

TEACHING A ROBOT TO BUILD DUPLO STRUCTURES

Using Reinforcement Learning

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Jesse Bristow
1875955

Supervised by Prof Benjamin Rosman and Dr Steven James

January 28, 2025



Abstract

Consider the task of configuring a robot arm to build Duplo structures to recreate a desired goal structure. This is a challenging task because it requires long-term planning. Briefly, we solve the task in the following manner. First, we provide images of the desired goal structure we wish to recreate. Using these images, we generate a list of sequential block placing actions using a Simulation Environment and a learned model. Finally, we execute this list of actions using a YuMi robot and Duplo blocks to recreate the structure. We experiment with different design decisions such as biasing our training data as well as providing the goal structures in the form of an instruction manual to see how they affect the overall system. The instruction manual outlines some high-level steps required to recreate a goal structure by providing ordered images of intermediate structures that can be constructed to recreate the final goal structure. Our final model manages to recreate unseen goal structures to a high degree of accuracy. We also show that the use of an instruction manual is beneficial to the performance of the model. We contribute a novel solution to the task of configuring a robot arm to build Duplo structures.

Declaration

I, Jesse Bristow, declare that this thesis is my own, unaided work. It is being submitted for the Degree of Masters of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

Signed

A handwritten signature in black ink, appearing to be 'Jesse Bristow', written over a horizontal line.

on the 28th day of January 2025 in Johannesburg.

Acknowledgements

I would like to thank my supervisors Professor Benjamin Rosman and Doctor Steven James, my colleagues at the Witwatersrand RAIL lab and my family and friends for their support. I also want to acknowledge Nishen Pather, rest in peace bro.

Contents

Preface

Abstract	i
Declaration	ii
Acknowledgements	iii
Table of Contents	iv
List of Abbreviations	vi

1 Introduction 1

2 Background 4

2.1 Reinforcement Learning	4
2.2 Q-learning	7
2.3 Deep Q-Networks	9
2.4 Curriculum Learning	12
2.5 Conclusion	13

3 Methodology 14

3.1 Research Questions	15
3.2 Separation of Pipeline	15
3.3 Simulation Environment	15
3.3.1 Long-term Planning	21
3.3.2 State Space	22
3.3.3 Action Space	22
3.3.4 Reward Function	22
3.4 Network Architecture	23
3.5 Training of Agent	24
3.6 Images to Voxel Representation	25
3.7 Instruction Manuel	27
3.7.1 Training	31
3.8 Robot arm	32
3.9 Final Pipeline	36
3.9.1 Final Pipeline without Instruction Manual	36
3.9.2 Final Pipeline with Instruction Manual	38
3.10 Conclusion	39

4 Experiments 40

4.1	Hyper-Parameters	40
4.1.1	Image to Voxel Representation	40
4.1.2	Reinforcement Learning Agent	41
4.2	Metrics	41
4.3	Experiment 1	42
4.3.1	Generation of Structures without Bias	43
4.3.2	Generation of Structures with Bias	43
4.3.3	Results	43
4.3.4	Discussion of Results	45
4.4	Experiment 2	46
4.4.1	Generation of Structures	46
4.4.2	Results	47
4.4.3	Discussion of Results	50
4.5	Experiment 3	51
4.5.1	Training using Voxelised Instruction Manual	52
4.5.2	Generation of Structures	52
4.5.3	Results	53
4.5.4	Discussion of Results	55
4.6	Experiment 4	57
4.6.1	Generation of Training Data	58
4.6.2	Results	59
4.6.3	Discussion of Results	60
4.7	Experiment 5	62
4.7.1	Results	62
4.8	Conclusion	65
5	Related Work	66
6	Conclusion and Future Work	69

List of Abbreviations

AI - Artificial Intelligence

MDP - Markov Decision Process

RL - Reinforcement Learning

DQN - Deep Q-Networks

NN - Neural Network

CNN - Convolutional Neural Network

MLP - Multi-layered Perceptron

ReLU - Rectified Linear Unit

Chapter 1

Introduction

Currently AI (Artificial Intelligence) algorithms, unlike humans, struggle to solve long-term planning tasks [5]. Imagine a robot could construct furniture or basic shelters given some initial images of the desired outcome. This would improve the building capabilities of the human species and allow for massive growth. Basic housing could be constructed for those in need more efficiently and effectively, and human injury suffered during construction tasks would be reduced. In hopes of achieving this end goal, we present a method to take the first steps. We set up a YuMi robot to build Duplo structures, given images of what the structures should look like.

This is a challenging task as it requires long-term planning. We extend on previous methods that have been used to solve similar problems. One such method has been to generate LEGO structures using Generative models which produce unique, previously unseen structures [1]. Unlike this method we create a model that can replicate a given goal structure. In another approach, they create a model that can recreate a given goal structure in a 2D simulation environment [2]. We extend on this method by creating a model that can recreate given goal structures in a 3D simulation environment. A 3D simulation environment is used to represent a space in which a YuMi robot can place Duplo blocks. In an approach similar to ours, they also create a model to recreate a given goal structure in a 3D simulation environment using sequential block placing actions [3]. We extend on this approach by creating an agent that can place blocks of different sizes and shapes, build structures that are physically stable and produce lists of sequential block placing actions that the YuMi robot can physically execute given its movement constraints.

Briefly, our solution to the problem is as follows. First, images of a goal Duplo structure are read in. These images are then used to produce a voxel representation of the structure, which is essentially a 3D model of the structure. By using this voxel representation, a simulation environment and a learned model, a sequential list of block placing actions is produced which can be executed in order to recreate the goal structure. Finally, this list of actions is sent to the YuMi robot in order to rebuild the structure in real-life. We also analyse the effects of providing these images in the form of an in-

struction manual. The instruction manual outlines some high-level steps required to recreate a goal structure by providing ordered images of intermediate structures that can be constructed to recreate the final goal structure.

Below is an image showing the YuMi robot building a Duplo structure in Figure 1.1.

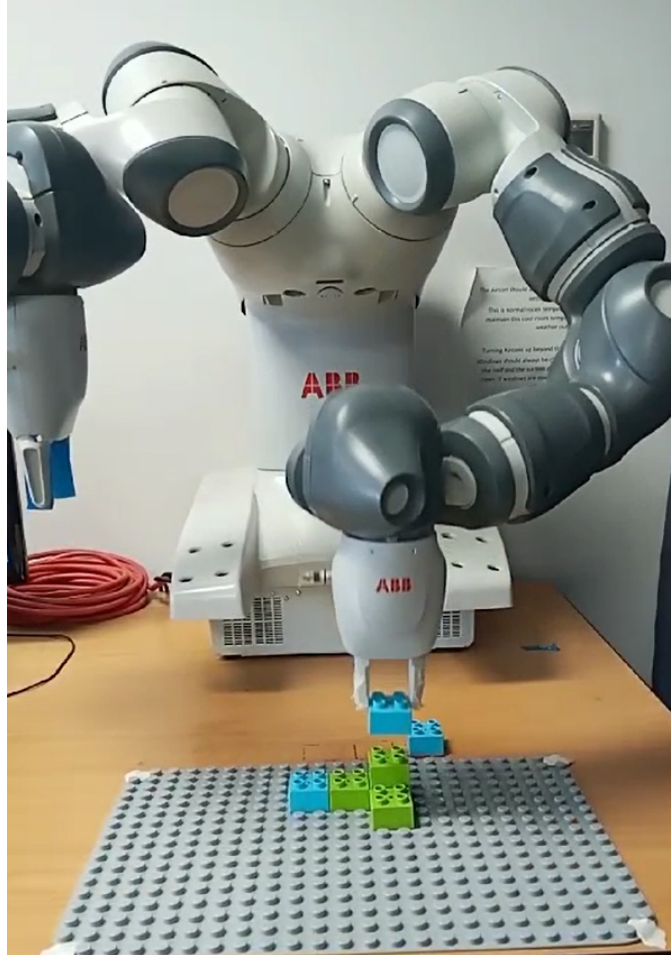


Figure 1.1: An illustration showing the YuMi robot building a Duplo structure.

Our final model manages to recreate unseen goal structures to a high degree of accuracy. We also show that the use of an instruction manual is beneficial to the performance of the model compared to when we only provide images of the final goal structure.

We contribute the following:

- A novel solution to the task of building Duplo structures using a YuMi robot given images of what the Duplo structure should look like.
- A general solution that can adapt to the YuMi robot’s physical movement constraints.

- An analysis of the use of an instruction manual to input the goal Duplo structure.

In the rest of the paper, we discuss the following. Background knowledge required to understand our solution, in the Background Chapter (Chapter 2). An detailed explanation of how we solve the problem, in the Methodology Chapter (Chapter 3). The experiments we ran and the results achieved, in the Experiments Chapter (Chapter 4). Similar problems and similar approaches to our approach and the problem we solve, in the Related Work Chapter (Chapter 5). A conclusion to highlight what we solved, the outcomes of our research and future work, in the Conclusion and Future Work Chapter (Chapter 6).

Chapter 2

Background

In this chapter, we discuss some of the background knowledge required to understand the rest of the paper. We discuss Reinforcement Learning, the Q-Learning Algorithm, the Deep Q-Network algorithm and Curriculum Learning.

2.1 Reinforcement Learning

Reinforcement learning (RL) is the main machine learning technique that we use to solve this problem. Reinforcement learning involves allowing an agent to interact with an environment and then rewarding the agent based on these interactions. An illustration showing how an agent acts within an environment is shown in Figure 2.1 below.

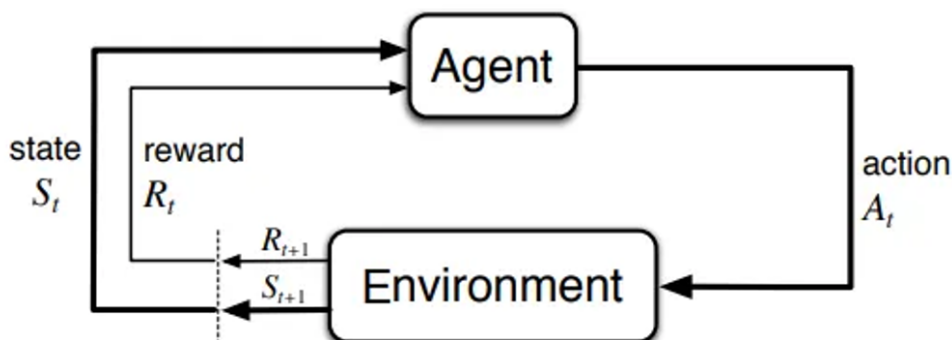


Figure 2.1: An illustration showing how an agent takes an action A_t in some state S_t in the environment, receives some reward R_{t+1} , and transitions to some state S_{t+1} [4].

A Markov Decision Process (MDP) is used to formally describe an environment for Reinforcement learning. An MDP consists of a set of states, a set of actions that can be performed in each state, a reward function that associates an expected reward with each state and action pair and finally a transition function which determines the probability

of the next state of a system given the current one and the action that was performed [4; 12; 13]. More formally an MDP can be defined as the tuple $\langle \mathbf{S}, \mathbf{A}, \mathbf{P}, \mathbf{R}, \gamma \rangle$, where:

- $\mathbf{S}$ is the set of states,
- $\mathbf{A}$ is the set of actions,
- $\mathbf{P}$ is the transition function, more formally written as $\mathbf{P}(s'|s, a) = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$, where S_t is the state at time step t , A_t is the action taken at time step t , $s' \in \mathbf{S}$, $s \in \mathbf{S}$, and $a \in \mathbf{A}$,
- $\mathbf{R}$ is the reward function, more formally written as $\mathbf{R}(s, a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$, where R_t is the reward at time step t , S_t is the state at time step t , A_t is the action taken at time step t , $s \in \mathbf{S}$, and $a \in \mathbf{A}$
- and γ is the discount factor, where $\gamma \in [0, 1]$.

These rewards then act as an incentive for the agent to act in a specific way or achieve some desired goal. The goal of the agent is to maximize the rewards it receives while interacting within the environment. In practice, the agent generally learns to maximize the discounted cumulative rewards it will receive [4]. These discounted cumulative rewards, also known as discounted returns, are the discounted sum of rewards from some time step t until the environment terminates, and can more formally be written as follows.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.1)$$

In Equation 2.1, G_t is the discounted returns at time step t , R_t is the reward at time step t , and γ is the discount factor [4]. The reason we use a discount factor is so that the returns do not tend toward infinity in a continuous environment that never terminates. The discount factor is also used to weight how important the immediate rewards should be compared to the future rewards.

A policy in RL is a mapping from states to action probabilities, and is more formally described by the following equation:

$$\pi : A \times S \rightarrow [0, 1], \pi(a|s) = \Pr(A_t = a | S_t = s) \quad (2.2)$$

In Equation 2.2, π is the policy function, A is action space, S is the state space, a is the action that is provided as input to the function, s is the state that is provided as input to the function, S_t is the state at time step t and A_t is the action taken at time step t [4]. This policy function returns the probability that the agent should choose the specific action a given that it is in state s . A policy is used to inform an agent how to act in an environment. The main problem in an RL setting is to find the optimal policy that allows the agent to accumulate the highest possible total reward in some environment [4]. To define what an optimal policy is and describe how to find it, we first need to introduce value functions. Value functions are used to predict the expected cumulative reward that an agent can obtain from a given state onwards by following a certain policy. There are two main types of value functions:

- State Value Function $V(s)$: Which estimates the expected cumulative reward that an agent can obtain given that it is in some state at some time step and is following some policy. More formally described by Equation 2.3 below [4].

$$V_{\pi}(s) = \mathbb{E}_{\pi}\{G_t | S_t = s\} \quad (2.3)$$

where π is the policy the agent is following, s is some state, G_t is the expected discounted returns at time step t and S_t is the state at time step t .

- Action Value Function $Q(s, a)$: Which estimates the expected cumulative reward that an agent can obtain given that it has taken some action in some state at some time step, and is following some policy. More formally described by Equation 2.4 below [4].

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}\{G_t | S_t = s, A_t = a\} \quad (2.4)$$

where π is the policy the agent is following, s is some state, a is some action, G_t is the expected discounted returns at time step t , S_t is the state at time step t and A_t is the action at time step t .

Policies can be partially ordered for finite MDPs in the following way [4].

$$\pi \geq \pi' \text{ if and only if } V_{\pi}(s) \geq V_{\pi'}(s) \text{ for all } s \in S$$

In other words, this states that a policy π is defined as better than or equal to another policy π' if and only if the expected return following π is greater than or equal to the expected return following π' for all states s . There will always be one or more policies that are better than all the rest and these are known as the optimal policies, usually denoted as π_* . All state value functions that follow an optimal policy are the same and are the optimal state value function. More formally written as follows [4].

$$V_*(s) = \max_{\pi} V_{\pi}(s) \quad \forall s \in S$$

The same goes for the action value functions. All action value functions that follow an optimal policy are the same and are the optimal action value function. More formally written as follows [4].

$$Q_*(s, a) = \max_{\pi} Q_{\pi}(s, a) \quad \forall s \in S, a \in A$$

Value functions are useful because any policy that acts greedily with respect to the optimal state value function or optimal action value function is an optimal policy. This means that given an optimal state value function, a policy that selects an action that leads to the next state with the highest value out of all possible next states given any current state, would be an optimal policy. Given an optimal action state function, an optimal policy could be defined as follows, in Equation 2.5 [4].

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a}{\operatorname{argmax}} Q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \quad (2.5)$$

There are many techniques and algorithms [4; 6; 14; 15; 16; 17; 18] that can be used to learn the state value functions and action value functions such as Policy iteration, Value iteration, Q-learning, SARSA, Deep Q-Networks (DQN), Policy Gradient methods, Actor-Critic methods and more. We will discuss Q-learning and DQNs in the following sections.

In some RL learning algorithms, an agent is left to act in the environment and only the state or state-action pair values associated with the states that the agent visits are updated. In order to describe how the agent is made to act within the environment, we introduce an important trade-off in RL, which is the trade-off between exploitation and exploration. This trade-off involves allowing the agent to sometimes take an action that is random instead of exploiting an action the agent believes is the best action during training. This improves the training of the model because it allows the agent to explore the action space and find new strategies, while also allowing the agent to learn more about the actions that perform well, instead of learning about poor performing actions which will not be useful. To implement exploitation versus exploration, the epsilon greedy policy [4; 19; 20] is often used which can mathematically be described as follows, in Equation 2.6.

$$A_t \leftarrow \begin{cases} \underset{a}{\operatorname{argmax}} Q_t(s, a) & \text{with probability } 1 - \epsilon \\ a \sim \operatorname{Uniform}(\{a_1 \dots a_k\}) & \text{with probability } \epsilon \end{cases}, \text{ where } \epsilon \in [0, 1] \quad (2.6)$$

In Equation 2.6, A_t is the action chosen at time step t , and Q_t is the action value function at time step t . The epsilon greedy policy allows us to select a random action with probability ϵ or the best-known action with probability $1 - \epsilon$ at time step t .

2.2 Q-learning

Q-learning is a specific algorithm used to try and learn an optimal action value function, which can then be used to derive an optimal policy as shown by Equation 2.5. The algorithm does this by learning the value of state and action pairs where the values represent the expected cumulative reward that will be achieved if that action is taken in that state. These values are referred to as Q-values. Thus, by learning these Q-values it allows the agent to perform more optimally in an environment because the agent has an idea of how much reward it will receive after taking a specific action in a specific state, and therefore it can always take the action which it presumes will result in the best cumulative reward [4; 14].

To learn these Q-values the agent is made to act in the environment by following the epsilon greedy policy, defined by Equation 2.6, for some number episodes or until some convergence criteria is met. An episode encapsulates one complete interaction cycle between the agent and the environment, from initialization to termination. For each action that is taken by the agent, the Q-value associated with that action and the state in which it was taken, is updated using an update rule. The update rule is described as follows [14].

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (2.7)$$

In Equation 2.7, Q is the action value function also known as the Q-function which maps state and action pairs to their Q-values. S_t represents some state at time step t . A_t represents some action at time step t . $Q(S_t, A_t)$ represents the Q-value of the state and action pair, (S_t, A_t) . α represents the learning rate. R_{t+1} represents the reward received at time step $t + 1$. γ represents the discount factor. S_{t+1} represents the state at the next time step $t + 1$ after taking the action A_t in state S_t . $\max_a Q(S_{t+1}, a)$ represents the Q-value associated with the state and action pair (S_{t+1}, a) , where a is the action that produces the maximum Q-value out of all possible actions that can be taken. The value $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ in the bracket can be described as the target value that we expect $Q(S_t, A_t)$ to be at time step t , and can be derived in the following way.

Equation 2.1 can be rewritten as:

$$\begin{aligned} G_t &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ \therefore G_t &= R_{t+1} + \gamma(G_{t+1}) \end{aligned} \quad (2.8)$$

Using Equation 2.4 and substituting Equation 2.8, it is seen that

$$Q_\pi(S_t, A_t) = \mathbb{E}_\pi\{G_t | S_t, A_t\} = \mathbb{E}_\pi\{R_{t+1} + \gamma(G_{t+1}) | S_t, A_t\} \quad (2.9)$$

Since

$$Q_\pi(S_{t+1}, A_{t+1}) = \mathbb{E}_\pi\{G_{t+1} | S_{t+1}, A_{t+1}\} \quad (2.10)$$

Equation 2.9 can be rewritten by substituting Equation 2.10 as

$$Q_\pi(S_t, A_t) = \mathbb{E}_\pi\{R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) | S_t, A_t\} \quad (2.11)$$

where S_{t+1} is the state transitioned to after taking action A_t in state S_t , and A_{t+1} is the action taken in S_{t+1} by following policy π . As a result, it can be said that

$$Q_\pi(S_t, A_t) \approx \mathbb{E}_\pi\{R_{t+1} + \gamma \max_a Q_\pi(S_{t+1}, a) | S_t, A_t\} \quad (2.12)$$

Thus the target value $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ in the bracket of Equation 2.7 [14] can be described as a discounted approximation to the actual cumulative reward that would be

achieved from time step t onwards. The value $Q(S_t, A_t)$ in this bracket can be described as the current Q-value at time step t . Thus, the value in this bracket represents how far off the current Q-value is from a discounted approximation to the actual cumulative reward that would have been achieved if this specific action was taken in this specific state. The sign of the value in the bracket represents if our current Q-value needs to be decreased or increased. The α value is then used to control how much the current Q-value is adjusted by, in order to get it closer to the value it is currently expected to be.

Algorithm 1 Q-learning algorithm [4]

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S$, $a \in A$, arbitrarily except that $Q(\text{terminal}, \bullet) = 0$
Loop for each episode:
 Initialize S
 Loop for each step of episode:
 Choose action A using ϵ -greedy policy
 Take action A , observe R, S'
 $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
 $S \leftarrow S'$
 until S is terminal

Algorithm 1 shows the pseudocode for the Q-learning algorithm.

2.3 Deep Q-Networks

The specific reinforcement learning algorithm we use in our approach is known as Deep Q-Network (DQN), also known as Deep Q-learning Network [6]. A DQN is based on the Q-learning algorithm, and it attempts to overcome the limitations of the basic Q-learning implementation. The basic Q-learning implementation requires a Q-value for every single state and action pair, although this can quickly become a severe limitation if either the state space required is large or the action space required is large, because then it becomes infeasible to learn so many Q-values. A DQN overcomes this limitation by using a neural network (NN) in order to approximate the Q-function. The state is given as the input and the Q-value for each possible action is generated as output. An illustration showing the comparison between Q-learning and Deep Q-learning is shown below in Figure 2.2.

Tabular Q-learning

States	Actions			
	1	2	...	n
0	$Q(0,1)$	$Q(0,2)$	...	$Q(0,n)$
1	$Q(1,1)$	$Q(1,2)$	...	$Q(1,n)$
...	...	...	...	...
m	$Q(m,1)$	$Q(m,2)$	...	$Q(m,n)$

Deep Q-learning

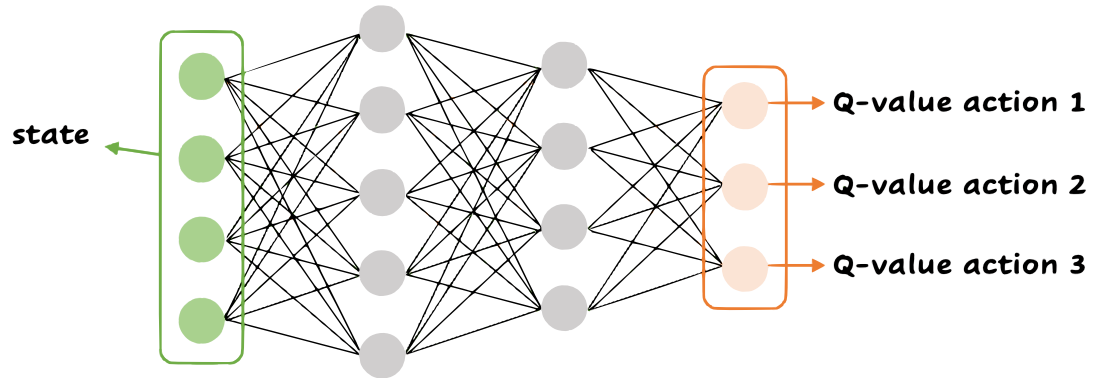


Figure 2.2: An illustration showing how a value is learnt for every state-action pair with the regular Q-learning algorithm versus how the Deep Q-learning algorithm uses a NN to approximate this table. The state is provided as input to this NN while the Q-value for each possible action is outputted by the NN.

As a result, not every state-action pair value would need to be learnt or visited because the NN learns to generalise states. This means that two similar states will learn to predict similar Q-values even if one of the states has not been explicitly encountered or learnt about previously.

The DQN algorithm is similar to Q-learning whereby the Q-values are learnt by making the agent act in the environment by following the epsilon greedy policy, defined by Equation 2.6, for some number episodes or until some convergence criteria is met. Instead of updating the Q-value associated with each action as it is taken and the state in which it was taken, like with Q-learning, the DQN algorithm samples data from a replay buffer [6; 21] in order to update the Q-values. The introduction of the replay buffer was one of the two major breakthroughs that resulted in the DQN algorithm gaining popularity. The replay buffer works as follows. For each action a taken by the agent in the environment, the action a , the state s in which the action was taken, the reward r that was received for the action taken and the next state s' that the agent transitions to after taking the action, are all stored as a tuple $\langle s, a, r, s' \rangle$ in the replay buffer. When updating the Q-values during training, a random batch of stored tuples from past agent-environment interactions is sampled from replay memory to update

the Q-values associated with each state-action pair in each tuple sampled [6]. The purpose of using replay memory is to try to ensure that the data the NN learns on is independent and identically distributed [22]. If the data was not randomly sampled from the replay memory, the NN would be training on sequential and highly correlated data which would greatly hinder the training. The use of replay memory also allows for more efficient use of previous memory as data can be learnt on multiple times. This is beneficial because the updating of Q-values is incremental and so using data repeatedly can help convergence.

As a result of using a NN, to update the Q-values or more specifically the weights of the NN, we need to define an error instead of using an update rule like with the Q-learning algorithm. Given some tuple $\langle s, a, r, s' \rangle$, the error for every output neuron not corresponding to action a will be 0 and for the output neuron corresponding to action a , the error is loosely defined as follows in Equation 2.13.

$$\text{error} = \text{target value} - \text{current value} \quad (2.13)$$

More specifically, the error for the tuple $\langle s, a, r, s' \rangle$, known as the Temporal Difference error (*td error*), is defined by Equation 2.14 when s' is the terminal state, and Equation 2.15 when s' is a non-terminal state [6].

$$\text{td error} = r - Q(s, a; \theta_t) \quad (2.14)$$

In Equation 2.14, $Q(s, a; \theta_t)$ is the Q-value returned by the output neuron corresponding to action a after inputting state s to the NN with weights θ_t at time step t . The weights θ_t are the weights for the policy network, which will be discussed below, at time step t . r corresponds to the *target value* from Equation 2.13 and $Q(s, a; \theta_t)$ corresponds to the *current value* from Equation 2.13.

$$\text{td error} = r + \gamma \max_{a'} Q(s', a'; \theta'_t) - Q(s, a; \theta_t) \quad (2.15)$$

In Equation 2.15, $\gamma \max_{a'} Q(s', a'; \theta'_t)$ is the maximum Q-value outputted by the NN with weights θ'_t at time step t , after inputting state s' . $Q(s, a; \theta_t)$ is the Q-value returned by the output neuron corresponding to action a after inputting state s to the NN with weights θ_t at time step t . The weights θ_t and θ'_t are the weights for the policy network and target network respectively, at time step t . Both the policy network and target network will be discussed below. $r + \gamma \max_{a'} Q(s', a'; \theta'_t)$ corresponds to the *target value* from Equation 2.13 and $Q(s, a; \theta_t)$ corresponds to the *current value* from Equation 2.13.

By following the reasoning behind the target value in the Q-learning section, see derivation of Equation 2.12, $r + \gamma \max_{a'} Q(s', a'; \theta'_t)$ can be used as the target value for the DQN algorithm.

The second major breakthrough that led to the DQN algorithm gaining popularity, was the introduction of a target network [6] to be used in conjunction with the policy network. The policy network is the NN that is continually updated during training and used to select actions based on state. The target network is a copy of the policy

network with frozen weights that is updated every n episodes, where n is a hyper-parameter. The target network is used to determine the target value in Equation 2.15. The purpose of the target network is to ensure that the target values remain fairly stable when optimising the policy network, and as a result increase the stability of training as a whole. If the target network were omitted, and the policy network itself was used to determine the target values, then the policy network would essentially be chasing a moving target during training, as the policy network is continually updated.

Algorithm 2 DQN algorithm [6]

Algorithm parameters: target network update frequency TU , small $\epsilon > 0$

Initialize policy network Q with random weights

Initialize target network T with a copy of Q 's weights

Initialize total steps taken $TS \leftarrow 0$

Initialize replay memory D

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose action A using ϵ -greedy policy with Q

 Take action A , observe R, S'

 Store tuple $\langle S, A, R, S' \rangle$ in D

 Sample random batch of tuples $\langle S_j, A_j, R_j, S'_j \rangle$ from D

 set $y_j = \begin{cases} R_j & \text{for terminal } S'_j \\ R_j + \gamma \max_{a'} T(S'_j, a') & \text{for non-terminal } S'_j \end{cases}$

 Perform backpropagation on Q using TD error of $(y_j - Q(S_j, A_j))$

 If $TS \bmod TU == 0$:

 Update T weights with current Q weights

$S \leftarrow S'$

$TS \leftarrow TS + 1$

 until S is terminal

Algorithm 2 shows the pseudocode for the DQN algorithm.

2.4 Curriculum Learning

In this section we discuss curriculum learning [29] which is a training strategy applied to machine learning problems. Curriculum learning is inspired by the way humans learn, whereby simpler concepts are learnt before more complex ones. In machine learning, curriculum learning is implemented by presenting training examples to a model in an sequence, starting from easier tasks and gradually progressing to more difficult ones. This methodology is different to the conventional methods whereby all training examples are presented to a model at once in a random order. By presenting the training examples in this structured manner, it can reduce the initial learning burden a model and as a result, lead to faster convergence [30]. The use of curriculum learning has also been shown to assist models, by improving their robustness. This is

due to the fact that models can learn to prevent overfitting to noise in a noisy environment by presenting simple examples first.

One challenge that exists when implementing curriculum learning is developing a metric to determine the difficulty of a training example. Several metrics that exist are as follows [29]. Model uncertainty measures how confident a model is in its predictions. Intrinsic complexity is based on feature dimensionality or noise level. Human-defined heuristics require humans with domain knowledge to rank training examples based on difficulty.

Two common approaches are used when designing a curriculum [29; 30; 31]. One approach is to design a predefined curriculum, which involves pre-defining the sequence of training examples based on certain metrics or human domain knowledge. The other approach is adaptive whereby the model itself determines the sequence of training examples based on its current performance. This approach eliminates the need for a predefined sequence of training examples but is more difficult to implement in general.

2.5 Conclusion

In conclusion we discussed Reinforcement Learning, which is the main branch of machine learning applied to solve our problem, Curriculum Learning, Q-learning and finally Deep Q-Network, which is the Reinforcement Learning algorithm applied to learn our agent. MDPs which are used to formally describe an environment for Reinforcement Learning, policy functions which are used to inform an agent how to act in an environment and value functions which can be learned in order to find optimal policy functions are discussed in Section 2.1. Section 2.2 contains a description of how optimal value functions can be learned through the use of the Q-learning algorithm. Provided within this Section are, explanations for how this algorithm works, the equations used within Q-learning and the psuedo-code for the algorithm. The DQN algorithm, which is based on the Q-learning algorithm is explained within Section 2.3. The limitations of the Q-learning algorithm that are overcome by the DQN algorithm are discussed along with a description of the experience replay buffer and target network. Psuedo-code is also provided to show how the algorithm can be implemented. A description of curriculum learning is presented in Section 2.4, along with the strengths that curriculum learning can provide. Common approaches used for assigning difficulty to training examples, as well as designing curriculums are discussed in this Section.

Chapter 3

Methodology

As stated before, training a YuMi robot to rebuild Duplo structures given images of the structure is a challenging task, because it requires long-term planning. One reason it requires long-term planning is to build *Bridge Structures* as seen in Figure 3.1 below.

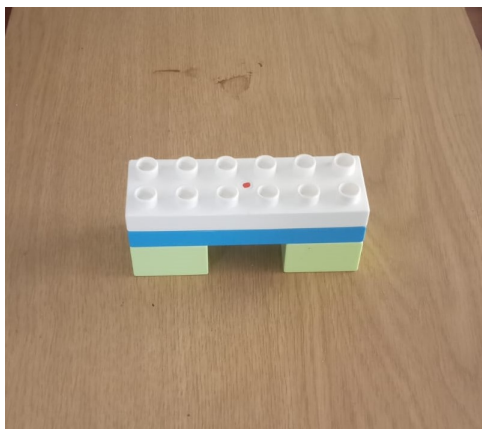


Figure 3.1: An illustration showing the *Bridge Structure*. In order to build this structure, the robot must place the green blocks first before placing the 6×2 Duplo block. Thus, this is a scenario in which the agent needs to plan into the future before taking actions.

The main task we attempt to solve is creating a novel solution for building complex structures in a simulation environment given images of a goal structure, that specifically works in conjunction with a YuMi robot given its physical constraints, as well as handle complex substructures such as the *Bridge Structure* that can exist within a goal structure. Our main experimental goal is to study the effectiveness of an instruction manual in this novel setting. In this chapter we discuss the methodology we follow in order to solve this challenging problem. We use Duplo blocks to construct these structures in real-life instead of LEGO blocks because the increase in size allows for more leeway when placing the blocks. As a result, less accuracy is required from the YuMi robot when placing blocks, which allows for more successful constructions. We also discuss the research questions we answer through our methodology.

3.1 Research Questions

We present a solution to the novel task of configuring a YuMi robot to rebuild a structure made from Duplo blocks, given images of what the structure should look like. This leads us to the following research questions;

- 1) Can an agent learn to recreate a list of unseen goal structures of stacked blocks using sequential block placing actions completely, or to a high degree of accuracy, in the simulation environment using Reinforcement Learning?
- 2) Will the use of the instruction manual improve performance compared to only providing the agent with images of the final goal structure?
- 3) Will biasing the data our RL agent trains on to include more *Bridge Structures*, as shown in Figure 3.1, allow the agent to recreate *Bridge Structures* more reliably as well as improve performance as a whole?
- 4) Can a YuMi robot be configured to execute a list of sequential block placing actions produced from the simulation environment in order to create a structure of Duplo blocks?

3.2 Separation of Pipeline

The pipeline is split into three sections during development. The first section involves developing a model that takes images of a structure and produces a voxel representation of the structure. The second section involves developing a model that takes the voxel representation of a structure and then produces a sequential list of actions to recreate the structure in a simulation environment. The third section involves configuring a YuMi robot to execute the sequential list of actions to recreate a structure using Duplo blocks. The sections are brought together in the final pipeline as shown in Figure 3.18.

3.3 Simulation Environment

We create a simulation environment to facilitate learning and assist in generating the sequential list of actions the YuMi robot executes. More specifically, this simulation environment is used to learn which blocks to place where and in what order, given some goal structure to create. Once learning has been completed, the simulation environment and a trained agent are used to determine which blocks to place where and in what order, given a goal structure to recreate. This list of actions is then sent to the YuMi robot to be executed in real-life. The physical robot arm is abstracted from the simulation environment to focus on training and inference without needing the physical robot and as a result, saving the operational life of the robot. The simulation environment allows for blocks to be placed within a 3D space, where the x -axis and y -axis go

along the horizontal plane and the z -axis is along vertical plane and represents height, as shown in Figure 3.2 below. This 3D space is stored as a 3D matrix with a 0 at some index if a block does not exist there and a 1 at some index if a block exists there. The dimensions of this 3D space are $n \times n \times n$, where $n \in \mathbb{N}$ and n is provided as a parameter during initialisation of a simulation environment.

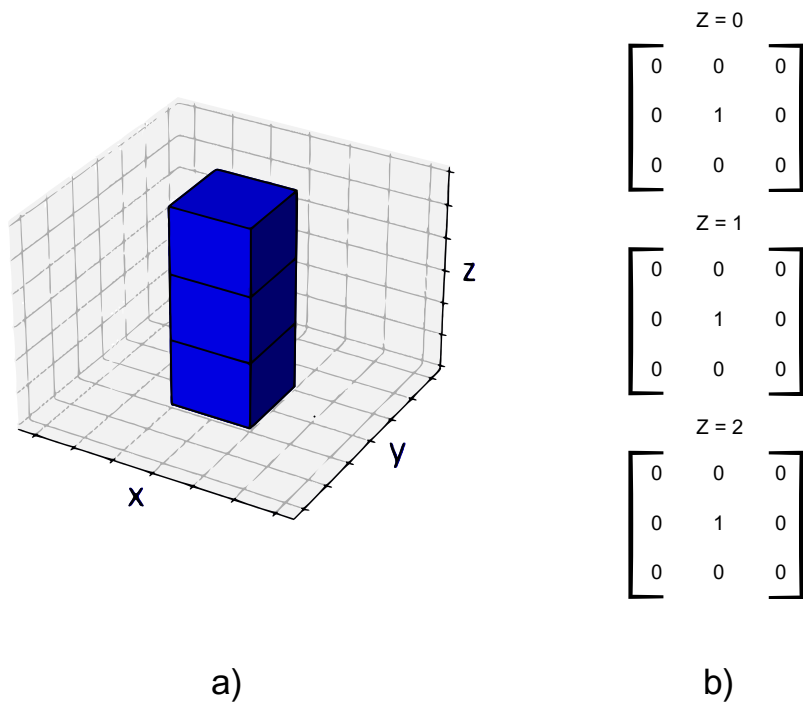


Figure 3.2: An illustration showing a representation of the 3D space in which three blocks have been stacked on top of each other. Image a) shows the column structure in the 3D space while image b) shows how this structure is stored as a 3D matrix. As shown in image a), the x -axis and y -axis go along the horizontal plane and the z -axis is along vertical plane.

The simulation environment also considers certain placement restrictions to prevent certain actions from being taken. These restrictions are based on an approximation to one of the physical robot arm's placement restrictions, if a block can be placed without breaking the structure, and if a block can be placed into the 3D space without any parts of it existing outside of the boundary of this 3D space. The physical robot arm's placement restriction we approximate is due to the fact that the YuMi robot has pincer grippers as shown in Figure 3.3 below, which prevents the robot from placing a block if blocks are already placed in areas surrounding the position we wish to place the block.



Figure 3.3: An illustration showing YuMi robot's pincer grippers.

These simulation environment placement restrictions are described in more detail as follows.

- If there is either a block placed to the left or right of (in the y -axis direction) the position where the gripper could be holding a block it is about to place, for every position at which the gripper could be holding the block when it is rotated along the y -axis, then the block cannot be placed when using the gripper rotated along the y -axis. If there is either a block placed in front of or behind (in the x -axis direction) the position where the gripper could be holding a block it is about to place, for every position at which the gripper could be holding the block when it is rotated along the x -axis, then the block cannot be placed when using the gripper rotated along the x -axis. If a block cannot be placed when using either the gripper rotated along the y -axis or x -axis, then this block is restricted from being placed in the simulation environment. This restriction is referred to as the *Gripper Restriction*. Examples of actions that violate the *Gripper Restriction* are shown in Figure 3.4 below.

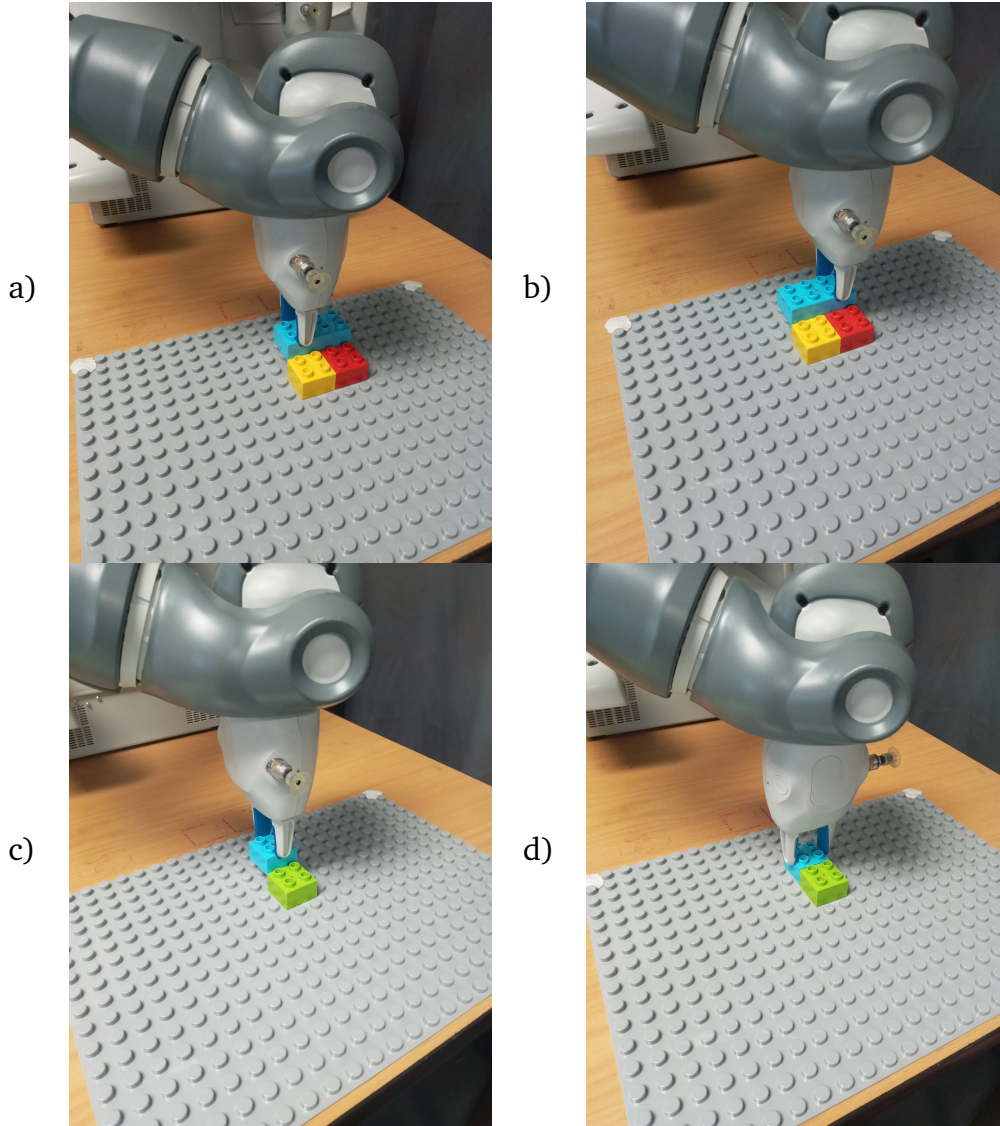


Figure 3.4: Illustration showing examples of actions that violate the *Gripper Restriction*. In images a) and b), the gripper can only pick up the 4×2 Duplo block when the gripper is rotated along the x -axis. In both positions where the gripper can pick this block up, there is a block next to this position along the x -axis. Since the block cannot be placed with the gripper rotated along the x -axis or the y -axis, this block placement action would violate the *Gripper Restriction*. Therefore, this 4×2 Duplo block placement would be prevented from being executed in the simulation environment and as a result, it would never be sent to the YuMi robot to be executed in real-life. In image c) the block cannot be placed with the gripper rotated along the x -axis because there is a block next to the placement position along the x -axis. Although in image d), the block can be placed with the gripper rotated along the y -axis because there is no block next to the placement position along the y -axis. If an action were to violate the *Gripper Restriction*, then the YuMi robot's gripper would clip an adjacent block resulting in a collision and termination of the program.

- If a block is to be placed above the ground level ($z > 0$) such that the ends of the block are not placed on top of previously placed blocks, then this block is restricted from being placed as it will prevent structures from becoming unstable and breaking. This restriction is referred to as the *Stability Restriction*. Examples of actions that will violate the *Stability Restriction* are shown in Figure 3.5 below.

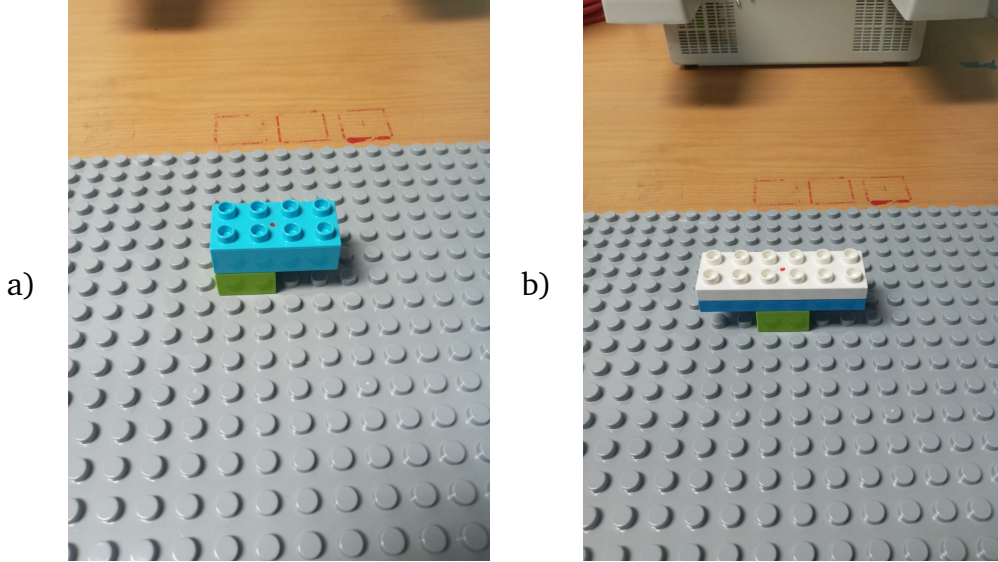


Figure 3.5: Illustration showing examples of actions that violate the *Stability Restriction*. The action taken to place the 4×2 Duplo block in image a) would violate the *Stability Restriction* because the one end of the block has not been placed on top of a previously placed block. The action taken to place the 6×2 Duplo block in image b) would also violate the *Stability Restriction* because both ends of the block have not been placed on top of a previously placed block. Therefore, both of these block placement actions would be prevented from being executed in the simulation environment and as a result, they would never be sent to the YuMi robot to be executed in real-life.

- If a block to be placed will have some parts of it existing outside the 3D space of the simulation environment once placed, the block is restricted from being placed. This restriction is referred to as the *Out of Bounds Restriction*.

The actions that can be taken in the simulation environment work as follows. Three different blocks can be placed in the simulation environment, namely a 1×1 simulation block, a 2×1 simulation block and a 3×1 simulation block. The 1×1 simulation block represents a 2×2 Duplo block, the 2×1 simulation block represents a 4×2 Duplo block and the 3×1 simulation block represents a 6×2 Duplo block. Simulation blocks of smaller dimensions are used to represent the Duplo blocks because it results in a smaller state space which benefits the RL agent training process. We only work with blocks of these sizes and as a result, we can scale the sizes down without losing any functionality. The 2×1 and 3×1 simulation blocks can be rotated and placed along the x -axis as well as the y -axis. Once placed, the 1×1 simulation block will fill one index, the 2×1 simulation block will fill two indices, and the 3×1 simulation block will fill three indices in the 3D space of our simulation environment. Different types of blocks that

can be placed in the 3D space of our simulation environment are shown in Figure 3.6 below.

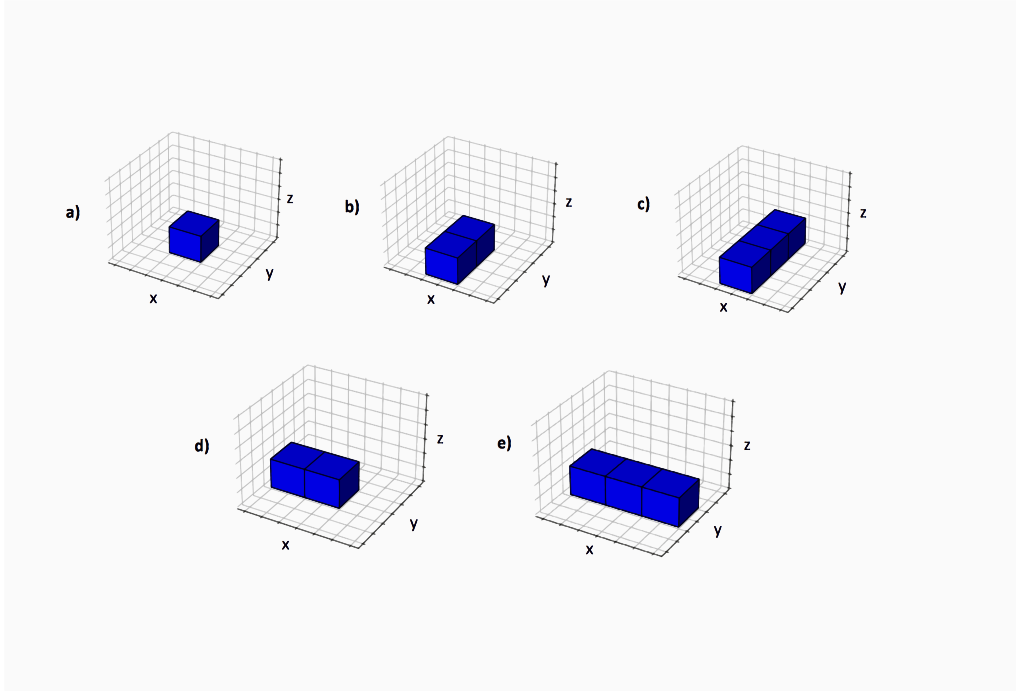


Figure 3.6: An illustration showing the different types of blocks that can be placed in the 3D space of our simulation environment, and the different rotations in which they can be placed. Image a) shows the 1×1 simulation block. Image b) shows the 2×1 simulation block placed along the y -axis. Image c) shows the 3×1 simulation block placed along the y -axis. Image d) shows the 2×1 simulation block placed along the x -axis. Image e) shows the 3×1 simulation block placed along the x -axis.

Each block, with each of its possible rotations, can be placed in the simulation environment by dropping the block from maximum height ($\max z$) at any (x, y) , and it will be placed where it lands on previously placed blocks or on the ground level ($z = 0$). Before allowing a block to be placed we check if it will violate any of the previously mentioned restrictions. If it does, then the block is prevented from being placed.

We make one further approximation to the robot’s physical restrictions. We need to make sure that the YuMi robot builds the structure from the bottom-up, as to avoid knocking the structure over if it were to place a block at a level below previously placed blocks. Blocks are not restricted from being placed in the simulation environment at a level lower than previously placed blocks. Instead, the simulation environment is used to reorder a given list of actions to ensure that the actions place blocks from the bottom-up, while still retaining the specific order in which blocks will be placed within a layer. Thus, given a goal structure to recreate, first the simulation environment and a trained agent are used to generate a sequential list of actions to recreate this structure and then this list of actions is reordered using the simulation environment, before sending the list of actions to the YuMi robot to rebuild the structure in real-life. The reordered list of actions is guaranteed to produce the same structure as the list of actions before they are

reordered. This is because for a block to be placed successfully, the only requirements are that the blocks that are meant to be placed below this block have already been placed, the blocks that are meant to be placed on the same layer but before this block have already been placed and that none of the blocks that are meant to be placed on the same layer but after this block have been placed. Thus for each block to be placed, these requirements will remain satisfied after the reordering process because blocks on lower levels will always be placed before placing some block, and the order in which blocks must be placed within a layer will be retained. Although, the reordered list of actions is only guaranteed to produce the same structure because of the blocks available for placing. If the robot had access to different types of blocks, especially blocks that span multiple layers, the reordering process would have to be re-engineered. The reason blocks within the same layer have to be placed in a specific order is discussed in the following Subsection (Subsection 3.3.1).

3.3.1 Long-term Planning

This problem requires long-term planning because it is an ordering problem. It is an ordering problem partly due to the *Gripper Restriction*. Not only is the agent required to place blocks where blocks have been placed in the goal structure, the agent is also required to place these blocks in the correct order. The reason for this is illustrated in Figure 3.7 below.

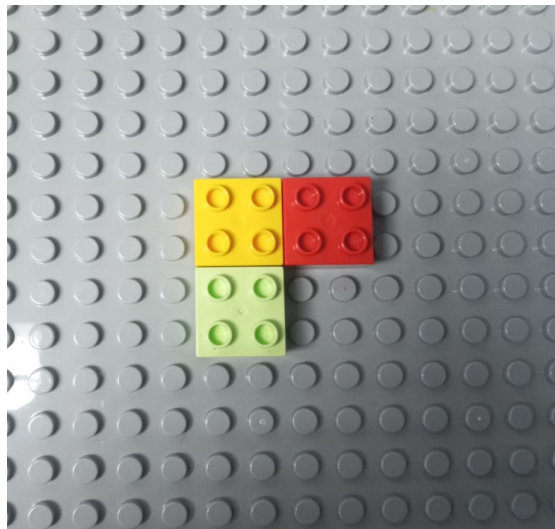


Figure 3.7: An illustration showing a structure that requires long-term planning to build. If the green block and red block were placed first and second, the YuMi robot would not be able to place the yellow block as this block placing action would violate the *Gripper Restriction*. Instead, the YuMi robot could complete the structure by placing the blocks in the following order, green block → yellow block → red block. This shows that the agent needs to plan into the future before taking any given action in order to complete a given goal structure successfully.

Another scenario that occurs, which results in this problem being an ordering problem, is when the agent is required to build the *Bridge Structure*, shown in Figure 3.1.

3.3.2 State Space

We represent our state space as a concatenation of the voxel representation of the goal structure and the voxel representation of the current structure that has been built. The voxel representation of the goal structure will be provided from the first part of the pipeline, discussed above, which will convert the images of the goal structure into voxel representation. The voxel representation of the current structure built is extracted from the simulation environment as the 3D space in which the structures are built is stored as a 3D matrix. The dimensions of these voxel representations will be $n \times n \times n$, where the input images are of resolution $n \times n$.

3.3.3 Action Space

An action will consist of placing a specific type of block at a specific location, with a specific rotation, in the simulation environment. A discrete action space is used. The action space will consist of the following actions. An action for every block that can be placed, at every rotation it can be placed with, and at every (x, y) location it can be placed in our simulation environment. Actions that will violate the *Out of Bounds Restriction* are omitted. As mentioned before, an action consists of dropping the block from maximum height ($\max z$) at any (x, y) , and will be placed where it lands on previously placed blocks or on the ground level ($z = 0$), in the simulation environment.

The total number actions in the action space will be $t \times m \times n - r$, where t is the number of different blocks that can be placed at different rotations, m is the number of x locations in the simulation environment, n is the number of y locations in the simulation environment and r is the number of actions that violate the *Out of Bounds Restriction*. Each action is represented by a single integer, thus the action space is described more formally as shown in Equation 3.1.

$$A = \{a | 0 \leq a < t \times m \times n - r, a \in \mathbb{Z}\} \quad (3.1)$$

3.3.4 Reward Function

Given an action taken in the simulation environment, the voxel representation of the goal structure, and the voxel representation of the current structure built, we provide the following rewards.

- +1 for each index filled by placing a block in the current structure built voxel representation, that is also filled in the goal structure voxel representation. This reward is referred to as *Reward Correct*
- -2 for each index filled by placing a block in the current structure built voxel representation, that is not filled in the goal structure voxel representation. This reward is referred to as *Reward Incorrect*

- -3 if the action taken violates either the *Gripper Restriction*, or the *Stability Restriction*. This reward is referred to as *Reward Invalid*
- -0.5 for taking an action. This reward is referred to as *Reward Step*

The *Reward Correct* provides an incentive to the agent to place blocks in correct positions with respect to the goal structure. The *Reward Incorrect* provides an incentive to the agent to not place blocks in incorrect positions with respect to the goal structure. The *Reward Invalid* provides an incentive to the agent to not take invalid actions that would be restricted by the simulation environment. The *Reward Step* provides an incentive to the agent to use as few actions as possible in order to recreate a goal structure, which in turn encourages the agent to prioritise using the 3×1 simulation blocks over 2×1 simulation blocks and 1×1 simulation blocks, and prioritise using the 2×1 simulation blocks over 1×1 simulation blocks.

In summary, the use of the reward function defined above allows us to learn an agent that can place blocks in correct positions and in the correct order.

3.4 Network Architecture

The architecture of the NN that is learnt for our RL agent is illustrated in Figure 3.8 below.

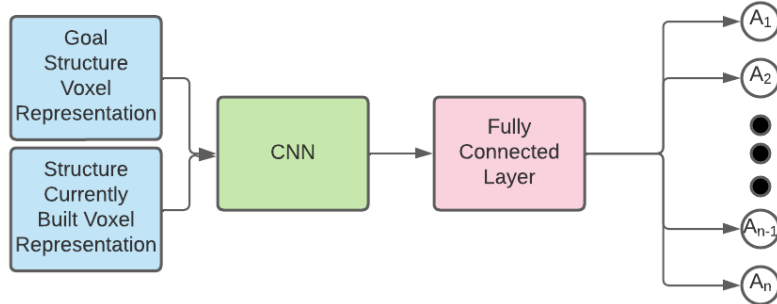


Figure 3.8: An illustration showing the architecture of the NN that is learnt for our RL agent. The state consisting of a concatenation of the goal structure and currently built structures voxel representations is provided as input to the network. The input is first fed through a Convolutional Neural Network (CNN) before going through a fully connected layer which finally provides data to the output neurons. Each output neuron corresponds to a specific action and outputs the Q-value associated with this action and inputted state.

The DQN algorithm is used to learn this network as the DQN algorithm is known to work well in scenarios with discrete actions spaces.

The architecture of the NN we use for our RL agent is as follows. We input a $n \times n \times n \times 2$ image, which is a concatenation of the voxel representation of some goal structure and

the voxel representation of the structure currently built, where both of these voxel representations are of size $n \times n \times n$. The first hidden layer convolves 16 $5 \times 5 \times 5$ filters with stride 1 with the input image and then applies 3D Batch Normalisation to the output. The second hidden layer convolves 32 $3 \times 3 \times 3$ filters with stride 1 and again applies 3D Batch Normalisation to the output. The third hidden layer convolves 32 $1 \times 1 \times 1$ filters with stride 1 and again applies 3D Batch Normalisation to the output. The output layer is a fully-connected linear layer with a single output for each possible action the agent can take. The rectified linear unit (ReLU) activation function [32] is applied after each layer except the output layer.

3.5 Training of Agent

The training of our agent in the simulation environment is performed as follows. Goal structures in voxel representation will be randomly generated for training. For one episode of training, a random goal structure will be selected from the generated list. The agent will attempt to recreate this goal structure in the simulation environment while simultaneously optimising its networks (performing DQN learning update step) to perform better. The agent will attempt to recreate the goal structure and optimise the NN as follows. Step one is to construct the state as a concatenation of the goal structure voxel representation and the voxel representation of the structure currently built. From here, an action to take given the current state is determined using the NN being learned. Then, this action is executed in the simulation environment and we receive the next state and a boolean stating whether this next state is the terminal state or not. Finally, the NN is optimised by performing the DQN learning update step. If the next state received is the terminal state, then the agent has finished its attempt at recreating the goal structure and completed the episode. If the next state received is a non-terminal state, then the process is repeated from step one onwards. The agent will train for n episodes. These steps are illustrated in Figure 3.9 below.

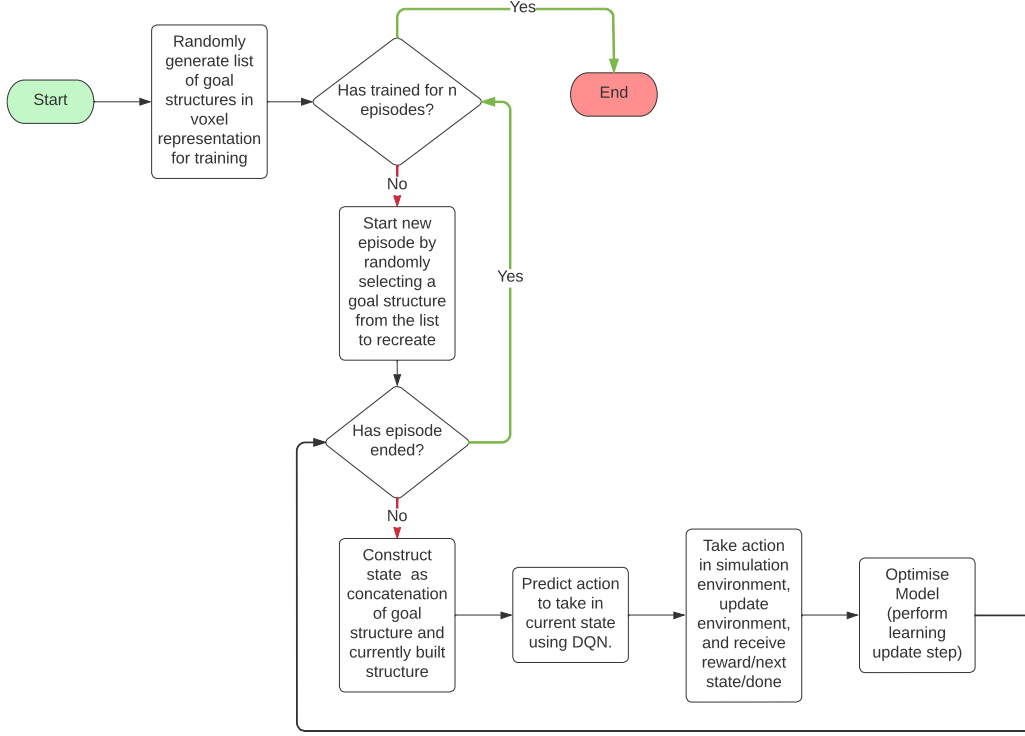


Figure 3.9: Flowchart showing the steps taken to train the agent in the simulation environment.

3.6 Images to Voxel Representation

As mentioned before, the first part of the pipeline involves converting images of some structure into a voxel representation of that structure. This problem is solved by training a NN that takes in a concatenation of the images as input and outputs the voxel representation. For a structure, three images are provided, namely a front, left and top-down view of the structure. These images can be produced by generating a random 3D structure and then extracting the images from the different views, or by manually creating them. A random 3D structure is generated by executing random block placing actions in the simulation environment until some number of indices have been filled within the 3D space of the simulation environment. Each image is a binary image that contains a 0 at some pixel, if no block exists at that position and a 1 at some pixel, if a block exists at that position. Each pixel corresponds to a single block. Although binary images are not ideal for real-life application, they are used as a first step on the path toward using natural images. The use of natural images is future work and has been approached in numerous ways already [33; 34]. The resolution of an image depends on the size of the environment in which we wish to build the structure. The voxel representation produced is a 3D matrix with a 0 at some index if a block does not exist there and a 1 at some index if a block exists there. This approach has some shortcomings, whereby not all structures can be inferred accurately from input images. The reason for this is because many 3D structures that share the same front, left and

top-down views exist. In the next Section we discuss instruction manuals, which can be used to remedy this issue. An illustration of a structure in voxel representation form that has been perfectly constructed given binary images of the structure is shown in Figure 3.10 below.

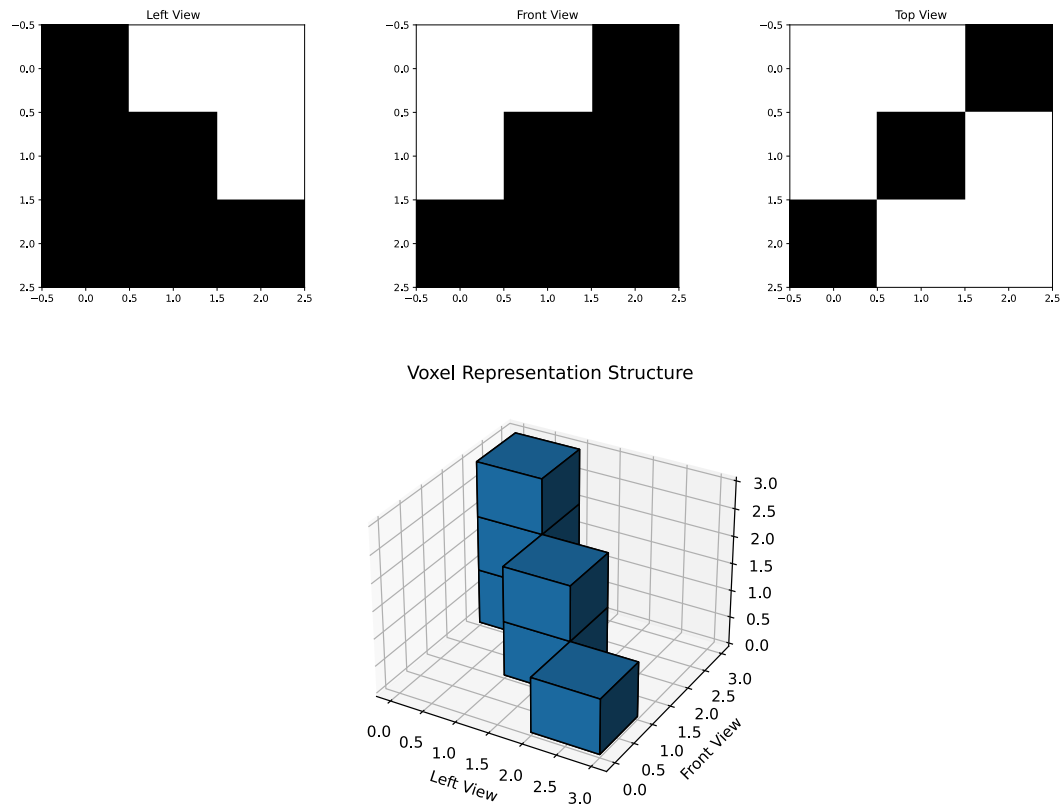


Figure 3.10: An Illustration showing a structure in voxel representation form that has been perfectly constructed given binary images of the structure. Ideally, the NN we train to convert images of a structure into voxel representation would read in the Left View, Front View and Top View as seen in this image, and produce the voxel representation structure shown.

An illustration of the NN that is learnt is shown in Figure 3.11 below.

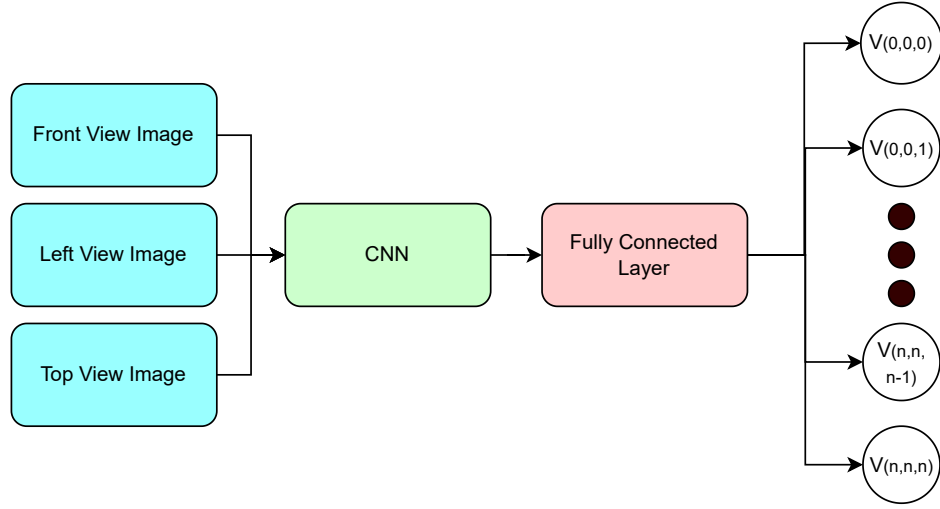


Figure 3.11: An Illustration showing the NN that is trained to convert images of a structure into a voxel representation of the structure. The input consists of a concatenation of all three images. Each output neuron corresponds to a specific index in the voxel representation and the value that is entered into the voxel representation will be the final value of the output neuron rounded to either 0 or 1.

The architecture of the NN that converts images to voxel representation is as follows. We input a $n \times n \times 3$ image, which is a concatenation of the front-view, left-view and top-view images of some goal structure, where each of these images is of resolution $n \times n$. The first hidden layer convolves 16 5×5 filters with stride 1 with the input image and then applies 2D Batch Normalisation to the output. The second hidden layer convolves 32 3×3 filters with stride 1 and again applies 2D Batch Normalisation to the output. The third hidden layer convolves 32 1×1 filters with stride 1 and again applies 2D Batch Normalisation to the output. The output layer is a fully-connected linear layer with a single output for each of the $n \times n \times n$ indices in the voxel representation. The rectified linear unit (ReLU) activation function is applied after each layer except the output layer. As stated above, we round the values outputted in the final layer to 0 or 1.

We refer to a model that converts images of some structure into a voxel representation of this structure as an image to voxel model.

3.7 Instruction Manual

An instruction manual contains information on a goal structure we wish the robot to build. More specifically, the instruction manual consists of the intermediate structures

that can be built to rebuild a final goal structure. The intermediate structures are provided in the order in which they need to be built. An instruction manual essentially highlights high level steps required to build a final goal structure. As such, the instruction manual can be seen as a form of curriculum learning. Each intermediate structure consists of n blocks, where n is some interval number. Each intermediate structure is described by three binary images within an instruction manual, namely a front, a left and a top-down view image of the structure. The instruction manual can be used to remedy the issue whereby not all 3D structures can be fully described by only three binary images. The reason for this is because individual parts of a structure can be described by intermediate structures within an instruction manual. An example of what an instruction manual could look like is illustrated in Figure 3.12 below.

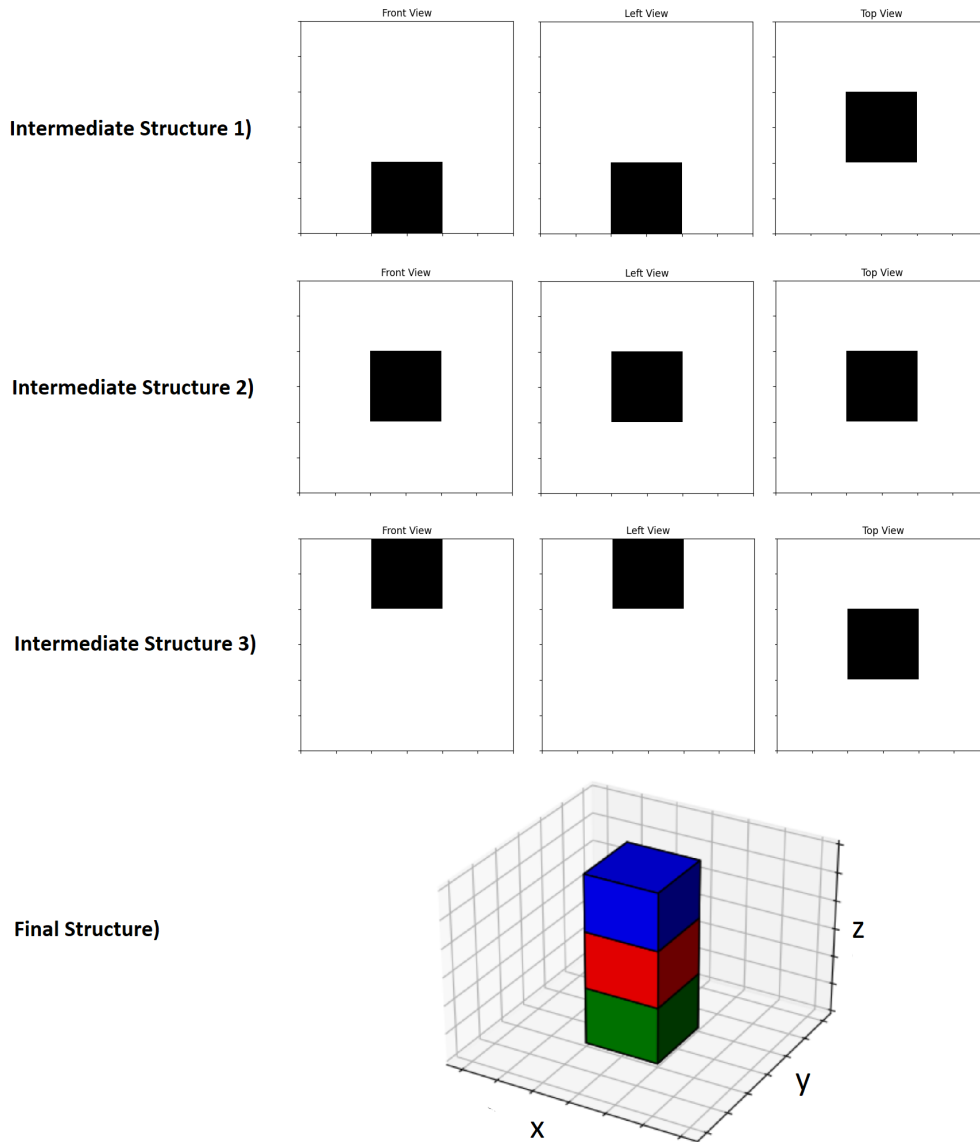


Figure 3.12: An illustration showing an example of an instruction manual. The intermediate structures images provide steps at interval number of 1 to produce the final structure. In this example, the green block will be placed if the first intermediate structure is completed perfectly, the red block will be placed if the second intermediate structure is completed perfectly and the blue block will be placed if the third intermediate structure is completed perfectly.

The instruction manual is used in the pipeline by converting it into a voxelised instruction manual which then is used for subsequent sections in the pipeline. A voxelised instruction manual consists of intermediate structures in voxel representation form which are constructed using the intermediate structures in an instruction manual in

the following way. The n^{th} intermediate structure in the voxelised instruction manual is created by converting the first n intermediate structures in the instruction manual into voxel representation, using a trained image to voxel model, and then appending these voxel representations into a single voxel representation. In other words, a structure at some position in the voxelised instruction manual will be the structure that would have been created if all the steps in the instruction manual had been followed perfectly up until that point. An example of an instruction manual that has been converted into a voxelised instruction manual is shown in Figure 3.13 below.

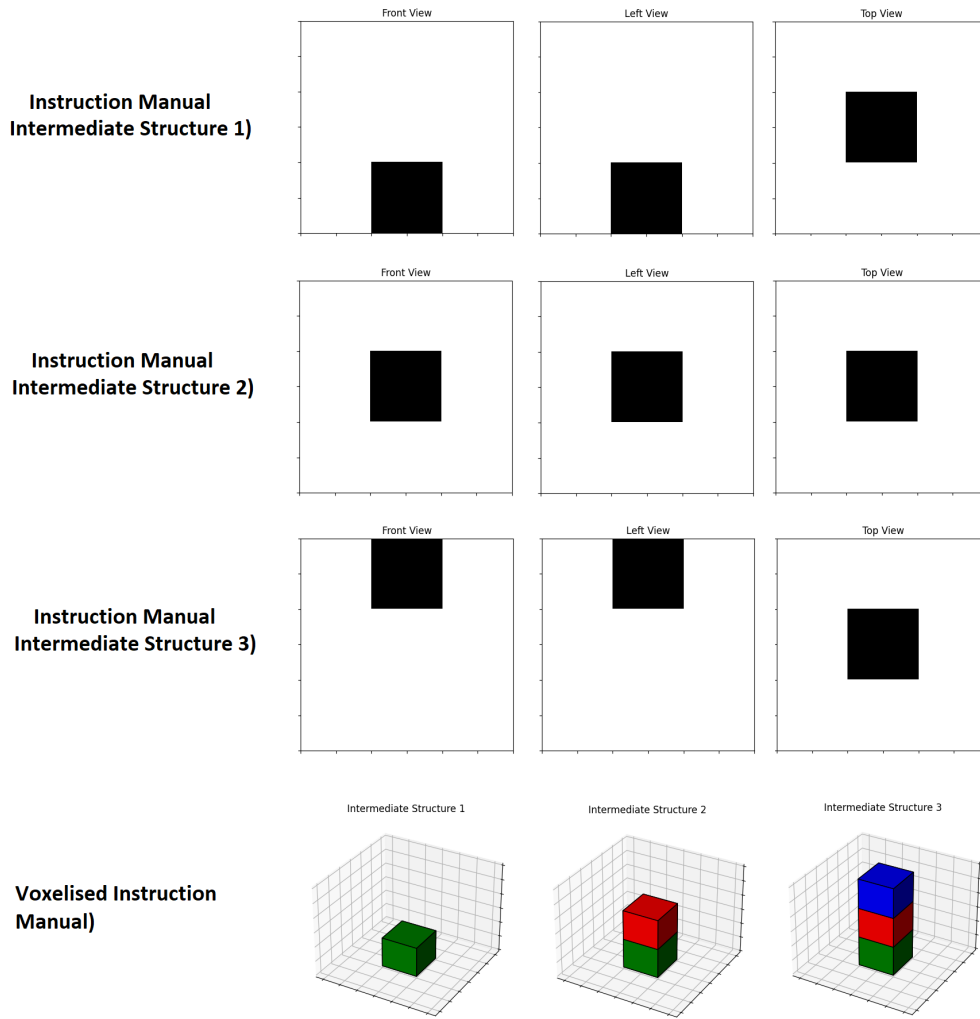


Figure 3.13: An illustration showing an instruction manual that has been converted into a voxelised instruction manual. Using the three intermediate structures represented by images in the instruction manual, the three intermediate structures in the voxelised instruction manual are constructed by following the steps above.

3.7.1 Training

To train this NN a dataset is needed. We create data in the following way. First, some random 3D structures are created in our simulation environment which are in the voxel representation format. For each structure the front, left and top-down view images are determined. Each data-point in our dataset will correspond to a single structure where

the features will be these images, and the labels will be the voxel representations of the structures. Finally, the NN is trained on this dataset for n episodes, where n is a hyper-parameter.

3.8 Robot arm

We setup a YuMi robot to place Duplo blocks according to a given list of actions in order to build a Duplo structure. The robot is controlled through the use of the python library called yumipy. This library allows us to change joint angles, open and close the grippers, as well as move the grippers to specified co-ordinates.

The space in which the robot can place blocks is represented by three axes, namely the x -axis, y -axis and z -axis, as shown in Figure 3.14 below.

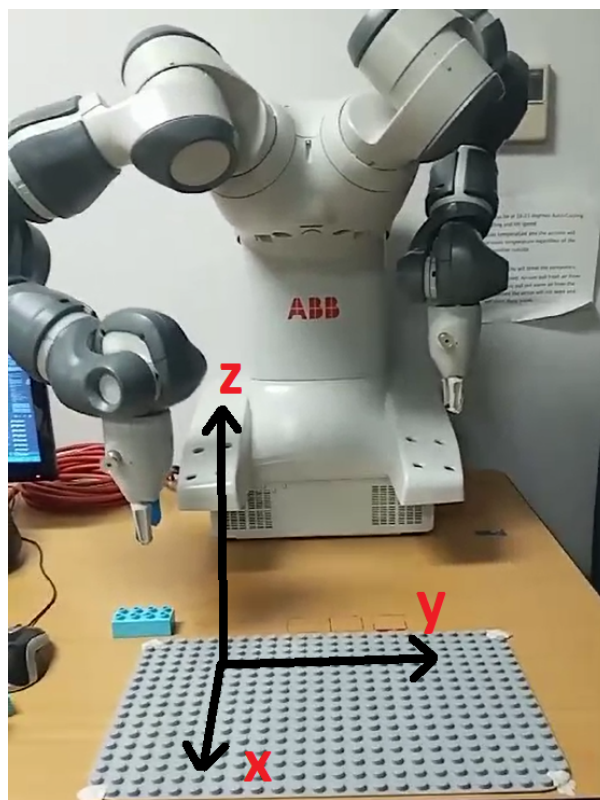


Figure 3.14: An Illustration showing the axes that represent the space in which the robot can place blocks. The x -axis runs along the horizontal plane parallel to the direction the robot is facing. The y -axis runs along the horizontal plane perpendicular to the direction the robot is facing. The z -axis runs along the vertical plane.

The robot can place 6×2 , 4×2 and 2×2 Duplo blocks. All blocks that are to be placed are put in specific positions within reach of the robot, as shown in Figure 3.15 below.

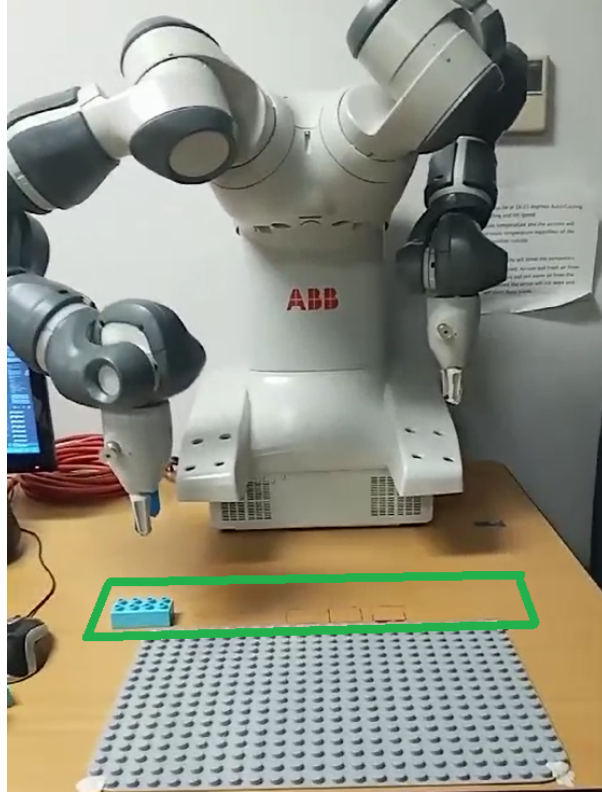


Figure 3.15: An Illustration showing the area in which blocks to be placed must be positioned prior to being picked up. Blocks to be placed are put in hard-coded positions within the enclosed green rectangle.

The coordinates of these positions are hard-coded so that the robot can always pick up blocks that are to be placed reliably, and with a high degree of accuracy. Positions at which blocks can be placed are also hard-coded so that the blocks can be placed on the Duplo mat, or on top of previously placed blocks reliably and accurately. If these placement positions were not hard-coded, blocks to be placed would not line up accurately with the Duplo mat or previously placed blocks, and as a result, would not clip into the Duplo mat or previously placed blocks.

When placing a block, we first determine where along the block we must position the gripper to pick it up. The rotation of the gripper required to place the block is also determined. The gripper can be rotated such that the pincers align with the x -axis, or with the y -axis.

For 2×2 Duplo blocks, the gripper is always positioned in line with the centre of the block before picking it up. If no blocks have been placed immediately next to the position at which we plan to place the block along the x -axis, then the block is placed with the pincer aligned along the x -axis. If no blocks have been placed immediately next to the position at which we plan to place the block along the y -axis, then the block is placed with the pincer aligned along the y -axis.

For 4×2 Duplo blocks, the block is picked up at one of the two ends. To check if the block can be picked up at some end we do the following. If the block is to be placed with the

long side running along the y -axis, and no blocks have been placed immediately next to the position at which the end will be placed along the x -axis, then the block can be picked up at this end. If the block is to be placed with the long side running along the x -axis, and no blocks have been placed immediately next to the position at which the end will be placed along the y -axis, then the block can be picked up at this end. The block is picked up at the first feasible end found. Before placing the block, the gripper is rotated depending on the rotation required to place the block with. If the block is to be placed with the long side running along the y -axis, then the block is placed with the pincer aligned along the x -axis. If the block is to be placed with the long side running along the x -axis, then the block is placed with the pincer aligned along the y -axis.

For 6×2 Duplo blocks, the block is picked up at one of the two ends or in the middle. To check if the block can be picked up at some end or in the middle we do the following. Let k denote the end or middle we are checking. If the block is to be placed with the long side running along the y -axis, and no blocks have been placed immediately next to the position at which k will be placed along the x -axis, then the block can be picked up at k . If the block is to be placed with the long side running along the x -axis, and no blocks have been placed immediately next to the position at which k will be placed along the y -axis, then the block can be picked up at k . The block is picked up at the first feasible position found. Before placing the block, the gripper is rotated depending on the rotation required to place the block with. If the block is to be placed with the long side running along the y -axis, then the block is placed with the pincer aligned along the x -axis. If the block is to be placed with the long side running along the x -axis, then the block is placed with the pincer aligned along the y -axis.

Examples of how a gripper will pick up a block are shown in Figure 3.16 below.

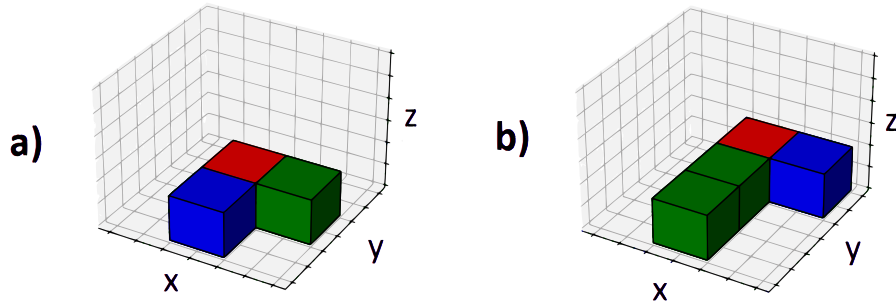


Figure 3.16: An illustration showing where the YuMi robot would pick blocks up in order to place them. In image a) we are analysing the 2×1 simulation block (which represents a 4×2 Duplo block) that is the combination of the red block and the green block. This 2×1 block is placed with the long side going along the x -axis, therefore the YuMi robot would place this block with its gripper rotated along the y -axis. The YuMi robot cannot place the block by holding it on the red end because of the adjacent blue block along the y -axis. The YuMi would place the block by holding it on the green end as there are no blocks adjacent to this position along the y -axis. In image b) we are analysing the 3×1 simulation block (which represents a 6×2 Duplo block) that is the combination of the red block and the green blocks. This 3×1 block is placed with the long side going along the y -axis, therefore the YuMi robot would place this block with its gripper rotated along the x -axis. The YuMi robot cannot place the block by holding it on the red end because of the adjacent blue block along the x -axis. The YuMi would place the block by holding it on the green end or in the middle as there are no blocks adjacent to these positions along the x -axis.

We are guaranteed that there will always be at least one feasible way in which we can place any given block, due to that fact that the simulation environment restricts actions that violate the *Gripper Restriction*. If the above checks for gripper rotation and position at which to pick a block up were not done, there would be a risk of a gripper pincer clipping a block placed adjacent to the position we plan to place a block, resulting in a collision and termination of the building process.

For a single action from the list of actions to execute, the position the block is meant to be placed in is determined in the same way it is done in the simulation environment. Essentially, the position the block would land in if it were to drop the block from the (x, y) position specified by the action is determined. From here, the coordinate of this position in the space in which the robot can place blocks is calculated. Finally, a block

is picked up from a hard-coded position where a known unplaced block is, and then placed at this coordinate. This is done for each action in the list of actions in order.

Below the YuMi robot is shown placing a 4×2 Duplo block in the Figure 3.17.

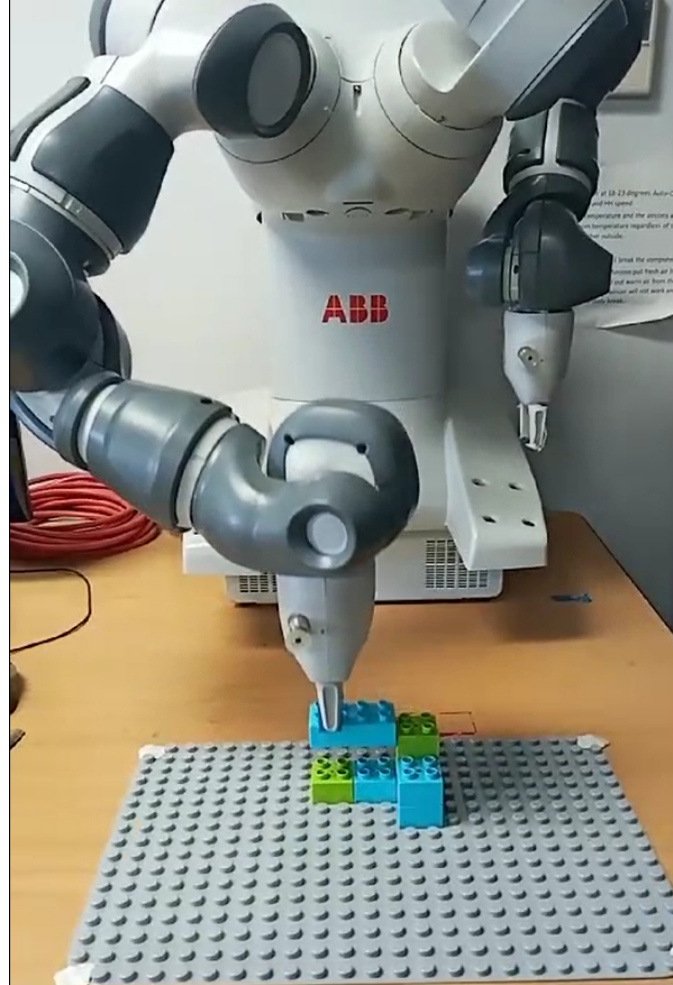


Figure 3.17: An illustration showing the YuMi robot place a 4×2 Duplo block. The robot is gripping the block at the first feasible end found. The first feasible end is found by following the steps described above.

3.9 Final Pipeline

In this Section we discuss how the final pipeline flows with and without the use of instruction manuals.

3.9.1 Final Pipeline without Instruction Manual

The final pipeline without the use of the instruction manual works as follows. First, images of a goal structure are provided. These images of the goal structure are then converted into voxel representation using a trained image to voxel model. A trained

agent is then used as follows to recreate the goal structure as accurately as possible in the simulation environment. The following steps are repeated until the agent has placed as many blocks as there are in the goal structure. Step one is to construct the state as a concatenation of the goal structure voxel representation and the voxel representation of the structure currently built. From here an action to take given the current state is determined, using the learnt NN. Then this action is executed in the simulation environment and we receive the next state and a boolean stating whether this next state is the terminal state or not. Finally, the action taken is appended to a list of all actions currently taken to recreate this goal structure. If the next state received is the terminal state, then the agent has finished its attempt at recreating the goal structure and the list of actions taken can be sent to the YuMi robot to rebuild the goal structure in real-life. A state is considered terminal once the model has filled as many indices in the structure it is building as there are indices filled in the provided structure, or after more actions have been taken than there are indices filled in the provided structure. If the next state received is a non-terminal state, then the process is repeated from step one onwards. These steps are illustrated in Figure 3.18 below.

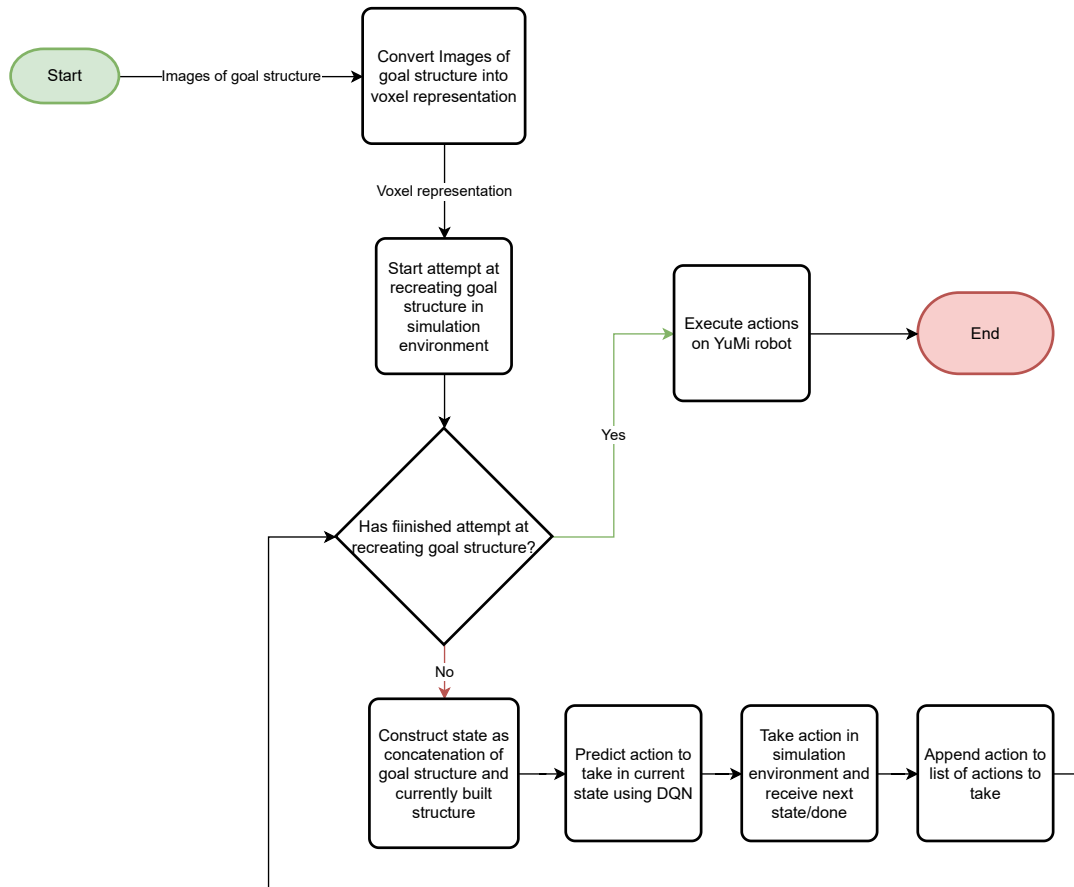


Figure 3.18: Flowchart showing the steps that would be taken during the use of the completed framework without the use of the instruction manual.

3.9.2 Final Pipeline with Instruction Manual

In order to describe how the final pipeline works with the use of instruction manuals, we first provide the steps required to recreate a structure in voxel representation form within the Simulation Environment. The steps are the same as the steps illustrated in Figure 3.18 except without the need to convert images into voxel representation or execute the list of actions using the YuMi robot, as shown in Figure 3.19 below.

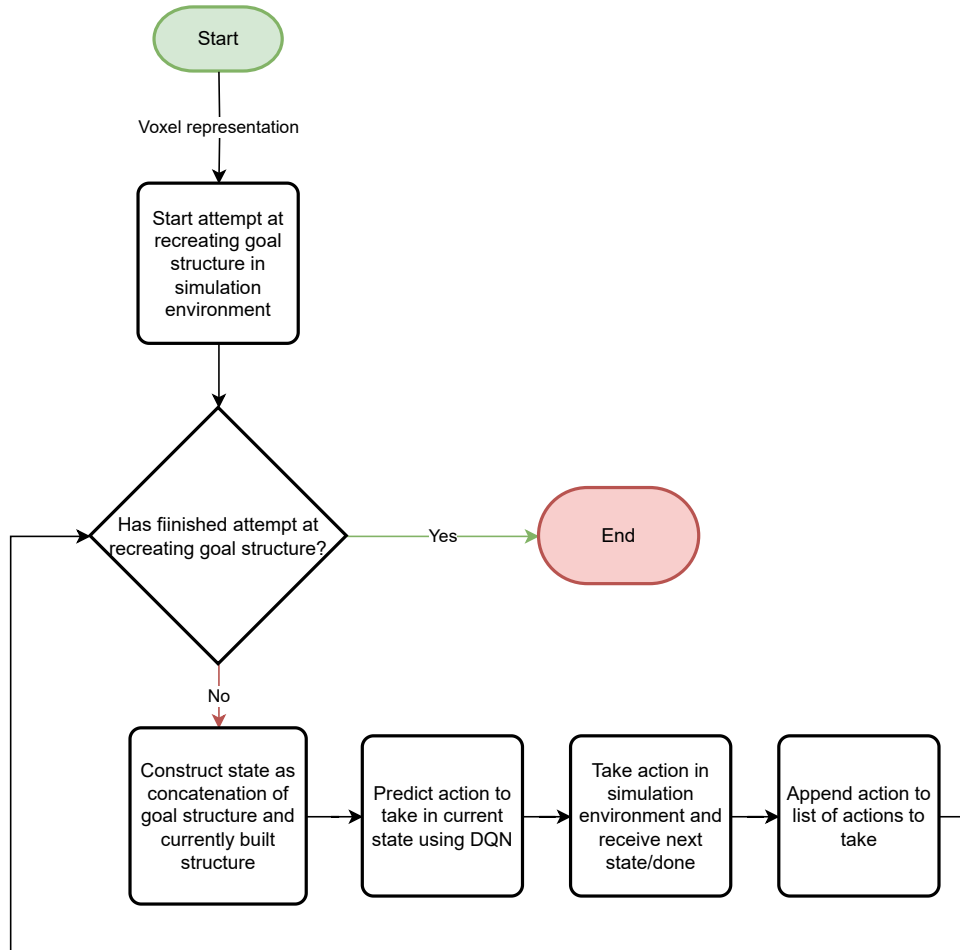


Figure 3.19: Flowchart showing the steps that would be taken to recreate a structure in voxel representation within the simulation environment.

The final pipeline works with the use of instruction manuals in one of the following two ways.

- The instruction manual is converted into a voxelised instruction manual and then the last structure in the voxelised instruction manual is extracted. The last structure in any voxelised instruction manual will be the full voxelised structure described by the instruction manual. From here, an attempt to recreate this structure

is performed by following the steps shown in Figure 3.19. Once the structure has been rebuilt in the simulation environment, the list of actions taken can be sent to the YuMi robot to rebuild the goal structure in real-life. We refer to this approach as *Method 1* throughout the rest of the document.

- The instruction manual is converted into a voxelised instruction manual. From here, we provide the model with the first structure in the voxelised instruction manual to recreate by following the steps shown in Figure 3.19. Once the model has recreated this structure, we provide the model with the next structure in the voxelised instruction manual. The last step is then repeated until the model has recreated the last structure within the voxelised instruction manual. Finally, the list of all actions taken to recreate these structures can be sent to the YuMi robot to rebuild the goal structure in real-life. We refer to this approach as *Method 2* throughout the rest of the document.

3.10 Conclusion

In conclusion we discussed each section within the pipeline, as well as how the NNs are trained. We also discussed how the instruction manual works and how the final pipeline flows.

Chapter 4

Experiments

In this chapter we present and discuss the results of our research. We experiment with biasing the data we train our RL agent on to include more *Bridge Structures*, as shown in Figure 3.1, in Experiment 1 (Section 4.3). An analysis on how well an agent learns to recreate structures of different sizes is provided in Experiment 2 (Section 4.4). An experiment on the use of instruction manuals abstracted from the image to voxel model is provided in Experiment 3 (Section 4.5). An experiment on the use of instruction manuals when the image to voxel model and our RL agent are used in conjunction is provided in Experiment 4 (Section 4.6), and lastly, how well the YuMi robot can build structures of Duplo blocks is tested in Experiment 5 (Section 4.7).

4.1 Hyper-Parameters

In this section we present the hyper-parameters that we use for all the experiments that follow. In Section 4.1.1, the hyper-parameters that are used during the training of the NN that converts images to voxel representation are presented. In Section 4.1.2, the hyper-parameters that are used during the training of our RL agent are presented.

4.1.1 Image to Voxel Representation

The hyper-parameters used during the training of the NN that converts images to voxel representation are shown in Table 4.1 below.

Hyper-parameter	Value
Loss Function	MSE
Optimizer	SGD
Learning Rate	0.1
Episodes	100000

Table 4.1: Table showing hyper-parameters used when training image to voxel models.

4.1.2 Reinforcement Learning Agent

The hyper-parameters used during training of our RL agent are shown in Table 4.2 below.

Hyper-parameter	Value
Loss Function	SmoothL1Loss
Optimizer	SGD
Learning Rate	0.0001
Batch Size	128
Target Update	5000000
Replay Memory Size	10000
Exploration Rate (ϵ)	0.2
Discount Factor (γ)	0.99999

Table 4.2: Table showing hyper-parameters used when training RL agents.

An exploration rate (ϵ) of 0 is used during inference for all experiments below, unless otherwise stated.

4.2 Metrics

We assess the performance of an agent given some goal structure and the structure built by the agent to recreate the goal structure, by using the following metrics.

- Given a goal structure in voxel representation and a structure built in voxel representation, we calculate an accuracy metric with bias removed, described more formally by Equation 4.1.

$$accuracy = \frac{c}{t - e} \quad (4.1)$$

In Equation 4.1, c represents the number of indices that are filled in the both the goal structure voxel representation and the structure built voxel representation. t represents the total number of indices in either the goal structure voxel representation or the structure built voxel representation, as both voxel representations contain the same number of indices. e represents the number of indices that are not filled in both the goal structure voxel representation and the structure built voxel representation. This accuracy metric allows us to compare how accurately goal structures of different sizes have been recreated while removing the bias introduced by the open spaces around both the goal structure and the structure built. If the accuracy were calculated by dividing the total number of indices that are both filled or both not filled in the goal structure voxel representation and the structure built voxel representation, by the total number of indices in one of these voxel representations, then the accuracy for smaller sized goal structures will inherently be larger due to having more indices not filled in both the goal structure voxel representation and the structure built voxel representation.

- Given a goal structure and a structure built, we determine the number of *Bridge Structures* in the goal structure that have been successfully recreated in the structure built. We are interested in this metric because it gives us an indication of how well an RL agent has learnt to plan into the future. The reason building these *Bridge Structures* requires long-term planning is described in Subsection 3.3.1.

4.3 Experiment 1

In this experiment we test if biasing the data we train our RL agent on to include more *Bridge Structures*, as shown in Figure 3.1, will improve an agents ability to recreate *Bridge Structures* in the simulation environment, and thus improve performance of an agent as a whole based the accuracy Equation 4.1.

Three sets of random goal structures are generated for this experiment. One set of random goal structures is generated without bias, and referred to as the *non-biased training structures*. The other two sets of random goal structures are generated with bias, and referred to as the *biased training structures* and *biased testing structures* respectively. One model is trained on the *non-biased training structures*, and referred to as the *non-biased model*. Another model is trained on the *biased training structures*, and referred to as the *biased model*. Both the *non-biased model* and *biased model* are tested on the *biased testing structures* to compare results.

An illustration of a random goal structure is presented in Figure 4.1 below.

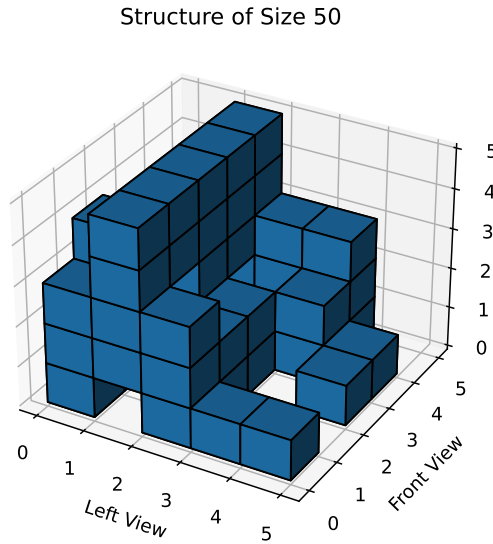


Figure 4.1: Figure showing a random goal structure. Structures generated with bias and without bias will look similar, except structures generated with bias will contain more *Bridge Structures*.

4.3.1 Generation of Structures without Bias

The *non-biased training structures* are generated as follows. 5000 random structures are generated where the voxel representation of every structure is of size $5 \times 5 \times 5$. A random structure is generated by executing random block placing actions in the simulation environment until 50 indices have been filled within the 3D space of the simulation environment.

4.3.2 Generation of Structures with Bias

The *biased training structures* and *biased testing structures* are generated as follows. 5000 random structures are generated for the *biased training structures*, and 2500 random structures for the *biased testing structures*. The voxel representation of every structure is of size $5 \times 5 \times 5$. A random structure is generated in the 3D space of the simulation environment in the following way. If there exists an action that will complete a *Bridge Structure*, a random action out of all possible actions that will complete a *Bridge Structure* is executed. If no actions will complete a *Bridge Structure*, then any random action is executed. Actions are performed in the manner described above until 50 indices have been filled within the 3D space of the simulation environment. By performing actions in the manner described above we prioritise creating *Bridge Structures* where possible, thus creating structures with more *Bridge Structures* than if actions were performed completely at random. The aim of creating structures in this manner is to provide the agent with more opportunities to encounter a *Bridge Structure* during training, thus allowing it to learn how to recreate these structures faster and to a higher degree of accuracy on average.

In Table 4.3, the total number of filled indices and total number of *Bridge Structures* that exist over all the structures within the *non-biased training structures*, *biased training structures* and *biased testing structures* are shown.

Structures	Total Indices Filled	Total Bridge Structures
Non-Biased Training	250000	7562
Biased Training	250000	29490
Biased Testing	125000	14586

Table 4.3: Table showing showing the total number of filled indices and total number of *Bridge Structures* that exist over all the structures within the *non-biased training structures*, *biased training structures* and *biased testing structures*.

4.3.3 Results

We average results over three different runs for both the *non-biased model* and the *biased model*. For each run, an agent is trained in the simulation environment for 80000000 time steps on its respective training structures. Every 100000 time steps the model we are training is used to rebuild every structure from the *biased testing structures* and record the following results. The average accuracy over the *biased testing structures*, using the Equation 4.1 to calculate the accuracy for single testing structure.

The total number of successful *Bridge Structures* recreated over the total number of *Bridge Structures* that exist in the *biased testing structures* as a percentage. These results are presented in the Figures 4.2 and 4.3 below.

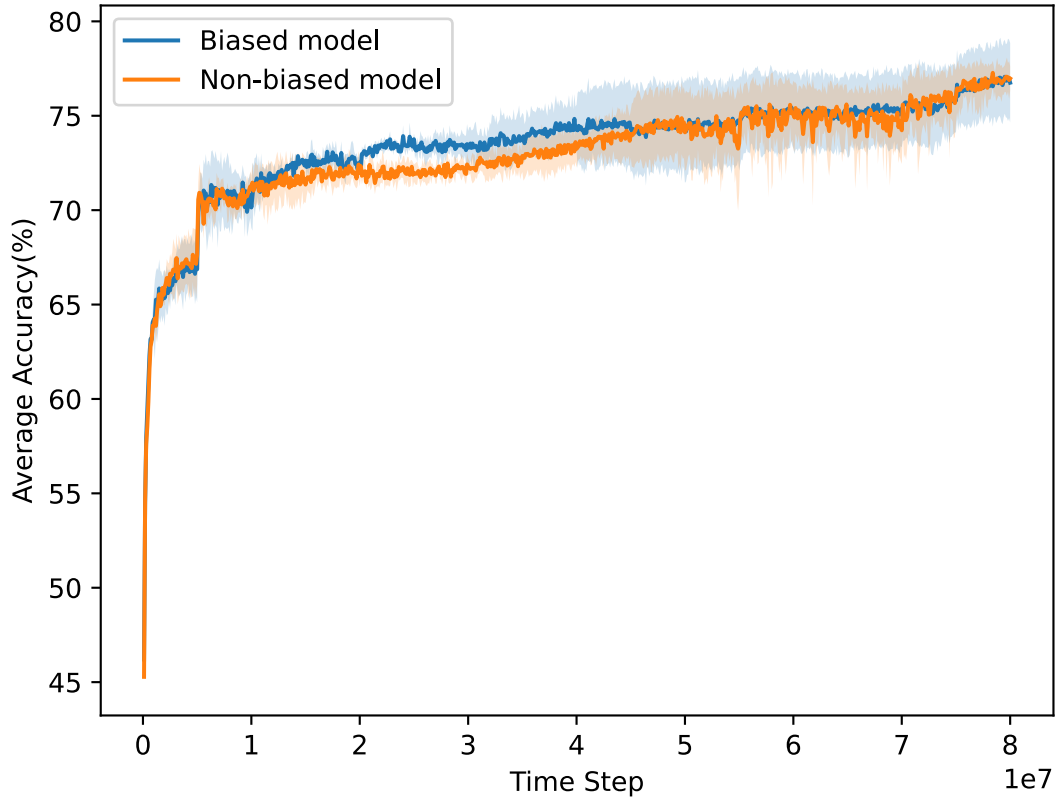


Figure 4.2: Figure showing average accuracy as a percentage versus time step during training for the *non-biased model* and the *biased model*. The shading in this figure represents the average error over the three training runs of a model which is calculated as follows. Each time the accuracy for each run is recorded, we calculate the absolute value of the difference between this accuracy and the mean of the accuracies over all runs at this time step. Finally, these differences are averaged to get the average error at this time step.

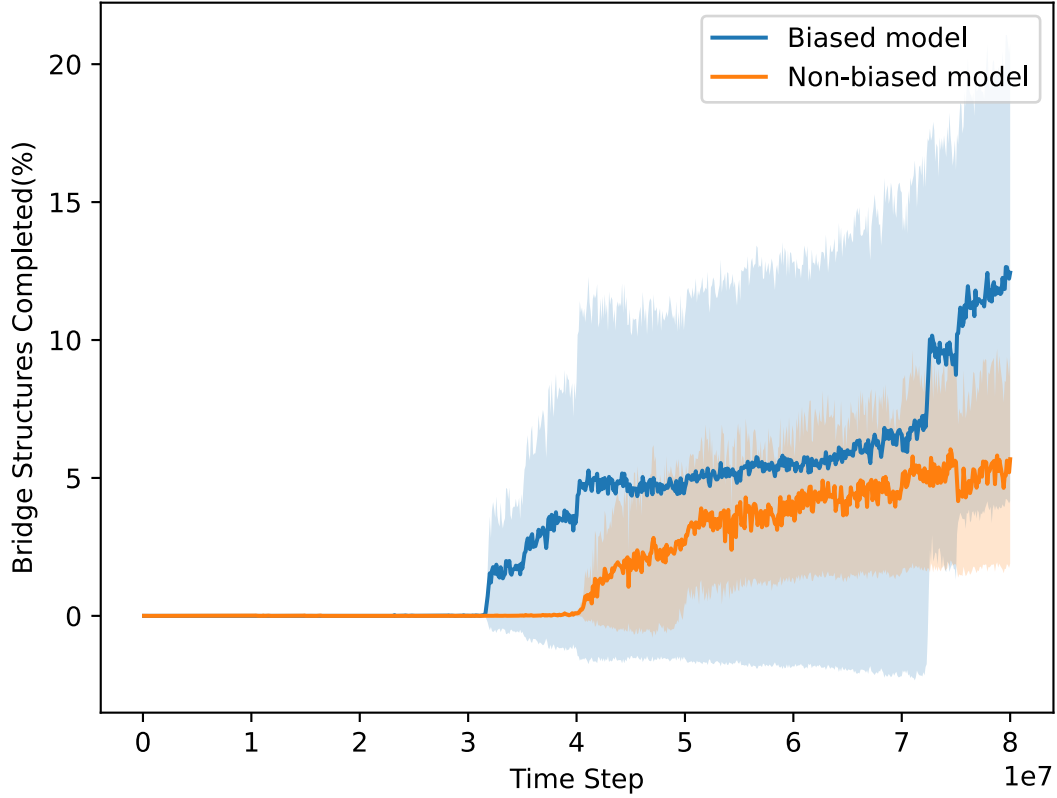


Figure 4.3: Figure showing percentage of total *Bridge Structures* completed successfully versus time step during training for the *non-biased model* and the *biased model*. The shading in this figure represents the average error over the three training runs of a model which is calculated as follows. Each time the percentage of *Bridge Structures* completed for each run is recorded, we calculate the absolute value of the difference between this percentage and the mean of the percentages over all runs at this time step. Finally, these differences are averaged to get the average error at this time step.

4.3.4 Discussion of Results

In Figure 4.2, there is no noticeable difference in the end performance of the *biased model* versus the *non-biased model* after training, based on accuracy. Both models learn at about the same rate on average and with about the same consistency. We note this based on the fact that both learning curves increase at about the same rate on average and with the same average error over all the training runs.

In Figure 4.3, the *biased model* manages to complete *Bridge Structures* more reliably compared to the *non-biased model* after training. The *biased model* learns to complete its first *Bridge Structure* faster than the *non-biased model* on average based on the fact that the learning curve spikes for the first time from about 3×10^7 time steps for the *biased model*, and from about 4×10^7 time steps for the *non-biased model*. The *biased*

model learns to complete *Bridge Structures* faster than the *non-biased model* on average based on the fact that the learning curve of the *biased model* is always above the learning curve of the *non-biased model* after about 3×10^7 time steps.

Although the *biased model* manages to complete *Bridge Structures* more reliably compared to the *non-biased model*, there is no significant difference in the end performance between the two models based on accuracy most likely due to the following reason. The amount of *Bridge Structures* is a lot less than the amount of blocks in total that exist over all the *biased testing structures*, as seen in Table 4.3, and both models recreate less than 13% of the total *Bridge Structures* that exist.

We conclude that it can be beneficial to bias the data we train our RL agent on to include more *Bridge Structures*. Although the overall performance does not increase based on accuracy, the RL agent can learn to recreate *Bridge Structures* faster, and more reliably if it is trained on data that is biased to have more *Bridge Structures* compared to data generated completely at random.

4.4 Experiment 2

Having established that our RL agent can recreate structures to a high degree of accuracy as seen in Experiment 1 (Section 4.3), in this experiment we consider the effect of training models on goal structures of a different sizes, based on the accuracy Equation 4.1.

We train three different models on sets of random goal structures. These models are tested on random unseen goal structures. One model is trained and tested on structures of size 25, another model on structures of size 50, and one last model on structures of size 75. A structure of size n is a structure that has a voxel representation with n indices filled.

4.4.1 Generation of Structures

To generate training and testing structures of size n we do the following. 5000 random goal structures are generated for the training structures, and 2500 random goal structures for the testing structures. These goal structures are generated in the same way the goal structures are generated in Subsection 4.3.2, i.e. with bias toward *Bridge Structures*, except actions are taken until n indices have been filled within the 3D space of the simulation.

Training and testing structures of size 25, 50 and 75 are generated in the manner described above. Examples of these structures are shown in Figure 4.4 below.

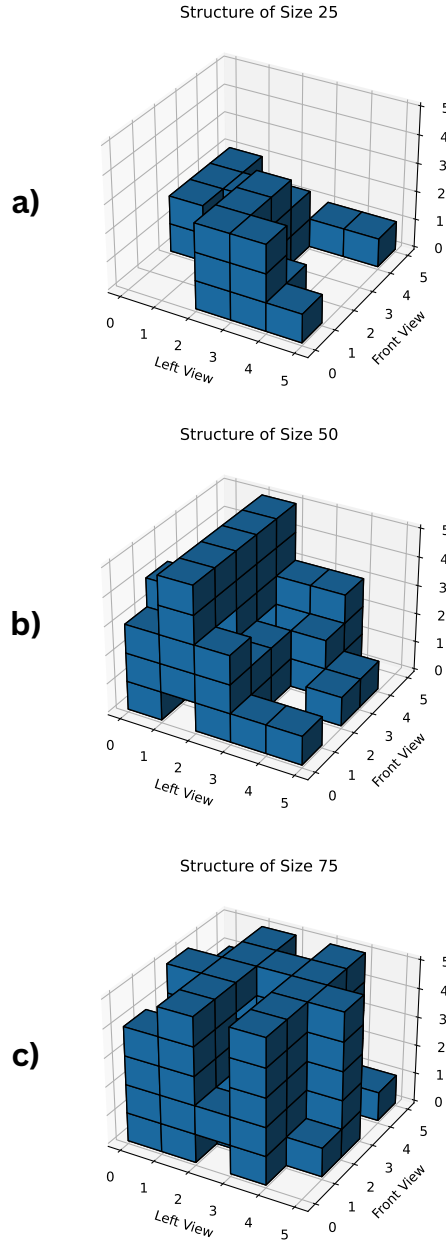


Figure 4.4: An illustration showing structures of different size. Image a) shows a structure of size 25, image b) shows a structure of size 50 and image c) shows a structure of size 75. 5000 training structures and 2500 testing structures are generated for each size.

4.4.2 Results

We average results over three different runs for all the models. For each run, an agent is trained in the simulation environment for 80000000 time steps on its respective training structures. Every 100000 time steps the model we are training is used to rebuild every structure from its respective testing structures and record the following results. The

average accuracy over its respective testing structures, using the Equation 4.1 to calculate the accuracy for a single testing structure. The total number of successful *Bridge Structures* recreated over the total number of *Bridge Structures* that exist within its respective testing structures as a percentage. These results are presented in the Figures 4.5 and 4.6 below.

After the models have been trained, each model is used to recreate each structure from its respective testing structures. Each structure created is compared to its corresponding testing structure, and the blocks that are filled in the testing structure but are not filled in the created structure are analysed. More specifically, the number of these blocks that cannot be filled in the created structure due to the *Gripper Restriction* or blocks above prohibiting placement of other blocks are counted. This result is then compared to how many blocks are filled in the testing structure and not in the created structure as a whole. These results give an indication of how well the agent learns the complex orderings in which blocks should be placed, and also give indication of how much of a model's lack of performance is due to the model's inability to learn these complex orderings. The results are presented in the Table 4.4 below as percentages over the total number of blocks in the testing structures. These results are averaged over the testing structures of size 25, 50 and 75 and all the models trained on these respective sizes independently.

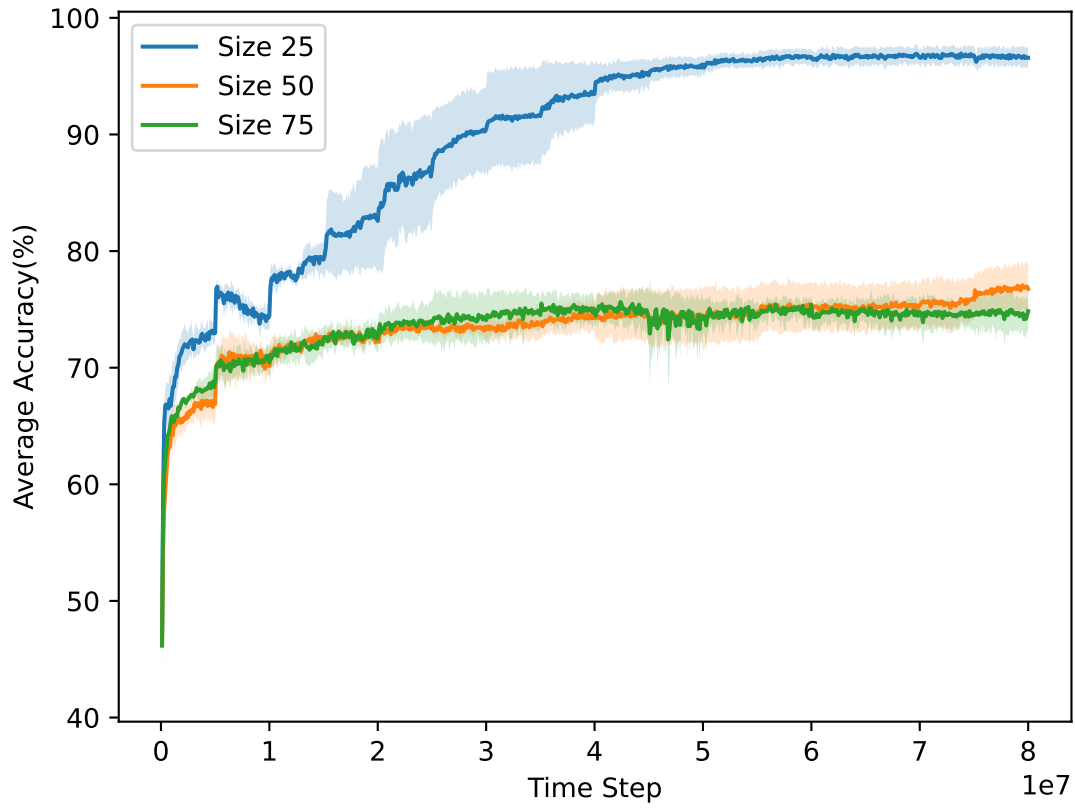


Figure 4.5: Figure showing average accuracy as a percentage versus time step during training for the models trained and tested on structures of size 25, 50, and 75. The shading in this figure represents the average error over the three training runs of a model which is calculated as described in Figure 4.2.

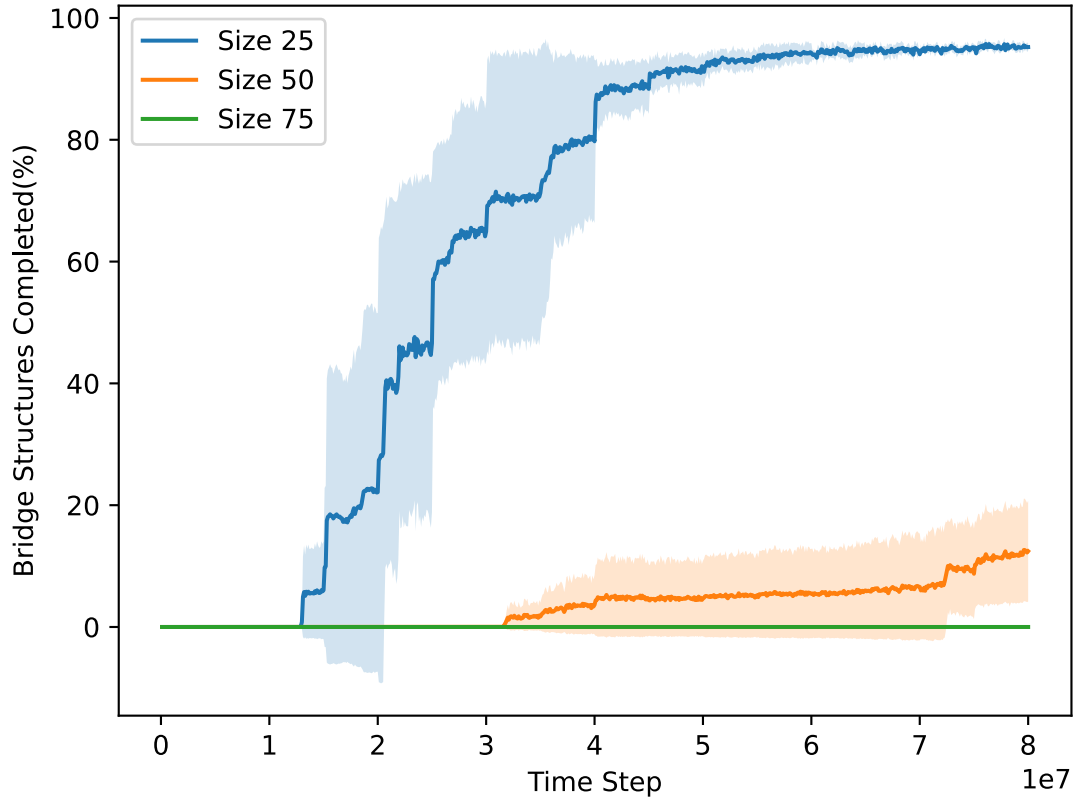


Figure 4.6: Figure showing percentage of total *Bridge Structures* completed successfully versus time step during training for the models trained and tested on structures of size 25, 50, and 75. The shading in this figure represents the average error over the three training runs of a model which is calculated as described in Figure 4.3.

RL Agent	Unfilled Blocks Counted (%)	All Unfilled Blocks (%)
Size 25	1.24	1.82
Size 50	11.20	16.45
Size 75	13.48	17.63

Table 4.4: Table showing the average percentage of unfilled blocks counted and the average percentage of all unfilled blocks, over the number of blocks within the testing structures for the models trained on structures of size 25, 50 and 75. Blocks are considered unfilled if they are filled within a testing structure but not filled when recreating the structure. Unfilled blocks are counted if they cannot be filled due to the *Gripper Restriction* or blocks above prohibiting placement of other blocks.

4.4.3 Discussion of Results

Below, we refer to results for a model based on the size of the structures it is trained and tested on.

In Figure 4.5 the performance is excellent when using size 25, but then dramatically drops off for size 50 and 75. This could be due to the model’s inability to learn the complex orderings in which actions must be executed as the size increases. Although the reason complex orderings are necessary, is only partly due to the presence *Bridge Structures* within structures, there is evidence for the previous statement in Figure 4.6, which shows that *Bridge Structures* are completed much more reliably when using size 25 compared to using size 50 or 75.

There is a significant increase in average error between time step 1×10^7 and time step 4×10^7 when using size 25. This is likely to be the period when the agent is learning about complex orderings of actions to perform. This is because to learn about the complex orderings, the agent must first execute them by chance a few times. Thus, the stage at which it has executed enough of these complex orderings will also be random. Based on the results in Figure 4.6, we can confirm this is a period where complex orderings of actions are learnt because this is when the model starts to recreate *Bridge Structures* more reliably. This also gives indication to the fact that these complex orderings are not learnt when using size 50 or 75, since we do not see these increases in average error when using either of these sizes.

In Table 4.4, it can be seen that a significant percentage of blocks filled in the testing structures of size 50 and 75 are not filled when attempting to recreate the structures due to the *Gripper Restriction* or blocks above prohibiting placement of other blocks. The reason blocks cannot be placed due to the *Gripper Restriction* is because blocks within the layer have been placed in the incorrect order, as discussed in Subsection 3.3.1. In order for the blocks placed above to not prohibit placement of other required blocks, the required blocks are to be placed first. Therefore, we provide further evidence that the models using size 50 and 75 have not learnt about the complex orderings in which blocks must be placed to create a structure. It can also be seen that a significant percentage of all unfilled blocks are not filled due to the models inability to learn about these complex orderings.

We conclude that performance decreases as the size increases, which we show is largely due to the model being unable to learn about the complex orderings in which actions must be executed as the size increases.

4.5 Experiment 3

As seen in Experiment 2 (Section 4.4), the agents performance deteriorates as the size of the goal structures increase. To achieve our future goal of constructing shelters or furniture, it is important that the agent is able to rebuild structures of a large size. In this experiment we attempt to remedy this issue. We test if the use of instruction manuals abstracted from the image to voxel model is beneficial to the performance of the agent when the goal structures are large, based on the accuracy Equation 4.1. The use of instruction manuals is abstracted from the image to voxel model by using voxelised instruction manuals.

In general, a voxelised instruction manual is generated from an instruction manual, as shown in Section 3.7. In this experiment, voxelised instruction manuals are generated in a manner abstracted from actual instruction manuals. The voxelised instruction manuals are generated from lists of actions taken to build goal structures from the bottom-up, and an interval number. Given a list of actions taken to build a goal structure from the bottom-up, and some interval number, a voxelised instruction manual is generated in the following way. We start off with an empty structure and take actions from the start of the given list of actions in order, until there are as many filled indices in this structure as the given interval number. Next, this structure is inserted into our voxelised instruction manual. From here actions are taken from the last action executed until another n indices have been filled within this structure where n is equal to the interval number. Again, this structure is inserted into our voxelised instruction manual. We follow these steps until all of the actions in the list of actions have been executed. As a result, the difference between the number of indices filled in any two subsequent structures within the voxelised instruction manual will be at least equal to the given interval number.

One model is trained on a set of random goal structures and tested on a set of random goal structures as well as three different sets of random goal structures in voxelised instruction manual form. The three different sets of random goal structures in voxelised instruction manual form are generated using interval number 1, 10 and 30 respectively.

Three models are trained on sets of random goal structures in voxelised instruction manual form. These models are also tested on sets of random goal structures in voxelised instruction manual form. One model is trained and tested on structures in voxelised instruction manual form using interval number 1, another model using interval number 10, and one last model using interval number 30.

We compare the results of a model trained and tested without the use of voxelised instruction manuals, a model trained without but tested with the use of voxelised instruction manuals and models both trained and tested with the use of voxelised instruction manuals.

4.5.1 Training using Voxelised Instruction Manual

When training a model using goal structures provided in the form of voxelised instruction manuals, we follow the same steps described in Section 3.5, except the goal structures are generated in the form of a voxelised instruction manuals, and during an episode we attempt to recreate the goal structure using the steps described in *Method 2* from Subsection 3.9.2.

4.5.2 Generation of Structures

To generate the training and testing structures not in voxelised instruction manual form we do the following. 5000 random goal structures are generated for the training structures, and 2500 random goal structures for the testing structures. These goal structures are generated in the same way the goal structures are generated in Subsection 4.3.2,

except actions are taken until 75 indices have been filled within the 3D space of the simulation.

To generate training and testing structures that are in voxelised instruction manual form using interval number n , we do the following. First, these training and testing structures are set equal to copies of the training and testing structures generated above, respectively. Next, each of these copied structures are converted into voxelised instruction manual form, using an interval number of n .

Training and testing structures that are in voxelised instruction manual form are generated using interval numbers 1, 10 and 30, in the manner described above.

4.5.3 Results

We average results over three different runs for all the models. For each run, an agent is trained in the simulation environment for 80000000 time steps on its respective training structures. Every 100000 time steps the model we are training is used to rebuild every structure from its respective testing structures and record the following result. The average accuracy over its respective testing structures, using the Equation 4.1 to calculate the accuracy for single testing structure. These results are presented in the Figures 4.7 and 4.8 below.

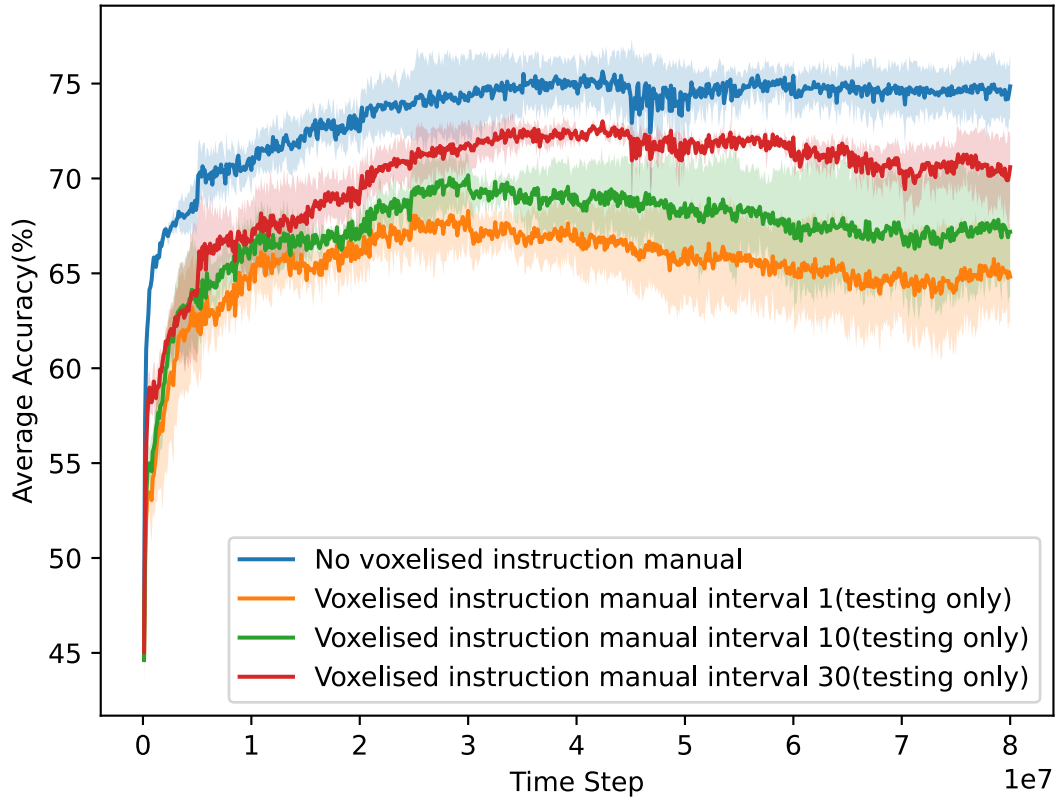


Figure 4.7: Figure showing average accuracy as a percentage versus time step during training for the model trained on structures not in voxelised instruction manual form. The results shown are for this model tested on structures not in voxelised instruction manual form, as well as structures in voxelised instruction manual form using interval number 1, 10 and 30. The shading in this figure represents the average error over the three training runs of a model which is calculated as described in Figure 4.2.

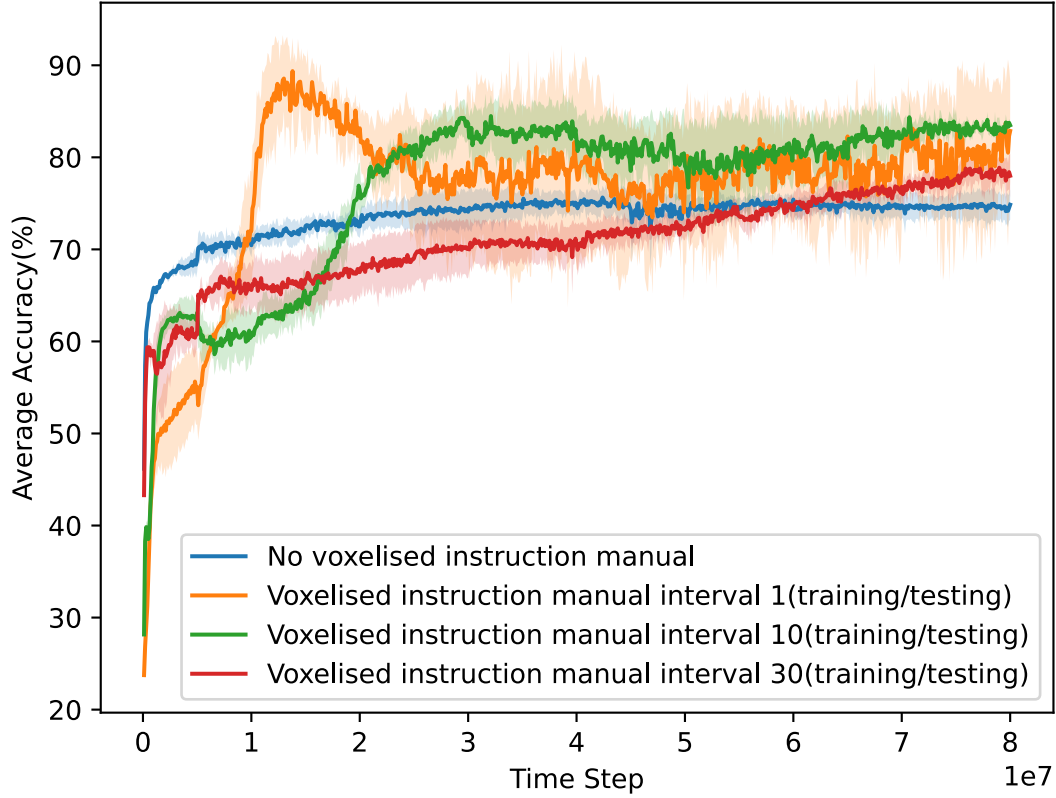


Figure 4.8: Figure showing average accuracy as a percentage versus time step during training for the one model trained and tested on structures not in voxelised instruction manual form, and the three models trained and tested on structures in voxelised instruction manual form, using interval number 1, 10 and 30. The shading in this figure represents the average error over the three training runs of a model which is calculated as described in Figure 4.2.

4.5.4 Discussion of Results

When analysing the results in Figure 4.7, we refer to the results produced by the model trained and tested without the use of voxelised instruction manuals by saying, when no interval number is used. We refer to the results of this model when tested using voxelised instruction manuals based on the interval number used to create these voxelised instruction manuals.

In Figure 4.7, it can be seen that it is not beneficial to use any interval number compared to using no interval. This is likely due to the fact that the model is only exposed to goal structures of size 75 during training, and never exposed to goal structures of a smaller size. Although smaller sized structures are easier to recreate, the model has only learnt to take actions in states where the goal structures are of size 75 and not any other size. This is also the model performs worse as the interval number used decreases. As the

interval number decreases, the size of goal structures initially provided to the model also decrease. As the size of the goal structures decrease, the less the states resemble states in which the model has learnt to act in.

When analysing the results in Figure 4.8, we refer to the results produced by the model trained and tested without the use of voxelised instruction manuals by saying, when no interval number is used. We refer to the results of a model trained and tested on voxelised instruction manuals based on the interval number used to create these voxelised instruction manuals.

In Figure 4.8, when using interval 1, the model performs better than all the other models ever do at around 1×10^7 time steps. From here onwards the model starts to perform worse. This is likely due to a lack of robustness of the model which leads to instability during training. This means that although the model may perform well when errors have not been made during the reconstruction process of a structure, the model struggles as soon as errors have been made. As a result, the accuracy to which the model recreates a structure can drop drastically as soon as errors are made, therefore inducing fluctuating performances during training. Evidence for this lack of robustness is shown in Table 4.5 below.

RL Agent	Accuracy when $\epsilon = 0.0$ (%)	Accuracy when $\epsilon = 0.2$ (%)
No Interval	74.85	69.28
Interval 30	78.00	72.57
Interval 10	83.48	72.18
Interval 1	82.86	53.04

Table 4.5: Table showing the average accuracy to which the models from Figure 4.8 recreate their respective testing structures when the exploration rate (ϵ) is set to 0.0, versus 0.2.

In Table 4.5, we see that performance drops off drastically when $\epsilon = 0.2$ compared to when $\epsilon = 0.0$, for the model using interval 1. This shows us that the model lacks robustness, because as soon as slight errors are introduced, due to random actions in this case, the performance of the model drops significantly. This issue can potentially be overcome by increasing the replay buffer size, thus allowing the model to continually train on a wide variety of experiences, or by dedicating a section of the replay buffer to negative experiences.

We see that using interval number 10 results in the best end performance, using interval number 1 is second best, using interval number 30 is third best and using no interval number is the worst. That being said, it is likely that using interval 1 would result in the best end performance, if the model did not suffer from instability during training. These results are expected because the smaller the difference in size between a goal structure and a structure currently built, the easier it becomes to recreate the goal structure, given that the structure currently built is correct. This is because the smaller this difference in size, the less actions it will take to recreate this goal structure.

It can also be seen that the lower the interval number used, the slower training is initially. Although the task becomes easier as the interval number decreases, the model is also required to learn about more states. This is because for some interval number, the model is required to learn about goal structures of sizes between this interval number and 75.

We conclude that it is not beneficial to provide goal structures in voxelised instruction manual form to a model trained on full goal structures, compared to providing full goal structures to this model. That being said, we conclude that it can be beneficial to provide goal structures in voxelised instruction manual form to a model trained using voxelised instruction manuals, compared to providing full goal structures to a model trained on full goal structures. Although these benefits exist, it is shown that there may be some side effects which include training becoming more unstable or slower initially, as the interval number used decreases.

4.6 Experiment 4

In Experiment 3 (Section 4.5), we see that the use of instruction manuals, abstracted from the image to voxel model can improve performance of our RL agent in the simulation environment. More specifically, we see that it is best to provide goal structures in the form of voxelised instruction manuals to an RL agent trained on voxelised instruction manuals. This is useful knowledge, but in reality, it would be more beneficial to provide images of the goal structure. Thus, in this experiment we test if providing goal structures in the form of instruction manuals when using the image to voxel model and our RL agent in conjunction, can be beneficial compared to providing images of full goal structures, while leveraging knowledge of the results in Experiment 3.

To use the image to voxel model and our RL agent in conjunction, either images of a final goal structure, or images in the form of an instruction manual are read in, and then these images are converted into a goal structure either in voxel representation or voxelised instruction manual form, using the image to voxel model. Finally, the RL agent is used to recreate this goal structure. We test how these models perform in conjunction in the following scenarios.

- When images of a final goal structure are provided and converted into a voxel representation before reconstruction.
- When images of a goal structure are provided in the form of an instruction manual and converted into a voxel representation before reconstruction.
- When images of a goal structure are provided in the form of an instruction manual and converted into voxelised instruction manual form before reconstruction.

We test converting the instruction manual into voxelised instruction manual form as it is known to be beneficial to provide goal structures in this form to an RL agent based on the results in Experiment 3. An RL agent trained on voxelised instruction manuals is used to recreate goal structures provided in the form of a voxelised instruction manual, as we know it is better to do this rather than use an RL agent not trained using voxelised

instruction manuals, based on the results in Experiment 3. We also test providing the above-mentioned instruction manuals with different intervals.

Instruction manuals are converted into voxel representation of the final goal structures by first converting the instruction manuals into voxelised instruction manuals, using a trained image to voxel model, and then extracting the last voxel representation within each of these.

In this experiment the RL agent trained on full goal structures from Experiment 3 is used, as well as the three RL agents trained with the use of voxelised instruction manuals. One image to voxel model is trained on full structures. Three image to voxel models are trained on intermediate structures within instruction manuals that are generated using an interval of 1, 10 and 30, respectively.

We test the image to voxel model trained on full structures in conjunction with the RL agent trained on full goal structures, on a set of random goal structures. We test the three image to voxel models trained on instruction manuals in conjunction with the RL agent trained on full goal structures, on a set of random instruction manuals. Lastly, we test the three image to voxel models trained on instruction manuals in conjunction with the RL agents trained on goal structures in voxelised instruction manual form, on a set of random instruction manuals.

4.6.1 Generation of Training Data

We generate the full structures used for training and testing the one image to voxel model in the following way. First, 5000 random structures are generated for the training structures, and 2500 random structures for the testing structures. These goal structures are generated in the same way the goal structures are generated in Subsection 4.3.2, except actions are taken until 75 indices have been filled within the 3D space of the simulation. Next, each of these structures are converted into a data-point where the label is the structure, and the features are the front-view, left-view and top-view images of the structures. The dimensions of the structure voxel representations are $5 \times 5 \times 5$ and the dimensions of the images are 5×5 . These structures are converted into data-points so that they can be used for training our image to voxel model as well as inference.

Instruction manuals are generated for training and testing the other three image to voxel models. More specifically, three sets of training and testing instruction manuals are generated using an interval of 1, 10 and 30 respectively, in the following way. To generate training and testing instruction manuals with interval n , we convert each of the structures from the full training and testing structures above into instruction manual form using interval n . The steps followed to convert some structure into instruction manual form using some interval number, are shown below.

Given a structure, the actions taken to build this structure from the bottom-up, and an interval number, this structure in instruction manual form can be generated in the following way.

We start off with an empty structure and take actions from the start of the given list of actions in order, until there are as many filled indices in this structure as the given

interval number. Next, this structure is inserted into the instruction manual. Again, starting from an empty structure, actions are taken from the last action we executed until there are as many filled indices in this structure as the given interval number. Again, this structure is inserted into the instruction manual. The last two steps are repeated until we have inserted the structure that is produced after taking the last actions in the list of given actions. In the end, each structure in the instruction manual will contain at least as many filled indices as the given interval number. Finally, each of these structures are converted into a data-point where the label is the structure, and the features are the front-view, left-view and top-view images of the structures. These structures are converted into data-points so that they can be used for training our image to voxel models as well as inference.

4.6.2 Results

We generate results for using an image to voxel model in conjunction with an RL agent in the following way. First, three separate image to voxel models are trained on their respective training data to test with the three versions of this RL agent we trained in Experiment 3. For each of these image to voxel models, each of these RL agents, and each structure in the respective testing data, we pass the images of this structure through the image to voxel model and recreate the produced structure using the RL agent. Depending on the image to voxel model and RL agent being tested, we either pass through images of the full structure or images in the instruction manual describing this structure. If the images are from an instruction manual, the produced structure is either a voxel representation or in voxelised instruction manual form. Next, the following four accuracy values are recorded using Equation 4.1. The accuracy to which the image to voxel model predicts the structure that we generated the images from, and refer to this as the IV (which stands for image to voxel) accuracy. The accuracy to which the RL agent recreates the structure that we generated the images from and refer to this as the RL (which stands for reinforcement learning) accuracy. The accuracy to which the RL agent recreates the structure produced from the image to voxel model and refer to this as the PS (which stands for produced structure) accuracy. The accuracy of the structure created by the RL agent given the structure produced from the image to voxel model compared to the initial structure that we generated the images from and refer to this as the IS (which stands for initial structure) accuracy. All of these accuracy values are averaged and the results are presented in Table 4.6 below.

The IV accuracy gives us an indication of well the image to voxel model is performing. The RL accuracy gives us an indication of how well the RL agent is performing. The PS accuracy gives us an indication of how well the RL agent can recreate structures produced by the image to voxel model. The IS accuracy gives us an indication of how well the image to voxel model and the RL agent perform in conjunction.

Image to Voxel Model	RL Agent	IV(%)	RL(%)	PS(%)	IS(%)
No Interval	No Interval	65.30	74.85	73.87	56.17
Interval 1	No Interval	90.91	74.85	67.65	67.56
Interval 10	No Interval	88.52	74.85	68.63	67.66
Interval 30	No Interval	76.68	74.85	68.06	63.19
Interval 1	Interval 1	90.91	82.86	43.46	43.74
Interval 10	Interval 10	88.52	83.48	71.26	67.30
Interval 30	Interval 30	76.68	78.00	63.29	55.88

Table 4.6: Table showing average IV, RL, PS and IS accuracies achieved as a percentage when using specific image to voxel models in conjunction with specific RL agents. In this table, the image to voxel models and RL agents trained on full structures are referred to by saying No Interval. The image to voxel models trained on instruction manuals of interval n are referred to by saying Interval n . The RL agents trained on voxelised instruction manuals of interval n are referred to by saying Interval n .

Image to Voxel Model	RL Agent	IV(%)	RL(%)	PS(%)	IS(%)
Interval 10	No Interval	88.52	74.85	68.63	67.66
Interval 10	Interval 10	88.52	83.48	71.26	67.30
Interval 30	No Interval	76.68	74.85	68.06	63.19
Interval 30	Interval 30	76.68	78.00	63.29	55.88

Table 4.7: Table highlighting results for four different image to voxel model and RL agent pairings used in conjunction from Table 4.6.

4.6.3 Discussion of Results

When analysing the results in Table 4.6, we refer to the image to voxel model trained on full structures as the image to voxel model trained with no interval. We refer to the RL agent trained on full structures as the RL agent trained with no interval. We refer to the image to voxel models trained on instruction manuals with interval n as the image to voxel models trained with interval n . We refer to the RL agents trained on voxelised instruction manuals with interval n as the RL agents trained with interval n . We refer to the usage of an image to voxel model in conjunction with an RL agent, by saying the (name of the image to voxel model in the table)-(name of the RL agent in the table) conjunction, for example, the Interval 1-No Interval conjunction.

In Table 4.6, the IV accuracy increases as the interval used when training our image to voxel model decreases, and is worst when no interval is used. This is expected because as the interval decreases, the more accurately we can describe the intermediate structures in the instruction manual. When no interval is used, only the overall shape of the full structure is described using the images, and we have no way of predicting how blocks are assembled within the structure itself.

The PS accuracy is highest with the No Interval-No Interval conjunction. This is likely to be because the produced structures from the image to voxel model do not capture the intricacies of the initial structure which require complex orderings of actions to

recreate. As a result, it is easier for the RL agent to recreate these produced structures to a higher degree of accuracy.

The IV accuracy as well as the RL accuracy is high when using an interval of 1 to train both our image to voxel model and our RL agent. That being said the PS and IS accuracies are drastically low with the Interval 1-Interval 1 conjunction. This is likely due to that fact that when a produced intermediate structure cannot be recreated with a valid action, the RL agent starts to struggle. This is due to the models lack of the robustness discussed in Subsection 4.5.4.

By referring to Table 4.7, it can be seen that the PS accuracy from the Interval 30-Interval 30 conjunction is worse than the PS accuracy from the Interval 30-No Interval conjunction. Since both RL agents are required to recreate the same structures produced by the image to voxel model, this shows that the RL agent performs worse when recreating intermediate structures of size 30 in order to recreate the full structures, compared to recreating the full structures in one go, even though the RL accuracy of the RL agent trained with interval 30 is higher than the RL accuracy of the RL agent trained with no interval. This likely stems from the fact that the image to voxel model trained with interval 30 produces structures to a low degree of accuracy. This potentially leads to more cases where it is impossible to place certain blocks in new intermediate structures, as these new intermediate structures are introduced during the rebuilding process. Given the full structure in one go, the RL agent can plan ahead to prevent finding itself in situations where it is impossible to place many of the blocks it needs to place. The opposite is true when comparing the PS accuracy from the Interval 10-Interval 10 conjunction to the PS accuracy from the Interval 10-No Interval conjunction. This issue presented above is circumvented in this case because the image to voxel model trained with interval 10 produces structures to a high degree of accuracy.

The image to voxel models trained with interval 30 or no interval used in conjunction with RL agents do not produce the best IS accuracies because the performance of these image to voxel models is relatively poor.

We note that the highest IS accuracies are those from the Interval 1-No Interval conjunction, the Interval 10-No Interval conjunction and the Interval 10-Interval 10 conjunction.

Overall, it is likely beneficial to incorporate the image to voxel models into the RL agent training process. This is based on the fact that the PS accuracy is lower than the RL accuracy for all the image to voxel model and RL agent pairings we test in conjunction. This shows that the RL agents recreate structures to a higher degree of accuracy, if these structures are generated in the same manner as the structures they trained on, compared to if these structures are produced by the image to voxel models. This issue could potentially be overcome if the RL agent is trained on structures that are produced by the image to voxel models. By achieving a better PS accuracy, we are likely to see a better IS accuracy as well.

We conclude that providing goal structures in the form of instruction manuals can be beneficial compared to providing images of full goal structures based on accuracy on average. More specifically, we conclude that it can be beneficial to either convert these

instruction manuals to voxel representations of the structures or convert them into voxelised instruction manuals before recreating the structures.

4.7 Experiment 5

Based on the previous experiments, we see that the RL agent and image to voxel model can be used in conjunction to recreate structures given images of these structures. This is useful, but it is important that these structures can be recreated in real-life too. In this experiment, we test how well the YuMi robot can execute lists of actions produced from an RL agent acting in the simulation environment. Ten random structures of size 10 are created where the size of the voxel representations is $3 \times 3 \times 3$. The images of these structures are fed through a trained image to voxel model, and the produced structures are recreated using a trained RL agent to generate the lists of actions. Finally, these lists of actions are executed using the YuMi robot and Duplo blocks. We record how many of these lists of actions the robot manages to execute successfully. We do not apply the use of instruction manuals in this experiment as the structures are of a small size.

4.7.1 Results

The robot manages to execute 6 out of the 10 lists of actions successfully. In two of the failed cases, the robot fails to place the last 3×1 block in the third layer. In one of the failed cases, the robot fails to place a 1×1 block in the third layer. In the last failed case, the robot fails to pick up a 3×1 block it is about to place.

In Table 4.9 below, the failed cases mentioned above are shown.

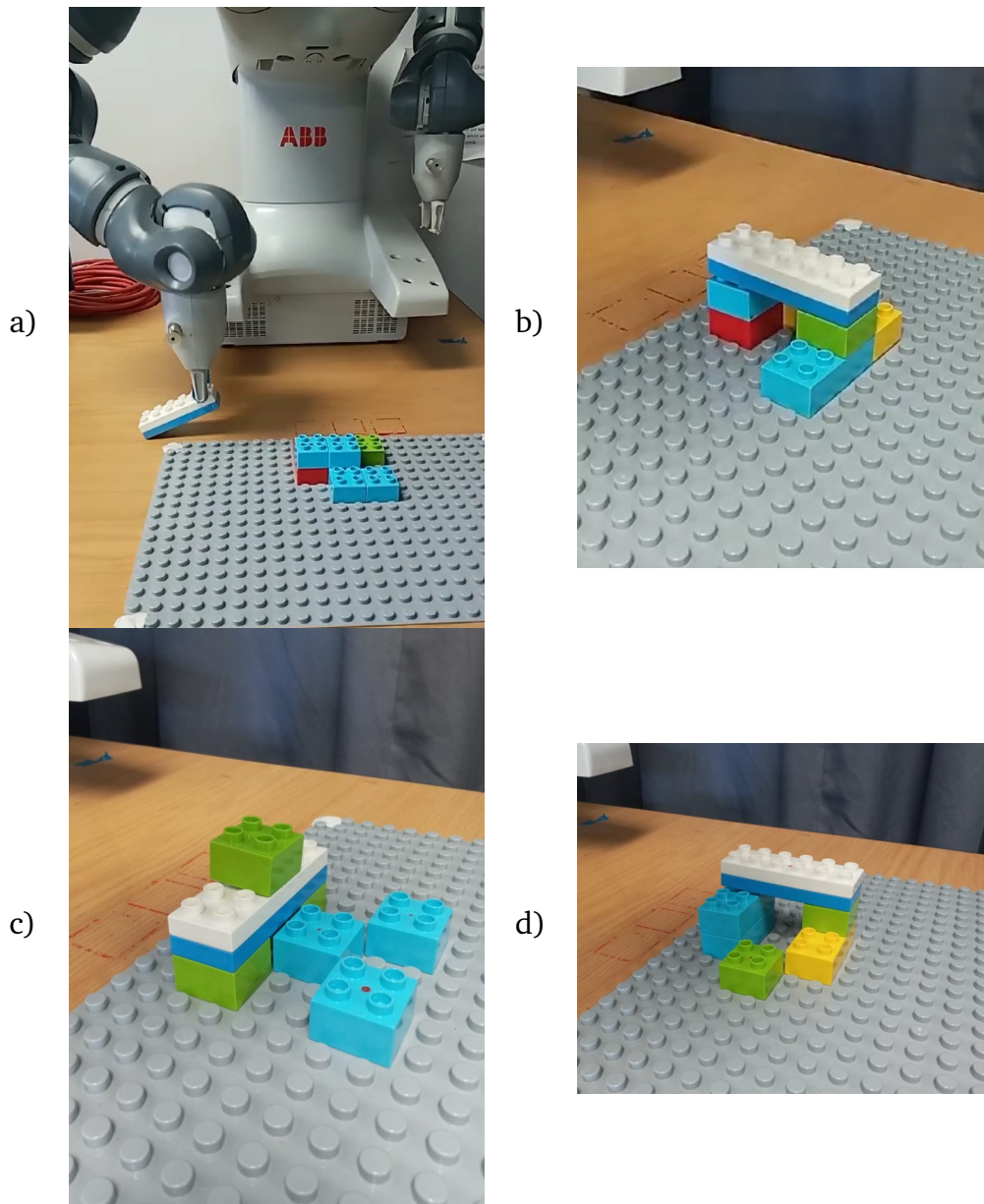


Figure 4.9: An illustration of the failed cases when rebuilding structures using the YuMi robot. In image a) the robot fails to pick up the 3×1 block. In images b), c) and d), the robot fails to place the block in the third layer successfully, as the blocks are not fully connected to the blocks below them.

Below an example of the robot successfully rebuilding a structure is shown in Figure [4.10](#).

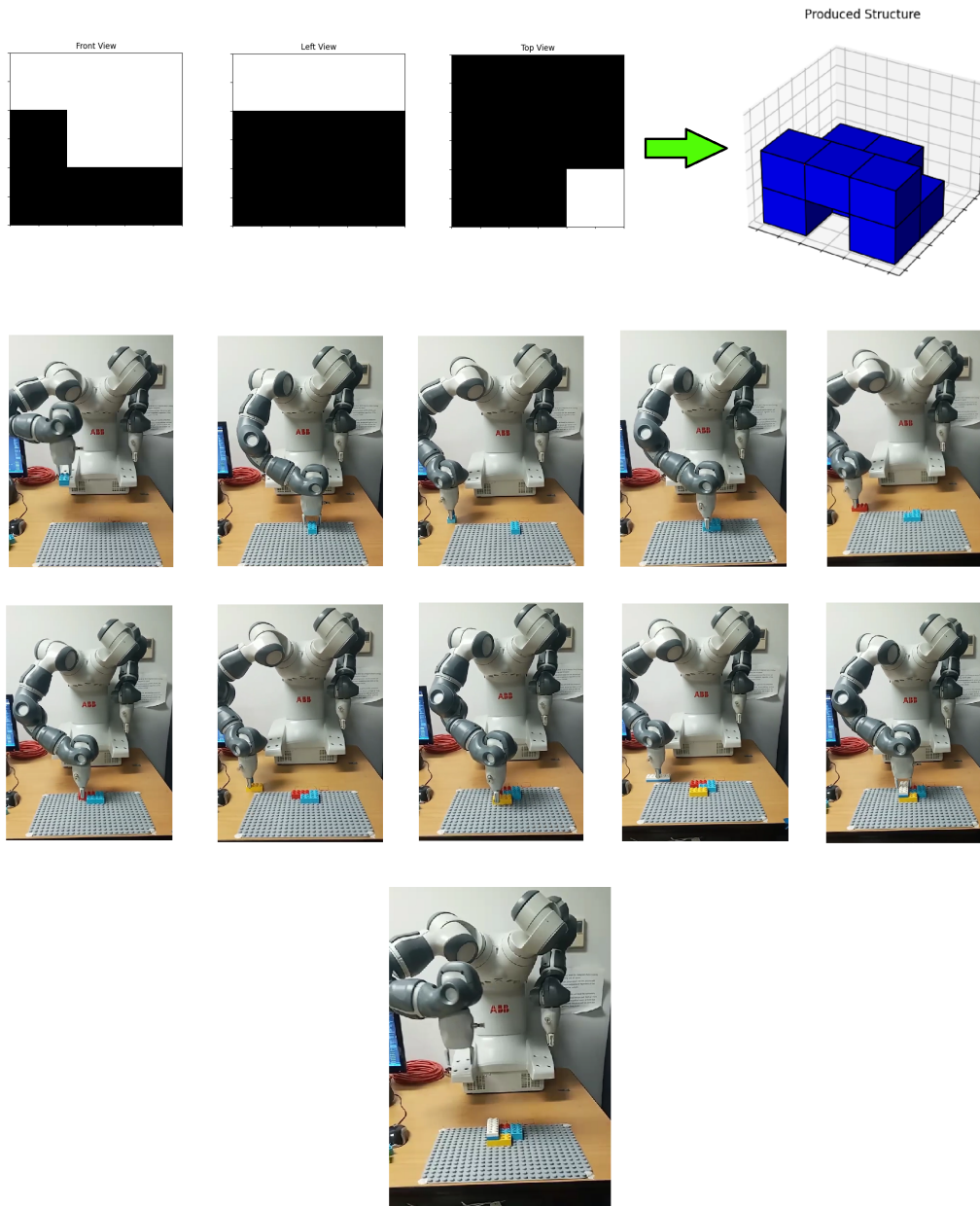


Figure 4.10: An illustration showing an example of the robot successfully rebuilding a structure. We input the front, left, and top-view images of the structure. Next, we get the structure in voxel representation by feeding the images of the structure through a trained image to voxel model. From here, the structure is rebuilt in the simulation environment using a trained RL agent to generate a list of actions that can be taken to rebuild this structure. Finally, the structure is rebuilt using the robot and Duplo blocks by executing this list of actions.

4.8 Conclusion

In conclusion we show that it can be beneficial to bias the data we train our RL agent on to include more *Bridge Structures*, the performance of our RL agents decrease as the size of structures increase, the use of voxelised instruction manuals abstracted from the image to voxel model can be beneficial to the performance of our RL agent, the use of instructions manuals can be beneficial when using an image to voxel model in conjunction with our RL agent and finally we show that the YuMi robot can successfully rebuild structures using Duplo blocks.

Chapter 5

Related Work

In this chapter we discuss related problems and approaches to these problems in comparison to our approach.

As stated before, there have been many approaches that address the task of construction via sequential actions. One method has been to learn Generative models [23] on LEGO structures. These Generative models are used to sequentially assemble new, previously unseen LEGO structures [1]. Data on human-built structures is used to learn the Generative models. The Generative models are based on graph-structured neural networks [24] and all the LEGO structures are represented as graphs. More specifically, a LEGO structure is represented as a directed graph, whereby the nodes represent LEGO bricks and the edges represent connections between two bricks. A new LEGO structure graph is generated by sequentially adding nodes and edges between these nodes until a termination criterion is met. To add a node to the graph, first the probability of adding a node of each type is calculated by feeding an embedded vector of the graph through a standard multi-layered perceptron (MLP) with a softmax activation. Then the node with the highest probability is chosen to be added to the graph. To add an edge to a newly added node, first the probabilities of either not adding an edge, adding an incoming edge or adding an outgoing edge to the node are calculated. If the option of adding an edge has the highest probability, then a destination node to connect the edge to is chosen by calculating a score for each potential destination node, and then choosing the node with the highest score. This score is calculated by feeding an embedded vector of this node and the newly added node through an MLP. Finally, once a destination node is chosen, then two separate MLPs are used to determine the x and y coordinate offsets respectively (the x and y coordinate offsets describe how much the two bricks that are to be connected will overlap in the x and y direction respectively). These two separate MLPs both take in an embedded vector of the destination node and the newly added node as input. In contrast to this method, we use Reinforcement Learning in order to recreate a given goal structure, instead of using Generative models to create new unseen structures.

In another approach an attempt is made to learn an agent to recreate a given goal structure in a 2D simulation environment using sequential block placing actions [2]. The

learning is done using Reinforcement Learning. The goal structure is made of stacked 2D blocks and is provided to the agent as an image. At the start of an episode a block is spawned at some random position along the top boundary of the environment. The agent has three actions which are to move the spawned block down, left, or right. A spawned block is considered placed once it has collided with another placed block or with the bottom boundary of the environment. Once a block has been placed, the next block to be placed is spawned. The reward feedback given to the agent is based on how similar the structure the agents creates is to the goal structure. The model that is learnt in this approach is a DQN. In contrast to this approach we use a 3D simulation environment, thus allowing us to represent a space in which the YuMi robot can place Duplo blocks.

Another method has been to create an agent that learns to recreate a given goal structure in a 3D simulation environment using sequential blocks placing actions [3]. The learning is done using Reinforcement Learning. Binary images of the given goal structure are provided to the agent. The structure the agent builds is stored as a directed graph whereby the nodes represent blocks, and the edges represent connections between two blocks. During construction, blocks are placed one by one by the agent. When placing a new block, the agent first chooses another block to connect it with, and then the agent chooses the offset and overlap of this connection. The binary images of the given goal structure are fed through a convolutional neural network (CNN) [25] and then the output is combined with the graph of the structure built before being fed through a series of MLPs and graph neural networks which finally produce probabilities for which block placement action should be executed next. They also learn a separate graph neural network in order to predict which block placement actions would be valid given the structure built to facilitate the network that predicts blocks placement actions. The agent is rewarded based on how similar a voxel representation of the structure built is to a voxel representation of the goal structure. In our approach we analyse the effect of inputting the given goal structure as a set of images in the form of an instruction manual. The instruction manual is used to provide guidance and assist construction because it provides the agent with intermediate goal structures that can be constructed to recreate a final goal structure. We transform the images of the goal structures into voxel representation. In contrast to their approach, these voxel representations are fed through our action prediction network instead of feeding the images themselves through the network. Instead of using a separate network to predict valid block placement actions, we give the agent a negative reward for taking an invalid action. We extend on their approach by creating an agent that can place blocks of different sizes and shapes, build structures that are physically stable and produce lists of sequential block placing actions that the YuMi robot can physically execute given its movement constraints.

Another method has been to generate LEGO building instructions from multiple photographic images of real objects [9]. They do this by first taking multiple images of some object from different angles. Using these images and the shape from silhouette method, they construct an octree voxel-based 3D model [26; 27] with colour information. Using this voxel model, they then construct a hollowed-out LEGO model using

simulated annealing, which is a stochastic global optimization method [28]. Finally, using this LEGO model, they produce building instructions to rebuild it in real-life. The building instructions they produce illustrate the blocks that need to be placed for each layer in the LEGO model. In contrast to their approach, the building instructions we produce are in the form of a sequential list of block placing actions. We produce a list of actions because the order in which the blocks are placed is important. If the order of block placements were changed, then it is not guaranteed that the YuMi robot will be able to place all of the blocks successfully, due to the physical constraints of the robot. In their approach, the stability of the LEGO structure produced is not guaranteed. In this aspect we also extend on their approach by ensuring the structures built are structurally stable.

Similar to the previous method, another method has been to generate 3D LEGO models and their corresponding building instructions using an image of the desired structure [10]. In this method they start off by converting the image of the desired structure into a latent representation. Using this latent representation, they predict a 3D voxel model of the structure. They apply an algorithm to produce the LEGO model given this 3D voxel model of the structure. Briefly, this algorithm works by converting each voxel in the voxel model to a 1×1 LEGO brick before finding optimal groups of bricks that can be combined into larger cuboid bricks. The building instructions produced illustrate the blocks that need to be placed for each layer in the LEGO model. We extend on this approach in the same way we extend on the previously described approaches, whereby we produce our building instructions in the form of a sequential list of actions, such that this list of actions can be physically executed by a YuMi robot given its movement constraints.

Another method has been to generate a LEGO brick layout given a 3D object [8]. In this approach they focus on initially creating a layout where all the bricks are connected such that they form a single connected component. From here, they iteratively perform stability analysis on the structure, and locally reconfigure weak sections in the structure to gradually improve the stability. In contrast to this approach, we ensure stability of the structures built by ensuring that no blocks can be placed in an over-hanging manner. In other words, the ends of each block placed must either be on the ground level or placed on top of previously placed blocks.

Lastly, one method has been to generate building instructions for fragile block models, by assembling the order in which the blocks should be placed [7]. They do this by segmenting the structure into components based on weak areas in the structure, and then determining the order in which these components should be assembled. They also try to segment the structure into as few components as possible, while still allowing for an effective component assembly order to be produced. In our approach, we also determine the order in which blocks should be placed, but in extension to their approach, we consider the physical constraints of the YuMi robot. This is done so that the list of block placements can be executed by the robot successfully.

Chapter 6

Conclusion and Future Work

In conclusion we produce a working pipeline to recreate structures using a YuMi robot and Duplo blocks, given images of what the structures should look like, as well as answer all of the research questions posed. We show that an agent can learn to recreate a list of unseen goal structures of stacked blocks using sequential block placing actions to a high degree of accuracy, in the simulation environment using Reinforcement Learning. We show that the use of the instruction manual can improve performance compared to only providing the agent with images of the final goal structure. We show that it can be beneficial to bias the data we train our RL agent on to include more *Bridge Structures*. Finally, we show that a YuMi robot can be configured to execute a list of sequential block placing actions produced from the simulation environment in order to create a structure of Duplo blocks.

There is a lot of future work that can be done to extend our current approach. We can introduce colours such that the RL agent is required to recreate the structures while also matching the colours of the blocks in the structure. We can also introduce different types of blocks of differing sizes and shapes. Instead of providing binary images of structures to recreate, we can provide natural images of structures we wish to recreate [33; 34]. Instead of hard-coding positions of blocks to be placed by the robot, as well as hard-coding positions in which blocks can be placed, we can implement a framework that allows for blocks to be picked up from anywhere and placed anywhere. We can explore the use of different types of robots or different types of grippers to rebuild the structures in real-life. We can explore the use of more complex simulation environments to more accurately approximate the physical restrictions of the robot, or the stability of structures [35]. We can also explore different types of RL algorithms for our RL agent, or different types of models for our image to voxel representation part of the pipeline. Given more time we could also train the models for longer, and average results over more training runs.

References

- [1] R. Thompson, E. Ghalebi, T. DeVries and GW.Taylor. Building LEGO Using Deep Generative Models of Graphs. NeurIPS, 2020.
- [2] W. Li, J. Bohg and M. Fritz. Acquiring Target Stacking Skills by Goal-Parameterized Deep Reinforcement Learning. CoRR, 2017.
- [3] H. Chung, J. Kim, B. Knyazev, J. Lee, GW. Taylor, J. Park and M. Cho. Brick-by-Brick: Combinatorial Construction with Deep Reinforcement Learning. NeurIPS, 2021.
- [4] RS. Sutton and AG. Barto. Reinforcement Learning: An Introduction. The MIT Press, second edition, 2018.
- [5] JA. Arjona-Medina, M. Gillhofer, M. Widrich, T. Unterthiner, J. Brandstetter and S. Hochreiter. Rudder: Return decomposition for delayed rewards. NeurIPS, 2019.
- [6] V. Mnih, K. Kavukcuoglu, D. Silver, AA. Rusu, J. Veness, MG. Bellemare, A. Graves, M. Riedmiller, AK. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wiestra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–33, 2015.
- [7] M. Zhang, Y. Igarashi, Y. Kanamori, and J. Mitani. Component-based building instructions for block assembly. Computer-Aided Design and Applications, 14(3):293–300, 2017.
- [8] SJ. Luo, Y. Yue, CK. Huang, YH. Chung, S. Imai, T. Nishita. BY. Chen. Legolization: Optimizing LEGO Designs. ACM Transactions on Graphics, 2015.
- [9] T. Kozaki, H. Tedenuma, and T. Maekawa. Automatic generation of LEGO building instructions from multiple photographic images of real objects. Computer-Aided Design, 70:13–22, 2016.
- [10] K. Lennon, K. Fransen, A. O’Brien, Y. Cao, M. Beveridge, Y. Arefeen, and N. Singh. Image2Lego: Customized LEGO® Set Generation from Images. 2021.
- [11] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, AA. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hass-

- abis, C. Clopath, D. Kumaran, and R. Hadsell, Overcoming catastrophic forgetting in neural networks. *Proc Natl Acad Sci USA* 114, 3521–3526, 2017.
- [12] R. Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.
 - [13] ML. Puterman and EA Feinberg. Markov decision processes: discrete stochastic dynamic programming. *SIAM review*, 38(4):689, 1996.
 - [14] C. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
 - [15] RS. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *NeurIPS*, 2000.
 - [16] V. Konda and N. Tsitsiklis. Actor-critic algorithms. *NeurIPS*, The MIT Press, 1999.
 - [17] T. Degris, M. White, and RS. Sutton. Off-policy actor-critic. In *Proceedings of the 29th International Conference on Machine Learning*, 2012.
 - [18] V. Konda and N. Tsitsiklis. Actor-critic algorithms. *NeurIPS*, 2000.
 - [19] C. Dann, Y. Mansour, M. Mohri, A. Sekhari and K. Sridharan. Guarantees for epsilon-greedy reinforcement learning with function approximation. *Proceedings of the 39th International Conference on Machine Learning*, 162, 4666-4689, 2022.
 - [20] N. Hariharan and G. Anand. A Brief Study of Deep Reinforcement Learning with Epsilon-Greedy Exploration. *International Journal of Computing and Digital Systems*, 2022.
 - [21] LJ Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8 293–321, 1992.
 - [22] CM. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
 - [23] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Farley, S. Ozair, A. Courville and Y. Bengio. Generative Adversarial Nets. *NeurIPS*, 2014.
 - [24] TN. Kipf, and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. *International Conference on Learning Representations*, 2017.
 - [25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324, 1998.
 - [26] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing* 19(2):129–147, 1982.

- [27] J. Wilhelms, A. Van Gelder. Octrees for faster isosurface generation. *ACM Trans Graph* 11(3):201–227, 1992.
- [28] A. Zhigljavsky, A. Žilinskas. *Stochastic Global Optimization*. Springer, 2008.
- [29] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, ME. Taylor, P. Stone. Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. *Journal of Machine Learning Research* 21 (181), 1-50, 2020.
- [30] P. Soviany, RT. Ionescu, P. Rota, N. Sebe. Curriculum Learning: A Survey. *International Journal of Computer Vision*, 2021.
- [31] G. Hacohen, D. Weinshall. On the Power of Curriculum Learning in Training Deep Networks. *Proceedings of the 36th International Conference on Machine Learning*, 2019.
- [32] V. Nair and GE. Hinton. Rectified Linear Units improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, 2010.
- [33] R. Tahir, AB. Sargano, Z. Habib. Voxel-Based 3D Object Reconstruction from Single 2D Image Using Variational Autoencoders. *Mathematics*, 9, 2288, 2021.
- [34] W. Xiong, F. Huang, H. Zhang and M. Jiang. 3D voxel reconstruction from single-view image based on cross-domain feature fusion. *Expert Systems With Applications*, 256, 124957, 2024.
- [35] Z. Zhou, J. Song, X. Xie, Z. Shu, L. Ma, D. Liu, J. Yin, S. See. Towards Building AI-CPS with NVIDIA Isaac Sim: An Industrial Benchmark and Case Study for Robotics Manipulation. *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024.