EXERCISE 13
Q-learning and deep-Q learning

Tue Herlau
tuhe@dtu.dk

2 May, 2025

Objective: The main goal of todays exercise is to implement and play with deep Q learning. Deep-Q learning can be though of as a combination of deep function approximators with (basic) model-based learning and an idea for reducing bias in Q-values. We will begin by exploring these two ideas in isolation in the tabular format to see they solve ideas present in any form of Q-learning (hence, it would also be a good idea to use them for the function approximation method) and only then implement DQN. Note that todays exercise can be completed using either Keras or Pytorch. (30 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex13.html

Contents

1 Theoretical problem: Baselines

The idea in Dueling Q-networks is that we write the Q-function as:

$$Q(s,a) = h(s) + g(s,a) - \max_a g(s,a)$$

Where h and g are two functions that the dueling Q-learning method then learn[1].

Assume that we have successfully found the optimal Q-functions, i.e., the Q-function satisfy the Bellman optimality equation:

$$Q(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q(s',a')|s,a]$$

Show that h must be equal to the optimal value function, i.e., that it satisfies:

$$V(s) = \max_a \mathbb{E}[r + \gamma V(s')|s,a]$$

2 Dyna-Q (`dyna_q.py`)

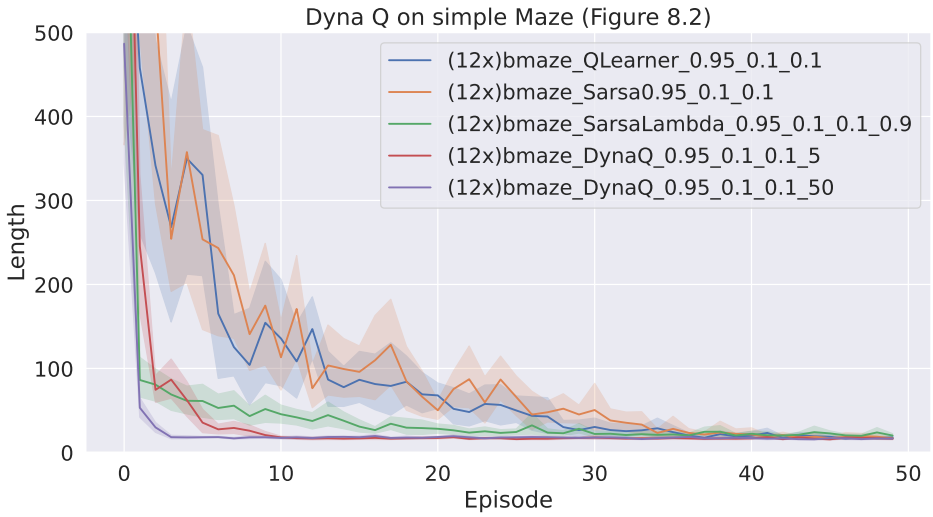
The Dyna-Q method is the standard Q-learning algorithm augmented with off-policy learning from samples drawn from a model. In practice, models of the environment are not easy to come by, so our model of the environment will simply consist of old observed transitions we append to a list (the *replay buffer*). Note in this formulation we don't use that the model is deterministic, and the method can therefore be used on any environment with ease.

Problem 1 Dyna-Q agent

Complete the implementation of the Dyna-Q agent and evaluate it on the simple MazeGrid environment as in [SB18, Example 8.2].

[1] it is not important how these functions are learned right now but if you are curious, you can implement it during the last of Todays exercise

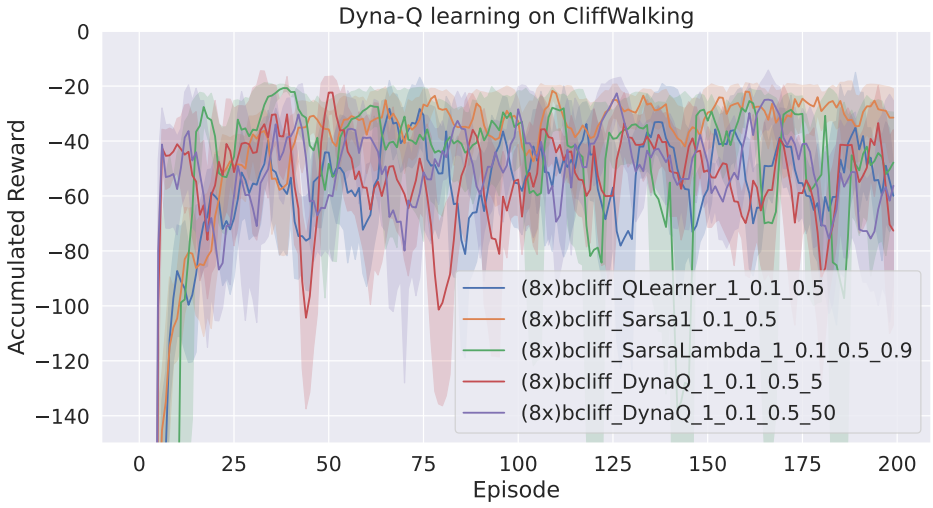
Info: Your results should be comparable to those in [SB18], however we don't have to initialize with a particular seed to show superior performance.



Problem 2 Dyna-Q agent on Cliffwalk

The use of a replay buffer (or model in the current terminology) also works for non-deterministic environments and offer a large benefit. To test this, complete the code in the second part of the exercise and test Dyna-Q on the windy gridworld environment

Info: I obtain the following results, which we can compare to our previous results where we compared against Sarsa. What are your conclusions of the relative performance of Q-learning, Sarsa and Dyna-Q? Why does Sarsa still seem to obtain higher asymptotic performance (and how would you fix it)?



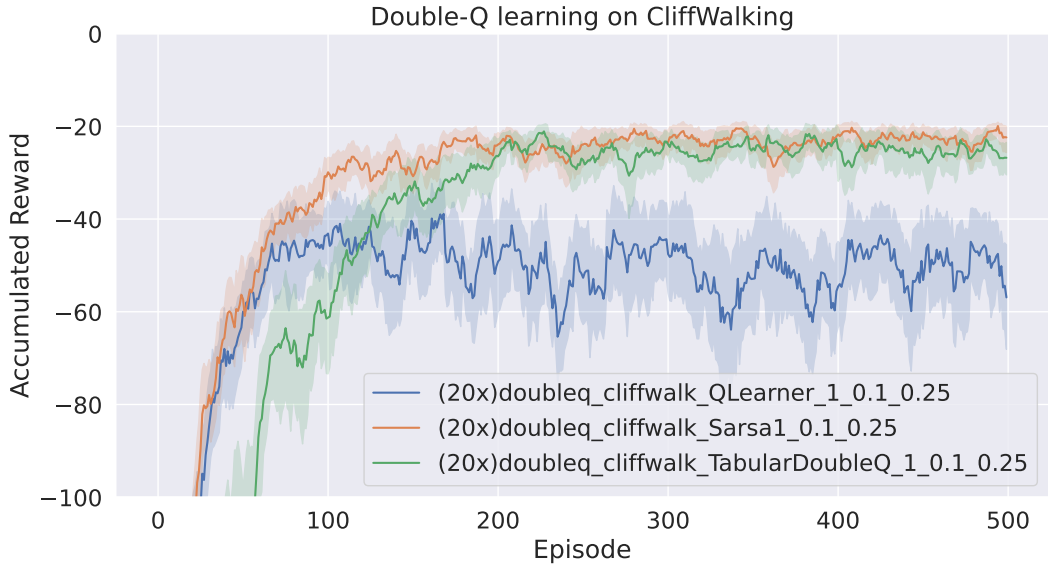
3 Tabular double-Q learning (`tabular_double_q.py`)

Our next warm-up method will be the tabular double-Q learning method from [SB18, Section 6.7]. The datastructure for the Q-values will be exactly as before and have already been initialized in the code, but you have to complete the update equations, however take care which of the Q-values you use (Q_1 or Q_2) in the update rules. We will first test the method on the Cliffwalking environment. It should be noted that the Q-values found by double-Q will converge to the same value as regular Q-learning, but the convergence properties are different and the resulting method will be more robust which is particularly important in conjunction with function approximators.

Problem 3 Tabular double-Q agent on Cliffwalk

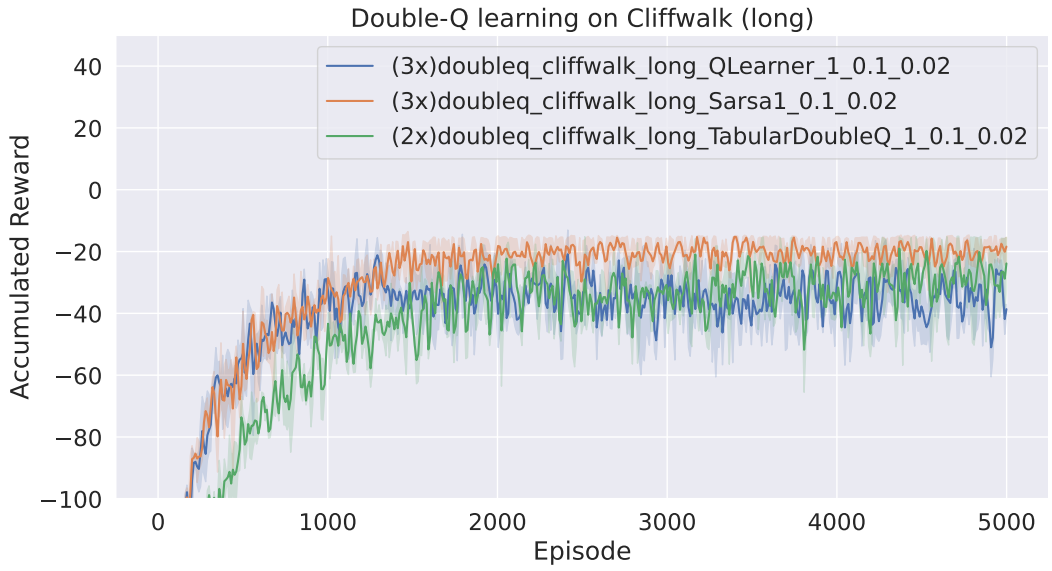
Complete the code for the double-Q agent and test it on the cliffwalk example.

Info: I obtain the following results using a value of $\alpha = 0.25$. Note convergence of double-Q learning takes roughly twice as long as Sarsa and Q-learning. Can you explain why?



The double Q-learning algorithm seems to converge to the same performance of Sarsa, which is quite strange in the context of the problem (recall the reason Q-learning performs worse than Sarsa is because Sarsa will adapt to the $\varepsilon = 0.1$-greedy exploration and steer clear of the cliff). I actually find it difficult to explain why double-Q perform so relatively well on this problem, but perhaps you can come up with an explanation?

The effect is entirely due to the (relative) high exploration rate α, as can be verified by a long simulation using a lower ($\alpha = 0.02$ rate of exploration as shown below:



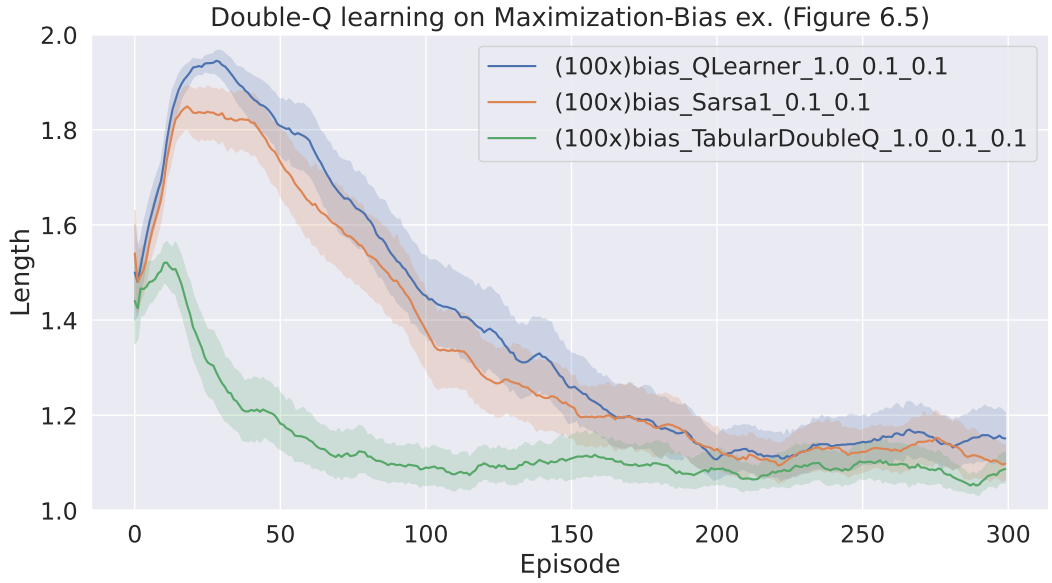
3.1 Maximization-bias example (`maximization_bias_environment.py`)★

We will next reproduce the results of the Maximization Bias example from [SB18, Example 6.7]. We will use a finite version of the environment because it will be quick to implement using our existing tools we encountered in an earlier exercise, however the discretization should have no effect. Complete the code for building the environment (i.e. what happens in state A) and run the simulations `tabular_double_q.py`

Problem 4 Maximization-Bias example

Implement the remaining part of the maximization-bias example (recall the format of the MDP is a dictionary of the form `{..., (state, reward): probability, ... }`) and re-produce the example figure.

Info: Since we don't have an easy way to get the number of left-actions in a given state, I simply plot the trajectory length which measures the same thing. You should get a result comparable to the following, which indicates gross over-estimation (and therefore a poor policy) during the initial training.



4 Deep Q learning★

We will now combine the previous two ideas, a replay buffer and double-Q learning, with a neural function approximator to obtain variants of the deep Q learning (DQN). There are three annoying things to be aware of when working with DQN,

- All sensible implementations have to rely on a DQN framework such as Keras or pytorch

- The methods will have many tunable parameters and the simulations are likely to take a long time to converge
- There are no convergence guarantees and results are often highly dependable on seeds

4.1 Implementation details

To somehow limit these annoyances we will work on a very small environment (Cart-pole) such that simulations can complete in a few minutes and secondly, abstract all the learning methods to a separate class. However note that these issues aside, we will end up with a real DQN implementation, and it should converge to a good controller on an Atari game given a few days of training.

4.1.1 The replay buffer

The replay buffer plays the role of the memory list in Dyna-Q. An example of how to use it is given below:

```
        # deepq_agent.py
        self.memory = BasicBuffer(replay_buffer_size) if buffer is None else buffer
        self.memory.push(s, a, r, sp, done) # save current observation
        """ First we sample from replay buffer. Returns numpy Arrays of dimension
        > [self.batch_size] x [...]]
        for instance 'a' will be of dimension [self.batch_size x 1].
        """
        s,a,r,sp,done = self.memory.sample(self.batch_size)
```

The main difference is it has a maximum size (first-in first-out, to avoid using all the memory on the computer) and an explicit sample function. The sample function takes an integer argument (`batch_size`) and return that number of samples stacked together. In other words, if s is n-dimensional, then `s` in the above will be a numpy `Array` of size

$$(\texttt{batch\_size} \times n)$$

(and similar `a`, `r`, and so on). In the following, assume all occurrences of s, r, etc. will have a batch-dimension.

4.1.2 The deep network

To make things easier I have abstracted everything relating to pytorch into a separate class so as to make it more clear which parts of DQN has to do with the actual method, and which has to do with the pytorch. This means you have to implement a single class with three relevant methods, see `dqn_network.py`. The first two evaluate the Q-values and fit the network:

```python
# irlc/ex13/lecture_12_examples.py
    # Initialize a network class
    self.Q = Network(env, trainable=True) # initialize the network
    """ Assuming s has dimension [batch_dim x d] this returns a float numpy Array
    array of Q-values of [batch_dim x actions], such that qvals[i,a] = Q(s_i,a) """
    qvals = self.Q(s)
    actions = env.action_space.n # number of actions
    """ Assume we initialize target to be of dimension [batch_dim x actions]
    > target = [batch_dim x actions]
    The following function will fit the weights in self.Q by minimizing
    > ||self.Q(s)-target||^2
    (averaged over Batch dimension) using one step of gradient descent
    """
    self.Q.fit(s, target)
```

The fit method might seem a bit odd as it has dimension of actions, but if `target` is initialized by having most entries equal to $Q(s,a)$, and only differ for the relevant actions where we want to adapt $Q_{\boldsymbol{w}}(s_i, a_i)$ towards y_i, it can implement the Q-learning objective function.

Finally, to implement double-Q learning, we have to adapt weights in one network from another. This can be done using:

```python
# irlc/ex13/lecture_12_examples.py
    self.Q2 = Network(env, trainable=True)
    """ Update weights in self.Q2 (target, phi') towards those in Q (source, phi)
    with a factor of tau. tau=0 is no change, tau=1 means overwriting weights
    (useful for initialization) """
    self.Q2.update_Phi(Q2, tau=0.1)
```

4.1.3 Scheduling of learning and exploration rate

It is common in Q learning to schedule learning and exploration rate, and in particular starting out with a high exploration rate early is of importance. The exploration rate ε will therefore be a function in our implementation of the number of episodes (specifically, it will decrease linearly, however $\frac{1}{n}$ is another popular choice). The methods appears less sensitive to scheduling α, however it is important to select a suitably low value to avoid divergence.

4.1.4 Other details

For any real use a deep Q learning method should allow us to manage experiments (which the `train` method already does), but also store simulations and later resume them. For completeness, and in case anyone want to try to the Atari example, I have implemented this functionality using `save` and `load` methods which should be used for long simulations. For our purposes these can be ignored.

4.2 Classical deep Q learning with replay buffer (`deepq_agent.py`)

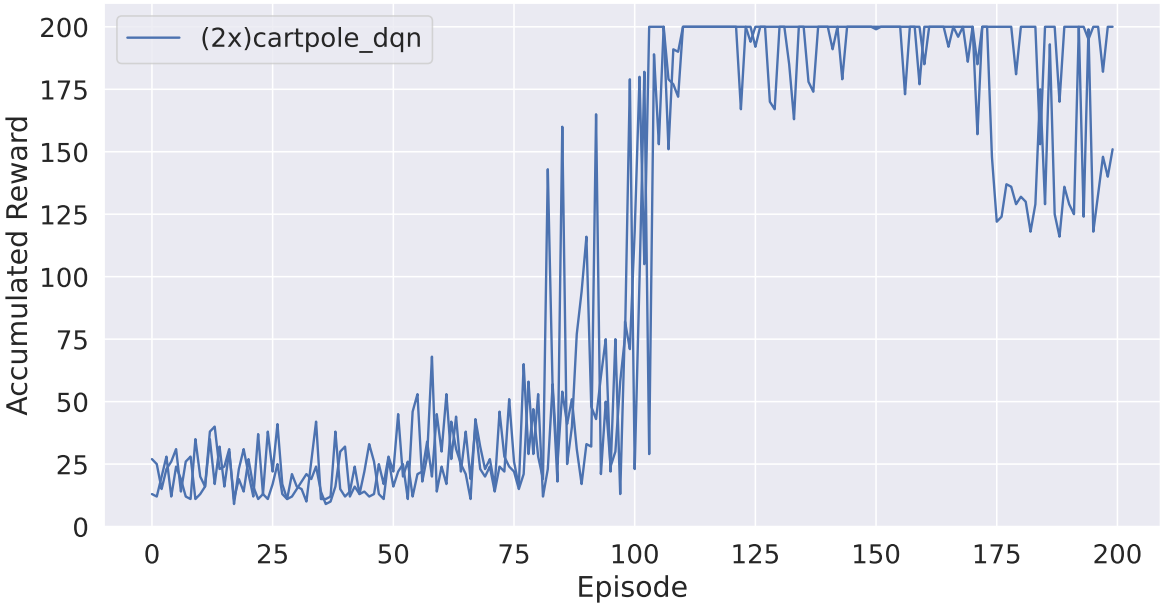
The first method we will implement will be the basic deep-Q learning method with a replay buffer. The implementation is actually very short (only about 30-40 lines) and I recommend reading through it for completeness. One thing to be aware of is that when s is a terminal state, i.e. $S_T = s$, then $Q(s, a)$ should be zero. In the tabular implementation this was guaranteed, but here you should multiply by `1-done` to ensure this is the case.

Note the simulation script save intermediate results thus allowing us to continue simulations later; this is not important for our short example, however it would be a good idea for more time-consuming simulations.

Problem 5 Basic deep Q learning

Complete the implementation of the Q-learning method and evaluate it on the Cartpole environment.

Info: I get the following results:



As we see the Agent solves the environment, but it is not very stable and individual runs has a fair amount of variability. I have not been able to find the source of this variability, but I suspect it might be the network architecture (fewer layers), layer initialization or something along those lines. It might also be something specific with my computer, since the parameters are taken from online implementations which seem to perform slightly less well than reported when I run them.

4.3 Double-Q learning (`double_deepq_agent.py`) ★

For double-Q learning we need to introduce a target network (This network plays the role of $\hat{q}_{\phi'}(s, a)$ in the slides) and it will be denoted `target` in the code (see included

comments). Asides this most of the code will be similar to the DQN agent which we can inherit from.

To adapt the weights, use the function in the DQN Network class (see example above).

Problem 6 Double deep Q learning

Complete the implementation of the double deep-Q learning method and evaluate it on the Cartpole environment.

Info: I get the following results:

Once more we see the agent solves the environment but the stability could be improved. The tendency over many simulations is to be slightly better than regular DQN.

4.4 Dueling-Q networks (`duel_deepq_agent.py`★★)

Dueling deep-Q is a different way to parameterize the Q-network by separating it into a value function and the so-called advantage. The Q-function is then defined as [WSH+15]

This method generally converge slightly faster than regular Q-learning.

To implement the method, all we need to do is to adapt the parameterization of the Q-network. You can find hints here:

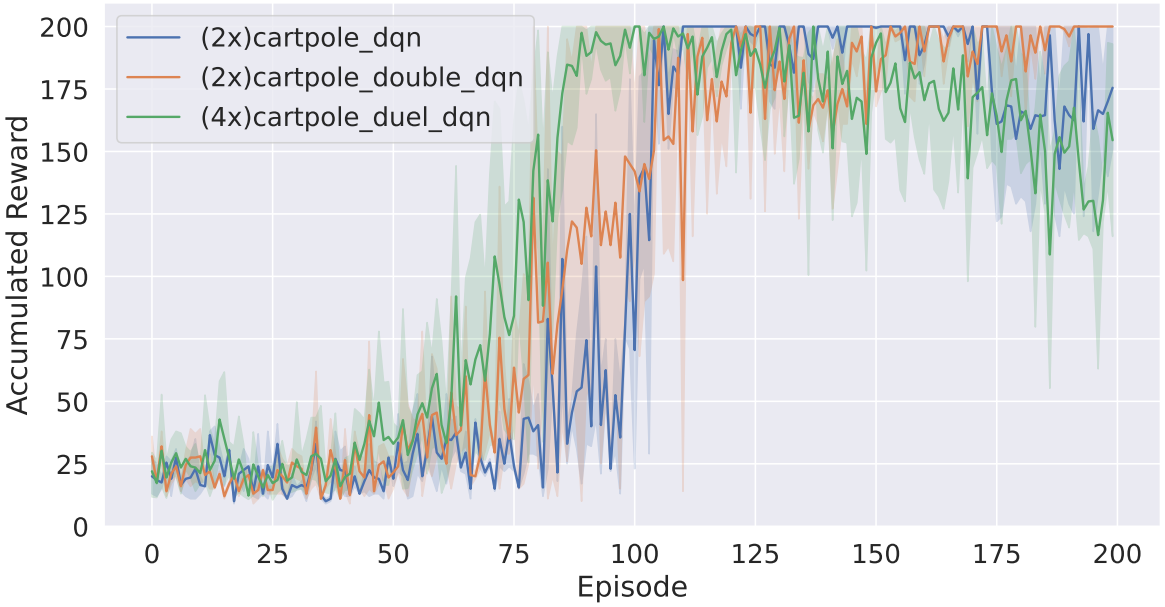
Pytorch

- `https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling`

Problem 7 Double deep Q learning

Adopt ideas from existing solutions to change the network class and implement Dueling deep-Q learning

Info: I get the following results:



These results are on average less stable than DQN and double-DQN (but perhaps with a slightly quicker learning). My suspicion is that the networks are too deep for this simple task.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).

[WSH+15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.