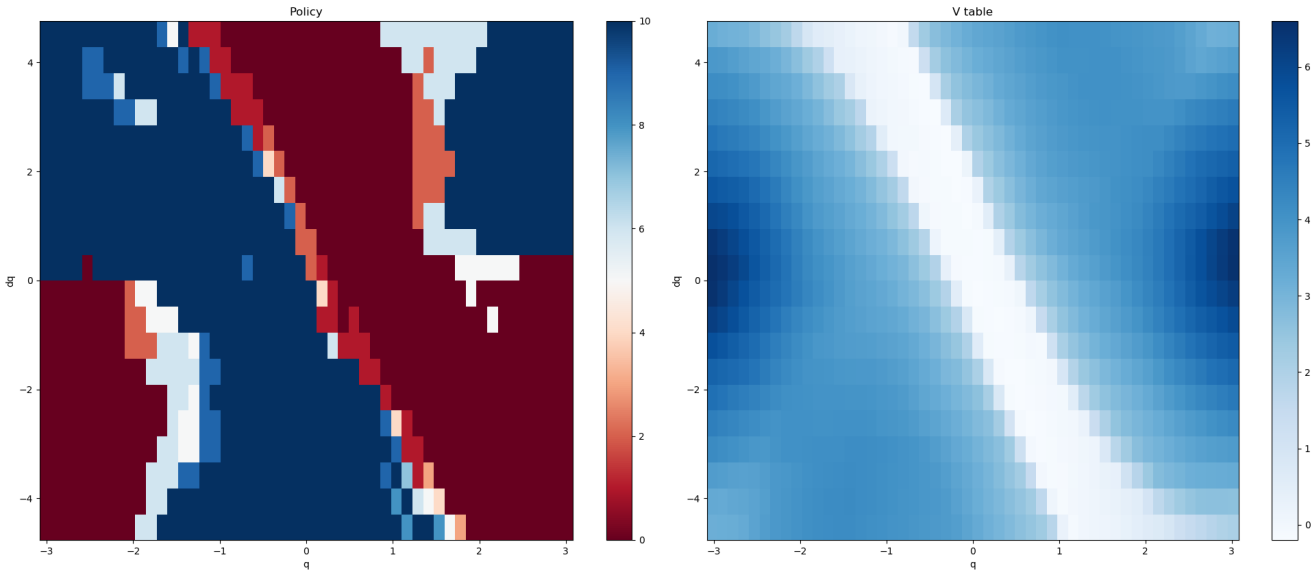

Figure 5. Joints: 1, episodes: 500, torque: 2. Policy table (left) and Value table (right).

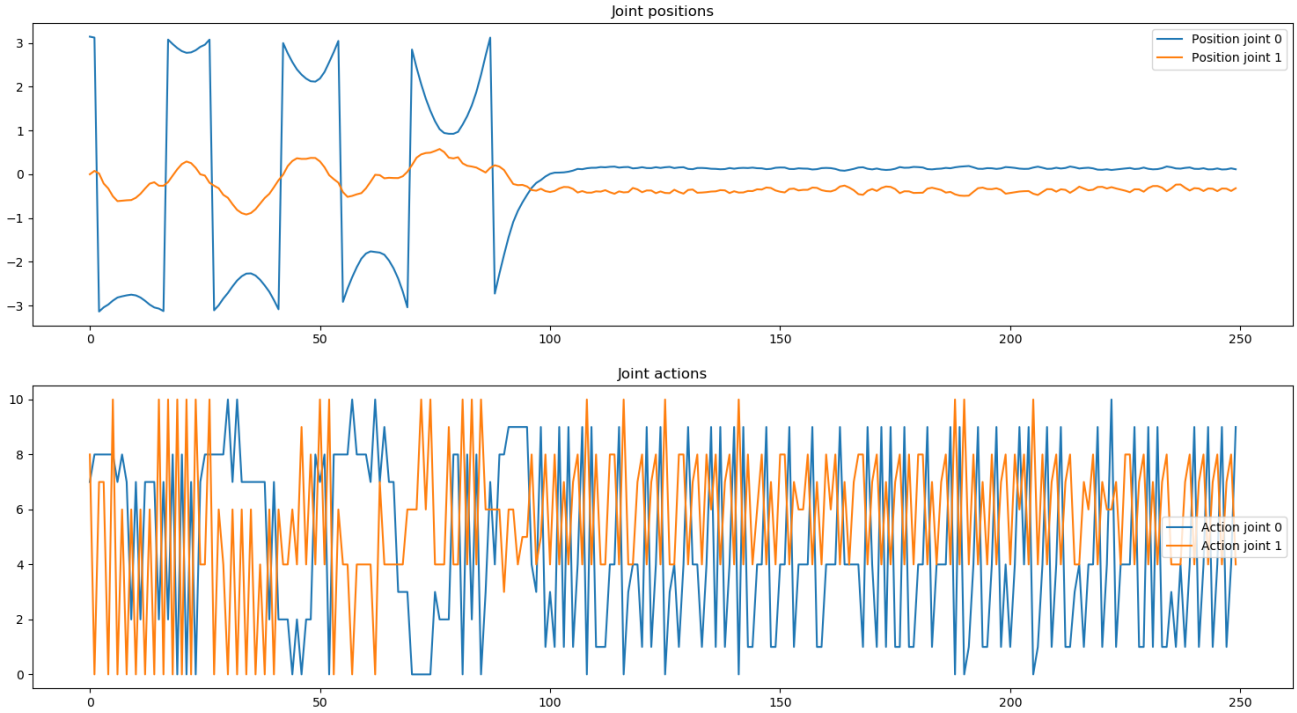

Figure 6. Joints: 2, episodes: 500, torque: 5. Joint angles (above) and joint torques (below) are illustrated.

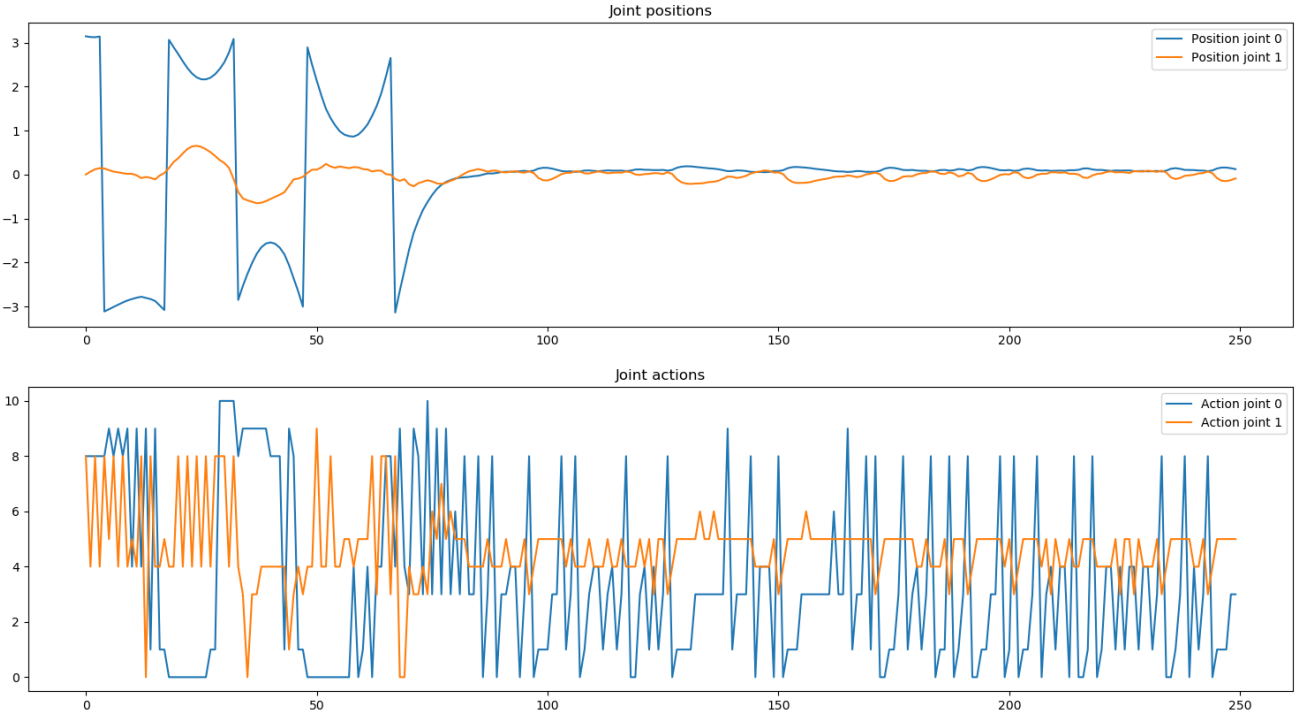

Figure 7. Joints: 2, episodes: 2500, torque: 5. Joint angles (above) and joint torques (below) are illustrated.

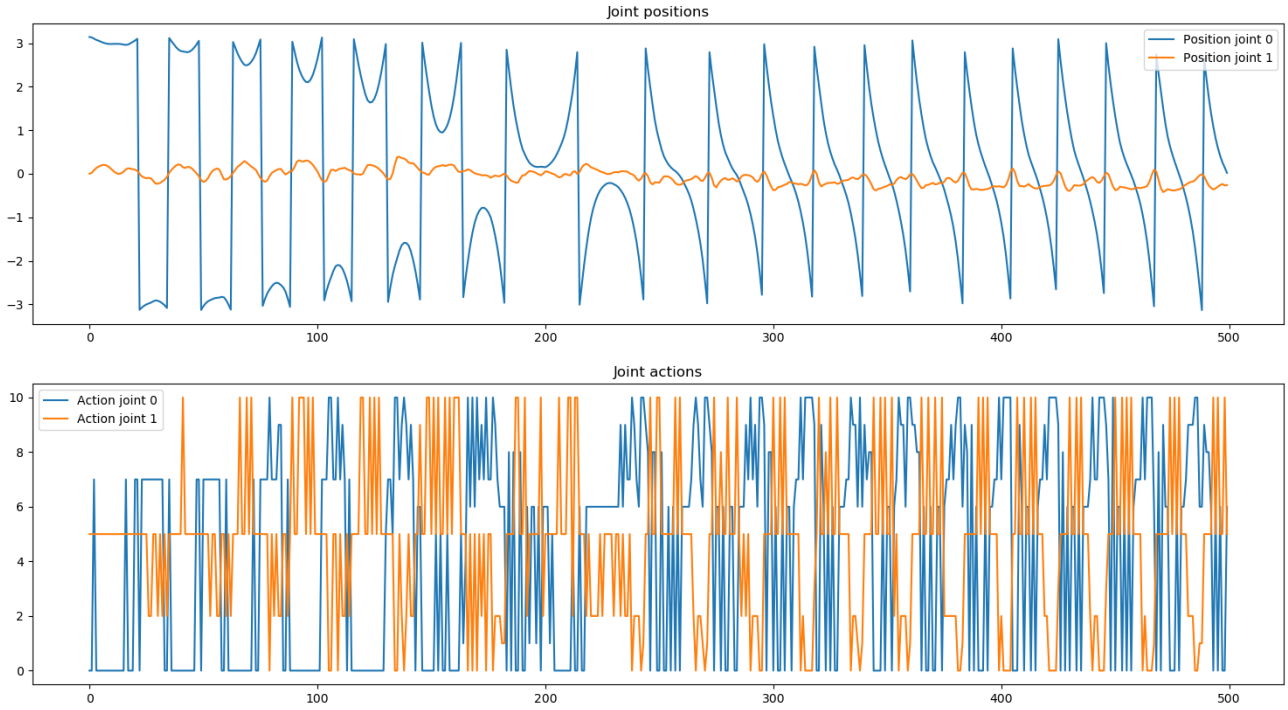

Figure 8. Joints: 2, episodes: 500, torque: 2. Joint angles (above) and joint torques (below) are illustrated.

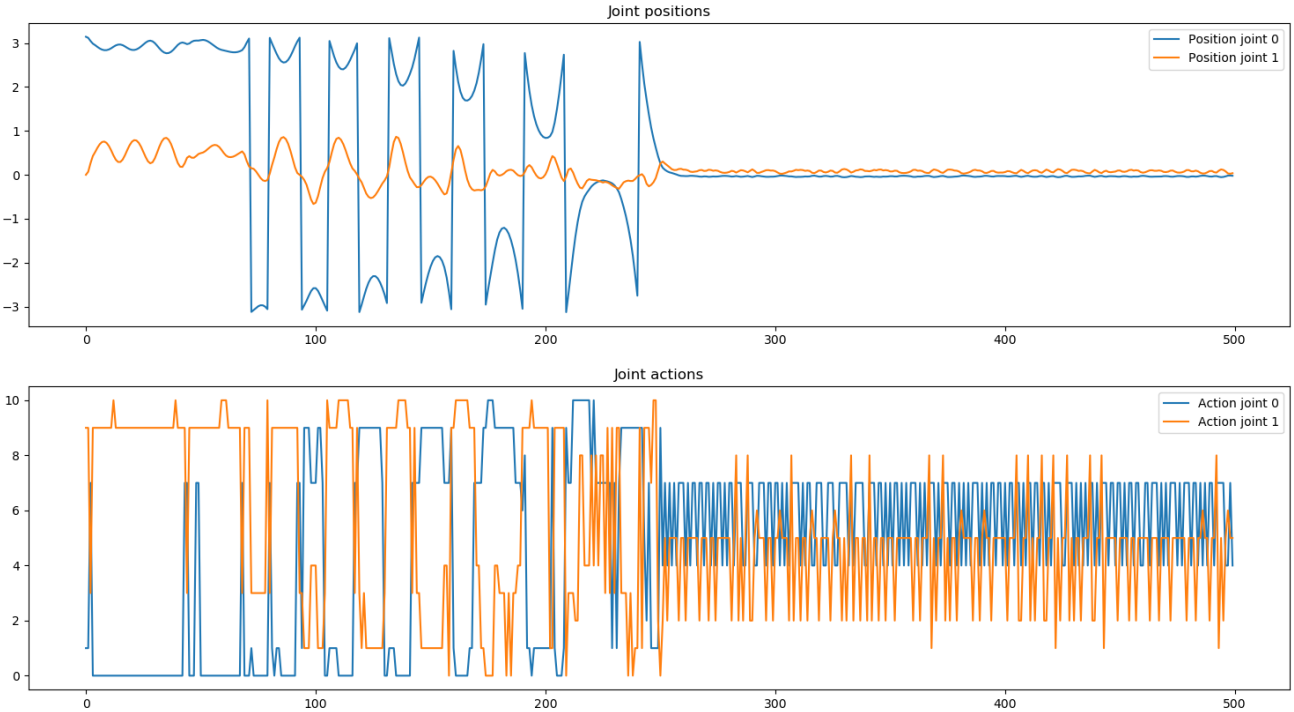

Figure 9. Joints: 2, episodes: 2500, torque: 2. Joint angles (above) and joint torques (below) are illustrated.

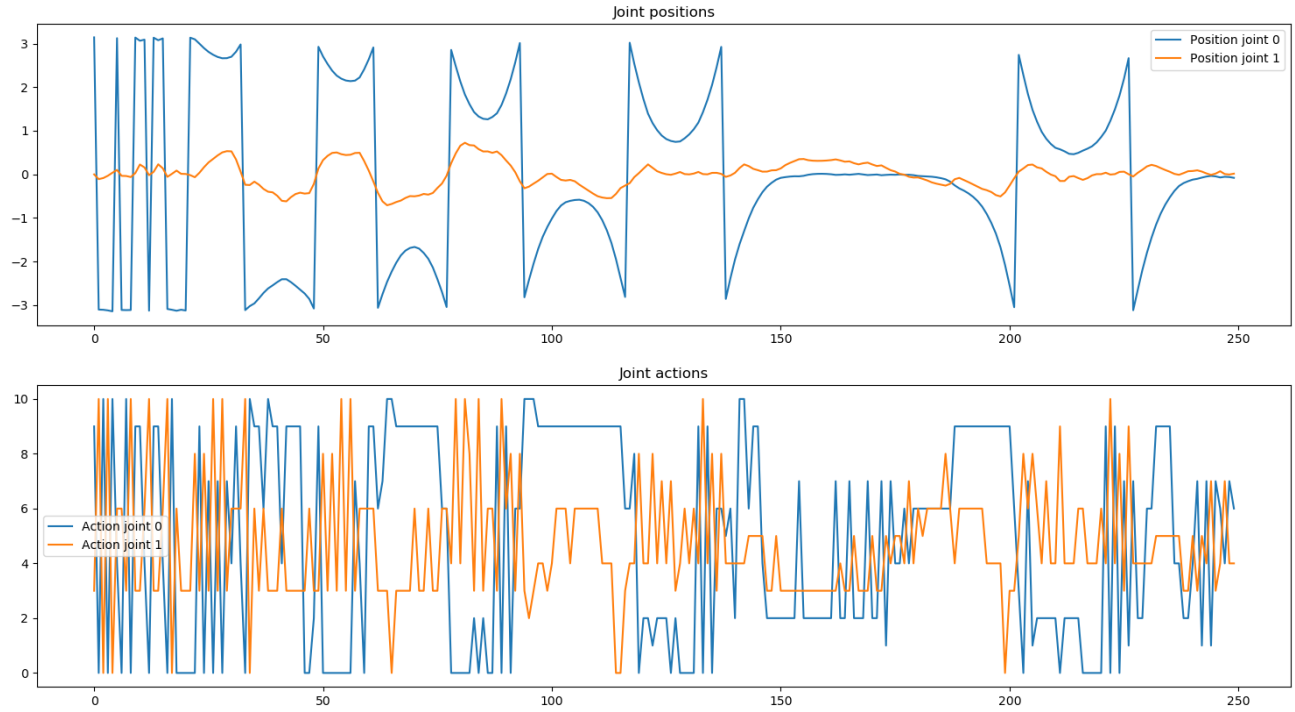

Figure 10. Joints: 2, episodes: 500, torque: 5. Here the done flag is used. Joint angles (above) and joint torques (below) are illustrated.

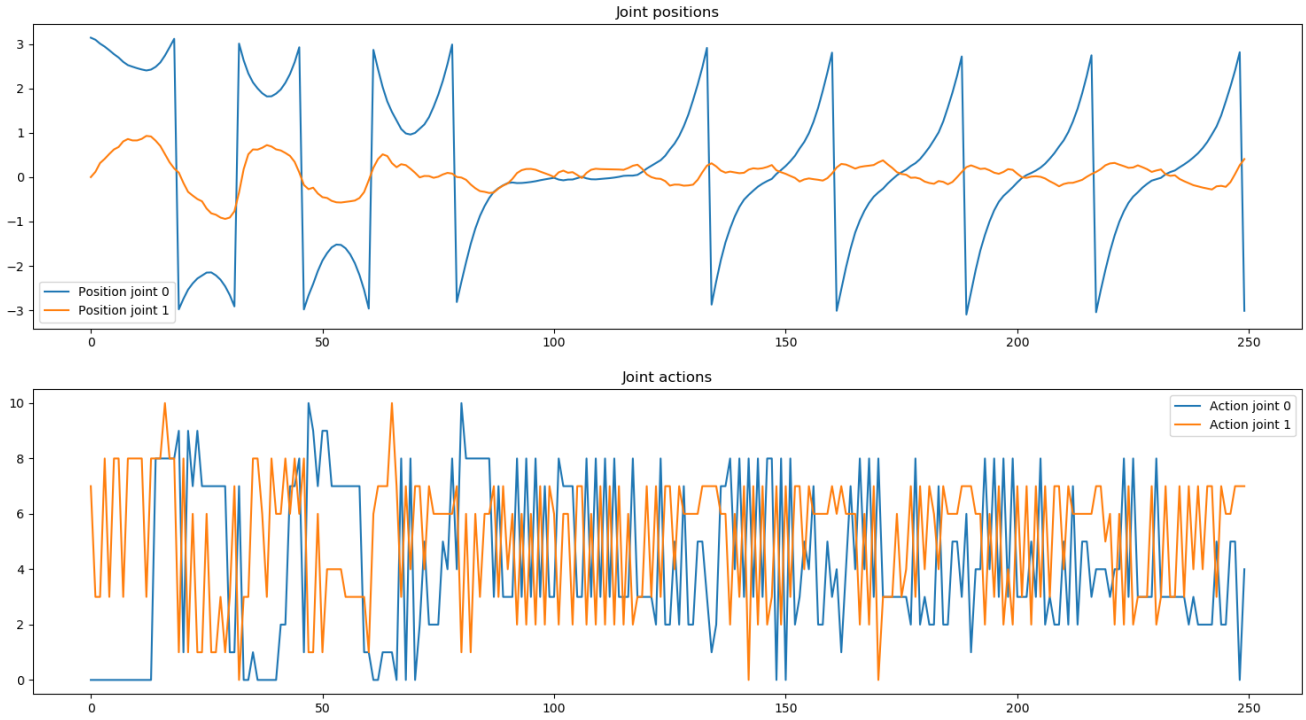

Figure 11. Joints: 2, episodes: 500, torque: 5. Here the network performing the best run is saved. Joint angles (above) and joint torques (below) are illustrated.

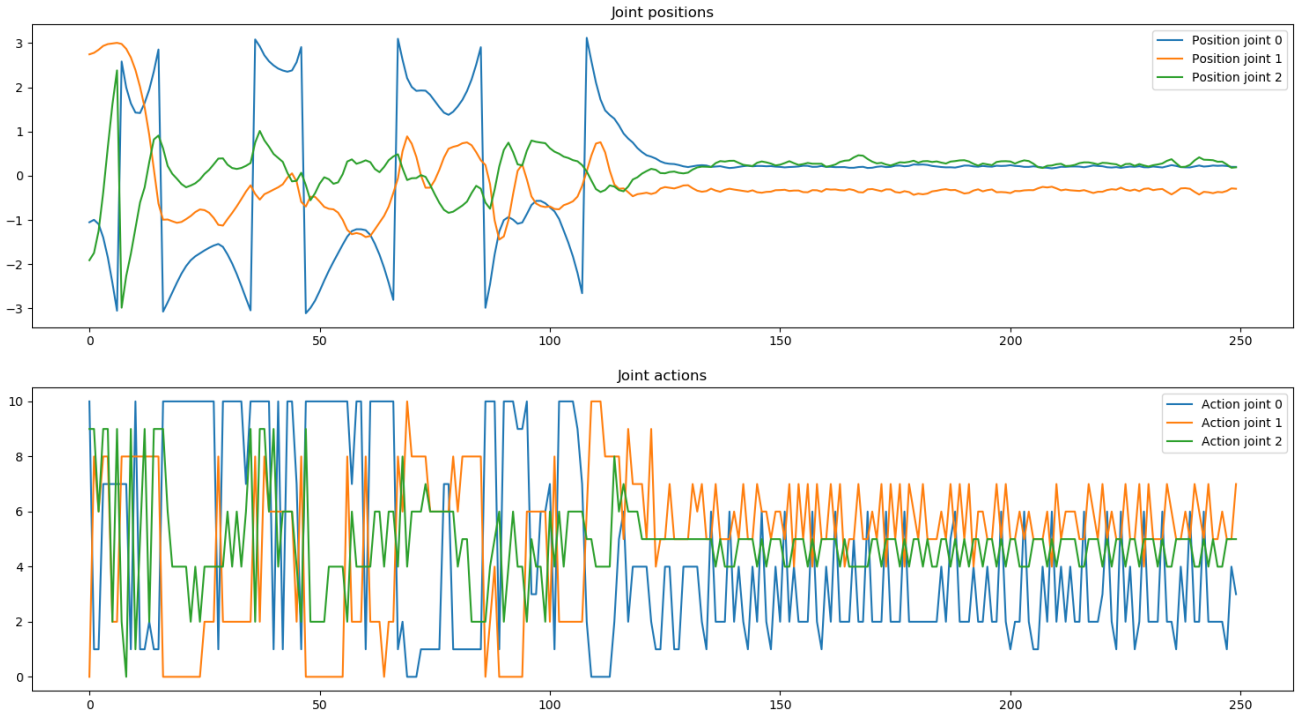

Figure 12. Joints: 3, episodes: 10000, torque: 10. Joint angles (above) and joint torques (below) are illustrated.