TÉCNICO LISBOA

From rocks to walls: a machine learning approach for lunar base construction

André Tomaz de Menezes

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Prof. Alexandre José Malheiro Bernardino
Prof. Rodrigo Martins de Matos Ventura

September 2020

Acknowledgments

I would like to thank my supervisors, Alexandre Bernardino and Rodrigo Ventura, and Pedro Vicente for all the support provided during the development of this work. I would also like to acknowledge VisLab for providing the hardware used for the the presented experiments.

I want to express my gratitude to all my friends and family, specially my parents, who have always been there, helping to shape the journey that led me to the accomplishment of this thesis. Finally, I am deeply grateful to Marta Guimarães, whose unconditional support was and always is the greatest contribution for me to keep moving forward and improving as a person.

Resumo

A utilização de recursos *in-situ* é um aspecto chave para uma exploração humana eficiente de ambientes extraterrestres. Um método de baixo custo para a construção de estruturas preliminares é o empilhamento de rochas não processadas, encontradas localmente, sem recurso a argamassa. Esta tese foca-se na aprendizagem autónoma desta tarefa complexa. Abordagens anteriores recorrem a modelos previamente adquiridos, que podem ser difíceis de obter no contexto de uma missão. Em alternativa, propomos uma abordagem sem modelos e baseada em dados. O problema é abstraído para a tarefa de selecionar a posição para cada pedra, apresentada numa sequência, ser colocada sobre a estrutura atualmente construída. O objetivo é cunstruir um muro que aproxime um volume objetivo, dada a percepção tridimensional da estrutura, do próximo objeto e do objetivo. Um agente é desenvolvido para aprender esta tarefa utilizando aprendizagem por reforço. O algoritmo *Deep Q-networks* é usado, onde a *Q-network* estima um mapa de valor correspondente ao retorno esperado de colocar o objeto em cada posição de uma vista superior da estrutura. A *q-funtion* aprendida capta o objetivo e a dinâmica do ambiente. O comportamento que surge é, de certo modo, consistente com a teoria existente para esta tarefa. A política aprendida supera heurísticas criadas com conhecimento prévio da tarefa, tanto em termos de estabilidade da estrutura como de semelhança com o volume objetivo. Apesar da simplificação da tarefa, a política aprendida com esta abordagem pode ser aplicada numa situação real como o planeador de alto nível num *pipeline* de construção autónoma.

Palavras-chave: aprendizagem por reforço, utilização de recursos *in-situ*, construção autónoma, rochas naturais

Abstract

In-situ resource utilization is a key aspect for an efficient human exploration of extraterrestrial environments. A cost-effective method for the construction of preliminary structures is dry stacking with locally found unprocessed rocks, which is a challenging task. This thesis focus on learning this task from scratch. Former approaches rely on previously acquired models, which may be hard to obtain in the context of a mission. In alternative, we propose a model-free, data-driven approach. We formulate an abstraction of the problem as the task of selecting the position to place each rock, presented to the robot in a sequence, on top of the currently built structure. The goal is to assemble a wall that approximates a target volume, given the 3D perception of the currently built structure, the next object and the target volume. An agent is developed to learn this task using reinforcement learning. The Deep Q-networks algorithm is used, where the Q-network outputs a value map corresponding to the expected return of placing the object in each position of a top-view depth image. The learned q-function is able to capture the goal and dynamics of the environment. The emerged behaviour is, to some extent, consistent with dry stacking theory. The learned policy outperforms engineered heuristics, both in terms of stability of the structure and similarity with the target volume. Despite the simplification of the task, the policy learned with this approach could be applied to a realistic setting as the high level planner in an autonomous construction pipeline.

Keywords: reinforcement learning, dry stacking, *in-situ* resource utilization, autonomous construction, model-free

Contents

List of Tables

List of Figures

Nomenclature

Notation

$\mathbf{A}$	Matrix in $\mathbb{R}^{\mathrm{m}\times\mathrm{n}}$ with elements a_{ij}, or higher dimensional array.
$\mathbf{a}$	Vector in $\mathbb{R}^{\mathrm{n}}$ with elements a_i.
$\mathcal{A}$	Set.
a, A	Scalar constant.
$\lvert\mathcal{A}\rvert$	Number of elements in set $\mathcal{A}$
A	Random variable.
a	Value of a variable.
$a_{\max}, a_{\min}$	Maximum and minimum values a variable can take.
$\mathbb{E}[A]$	Expected value of random variable A.
$\Pr\{A = a\}$	Probability of random variable A taking value a.

Greek symbols

α	Exponent used for prioritization in experience replay.
β	Exponent used for prioritization bias compensation in experience replay.
δ	Temporal-difference error.
ε	Probability of taking a random action in the ε-greedy policy.
η	Learning rate.
γ	Discount factor.
π	Policy.
π_*	Optimal policy.
τ	Temperature parameter in the Boltzmann distribution.
θ	Rotation, in radians.

Roman symbols

$\mathcal{A}$	Action space.
a	An action.
A_t	Action at time step t.
$\mathcal{G}$	Goal space.
g	A goal.
G_t	Return from time step t.
$\mathbf{q}$	Unit quaternion $\left[x_0 \sin\frac{\theta}{2} \quad x_1 \sin\frac{\theta}{2} \quad x_2 \sin\frac{\theta}{2} \quad \cos\frac{\theta}{2}\right]$, representing a rotation of θ around unit axis $\mathbf{x}$.
q_π	Action-value function of policy π.
q_*	Action-value function of the optimal policy.
Q	Approximation of the action-value function.
$\mathcal{R}$	Reward space.
r	A reward.
R_t	Reward at time step t.
$\mathcal{S}$	State space.
s	A state.
S_t	State at time step t.
v_π	Sate-value function of policy π.
v_*	State-value function of the optimal policy.
V	Approximation of the state-value function.
$\mathbf{w}$	Weight vector of a neural network.
$\mathbf{x}$	Position in $\mathbb{R}^3$.

Glossary

ANN artificial neural network.

CNN convolutional neural network.

DL deep learning.

DNN deep neural network.

DQN deep Q-network.

DRL deep reinforcement learning.

FCN fully convolutional network.

GPI generalized policy iteration.

GVF general value function.

IoU intersection over union.

ISRU in-situ resource utilization.

MDP Markov decision process.

ML machine learning.

OR occupation ratio.

ReLU rectified linear unit.

RL reinforcement learning.

SGD stochastic gradient descent.

TD temporal-difference.

URDF unified robot description format.

UVFA universal value function approximator.

Chapter 1

Introduction

1.1 Motivation

For a long term human exploration of extraterrestrial environments, such as the Moon, it is essential to use native materials as replacement to resources otherwise brought from Earth, at great expense. This is commonly referred to as in-situ resource utilization (ISRU), and its applications range from the production of propellant and life support gases to the construction of planetary infrastructures [1]. This thesis focus on the later.

Initial settlement infrastructures, such as roads, platforms and shade walls, may be built with unprocessed or minimally processed local rocks using the ancient method of dry stacking [2]. Although rudimentary, this technique has been proven to produce robust and long lasting structures [3], while requiring very low pre-processing time and energy.

Due to the increased risk and limitations imposed by human missions [4], it is important to have systems with the capability of autonomously setting up the preliminary infrastructures.

However, assembling a structure from a set of irregularly shaped rocks is a complex task, usually performed by experienced humans and requiring some amount of intuition. It is not clear how to translate the solution into an autonomous system. Additionally, this system is targeted at unexplored environments, in which models of the objects and dynamics may not be available.

An increasingly common approach to autonomously learning such difficult tasks is reinforcement learning (RL). Concretely, the field of deep reinforcement learning (DRL) has seen a number of breakthroughs in recent years, such as the deep Q-network (DQN) algorithm [5] and its subsequent improvements [6].

1.2 Objectives

With this thesis, we aim to show that it is possible for an autonomous agent to learn the complex task of dry stacking with irregular rocks with a model-free approach. The agent must learn from scratch, without relying on previously acquired models of the objects and environment dynamics.

1.3 Related Work

Some previous works exploit a physics engine to plan a stable structure from a set of pre-scanned irregular rocks. Lambert and Kennedy [7] developed an application that outputs an assembly plan for a set of prism-shaped rocks, a 2-D simplification of the problem, where the pose of each rock is obtained using simulated annealing to minimise a simple heuristic based on the vertical coordinate of the rock position. Nielsen and Dancu [8] propose a system for assisting real-time dry stone wall construction, where a good placement is found in simulation by dropping the 3D model of a stone at different positions and with different orientations. The reached poses are evaluated using the distribution of contact points and dot product between the contact normals and the vertical direction. Furrer *et al.* [9] present a method to select the best next object and pose by applying gradient descent from several random initial poses to optimize a cost function that takes into account: i) the area of the support polygon, ii) the deviation between the support polygon normal and the gravity direction, iii) the kinetic energy of the structure after the placement and iv) the distance between centers of mass of next and previous objects. This approach is validated in a real world setting with a pipeline capable of constructing vertical stacks of up to four irregular rocks with a robotic manipulator. Liu *et al.* [10] propose an approach that successively reduces a finite set of stable poses, generated by the simulator, by applying a hierarchical sequence of filters based on heuristics. Concretely, the area and normal of the support polygon, the height relative to neighboring objects, the top surface sloping and interlocking (number of objects in contact with the current one). In an experimental setting, this method is shown to outperform the work of Furrer *et al.* [9] in building vertical stacks and to be able to construct walls of four courses. This was preceded by authors' previous works in a simplified 2-D setting [11, 12], including a RL approach [13] in which the DQN algorithm is used to learn the values of each stable pose, which are used instead of the sequence of heuristics.

To the best of our knowledge, there is no previous work that explores a completely model-free approach to this problem.

1.4 Contributions

We propose a formulation for the problem of autonomously building a dry stack wall with irregular 3D blocks that is compatible with a model-free RL approach. We show that this formulation provides enough information and freedom for an agent to learn the task, without relying on previously acquired object models or internal physical simulation. A deep neural network architecture is developed to learn a mapping of the state representation to action values using DQN. The implementation of our approach is openly available[1].

[1] Source code available at `https://github.com/menezesandre/stackrl`.

1.5 Thesis Outline

An introduction to the concepts used in this work, within the topics of deep learning (DL) and RL, is provided in Chapter 2. A theoretical formulation of the approach, including the RL environment and the agent to learn it, is proposed in Chapter 3 and its implementation is described in Chapter 4. The experiments performed to validate the approach are presented in Chapter 5. Finally, in Chapter 6, the advantages and shortcomings of the approach are discussed.

Chapter 2

Background

This Chapter provides an introduction to the concepts used in the development of this work. Section 2.1 defines deep neural networks and presents the relevant architectures. Section 2.2 introduces the topic of reinforcement learning and describes the used algorithms.

2.1 Deep Learning

DL is the field of machine learning (ML) that applies artificial neural networks (ANNs) with multiple hidden layers, also called deep neural networks (DNNs). The usage of a multilayered architecture enables the model to learn representations of data with increasing levels of abstraction, which makes it possible to approximate complex nonlinear functions directly from raw input (e.g. images). In opposition to traditional ML methods, where feature engineering is required to extract relevant information from raw data, the process of feature extraction is encapsulated in the model and learned according to the target output [14].

An artificial neuron is a unit conceived to model the biological neurons. It receives signals and outputs a signal in response. The output of a neuron is obtained by computing a weighted sum of its input signals and then applying a non-linear activation function, such as the rectified linear unit (ReLU) defined as $f(x) = \max(0, x)$. An ANN is, as the name suggests, a network of these neurons. The network structure is divided in input units that receive signals from the input of the network, hidden units that are only connected to other neurons (in terms of input and output) and output units, whose output signals make up the output up the network. The weights of the neurons, used for the weighted sum of input signals, are the trainable parameters of the ANN that must be adjusted in order to achieve the intended output. Figure 2.1 provides a representation of an ANN and the artificial neuron.

The key aspect of an ANN is that, as long as the non-linear activation functions are differentiable, its output is differentiable with respect to each of its weights. The gradient is computed by applying the chain rule. Let $\mathbf{w}^{[i]}$ represent the weights array of the i^{th} layer and $\mathbf{x}^{[i]}$ its output, corresponding to the input of the $(i+1)^{\text{th}}$ layer. Given a function of the network's output $f(\mathbf{x}^{[N]})$ (where N is the index of the

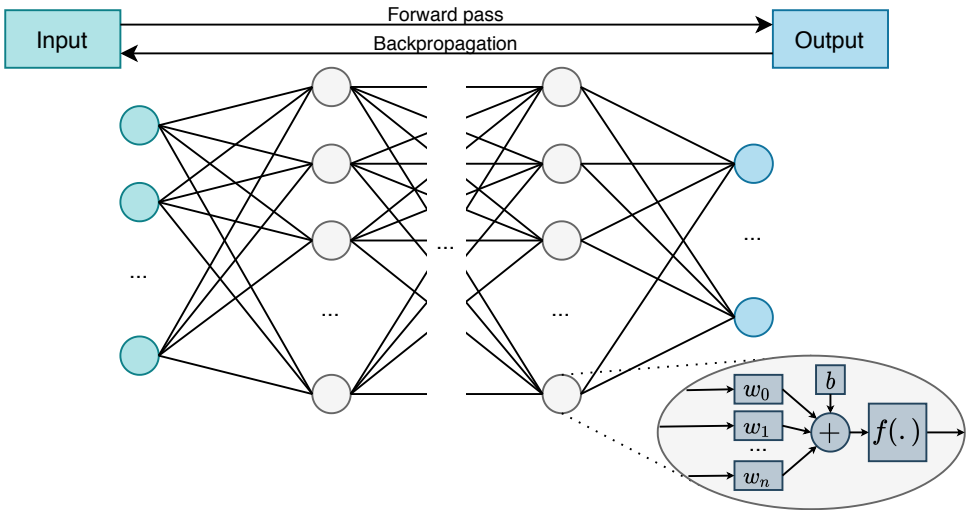


Figure 2.1: Artificial neural network, represented with turquoise input units, grey hidden units and blue output units. The expanded unit illustrates the operation performed by a neuron: a weighted sum of its inputs and a bias, followed by an activation function.

last layer) its gradient with respect to the weights of any layer is given by

$$\nabla_{\mathbf{w}^{[i]}} f(\mathbf{x}^{[N]}) = \nabla_{\mathbf{w}^{[i]}} \mathbf{x}^{[i]} \prod_{j=i}^{N-1} \left(\nabla_{\mathbf{x}^{[j]}} \mathbf{x}^{[j+1]} \right) \nabla_{\mathbf{x}^{[N]}} f(\mathbf{x}^{[N]}), \tag{2.1}$$

where $\nabla_{\mathbf{x}^{[i]}} \mathbf{x}^{[i+1]}$ is the gradient of a layer's output with respect to its input and $\nabla_{\mathbf{w}^{[i]}} \mathbf{x}^{[i]}$ with respect to its weights. With this process, named backpropagation, it is possible to obtain the gradient of the function with respect to all weights of the network.

Backpropagation makes it possible to apply gradient descent to minimize a given loss function: iteratively applying small updates to the weights in the direction opposite to the gradient, which decreases the loss, until a minimum is found. This loss is typically a metric of how much the network deviates from the desired function. This function depends on a set of examples of inputs and target outputs. The loss is calculated by applying the network to each example input and matching the output with the target. The losses for each example of the set are then combined (e.g. averaged) and the gradient of the total loss w.r.t. the weights is calculated and applied, in order to decrease the deviation between the network and the set of examples.

Instead of computing the gradient of the loss for the complete set of examples, this may be estimated using a smaller batch sampled from the set. As the batch follows the distribution of the set from which it was sampled, the expected value of the gradient is the same as for the total loss. This noisy estimates are used for stochastic gradient descent (SGD), a variation of gradient descent in which an update is performed using the gradient from each batch. This approach has been shown to provide faster convergence for large sets of examples [14].

2.1.1 Convolutional Neural Networks

A convolutional neural network (CNN) is a special case of a DNN that employs convolutional and pooling layers, which are inspired by the way the visual cortex processes signals [14]. Figure 2.2 illustrates the operations performed by these layers.

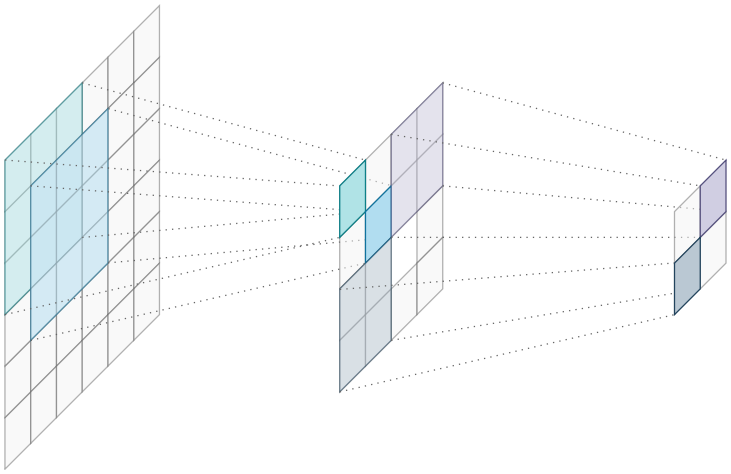


Figure 2.2: Convolutional and pooling layers. In a convolutional layer, each unit is connected to a local patch of the input (represented by turquoise and blue). The pooling layer performs an operation (e.g. $\max$) that combines a local patch (represented by grey and purple) to reduce dimensions.

A convolutional layer is characterized by local connections and shared weights. Its neurons are organized in feature maps. All neurons in a feature map share the same set of weights, and each neuron only uses signals from a local patch of the input to compute its output. In practice, each feature map is the result of convolving a learnable filter (defined by the shared set of weights) through the input of the layer. A convolutional layer outputs a stack of such feature maps, resulting in an array with the same dimensionality as the input. The layer is thus parameterized by the size of the filter (also called the kernel size), which defines the number of input signals received by each neuron, the strides, that determine the shift in the region that neighboring neurons receive as input, and the number of filters, corresponding to the number feature maps.

The convolutional nature of these layers makes them shift invariant: if a pattern in the input produces a shape of activations in a feature map, the result of shifting the pattern in the input is a feature map with the same shape of activations shifted accordingly. The usage of local connections enables the detection and extraction of local features, that are hierarchically assembled into higher level features through successive layers [14]. A convolutional layer can also be seen as a regularized version of the fully connected layer, in which the weights of the input signals outside the local patch are set to zero and the set of weights of each neuron in a feature map are restricted to be equal. This regularization enables a better generalization, while the local connections and shift invariance make this layer architecture specially suitable to extract features when the data is arranged in a meaningful way (e.g. a 2D image, where the dimensions represent space, or a 1D discrete signal, where the dimension represents time).

A pooling layer performs an operation that combines the features in a local patch. One commonly used operation is taking the maximum of each feature map over the patch. With this operation, the

semantics of the patch are captured in a single feature vector, which allows a reduction of the size of the input. These layers are also defined by a pool size, that defines the size of the region from which the features are merged, and strides (as in convolutions). A usual setting is a pooling layer with kernel size 2 and strides 2, that reduces the dimensions of the input to one half. This is done to allow subsequent convolutional layers to assemble the most meaningful features of a wider region, making the network more robust to variations in the disposition of relevant information [14].

A CNN is typically constituted by a series of convolutional layers intercalated with pooling layers, which extract the features from the input, and then a series of fully connected layers that map these features to the desired output.

Fully Convolutional Networks

A fully convolutional network (FCN) is a CNN that, in its whole, retains the shift invariance property of the convolutional layers. Such networks may take as input arrays of arbitrary size and return arrays with the same dimensionality and sized accordingly [15]. This is accomplished with an architecture that uses exclusively the convolutional and pooling layers described above, along with (optionally) upsampling layers that increase the size (and hence the resolution) of their input. An architecture with these characteristics is useful for tasks in which dense feature representations are needed (i.e. a vector of features for each region of the input, rather than one global vector of extracted features).

Long *et al.* [15] proposed a fully convolutional architecture for pixelwise classification of images (i.e. semantic segmentation). A regular CNN is converted to a FCN by replacing the final fully connected layers with additional convolutional layers. The low resolution of the output feature maps, caused by the size reductions (e.g. pooling), is addressed by upsampling the output. This is accomplished using deconvolutions, which can be seen as reversed convolutional layers, that express a learnable upsampling operation (e.g. bilinear). The network is thus constituted by a contracting path, corresponding to the usual convolutional and pooling layers, and an expanding path through the upsampling layers. The output is further refined by combining higher resolution features from layers in the contracting path with with the high level upsampled features.

Ronneberger *et al.* [16] extended this architecture by increasing the capacity of the expanding path. While the contracting path remains the sequence of convolutional and pooling layers, the expanding path is now a sequence of convolutional and upsampling layers, with the same capacity. This allows the network to propagate the information contained in the high level, low resolution features into successively higher resolution feature maps, while combining it with the features of corresponding resolution from the contracting path. This architecture has symmetric downsampling and upsampling paths, resulting in a u-shaped network and hence named U-net.

2.1.2 Siamese Networks

Siamese networks [17] are used to compute the similarity between two inputs. Each input is passed through a branch of the network that extracts a feature vector. The features from both branches are then

used to compute a similarity metric, either learned or predefined. The branches are two sub-networks with the same architecture and shared weights. These sub-networks must learn to extract the relevant features from the inputs, so that inputs of the same class are close in the feature space while different classes are distant. This architecture is useful for problems with a very large or unknown number of classes and with little to no data available from some of them [18]. While a regular network would need many examples of a class to learn to classify it, a trained Siamese network may generalize to an unknown class using a single example of that class to match with other inputs. Figure 2.3 depicts the Siamese architecture.

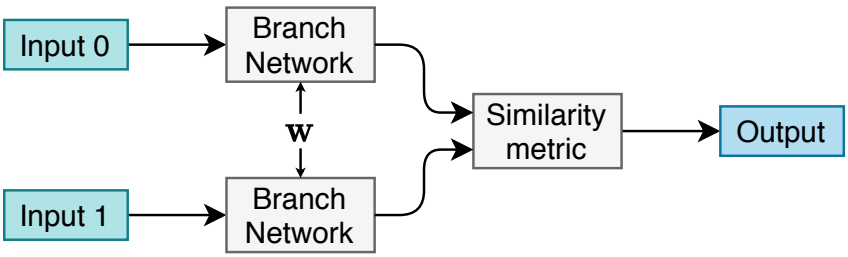


Figure 2.3: Siamese neural network. The branches are networks with shared weights and architectures, that extract features from the inputs. The features are matched with a similarity metric, either learned or predefined.

Fully Convolutional Siamese Networks

Bertinetto *et al.* [19] extended the concept of Siamese networks to the problem of tracking arbitrary objects, introducing a fully convolutional Siamese architecture. Each branch of the network is by itself a FCN, extracting feature maps from the object and from the scene where the object must be found. The feature maps are then cross-correlated, which can be visualized as sliding the feature maps of the object through the feature maps of the scene and computing a heatmap with a metric of the match for each possible offset. With this operation, not only each branch is fully convolutional but the whole network is fully convolutional with respect to the scene image (because the cross-correlation has the same effect as a convolutional layer where the filters are the feature maps of the object), meaning that a translation in the scene produces a corresponding translation in the output heatmap.

2.2 Reinforcement Learning

RL is a framework in which an agent interacts with an environment in order to learn a behaviour that maximizes a reward signal [20, Sec. 1.1]. The accumulated rewards only indicate how good a behaviour was, not whether it was the best behaviour. This evaluative feedback differs from the instructive feedback (i.e. error w.r.t. the correct output) used in supervised learning, creating the need for an active exploration to find a good behaviour. [20, Ch. 2]

2.2.1 Overview

A RL problem can be formulated as a Markov decision process (MDP). The environment is defined by the set of possible states $\mathcal{S}$, the set of available actions in a state $\mathcal{A}(s)$, the set of possible rewards $\mathcal{R} \subset \mathbb{R}$ and the dynamics given by the function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ defined as

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}, \tag{2.2}$$

which represents the transitions probability distribution. The state is what is perceived by the agent, which may be an incomplete representation of the full environment's state. For simplicity of notation, it is henceforth assumed that the set of available actions does not depend on the state and can be represented as $\mathcal{A}$. At each discrete time step t, the agent observes the environment's state $S_t \in \mathcal{S}$ and takes an action $A_t \in \mathcal{A}$ accordingly. This causes the environment to shift to state S_{t+1}, which comes with the reward $R_{t+1} \in \mathcal{R}$ [20, Sec. 3.1]. The tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ represents the transition at time step t. This process is represented in Figure 2.4.

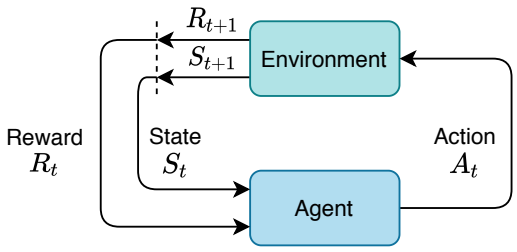


Figure 2.4: Interaction between agent and environment in a MDP

The goal of the agent is to maximize the cumulative reward. This can be formalized by defining the return G_t as a function of the reward sequence from time step t, and the goal as maximizing the expected value of G_t [20, Sec. 3.3]. The return can be simply defined as the sum of the rewards until the final time step T,

$$G_t = \sum_{k=t+1}^{T} R_k. \tag{2.3}$$

However, if there is no time limit (i.e. $T = \infty$) the sum defined in (2.3) may be divergent. For this reason, a discount factor $\gamma \in [0, 1]$ is introduced, creating the discounted return

$$G_t = \sum_{k=t+1}^{\infty} \gamma^{k-t-1} R_k \tag{2.4}$$

Given that $\mathcal{R}$ is bounded, the sum is always convergent if either $\gamma < 1$ or T is finite. If we consider $R_t = 0 \ \forall \ t > T$, this expression is general for any time limit T and is equivalent to (2.3) for $\gamma = 1$ [20, Sec. 3.4]. This representation is useful because it can be expressed recursively (in terms of the next

time step) as

$$\begin{aligned} G_t &= R_{t+1} + \gamma \sum_{k=t+2}^{\infty} \gamma^{k-t-2} R_k \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.5}$$

The way an agent acts is captured by its policy $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, defined as

$$\pi(a|s) = \Pr\{A_t = a \mid S_t = s\}, \tag{2.6}$$

which gives the probability distribution over actions for a given state [20, Sec. 3.5]. This is a stochastic representation of the behaviour. In alternative, if a policy is deterministic, it can be represented as the function $\pi : \mathcal{S} \to \mathcal{A}$ that maps directly states to actions (i.e. $a = \pi(s)$).

The expected return of following the policy π from a state s is expressed by the state-value function $v_\pi : \mathcal{S} \to \mathbb{R}$, defined as

$$v_\pi(s) = \mathbb{E}[G_t \mid S_t = s, \pi]\,. \tag{2.7}$$

Using the relation in (2.5) and the fact that, by definition, $\mathbb{E}[G_{t+1} \mid \pi] = v_\pi(S_{t+1})$, we can express the state-value in terms of the next state's value, as

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, \pi] \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(.|s)]\,. \end{aligned} \tag{2.8}$$

Similarly, we can define the value of taking an action a from a state s and then following π as the action-value function $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$,

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t \mid S_t = s, A_t = a, \pi] \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]\,. \end{aligned} \tag{2.9}$$

The optimal policy π_* is the one for which the expected return is greatest. That is, $v_*(s) = \max_\pi v_\pi(s)$ and $q_*(s, a) = \max_\pi q_\pi(s, a)$. The expected return from a state s given the optimal policy must be the same as the expected return of taking the best possible action in that state. This is expressed by

$$v_*(s) = \max_a q_*(s, a). \tag{2.10}$$

Using (2.9) (applied to q_*) and (2.10), we can define the Bellman optimality equations for the state-value and action-value functions [20, Sec. 3.6]. Respectively, replacing q_* in (2.10), we obtain

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a], \tag{2.11}$$

and replacing v_* in (2.9) gives

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right]. \tag{2.12}$$

Both equations (2.11) and (2.12) could be expressed in terms of p (2.2), which is useful for model-based RL. In that case, a model of the environment (previously known or learned) would be used to compute the optimal policy. However, these methods are out of the scope of this thesis. In opposition, in model-free RL the model is implicitly captured by fitting the value function and/or policy to the collected experience.

Most RL methods rely on some form of the generalized policy iteration (GPI) [20, Sec. 4.6]. The core idea is to alternate between fitting the value function to the expected return of the current policy and updating the current policy to maximize the value function. This includes the special cases in which the value function fits to a policy represented directly in terms of the value or the policy is updated directly using an implicit estimation of the value function. In the case of model-free learning, the expected return is estimated from the experience. Also, when a model is not available, knowledge about the state-value function alone is not useful for the policy improvement (i.e. it is of no use to know which state is best if there is no knowledge about which action leads to that state), so the action-value function must be estimated for the GPI [20, Sec. 5.2].

In order to approximate the optimal policy using the collected experience, it is necessary to perform a wide exploration of the state and action spaces. In the context of GPI, it is easy to understand that if the agent follows the deterministic greedy policy for the current action-value estimates Q

$$\pi(s) = \arg\max_a Q(s,a), \tag{2.13}$$

it may never experience potentially better state-action pairs and converge to a sub-optimal policy. For this reason, a stochastic policy $\pi(a|s) > 0 \;\forall\; a \in \mathcal{A},\; s \in \mathcal{S}$ must be used to model the agent's behaviour while learning [20, Sec. 5.4]. Some examples of such policies are the ε-greedy policy

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s,a') \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases} \tag{2.14}$$

where a random action is taken with probability ε, and the softmax (or Boltzmann) policy

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}, \tag{2.15}$$

where the current value estimates Q are used as the logits for the action probability distribution. The parameters $\varepsilon \in [0,1]$ and $\tau \in \{x \in \mathbb{R} :\; x > 0\}$ control the exploration. As these parameters approach zero, the stochastic policy becomes closer to the deterministic greedy policy.

This need for exploration raises one difficulty: the agent must learn the values of the optimal policy q_* from experience collected with the exploratory policy. The simplest approach is to use a stochastic policy

that is close to the greedy policy (e.g. ε-greedy) and improve it until a near-optimal policy is achieved. In some cases, the greedy policy of the values estimated for this near-optimal policy may even be optimal. Such approach is used in the on-policy methods, where the policy being improved is the same as the one used to collect experience. A more general approach is to learn the optimal policy independently of the behaviour that originated the experience, which is used in the off-policy methods. These methods are in general more complicated and slower to converge, but the disassociation between collecting and learning is very powerful [20, Sec. 5.5]. Besides allowing the agent to learn the optimal policy (instead of near-optimal), it adds great flexibility in terms of the origin of the experience. It is possible, for instance, to learn from experience collected by an expert or to reuse data collected with old policies.

There are two main groups of methods to learn the value from experience. In broad terms, Monte Carlo methods estimate the value of a state (or state-action pair) using the experienced returns obtained from that state (state-action pair) [20, Ch. 5]. Updates to the value estimates need experience from a complete episode (i.e. up to $t = T$) for the returns to be calculated. On the other hand, temporal-difference (TD) methods rely on the relation from (2.5) to update the value estimates using collected rewards in combination with the value estimates of subsequent states [20, Ch. 6]. This is commonly referred to as bootstrapping and makes it possible to learn from each transition, eliminating the need to collect a complete episode for each update.

2.2.2 Q-learning

The Q-learning algorithm, introduced by Watkins [21], represented a breakthrough in RL for combining off-policy and TD learning in a model-free method. This algorithm uses the Bellman optimality equation to estimate q_* directly, regardless of the behaviour that originated the transitions.

From (2.12), for a transition $(S_t, A_t, R_{t+1}, S_{t+1})$, the value of $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ can be used for the estimate $Q(S_t, A_t)$. Accordingly, the TD error for that transition can be defined as

$$\delta_t = \begin{cases} R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) & \text{if } S_{t+1} \text{ is not terminal} \\ R_{t+1} - Q(S_t, A_t) & \text{otherwise,} \end{cases} \tag{2.16}$$

where the fact that there is no return from a terminal state is contemplated by the condition.

In its original form (for which the convergence was formally proved by Watkins and Dayan [22]), Q-Learning uses a tabular representation of Q. That is, the set of current estimates $\{Q(s, a) \; \forall \; s \in \mathcal{S}, \; a \in \mathcal{A}\}$ is stored in memory. The agent interacts with the environment following an exploratory policy based on Q (e.g. ε-greedy) and, for each collected transition, updates the current estimate according to the rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \eta_t \delta_t, \tag{2.17}$$

where $\eta_t \in (0, 1]$ is the learning rate at time step t.

2.2.3 Double Q-learning

In stochastic environments, the observed transitions can be noisy, which causes the value estimates to be noisy as well. The Q-learning algorithm uses $\max_a Q(S_{t+1}, a)$ as the estimate of $v_*(S_{t+1})$. However, if Q is noisy, the $\max$ operator tends to select positive noise. As this value is used in the update of Q, the positive noise is propagated, creating an overestimation bias that can harm the performance of the algorithm. To solve this problem, van Hasselt [23] introduced the Double Q-learning algorithm.

This variation of Q-learning uses two different estimators Q^A and Q^B. At each time step, an action is taken according to an exploratory policy based on both estimates (e.g. the ε-greedy policy for $Q^A + Q^B$). The collected transition is then used to update either Q^A or Q^B (e.g randomly chosen) according to the rule (2.17). The key difference is that the TD error used for the update of Q^A is

$$\delta_t^A = \begin{cases} R_{t+1} + \gamma Q^B\left(S_{t+1}, \arg\max_a Q^A(S_{t+1}, a)\right) - Q^A(S_t, A_t) & \text{if } S_{t+1} \text{ is not terminal} \\ R_{t+1} - Q^A(S_t, A_t) & \text{otherwise.} \end{cases} \tag{2.18}$$

Similarly, δ_t^B is calculated by switching Q^A and Q^B in (2.18). The estimation the next state's optimal value, previously done with the $\max$ operation, is now separated in two phases. The estimator being updated provides the estimate of the optimal action $\arg\max_a Q^A(S_{t+1}, a)$, while the other estimator Q^B, learned from a different set of transitions, provides an unbiased evaluation of that action. In the limit, both estimators converge to q_*, while suppressing the propagation of the overestimation bias.

2.2.4 Deep Q-networks

The tabular representation of Q presented in Section 2.2.2 is very limited. For environments with very large state spaces, the tabular representation of Q would need a large amount of memory to be stored and it may be infeasible to explore the complete state space, as required for the convergence of Q. This is the case for any application in which the state is represented as a combination of sensory inputs. As an example, if the agent's input is a small 28×28 gray-scale image with byte precision (each pixel takes 256 possible values), the size of the state space would be $|\mathcal{S}| = 256^{28\times 28} \approx 10^{1888}$. To extend RL to such applications, it is necessary to represent the value as a function approximation learned from experience and that generalizes to unseen states [20, Pt. II]. The problem of function approximation and generalization is central in the field of machine learning, and any algorithm used in supervised learning could, in practice, be used for the value function approximation. In the last decade, the use of DNNs (see Section 2.1) has become widely popular due to its success in a wide range of tasks (e.g. [24]). To overcome the limitations of the Q-learning algorithm, Mnih *et al.* [5] developed an agent architecture, DQN, that combines Q-learning with function approximation using a DNN.

The introduction of function approximation in RL, and specially in off-policy TD learning [20, Sec. 11.3] is known to raise instability and divergence. Some of the identified causes for this problem are the correlations created by the sequential nature of the experience used to learn, the constant variations in the experience distribution caused by the changing policy and the correlation between the two terms of

the TD error (the current estimate $Q(S_t, A_t)$ and the target $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$) [5]. To address this issues, the DQN algorithm adds two concepts to Q-learning: a target estimator and experience replay [25].

The Q-network can be represented with $Q(s, a \,|\, \mathbf{w})$, an estimate of $q_*(s, a)$ given the network weights vector $\mathbf{w}$. The target network $Q(s, a \,|\, \mathbf{w}^-)$ is a copy of the Q-network, whose weights vector $\mathbf{w}^-$ is updated with the values of $\mathbf{w}$ every C steps and kept fixed otherwise. The TD error is now defined as

$$\delta_t = \begin{cases} R_{t+1} + \gamma \max_a Q(S_{t+1}, a \,|\, \mathbf{w}^-) - Q(S_t, A_t \,|\, \mathbf{w}) & \text{if } S_{t+1} \text{ is not terminal} \\ R_{t+1} - Q(S_t, A_t \,|\, \mathbf{w}) & \text{otherwise.} \end{cases} \tag{2.19}$$

With this formulation, each period of C steps corresponds to the simple optimization problem of minimizing the error of $Q(. \,|\, \mathbf{w})$ with respect to a fixed target.

With experience replay, the transitions are no longer immediately used for the update and discarded. At each step, the collected transition is stored in a replay memory, represented by the set of time steps $\mathcal{M} = \{i : (S_i, A_i, R_{i+1}, S_{i+1}) \in \text{memory}\}$. When the memory reaches its full capacity, the oldest transitions are replaced with the new ones. Independently of the experience collection, at each step (or n steps) a minibatch of transitions, represented by $\mathcal{B}_t = \{i : (S_i, A_i, R_{i+1}, S_{i+1}) \in \text{minibatch}\}$, is randomly sampled from the replay memory and $\mathbf{w}$ is updated according to

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \left(\frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \frac{1}{2} \delta_i^2 \right), \tag{2.20}$$

where a SGD step with mean squared error loss is used for simplicity of representation, although different loss functions (e.g. Huber [26]) and optimization algorithms (e.g. Adam [27]) could be used. The use of a random set of transitions from the replay memory eliminates the correlation caused by sequentiality and ensures that the data distribution is not affected by the changes in the current policy. Additionally, experience replay improves the sample efficiency of the algorithm (i.e. the ratio between the amount of learning and the agent-environment interactions needed). A diagram representation of the algorithm is shown in Figure 2.5.

In its implementation, the Q-network is a function $q_{\mathbf{w}} : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|}$ that, given a state, estimates the values for all actions. Although requiring a fixed set of actions, this allows the estimation for the full action space with a single forward pass.

Double DQN

The overestimation bias introduced by the $\max$ operator, discussed in Section 2.2.3, persists in DQN. To overcome this, van Hasselt *et al.* [28] applied the concept of Double Q-learning to DQN, resulting in the Double DQN algorithm. Instead of introducing an additional estimator, as in Double Q-learning, this algorithm takes advantage of the already present target network. Here, the current network provides the estimate of the optimal action and the target network evaluates this action. The TD error is then given

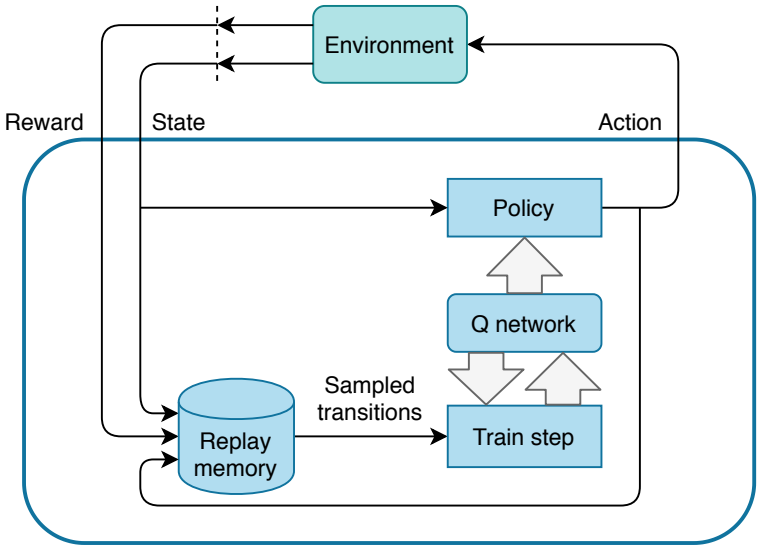


Figure 2.5: DQN algorithm.

by

$$\delta_t = \begin{cases} R_{t+1} + \gamma Q\left(S_{t+1}, \arg\max_a Q(S_{t+1}, a \,|\, \mathbf{w}) \,|\, \mathbf{w}^-\right) - Q(S_t, A_t \,|\, \mathbf{w}) & \text{if } S_{t+1} \text{ is not terminal} \\ R_{t+1} - Q(S_t, A_t \,|\, \mathbf{w}) & \text{otherwise.} \end{cases} \tag{2.21}$$

Prioritized Experience Replay

Schaul *et al.* [29] introduced the idea of prioritizing experiences that potentially contain more information to be learned, which was shown to speed up learning and improve performance. As it is not possible to directly predict how much the agent will learn from a transition, a good estimate for this is the TD error from the last time it was used in the update. The difference between the target and predicted values indicates how unexpected that transition was.

The introduction of prioritization can create two issues. First, the experience effectively used in the updates may be reduced to a small set of high error transitions, causing a loss of diversity. Second, the distribution of the data used to learn is affected by prioritization, which creates a bias in the estimation of the expected returns. The first issue is tackled by using stochastic prioritization: instead of sampling with uniform probability as in regular experience replay, the probability of transition i being sampled is given by

$$\Pr\{i \in \mathcal{B}_t\} = \frac{P_i^\alpha}{\sum_{j \in \mathcal{M}} P_j^\alpha}, \tag{2.22}$$

where the exponent α controls the prioritization ($\alpha = 0$ corresponding to uniform sampling) and the priority P_i is computed from the TD errors. In its simplest form, the priority is proportional to the absolute value of the TD error,

$$P_i = |\delta_i| + \epsilon, \tag{2.23}$$

where ϵ is a small constant that prevents the sampling probability from becoming zero (in practice, the value used for ϵ controls the minimum of the sampling probability distribution). To correct the bias in

the expected return, importance sampling [20, Sec. 5.5] is used. The importance sampling ratio is the correction from the sampling probability distribution to the real (uniform) data distribution, given by

$$W_i = \frac{\Pr\{i \in \mathcal{B}_t \mid \text{uniform}\}}{\Pr\{i \in \mathcal{B}_t \mid \text{prioritized}\}} = \frac{1}{|\mathcal{M}|} \frac{\sum_{j \in \mathcal{M}} P_j^\alpha}{P_i^\alpha}. \tag{2.24}$$

To avoid arbitrarily large steps (i.e. if the prioritized sample probability is very small), the weights are normalized to be bounded by one. Additionally, a exponent β is introduced to define how much the prioritization bias is compensated, with $\beta = 1$ corresponding to full compensation. This is used because an unbiased estimation of the expected returns is only needed near convergence, while a bias towards prioritized samples may be allowed to accelerate initial learning. Accordingly, β may be set to an initial value $\beta_0 < 1$ and annealed to one as learning progresses. The weighted update and corresponding normalized weights are given by

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \left(\frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \frac{\omega_i}{2} \delta_i^2 \right), \tag{2.25}$$

$$\omega_i = \left(\frac{W_i}{\max_j W_j} \right)^\beta = \left(\frac{\min_j P_j}{P_i} \right)^{\alpha\beta}. \tag{2.26}$$

Dueling DQN

The action-value function can be factorized into the state-value function and an advantage function [30], such that

$$q_*(s, a) = v_*(s) + a_*(s, a). \tag{2.27}$$

From (2.10),

$$\max_a a_*(s, a) = 0. \tag{2.28}$$

This factorization is motivated by the fact that the state by itself may be good or bad and sometimes the action taken has little influence on the value. With separate representations, the state-value can be learned faster, instead of implicitly learning it for each state-action pair.

Wang *et al.* [31] introduced this factorization into the DQN architecture. The value $Q(s, a \mid \mathbf{w})$ is given by $V(s \mid \mathbf{w}_0, \mathbf{w}_v)$ and $A(s, a \mid \mathbf{w}_0, \mathbf{w}_a)$, where the weights vector $\mathbf{w}_0$ is common to both estimators and $\mathbf{w} = (\mathbf{w}_0, \mathbf{w}_v, \mathbf{w}_a)$. Summing the state-value and advantage directly, as in (2.27), would not perform well with gradient descent on Q, because one value of Q may be given by infinite combinations of V and A. One way to avoid this would be to force the advantage to fulfill (2.28) by using $A(s, a \mid \mathbf{w}, \mathbf{w}_a) - \max_{a'} A(s, a' \mid \mathbf{w}, \mathbf{w}_a)$. However, this representation is very sensitive to changes in the maximum value of A. A more robust approach is to use the mean, which changes slower, giving

$$Q(s, a \mid \mathbf{w}) = V(s \mid \mathbf{w}_0, \mathbf{w}_v) + \left(A(s, a \mid \mathbf{w}_0, \mathbf{w}_a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a' \mid \mathbf{w}_0, \mathbf{w}_a) \right). \tag{2.29}$$

The resulting network architecture, named dueling architecture, can be seamlessly introduced in the DQN algorithm with any of the other improvements discussed above, and was shown to outperform the original network architecture.

2.2.5 Universal Value Functions

The transition probability distribution given by (2.2) can be factorized as

$$\begin{aligned} p(s', r \mid s, a) &= \Pr\{R_{t+1}=r \mid S_t=s, A_t=a, S_{t+1}=s'\}\Pr\{S_{t+1}=s' \mid S_t=s, A_t=a\} \\ &= p(r \mid s, a, s')p(s' \mid a, s). \end{aligned} \tag{2.30}$$

While the state-transition probability $p(s' \mid a, s)$ is purely a function of the environment's "physics", the reward distribution $p(r \mid s, a, s')$ is defined by the underlying goal. Therefore, a more general representation of the environment may be given by

$$p(s', r \mid s, a, g) = p(r \mid s, a, s', g)p(s' \mid a, s), \tag{2.31}$$

where $g \in \mathcal{G}$ represents the goal. If the reward is deterministic, then a function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{G} \to \mathbb{R}$ can be defined such that

$$p(s', r \mid s, a, g) = \begin{cases} p(s' \mid a, s) & \text{if } r = r(s, a, s', g) \\ 0 & \text{otherwise.} \end{cases} \tag{2.32}$$

Accordingly, a general value function (GVF) [32] can be defined as $v_{g,\pi}(s) = \mathbb{E}[G_t \mid S_t=s, \pi, g]$ or $q_{g,\pi}(s, a) = \mathbb{E}[G_t \mid S_t=s, A_t=a, \pi, g]$, the expected return given the goal. For an environment with a defined state-transition probability, many different GVFs may be learned according to different goals. Each goal is associated with the optimal policy π_{g*}, corresponding to value functions v_{g*} and q_{g*}.

Schaul *et al.* [33] proposed to unify the set of possible GVFs, $\{v_{g*} : g \in \mathcal{G}\}$ or $\{q_{g*} : g \in \mathcal{G}\}$, into an universal value function approximator (UVFA). The idea is to use a DNN that takes the goal as an additional input, such that $Q(s, a, g \mid \mathbf{w})$ is an estimation of $q_{g*}(s, a)$ (similarly, $V(s, g \mid \mathbf{w})$ estimates $v_{q*}(s, a)$). With this formulation, the function approximation allows to generalize not only over a potentially large state space $\mathcal{S}$, but also over a potentially large set of goals $\mathcal{G}$.

An UVFA can be learned using the DQN algorithm (Section 2.2.4) by appending the goal to the state perceived by the agent and choosing an appropriate Q-network architecture $q_{\mathbf{w}} : \mathcal{S} \times \mathcal{G} \to \mathbb{R}^{|\mathcal{A}|}$.

Chapter 3

Methodology

This work focus on the problem of building a dry stack wall using irregular objects. The proposed approach is to learn a planing policy from scratch using model-free reinforcement learning.

To achieve this, it is first necessary to capture the problem in a RL environment. The problem is formulated as a MDP (see Section 2.2.1) with discrete state and action spaces. Concretely, the state consists in elevation maps from an overhead view of the structure and a view of the new object to be placed, and the action is defined as the pixel indexes in the overhead elevation map that correspond to the new position for the object. It is also necessary to design a system of rewards to lead the agent to the intended behaviour. These topics are discussed in Section 3.1.

Then, an agent is developed to learn a suitable behaviour for the environment. The proposed agent follows the DQN algorithm to learn a value estimator that maps states to action values. This value estimator consists in a deep neural network based on the fully convolutional Siamese architecture (see Section 2.1.2), which takes as input the elevation maps of the overhead and object views and outputs a value map, containing the values of placing the object in each position. As this task differs from the usual utilization of Siamese networks (i.e. the two inputs are semantically different) a modified architecture is used, in which the branches do not share weights. The development of the agent is described in Section 3.2.

3.1 Environment

The environment must capture the problem of building a dry stack structure with irregular blocks in a target volume. In order to frame it as a learnable MDP, the problem is simplified to choosing the position for each object in a random sequence of irregular blocks. With this simplification, the agent is merely solving a planing problem, while the picking and placing are encapsulated in the environment.

At each time step, a new object from the sequence is presented to the agent, as well as the currently built structure and the target volume. The agent must choose a position $\mathbf{x} \in \mathbb{R}^3$ to place the new object, with freedom restricted to the horizontal coordinates x_0 and x_1. The object's vertical coordinate x_2 is defined so that the object is laid on top of the current structure, and with the same orientation as it was

presented. To simulate the placement, the velocity of the object is artificially controlled until a support polygon is achieved (at least three contact points). The simulation is then run freely until all objects stabilize. This process is repeated until all objects in the sequence were placed. Figure 3.1 shows a visualization of the environment.

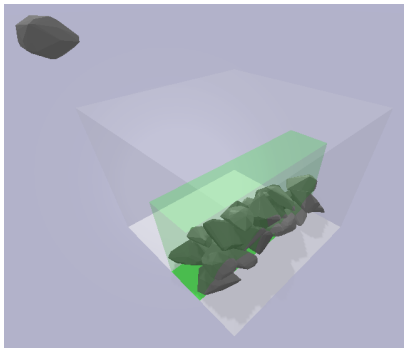


Figure 3.1: Visualization of the environment. The white box represents the region where the agent can observe the currently built structure, which corresponds to the region where placements are allowed. The green box represents the target volume. The new object is presented at the top left corner of the image.

The concept of elevation maps, which are used as the state representation, is defined in Section 3.1.1. A formal definition of the environment as a MDP is provided in Section 3.1.2 and the reward shaping is discussed in Section 3.1.3. In Section 3.1.4 some extensions to the formulation of the state and action spaces are proposed to increase the flexibility provided to the agent.

3.1.1 Elevation maps

In order to define the state space of the environment, it is first necessary to define elevation maps. An elevation map is a discretized and truncated representation of a function $h : \mathbb{R}^2 \to \mathbb{R}$ that defines a surface $\left\{\mathbf{x} \in \mathbb{R}^3 : x_2 = h(x_0, x_1)\right\}$. The horizontal coordinates (x_0 and x_1) are discretized into a $\mathrm{m} \times \mathrm{n}$ grid with cell size $\mathrm{d} \in \mathbb{R}$, such that an horizontal position $[x_0 \quad x_1]$ is represented by the cell with indexes (i, j) given by

$$\begin{aligned} i &= \mathrm{trunc}(\frac{x_0}{\mathrm{d}}), \quad 0 \leq x_0 < \mathrm{d} \cdot \mathrm{m} \\ j &= \mathrm{trunc}(\frac{x_1}{\mathrm{d}}), \quad 0 \leq x_1 < \mathrm{d} \cdot \mathrm{n}, \end{aligned} \tag{3.1}$$

where the trunc operation removes the decimal part of a number. An elevation map $\mathbf{H} \in \mathbb{R}^{\mathrm{m} \times \mathrm{n}}$ is an array whose elements represent the value of the function h in each cell. For simplicity of notation, this is represented as the value in the vertices of the grid $[x_0 \quad x_1] = [i \cdot \mathrm{d} \quad j \cdot \mathrm{d}]$. Each element of $\mathbf{H}$ is then given by

$$h_{ij} = h(i \cdot \mathrm{d}, j \cdot \mathrm{d}). \tag{3.2}$$

These elements are henceforth referred to as the pixels of the elevation map, and d as the pixel size.

With this definition, a point at the origin ($\mathbf{x} = [0 \quad 0 \quad 0]$) is represented in the elevation map by the pixel h_{00} with zero elevation (i.e. $h_{00} = 0$). The dimensions of the array $\mathbf{H}$ are aligned with the horizontal

axis of the coordinate system. That is, two points with the same x_0 are represented in the same column of $\mathbf{H}$ and two points with the same x_1 in the same row. This coordinate system is given by a 3D frame that is defined by the view of the elevation map (see Figure 4.2(b) for a representation of the frame in the view). This frame, in $\mathbb{R}^3$, is henceforth referred to as $\mathrm{F}_{\mathbf{H}}$. A point defined with respect to the frame $\mathrm{F}_{\mathbf{H}}$ is denoted as $\mathbf{x}^{\mathbf{H}}$.

An elevation map $\mathbf{H}$ is then defined by its dimensions $\mathrm{m} \times \mathrm{n}$, its pixel size d and its view, given by the transformation between a given base frame and $\mathrm{F}_{\mathbf{H}}$.

Two elevation maps $\mathbf{H}$ and $\mathbf{H}'$, with the same pixel size d, are said to have a correspondence if their views are such that the frames $\mathrm{F}_{\mathbf{H}}$ and $\mathrm{F}_{\mathbf{H}'}$ have the horizontal axis with the same orientation. In practice, this means that any $\Delta\mathbf{x}$ in the 3D space is represented by the same interval of pixels in both maps. With this condition assured, two elevation maps can be matched directly by overlapping them, knowing that a feature represented in a given interval of pixels in one map would correspond to the same interval in the other.

Let

$$\mathcal{E}_{\mathrm{m,n,h_{max}}} = \left\{\mathbf{H} \in \mathbb{R}^{\mathrm{m}\times\mathrm{n}} : h_{ij} = \min\left(\max\left(h(i\cdot\mathrm{d}, j\cdot\mathrm{d}), 0\right), \mathrm{h_{max}}\right)\right\} \tag{3.3}$$

represent the set of elevation maps of size $\mathrm{m} \times \mathrm{n}$ that capture a truncated version of the function h, such that the values are clipped between 0 and $\mathrm{h_{max}}$. This clipping can be seen as setting a maximum and minimum visible depth. For a clipped elevation map $\mathbf{H} \in \mathcal{E}_{\mathrm{m,n,h_{max}}}$, the set of indexes of its pixels is

$$\mathcal{P}_{\mathbf{H}} = \{0, 1, \ldots, \mathrm{m}-1\} \times \{0, 1, \ldots, \mathrm{n}-1\} \tag{3.4}$$

and the set of indexes of the non-empty pixels is defined as

$$\mathcal{F}_{\mathbf{H}} = \{(i,j) \in \mathcal{P}_{\mathbf{H}} : h_{ij} > 0\}. \tag{3.5}$$

3.1.2 Environment formulation

Physically, the environment is constituted by: a static ground plane $\{\mathbf{x} \in \mathbb{R}^3 : x_2 = 0\}$, with $\mathbf{x}$ defined in the base frame of the environment; the complete set of objects $\mathcal{O}$, from which T [1] objects are sampled at the beginning of each episode to be sequentially introduced in the world at position $\mathbf{x}_{\mathrm{new}}$ and with orientation given by the unit quaternion $\mathbf{q}_{\mathrm{new}} = [0 \quad 0 \quad 0 \quad 1]$[2]; and the force of gravity $\mathbf{g} = [0 \quad 0 \quad -\mathrm{g}]$, applied to all bodies at every instant. The state-transition probability $p(s' \mid a, s)$ (probability of reaching state s' after performing action a in state s) is implicitly given by a physics engine. The reward distribution $p(r \mid s, a, s', g)$ (probability of receiving reward r when transitioning from state s to s' with action a, given the goal g) is discussed in Section 3.1.3.

The state space $\mathcal{S} = \mathcal{E}_{\mathrm{m_O,n_O,h_O}} \times \mathcal{E}_{\mathrm{m_N,n_N,h_N}}$ is the combination of the sets of possible clipped elevation maps of the overhead view O and the new object view N. A state can be represented as the tuple $s = (\mathbf{O}, \mathbf{N})$. The vies of the overhead elevation map $\mathbf{O} \in \mathcal{E}_{\mathrm{m_O,n_O,h_O}}$ is aligned with the base frame of

[1] Here T is used instead of T as in Section 2.2 to denote that it is a defined constant, rather than a random variable.
[2] The convention adopted is that the scalar part of the quaternion is its last element.

the environment (i.e. the transform from the base frame to $\mathrm{F}_{\mathbf{O}}$ is the identity, $\mathbf{x}^{\mathbf{O}} = \mathbf{x}\ \forall\ \mathbf{x} \in \mathbb{R}^3$) and represents a discrete view of the current structure (see Figure 3.3), capturing the volume

$$\mathcal{V}_{\mathbf{O}} = \left\{\mathbf{x} \in \mathbb{R}^3 : 0 \leq x_0 < \mathrm{d} \cdot \mathrm{m}_{\mathbf{O}},\ 0 \leq x_1 < \mathrm{d} \cdot \mathrm{n}_{\mathbf{O}},\ 0 \leq x_2 \leq \mathrm{h}_{\mathbf{O}}\right\}. \tag{3.6}$$

The new object's elevation map $\mathbf{N} \in \mathcal{E}_{\mathrm{m}_{\mathbf{N}},\mathrm{n}_{\mathbf{N}},\mathrm{h}_{\mathbf{N}}}$ captures the volume centered in $\mathbf{x}_{\mathrm{new}}$

$$\mathcal{V}_{\mathbf{N}} = \left\{\mathbf{x}_{\mathrm{new}} + \mathbf{x} \in \mathbb{R}^3 : -\frac{\mathrm{d} \cdot \mathrm{m}_{\mathbf{N}}}{2} \leq x_0 < \frac{\mathrm{d} \cdot \mathrm{m}_{\mathbf{N}}}{2},\ -\frac{\mathrm{d} \cdot \mathrm{n}_{\mathbf{N}}}{2} \leq x_1 < \frac{\mathrm{d} \cdot \mathrm{n}_{\mathbf{N}}}{2},\ -\frac{\mathrm{h}_{\mathbf{N}}}{2} \leq x_2 \leq \frac{\mathrm{h}_{\mathbf{N}}}{2}\right\}. \tag{3.7}$$

The dimensions $\mathrm{m}_{\mathbf{N}} \times \mathrm{n}_{\mathbf{N}}$ and $\mathrm{h}_{\mathbf{N}}$ are set so that every object in $\mathcal{O}$ fits in $\mathcal{V}_{\mathbf{N}}$. The position $\mathbf{x}_{\mathrm{new}}$ is set so that $\mathcal{V}_{\mathbf{O}} \cup \mathcal{V}_{\mathbf{N}} = \emptyset$. The elevation map $\mathbf{N}$ represents a bottom view of the object (i.e. of the side that will be in contact with the structure, see Figure 3.3), which means the vertical axis of $\mathrm{F}_{\mathbf{N}}$ has the opposite direction of the vertical axis of the base frame. To keep the correspondence between $\mathbf{O}$ and $\mathbf{N}$, the horizontal axes of $\mathrm{F}_{\mathbf{N}}$ must have the same direction as the base frame (Figure 3.2 illustrates the horizontal axes with the same direction and the vertical axis with the opposite). This means that $\mathrm{F}_{\mathbf{N}}$ is not a right-handed coordinate system, and the transformation from the environment base frame is given by

$$\mathbf{x}^{\mathbf{N}} = (\mathbf{x} - \mathbf{x}_{\mathbf{N}}) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \tag{3.8}$$

where $\mathbf{x}_{\mathbf{N}} = \mathbf{x}_{\mathrm{new}} - \begin{bmatrix} \frac{\mathrm{d} \cdot \mathrm{m}_{\mathbf{N}}}{2} & \frac{\mathrm{d} \cdot \mathrm{n}_{\mathbf{N}}}{2} & -\frac{\mathrm{h}_{\mathbf{N}}}{2} \end{bmatrix}$ is the position of the origin of $\mathbf{N}$ in the base frame.

The action space $\mathcal{A} = \{0, 1, \ldots, \mathrm{m}_{\mathbf{O}} - \mathrm{m}_{\mathbf{N}}\} \times \{0, 1, \ldots, \mathrm{n}_{\mathbf{O}} - \mathrm{n}_{\mathbf{N}}\}$ is a discretization of the set of possible positions for the new object inside $\mathcal{V}_{\mathbf{O}}$, with freedom restricted to the horizontal coordinates x_0 and x_1. An action is represented by the tuple $a = (i, j)$ and can be visualized as overlapping the elevation maps $\mathbf{N}$ with $\mathbf{O}$ with an offset of i rows and j columns (see Figure 3.3(a)). The position given by an action tuple is obtained by computing the corresponding position of the origin of $\mathrm{F}_{\mathbf{N}}$ with respect to $\mathrm{F}_{\mathbf{O}}$ and then transforming the position of the object with respect to $\mathrm{F}_{\mathbf{N}}$ to $\mathrm{F}_{\mathbf{O}}$ (which coincides with the base frame). According to the proposed visualization, the horizontal coordinates of the origin of $\mathrm{F}_{\mathbf{N}}$ in $\mathrm{F}_{\mathbf{O}}$ are obtained by multiplying the pixel indexes (given by the action tuple) with the pixel size d. The vertical coordinate is set so that the object is placed on top of the current structure. Given an action $a = (i, j)$, the position of $\mathbf{N}$ in the base frame is

$$\mathbf{x}_{\mathbf{N}} = \begin{bmatrix} i \cdot \mathrm{d} & j \cdot \mathrm{d} & \max\limits_{k,l \in \mathcal{F}_{\mathbf{N}}} (o_{i+k,j+l} + n_{kl}) \end{bmatrix}, \tag{3.9}$$

where the third component is obtained with the process illustrated in Figure 3.2, to compute the vertical distance between $\mathbf{N}$ and $\mathbf{O}$ at which the surfaces are in contact. The position of the object with respect to $\mathrm{F}_{\mathbf{N}}$ is $\mathbf{x}^{\mathbf{N}}_{\mathrm{place}} = \begin{bmatrix} \frac{\mathrm{d} \cdot \mathrm{m}_{\mathbf{N}}}{2} & \frac{\mathrm{d} \cdot \mathrm{n}_{\mathbf{N}}}{2} & \frac{\mathrm{h}_{\mathbf{N}}}{2} \end{bmatrix}$. Using the inverse of the transformation in (3.8) and replacing $\mathbf{x}_{\mathbf{N}}$ with the value obtained from (3.9), the position of the object in the base frame is

$$\mathbf{x}_{\mathrm{place}} = \begin{bmatrix} \mathrm{d}\left(i + \frac{\mathrm{m}_{\mathbf{N}}}{2}\right) & \mathrm{d}\left(j + \frac{\mathrm{n}_{\mathbf{N}}}{2}\right) & \max\limits_{k,l \in \mathcal{F}_{\mathbf{N}}} (o_{i+k,j+l} + n_{kl}) - \frac{\mathrm{h}_{\mathbf{N}}}{2} \end{bmatrix}. \tag{3.10}$$

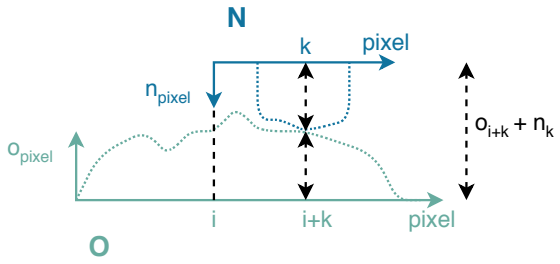
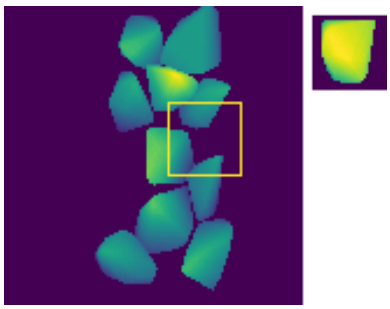


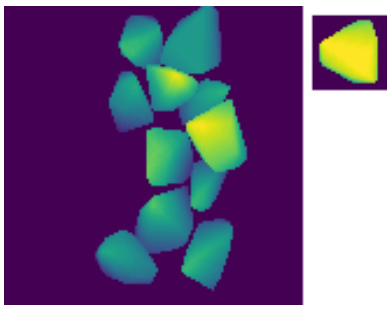
Figure 3.2: Process of obtaining the vertical distance between the object and overhead elevation maps. A simplified 1D version of the elevation maps is used for illustration. The horizontal axis represents the pixel index and the vertical axis the corresponding elevation value. $\mathbf{N}$ is positioned over $\mathbf{O}$ with an offset of i pixels. The vertical distance between $\mathbf{O}$ and $\mathbf{N}$ at which the surfaces are in contact is given by $o_{i+k} + n_k$. The pixel k is the non-empty pixel of $\mathbf{N}$ for which the sum of the elevation values is maximum.

The goal space $\mathcal{G} = \mathcal{E}_{\mathrm{m_O,n_O,h_O}}$ is the set of possible goal elevation maps. A goal elevation map $\mathbf{G} \in \mathcal{G}$ shares the view of $\mathbf{O}$ ($\mathbf{x}^{\mathbf{G}} = \mathbf{x}^{\mathbf{O}}$) but represents a virtual target structure (see Figure 3.3(c)), such that the goal of the environment is to assemble a real structure that approximates the target. This goal is formalized in Section 3.1.3.

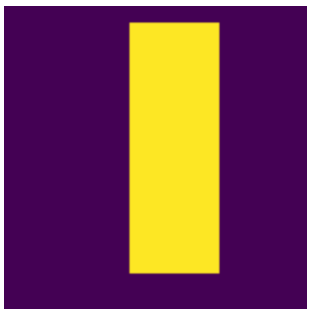
An example of a state-transition (s, a, s') is represented in Figure 3.3, along with the corresponding goal g. A visualization of the environment at the same consecutive time steps is provided in Figure 3.4.



(a) State s (left: $\mathbf{O}$; right: $\mathbf{N}$) with a visualization of action a (corresponding to the pixel indexes of the top left corner of the yellow box).



(b) State s' (left: $\mathbf{O}$; right: $\mathbf{N}$).

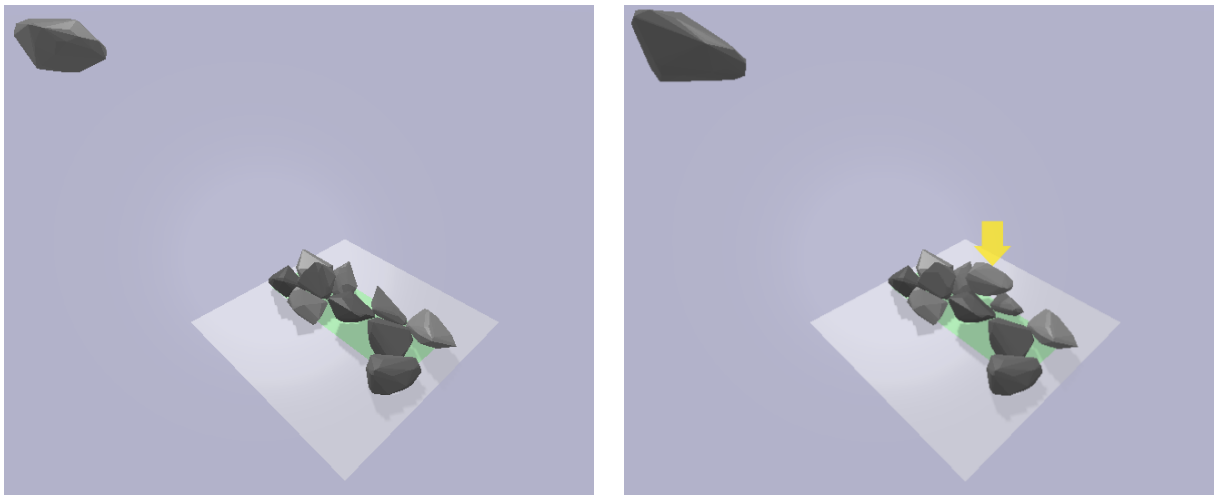


(c) Goal $g = \mathbf{G}$.

Figure 3.3: Visualization of a state-transition (s, a, s') and goal g.

3.1.3 Reward Shaping

Reward shaping is a key aspect of RL, as the learned optimal behaviours are critically affected by the way rewards are distributed. Firstly, it must capture the intended goal in a clear way, that leaves little room for the agent to "cheat" the environment. Secondly, it should be distributed in a way that facilitates the learning process towards the desired behaviour. The reward shape is a trade-off between these two factors. A sparse reward sent when a goal is achieved is very clear. However, if the goal is hard to achieve, it may be difficult for the agent to learn how to behave until a reward is received. On the other hand, rewarding behaviours that take the agent towards the goal may accelerate learning, but if not carefully shaped it may also lead to unintended behaviour that exploits the rewards in an unexpected way.



(a) Environment at time step t ($S_t = s$).

(b) Environment at time step $t+1$ ($S_{t+1} = s$).

Figure 3.4: Visualization of the environment at two consecutive time steps. The next object is presented in $\mathbf{x}_{\text{new}}$, at the top left corner. The white area marks the region covered by the overhead view. The green area marks the region occupied by the target volume. The marker in the right image points to the newly placed object.

As stated before, the goal of the environment is to assemble the objects in a way that approximates the target volume. This goal may be formalized in two ways: either obtaining a structure that matches the target (i.e. $o_{ij} \approx g_{ij} \ \forall \ i,j \in \mathcal{P}_{\mathbf{O}}$) or the more relaxed goal of filling the target (i.e. $o_{ij} \geq g_{ij} \ \forall \ i,j \in \mathcal{F}_{\mathbf{G}}$). If the number of objects is such that $\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} o_{ij} \leq \sum_{i,j \in \mathcal{P}_{\mathbf{O}}} g_{ij}$ at any time, the solution that maximizes both goals is the same and corresponds to maximizing the intersection between the current and target volumes. The formulations differ because the first explicitly penalizes objects that are not directly employed to reach the target. In either case, the goal is hard to reach, or even impossible depending on the settings. In order to generate rewards, a metric of proximity to the goal must be defined. With such a metric, each episode may yield a return corresponding to how well the goal was approximated.

The similarity between the current and target structures may be evaluated using the intersection over union (IoU) of the two volumes. This may be computed from the elevation maps $\mathbf{O}$ and $\mathbf{G}$ as

$$\text{IoU} = \frac{\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} \min\left(o_{ij}, g_{ij}\right)}{\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} \max\left(o_{ij}, g_{ij}\right)}. \tag{3.11}$$

The occupation of the target volume is given by the occupation ratio (OR), defined in terms of $\mathbf{O}$ and $\mathbf{G}$ as

$$\text{OR} = \frac{\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} \min\left(o_{ij}, g_{ij}\right)}{\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} g_{ij}}. \tag{3.12}$$

Both these metrics lay in the range $[0, 1]$, with 1 corresponding to a fully achieved goal. The values of the metrics at time step t (i.e. computed with the elevation maps $\mathbf{O}$ and $\mathbf{G}$ observed at t) are denoted as IoU_t and OR_t. The value of each of the metrics at the terminal state, IoU_{T} or OR_{T}, evaluates how close the final state is to each of the formulated goals (approximating or filling the target, respectively).

Although these metrics capture the goal of occupying the target volume, they ignore an implicit aspect

of the goal statement: the structure should be a stable assembly of the irregular objects. An example of an undesired behaviour that would be rewarded is to place the objects in a way that makes it likely for them to fall inside the target, rather than not falling. A way to avoid this is to discount the contribution of each object to the reward according to the distance between its original pose, where it was placed, and its current pose. This way, an object that falls off from its place no longer contributes to the reward, even if it falls inside the target. The choice of only discounting the reward contribution, rather than applying a separate penalty, avoids the need to tune the weights of positive and negative contributions. A poorly tuned reward could, for example, lead the agent to divert from the intended behaviour just to avoid the potential penalties.

It is not trivial to distinguish the contribution of each object to the elevation map $\mathbf{O}$, used to compute the IoU and the OR. However, it is not necessary to use those specific quantities. A metric that can be used in replacement of the intersection between structures ($\sum_{i,j \in \mathcal{P}_\mathbf{O}} \min(o_{ij}, g_{ij})$) is the number of objects inside the target. Intuitively, a greater volume of the intersection is obtained when a larger number of objects is inside the target. These quantities are not proportional: the volume of the objects is variable, while the replacement metric considers equal contributions, and the volume of the intersection includes empty space beneath objects (and consequently is dependent on how the objects are disposed), which is ignored in the replacement. Despite these facts, the metrics are expected to be highly correlated. The number of objects inside the target at time step t can be defined as

$$\mathsf{I}_t = \sum_{i=0}^{t-1} b_t^{[i]}, \tag{3.13}$$

where

$$b_t^{[i]} = \begin{cases} 1 & \text{if center of mass of object } i \text{ is inside target at time step } t \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

The superscript $[i]$ indicates that the contribution refers to the object presented at time step i, such that the first state in which that object is in the structure is at time step $i+1$. To completely define metrics equivalent to the IoU and OR, it is also necessary to define approximations to the union of the volumes and to the target volume. Taking the episode length T as the number of objects that would ideally be inside the target by the end of the episode, this value can be used as the replacement for the target volume. Accordingly, the current number of objects, given by the time step index t, may be used as the replacement for the current volume. The union is then given by the property $|\mathcal{X} \cup \mathcal{Y}| = |\mathcal{X}| + |\mathcal{Y}| - |\mathcal{X} \cap \mathcal{Y}|$. The metric that replaces the IoU is then defined as

$$\mathsf{IoU}_t^{\sim} = \frac{\sum_{i=0}^{t-1} b_t^{[i]}}{\mathrm{T} + t - \sum_{i=0}^{t-1} b_t^{[i]}}, \tag{3.15}$$

and, for the OR,

$$\mathsf{DOR}_t^{\sim} = \frac{\sum_{i=0}^{t-1} b_t^{[i]}}{\mathrm{T}}. \tag{3.16}$$

With this metrics, it is trivial to apply a discount to each object's contribution. This discount can be

defined as

$$\mu_t^{[i]} = \max\left(0, 1 - \left(\frac{|\Delta \mathbf{x}_t^{[i]}|}{\Delta \mathrm{x}_{\max}}\right)^{\mathrm{c_x}}\right) \max\left(0, 1 - \left(\frac{|\Delta \theta_t^{[i]}|}{\Delta \theta_{\max}}\right)^{\mathrm{c}_\theta}\right), \tag{3.17}$$

where $\Delta \mathbf{x}_t^{[i]}$ and $\Delta \theta_t^{[i]}$ are the translation and rotation distances between the original pose of object i and its pose observed at time step t. The parameters $\Delta \mathrm{x}_{\max}$ and $\Delta \theta_{\max}$ represent the maximum allowable distances. If the object exceeds one (or both) of these distances, its contribution to the reward is zero. The exponents $\mathrm{c_x}$ and c_θ control how μ decreases with the distances. Higher values ($\mathrm{c_x}, \mathrm{c}_\theta > 1$) make it less sensitive to small displacements. The discounted metrics can then be defined as

$$\mathsf{DIoU}_t = \frac{\sum_{i=0}^{t-1} b_t^{[i]} \cdot \mu_t^{[i]}}{\mathrm{T} + t - \sum_{i=0}^{t-1} b_t^{[i]}}, \tag{3.18}$$

$$\mathsf{DOR}_t = \frac{\sum_{i=0}^{t-1} b_t^{[i]} \cdot \mu_t^{[i]}}{\mathrm{T}}. \tag{3.19}$$

The range of these metrics is also $[0, 1]$, with the value 1 only reachable in the terminal state of a perfect episode (i.e. all objects inside target and with no displacements from their original poses).

Any of the discussed metrics may be used to distribute rewards at each step by immediately rewarding each contribution to the final value. This is accomplished using

$$R_t = \begin{cases} M_t - M_{t-1} & \text{if } 0 < t \leq \mathrm{T} \\ 0 & \text{if } t = 0, \end{cases} \tag{3.20}$$

where M is one of the metrics (IoU, OR, DIoU or DOR). This way, an action that increases the metric (e.g. successfully places an object inside the target) is immediately rewarded, while an action that makes it decrease (e.g. causes part of the existing structure to collapse) is penalised. At the same time, the undiscounted episode return (2.3) is still the final value of the metric:

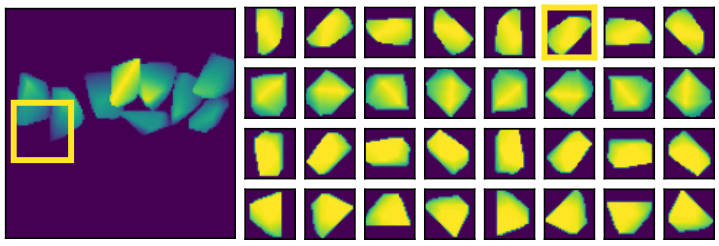
$$G_0 = \sum_{t=1}^{\mathrm{T}} R_t = \sum_{t=1}^{\mathrm{T}} M_t - \sum_{t=0}^{T-1} M_t = M_{\mathrm{T}} - M_0 = M_{\mathrm{T}}. \tag{3.21}$$

The four metrics produce rewards that contain different amounts of information: the OR exclusively rewards increases in target occupation; the IoU additionally penalizes improper use of the objects (i.e. placing outside of the target), while the DOR penalizes poor placements; the DIoU includes all of the previous information. While more informative rewards should help an agent to learn faster, rewards that contain too much information may be hard to relate with the observed events and consequently hinder learning.

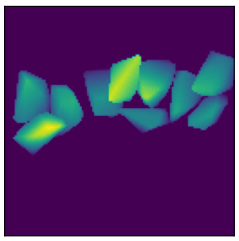
3.1.4 Extensions to state and action spaces

Two modifications can be easily made to the formulation of the state and action spaces in order to increase the freedom of the agent. One is to provide a set of different orientations for the object to be placed in. The second is to present more than one object at a time (in the limit, presenting the whole list

of objects to be placed) so that the agent may choose which one to place next. Both these extensions are accomplished by producing a set of elevation maps like $\mathbf{N}$, from different objects and orientations, and sending them to the agent as a batch of images. The action then includes the batch index of the image that corresponds to the object and orientation to be used. An example of a transition under both these extensions is presented in Figure 3.5.



(a) State s (left: $\mathbf{O}$; right: batch of images like $\mathbf{N}$) with a visualization of action a (corresponding to the pixel indexes of the top left corner of the yellow box in $\mathbf{O}$ and the batch index of the image with the yellow frame).



(b) Overhead view $\mathbf{O}$ in state s'.

Figure 3.5: Visualization of a state transition (s, a, s') of the modified environment. The batch of images like $\mathbf{N}$ presents four possible objects (rows) and eight possible orientations (columns), corresponding to different rotations around the vertical axis.

This formulation is not as practical to be learned with the proposed approach, because one action choice requires an evaluation for each of the object images in the batch. However, a policy learned for the original formulation may be applied to this extensions with a greedy approach. This is done by separately estimating the value of the best position for each object/orientation and then selecting the one that presents the highest value over the batch. This is expected to work better for the choice of orientation, because it does not change the meaning of the action: it is still a pose choice, just with added freedom. With the choice of the next object, the planning task is not only a sequence of pose choices but also the choice of the ordering for the objects. An agent that was not trained for this is likely to end up choosing the easiest objects first, ignoring the fact that the remaining will have to be used later.

3.2 Agent

The agent takes as input a tuple of arrays $(\mathbf{M}, \mathbf{N})$, where $\mathbf{M}$ is a $\mathrm{m}_\mathbf{O} \times \mathrm{n}_\mathbf{O} \times 2$ array containing the overhead elevation maps of the current and target structures ($\mathbf{O}$ and $\mathbf{G}$ respectively) with elements given by

$$m_{ijk} = \begin{cases} o_{ij} & \text{if } k = 0 \\ g_{ij} & \text{if } k = 1, \end{cases} \tag{3.22}$$

and $\mathbf{N}$ is the $\mathrm{m}_\mathbf{N} \times \mathrm{n}_\mathbf{N}$ elevation map with the bottom view of the new object.

The action to be returned, a tuple with the discretized horizontal coordinates, can be visualized by overlapping the object view on the overhead view with the action tuple as offset. From this observation, a natural candidate to map the state to action values is a cross-correlation like operation that slides $\mathbf{N}$

through $\mathbf{O}$ and $\mathbf{G}$, outputting a value for each possible offset. Such an operation provides the intended codomain and captures the underlying spatial relations in the observed state. The sets

$$\begin{aligned} \mathcal{X} &= \mathbb{R}^{\mathrm{m_O} \times \mathrm{n_O}} \times \mathbb{R}^{\mathrm{m_N} \times \mathrm{n_N}}, \\ \mathcal{Y} &= \mathbb{R}^{(\mathrm{m_O} - \mathrm{m_N}+1) \times (\mathrm{n_O} - \mathrm{n_N}+1)}, \end{aligned} \tag{3.23}$$

are defined so that a function performing an operation of this kind can be expressed as $f : \mathcal{X} \to \mathcal{Y}$. Figure 3.6 shows the process of choosing an action from the state, where the value estimator would employ these functions.

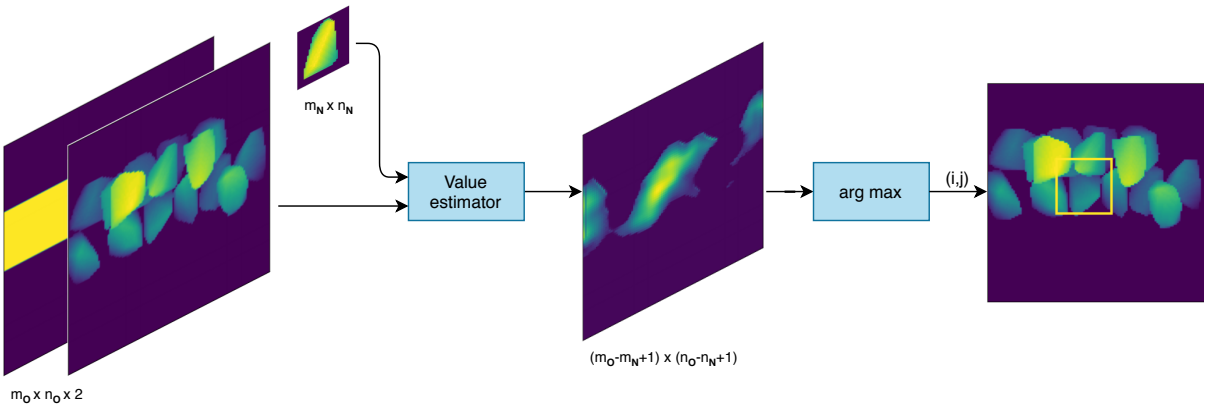


Figure 3.6: Process of choosing an action from the state. The image in the right is a visualization of the action, where $a = (i, j)$ gives the pixel indexes of the top left corner of the yellow box.

This idea is explored in Section 3.2.1, where different heuristics are formulated to match this format, and extended in Section 3.2.2 with the usage of DNNs.

3.2.1 Heuristics

The heuristics used in related works (see Section 1.3) frequently include the height of the position, so that the object is as low as possible, and an evaluation of the support polygon, to maximize its area and the dot product between its normal and the direction of gravity. The heuristics used in this work are inspired by these two criteria. As it is not possible to directly evaluate the support polygon, this is here approximated by an evaluation of the match between the surfaces of the object and structure. Additionally, with this formulation, the provided target volume must be taken into consideration. This section starts by presenting two heuristics based on the height criterion (cross-correlation and height) and then two heuristics based on the support criterion (difference and correlation coefficients). These heuristics are formulated as cost functions, such that the best positions according to each heuristic are the minima of the functions. Finally, the target criterion is presented, to be applied on top of each of the heuristics.

Cross-correlation

The first considered heuristic function is the actual cross-correlation between $\mathbf{O}$ and $\mathbf{N}$, defined as the function $\mathsf{C} : \mathcal{X} \to \mathcal{Y}$ and given by

$$\mathsf{C}_{ij}(\mathbf{O}, \mathbf{N}) = \sum_{k,l \in \mathcal{P}_\mathbf{N}} o_{i+k,j+l} \cdot n_{kl}, \tag{3.24}$$

where $\mathcal{P}_\mathbf{N}$ is the set of pixels of $\mathbf{N}$, defined in (3.4). By definition, the values in the elevation maps are always greater or equal to zero. For this reason, each element of the cross-correlation output $\mathsf{C}_{ij}(\mathbf{O}, \mathbf{N})$ can be here interpreted as a weighted sum of the elements $\{o_{i+k,j+l} : k, l \in \mathcal{P}_\mathbf{N}\}$, with the weights given by $\{n_{kl} : k, l \in \mathcal{P}_\mathbf{N}\}$. Elements $o_{i+k,j+l}$ that do not coincide with the object (i.e. $n_{kl} = 0$) are not considered in the sum, while the remaining are weighted according to the object shape. Effectively, this results in a blurring operation, with the filter shape corresponding to the shape of the object. The local minima of the cross-correlation output are likely to correspond to concavities in the structure where the object fits, because sharper local minima in the elevation map $\mathbf{O}$ would have been blurred out. Additionally, the global minimum is also the lowest point in the blurred surface, which is consistent with the height criterion previously referred.

Height

As previously shown in Figure 3.2, the actual height of a position can be directly calculated from the elevation maps $\mathbf{O}$ and $\mathbf{N}$. From (3.9), a height function $\mathsf{H} : \mathcal{X} \to \mathcal{Y}$ can be defined as

$$\mathsf{H}_{ij}(\mathbf{O}, \mathbf{N}) = \max_{k,l \in \mathcal{F}_\mathbf{N}} (o_{i+k,j+l} + n_{kl}), \tag{3.25}$$

where $\mathcal{F}_\mathbf{N}$, defined in (3.5), is the set of non-empty pixels of $\mathbf{N}$. The local minima are positions where the height is the lowest in a neighborhood, and the global minimum is the lowest placement. This heuristic is expected to be sharper than (3.24), as it calculates the actual height of each position rather than performing a blurring operation.

Difference

A metric that can be used to evaluate the match between two surfaces is the sum of squared (or absolute) differences. Taking $\mathbf{O}$ and $\mathbf{N}$ positioned as in Figure 3.2, the vertical distance between surfaces may be used as a difference to be minimized. A difference based heuristic $\mathsf{D} : \mathcal{X} \to \mathcal{Y}$ is then defined as

$$\begin{aligned} \mathsf{D}_{ij}(\mathbf{O}, \mathbf{N}) &= \frac{1}{|\mathcal{F}_\mathbf{N}|} \sum_{k,l \in \mathcal{F}_\mathbf{N}} \left| \max_{u,v \in \mathcal{F}_\mathbf{N}} (o_{i+u,j+v} + n_{uv}) - o_{i+k,j+l} + n_{kl} \right|^{\mathrm{c}_d} \\ &= \frac{1}{|\mathcal{F}_\mathbf{N}|} \sum_{k,l \in \mathcal{F}_\mathbf{N}} (\mathsf{H}_{ij}(\mathbf{O}, \mathbf{N}) - o_{i+k,j+l} + n_{kl})^{\mathrm{c}_d}, \end{aligned} \tag{3.26}$$

where the exponent c_d is used to control the sensitivity of the metric. This formulation includes the absolute ($\mathrm{c}_d = 1$) and squared ($\mathrm{c}_d = 2$) differences. The heuristic function defined in (3.25) is used to

replace the $\max$ operation to point out the relation between these heuristics. Taking the absolute value (in the first line) is redundant because the argument is never less than zero (i.e. $\max_{x'} f(x') - f(x) \geq 0 \; \forall \; x$). In order to prioritize larger support polygons, this heuristic can be extended by using a weighted sum of differences, where the values from pixels farther from the center are given more importance. These weights may be defined as

$$w_{ij} = \sqrt{\left(i - \frac{\mathrm{m_N}}{2}\right)^2 + \left(j - \frac{\mathrm{n_N}}{2}\right)^2}. \tag{3.27}$$

The extended heuristic is expressed as

$$\mathsf{D}_{ij}(\mathbf{O}, \mathbf{N}) = \frac{1}{\sum_{k,l \in \mathcal{F}_\mathbf{N}} (w_{kl})^{\mathrm{c}_w}} \sum_{k,l \in \mathcal{F}_\mathbf{N}} (w_{kl})^{\mathrm{c}_w} \left(\mathsf{H}_{ij}(\mathbf{O}, \mathbf{N}) - o_{i+k,j+l} + n_{kl}\right)^{\mathrm{c}_d}, \tag{3.28}$$

where the exponent c_w controls how strongly the weights are applied, with $\mathrm{c}_w = 0$ corresponding to the expression in (3.26).

Correlation coefficients

The match between the surfaces may also be evaluated using the correlation coefficients. Note that although this operation is similar to the cross-correlation defined above, for this application the outputs have fundamentally different meanings. As previously stated, the elements of the elevation maps are always greater or equal to zero. For this reason, only by taking the mean out it is possible to actually evaluate the match between the surfaces. This heuristic, defined as $\mathsf{CC} : \mathcal{X} \to \mathcal{Y}$, is given by

$$\mathsf{CC}_{ij}(\mathbf{O}, \mathbf{N}) = \frac{\sum_{k,l \in \mathcal{P}_\mathbf{N}} \left(o_{i+k,j+l} - \overline{o}_{ij}\right) \left(n_{kl} - \overline{n}\right)}{\sqrt{\left(\sum_{k,l \in \mathcal{P}_\mathbf{N}} \left(o_{i+k,j+l} - \overline{o}_{ij}\right)^2\right) \left(\sum_{k,l \in \mathcal{P}_\mathbf{N}} \left(n_{kl} - \overline{n}\right)^2\right)}} \tag{3.29}$$

where the mean values are

$$\begin{aligned} \overline{o}_{ij} &= \frac{1}{|\mathcal{P}_\mathbf{N}|} \sum_{k,l \in \mathcal{P}_\mathbf{N}} o_{i+k,j+l}, \\ \overline{n} &= \frac{1}{|\mathcal{P}_\mathbf{N}|} \sum_{k,l \in \mathcal{P}_\mathbf{N}} n_{kl}. \end{aligned} \tag{3.30}$$

Notice that the vertical axes of $\mathbf{O}$ and $\mathbf{N}$ are in opposite directions, so a perfect fit corresponds to total negative correlation (i.e. $\mathsf{CC}_{ij}(\mathbf{O}, \mathbf{N}) = -1$).

Target

The goal given by $\mathbf{G}$ must be included in the process of choosing an action. This is done by first creating a binary map with the region where the goal is not fulfilled, with elements

$$b_{ij} = \begin{cases} 1 & \text{if } o_{ij} < g_{ij} \\ 0 & \text{otherwise.} \end{cases} \tag{3.31}$$

This is the region where additional objects are needed in order to achieve the goal. The overlap between the object and this target area is defined as $\mathsf{G} : \mathcal{X} \to \mathcal{Y}$, given by

$$\mathsf{G}_{ij}(\mathbf{O} < \mathbf{G}, \mathbf{N}) = \frac{1}{|\mathcal{F}_{\mathbf{N}}|} \sum_{k,l \in \mathcal{F}_{\mathbf{N}}} b_{i+k,j+l}, \tag{3.32}$$

where $\mathbf{O} < \mathbf{G}$ denotes the binary map defined in (3.31). The choice of actions is then reduced to the set

$$\mathcal{A}_g = \{(i,j) \in \mathcal{A} : \mathsf{G}_{ij}(\mathbf{O} < \mathbf{G}, \mathbf{N}) > \mathsf{G}_{\min}\}, \tag{3.33}$$

where $\mathcal{A}$ is the original action space (see Section 3.1.2) and $\mathsf{G}_{\min}$ is a defined threshold.

The global minimum of an heuristic function f can be obtained as $\arg\min_{i,j \in \mathcal{A}} f_{ij}(\mathbf{O}, \mathbf{N})$. However, simply replacing $\mathcal{A}$ by $\mathcal{A}_g$ in the support of the $\arg\min$ may yield a point in the edge of $\mathcal{A}_g$ that is not actually a local minimum of $f(\mathbf{O}, \mathbf{N})$. In some cases, this may be problematic. An example of why this is a problem can be visualized as placing an object in the edge of the structure (e.g. because it is the lowest position in $\mathcal{A}_g$) and it falling outwards (e.g. because the adjacent position, outside $\mathcal{A}_g$, is lower). To avoid this, it is necessary to verify that the chosen position corresponds to a local minimum. The intended behaviour may be obtained by first computing the set of local minima of $f(\mathbf{O}, \mathbf{N})$ and then using the lowest of the local minima that is inside $\mathcal{A}_g$. The set of local minima of order d is defined as

$$\mathcal{A}_{\min} = \left\{ (i,j) \in \mathcal{A} : f_{ij}(\mathbf{O}, \mathbf{N}) = \min_{k,l \in N_d(i,j)} f_{k,l}(\mathbf{O}, \mathbf{N}) \right\}, \tag{3.34}$$

where $N_d(i,j)$ is a neighborhood of (i,j), defined as

$$N_d(i,j) = \{i-d, \ldots, i, \ldots, i+d\} \times \{j-d, \ldots, j, \ldots, j+d\}. \tag{3.35}$$

Note that this definition includes flat regions of the heuristic, which are not problematic according to the identified failure mode. The action may then be chosen according to

$$\pi(s) = \underset{i,j \in \mathcal{A}_g \cap \mathcal{A}_{\min}}{\arg\min} f_{ij}(\mathbf{O}, \mathbf{N}), \tag{3.36}$$

where f may can be replaced by any of the discussed heuristics and the state s includes the elevation maps $\mathbf{O}$, $\mathbf{G}$ and $\mathbf{N}$.

3.2.2 Neural Network

The value estimator in Figure 3.6 may be extended to a DRL setting with the usage of an appropriate DNN, based on the fully convolutional Siamese architecture [19] (see Section 2.1.2) as represented in Figure 3.7. The values can then be learned by interacting with the environment, using the DQN algorithm.

In this case, the inputs of the two branches are semantically different: one contains the overhead view of the current structure and target volume and the other is the view of the object to be placed. This

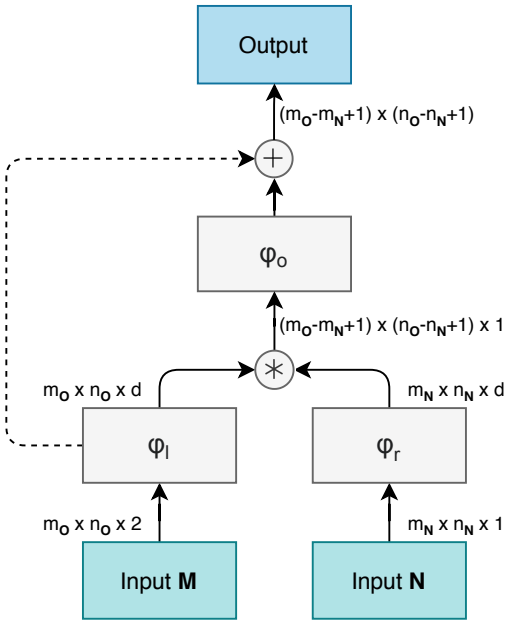


Figure 3.7: Network architecture. The modules φ_l and φ_r are the left and right branches of the network, that perform a fully convolutional dense feature extraction to the elevation maps $\mathbf{M}$ and $\mathbf{N}$. The module φ_o performs a fully convolutional transformation to the result of the cross-correlation between the feature maps, in order to estimate action advantages. An additional scalar is extracted by the module φ_l to estimate the state value $V(s)$, that is then combined with the action advantage estimates $A(s, a)$.

differs from the usual utilization of Siamese networks, where both inputs have the same meaning and must be matched (e.g. if the goal was to find an instance of the new object in the current structure). Therefore, a modification of the Siamese architecture is used, where the branches do not share weights and may even have different architectures. The only restriction is that each branch outputs an array with the same height and width as its input, and that both branches output arrays with the same depth. This means that each branch must be a FCN providing a dense (pixelwise) feature extraction, as described in Section 2.1.1.

The network architecture is extended with one additional fully convolutional module, placed at the output of the cross-correlation. This is motivated by the fact that the network must learn a specific value function, estimated from the collected rewards. In opposition to learning similarity, in which a metric must be as high as possible for a good match and low otherwise, this network should fit the specific values corresponding to the expected return. With the additional module, the output of the cross-correlation may be mapped to the intended values, giving the branches more freedom to express the features without the restriction of the cross-correlation fitting the value function.

Additionally, the network is adapted to match the dueling architecture (see Section 2.2.4). In this case, it is intuitive that the intrinsic value of a state is greatly influenced by the current structure: not only in terms of the availability of potentially good positions for the new object, but also in the stage of the episode (i.e. an advanced episode, indicated by a structure with many objects, means lower expected return because the terminal state is closer). Although the shape of the object by itself may also be indicative of the expected quality of the available positions, this can be seen as part of the action advantage. This observation is translated into the architecture by extracting a scalar value from the

branch of the overhead view, which is then combined with the output of the network according to (2.29). This value is extracted by applying fully connected layers to the higher level, lower resolution feature maps in the branch (i.e. the end of the contracting path).

Chapter 4

Implementation

This Chapter describes the implementation[1] of the approach proposed in Chapter 3. Section 4.1 provides the details about the environment implementation, while Section 4.2 describes the components of the agent.

4.1 Environment

The environment is implemented as a Python class that provides a step method, which receives an action a and returns the next state s' and reward r, and a reset method, which sets up a new episode and returns the first state (corresponding to an empty overhead view and the first object in the sequence). These are described in detail in Section 4.1.1. The process of generating the models used in the environment is presented in Section 4.1.2.

4.1.1 Methods

The environment step is divided in different sub methods, as represented in Figure 4.1. The overall process, from the agent's action to the returned result, is described in Algorithm 1. The keywords **Data** and **Input** are used to differentiate attributes of the environment, shared by all methods and across different calls to the same method (e.g. the current state; the list of objects; the identifier of the last presented object), and the external input a. The process of resetting (or setting for the first time) the environment is described in Algorithm 2, and includes removing all objects from simulation, uniformly sampling a sequence of T objects from $\mathcal{O}$ to be used in the episode and generating a new goal. The set of objects $\mathcal{O}$ is previously generated using the method described in 4.1.2. A more detailed discussion of each sub method is provided below. This implementation exposes the OpenAI Gym[2] common interface, which makes it readily usable within several existing RL frameworks.

[1] Source code available at `https://github.com/menezesandre/stackrl`.
[2] G. Brockman *et al.* OpenAI Gym. `https://gym.openai.com/`, version 0.17.2.

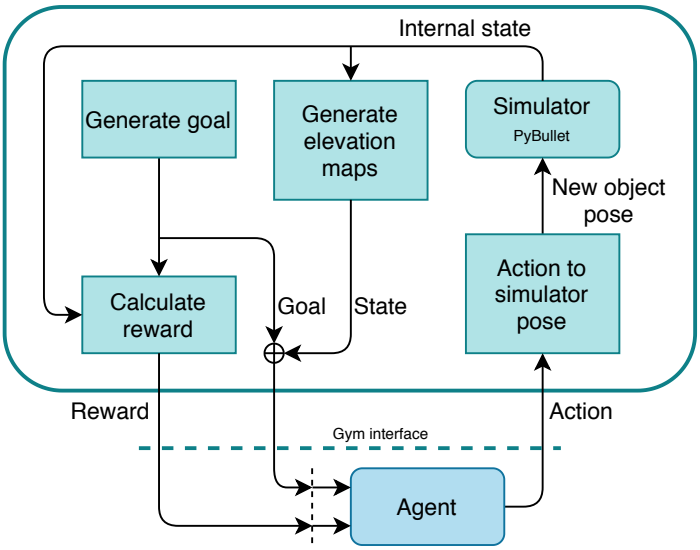


Figure 4.1: Environment implementation.

Algorithm 1: Step environment

Data : Current state and goal s, g; terminal flag $done$; new object's pose $\mathbf{x}_{\text{new}}, \mathbf{q}_{\text{new}}$; new object o_{new}; list of objects to be placed L_{new}; list of placed objects L_{place}.
Input : Action a.
Output: Next state and goal s', g; reward r; terminal flag $done$.

```
if done then
    Reset environment (Algorithm 2)          // Initializes s, g, done, L_new, L_place = ∅
else
    x_place ← Convert a to position
    Set pose of o_new to x_place, q_new      // Place the object presented in the last step
    while # contact points of o_new < 3 do
        Step simulation
        Set velocity of o_new to 0
    x_place, q_place ← Get pose of o_new
    Append o_new, (x_place, q_place) to L_place
    while any o_i ∈ L_place is moving do
        Step simulation
    if L_new is not empty then
        o_new ← Pop from L_new
        Insert o_new in the simulator at pose x_new, q_new   // Present new object to be placed
         in the next time step
    else
        done ← True
    s' ← Generate elevation maps
    r ← Calculate reward from s, s', g, L_place
    s ← s'
```

Action to simulator pose

The received action tuple is converted to the position in the base frame of the simulator, $\mathbf{x}_{\text{place}}$, using (3.9) and (3.10). The orientation is kept, so the pose used to place the new object is given by the tuple $(\mathbf{x}_{\text{place}}, \mathbf{q}_{\text{new}})$.

Algorithm 2: Reset environment

Data : Current state and goal s, g; terminal flag $done$; new object's pose $\mathbf{x}_{\text{new}}, \mathbf{q}_{\text{new}}$; new object o_{new}; list of objects to be placed L_{new}; list of placed objects L_{place}; complete set of objects $\mathcal{O}$; episode length T

Output: Initial state and goal s, g

while L_{place} *is not empty* **do**
  $o_i \leftarrow$ Pop from L_{place}
  Remove o_i from simulator

for $i = 1$ **to** T **do**
  $o_i \leftarrow$ Sample from $\mathcal{O}$
  Append o_i to L_{new}

$o_{\text{new}} \leftarrow$ Pop from L_{new}

Insert o_{new} in the simulator at $\mathbf{x}_{\text{new}}, \mathbf{q}_{\text{new}}$ `// Present the first object`

$s \leftarrow$ Generate elevation maps

$g \leftarrow$ Generate goal

$done \leftarrow$ False

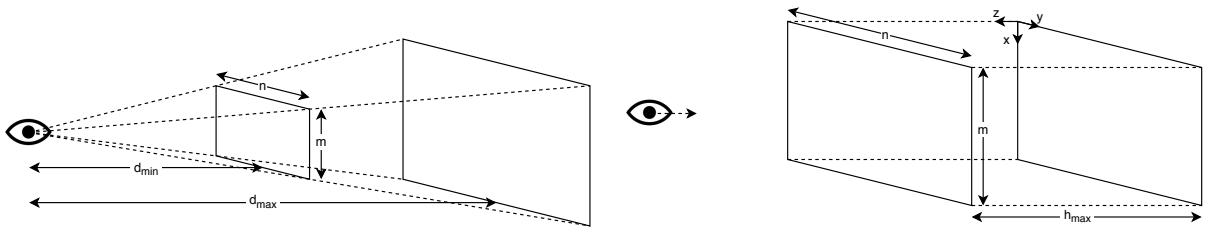
Simulator

The simulation is performed using PyBullet[3]. At each environment step, the new object's pose is set to $(\mathbf{x}_{\text{place}}, \mathbf{q}_{\text{new}})$. The simulation is then run step by step until the new object has at least three contact points with the structure, while its velocity is set to zero after each step. This process simulates a careful placement of the object in its position, by controlling its velocity until a support polygon (at least triangle) is achieved. After this, the new object is appended to the list of placed objects along with its current pose. The simulation is then run until all objects stop moving. In practice, this is achieved by, after each simulator step, checking whether the velocities of all objects are bellow a threshold (defined as $0.01ms^{-1}$). To avoid the overhead of checking all objects at each step, the velocities are checked sequentially from the last to the first object in the list of placed objects and the simulation is continued once a value above threshold is found. As the last object is the most likely to be moving, this results in only one check for most simulator steps.

Generate elevation maps

A synthetic camera is used in the physics simulator to generate depth images of the overhead view of the structure and the bottom view of the object. An elevation map differs from a depth image in two key aspects. First, the quantity represented increases as an object approaches the view point, while the depth represents the distance from the view point. Secondly, a depth image is a perspective view of the scene, while an elevation map is by definition (see Section 3.1.2) an orthogonal view. These differences are represented in Figure 4.2. It is possible to transform a perspective view into an orthogonal view using the perspective matrix. However, an orthogonal view can be easily approximated in the simulator by positioning the synthetic camera far away from the scene, such that the minimum and maximum visible depths $d_{\min}$ and $d_{\max}$ verify $d_{\max} - d_{\min} \ll d_{\min}$. With that condition assured, it is trivial to convert a depth image to an elevation map: for each depth value d the corresponding elevation is $h = d_{\max} - d$.

[3] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, version 2.9.6.



(a) Depth image.

(b) Elevation map with a representation of its frame.

Figure 4.2: Comparison between a depth image with perspective view and an elevation map with orthogonal view. Both are represented by a $\mathrm{m} \times \mathrm{n}$ matrix, with values ranging from $d_{\min}$ (near) to $d_{\max}$ (far) in the depth image and from 0 (far) to $h_{\max}$ (near) in the elevation map.

At each time step, two depth images (structure and new object) are captured after the simulation is run. The images are then converted to elevation maps. To avoid large memory requirements (to store the transitions in a replay memory, see Section 2.2.4), the elevation maps are converted to unsigned integers of 8 bits. The values are then given by

$$h^{\mathrm{uint8}} = \mathrm{trunc}(\frac{h}{h_{\max}} 2^8), \tag{4.1}$$

where the trunc operation removes the decimal part of a number. With this conversion, each elevation map represents a volume discretized in $\mathrm{d} \times \mathrm{d} \times \frac{h_{\max}}{256}$ sized voxels. To ensure consistency in the voxel size of both elevation maps, the same value is used to convert both maps, corresponding to $h_{\max} = \max(\mathrm{h}_{\mathbf{O}}, \mathrm{h}_{\mathbf{N}})$. This value is a constant defined by the parameters of the elevation maps (see Section 3.1.2).

Generate goal

Although in its formulation the goal can represent an arbitrary structure, it is here simplified to the set of structures that can be represented as

$$g_{ij} = \begin{cases} h_g & \text{if } (k_0 \leq i < k_0 + l_0) \wedge (k_1 \leq j < k_1 + l_1) \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

That is, the set of rectangular prisms with height $h_g \leq \mathrm{h}_{\mathbf{O}}$ and aligned with the base frame. The parameters (k_0, k_1) and (l_0, l_1) control the position and extents of the target structure. In this case, the set of non-empty pixels of $\mathbf{G}$ is given by $\mathcal{F}_{\mathbf{G}} = \{k_0, k_0{+}1, \ldots, k_0{+}l_0{-}1\} \times \{k_1, k_1{+}1, \ldots, k_1{+}l_1{-}1\}$. The parameters may be computed with different degrees of randomness: the extents l_0 and l_1 can be fixed, in which case the position is randomly chosen and the extents are randomly swapped (i.e. applying a $90°$ rotation); the area $l_0 \times l_1$ may be fixed, in which case l_0 is also randomly chosen and l_1 is set to keep the fixed area; nothing is fixed, in which case l_1 is also randomly chosen. The target parameters are always set so that the target structure is completely inside the map, $k_0 + l_0 < \mathrm{m}_{\mathbf{O}} \wedge k_1 + l_1 < \mathrm{n}_{\mathbf{O}}$, and so that the new object map fits in the target, $l_0 \geq \mathrm{m}_{\mathbf{N}} \wedge l_1 \geq \mathrm{n}_{\mathbf{N}}$. If one or both extents are randomly chosen, high aspect ratios are made more likely by sampling l_0 from a distribution skewed towards high

values and l_1 towards low values (l_0 and l_1 are randomly swapped afterwards).

The target structure and corresponding goal elevation map are generated at the beginning of each episode, and kept constant until time step T.

With this representation, the spatial relation between the current and target structures are clearly presented to the agent.

Calculate reward

The reward may be calculated using one of the four metrics discussed in Section 3.1.3. The IoU and the OR are directly computed from the elevation maps $\mathbf{O}$ and $\mathbf{G}$ according to (3.11) and (3.12). At each time step, the metric is computed from the new elevation maps and the reward is calculated as the difference between the current value and the value stored from the previous time step. The current value is then stored in replacement of the previous one, to be used on the next time step. When the environment is reset, the previous value is set to zero.

When using the discounted metrics, the condition in (3.14) may be expressed in terms of the target parameters as

$$b_i = \begin{cases} 1 & \text{if } \mathbf{x}_i \in \mathcal{T}_g \\ 0 & \text{otherwise,} \end{cases} \tag{4.3}$$

where $\mathbf{x}_i$ is the position of object i and

$$\mathcal{T}_g = \left\{ \mathbf{x} \in \mathbb{R}^3 : \mathrm{d}\, k_0 \leq x_0 < \mathrm{d}\,(k_0 + l_0),\ \mathrm{d}\, k_1 \leq x_1 < \mathrm{d}\,(k_1 + l_1),\ 0 \leq x_2 < h_g \right\} \tag{4.4}$$

is the target volume of goal g, corresponding to an elevation map with parameters (k_0, k_1), (l_0, l_1) and h_g. The rotation distance $\Delta\theta$ is obtained from the original and current orientation quaternions[4] ($\mathbf{q}_{\mathrm{place}}$ and $\mathbf{q}$, respectively) as

$$\begin{aligned} \Delta\mathbf{q} &= \mathbf{q}_{\mathrm{place}} \cdot \mathbf{q}^{-1}, \\ \Delta\theta &= 2 \arccos\left(\Delta q_3\right). \end{aligned} \tag{4.5}$$

The process of calculating the reward is presented in Algorithm 3.

4.1.2 Model generation

In order to allow the agent to learn a generalization, the experience provided by the environment must be diverse. This includes the object models used. As it is hard to find a large number of models of real irregular objects (e.g. natural rocks), these models are synthetically generated. This allows the generation of a dataset of models from which the sequences used in each episode are sampled.

The model generation process is based on the method presented by Thangavelu *et al.* [11] to generate datasets of irregular convex polygons, used for 2D construction. It can be defined as generating

[4] The used convention is that the scalar part of the quaternion is the last component.

Algorithm 3: Calculate reward (discounted metrics)

Data : Reward memory M^-; boolean indicating whether to compute the discounted IoU (OR otherwise); episode length T.
Input : Goal g; list of placed objects L_{place}.
Output: Reward r.

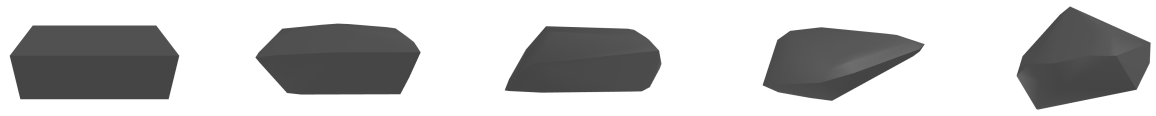
```
numerator ← 0
denominator ← T
for o_i, (x_place, q_place) ∈ L_place do
    x_i, q_i ← Get current pose of o_i
    if x_i ∈ T_g then
        |Δx| ← |x_place − x_i|
        Δθ ← Compute rotation between q_place and q_i
        d ← Compute discount from |Δx| and Δθ
        numerator ← numerator + d
    else if IoU then
        denominator ← denominator + 1
M ← numerator/denominator
r ← M − M⁻
M⁻ ← M
```

blocks deformed according to an irregularity parameter ς. As such, the starting point is a perfect brick: a box with extents $1 \times \frac{1}{2} \times \frac{1}{3}$, scaled to be inscribed in a sphere with radius $\mathrm{r}_{\max}$. The parameter $\mathrm{r}_{\max}$ is defined to bound the size of the objects (e.g. to create a set of objects that fit in a given gripper). A random displacement $\Delta\mathbf{x}$ is then applied to each of the eight vertices of the box, with each component Δx_i sampled from a truncated normal distribution with zero mean, standard deviation $\sigma = \varsigma\,\mathrm{r}_{\max}$ and support $\Delta x_i \in [-\mathrm{r}_{\max}, \mathrm{r}_{\max}]$. A sequence of n subdivisions and random displacements of the additional vertices is then applied, were the parameter n controls the level of detail of the objects. A subdivision consists in dividing each face in four smaller faces, such that a new vertex is added in the middle of each edge and in the center of each face. This reduces the length of each edge by a factor of $\frac{1}{2}$. The distribution from which the displacements of the new vertices are sampled is scaled accordingly: after the i^{th} subdivision, the truncated normal distribution has standard deviation $\sigma = \varsigma\, 2^{-i}\mathrm{r}_{\max}$ and support $\Delta x_i \in \left[-2^{-i}\mathrm{r}_{\max}, 2^{-i}\mathrm{r}_{\max}\right]$. With this process, the displacement of the original vertices greatly affects the overall shape of the object, while the following subdivisions and displacements successively increase the detail of the irregular shape. In the end of the "irregularization" process, the model is converted to its convex hull. This is done for two reasons: first, the usage of convex shapes makes the collision computations more efficient in the physics engine; secondly, the convex hulls actually resemble more natural rocks (often subject to erosion) than the obtained irregular shapes. If any of the extents of the oriented bounding box (i.e. the box with minimum volume that bounds the model) exceeds the diameter $2\,\mathrm{r}_{\max}$ due to the random displacements, the model is scaled down to fit in a bounding box with maximum extent $2\,\mathrm{r}_{\max}$.

The meshes of the models are generated according to the described process using Trimesh[5]. Each mesh is positioned and oriented in the model's frame so that its oriented bounding box is centered at the origin and aligned with the axes, with the largest extent aligned with the first axis and the smallest

[5]M. Dawson-Haggerty *et al.* Trimesh. `https://trimsh.org/`, version 3.8.1.

with the third (vertical). A value for the density is uniformly sampled from an interval $[\rho_{\min}, \rho_{\max}]$, which simulates diverse materials or materials with variable composition or porosity. The mass and moments of inertia are then calculated, using the mesh and its density. Each generated model is saved as a pair of files: the mesh is exported as a Wavefront .obj file and the model information, including mass, centroid position, moments of inertia, friction and a pointer to the mesh file, are saved in an unified robot description format (URDF) file. Some examples of models generated with different values of ς are presented in Figure 4.3.



(a) $\varsigma = 0.$ (b) $\varsigma = 0.1.$ (c) $\varsigma = 0.25.$ (d) $\varsigma = 0.5.$ (e) $\varsigma = 0.75.$

Figure 4.3: Models generated with different values of the irregularity parameter ς (3 subdivisions performed).

4.2 Agent

The implementation of the DQN agent comprises the Q-network, the replay memory, and the methods used to train the network. All components are implemented using Tensorflow[6]. The overall training process, corresponding to the DQN algorithm with the extensions presented in Section 2.2.4, is described in Algorithm 4.

The details about the network architecture and replay memory are presented in Sections 4.2.1 and 4.2.2, respectively. Section 4.2.3 describes how the training loop is designed to improve computational efficiency.

4.2.1 Network Architecture

The inputs of the network, the 8 bit elevation maps, are converted to floats in the range $[0, 1]$ by dividing the value of each pixel by 256.

The branches of the network (φ_l and φ_r in Figure 3.7) are defined by the U-net architecture [16] (see Section 2.1.1), which have been shown to perform well for dense feature extraction. It consists of a contracting path and a symmetric expanding path. In each level of the contracting path, two convolutional layers with 3×3 kernel size and the same number of filters are applied, followed by a max pooling layer with size 2×2 and stride 2, which reduces each dimension of its input to a half. After each down sampling (by the max pooling layer), the number of feature maps is doubled (i.e. the number of filters of the convolutional layers is increased by a factor of two relative to the previous level). At the last level, two more convolutions are applied and then the expanding path begins. At each level of the

[6] M. Abadi *et al.* TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org/`, version 2.3.0.

Algorithm 4: Training loop

Data : Maximum steps $t_{\max}$; initial collect steps t_{init}; target update period t_{tu}; replay memory size N; exploration parameter ε; minibatch size K; prioritization parameters α and β; discount factor γ; learning rate η.

$\mathcal{M} \leftarrow$ Initialize empty replay memory with size N
$\mathbf{w} \leftarrow$ Randomly initialize network weights
$\mathbf{w}^- \leftarrow \mathbf{w}$
$S_0 \leftarrow$ Reset environment (Algorithm 2)
$R_0 \leftarrow 0$
$done_0 \leftarrow$ False
for $t = 0$ **to** $t_{\max}$ **do**
  $A_t \sim \pi(S_t|\mathbf{w}, \varepsilon)$ (2.14)
  Append $(S_t, R_t, done_t, A_t)$ to $\mathcal{M}$ with priority $\max_{i \in \mathcal{M}} P_i$
  $S_{t+1}, R_{t+1}, done_{t+1} \leftarrow$ Step environment with action A_t (Algorithm 1)
  if $t > t_{\text{init}}$ **then**
    $\mathcal{B}_t \leftarrow$ Sample K transitions from $\mathcal{M}$ with distribution $\Pr\{i \in \mathcal{B}_t\} = \frac{P_i^\alpha}{\sum_{j \in \mathcal{M}} P_j^\alpha}$ (2.22);
    $loss \leftarrow 0$
    for $i \in \mathcal{B}_t$ **do**
      $\omega_i \leftarrow \left(\frac{\min_{j \in \mathcal{M}} P_j}{P_i}\right)^{\alpha\beta}$ (2.26)
      if not $done_{i+1}$ **then**
        $\delta_i \leftarrow R_{i+1} + \gamma Q\left(S_{i+1}, \arg\max_a Q(S_{i+1}, a \,|\, \mathbf{w}) \,|\, \mathbf{w}^-\right) - Q(S_i, A_i \,|\, \mathbf{w})$ (2.18)
      else
        $\delta_i \leftarrow R_{i+1} - Q(S_i, A_i \,|\, \mathbf{w})$
      $P_i \leftarrow |\delta_i| + \epsilon$ (2.23)
      $loss \leftarrow loss + \frac{\omega_i}{2}\delta_i^2$ (2.25)
    $\mathbf{w} \leftarrow$ Backpropagate$(\mathbf{w}, \eta, \frac{loss}{|\mathcal{B}_t|})$
    if $t \bmod t_{\text{tu}} = 0$ **then**
      $\mathbf{w}^- \leftarrow \mathbf{w}$

expanding path, a deconvolution with kernel size 2×2 and stride 2 is applied, so that each dimension of its output is doubled and the number of feature maps is halved. The upsampled feature maps are then concatenated to the feature maps from the corresponding level in the contracting path, and two more 3×3 convolutions are applied, with number of filters corresponding to the current level. This process is repeated until the output has the same size as the input. The feature maps extracted through this contracting and expanding path may then be mapped to the desired number of features using a 1×1 convolutional layer. For an input with size $\mathrm{m} \times \mathrm{n}$, the feature maps at the i^{th} level (either at the expanding or contracting path) have size $2^{-i}\mathrm{m} \times 2^{-i}\mathrm{n}$ and $2^i\mathrm{C}$, where level zero is the level of the input and output and C is the number of filters of the convolutional layers at level zero. The branches of the network, φ_l and φ_r, are thus characterized by the hyperparameters C_l and C_r, the number of filters of the first level, and the indexes of the last level N_l and N_r, such that the size and number of channels of the feature maps in the last level are $2^{-\mathrm{N}}\mathrm{m} \times 2^{-\mathrm{N}}\mathrm{n}$ and $2^{\mathrm{N}}\mathrm{C}$. Both modules must output the same number of channels to be cross-correlated, so the last 1×1 convolutions must have the same number of filters in both branches. Alternatively, the extracted features may be used directly if $\mathrm{C}_l = \mathrm{C}_r$, in which case the last layer could be dropped.

The state-value estimate $V(s)$ is computed from the output of the last level of φ_l, a stack of $2^{\mathrm{N}_l}\mathrm{C}_l$

feature maps of size $2^{-\mathrm{N}_l}\mathrm{m}_\mathbf{O} \times 2^{-\mathrm{N}_l}\mathrm{n}_\mathbf{O}$. The stack of feature maps is converted to a single feature vector by applying a global average pooling layer, that outputs a vector with $2^{\mathrm{N}_l}\mathrm{C}_l$ elements containing the average activation of each feature map. With this operation, a stack of feature maps extracted from arbitrarily sized input is reduced to a fixed size vector (i.e. the dependency on $\mathrm{m}_\mathbf{O}$ and $\mathrm{n}_\mathbf{O}$ is removed). This vector is then passed through a series of fully connected layers, the last of which containing only one neuron that outputs the scalar value. This part of the network is defined by the hyperparameters N_v, the number of intermediate fully connected layers, and C_v, the number of units in each of these layers.

Finally, the output module φ_o is just a stack of N_o convolutional layers with size 3×3 and C_o filters, terminated with a 1×1 convolution with a single filter, that maps the C_o feature maps into a single channel.

All convolutional and fully connected layers use the ReLU as activation, with the exception of the 1×1 convolutions at the end of each module and the single neuron layer that produces the state value estimate, which use no activation. The input of each convolutional layer is zero-padded, so that the output retains the size of the input.

The whole set of hyperparameters is reduced by setting fixed values or fixed relations. The depths of the state value estimator and the output module are set to $\mathrm{N}_v = 1$ and $\mathrm{N}_o = 2$. The number of filters that characterize each module are set to the same value, $\mathrm{C} = \mathrm{C}_l = \mathrm{C}_r = \mathrm{C}_o$, and the 1×1 convolution at the end of the branches is not used. The number of units of the fully connected layer (in the state value estimator) is set to match the length of its input $\mathrm{N}_v = 2^{\mathrm{N}_l}\mathrm{C}$. Noting that the smaller input contains less information and requires less downsampling for a considerable portion to be captured by a filter (in a convolutional layer), the number of levels in φ_r may be smaller than φ_l. Accordingly, the relation between the number of levels is set so that the size of the feature maps in the last level of both branches is similar, $2^{-\mathrm{N}_l}\mathrm{m}_\mathbf{O} \times 2^{-\mathrm{N}_l}\mathrm{n}_\mathbf{O} \approx 2^{-\mathrm{N}_r}\mathrm{m}_\mathbf{N} \times 2^{-\mathrm{N}_r}\mathrm{n}_\mathbf{N}$. Defining the reference number of levels $\mathrm{N} = \mathrm{N}_l$, the number of levels of φ_r may be defined as $\mathrm{N}_\mathrm{r} = \mathrm{N} - \mathrm{trunc}\left(\frac{1}{2}\log_2 \frac{\mathrm{m}_\mathbf{O}\mathrm{n}_\mathbf{O}}{\mathrm{m}_\mathbf{N}\mathrm{n}_\mathbf{N}}\right)$. With these restrictions, the complexity of the network is controlled by the two hyperparameters N and C, that define the depth and "width" of the network (depth in the sense of the length of the path from input to output, width as in the amount of information that fits through that path).

A complete diagram of the used network can be found in Figure C.1 of Appendix C.

4.2.2 Replay Memory

As described in Algorithm 4, the replay memory stores tuples with $(R_t, S_t, A_t, done_t)$, rather than the complete transition. This avoids storing each state two times: as the next state in one transition and as the current state in the next. The state is the largest element of the transition, so this modification almost halves the memory requirement for a given replay memory size. When sampling a transition i from the memory, the elements (S_i, A_i) are taken from the entry i of the memory, while $(R_{i+1}, S_{i+1}, done_{i+1})$ are taken from the next entry. This raises an issue: episode boundaries (i.e. transitions like (S_T, A_T, R_0, S_0)) that wouldn't otherwise be stored may now be sampled as a pair of consecutive entries. To avoid this, the priority of each new entry is set to zero (i.e. not sampleable) when it is added, and is only set

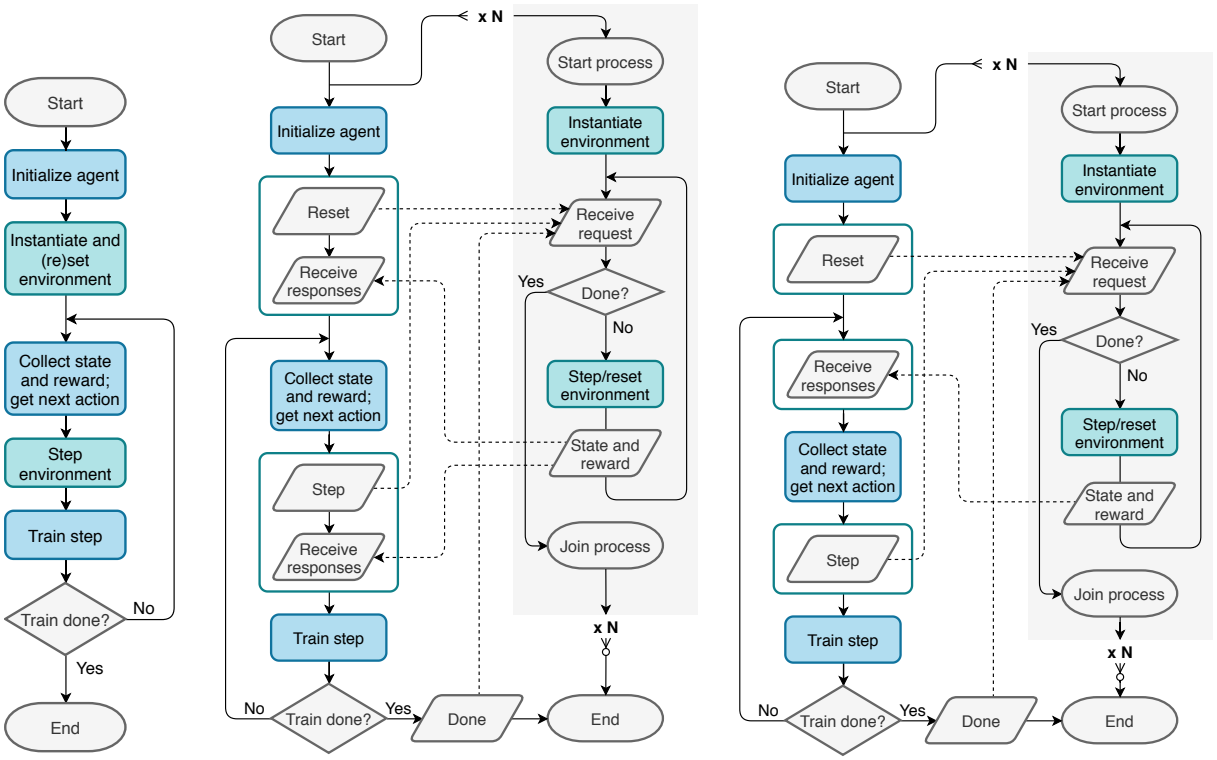
to $\max_{i \in \mathcal{M}} P_i$ on the subsequent step, when a next state is available. If an added entry is a terminal state, then the priority is not updated in the next step and is kept as zero. The values of $\max_{i \in \mathcal{M}} P_i$ and $\min_{i \in \mathcal{M}} P_i$ (used as the priority of new entries and to compute the weights for the prioritization bias correction, respectively) are stored, and recomputed whenever an update introduces a larger or smaller priority or when the entries $\arg\max_{i \in \mathcal{M}} P_i$ or $\arg\min_{i \in \mathcal{M}} P_i$ are updated.

In the training loop, the sampling method of the replay memory is wrapped in a Tensorflow generator dataset, which optimizes the input pipeline by parallelizing the data fetching and consumption. In practice, this means that a minibatch used for training in time step t may have been sampled with priorities from a previous step. This delay is limited by the maximum number of prefetched minibatches: when this number is reached, the next minibatch is only sampled when the first one was consumed. Setting a small number (e.g. 3) is sufficient to amortize the sampling time, while the introduced prioritization delay may be neglected.

4.2.3 Parallelization of simulation and network updates

Although not being an asynchronous method, the DQN algorithm has separate training and collecting processes that only need to be synchronized at a step level. That is, the minibatch sampling at time step t must occur after adding transition t in memory and the action at $t+1$ must be chosen with the parameters updated at t, but the processes may run separately between these events. To take advantage of this fact, some alterations were made to the training loop. First, a multiprocessing wrapper of the environment is implemented, such that an instance at the main process provides the same interface as the regular implementation, but the computations are actually running in a server process. Secondly, noting that the outcome of the environment step is only required in the next time step (after the training iteration), the operations are repositioned to to allow a parallelization between the training step (backpropagation) and the step of the environment (running the simulation). The alterations performed in the training loop are depicted in Figure 4.4. As these two processes make up most of the time needed to complete an iteration, this reduces the iteration time from $t_{\text{collect}} + t_{\text{train}}$ to $\max(t_{\text{collect}}, t_{\text{train}}) + \epsilon$, where ϵ is the overhead created by multiprocess communication, small when compared to the collect and train times.

Additionally, this implementation of the environment allows multiple simulations to run in parallel. In this case, the wrapper instance in the main process returns batches of states and rewards, stacked from the multiple processes, and receives batches of actions that are then unstacked and sent to each environment. The ratio of sampled transitions per transition added to the memory, that would otherwise be controlled by setting a number of collect steps performed between training steps, may also be defined using the number of parallel environments, while decreasing the elapsed time per collected sample.



(a) Single process.

(b) Multiprocessing (N environments). Calls to the environment instance are replaced by requests to server processes where the environments are running.

(c) Waiting for the servers' responses is repositioned to allow parallelization between the agent's train step and environments' steps.

Figure 4.4: Training loop flowchart.

Chapter 5

Results

This chapter presents the experiments performed to validate the proposed approach. Section 5.1 describes the overall setup for the experiments, including the general parameters used to generate models and for the environment and the metrics used. In Section 5.2, the baseline results are established using the heuristics discussed in Section 3.2.1. Finally, Section 5.3 explores the learning process and the learned policies.

5.1 Setup

5.1.1 Model generation

The models used in the experiments are generated as described in Section 4.1.2. Sets of 500 models are generated with ς ranging from 0.05 to 1 with steps of 0.05, resulting in a global set of 10000 objects. The global set is completed with the addition of a model of the perfect brick, generated with $\varsigma = 0$ (it is not necessary to generate many of these models because the generation is deterministic). The density range, from which the density of each model is uniformly sampled, is set as $[2200, 2600]$ (kg m^{-3}) based on the bulk density range of the lunar highland crust [34]. The size parameter is set as $\mathrm{r}_{\max} = 0.0625$m, so that all objects are bounded by boxes with at most 0.125m extents. The number of subdivisions performed is $\mathrm{n} = 3$.

In order to characterize the generated models, three shape metrics are used: the volume of the mesh, presented as a dimensionless ratio with respect to the volume of the perfect brick; the aspect ratio, which is here defined as the ratio between the largest and smallest extents of the oriented bounding box; and the rectangularity, here defined as the ratio between the volume of the mesh and the volume of its oriented bounding box (such that a box shaped mesh has maximum rectangularity). The perfect brick, generated with irregularity $\varsigma = 0$, is characterized by volume ratio 1, aspect ratio 3 and rectangularity 1. The shape distribution of the models is represented in Figure 5.1.

As expected, the rectangularity decreases as ς increases, and the variances of volume and aspect ratio increase. From $\varsigma \approx 0.5$ upwards, the shape distribution doesn't change significantly. A sample of objects generated with $\varsigma \geq 0.5$ is presented in Figure 5.2. The volume and the aspect ratio have

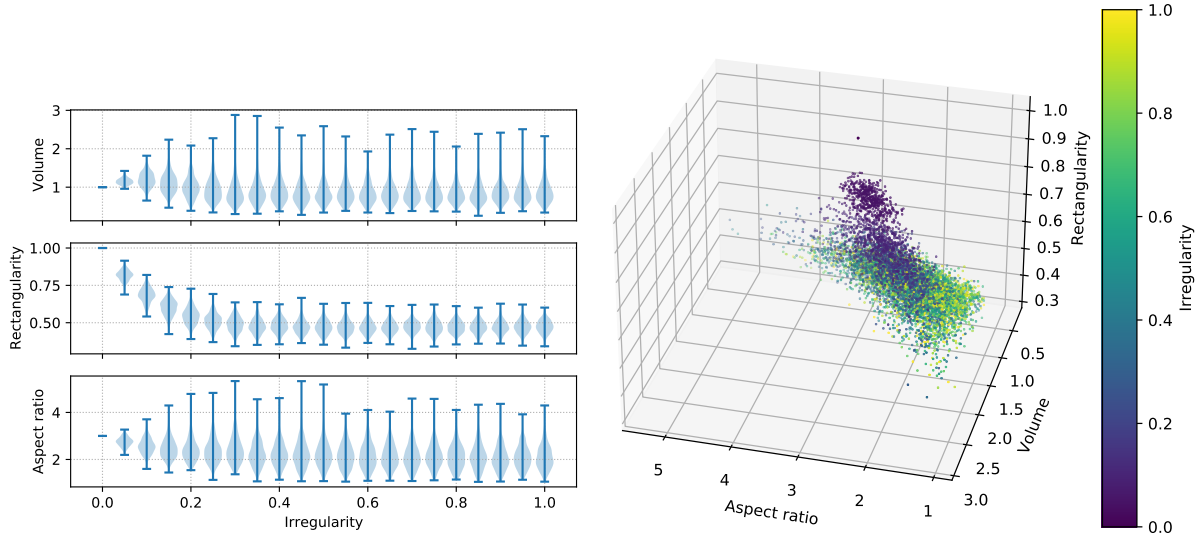


Figure 5.1: Shape distribution of the generated models.

a negative correlation, which is explained by the truncated displacements and size bounds: as the maximum extent is bounded, higher aspect ratios only occur for small values of the minimum extent, and consequently smaller volumes. This relation is visible in Figure 5.1 as a clear boundary in the distribution. The bias introduced in the shape distribution by the size limit is considered acceptable, as it could be seen as excluding unusable objects (e.g. to large to be handled by the robot) from the set.

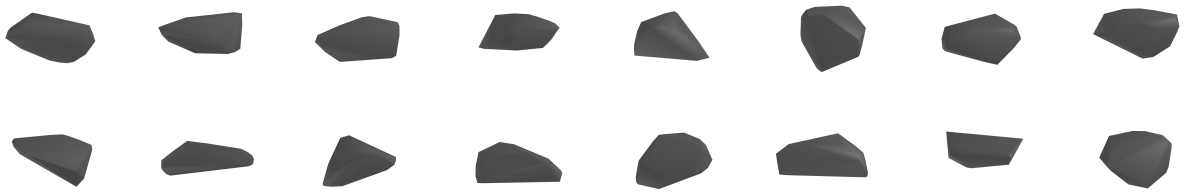


Figure 5.2: Sample of used models ($\varsigma \geq 0.5$).

5.1.2 Environment

In accordance to the scale of the generated models, the dimensions of the volume $\mathcal{V}_{\mathbf{N}}$, represented by the object elevation map $\mathbf{N}$, are set to $\mathrm{m}_{\mathbf{N}}\mathrm{d} = \mathrm{n}_{\mathbf{N}}\mathrm{d} = \mathrm{h}_{\mathbf{N}} = 0.125$m so that every object in the set is fully captured by the map. The dimensions of the area covered by $\mathbf{O}$ are set as $\mathrm{m}_{\mathbf{O}}\mathrm{d} = \mathrm{n}_{\mathbf{O}}\mathrm{d} = 0.5$m, four times greater than $\mathbf{N}$ in each dimension. The value of the pixel size d represents a trade-off between the resolution of the representations (and consequently of the discretized positions in the action space) and the computational requirements, in terms of memory and number of operations. The value used in the experiments (unless stated otherwise) is $\mathrm{d} = 2^{-8}$m $(\approx 4\text{mm})$, resulting in map dimensions $\mathrm{m}_{\mathbf{N}} \times \mathrm{n}_{\mathbf{N}} = 32 \times 32$ and $\mathrm{m}_{\mathbf{O}} \times \mathrm{n}_{\mathbf{O}} = 128 \times 128$ and a set of $97 \times 97 = 9409$ possible actions at each step. The goals are generated with a fixed area of one fourth of the total observable area, $l_0 \times l_1 = 0.25 \cdot \mathrm{m}_{\mathbf{O}} \times \mathrm{n}_{\mathbf{O}}$ $(= 4 \cdot \mathrm{m}_{\mathbf{N}} \times \mathrm{n}_{\mathbf{N}})$. The target height is set as $h_g = 2\,\mathrm{h}_{\mathbf{N}} = 0.25$m. Note that although this is only two times the maximum dimension of the objects, the height is typically the smallest dimension due to the oriented

bounding box alignment (e.g. 0.25m corresponds to seven courses of the perfect brick). The maximum height visible in $\mathbf{O}$ is set as $\mathrm{h}_{\mathbf{O}} = h_g + \mathrm{h}_{\mathbf{N}} = 0.375$m, so that the target is visible with a margin of the maximum dimension of an object. This corresponds to a height resolution (in the discretized 8 bit maps) of $0.375 \cdot 2^{-8}$m $(\approx 1.5\text{mm})$. The number of objects necessary to approximate the target is estimated with the number of perfect bricks that fit in the volume $0.25 \cdot \mathrm{m}_{\mathbf{O}}\mathrm{d} \cdot \mathrm{n}_{\mathbf{O}}\mathrm{d} \cdot h_g$, corresponding to $\mathrm{T} = 76$. Unless specified, the models used are sampled from the set of 5500 objects generated with $\varsigma \geq 0.5$. The step size of the physics engine (i.e. the amount of time proceeded in each simulation step) is set to 0.01s, which was found to be a good trade-off between simulation speed and accurate behaviour. The gravitational acceleration is defined as $\mathrm{g} = 9.8\text{ms}^{-2}$, the value at the surface of the Earth (see Section 5.2.2 for a comparison with $\mathrm{g} = 1.6\text{ms}^{-2}$, the value at the lunar surface).

5.1.3 Metrics

The metrics used to evaluate the policies are the IoU, defined in (3.11), and the average discount

$$\mathsf{AD}_t = \frac{1}{t}\sum_{i=0}^{t-i} \mu_t^{[i]}, \tag{5.1}$$

where $\mu_t^{[i]}$ is the discount defined in (3.17) applied to the i^{th} object at time step t. While the first evaluates the similarity between the current and target structures, the second quantifies the effectiveness of the policy by evaluating the proximity between the planed placements and the actual positions of the objects in the current structure. A policy is considered good if it "intentionally" achieves a high similarity with the target structure, where the intentionality is measured by the proximity between planed and actual poses. The evaluations are reported as the average final value of the metrics, $\overline{\mathsf{IoU}}_{\mathrm{T}}$ and $\overline{\mathsf{AD}}_{\mathrm{T}}$ (for the reference $\mathrm{T} = 76$ or a different defined value), which are an evaluation of the result of the assembly plan. However, in difficult settings it may be hard to surpass some asymptotic performance boundary, which makes policies of different quality reach the end of the episode with similar values. To differentiate these, an evaluation of the metrics' evolution is included in the form of the area under the curves, $\mathsf{A}(\mathsf{IoU})$ and $\mathsf{A}(\mathsf{AD})$, defined as

$$\mathsf{A}(M) = \sum_{t=1}^{\mathrm{T}} \overline{M}_t, \tag{5.2}$$

where M is one of the metrics. A higher value of $\mathsf{A}(\mathsf{IoU})$ means that the policy reaches a high IoU (close to the asymptotic boundary) sooner, which is preferable. Higher $\mathsf{A}(\mathsf{AD})$ means that the proximity between planed and actual positions holds for longer, even if it inevitably decreases towards the end of the episode. The average values are computed over a set of N evaluation episodes, such that

$$\overline{M}_t = \frac{1}{\mathrm{N}}\sum_{i=1}^{\mathrm{N}} M_{i,t}, \tag{5.3}$$

where $M_{i,t}$ is the value of the metric in time step t of the i^{th} episode.

For the training rewards, the four metrics discussed in Section 3.1.3 are considered. The rewards

are generated according to the difference between the values of the metric in consecutive time steps, as in (3.20).

The parameters of the discounted metrics are defined as follows. The maximum translation distance is set to $\Delta \mathrm{x}_{\max} = 0.125$m, the distance needed for an object to have zero overlap with its original position (corresponding to the maximum dimension of the objects). The maximum rotation distance is set to $\Delta\theta_{\max} = \pi$rads, corresponding to opposite orientations. The exponents are set to $\mathrm{c}_{\mathbf{x}} = \mathrm{c}_{\theta} = 2$, which means the discounts are applied as a squared difference.

In Section 3.1.3, the discounted metrics were formulated based on the assumption that the number of objects inside the target is approximately equivalent to the intersection between current and target structures. In order to validate this, a sequence of 100 episodes is run with a policy that randomly picks an action from $\mathcal{A}$ at each time step. Each state is simultaneously evaluated with the number of objects inside the target, $\sum_{i=0}^{t-i} b_i$, and the intersection between structures, $\sum_{i,j \in \mathcal{P}_{\mathbf{O}}} \min(o_{ij}, g_{ij})$. At each step, two rewards are yielded as the difference between consecutive values of each of the metrics. The two sequences of rewards present a correlation coefficient of 0.76, which confirms the assumption made.

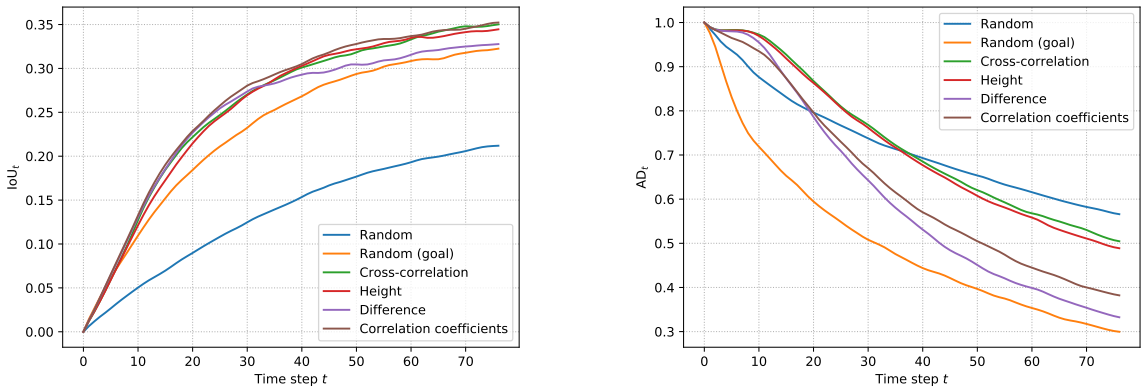
5.2 Baseline

The baseline results are obtained for three levels of (expected) performance. The first corresponds to the policy that randomly samples actions from $\mathcal{A}$. The second introduces the notion of the goal by restricting the set of actions to be sampled to $\mathcal{A}_g$ (3.33), the set of positions in the target region. Finally, some knowledge about the problem is added to the action choice with one of the four heuristics discussed in Section 3.2.1.

The minimum overlap with the target region, used to compute $\mathcal{A}_g$ (3.33), is set to $\mathrm{G}_{\min} = 0.75$, meaning that at most one fourth of the object's area is allowed to be outside of the target. The neighborhood size used to compute the set of local minima $\mathcal{A}_{\min}$ (3.34) is set to $d = 1$. This value is enough to significantly (not totally, because the estimates are noisy) suppress the effect of choosing a minimum in the edge of $\mathcal{A}_g$ that is not actually a local minimum of the function, while the computational overhead of computing the local minima is kept low. The exponents of the difference based heuristic D (3.26) are set to $\mathrm{c}_d = \mathrm{c}_w = 2$, as these were the values that held a better performance amongst the possible combinations of absolute and squared difference ($\mathrm{c}_d \in \{1, 2\}$) and linear, quadratic or no weighting ($\mathrm{c}_w \in \{0, 1, 2\}$).

Each of the six considered policies is tested with the same set of 100 episodes (i.e. same targets and sequences of objects), with the environment setup described in Section 5.1.2. The curves $\overline{\mathrm{IoU}}_t$ and $\overline{\mathrm{AD}}_t$ are shown in Figure 5.3. The evaluation metrics are reported in Table 5.1. Figure 5.4 shows examples of structures obtained with each policy for a given target and sequence of objects.

The three distinct levels of performance are visible in the results. As expected, the random policy doesn't approximate the target structure, which is translated by a low IoU. The distribution of the objects through a larger area allows lower displacements from the original poses, resulting in higher AD. The introduction of the goal to reduce the set of possible actions greatly increases the IoU, but the absence



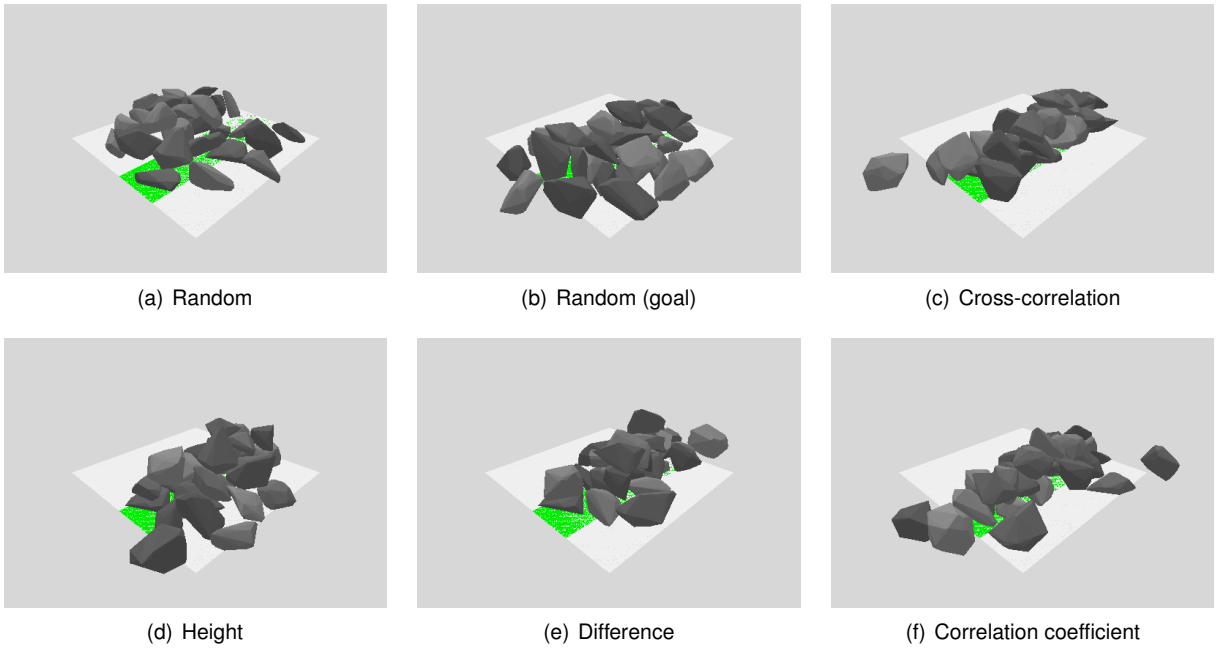
(a) Intersection over union.

(b) Average discount.

Figure 5.3: Evolution of the evaluation metrics throughout the episode for the baseline policies. Each point represents the average over 100 episodes.

Table 5.1: Evaluation metrics for the baseline policies. Values reported as the average and standard deviation over 100 episodes.

	$\overline{\text{IoU}}_{76}$	A(IoU)	$\overline{\text{AD}}_{76}$	A(AD)
Random	0.212 ± 0.029	10.3 ± 0.2	0.565 ± 0.039	54.8 ± 0.5
Random (goal)	0.323 ± 0.028	17.8 ± 0.2	0.299 ± 0.040	38.0 ± 0.6
Cross-correlation	0.351 ± 0.024	19.8 ± 0.2	0.503 ± 0.036	55.2 ± 0.3
Height	0.345 ± 0.029	19.6 ± 0.2	0.488 ± 0.040	54.6 ± 0.4
Difference	0.328 ± 0.033	19.3 ± 0.3	0.331 ± 0.033	46.0 ± 0.4
Correlation coefficients	0.352 ± 0.031	20.2 ± 0.2	0.381 ± 0.041	48.2 ± 0.5



(a) Random

(b) Random (goal)

(c) Cross-correlation

(d) Height

(e) Difference

(f) Correlation coefficient

Figure 5.4: Examples of structures obtained with the baseline policies for the same target and sequence of objects, at time step $t = 30$.

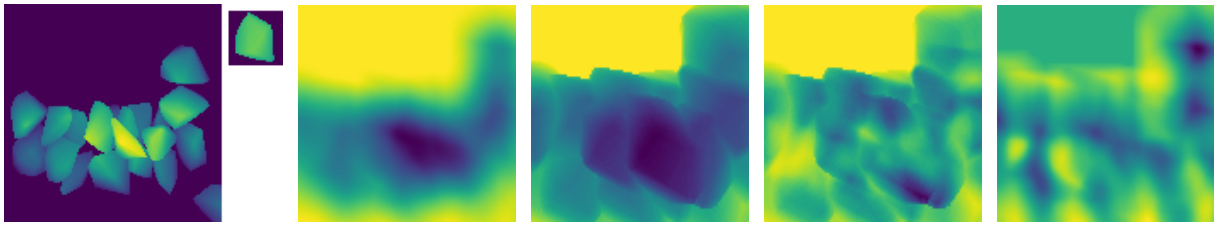
of a pose choice criterion results in a highly ineffective policy and large displacements from the original poses. The heuristics improve the effectiveness of the planed placements, resulting in a further improvement of the IoU and higher values of AD. Despite its simplicity, the cross-correlation based heuristic produces the best combination of IoU and AD.

From Figure 5.3, it is possible to split an episode in roughly three stages. In the first stage, it is fairly easy to place the objects inside the target without displacements. This is observable in the initial section of the $\overline{\text{AD}}_t$ curves (Figure 5.3(b)), in which the values obtained by the heuristic policies stay close to one. This stage corresponds to filling the target area with a first course of objects and may be roughly defined as the first 10 steps of an episode, which is both consistent with the observed curves and the number of perfect bricks that fit in a course with the target area defined in Section 5.1.2. After the first course, the episode may be further divided in two more stages from the $\overline{\text{IoU}}_t$ curves (Figure 5.3(a)). Initially, the action selection criteria introduced by the heuristics produce better placements, which is translated by the greater slopes in the $\overline{\text{IoU}}_t$ curves relative to the policy that randomly selects positions in the target. After a certain point, the different position choices no longer produce significantly different results and the curves become approximately parallel, each converging to an asymptotic boundary. This point may be defined as 30 steps, which in the ideal case (perfect brick) would correspond to three courses. On the third stage, it is increasingly difficult to find a good position, regardless of the selection criterion. This observation also confirms the importance of reporting the area under the curves to differentiate distinct levels of performance before the asymptotic boundary.

To quantify the relation between the different heuristics, a sequence of 4000 states is used to simultaneously generate value maps according to each heuristic. The correlation coefficients of the estimated values are presented in Table 5.2 and Figure 5.5 shows an example of the value maps for a given state. All pairs present positive correlation, which means that to some extent the heuristics capture similar features of the state. The most distinct from the remaining is the correlation coefficients based heuristic (CC). The cross-correlation (C) and height (H) based heuristics are the most similar, as both take into account the elevation of a position. While C produces a smooth value map, corresponding to filtering the overhead elevation map with the object, H yields a sharper value map with the exact heights of each position. This makes the first slightly more robust to irregular objects, while the latter is specially suitable for more regular objects.

Table 5.2: Correlation coefficients of the values estimated by each of the heuristics, for a sequence of 4000 states.

	C	H	D	CC
C	1	0.91	0.5	0.54
H	0.91	1	0.76	0.37
D	0.5	0.76	1	0.09
CC	0.54	0.37	0.09	1



(a) State (b) Cross-correlation (c) Height (d) Difference (e) Correlation coefficient

Figure 5.5: Examples of value maps obtained with the heuristics for a given state ($t = 20$). Yellow represents highest value, corresponding to minimum of the heuristics (the action choice is restricted to local minima in the target region, not depicted here).

5.2.1 Influence of elevation map resolution

The same set of 100 episodes is run again with different values of the elevation maps' pixel size d, and consequently different numbers of possible actions. The additional values considered are $\mathrm{d} \in \{2^{-9}, 2^{-7}\}$ [m], corresponding to half and double the original pixel size. Halving the pixel size corresponds to increasing the number of pixels (and consequently the required memory and number of operations) by a factor of 4. The performance metrics obtained with the cross-correlation heuristic for the different values of d are presented in Table 5.3. While using the pixel size $\mathrm{d} = 2^{-8}$m represents an improvement over the larger $\mathrm{d} = 2^{-7}$m, further subdividing it doesn't seem to improve performance and strongly increases the computational costs.

Table 5.3: Evaluation metrics obtained with different pixel sizes, for the cross-correlation heuristic policy. Values reported as the average and standard deviation over 100 episodes.

d [m]	$\overline{\mathrm{IoU}}_{76}$	A(IoU)	$\overline{\mathrm{AD}}_{76}$	A(AD)
2^{-7}	0.35 ± 0.025	19.8 ± 0.2	0.497 ± 0.034	55.0 ± 0.3
2^{-8}	0.351 ± 0.024	19.8 ± 0.2	0.503 ± 0.036	55.2 ± 0.3
2^{-9}	0.348 ± 0.029	19.8 ± 0.2	0.497 ± 0.039	55.2 ± 0.4

5.2.2 Influence of gravitational acceleration

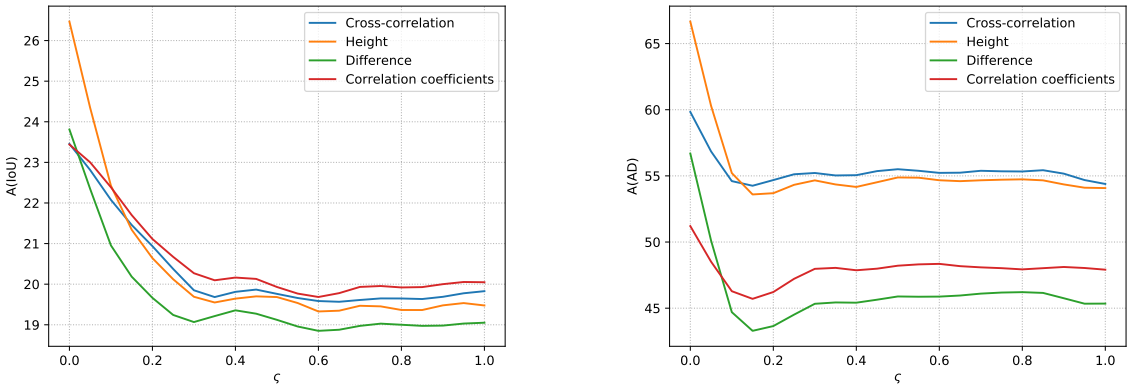
The set of 100 episodes is repeated with the gravitational acceleration set to $\mathrm{g} = 1.6\mathrm{ms}^{-2}$. Table 5.4 reports the performance of the cross-correlation heuristic under each of the gravity values. While a slightly greater similarity with the target is achieved with lower g, the structures obtained under higher g are in general closer to the planed positions. The results are close, which means the difficulty of learning a policy for any of the gravity values should be similar. An important difference is that the simulation progresses slower for lower gravity. Concretely, for the experiment with $\mathrm{g} = 1.6\mathrm{ms}^{-2}$ the environment takes on average 118 simulation steps from the placement until the objects settle, while for $\mathrm{g} = 9.8\mathrm{ms}^{-2}$ it takes on average 68 steps, making the simulations generally faster for the later. This makes this value preferable for faster experiments, while it may be assumed that equivalent results would be obtained for different values of g.

Table 5.4: Evaluation metrics obtained with different values of gravitational acceleration, for the cross-correlation heuristic policy. Values reported as the average and standard deviation over 100 episodes.

g [ms^{-2}]	$\overline{\text{IoU}}_{76}$	A(IoU)	$\overline{\text{AD}}_{76}$	A(AD)
9.8	0.351 ± 0.024	19.8 ± 0.2	0.503 ± 0.036	55.2 ± 0.3
1.6	0.358 ± 0.024	20.2 ± 0.2	0.483 ± 0.037	54.3 ± 0.4

5.2.3 Influence of irregularity

To evaluate the effect of the irregularity in the performance, each policy is evaluated on 21 sets of 25 episodes, each set using sequences of sampled objects with a specific value of $\varsigma \in \{0, 0.05, \ldots, 1\}$. The evolution of A(IoU) and A(AD) with the increase of the irregularity ς is depicted in Figure 5.6. As expected, the height based heuristic performs particularly well for regular objects. As for the shape distribution (see Figure 5.1), the performance is similar with objects generated with irregularity greater than $\varsigma \approx 0.5$.



(a) Intersection over union.

(b) Average discount.

Figure 5.6: Influence of irregularity on the evaluation metrics, for the baseline policies. Each point represents the average over 25 episodes.

5.3 Learning

Based on the conclusions from Section 5.2, the learning is focused on the first two stages of construction, roughly corresponding to the assembly of a structure with 30 objects. This is the segment of the task in which it is clear that a better policy produces better results, which means it is learnable.

The environment is setup according to the description in Section 5.1.2, with episode length $\mathrm{T} = 30$. In order to accelerate the collection of experience when training, the simulation step size is set to 0.0125s, corresponding to a 20% reduction in the number of steps necessary to elapse a given time period in simulation. This is the largest step size that was verified not to introduce incorrect behaviour in the simulation (e.g. objects penetrating the ground plane). As the observations are only taken in the end of each simulator run (when the objects settle), the accuracy of the process is not too relevant as long as

the final result is not physically incorrect.

The rewards are scaled upwards to be of order one, which is closer to the expected initial value estimates from the randomly initialized network. If the rewards are very small, the initially random estimates of the next state's value completely dominate the TD target $R_{t+1} + \gamma V(S_{t+1})$. This requires an initial, sometimes long, learning stage that corresponds to gradually scaling down the network's output, which delays the process of actually learning from the rewards to later iterations. Taking the number of perfect bricks that fit in the target as a reference, the value of the IoU or OR increases by approximately $\frac{1}{76}$ when an object is successfully placed in the target, so the rewards based on these metrics must be scaled by a factor of 76 to be of order one. The discounted metrics DIoU and DOR are defined to ideally reach 1 at the final time step T, so scaling the rewards by a factor of $\mathrm{T} = 30$ results in rewards that ideally take the value 1 at each time step.

The network used to learn the value function follows the architecture described in Section 3.2.2. The size of the network is defined by $\mathrm{N} = 5$ and $\mathrm{C} = 16$, resulting in a network with 2126226 trainable parameters. The value of N is set so that each neuron in the last convolutional layer of the bottom level has a field of view with size similar to the input of the network. The value of C is a trade-off between representation capacity and computational cost. For each experiment, the network's weights are initialized with values sampled from normal distributions with variance scaled according to the method proposed by He *et al.* [35] and the biases are initialized as zero. The parameters are optimized using Adam [27] with learning rate $\eta = 6.25 \cdot 10^{-5}$ and exponential decay rate for the first and second moment estimates $\beta_1 = \beta_2 = 0.95$. This learning rate is the largest value in $\left\{0.001 \cdot 2^{-i} : i \in \mathbb{N}\right\}$ that provided consistent convergence. The function being optimized is the Huber loss [26] of the TD error, which is more robust to outliers than the squared error.

The training process follows the Double DQN algorithm with prioritized experience replay (see Algorithm 4). The replay memory size is defined as $N = 400000$ transitions, which is the largest size that satisfies the memory constraints. Two transitions are collected between network updates, corresponding to two environments running on parallel processes. Each network update is performed using a minibatch of size $K = 32$. With these settings, each transition is replayed on average 16 times before being replaced in the replay memory. The target network is updated every 10000 iterations. The exploration is performed using an ε-greedy policy with ε linearly annealing from 1 to 0.1 throughout the first 400000 training iterations. A set of 20000 transitions is collected with a random policy before training starts. The prioritization exponent is set as $\alpha = 0.6$ and the priority bias compensation β is linearly annealed from 0.4 to 1 over the first 400000 iterations. The discount factor is set as the probability of a transition not being terminal, given by $\gamma = 1 - \frac{1}{\mathrm{T}} \approx 0.967$.

The obtained policies may be qualitatively characterized in terms of the level of behaviour, based on the observations from Section 5.2. One of the learning objectives should be for an agent to capture the notion of the goal, which is translated by the assignment of higher values to positions in the target. The other is to capture the dynamics of the environment in order to choose the positions that lead to the best final structure. The later may be observed in the behaviour through the two considered stages of construction: first, a good foundation for the structure must be laid, by filling the target area with the

first course; afterwards, the structure must be gradually increased by choosing good positions for the following objects.

On the used hardware (CPU: Intel(R) Core(TM) i7-7820X, GPU: GeForce GTX 1080 Ti) and for the described setup, one training iteration takes around 0.15s of continuous process time (CPU for the simulation and GPU for inference and backpropagation). This results in almost 42 hours of continuous process time to complete 10^6 iterations and in practice a training session of this length takes longer (e.g. if more than a session is running simultaneously). This makes the experiences extremely costly in terms of computation. Due to this limitation, only one or two runs are presented to illustrate most of the conclusions. It is also possible that an extensive hyperparameter search could allow further improvements to the results.

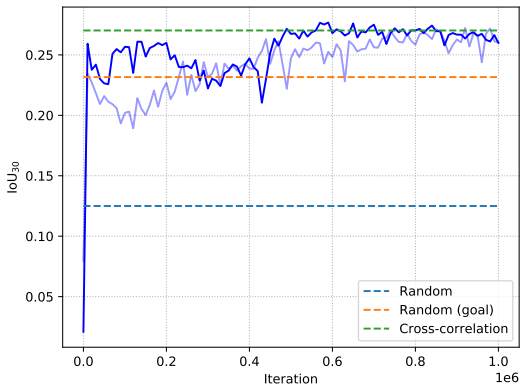
5.3.1 Main experiments

A set of training sessions is run with rewards generated by each of the four metrics discussed in Section 3.1.3, to evaluate the behaviours that emerge from each reward shape. After each 10000 iterations, a set of 100 episodes is run separately (without collecting the transitions) with the current greedy policy (no exploration) and evaluated using the IoU. This allows a comparison between the evolution of the different training sessions independently of the random behaviour introduced by exploration and the metric used to generate the rewards. Two different random initializations are used with each metric. The evolution of the evaluations during training is presented in Figure 5.7. The policies obtained after 10^6 iterations are then evaluated on the same set of 100 episodes to observe the evolution of the IoU and AD during the episode. Figure 5.8 depicts the best (out of the two runs) $\overline{\text{IoU}}_t$ and $\overline{\text{AD}}_t$ curves obtained with the policies learned from each metric. The corresponding results are reported in Table 5.5. The learning curves of the policies used for Figure 5.8 are highlighted in Figure 5.7 with the stronger line. When the training sessions are referred to as first and second run, the second is the best one (stronger line).

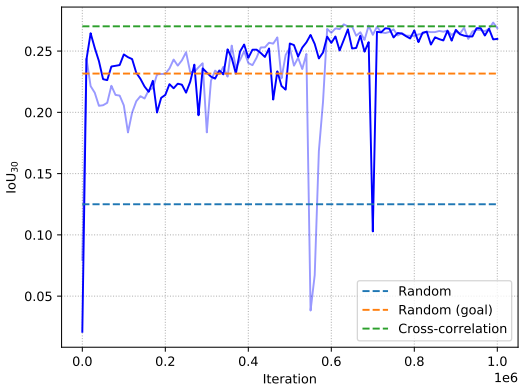
Table 5.5: Evaluation metrics for the learned policies. Values reported as the average and standard deviation over 100 episodes.

	$\overline{\text{IoU}}_{30}$	A(IoU)	$\overline{\text{AD}}_{30}$	A(AD)
Learned from IoU	0.266 ± 0.026	5.17 ± 0.1	0.584 ± 0.064	23.0 ± 0.4
Learned from OR	0.265 ± 0.027	5.08 ± 0.1	0.569 ± 0.059	21.8 ± 0.4
Learned from DIoU	0.285 ± 0.029	5.38 ± 0.1	0.703 ± 0.056	26.2 ± 0.3
Learned from DOR	0.281 ± 0.031	5.37 ± 0.1	0.682 ± 0.058	25.7 ± 0.3
Random (goal)	0.232 ± 0.025	4.31 ± 0.1	0.507 ± 0.064	20.2 ± 0.5
Cross-correlation	0.27 ± 0.024	5.03 ± 0.1	0.769 ± 0.046	27.1 ± 0.2

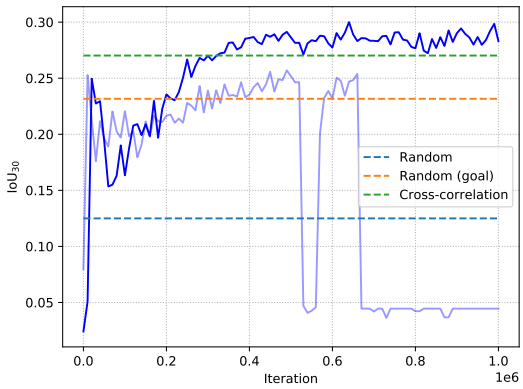
From the learning curves in Figure 5.7, it is observable that the notion of the goal is easily (and quickly) learned from the rewards generated with any of the metrics. This corresponds to reaching the second level of baseline performance (given by the policy that samples actions from the target region), which always happens on the first 10^4 iterations (also, see the value maps after 10^4 iterations



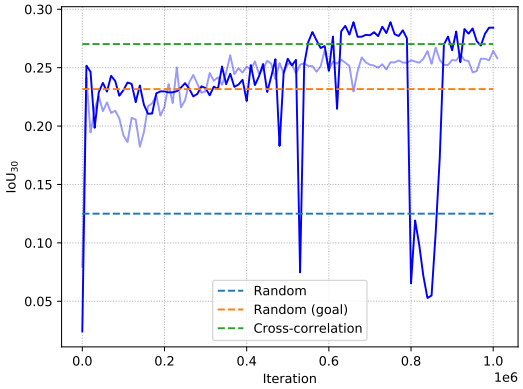
(a) Learning from IoU



(b) Learning from OR

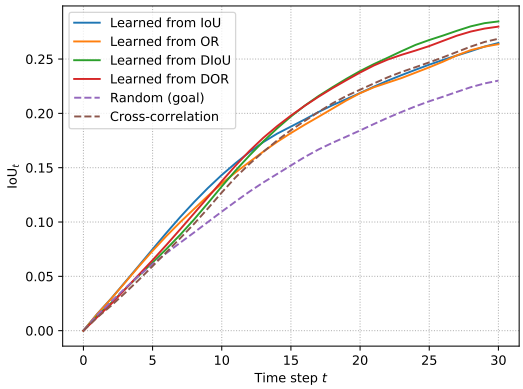


(c) Learning from DIoU

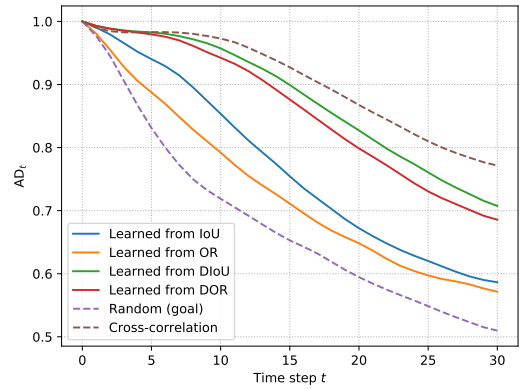


(d) Learning from DOR

Figure 5.7: Evolution of the evaluation results during training with rewards generated by each of the metrics. Each plot shows two runs with different initializations, with the one that produces the best final policy highlighted with the stronger line. The three levels of baseline performance are presented for reference.



(a) Intersection over union.



(b) Average discount.

Figure 5.8: Evolution of the evaluation metrics throughout the episode for the policies learned after 10^6 iterations. Each point represents the average over 100 episodes.

in Appendix A). Afterwards, some placing criterion is gradually learned and the performance becomes comparable with the third level, given by the cross-correlation heuristic. Although all policies reach this level, only the policies learned from the discounted metrics (DIoU and DOR) were able to significantly outperform the cross-correlation heuristic in the evaluation during training.

Sometimes, during training, the performance breaks down. This happens when one of the layers in the network's critical path stops producing any activation for most inputs it receives, which results in the network predicting equal values for all actions most of the times. [1] Frequently, there is still some inputs to which the layer responds, which allows the network to recover and sometimes even transform its representation into something better (according to the evaluation criterion). However, it may also happen that the layer stops producing activations for all inputs it receives, which means that nothing is backpropagated to the layer's weights nor to the previous layers. In this case, learning stops completely for that segment of the network and the collapse is unrecoverable. An example of this is the worst run in Figure 5.7(c), in which it is verified that the output of the left branch (that extracts features from the new object) is all zero for any object, resulting in the cross-correlation output being all zero and, consequently, the same advantage for all actions.

The curves from Figure 5.8 allow an initial analysis of the obtained behaviours. From Figure 5.8(b), the difference between the policies learned with the discounted (DIoU and DOR) and not discounted (IoU and OR) metrics is clear. The later were not explicitly directed towards minimizing the displacements, which is reflected in an $\overline{\text{AD}}_t$ curve closer to the random policy. Despite this, these policies manage to achieve a similarity with the target structure close to the cross-correlation heuristic, as seen in Figure 5.8(a). The fact that these policies do not keep the value of AD close to 1 on the first steps of the episode, along with the higher slope in the beginning of the $\overline{\text{IoU}}_t$ curve, bring up a previously ignored fact about the IoU and OR: as the volume considered to be the structure includes empty spaces beneath the objects, these metrics may be maximized by pilling the objects with large spaces between them. This behaviour makes the intersection increase faster initially, but produces unstable structures (intuitively, a robust structure should be compact) and ultimately undermines the performance. On the other hand, the policies learned with the discounted metrics show an $\overline{\text{AD}}_t$ curve close to the cross-correlation heuristic, with an initial segment close to one. This indicates that a first course is laid, upon which the subsequent objects are (somewhat) carefully placed. With this behaviour, the policies reach a greater similarity with the target, translated by the largest final IoU. Between the policies learned with the discounted metrics, the one learned with the DIoU presents a slightly better performance, which may be explained by the fact that this metric is more informative. On the other hand, this fact also makes the value function harder to learn, which could make it more prone to a network collapse (although the fact that this happened in one out of two runs is not conclusive). The behaviour learned from this metric is studied to more depth bellow.

[1] A layer is in the network's critical path if any of the input information has necessarily to pass through that layer to reach the output. This excludes layers in the lower levels of the branches, due to the skip connections of the U-net.

Consistency of learned value functions

Despite the differences in performance, the learned value functions share a somewhat consistent foundation across runs and different reward metrics (see the similarities in Appendix A, Figures Figure A.2 to Figure A.9). This is due to the shared goal of placing objects in the target. To quantify this, a sequence of 1920 states is simultaneously evaluated by the networks obtained by each of the eight runs after $4 \cdot 10^5$ iterations, and the obtained pixelwise values are correlated. The correlation coefficients obtained for all 28 pairs are verified to be greater or equal to 0.9. As the training progresses, each network develops a more specific shape for the value estimates, but the foundation remains similar. Repeating the previous experiment with the networks obtained after 10^6 iterations, 13 out of the 28 possible pairs retain a correlation coefficient greater than 0.9, while all the remaining pairs (with the exception of the run with DIoU that collapsed) present correlation coefficients greater than 0.72.

Utilization of network capacity

In order to quantify the utilization of the representation capacity of the networks, some of the feature maps obtained inside the network are analysed. Concretely, the analysed feature maps are the outputs of the left and right branches (arrays with dimensions $128 \times 128 \times 16$ and $32 \times 32 \times 16$, respectively) and the outputs of the last layer of the bottom level of each branch, corresponding to the end of the contracting path (arrays with dimensions $8 \times 8 \times 256$ for the left branch and $8 \times 8 \times 64$ for the right one). Each pixel is seen as an instance of a features vector. For each trained network, a sequence of 3 episodes of length $\mathrm{T} = 60$ is played. Here, longer episodes are used to expose the networks to a wider variety of states. The pixels of the outputs of each of the referred layers, obtained for each state, are stored as a list of feature vectors. This results in lists with length $3 \cdot 60 \cdot 128 \cdot 128 = 2949120$ and $3 \cdot 60 \cdot 32 \cdot 32 = 184320$ for the outputs of the left and right branches and $3 \cdot 60 \cdot 8 \cdot 8 = 11520$ for the outputs of the bottom layers. From the obtained sequences of feature vectors, two metrics may be computed. For each sequence, the number of features that are not always zero represents the number of feature maps in the corresponding layer that produce any activation. Using principal component analysis, the number of components needed to explain 99% of the variance of the sequence of feature vectors may be used as a metric of the utilization of the capacity. This number of components corresponds to the number of feature maps that the layer would need to represent a linear combination of the current output while losing only 1% of the representation. Table 5.6 reports these metrics for the best performing networks obtained with each reward shape after 10^6 training iterations. From the number of components needed to represent each layer's output, it is clear that the networks are far from their full representation capacity. Often, and specially for the bottom layers, the representation is compressed to only use a subset of the available channels. This indicates that the used size of the networks is not a limitation and that longer training sessions could improve the representation, and consequently the performance.

Table 5.6: Analysis of the utilization of the network's representation capacity after 10^6 iterations, for best performing networks obtained with each reward shape. For each of the selected layers, the table reports the number of principal components that capture 99% of the feature maps' variance (#PC) and the number of channels that produce any activation for the given input (#Act.).

Learned from	Left output (16 ch.)		Left bottom (256 ch.)		Right output (16 ch.)		Right bottom (64 ch.)	
	#PC	#Act.	#PC	#Act.	#PC	#Act.	#PC	#Act.
IoU	14	16	6	51	12	16	32	54
OR	9	16	59	255	6	14	12	27
DIoU	11	16	4	19	9	12	39	50
DOR	10	16	7	59	10	16	13	33

Behaviour learned from DIoU

As the best performing policy emerged from the rewards generated with the DIoU, the behaviours learned from this metric are further analysed. First, to evaluate the consistency, two additional training sessions are run with different initializations. The best one out of the four (the two runs originally reported in Figure 5.7(c) and the two new ones) is then trained for longer, in order to check if the performance can be further improved. The learning curves are presented in Figure 5.9 and show the initial evolution of the four sessions and the further training of the best one.

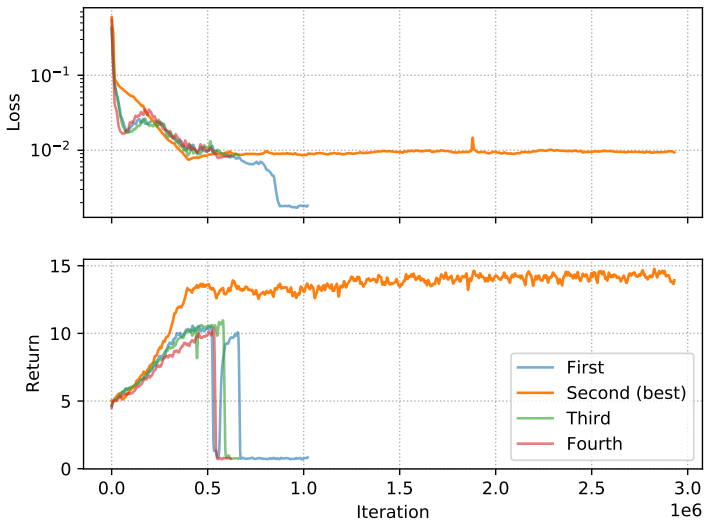


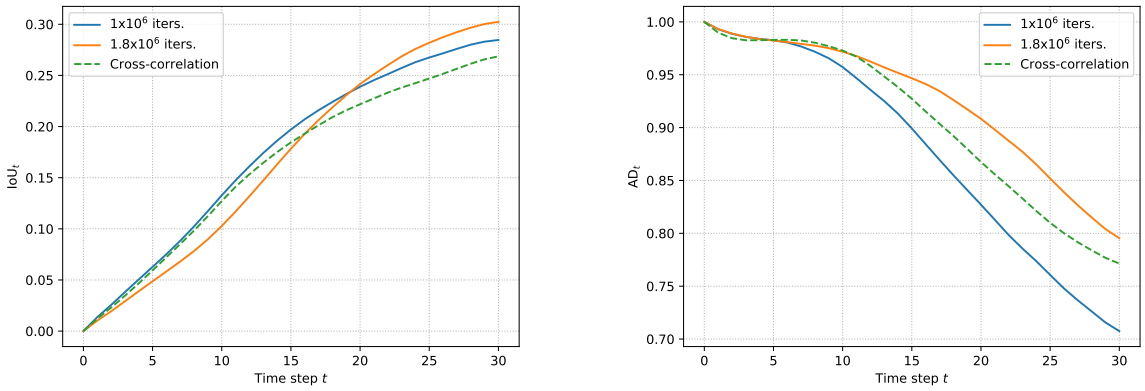
Figure 5.9: Learning curves of the four runs with rewards generated by the DIoU. Values reported every 100 iterations. The loss is the average over each 100 iterations. The return is the average undiscounted episode return over the last 200 training episodes. All curves are smoothed with a Gaussian filter with standard deviation 25 (corresponding to 2500 iterations) and truncated at 4 standard deviations to make the plots clearer.

From the learning curves in Figure 5.9, it is clear that the most frequent evolution is closer to the first run, which suggests that the second run is the result of a "lucky" initialization. It may be concluded that a value function for the DIoU is hard to learn, which results in frequent collapses of the network when learning with the defined settings.

It is visible from the loss curves that the network frequently converges to a local minimum early in

training. On the other hand, the second run gradually decreases the loss until the end of the exploration annealing period. The later behaviour could be made more frequent by, for example, using a smaller learning rate and increasing the exploration annealing period. This is consistent with the idea that an extensive (and expensive) hyperparameter tuning would allow improvements to the obtained results.

Although the evaluations using the IoU (Figure 5.7(c)) do not appear to change significantly after a certain point, the return curve (which is the sum of rewards generated by the DIoU) of the second run in Figure 5.9 shows a slight increasing tendency. This suggests that learning could continue to slowly improve the policy over a training session with length of a larger order of magnitude. This improvement is explained by an increasing effectiveness that results in lower displacements from the original poses. This is illustrated in Figure 5.10, which compares the evolution of the evaluation metrics during an episode (averaged over the same set of 100 episodes) for the policy after 10^6 and $1.8 \cdot 10^6$ iterations. The $\overline{\text{AD}}_t$ curves clearly reflect this difference, with the later policy achieving a larger $\overline{\text{AD}}_{\text{T}}$ and $\text{A}(\text{AD})$, even surpassing the cross-correlation heuristic in this aspect. From the $\overline{\text{IoU}}_t$ curve it is possible to interpret the behaviour development: the curve increases slower in the beginning because the agent learned to create a more compact first course, which ultimately leads to a better final result. The checkpoint at $1.8 \cdot 10^6$ iterations is selected to illustrate the behaviour because it was verified to achieve the best average performance under both evaluation criteria. However, it is also verified that the variance of the results decreases for later iterations, which means that the policy becomes more consistent.



(a) Intersection over union.

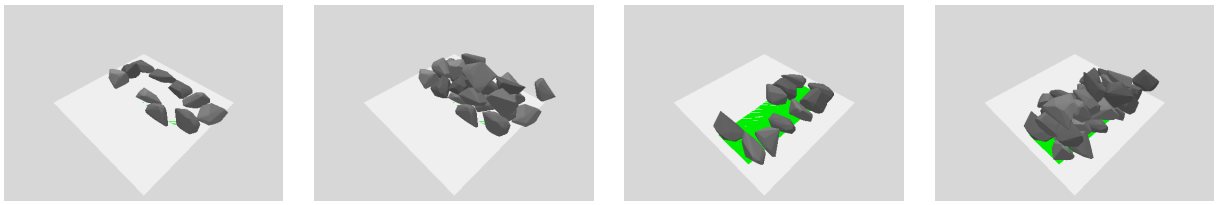
(b) Average discount.

Figure 5.10: Evolution of the evaluation metrics throughout the episode for the policy learned with the DIoU after 10^6 and $1.6 \cdot 10^6$ iterations. Each point represents the average over 100 episodes.

Figure 5.11 shows examples of structures obtained with the policy learned after $1.8 \cdot 10^6$ iterations. Figure 5.12 depicts value maps obtained with the trained network. In order to characterize the learned position choice criteria, the value maps generated by the network are matched with value maps generated by the the cross-correlation and correlation coefficients based heuristics. These heuristics are selected because each captures one of the predefined criteria: prioritizing lower positions and finding a good fit for the object in the structure. For each state in a sequence of 100 episodes with length $\text{T} = 30$, the values obtained for the pixels in the target region are correlated[2]. Figure 5.13 shows the distribu-

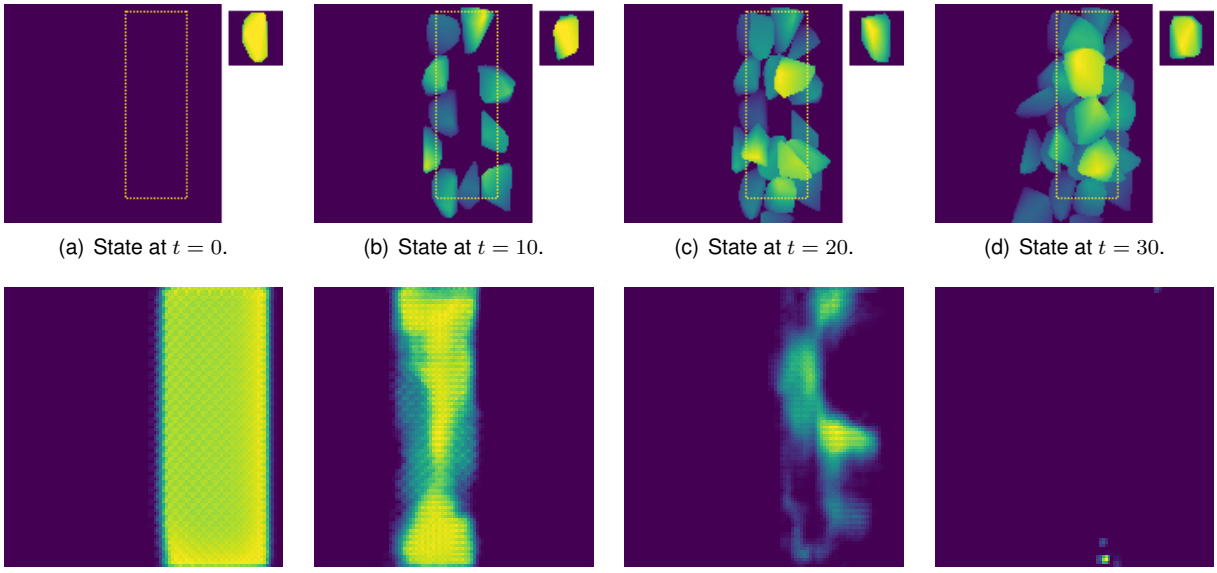
[2]The correlation is only computed for the pixels in the target region because that's the region to which the heuristics apply

tion of the obtained correlation coefficients per episode step (over the 100 episodes). This presentation allows an understanding of how the learned behaviour adapts to each stage of the episode.



(a) First episode, $t = 10$. (b) First episode, $t = 30$. (c) Second episode, $t = 10$. (d) Second episode, $t = 30$.

Figure 5.11: Examples of structures obtained with the policy learned from rewards generated by the DIoU, after $1.8 \cdot 10^6$ training iterations. Images show two time steps ($t \in \{10, 30\}$) for each of two episodes.



(a) State at $t = 0$. (b) State at $t = 10$. (c) State at $t = 20$. (d) State at $t = 30$.

(e) Value map, $t = 0$. (f) Value map, $t = 10$. (g) Value map, $t = 20$. (h) Value map, $t = 30$.

Figure 5.12: Examples of value maps obtained with the policy learned from rewards generated with the DIoU, after $1.8 \cdot 10^6$ training iterations. Images show four states ($t \in \{0, 10, 20, 30\}$) and the corresponding estimated value maps.

One very interesting observation is the behaviour learned for the first steps. The agent learns to start an episode by enclosing the target area with the first objects. This allows the following objects to be supported by an inwards sloping surface, which prevents them from falling out and increases the overall stability. This behaviour, learned from scratch, is consistent with the theory of dry stacking (although typically the largest objects would be selected for this purpose, which is not allowed here). After this first stage, the agent tries to pack the objects as tightly as possible to fill the inside of the first course and build upon it. As the episode progresses, the actions seen as advantageous become sparser, as it can be visualized in Figure 5.12. The value maps for the first steps are highly correlated with the cross-correlation heuristic. This is expected because the behaviour of enclosing the target area and then filling its inside is consistent with placing the objects as low as possible. The similarity with the correlation coefficients heuristic increases during the first steps, as the target area fills up and the objective of tightly packing the objects in the first course becomes coincident with finding a good fit for the object in

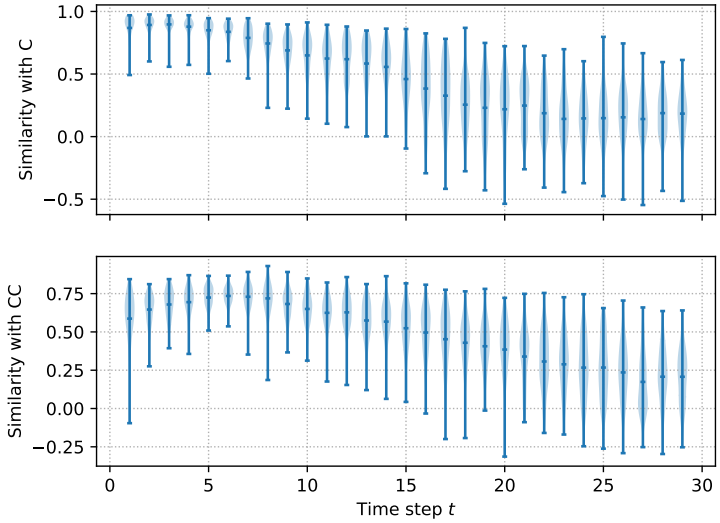


Figure 5.13: Correlation coefficients between the values estimated by the learned policy and heuristics during an episode. The top plot shows the similarity with the cross-correlation heuristic (C) and the bottom one with the correlation-coefficient heuristic (CC). Each column represents the distribution obtained for 100 episodes.

the structure. As the number of objects in the structure increases, the correlation with both heuristics decreases (although remaining positive on average). From the obtained results (see Figure 5.10), it is clear that the learned value function captures a better understanding of the environment dynamics than the defined heuristics, which is translated by generally more stable placements.

5.3.2 Ablation

This Section presents experiments that illustrate the improvements introduced by some of the choices previously used.

Dueling architecture

To evaluate the effect of the dueling architecture, a pair of training sessions is run with the same setup and network initialization, the only difference being that the dueling part is removed from one of the networks. This corresponds to the fully connected layers that estimate the state-value and its addition to the output of the network. The resulting non dueling network has 2060177 trainable parameters, 66049 less than the original, but this change doesn't significantly reduce the computational cost. The rewards for both training sessions are generated using the DOR metric and the length of the training episodes is $\mathrm{T} = 40$.

The learning curves over 10^6 iterations are presented in Figure 5.14, and an example of the value maps returned by the networks are shown in Figure 5.15. Although corresponding to one run only, the results are perfectly illustrative of the improvement introduced by the dueling architecture. The fact that the state-value is estimated separately allows the action advantage, given by the cross correlation, to

be focused on the actions inside the target. In the case of the non dueling architecture, the output of the cross-correlation is forced to take into account the returns over the complete action space, because even if the action doesn't bring advantage there is always the possibility of future rewards. This leaves less capacity to express the different values of each immediate action.

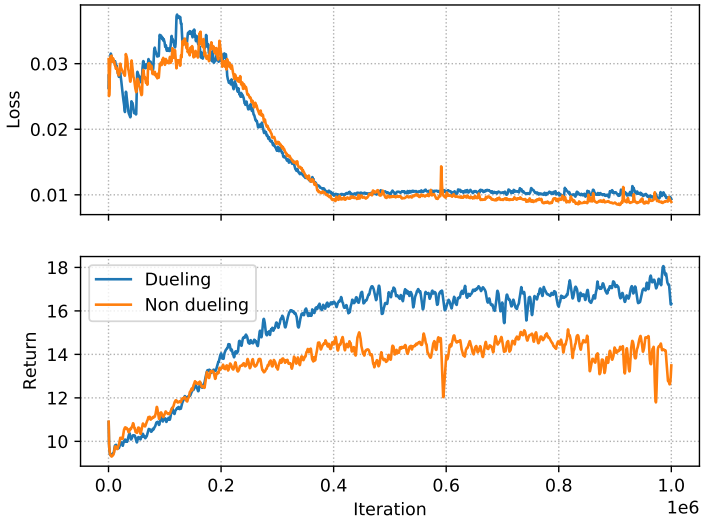
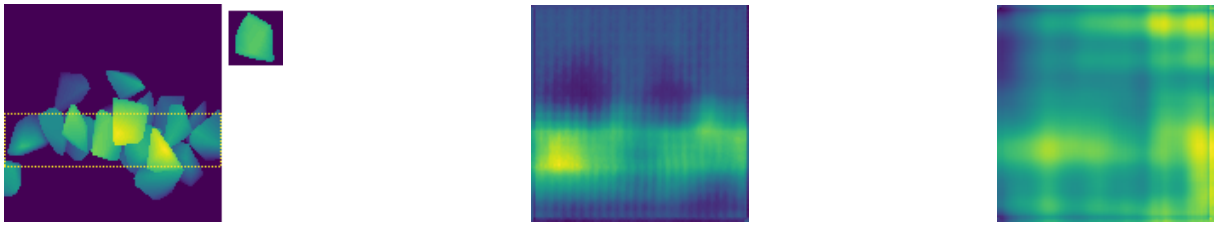


Figure 5.14: Learning curves with and without the dueling architecture. Values reported every 100 iterations. The loss is the average over each 100 iterations. The return is the average undiscounted episode return over the last 200 episodes. All curves are smoothed with a Gaussian filter with standard deviation 5 (corresponding to 500 iterations) and truncated at 4 standard deviations to make the plots clearer.



(a) State (b) Dueling (c) Non dueling

Figure 5.15: Examples of value maps obtained with and without the dueling architecture for a given state ($t = 20$), from networks trained over 10^6 iterations.

Reward scaling

Figure 5.16 presents an example of a learning curve obtained without reward scaling, compared with the exact same setup (including network initialization) and rewards scaled as previously described. Rewards are generated using the DIoU metric. The process of learning from the rewards only starts around iteration $4 \cdot 10^5$, when the network's output is small enough to be comparable with the rewards. Figure 5.17 shows examples of value maps returned by the networks at two points in training: before and after the run without reward scaling starts to fit to the rewards. At iteration $4 \cdot 10^5$, the network trained with scaling already shows an understanding of the environment, while the one trained without shows no relation

with the actual values of the actions. At iteration $6 \cdot 10^5$, the network trained with no scaling already reflects the values of the actions, but at this point the one trained with scaled rewards already learned a more refined estimate of the values. It is not clear that the asymptotic result without scaling would be worst, but the learning process is definitely accelerated by scaling the rewards.

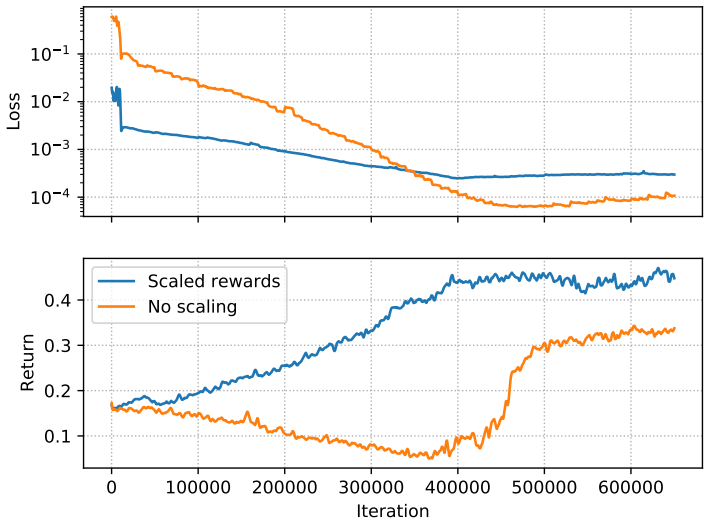
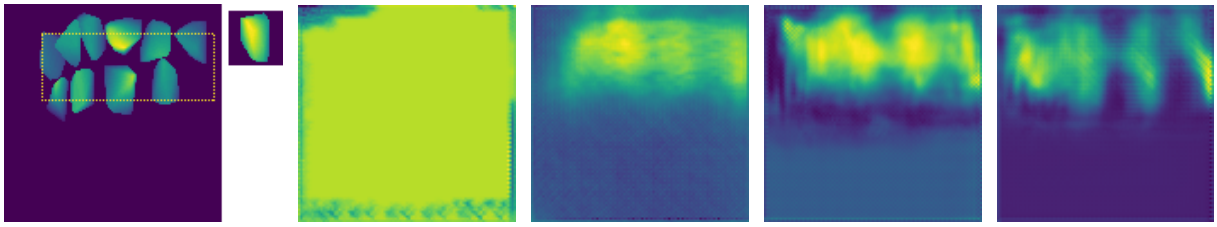


Figure 5.16: Learning curves with and without reward scaling. The return is the average undiscounted episode return over the last 200 episodes. The curves corresponding to scaled rewards are scaled back down in order to match the quantities represented by the ones without scaling. All curves are smoothed with a Gaussian filter with standard deviation 5 (corresponding to 500 iterations) and truncated at 4 standard deviations to make the plots clearer.



(a) State (b) No scaling, $4 \cdot 10^5$ iterations (c) No scaling, $6 \cdot 10^5$ iterations (d) Scaled rewards, $4 \cdot 10^5$ iterations (e) Scaled rewards, $6 \cdot 10^5$ iterations

Figure 5.17: Examples of value maps obtained with and without reward scaling at two training points (before and after the network starts fitting to the unscaled rewards), for a given state ($t = 10$).

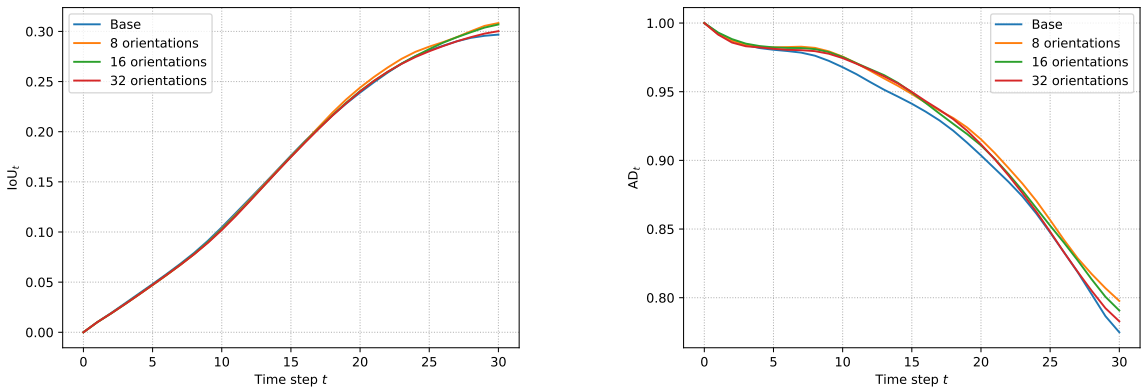
5.4 Extensions to state and action spaces

Additional experiments were performed in order to evaluate the improvements allowed by the extensions to the state and action spaces discussed in Section 3.1.4. A set of 100 episodes is run under each of different settings, using the policy learned from DIoU rewards after $2.71 \cdot 10^6$ iterations. This checkpoint is used here instead of $1.8 \cdot 10^6$ iterations because it was also verified to be a local maximum in terms of average performance and, additionally, the results obtained for this experiments make the conclusions

more evident. The results obtained after $1.8 \cdot 10^6$ iterations are equivalent, but the differences between the different settings are much smaller (i.e. the average curves are too close).

Orientation

To evaluate the effect of providing orientation freedom, experiments were performed providing 8, 16 and 32 possible orientations for each object. The different orientations are obtained with a rotation of the object around the vertical axis, such that the complete turn around the axis is divided in equal intervals (e.g. for 8 possible orientations, the rotation interval between consecutive orientations is $\Delta\theta = \frac{\pi}{4}$rads). The results obtained for each setting are shown in Figure 5.18. It is clear that providing the freedom for the agent to select the orientation improves the performance, as all obtained curves are above the one obtained with the base formulation. From the initial evolution of the $\overline{\text{AD}}_t$ curve, it is possible to understand that this extension allows the agent to assemble a tighter first course, which leads to the overall improvement of the final result. There is no need to provide a high resolution in terms of the orientation (i.e. small angles between possible orientations) to see this performance improvement, as the best results are obtained when the agent is provided 8 possible orientations to select from.



(a) Intersection over union.

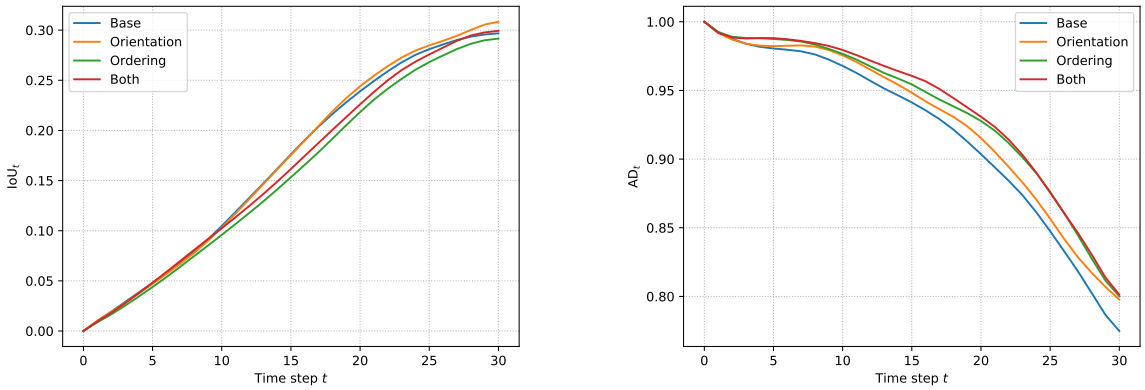
(b) Average discount.

Figure 5.18: Evolution of the evaluation metrics throughout the episode with different levels of orientation freedom. Values obtained with the policy learned from DIoU rewards after $1.8 \cdot 10^6$ training iterations. Each point represents the average over 100 episodes.

Ordering

In order to evaluate the result of providing the choice of the next object to the agent, additional experiments were performed, first with fixed orientation and then providing 8 possible orientations. The results obtained with each extension and both simultaneously are compared in Figure 5.19. As expected, the agent does not respond so well to the choice of the next object. The $\overline{\text{AD}}_t$ curve holds closer to 1 for longer, which is explained by the fact that the agent starts by selecting the "easiest" objects first, which yield higher expected returns. This behaviour ignores the fact that the "hardest" objects are being left to the end of the episode, which leads to a faster decrease of the $\overline{\text{AD}}_t$ towards the end of the episode and

a generally lower final value for the IoU. Providing the choice of orientation along with the choice of the object improves these results, but the negative effect of greedily selecting the next object remains.



(a) Intersection over union.

(b) Average discount.

Figure 5.19: Evolution of the evaluation metrics throughout the episode with the different extensions to state and action spaces. Values obtained wih the policy learned from DIoU rewards after $1.8 \cdot 10^6$ training iterations. Each point represents the average over 100 episodes.

Chapter 6

Conclusions

The major achievement of this work was to develop a setup in which a dry stacking policy can be learned with a model-free approach. An agent is able to capture the goal of the environment and its dynamics, and even develop a strategy that is, to some extent, consistent with the existing dry stacking theory [3]. Despite this remarkable learning result, the obtained performance is still bounded by the limited action space, which is far from the available to a human performing the task.

With this achievement, we can confirm that it is possible to learn this complex task from scratch, without relying on any previously acquired models of the objects and environment dynamics. Additionally, the RL approach is validated by the emergence of strategy, which requires an understanding of the sequentiality of the task in addition to the dynamics that rule each single transition.

One of the major drawbacks for this work was the computational cost of the experiments, that include intensive physical simulations and expensive operations with large neural networks. This factor made the path to the presented results a very prolonged process of trial and error, which ultimately prevented further improvements to the solution.

6.1 Future Work

As stated, the results suggest that further improvements could be achieved with a better choice of parameters. To find the optimal set of hyperparameters, a more thorough parameter search would need to be performed. This search would not need to be a brute force grid search, but wold require several iterations of performing training sessions, taking conclusions and adjusting the parameters accordingly.

The architecture of the network used as the value estimator could also be developed. Although not commonly used in DRL, there is a number of recent advances in the field of DL that may be used to improve the network representation. As an example, the utilization of residual connections [36] could contribute to avoid the frequent network collapses observed in training (see Figure 5.9). Additionally, a fully convolutional alternative to the U-net like architecture is the usage of atrous (dilated) convolutional layers [37], which has also been shown to perform well for dense feature extraction.

This work employed the DQN algorithm with some of its improvements. However, there is a number

of other modifications to the algorithm that have been shown to improve the performance [6]. One that could be readily introduced in the proposed approach is n-step bootstrapping [20, Ch. 7], which only requires small modifications to the process of sampling transitions and computing the TD targets.

The proposed approach can be seen as a high level planning method. Even though the learned policy captures a good representation of the task, the performance is bounded by the feedforward execution. This work could be applied to a realistic setting by introducing it in a pipeline with a lower level feedback controller and a robotic manipulator to perform the placements. Such controller could be used to produce the careful placements simulated in the environment. After the manipulator is moved to the position given by the planer, the object can be carefully placed by applying torques to the joints with magnitude just under the necessary to counteract the gravity of the object. A stable position is reached when the object appears to have no weight to the manipulator. This is a better criterion than achieving at least three contact points. Several possibilities can be explored from this process. A maximum displacement could be defined, so that if no stable position is reached after this displacement the placing is aborted and the manipulator retracts with the object (and a different position is tried). Additionally, different abortion criteria could be defined from the available sensory inputs, e.g. in order to avoid a structure collapse. All of these ideas for an autonomous system are consistent with the usual process of building dry stack walls (by humans), that includes carefully trying positions, checking the stability and repositioning if necessary.

Another point that could be explored is the selection of the next object to be placed. As shown, simply applying the learned policy to greedily select the next object results in leaving the “worst” objects (according to the learned value function) to the end of an episode, which is likely to be a poor sorting criterion. To improve this, the sorting of the objects would have to be included in the learning process, which would require to rethink the agent architecture.

Bibliography

[1] K. R. Sacksteder and G. B. Sanders. In-Situ Resource Utilization for lunar and Mars exploration. In *Collection of Technical Papers - 45th AIAA Aerospace Sciences Meeting*, volume 6, 2007.

[2] M. Thangavelu. Living on the Moon. In *Encyclopedia of Aerospace Engineering*. John Wiley & Sons, Ltd, Chichester, UK, 2012.

[3] J. Vivian. *Building Stone Walls*. Storey Publishing, second edition, 1976.

[4] C. S. Allen, R. Burnett, J. Charles, F. Cucinotta, R. Fullerton, R. Goodman, A. D. Griffith, J. J. Kosmo, M. Perchonok, J. Railsback, S. Rajulu, D. Stilwell, G. Thomas, T. Tri, R. Wheeler, A. Mueller, and A. Simmons. Guidelines and Capabilities for Designing Human Missions. *Nasa/Tm–2003–210785*, (January), 2003.

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

[6] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. AAAI press, 2018.

[7] M. Lambert and P. Kennedy. Using artificial intelligence to build with unprocessed rock. In *Key Engineering Materials*, volume 517, 2012.

[8] S. A. Nielsen and A. Dancu. Fusing design and construction as speculative articulations for the built environment. In Ajla Aksamija, John Haymaker, and Abbas Aminmansour, editors, *Future of Architectural Research: Proceedings of the Architectural Research Centers Consortium Conference*, Chicago, 2015. Perkins+Will.

[9] F. Furrer, M. Wermelinger, H. Yoshida, F. Gramazio, M. Kohler, R. Siegwart, and M. Hutter. Autonomous robotic stone stacking with online next best object target pose planning. In *Proceedings - IEEE International Conference on Robotics and Automation*. Institute of Electrical and Electronics Engineers Inc., 2017.

[10] Y. Liu, J. Choi, and N. Napp. Planning for Robotic Dry Stacking with Irregular Stones. *12th Conference on Field and Service Robotics (FSR19)*, 2019.

[11] V. Thangavelu, Y. Liu, M. Saboia, and N. Napp. Dry Stacking for Automated Construction with Irregular Objects. In *Proceedings - IEEE International Conference on Robotics and Automation*. Institute of Electrical and Electronics Engineers Inc., 2018.

[12] Y. Liu, M. Saboia, V. Thangavelu, and N. Napp. Approximate stability analysis for drystacked structures. In *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2019-May. Institute of Electrical and Electronics Engineers Inc., 2019.

[13] Y. Liu, S. M. Shamsi, L. Fang, C. Chen, and N. Napp. Deep Q-Learning for Dry Stacking Irregular Objects. In *IEEE International Conference on Intelligent Robots and Systems*. Institute of Electrical and Electronics Engineers Inc., 2018.

[14] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[15] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 07-12-June-2015. IEEE Computer Society, 2015.

[16] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9351. Springer Verlag, 2015.

[17] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. Signature Verification using a "Siamese" Time Delay Neural Network. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan-Kaufmann, 1994.

[18] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, volume I. IEEE Computer Society, 2005.

[19] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. Torr. Fully-convolutional siamese networks for object tracking. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9914 LNCS. Springer Verlag, 2016.

[20] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, second edition, 2018.

[21] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.

[22] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[23] H. van Hasselt. Double Q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*. Curran Associates, Inc., 2010.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

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

[26] P. J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35 (1):73–101, 1964.

[27] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2015.

[28] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-Learning. In *30th AAAI Conference on Artificial Intelligence, AAAI 2016*. AAAI press, 2016.

[29] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2016.

[30] Leemon C. Baird III. Advantage Updating. Technical Report WL–TR-93-1146, Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1993.

[31] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, New York, New York, USA, 2016. PMLR.

[32] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup. Horde: A Scalable Real-Time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.

[33] T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal Value Function Approximators. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, Lille, France, 2015. PMLR.

[34] W. S. Kiefer, R. J. MacKe, D. T. Britt, A. J. Irving, and G. J. Consolmagno. The density and porosity of lunar rocks. *Geophysical Research Letters*, 39(7), 2012.

[35] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

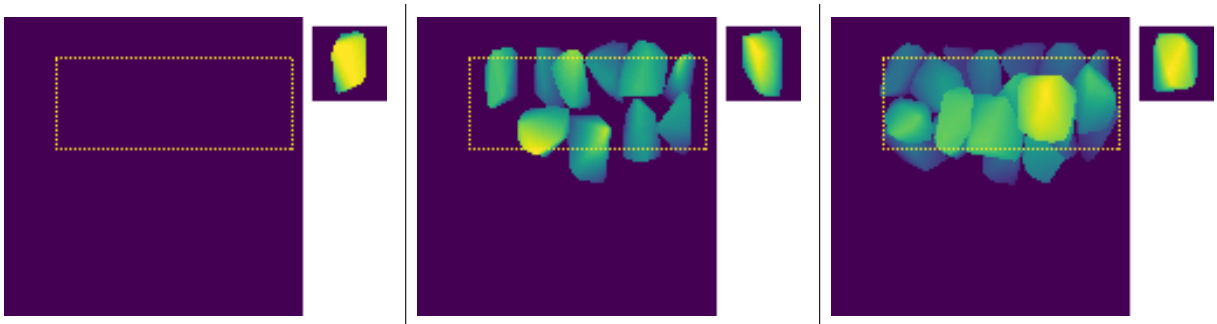
[36] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-December. IEEE Computer Society, 2016.

[37] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking Atrous Convolution for Semantic Image Segmentation. 2017.

Appendix A

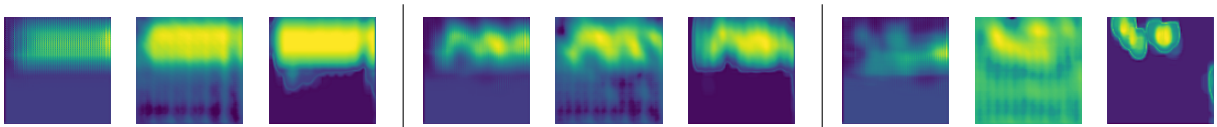
Examples of value maps

Figure A.1 represents three states of an episode ($t \in \{0, 10, 20\}$). Each of the remaining figures represents the value maps estimated by the network at different points in training (iterations 10^4, $4 \cdot 10^5$ and 10^6) for each of the represented states.



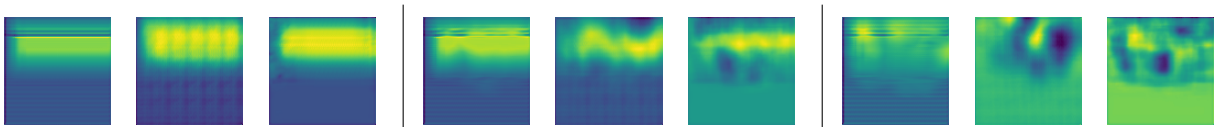
(a) $t = 0$ (b) $t = 10$ (c) $t = 20$

Figure A.1: States. Goal represented with the dotted line.



(a) 10^4 (b) $4 \cdot 10^5$ (c) 10^6 (d) 10^4 (e) $4 \cdot 10^5$ (f) 10^6 (g) 10^4 (h) $4 \cdot 10^5$ (i) 10^6

Figure A.2: Value maps from first run with IoU rewards, after the given number of iterations.



(a) 10^4 (b) $4 \cdot 10^5$ (c) 10^6 (d) 10^4 (e) $4 \cdot 10^5$ (f) 10^6 (g) 10^4 (h) $4 \cdot 10^5$ (i) 10^6

Figure A.3: Value maps from second run with IoU rewards, after the given number of iterations.

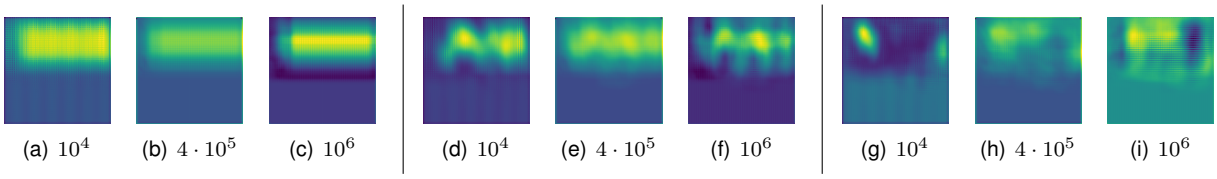


Figure A.4: Value maps from first run with OR rewards, after the given number of iterations.

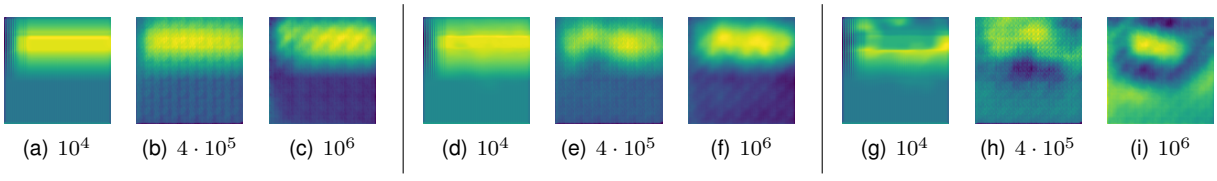


Figure A.5: Value maps from second run with OR rewards, after the given number of iterations.

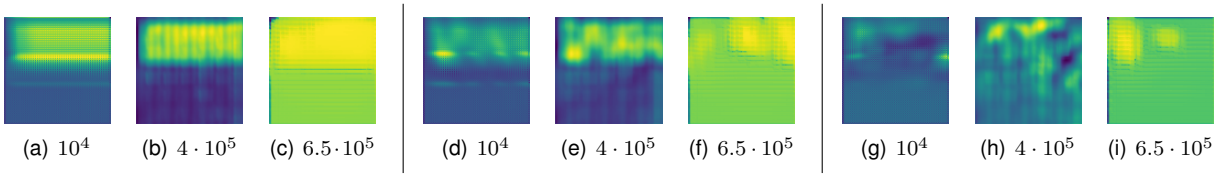


Figure A.6: Value maps from first run with DIoU rewards, after the given number of iterations. Here the third training point is at $6.5 \cdot 10^5$ iterations, just before the unrecoverable collapse of the network.

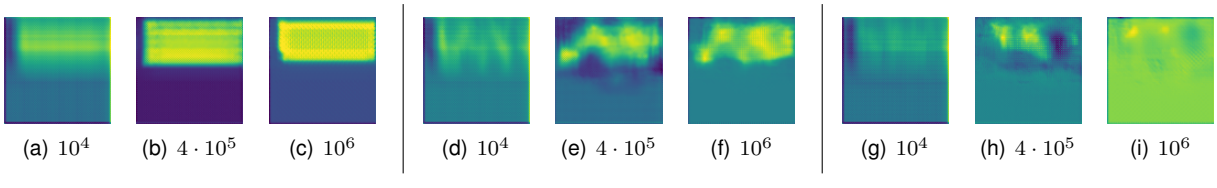


Figure A.7: Value maps from second run with DIoU rewards, after the given number of iterations.

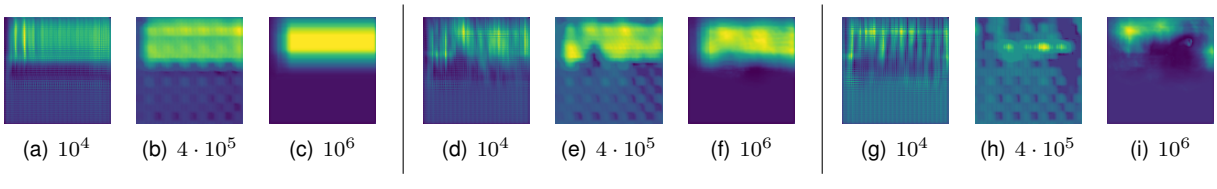


Figure A.8: Value maps from first run with DOR rewards, after the given number of iterations.

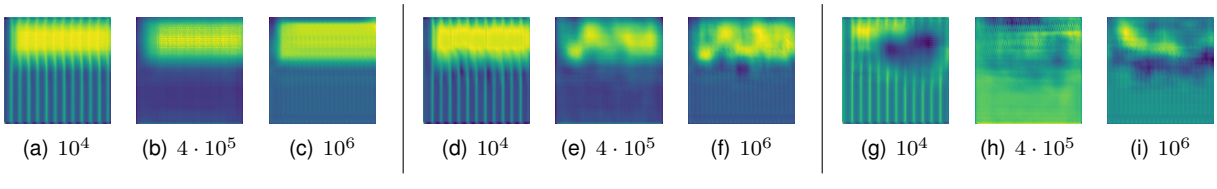


Figure A.9: Value maps from second run with DOR rewards, after the given number of iterations.

Appendix B

Examples of structures

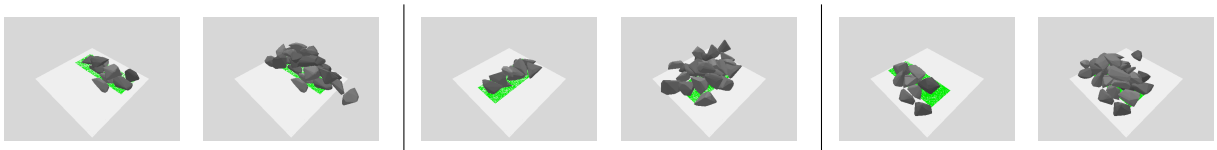
This Appendix presents examples of structures obtained with some of the considered policies, for three distinct goals and sequences of objects. Images represent the structure at time steps $t \in \{10, 30\}$.

(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.1: Structures obtained with the policy that samples actions from the target region.

(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.2: Structures obtained with the policy based on the cross-correlation heuristic.



(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.3: Structures obtained with the policy learned from IoU.

(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.4: Structures obtained with the policy learned from OR.

(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.5: Structures obtained with the policy learned from DIoU.

(a) $t = 10$ (b) $t = 30$ (c) $t = 10$ (d) $t = 30$ (e) $t = 10$ (f) $t = 30$

Figure B.6: Structures obtained with the policy learned from DOR.

Appendix C

Complete network diagram

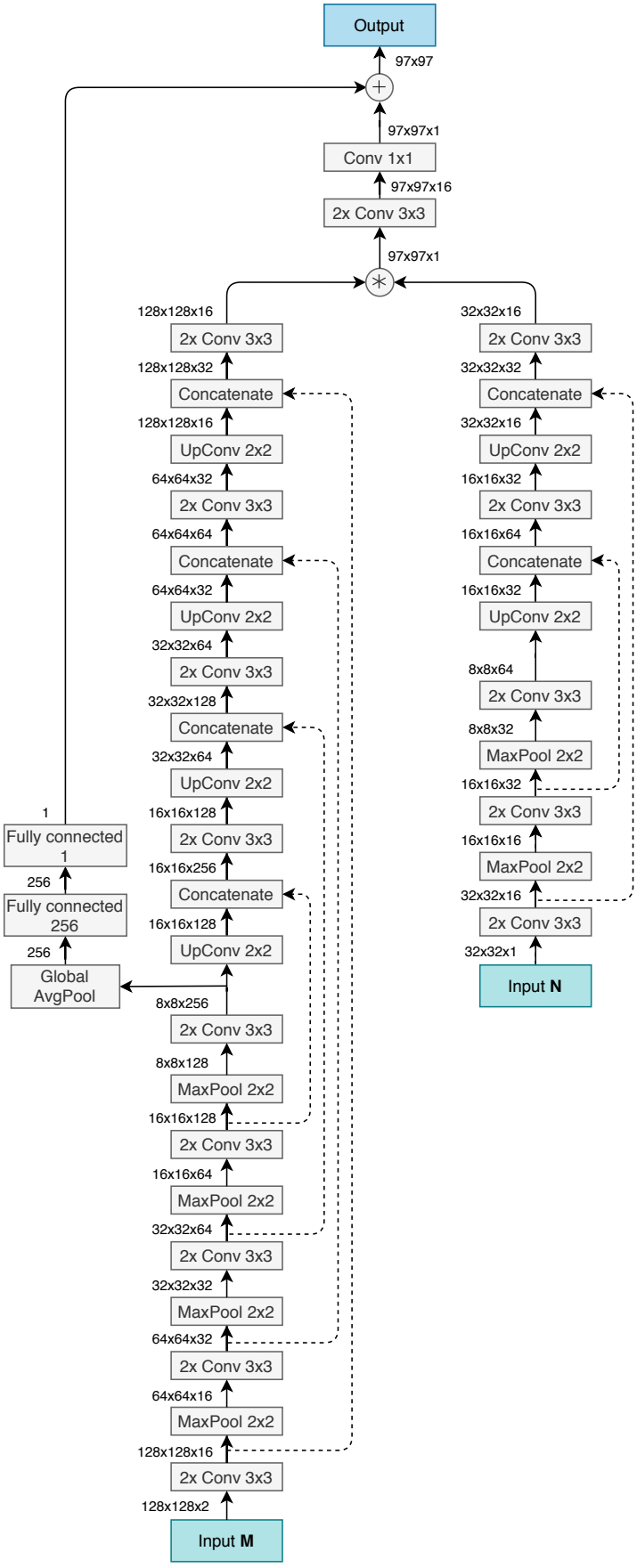


Figure C.1: Diagram of the network used as the value estimator