Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Curriculum Learning for Robot Manipulation using Deep Reinforcement Learning

Author:
Diego MENDOZA BARRENECHEA

Supervisor:
Dr. Edward JOHNS

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science / Artificial Intelligence of Imperial College London

June 2017

Abstract

In this project we evaluate the performance of using a curriculum learning strategy to train agents for robot manipulation. This strategy is tested with an off-policy deep reinforcement learning algorithm called deep Q-learning which iteratively approximates the optimal action-value function using deep neural networks. We found that a simple training strategy consisting in increasing the number of robot joints that the agent can control gives better results, and other encouraging results using other curricula which could be improved by using more prior knowledge about the problem to design the curricula more carefully.

Acknowledgements

I would like to thank my supervisor Edward Johns, for letting me delve deeper into such an interesting subject, for his valuable advice, guidance, and genuine interest in making the most of this project. I would also like to thank my robotics and machine learning professors, whose passion for their work made me excited about working at the crossroads of these disciplines.

I would also like to thank my flatmates, my friends and my family for their affective support throughout this year in London.

Contents

1 Introduction **1**

2 Background **3**

2.1 Reinforcement Learning 3

2.1.1 Markov Decision Process 4

2.1.2 Return 4

2.1.3 Value functions 5

2.1.4 Optimal Policy 5

2.1.5 Q-Learning 5

2.2 Deep Reinforcement Learning 7

2.2.1 Action-value function approximation 7

2.2.2 Deep Q-Networks 7

2.2.3 End-to-end models 9

2.3 Curriculum Learning 9

2.4 Related Work 10

2.4.1 Simulated robot manipulation using deep reinforcement learning . . 10

2.4.2 Curriculum learning for deep reinforcement learning 11

2.4.3 Automated curriculum generation 11

3 Experimental Setup **13**

3.1 Robotics Environment 13
3.2 Tasks 14
3.3 Algorithms implemented 14
3.3.1 Network architecture 16
3.3.2 Reward function 17
3.3.3 Number of steps per episode 18
3.3.4 Asynchronous data collection and network updates 18
3.3.5 Implemented algorithm summary 18
3.4 Curricula evaluated 20
3.5 Strategy comparison 20

4 Experimental Results and Evaluation 23

4.1 Object reaching task 24
4.1.1 *Decreasing joint angular velocity* curriculum 24
4.1.2 *Initializing robot position closer to the target* curriculum 29
4.1.3 *Increasing number of moving joints* curriculum 34
4.2 Object pushing task 39
4.2.1 *Decreasing joint angular velocity* curriculum 39
4.2.2 *Initializing robot position closer to target* curriculum 43
4.2.3 *Increasing number of moving joints* curriculum 46

5 Conclusion and Future Work 49

A Code running instructions 53

List of Figures

1.1 Curriculum learning consisting in initializing the robot end-effector position close to the target object, then initializing further away from it. The last task is the goal task to learn. 2

2.1 Reinforcement learning is suitable for problems involving interactions between an agent and an environment. 3

2.2 Simple DQN. Each neuron in the output layer corresponds to the Q value of the input vector s and an action a. Thus, we get the Q value of all possible actions given the state at each forward pass. 8

3.1 Simulated environment on V-REP for robot manipulation using a Mico arm . 14

3.2 Robot executing a policy for the reaching task. 15

3.3 Robot arm executing a policy for the pushing task. 15

3.4 DQN architecture used for the robot manipulation problem. Neurons in the output layer are split in sub-sets of 3, where each one is interpreted as the Q value of one of the three joint actions: set joint angle velocity to the fixed positive value, to the fixed negative value, or set it to 0. Thus, robot actions are composed of six joint actions that are updated at each time-step. 16

4.1 **Decreasing joint angle velocity** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 25

4.2 **Decreasing joint angle velocity** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards. 26

4.3 **Decreasing joint angle velocity** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 27

4.4 **Decreasing joint angle velocity** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards. 28

4.5 **Further initial states** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 30

4.6 **Further initial states** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards. 31

4.7 **Further initial states** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 32

4.8 **Further initial states** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards. 33

4.9 **Increasing number of joints** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 35

4.10 **Increasing number of joints** curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards. 36

4.11 **Increasing number of joints** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 37

4.12 **Increasing number of joints** curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards. 38

4.13 **Decreasing joint angle velocity** curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 40

4.14 **Decreasing joint angle velocity** curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards. 41

4.15 **Decreasing joint angle velocity** curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 42

4.16 **Initial states further away from object** curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 44

4.17 **Initial states further away from object** curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 45

4.18 **Increasing number of joints** curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 47

4.19 **Increasing number of joints** curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards. 48

Chapter 1

Introduction

In this project, we evaluate the performance of curriculum learning for robotic manipulation tasks using deep reinforcement learning algorithms.

So far, a large number of robots have been successfully deployed in industry to perform manipulation tasks. These robots are designed perform very specific tasks in controlled environments. Robots could be very useful outside factories, such as at home, at the office, or in the street, to perform tasks such as cleaning, handling and tidying up. Unfortunately, the variability of the environments, objects and obstacles that the robot may encounter make these tasks too complex and are out of the reach of current robot capabilities.

For robots to be able to perform useful tasks in environments that were not designed specifically for robots, they have to able to adapt successfully to new environments and objects. Moreover, they have to be safe for people around them. In controlled environments, we can use the prior knowledge about the environment and the objects that the robot will encounter to reduce the complexity and build hand-engineered modules, such as sensors, state estimators and low-level control systems that solve the task. This approach struggles to deal with the wide range of environments and objects encountered in everyday life. More general approaches with fewer assumptions are needed to deal with this problem.

Reinforcement learning algorithms have a more general approach to solve problems, by making the intelligent agent learn on its own what the optimal strategy to solve the task is, by learning from its interactions with its environment. This approach was successfully applied to simple problems and different games. Given that robotics problems often involve high-dimensional, continuous action and state spaces, only simplified robotics problems have been solved using this approach, such as using hand-engineered features to describe policies [].

More recently, deep neural networks have been successfully used in combination with reinforcement learning to solve complex problems. Indeed, neural networks have been used as function approximators for different reinforcement learning algorithms, building

on their ability to approximate complex functions and deal with high dimensional input spaces, which have lead to state-of-the-art models for supervised and unsupervised learning problems as computer vision and natural language processing.

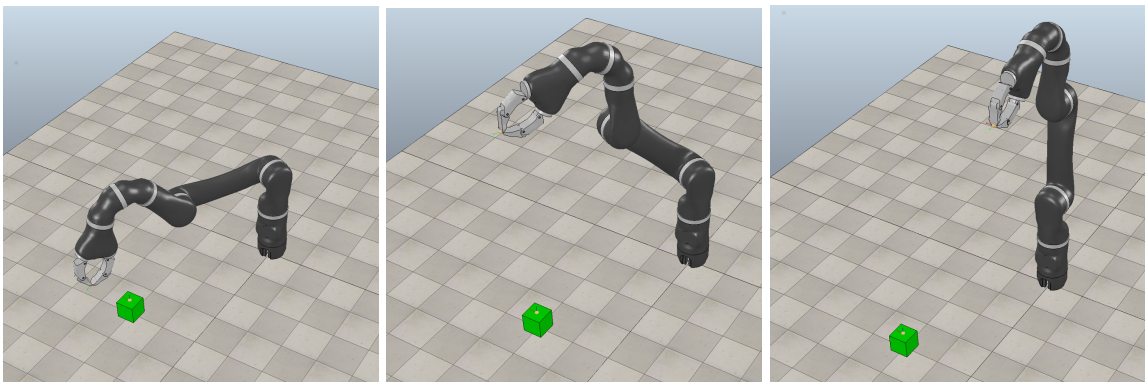
Deep reinforcement learning has given state-of-the-art results at game AIs that can surpass human performance, at games such as the game of Go [Silver, 2015] and Atari games [Mnih et al., 2015]. It has also been applied to robotics, to build end-to-end robot controllers [Levine et al., 2015].

Nevertheless, deep reinforcement learning depends heavily on the amount of data and the computing resources available. Unlike supervised or unsupervised learning settings, data is generated at training time, from the agent interactions with the environment. Therefore, optimizing data collection and exploitation are necessary to scale up deep reinforcement learning to more complex and useful robotics tasks.

In this project we evaluate the effectiveness of using a curriculum learning approach to improve data exploitation and reduce the amount of data needed for robot manipulation tasks. Curriculum Learning [Bengio et al., 2009] is a training strategy that consists in exposing the learning agent to a sequence of tasks of increasing difficulty before exposing it to the final task. It is inspired by the way children are exposed to concepts of increasing difficulty as they grow up, so that they can eventually learn complex concepts more easily.

We explore different curricula, such as increasing the number of moving joints, increasing speeds, and using easier initial robot configurations, for manipulation tasks such as object reaching and object pushing. We will implement a deep Q-learning algorithm on a 6 DoF robotic arm in a simulated environment to evaluate this approach.

We obtained an agent with better performance when trained with a simple curriculum consisting in increasing the number of joints that it can control, and encouraging results for other curricula. We also found that prior knowledge about the problem given to the training algorithm through the curriculum has to be accurate, since the curriculum may prevent learning an optimal policy otherwise.



(a) Sub-task 1 **(b)** Sub-task 2 **(c)** Sub-task 3

Figure 1.1: Curriculum learning consisting in initializing the robot end-effector position close to the target object, then initializing further away from it. The last task is the goal task to learn.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning is a machine learning approach to solve problems involving interactions between an intelligent agent and an environment. The agent is capable of executing actions that affect its environment and getting back observations of its state. During these interactions, the agent and the environment states are constantly changing, and according to the task we want the agent to complete, rewards are given to the agent when a goal state has been reached, or when the agent is getting closer to such a state, see Figure 2.1.

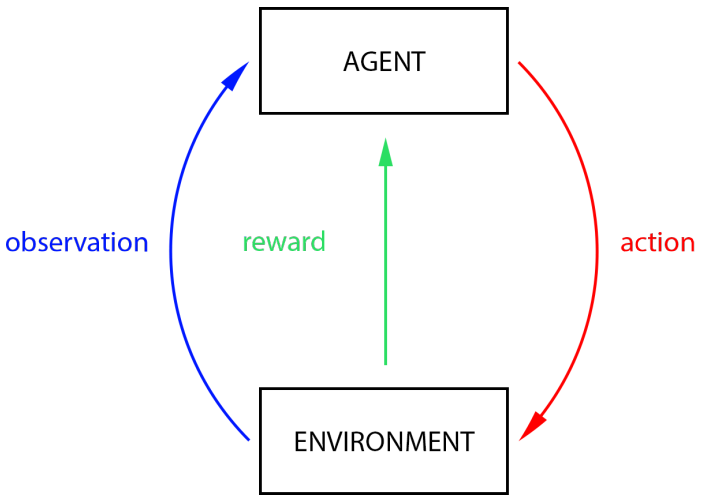


Figure 2.1: Reinforcement learning is suitable for problems involving interactions between an agent and an environment.

The learning agent will then try to learn sequences of actions that maximize the cumulative rewards by learning from its interactions with the environment, using a trial and error approach [Sutton and Barto, 1998].

2.1.1 Markov Decision Process

More formally, we can discretise the interactions between the agent and the environment into transitions. At time-step t, the agent in state s_t executes the action a_t that affects its environment, receives a reward r_t and observes a new state s_{t+1}.

Reinforcement learning problems with fully observable states are Markov Decision Processes $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{P}(s_{t+1}|s_t, a_t)$ is the probability of ending in state s_{t+1} after picking action a_t from the state s_t, $\mathcal{R}_{a_t}(s_t, s_{t+1})$ is the function defining the reward received during the transition from state s_t to state s_{t+1} by executing action a_t, and γ is the discount factor.

Thus, reinforcement learning problems have the Markov property, which says that the next state is conditionally independent of past states given the current state:

$$\mathbb{P}(s_{t+1}|s_1, ..., s_t) = \mathbb{P}(s_{t+1}|s_t) \tag{2.1}$$

2.1.2 Return

The sequence of actions executed by the agent is determined by the strategy it follows, which is formalised as a *policy* π. A policy can be defined as a mapping from the set of states $\mathcal{S}$ to the set of actions $\mathcal{A}$ for deterministic environments or as a conditional probability distribution over actions given the current state for stochastic environments.

$$a = \pi(s) \tag{2.2}$$

$$\text{or } \pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \tag{2.3}$$

We can define the *return* as the sum of rewards received by the agent after an episode of interactions, consisting of T time-steps:

$$R_{undisc} = \sum_{t=1}^{T} r_t \tag{2.4}$$

We can also define a *discount factor* γ and a *discounted return* that gives more weight to immediate rewards:

$$R = \sum_{t=0}^{T-1} \gamma^t r_{t+1},\ 0 \leq \gamma \leq 1 \tag{2.5}$$

The target of the training algorithm will then be to find a policy that maximizes the (discounted) return. Thus, a low γ gives more value to short-term rewards, whereas a high γ encourages the agent to maximize its return in the long-term. To do this, the agent has to explore the state and action spaces to observe the rewards associated to them and figure out which sequences of states and actions are associated to high returns.

2.1.3 Value functions

To compare different policies, we define some value functions. The *state-value function* $V_\pi(s)$ as the expected return when the agent is in state s and follows the policy π afterwards:

$$V_\pi(s) = \mathbb{E}_\pi[R|s] \tag{2.6}$$

Similarly, we define the *action value function* $Q_\pi(s, a)$ as the expected return when the agent is in state s, picks action a and follows the policy π afterwards:

$$Q_\pi(s, a) = \mathbb{E}_\pi[R|s, a] \tag{2.7}$$

2.1.4 Optimal Policy

Ideally, the agent will learn the optimal policy π^*, which is the one that maximizes the expected return over whole episodes.

$$\pi^*(s) = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_\pi[R] \tag{2.8}$$

Its associated value functions are called the optimal value functions $V_{\pi^*}(s)$ and $Q_{\pi^*}(s, a)$. By defining a partial ordering over policies according to their associated state-value functions:

$$\pi \leq \pi' \Leftrightarrow \forall s\ V_\pi(s) \leq V_{\pi'}(s) \tag{2.9}$$

we can find the optimal state-value function and the optimal action-value function:

$$V^*(s) = \max_\pi V_\pi(s) = V_{\pi^*}(s) \tag{2.10}$$

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) = Q_{\pi^*}(s, a) \tag{2.11}$$

The best policy π associated to a value function is obtained by acting greedily: picking the actions leading to the accessible state s_{t+1} with the highest state value $V(s_{t+1})$, and the action that maximizes $Q_\pi(s, a)$, $a = \operatorname{argmax}_a Q_\pi(s, a)$. Thus, if we find an optimal value function, $V^*(s)$ or $Q^*(s, a)$, we can find the optimal policy π^*.

2.1.5 Q-Learning

Value functions can be rewritten as the sum of immediate reward and the discounted value at the next time-step, called the Bellman equation [Bellman, 1952]. For example, for the action-value function:

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t, a_t] \tag{2.12}$$

Based on this equation, we can iteratively approximate the value functions with a dynamic programming approach. At each iteration, the previous estimation of the value function is slowly updated towards a target value that incorporates the observed immediate rewards. This method, called temporal difference (TD) learning uses the following updating rule:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\delta \tag{2.13}$$

where α is the *learning rate*, which controls how each observed reward impacts the current estimate at each update, and δ is called the *temporal difference (TD) error*, corresponding to the difference between the initial estimate and the target value:

$$\delta = Q_{target} - Q_\pi(s_t, a_t) \tag{2.14}$$

According to the Bellman equation, a good target value is:

$$Q_{target} = r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) \tag{2.15}$$

and the resulting learning algorithm is called *SARSA* (from **S**tate-**A**ction-**R**eward-**S**tate-**A**ction, which are the transition elements needed for the updates). SARSA is an *on-policy* algorithm as it learns from transitions generated by the policy associated to the current action-value estimate Q_π to interact with the environment.

The optimal value function Q^* can also be defined recursively, given that the next action is obtained by greedily maximizing the action-value function:

$$Q^*(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t] \tag{2.16}$$

the associated algorithm, called *Q-learning*, approximates the optimal action-value function Q^*:

$$Q_{target} = r_t + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1}) \tag{2.17}$$

Q-learning is an *off-policy* learning algorithm, as it can learn from transitions (previous state, action, reward, next state) that may have been generated by policies different from the current one.

Q-learning summary

To sum up, Q-learning uses the following rule to iteratively approximate the optimal action-value function Q^*:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha(r_t + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t)) \tag{2.18}$$

using transitions $e_t = (s_t, a_t, r_t, s_{t+1})$.

2.2 Deep Reinforcement Learning

2.2.1 Action-value function approximation

Reinforcement learning problems that have a small number of states, such as chess or grid-world problems, might be solved using a 2-entry table to store the action-values $Q(s, a)$ and updating them using Q-learning. However, for more complex problems with large finite state spaces, such as the game of Go, or continuous state space, such as robotics problems, this solution does not scale. Instead, we can use an estimator $Q_\theta(s, a)$ of the action-value function $Q_\pi(s, a)$ defined by a set of parameters θ that requires less storage capacity:

$$Q_\theta(s, a) \approx Q_\pi(s, a) \tag{2.19}$$

The goal is then to update θ to minimize the Q-learning TD error δ:

$$\delta = r_t + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t) \tag{2.20}$$

To do this, we can minimize the mean squared error cost function:

$$L(\theta) = \mathbb{E}[(r_t + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))^2] \tag{2.21}$$

Then we can update the model parameters using gradient descent:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial L}{\partial \theta_i}(\theta) \tag{2.22}$$

$$\frac{\partial L}{\partial \theta_i}(\theta) = \mathbb{E}\left[(r_t + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)) \frac{\partial Q_\theta(s_t, a_t)}{\partial \theta_i}\right] \tag{2.23}$$

2.2.2 Deep Q-Networks

Deep learning models are known to be flexible enough to approximate complex functions with high-dimensional input spaces with a few number of parameters, by stacking layers of linear and non-linear transformations. Neural networks have recently given state-of-the-art models for regression and classification, in problems such as object recognition and machine translation. Deep neural networks have also been applied to approximate action-value functions with very encouraging results: Deep Q-Networks (DQN) were successful at training agents to play some Atari games with a human-level performance [Mnih et al., 2015] [Mnih et al., 2013]. However, some improvements over the basic, naive Q-learning algorithm where necessary to achieve these results: experience replay and target networks.

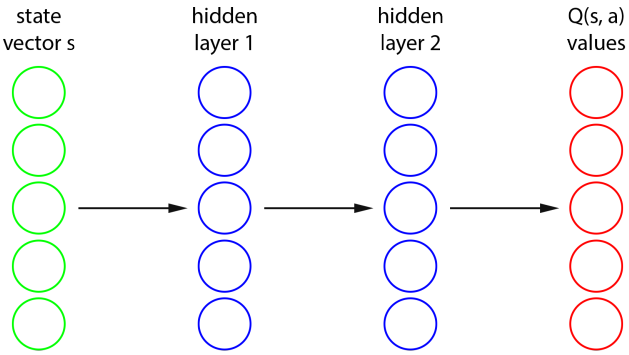


Figure 2.2: Simple DQN. Each neuron in the output layer corresponds to the Q value of the input vector s and an action a. Thus, we get the Q value of all possible actions given the state at each forward pass.

Experience replay

The data collected by the algorithm is a sequence of temporally correlated transitions. In order to break the correlation, we can use a approach called *experience replay* [Lin, 1992], which consists in storing the observed transitions $e_t = (s_t, a_t, r_t, s_{t+1})$ in a buffer and randomly sampling a mini-batch of transitions for each network parameters update.

Another benefit is that data is re-used: interesting transitions are used for multiple network updates. Indeed, an efficient data exploitation greatly reduces training times for reinforcement learning algorithms, since data collection through interactions with the environment often takes an important part of overall training time and resources, sometimes more than the actual parameter updates. This is contrary to supervised and unsupervised learning problems, which are often limited to training model parameters using a (large) dataset already available.

The resulting objective function is the mean squared error over the mini-batch sampled at each training step:

$$L(\theta) = \mathbb{E}_{mini-batch}[(r_t + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))^2] \tag{2.24}$$

which is thus minimized using stochastic gradient descent, similarly to other machine learning setups.

Exploration-exploitation trade-off

For the action a_t to execute next, it may seem natural to pick the action that maximizes $Q(s_t, a_t)$ i.e. act greedily, but it is also important to explore less visited states that may eventually lead to higher returns. There is a exploration-exploitation trade-off which is can be dealt with an ϵ-greedy exploration strategy, which consists in picking a random action with an ϵ probability, and acting greedily with an $1 - \epsilon$ probability. ϵ is typically

close to 1 at the beginning to encourage exploration and decreases to a very small value to encourage convergence of the learned policy towards the optimal one.

Target network

Small changes to the Q-values may have a big impact on the associated policy. To avoid policy oscillations during deep Q-learning updates and encourage convergence, we can use a secondary neural network with old parameters to retrieve the target values Q_{target}. The objective function to train the main network becomes then:

$$L(\theta) = \mathbb{E}_{mini-batch}\left[(r_t + \gamma \max_{a_{t+1}} Q_{\theta_{target}}(s_{t+1}, a_{t+1}) - Q_{\theta_{main}}(s_t, a_t))^2\right] \tag{2.25}$$

and we update the *target network* by periodically copying the parameters from the main network:

$$\theta_{target} \leftarrow \theta_{main} \tag{2.26}$$

or smoothly updating the target network parameters towards the main network at each step:

$$\theta_{target} \leftarrow (1 - \tau)\,\theta_{target} + \tau\,\theta_{main} \tag{2.27}$$

where τ is the smooth update rate, typically close to 1, e.g. $\tau = 0.99$.

2.2.3 End-to-end models

Deep neural networks can be used to approximate value functions, but also policies and even build a model of the environment. Building on the accuracy and flexibility of convolutional neural networks to analyse images and recurrent neural networks to extract temporal features in sequential data, neural networks are well-suited for optimizing these functions in an end-to-end fashion, i.e. going from the raw sensor signals that reflect the current state to the low-level actions to execute. For example, in [Mnih et al., 2015], the input of the DQN is the raw pixels and the outputs are the Q-values of the actions associated to the Atari joystick and button positions. An end-to-end model is also proposed for optimal control for robotics in [Levine et al., 2015], where a deep neural network approximating policies takes raw images as input and maps them to motor torques.

2.3 Curriculum Learning

Curriculum learning [Bengio et al., 2009] is a training strategy for machine learning algorithms, consisting in training the model with a sequence of examples of increasing difficulty, instead of uniformly sampling examples regardless of their difficulty, or directly training the model with the difficult examples. This strategy was inspired by human learning

during childhood which follows a curriculum defined by parents and education systems, that expose children to simple concepts first and gradually move towards more complex concepts, in order to make them easier. Shaping uses a similar strategy to train animals.

The idea of *starting small* was introduced in [Elman, 1993] to help neural networks learn complex concepts by being exposed to a sequence of concepts of increasing difficulty. This approach, inspired by human learning during childhood, was applied successfully to learning a simple grammar with a recurrent neural network.

This training strategy was further developed in [Bengio et al., 2009]. Experimental results for tasks such as shape recognition and language modelling showed that using a curriculum, i.e. a sequence of sets of examples of increasing difficulty, can speed up learning and improve generalization (better performance on the test set of target hard examples). This two-fold benefits are also observed when using unsupervised pre-training methods like greedy layer-wise pre-training of networks for supervised tasks [Hinton et al., 2006] [Vincent et al., 2008], which takes another approach to learn more complex concepts. Indeed, unsupervised pre-training can be interpreted as learning abstract representations of the input data with an increasing level of abstraction in each new layer. For example, in an object recognition task, the first layer may correspond to edge detectors and the following layers contain combinations of these features of increasing complexity, such as object parts and eventually the objects themselves. Formally, unsupervised pre-training methods put deep network in the basins of attraction of a good local minima in parameter space in terms of test set error, thus accelerating convergence during supervised training and acting as a regularizer [Erhan et al., 2009]. Curriculum learning might have a similar optimisation effect.

Even though the benefits of curriculum learning have been observed for many tasks, the curricula that were tested in experimental settings were hand-crafted and contained prior knowledge specific to the target task, and the schedule used was defined in terms of epochs or network updates, instead of being adapted to the performance of the network at each task.

2.4 Related Work

2.4.1 Simulated robot manipulation using deep reinforcement learning

This projects is a follow-up to a previous project that on the deep reinforcement learning for robot manipulation in a simulation setting [James and Johns, 2016a] [James and Johns, 2016b]. It focused on the implementation of an end-to-end deep Q-learning model to control the Mico robot arm in simulation, that takes raw images generated in simulation as input vector. In contrast, we directly used the joint angles, end-effector position and object position to simplify the problem for curriculum learning evaluation.

2.4.2 Curriculum learning for deep reinforcement learning

[Wu and Tian, 2016] used curriculum Learning with shaping rewards to train an Asynchronous Advantage Actor-Critic (A3C) to play the First person shooter (FPS) video-game Doom and achieved a very good performance. The curricula used consisted in increasing the complexity of maps, and making the bots stronger by increasing their moving speed, their initial health and their initial weapon. The difficulty level is increased or decreased depending on whether the agent has a good or a bad performance with the current difficulty, respectively.

2.4.3 Automated curriculum generation

Some methods have been proposed recently to automate the process of curriculum generation. [Held et al., 2017] introduce a method to iteratively generate goals for reinforcement learning agents to perform, using Generative Adversarial Networks (GAN) [Goodfellow et al., 2014]. The objective is to maximize the reward A GAN is trained along with the RL agent to generate goals with the right difficulty according to the agent's performance, so that it keeps learning by gradually completing challenging tasks. This method of generating goals with an appropriate schedule optimises the training time and has given state-of-the-art results at some reinforcement learning problems. Another benefit of this approach is that it is suitable for problems with sparse rewards.

Another approach was proposed in [Sukhbaatar et al., 2017] for reinforcement learning tasks, which consists in using two sub-agents A and B: A proposes a task to B, which tries to complete it. This is first done with a unsupervised approach, by ignoring the rewards of the actual problem. Instead, the environment is explored by rewarding A when it proposes a task difficult for B but relatively easy for A, and rewarding B for optimally completing the task proposed. By doing this, A will actually propose suitable tasks of increasing difficulty according to B's performance and B will get gradually better at executing difficult tasks in its environment. Both A and B can be implemented using neural networks. After the self-play episodes, B is trained with the problem reward function and experimental results show that it benefits from the unsupervised pre-training as the agent gets higher rewards in fewer episodes.

[Florensa et al., 2017] proposes another approach to generate a curriculum automatically by sampling initial states close to goal states, and previous initial states that have lead to goal states. By doing this, the agent is able to learn from the episodes starting close to goal states and progressively starting further away from the goal states and sampling states according to the agent performance. Similarly, a curriculum consisting in initializing the robot end-effector close to goal states and progressively initializing it further away from the goal state was used in [Popov et al., 2017]. For this project, we used some of these ideas, as we will describe later.

Chapter 3

Experimental Setup

In this part we will describe the experimental setup used to evaluate and compare different curriculum learning strategies, consisting of a deep Q-learning algorithm coupled to a robotics simulator.

3.1 Robotics Environment

Experiments were run in simulation using the V-REP (Virtual Robot Experimentation Platform) simulator [Coppelia Robotics, 2017]. A model of the Mico 6 DoF robot arm with its gripper was retained for all experiments, as shown in Figure 3.1, since the Dyson Robotics Lab team is working with Mico robotic arms, and could be used to deploy a policy learned in simulation.

The robot will be controlled by setting joint angular velocities. To do this, the control loop is disabled and the maximum torque is set to a high value (10^{10} N.m), so that the target velocity will be reached instantaneously.

We used the V-REP remote API to control the simulations from an external Python script via socket communication, which implements some basic environment functions similar to OpenAI Gym environments [Brockman et al., 2016], such as executing actions, getting observations and rewards, and resetting the environment.

V-REP was used in synchronous mode, meaning that each simulation time-step is triggered from the Python script. We used the default time-step of 50 ms. This means that the agent will execute an action and receive a new observation at 20 Hz in simulation time.

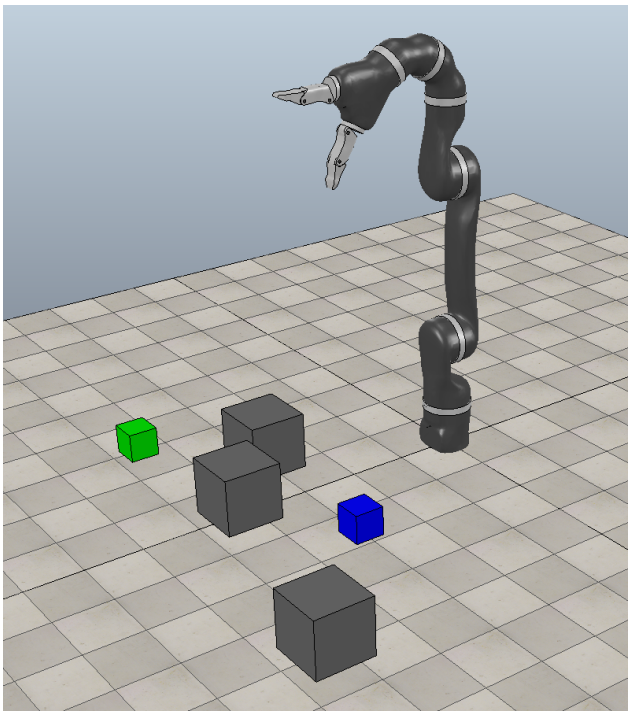


Figure 3.1: Simulated environment on V-REP for robot manipulation using a Mico arm

3.2 Tasks

We chose two robot manipulation tasks:

1. reaching a target object, such as a small cube, as shown in 3.2. The position of the object does not change between episodes, but the object will dynamically react to the forces applied to it, e.g. the cube may be pushed by the robot arm trying to reach it.

2. pushing an object to a pre-defined target position, as shown in 3.3. The initial position of the object and the target position do not change between episodes. This task is more complex than the previous one, since the robot has to reach the object first.

These tasks were chosen to evaluate curriculum learning strategies for their simplicity, so that we can do some comparisons with reasonable training times.

3.3 Algorithms implemented

We will run our experiments using Deep Q-Networks to control the robot arm, implemented in Python (3.5) using the Tensorflow library (1.2).

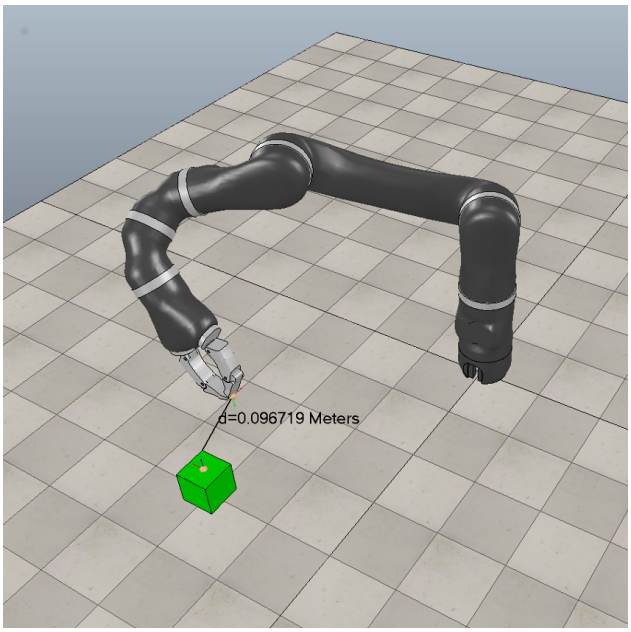


Figure 3.2: Robot executing a policy for the reaching task.

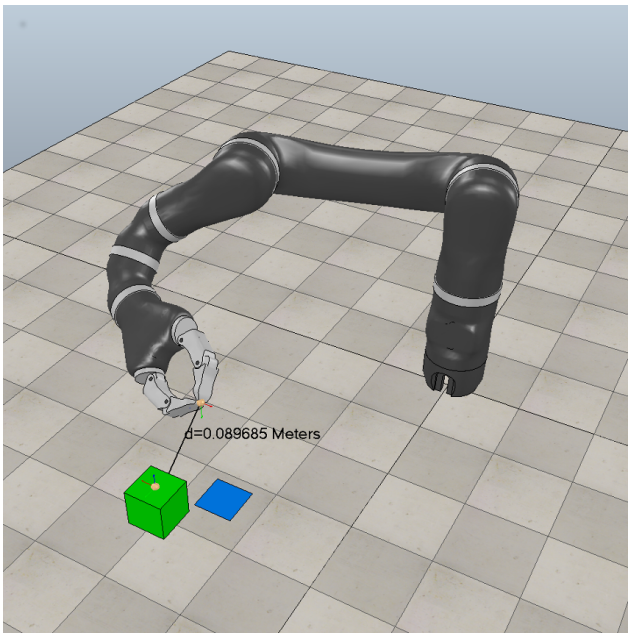


Figure 3.3: Robot arm executing a policy for the pushing task.

Following the recent success of Deep Q-Networks, most notably for Atari game AIs [Mnih et al., 2015], we decided to implement a similar algorithm to evaluate curriculum learning.

3.3.1 Network architecture

The retained DQN architecture was a deep network that takes the environment state as input and outputs the action-values (Q values) associated to each action.

To do this, we used a fully connected neural network with three hidden layers between the input layer and the output layer. The input layer is the state vector, consisting of the normalized angles of each joint, the 3D position of the robot end-effector (x, y, z values in meters) and the 2D position of the object to interact with (x, y values in meters, corresponding to the cube location in the $z = 0$ plane). The hidden layers are composed of 50 rectified linear units (ReLU) each. The final layer is split into 6 subsets (the number of joints that the robot can move) of 3 neurons each. Each subset corresponds to three actions available to each neuron: turning the joint clockwise at the fixed speed, turning it anticlockwise at the fixed speed, or staying still (zero velocity). The joint velocity value is always fixed before training, and typically equal to 1 rad/s. At each time step, the velocity of each joint is simultaneously and independently updated according to the maximum Q-value in its associated output subset, as shown in Figure 3.4.

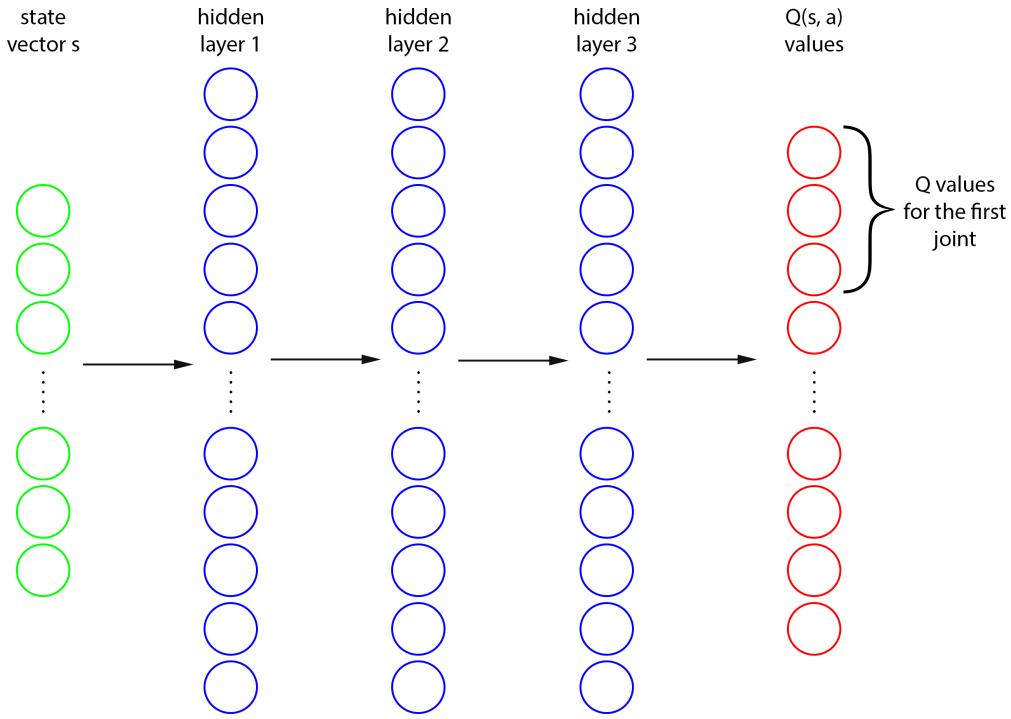


Figure 3.4: DQN architecture used for the robot manipulation problem. Neurons in the output layer are split in sub-sets of 3, where each one is interpreted as the Q value of one of the three joint actions: set joint angle velocity to the fixed positive value, to the fixed negative value, or set it to 0. Thus, robot actions are composed of six joint actions that are updated at each time-step.

We use an Adam optimizer to train the network, with a learning rate of 10^{-4}. We implemented an experience replay buffer with a size of the order of magnitudes of 10^4 transitions, which often means that training episodes are used for all following episodes.

We used an ϵ-greedy epsilon strategy to fill the buffer, with ϵ set to zero at the beginning to explore the state space by executing completely random actions for a few thousand transitions. ϵ is then decayed exponentially to a minimum final value, typically $\epsilon_{min} = 0.01$ for each joint:

$$\epsilon(n) = \epsilon_{min} + (1 - \epsilon_{min})\, e^{-\frac{n}{N_\epsilon}} \tag{3.1}$$

where is N_ϵ is a constant. We set this constant so that $\epsilon \approx \epsilon_{min}$ by the end of the episode. The iterative approach consists in decreasing the value of ϵ by a fixed amount:

$$\epsilon_0 = 1 \tag{3.2}$$

$$\epsilon_{n+1} = \begin{cases} \epsilon_n, & \text{if } n < N_{start} \\ (1 - \frac{1}{N_\epsilon})\, \epsilon_n, & \text{otherwise} \end{cases} \tag{3.3}$$

3.3.2 Reward function

For robotics problems, it is often the case that it is easy to tell whether a state is a goal state, but it is not clear whether it belongs to an optimal trajectory leading to a goal state. Reinforcement learning is interesting in such situations, since we can define a *sparse* reward function using little prior knowledge, i.e. the reward signal to 1 for goal states and 0 for all other states. Unfortunately, for robotics problems, state spaces are continuous and usually high dimensional, thus it may take a very long time before the robot gets any reward, since the probability that a random sequence of actions will lead to a goal state is very low. To deal with this problem, we can provide the algorithm with non-zero rewards for states belonging to a path leading to the goal state. These *shaping* rewards encourage the exploration of states *closer* to a goal state. By giving more prior knowledge to the agent through the reward signal, we reduce the complexity of the problem. Indee, some robotics tasks in constrained environments have been solved using this strategy. Nevertheless, using shaping rewards requires careful design, as they introduce a bias in training that may lead to sub-optimal strategies or even prevent the agent from reaching a goal state, as described in [Popov et al., 2017].

We chose to use shaping rewards based on the distance between the end-effector and the surface of the object. The reward reaches a maximum value r_{max} when touching the object and decays exponentially as the end-effector moves away from the object:

$$r_{1,t} = r_{1,max}\, e^{-\frac{d_{a,t}}{D}} \tag{3.4}$$

For the object reaching task, it is straightforward to directly use this reward function by using the distance $d_{a,t}$ between the end-effector and the object. For the object pushing to target position task, we used a reward function with two components, corresponding to the distance $d_{a,t}$ between the end-effector and the object and the distance $d_{b,t}$ between the object position and the target position, similar to the approach taken in [Gu et al., 2016]:

$$r_{2,t} = r_{a,t} + r_{b,t} \tag{3.5}$$

$$r_{a,t} = r_{a,max}\, e^{-\frac{d_{a,t}}{D}} \tag{3.6}$$

$$r_{b,t} = r_{b,max}\, e^{-\frac{d_{b,t}}{D}} \tag{3.7}$$

Furthermore, we used a small maximum reward r_{max} in order to keep the discounted return within the range $[1, +1]$ to prevent the Q-values from being too large and avoid instability during backpropagation due to large gradients.

3.3.3 Number of steps per episode

When using shaping rewards, the number of steps that the agent is allowed to execute during each episode has a great impact on the undiscounted rewards and the Q-values. Indeed, if we use only positive rewards and *terminal* goal states, i.e. episodes end when a goal state is reached, the agent may maximize its returns by simply avoiding terminal states until the maximum number of steps per episode is reached. One way to deal with this problem and force the agent to find the optimal trajectory is to set negative rewards (for non-terminal states), as the agent will try to reach a terminal state in the smallest number of steps. Since we are using ReLU activation functions, it is easier to approximate positive Q-values, thus positive rewards are preferred. Unfortunately, with positive rewards, all episodes have to last the same number of steps to avoid the problem presented above, and the agent may stay in a goal state for many steps if the number of steps allocated to each episode is greater that the minimal number of steps needed to reach the goal. Thus, it is better to avoid excessive overestimation of the number of steps needed in order to use training time for policy optimization through exploration.

3.3.4 Asynchronous data collection and network updates

Following the results on asynchronous data collection and network updates presented in [Gu et al., 2016], we decided separate these into two different threads: a collector thread interacts with the environment and stores transitions into the experience replay buffer, and a trainer thread samples transitions from the buffer to train the DQN. This approach is particularly useful for real environments, for which the robot has to interact in real time with its environment, since long gradient backpropagation times are only executed in the trainer thread. Nevertheless, this approach may greatly reduce overall training time. We also implemented multiple network parameter updates per environment step to further reduce training time, as described in [Popov et al., 2017].

3.3.5 Implemented algorithm summary

The final algorithm is shown in Algorithm 1.

Algorithm 1 DQN training algorithm implemented

// main thread (collector thread)
  Initialize main network parameters θ_{main}
  Copy main to target parameters $\theta_{target} \leftarrow \theta_{main}$
  Initialize experience replay buffer B
  Initialize exploration rate $\epsilon \leftarrow 1$
  Initialize $transitions \leftarrow 0$
  Initialize $episode \leftarrow 0$
  while $transitions < N$ **do**
    $episode \leftarrow episode + 1$
    Update collector network with main network parameters $\theta_{collector} \leftarrow \theta_{main}$
    Get initial state s_1
    if $transitions <$ buffer start size **then**
      $\epsilon \leftarrow (1 - \frac{1}{N_\epsilon})\,\epsilon$
    end if
    for $t = 1, T$ **do**
      Get Q-values $Q_{\theta_{collector}}$
      for $j = 1, J$ **do**
        Sample $x \sim unif(0, 1)$
        if $x < \epsilon$ **then**
          Pick a joint action greedily $a_{t,j} = \text{argmax}_{a_j}\, Q_{\theta_{collector}}(s_t, a_j)$
        else
          Pick a random joint action $a_{t,j}$
        end if
      end for
      Execute robot arm action a_t and get the new state s_{t+1} and reward r_t
      Store the transition (s_t, a_t, r_t, s_{t+1}) in the buffer B
      $transitions \leftarrow transitions + 1$
    end for
  end while
// trainer thread
  Load main and target network parameters θ_{main}
  Wait for the experience replay buffer B to reach its start size
  while $transitions < N$ **do**
    $last\ transition \leftarrow transitions$
    $update \leftarrow 0$
    while $last\ transition = transitions$ **do**
      if $update < N_u$ **then**
        Sample a mini-batch of n transitions from B
        Get target Q values from target network:
        $Q_{target,i} \leftarrow r_t + \gamma \max_{a_{t+1,i}} Q_{\theta_{target}}(s_{t+1,i}, a_{t+1,i})$
        Update the main network parameters θ_{main} to minimize
        $L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(Q_{target,i} - Q_{\theta_{main}}(s_{t,i}, a_{t,i}))^2$
        Smoothly update the target network parameters $\theta_{target} \leftarrow \tau\theta_{main} + (1-\tau)\theta_{target}$
        $update \leftarrow update + 1$
      end if
    end while
  end while

3.4 Curricula evaluated

For robotics problems, there are many axes of difficulty, such as the number of robot joints, the velocities applied and the initial robot configuration. Each curriculum is composed of *sub-tasks*, going from the easiest task to the hardest task, corresponding to the target task.

We decided to test three different curricula:

- reducing joint angular velocities. We first set the non-zero joint angular velocities to 4 rad/s, and subsequently to 2 rad/s then 1 rad/s. This last value was retained for training experiments without curriculum.
- increasing the number of joints. We progressively increase the number of moving joints, from 1 to 6, from the base joint up to the end-effector joint.
- initializing the robot in a position close to the target object and progressively setting the initial position further away from it.

3.5 Strategy comparison

In order to evaluate curriculum learning, we decided to track some variables during training as well as to include test episodes periodically between training episodes. These include the number of successful episodes during training, the undiscounted returns received during training, the step at which a goal state was first reached (when a goal state is not reached, the step number is the maximum number of steps per episode), and the evolution of the exploration rate ϵ. Goal states correspond to a distance smaller than 5cm between the end-effector and the object for the reaching task, and to a distance smaller than 5cm between the object position and the target position for the object pushing task.

Since the main objective of using curriculum learning is to reduce training time, we decided to set a fixed number of environment transitions that the algorithm would see during training, regardless of the curriculum followed. For each curriculum, the number of transitions was divided and allocated equally to each sub-task. For example, for a curriculum with three different joint angle velocities, each velocity would be used for about one third of the total number of transitions, rounded to the upper bounding number of steps corresponding to an entire training episode. This way, we can fairly compare curricula with different numbers of transitions per episode. All curves are also compared with respect to the number of transitions the algorithm has seen during training.

We also set the sum of the sub-task buffer initial start sizes to be equal to the buffer start size of the baseline without curriculum, in order to obtain comparable final agents that roughly execute the same overall number of random actions.

The learned policy is periodically evaluated during training at *testing* episodes, typically for 20 testing episodes every 50 training episodes. During test episodes, training is momentarily paused and the policy is evaluated in target task conditions, and we keep track of the success rate for testing episodes as well as the mean undiscounted reward received during each batch of testing episodes.

Chapter 4

Experimental Results and Evaluation

In this chapter, we describe the results obtained under the experimental conditions described previously and evaluate them.

We will present results corresponding to the target object reaching task first, then we will present the results corresponding to the object pushing to target position task. For each task, we will go through each curriculum, and evaluate their performance. We will compare results to baselines not using curriculum learning. Both shaping and sparse rewards were used in order to evaluate curriculum learning for problems with sparse rewards, and compare them to using shaping rewards. We also compared curriculum learning with shaping rewards with a baseline without curriculum and with sparse rewards, in order to evaluate the joint performance, since the ultimate aim is to reduce the training time needed to obtain a reinforcement learning with a good performance at robotics tasks.

4.1 Object reaching task

For the target reaching task, we implemented our deep Q-learning algorithm and trained it first to complete the task without curriculum. Different training hyper-parameters where tested. The following hyper-parameters where retained because the resulting model gave a model that consistently reached the cube following trajectories seemingly close to the optimal one. The models were trained for about 250 000 environment transitions. Without curriculum, it corresponded to 5000 episodes and 11.83 hours of training, using Tensorflow on a Nvidia GeForce GTX TITAN X GPU.

4.1.1 *Decreasing joint angular velocity* curriculum

We evaluated the decreasing velocities curriculum first, with sparse rewards, and compared it to a sparse rewards baseline without curriculum, and we obtained the results shown in 4.1.

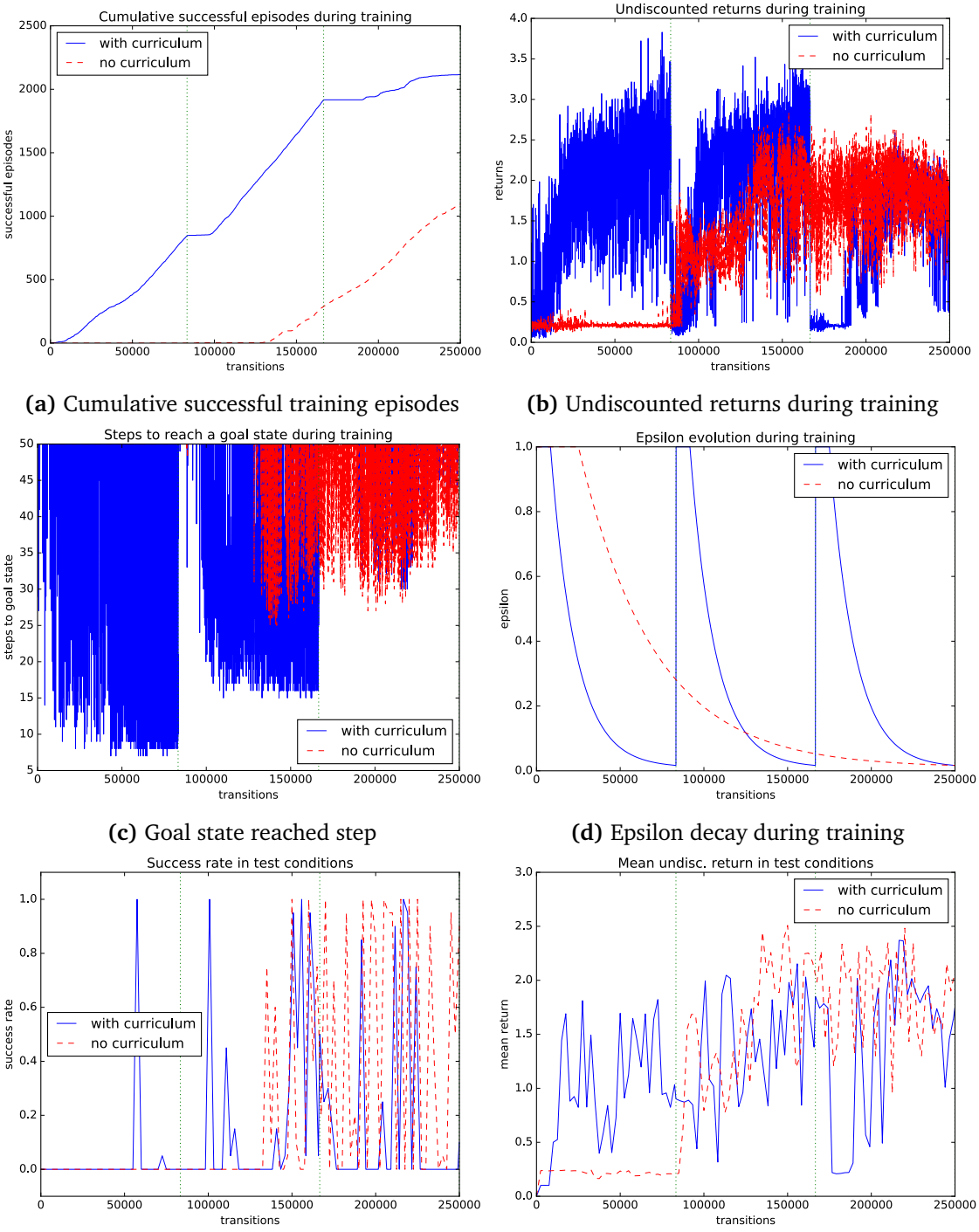
We can see in Figure 4.1a and Figure 4.1c that, using the curriculum, the agent reaches goal states earlier during training. The agent even reaches the goal state in testing conditions as shown in Figure 4.1e, i.e. with the final task velocity, even tough it had only with higher velocities during the first few testing episodes. Nevertheless, the baseline reaches goal states more consistently and the end of training, whereas the curriculum learning agent becomes less successful, shown by the steeper slope of the baseline in Figure 4.1a, the fewer steps to reach the goal in Figure 4.1c, and the higher test success rates in Figure 4.1e, as well as the slightly higher obtained returns obtained in the last few training and testing episodes, as shown in Figures 4.1b and 4.1f.

We also compared the performances of training with the curriculum and sparse rewards, and training with shaping rewards without curriculum, as shown in Figure 4.2. As expected, the performance of the baseline without curriculum is even better, since it surpasses the curriculum agent more consistently and earlier, as shown in Figures 4.2e and 4.2f.

We also trained an agent with both the curriculum and shaping rewards (Figure 4.3). Even if the curriculum agent takes more time to recover from the last sub-task switching (around transitions 17,000 to 20,000), as shown in 4.3a, 4.3e, and 4.3f, its final performance is much more consistent, and comparable to the reference trained without curriculum.

Nevertheless, when compared to a baseline agent trained with shaping rewards, in Figure 4.4, we can see that the baseline eventually performs slightly better than the curriculum agent.

To sum up, even though the agents trained with the curriculum effectively reach goal states earlier and more often compared to the reference at the beginning of the training,



(a) Cumulative successful training episodes

(b) Undiscounted returns during training

(c) Goal state reached step

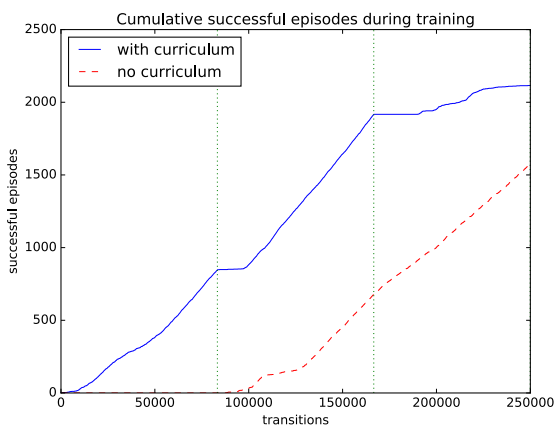
(d) Epsilon decay during training

(e) Success rate at test episodes

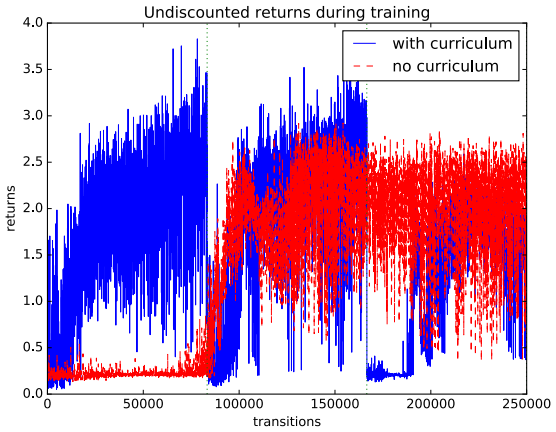
(f) Mean return at test episodes

Figure 4.1: Decreasing joint angle velocity curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

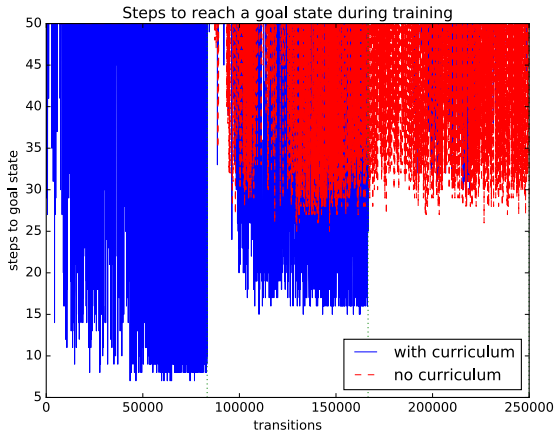
the final model did not really benefit from it, since its performance is surpassed by the agent trained without curriculum. We observe that the last sub-task switching, which corresponds to decreasing the angular velocity from 2.0 rad/s to 1.0 rad/s greatly reduces



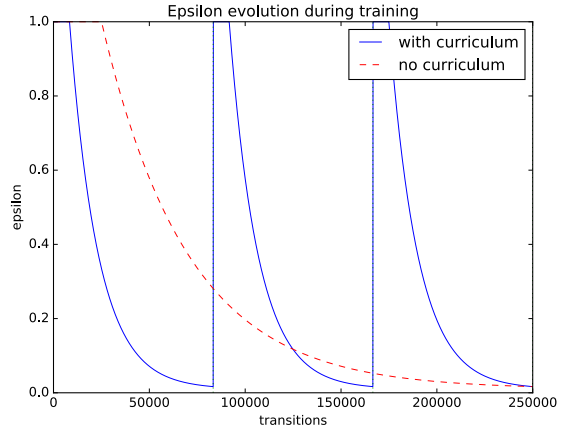
(a) Cumulative successful training episodes



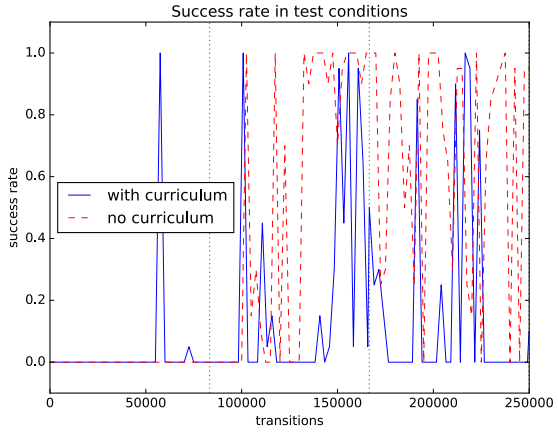
(b) Undiscounted returns during training



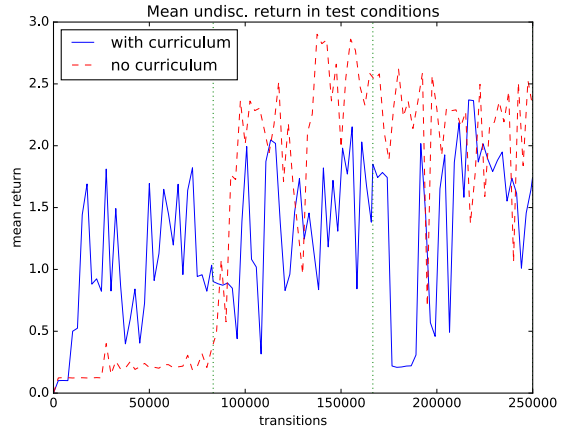
(c) Goal state reached step



(d) Epsilon decay during training



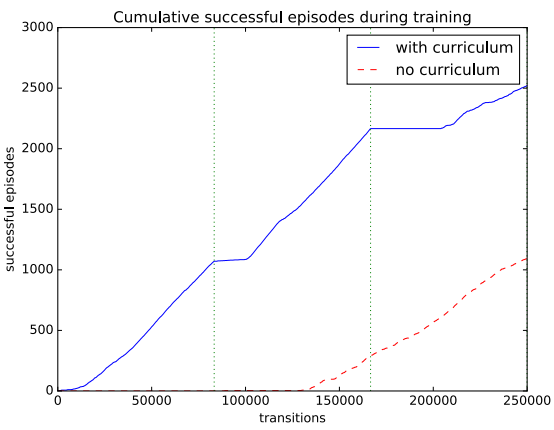
(e) Success rate at test episodes



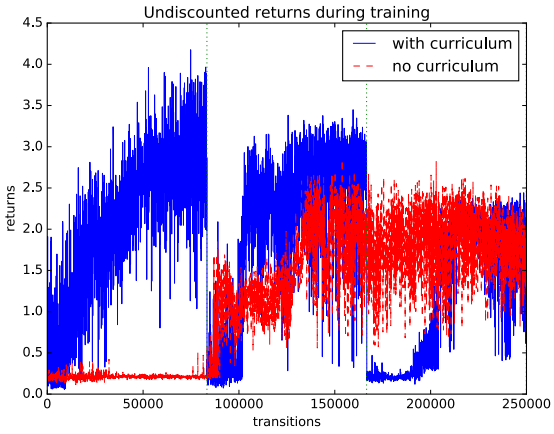
(f) Mean return at test episodes

Figure 4.2: Decreasing joint angle velocity curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards.

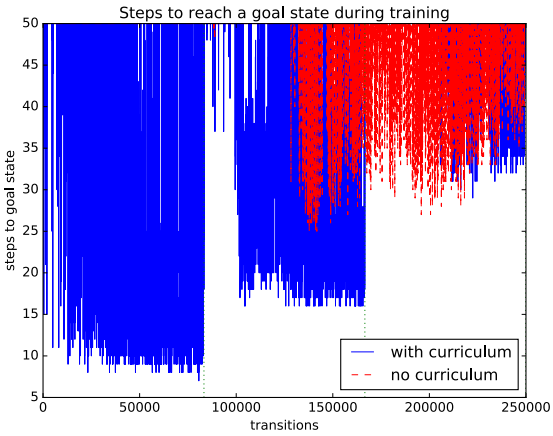
the agent performance for a large number of transitions. Nevertheless, if we only consider the number of transitions corresponding to the target task (i.e. last curriculum sub-task), the curriculum agent has a better performance. Since the agent benefits from the simpler



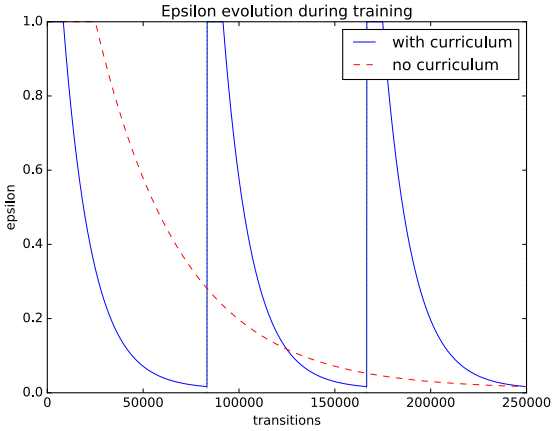
(a) Cumulative successful training episodes



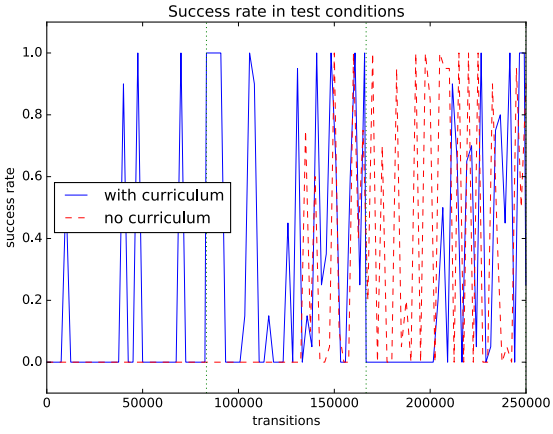
(b) Undiscounted returns during training



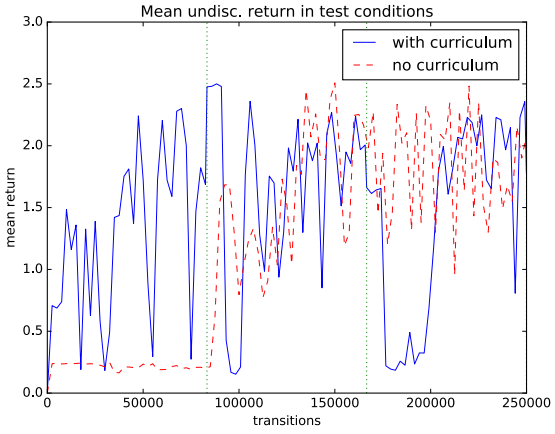
(c) Goal state reached step



(d) Epsilon decay during training



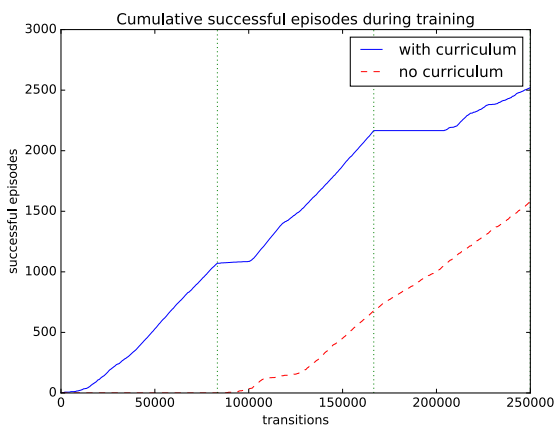
(e) Success rate at test episodes



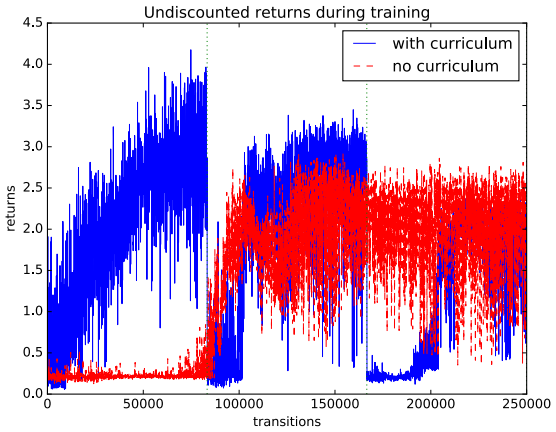
(f) Mean return at test episodes

Figure 4.3: Decreasing joint angle velocity curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

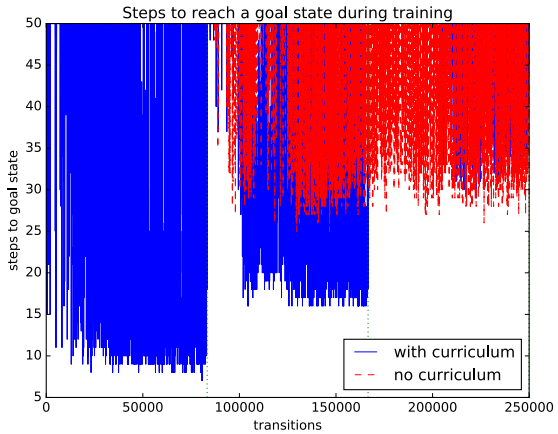
tasks, curriculum learning effectively works as a pre-training technique. It seems likely that the agent would benefit more from a curriculum that allocates training transitions to sub-tasks unequally. For instance, the agent may achieve a better performance if more



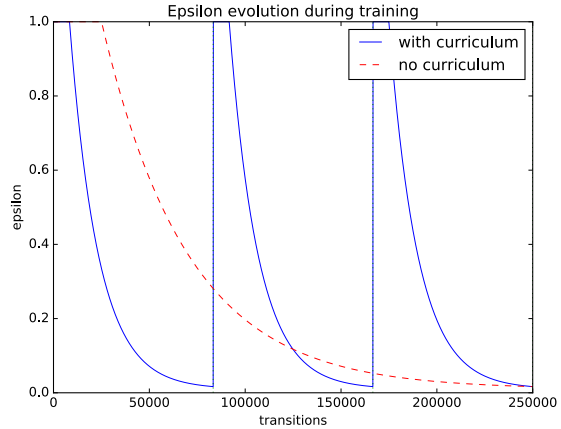
(a) Cumulative successful training episodes



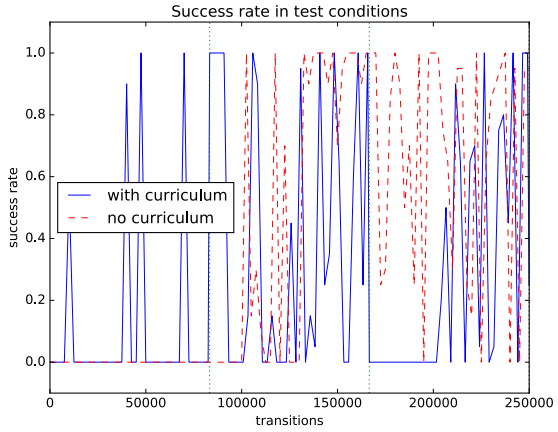
(b) Undiscounted returns during training



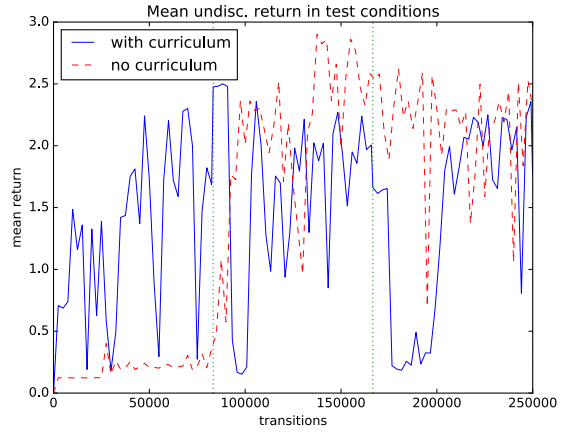
(c) Goal state reached step



(d) Epsilon decay during training



(e) Success rate at test episodes



(f) Mean return at test episodes

Figure 4.4: Decreasing joint angle velocity curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards.

transitions are allocated to the last sub-task. Indeed, the agent would make much more interactions in target task conditions while still building from previous experience at simpler tasks.

4.1.2 *Initializing robot position closer to the target* curriculum

Now we evaluate the performance of the curriculum consisting in initializing the position of the robot end effector closer to the cube and progressively initializing it further away from it. We tested using three different initial robot configurations, the last one corresponding to the initial configuration used for all other curricula.

We compared the curriculum agents trained with both sparse and shaping rewards and compared them to the the corresponding baseline agents, as shown in 4.5, 4.6, 4.7, and 4.8. The curriculum agent fails to learn the task, as the number of successful episodes is very little, during both training and testing episodes.

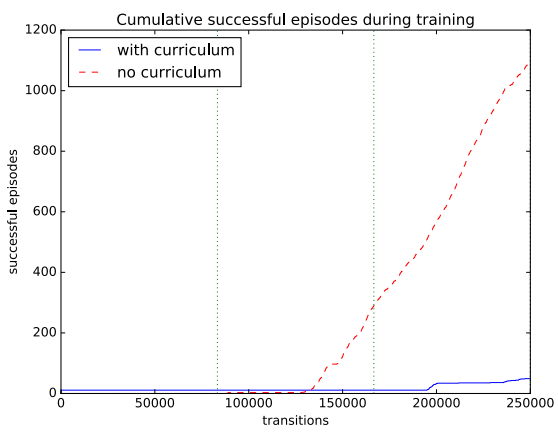
These results were not expected, since intuitively initializing the end-effector increases the probability of the end-effector to reach the cube, even with random actions. We hypothesize that it is due to the fact that the maximum number of steps per training episode is too small for the first two sub-tasks (5 and 20 steps per episode, compared to 50 steps per episode for the final sub-task).Indeed, we designed the maximum number of steps for these sub-tasks by running some random episodes from the initial positions and taking the smallest number of steps for which a goal state was reached, since we wanted to avoid overestimation, as mentioned in Subsection 3.3.3. Another advantage of giving the agent less steps per episode when it is close to the target is that it is more likely that the Q-values learned for the first few sub-tasks will approximate more closely the Q-values of the

Since we get some successes during training for the first sub-task, the intuition we have is that the goal states were reached using the stochasticity of the environment, and the robot is not able to reproduce these episodes on its own. Experiments allocating more steps per episode for each sub-task is necessary to really evaluate this curriculum.

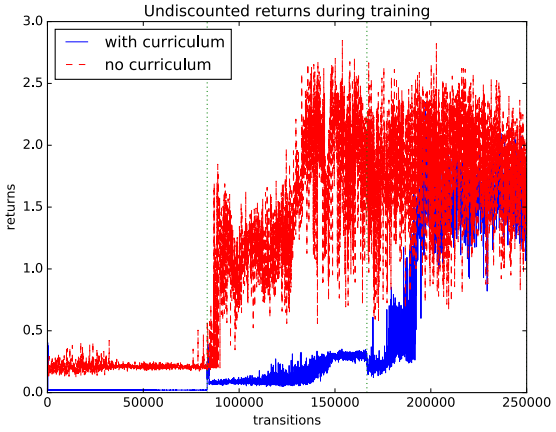
We have to note that it is difficult to compare the curves corresponding to the steps to reach a goal state, and the undiscounted returns during training, since the number of steps per episode is not the same.

Nevertheless, as mentioned before, if we only consider the transitions in target task conditions, the curriculum agent manages to obtain higher rewards and successful episodes earlier. However, this may also be due to the time-steps constant of the ϵ decay function. Indeed, the final sub-tasks starts exploiting the learned policy sooner than the non-curriculum agent.

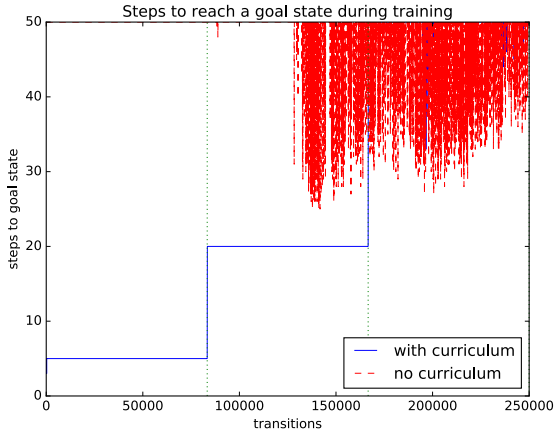
To sum up, more experiments giving more steps per episode to initial sub-tasks are needed to conclude whether this curriculum is not effective. A good approach to set the number of steps to execute according to the initial position of the robot could be to adapt the method followed in [Popov et al., 2017]: sending actions defined by a human operator (or by another control method, such as an inverse kinematics approach) at each time step and record the number of steps needed to go from the furthest state to the goal. Then, intermediate states belonging to the obtained trajectories can be used as initial states and the remaining time-steps are given to the learning agent.



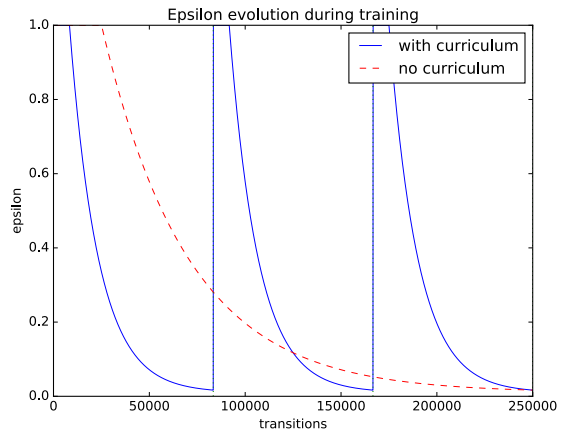
(a) Cumulative successful training episodes



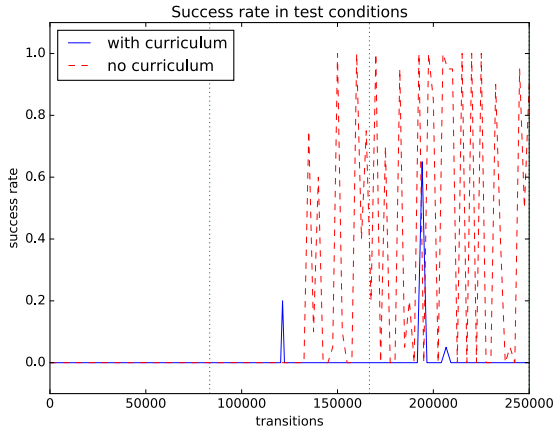
(b) Undiscounted returns during training



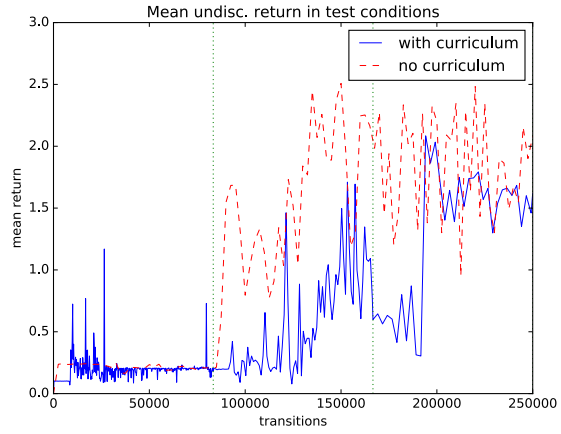
(c) Goal state reached step



(d) Epsilon decay during training

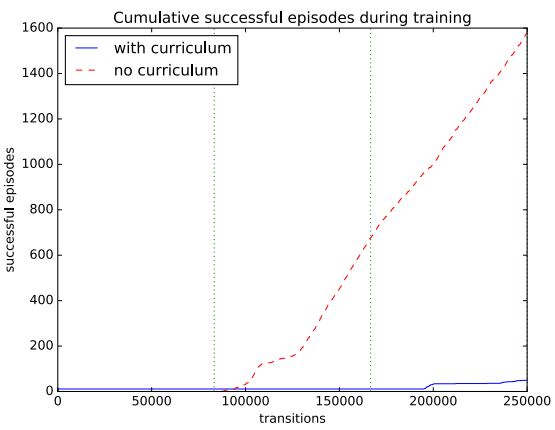


(e) Success rate at test episodes

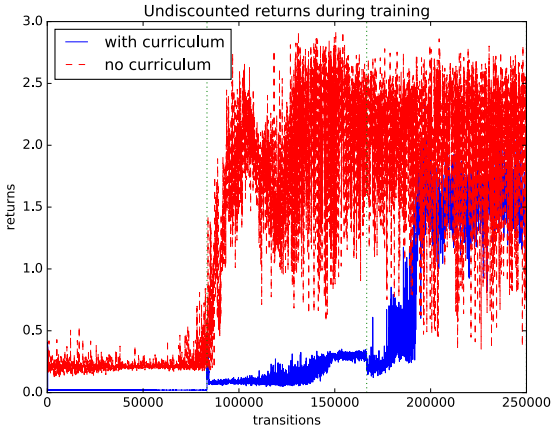


(f) Mean return at test episodes

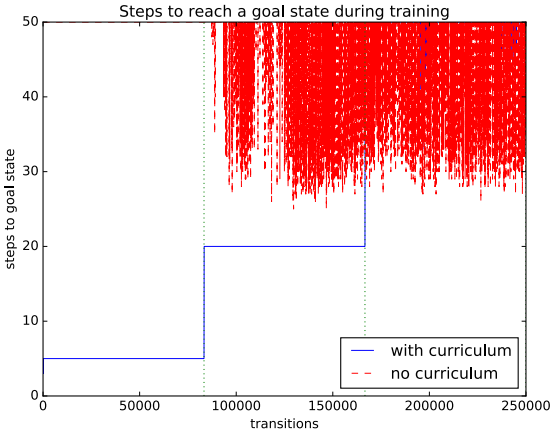
Figure 4.5: Further initial states curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



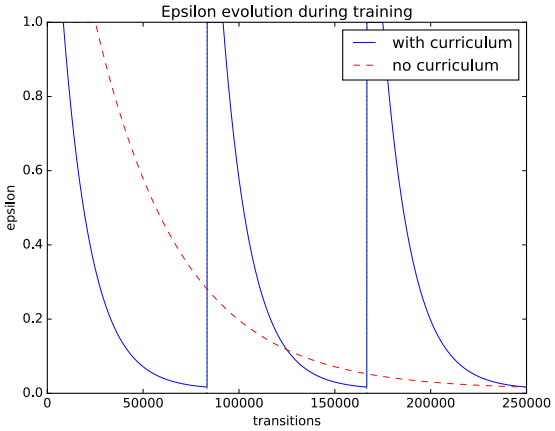
(a) Cumulative successful training episodes



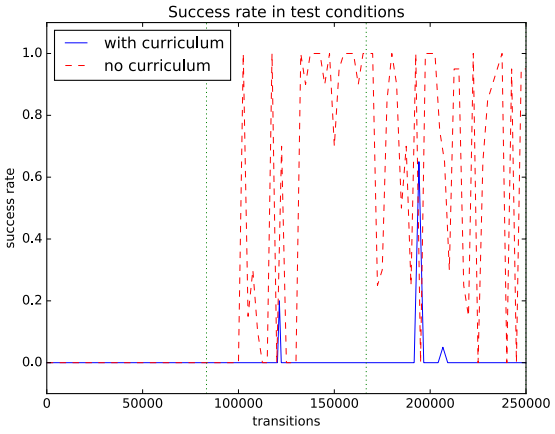
(b) Undiscounted returns during training



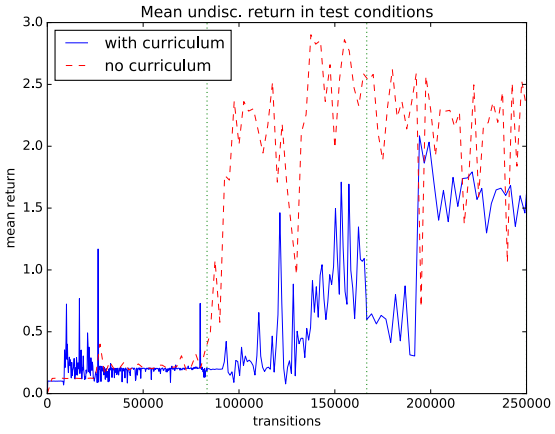
(c) Goal state reached step



(d) Epsilon decay during training

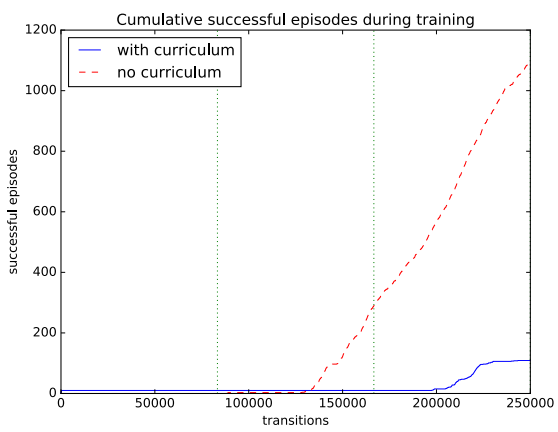


(e) Success rate at test episodes

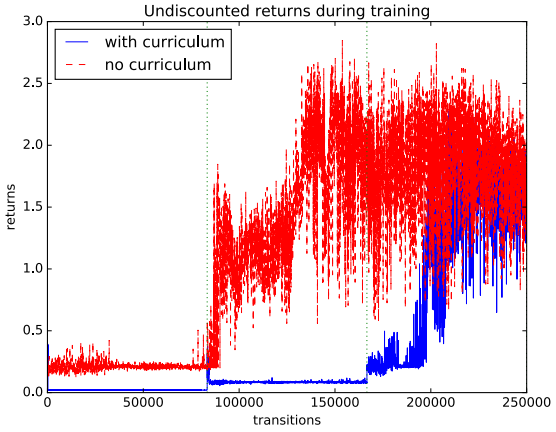


(f) Mean return at test episodes

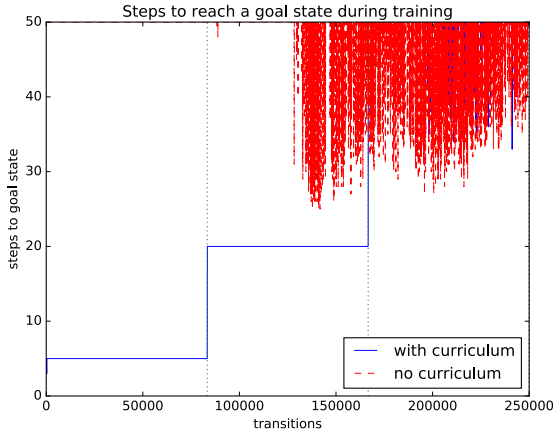
Figure 4.6: Further initial states curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards.



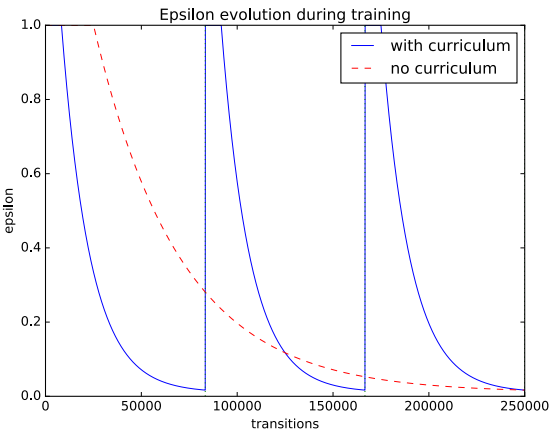
(a) Cumulative successful training episodes



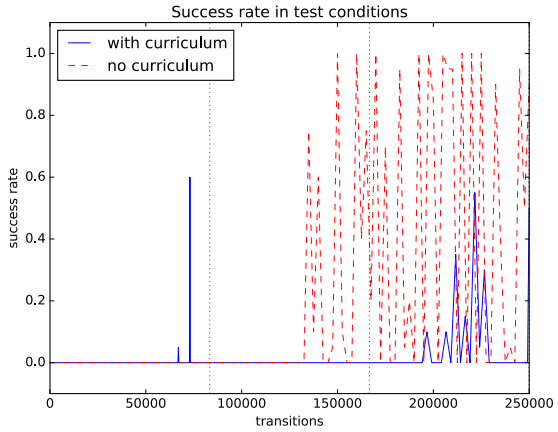
(b) Undiscounted returns during training



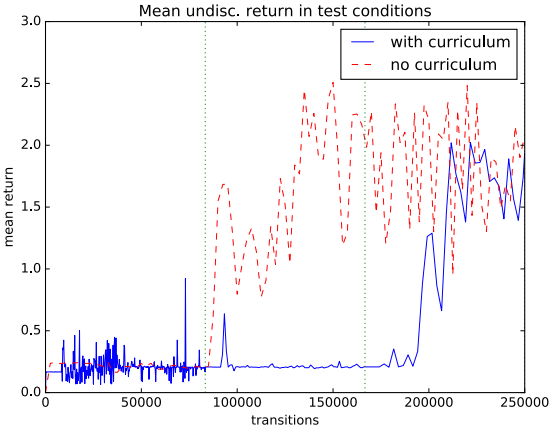
(c) Goal state reached step



(d) Epsilon decay during training

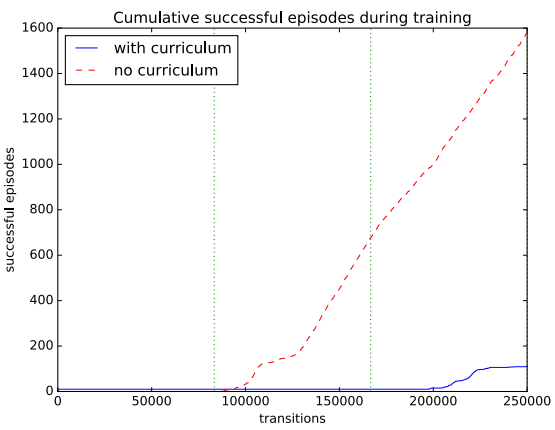


(e) Success rate at test episodes

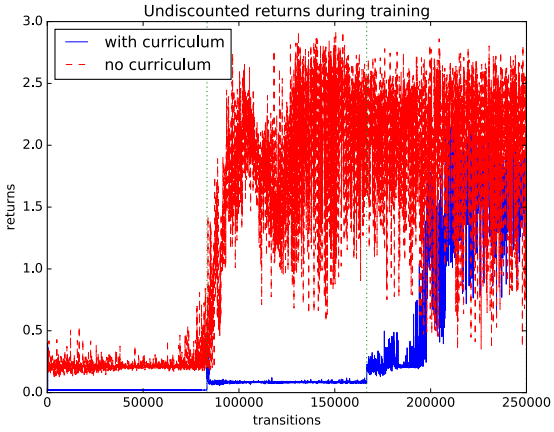


(f) Mean return at test episodes

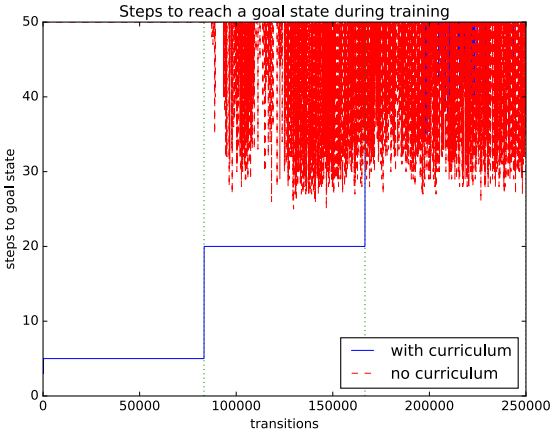
Figure 4.7: Further initial states curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



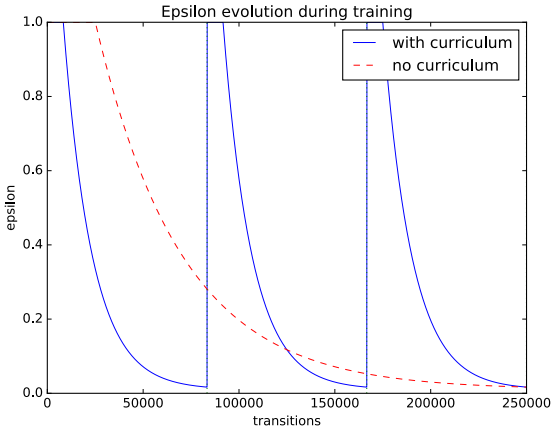
(a) Cumulative successful training episodes



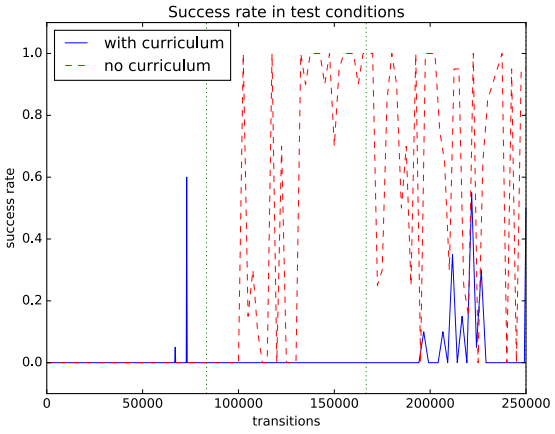
(b) Undiscounted returns during training



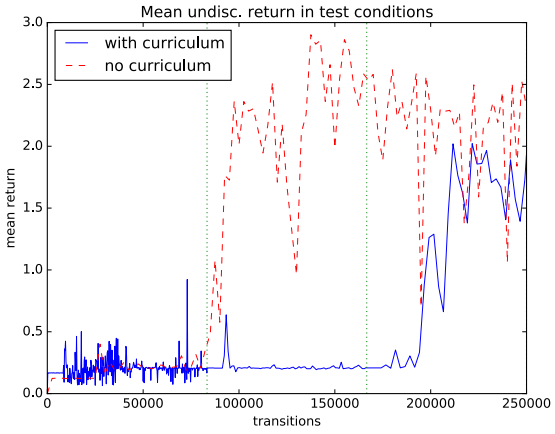
(c) Goal state reached step



(d) Epsilon decay during training



(e) Success rate at test episodes



(f) Mean return at test episodes

Figure 4.8: Further initial states curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **shaping** rewards.

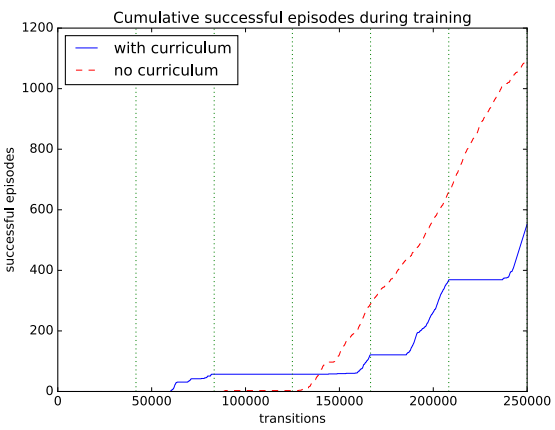
4.1.3 *Increasing number of moving joints* curriculum

The last curriculum evaluated consists in progressively increasing the number of joints that the agent can control. Sub-task switching consists in increasing the size of the output layer of the DQN to control the corresponding joint. The state vector corresponding to the input layer remains unchanged. Figure 4.9 shows the results corresponding to the curriculum agent compared to the baseline. We can see that the curriculum agent gets some successes and bigger returns during training before the baseline, which may be due to the fewer exploration episodes though. More interestingly, we clearly see the impact of sub-task switching on the performance of the learned policy in test conditions: success rate and mean returns greatly decrease during episodes corresponding to higher ϵ value, i.e. exploration episodes. We can observe than despite this, the agent is able to quickly recover as ϵ decreases, and even surpass the baseline performance: it gets more return and successes during the last few test episodes, and even during training the cumulative successes have a steep increase greater than the baseline, and the goal is reached in fewer steps, as shown in Figures 4.9f, 4.9e, 4.9, and 4.9c

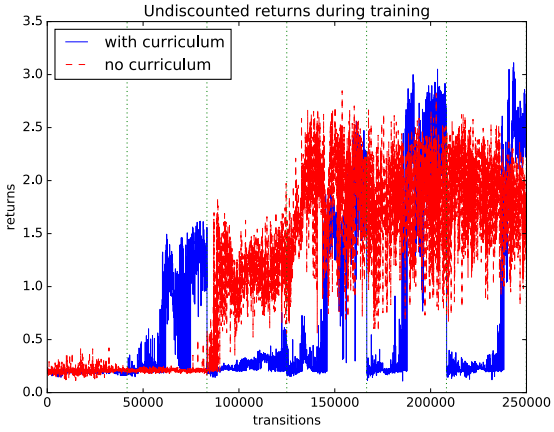
The curriculum as a good performance even compared to the shaping rewards baseline, as shown by the results in Figure 4.10: test and training returns, as well the test and training success rate, are eventually higher for the curriculum agent, and the goal is reached in fewer steps.

Surprisingly, using shaping rewards with curriculum learning does not improve the performance of the agent, which is not better than the baselines without curriculum learning, as shown in 4.11 and 4.12. In this case, curriculum learning with sparse rewards gives the best results. We have the intuition

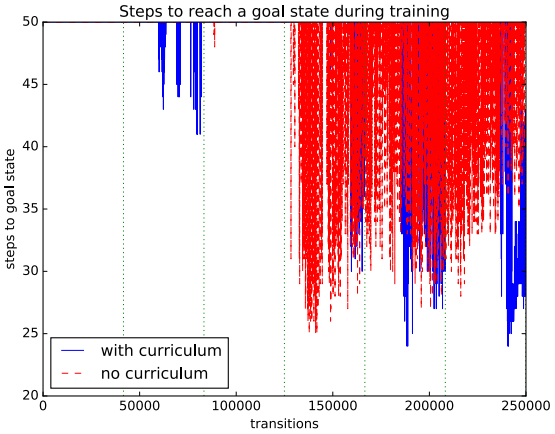
To sum up, increasing the number of moving joints is the curriculum that worked the best so far, even better than shaping rewards, and achieves a better performancs without shaping rewards. Further experiments should to be done with the other curricula, as the maximum number of steps and the schedule of sub-task switching may have a big impact on the training and performance of the final agent.



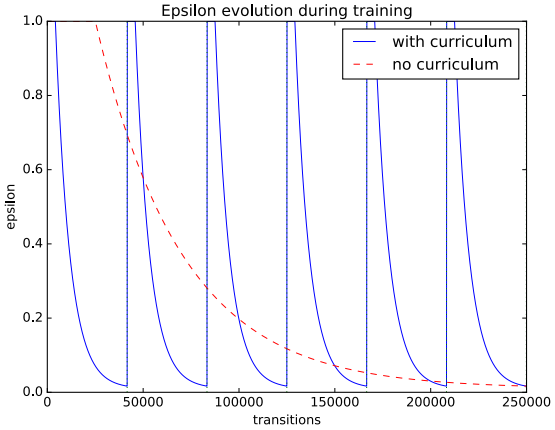
(a) Cumulative successful training episodes



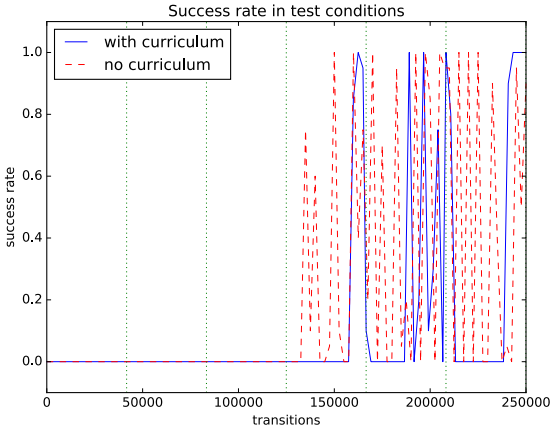
(b) Undiscounted returns during training



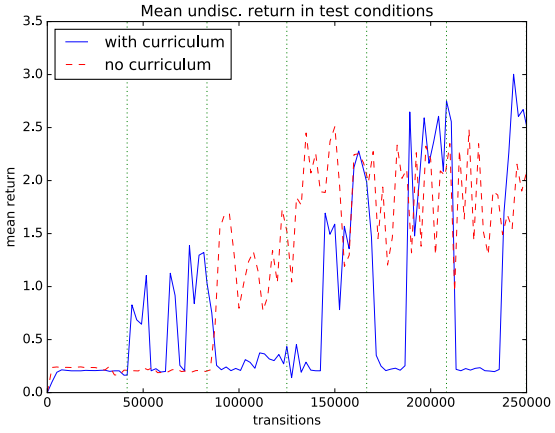
(c) Goal state reached step



(d) Epsilon decay during training

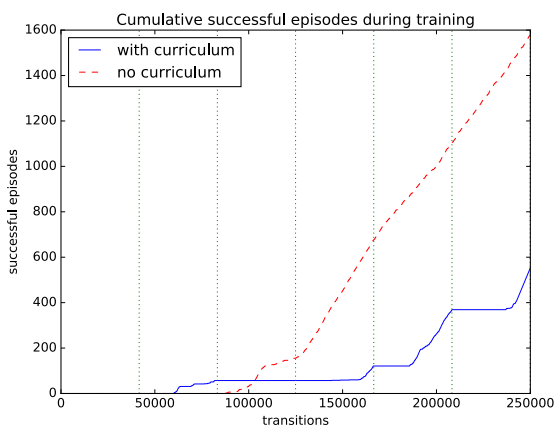


(e) Success rate at test episodes

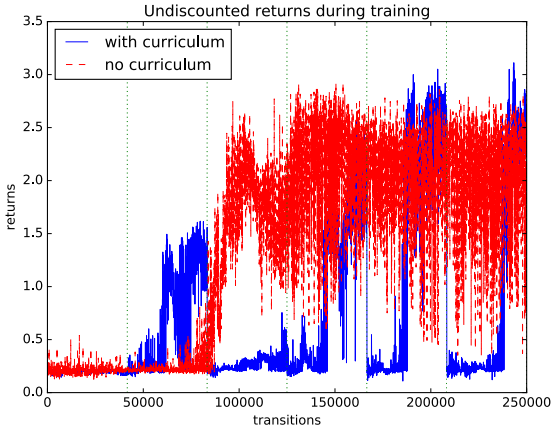


(f) Mean return at test episodes

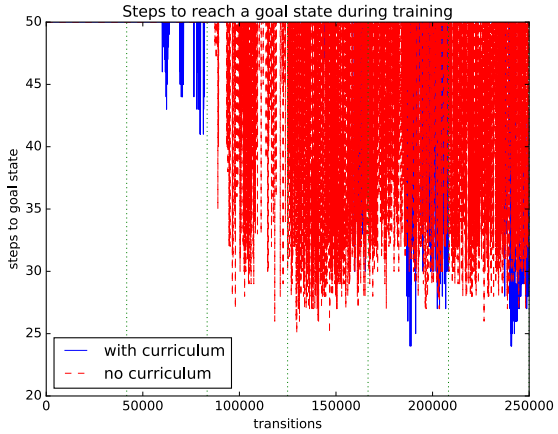
Figure 4.9: Increasing number of joints curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



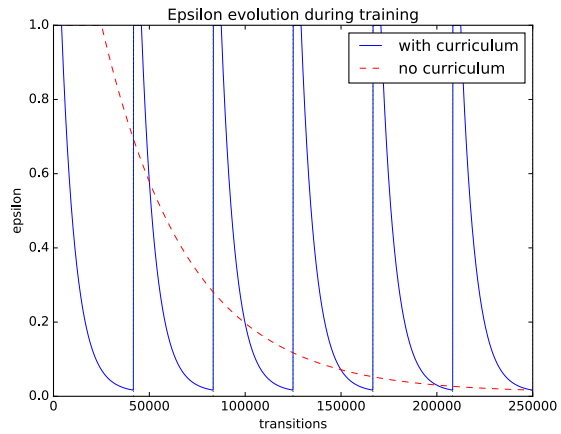
(a) Cumulative successful training episodes



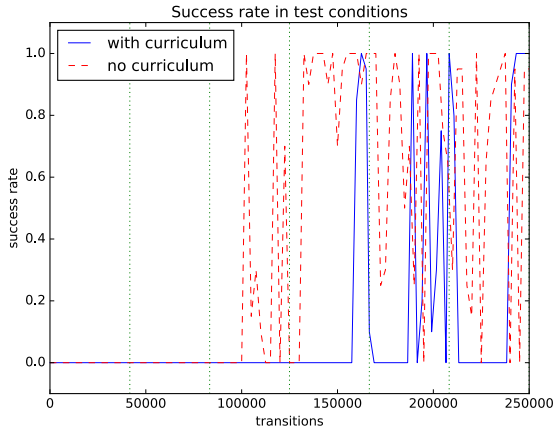
(b) Undiscounted returns during training



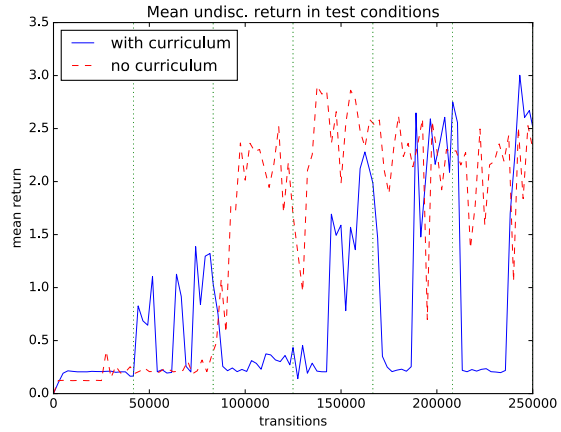
(c) Goal state reached step



(d) Epsilon decay during training

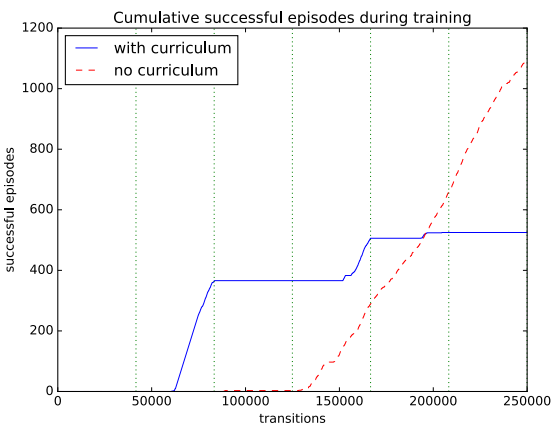


(e) Success rate at test episodes

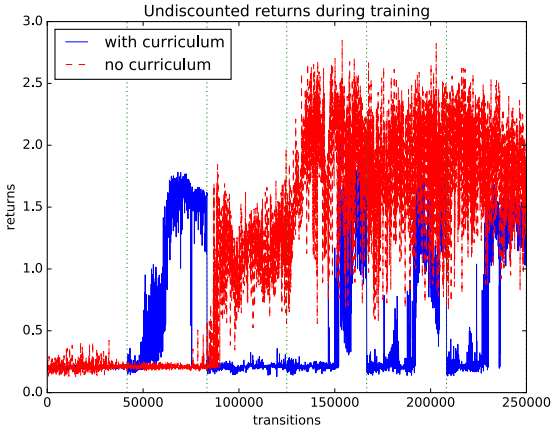


(f) Mean return at test episodes

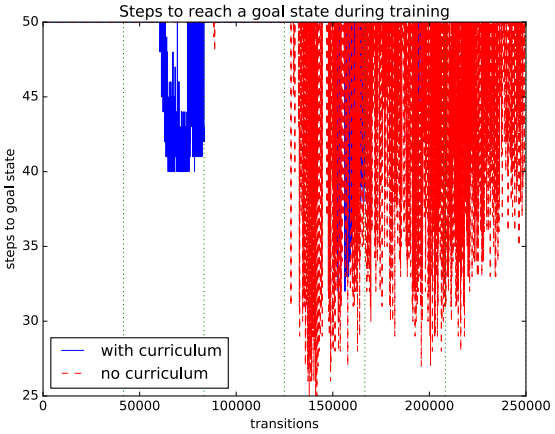
Figure 4.10: Increasing number of joints curriculum with **sparse** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards.



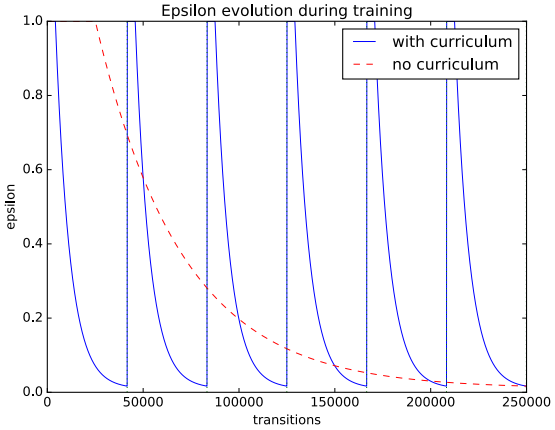
(a) Cumulative successful training episodes



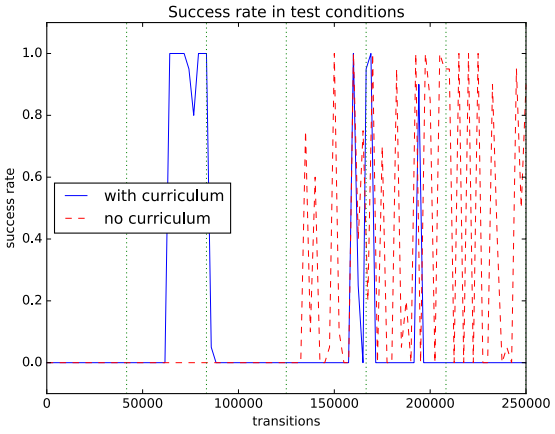
(b) Undiscounted returns during training



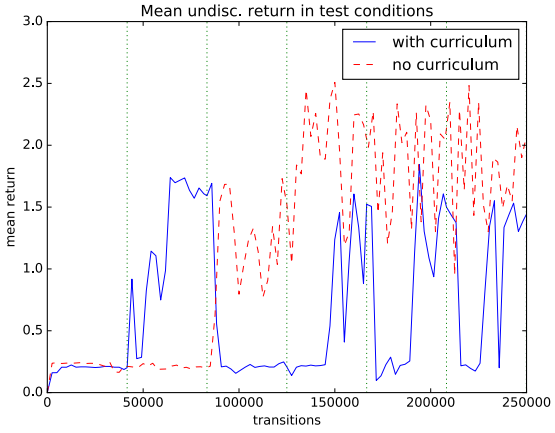
(c) Goal state reached step



(d) Epsilon decay during training

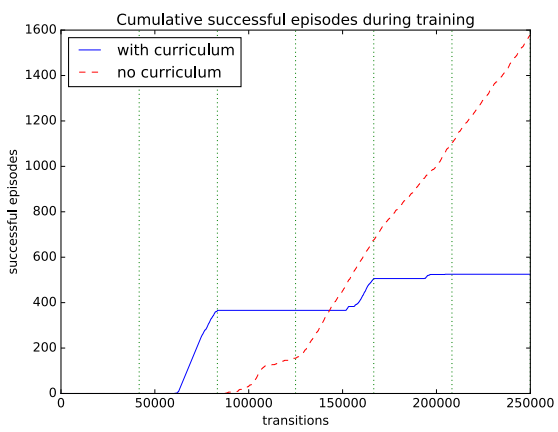


(e) Success rate at test episodes

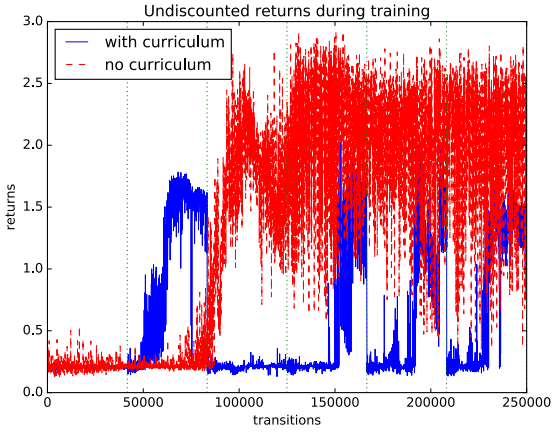


(f) Mean return at test episodes

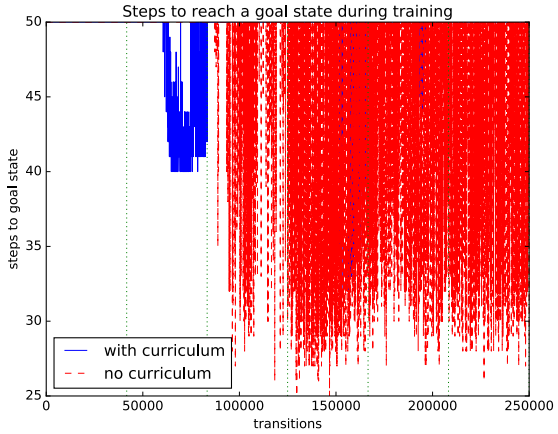
Figure 4.11: Increasing number of joints curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



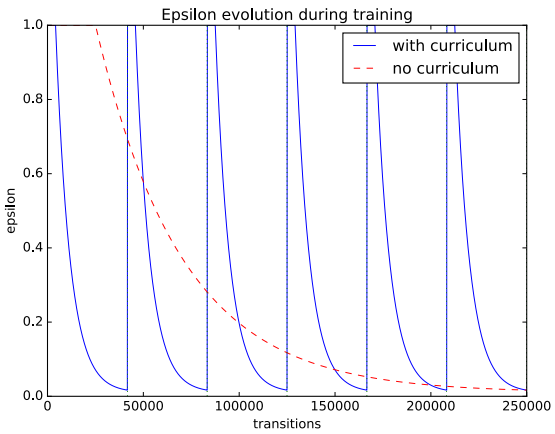
(a) Cumulative successful training episodes



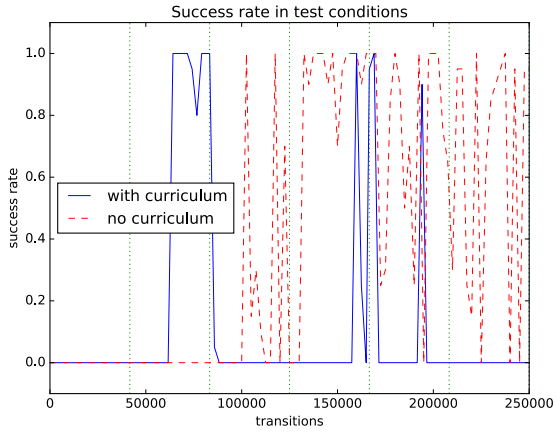
(b) Undiscounted returns during training



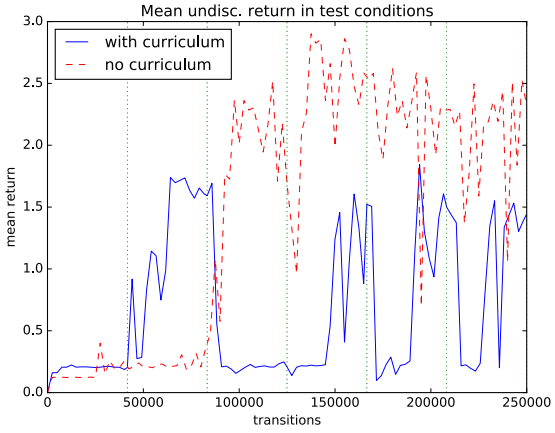
(c) Goal state reached step



(d) Epsilon decay during training



(e) Success rate at test episodes



(f) Mean return at test episodes

Figure 4.12: Increasing number of joints curriculum with **shaping** rewards performance at the reaching task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards.

4.2 Object pushing task

Following the results obtained for the target reaching task, we evaluated the similar curricula for task consisting in not only reaching the cube, but also pushing it to a target position. To keep the task relatively simple, the initial position of the cube and the target position remain unchanged throughout the training. We used the hyper-parameters used for the previous taskT, without training the baseline to be successful at the task, and implemented our deep Q-learning algorithm with Tensorflow on a Nvidia GeForce GTX 1060.

4.2.1 *Decreasing joint angular velocity* curriculum

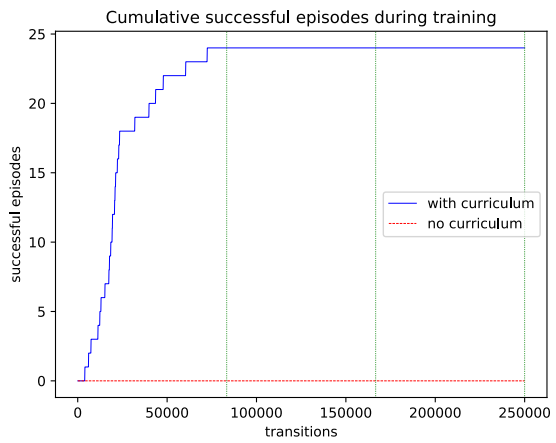
We evaluated the performance of decreasing the angular velocity first and we obtained the results shown in 4.13. Even if the baseline does not see any successful training episode, the curriculum agent is able to be successful at a few training episodes, and gets higher returns during training episodes, especially for the first two sub-tasks. The curriculum agent also obtains higher returns during testing episodes, but is unable to achieve successful test episodes. This might be due to a too restrictive steps per episode limit, similar to the case described in 4.1.2. Indeed, the agent is able to increase the return obtained during the first two sub-tasks but is unable to reach a goal state. Remarkably, the curriculum agent performance decreases during the last subtask, thus it fails to build on the experience gathered from previous sub-tasks.

Nevertheless, the curriculum learning agent performs slightly better than the baseline, and further experiments giving more steps per episode to complete the task are necessary.

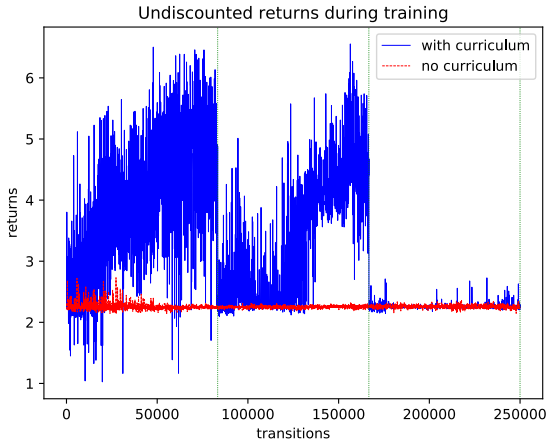
The curriculum agent also performs better that the shaping rewards baseline, which also performs poorly, as Figure 4.14 shows.

Using shaping rewards increases the performance of the curriculum agent during training, but has little impact on the performance at the target task. 4.15

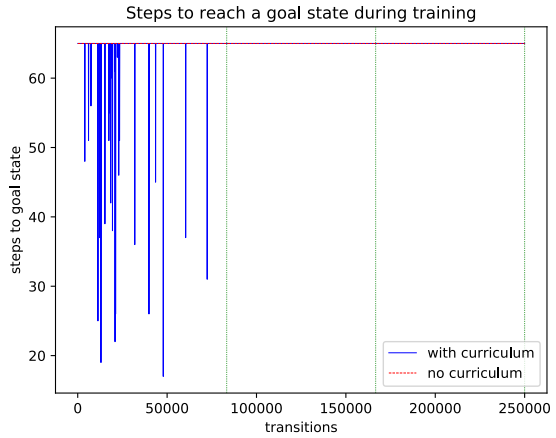
To conclude, even if we hoped that using a curriculum might make the agent learn to complete the task within the allocated training transitions, it seems necessary to tune the baseline model hyper-parameters so that it is able to complete the task without curriculum first, and then evaluate curriculum learning. Another approach could be to give the algorithm more time-steps per episode and then try to make the agent learn the task using a curriculum.



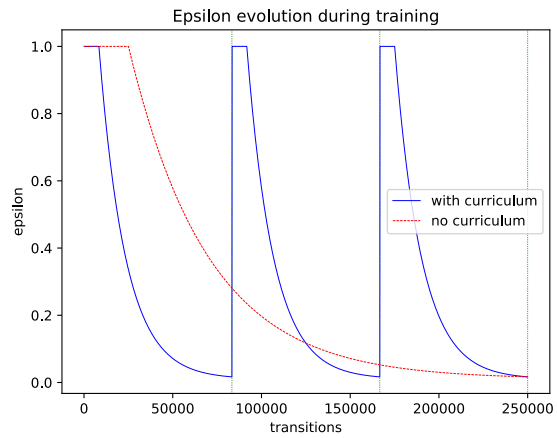
(a) Cumulative successful training episodes



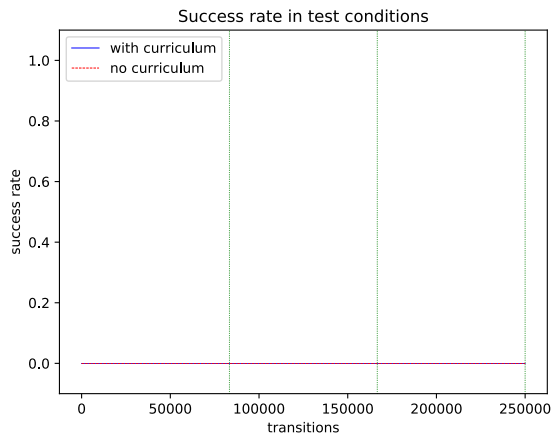
(b) Undiscounted returns during training



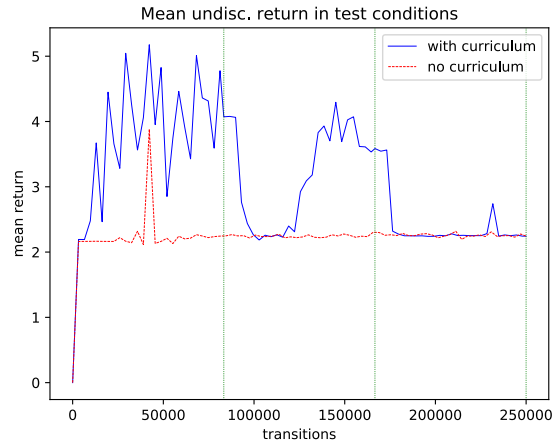
(c) Goal state reached step



(d) Epsilon decay during training

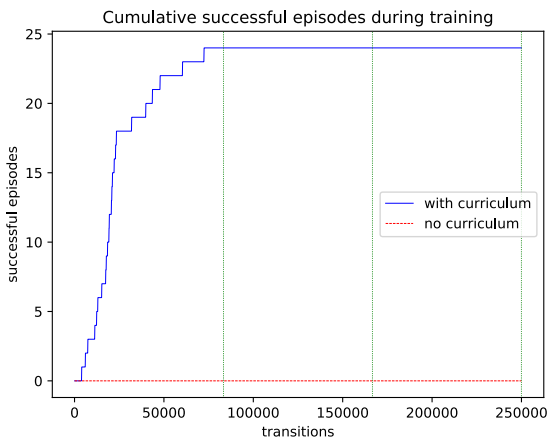


(e) Success rate at test episodes

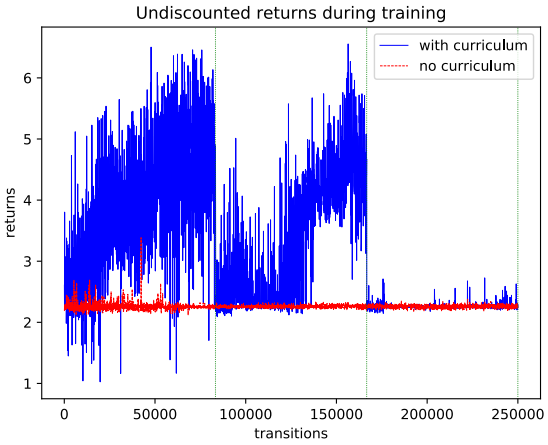


(f) Mean return at test episodes

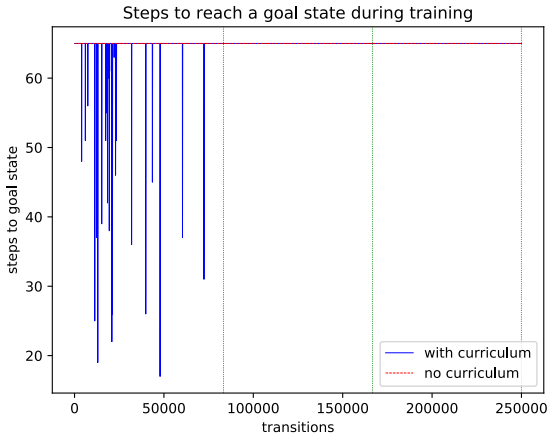
Figure 4.13: Decreasing joint angle velocity curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



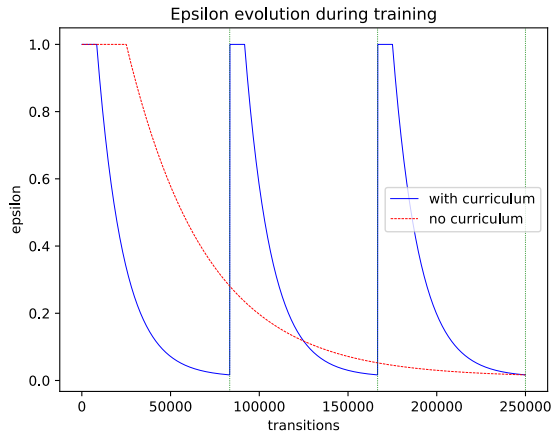
(a) Cumulative successful training episodes



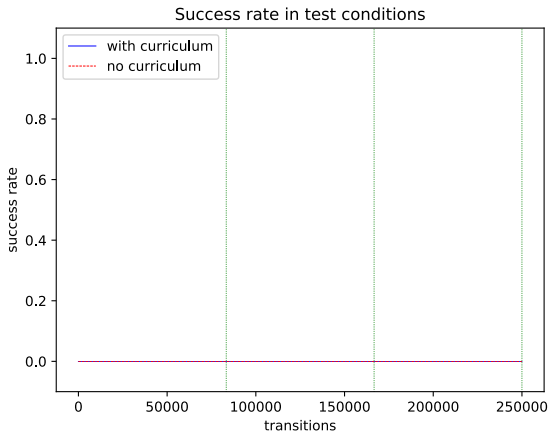
(b) Undiscounted returns during training



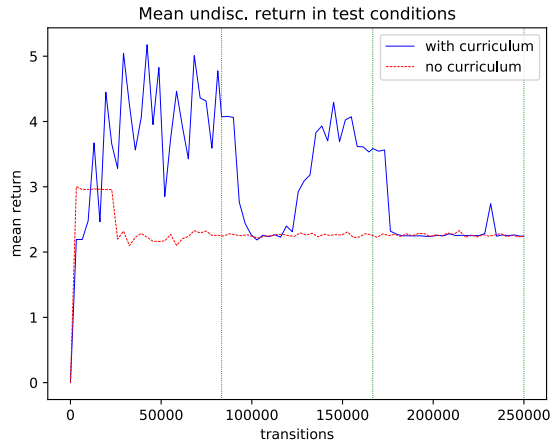
(c) Goal state reached step



(d) Epsilon decay during training

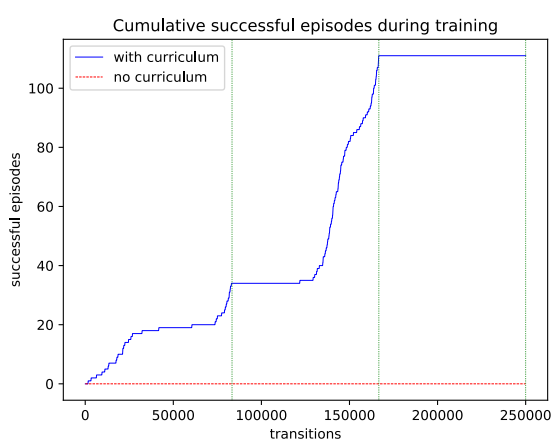


(e) Success rate at test episodes

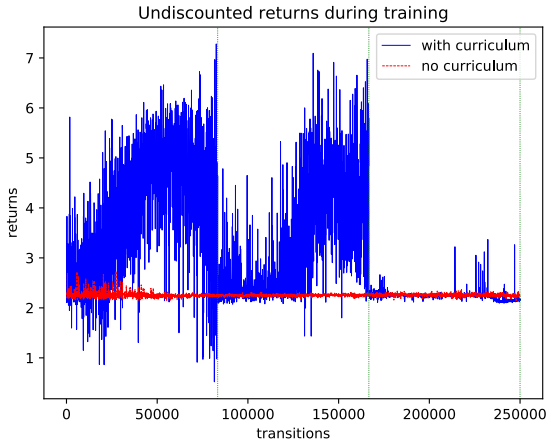


(f) Mean return at test episodes

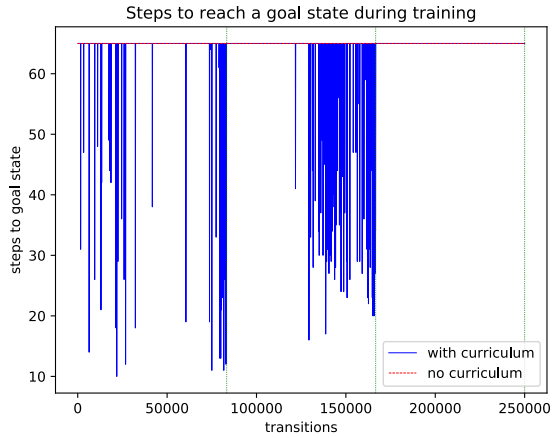
Figure 4.14: Decreasing joint angle velocity curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum with **shaping** rewards.



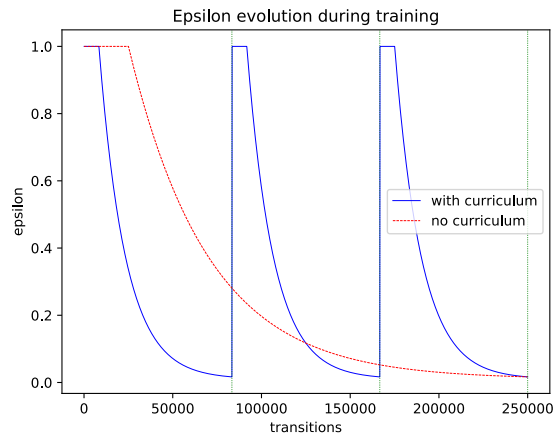
(a) Cumulative successful training episodes



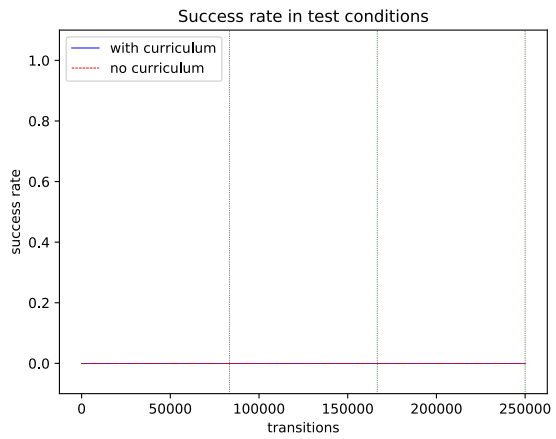
(b) Undiscounted returns during training



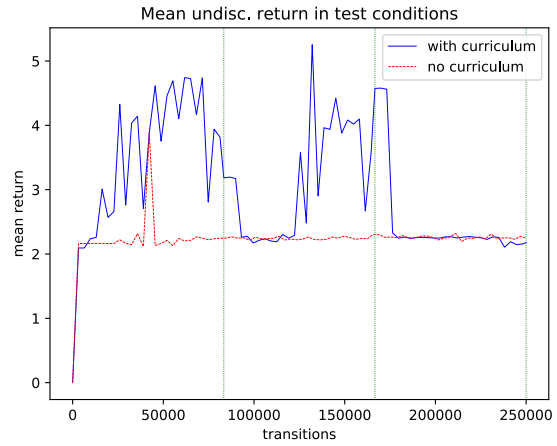
(c) Goal state reached step



(d) Epsilon decay during training



(e) Success rate at test episodes



(f) Mean return at test episodes

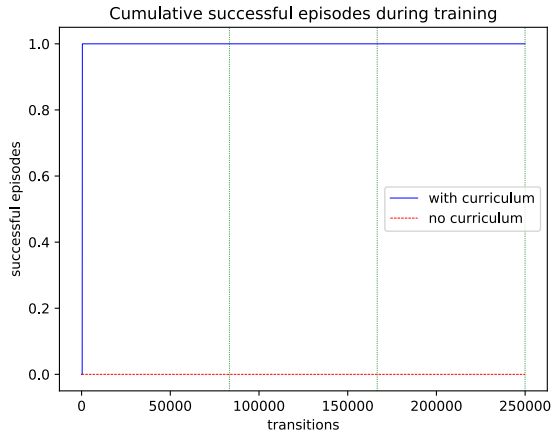
Figure 4.15: Decreasing joint angle velocity curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

4.2.2 *Initializing robot position closer to target* curriculum

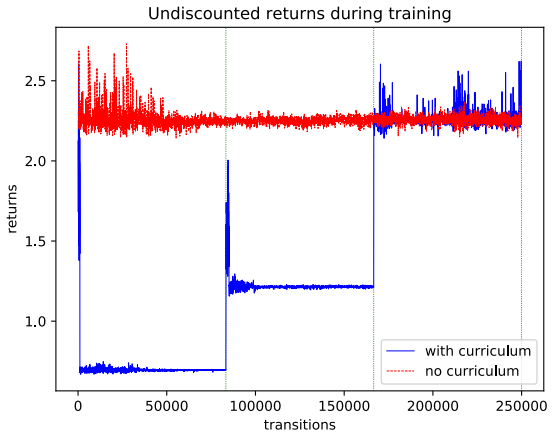
Initializing the robot state closer to the target gives a slightly better performance compared to the baseline, as shown in Figure 4.16. From the returns obtained during training and testing episodes, which are maximal when the agent performs random actions the most, we confirm that stochasticity is indeed needed to get closer to a goal state.

Similar results are obtained when combinining curriculum learning with shaping rewards, as Figure 4.17 shows.

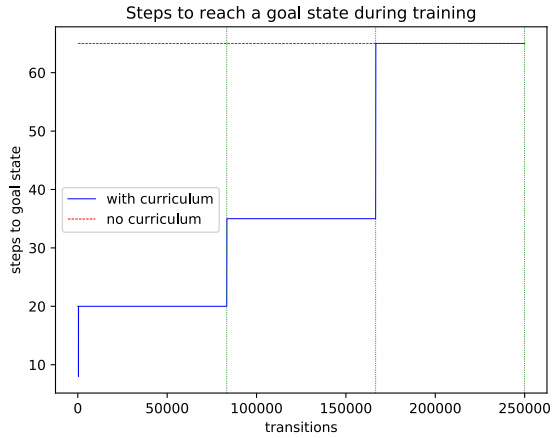
Like we said before, more steps per episode seem to be needed to reach goal states without relying in the random transitions of the environment.



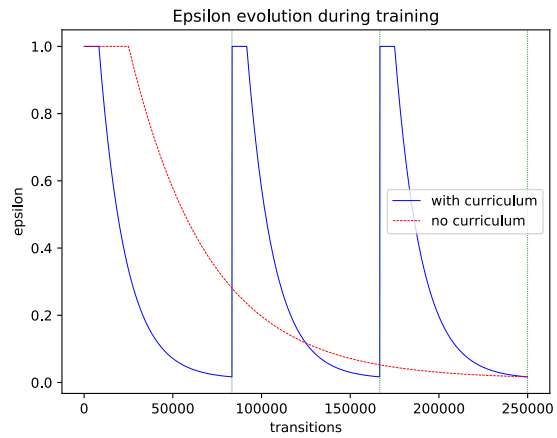
(a) Cumulative successful training episodes



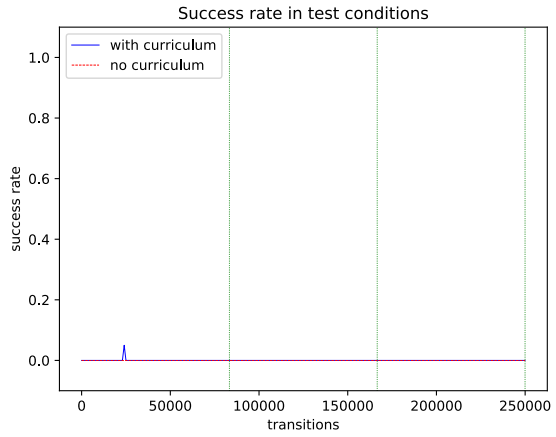
(b) Undiscounted returns during training



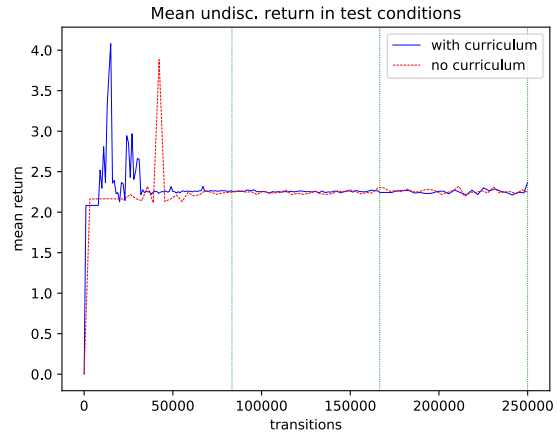
(c) Goal state reached step



(d) Epsilon decay during training



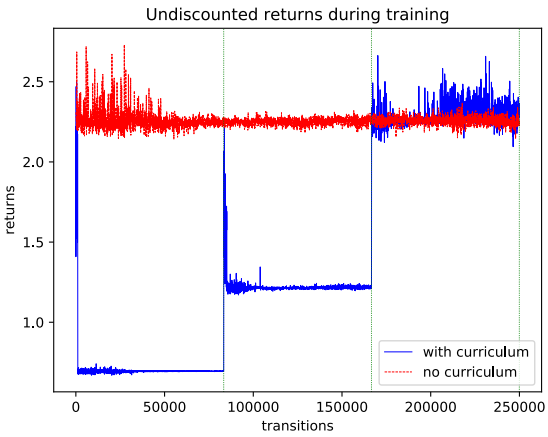
(e) Success rate at test episodes



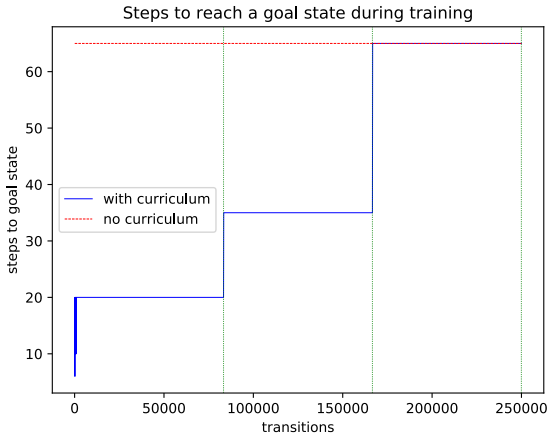
(f) Mean return at test episodes

Figure 4.16: Initial states further away from object curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

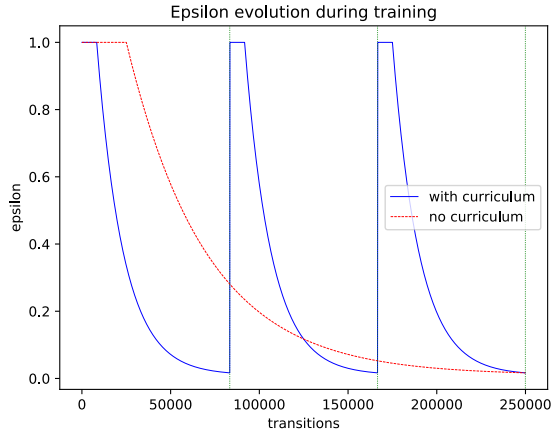
(a) Cumulative successful training episodes



(b) Undiscounted returns during training

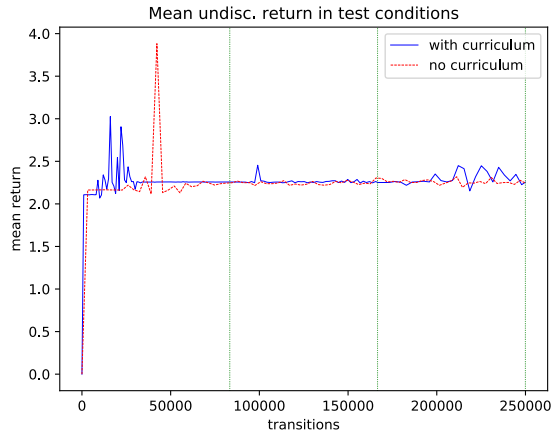


(c) Goal state reached step



(d) Epsilon decay during training

(e) Success rate at test episodes



(f) Mean return at test episodes

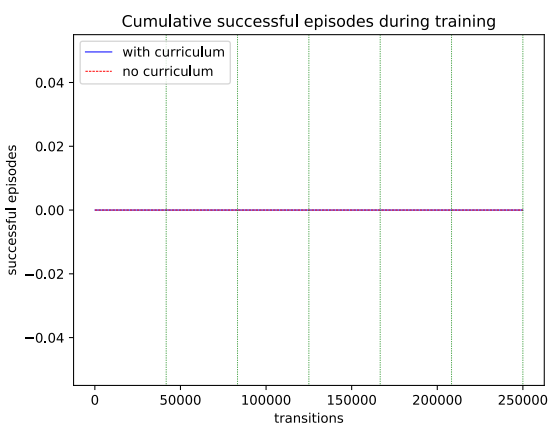
Figure 4.17: Initial states further away from object curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

4.2.3 *Increasing number of moving joints* curriculum

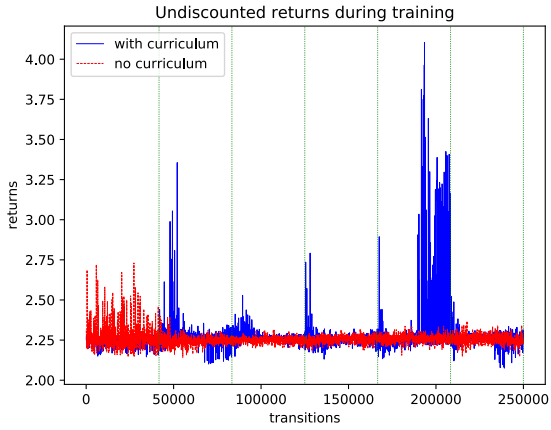
The performance of using a curriculum that increases the number of moving joints is better than the baseline, since Figure 4.18 shows that the agent obtains higher returns during training and testing episodes.

However, the performance on the final task is similar to that of the baseline. Using shaping rewards improves the performance on the final task, as shown in Figure 4.19, but in neither case the agent reaches a goal state.

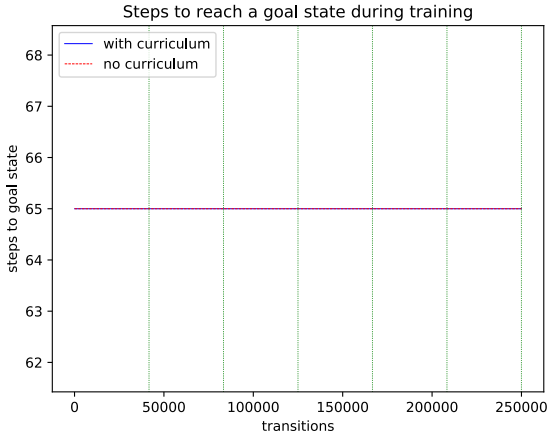
The performance of this curriculum also needs to be evaluated with an algorithm that gives the agent more steps per episode, similar to the previous cases.



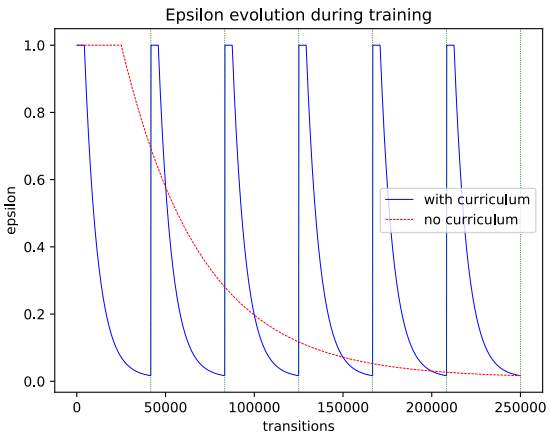
(a) Cumulative successful training episodes



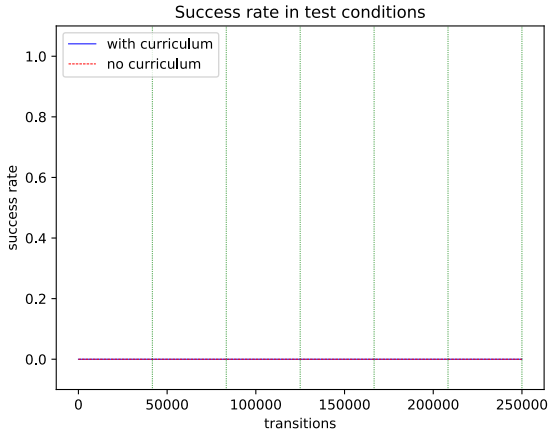
(b) Undiscounted returns during training



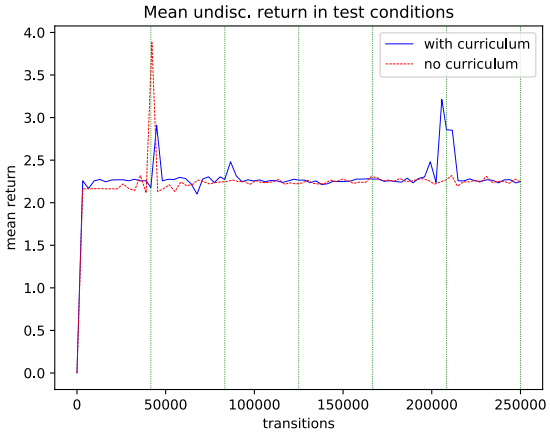
(c) Goal state reached step



(d) Epsilon decay during training

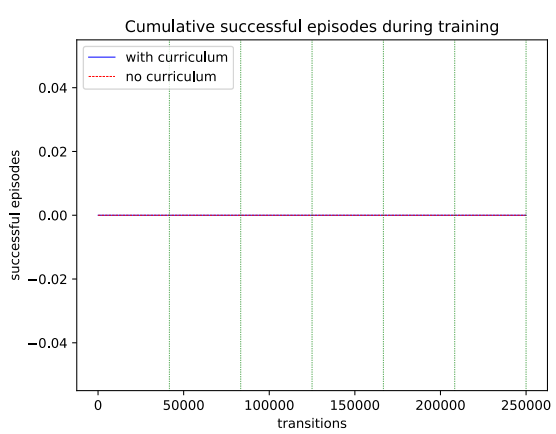


(e) Success rate at test episodes

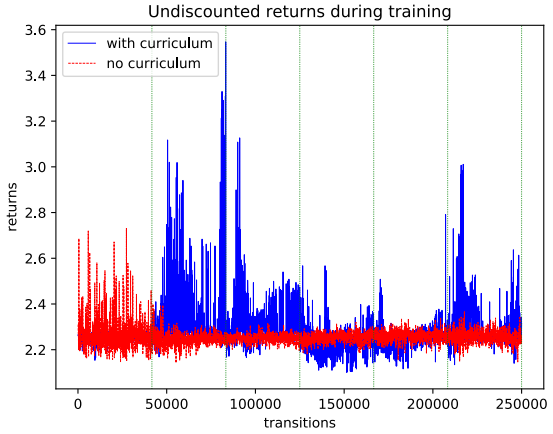


(f) Mean return at test episodes

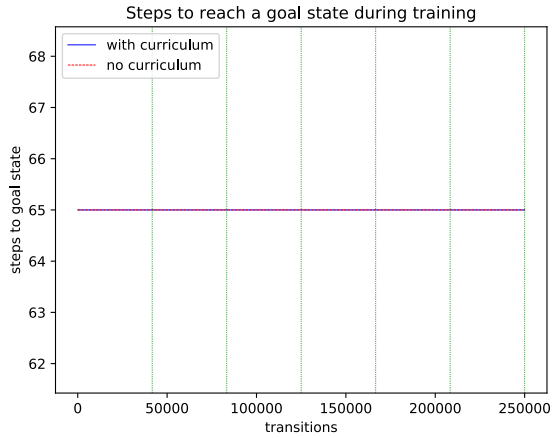
Figure 4.18: Increasing number of joints curriculum with **sparse** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.



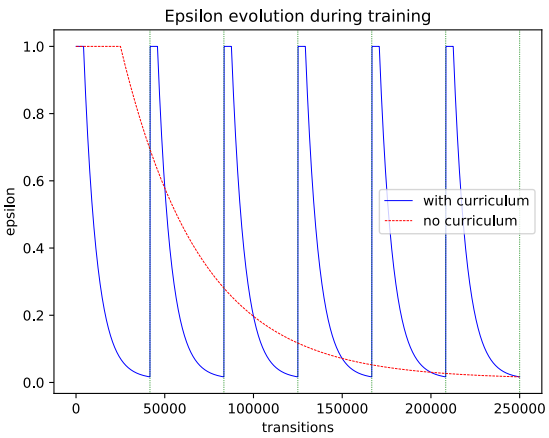
(a) Cumulative successful training episodes



(b) Undiscounted returns during training

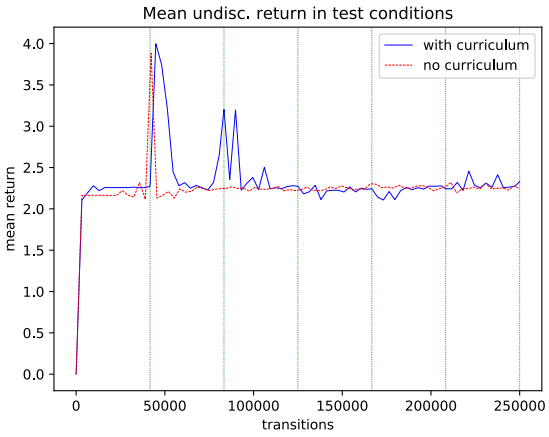


(c) Goal state reached step



(d) Epsilon decay during training

(e) Success rate at test episodes



(f) Mean return at test episodes

Figure 4.19: Increasing number of joints curriculum with **shaping** rewards performance at the pushing task. Vertical lines represent sub-task switching. Baseline without curriculum, with **sparse** rewards.

Chapter 5

Conclusion and Future Work

A few curricula where tested for two different tasks. For the reaching task, a baseline was first designed so that the agent learned a good policy without using curriculum learning.

For the reaching task, we found that a simple curriculum such as increasing the number of joints that the robot can regular intervals can improve the performance of the final agent, even compared to using shaping rewards. We also found that using shaping rewards does not necessarily improve the performance of the agent trained with curriculum learning. Other curricula gave performances similar to the baseline training without curriculum, despite the fact that the agent was trained on the target task for fewer episodes.

These results show that curriculum learning can guide early exploration by performing simpler tasks, and that the final agent is able to benefit from them. However, careful curriculum design is necessary in order to make the most out of preceding sub-tasks. Indeed, training the agent longer on the final sub-task or making the transitions between sub-tasks smoother could further improve its performance.

For the pushing task, further experiments with more steps per episode would let us evaluate the curriculum learning approach more accurately. Indeed, curriculum design requires more prior knowledge about the problem, such as the minimum number of time-steps needed to consistently reach the cube. A human operator or another alternative control method may be used to get more accurate information about the problem and optimal policies. Badly designed curricula may prevent the agent from learning the sub-task and lead to poor performances. Because of a lack of time, we were not able to train an agent to solve the pushing task before testing curriculum learning.

To conclude, curriculum learning is a method to give more prior knowledge about the problem to the learning agent, which can reduce the complexity of the problem and reduce the overall learning time. However, if it is not defined correctly, it may prevent the agent from learning a good policy.

Some possible future work could be identifying the most important traits of successful

curricula, as well as optimizing the sub-task switching schedule or the number of sub-tasks. To reduce training times, further optimization to the DQN algorithm, such as prioritized experience replay and duelling architectures could be implemented. It may also be interesting to evaluate curriculum learning with other deep reinforcement learning algorithms, such as Normalized Advantage Function (NAF) and Asynchronous Advantage Actor Critic (A3C) [Mnih et al., 2016], which are more easily adapted to continuous state spaces.

It is important to note that an important computational resources are required to evaluate and compare different curricula performances and training times required for an accurate comparison in good experimental conditions greatly limits the number of experiments we can perform.

Bibliography

[Bellman, 1952] Bellman, R. (1952). On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–9. pages 5

[Bengio et al., 2009] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1–8, New York, New York, USA. ACM Press. pages 2, 9, 10

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. pages 13

[Coppelia Robotics, 2017] Coppelia Robotics (2017). V-REP user manual. pages 13

[Elman, 1993] Elman, J. L. (1993). Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99. pages 10

[Erhan et al., 2009] Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. pages 10

[Florensa et al., 2017] Florensa, C., Held, D., Wulfmeier, M., and Abbeel, P. (2017). Reverse Curriculum Generation for Reinforcement Learning. pages 11

[Goodfellow et al., 2014] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks. pages 11

[Gu et al., 2016] Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2016). Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. pages 17, 18

[Held et al., 2017] Held, D., Geng, X., Florensa, C., and Abbeel, P. (2017). Automatic Goal Generation for Reinforcement Learning Agents. pages 11

[Hinton et al., 2006] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554. pages 10

[James and Johns, 2016a] James, S. and Johns, E. (2016a). 3D Simulated Robot Manipulation Using Deep Reinforcement Learning. Technical report. pages 10

[James and Johns, 2016b] James, S. and Johns, E. (2016b). 3D Simulation for Robot Arm Control with Deep Q-Learning. pages 10

[Levine et al., 2015] Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2015). End-to-End Training of Deep Visuomotor Policies. pages 2, 9

[Lin, 1992] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321. pages 8

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. pages 50

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. pages 7

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533. pages 2, 7, 9, 15

[Popov et al., 2017] Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. (2017). Data-efficient Deep Reinforcement Learning for Dexterous Manipulation. pages 11, 17, 18, 29

[Silver, 2015] Silver, D. (2015). Reinforcement Learning Course. pages 2

[Sukhbaatar et al., 2017] Sukhbaatar, S., Kostrikov, I., Szlam, A., and Fergus, R. (2017). Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play. pages 11

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (1998). Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054. pages 3

[Vincent et al., 2008] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 1096–1103, New York, New York, USA. ACM Press. pages 10

[Wu and Tian, 2016] Wu, Y. and Tian, Y. (2016). Training Agent for First-Person Shooter Game with Actor-Critic Curriculum Learning. pages 11

Appendix A

Code running instructions

Here we briefly describe the code files provided:

1. $MicoRobot\_last.ttt$ is the V-REP scene used for the experiments.
2. $robotenv.py$ implements an environment class (similar to OpenAI Gym environments) which acts as an interface between the following Python scripts and the V-REP remote API.
3. $dql\_algorithm.py$ implements the deep Q-learning algorithm described in Algorithm 1.
4. $curriculum.py$ implements a class with a $run()$ method that can run the algorithm multiple times, save and load the trained models to evaluate a curriculum.
5. $comparison\_plot.py$ is a script that can run multiple curricula and compare them to baselines without curriculum that may have been run previously. This script ensures that curricula and the baselines were run under equal conditions.
6. $main.py$ is a script to run a curriculum individually or test for a minimal number of episodes to detect errors before running the scripts for the total number of training episodes.
7. $visualize\_model.py$ is a script that allow to run a previously trained model for a few episodes to visualized the learned policy.
8. $trained\_models\_and\_results$ contains models trained previously.
9. RUN and $run\_script.sh$ are bash scripts used to run the process in the background and save the process output and errors to a log file. Thus, the command to run the curricula evaluation script is simply $./RUN$.
10. $vrep.py$, $vrepConst.py$, and $remoteApu.so$ are V-REP files that need to be included in order to be able to use the V-REP remote API.

Note: Debugging the $robotenv$ environment methods was very time-consuming, since the amount of information given by V-REP on its website is limited and its user base is not very big. For future students working on reinforcement learning for robotics with V-REP, you may want to consider taking a look at the *V-REP Tethering Python Wrapper* "vrepper" `https://github.com/ctmakro/vrepper`, which would have been a huge time saver if it had been found earlier.