Learning Options in Deep Reinforcement Learning

Jean Merheb-Harb

Computer Science
McGill University, Montreal

December 8, 2016

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Master of Science.

Acknowledgements

First and foremost, I would like to thank my supervisor, Doina Precup, for all the encouragement and for teaching me so much. A special thanks to Pierre-Luc Bacon, for the great discussions we've had and enduring my constant bombardment of questions.

Abstract

Temporal abstraction is a key idea in decision making that is seen as crucial for creating artificial intelligence. Humans have the ability to reason over long periods of time and make decisions on multiple time scales. Options are a reinforcement learning framework that allow an agent to make temporally extended decisions. In this thesis, we present the deep option-critic, which combines the powerful representation learning capabilities of deep learning models with the option-critic to learn both state and temporal abstraction automatically from data in an end-to-end fashion. We apply the algorithm on the Arcade Learning Environment, where the agent must learn to play Atari 2600 video games, and analyze performance and behaviours learned using the options framework.

Résumé

L'abstraction temporelle est une idée clée dans la prise de décision qui est considérée comme cruciale pour la création de l'intelligence artificielle. Les humains ont la capacité de raisonner sur de longues périodes de temps et prendre des décisions sur différentes échelles de temps. Les options sont un cadre conceptuel d'apprentissage par renforcement qui permettent à un agent de prendre des décisions temporellement étendues. Dans cette thèse, nous présentons l'option-critique profond, qui combine les puissantes capacités d'apprentissage de la représentation des modèles d'apprentissage profonds avec l'option-critique, donnant la capacité d'apprendre l'abstraction temporelle et d'état automatiquement à partir des données d'une manière de bout en bout. Nous appliquons l'algorithme sur le "Arcade Learning Environment", où l'agent doit apprendre à jouer aux jeux vidéo du Atari 2600, et nous analysons les performances et les comportements appris en utilisant le cadre conceptuel d'options.

Contents

1

Introduction

Creating models capable of hierarchical decision making has been a challenging task. Humans have the ability to reason over long periods of time and make decisions on multiple time scales. We can choose to go to the store, and within that decision, choose to get up from our seat to leave. Even within that decision, we can break it down into the smallest actions such as contracting certain muscles. However, all reinforcement learning (RL) algorithms seen to date make decisions for a single time-step, evaluating the entire future ahead every time. Temporally extended decisions are seen as a crucial component for scaling up, both to improve the way we act and to learn more efficiently. Most RL algorithms are currently only able to reason at the smallest time scale possible, similar to making decisions about which muscle contraction will lead to successfully going to the store. This is an inefficient way to reason, as one would have to think about the effect of every muscle contraction on the world.

Reinforcement learning algorithms are usually applied at a single time scale. Algorithms like Q-learning and SARSA learn to give values for each action in every state, and then usually choose the action with the highest value. Multiple issues arise from such a framework. The actions are applied for a single time-step, meaning the agent must make a new decision as soon as possible, akin to choosing your entire life's path at every fraction of a second. Also, RL's main task is credit assignment,

propagating rewards received to past actions that led to such a situation. It's difficult to assign credit to an action if it consists of something as simple as contracting a muscle, whereas hierarchical models would allow one to assign credit many time-steps back. The ability to make temporally extended decisions is a fundamental building block for good decision making.

Temporal abstraction has been applied to reinforcement learning in different ways, but automatically learning the high-level decisions remains very challenging. R. S. Sutton, Precup, and S. Singh 1999 introduced the options framework, where an agent is capable of using temporally extended decisions to act and plan in an environment. Some of the methods for learning options required state clustering or partitioning, and having the options learn to go to the transition points between clusters (Hauskrecht et al. 1998). Another method tried to first learn sub-goals, then have the options learn to get to them, and finally chain the sub-goals to get to the main goal (Konidaris and Barreto 2009, McGovern and Barto 2001, Stolle and Precup 2002). However, all of these methods either required some hand-engineered methods using human knowledge, some type of segmentation in the learning process, or even a list of all states hindering the algorithm's scalability. Bacon, Harb, and Precup 2016 has introduced the first algorithm capable of learning options on-line, within a single learning process, simultaneously while learning values and state abstraction.

Reinforcement learning algorithms can use models to approximate its estimated values. The popular field of deep learning consists of powerful models capable of learning non-linear functions. Neural networks, composed of stacked layers of linear models, allows one to build abstractions as hierarchies of features. Throughout the years, more complex but intuitive models were developed, allowing people to better extract information from image and temporal data. Convolutional neural networks (CNN) are the most powerful models capable of extracting information from images. They start by learning to recognize simple shapes and combine these to recognize more complex figures. After a few layers of abstraction, these models are capable of

learning to recognize anything.

The combination of reinforcement learning and deep learning was popularized when DQN (Mnih, Kavukcuoglu, et al. 2015) was developed, a reinforcement learning agent capable of learning to play a variety of games directly from pixels with no previous or hard-coded knowledge. Using a CNN to recognize shapes and objects from pixels, the agent learns to assign value to what it sees from experience with the environment.

This thesis combines the option-critic architecture with DQN, allowing an agent to learn state and temporal abstraction simultaneously.

1.1 Outline

Chapter 2 starts with an introduction of Markov decision processes and reinforcement learning, explaining how an agent can learn about an environment and learn to improve itself to maximize its expected rewards. In Chapter 3, we go through some deep learning methods, which are powerful models used for recognition and classification. At the end of the chapter, we merge deep learning and reinforcement learning, allowing RL agents to learn to recognize objects using DL tools and use what it sees in the environment to help make its decisions. Chapter 4 introduces temporal abstraction in reinforcement learning, that is, for an agent to make decisions on a longer timescale. We do this through the options framework and the option-critic architecture. Finally, in chapter 5, we put everything together and learn a model with the option-critic algorithm while using deep learning models. We then present and analyze results of the algorithm applied to some games in the Atari domain.

2

Sequential Decision Making under Uncertainty

2.1 Markov Decision Process

A **Markov Decision Process** (MDP) is a framework for decision making with stochasticity. At each time-step, an agent is in a state, chooses an action, which will send him to a new state.

More specifically, an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, r, \mathcal{P}, \gamma \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a function that maps a state-action pair to a real-valued reward, $\mathcal{P}(s'|s, a)$ is the transition probability of going from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ when taking the action $a \in \mathcal{A}$, and γ is the discount factor where $\gamma \in [0, 1]$.

If γ is 1, the MDP is said to be in the undiscounted setting, meaning all rewards received, regardless of how far in the future, have the same importance. When γ is smaller than 1, the agent values early rewards more than later ones.

A state is called terminal if the MDP ends when the agent gets to that state. If such a state is present, the MDP is said to be episodic.

The key property of MDPs is that they are Markovian. This means that transitions are only conditionally dependent on the current state and action, and completely independent of past information. That is, $P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, ..., s_0, a_0)$

2.1.1 Policies

A policy π represents the behaviour of an agent, a mapping from state to action.

In the deterministic policy setting, a state will always result in the same action choice for a given policy, $\pi : \mathcal{S} \mapsto \mathcal{A}$. In the stochastic setting, the action is sampled from a state dependent distribution, $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$, where each action has a probability of getting selected.

2.1.2 Value Function

The value of a state is the expected discounted sum of rewards by following a certain policy, while starting at the state s, calculated as follows.

$$V^{\pi}(s) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big]$$

The state-action value function $Q^{\pi}(s, a) : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is similar to the value function, but assumes that an action a is taken at the first step. This function is often used when deciding which action to take by comparing the values of each action.

2.1.3 Bellman Equation

Instead of looking at the state values as the expected return of the entire future, the Bellman equation (Bellman 1956) allows one to view it in a recursive manner, by using the values of the next state. The equation is given by the following.

$$V^{\pi}(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V^{\pi}(s')$$

As we can see, the state value is equivalent to the expected reward in the next time step plus the decayed weighted sum of the next state value. We can also express the equation in a vectorized form.

$$V^{\pi} = R^{\pi} + \gamma P^{\pi} V^{\pi}$$

If we have all information on the MDP, we can solve for the values of all states, in a closed form solution.

$$V^{\pi} = (I - \gamma P^{\pi})^{-1} R^{\pi}$$

2.1.3.1 Bellman Operator

The Bellman operator transforms an arbitrary value function so that consecutive states obey the Bellman equation.

$$T^{\pi} V(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V(s')$$

A single step of this operator brings the state values closer to the equivalence, of a given policy. Iterating this process converges the state values to the correct values of the policy. It is an alternative to the closed form solution seen previously.

2.1.4 Dynamic Programming Algorithms

The Bellman equation and operator are great tools for solving values, but if we want an agent to optimize rewards, it must improve its policy along the way. We can use dynamic programming algorithms to solve for optimal control strategies while learning the state values. Two of the most used methods that use the Bellman operator are value iteration and policy iteration.

2.1.4.1 Value Iteration

Value iteration (VI) (Howard 1960) is an algorithm that can solve the optimal state-action values and optimal policy of an MDP, by using the Bellman operator. The control is done by using the greedy policy, where the agent chooses the action with the highest Q-value. The algorithm consists of two steps.

First, starting with arbitrary state values, we update the Q-values of every state action by doing one step of the Bellman operator.

$$Q_{t+1}(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_t(s)$$

Second, we update the state values by taking the highest Q-value for each state. Since we're trying to get the optimal policy, the agent will take the action with the highest value.

$$V_{t+1}(s) = \max_a Q_{t+1}(s, a)$$

Finally, we repeat these two steps until convergence. As we can see, VI consists of iterating between one step of the Bellman operator and one step of policy improvement.

2.1.4.2 Policy Iteration

Another popular algorithm that solves control in MDPs is Policy Iteration (PI) (Howard 1960). The major difference with VI is that we solve for the state values of a fixed policy, and only then do we improve the policy. Also, PI directly updates a policy, remembering exactly which action to select in each state, whereas VI indirectly finds it from the learned Q-values.

We start by randomly initializing a policy π. That is, each state has a deterministic action choice. Then we iterate between the two following steps.

First, we update the values of each state for the current policy. Either by applying the Bellman operator until the values converge, or by closed form solution from the set of linear equations.

$$V_{t+1}^{\pi_t}(s) = r(s, \pi_t(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi_t(s)) V_{t+1}^{\pi_t}(s')$$

The second step is to update the policy by choosing the action with the highest value. But since we aren't calculating the state-action values, we simply calculate them by using the one-step value function.

$$\pi_{t+1}(s) = \text{argmax}_a \Big(r(s, a) + \gamma \sum_{s'} P(s'|s, a) V_{t+1}^{\pi_t}(s') \Big)$$

2.2 Reinforcement Learning

Reinforcement learning is a framework in which an agent tries to solve an MDP, when no information about transitions or rewards is known. The challenge is to simultaneously explore the environment to learn about it, and exploit it to receive the rewards. The agent must go through the environment and sample transitions, to learn the state values and then take more informed decisions later on.

2.2.1 Monte Carlo

One way to estimate the value of a state is simply to sample entire trajectories and average their sum of discounted rewards for each state. As we get more samples, our value estimates become more accurate.

There are a few problems with this method however. Sampling entire trajectories is expensive, the environment needs to be episodic, and the variance is very high.

2.2.2 Temporal Difference Learning

An alternative to Monte Carlo methods is Temporal Difference (TD) (R. S. Sutton 1988). Instead of sampling an entire trajectory, we can use the Bellman equation to find the difference in estimation between two consecutive states.

The Bellman equation states that $V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V^\pi(s')$, where the value of the next state is an expectation over all states. Since we don't know the transition probabilities, the agent can sample a state from that expectation. Giving us

$$V^\pi(s) = r(s, \pi(s)) + \gamma V^\pi(s')$$

where s' is our sampled state from acting in the environment. Since our values are being learned, both sides of the equation won't be equal. We call this error the temporal difference. $\text{TD} = R(s, \pi(s)) + \gamma V^\pi(s') - V^\pi(s)$

To calculate the value of each state for a given policy, we can perform the following algorithm. The agent acts through the environment and updates its state values by using the TD, similar to how we used the Bellman operator in the previous algorithms. The difference is that we are using samples from the environment and have no information on the rewards and transitions.

Initialize $Q(s, a)$
repeat
  $s \leftarrow s_0$
  repeat
    $a \leftarrow \pi(s)$
    Apply action a on environment and observe s', r
    $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$
    $s \leftarrow s'$
  until s' *is terminal*
until *forever*

Algorithm 1: One-step tabular TD

2.2.3 On-Policy and Off-Policy Learning

The TD algorithm learns the values of all states given a policy π. However, in most settings, we will not only want to learn the values of states, but also improve our policy to maximize rewards. There are two types of ways we can learn a policy.

First is **on-policy** learning, where we improve an agent's policy which is being used directly on the environment. Its job is to learn a policy that will need to explore well enough to learn, but also act greedily to get rewards.

The alternative is **off-policy** learning, where one agent acts using a policy that explores well, and a second agent uses the experiences as data to improve its own policy. Since the samples from the policy acting on the environment come from a different distribution than if the second policy were to act, we usually need to use sampling methods, such as importance sampling, to eliminate bias from the action and state samples.

2.2.4 Online and Offline Learning

Online learning is when an agent updates its estimates at every time-step using only the last transition and reward samples. If it uses data from the past or updates only at certain time-steps, such as at the end of an episode, it is called offline learning.

2.2.5 SARSA

SARSA (Rummery and Niranjan 1994) is an on-policy algorithm that learns the value of policy while acting in an environment. It uses TD to learn the state-action values as it acts. To calculate the TD in SARSA, one needs a state and action, the resulting reward, the following state, and the following action, hence the name.

The policy is derived from the state-action value function Q. When choosing a policy from the Q-values, the optimal action is to choose the action with the highest value. However, as we need to learn about the environment to potentially improve our policy, we must act randomly sometimes. A popular policy used in this setting is ϵ-greedy. This means to choose a random action $\epsilon\%$ of the time and take the best action otherwise. Finally, since this algorithm is on-policy, the values will reflect the fact that some randomness is present in the policy. So in states that present risk and ask careful action selection, the randomness will diminish state value. This results in more conservative policies.

2.2.6 Q-Learning

SARSA's off-policy counterpart is Q-Learning (Watkins and Dayan 1992). In this setting, the agent uses one policy to move through an environment and uses those experiences to learn a second policy that will act differently.

The second value function we're learning assumes its own policy will be followed. This means that the risk presented by exploration can be disregarded, since the optimal policy will not be selecting random actions in risky states. As opposed to SARSA,

Initialize $Q(s, a)$
repeat
  $s \leftarrow s_0$
  $a \leftarrow$ Policy from Q (e.g. ϵ-greedy)
  repeat
    Apply action a on environment and observe s', r
    $a' \leftarrow$ Policy from Q (e.g. ϵ-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until s' *is terminal*
until *forever*

Algorithm 2: One-step tabular SARSA

states that demand careful attention will now have high value as we can expect the policy to act correctly.

Initialize $Q(s, a)$
repeat
  $s \leftarrow s_0$
  repeat
    $a \leftarrow$ Policy from Q
    Apply action a on environment and observe s', r
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
  until s' *is terminal*
until *forever*

Algorithm 3: One-step tabular Q-Learning

2.2.7 Double Q-Learning

Q-learning is known to empirically suffer from over-estimations of Q-values. The reason for this is that the target $r(s, a) + \gamma \max_a Q(s', a)$ has a max operator over the same Q-function that we are learning. When starting the learning process, there initial Q-values will have a distribution of errors, but the max operator will take the action with the highest error available, since it sees it as the best action. This will then update other states' value with the high error, propagating it to many states as

we perform updates.

To fix this, Van Hasselt 2010 created the Double Q learning algorithm, which has two Q-functions. We learn both Q-functions on the same environment, and use each to learn the other. Instead of the regular gradient update

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

we interchangeably have

$$Q_A(s,a) \leftarrow Q_A(s,a) + \alpha[r(s,a) + \gamma Q_A(s', \operatorname{argmax}_{a'} Q_B(s',a')) - Q_A(s,a)]$$

and

$$Q_B(s,a) \leftarrow Q_B(s,a) + \alpha[r(s,a) + \gamma Q_B(s', \operatorname{argmax}_{a'} Q_A(s',a')) - Q_B(s,a)]$$

At each update step, one of the two functions is sampled randomly to update.

From the update functions, we can see that instead of taking the highest value of the next state, which leads to over-estimation, we take the value of the action with the highest value in the second network. On the early stages of learning, when values aren't accurate, the best action from the second network probably won't be the same as the best for the current network, stopping the over-estimation. However, later on in the training, when both networks have learned for a while, the best action from the second network will most likely be the same as the first one, leading to the correct update.

2.2.8 Function Approximation

All algorithms we've seen to this point were in the tabular setting. A tabular environment is one where each state is simply an index or a one-hot encoding, with no notion of similarity or information. There are major issues with this approach. When learning about one state, the agent cannot share the new knowledge to similar states, slowing training capacity. Furthermore, the state spaces can quickly become very

large and intractable to compute. For example, in pixel space, every combination of pixel colors would be seen as a different state.

The alternative is called function approximation. We can use state features as information, and learn a function to approximate state values and learn policies. The functions can learn to give values to certain features and use this to correctly predict the value of a state never seen before, by knowing its features. The algorithms seen remain very similar, but we use models to learn the values instead of directly remembering a value for every state. The most heavily used methods are gradient based, where we calculate the gradient of the weights in our function with regards to the TD, to minimize it.

2.2.9 Policy Gradient Methods

We've seen methods such as Q-learning and SARSA, which learn state-action values and use a greedy policy over these to select actions. However, it's also possible to learn policies themselves.

2.2.9.1 REINFORCE

The first method, REINFORCE (Williams 1992), learns a policy solely from experience and does not require a value function at all. It is analogous to the Monte Carlo method, that required an entire sample trajectory to update the value function of a state, however, in REINFORCE, we use the trajectory to update a policy.

The objective is to maximize the expected cumulative reward.

$$\rho(s) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t \middle| s\right]$$

We can call the expected cumulative reward of a specific trajectory τ, $R(\tau)$. The probability of each trajectory is given by the policy, since it dictates what actions the agent will take. So we can reformulate the objective as the following.

$$\rho(s) = \sum_{\tau} P_\theta(\tau) * R(\tau)$$

where θ represents the parameters of our policy, which we are trying to learn. To maximize the objective, we take calculate gradient of ρ with regards to θ.

$$\frac{\partial \rho(s)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{\tau} P_\theta(\tau) * R(\tau)$$

$$= \sum_{\tau} \frac{\partial P_\theta(\tau)}{\partial \theta} * R(\tau) + P_\theta(\tau) * \frac{\partial R(\tau)}{\partial \theta}$$

Since $R(\tau)$ is independent from θ, its gradient is 0, giving us

$$= \sum_{\tau} R(\tau) * \frac{\partial P_\theta(\tau)}{\partial \theta}$$

We then use the log trick to change the sum of gradients of all states to an expectation over them, allowing us to sample trajectories.

$$= \sum_{\tau} \frac{P_\theta(\tau)}{P_\theta(\tau)} * R(\tau) * \frac{\partial P_\theta(\tau)}{\partial \theta}$$

$$= \sum_{\tau} P_\theta(\tau) * R(\tau) * \frac{\partial P_\theta(\tau)}{P_\theta(\tau) \partial \theta}$$

$$= \sum_{\tau} P_\theta(\tau) * R(\tau) * \frac{\partial \log(P_\theta(\tau))}{\partial \theta}$$

$$= \mathbb{E}_\tau \left[R(\tau) * \frac{\partial \log(P_\theta(\tau))}{\partial \theta} \right]$$

Intuitively, this gradient shows that we can sample a trajectory using our policy, and increase the probability of doing that trajectory scaled by the return. As we see different trajectories, the ones with high returns will be increased more than the ones with low returns, resulting in a policy that chooses better trajectories.

2.2.9.2 Policy Gradient

Instead of using entire trajectories as samples of state values, we can also use a value estimate. These value estimates can be learned by TD, and the policy is then optimized using these estimates.

As stated in 2.2.9.1, the objective of a policy is to maximize.

$$\rho(s) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t \middle| s \right] = \sum_{a} \pi(a|s) Q^\pi(s,a) = V^\pi(s)$$

To maximize the formula, we calculate the gradient as follows (R. S. Sutton, McAllester, et al. 1999).

$$\frac{\partial V^\pi(s)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_a \pi(a|s) Q^\pi(s,a)$$

$$= \sum_a \left[\frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a) + \pi(a|s) \frac{\partial Q^\pi(s,a)}{\partial \theta} \right]$$

$$= \sum_a \left[\frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a) + \pi(a|s) \frac{\partial R(s,a) + \sum_{s'} \gamma P(s'|s,a) V^\pi(s')}{\partial \theta} \right]$$

As the reward and transition functions dependent only on the MDP and not the policy, their gradients are 0.

$$= \sum_a \left[\frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a) + \pi(a|s) \sum_{s'} \gamma P(s'|s,a) \frac{\partial V^\pi(s')}{\partial \theta} \right]$$

If we unroll the recursive value function a few times, we see that the left part of the equation ($\sum_a \frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a)$) comes back multiple times, for every time we encounter the state s in the future. Instead, we can sum the discounted probabilities of seeing each state at every step in the future. We can then change the gradient to the following.

$$\frac{\partial V^\pi(s)}{\partial \theta} = \sum_x \sum_{k=0}^{\infty} \gamma^k \Pr(s \to x, k, \pi) \sum_a \frac{\partial \pi(a|x)}{\partial \theta} Q^\pi(x,a)$$

where $\Pr(s \to x, k, \pi)$ is the probability of going from state s to state x in exactly k steps, while following policy π. We also notice that we go from $\frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a)$ to $\frac{\partial \pi(a|x)}{\partial \theta} Q^\pi(x,a)$ as the first term was only present because of the initial state. When unrolling the recursive part, we end up seeing the gradient with regards to all states. Finally, we define the term $d^\pi(s)$ as the discounted weighting of states encountered starting at state s_0, $d^\pi(s) = \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi)$. The final gradient is then given by:

$$\frac{\partial V^\pi(s)}{\partial \theta} = \sum_x d^\pi(x) \sum_a \frac{\partial \pi(a|x)}{\partial \theta} Q^\pi(x,a)$$

It's important to note that in practice most algorithms use $d^\pi(x)$ as a stationary distribution, which results in an expectation over states. Algorithms then use this to

sample states from experience. This is **not** the correct distribution, as $d^\pi(x)$ is not the stationary distribution but a weighting that depends on γ. Since it's a re-weighting of gradients, it presents a bias in the estimator. However, empirical evidence shows that the algorithms sampling from the stationary distribution have good and even performance than the unbiased methods (Thomas 2014).

To reduce variance, one can add a baseline. Any function that isn't dependent on actions can be used, shown as follows with $V(s)$ as an example.

$$\sum_s d^\pi(s) \sum_a \frac{\partial \pi(a|s)}{\partial \theta} [Q^\pi(s,a) - V(s)]$$

Splitting the subtraction and only taking the right side.

$$-\sum_s d^\pi(s) \sum_a \frac{\partial \pi(a|s)}{\partial \theta} V(s)$$

$$= -\sum_s d^\pi(s) V(s) \sum_a \frac{\partial \pi(a|s)}{\partial \theta}$$

$$= -\sum_s d^\pi(s) V(s) \frac{\partial}{\partial \theta} \sum_a \pi(a|s)$$

$$= -\sum_s d^\pi(s) V(s) \frac{\partial}{\partial \theta} 1 = 0$$

This shows that any term not dependent on a will have no effect on the expectation of the gradient.

Also, the sum over action gradients can be computationally expensive in large action spaces. We can use a log trick to turn the sum over action gradients into an expectation of the log-gradient.

$$= \sum_s d^\pi(s) \sum_a \frac{\pi(a|s)}{\pi(a|s)} \frac{\partial \pi(a|s)}{\partial \theta} Q^\pi(s,a)$$

$$= \sum_s d^\pi(s) \sum_a \pi(a|s) \frac{\partial \pi(a|s)}{\pi(a|s) \partial \theta} Q^\pi(s,a)$$

$$= \sum_s d^\pi(s) \sum_a \pi(a|s) \frac{\partial \log(\pi(a|s))}{\partial \theta} Q^\pi(a|s)$$

$$= \sum_s d^\pi(s) \mathbb{E}_a \frac{\partial \log(\pi(a|s))}{\partial \theta} Q^\pi(s,a)$$

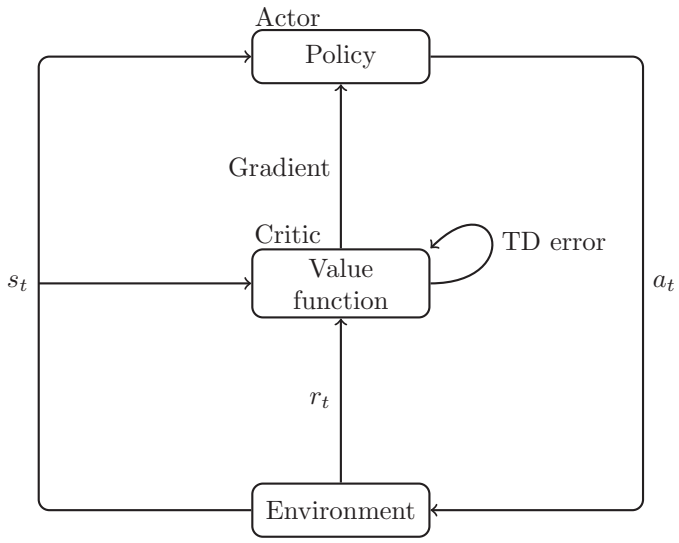


Figure 2.1: The actor critic architecture. An agent chooses an action based on the environment state, transitioning to a new state. During the transition, a reward signal is received, which is used to learn a value function, using the TD error. The value function is then used to improve the agent's policy.

Now with an expectation over actions, we can use action samples as we act as opposed to having to calculate the gradient of every action in a given state.

2.2.9.3 Actor-Critic

Actor-critic is an algorithm that learns both a policy (the actor) and a value function (the critic). It learns a policy like in REINFORCE, but uses a learned value function to calculate the gradient instead of using a sample trajectory, as explained with the policy gradient. We simultaneously learn the value function of the current policy using TD. Figure 2.1 shows the architecture and relation between each component of the actor-critic.

As explained in 2.2.9.2, the gradient for the actor is

$$\frac{\partial \rho}{\partial \vartheta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \vartheta} Q^\pi(s,a) = \sum_s d^\pi(s) \mathbb{E}_a \frac{\partial \log(\pi(a|s))}{\partial \vartheta} Q^\pi(s,a)$$

And the critic's gradient is

$$\frac{\partial Q(s,a)}{\partial \theta} = \frac{\partial (R(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)} V(s') - Q(s,a))^2}{\partial \theta} = -\sigma * \frac{\partial Q(s,a)}{\partial \theta}$$

The actor-critic algorithm can be trained either on-policy or off-policy. In the on-policy setting, we simply use samples from the latest action taken on the environment. In the off-policy setting (Degris, White, and R. Sutton 2012) however, importance sampling will be required to correct for the discrepancy in the probability of seeing the samples. This is necessary as the policy in the past will have led to different action choices and transitions than the current policy would lead to.

As explained previously, function approximation can be used to approximate the value of states while generalizing for previously unseen states. In the actor-critic case, using function approximation would require two functions, one for the critic and one for the actor. We then optimize each function, where both require each-other.

Initialize $Q_\theta(s,a)$, $\pi_\vartheta(a|s)$
repeat
  $s \leftarrow s_0$
  repeat
    $a \sim \pi_\vartheta(a|s)$
    Apply action a on environment and observe s', r
    $\theta \leftarrow \theta + \alpha\Big((r + \gamma V(s') - Q_\theta(s,a)) * \frac{\partial Q_\theta(s,a)}{\partial \theta}\Big)$
    $\vartheta \leftarrow \vartheta + \frac{\partial \log(\pi_\vartheta(a|s))}{\partial \vartheta} Q_\theta(s,a)$
    $s \leftarrow s'$
  until s' *is terminal*
until *forever*

Algorithm 4: One-step online Actor-Critic with function approximation

3

Deep Learning

Deep learning is a class of machine learning models that can learn non-linear functions. We use these types of models to learn knowledge representations that will then be used by the RL algorithms. In high dimensional environments, DL models are especially useful, allowing the RL agent to reason in a high level and generalizing in the recognition of objects and scenarios. We'll start by introducing the linear regression, which is not a deep learning model itself, but is their fundamental building block.

3.1 Linear Regressions

A linear regression is a model that learns the linear relationship between a set of explanatory variables (inputs) and a target variable (output). That is, learns a weight for every input variable to explain how to get the target, like follows.

$$y_i = x_i^T W + \epsilon_i$$

where W is the weight matrix which we're trying to learn, x is the data, with a number of features for each data point, i is the i^{th} data point and ϵ_i is noise, or variable that explains the gap between the prediction and the real target. The objective is usually to minimize the sum of squares of ϵ (OLS).

Multivariate linear regressions are linear models with multiple outputs for the same inputs, where you try to learn different types of targets with the same data. A

set of weights is learned for each of the different outputs.

3.2 Neural Networks

Neural networks are a class of models capable of learning non-linear functions. The most basic neural networks are stacked multivariate linear models with non-linear activation functions between layers. That is, each layer is a multivariate model with an arbitrary number of outputs, which instead of learning a target, simply creates new features as a composition of inputs. The following layer takes these outputs as its inputs, creating a combination of the combinations. This allows neural networks to extract complex structures from data by learning to create a hierarchy of features relevant to model data. Finally, since stacked linear models remains linear, we run the outputs of each layer through activation functions which perform non-linear transformations on the output, as explained in the next section.

Starting with X_0 as the initial input, the following formula shows how each output is a function of the following layer, with σ as the activation function.

$$\mathbf{X}_l = \sigma(\mathbf{W}_l \mathbf{X}_{l-1} + \mathbf{b}_l)$$

When creating a neural network, one chooses the number of layers (number of stacked multivariate linear models) and the number of outputs for each of them, except the final layer since it has a fixed output, as shown in Figure 3.1. These layers are sometimes called fully connected layers.

3.2.1 Activation Functions

An activation function is a non-linear function which is applied to each output neuron in a given layer. The three most heavily used activation functions are sigmoid, hyperbolic tangent (tanh) and rectified linear units (ReLU).

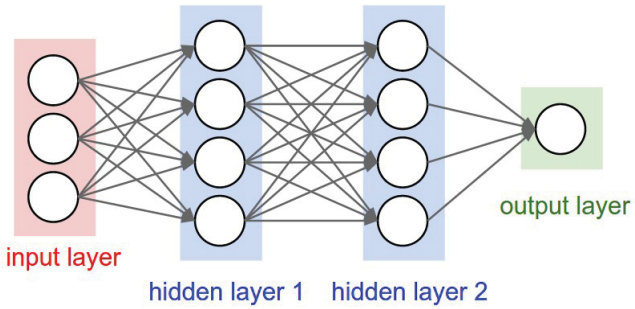


Figure 3.1: Example of a neural network architecture. source: http://cs231n.github.io/assets/nn1/neural_net2.jpeg

The sigmoid function is an S shaped function that is bounded by 0 and 1, smooth, differentiable, and mirrored along the y=0 axis. It is usually used when a probability is required as the output.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Tanh has the same properties as the sigmoid function, but is bounded by -1 and 1.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

One of the main reasons why sigmoid and tanh are used is due to the property that their derivative can be calculated as a function of the output.

$$\frac{\partial \text{sigmoid}(x)}{\partial x} = \text{sigmoid}(x) * (1 - \text{sigmoid}(x))$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

The ReLU function simply keeps the value of x if it is larger than 0, otherwise, it outputs 0.

$$\text{ReLU}(x) = \max(0, x)$$

One of the reasons why this function is used is that its gradient remains the same regardless of the scale of x, whenever $x > 0$. Alternatively, the gradient from sigmoid and tanh is very small when x is at extreme values, slowing down the training

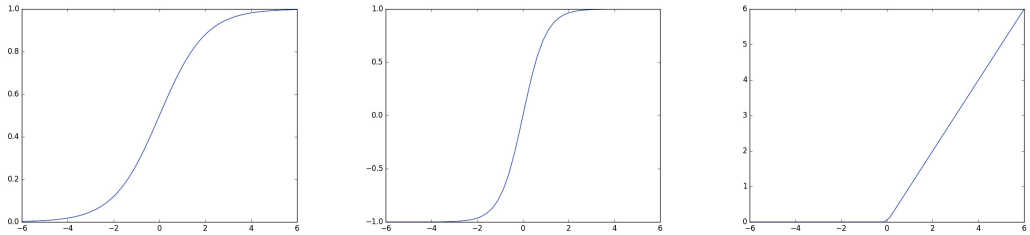


Figure 3.2: Left: Sigmoid function, which is bounded by 0 and 1. Middle: Tanh function, which is bounded by -1 and 1. Right: ReLU function, which is bounded by 0 and is linear (slope of 1) otherwise.

time. Also, having an output of 0 allows the outputs to be more sparse, improving computation time.

Finally, as opposed to the activation functions that apply directly to each node, there is the softmax function that compares the value of all outputs of a layer.

$$\text{softmax}(\mathbf{y})_i = \frac{\exp^{y_i/t}}{\sum_{y_j \in \mathbf{y}} \exp^{y_j/t}}$$

There's a temperature t parameter that can affect the entropy of the distribution, either making the distribution flatter and more stochastic or closer to a delta. The output sums up to one, giving a probability distribution over the inputs of the softmax. In reinforcement learning it can be used to calculate the probability of selecting each action for a given policy.

3.3 Convolutional Neural Networks

A major problem with vanilla neural networks is that they can only work on vectorized data. Any data with multiple axes such as images or temporal data must be flattened before being used. One can see that such a restriction can be costly. This means that spatial and temporal information is lost.

For example, in a image recognition task, it doesn't matter whether an object is centered or slightly to the side. However, if we're flattening the image, the resulting vectors will be hard compare and see the similarity. A neural network would learn to

give a particular weight to each pixel in an image, regardless of neighbouring pixels, discarding valuable information. Furthermore, a neural network has a weight for each input, which means it has to learn to recognize different shapes for every location on an image. Convolutional neural networks (CNN) fix these issues.

CNNs are translation-invariant neural networks which shared weights across inputs. They are composed of small filters that convolve over an entire image and signal how strong of a match each area is. Essentially, it's like performing a small linear regression on a subset of the inputs, and performing this on all possible areas. If the area resembles the filter, the output will be high. These filters can learn to recognize simple shapes once, and then recognize it anywhere on future images. This is a powerful property that greatly helps for generalization.

For each filter, there's an output, resembling a heat-map, signaling the areas matching the filters. The following layer then has filters of its own and the process is repeated. When the input has more than one channel, the filter is a three dimensional tensor, searching for a combination of patterns on each channel. Like fully connected layers, CNNs learn a hierarchical representation of features. In face recognition, it's akin to learning to see simple lines and colors, then the eyes, nose and mouth, and finally a face.

In a given layer that has K channels and with filters of I by J dimensions, we would have the following function for a single output channel. For every new channel we want to output, we need the same function with a new set of weights.

$$\text{Conv}(\mathbf{x})_{x,y} = \sum_{i=0}^{I} \sum_{j=0}^{J} \sum_{k=0}^{K} w_{i,j,k} * \mathbf{x}_{x+i-((I-1)/2),y+j-((J-1)/2),k}$$

Finally, after a few convolutional layers there usually follows a few fully connected layers before the final outputs.

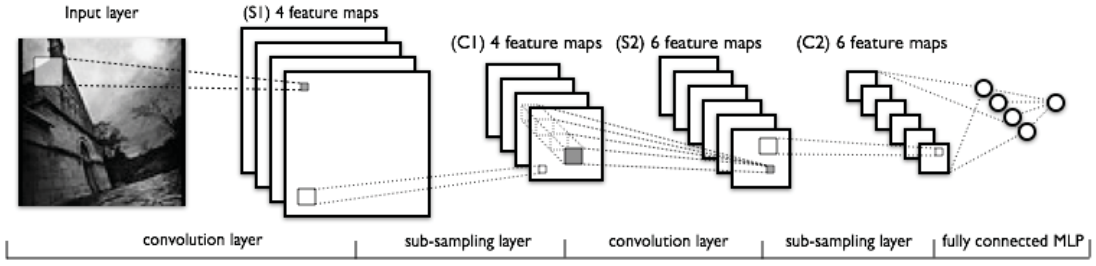


Figure 3.3: Example of a convolutional neural network architecture. Source: http://deeplearning4j.org/convolutionalnets.html

3.3.1 Pooling

Pooling is a function that helps with dimensionality reduction by merging neighbouring outputs of a convolutional layer. The most used pooling functions are the max and mean functions.

For example, performing a 2x2 pooling reduces the dimensionality of the output 4-fold. And doing so on sequential layers reduces it exponentially. The loss of information while doing so is usually minimal as each neighbour in the operation is taken into consideration.

3.3.2 Stride

Stride is a similar function to pooling, but applies the convolution to every n^{th} neighbour, reducing computation by n times. It would change the convolutional function to the following. The image indices are multiplied by n, skipping everything in between.

$$\text{Conv}(\mathbf{x})_{x,y} = \sum_{i=0}^{I}\sum_{j=0}^{J}\sum_{k=0}^{K} w_{i,j,k} * \mathbf{x}_{(x*n)+i-((I-1)/2),(y*n)+j-((J-1)/2),k}$$

The effects on dimensionality are the same as in pooling. The major difference is that pooling uses n neighbors and merges them, while stride simply selects the first input and ignores the $n-1$ following ones. There is a trade-off between loss of information and computation speed, but the speedups are substantial while the information loss is usually quite negligible as convolutional layers have heavy overlapping.

3.4 Learning Deep Learning Models

There is a handful of optimization methods to learn neural networks, but the most heavily used method is Stochastic Gradient Descent (SGD).

In SGD, we calculate the gradient of each parameter of a function and take a step in the direction minimizing or maximizing it depending on our objective. After repeating this for a number of steps, we should converge to a solution, if the steps are small enough.

$$W := W - \alpha \nabla_W L(W)$$

where α is the step size that scales the magnitude of change of each weight and $L(W)$ is the loss function which we're optimizing.

If an optimization problem is convex, such as a linear regression, SGD will converge to the global optimum. Neural network optimization, however, is non-convex and SGD will only converge to a local optimum.

To calculate the gradient of each weight, we used backpropagation. This method uses the calculus chain rule to propagate the gradient from the last layer to the first. It eliminates the need to calculate each gradient at a time, which would be an extremely heavy computation. The chain rule is given by

$$\frac{\partial f(g(x,W))}{\partial W} = \frac{\partial f(g(x,W))}{\partial g(x,W)} * \frac{\partial g(x,W)}{\partial W}$$

We then start by calculating the gradients of the weights of the last layer, which need to compute the derivative of the loss function. Next, we get the gradients for the second to last layer, which needs the gradient of the last layer, partially calculated when we got the last gradients. As we can see, each layer needs gradients that were calculated at the following layer, so we can go backwards and efficiently calculate the gradients for all weights.

In practice, there are many tools to calculate the gradients automatically. One such tool that we use in the experiments is Theano (Bergstra et al. 2010). It's a

mathematical framework that allows researchers to create symbolic computational graphs and perform automatic differentiation to easily get the gradient of any function.

3.4.1 RMSProp

Vanilla SGD is slow to converge as it does not use any information on previously calculated gradients.

RMSProp Tieleman and Hinton 2012 is a gradient descent optimization algorithm that uses recent gradients to accelerate the learning process. This is done by normalizing the gradients by the magnitude of the running average of the sum of squares of gradients.

The update is as follows, where decay is a parameter determining the importance of past gradients in the running average, α is the learning rate, and ϵ is a small constant that helps with stabilization in cases where the denominator is close to 0.

$$g := \text{decay} * \text{g} + (1 - \text{decay}) * \left[\frac{\partial E}{\partial W}\right]^2$$

$$W := W - \alpha \frac{\partial E / \partial W}{\sqrt{\text{g} + \epsilon}}$$

There also exists a modified version of RMSProp first used in Graves 2013, called adaptive RMSProp. In this version, the gradients aren't normalized by the running average but by the standard deviation of the recent gradients. This has the effect of decreasing the gradient in directions that have been oscillating. To calculate the standard deviation, we also need to calculate the running average.

$$f := \text{decay} * \text{f} + (1 - \text{decay}) * \frac{\partial E}{\partial W}$$

$$W := W - \alpha \frac{\partial E / \partial W}{\sqrt{\text{g} - \text{f}^2 + \epsilon}}$$

noop (0)	fire (1)	up (2)	right (3)	left (4)
down (5)	up-right (6)	up-left (7)	down-right (8)	down-left (9)
up-fire (10)	right-fire (11)	left-fire (12)	down-fire (13)	up-right-fire (14)
up-left-fire (15)	down-right-fire (16)	down-left-fire (17)	reset* (40)	

Figure 3.4: All action indices on the Atari 2600.

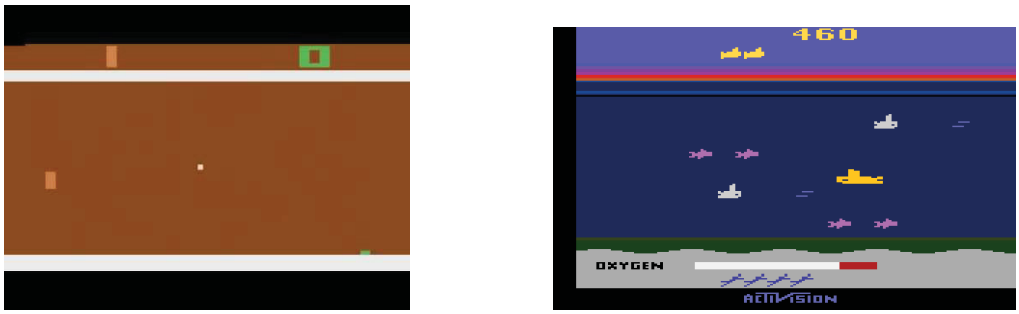
3.5 Arcade Learning Environment

The Arcade Learning Environment (ALE) (Bellemare et al. 2012) is an emulator of the popular arcade platform, the Atari 2600, which can run a set of 58 games. It is specifically designed to train and test machine learning algorithms. The ALE can speed up the simulator as much as possible to train quickly and has channels that can communicate with both C++ and Python.

3.5.1 Games

The games available have a large range of styles, from simple games such as Pong or Breakout where one must use a paddle to return a ball, to more complex puzzle games such Montezuma's Revenge where one must navigate through a series of rooms and must collect objects while avoiding enemies. The variety of games is a good test for AI algorithms as they provide different types of challenges and allow researchers to find flaws in the learning processes. Furthermore, it requires algorithms to be as general and robust as possible to be able to learn on different games without any hand-engineering. When building models that try to learn to play these games, they take the frames as the state, have a variety of actions from which to select (see Figure 3.4) and receive a reward directly from the game, which it tries to maximize.

Two of the games that we will mostly test on are Pong and Seaquest. Figure 3.5 shows example frames from each game.



((a)) Pong ((b)) Seaquest

Figure 3.5: Example of frames from Atari 2600 games.

3.5.1.1 Pong

Pong is a classic game where two opponents on opposite sides of the screen control a paddle and try to hit a ball to the other side. If a player allows the ball to go past their paddle, the opposite player gets a point. The first player to get 21 points wins the game. In the training setting, the learning agent controls one paddle and a fixed computer algorithm controls the second. Pong was selected as it presents the easiest environment to optimize and allows us to know the algorithms capacity to learn at all.

3.5.1.2 Seaquest

Seaquest is a single player game where the player controls a submarine and must save humans by floating to them. Furthermore, there are sharks and evil submarines that attack the player. If there is contact with an enemy, the player dies. The player can also shoot bullets at the enemies. Finally, there is an oxygen bar that depletes with time, and if it runs out, the player dies. To fill it up, the player must bring the submarine back to surface. Seaquest was selected as a more challenging game

3.6 Deep Q Networks

Deep Q Networks (DQN) (Mnih, Kavukcuoglu, et al. 2015) is an algorithm where an RL agent can learn to play most ALE games directly from pixel space with no previous knowledge of the game and no hand-engineering. Simply put, DQN is Q-Learning with a Convolutional Neural Network as function approximation. This was a major achievement as the agent learned temporal credit assignment on images it had to learn to recognize.

To stabilize the learning, a few mechanisms had to be implemented. The most important one is experience replay (Lin 1992), which consists of a memory of the last million transitions experienced. This memory is then sampled from uniformly to create a training batch of 32 transitions at every 4 time-steps. There are two reasons for such a mechanism. First is that neural networks generally don't learn well when data is correlated and the uniform sampling behaves as a shuffling of the data. Second is that they can experience forgetting when dealing with non-stationary distributions of data. When performing many gradient updates without seeing certain data, it's easy to see that the learned weights for certain scenarios might get overwritten. Therefore, having a memory of the last million frames ensures that the network will keep seeing old transitions.

Another important mechanism is the frozen target network. In Q-learning, the policy is ϵ-greedy, meaning the agent takes the action with the highest Q-value most of the time. The problem here is that if two actions have similar values in a state and a gradient update is performed, flipping the preferred action, the behaviour can change drastically. Since we're using a few random samples to determine the gradient update, it's easy to be unlucky and have a gradient update that drastically changes the behaviour for the worst. Furthermore, the sample could be of an extreme situation, highly unbalancing the expected value of a state. The solution is to use a frozen network, which is a second set of parameters that remains fixed for a number of

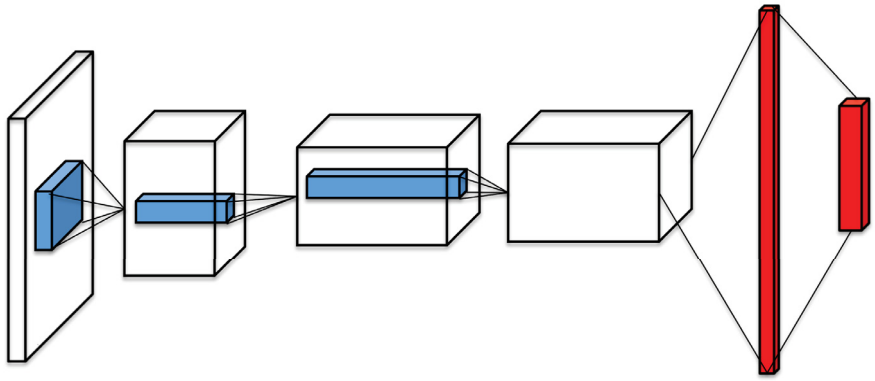


Figure 3.6: Diagram showing the DQN architecture. The CNN learns a representation that is shared for all Q-value functions. Source: Wang, Freitas, and Lanctot 2015.

steps, and is used as the estimate of the next state's value when performing the TD update. Remaining fixed forces the acting network to stay relatively close to the fixed networks' estimates, as to not change too drastically. In the implementation, the acting network updates the frozen network every 10000 frames..

The DQN algorithm also uses reward clipping, where positive rewards are seen as 1, negative ones as -1 and 0 remains as is. This is to ensure that reward scales across different games does not affect the learning, as the gradients are proportional to the reward size.

DQN repeats the chosen action for four consecutive frames. This allows the network to play four times as many games, as it spends no time processing the repeated actions. Furthermore, the images fed to the convolutional neural network belong to every fourth frame, skipping three at a time. Also, as these games are built for humans, which can't process information of every frame, there is generally very little loss of information by skipping three frames.

The architecture of the convolution network is as follows. The input is four consecutive frames (ignoring the skipped ones between each pair). The first layer consists of 32 filters of 8x8 dimensions with a stride of 4. The second layer has 64 4x4 filters with a stride of 2. The third layer has 64 3x3 filters with no stride. None of the layers

use pooling and they all use rectified linear units as activation functions. The output of the last layer is then fed to a fully connected layer of 512 units, using the same activation function. Finally, these are fed to the output layer, a fully connected layer with one output per action and with no activation function. Each of these outputs represents the Q-value of different actions. Depending on the game, there is a different number of actions available.

Adaptive RMSProp is used as the learning algorithm, with a learning rate of 0.00025, a decay of 0.95, and an epsilon of 0.01. As stated earlier, this substantially accelerates learning speeds as RMSProp uses past gradients to reduce the size of fluctuating ones.

The ϵ parameter is decayed linearly over the first million frames, starting at 100% and ending at 10%, remaining at that level for the rest of training. This allows for exploration throughout the training process. Starting out with high randomness allows the agent to try a large combination of moves and learn which are valuable early in the process. However, as the agent learns to act well in the early stages of a game, randomness is still required to learn potentially better actions in later stages.

Finally, the model is trained for 50 million frames.

3.6.1 Double DQN

As explained in 2.2.7, double Q-learning is a variant of Q-learning where we use two Q functions to decrease over-estimation. Double Q-learning is introduced to the Atari games in Van Hasselt, Guez, and Silver 2015, which led to substantial performance improvements.

The algorithm is slightly changed when applied to DQN. In the original Double Q-learning paper, we sample one of two networks as the trainee and use the other network for the greedy policy. In DQN however, we already have a second network, the frozen network, and as it's routinely updated there is no need to learn it. This

eliminates the need to sample one of the two networks to train. Instead, we only learn the first network and change the target from

$$Y^{\text{DQN}} = R_t + \gamma \max_a Q(s_{t+1}, a; \theta^-)$$

to

$$Y^{\text{Double DQN}} = R_t + \gamma Q(s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a; \theta); \theta^-)$$

where θ^- represents the parameters of the frozen network.

4

Temporal Abstraction

4.1 The Options Framework

Options (R. S. Sutton, Precup, and S. Singh 1999) are a framework in RL that allow for hierarchical decision making. The general idea is that an agent has a policy over policies. The agent chooses a long-term policy (the option), and then follows this policy for some time, making small decisions within it. Once the option is complete, the agent chooses a new option to execute. The option can be seen as a decision such as grabbing a cup, and the actions within it are the finer details of which muscles to use to grab the cup.

As opposed to regular policies that simply consist of a state to action mapping, each option has three components. A policy π that represents the behaviour $\pi_\omega : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ where ω is a specific option, a termination function β that represents the probability of terminating the current option and selecting a new one $\beta_\omega : \mathcal{S} \mapsto [0, 1]$, and an initiation set containing all states from which the option can start from $\mathcal{I}_\omega \subseteq \mathcal{S}$. Finally, we must have a policy over options $\pi_\Omega : \mathcal{S} \times \Omega \mapsto [0, 1]$ where Ω is the set of options.

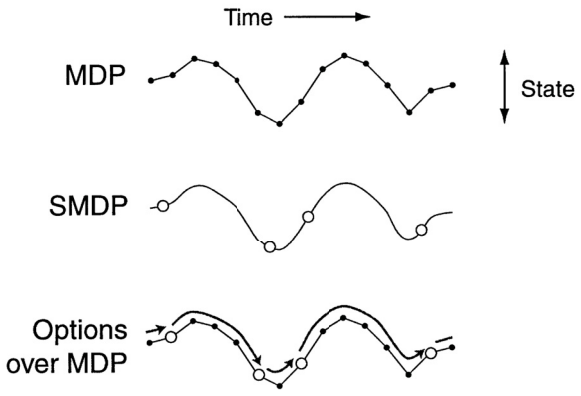


Figure 4.1: Semi-Markov Decision Process. The use of options makes states depend on past decisions, which is not entirely markovian.

4.1.1 Semi Markov Decision Processes

In traditional reinforcement learning, we work under MDPs, where the current state is completely independent from the past and each action is applied for exactly one step. However, a policy over options selects an option that will be executed for a random number of steps. We call this framework the Semi-Markov Decision Process (SMDP) (Puterman 1994). The option policies themselves are applied at every time step, being more like an MDP, but the past information about which option was executed does affect the future, meaning the states aren't entirely Markovian in this setting.

4.2 The Option-Critic Architecture

In this section, we present the option-critic architecture. This framework allows an agent to discover options automatically as it acts in the environment. The name comes from the fact that it's similar to the actor-critic architecture, where an agent has a policy (the actor) and a critic that evaluates the policy. The option-critic has a policy over many options, each of which can be seen as an actor, and a critic which evaluates all options and their policies. Figure 4.2 shows the difference and similarities between the actor-critic and option-critic architectures.

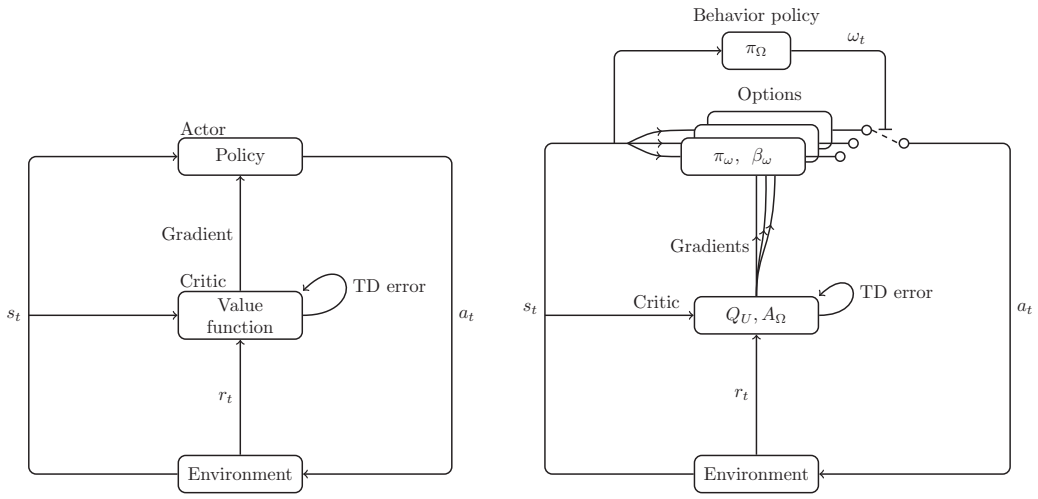


Figure 4.2: A comparison of the Actor-Critic and Option-Critic architectures. Instead of having a single actor, the Option-Critic architecture has a behavior policy that chooses which of many "actors" we should use.

In this architecture, the policy over options is an ϵ-greedy policy learned by Q-learning. This policy is in charge of selecting a temporally extended option, similar to how traditional policies select an action, which allows the agent to make longer term decisions. When the agent chooses an option, it executes it and follows its policy until termination. Each option policy is a stochastic distribution from which actions are sampled. Termination is a Bernoulli random variable that is conditionally dependent on state and option. When we terminate an option, we go back to the ϵ-greedy policy over options, which chooses a new option to execute. Finally, the initiation set of each options is the set of all states, meaning any option can be launched at any state.

The function for the Q-value of a state-option pair is given by the following, simply being an expectation over the probability of selecting each action.

$$Q_\Omega(s, \omega) = \sum_a \pi_{\omega,\theta}(a|s) Q_U(s, \omega, a)$$

The Q-value of a state-option-action triple can be calculated from the one-step return. Within the expectation over next states, we have the utility term $U(\omega, s')$, which sums the cases when the option terminates and when it doesn't. If it does, then we simply have the value of the next state, since the policy over policies will choose a new option to execute. If the option continues, then we have the Q-value of the next

state with the same option.

$$Q_U(s,\omega,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a)U(\omega,s')$$

$$U(\omega,s') = (1-\beta_{\omega,\vartheta}(s'))Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')V_\Omega(s')$$

As in the actor-critic algorithm, we can calculate the gradient of every component as to maximize the expected return for all states.

4.2.1 Intra-Option Policy Gradient

The first gradient we must calculate is the intra-option policy gradient. This is the policy gradient for one of the options. Its target is to maximize the expected return of a state, given by the following function.

$$\rho(\theta) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t \middle| s_0, \omega_0, \Omega\right] = Q_\Omega(s_0,\omega_0)$$

Which we then derive as follows.

$$\frac{\partial Q_\Omega(s,\omega)}{\partial\theta} = \frac{\partial}{\partial\theta}\sum_a \pi_{\omega,\theta}(a|s)Q_U(s,\omega,a)$$

$$= \sum_a \left(\frac{\partial \pi_{\omega,\theta}(a,s)}{\partial\theta}Q_U(s,\omega,a) + \pi_{\omega,\theta}(a|s)\frac{\partial Q_U(s,\omega,a)}{\partial\theta}\right)$$

$$= \sum_a \left(\frac{\partial \pi_{\omega,\theta}(a,s)}{\partial\theta}Q_U(s,\omega,a) + \pi_{\omega,\theta}(a|s)\sum_{s'}\gamma P(s'|s,a)\frac{\partial U(s,\omega)}{\partial\theta}\right)$$

Expanding $\frac{\partial U(s,\omega)}{\partial\theta}$, we get

$$\frac{\partial U(s,\omega)}{\partial\theta} = (1-\beta_\omega(s'))\frac{\partial Q_\Omega(s',\omega)}{\partial\theta} + \beta_\omega(s')\frac{\partial V_\Omega(s')}{\partial\theta}$$

$$= (1-\beta_\omega(s'))\frac{\partial Q_\Omega(s',\omega)}{\partial\theta} + \beta_\omega(s')\sum_{\omega'}\pi_\Omega(\omega'|s')\frac{\partial Q_\Omega(s',\omega')}{\partial\theta}$$

$$= \sum_{\omega'}((1-\beta_\omega(s'))\mathbf{1}_{\omega'=\omega} + \beta_\omega(s')\pi_\Omega(\omega'|s'))\frac{\partial Q_\Omega(s',\omega')}{\partial\theta}$$

We then insert it back into the gradient function.

$$\frac{\partial Q_\Omega(s,\omega)}{\partial\theta} = \sum_a \frac{\partial \pi_{\omega,\theta}(a,s)}{\partial\theta}Q_U(s,\omega,a)$$
$$+ \pi_{\omega,\theta}(a|s)\sum_{s'}\gamma P(s'|s,a)\sum_{\omega'}((1-\beta_\omega(s'))\mathbf{1}_{\omega'=\omega} + \beta_\omega(s')\pi_\Omega(\omega'|s'))\frac{\partial Q_\Omega(s',\omega')}{\partial\theta}$$

$$= \sum_a \frac{\partial \pi_{\omega,\theta}(a,s)}{\partial \theta} Q_U(s,\omega,a) + \sum_{s'}\sum_{\omega'} \Pr(s \to s', \omega \to \omega', 1) \frac{\partial Q_\Omega(s',\omega')}{\partial \theta}$$

And similarly to the policy gradient theorem, we have a recursion where we have a tree of gradients. Many state-option combinations are seen multiple times, so we can sum their gradients discounted by the probability of seeing them at every number of steps.

$$\begin{aligned}\frac{\partial Q_\Omega(s,\omega)}{\partial \theta} &= \sum_s \sum_\omega \sum_{k=0}^{\infty} \Pr(s \to s', \omega \to \omega', k) \sum_a \frac{\partial \pi_{\omega,\theta}(a|s)}{\partial \theta} Q_U(s,\omega,a) \\ &= \sum_s \sum_\omega d_\Omega(s,\omega) \sum_a \frac{\partial \pi_{\omega,\theta}(a|s)}{\partial \theta} Q_U(s,\omega,a)\end{aligned}$$

4.2.2 Termination Gradient

For the termination gradient, we derive the performance metric that evaluates the expected sum of rewards when transitioning into a new state s', but already having executed an option ω in the past. We parametrize the termination function with ϑ to distinguish it from the parameters of the intra-option policy θ.

$$\rho(\vartheta) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t \middle| s', \omega, \Omega\right] = U(s',\omega)$$

We start by expanding U.

$$U(s',\omega) = (1 - \beta_{\omega,\vartheta}(s')) Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s') V_\Omega(s')$$

$$= (1 - \beta_{\omega,\vartheta}(s')) \sum_a \pi_{\omega'}(a|s') \left(r(s',a) + \sum_{s''} \gamma P(s''|s',a) U(\omega, s'') \right)$$

$$+ \beta_{\omega,\vartheta}(s') \sum_{\omega'} \pi_\Omega(\omega'|s') \sum_a \pi_{\omega'}(a|s') \left(r(s',a) + \sum_{s''} \gamma P(s''|s',a) U(\omega', s'') \right)$$

Now to calculate the gradient.

$$\begin{aligned}\frac{\partial U(\omega,s')}{\partial \vartheta} &= \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta} (V_\Omega(s') - Q_\Omega(s',\omega)) \\ &\quad + (1 - \beta_{\omega,\vartheta}(s')) \sum_a \pi_{\omega'}(a|s') \sum_{s''} \gamma P(s''|s',a) \frac{\partial U(\omega,s'')}{\partial \vartheta} \\ &\quad + \beta_{\omega,\vartheta}(s') \sum_{\omega'} \pi_\Omega(\omega'|s') \sum_a \pi_{\omega'}(a|s') \sum_{s''} \gamma P(s''|s',a) \frac{\partial U(\omega,s'')}{\partial \vartheta}\end{aligned}$$

$$= \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta}(V_\Omega(s') - Q_\Omega(s',\omega))$$
$$+ \sum_{\omega'}\sum_{s''}((1-\beta_{\omega,s}(s'))\mathbf{1}_{\omega'=\omega} + \beta_{\omega,\vartheta}(s')\pi_\Omega(\omega'|s'))\sum_a \pi_{\omega'}(a|s')\gamma P(s''|s',a)\frac{\partial U(\omega',s'')}{\partial \vartheta}$$
$$= \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta}(V_\Omega(s') - Q_\Omega(s',\omega)) + \sum_{\omega'}\sum_{s''}\Pr(s' \to s'', \omega \to \omega', 1)\frac{\partial U(\omega',s'')}{\partial \vartheta}$$

We now have the same recursion as the other gradients.

$$\frac{\partial U(\omega,s')}{\partial \vartheta} = \sum_{\omega'}\sum_{s''}\sum_{k=0}^{\infty}\Pr(s' \to s'', \omega \to \omega', k)\frac{\partial \beta_{\omega',\vartheta}(s'')}{\partial \vartheta}(V_\Omega(s'') - Q_\Omega(s'',\omega'))$$
$$= \sum_{\omega'}\sum_{s''} d_\Omega(s',\omega)\frac{\partial \beta_{\omega',\vartheta}(s'')}{\partial \vartheta}(V_\Omega(s'') - Q_\Omega(s'',\omega'))$$

Finally giving us the following theorem.

$$\frac{\partial U(\omega_0,s_1)}{\partial \vartheta} = -\sum_{s',\omega} d_\Omega(s',\omega)\frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta}A_\Omega(s',\omega)$$

where A_Ω is the advantage over options $A_\Omega(s',\omega) = Q_\Omega(s',\omega) - V_\Omega(s')$.

Intuitively, the gradient makes a lot of sense. It's stating that if the state-option's value is higher than that of the state, then we should decrease the probability of terminating it. If it's lower than the state's value, then we increase termination probability. It's also interesting that the advantage function naturally appears in the gradient as opposed to the policy gradient, where people add it as a baseline.

4.2.3 Estimating Q_U

When the action space is large, it might be difficult to learn Q_U. There would be a total of $N_\Omega * N_A$ independent outputs to learn. Instead calculating it directly, it's possible to estimate Q_U from Q_Ω. Since $Q_U(s,\omega,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a)U(\omega,s') = r(s,a) + \gamma\, \mathbb{E}_{s'\sim P}[U(\omega,s')|s,a]$, we can simply use the last transition and reward as a sample for the expectation.

4.2.4 Algorithm

Putting it all together, we get the following pseudo-code for the option-critic architecture in a function approximation setting.

$s \leftarrow s_0$
Choose ω according to a policy over options $\pi_\Omega(s)$
repeat
  Choose a according to $\pi_{\omega,\theta}(a|s)$
  Take action a in s, observe s', r

$$\hat{y} = \begin{cases} r & \text{s' is terminal,} \\ r + \gamma \left[(1 - \beta_{\omega,\vartheta}(s')) Q_\Omega(s', \omega) + \beta_{\omega,\vartheta}(s') \max_{\bar{\omega}} Q_\Omega(s', \bar{\omega}) \right] & \text{otherwise.} \end{cases}$$

$$\xi \leftarrow \frac{\partial (\hat{y} - Q_{U,\xi}(s, \omega, a))^2}{\partial \xi}$$

$$\theta \leftarrow \theta + \alpha_\theta \frac{\partial \log \pi_{\omega,\theta}(a|s)}{\partial \theta} Q_{U,\xi}(s, \omega, a)$$

$$\vartheta \leftarrow \vartheta - \alpha_\vartheta \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta} \left(Q_\Omega(s', \omega) - V_\Omega(s') \right)$$

  if $\beta_{\omega,\vartheta}$ *terminates in* s' **then** choose new ω according to the policy over options $\pi_\Omega(s')$
  $s \leftarrow s'$
until s' *is terminal*

Algorithm 5: Option-critic with intra-option Q-learning

4.2.5 Regularization

It is easy to add regularization functions to the gradients to moderate the learning behavior.

4.2.5.1 Termination Regularization

If a greedy policy over options is chosen, the average termination probability can slowly increase to 100% while training. This is caused by the fact that the value of a state is $V(s) = \max_\omega Q_\Omega(s, \omega)$. Looking at the termination gradient, we can see that

$$\frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta} \left(Q_\Omega(s', \omega) - V_\Omega(s') \right) = \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta} \left(Q_\Omega(s', \omega) - \max_\omega Q_\Omega(s', \omega) \right)$$

where

$$Q_\Omega(s', \omega) - \max_\omega Q_\Omega(s', \omega) \leq 0$$

As the derivative of the termination function is multiplied by this value, the gradient descent will always make the termination probability increase, leading to an option

termination at every step at convergence.

To deal with this, a simple regularization parameter can be implemented. The new gradient update is

$$\vartheta \leftarrow \vartheta - \alpha_\vartheta \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta} \Big(Q_\Omega(s', \omega) - \max_\omega Q_\Omega(s, \omega) + \eta_\beta \Big)$$

This regularization can be seen as an incentive to extend the current option whenever its value is close to the best one. That is, if the Q-value of the current option is within η_β of the maximum Q-value, then the derivative will be multiplied by a positive number, extending the option by decreasing the termination probability.

As explained, the need for regularization stems from the greedy policy. In the case of a stochastic policy over options, the value function is a weighted sum over all Q-values, making the value smaller than the highest Q-value. In this case, the gradient will naturally decrease the termination probability of options with Q-values larger than the value of the state, eliminating the need for this regularization.

4.2.5.2 Entropy Regularization

Another problem encountered was that option policies often learned to output deterministic action distributions early in the training process. This isn't necessarily bad behaviour, but it does limit exploration, which stops the agent from discovering better policies. Motivated by the Asynchronous Advantage Actor Critic paper Mnih, Badia, et al. 2016, we add an entropy regularization to the intra-option gradient.

In information theory, entropy represents how much information can be gained from a message, meaning how unpredictable it is. If one knows what to expect, then no information can be gained. Therefore, when the distribution is a delta, we know what's to come. Entropy is always greater or equal to 0. The highest entropy comes from a uniform distribution, while the lowest is from a delta. Using it as a regularization function causes the policy to output a more stochastic distribution.

The entropy cost is calculated as follows

$$H(\pi_{\omega,\theta}(\cdot|s)) = -\sum_{a} \pi_{\omega,\theta}(a|s) \log(\pi_{\omega,\theta}(a|s))$$

The gradient now becomes

$$\theta \leftarrow \theta + \alpha_\theta \left[\frac{\partial \log \pi_{\omega,\theta}(a|s)}{\partial \theta} Q_U(s,\omega,a) + \eta_H * \frac{\partial H(\pi_{\omega,\theta}(\cdot|s))}{\partial \theta} \right]$$

4.2.6 Analysis of the Algorithm

The option-critic architecture, allowing us to automatically discover options, has its advantages, but also has drawbacks. On one hand, options help propagate value updates back at a larger timescale, accelerating the learning. They also help generalization in the multi-task setting, as the agent can use previously learned options to navigate the environment and explore more efficiently, as shown in Bacon, Harb, and Precup 2016.

One the other hand, each option has to be learned independently, adding an overhead cost proportional to the number of options. Furthermore, there is no shared knowledge between options. If one options learns not to take a certain action in a state, it will have to be learned again in other options which would perform similarly.

5

Experiments

In this chapter, we introduce the Deep Option-Critic. Using the option-critic architecture described previously and combining it with the DQN framework, we get an option-critic algorithm that has the ability to learn to play ALE games directly from pixels, while learning options and control over them.

5.1 The Deep Option-Critic Algorithm

To allow for a fair comparison with the DQN results, the architecture and engineered methods were kept as consistent as possible. The architecture of the Option-Critic uses a shared network for all components. The reasoning is that since we start from pixel space, there is no need to learn to recognize the same objects and movements independently for each function. The shared network will work as a feature extraction network, which creates a feature vector that is then fed to each of the option-critic components.

To minimize the number of parameters, we never explicitly use a $Q(s, \omega, a)$, as explained in Subsection 4.2.3. Because of this, we only have three main components, $Q_\Omega(s, \omega)$, $\beta_\omega(s)$ and $\pi_\omega(a|s)$.

The convolutional neural network consists of the same layers as DQN, up to the fully connected layer of 512 neurons, which is used as the shared feature vector.

The policy used over options is ϵ-greedy, and is learned off-policy, by using Q-learning. We used the same ϵ schedule as in DQN. Also as in DQN, we use experience replay to stabilize the network's learning. The gradient descent uses RMSProp with a learning rate of 0.00025, an epsilon of 0.01 and a decay of 0.95.

The termination function is a linear function with a sigmoid activation function to transform the output into a probability between 0 and 1. The is a termination function for each option.

The option policies are linear functions with softmax activations. This transforms the outputs into a distribution, where the sum of outputs is 1. The action used in the environment is then sampled from that distribution.

The termination and option policies are learned online, using the most recent sample in the trajectory to calculate the gradient. The gradient descent on both is vanilla SGD with a learning rate of 0.00025.

All three networks are learned by gradient descent. The shared CNN however, was learned only from the TD error used to learn the policy over options. That is, when performing a step of gradient descent of the policy over options, we use backpropagation throughout the entire CNN, however when performing the descent on the two other networks, we disconnect the backpropagation before the CNN. This means that the feature vector that is output from the CNN is fully learned from the TD error and not from the option policies. This was originally a concern, but empirical evidence showed that the feature vector has sufficient information to learn good policies along with good Q-values. Performing gradient descent from all three networks onto the CNN is left as future work along with architectural choices, such as having a separate CNN for each network.

Algorithm 6 shows a simple case of option-critic where intra-option Q-learning is used in the tabular setting to learn a critic and to derive a policy over option π_Ω. We write α, α_θ and α_ϑ for the learning rates of the critic, intra-option policies and termination functions respectively. ξ^- represents the weights of the frozen network.

Initialize weights θ, ϑ and ξ randomly
$\xi^- \leftarrow \xi$
$s \leftarrow s_0$
$\phi \leftarrow \phi_\xi(s)$
Choose ω according to an ϵ-greedy policy over options $\pi_{\Omega,\theta}(\phi)$
repeat
  Choose a according to $\pi_{\omega,\theta}(a|\ \phi)$
  Take action a in s, observe s', r
  Store s, a, ω, r, s' in Experience Replay
  $\phi' \leftarrow \phi_\xi(s')$

  1. Options evaluation:(**Every** 4 steps)
  Sample 32 transitions (s, a, ω, r, s') from Experience Replay

$$\hat{y} = \begin{cases} r & \text{s' is terminal,} \\ r + \gamma \left[(1 - \beta_{\omega,\vartheta}(s')) Q_{\Omega,\xi^-}(s', \omega) + \ \beta_{\omega,\vartheta}(s') \max_{\bar{\omega}} Q_{\Omega,\xi^-}(s', \bar{\omega}) \right] & \text{otherwise} \end{cases}$$

  Perform RMSProp gradient descent on $\dfrac{\partial (\hat{y} - Q_{\Omega,\xi}(s, \omega))^2}{\partial \xi}$

  2. Options improvement:(**Every** 1 step)

$$\theta \leftarrow \theta + \alpha_\theta \frac{\partial \log \pi_{\omega,\theta}(a|\ \phi)}{\partial \theta} Q_U(s, \omega, a)$$

$$\vartheta \leftarrow \vartheta - \alpha_\vartheta \frac{\partial \beta_{\omega,\vartheta}(\phi')}{\partial \vartheta} \left(Q_{\Omega,\xi}(s', \omega) - \max_{\bar{\omega}} Q_{\Omega,\xi}(s', \bar{\omega}) \right)$$

  if $\beta_{\omega,\vartheta}$ *terminates in* s' **then** choose new ω according to ϵ-greedy $Q_{\Omega,\xi}(s', \cdot)$
  Every 10000 steps $\xi^- \leftarrow \xi$
  $\phi \leftarrow \phi'$
until s' *is terminal*

Algorithm 6: Deep Option-critic with intra-option Q-learning on Atari

In all following plots depicting training curves in the remainder of this chapter, we track the average score during the testing phase at the end of each epoch. To smooth the curves and have an idea of the performance without outliers we also plot the moving average of the training curve with a sliding window of size 10. Finally, we plot two lines, the test performance scores of DQN and Double DQN, as reported in Van Hasselt, Guez, and Silver 2015, used to evaluate the quality of the option-critic's performance. The Y axis is the average score the algorithm achieved in a game, which is at different scales depending on the game. The X axis is the number of epochs of

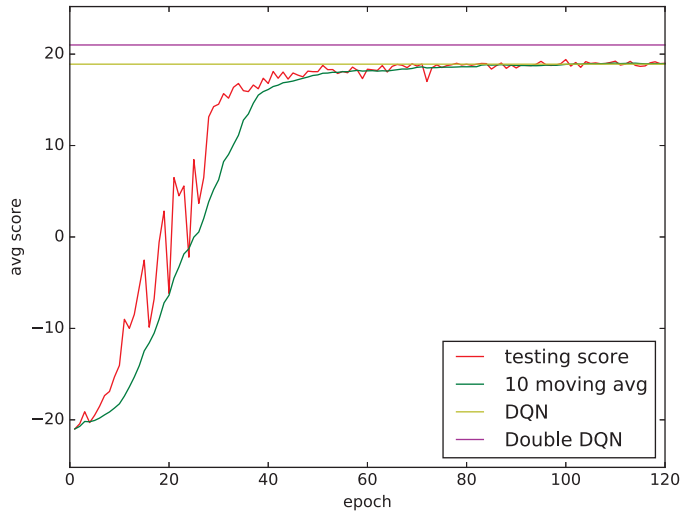


Figure 5.1: Option critic training curve on Pong with fixed options with no regularization.

training, where one epoch is 250000 frames of training.

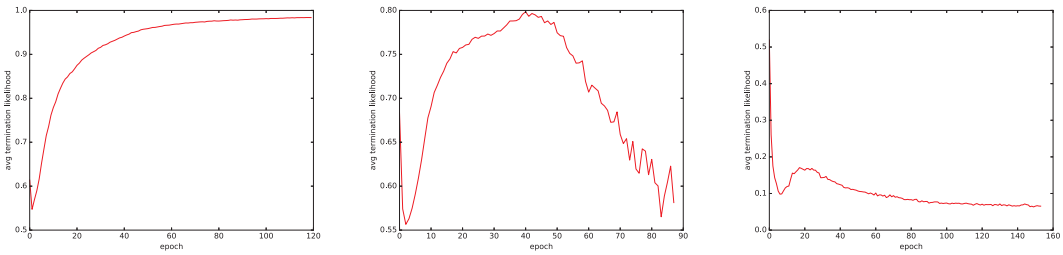
5.2 Fixed Option-Critic

The first algorithm tested was simply to learn the termination function, using fixed options. In this setting, there is one option per action, where each option consists of repeating the same action. The algorithm remains the same, except there is no intra-option policy gradient step.

In this restricted setting, we tested the algorithm on Pong, a simple game where two players each have a paddle and try to hit a ball past the opponent's paddle. There are three possible actions, up, down and no-op (to do nothing).

The first experiment was to run the option-critic algorithm without any regularization on Pong. The final performance, as seen in Figure 5.1, converged to a level similar to DQN. The percentage of steps where termination was executed, leading to the choice of a new option, was also tracked through training.

As we can see in Figure 5.2(a), the probability of terminating simply kept increas-



((a)) 0.0 regularization ((b)) 0.01 regularization ((c)) 0.05 regularization

Figure 5.2: Option critic termination likelihood throughout training time on Pong with fixed options with different regularization levels.

ing during training. As explained earlier, this is due to the advantage function in the Q-value gradient, which we fix with a termination regularization. Termination allows the agent to reevaluate the choice in options, and select the best one, however, it also has a computational cost. In the ALE setting, we have to run the screen through a CNN every time we terminate, to select the option. Furthermore, it was found that the agent changes options only 58% of the time when terminating. This means it re-selects the same option 42% of the time. This is not to say that choosing the same option in sequential steps is bad, on the contrary, the agent should terminate an option when an important step to re-evaluate and choose what's best, which might be the same option. However, at a termination rate of almost 100%, the agent ends up re-selecting an option at almost half of all steps.

To reduce termination likelihood, we tried training models with a termination regularization of 0.01 and 0.05. As we can see in Figure 5.2, termination likelihood is much lower than the original algorithm without regularization. Furthermore, we can see that the likelihood starts by increasing, then plateau's, and finally decreases. In the 0.05 regularization setting, the plateau happens very early in the training. The magnitude of the advantage function is what causes the rate of change in the termination function. That being said, the rate of change clearly decreases as training time passes, meaning the advantage function slowly shrinks. There is probably a combination of two effects contributing to this phenomenon.

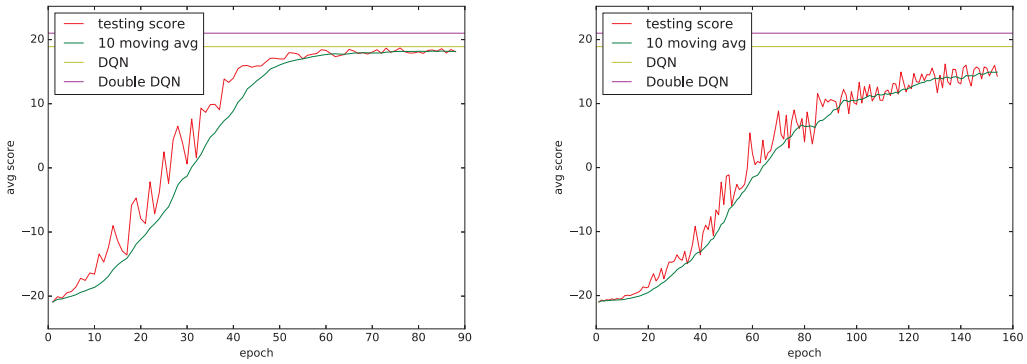


Figure 5.3: Option critic training scores on Pong with fixed options with regularization of 0.01 and 0.05 respectively.

First is that Q-values are initially random, and with the advantage function being 0 if the optimal option is selected, the advantage function will have $\frac{n-1}{n}\%$ probability of being negative, with n being the number of actions.

Second, as the behavior is learned and Q-values become more accurate, the difference in Q-values for different actions in a given state will be negligible, unless we're in a state that crucially requires a certain action to be performed to get some reward. In such a state, performing the wrong action will lead to a lower reward, meaning a low Q-value. Therefore, in most states, which don't have this crucial decision, the advantage function will be small, and the termination regularization will decrease termination probability.

Finally, Figure 5.3 shows the training curves of the model with termination regularization of 0.01 and 0.05, respectively. The most notable difference is the training speed. As regularization is increased, the model takes longer to train. The current hypothesis for the cause is that exploration is modified by the regularization, which can hinder training in some games. As we will see in Subsection 5.3, the termination regularization does not negatively affect all games, and even affect some positively.

As for Pong, the game consists of having the paddle at the right area to return the ball. Furthermore, the direction of the ball returned depends on the location on which it hit the paddle, meaning precision is needed for skillful play. By moving

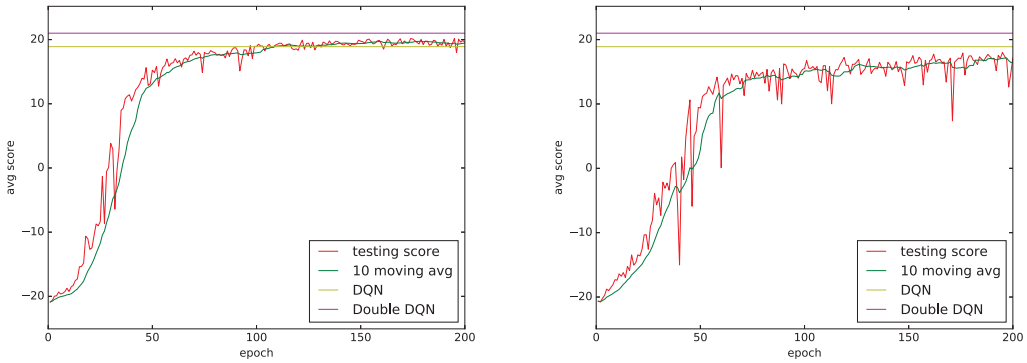


Figure 5.4: Full option critic training scores on Pong with 2 and 8 learned options respectively.

entirely randomly, the paddle ends up returning the ball and seeing valuable moves relatively quickly. However, if the fixed option of consistently pushing the same button is extended by termination regularization, the paddle will end up swinging quickly from one side to the other, making it more difficult to hit the ball, and thus slowing training down. The final step is to learn the full option-critic, including the option policies themselves, in an end-to-end fashion.

5.3 Full Option-Critic

First, we start with Pong. Figure 5.4 shows the training curves when learning 2 and 8 options. The model with 2 options learns quite quickly and gets to a performance level slightly higher than DQN. However, the model with 8 options learns at a slower pace and doesn't quite reach DQN performance. As Pong is a simple game with only 3 actions, the current hypothesis is that having more options than needed simply hurts training speed because the model will have to learn the same policy multiple times independently.

The model was then trained on Seaquest. We first tried it with 8 options and no regularization. Figure 5.5 shows that performance was quite unstable as the score fluctuated frequently between epochs, but performance levels were similar to DQN.

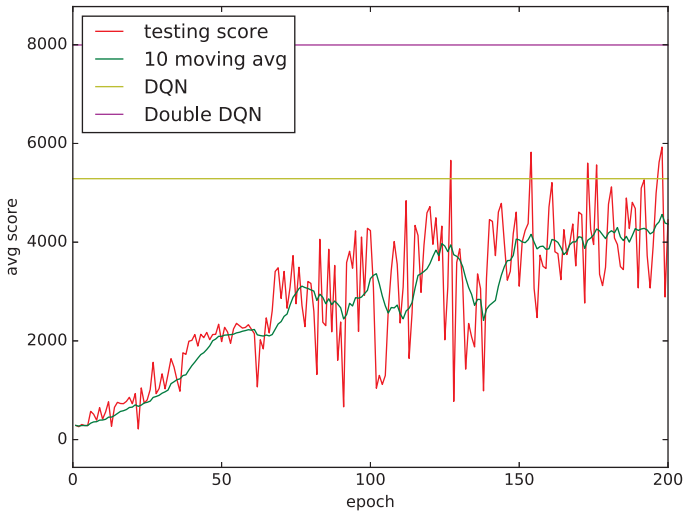


Figure 5.5: Full option critic training scores on Seaquest with 8 learned options, and no regularization.

5.3.1 Termination Analysis

As in the fixed option setting, termination probability steadily increases throughout training, when no regularization is used. We then trained the model with termination regularization at 0.01. Figure 5.6 shows that the termination probability slowed early in the training, but quickly increased later on, signaling that the regularization parameter wasn't high enough. However, Figure 5.7 shows its performance levels and that the model learned much faster and more stable than the model without regularization.

We tested higher levels of termination regularization on the game of Asterix as well. Figures 5.8 and 5.9 show performance and termination rates over 100 epochs of training. We find similar results as before, where training seems to stabilize with an increase in regularization. Also, termination rates are much lower with the 0.1 regularization, as expected. However, we can see that the model with more regularization took longer to reach DQN levels. A possible cause for this is that option policies aren't yet good early in the training and strong regularization forces them to be extended, forcing the agent to use long sequences that can lead it to bad situations.

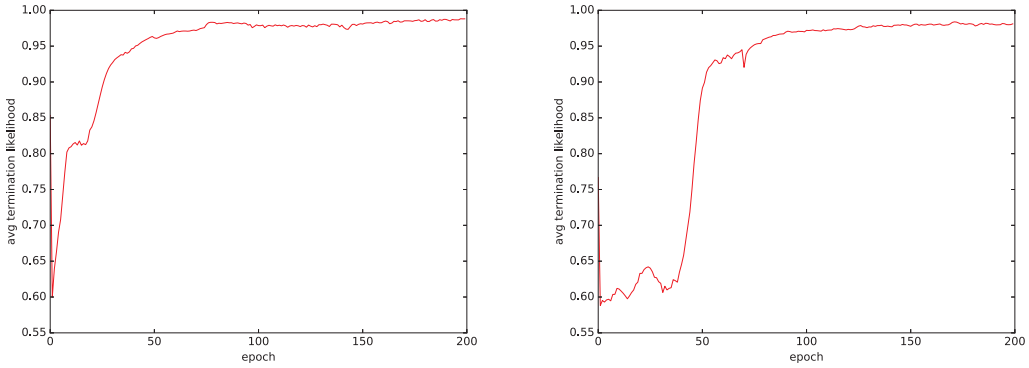


Figure 5.6: Termination rates during training on Seaquest with 0 and 0.01 termination regularization parameters, with 8 options.

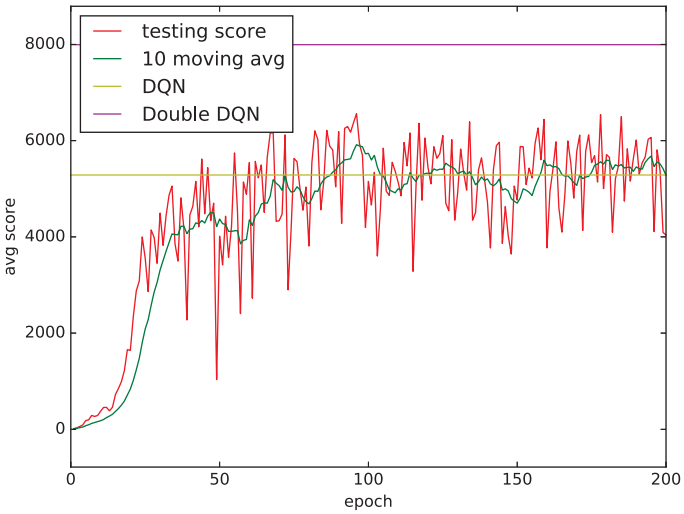


Figure 5.7: Full option critic training scores on Seaquest with 8 learned options, and 0.01 termination regularization.

When termination probability is high, the agent can quickly switch to an option that will perform a better action.

Figure 5.10 has two matrices with heat maps showing the distribution of how often each action was selected in each option. The distribution is applied on each column. It shows the agent learned deterministic policies when termination was low, and slightly more balanced distributions over actions when regularization was strong. When termination probability is high, each option can learn to represent an action, and then allow the policy over options to simply choose actions at each time-step. But

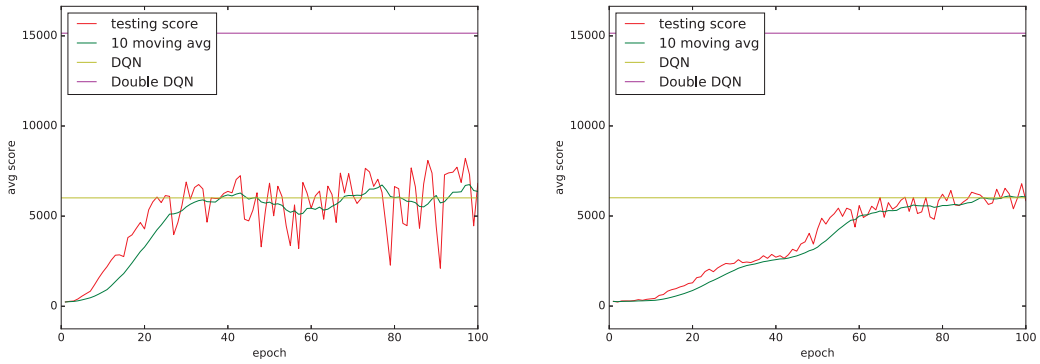


Figure 5.8: Full option critic training scores on Asterix with 0.01 and 0.1 termination regularization parameters, with 8 options.

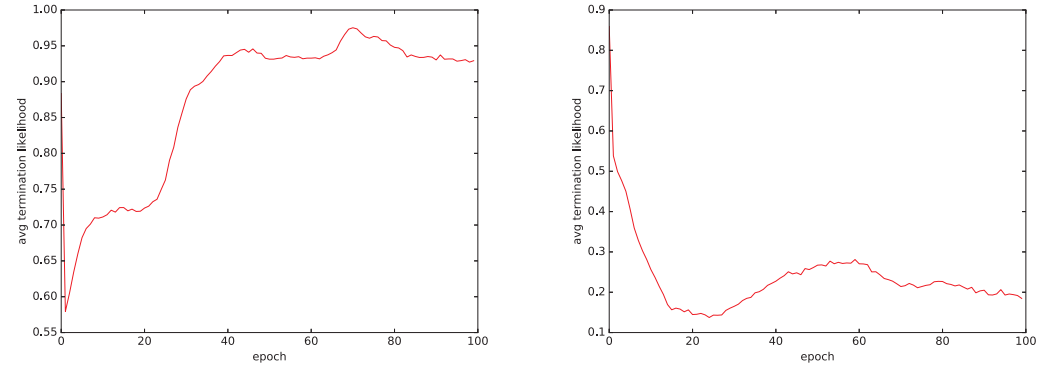


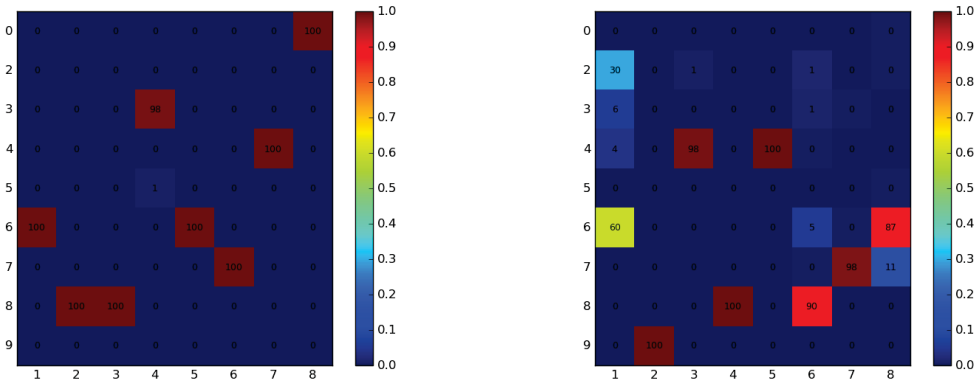
Figure 5.9: Termination rates during training on Asterix with 0.01 and 0.1 termination regularization parameters, with 8 options.

when regularization is strong, the options has to be longer, which forces the policy to be more than just repeating a single action, as it will have to act in many different states.

5.3.2 Option Policy Analysis

Returning to the agent playing Seaquest without regularization, Figure 5.11 shows the distribution of the sampled actions for each option during a single run after training was complete.

The first thing to notice is that the table is quite sparse, and a few options are almost or completely deterministic. Furthermore, those options are the most frequently



((a)) 0.01 termination regularization ((b)) 0.1 termination regularization

Figure 5.10: Action selection distribution per option after training on Asterix with 8 options.

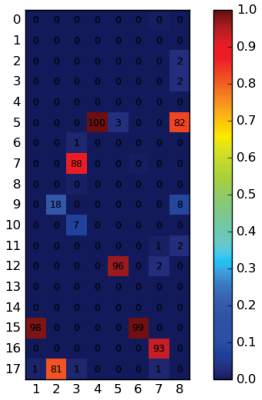


Figure 5.11: The distribution of sampled actions in each of the 8 options from the trained model on Seaquest with no regularization. The action indices represent the moves from Figure 3.4.

used ones. This originally created a concern about the learned shared feature vector. It was thought possible that the CNN output might learn a vector only good enough for the Q-value function and not one with enough information to allow the option policies to learn a state dependent distribution. If that was the case, the options would surely learn to be deterministic and let the greedy policy over options simply choose deterministic actions, exactly like Q-learning.

To test this, we then trained the same model but with only 2 options. However, it learned a similar, almost deterministic distribution in each option, as seen in Figure 5.12. The training curve from same figure shows that the model was also capable of

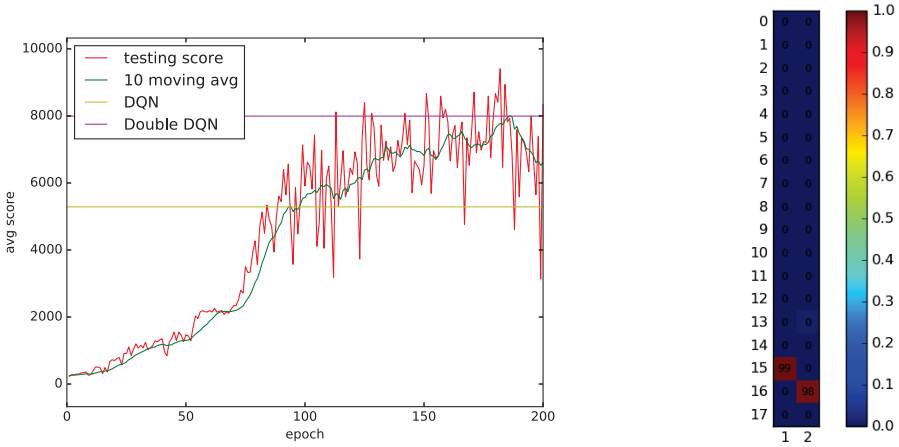


Figure 5.12: Left: Full option critic training scores on Seaquest with 2 learned options, and no regularization. Right: The distribution of sampled actions in each of the 2 options from the trained model on Seaquest with no regularization. The action indices represent the moves from Figure 3.4.

reaching DQN levels even by only using 2 of 18 actions.

5.3.2.1 Entropy Regularization

Even having 2 options didn't allow the options to have good distributions on actions, which enforced the hypothesis that the learned feature vector wasn't good enough for the policies. A second hypothesis was that the options learned to become deterministic because it's the easiest thing to learn when values aren't accurate, as is the case early in training. Also, as mentioned in 4.2.5.2, Mnih, Badia, et al. 2016 faced the problem that the actor learned a deterministic policy early in training, which limited exploration.

We then implemented an entropy regularization added to the intra-option policy gradient. Testing on Seaquest with 8 options, we compare the effect of using no regularization with using entropy regularization of 0.01 in Figure 5.13. The first thing to notice is that the model with entropy regularization learned much faster than the model with none. This is aligned with the theory that the options became deterministic too early, hindering on exploration. The regularization probably kept the options from being deterministic early in the training. That being said, Figure

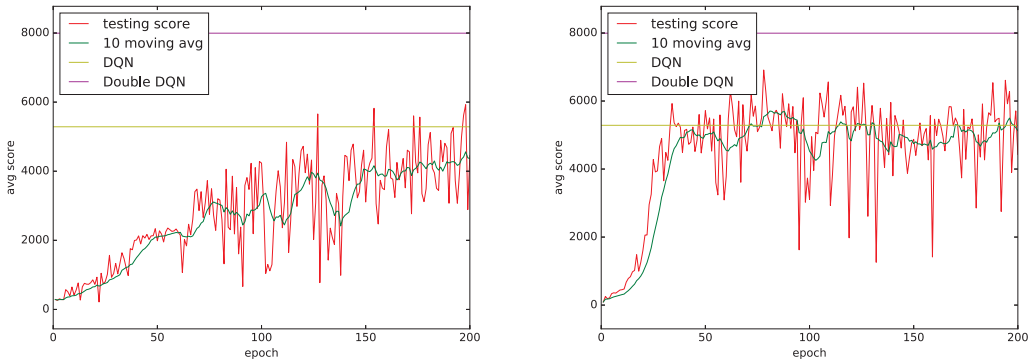


Figure 5.13: Model with 8 options trained on Seaquest with 0 and 0.01 entropy regularization.

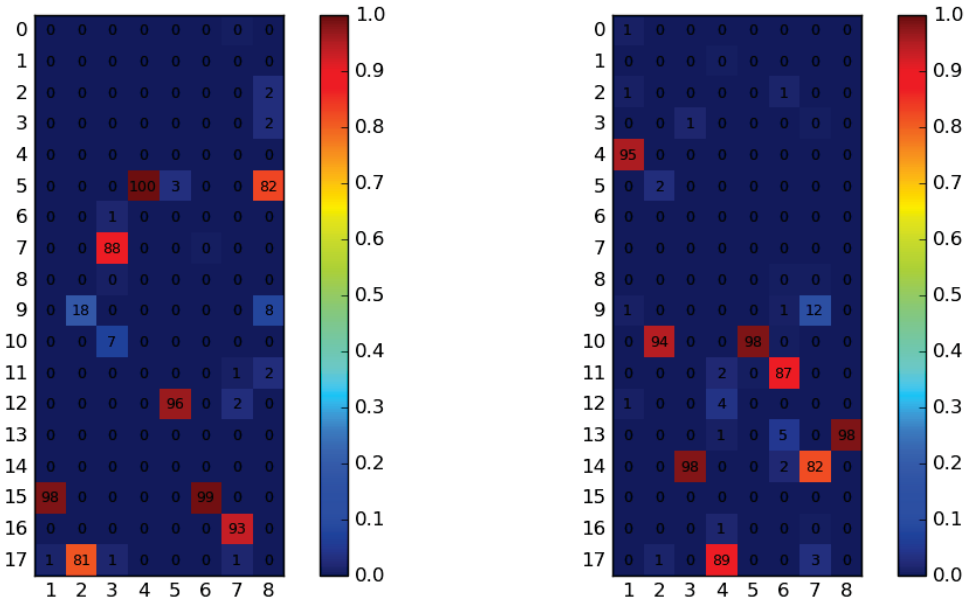


Figure 5.14: Action distributions with 8 options trained on Seaquest with 0 and 0.01 entropy regularization.

5.14 shows that the distribution over actions remained almost exactly the same at the end of training. Entropy regularization clearly had an effect on performance, but was enough to stop the options from being determinisitic.

5.3.2.2 Intra-Option Policy Baseline

At this point however, a third hypothesis came up. In most games, when maximizing rewards, Q-values will end up being positive, and in some games, Q-values are

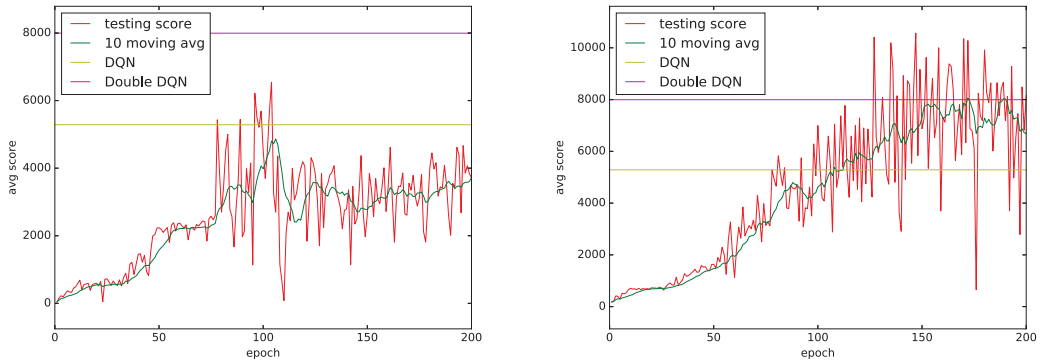
always positive. With the intra-option policy gradient, following the log trick, being $\frac{\partial \rho(\theta)}{\partial \theta} = \mathbb{E}_s \mathbb{E}_a \frac{\partial \log(\pi_{\omega,\theta}(a|s))}{\partial \theta} Q_U(s, \omega, a)$ and sampling the action from the policy to calculate the gradient, the gradient update will almost always make the probability of $\pi(a|s)$ increase for the sampled action. If the sum over actions was kept, as opposed to sampling the action, the gradient for each action would be multiplied by Q, and the highest Q-value would force the sum of gradients to point more towards the best action, increasing its probability as opposed to the other actions. But when sampling the action from π, the gradient will increase the probability of that action as long as its Q-value is positive. This will increase its probability, and the next time the agent visits the state, the probability of choosing that same action is higher than last time, meaning it'll probably get updated and increased again. This would cause a "rich get richer" phenomenon, which would perfectly explain why the option policies became almost deterministic.

To tackle this, a baseline was added to the intra-option gradient, becoming

$$\frac{\partial \rho(\theta)}{\partial \theta} = \mathbb{E}_s \mathbb{E}_a \frac{\partial \log(\pi_{\omega,\theta}(a|s))}{\partial \theta} [Q_U(s, \omega, a) - Q_\Omega(s, \omega)]$$

In this case, $Q_\Omega(s, \omega)$ is equivalent to the usual $V(s)$ as the policy gradient baseline. Instead of multiplying by $Q_U(s, \omega, a)$, which is usually positive, we now use its advantage over $Q_\Omega(s, \omega)$. If a poor action, relative to the state option value, is sampled, the gradient will actually decrease its probability, as its advantage will be negative. This should help the option policy distributions be more balanced. It shouldn't entirely stop policies from being deterministic however, since such a policy could happen to be useful in some environments.

Now to compare some models with exactly the same hyper-parameters except using a baseline versus not. The first model is trained on Seaquest with 8 options, 0.01 termination regularization and 0.01 entropy regularization. Figure 5.15 shows the difference in performance, which is quite substantial. The model using a baseline ends up surpassing double DQN levels. Furthermore, Figure 5.16 shows the difference



((a)) No baseline ((b)) Uses baseline

Figure 5.15: Performance effects of baselines on models with 8 options trained on Seaquest with 0.01 termination and 0.01 entropy regularization.

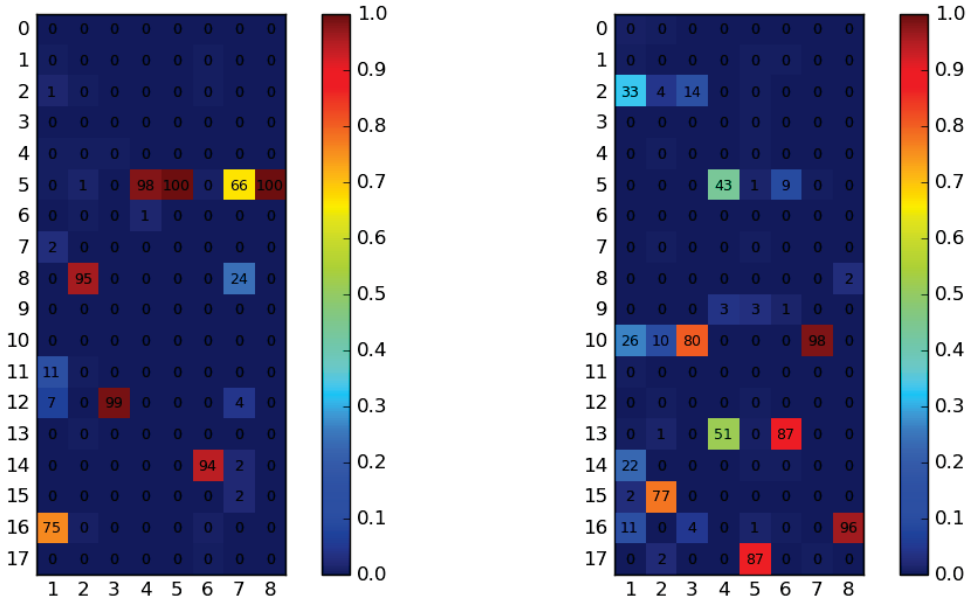
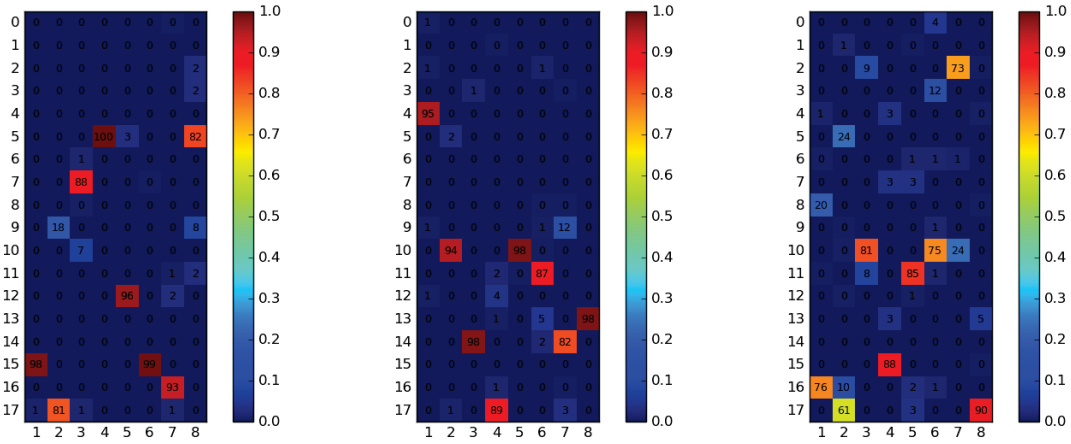


Figure 5.16: Action selection distributions effects of baselines on models with 8 options trained on Seaquest with 0.01 termination and 0.01 entropy regularization. The model on the left is not using a baseline.

in action selection distribution across options in both models. This was the first time that action distribution were this balanced.

Figure 5.17 has more graphs showing the effect of baselines. The first graph has no regularization, while the second has 0.01 entropy regularization. The effect on the distribution is minimal, but the third graph shows the distribution being much more balanced when using 0.01 entropy regularization and a baseline.

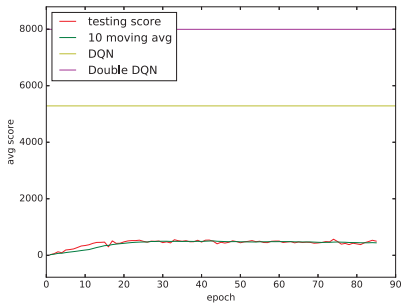


((a)) No regularization ((b)) 0.01 entropy reg. ((c)) 0.01 entropy reg. and baseline

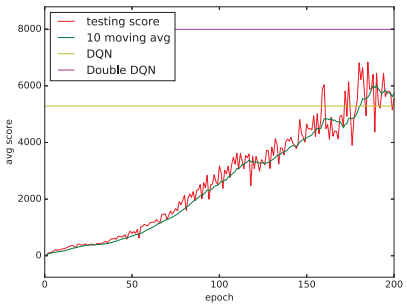
Figure 5.17: Action selection distributions effects of baselines on models with 8 options trained on Seaquest.

Another example showing the effects of the baseline is Figure 5.18. The combination of regularization hyper-parameters appears to have hurt training progress on the model with 2 options. However, there is consistent improvement on the model when using a baseline. Furthermore the action selections shows it's well balanced. Looking at the distribution more closely, we can see that option 1 uses actions 2, 7, 10, 14 and 15, while option 2 uses actions 5, 8, 13, 16, 17. All actions in option 1 use the up button, while all actions from option 2 use the down button. This means the model learned opposite behaviors for each option.

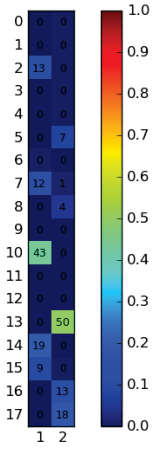
Finally, Figure 5.19 shows the performance plots of a few games with a combination of regularization parameters and a baseline, showing that the algorithm can train in a variety of games with the same hyper-parameters. However, Montezuma's Revenge remains unsolved as its main challenge is exploration.



((a)) No baseline.

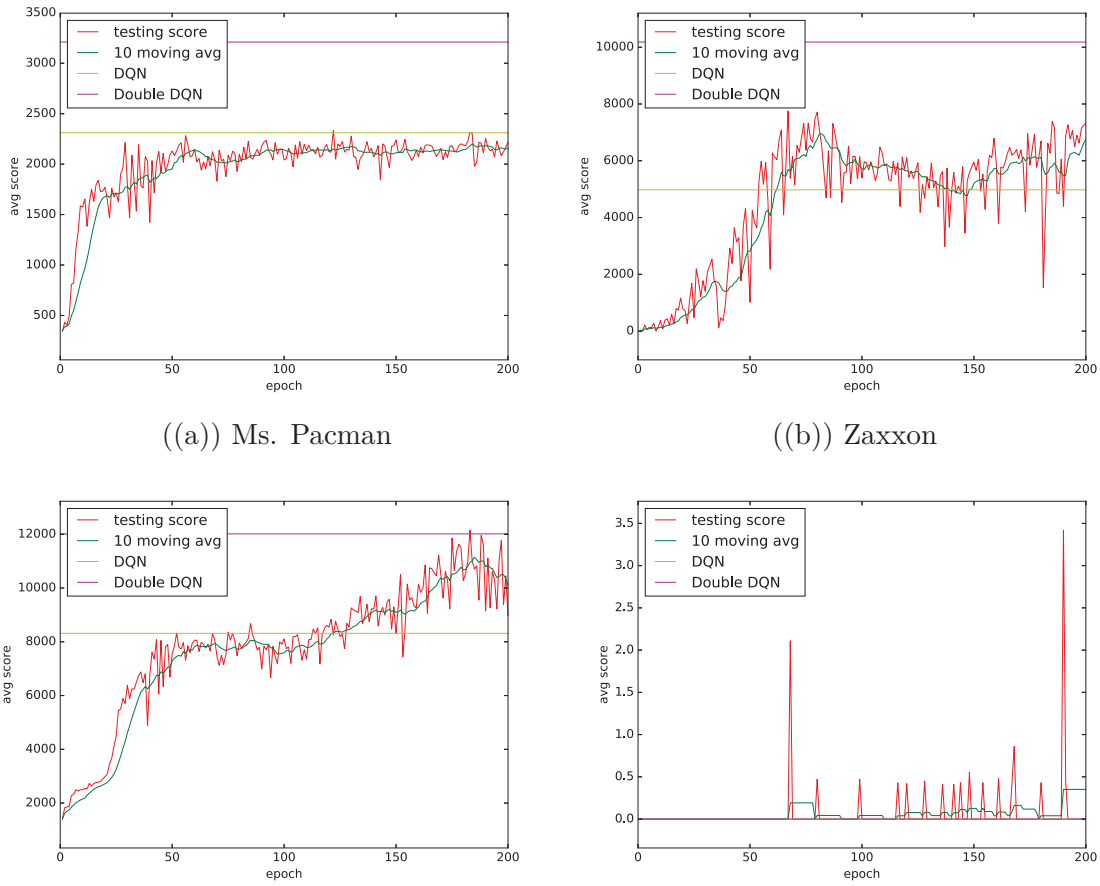


((b)) Uses baseline.



((c)) Uses Baseline. Same as (b).

Figure 5.18: Left and Middle: Performance of models with 2 options trained on Seaquest. Both models have 0.01 termination and entropy regularization. Right: Action distribution of model using a baseline.



((a)) Ms. Pacman

((b)) Zaxxon

((c)) Riverraid, 2 options

((d)) Montezuma's Revenge

Figure 5.19: Training curves on different games with the same set of hyper-parameters: 0.01 termination regularization, 0.01 entropy regularization, and an added baseline. Each game, except Riverraid, was trained with 8 options.

6

Conclusion

In this thesis, we introduced the Deep Option-Critic Architecture, a combination of the option-critic and deep learning models that allows a reinforcement learning agent to jointly learn state and temporal abstraction.

For the first time, options have been learned automatically in an on-line and scalable manner. The algorithm shows that the option-critic can run in very high dimensional state spaces. Having a scalable temporal abstraction framework can have a great impact. This can be useful for fields such as transfer learning, where an option learned in a previous task would allow an agent to execute cohesive sequences of actions in new tasks, substantially improving the learning speed.

We showed the deep option-critic's learning capacity in the Arcade Learning Environment. The algorithm was capable of learning options in different games without any previous knowledge. It achieved DQN or better levels of performance in the Atari domain. We showed the potential effects of regularization on the behaviour of different options and on the performance of the agents. Termination regularization was shown to help extend options which had positive or negative effects depending on the game. Entropy regularization had more of an effect, since it helps the agent explore for a longer period of time. However, it didn't affect the action distributions as much as was hoped. Finally, we showed that using a sampled action, for the expectation over actions in the policy gradient, strongly pushed options to become deterministic.

The addition of a baseline had a substantial effect, both in helping the distribution over actions in options be more balanced and improving performance.

6.1 Future Work

Many architectural questions remain to be explored. In our experiments, we only trained the CNN with the gradient from the Q-functions, disregarding the termination and option policy gradients. Perhaps by combining all three function's gradients, the feature representation appropriate for all functions would be learned much faster. It would also be possible to have a separate CNN for each function.

The termination regularization used was almost always 0.01. It's quite possible that appropriate regularization is game-dependent, since its affect is scaled relatively to advantage values of each game. Also, this regularization was just one approach to push the model to extend option lengths. Another method to be tried is a deliberation cost, where a termination would trigger a negative reward. The agent would then learn to extend options on its own to maximize reward.

It's also important to note that no hyper-parameter search was performed due to the heavy computational demand. Such a search would most probably increase performance quite substantially, due to the sensitivity of performance.

Bibliography

Bacon, Pierre-Luc, Jean Harb, and Doina Precup (2016). "The option-critic architecture". In: *In preparation*.

Bellemare, Marc G et al. (2012). "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research*.

Bellman, Richard (1956). "Dynamic programming and Lagrange multipliers". In: *Proceedings of the National Academy of Sciences* 42.10, pp. 767–769.

Bergstra, James et al. (2010). "Theano: A CPU and GPU math compiler in Python". In: *Proc. 9th Python in Science Conf*, pp. 1–7.

Degris, Thomas, Martha White, and Richard Sutton (2012). "Off-Policy Actor-Critic". In: *International Conference on Machine Learning*.

Graves, Alex (2013). "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850*.

Hauskrecht, Milos et al. (1998). "Hierarchical solution of Markov decision processes using macro-actions". In: *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 220–229.

Howard, Ronald A (1960). *Dynamic programming and Markov processes*. MIT press Cambridge.

Konidaris, George and Andre S Barreto (2009). "Skill discovery in continuous reinforcement learning domains using skill chaining". In: *Advances in Neural Information Processing Systems*, pp. 1015–1023.

Lin, Long-Ji (1992). "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3-4, pp. 293–321.

McGovern, Amy and Andrew G Barto (2001). "Automatic discovery of subgoals in reinforcement learning using diverse density". In:

Mnih, Volodymyr, Adria Puigdomenech Badia, et al. (2016). "Asynchronous methods for deep reinforcement learning". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML-16)*.

Mnih, Volodymyr, Koray Kavukcuoglu, et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Puterman, Martin L (1994). "Markov Decision Processes: Discrete Stochastic Dynamic Programming". In:

Rummery, Gavin A and Mahesan Niranjan (1994). *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering.

Stolle, Martin and Doina Precup (2002). "Learning options in reinforcement learning". In: *International Symposium on Abstraction, Reformulation, and Approximation*. Springer, pp. 212–223.

Sutton, Richard S (1988). "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1, pp. 9–44.

Sutton, Richard S, David A McAllester, et al. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation." In: *NIPS*. Vol. 99, pp. 1057–1063.

Sutton, Richard S, Doina Precup, and Satinder Singh (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1, pp. 181–211.

Thomas, Philip (2014). "Bias in Natural Actor-Critic Algorithms." In: *ICML*, pp. 441–448.

Tieleman, Tijmen and Geoffrey Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2.

Van Hasselt, Hado (2010). "Double Q-learning". In: *Advances in Neural Information Processing Systems*, pp. 2613–2621.

Van Hasselt, Hado, Arthur Guez, and David Silver (2015). "Deep reinforcement learning with double Q-learning". In: *CoRR, abs/1509.06461*.

Wang, Ziyu, Nando de Freitas, and Marc Lanctot (2015). "Dueling network architectures for deep reinforcement learning". In: *arXiv preprint arXiv:1511.06581*.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4, pp. 229–256.