Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Spatially Equivariant Action Maps in Behavioral Cloning

Author:
Emir Sahin

Supervisor:
Dr. Edward Johns

Second Grader:
Prof. Andrew Davison

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing (Artificial Intelligence and Machine Learning) of Imperial College London

September 2022

Abstract

Researchers have recently studied how a robot can learn from images alone, without prior knowledge of objects within a scene. This study led to Robot Learning implementations where a robot's actions are predicted directly from an image through a neural network. However, across all these implementations, actions have been represented in several ways without a clear consensus on which representation is best to represent a robot's actions using a neural network.

In this paper, we consider a simple robot grasping problem where our goal is to learn a function from an image of a scene onto an action that will grasp a target object within that scene. We aim to leverage the properties of a grasping workspace to create an adaptable and understandable action representation. Specifically, we focus on the equivariant properties of the $SE(2)$ geometric group: a geometric group into which one can map a planar grasp problem.

We propose using Group-CNNs (G-CNN), a subset of Convolutional Neural Networks (CNN) that transforms convolutional kernels based on a Symmetric Group G, which we can use to exploit the rotational qualities of an $SE(2)$ space. However, we recognize through experimentation that the $SE(2)$-equivariant properties of a grasping problem can not be exploited by predicting end-effector velocities.

Due to this, we propose Spatially Equivariant Action Maps (EAM). This action representation maps end-effector positions within a workspace in a 2D array that is the same size as the input image. Then, the gripper moves to the proposed position using Inverse Kinematics. We recognize that the optimal grasp function using an EAM is $SE(2)$ equivariant and can be modeled using G-CNNs.

Structuring the grasping task as a Behavioral Cloning problem, we utilize a Fully Convolutional Network (FCN) built by G-CNNs to generate EAMs. Through experimentation, we show the high accuracy of this methodology even at low demonstration numbers and how the method exploits equivariance to adapt to unseen target positions thanks to the alignment between the input and the output of the FCN.

Acknowledgments

First, I would like to remember and thank my grandmother, Ayfer Şahin, whom we lost during the project. I love you and will never forget you.

I would like to thank Dr. Edward Johns for his guidance throughout this project and his willingness to answer my wild and incomprehensible questions every time.

I would like to thank my parents, Göksan and Esra Şahin, my grandmother Evin Tengiz, my grandfather Mesut Şahin, and my brother Engin Şahin for being there for me and supporting me no matter the situation.

I would like to thank Caroline Gilchrist, the Student Wellbeing Advisor at the Department of Computing for helping me through a few rough patches.

I would like to thank Francesco Bellardinelli, my Personal Tutor, who has supported me through my whole masters degree.

I would like to acknowledge Pietro Vitiello whom I've shared ideas throughout this project.

Finally, I would like to thank all of my friends, new and old, who supported me throughout this journey.

Contents

Chapter 1

Introduction

1.1 Motivation

Imagine a future where robots work alongside humans, assisting them on any task, trivial to complex: cleaning, driving, manufacturing. Such a future is not challenging to imagine. After all, robots have been prevalent in the industry since 1961, when GM first introduced them to the assembly line [1], and more recently, they have been increasing in popularity in households [2]. Given the increasing demand for robots in manufacturing and households, it is not hard to see how robotics technology has evolved quickly from solving simple, hard-coded tasks to tackling problems requiring adaptive decision-making.

However, as prominent as robots may be in the industry, they are far from being considered widely adopted in the household. Humans' tumultuous day-to-day activities require much adaptability: for robots to be alongside humans on a day-to-day basis, they need to be highly adaptable, communicate clearly, and tackle a range of simple to complex tasks.

At first sight, this seems trivial - after all, factory robots build vehicles, electronics, computer chips, and more with barely any human intervention [2]. A majority of the building process is streamlined and planned ahead of time. This results in highly repeatable and accurate tasks from factory robots [3]. However, factories are closed settings where external factors do not interfere heavily relative to the life inside of a household. Factory robots do not have to adapt to new settings constantly or solve novel tasks. They are intricately programmed and tested before they are even used on the factory floor [4].

1.1.1 Adaptability to New Actions

Adapting to and emulating a novel action is relatively trivial for a human: one can comprehend an action or a sequence of actions in a few demonstrations, then emulate it quickly, depending on how complex the action is. Consider the simple task of pouring water into a glass. It can be broken down into a few simple actions: grasp

the handle, raise the water container over the glass, then tilt the container until the glass is full. Through a single observation and without any additional information, a human should be able to understand the sequence of actions it has to take and emulate it perfectly afterward, knowing when to stop pouring.

Let us consider a robot arm attempting the same task. With several degrees of movement thanks to multiple joints, these arms can perform various actions ranging from grasping, pulling, and pushing using Robot Control [5]. The same pouring task can be performed flawlessly by deconstructing the task into a series of actions for each joint of the robot. However, this requires a high amount of fine-tuning and will only work for a specific glass, a specific jug, and a specific amount of water. A typical household is unlikely to have enough time to break down the water pouring task nor the expertise required to code this information into the robot arm. Even on just the task of grasping an object, robots perform very inaccurately in comparison to humans [6].

There have been approaches to make Robot Control more adaptable through methods such as Statistical Localisation and Mapping (SLAM)[7], Extended Kalman Filters (EKF) [8], and Model Predictive Control (MPC) [9]. These methods have proven successful in increasing the adaptability of robots in complex applications. However, robots still struggle with novel tasks, environments, discontinuities, or anything that may cause a large amount of uncertainty within a task [10].

Robot Learning, stemming from the combination of machine learning and robotics, is a sub-division of robotics that aims to tackle, among other things, the adaptability issues of traditional robotics. Robot Learning leverages artificial intelligence to allow robots to execute tasks without specific instructions autonomously. One can extend a robot's abilities to additional assignments without manually programming the actions required [11].

It is common to see Robot Learning algorithms based on images and visuals alone without preliminary data regarding the location of objects within a scene[12, 13, 14]. Visual Robot Learning can use a neural network trained to predict a robot's actions based solely on an image and utilize this network for control in reinforcement learning, imitation learning, and language-conditioned learning tasks [15, 16, 17]. However, research has shown that the performance of vision-based grasping systems is heavily influenced by the amount of variation and data available during training[18].

1.1.2 Action Representations

Traditional Robot Control is highly accurate but not adaptable, while Robot Learning is highly adaptable but relies on training data to perform accurately. This paper aims to combine these two fundamental methodologies by exploring an important factor of Visual Robot Learning: the representation of an action. If an action is what a robot must do to reach its next state, then an action representation is the information fed into a controller to get that action. It can be an image representing a new state,

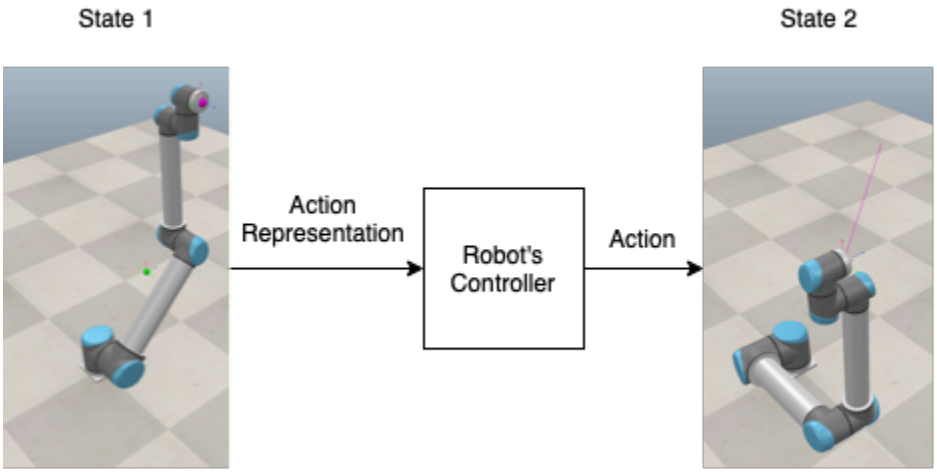

Figure 1.1: Action Representation.

positional arguments for the end-effector, velocities, torques, and more. Fig. 1.1 provides a diagram for visual explanation.

An important concept to consider regarding adaptability is equivariance. While it is further discussed under Sec. 4, equivariance allows the output to transform the same way the input does. So, if a Robot Learning algorithm is translation-equivariant, it does not need to learn each specific translation within the input to generalize the output: equivariance will automatically translate the output.

A majority of robot grasping tasks are in translationally and rotationally equivariant spaces. So, we can increase accuracy and decrease sample rates by leveraging these equivariant properties [19, 20, 21]. However, as the input and output space must align, not all action representations can leverage these equivariant properties.

One last consideration is the ease of understanding regarding an action representation. In a future where robots work alongside humans, we would not expect people to have intricate knowledge of robotics. However, communication of actions and the decision process behind a robot is important for wider adoption. A non-expert is unlikely to understand how joint velocities will move an end-effector. However, comprehending a visual positional argument of where the robot wants the end-effector to go is very simple. Due to this, an action representation should be easy to understand.

The paper proposes Spatially Equivariant Action Maps (EAM), a methodology that aims to provide understandable action representations that combine Robot Learning and Robot Control. EAM maps positional arguments for an end-effector into a matrix aligned with the visual input. The map indicates a position an end-effector must reach using Robot Control and leverages the equivariant properties of Convolutional Neural Networks of a robot grasping task to decrease the number of representations needed to learn novel positions and rotations of a target.

1.2 Objectives

This research aims to introduce a novel action representation for Visual Robot Learning and experiment with this action representation in a simple robot grasping environment to determine its strengths and weaknesses. More specifically, our goals are summarized below:

- Combine the accuracy of Robot Control and adaptability of Robot Learning.
- Create an action representation that is understandable to a larger majority of people rather than just robotics experts.
- Build a grasping simulation environment to test the accuracy, robustness, and application of the proposed action representation.
- Understand the equivariant properties of a robot grasping task and how they can be utilized in an action representation to improve the adaptability of models.

The focus on equivariance is due to its ability to make the network more sample efficient and adaptive. By allowing the output of a network to transform with the input, it allows the generalization of data into novel inputs. The aim is to increase the adaptability of networks by leveraging equivariant properties. We further discuss equivariance in Sec. 4 and Sec. 7.

1.3 Contributions

The main contribution of this paper is the Spatially Equivariant Action Maps (EAM, Sec. 7) and the methodology of utilizing them in a Behavioral Cloning (Sec. 2.2.1) setting. This method predicts gripper positions by mapping the robot workspace into a 2D matrix. Furthermore, it successfully leverages a robot grasping space's translational and rotational equivariance properties to predict gripper positions in unseen states accurately.

In this paper, we provide the mathematical ground for leveraging equivariant properties of robot grasping spaces and how EAMs can do so. Further, we provide a methodology for setting up environments and data generation techniques that will work with EAMs. In addition, we provide network structures that leverage rotational equivariant and can create EAM representations, as well as experiments that prove the equivariant properties of EAMs.

We can briefly list the contributions of this paper below:

- Proposing Spatially Equivariant Neural Networks as a new action representation for visual robot learning. It maps (x,y) in the workspace into pixels of a 2D matrix.
- Providing the methdology to generate Expert Demonstrations to train a network to generate EAMs.

- Suggest and test various Fully Convolutional Network architectures that work well with EAMs to generate translationally and rotationally equivariant action representations.

1.4 Structure of the Report

At first, the paper provides background knowledge in reinforcement learning, imitation learning, and deep learning, as these concepts will be used throughout the paper. Afterward, related work to robot action representations will be provided to give an insight into the works that inspired this one. Then, we will provide a mathematical basis for equivariance and group theory, which EAMs leverage as an action representation. Then, we will introduce an end-effector velocity model with G-CNNs that fail to use equivariance to generalize grasping into unseen target positions. Afterward, we will introduce the concept of Spatial Action Maps and provide the mathematical background to prove their translational and rotational equivariant properties. Finally, we introduce EAM, a representation that leverages Spatial Action Maps and G-CNNs, and test its equivariant properties with various experiments.

Chapter 2

Background

This chapter will inform on and elaborate on background knowledge that is crucial in understanding the domain of the paper. It will start by explaining reinforcement learning and then introducing imitation learning. After, the paper will explain several relevant deep learning concepts. Finally, it will briefly describe Inverse Kinematics as it's the process of calculating robot parameters to place a robot's gripper to a specified position.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a set of experience-based mathematical frameworks that allow agents to learn tasks by minimizing cost functions [22]. An agent is placed in an environment that is free to explore by deciding on actions at discrete time intervals based on the current state of the environment. As the agent performs various actions and changes states based on a probability distribution, it incurs rewards that it tries to maximize over the long run.

RL utilizes Markov Decision Processes can be described with Markov Decision Processes (MDP). An essential aspect of MDPs is the Markov Property which indicates that a state only depends on the previous state and the action taken. So, a state transition will be independent of all actions taken and states visited other than the current step. This property is defined as:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_t, ..., S_1] \tag{2.1}$$

As the Markov Property signifies that a state contains all the necessary information to determine the following action, it allows one to create policies that map states to actions. Considering this information, we can formally define an MDP with a tuple M = $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, p_0, \gamma)$:

- $\mathcal{S}$ is a set of states where $s, s' \in \mathcal{S}$ are the current and next state respectively;
- $\mathcal{A}$ is a set of actions available where $a \in \mathcal{A}$ is the action taken at state s;

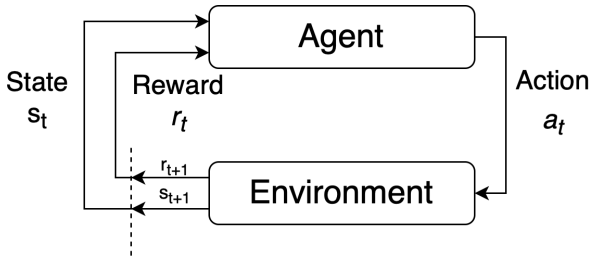

Figure 2.1: Agent and environment interactions in RL diagram. An agent observes a state s_t and chooses action a_t at each time step t, resulting in state s_{t+1} and reward r_t.[22]

- $\mathcal{R}$ is a set of rewards where $r \in \mathcal{R}$ is the reward for using action a in state s to reach state s';
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is a set of probabilities $p(s'|s, a)$ where action a in state s will lead to state s';
- p_0 is is the initial state distribution $\mathcal{S} \rightarrow [0, 1]$,
- γ is the discount value that deteriorates future rewards.

In a typical scenario, the aim is to determine the best policy π: $\mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ that maximizes the overall reward, which we can determine by calculating the expected rewards over the trajectories given in a policy:

$$R = \mathbb{E}_\pi[\mathcal{R}(s, a)] = \mathbb{E}_{s_t, a_t \sim \pi}[\sum_{t=0}^{T} \gamma^t \mathcal{R}(s_t, a_t)], \tag{2.2}$$

where T is the time horizon, $s_0 \sim p_0, a_t \sim \pi(\cdot|s_t)$, and $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$. γ assures even though T may be infinite, the return will be bounded. Hence, the objective of an agent is to maximize the expected reward from a state s: $\max_\pi \ \mathbb{E}_\pi[\mathcal{R}(s, a)]$.

In RL, to determine the optimal policy, we utilize functions that calculate the value of a state or state-action pair in the long run; hence these functions are called Value functions. RL algorithms use two main value functions during training: the state-value function and the action-value function.

The state-value function maps a state's expected return based on policy π and the higher the value, the better the state at time t (2.3).

$$V_\pi(s_t) = \mathbb{E}[R_t(s_t)] \tag{2.3}$$

During each stage of training, the value function is used for Policy Evaluation and Policy Improvement. Policy Evaluation picks the value function for a policy π, and Policy Improvement uses this value function to adjust the action of each state to transition into a higher value state. This process is repeated until the optimal value function is determined:

$$V^*(s_t) = \max_\pi V_\pi(s_t) \tag{2.4}$$

In most cases, it is hard to determine the agent will transition to after taking an action due to unknown environment dynamics resulting in the use of action-value functions. The action-value function maps the expected return of each action from a given state based on policy π (2.5) to find the optimal policy π^* (2.6):

$$Q_\pi(s_t) = \mathbb{E}[R_t|s_t, a_t|] \tag{2.5}$$

$$\pi_* = arg\max_\pi Q_\pi(s_t, a_t) \tag{2.6}$$

There are two categories of RL algorithms depending on the agent's knowledge of the environment: model-based and model-free RL.

2.2 Imitation Learning

As explained previously, in the RL framework, an agent aims to learn the optimal policy π for an environment by maximizing the rewards it gains through interacting with that environment. A pre-defined and catered reward function determines the rewards the agent gains from the environment. However, creating complex reward functions for complex tasks may be extremely difficult, and learning from these complex reward functions may take a long time through trial and error. This is where the Imitation Learning (IL) framework comes in: rather than using a cost function to train an agent, we provide the agent with state-action trajectories (called demonstrations from here on) of an expert: $\tau = (s_0, a_0), (s_1, a_1), ..., (s_N, a_N)$. In this framework, the agents aim to learn a policy π that results in similar behaviour to a set of expert demonstrations $D = \tau_1, \tau_2, ...$ So, we can formally define the IL framework with an MDP without a reward function. Two popular methods of implementing the IL framework are Behavioral Cloning (BC)[23] and Inverse Reinforcement Learning (IRL).

2.2.1 Behavioral Cloning

In BC, the agent aims to emulate the expert's behavior (policy π_θ) through supervised learning by regressing over the state-action pairs from the expert trajectories τ. Specifically, the agent uses the set of expert demonstrations D with a supervised learning algorithm to learn the optimal parameters θ^* to emulate the expert's policy π_e (2.7. The regressor used to determine π may be based on MLE (Maximum Likelihood Estimation) or Deep Neural Networks (explained in Sec. 2.3) that this paper will utilize.

$$\pi^* = arg\min_\pi \mathbb{E}_{s\ D}[||\pi_e - \pi_\theta||] \tag{2.7}$$

$$\theta^* = arg\min_\theta \sum_{i=\tau} ||(s_e, a_e) - (s_i, a_i)|| \tag{2.8}$$

BC aims to imitate the expert's behavior (state-action pairs) without directly interacting with the environment. However, if the state-distribution of the expert policy π_e

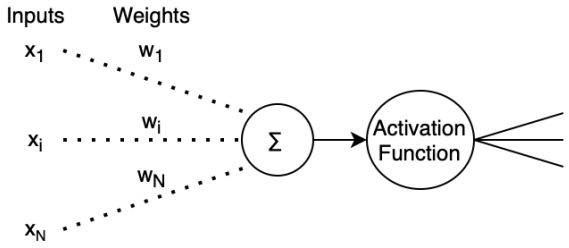

Figure 2.2: Diagram of a neuron.

and the optimal policy for the agent π^* are different, BC will lead to a compounding error problem called covariate shift. This occurs because during training time, the agent is only trained on state-action pairs from π_e but during testing, π^* is executed, which may lead the agent to states not observed in π_e. Simplest way to tackle this problem is to gather a large number of different demonstrations from π_e to have data on more states. However, there are smarter approaches to tackle this problem such as DAgger [24]. DAgger implements an iterative learning policy which provides the agent with expert demonstrations based on its own policy π. However, this requires an interactive expert which will provide demonstrations based on the path the agent takes during training.

Do note that this paper will build on the BC framework to tackle the robot gripping problem by providing expert demonstrations as the aim is to determine a action representation rather than a novel Imitation Learning approach.

2.3 Deep Learning

2.3.1 Artificial Neural Networks

Artificial Neural Networks (ANN/NN) are models inspired by the neuron activation in the brain[25]. Specifically, they are models that map an input vector x to a target value y by processing x through multiple connected hidden layers of neurons.

A neuron is the basic building block of a NN: it receives several inputs from previous neurons, computes the weighted sum, and passes the weighted sum through a non-linear function (activation function) to produce a single output (Fig 2.2). Neurons communicate by weighing connections that feed the output of a neuron in one hidden layer to the input of a neuron in another hidden layer. A NN trains by optimizing the weight of a connection between neuron i and j ($w_{i,j}$), which is multiplied by the input fed into neuron j from neuron i. The fundamental NN model is the Multilayer Perceptron (MLP). MLPs are generated by stacking several layers of neurons, and connecting layers with the previous and the next layer of neurons - neurons on the same layer are not connected. The ith layer of an MLP consists of n_i neurons, called the layer's width. Each of the n_i neurons connects to the neurons of layer j with weights $W_{i,j}$. The computation of the ith layer is in Eq. 2.9 and the structure of an

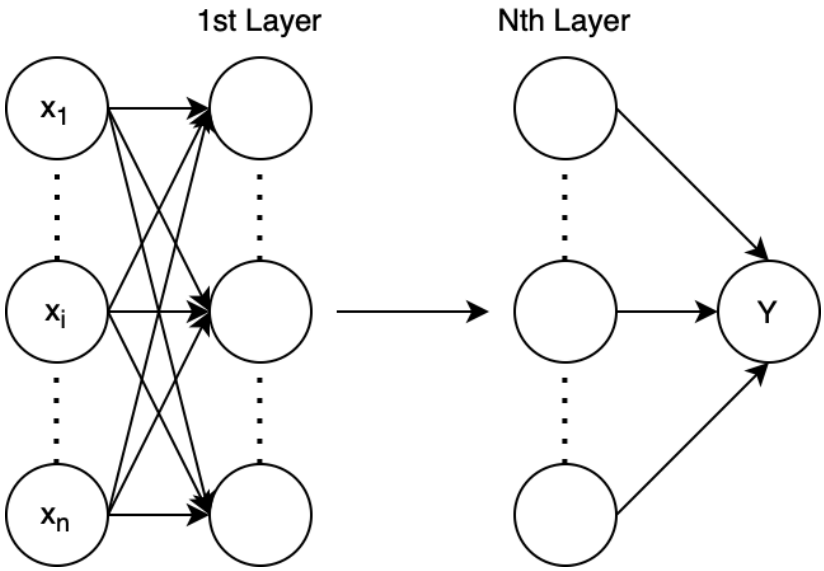

Figure 2.3: Diagram of an MLP with n inputs, N layers, and a single output.

MLP is in Fig. 2.3.

$$y_i = \sum_{i=0}^{n_i-1} w_{i-1,i} * x_i + b_j \tag{2.9}$$

A NN is trained by tweaking the weights and biases found in the neurons and connections to increase the likelihood of generating the output y_{train} from the input x_{train} while generalizing on an independent test set of (x_{test},y_{test}) values. This is done through gradient descent with various Loss Functions ($\mathcal{L}()$). Loss functions differ based on the NN's input and output, but traditional loss functions include Mean Squared Error(MSE) for regression (Eq. 2.10 and Cross-Entropy(CE) for classification tasks (Eq. 2.11).

$$\mathcal{L}_{MSE} = 1/N * \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{2.10}$$

$$\mathcal{L}_{CE} = \sum_{i=1}^{N} (\hat{P}(i) \log P(i)) \tag{2.11}$$

To train a NN, we must first calculate the gradient of the loss after a forward pass (calculating an output with a set of inputs given the current parameters). Then, we back-propagate these gradients to the weights of the NN. Then, gradient descent can be performed to tweak the network to reduce the Loss value in the next forward pass.

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are NNs aimed at spatial pattern recognition using the convolution operation. CNNs are hierarchical and multi-layered networks

that capture the Spatial and Temporal dependencies in the input by applying relevant filters. This is done by exploiting the spatial locality of inputs by sharing weights across space dimensions - weight sharing also reduces the number of parameters required in comparison to a fully connected network.

Two primary components of a CNN are the convolutional layer and the pooling layer. These layers perform a similar operation where a 3D array input (called feature maps) is fed into the layer to output another 3D array.

As the name implies, a convolutional layer utilizes convolutions as a feature extraction mechanism to detect features in the layer's input. A layer consists of multiple feature maps (channels); a feature map is a set of neurons where each neuron is connected to a different area of the layer's input. Rather than using weights for each neuron, a convolutional layer utilizes trainable kernels/filters associated with each layer.

An example of a convolution between a feature map and kernel is in Fig. 2.4 where the kernel window of weights slides over the input feature map. The convolutional layer determines patterns between the input feature maps and the kernels associated with that layer, resulting in the output feature maps. Eq. 2.13 describes this operation on F_j^{in} input and F_i^{out} output feature maps. The $k_{i,j}$ represents the kernel associated with the ith output jth input, b_i represents the bias vector, and $\star$ is a 2D convolution seen in Fig. 2.4. Eq. 2.12 is the definition of a convolution in regards to a feature map F and a kernel k where C stands for the number of channels.

$$(F \star k) = \sum_{y} \sum_{c=1}^{C} f_c(y) k_c(y - x) \tag{2.12}$$

$$F_i^{out} = \sum_{j=1}^{N_{in}} F_j^{in} \star k_{i,j} + b_i \tag{2.13}$$

A pooling layer is similar to a convolution: a smaller kernel slides over a feature map, replacing the neighborhood with a summary statistic such as maximum or average. Typically, this is applied on non-overlapping windows on the feature map, unlike convolutions. The main aim of a pooling layer is to introduce invariance into the spatial translation of features and reduce the size of the inputs.

CNNs are very popular in image processing tasks, hence they are widely adopted. LeNet [26] is the first implemented CNN model while AlexNet [27] popularized the use of CNNs in both research and industry.

Fully Convolutional Networks (FCNs) such as UNet [28] are a subcategory of NNs and CNNs that only perform convolutional operations (including downsampling and upsampling. So, unlike most CNN frameworks such as AlexNet [27], FCNs lack fully connected layers. FCNs are primarily used for image segmentation [28] [29] as the output of an FCN tends to match the size of the input matrix.

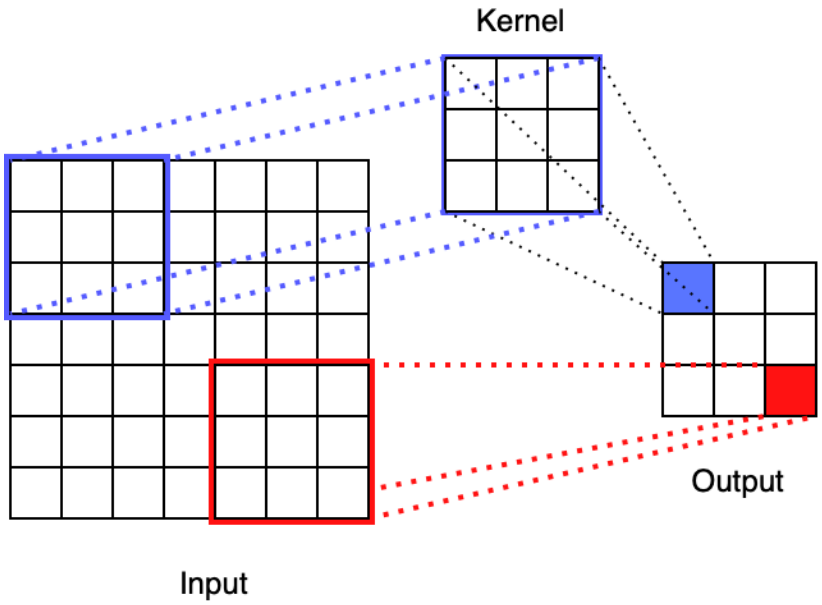

Figure 2.4: Diagram of a convolution being applied on a feature map of size 6x6 with a kernel of 3x3 with a stride value of two.

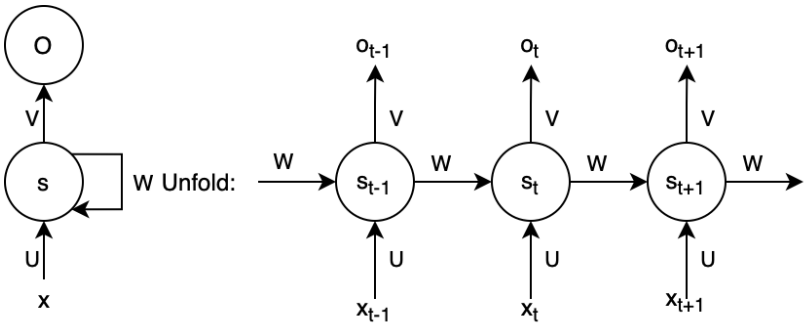

Figure 2.5: Diagram of an RNN under folded and unfolded views. Adapted from [30].

Typical FCN architecture tends to include a downsampling encoder and an upsampling decoder. The downsampling path extracts and interprets the context, and the upsampling path allows for localization. In the case of UNet [28], each step of the downsampling path feeds its output back into the matching step of the upsampling path.

2.3.3 RNNs

NNs and CNNs assume that all inputs are independent of one another. However, this is not the case for many tasks in the real world. Hence, Recurrent Neural Networks (RNNs) are NN aimed at processing sequential datasets [30] by sharing parameters across time. The recurrence of an RNN allows it to perform the same task for every element in a sequence, making the output dependent on previous inputs and calculations.

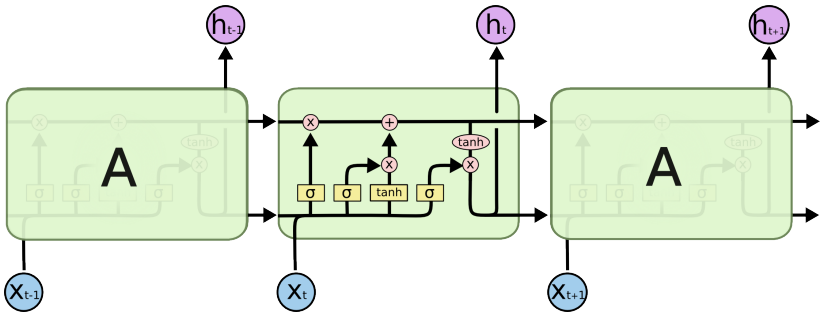

Figure 2.6: Diagram of three LSTM layers. Taken from [32]

Figure 2.5 demonstrates the folded and unfolded views of an RNN network where x is a sequence of values that is the input, U is the input-to-hidden matrix, W is the hidden-to-hidden matrix, and V is the hidden-to-output matrix. Each matrix U, W, and V are the weight of neurons associated with the network. To update neuron weights, RNNs use a technique called Back Propagation Through Time (BPTT), which is, essentially, the application of backpropagation across time steps of a sequence. However, if the sequences are very long, the gradients may vanish or explode in older nodes, based on the weights of these nodes. To tackle the gradient vanishing problem, a type of RNN called Long Short-Term Memory [31] were invented. This paper utilizes LSTMs to deal with sequential data rather than a simple RNN.

LSTMs implement the usage of gated cells to tackle the gradient vanishing problem caused by BPTT in regular RNNs. A gated cells make decisions on what data is allowed to be written to, read from, and stored on itself by switching between closed and opened states. These gates allow LSTM to keep details about older parts of a sequence by providing memory to maintain a constant error.

The diagram for an LSTM can be viewed on Fig. 2.6. A sequence vector is carried from the output of a layer to the input of the next layer. Small circles within a layer indicates vector addition or multiplication while yellow boxes are learned NN layers. Merging lines denote concatenation and forking lines denote duplication.

2.4 Robot Control and Kinematics

For this paper, a robot refers to a robotic arm. A robotic arm is a combination of links and joints. Links are semi-rigid bodies or a combination of semi-rigid bodies that have a relative motion relative to the other parts of the machine. On the other hand, a joint is a connection between two or more links, which allows motion between connected links. Each joint provides a robot with a degree of freedom (DoF); DoF refers to how a robot can move. The number of DoF equals the number of independent displacements possible within a robot. Fig. 2.7 provides a diagram of links and joints within a robot arm. An *end-effector (EE)* is the gripper or component found at the end of a robot arm. It is the point where a robot interacts with its environment.

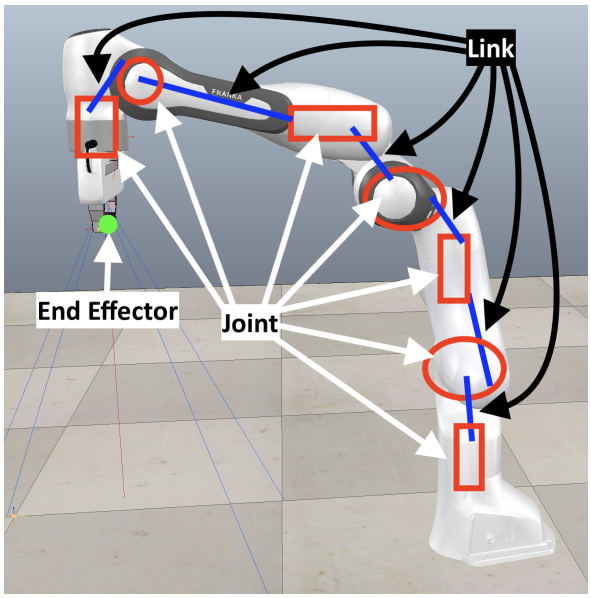

Figure 2.7: Diagram of a Panda robot arm within a simulation environment. Red Circles and Squares indicate joints while Blue lines indicate links. The green-dot at the end of the robot indicates the end-effector.

Kinematics is a branch of mechanics that concerns the motions of a system of bodies; it does not take into account the forces involved in actuating these motions and, instead, focuses on the position, velocity, and acceleration of a system of bodies with respect to time.

In Robotics, the EE position is defined in Cartesian space. However, as an arm combines joints and links, one must map the end-effector position to all the joints. *Inverse Kinematics (IK)* solves kinematic equations of a system of bodies to move the EE to a specified position[33]. Specifically, IK maps cartesian space into joint space $J(\theta_i)$ where i is a joint within a robot arm. So, it calculates the joint positions and rotations for a given end-effector position.

Chapter 3

Related Work

This chapter will describe various work in the RL and IL domains that utilize various action representation methods that inspired the final implementation in this paper. While each work described is different, the sections cluster similar work together, leading up to Spatial Action Representations.

3.1 Direct Action Representations

"Direct Action Representations" (DAR) are representations that directly correlate to an action the robot can perform. These include but are not limited to joint velocities, end-effector velocities, and joint torques.

DARs are common in projects based on BC. For example, when James et al. [13] explore robust methods to apply simulation-based Robot Learning models to the real world, they utilize a vision-based Robot Learning algorithm that uses a CNN trained to map images into actions. They feed the algorithm 3rd-person images of the environment and the robot to output joint velocities for actions and target position, end-effector positions as auxiliary outputs. While the method works successfully, joint velocities may differ from arm to arm and are hard for humans to grasp without context. The paper also reveals the importance of auxiliary outputs.

Zhang et al. [34] demonstrate the use of Virtual Reality Telemetry teleportation in imitation learning. In their setup, a human uses VR hand controllers to move a robot arm from the 3rd-person perspective view of the end-effector. The paper proposes a neural network architecture that takes RGB images and depth information into action while including auxiliary predictions to accelerate learning, much like James et al. The input to the network also includes the five most recent steps of the end effector, implying the importance of previous time steps. On top of temporal inputs, the model uses both RGB and Depth images. Zhang et al. use end-effector velocities for the actions even though the input is not from the perspective of the end-effector. This mismatch is solved by including the end-effector's position in the input since predicting joint velocities from various 3rd-person angles may prove troublesome

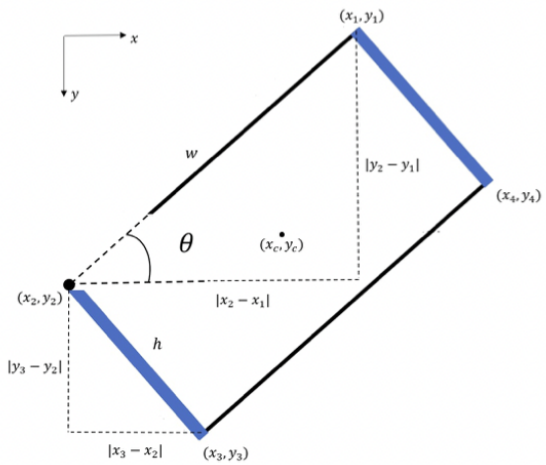

Figure 4: Rectangle encoding using its vertices coordinates

Figure 3.1: Rectangle encoding using its vertices coordinates [35]

for a single model. Finally, we will see the addition of L1 loss on top of L2 loss that Zhang et al. apply in other papers.

Both James et al. [13] and Zhang et al. [34] show the importance of auxiliary data while training Robot Learning networks. However, their approaches are significantly different as James et al. make the network predict the positions of the end-effector and the target while Zhang utilizes predictions.

Ribeiro et al. [35] introduce an end-effector mounted camera solution with two critical aspects. One is similar to James et al.'s solution of feeding images into a CNN to determine end-effector velocities - the results demonstrate the high accuracy of utilizing end-effector velocities for end-effector mounted view-points. The other is that Ribeiro et al. use a CNN to output the gripper's final positional and rotational values, using desired and current images of the end-effector. Fig. 3.1 further defines these output values based on the Cornell Grasping Dataset. Furthermore, the paper separates the end-effector's translational and rotational velocities of some models. However, the focal takeaways should be the importance of features and positional estimations in visual Robot Learning.

3.2 Feature-Focused Action Representation

This section will further explore action representation methods focusing on features in a visual image - both hand-crafted and learned. There are methods like the one proposed by Finn et al. [36] where an automated state-space construction approach is utilized to learn state representations directly from camera images. Their method initially trains a controller based on joint-angles and end-effector positions, outputting motor torques. The method determines states using a guided search policy based on a linear Gaussian controller; this initial controller collects images of the environment to extract features using a deep spatial autoencoder. Then, using what

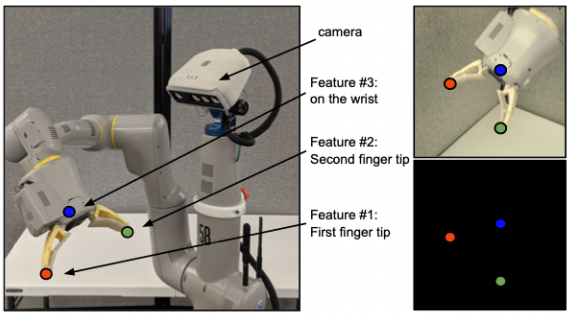

Figure 3.2: Action Image features taken from [37].

Finn et al. [36] describe as a 'predictiveness' heuristic and Kalman filters, the method prunes feature points to finally train a vision-based controller based on images and extracted features. Finn et al. use joint torques as actions but focuses on an 'action representation' based on unsupervised extracted features, which augment the final model significantly.

Another use of unsupervised feature extraction is proposed by Devin et al. [14], though the focus is on generalizable learning. The method extracts two types of features: task-independent and task-dependent features. An RPN extracts task-independent features to determine task-relevant objects and their positions as Devin et al. assume all objects relevant to a task will share similar semantic features. After picking a task-relevant object, a separate network only uses the semantic features of the task-relevant object to train on several trajectories. While the paper's main objective is unrelated, using extracted features from one network to train another is similar to Finn et al.'s [36] implementation as well. One is based on previous trajectories to improve task accuracy, while one focuses on utilizing the features for task-specific objects. However, not all methods use unsupervised features, such as those proposed by Khansari et al. [37] and Borja-Diaz et al. [38] utilize hand-crafted features to train robot controllers. Khansari et al. [37] propose a novel grasp proposal representation called "Action Image." Action Image aims to apply to all standard sensors, domain invariant, generalizable, and translation-invariant. Rather than proposing motor torques or velocities, the model proposes final gripper positions (similar to [35] and papers under 3.3). To do so, Khansari et al. propose utilizing three feature points to train on: the end of either fingertip of the gripper and the point where the fingertips connect to the robot hand (Fig. 3.2.) These features represent the final gripper positions of a proposed grasp candidate that an actor-critic network to improve grasp proposals. While Khansari et al. [37] do not directly use the features to generate a grasp proposal, the features are still crucial in improving the grasp proposal network. The Action Image method is directly related to a 3rd-person view as we can not directly use the suggested kinematics transformations in an end-effector setting. However, the paper does demonstrate the importance of features associated with grasp proposals.

As mentioned previously, Borja-Diaz et al. [38] use hand-crafted features. Nevertheless, their approach is significantly different from the grasp proposal approach

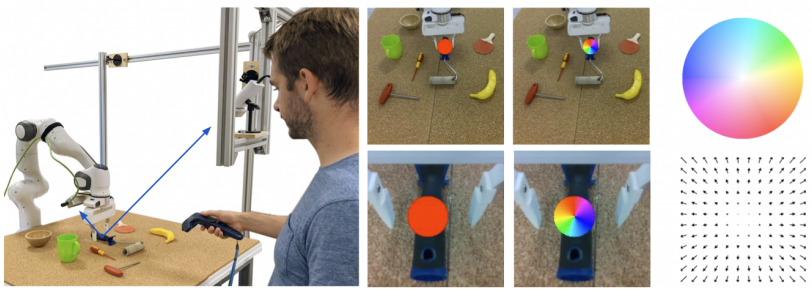

Figure 3.3: Affordance Regions representation taken from [38]. The colored circle indicates how the gripper should move to reach an affordance region's center.

utilized by Khansari et al. [37]. Borja-Diaz et al.'s methodology is based on two separate networks: one for approaching an object and one for interacting with the object. While a crucial part of the paper is the intuition behind how unlabeled data is used based on gripper movements, the focus for action representations should be on how these two networks generate action representations. The first network learns 'affordances' [39] to determine interaction regions within a scene that are used to estimate 2D pixel coordinates through a vector from each affordance pixel to the center of the region. The robot is guided to affordance regions using an action-planner where it switches to the second model based on reinforcement learning to grasp and interact with objects based on a human's grasping and dropping pattern. The main feature is the regions generated for interaction (Fig. 3.3 and how they relate to spatial positions of the gripper (Sec. 3.3). The utilization of two separate networks for reaching and interaction is similar to Devin et al.'s [14] approach.

Oh et al. [40] propose a method called View-Action for First-Person vision, though it is not under the domain of Robot Learning. View-Action utilizes three networks: an LSTM policy network, a Forward Model, and an Inverse Model. The method trains the forward and inverse model together, so the forward model accurately predicts the next state based on a state-action pair, and the inverse model predicts actions given two consecutive states. As the action is fed into the forward model using deconvolutional layers, the forward model can still be used to extract features from a given state, which is utilized in training the LSTM policy model. It should be noted that Oh et al.'s [40] action space is made out of simple positional movements, likely making this method easier to implement, though utilizing deconvolutional layers and forward-inverse network pairs to generate features can prove useful in Robot Learning as well. Note that this approach may be similar to utilizing a GAN to generate features.

3.3 Spatial Action Representations

Previous papers such as [37, 38, 35] use spatial aspects for action representations. The papers discussed in this section will relate specifically to spatial action predictions rather than direct action predictions. Pardo et al. [41] utilize a concept called Q-Frames. This methodology input frames from a game and outputs 2D Q-frames

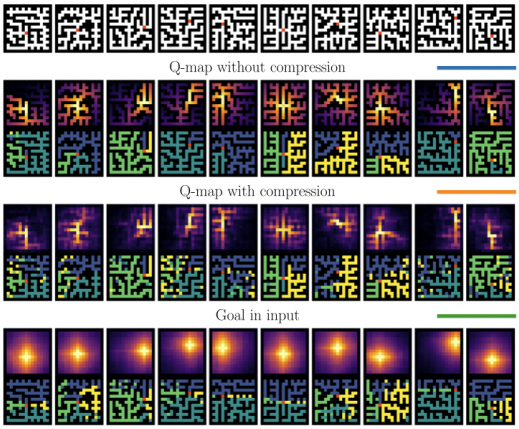

Figure 3.4: Implementation of Q-frames taken from [41].

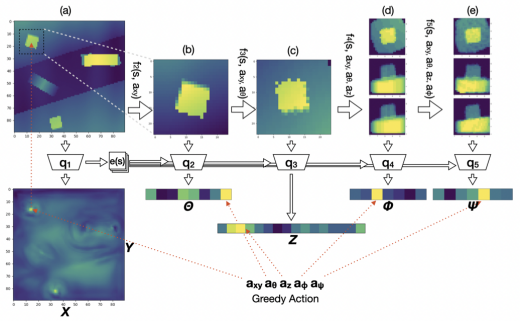

Figure 3.5: Implementation of ASR taken from [43].

where rows and columns represent goal locations in a given frame, and the number of output frames indicates the number of actions required. Note that the Q-frames consider an 'all-goals' approach where goals are defined as possible end locations that compare the number of actions required to reach the state in the next training frame (Fig. 3.4. Directly applying this to a gripping problem may prove hard as the number of achievable goals that can be generated is much larger than in a 2D game. The first application of a spatial action map, similar to Q-frames [41], in robotics, comes from Wu et al. [42]. The method is in a 2D action space for mobile robots where spatial action maps are generated based on the input image of the current state to represent possible actions in a pixel map. In this approach, each pixel represents a navigational end-point in the actual scene.

Wang et al. [43] use a similar method to [42] for the particular euclidian group in three dimensions (SE(3)). However, the spatial action map utilization in the paper is only for SE(2) spaces where each pixel value corresponds to a single (x,y) position in the environment. Wang et al. propose an Augmented State Representation (ASR) to tackle rotations. In ASR, there are two separate networks: one for translational outputs and one for rotational outputs. The translational network utilizes the spatial action map to determine the appropriate pixel location, which is then cropped from the input image and fed into the rotational network that proposes a rotational component for the final gripping location. Wang et al. [43] use more networks to tackle various components of the SE(3) action space which can be seen in Fig. 3.5. Zhu et al. [44] utilizes equivariant properties of SE(2) action spaces [45] to apply the ASR representation proposed by Wang et al. [43]. Utilizing equivariant CNNs [46] to train a network on generating spatial action maps [42] and cropped rotation identifiers [43]. The work demonstrates the efficiency improvements caused by implementing equivariant CNNs in spatial action spaces utilizing the ASR representations.

Chapter 4

Group-Equivariant CNNs

The EAM (Sec. 7) representation presented in this paper aim to leverage the translational and rotational equivariance of the 2-dimensional Euclidian Space present in a robot grasping task. This chapter aims to provide a basic understanding of Group Theory, Equivariance, and Equivariant CNNs to create a baseline understanding of the relevant concepts. Majority of the theory and concepts explained this chapter are based on [30, 19, 47, 20].

4.1 Group Theory and Symmetry

Understanding Symmetry and Symmetric Groups is essential in utilizing the equivariant rotational properties of two-dimensional euclidian spaces (shall be referred to as E(2) from this point on.) The paper will introduce the basics of Group Theory to transition into Symmetric Groups relevant to the Robot Grasping task.

4.1.1 Group Theory

A group can be formally defined as a set of elements G. G has a binary operator that is the called group operator $\cdot$ which satisfies the following axioms:

- An identity element $e \in G$ exists where $e \cdot i = i \cdot e = i$ for any $i \in G$. (Identity)

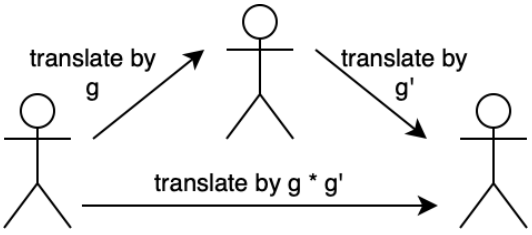

Figure 4.1: Diagram of Group $\mathbb{R}^2$.

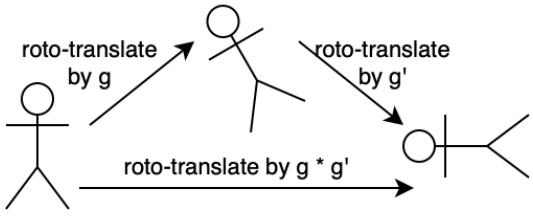

Figure 4.2: Diagram of Group SE(2).

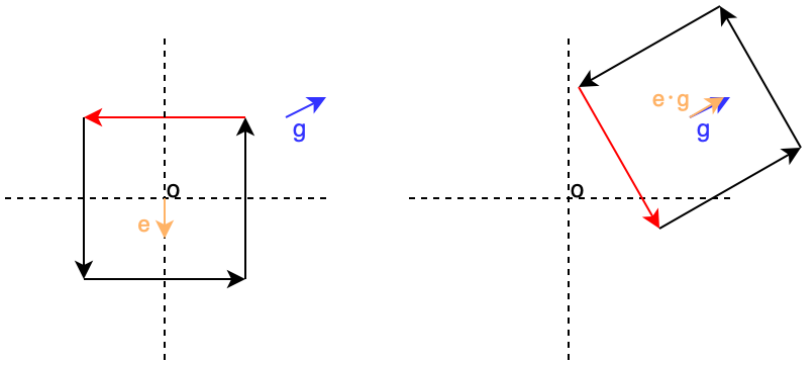

Figure 4.3: A set of vectors with identity vector e represented in SE(2) being shifted to point g in SE(2).

- The group product $\cdot$ is associative for $i, j, k \in G$: $i \cdot (j \cdot k) = (i \cdot j) \cdot k$. (Associativity).
- For any $i, j \in G$, the group product $i \cdot j = ij \in G$. (Closure)
- For any $i \in G$, there is an inverse transformation of i: $i^{-1} \in G$. $i \cdot i^{-1} = i^{-1} \cdot i = e$. (Inverse)

A simple group to consider is the Translation Group $\mathbb{R}^2$ where the group operation is $+$. This group consists of all possible translations in the vector space $\mathbb{R}^2$ (Fig. 4.1). Given the group operator is $+$, then the group operation and inverse can be defined by $g \cdot g' = (x + x')$ and $g^{-1} = (-x)$ respectively.

Building upon $\mathbb{R}^2$, we can introduce the group most relevant to a robot gripping task: the 2D Special Euclidian motion group ($SE(2)$, Fig. 4.2). $SE(2)$ couples $\mathbb{R}^2$ with the orthogonal group (symmetry group of the circle Sec. 4.1.2) $SO(2)$. As such, we can formally define $SE(2)$ with the following group operator and inverse:

$$\begin{aligned} g \cdot g' = (x, R_\theta) \cdot (x', R_{\theta'}) &= (R_\theta x \ + x, R_{\theta+\theta'}) \\ g^{-1} &= (-R_\theta^{-1} x, R_\theta^{-1}) \end{aligned} \tag{4.1}$$

where $g = (x, R_\theta)$.

Another important aspect of Group Theory are representations. Define ρ as a group representation that serves as a group homomorphism between G and general linear group $GL(V)$: $\rho : G \to GL(V)$.

$$\rho(g') \circ \rho(g)[v] = \rho(g' \cdot g)[v] \tag{4.2}$$

where v is some vector that is translated by $\rho(g)$ parameterized with $g \in G$.

Using this notion of representations, we can define Left-Regular representations $\mathcal{L}_g$ that transforms functions f applying the inverse group action to f's domain (Eq. 4.3). An example application of $\mathcal{L}_g$ is represented in 4.4.

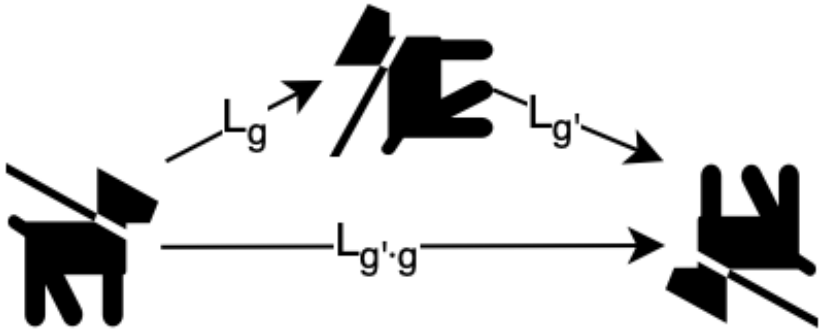

Figure 4.4: Function is $f \in \mathbb{L}_2(\mathbb{R}^2$ (a 2D image.) $G = SE(2)$. The representation is the roto-translation of the image: $\mathcal{L}_g(f)(y) = f(R_\theta^{-1}(y - x))$

$$\mathcal{L}_g[f](x) = f(g^{-1} \cdot x) \tag{4.3}$$

4.1.2 Symmetry Groups

Symmetry Groups are essential in understanding G-CNNs (Sec. 4.2) that we use to leverage rotational and flip equivariances. We first need to define a symmetry transformation to understand a Symmetry Group.

Symmetry is an invariant transformation of an object where a symmetrical change does not lead to any visible changes in the object. We can consider the example of flipping or rotating a cube by 90 degrees about its center. Given all symmetries of an object, we can define some axioms:

- Identity transformation of an object will be a symmetry. (Identity)
- If we combine two separate symmetry transformations by applying them consecutively, then the resulting transformation itself is a symmetry. (Closure)
- The inverse of symmetry is still a symmetry. (Inverse)

You will notice that these axioms are very similar to group axioms. So, a Symmetry Group is just a set of these symmetry transformations. Using the axioms above, we can formally define a symmetry group G with the following conditions:

- An identity transformation $e \in G$ exists where $x \sim_e x$ for every $x \in \mathcal{X}$ (Identity)
- Elements of G are associative where $i(jg) = (ij)g \; \forall \; i, j, g \in G$. (Associativity)
- For any $i, j \in G$, the application of i and j create transformation $ij \in G \; \forall \; i, j \in G$. (Closure)
- For any transformation $i \in G$, the inverse transformation of i: $i^{-1} \in G$. Application of i^{-1} with i on $x \in \mathcal{X}$ results in an identity transformation. (Inverse)

This paper will focus on two main symmetric groups: cyclic groups C and dihedral groups D. The group C_n includes the rotational symmetries of a n-gon. The group D_n is the entire symmetry group of a regular n-gon that includes rotational and flip

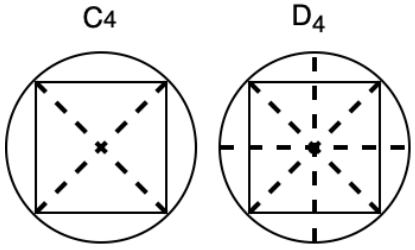

Figure 4.5: Demonstrative diagram for Cyclical Group C_4 and Dihedral Group D_4.

symmetries. Fig. 4.5 provides diagrams for the groups D_4 and C_4. Note that both the C_n and D_n groups can be defined within the $SE(2)$ group: $C_n, D_n \in SE(2)$.

4.2 Equivariant CNNs

4.2.1 Group CNN

Using the notion of groups, we can define the images and feature maps in a regular CNN as functions $f : \mathbb{Z}^2 \to \mathbb{R}^K$ in a rectangular domain. At each pixel location, the kernel stack returns a vector $f(x, y)$ of dimensions equal to the number of channels.

Let's use representation $\mathcal{L}_g$ to denote a transformation to $g \in G$ acting on the set of kernels. $\mathcal{L}_g$ transforms the kernel f to G-space to create $\mathcal{L}_g f$, which is equal to $g^{-1}x$ on the original kernel f. Eq. 4.4 provides the equation for the transformation $L_g f$.

$$\mathcal{L}_g f(x) = [f \circ g^{-1}](x) = f(g^{-1}x) \tag{4.4}$$

Unlike a regular CNN, whose kernels are a function of the group $\mathbb{Z}^2$, G-CNNs use the kernels of a group G. The equation of a CNN that is defined using group representation (Eq. 4.4), can apply to any group G by replacing an element of $\mathbb{Z}^2 : x$ with a component of $G : i$ to calculate $g^{-1}i$.

Like a regular CNN, the G-CNN works by transforming the kernel over a space G. Let's consider input in Symmetric Group C_4. Fig. 4.6 visualizes how the filters would be applied to the input to tackle the rotational transformations of C_4. As C_4 is a subgroup of $SE(2)$, it has both translational and rotational components. To tackle this, G-CNN translates four separate kernels rotated by 90 degrees over the input. For a different group G, feature maps will transform according to group representation $\mathcal{L}_g$ as seen in Eq. 4.4.

4.2.2 Equivariant G-CNNs

We can define equivariance with a basic understanding of Symmetric Groups. In simple terms, a function is equivariant if the output changes by the same amount the input changes - do note that this is different from invariance, where the output

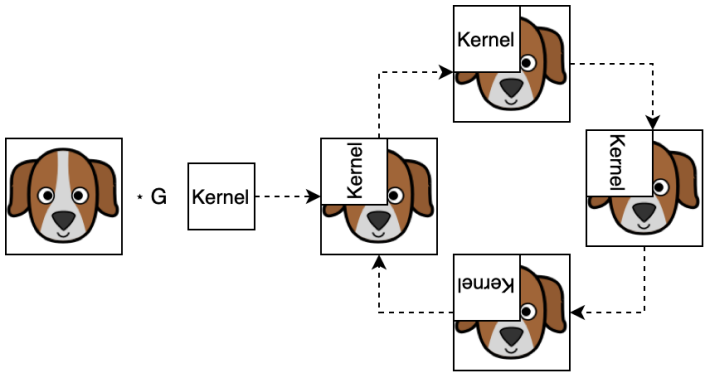

Figure 4.6: Demonstration of rotated kernels being applied on an input in group C_4.

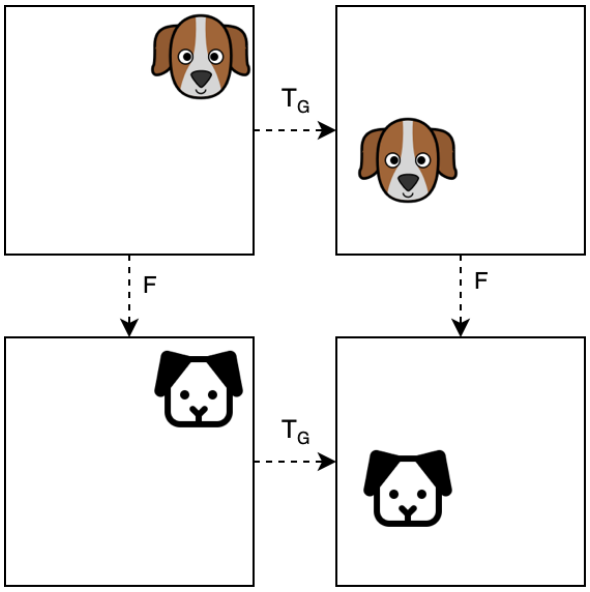

Figure 4.7: Translational Equivariance on shift T_G and function F. Note how the result is the same whether the transformation or the function is applied first.

does not change when the input does change. A function is equivariant under a group G if the input and output domains adjust with that group's action (Eq. 4.5), f is a function and t represents a transformation.). Fig. 4.7 represents translational equivariance, as an example.

$$f(t_g(x)) = t_g(f(x)) \; \forall \; g \in G \tag{4.5}$$

Convolutional Neural Networks are inherently equivariant to translations due to the operations of convolution layers. As a reminder, a convolutional layer has a kernel that translates over different pixel positions over an input, storing a computed value. Since the kernel translates over the whole image, it trains over all the translated versions of the input (Eq. 2.13). So, when a new input is just a translated version of the old input, the output value will just be translated based on pixel locations. In the case of Fig. 4.7, the function F will be a convolution.

Based on Cohen et al.'s [19] work, we can prove the translational equivariance of a CNN by using the group representation $\mathcal{L}_g$ as a translation t in group T based on Eq. 4.4:

$$\mathcal{L}_t(f) = f \circ t^{-1} \tag{4.6}$$

$$(\mathcal{L}_t(f))(x) = f(t^{-1}(x)) \tag{4.7}$$

Using this notation, we can update our definition of a convolution from Eq. 2.12 to (assuming C = 1):

$$((\mathcal{L}_t f) \star k)(x) \tag{4.8}$$

Then, using the definition of equivariance Eq. 4.5 and Eq. 4.8 we can solve the following set of equations to prove translational equivariance of convolutional layers [19]:

$$\begin{aligned} &((L_t) \star k)(x) \\ &= \sum_y f(y-t)k(y-x) \\ &= \sum_y f(y)k(y+t-x) \\ &= \sum_y f(y)k(y-(x-t)) \\ &= (L_t(f \star k))(x) \end{aligned} \tag{4.9}$$

While the above proof is based on a regular CNN in the group $\mathbb{Z}^2$, the same principles apply to G-CNNs for a symmetric group G by changing $x, y \in \mathbb{Z}^2$ with $i, j \in G$ [19]. Eq. 4.9 can be used on a different group transformation by simply changing the translation group T into any transformation (rotation, flip) group T_G. So, a convolution will be equivariant for a Symmetric Group G if it calculates the convolution for each element in G; this is done by using multiple rotated versions of kernel k (Fig. 4.6). In practice, we must permute the elements of the numerous spun filters to extend equivariance properties to subsequent layers, as not doing so will change the output space to G from Euclidian space. The version of G-CNNs implented in this paper are based on Weiler et al.'s work [46] called e2CNN based on the $SE(2)$ group.

As stated previously, this research focuses on a robot grasping task, meaning the most relevant Group is the $SE(2)$ Group, which has cyclical and flip symmetries. Due to this, the most relevant Symmetric Groups that can be utilized with G-CNNs are Cyclical Groups. For a cyclical group, the elements $g \in C_n$ or $g \in D_n$ represent rotations within the cyclical group.

In the G-CNN framework, one needs to define representations of how the elements of G apply to the data [44]. There are three representations to consider. First, the regular representation that permutes the elements an n-dimensional vector $(x_1, x_2, ..., x_n) \in \mathbb{R}^n$, $\rho_{reg}(g)x = (x_{n-m+1}, ..., x_n, x_1, x_2, ..., x_{n-m})$ where g is the m-th element in G. Second, the trivial representation that makes no change on a scalar $x \in \mathbb{R}$, $\rho_{tri}(g)x = x$.

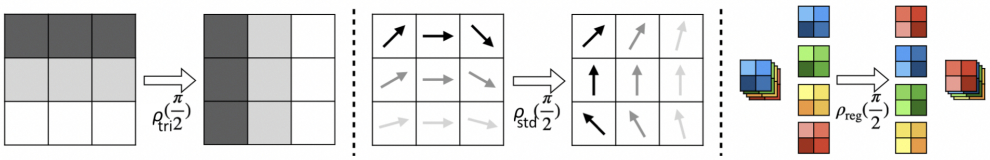

Figure 4.8: Diagram of the actions of e2cnn representations, taken from [48]. On the left, C_n acts on a 1-channel feature map with identical mapping.In the middle, C_n acts on a vector field by rotating the vector at each pixel with ρ_{std}On the right, C_n acts on a 4-channel feature map by permuting the order of the channels using ρ_{reg}

Third, the standard representation ρ_{std} that simply rotates a vector x. These representations will be crucial when defining the equivariant models we implement under Sec. 7 and Sec. 5. Fig. 4.8 provides a diagram of each representation.

Chapter 5

Preliminary Models

This chapter introduces two models visual imitation learning models built during the process of this thesis. While there have been many models worked on, these two models are the ones that led to the final implementation that will be explained in Sec.7. One is the baseline model that purely predicts end-effector (EE) velocities, similar to the model utilized by James et al. [13]. The other is the equivariant version of the baseline model, where each CNN has changed with a G-CNN network that allowed various Symmetric Groups to be tested in predicting end-effector velocities.

5.1 Experimental Setup

The simulation environment is built on CoppeliaSim [49] using PyRep [50]. The robot arm is a 7-DOF Panda arm with a two-fingered gripper; in this experimental setup, the gripper has a vision sensor, with a 64x64 resolution, placed near the end-effector to capture visual information from the environment.

The goal of the environment is simple: moving the end-effector of the Panda arm to a position where it can grasp randomly placed objects on the tabletop using visual inputs. Fig. 5.1 and Fig 5.2 provide an overview of the experimental setup demonstrating the position of the arm and one possible location of the cube; Fig. 5.3 and Fig. 5.4 show a possible end-state of start-state indicated in Fig. 5.1.

As the models introduced in Sec. 5.2 are all behavioral cloning models, there is a need for expert demonstrations. Manually generating trajectories is not feasible considering the number of samples needed, so we used the Inverse Kinematics (IK) module in PyRep to generate trajectories towards randomly placed cubes.

We had two primary considerations in generating usable trajectories for the behavioral cloning models. One, the expert paths generated would have to keep the target object within the point of view of the vision sensor at all times. Otherwise, the trained model may move to positions where the target is not within sight, resulting in randomized movement that will lead to failure. Two, expert trajectories must include variance to handle the covariate shift problem. Hence, given a starting state,

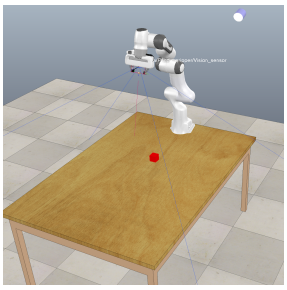

Figure 5.1: Setup, 3rd Person View.

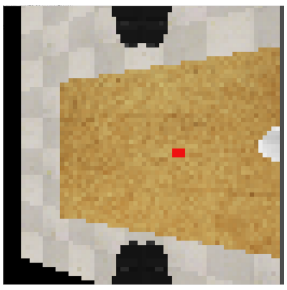

Figure 5.2: Setup, 1st Person View.

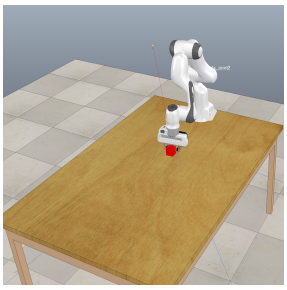

Figure 5.3: Setup, 3rd Person View.

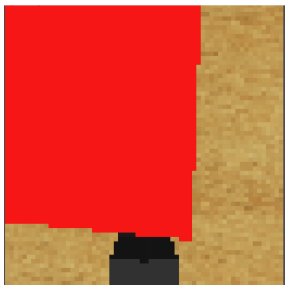

Figure 5.4: Setup, 1st Person View.

we must generate multiple similar paths towards the cube so the output model can better generalize a course in a similar situation.

Based on these two considerations, the setup includes several environmental restrictions to generate good trajectories. These are as follows:

- **Grasping Angle:** The trajectories are restricted such that the end-effector will always be perpendicular to the table's surface. This angle restriction ensures similarity in captured images during the approach and increases the likelihood of keeping the cube within vision throughout the trajectory.
- **Linear Trajectories:** The expert demonstrations will only consist of linear trajectories to the target. Combined with the grasping angle, this ensures the cube is always within vision. Plus, a linear path is similar to a human's route to a goal as it's the shortest path and is easier to visualize.
- **Target within the Object:** The IK module will generate a path to a random position within the object rather than picking the center of the object. The random coordinate is chosen from an area within the object that is optimal to grasp. This will increase the variance in the trajectories generated to capture the target object.
- **Restricted Cube Positions:** During expert demonstration generation and testing, the randomized cube positions will be restricted to be on top of the table and visible from the vision sensor placed on the end-effector.

We gather expert demonstrations before the training of the BC models. As the target of the IK module is not the object itself but a random optimal position within the target object, we collect multiple approaches to a single object position for each random object position. For example, if we want to generate 100 expert trajectories, we can use 25 separate object locations and four locations within the object.

As BC aims to emulate expert demonstrations, the model will only train on complete trajectories without significantly augmenting the training data. Given the structure of the models in Sec. 5.2, this may cause issues training the stopping output variable if the only time the variable is adjusted is during the final state of a trajectory τ. To successfully train either model on stopping, we placed a depth sensor on the Panda

gripper to record the distance between the gripper and the target object. Once the target object is within a grasping distance (even if the trajectory is not done), the state will record the stopping variable as one instead of zero. This adjustment ensures the network has enough data points where the gripper must stop.

5.2 Models

The baseline network is a CNN-LSTM network that takes the visual state input and outputs end-effector velocities (linear and angular) and a binary stopping variable. The network input is RGB images (64, 64, 3) taken from the vision sensor mounted on the end-effector of the Panda arm. The state image is normalized over its RGB values before being fed into the model. The network's output includes two additional auxiliary outputs that are not used in training: position of the cube and end-effector relative to the base of the robot.

The LSTM layer provides temporal information as the desired movement of the end-effector will depend on its previous trajectory and the current visual state. LSTM has improved results in various applications [13, 40] and during the testing done to finalize a baseline model.

The loss functions (Eq. 5.2) used to train the network is a combination of MSE (Eq. 2.10) and BCE loss(Eq. 5.1). The y values in MSE loss refer to the end-effector velocities, cube position, and the end-effector position. The y-value in BCE loss refers to the binary stop variable which signals the network to stop. The λ value determines the weight of the BCE. MSE loss aims to train the network on moving the end-effector, while the BCE loss aims to train the network on stopping once the end-effector is in a grasping position. We found that $\lambda = .4$ works best to train the network on stopping while keeping acceptable end-effector velocity accuracy.

$$\mathcal{L}_{BCE}(y) = -1/N \sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i)) \tag{5.1}$$

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}_{MSE}(y, \hat{y}) + \lambda * \mathcal{L}_{BCE}(y) \tag{5.2}$$

Fig 5.5 is the diagram of the baseline network. The fully connected layer outputs all the results. The only outlier is the binary stopping variable, where the sigmoid of the stopping value is calculated to place it between 0 and 1.

The structure of the equivariant network is similar to that presented in Fig. 5.5. While the main difference is that G-CNNs are used instead of CNNs, small changes are made to the structure to utilize G-CNNs. First, we moved the pooling layer in every convolutional block after the ReLU activation to streamline the input and output type adjustments required by e2cnn [46]. Second, we adjusted the input size of the first fully connected layer to fit the larger output size of G-CNNs (due to the use of multiple kernels based on group G). Finally, we use various representations (as described in Sec. 4.2.2) to apply the G-CNNs. Specifically: we useρ_{tri} for the first

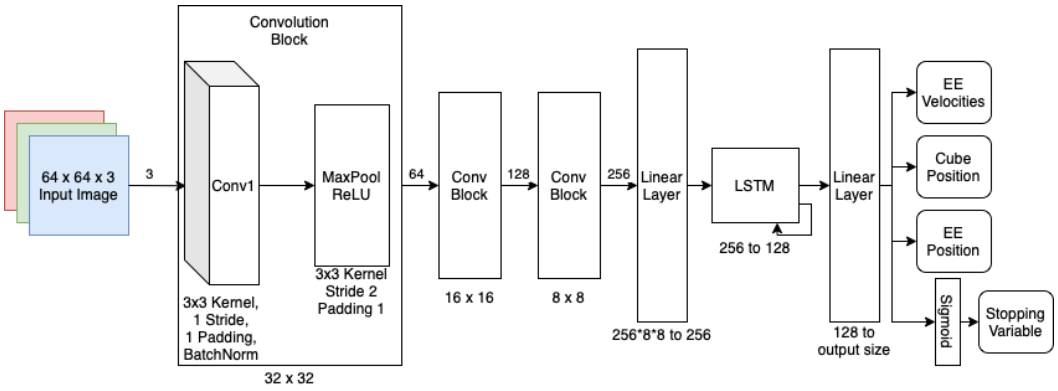

Figure 5.5: Baseline Network with convolutional layers, an LSTM layer, and linear layers to output EE velocities, cube position, EE position, and stopping variable.

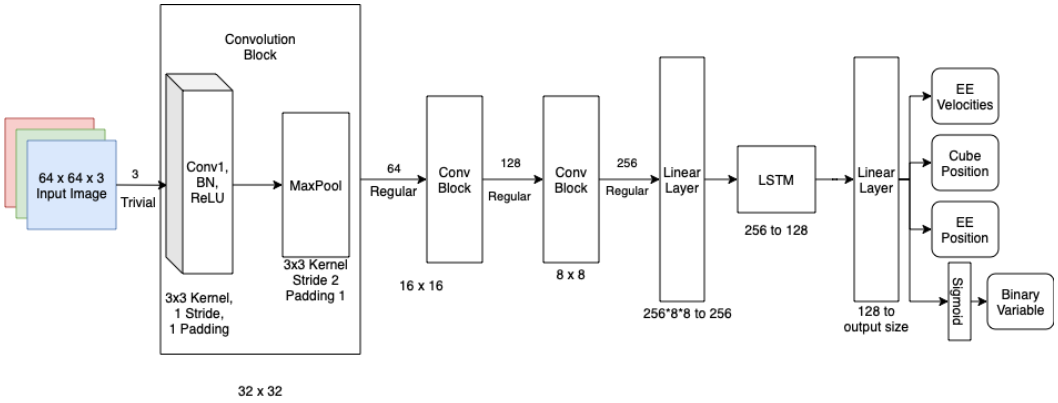

Figure 5.6: Equivariant Network with Group-Convolutional layers, an LSTM layer, and linear layers to output EE velocities, cube position, EE position, and stopping variable.

G-CNN's input as it is used to process the input image. Afterward, we use ρ_{reg} for all the inputs and outputs of the G-CNNs to leverage the equivariant properties of the $SE(2)$ group. Fig. 5.6 depicts the diagram for the equivariant baseline model.

Chapter 6

Preliminary Experiments

This chapter will detail several experiments. These experiments will determine the relative performance of equivariant and baseline models and the equivariant properties of each model. Note that the experiments are not focused on determining optimal hyper-parameters as the aim is to find the differences between the equivariant and the baseline representation.

For all the experiments detailed, we calculate accuracy in the same manner. A trained model works in a testing environment where a cube spawns randomly on the table-top, at a graspable position (Fig. 5.1). For the grasp to be accurate, the model must move the end-effector into the grasping area within the cube and turn on the stopping variable to freeze the arm's position or be within the grasping area within the object 150 steps. If the model processes both actions mentioned earlier, the trial is considered a success. The test runs fifty separate trajectories and counts the number of successes. Then, for accuracy, we calculate the number of successful trials divided by the number of total attempts. A higher accuracy value indicates a better model.

To calculate the accuracy of each model, we sampled trajectories four separate times to test four different instances of each model.

The stopping accuracy is calculated using Eq. 6.1 where $C_{t,s}$ is the number of stopped successful trajectories and C_t is the number of all successful trajectories.

$$A_s = \frac{C_{t,s}}{C_t} \tag{6.1}$$

In all the experiments, the equivariant model is tested in Symmetric Groups C_8 (Cyclical Group of 45 degrees) and D_4 (Dihedral Group of 90 degrees) due to their high accuracy in $SE(2)$ spaces [45].

Unless stated otherwise, the following hyperparameters were used during training: .0004 Learning Rate, 20 Epochs, 32 Batch Size, 1E-7 weight decay.

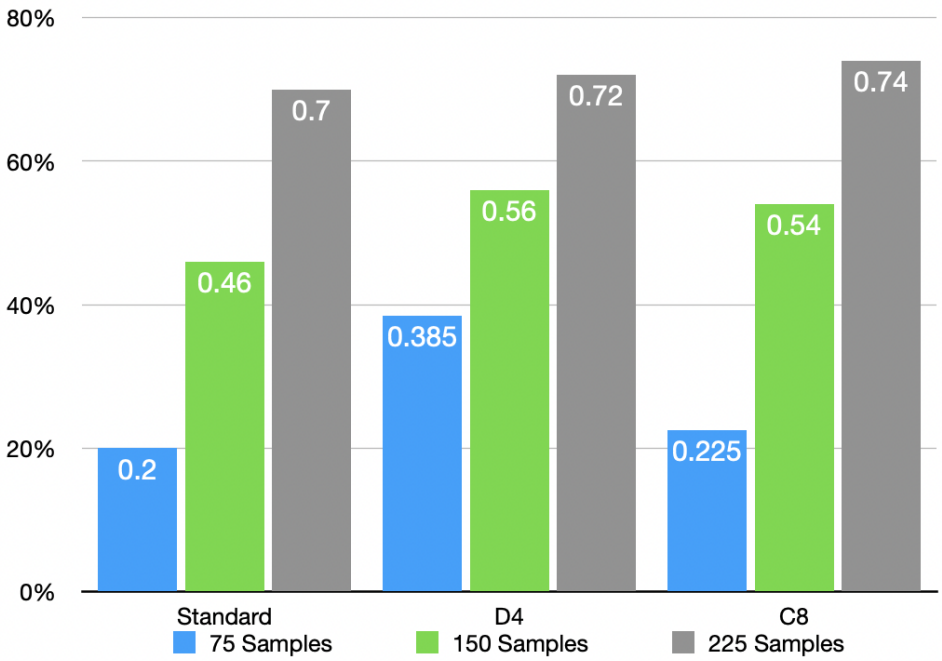

Figure 6.1: Diagram of accuracy results for end-effector models given 75, 150, and 225 expert trajectory samples of non-rotating cubes to train on.

Stopping Accuracies		
Standard	D_4	C_8
92%	81.5%	83%

Table 6.1: Stopping accuracies of end-effector models given 225 expert trajectory samples of non-rotating cubes during training.

6.1 Different Sample Rates

This experiment aims to determine the sample efficiency of the model. Specifically, it tests the accuracy of each model trained at different numbers of expert trajectories: 25, 50, and 75. As explained previously, three random spots within the cube are targeted to add variance to each trajectory. This increases the number of demonstrations to 75, 150, and 225.

We expect the models to perform better when trained on a higher number of demonstrations and, specifically, the G-CNN models to perform better at lower sample rates due to rotational equivariances.

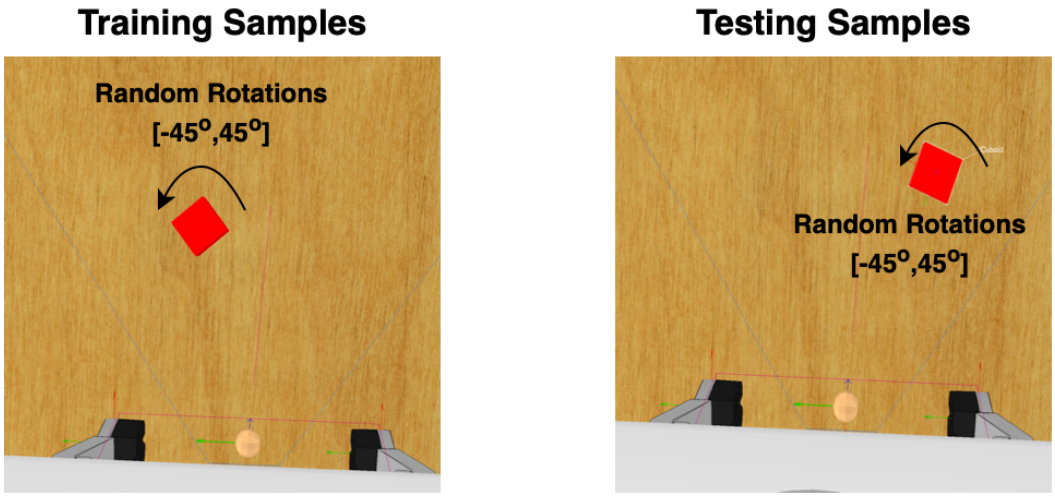

Figure 6.2: Example samples from the training and testing sets for the Rotated Cubes Test.

6.1.1 Results

Looking at the Fig. 6.1 reveals that there are no significant difference in the performance of CNN and G-CNNs while predicting end-effector velocities. However, what the graphs fail to demonstrate is the accuracy of stopping. Table 6.1 reveals that the standard CNN model is superior in stopping a trajectory than either G-CNN models.

The results do not match our expectations. It is true that the G-CNN models perform slightly better at lower sample rates compared to the standard CNN model. However, the differences are only significant at the 150 sample level. Furthermore, when stopping is considered, the G-CNN models perform significantly worse than the standard model.

6.2 Rotated Cubes

This experiment tests each model on rotated target objects to determine how each model handles rotational changes. This is done to see if the rotational equivariant properties of G-CNNs are being leveraged by the models.

To do so, we sample 800 episodes (200 separate locations and four random rotations for each position.) For training, the trajectories are generated so that the end-effector's final orientation matches the cube's orientation.

The rotations tested are between -45 and 45 degrees due to the inherent geometry of the cube, where all other revolutions will be symmetric to the sampled range. Look at Fig. 6.2 for a visual diagram.

The expectation is that the G-CNN models will perform better than the standard CNN models in this experiment.

Accuracies		
Standard	D_4	C_8
33%	52.5%	56%

Table 6.2: Table of accuracy results for end-effector models for the rotated cube experiment, given 800 expert trajectory samples of rotating cubes to train on.

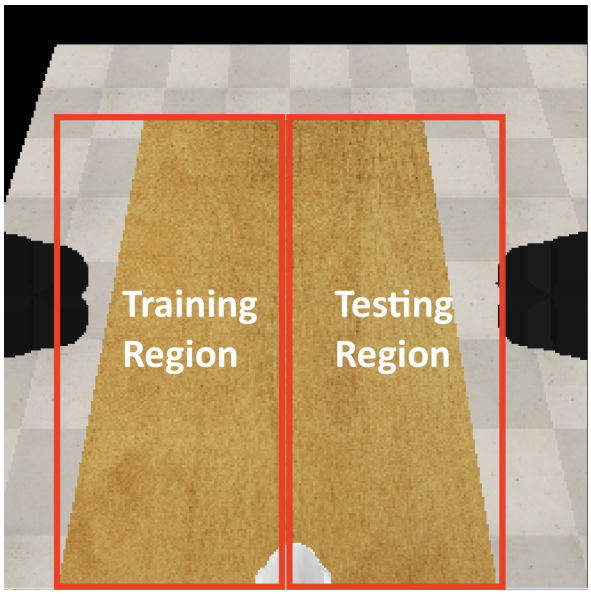

Figure 6.3: Diagram demonstrating the seperation of testing and training regions for the One-Sidded Training test.

6.2.1 Results

Table 6.2 depicts the results of the experiment. The worse accuracy across the board is due to the higher utilization of rotational velocities in the network output. The results reveal that the non-rotation-equivariant group CNN performs severely worse than its C_8 and D_4 counterparts. This discrepancy in the results is expected as C_8, and D_4 G-CNN models are equivariant to rotational changes in the input. So, the models are likely to recognize the cube more accurately as they are more sample-efficient for rotational tasks [46].

The difference in C_8 and D_4 accuracies is within the margin of error. So, either model performs similarly with a cube as a target. However, neither the C_8 nor the D_4 models learned to rotate to match the target's orientation. Instead, either model seems to be rotation invariant as they do not change their final rotation to grasp a cube, indicating that none of the proposed models can tackle rotations correctly.

Accuracies		
Standard	D_4	C_8
3%	4.5%	2.5%

Table 6.3: Table of accuracy results for end-effector models for the One-Sided Training experiment, given 225 expert trajectory samples of non-rotating cubes on. Training trajectories are only available on the left side of the environment while testing trajectories are only on the right side of the environment.

6.3 One-Sided Training

This final experiment is to gauge the equivariant properties of either model. Data will only be collected on the left side of the table, while tests will be done on the right side of the tabletop. Fig. 6.3 demonstrates this.

Since D_4 is equivariant in 90-degree rotations and flips while C_8 is equivariant in 45-degree rotations, the expectation is that the network will be able to generalize values trained on one side of the network to the other. Hence, the model should use a position on the left side of the network to correctly interpret a symmetric location (based on group C_8 or D_4) on the right side of the network.

6.3.1 Results

Fig. 6.3 present the results for the experiment. The results indicate that none of the models can generalize cube positions from one side of the state space to the other. The accuracy is not 0% during testing because the cubes spawned in the middle of the table-top, where the left and right spaces indicated in Fig. 6.3 meet. The lack of generalization between two sides of the state-space suggests a lack of translational equivariance between the input and the output of the models. This is further discussed in Sec. 6.4.

6.4 Preliminary Results

While each model reaches an accuracy of over 70% given 225 samples, there are some unexpected results from experiments (Sec. 6). The results indicate that over 100 separate target positions are required for a model that performs relatively well for a model with stopping to work. Furthermore, none of the models learn well when extra complexity is introduced through rotational changes. While the G-CNN models perform better given rotated targets, none of the proposed models rotate the end-effector to grasp an object. While further improvements in the model pipeline might fix this, there is a more significant issue with the proposed approaches.

The biggest issue with the results is the equivariance testing (Sec. 6.3). None of the models successfully learn to generalize into un-trained cube locations. While the rotational testing does indicate that the rotational kernels implemented by G-CNN

[46] somehow increase the sample efficiency or accuracy, as the G-CNN models perform better than the regular model, we are unable to observe equivariance over the whole action-space. This indicates two things: the G-CNN networks can learn various rotations of the target object but fail to use equivariant properties of the state space to generalize end-effector policies based on cube locations.

The failure to generalize results to different cube positions is likely due to two reasons: the end-effector velocities and Fully Connected layers. The Fully Connected layers in the models serve as an interpretation layer. It takes flattened feature maps output by the CNNs and uses them as regression variables. Hence, while the output feature maps may be equivariant, if there aren't any data points to indicate what an end-effector velocity should be given feature maps, the fully connected layer will fail to generalize the outputs accurately.

Using end-effector velocities as outputs do initially make sense. The space of end-effector velocities is Cartesian as they indicate x,y, and z values plus the relative rotations for each of these axes. However, a change in the input space does not reflect exactly into the EE velocity's cartesian space. Instead, the velocities related to the y-axis are always positive, and the z-axis is always negative since the target is in front of the end-effector. Hence, a change in the input will not cause an immediate difference in the output, indicating that while the feature maps are equivariant, the end-effector outputs are not. Furthermore, equivariance only applies in the same space/group (Sec. 4.2.2). The input is an image. However, the output is velocities - one can not directly transform velocities using positional changes. Considering these two issues and given the definition of equivariance in Eq. 4.5, we can infer that the action representation is not equivariant to the input. If we transform the cube's position by some transformation t, the output will not directly transform by t. Fig. 6.4 demonstrates this visually.

To tackle the lack of equivariance (and, hence, adaptability) in end-effector velocities and fully connected layers, we introduce the Spatially Equivariant Action Maps (SEAM or EAM) as an action representation in Sec. 7.

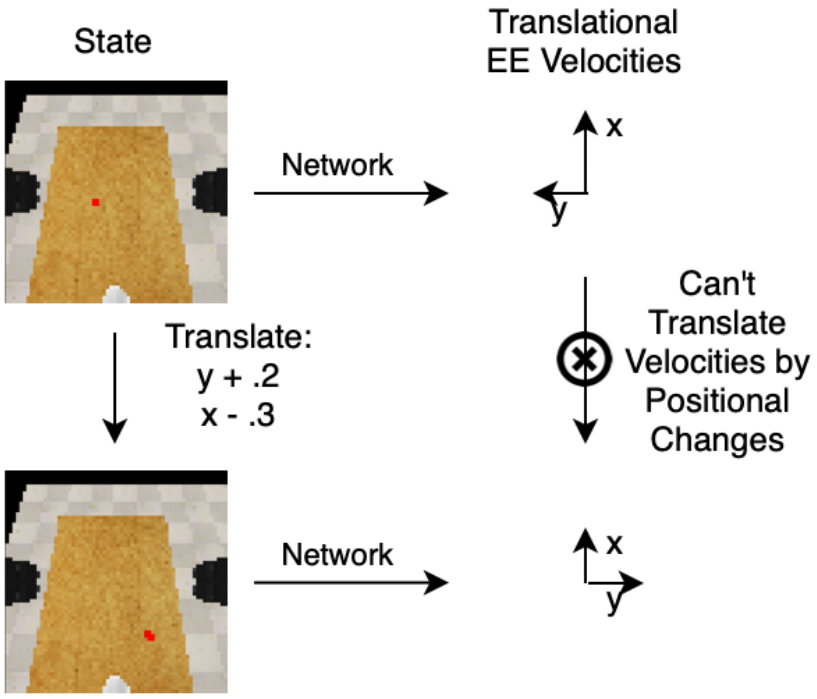

Figure 6.4: Diagram indicating that a change in x and y positional values do not directly reflect to changes in velocity values. So, transformations in an input image are not equivariant with output end effector velocities.

Chapter 7

Spatially Equivariant Action Maps

This chapter introduces the proposed novel action representation for visual imitation learning: Equviariant Action Maps (EAM). EAM aims to tackle issues related to equivariance in traditional grasping methodologies that utilize end-effector velocities by creating a more sample-efficient and equivariant action representation. The method is not superior to models that estimate end-effector velocities, as it has many drawbacks and requirements to work. However, it provides an equivariant and adaptive solution to a robot grasping problem by generating positional action maps that map end-effector positions to a 2D array that is the same size as the input image, based on Q-Maps [41] and Spatial Action Maps [42].

7.1 Problem Statement

The main objective is to introduce a novel action representation for visual behavioral cloning that is intuitive to human understanding. Keeping this in mind, we build our problem statement based on a robot grasping task to leverage the equivariant properties of its relevant SE(2) state space.

The goal of a visual grasping task is moving the end-effector of a robotic arm to a position where it can grasp a particular object using visual inputs. A grasping job can be formally defined with a grasp function Ξ, which maps from an image of a graspable object $s \in \mathbb{R}^{c \times h \times w}$ to a planar end effector location $a \in SE(2)$ where an object can be grasped.

To leverage the equivariant properties of the $SE(2)$, we need to introduce several assumptions:

Assumption 1 *The action space a must be aligned with the input space s. Specifically, the reference frame of an action a must align with the reference frame of state s.*

Assumption 2 $ga = g\Xi(s) = \Xi(gs)$ *where $g \in G$ is a transformation in symmetric group $G \in SE(2)$, gs is the image s transformed by g, and ga is the action a transformed by g.*

For the grasp function Ξ to be equivariant, transformations applied to image s must directly adjust the final gripper position generated using action a. To fulfill this, we need Assumption 1 to be valid for the transformations to be aligned: the action and states must be aligned. Due to the dynamics of grasp positions being independent of the reference frame from which we view a system, the Assumption 2 is satisfied in object grasping scenarios.

Based on these two assumptions, we can define a grasping task using the subgroup G of SE(2) with the following:

- **State Space:** A state $s \in \mathbb{R}^{c \times h \times w}$ a c-channel image. $g \in G$ is the group operator that acts on state s using representation ρ_{tri} to rotate the pixels without changing the pixel feature vector.
- **Action Space:** The action space can be defined as an array of 3D positions the end-effector of the gripper can reach. Simplifying the 3D positions into 2D (x,y) positions, we can map each state and action into array of size (h,w) of state s so the action space and the state space are aligned. Further discussed in Sec. 7.2.
- **Successful Grasp:** We assume that the goal of the environment is to move the end-effector to a position where it can successfully grasp a target object. The goal, in this case, is invariant to transformation g. gs denotes a transformed state, and ga indicates a transformed action. This is due to $goal(s, a) = goal(gs, ga)$ as once s transforms, a transforms.

7.2 Method

Using the problem statement above, we can formulate the visual object grasping task as a BC problem where an agent learns an expert policy π_e by regressing over a set of expert demonstrations τ of state-action pairs $(s, a), s \in S, a \in A$. The agent trains a deep neural network using expert trajectories to generate π that matches the expert's policy π_e.

To tackle this BC problem by leveraging the equivariant properties of $SE(2)$ space, we introduce a new action representation, Spatially Equivariant Action Maps (EAM).

7.2.1 Action Maps

Spatially Equivariant Action Maps are based on Q-Maps [41] and Spatial Action Maps [42]. EAMs try to estimate the final position of the end-effector given an input image $s \in \mathbb{R}^{c \times h \times w}$. However, rather than directly outputting the x, y, and z values, they generate an 2D array of size $(h \times w)$ to match the size of the input image. Each element in the generated array corresponds to an (x, y) coordinate in the state space, matching the input s. Each component of the EAM array indicates a value for their respective pixel positions. Each value represents the weight w given to a coordinate based on a state s. The element with the highest w indicates where the end-effector

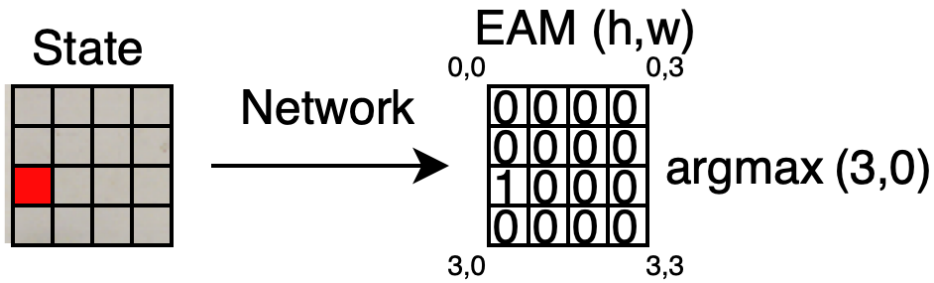

Figure 7.1: Diagram indicating an EAM given a state image of size (4×4) where the red box indicates the object to grasp. The argmax argument provides the position of the largest weight.

should move to grasp an object. Once a position is chosen, inverse kinematics is used to move the end effector to the respective location.

Refer to Fig. 7.1 for an example of an EAM given a state. The state s is an image of size (4×4) where the red square indicates the grasp-able item. For simplicity, let's assume that each 'pixel' in the environment has an area of 1 cm^2 with each side 1 cm long. The output EAM matches the size of the input with a 2D array of size (4×4). Given that the size of the environment is $4 * 4 = 16$ cm, each pixel indicates a position (x, y) on the environment. In the example, the object's position is $(3, 0)$. Once the EAM of the input is generated and the argmax is calculated, the resulting position is also $(3, 0)$. So, the end-effector will move to position $(3, 0)$ indicated by the weights in the EAM.

In a more complex scenario, the size of the environment (X, Y) must be divided into the size of the input (h, w) as it will match the size of the output. This is detailed in Eq. 7.1, where A is the action map of environment positions based on an EAM, $w \in W, h \in H$ are element location in A with column value w and row value h, and X, Y values indicate the set of X and Y coordinates of a square environment, relative to the base of the gripper. $x \in X, y \in Y$ indicate the Cartesian coordinates of the grasp position r.

$$A_{y,x} = [\frac{y - \min(Y)}{\max(Y) - \min(Y)}, \frac{x - \min(X)}{\max(X) - \min(X)}] = [h, w] \tag{7.1}$$

The EAM is aligned with the state space, allowing a network to leverage equivariant properties of $SE(2)$. Denoting EAM and its positional transfer function Eq. 7.1 as function ζ, we can show indicate the equivariance of EAM with Eq. 7.2, demonstration Fig. 7.2.

$$ga = g\zeta(s) = \zeta(gs) \tag{7.2}$$

Training an EAM is likely faster than training a model estimating end-effector velocities based on visual input. The only requirement is generating action maps based

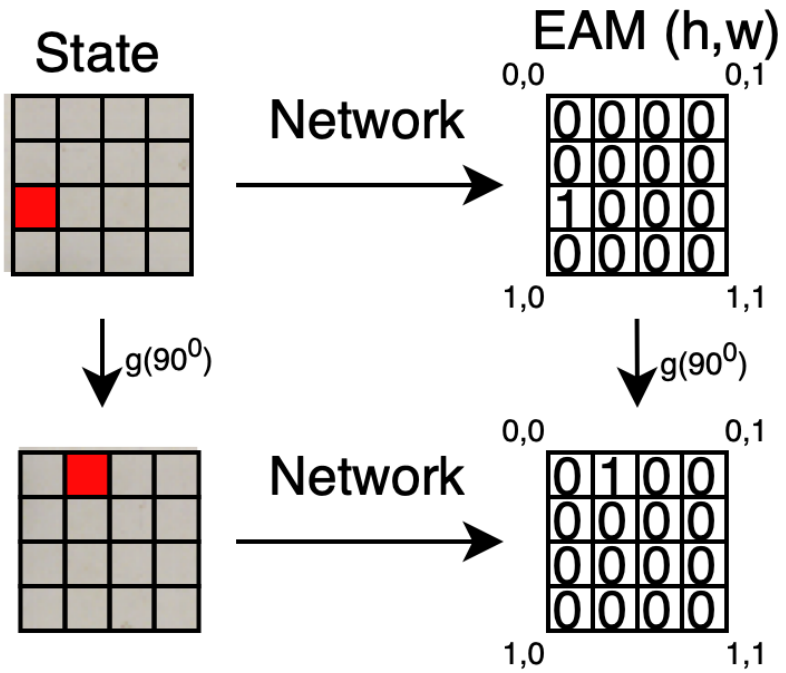

Figure 7.2: Diagram indicating the equivariant property of EAM based on group operator g for a $\pi/2$ rotation.

on a single image. Plus, EAMs can leverage equivariant properties, as just discussed. However, there are some significant drawbacks to EAMs. These are as follows:

- **Knowledge of the Environment:** To set-up an EAM representation, one needs inherent knowledge of the environment such as position and size of the environment space relative to the base of the gripper. On top of this, the visual input of the network must be aligned with the available positions within the environment, requiring knowledge of camera intrinsic.
- **Lack of Z-axis and rotations:** With the given implementation of EAMs, it is only applicable to 2D spaces. The maps are not able to convey rotational or height-wise positions. Suggestions will be further discussed under Sec. 9.2.
- **Reliance on Inverse Kinematics:** While a representation based on end-effector velocities allow more control over the robot gripper, EAMs rely on IK or action planners to generate a path to a position relative to the base of the robot. So, there may be issues with collision, depending on the environment.
- **Reliance on Simple Tasks:** EAMs only provide positional arguments. So, to tackle a complex task, the complex task must be divisible into a simple grasping task. Further discussion under Sec. 9.2.

Fig. 7.3 demonstrates EAMs in a simulation environment in a more complex situation. The following sections will explain how to generate expert demonstrations and what networks to use in order to generate such a model.

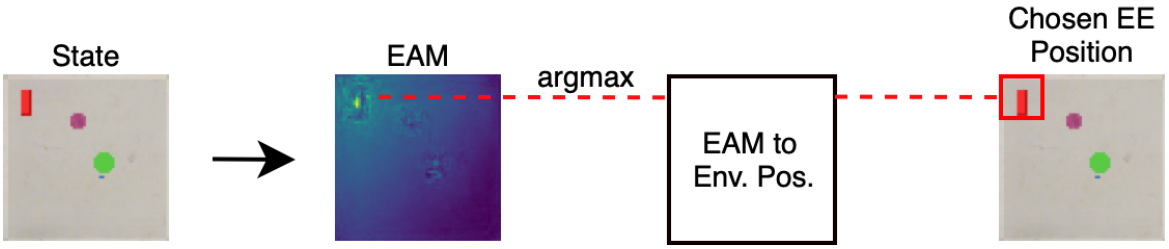

Figure 7.3: Diagram of an EAM in a simulation environment. Yellow indicates higher weight in EAM.

The EAM to Environment Positions module in Fig. 7.3 provides the mapping of pixel positions to environmental space using Eq. 7.1. However, if the robot's base and action region's rotation do not match up, Eq. 7.1 does not work. So, if a rotation matrix is used to align the action region and the robot's base region, we can use Alg. 1 to calculate a position in the environment based on the pixel location. In 1 for X_β, Y_β, β relates to a side. For X, if β is zero, then we consider the left side of the area. For Y, $\beta = 0$ indicates the top side of the area relative to the robot's base. This mapping is done to match the X and Y positions with the pixel coordinates used by the Pillow repository in Python.

Algorithm 1 Pixel Position to Environment Position

Given: Pixel location (h, w) and size of EAM (H, W)
$p_x = ((X_1 - X_0)/W * w) + X_0$
$p_y = ((Y_1 - Y_0)/H * h) + Y_0$
Return: $(p_x, p_y) \in (X, Y)$

7.3 Generating Expert Trajectories

EAMs, as described in the previous section, require an input image aligned with the environment's action space. Using this input, we can train a network to generate an array of weights, as seen in Fig 7.1. So, developing EAMs to train on will depend on the network used. The implementation of EAMs in this paper utilizes UNets [28] (see Sec. 7.4), so the demonstration generation will be for Fully Convolutional Networks.

To generate demonstrations, first, we must set up the grasping environment to have an overhead view of the available action space. Afterward, given the cartesian coordinate space (Y, X) of the 2D action space, we determine the $\max(Y), \min(Y)$ and $\max(X), \min(X)$ values. Then, the following steps are taken to generate an array of weights:

1. Given a state s, determine the grasp position $(\hat{y}, \hat{x})$ of the target object relative to the base of the robot.
2. Calculate the pixel positions $(\hat{h}, \hat{w})$ of $(\hat{y}, \hat{x})$ using Eq. 7.1 with $A_{\hat{y},\hat{x}}$.

3. Generate an array A of zero values of size (H, W) where H is the height of the image input and W is the width of the image input.
4. Set $A(\hat{h}, \hat{w})$ value to one, leaving rest of the weights at zero.
5. Save the array (can be saved as an Image using PIL.)

The weight values may be adjusted to account for how the array is saved. If PIL is used, the array value needs to be multiplied by 255 to account for PIL's image creation requirements.

Only giving weight to a single element may cause issues training the PIL network. Hence, a gradient array based on Manhattan distance was utilized for better generalization. The following set of instructions is for the gradient EAM implementation:. The weight using L1 distance is calculated by normalizing negative $L1$ (Eq. 7.3) distances between 1 and 0 (Eq. 7.4, where $L1_{max}$ is the maximum L1 value in the environment for $(\hat{h}, \hat{w})$).

$$L1([x_1, y_1], [x_2, y_2]) = |x_1 - x_2| + |y_1 - y_2| \tag{7.3}$$

$$w_{h,w} = \frac{-(L1([h, w], [\hat{h}, \hat{w}]) - (-L1_{max})}{0 - (-L1_{max})} \tag{7.4}$$

1. Given a state s, determine the grasp position $(\hat{y}, \hat{x})$ of the target object relative to the base of the robot.
2. Calculate the pixel positions $(\hat{h}, \hat{w})$ of $(\hat{y}, \hat{x})$ using Eq. 7.1 with $A_{\hat{y},\hat{x}}$.
3. Generate an array A of zero values of size (H, W) where H is the height of the image input and W is the width of the image input.
4. Set $A(\hat{h}, \hat{w})$ value to one, leaving rest of the weights at zero.
5. Set adjacent elements to $A(\hat{h}, \hat{w})$ to .75.
6. Use Eq. 7.4 to generate the normalized L1 values for each pixel and multiply it by .4 to decrease the weight.
7. Save the array (can be saved as an Image using PIL.)

Fig. 7.4 shows the implementation of two different EAM types. The brightness of the pixel indicates the weight of the pixel. The gradient helps with the recreation of the map as a single pixel value may be taught randomly. Providing higher pixel weights to adjacent pixels increases the generalization as the optimal pixel is not always in the same spot for an object.

7.4 Networks and Losses

The output EAM must be the same size as the input for the equivariant properties to work directly. To achieve this, we utilize a Fully Convolutional Network as the network for the EAM[42]. Specifically, we use an architecture similar to a shallow

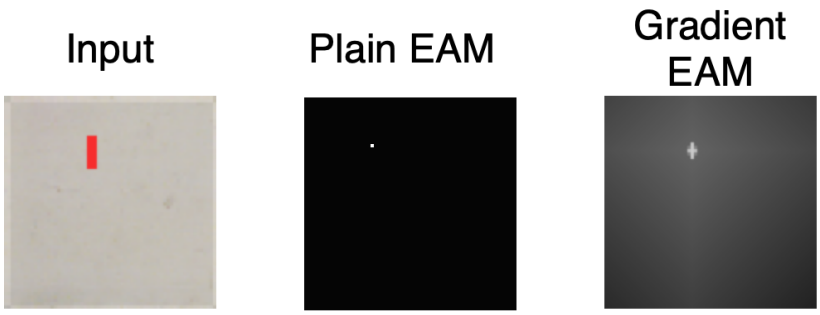

Figure 7.4: Diagram of the simple and gradient EAM types. The position of the rectangle is clearer on the gradient approach.

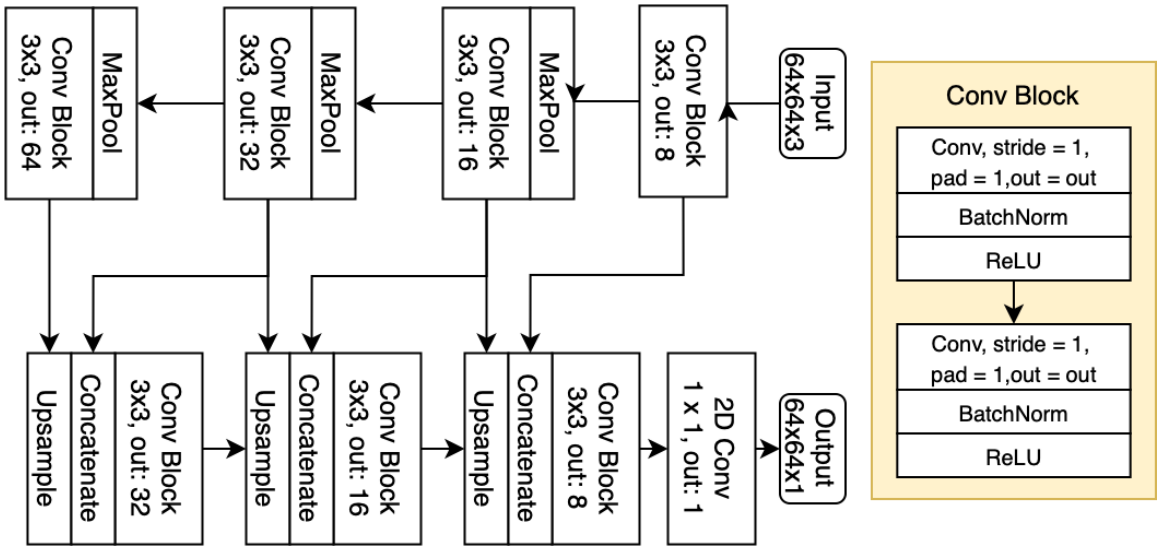

Figure 7.5: Diagram of the EAM network with regular convolutional layers.

UNet [28] for both the standard CNN and the G-CNN (to keep them similar) models as it has performed well in [42] and [43].

Fig. 7.5 and Fig. 7.6 indicate the structure of the utilized networks. In Fig. 7.6, deconvolutional and upsampling layers are present together as upsampling alone does not fully upsample smaller feature maps, increasing the parameter count versus the Standard CNN architecture.

For the equivariant variant of the presented architectures, only the input and the output layers use trivial representations ρ_{tri} as they input and output the final images. The rest of the layers use regular ρ_{tri} representations to utilize the full extent of the equivariant properties of G-CNN. Fig. 7.7 provides a complete overview of the system with the UNet architecture.

The pixel-to-position module is not necessary to train the EAM network. Rather, the UNet is trained to recreate an image similar to the generated EAM images. So, the network trains using the visual input of the environment and the EAM output generated rather than directly on the output positions. Due to this, we chose the

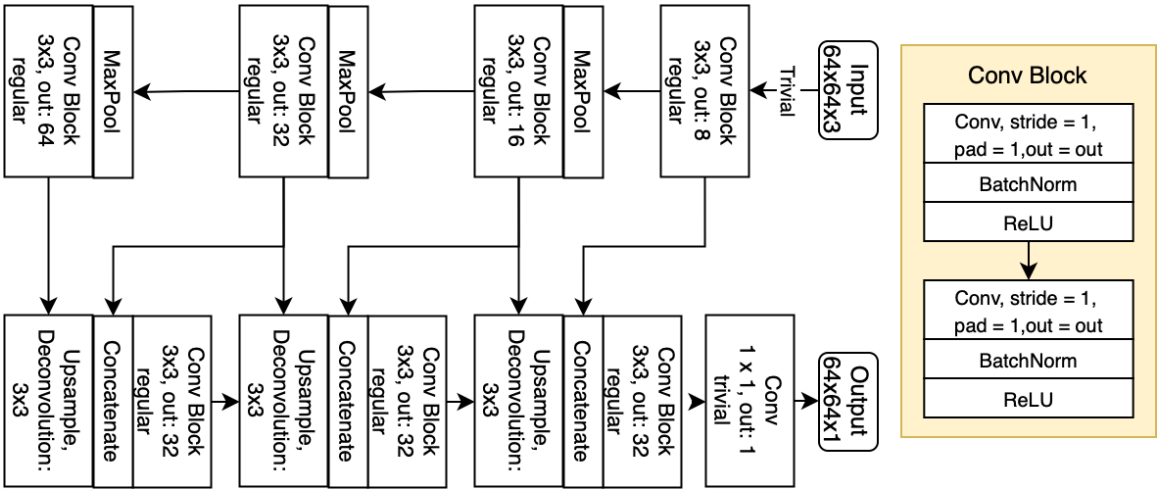

Figure 7.6: Diagram of the EAM network with G-CNN layers.

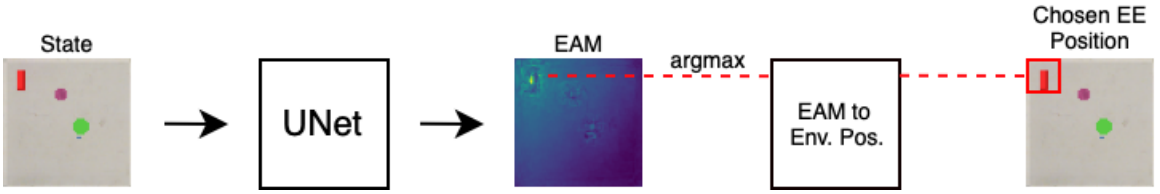

Figure 7.7: Full Diagram of the EAM Architecture

$L1$ (Eq. 7.5) and MSE (Eq. 7.6) loss functions to train the network. Both of these losses are often used in image upscaling and recreation tasks [51]. The $L1$ assists in the final positioning of pixels due to the inherent distance metric, while MSE assists with the final pixel values in our implementation. Without $L1$, pixel positions are less precise, while without MSE, the number of highly weighted pixels increases drastically.

$$\mathcal{L}_{L1}(p) = \frac{1}{N}\sum_{p=1}^{P} |w(p) - \hat{w(p)} \tag{7.5}$$

$$\mathcal{L}_{MSE}(p) = \frac{1}{N}\sum_{p=1}^{P} (w(p) - \hat{w(p)})^2 \tag{7.6}$$

For both of the loss equations, $p \in P$ is a pixel in the reconstruction while w indicates the weight of a given pixel. The final loss function is as follows:

$$\mathcal{L} = \mathcal{L}_{L1} + \mathcal{L}_{MSE} \tag{7.7}$$

Chapter 8

Implementation and Experiments

This section aims to evaluate the usability of the EAM method by testing its accuracy and various equivariant properties. Initially, it will introduce the setup and the relevant implementation details of EAM in the given environment. Afterward, it will detail various experiments before providing the results and evaluating them.

8.1 Experimental Setup

The simulation environment is built on CoppeliaSim [49] using PyRep [50]. The robot arm is a 7-DOF Panda arm with a two-fingered gripper; in this experimental setup, the gripper has a vision sensor, with a 64x64 resolution, placed near the end-effector to capture visual information from the environment.

We set the environment up for a robot grasping task. A square actionable region, where graspable objects will spawn, of size 50×50cm is chosen in front of the Panda arm. The panda arm's initial position is set up so that the image captured by the vision sensor aligns with the actionable region. Fig. 8.1 and Fig. 8.2 provide an overview of the environment and the vision sensor, respectively.

To implement EAM, we need the positional arguments of the actionable region relative to the robot's base. Specifically, this is for the EAM to Env Positions module in Fig. 7.7. While we can calculate the max and min X and Y values by hand, we placed two dummies on the top right-most and bottom left-most positions of the actionable region to make implementation more manageable. Then, we used PyRep to get their location relative to the robot's base. Since the robot's base's rotation matches the actionable region's rotation, we use the two dummies to determine the region's maximum and minimum X and Y values.

The camera's rotation matches the end-effector's rotation. If the gripper is rotated by θ degrees, then we turn the output EAM by $-\theta$ to keep the action region aligned with the robot's coordinate system.

During the data gathering phase, objects spawn in the actionable region. Once a

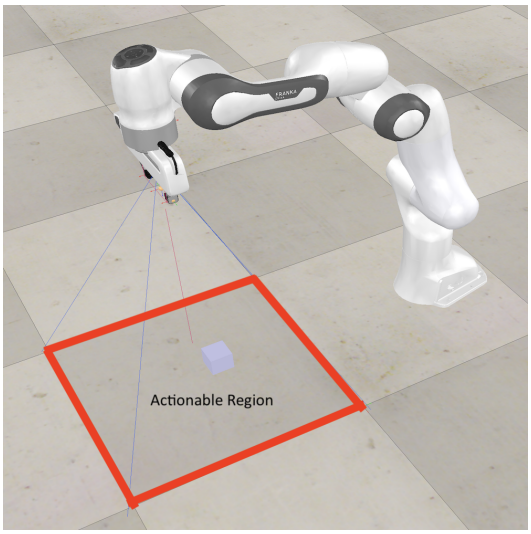

Figure 8.1: Overview of the Experimental for EAM.

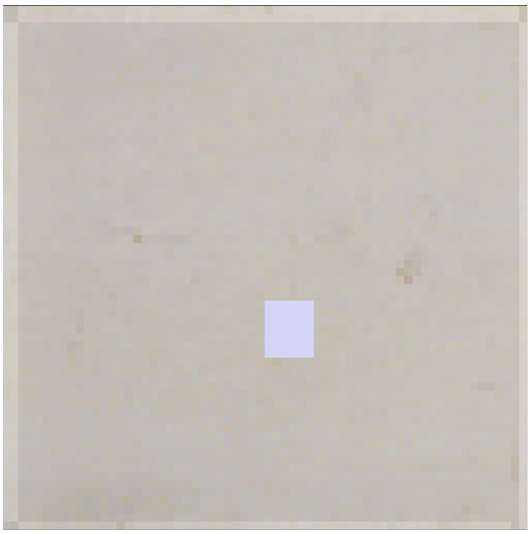

Figure 8.2: Camera view of the Setup for EAM.

target object generates, we use PyRep to determine its position relative to the robot's base. Then, we can apply the procedure outlined in Sec. 7.3.

During training, unless stated otherwise, the following parameters are used: 0.001 learning rate, 10^{-7} weight decay, 30 epochs, 16 batch size, adam optimizer.

During testing, we run 50 grasp attempts. Each attempt randomly positions the target object within the action region. Then, the EAM network estimates a location for the target object, which the PyRep Inverse Kinematics module executes to reach the target. As the network does not predict a height (Z) value, we use the height of the grasp region within an object to define the final position to execute the grasp. If the end-effector is within the grasp region of the target, the run is considered accurate. Accuracy is the value we get from dividing the number of correct runs by the total number of runs. Qualitative metrics are evaluated based on the experiment as well.

In order to increase the applicability of the accuracy results, each model is trained four separate times with four separately sampled data sets. The tests are ran on each model and the average accuracy is provided. This decreases the effect of outliers that may happen due to training and data distribution.

8.2 Accuracy on Sample Sizes

The equivariant properties of CNNs have mainly been used to increase the sample efficiency and accuracy of networks [20][44]. This experiment aims to test how the EAM model performs with a CNN and G-CNN (D_4 and C_8 groups) backbone by training on three sample sizes: 20, 75, and 150. This test will reveal the general robustness of the EAM method while revealing the performance differences between CNN and G-CNN applications.

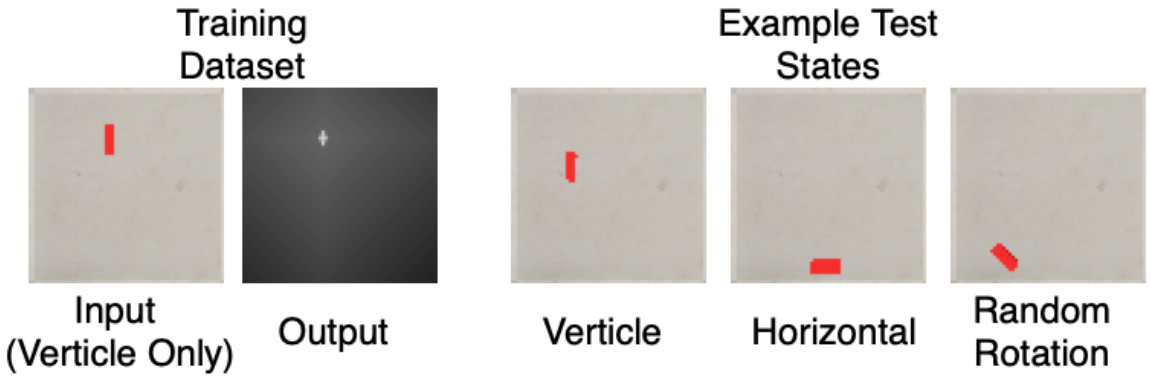

Figure 8.3: Diagram of the Training and Test Inputs for Sample Size Test.

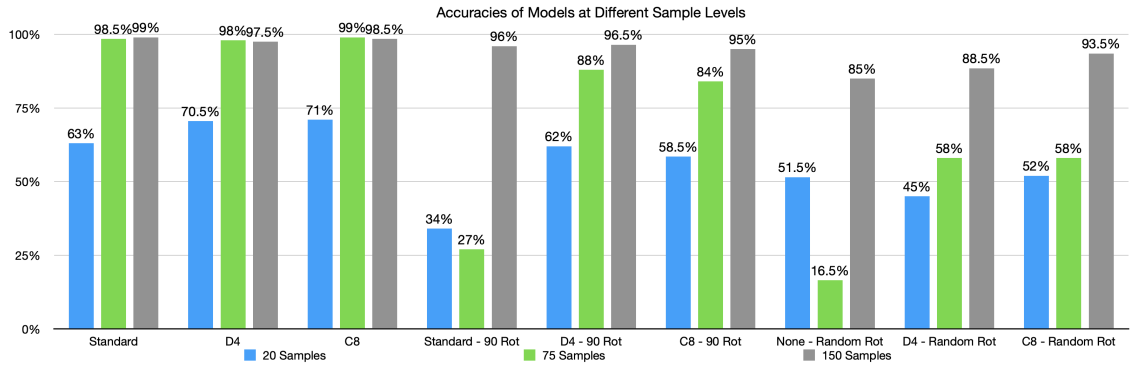

Figure 8.4: Diagram of accuracy results for EAM models trained with 20, 75, and 150 target location samples. 90 Rot refers to only 90 degree rotations while Random Rot refers to random rotations between -90 and 90 degrees.

Expert demonstrations of this experiment only include vertical rectangles as target objects. There are no distractors present during training or testing. However, the experiment considers the accuracy of horizontal, vertical, and rotated rectangles. Fig. 8.3 shows an example data point used to train a network.

The expectation is that the accuracy of each model will increase with more samples and the accuracy of G-CNN approaches to be higher in different rotations.

8.2.1 Results

Fig. 8.4 gives the accuracy results for various sample sizes. “90 Rot” indicates a horizontal rectangle, while “Random Rot” indicates a randomly rotated rectangle. Note that we tested 10 separate models of 20 and 75 samples for the standard CNN due to the unexpected results in rotated accuracies for these models.

Let us first consider how sample numbers affect the accuracy of each model. As expected, the more samples a model trains on, the higher its accuracy. Looking at Fig. 8.5 and Fig. 8.6, one can deduce that each model learns to pinpoint the location of the rectangles better given more samples. The only exception to this rule seems to

be the standard CNN application of EAM with rotated rectangles. The standard CNN EAM performs better with 20 samples than 75, given an unfamiliar rotation (expect for 90 degree rotations.)

The standard CNN's reconstructions (even at higher sample rates) generate a blob rather than a rectangle while both C_8 and D_4 generate position suggestions similar to the shape of a rectangle. This is likely due to their inherent ability to learn rotations, further discussed in Sec. 8.4

EAM models perform worse when rotation is introduced. The decrease in accuracies is solved by increasing sample rates. However, G-CNNs perform marginally better in environments with unseen rotations than the standard CNN. One can expect this considering the rotational equivariant properties of G-CNN networks. The G-CNN on the C_8 group performs better than the G-CNN in the D_4 group when the rectangle is randomly rotated as C_8 is equivariant in 45^o rather than D_4's 45^o.

What needs to be noted is the high accuracy at low sample rates in comparison to results presented in Sec. 6.1.1. The accuracy in both of the experiments are calculated in the same way, however, the EAM approach that leverage Robot Control requires less samples to achieve high accuracy while the end-effector model requires a lot more samples and multiple visual inputs regarding the position of the gripper over the length of the trajectory.

8.3 Translation Test

This experiment tests the translational equivariance properties of EAMs. We only generate expert samples for 25% of the possible workspace but test to see if the model can predict locations in the entire action space.

As the focus is on translational equivariance of EAMs, we use a cube and rectangle for the target object with 75 unique expert demonstrations and only one of the G-CNN groups (in this case, we went with D_4.) Fig. 8.7 demonstrates the experimental setup.

Both CNN and G-CNN are inherently translationally equivariant. So, we expect that both models will be able to construct maps for cubes in unseen locations. By ignoring rotations and more complex shapes, we can focus purely on the translational equivariance aspect of EAMs.

8.3.1 Results

Fig. 8.8 provides examples of accurate and inaccurate results generated by the translational equivariance test. Table 8.1 indicates the accuracy of each model during the test.

As the test runs on the entire workspace of the robot, if translational equivariance is not working, we would expect the final output to have an accuracy of 25% as models only train on objects in that part of the model. However, the model accuracies are

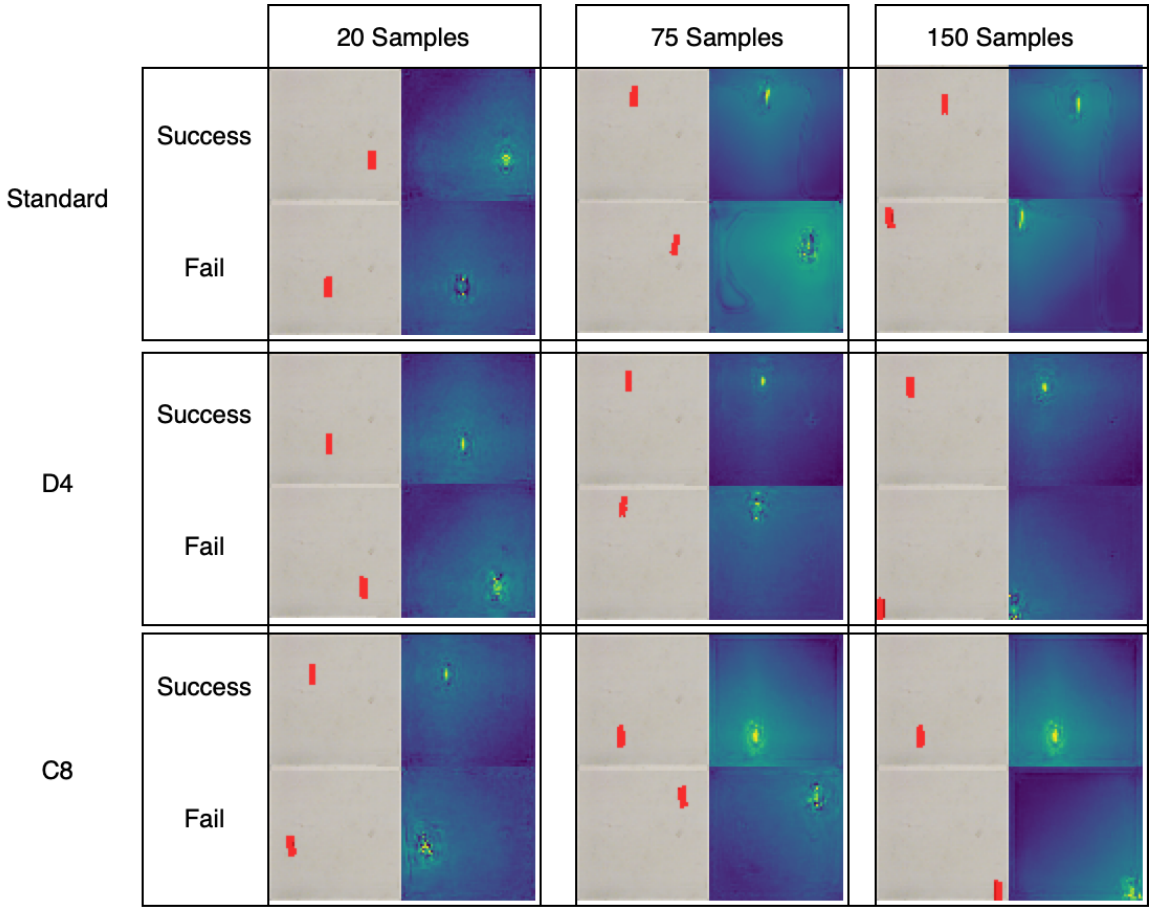

Figure 8.5: Examples of generated EAM maps from respective UNets trained on different sample rates without rotated rectangles. Tested on unrotated rectangles.

Model Accuracies			
CNN Square	CNN Rectangle	G-CNN Square	G-CNN Rectangle
97%	92%	96.5%	90%

Table 8.1: Table of EAM model accuracies for the translation test. Training data only includes targets within the lower 20% of the workspace as indicated in Fig.8.7. Testing uses the whole workspace for targets.

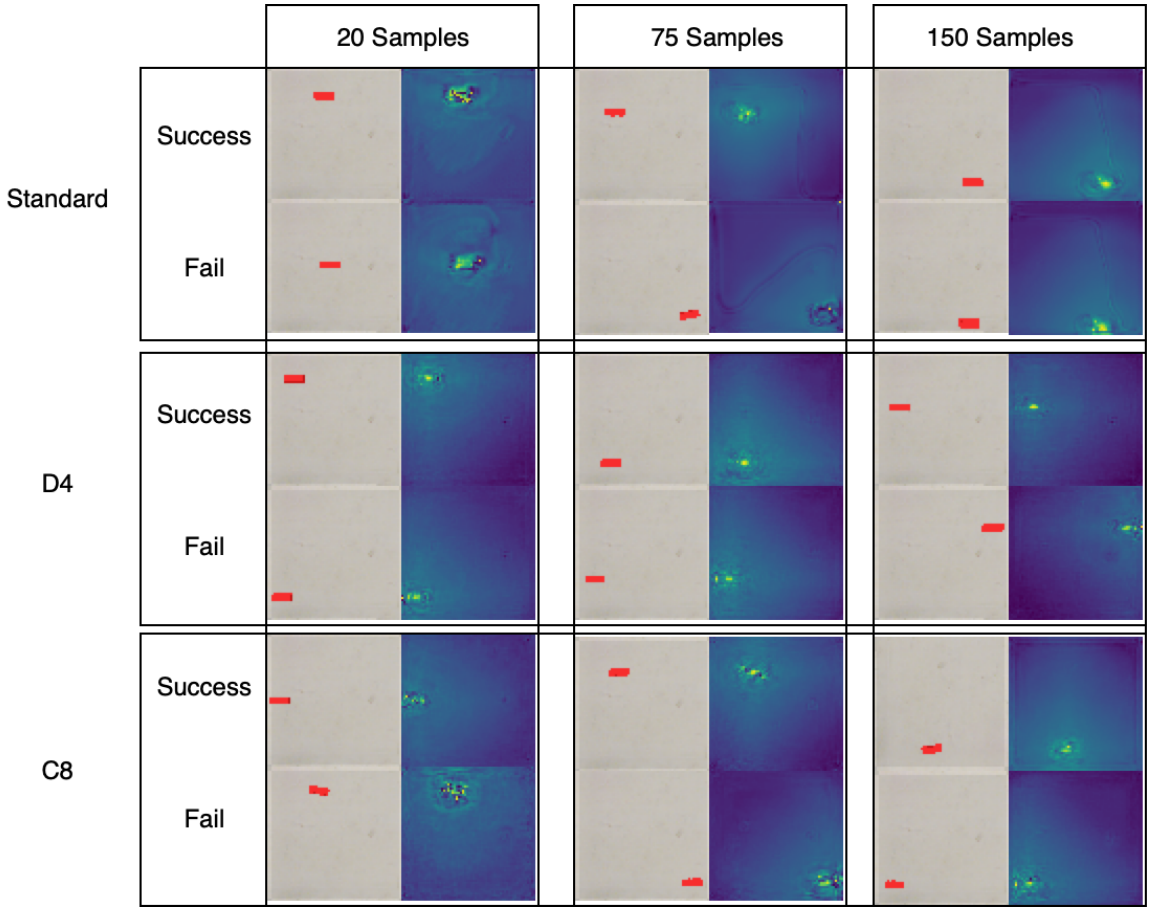

Figure 8.6: Examples of generated EAM maps from respective UNets trained on different sample rates without rotated rectangles. Tested on rectangles rotated by 90 degrees.

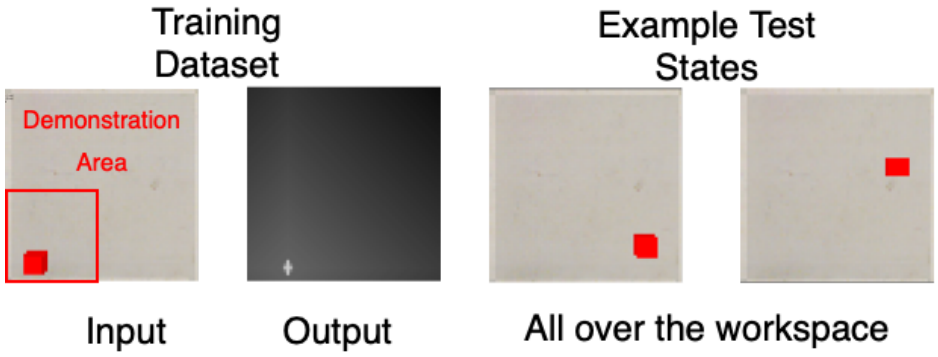

Figure 8.7: Diagram of translational equivariance test setup.

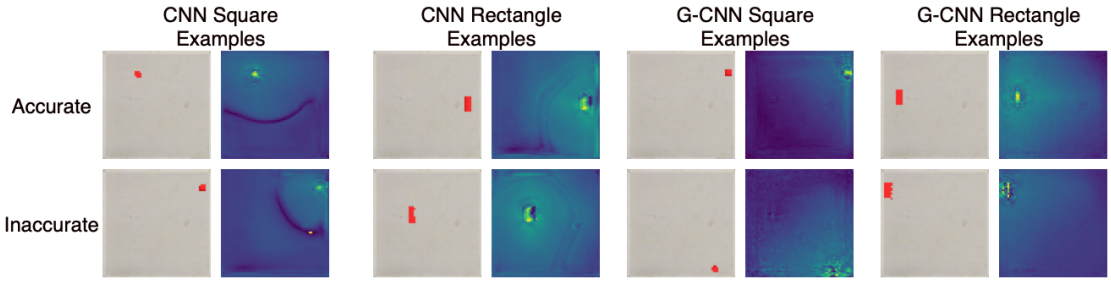

Figure 8.8: Diagram of output EAM examples from the Translation Equviariance test. Training data only includes targets within the lower 20% of the workspace as indicated in Fig.8.7.

over 90%, indicating that the translational equivariance properties of both CNNs work well in both the rectangle and the square shapes.

We can further confirm the results by looking at Fig. 8.8 as it provides map examples from each model. For squares, both the CNN and G-CNN generate small dots on the output EAM. We can see that issues occur when a single pixel value overshoots due to broken gradient estimations or if the cube is at an edge. For rectangles, CNNs seem to fail due to the large number of pixels used to represent the rectangle in EAM, while on the G-CNN, failures occur due to the low resolution of the camera changing the shape of the rectangle. These are specific cases as most of the position predictions are accurate.

Examining the rectangle maps reveals that G-CNN picks fewer pixel values along the shape of the rectangle, while CNN uses similar and thicker pixel outputs for rectangles.

8.4 Rotation Test

The fundamental difference between G-CNN and a standard CNN is Group Equivariance, as G-CNNs can be rotationally and flip equivariant based on their group G. While this test will not try to orient the end effector to match the orientation of the target rectangle, it will explore the results of how a standard, D_4, and C_8 network will map rotated rectangles in an EAM. The setup examines the results found during testing in Sec. 8.2. Note that, during training, rectangles are not rotated and are only aligned vertically. Refer to Fig. 8.3 to examine the setup.

This test will specifically examine the output maps of the network. As a reminder, standard CNN is not rotationally equivariant, D_4 is equivariant in flips and 90-degree rotations, and C_8 is equivariant in 45-degree rotations.

8.4.1 Results

To compare rotational equivariance, we shall first compare 90^o illustrated in Fig. 8.6 as both D_4 and C_8 are equivariant for 90^o rotations. Then, we will examine Fig. 8.9

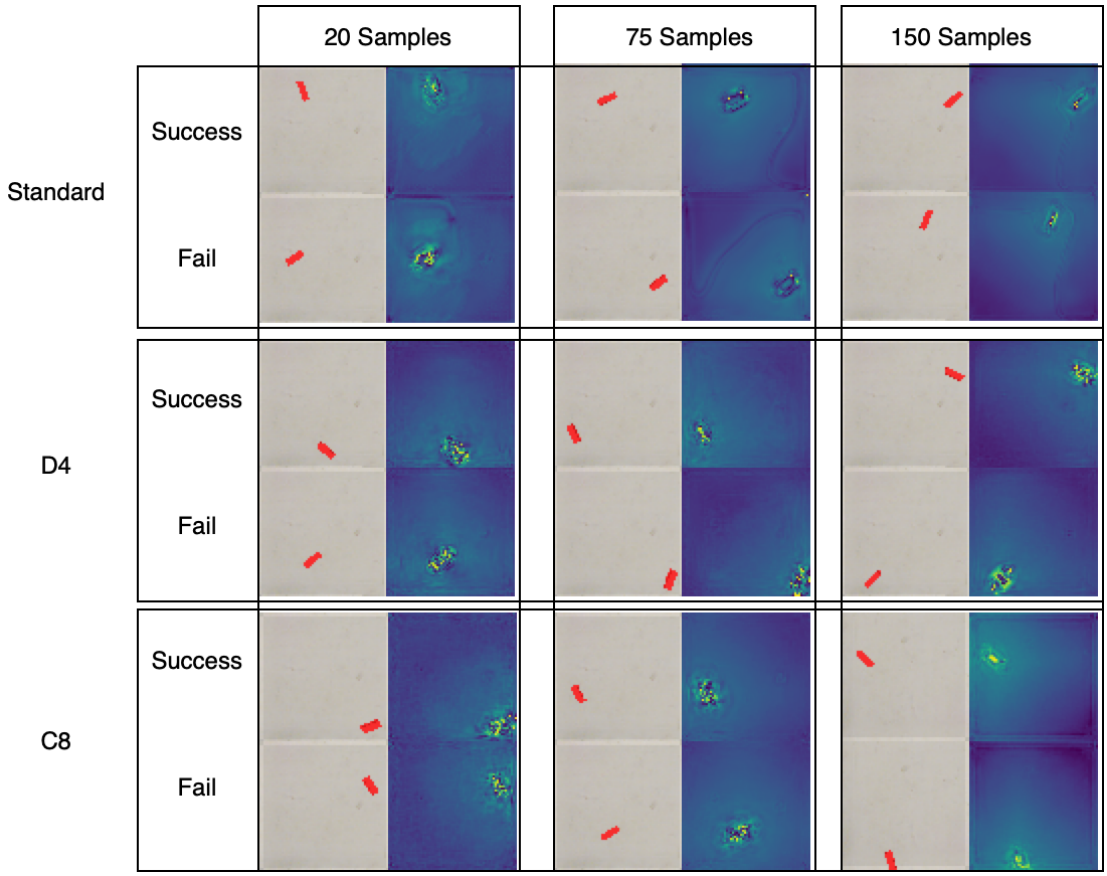

Figure 8.9: Examples of generated EAM maps from respective UNets trained on different sample rates without rotated rectangles. Tested on rectangles that are randomly rotated.

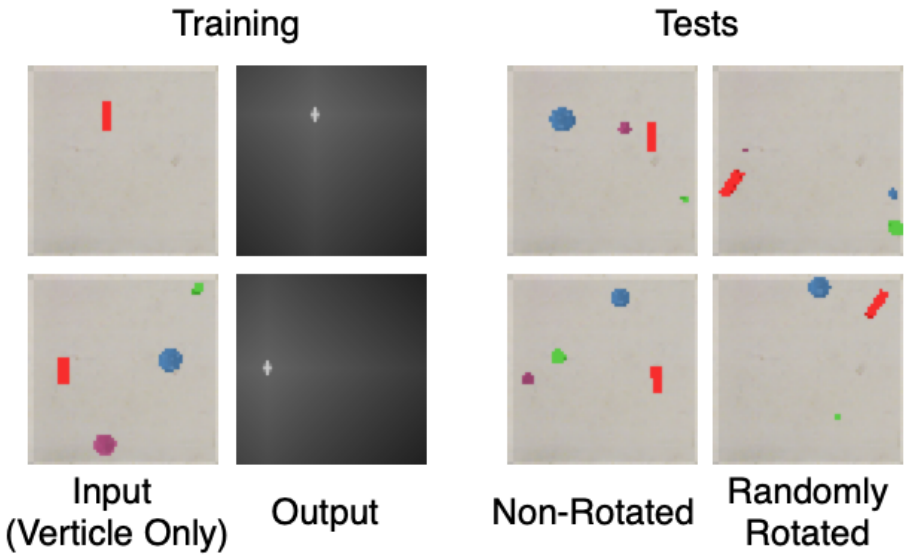

Figure 8.10: Diagram illustrating the Distractor Test's Setup

to determine if the C_8's rotational equivariance performs better than D_4's for smaller angles.

Fig. 8.6 reveals that the standard CNN's results generate blobs no matter the sample rate. In Fig. 8.5, when enough samples are provided, the standard UNet creates EAMs with visible rectangles - however, this does not seem to be the case at a 90^o angle. D_4 UNet performs similarly, even on failed examples, with various sample rates across the board. The output results are practically rotated versions of the results in Fig. 8.5, indicating the flip and 90-degree rotational kernels of D_4 work with EAMs, demonstrating the equivariant properties of G-CNNs and EAM representation. C_8 generates maps that seem to be between 90 and 45 degrees at low sample rates but improve the results as the number of samples increases.

Fig. 8.9 demonstrates how increasing the rotational complexity of the G-CNN improves adaptability. While D_4 fails to generate a rectangular position map for random rotations, C_8 can do so, even on failed attempts. Due to C_8's successfully generated maps compared to the mappings of both standard and C_8 UNets, the test demonstrates the rotational equivariant properties of G-CNNs and the EAM representation.

8.5 Distractors Test

This test is generated to test the robustness of the EAM approach when there are distractors or other objects present in an environment. This test aims to see if EAMs can be applied to more complex applications. For example, if we wish to train a robot using EAMs to stack objects, it would need to learn to ignore items based on the current state. While this experiment does not train on various objects, it trains to see if distractors affect the results of the EAM approach.

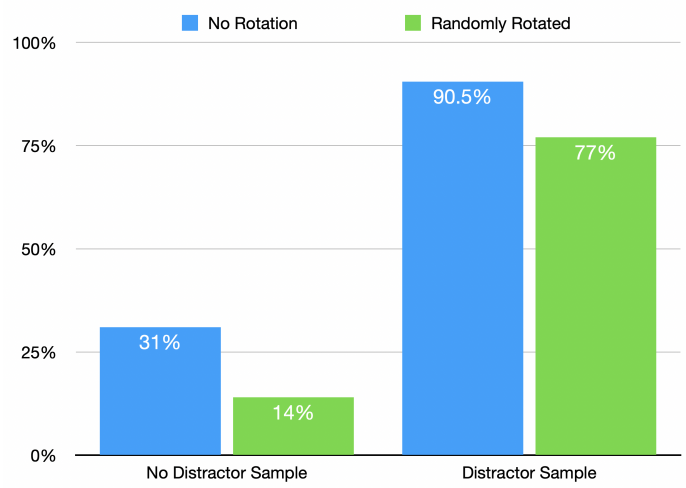

Figure 8.11: Diagram demonstrating the accuracies of a C_8-based EAM model tested with distractors in the workspace. Two models are trained at 150 samples where one of set of samples include random distractors within the workspace.

The model used for this experiment is the C_8 model, as it is the most accurate one in the tests so far and generates the clearest maps. The model will be trained on two separate 150 expert demonstrations. One set will be plain without any distractors, while the other set will have distractors of various sizes and locations present. The testing will only be done with distractors with no rotations and random rotations. Fig. 8.10 demonstrates the experimental setup.

8.5.1 Results

Fig. 8.11 reveals that the EAM model performs significantly worse if used in an environment with distractors if none were present during training. To see why this may be the case, we examine Fig. 8.12: It looks like the model can somewhat determine where the rectangle is. However, the unseen objects cause brighter pixels than the rectangle, likely due to unseen values on the kernels, which estimates an incorrect grasping position.

However, once distractors are included in training samples, the accuracy of the EAM model rises significantly. The results are not as good as the ones seen in Sec. 8.2, but they are relatively high. Examining maps generated in Fig. 8.12 reveals that training with distractors pushes the network to lower the pixel values of distractors. Decreasing pixel weights of unseen shapes seems to affect the mapping around the target when rotated, providing lower pixel weights for unseen positions.

8.6 Overall Results

While the chapter discusses four separate experiments, we used several more experiments during model generation. However, the listed four summarise the results. For

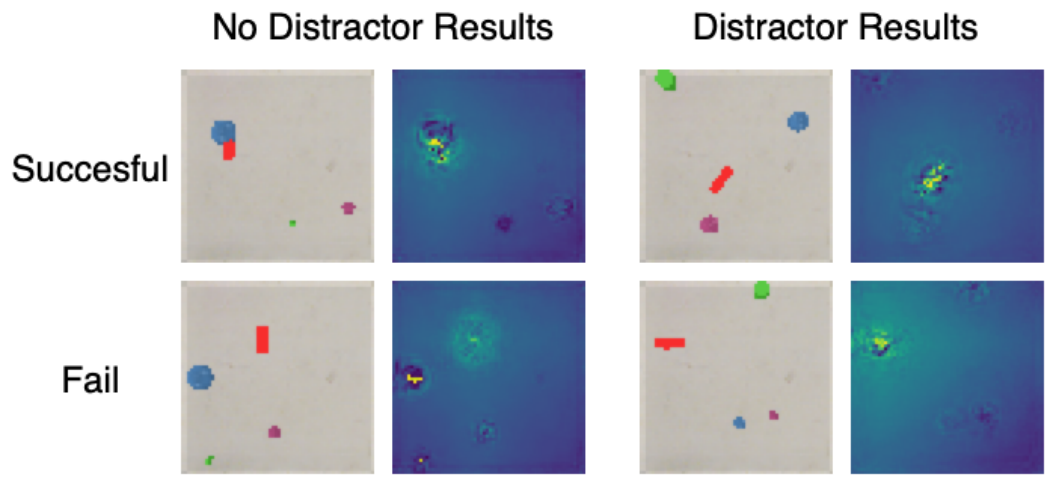

Figure 8.12: Examples of generated EAM maps for the Distractor Experiment. The map generating UNet, trained with 150 unrotated rectangle samples, is built with G-CNNs of C_8 Symmetric Group.

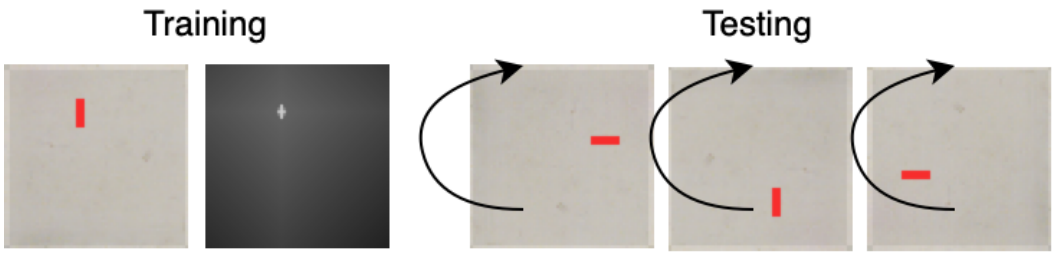

Figure 8.13: Diagram displaying the setup for the workspace rotation experiment.

example, one of the unexplained experiments had to do with rotating the camera on the gripper at 90-degree intervals, even though the model trained only on a single orientation. This experiment aimed to test the rotational equivariance of EAMs when the whole state space is rotated. However, due to Assumption 1, we need to rotate the pixel-to-position calculation (Alg. 1) to align with the new state space. So, the experiment became similar to the one in Sec. 8.4 where only target rotations mattered.

The experiments in Sec. 8.3 and Sec. 8.4 prove the equivariant properties of the EAM action representation. The results indicate that EAMs are useful in equivariant grasping tasks - and given an austere environment, the accuracy of EAM-based models is quite high. Depending on the application, training a standard CNN UNet to generate EAM representations is likely to work quite well, but G-CNN UNets perform better with more complex objects. The symmetric space C_8 is likely to be preferred to symmetric space D_4 in terms of accuracy and equivariant properties as well, considering the results in Fig. 8.9 and Fig. 8.4.

However, the reliance on inverse kinematics (need knowledge of the robot), the lack of rotation and height outputs, and the stringent requirements for workspace alignment are considerable downsides of the EAM approach. In future works (Sec.9.2), we describe two methods we started implementing to add rotations to the EAM grasping approach but would not be completed by the deadline of the paper.

Overall, the EAM action representation successfully allows leveraging equivariant properties in a robot grasping task. Results in Sec. 8.2 in comparison to the EE model's results in Sec. 6.1.1 reveal that EAMs require fewer samples to be highly accurate. Furthermore, comparing Sec. 8.3 to 6.3 reveals that, in comparison to traditional end-effector models, EAM representations can infer object locations in unfamiliar positions and rotations.

Chapter 9

Conclusion

Aiming to leverage the symmetries and equivariant properties present in the robot grasping task, we initially propose a simple end-effector velocity prediction method built with G-CNNs. While the method performs relatively well in simple tasks, it has issues stopping and orienting the end-effector for a simple cube. However, further tests into the equivariant properties of the model reveal that the model fails to generalize outputs for unseen inputs in symmetric spaces, even though the network ought to be equivariant. Due to the failure of the initial approach, we further explore how the equivariant properties of the Robot Grasping task can be leveraged for an action representation. This leads us to introduce Equviariant Action Maps, a position prediction action representation that maps positions on the workspace into a 2D matrix.

While the idea of utilizing spatial action maps is not new, EAM, to the best of our knowledge, is the first spatial action representation used in a behavioral cloning setting while leveraging equivariant properties. By using aligned mappings of positions based on the input visuals, EAM transforms the same way the visual input does. Furthermore, as EAM maps positions into a map, a human is able to understand the results easier than results that may be purely numeric.

The paper details experiments aimed at determining the accuracy, robustness, and equivariant properties of EAM. These experiments reveal that EAM representation is very accurate in simple grasping tasks, works when various distractors are present, and are equivariant in $SE(2)$ transformations (given a G-CNN with a symmetric group $G \in SE(2)$ is used). These results are likely due to combining traditional Robot Control for approach accuracy and Robot Learning to increase adaptability.

However, EAM representations are not close to perfect. They require the usage of inverse kinematics (meaning one needs knowledge of the robot) to move a gripper to the final target location, has stringent alignment requirements to be equivariant with the input, and can not tackle rotational or height-based positional predictions. Despite these setbacks, utilizing EAMs with an FCN provides highly accurate results thanks to Inverse Kinematics for movement calculations.

The paper does not explore whether EAM representation works with more complex tasks. So further experimentation and improvements are needed to grasp the extent of EAM representation's application. We discuss some further improvements in Sec. 9.2.

9.1 Ethical Considerations

The model-related data collection, training, and testing were done in an offline simulation environment. So, no external sources influenced the results of the project other than possible inaccuracies that may be caused by Single Event Upsets or hardware. Multiple training, testing, and data collection sessions reduce the outliers that these inaccuracies may cause. As the project was done in a simulation environment, nobody was in a situation where they could be harmed.

There has been no usage of human data in the project. Inverse Kinematics generates demonstrations in the simulation environment. Hence, there are no privacy concerns present.

CoppeliaSim, the simulation environment used in this project, allows all usage other than commercial usage under our educational license. PyRep and other referenced libraries are under the MIT license, indicating that they are open-source. Due to this, the project has no copyright or related legal issues.

There is an overarching concern of AI being susceptible to misuse. Specifically, using AI in the military or other applications may harm human lives. EAM is just a simple action representation applied to robot grasping tasks. Considering the current state of military technology, we argue that the findings of this paper are unlikely to improve militaristic technology. However, further research into AI technology may lead to a superintelligence that can take detrimental actions. While this report does not slow the approach to this concept, it only provides an honest and basic action representation that is unlikely to be used by a superintelligence.

If EAMs are utilized in a real-world environment, inaccuracies in positional estimation may cause issues. However, experiments with distractors have shown that the model is capable of learning to avoid distractors within the workspace. There are no discriminatory concerns as the model does not directly work with humans or objects.

The model provides an action representation that is more accessible to the public. Rather than providing joint torques or other numerical results, EAMs indicate positions on a map that can be easily understood and compared to the visual input. EAM maps allow people not knowledgeable in robotics, computer science, or AI to interpret the data.

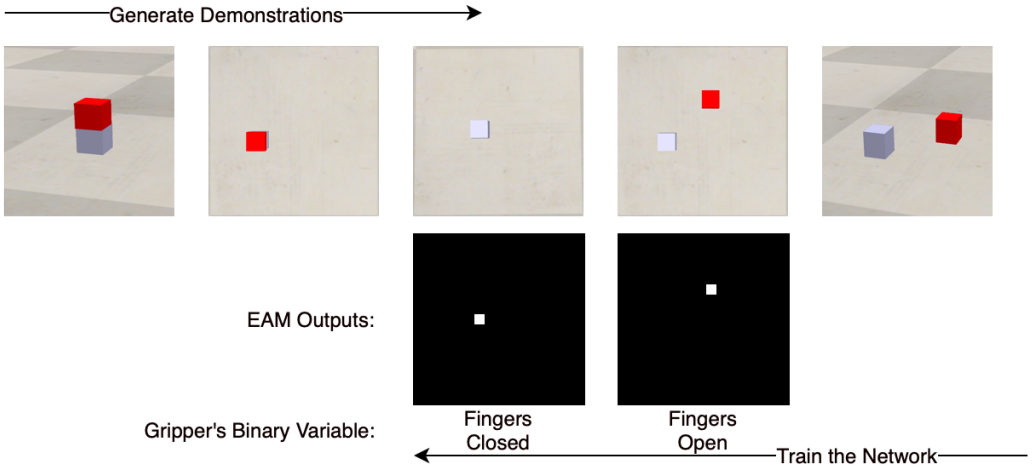

Figure 9.1: Diagram of using EAMs to train a cube stacking task. Generate expert demonstrations in reverse by placing cubes in random locations. Use a binary variable of whether the gripper fingers are closed to determine the next output of the EAM.

9.2 Future Works

Finally, we will consider means of improving the EAM approach. As mentioned multiple times during the paper, the EAM approach is not perfect and only works under stringent requirements. This chapter will discuss three main shortcomings of the EAM representation that may be fixed: EAM's application in more complex grasping problems, EAM's lack of height adjustment, and EAM's lack of rotational adjustment.

While describing the EAM and running various experiments, the focus was purely on a simple grasping task for a simple object. One can argue that the preliminary models and EAMs are over-engineering a solution for a problem that did not exist. However, as EAM is purely an action representation, there are ways it can be stacked to perform more complex actions. For example, suppose we wish to build a small structure using cubes and rectangles. In that case, we can generate demonstrations by starting with a built structure and pulling it apart by placing objects in random locations within the workspace. We collect the relevant workspace snapshot once an object is within the grippers and generate an EAM on the map where the end-effector was while grasping the object. Then, we feed the maps in reverse to teach a network to build the same structure. There is a lack of height input, which we can solve by using a predefined height at each step of the building process. While there is likely a better way of tackling this, we thought this was an initial step in tackling more complex problems. Fig. 9.1 provides a demonstration of how this experiment may be set-up. However, note that EAMs still only work with simple grasping tasks in this case. If a complex problem can not be simplified into multiple grasping tasks, then the EAM representation may not work.

Tackling the height component missing in EAMs is a more complex issue but will make EAMs more robust. There are two ways this may be tackled. First is using

3D G-CNNs [52] with a 3D representation of the environment. However, this will make the EAM model even more complex, requiring generating 3D maps. Another is switching to a depth sensor and including a secondary network. Rather than giving EAM a height component, another network may train to learn the relative height values of positions. If the EAM provides a position, however, we can purely utilize Robot Control with the depth sensor to adjust the final height position of the inverse kinematics movement.

Finally, we can tackle the missing rotational component of EAMs. Rather than trying to add a rotational component to EAMs, we can use a separate representation and network to determine the final rotation of the end-effector in order to grasp an object. In the last leg of the project, we started building this concept due to its relevance to rotational equivariance but did not have enough time. The concept is similar to ASR [43], where we utilize EAM to propose a position. Then, we use the pixel value of that position to crop the input around the target object. A separate network inputs the cropped image and determines the object's rotation using rotational bins. If the model utilizes eight rotational bins, we divide 360 degrees into eight equal segments, and the object's rotation is placed within the appropriate bin. A CNN or G-CNN network (without a linear layer) trains on the cropped image to generate a suitable rotational bin afterward. This approach only had around 30-35% accuracy in matching bins. To improve the accuracy and equivariant properties, we can adopt a binning strategy similar to Hara et al. [53] and Chalvatzaki et al. [54] where the sin and cos values of target rotation θ are calculated, then placed on a euclidian map of size -1 to 1. Then, the rotation network can learn to generate a similar map or bins based on sin and cos values, which can be reverted to the original rotation using arctan. Fig. 9.2 demonstrates the application of how rotations can be implemented into the EAM action representation.

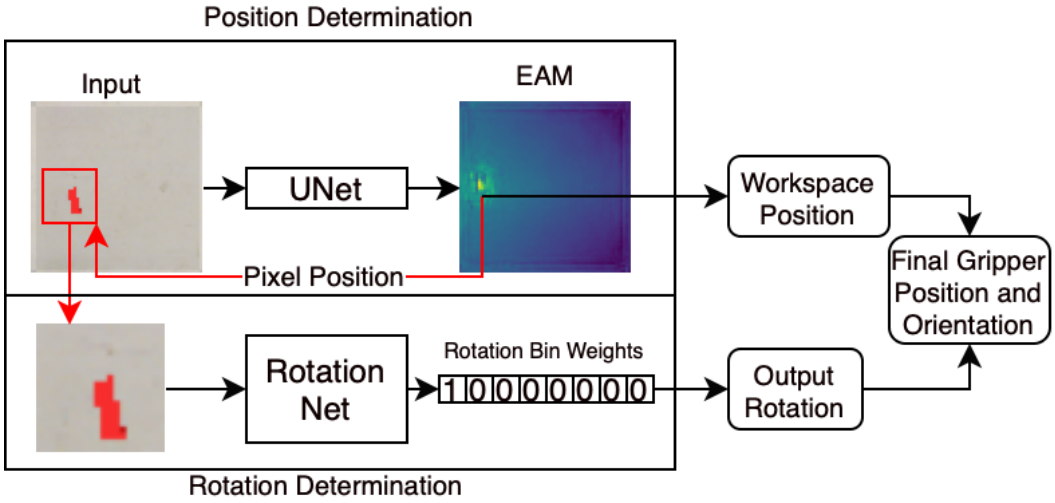

Figure 9.2: Diagram of the proposed rotation approach. The input is initially fed into the UNet to generate an EAM. Using the pixel position from the EAM, the original input is cropped and fed into the rotation net. The EAM output is used to generate the final workspace position of the end-effector while rotation net output provides the final output rotation of the end-effector.

Bibliography

[1] Yasuhiko Hashimoto. Half a century of kawasaki robotics: The kawasaki robot story, 2018. pages 1

[2] International Federation of Robotics. Ifr presents world robotics 2021 reports, 10 2021. pages 1

[3] Jonathan Tilley. Automation, robotics, and the factory of the future — mckinsey, 9 2017. URL https://www.mckinsey.com/business-functions/operations/our-insights/automation-robotics-and-the-factory-of-the-future. pages 1

[4] Mordechai Ben-Ari and Francesco Mondada. Robots and their applications. *Elements of Robotics*, pages 1–20, 2018. doi: 10.1007/978-3-319-62533-1_1. URL https://link.springer.com/chapter/10.1007/978-3-319-62533-1_1. pages 1

[5] Franka Emika. Panda user handbook. 2018. pages 2

[6] Sulabh Kumra and Christopher Kanan. Robotic grasp detection using deep convolutional neural networks. *IEEE International Conference on Intelligent Robots and Systems*, 2017-September:769–776, 11 2016. ISSN 21530866. doi: 10.48550/arxiv.1611.08036. URL https://arxiv.org/abs/1611.08036v4. pages 2

[7] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: Part i. *IEEE Robotics and Automation Magazine*, 13:99–108, 6 2006. ISSN 10709932. doi: 10.1109/MRA.2006.1638022. pages 2

[8] Ross Hartley, Maani Ghaffari, Ryan M. Eustice, and Jessy W. Grizzle. Contact-aided invariant extended kalman filtering for robot state estimation. *International Journal of Robotics Research*, 39:402–430, 4 2019. ISSN 17413176. doi: 10.48550/arxiv.1904.09251. URL https://arxiv.org/abs/1904.09251v2. pages 2

[9] Maria Vittoria Minniti, Ruben Grandia, Kevin Fäh, Farbod Farshidian, and Marco Hutter. Model predictive robot-environment interaction control for mobile manipulation tasks. *Proceedings - IEEE International Conference on Robotics and Automation*, 2021-May:1651–1657, 6 2021. ISSN 10504729. doi: 10.

48550/arxiv.2106.04202. URL https://arxiv.org/abs/2106.04202v1. pages 2

[10] Granta. Advantages and disadvantages of robotic automation — granta automation, 2017. URL https://www.granta-automation.co.uk/news/advantages-and-disadvantages-of-robotic-automation/. pages 2

[11] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *Robotics: Science and Systems*, 9 2017. ISSN 2330765X. doi: 10.48550/arxiv.1709.10087. URL https://arxiv.org/abs/1709.10087v2. pages 2

[12] Ajith Thomas and John Hedley. Fumebot: A deep convolutional neural network controlled robot. *Robotics*, 8, 2019. ISSN 22186581. doi: 10.3390/robotics8030062. pages 2

[13] Stephen James, Andrew J. Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. *PMLR*, pages 334–343, 10 2017. URL http://arxiv.org/abs/1707.02267. pages 2, 15, 16, 27, 29

[14] Coline Devin, Pieter Abbeel, Trevor Darrell, and Sergey Levine. Deep object-centric representations for generalizable robot learning. 2018. doi: 10.1109/ICRA.2018.8461196. pages 2, 17, 18

[15] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. 6 2018. doi: 10.48550/arxiv.1806.10293. URL https://arxiv.org/abs/1806.10293v3. pages 2

[16] Edward Johns. Coarse-to-fine imitation learning: Robot manipulation from a single demonstration. *Proceedings - IEEE International Conference on Robotics and Automation*, 2021-May:4613–4619, 5 2021. ISSN 10504729. doi: 10.48550/arxiv.2105.06411. URL https://arxiv.org/abs/2105.06411v2. pages 2

[17] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Cliport: What and where pathways for robotic manipulation. 9 2021. doi: 10.48550/arxiv.2109.12098. URL https://arxiv.org/abs/2109.12098v1. pages 2

[18] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *NIPS 2016 Tutorial*, 3 2016. ISSN 15267598. doi: 10.48550/arxiv.1603.02199. URL https://arxiv.org/abs/1603.02199v4. pages 2

[19] Taco S. Cohen and Max Welling. Group equivariant convolutional networks. *33rd International Conference on Machine Learning, ICML 2016*, 6:4375–4386,

2 2016. doi: 10.48550/arxiv.1602.07576. URL https://arxiv.org/abs/1602.07576v3. pages 3, 20, 25

[20] Jan E. Gerken, Jimmy Aronsson, Oscar Carlsson, Hampus Linander, Fredrik Ohlsson, Christoffer Petersson, and Daniel Persson. Geometric deep learning and equivariant neural networks. 5 2021. pages 3, 20, 47

[21] Anthony Simeonov, Yilun Du, Andrea Tagliasacchi, Joshua B. Tenenbaum, Alberto Rodriguez, Pulkit Agrawal, and Vincent Sitzmann. Neural descriptor fields: Se(3)-equivariant object representations for manipulation. 12 2021. URL http://arxiv.org/abs/2112.05124. pages 3

[22] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. 2000. pages 6, 7

[23] M Bain and C Sammut. A {Framework for Behavioural Cloning}. *Machine Intelligence*, 15, 1999. pages 8

[24] Michael Kelly, Chelsea Sidrane, Katherine Driggs-Campbell, and Mykel J. Kochenderfer. Hg-dagger: Interactive imitation learning with human experts. volume 2019-May, 2019. doi: 10.1109/ICRA.2019.8793698. pages 9

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521: 436–444, 5 2015. ISSN 0028-0836. doi: 10.1038/nature14539. pages 9

[26] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, R. Howard, Wayne Hubbard, and Lawrance Jackel. Handwritten digit recognition with a back-propagation network. *Advances in Neural Information Processing System 2*, 1989. pages 11

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012 alexnet. *Advances In Neural Information Processing Systems*, 2012. ISSN 10495258. pages 11

[28] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. 5 2015. pages 11, 12, 42, 44

[29] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:640–651, 11 2014. ISSN 01628828. doi: 10.48550/arxiv.1411.4038. URL https://arxiv.org/abs/1411.4038v2. pages 11

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 1. MIT Press, 1 edition, 2016. pages 12, 20

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997. ISSN 08997667. doi: 10.1162/NECO.1997.9.8.1735. URL https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735. pages 13

[32] Christopher Olah. Understanding lstm networks – colah's blog, 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/. pages 13

[33] A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir. Inverse kinematics techniques in computer graphics: A survey. *Computer Graphics Forum*, 37, 2018. ISSN 14678659. doi: 10.1111/cgf.13310. pages 14

[34] Tianhao Zhang, Zoe McCarthy, Owen Jowl, Dennis Lee, Xi Chen, Ken Goldberg, and Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. 2018. doi: 10.1109/ICRA.2018.8461249. pages 15, 16

[35] Eduardo Godinho Ribeiro, Raul de Queiroz Mendes, and Valdir Grassi. Real-time deep learning approach to visual servo control and grasp detection for autonomous robotic manipulation. *Robotics and Autonomous Systems*, 139, 2021. ISSN 09218890. doi: 10.1016/j.robot.2021.103757. pages 16, 17, 18

[36] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. volume 2016-June, 2016. doi: 10.1109/ICRA.2016.7487173. pages 16, 17

[37] Mohi Khansari, Daniel Kappler, Jianlan Luo, Jeff Bingham, and Mrinal Kalakrishnan. Action image representation: Learning scalable deep grasping policies with zero real world data. 2020. doi: 10.1109/ICRA40945.2020.9197415. pages 17, 18

[38] Jessica Borja-Diaz, Oier Mees, Gabriel Kalweit, Lukas Hermann, Joschka Boedecker, and Wolfram Burgard. Affordance learning from play for sample-efficient policy learning. *2022 International Conference on Robotics and Automation (ICRA)*, pages 6372–6378, 5 2022. doi: 10.1109/ICRA46639.2022.9811889. URL https://ieeexplore.ieee.org/document/9811889/. pages 17, 18

[39] James J. Gibson. The ecological approach to visual perception. *The Ecological Approach to Visual Perception*, 11 2014. doi: 10.4324/9781315740218. URL https://www.taylorfrancis.com/books/mono/10.4324/9781315740218/ecological-approach-visual-perception-james-gibson. pages 18

[40] Changjae Oh and Andrea Cavallaro. View-action representation learning for active first-person vision. *IEEE Transactions on Circuits and Systems for Video Technology*, 31, 2021. ISSN 15582205. doi: 10.1109/TCSVT.2020.2987562. pages 18, 29

[41] Fabio Pardo, Vitaly Levdik, and Petar Kormushev. Scaling all-goals updates in reinforcement learning using convolutional neural networks. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*, pages 5355–5362, 10 2018. ISSN 2159-5399. doi: 10.48550/arxiv.1810.02927. URL https://arxiv.org/abs/1810.02927v2. pages 18, 19, 38, 39

[42] Jimmy Wu, Xingyuan Sun, Andy Zeng, Shuran Song, Johnny Lee, Szymon Rusinkiewicz, and Thomas Funkhouser. Spatial action maps for mobile manipulation. *Robotics: Science and Systems*, 4 2020. doi: 10.15607/RSS.2020.XVI.035. URL http://arxiv.org/abs/2004.09141http://dx.doi.org/10.15607/RSS.2020.XVI.035. pages 19, 38, 39, 43, 44

[43] Dian Wang, Colin Kohler, and Robert Platt. Policy learning in se(3) action spaces. 10 2020. doi: 10.48550/arxiv.2010.02798. URL https://arxiv.org/abs/2010.02798v2. pages 19, 44, 61

[44] Xupeng Zhu, Dian Wang, Ondrej Biza, Guanang Su, Robin Walters, and Robert Platt. Sample efficient grasp learning using equivariant models. 2 2022. doi: 10.48550/arxiv.2202.09468. URL https://arxiv.org/abs/2202.09468v1. pages 19, 25, 47

[45] Dian Wang, Robin Walters, Xupeng Zhu, and Robert Platt. Equivariant q learning in spatial action spaces. 10 2021. doi: 10.48550/arxiv.2110.15443. URL https://arxiv.org/abs/2110.15443v1. pages 19, 31

[46] Maurice Weiler and Gabriele Cesa. General $e(2)$-equivariant steerable cnns. *Advances in Neural Information Processing Systems*, 32, 11 2019. ISSN 10495258. doi: 10.48550/arxiv.1911.08251. URL https://arxiv.org/abs/1911.08251v2. pages 19, 25, 29, 34, 36

[47] Taco S. Cohen, Maurice Weiler, Berkay Kicanaoglu, and Max Welling. Gauge equivariant convolutional networks and the icosahedral cnn. *36th International Conference on Machine Learning, ICML 2019*, 2019-June:2357–2371, 2 2019. doi: 10.48550/arxiv.1902.04615. URL https://arxiv.org/abs/1902.04615v3. pages 20

[48] Dian Wang, Robin Walters, and Robert Platt. $\{SO\}(2)$-equivariant reinforcement learning. 3 2022. doi: 10.48550/arxiv.2203.04439. URL https://arxiv.org/abs/2203.04439v1. pages 26

[49] Eric Rohmer, Surya P. N. Singh, and Marc Freese. Coppeliasim (formerly v-rep): A versatile and scalable robot simulation framework, 2013. URL https://www.coppeliarobotics.com/coppeliaSim_v-rep_iros2013.pdf. pages 27, 46

[50] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing v-rep to deep robot learning. 6 2019. doi: 10.48550/arxiv.1906.11176. URL https://arxiv.org/abs/1906.11176v1. pages 27, 46

[51] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for image restoration with neural networks. 2018. URL https://github.com/NVlabs/PL4NN. pages 45

[52] Marysia Winkels and Taco S. Cohen. 3d g-cnns for pulmonary nodule detection. 4 2018. doi: 10.48550/arxiv.1804.04656. URL https://arxiv.org/abs/1804.04656v1. pages 61

[53] Kota Hara, Raviteja Vemulapalli, and Rama Chellappa. Designing deep convolutional neural networks for continuous object orientation estimation. 2 2017. doi: 10.48550/arxiv.1702.01499. URL https://arxiv.org/abs/1702.01499v1. pages 61

[54] Georgia Chalvatzaki, Nikolaos Gkanatsios, Petros Maragos, and Jan Peters. Orientation attentive robotic grasp synthesis with augmented grasp map representation. 6 2020. doi: 10.48550/arxiv.2006.05123. URL https://arxiv.org/abs/2006.05123v2. pages 61