ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Where to Click: Exploring Reinforcement Learning for 3D Interactive Object Segmentation

Bachelor Thesis

C. Tschechtelin

August, 2023

Advisors: Prof. Dr. K. Schindler, Dr. T. Kontogianni

Department of Computer Science, ETH Zürich

Abstract

In this paper, we propose a way to automatically generate seeds to solve the interactive segmentation problem of a 3D point cloud. The main issue of the interactive segmentation problem is the high human effort when extracting consistent objects in a 3D scene. The model presented in this paper tries to reduce the needed human effort and only relies on few user seed suggestions. After the first click is made by the user, the presented model predicts automatically a sequence of additional user clicks to optimally segment the desired object. Since we cannot define globally optimal user points, we have to define this task as a Markov Decision Process (MDP). By using Reinforcement Learning in a deep Q network we are trying to approximate the optimal solution.

Contents

Chapter 1

Introduction

Segmenting the object of interest in a 3D point cloud is an important problem in computer vision. The segmentation of 3D scenes is needed for creating labeled data sets used for training deep learning networks. Without user input, automatic object segmentation has limitations in segmenting objects, therefore it's important to know the user's intention to segment the correct object. In interactive object segmentation a user gives the information necessary in the form of some clicks or a bounding box to extract the desired object from a 3D scene.
For many objects in complex environments, the user has to spend much time to correct the segmentation algorithm such that the resulting segmentation is satisfying. Thus, it is of great importance to reduce the human effort while maintaining the performance. In this work, we propose a technique to simulate the user interactions with the segmentation system to obtain the desired object. It is important to decide which points require labeling. Currently heuristics are used in interactive object segmentation to simulate the user's intention.
In Interactive Object Segmentation in 3D Point Clouds[8] the user clicks are simulated by determining where the biggest error of the currently predicted mask is. Then the simulation chooses the point in the center of the error region. Although reaching 90% IoU (Intersection over Union) between the predicted and the ground-truth mask in about 20 clicks is quite powerful, we try to optimize this heuristics by reducing the number of clicks necessary.
In this work, the following alternative approach is explored. We use reinforcement learning to determine points which would benefit the most from user feedback. Reinforcement learning in combination with deep neural networks have received a lot of attention in the past few years from solving Atari games on human-level and for some games even above human-level. In reinforcement learning, an agent is trained to take actions on certain states. We are introducing a RL network for our optimized click simulation.

Chapter 2

Related Works

2.1 Minkowski Engine

3D videos in Robotics and VR/AR applications are often created by a sequence of Depth Images or LIDAR scans. The Minkowski Engine is a 4-dimensional convolutional neural network that can directly process such 3D videos using high-dimensional convolutions instead of using 2D convolutions for each frame. The Minkowski Engine hereby adopts sparse tensors and implements generalized sparse convolutions.

The focus of the Minkwoski Engine is the 3D-video perception. It solves the following technical challenges in using 3D-videos for high-level perception tasks:

- 3D data requires different representations and processing this data is confusing for the user and hard to implement for larger systems.
- 3D convolutional neural networks are less efficient in terms of performance compared to 2D convolutional networks.
- There are only a limited number of fast large-scale 3D data libraries.

Since this paper is mostly based on interactive 3D segmentation[8] and the Minkowski Engine is a memory efficient framework for 3D convolutions for large point clouds, this work is built on top of the Minkowski Engine.[2]

2.2 Interactive Segmentation in 3D Point Clouds

In interactive segmentation tasks, a user chooses an object which is to be segmented into foreground, while the other points of the scene are supposed to be in the background. The user selects certain seeds of the scene and labels them with either 1 or 0 (1 for foreground and 0 for background), such that

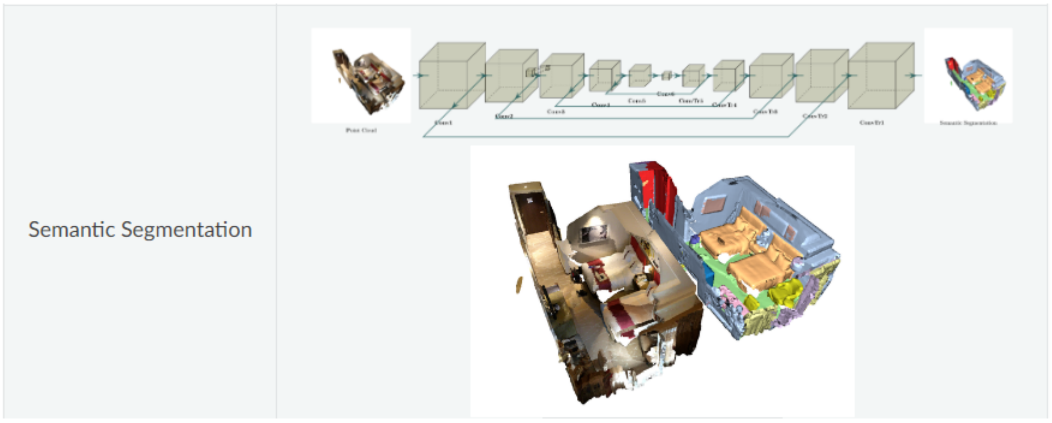


Figure 2.1: Example 3D segmentation network based on the Minkowski Engine. [2]

the underlying model can use this information from the user to predict the segmentation mask of the scene.[15]

In this paper we use a recent interactive segmentation model of 3D point clouds from this paper [8] which is based on the Minkowski Engine for our segmentation tasks. This segmentation model is compared to other 3D instance segmentation based on weakly-supervised information and not on fully-supervised information. This method does also not require training data from any target domain and adapts to new environments which do not have any training sets available.

In this chapter, we describe the most important features, which we also use in later chapters for our reinforcement learning project.

2.2.1 Structure of Segmentation System

The segmentation is generated in an iterative approach. The user clicks in each iteration on a point in the point cloud. After each click, the underlying model produces its best guess of a segmentation based on all the collected clicks in this run. After receiving the mask, the user provides a feedback via corrections, followed by a new update of the mask.

Input representation The input to the segmentation network consists of the colors of the scene $C \in \mathbb{R}^{Nx3}$ and two channels $T_p, T_n \in N \times 1$ which represent the negative and the positive click mask. It is important to note that the negative clicks correspond to the background clicks and the positive clicks to the foreground. After each seed added by the user, the system adds all points in a small radius around the seed coordinate to the input channels. In further sections of this paper we refer to this area as to the *clickmask* of the seed. Therefore T_p and T_n is defined as follows:

$$T_p(p) = \begin{cases} 1, & \text{if } |x_p - x_q| \leq \varepsilon \text{ and} \\ & |y_p - y_q| \leq \varepsilon \text{ and} \\ & |z_p - z_q| \leq \varepsilon \\ 0, & \text{otherwise} \end{cases}$$

x_q, y_q and z_q are the coordinates of any point in S_p which is the set of positive clicks chosen by the user. ε is the volume length. T_n is similarly defined with any point in S_n (set of negative clicks chosen by the user). After generating T_p and T_n, those channels are concatenated with the scene colors and result in an input of dimension $N \times 5$.

2.2.2 Automatic Click Generation

The project from [8] also includes a user click simulation for its segmentation tasks. During testing phase of the segmentation model, clicks are added based on the errors of the currently predicted mask. The click will be located on the coordinate with the largest error. By calculating pair-wise distances between all falsely labeled 3D points they get demanded error region. There are problems with this heuristics. Choosing the point which has the largest error region sometimes is not precise enough for objects with small emerged parts. Thus for particular objects, like a cup with a small handle, the heuristics is not the optimal choice to simulate user clicks since it would need a lot of predictions of the segmentation to be accurate.

The main goal of our paper is to replace this click simulation by using reinforcement learning methods. At the moment the model only needs on average 14.7 clicks per object in the ScanNet data set[3] to achieve an accurate segmentation of 90%.[8]. We try to reduce the number of clicks such that simulation of user clicks can happen more efficiently.

2.3 SeedNet Automatic Seed Generation System

SeedNet[10] is a paper that proposes a seed generation system for the interactive segmentation task of 2D images. A user enters an image and sparse seed information to the system and SeedNet creates additional seed points to obtain accurate segmentation results. The seed points intend to predict the user's intention and reduce the user's effort. SeedNet uses Random Walk (RW)[9] as an off-the-shelf interactive segmentation algorithm and is trained on multiple 2D data sets. The first click is generated randomly for each image, SeedNet then generates the other clicks with the trained agent. The agent predicts each click on a 20x20 grid of the whole scene. On average SeedNet can increase the segmentation results from 39.72 IoU (Which correspond the

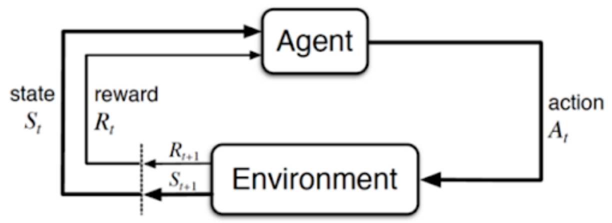


Figure 2.2: Graphical representation of the MDP model.[5]

precision of RW segmentation on the first random click) to a value of 60.97. In this work, we try to build our own click generator which is mostly based on the ideas of this Seed Generation System.

2.4 Markov Decision Process (MDP)

A Markov decision process (MDP) is defined as a stochastic decision-making process that uses a mathematical framework to model the decision-making of a system. It is used for scenarios where the results are either random or controlled by a decision maker. MDP chooses the next best action in a system by considering the current state and the environment of the system.

How does it work? The MDP model requires key elements such as an agent, states, actions, rewards and a policy. The agent is responsible for making decisions and performing actions. It operates in the environment, which has information about the states and also the transitions from one state to another. Additionally the agent receives rewards based on the actions that the agent made. The policy determines how the agent selects the action depending on its current state.
The MDP framework has the following key components:

- $S : states(s \in S)$
- $A : actions(a \in A)$
- $P_a(s, s') = P[s_{(t+1)} = s' | s_t = s, a_t = a]$ is the probability that action a in state s will lead to state s′ in the next time step.
- $R_a(s, s')$: reward after transitioning from s to s′ by executing action a.

The goal of MDP is to find an optimal policy function $\pi : S \to A$. The policy function defines the next action that an agent will choose when in state s. To be more precise: The object is to choose a policy π that will maximize the expected cumulative collected rewards.

$$E\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right]$$

where we choose $a_t = \pi(s_t)$ which is equal to actions given by the policy. γ is the discount factor $0 \leq \gamma \leq 1$ which is usually close to 1.[16]

2.5 Reinforcement Learning

Reinforcement learning is concerned how an agent takes actions in an environment to maximize its cumulative reward. It differs from supervised learning, because it does not need labelled input/output pairs. It instead tries to find an optimal policy by exploration of unknown knowledge and exploitation of current knowledge. Reinforcement Learning is a form of a Markov Decision Process (MDP) and has therefore similar definitions of state, action, probabilities and reward functions. There are several ingredients for Reinforcement Learning which are presented in this chapter.[17]

Exploration vs Exploitation The exploration vs exploitation trade-off is mostly based on a ε-greedy method. $0 < \varepsilon < 1$ is a parameter controlling this trade-off. With probability $1 - \varepsilon$ exploitation is performed. The agent chooses the action which it believes to have the best long-term effect. Else with a probability of ε it chooses the exploration strategy, where the action is selected uniformly at random. ε is a fixed parameter but can be adjusted while training.[17]

Value Function The goal of Reinforcement Learning is to find an optimal policy, such that an agent can find the best action in a given state. The Value function represents the value of the given state and it maintains a set of estimates of expected returns for some policy. Usually we speak of the current (on-policy) or the optimal (off-policy) one.

To define an optimal policy, we define the Value function as follows:

$$V^{\pi}(s) = E[R|s, \pi]$$

R stands for the return associated with the current state s and the policy π. By defining $V * (s)$ as the maximum possible value of $V^{\pi}(s)$, depending on the policy π, we can formally say in Reinforcement Learning, we want to find:

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

We equally define R as the reward in the Markov Decision Process section and then combine this with the Bellman Ford equation and we receive our state-value function, which we will need in future chapters.

$$\begin{aligned} V^{\pi}(s) &= E\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}\left(s_t, s_{t+1}\right)|s, \pi\right] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s') \end{aligned}$$

In MDP we are looking for the optimal policy and therefore we can use the Bellman optimality equation:

$$V^{\pi *}(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s'|s,\pi(s)) V\left(s'\right) \right)$$

[17]

2.6 Q Learning

Until now, we only evaluated the value of going to a particular state, where we took the stochasticity of the environment in consideration:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P\left(s'|s,\pi(s)) V\left(s'\right)\right) \right)$$

Q learning assesses the quality of an action, rather than determining the possible value of the state being moved to. Additionally the Q function produces the value for just one possible action. Therefore we should replace the value function with the action-value function or Q function.[13][17]

The optimal Q function also follows the Bellman optimality equation:

$$\begin{aligned} Q(s,a) &= E[R|s,a,\pi] \\ &= E[\sum_{t=0}^{\infty} \gamma^t R_{a_t}\left(s_t, s_{t+1}\right)|s,a,\pi] \\ &= R(s,a) + \gamma \max_{a'} Q(s',a') \end{aligned}$$

Here we calculate the maximum return of state s and action a as the sum of the immediate reward R and the return of the optimal policy until the end of the episode, which is equal to the maximum reward from the next state s′ discounted by γ.[1]

2.6.1 Temporal Difference Learning

Temporal difference Error is the value that helps the agent to get the optimal Q values over time. We know that the environment is stochastic and that the reward at time t is composed of **discounted rewards** in the future. This implies that future rewards have less value.

The **TD Error** is the difference between the optimal action-value function (Q*) and our current prediction (Q).[13]

$$\begin{aligned} TDerr(a,s) &= Q^*(s,a) - Q(s,a) \\ &= (R(s,a) + \gamma \max_{a'} Q(s',a')) - Q(s,a) \end{aligned}$$

We update the Q(s,a) value as follows in Q learning:

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha TDerr_t(a,s)$$

2.7 Deep Q Network

The Deep Q Network was developed by DeepMind in 2015. By combining reinforcement learning and deep neural networks it could solve a wide range of Atari games. The algorithm was developed by combining Q learning with deep neural networks and a technique called experience replay.

In this section whenever we talk about policy, the optimal network parameters are meant.

Most of the times, it is not practical to calculate an table containing Q values for each combination of state s and action a. Instead we train a function approximator, like a neural network, with parameters θ to estimate the Q values.[1]

General Policy Iteration In a Deep Q Network we alternate between policy evaluation and policy iteration. At first we should start with some arbitrarily initialized policy, evaluate the policy (E) and then derive a new policy from the evaluation process (I). We repeat this process until we reach an optimal policy. So we can say in a GPI sense that we derive our policy from our Q function and carry out policy evaluation via TD methods to obtain the next Q function.

$$\pi_0 \rightarrow^E Q_{\pi_0} \rightarrow^I \pi_1 \rightarrow^E Q_{\pi_1} \rightarrow^I \cdots \rightarrow^I \pi_* \rightarrow^E Q_{\pi_*}$$

Now let's add the parameter θ to our Q function, which denote the parameters of the neural network. Following GPI, we want to minimize the difference between our current Q and our target Q at each time step i. We take the mean squared error between both of them:[20]

$$\begin{aligned} L(\theta_i) &= \mathbb{E}_{s,a,r,s' \sim p(.)}[(y_i - Q_{\theta_i}(s,a))^2] \\ &= \frac{1}{N} \sum_{j \in N} (Q^*_{\theta_i}(s_j,a_j) - Q_{\theta_i}(s_j,a_j))^2 \end{aligned}$$

where

$$y_i = Q^*_{\theta_i}(s, a)$$

$$Q^*_{\theta_i}(s_j, a_j) = R(s_j, a_j) + \gamma \max_{a'_i} Q_{\theta_{i-1}}(s'_j, a'_j)$$

y_i is also called TD target and $y_i - Q$ the TD error as before. p is equal to the distribution over transitions {s, a, r, s′} collected from the environment. The parameters from the previous iteration θ_{i-1} are fixed and not updated. θ_{i-1} represent the parameters of the target network. In practice, the target network is a snapshot from a few iterations ago.[1]

With gradient ascent, we minimize the TD error.

$$\theta \leftarrow \theta - \alpha \frac{\vartheta L}{\vartheta \theta}$$

Experience Replay With experience replay, the network updates are more stable. We implement a circular buffer, also called replay buffer, and at every time step of data collection, we collect { s,a,r,s′} tuples and add them to the replay buffer. During training we are not only using the latest transition to compute the loss and its gradient, we are using a mini-batch of transitions sampled from the buffer. We improve the stability, because the transitions are uncorrelated to each other.[1]

2.8 Optimizations

2.8.1 Double Deep Q Networks

In the chapter of Deep Q Networks, we already mentioned the parameters of the target network and the idea to not update the parameters of this network. In this chapter, we explain further why this is helpful. Consider the target Q value:

$$Q^*(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

Specifically:

$$\max_{a'} Q(s', a')$$

Here we are taking the maximal estimated value and therefore implicitly estimating the maximum Q value in this function. This overestimation introduces a maximization bias in learning and the Moving Q Targets Problem.

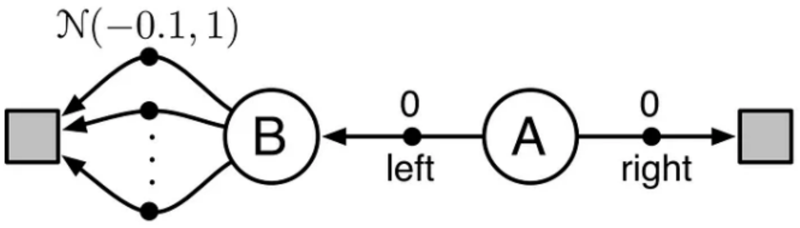


Figure 2.3: Example of Maximization Bias in Q Learning[4]

Maximization Bias in Q Learning We say that a Q learning algorithm is overestimating the value function (V) and the action-state function (Q). This problem will be explained with this example:

Let's say the agent chooses in state A the action to go right, because in earlier experiences agent A discovered that going to state B will result in a negative reward of -0.1. By only considering the expected rewards, the agent will profit more from going right than going left. But if we also include the variance in our analysis, agent A quite often receives a bigger reward by going to state B.[4]

Moving Q Targets By revisiting the TD Error from earlier:

$$\begin{aligned} TDerr(a,s) &= Q^*(s,a) - Q(s,a) \\ &= (R(s,a) + \gamma \max_{a'} Q(s',a')) - Q(s,a) \end{aligned}$$

The first component of the TD Error is calculated with the immediate reward plus the discounted max Q value for the next state. While training our agent, we update the weights of the network by using the TD error. But the same weights apply to both the target Q value (Q*) as well as the predicted Q value. So we end up approaching the target value, but also moving the target. This leads to high oscillation issues during the training process.

To handle both problems, we use two Deep Q Networks:

- The online network is responsible for the selection of the next action (depending on the next state) as always.
- The target network is responsible for the evaluation of the next action.

This approach is called using a Double Deep Q Network. The target Q values of the next state, i.e. $Q(s', a')$, is based on the prediction on the target network. The target network remains fixed for a certain number of steps. But the selection of the action on the next state, i.e. $\max_{a'}$ relies on a prediction of the online network. Basically, we decouple action selection from the target

Q value generation and can therefore substantially reduce the overestimation. Additionally since the target network remains fixed for some iterations, we also solve the moving target problem.[6]

2.8.2 Dueling Deep Q Network

In Q Learning, Q values are ways to see how well an action fits to a particular state, that's why it is also called action-value function. This metric is the same as the expected return of executing an action while being on a state.

$$\begin{aligned} Q(s,a) &= E[R|s,a,\pi] \\ &= E[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\,|s,a,\pi] \\ &= R(s,a) + \gamma \max_{a'} Q(s',a') \end{aligned}$$

Q values can be decomposed into two parts: the state value function (V(s) as in 2.6) and the advantage value (A(s,a)). We introduce a new variable, the advantage value:

$$A(s,a) = Q(s,a) - V(s)$$

Recall that the Q value represents the value of choosing an action at a certain state s. the state value function is assessing the value of being in a certain state. Then intuitively the advantage function describes how good an action is compared to other possible actions while being at the given state. In a Dueling Q Network, we change the representation of the Q function to a sum of the value function aggregated with the advantage function:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|A|}\sum_a A(s,a))$$

Note that we cannot simply add the advantage function, because the naive sum of the two is "unidentifiable", which means that we cannot recover V and A uniquely. It is empirically shown in Wang et al. that this would lead to poor practical performance.
By doing this transformation, we split the features of the Q network into two separate estimators. The value estimator describes which states are valuable and which are not, without having to learn the effect of each action for each state. This architecture is especially relevant in tasks where actions might not affect the environment in significant ways.[6] [19]

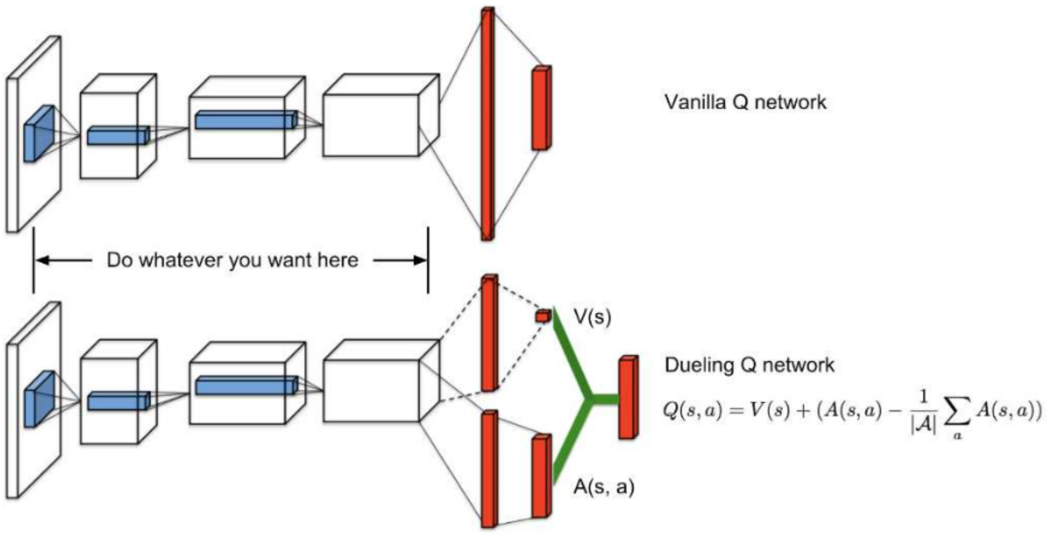

Vanilla Q network
Do whatever you want here
V(s)
Dueling Q network
$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|}\sum_a A(s,a))$
A(s, a)

Figure 2.4: Abstract model of Dueling Network Architecture[19]

Chapter 3

Method

3.1 Markov Decision Process

We first define in this work our MDP model. We tried to adapt the paper from SeedNet[10] to a MDP model in 3D space: As N we define the number of points in the 3D point cloud, which represents the scene.

3.1.1 State

We are given a 3D scene $P \in \mathbb{R}^{N \times 3}$ consisting of xyz coordinates and the corresponding RGB scene colors $C \in \mathbb{R}^{N \times 3}$. We are using the whole information of the picture, therefore it is necessary to include the scene colors in the state. At every step of the segmentation process, there are two things generated. One is the seed map and one is the resulting binary mask $B \in \mathbb{R}^{N \times 1}$ from the segmentation model. The segmentation network also needs the seed map as input, that's why we try in some experiments to include the seed map. In most experiments the seed map is excluded.

All in all, as a state S we have the dimension $S \in \mathcal{R}^{N' \times 4}$, where N' is equal to the number of points after the sparse voxelization of the Minkowski Engine.

3.1.2 Action

The agent predicts an action given a state. The action within an action space is to position a new seed point in the 3D scene. The position of the seed point is based in our experiments either on a regular 3D grid or on the voxelization of a part of the scene. The coordinate of the next seed is the center of the grid or the center of the voxel. The label is taken from the ground truth mask of the ScanNet[3] data set in our experiments.

Since it is hard to define a termination state, we are terminating one episode after 10 steps. Steps correspond in this case to an agent predicting a seed location.

Instead of taking the whole scene, we restrict the action area according to the object we try to segment. There are two variants evaluated in our experiments, but both are based on a bounding box around the object:

Bounding box : We first determine the center of all positive points as midpoint in our ground truth mask. After selecting the maximal and minimal xyz coordinates of the positive foreground points, we have the dimension of a bounding box including all foreground points. Since we often also need to click on the background for a good segmentation, we increase the diagonal of the bounding box by a stretch factor sf to include more negative background points. After having the min and max coordinates of the bounding box, we ignore all points which are outside of the bounding box for our action space.

Sparse Voxelization : We are using the built-in voxelization method of the MinkowskiEngine to rasterize the area where the clicks are located. One voxel is equal to one 3D grid in the action space. This method is a good replacement for the already implemented heuristics which also determines actions on a 0.05 voxelization of the image, but since its number of voxels per area is different for each object, we can't use this method to train our network on multiple scenes. The number of grids determine the action space and the network the corresponding dimensions. And to our knowledge, there is no voxelization method, where the number of voxels are fixed.

Regular 3D grid : We rasterize our bounding box into regularly distributed grids. This method is simple to implement and does create a fixed amount of grids, compared to the voxelization method. But we can not easily determine the label of each grid. In the sparse voxelization method we can reduce the scaling of the ground truth with the same voxelization as the scene coordinates and therefore easily determine the label of each grid. In the 3D grid method we are doubling the action space and let the agent guess the label of the seed.

3.1.3 Reward

The reward evaluates the action of the agent. In our work we try out different kind of reward systems.
The first version of our reward function is to take the general IoU value.

$$R_{IoU}(s,a) = IoU(M,G)$$

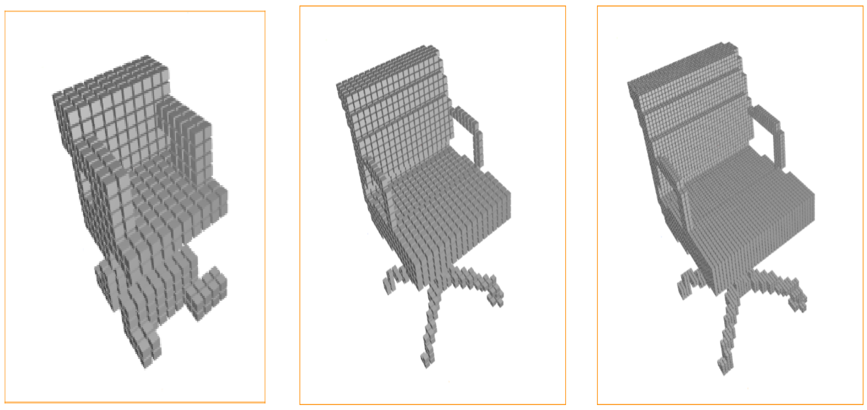


Figure 3.1: Example of the built-in voxelization method applied to a chair. The number of voxels per cm increase from left to right.

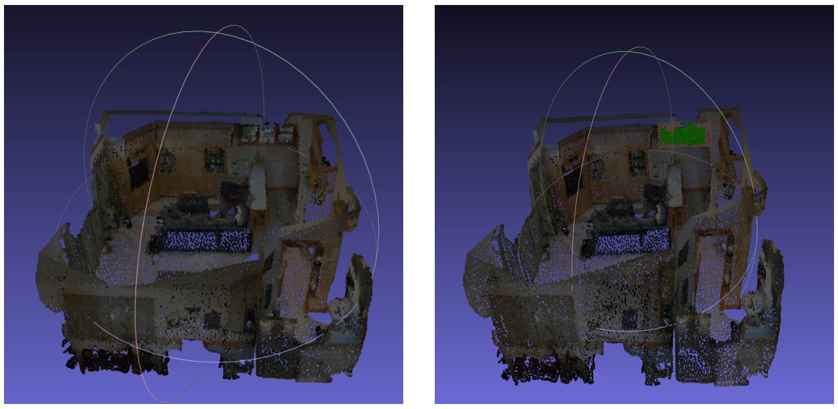


Figure 3.2: With a sparse voxelization of 0.02 (1 point/2 cm) and a stretch factor of 1.2, we have the following foreground points (green) and background points (pastel) as action space.

which is a common metric in image segmentation. It describes the precision of the segmentation M compared to the ground truth G in training.

The second version takes the relative change between the current IoU value and the IoU value before into consideration. With this reward function, small changes in IoU will be punished harder and it should be an incentive for the agent to focus more on different seed points, instead of taking the same actions.

$$R_{diff} = IoU(M, G) - IoU(M_{prev}, G)$$

To even further motivate the agent to not take the same coordinates for the seeds. We introduce an error action: Assume the agent selects a seed point twice in an episode, then the first click will be rewarded with one of the standard reward function mentioned above. The other one (and all further clicks on the same location) is counted as a bad click, because in our state we already inserted this click. Repeating an action is in this case not helpful and does not provide the agent a lot of additional information compared to deciding for another action in the action space.

$$R_{noDouble}(s_t, a_t) = \begin{cases} R & \text{if } a_t \neq a_i, \forall i < t \\ -1 & \text{otherwise} \end{cases}$$

where t refers to the time step during an episode.
In addition to the changes made so far, we divide the state into 4 regions as in the SeedNet paper[10]. The regions are called strong foreground (SF), weak foreground (WF), weak background (WB) and strong background (SB). We divided the state by introducing 2 boundaries, they correspond to boundaries within the background and within the foreground points. We get the boundary between strong foreground and weak foreground by first calculating the pairwise distances between center of foreground points to all other foreground points. Then we take the median of those distances as our boundary. We can be sure that there are an equal number of points in the SF, as in the WF section. Analogously we do it between the weak background and the strong background section.

By introducing different reward functions based on the location of the seed point, the prediction ought to be more precise.
By using an exponential IoU reward instead of an linear IoU reward, we give more attention to changes in high IoU values.

$$R_{exp} = \frac{exp^{k \cdot IoU(M,G)} - 1}{exp^{k} - 1}$$

where k is a constant value.

We can formally recap the reward function used as follows:

$$R_{\text{regions}} = \begin{cases} R_{\text{exp}} & \text{if } P_{\text{seed}} \in \text{SF or SB} \\ R_{\text{exp}} - 1 & \text{if } P_{\text{seed}} \in \text{WF or WB} \end{cases}$$

Where P_{seed} means the the current seed selected for evaluation. In the SeedNet paper they are differentiating between the different labels of the selected seed. In this work we use the same labels as in the ground truth mask. Therefore we only differentiate between the location of the selected seed. In our experiments we combine different variants of reward functions for evaluation.

3.2 Training of the Reinforcement Learning Agent

3.2.1 Deep Q Network (DQN)

With the proposed MDP formulation, we can train the agent by using reinforcement learning. To train the agent on one scene we iterate through a number of episodes. In each episode the agent chooses an action and the Deep Q Network evaluates the action with a reward. After 10-15 actions the episode ends and we reset the state and the input of the segmentation network.
We define the Q learning target with the given s, a, s':

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

where γ is the discount factor and s' and a' represent the state and the action of the next step.
With DQN we approximate the Q values with a deep Q neural network. The loss function for training can be expressed:

$$Loss(\theta) = \mathbb{E}[(R(s,a) + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)^2]$$

which is the same as the TD Error from the theory part. We use a mean squared error and gradient ascent to update the neural network.

To perform the exploitation vs exploration trade off, we use ε-greedy policy as a behaviour policy. The ε-greedy policy uses a random action with a probability of ε (exploration) and an action that is based on the max Q Value with a probability of $1 - \varepsilon$ (exploitation). At first $\varepsilon = 1.0$ which means the action is chosen randomly. Then during the first E steps, ε decreases linearly to a fixed final value for the rest of the training.
We create an experience replay buffer for training our agent. After each action during training, we collect the reward and the next state and store the replay memory (s, a, r, s') in our experience replay buffer. For updating our network we collect a batch of experiences from our buffer to perform Q learning on this batch.

3.3 Model Architecture

The DQN used in this study is similar to the one in [10], instead of using the TensorFlow Keras framework as in [12], we use the Minkowski Engine framework for Sparse Tensors. We expect a better synergy with the already existing interactive 3D object segmentation from [8]. We also use the double

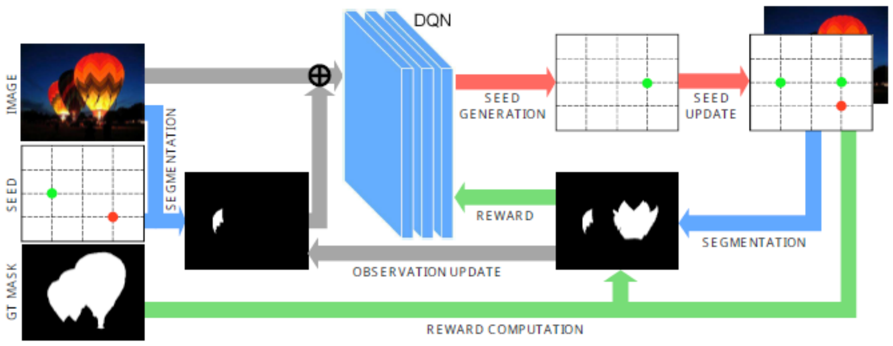


Figure 3.3: Our system is similar to the SeedNet[10] system: The scene and the segmentation mask are inputs to the DQN. For the reward function we compare ground truth and the resulting segmentation mask. This process is repeated 10 times per episode. The gray arrows represent state-related behavior, red arrows action-related behavior and green arrows indicate reward-related behavior.

DQN structure of [18] and the dueling DQN structure of [19]. As input to the network we try out several different sparse voxelization (1 point / 5 cm, 8 cm, 10 cm) of the Minkwoski Engine to reach a similar efficiency as in the SeedNet paper. We perform 3 convolutions each followed by a ReLU activation in the Network. The first layer has 32 output channels with a kernel of size 8 and a stride of 4, the second has the same properties but with 64 output channels and the last layer has 64 output channels with a kernel size of 3 and a stride of 1. After the convolutions, a flattening layer is used into 2 fully connected linear layers, the first 512-D layer is split into state-value and advantage function which is the last layer. The state-value function $V(s)$ is a scalar value and the advantage function comes out with the same dimensions as the action space.
The action is determined according to the Q Value with the maximum value. Then the action label (which is an integer between 0 and $\mathcal{A}$, where $\mathcal{A} \in \mathbb{N}$ is the size of the action space) is converted to grid coordinates. After the conversion we know where the new seed will be located.

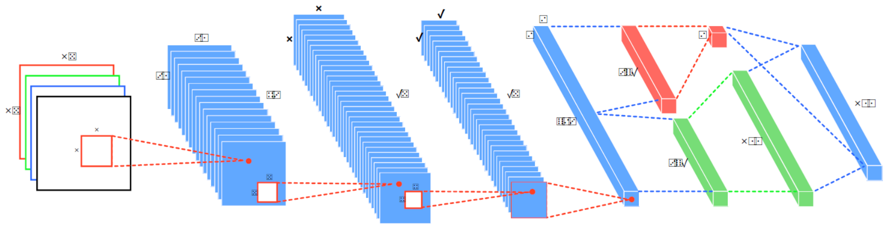


Figure 3.4: The architecture of our DQN is similar to the one used in [10]. The red layers are for the state value function and the green layers are for the advantage function.

Chapter 4

Experiments

4.1 Network learning

In our experiments, we tried to reach an acceptable IoU value for one particular scene of the ScanNet data set. In most of our experiments, 600 episodes for training are enough for the network to minimize the loss function. We first iterate through 2000 pre-training steps to only collect and store experiences in the buffer, the training does not proceed in this phase. ε is reduced within the first $E = 1000$ training steps from 1.0 to a final value of $\varepsilon = 0.2$. The parameters of the network are described in section 3.3. The discount factor is set to $\gamma = 0.9$. An episode ends after every 10 seed generation processes. For training we use an Adam optimizer[7] and utilize a learning rate of 1e-4. We set the stretch factor for our action space to $sf = 1.2$ and the sparse quantization for our action space is 0.05 (1 point/5 cm) which results in an action space of 597 possible seed locations.
We reduce the original point cloud from 24769 points through sparse voxelization of 0.05 to a density of 15880 points. Additionally for each click location generated by the agent, all points within a distance of 5 cm from the center is added to the input channels for the segmentation system.

We update the online network every 4 steps to be more efficient and the target network with an update rate of 1e-7. In [10] the target update frequency, the number of pre-training steps and the training steps are higher scaled but due to performance reasons and technical issues in this work we try a lower scale version as a proof of concept.

4.2 Evaluation metrics

All the evaluation in this section are based on the following metrics:

- **Episode Reward**: The sum of rewards collected during an episode.
- **Training Loss**: The sum TD Errors between target Q value and Q value of the online network during an episode.
- **Minimal/Maximal and Average IoU**: Metrics to evaluate our method with the heuristics already implemented in the segmentation system[8]

The first seed is generated using the heuristics of the segmentation network. After the first seed, the segmentation mask has a value of IoU = 22.8%.

4.3 Training Results

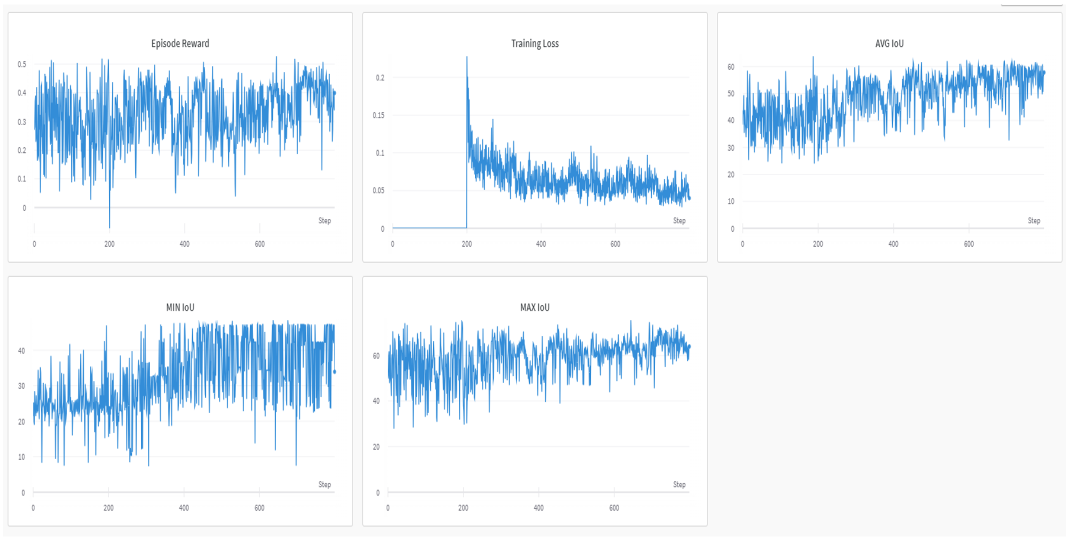


Figure 4.1: In this experiment, we used all the parameters mentioned in 4.1 and R_{diff} as our reward function.

As we observe in 4.1 the average IoU increases from a starting value of 22.8% to an average IoU of 55.5% at the end of the training period of 600 steps. During pre-training the average IoU was at around 40.0%, therefore we see an increase in precision during training. The maximal IoU during an episode is at 65%, which is compared to the results of the SeedNet system in 2.3 a solid result. We also observe the large variation of the minimal IoU and an increase in the episode reward.

4.4 Reward Evaluation

We observe that the R_{IoU} and the R_{diff} curve are similar. One can say the R_{IoU} curve is more accurate in terms of IoU compared to R_{diff}. Nonetheless

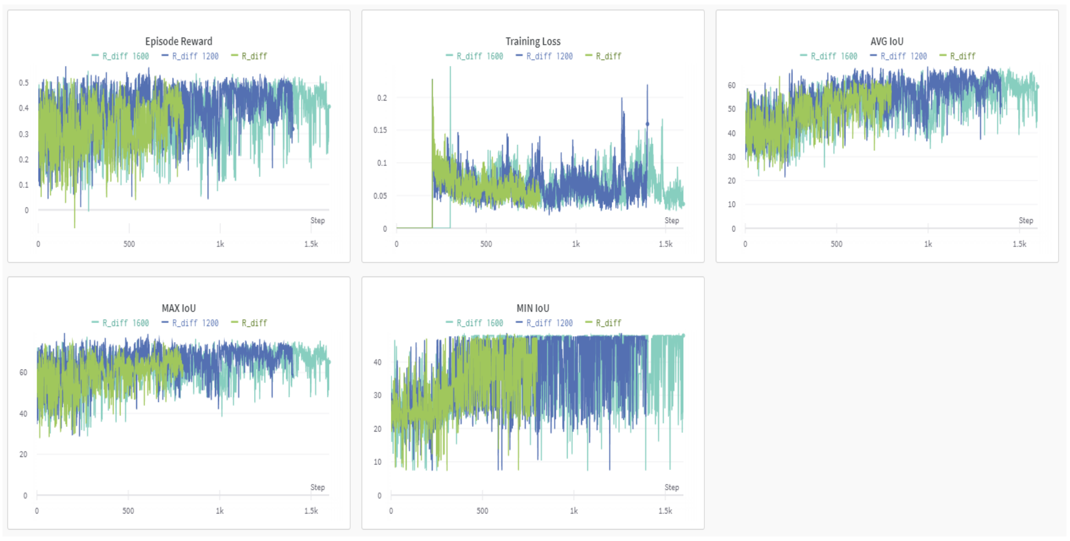


Figure 4.2: We increase the number of training iterations for our setup to a number of 1200 and 1600. By testing this setup on more training iterations as in 4.2, we realize that the average IoU curve and the loss curve do not increase by a lot.

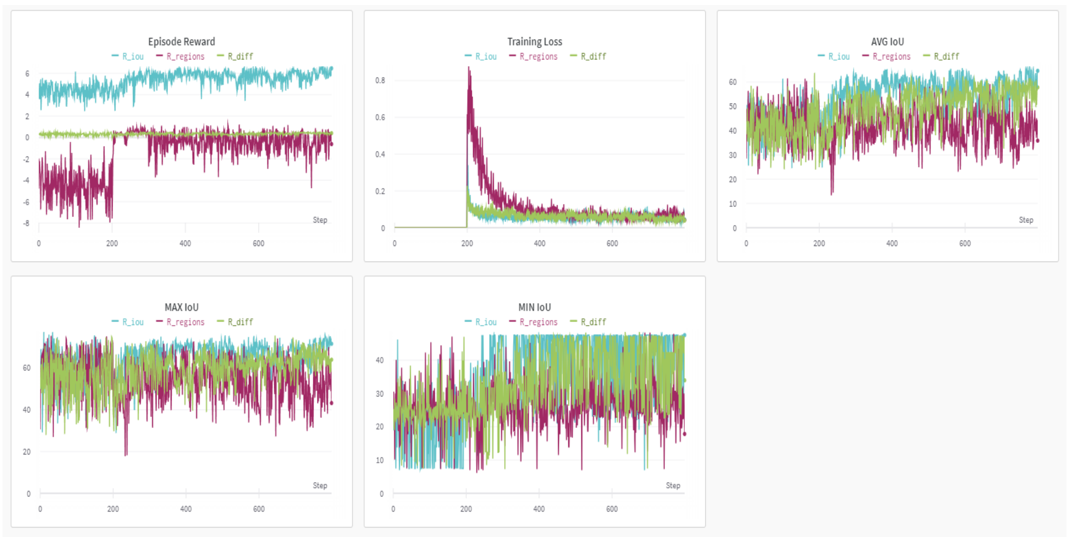
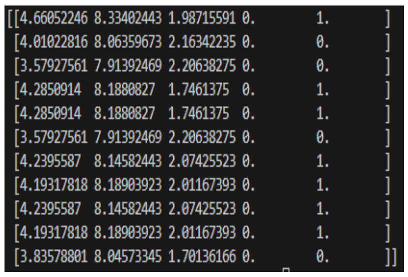


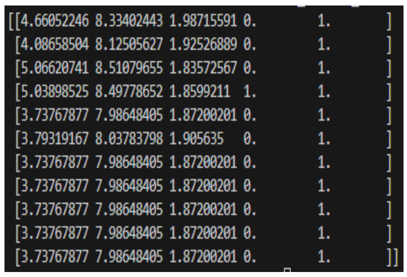
Figure 4.3: We evaluate the training curves of all the reward functions mentioned in 3.1.3. The blue curve is the reward R_{IoU}, green is R_{diff} and red represents $R_{regions}$.

do we prefer R_{diff} for our agent. R_{diff} shows better results in preventing the same action of the agent as shown in 4.4.

The curve of $R_{regions}$ does not satisfy our expectations. In SeedNet $R_{regions}$ is used for the agent to locate its seed predictions more precisely. But in our experiments, this setup did not increase the resulting IoU. The IoU curves are below the other reward functions and hence less accurate. To find a reasoning



```
[[4.66052246 8.33402443 1.98715591 0.         1.        ]
 [4.01022816 8.06359673 2.16342235 0.         0.        ]
 [3.57927561 7.91392469 2.20638275 0.         0.        ]
 [4.2850914  8.1880827  1.7461375  0.         1.        ]
 [4.2850914  8.1880827  1.7461375  0.         1.        ]
 [3.57927561 7.91392469 2.20638275 0.         0.        ]
 [4.2395587  8.14582443 2.07425523 0.         1.        ]
 [4.19317818 8.18903923 2.01167393 0.         1.        ]
 [4.2395587  8.14582443 2.07425523 0.         1.        ]
 [4.19317818 8.18903923 2.01167393 0.         1.        ]
 [3.83578801 8.04573345 1.70136166 0.         0.        ]]
```



```
[[4.66052246 8.33402443 1.98715591 0.         1.        ]
 [4.08658504 8.12505627 1.92526889 0.         1.        ]
 [5.06620741 8.51079655 1.83572567 0.         1.        ]
 [5.03898525 8.49778652 1.8599211  1.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]
 [3.79319167 8.03783798 1.905635   0.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]
 [3.73767877 7.98648405 1.87200201 0.         1.        ]]
```

Figure 4.4: We compare the predicted seeds of the last episode from R_{diff} (left) and R_{IoU} (right). The fourth column is the random bit and the fifth column is the label of the click.

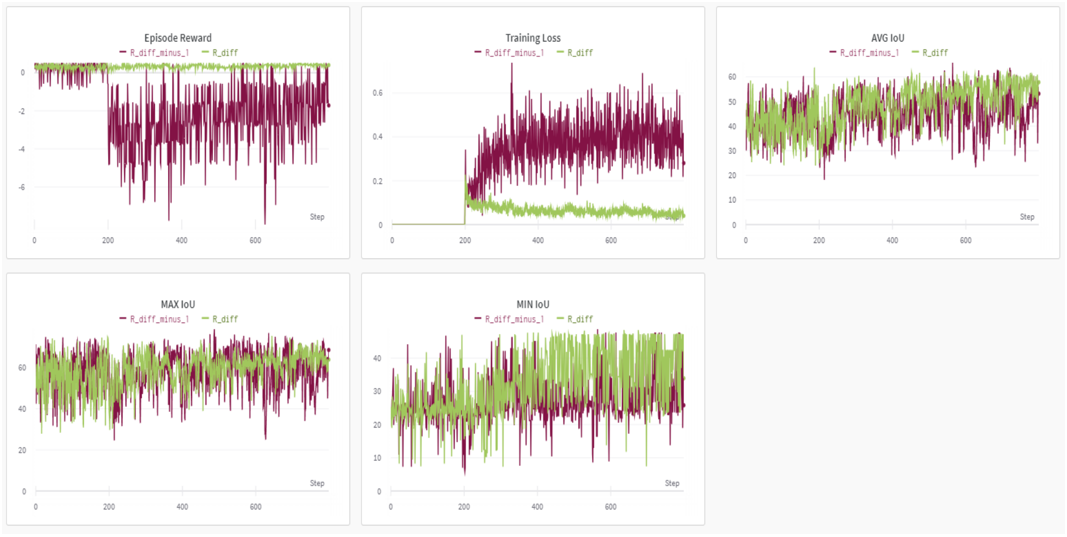


Figure 4.5: We evaluate the training curves of R_{diff} in green and $R_{noDouble}$ combined with R_{diff}

for this behaviour, our agent is already locating it's seed predictions in the correct region because of the limited region of our grid. Thus grouping the potential seed locations into several regions is not helpful. In a setup without limiting grid, the $R_{regions}$ might have more impact on the precision of the agent.

Evaluation Penalty In 3.1.3 we introduce $R_{noDouble}$ and in our experiments we observed the results in 4.5. As expected, we notice that the loss curve is relatively high and the reward curve is relatively low compared to the curves of the R_{diff}. The episode reward curve of $R_{noDouble}$ is showing a convergence towards 0 during training. We can therefore derive that the number of replicated actions during an episode must decrease. Arguably by comparing the IoU curves between both functions, we cannot observe any significant improvement on either side.

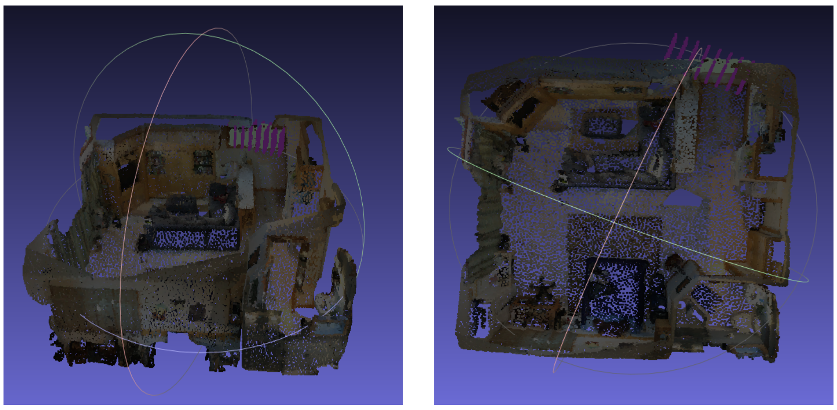


Figure 4.6: On both pictures, we see our grid example. The dots presented in pink are representing the coordinates belonging to our regular grid. The green dots represent the actual foreground voxels of the object. By comparing the pink with the green dots, we observe high inaccuracies between foreground points and grid points.

4.5 Grid Evaluation

Beside our original grid based on the voxelization method of the Minkowski Engine, we tested a more regularly distributed grid in the 3D space. To evaluate which grid works best as action space for our deep Q network.

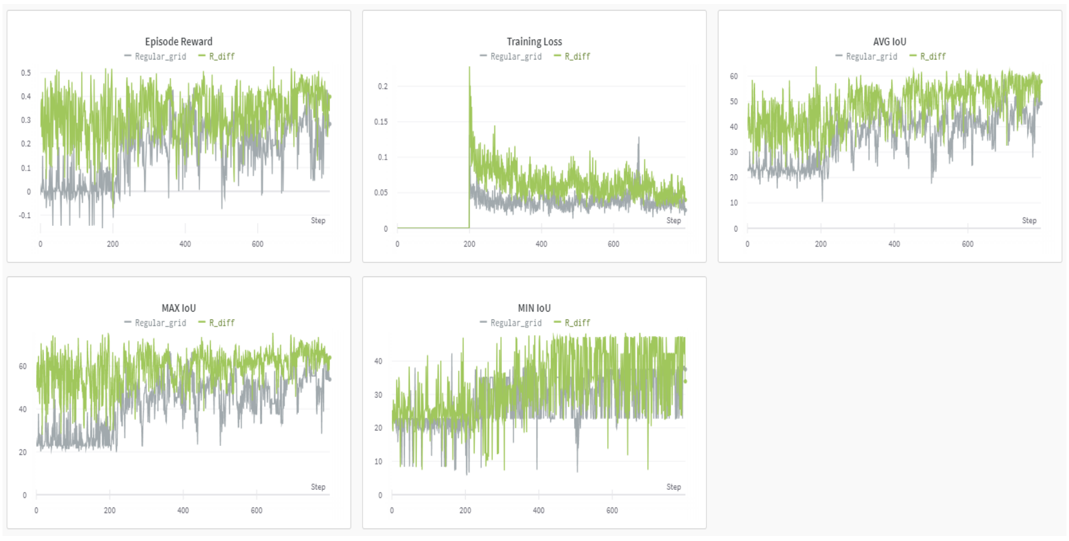


Figure 4.7: The results when evaluating both grid versions in 3.1.2. The grey curve represents the regular grid from figure 4.6 and the green one represents the voxelized action space.

We observe that the all the IoU curves of the regular grid are below the IoU curves of our voxelized version. This seems to confirm our initial thoughts. When looking at the episode reward function, we see a higher

Table 4.1: Performance evaluation for one scene of ScanNet data set:

	100	150	600	1000
update every step	18min	36min	115min	165min
update every 4 steps	5min	10min	40min	65min

Table 4.2: By updating our network only every 4 steps, we observe an improvement in performance.

growth between pre-training and actual training. This could indicate that there is again a problem with repeatedly choosing the same actions during training.

4.6 Issues and Future Work

In this section we discuss issues that arose during our experiments and present attempts to solve the problems.

4.6.1 Performance Issues

When training our network on the ScanNetV2[3] data set, we are observing issues with performance. The ScanNetV2 data set contains roughly 1200 scenes. It is currently the largest data set of 3D scenes. However compared to data sets for 2D segmentation tasks, it is still relatively small. Testing on a large part of the data set is important for improving the stability of our network on unseen data sets. The results in 4.1 show that training only on one scene requires quite some computational effort and therefore also a lot of time. In reinforcement learning projects like [10] one scene is trained with 5000 episodes. By making a projection on our time requirements it would require 5.5h to train one scene for 5000 episodes with the voxelization of 0.05 and update rate of 4 steps.

Potential Fix To improve the performance of our network, we can reduce the number of points in the 3D scene such that the state does not have a too large state as input. In the current voxelization of 0.05, the input state has a dimension of $S \in \mathbb{R}^{15'880 \times 4}$. When we compare this to the dimension of the SeedNet system[10] $S \in \mathbb{R}^{7'056 \times 4}$ we observe a large difference.

4.6.2 Learning Stability

In several experiments we observed a highly irregular learning stability, which slowed down, or even impeded, the training progress. In reinforcement learning the update rate of the target network does have a high impact on the learning stability. In [12] and in [10] the proposed solution is to update the target network by replacing the target network parameters completely by

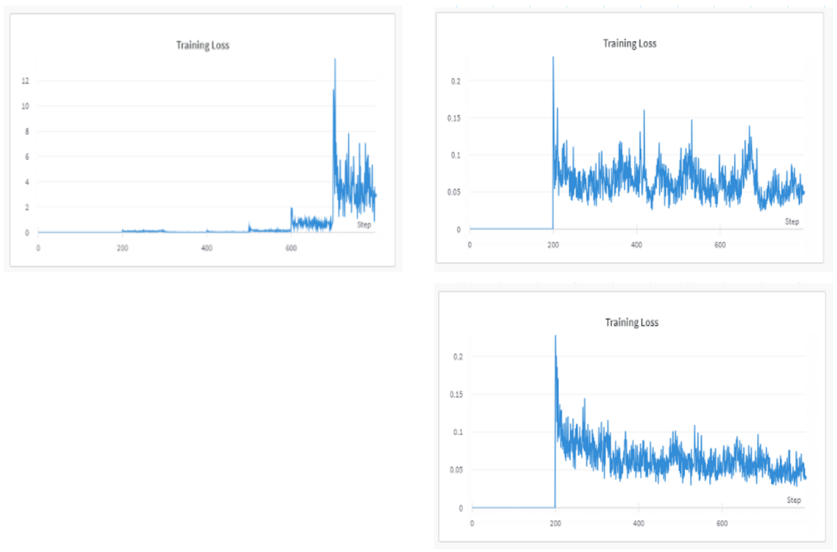


Figure 4.8: We evaluate the stability of our experiments, the first loss curve (top left) shows a complete update in the target network every 1000 steps, the one top right and bottom right show a update rate of 1e-4 and 1e-7 respectively. The loss curve on top is by some factors higher than the other curves.

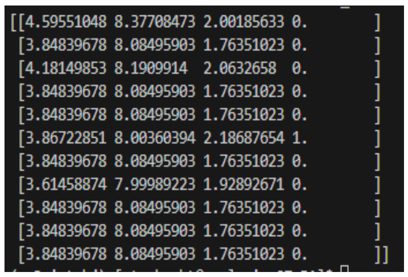
the parameters of the online network. When using this method our training results show a lot of irregularities and therefore loses stability. Our approach is to use a Deep Deterministic Policy Gradient (DDPG)[11]. DDPG maximizes the expected cumulative long-term reward by introducing a smoothing factor $0 \leq \tau \leq 1$ to update the target parameters. The target parameters are updated at every time step.

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

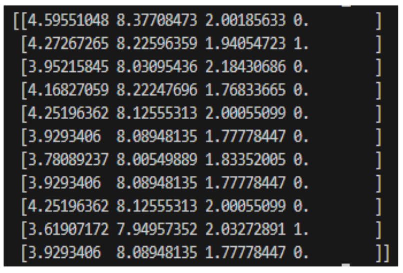
θ' and θ represent the weights of the target network and the current network. This update is called a *soft update* and it has been used in the paper for DDPG. Assuming $\tau = 0.0001$, the new weights of the target network will consist to 0.01% of the online network and to 99.99% of the previous target network. In 4.8 we observe a substantial change of the loss curve when running different versions of the update pattern.

4.6.3 Predicting the same Action

The following experiment states a problem that occurred quite frequently in our experiments. We trained the agent with a reward function stated in [10] and by looking at the seeds generated by the agent in the last episode, we realized that the action the agent has chosen was quite often the same action as earlier during an episode.



```
[[4.59551048 8.37708473 2.00185633 0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [4.18149853 8.1909914  2.0632658  0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [3.86722851 8.00360394 2.18687654 1.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [3.61458874 7.99989223 1.92892671 0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]
 [3.84839678 8.08495903 1.76351023 0.        ]]
```



```
[[4.59551048 8.37708473 2.00185633 0.        ]
 [4.27267265 8.22596359 1.94054723 1.        ]
 [3.95215845 8.03095436 2.18430686 0.        ]
 [4.16827059 8.22247696 1.76833665 0.        ]
 [4.25196362 8.12555313 2.00055099 0.        ]
 [3.9293406  8.08948135 1.77778447 0.        ]
 [3.78089237 8.00549889 1.83352005 0.        ]
 [3.9293406  8.08948135 1.77778447 0.        ]
 [4.25196362 8.12555313 2.00055099 0.        ]
 [3.61907172 7.94957352 2.03272891 1.        ]
 [3.9293406  8.08948135 1.77778447 0.        ]]
```

Figure 4.9: The first 3 columns are the xyz coordinates and the last column is the random bit of the action. Before experiment 43 (left) there are a lot of identical actions chosen for the last episode of one training. After initializing the weights in a random manner (right), the actions are varying more often.

To solve this issue we tried out multiple different approaches:

- Adapt the reward function such that we penalize actions which are chosen multiple times during an episode as discussed in 4.4.
- Insert chosen seed points into the state such that we have information on where the seed points were placed.
- Additionally to adapting the reward function increasing the learning rate or increasing the number of iterations. The agent should adapt faster to the penalized actions.
- As in [12] initialize the weights of the network randomly with a normal distribution.

As soon as the weights of the network were initialized on a normal distribution, the seed coordinates during an episode became more diverse. Although we have a lot more diversity from the agent, there are still recurrences of the same actions in the later phase of a training episode of 10 clicks. In 4.2 we increased the number of iterations, which did not completely erased this issue but we see a potential correlation between the number of iterations and the number of repeated actions.

4.6.4 Experiences Limit in Accuracy

All the experiences are randomly chosen actions such that we can use them later for training our agent. Training of the agent is based in all steps on a batch of 32 experiences in the experience buffer. It purely relies on the experiences in the buffer. Since during exploitation the agent determines the next action by forwarding the state to our deep Q network, the effect of the next action also depends on the experiences made so far. A cap in accuracy during pre-training can limit the effectiveness of the agent's predicted actions.

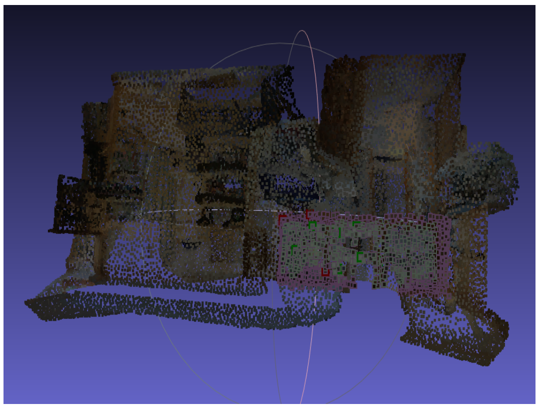


Figure 4.10: The big cubes in green and red show the predicted points of the our deep Q network. They were always placed inside the first half of all the voxels of the object.

In multiple runs we observed a cap in accuracy for this kind of pre-training. In our experiment we observed a cap in IoU accuracy of $\sim 30\%$ as shown in 4.11 on the left side. As explained in the passage before, we have to increase this cap such that the agent's prediction is improved.

We try to solve this issue with the following methods:

- We use R_{regions} instead of R_{diff} to provide an incentive for the agent to predict points more centrally. This variant only increases the accuracy during training and not during pre-training. This approach does not turn out to our satisfaction.
- Instead of choosing experiences in the pre-training buffer randomly, we determine the next action with a probability of $\delta = 0.5 = 50\%$ randomly and with a chance of $(1 - \delta) = 0.5 = 50\%$ with the heuristics in 2.2.2. With this method the decisions of our agent are approximating closer to the decisions of the heuristics.
- The final solution to this problem was to remove two errors in the code concerning the agent's function to convert the action space index to a xyz coordinate. The resulting xyz coordinate was always within the first half of all the action space as shown in 4.10. Additionally, at each step we added by mistake the click mask of the first seed to the segmentation mask. This caused a cap of IoU accuracy during pre-training and also during training.

After fixing the errors in the code, we also ran experiments with the modified pre-training phase, to increase the overall IoU of the experiments. The results of the training phase showed small improvements compared to the experiment without modified pre-training phase. This method also needs further investigation.

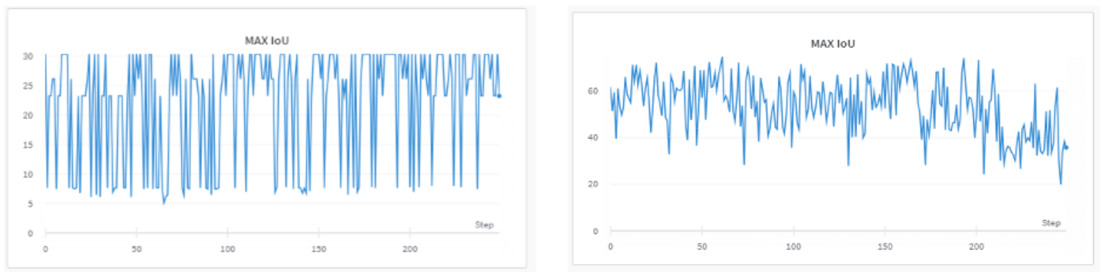


Figure 4.11: On the left side wee observe the cap in accuracy during the whole training process. By fixing a bug in the code, the IoU curve began to show expected behaviour.

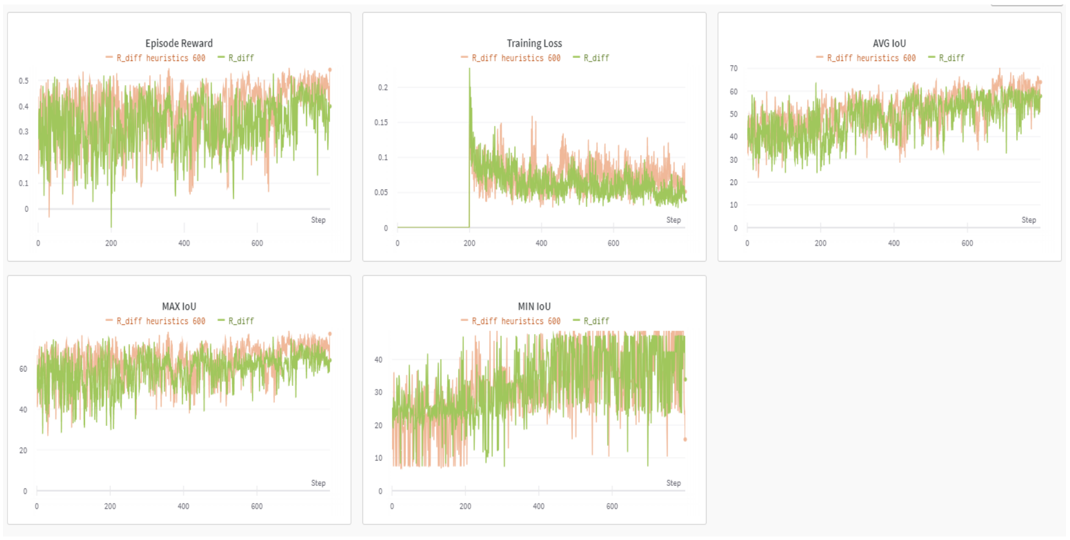


Figure 4.12: In green we see the training curves without modified pre-training phase in orange we see training curves with modified pre-training phase from 4.6.4

Future Changes Besides using our adapted pre-training phase which did show small improvements in accuracy, we can increase the effect of the collected experiences during pre-training with Prioritized Experience Replay (PER) from [14]. PER prioritizes experiences which are more significant instead of sampling them randomly. This might improve the accuracy of the agent.

Chapter 5

Conclusion

In our work we presented a technique to propose seed locations for interactive object segmentation in a 3D space. By using a deep Q network, which combines reinforcement learning and neural networks, we are able to generate different seed propositions. The network shows an efficient and stable learning pattern. Since it is built by using the Minkowski Engine[2], it synergizes well with the used segmentation model[8].
We evaluated different action spaces for our network to reach a higher accuracy. To make the predictions of the network non-repetitive, we adapted the reward function and randomized the initial network weights. By modifying the pre-training phase and testing a reward function, which differs between the regions of the action space, we attempted to improve the seed generations. By training our agent on a subset of the ScanNet data set, we observed that the average IoU increased from 22.8% to a value of 65% within 10 click simulations.
Simulating seed propositions for labeling is a significant task, they are needed to generate training data for computer vision tasks. It is therefore important to optimize this seed generation process. Using reinforcement learning for predicting the user's intention is a novel technique with a lot of potential in the future.

Bibliography

[1] Introduction to rl and deep q networks. https://www.tensorflow.org/agents/tutorials/0_intro_rl, 2023. [Online; accessed 27-July-2023].

[2] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks, 2019.

[3] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes, 2017.

[4] Ameet Deshpande. Deep double q-learning — why you should use it. https://medium.com/@ameetsd97/deep-double-q-learning-why-you-should-use-it-bedf660d5295, 2018. [Online; accessed 27-July-2023].

[5] Vijay Kanade. What is the markov decision process? definition, working, and examples. https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-markov-decision-process/, 2022. [Online; accessed 27-July-2023].

[6] Sergios Karagiannakos. Q-targets, double dqn and dueling dqn. https://theaisummer.com/Taking_Deep_Q_Networks_a_step_further/, 2018. [Online; accessed 27-July-2023].

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[8] Theodora Kontogianni, Ekin Celikkan, Siyu Tang, and Konrad Schindler. *Interactive Object Segmentation in 3D Point Clouds*. 2023.

[9] Grady L. Random walks for image segmentation, Nov 2006.

[10] Kyoung Mu Lee, Heesoo Myeong, and Gwangmo Song. Seednet: Automatic seed generation with deep reinforcement learning for robust interactive segmentation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1760–1768, 2018.

[11] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[12] Kavukcuoglu KṠilver Dėt alṀnih, V˙Human-level control through deep reinforcement learning. *Nature 518*, 2015.

[13] Sayak Paul. An introduction to q-learning: Reinforcement learning. https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/, 2019. [Online; accessed 27-July-2023].

[14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[15] Jinsheng Sun, Xiaojuan Ban, Bing Han, Xueyuan Yang, and Chao Yao. Interactive image segmentation based on feature-aware attention. *Symmetry*, 14(11), 2022.

[16] Wikipedia contributors. Markov decision process — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=1156832827, 2023. [Online; accessed 21-August-2023].

[17] Wikipedia contributors. Reinforcement learning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1170805603, 2023. [Online; accessed 21-August-2023].

[18] Chris Yoon. Double deep q networks. https://towardsdatascience.com/double-deep-q-networks-905dd8325412, 2019. [Online; accessed 21-August-2023].

[19] Chris Yoon. Dueling deep q networks. https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751, 2019. [Online; accessed 27-July-2023].

[20] Chris Yoon. Vanilla deep q networks. https://towardsdatascience.com/dqn-part-1-vanilla-deep-q-networks-6eb4a00febfb, 2019. [Online; accessed 27-July-2023].

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Where to Click: Exploring Reinforcement Learning for 3D Interactive Object Segmentation

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Tschechtelin	Cedric

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Breitenbach 30.08.23	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.