Approaches to Spatial Reasoning in Reinforcement Learning

A Project Report

submitted by

SAI PRAVEEN B

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **Approaches to Spatial Reasoning in Renforcement Learning**, submitted by **Sai Praveen B**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelors of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Balaraman Ravindran
Research Guide and Associate Professor
Dept. of Computer Science
IIT-Madras, 600 036

Place: Chennai

Date: 1st May 2017

ACKNOWLEDGEMENTS

I would like to acknowledge (and am deeply grateful for) the technical help and support provided by my mentor Prof. B. Ravindran and my fellow student (and collaborator) J.S Suhas. I would also like to thank Sahil Sharma for his valuable inputs on various issues related to this work.

Additionally, I would like to thank my mother for her unwavering emotional support.

ABSTRACT

KEYWORDS: Reinforcement Learning; Deep Learning; Representation Learning; Variational Autoencoders; Projection Networks; Q-networks; Video Prediction; Jacobian Exploration Bonus

In multi-task reinforcement learning problems, model-based approaches are often ignored over model-free approaches like Deep Q-Networks (*Mnih et. al*). However, in most cases, having the ability to build a robust model of environment can be very useful. We analyze this problem on two different levels: 2D Map-like environments (Fully-observable) and 3D Minecraft-like environments (Partially-observable). For the *fully-observable* scenario, we present a novel method to facilitate exploration in multi-task reinforcement learning using deep generative models. We supplement our method with a low dimensional energy model to learn the underlying MDP distribution and provide a resilient and adaptive exploration signal to the agent. To extend this concept to the more useful, *partially-observable* case, we present a new architecture that replaces convolutional networks as a more effective way to implicitly model 2D projections of 3D environments. We discuss the mechanisms that allow it to implicitly model both the 3D environment and it's dynamics. We evaluate our method on a new set of environments and provide an intuitive interpretation of our results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

For a long time, reinforcement learning didn't leave the domain of environments with small state spaces. In fact, for a long time, stateless machines, aka. multi-armed bandits, were the main topic of research.
Even with large state spaces, the function approximation algorithms before the age of neural networks were rather limited in flexibility. The state space had to be carefully observed and the parameters tuned with that context in mind. For good reason, tackling incredibly complex high dimensional state spaces like a racing game or the game of Go, was seemingly impossible at the time.
With the advent of the latest revision of neural networks, there came a wave of new algorithms incorporating neural networks for complex function approximation.
Neural networks could, with little or no tuning, fit incredibly complex functions like those involving dynamics simulations (3). Given that the most important part of model-free reinforcement learning is typically the Q-function (which is also complex and usually lacks global structure), the earliest applications used a neural network to approximate the Q-function in **Fitted Q-iteration** (17).

Then, with the next big wave in the machine learning industry, deep learning, applying it to Fitted Q-iteration seemed to be the obvious move. However, the fickleness of deep neural networks meant that, most of the time, the network diverged from the non-stationary target function.

Soon, a ground-breaking research paper by *DeepMind Labs*, titled *"Playing atari games with Deep Reinforcement Learning"* used a convolutional deep neural network to understand and play 8-bit *Atari* games by observing the screen pixels.
A barrage of modifications to the Deep Q-Network architecture followed, one of the most prominent of which is the paper titled *"Action Conditional Video Prediction"* which

is what the second part of this thesis is based on.

While Q-networks solved the problem of function approximation, the new problem was now about finding efficient ways of searching the extremely large state-spaces. Some of the best methods in this regard, were exploration bonuses (3) and a prioritized replay memory (16). This is where our thesis attempts to apply *model-based* methods to reinforcement learning. By attempting to fit a model to the dynamics of the environment, we demonstrate that in some domains, it is possible to significantly improve performance.

Since the architectures proposed here overwhelmingly deal with fitting a model to spatial environments, both fully-visible (*the agent knows it's location, like a 2D Map-based game similar to Pac-man*) and partially-visible (*the agent is unaware of it's actual location, like a 3D Minecraft game*), the model-based approach is referred to as *spatial reasoning*.

In this thesis, we try to evolve spatial reasoning approaches to *multi-task reinforcement learning*. So far, there has been extensive work on the problem of multi-task reinforcement learning, which is generally used to describe the simultaneous learning of multiple distinct environments/tasks. There are generally three types of reinforcement learning that involves multiple tasks:

1. Cases that require the simultaneous learning of multiple tasks where each task is clearly identified as a different task.
   For example, A neural network with multiple heads being trained to play SeaQuest, Breakout and Montezuma at the same time.

2. Cases that require the simultaneous learning of multiple tasks where the agent is unaware of which task it is solving currently.
   For example, there are two sets of possible rewards and transition probabilities. One of them is picked at random and the agent needs to solve it with no information about which one was picked.

3. Cases that require the simultaneous learning of one or more tasks and then require the same agent to solve another completely different task
   For example, an agent that has to learn to play Pong and SeaQuest, and then extend this knowledge to learn Space Invaders from scratch.

In the first part of this thesis, we focus on model-based methods to promote exploration in the anonymous multi-task domain (case 2). In particular, the domain used for comparison is the *Map-like environment* domain (definition below).

We attempt to extend the ideas presented in (2) by applying it to a larger state space: a set of grid-worlds, one of which is randomly chosen as the underlying environment.

Then, we extend this work by replacing the simplistic RBM model with a variational auto-encoder that can learn and generalize much more complex worlds.
Work done so far also does not focus on *problem identification*, i.e. the agent needs to explore not only states that maximize reward but also those that maximize information gain (about the rest of the state space). In certain worlds, there are certain *indicator states* that have low reward but, like a *signboard*, provide invaluable information that can lead the agent to following a faster path to the goal state.
We propose a method to find these special states and assign a bonus reward in order to make the agent visit these states more often.

The second part of this thesis explores spatial reasoning models for the *partially-observed* scenario.
For example, in a Minecraft game, if we define the state space as the position/orientation of the agent, the problem setting is essentially a POMDP, with the observation being a 2D projection of the 3D world. The internal state is invisible to the agent.

So far, there have been no significant model-based attempts to approach POMDPs where the observation is a projection of a *higher dimensional space* Most current approaches depend on the 2D features present in each scene. Each set of frames are treated as a set of independent pixels rather than as different views of the same 3D scene. In order to apply *spatial reasoning* to improve benchmarks on the worlds such as the I-world, we consider the original process of how the observations were generated:
In Computer Graphics, a method known as *"ray tracing"* is one of the most popular methods used to project a 3D scene onto a 2D plane. In keeping with this idea, the network proposed in this thesis uses a *memory unit* (a set of parameters that represent the color and shape of the objects in the scene) to model the scene and a neural network to model the interaction (*intersection unit*) between the agent's current position/orientation and the *memory unit*. The *Motivation* section explains the ideas behind the formulation of the ACPNN (Action-Conditional Projection Neural Network). We also present various modifications to the standard ACPNN framework and the effects on it's performance.

CHAPTER 2

RELATED WORK

Our Spatial Reasoning-based algorithm for 2D Fully-observable Map-like environments (*Multi-task*, 3.1.1) improves upon existing methods like (2) by proposing an exploration bonus to allow the agent to learn more optimal paths. There is extensive research in the field of exploration strategies for reducing uncertainty in the MDP. R_{max}, E^3 are examples of widely used exploration strategies. Bayesian Exploration Bonus assigns a pseudo reward to states calculated using frequency of state visitation. Thomson sampling samples an MDP from the posterior distribution computed using evidence(rewards and transition probabilities) that it obtains from trajectories. We follow a similar approach to sample MDPs. However, these algorithms assume there exists a single stationary MDP for each episode (which is the case in Single-Task RL or *STRL*). Our algorithm addresses the Multi-task RL (or *MTRL*) problem where each episode uses an MDP sampled from an arbitrary distribution on MDPs. Contrary to the STRL exploration strategies, our exploration bonus is designed to recognize states that have low rewards but are potentially useful for the agent to improve its certainty about the current MDP.
Recent advances in MTRL and Transfer Learning algorithms, like Value Iteration Networks(9) and Actor-Mimic Networks(21), attempt to identify common structure among tasks and generalize learning to new tasks with similar structure. In the context of MTRL and Transfer Learning on environments that give *image*-like observations, Value Iteration Networks (9) employ Recurrent Neural Networks for value iteration and learn kernel functions $f_R(x)$ and $f_P(x)$ to estimate the reward and transition probabilities for a state from its immediate surroundings. This has the effect of easily generalizing to new tasks which share the same MDP structure(R and P for a state can be determined using locality assumptions). Our work, unlike transfer learning algorithms, does not attempt to learn common structure across MDPs. Instead, we attempt to efficiently use a model-based algorithm to learn a distribution over the possible structure of the environment, and then use this model to boost the agent's performance.

Another class of MTRL algorithms focuses on deducing the current MDP using Bayesian Reasoning. Multi-class models proposed by (12) and (2), attempt to assign class labels to the current MDP given a sequence of observations made from it. (2) use a Hierarchical Bayesian Model(HBM) to learn a conditional distribution over class labels given the observations. The agent samples an MDP from the posterior distribution in a manner similar to Thomson sampling, and then chooses the action. We follow the same procedure for action selection, but incorporate exploration bonuses into it as well.
(6) proposes a novel method using Deep Recurrent Memory Networks to learn policies on Minecraft multi-task environments. They used a fixed memory of past observations. This model successfully learns policies on I-shaped environments where the color of a marker cell determines the goal location. However, the agent was limited to 90^o turns and moved in 1 block steps, which implies that the approach is to treat the frames as a sequence of unrelated images. Our approach, in contrast, attempts to tie all the frames together by considering them as different views of a latent higher dimensional collection of*objects*.

Our Spatial Reasoning model for 3D Video Prediction attempts to address this issue by proposing a structure that models both the scene elements and it's dynamics.
There have been previous (somewhat successful) attempts to visually represent 3D systems. These include the *EM* approach followed by (13) that assumes a linearly transformable internal representation that can model the the shift in the viewing angle. Another significant method is that of *Transforming Auto-encoders*(20) which uses a fully convolutional auto-encoder that also learns a probabilistic affine transformation of the intermediate representation. While the latter method also focuses on 3D image reconstruction, it's performance is limited by the use of a *de-convolutional* network to construct the image. In addition to this, *Transforming Auto-encoders* do not deal with learning a state-action embedding or with occlusion (the tests only had a single unobstructed object).
Another recent method that has promising results, in representing projections of higher dimensional objects, is *Deep Symmetry Networks*(19). *Deep Symmetry Networks* work with a generalized form of the convolutional neural network to apply kernels to learn filters in any Symmetry Group. However, while the network can learn a host of transformations that normal convolutional networks can't (like rotation and scaling), it is still limited to treating linear motion under 3D perspective as an affine transformation of the

objects in image space. Contrary this method, our approach attempts to learn a set of hidden parameters that represent the latent 3D scene itself.

CHAPTER 3

MOTIVATION

This thesis covers multiple approaches that were created to implement spatial reasoning. From the simplest to the most complex, these are:

1. RBM-based reasoning for Map-like environments.
2. Deep Generative Models for Map-like environments.
3. Projection Networks for 3D video prediction.

3.1 RBM-based reasoning for Map-like environments

Definition 3.1.1 ***Map-like environments***
In the context of this thesis, Map-like environments are a class of environments where, after an action, an agent receives information not only about the rewards ($r_{s_t,s_{t+1}}$) and transition probabilities $P(s^{'}, s_{t+1})$ of the target state s_{t+1} but also the rewards and transition probabilities of the states adjacent to the target state $S^{'} = \{s|s \in adj(s_t), s_t \in S\}$

Note that this definition also requires a symmetric relation among the states which represents state adjacency $adj(s, s^{'})$. For example, in a grid-like world, cells that are next to each other are defined as *adjacent*.

Intuitively, this model can be applied to environments where some information about the surrounding states can be gathered (like a robot with GPS and a camera moving around a map). In the environment this thesis considers, the L-world, at every time step the agent is given the rewards and transition probabilities of *adjacent* states.

Since a small subset of information about the system's dynamics ($P(.)$ and $R(.)$) are available, the logical next step is to reason about the rest of the parameters that make up $P(.)$ and $R(.)$.

The RBM (4) is especially good at this task since it develops an unsupervised probability model over all the outputs and assumes a hidden internal state to represent each possibility.

Hence, the RBM is used to predict the unseen parameters from the seen parameters (the challenge mitigated here is that the set of *seen* parameters need not be the same in every step and episode).

Once the unseen parameters are predicted, then methods like *Dynamic Programming*, *Value Function Iteration* etc, may be applied to converge to the final value function.

3.2 Deep Generative Models for Map-like environments

The deep generative model was first proposed as a solution to the many shortcomings of the above method.

3.2.1 Complexity

The standard RBM lacks the ability to handle complex patterns/environments which look less like rigid binary grids and more like real world images. To handle this, we use deep Variational Auto-encoders which, like RBMs, produce a representation Z from an image/input X.

The issue here is that the image is partially observed and the original VAE (10) is designed for images where every pixel has a definite value. In order to get around this, we propose a simple modification made to the VAE objective function that allows it to treat missing pixels in a neutral manner (no information is derived from blocked/masked pixels).

3.2.2 Handling Indicator States

Definition 3.2.1 ***Indicator States***
Indicator States are defined as states that have no reward/relatively small reward but which when observed provides non-trivial information about the other parts of the Map-like environment. Intuitively, the indicator states are like a signboard that the agent can use to improve it's current belief of the other states.

In cases where indicator states are present close to the agent's trajectory and provide information that the agent can use to shorten it's journey (no need to visit every possible goal location), the optimal path for the agent is through the indicator state.

The current formulation simply uses the seen states to derive information about the unseen states. It does not *actively* seek states which provide useful information about the other states.
In order to mitigate this, a reward bonus is developed that is provided to a particular location/state in a Map-like environment, based on the utility of the state in determining the other states.

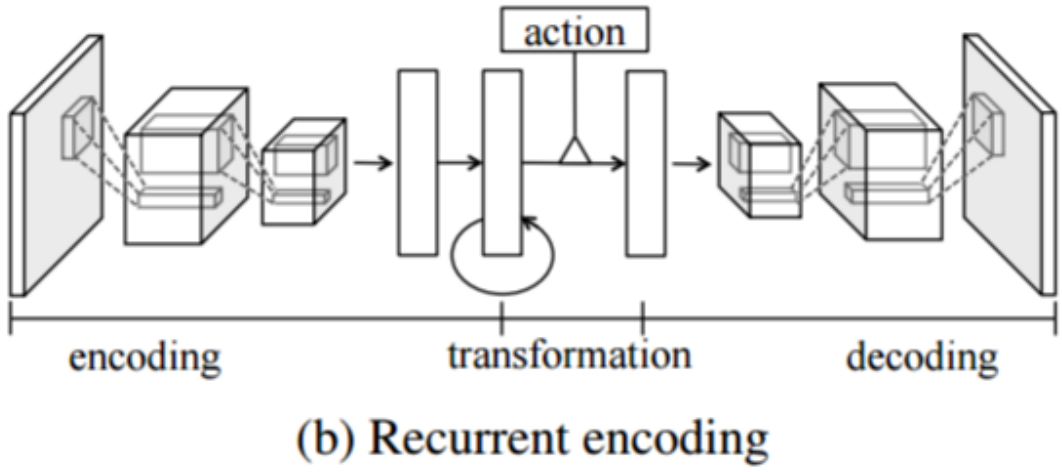
3.3 Projection Networks for 3D video prediction

3.3.1 Video Prediction

Video Prediction is an up and coming field where deep *convolutional* networks try to predict an entire frame of a game from it's past frames. Reconstruction of the image from the representation is typically done using a *de-convolutional network*

Barring some exceptions, a deep network typically has to learn the rules of a game to predict successive frames accurately. The ability to predict frames in the future opens up a lot of possibilities for RL agents as they can simulate possibilities from every state before having to actually select an action at that point.
The latest in this field is the paper on Action-Conditional Video Prediction which uses

Figure 3.1: **Action Conditional Video Prediction** for ATARI games - This architecture was built to predict frames of the ATARI series of games/



a complex Convolutional-Deconvolutional network with an action-dependent linear transform on the intermediate representation vector.

The shortcomings of this method include the limitations of the convolutional network. This architecture works well for 2D games like *SeaQuest* and *Freeway* where the successive positions of 2D sprites move linearly between frames, rarely changing structure or form (the 2D cars look exactly the same in every frame and move a well-defined constant amount each frame).
However, when we take the problem of video prediction in the 3D object case, there are problems that are immediately evident from the variable nature of the motion in 3D environments. The perspective projection (which is the most common) of a 3D system has very complex motion in pixel coordinates even though the motion is fairly simple in 3D space.

Since convolutional systems were originally built to identify patterns in a position-invariant manner, it makes no efforts to understand objects that change form-factors on the screen when they move, but implicitly are simply projections of constant (fixed shape) objects.
The convolutional network only looks at the 2D projected shape and develops filters for it.

Figure 3.2: **2D Motion** A comparison of images that illustrates the simple translational motion present in the ATARI game: SeaQuest

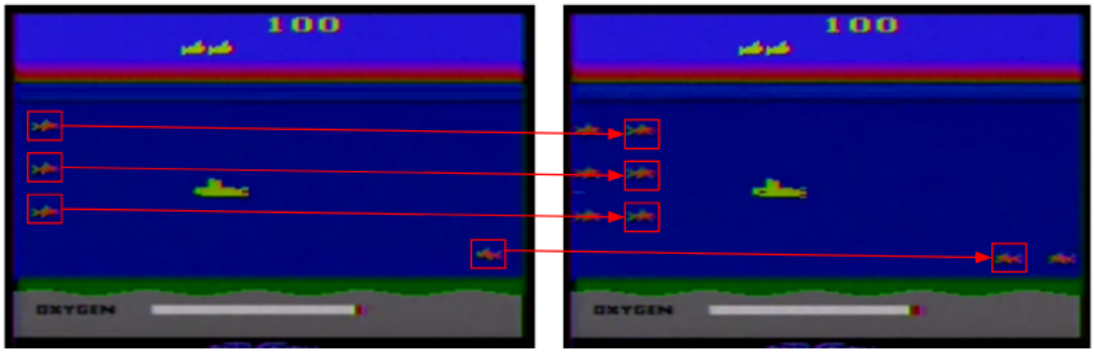
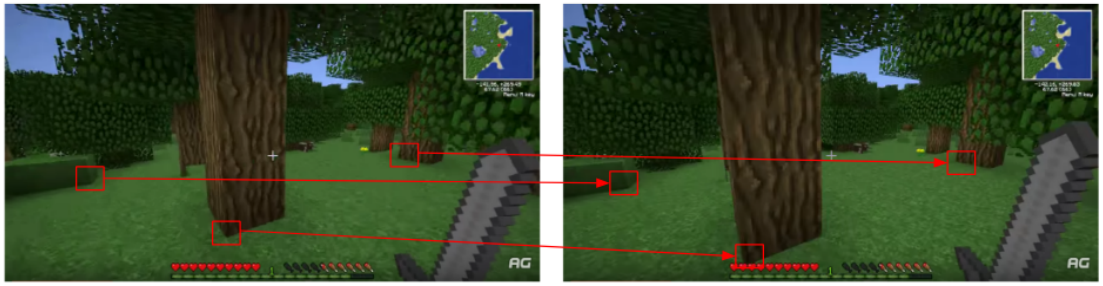


Figure 3.3: **3D Motion** A comparison of images that illustrates the complex motion present in Minecraft



When the camera moves and the object changes form factor, the filters used by the system to recognize the *same* object will change. In this manner, the neural network has to learn all possible forms of the object *and* each one's transformation.

3.3.2 Ray Tracing-inspired Network

Computer Graphics prescribes a standardized method to convert from 3D objects to a 2D projected image. *Ray Tracing* is a general method that can be applied to any form of projection depending on how the rays are constructed.

Intuitively, a neural network which explicitly models the 3D to 2D projection by mimicking the process of ray tracing will be able to model motion better.
This principle is similar to when convolutional networks were first introduced: weight sharing across filters would logically help learn translation-invariant patterns by general-

ising weights in one spot to the entire image.

This logic is the motivation for proposing such an architecture. Note that in this thesis, we only propose an architecture for the *deconvolutional* part (convert from the intermediate representation Z to image) while the convolutional network that converts frames to the intermediate representation remains largely the same (except that the intermediate representation now has two parts: State S and Context T.

To come up with the structure, some simplifying assumptions are made:

1. The world has simple solid colors (like Minecraft). A more complex architecture could possibly work with other worlds too, but given that this is the first-of-it's-kind architecture, we start out with simpler networks.
2. The motion is close to continuous: If the agent turns in 90 degree increments, the system will be unable to find correlations. In our experiments, the agent turns in 30 degree increments.
3. For our ACPNN architecture, the start position is always the same. The agent essentially does *not* use input at this stage, but rather attempts to fit it's model to predict the *outputs*. In Input-ACPNN, we use a convolutional network to process the input and predict both the state of the unit and the network.

A key feature here is the assignment of uniformly varying coordinates to each pixel in the image. This enforces the *linearity* of ray creation (during ray tracing) and helps immediately generalize the learning at one pixel to all others.

As shown in the figure, the structure is such that we imitate the process of ray tracing. The current state (position and orientation) and the pixel coordinate is fed to a linear model to create the *ray*.
Although we expect something like a *'ray'* to be created here, the network may converge onto some other form of useful linear element (we call this the *ray element* R).

The network maintain a set of static *memory* cells (which are called *scene elements*), each of which contains one 4-element color C and a vector representation V (the vector

representation intuitively represents it's shape/location).

The ray element R along with the memory cell representation V is then passed through the network to produce an attention A. This attention represents the significance of this scene element at this pixel given that the agent is in this state (in our ray tracing model, this is the intersection part).

A sum of all the scene element colors weighted C by their attention A gives the final color for the pixel.

While the initial state is assumed to be all 0s for the ACPNN model, the assumption is that it always starts out at the same spot. However, if we are to create a vector representation for a given scene, we need to include a component that uses the previous frames to predict the current state of the agent (initial state S_0) and the nature of the environment (scene context T).

This is implemented in the Input-ACPNN model, which contains a convolutional network that produces S_0 and T. The motivation here is that, if the network is trained end-to-end, the *input network* will learn the filters required on the first $T-1$ frames to extract information that will allow the output *projection network* to successfully simulate the scene from time T onwards. Since the only link (similar to an auto-encoder) between the input and output networks is the state S_0 and context T vectors and only T (not S_0) does not participate in the intersection unit, the context T will be a vector representation of the scene.
This architecture is simply an auto-encoder, but modified to work with entire 3D environments rather than just the scene.

CHAPTER 4

BACKGROUND

This thesis assumes the reader has some background in Neural Networks. The following sections give some background on Reinforcement Learning and advanced Machine Learning concepts.

4.1 Reinforcement Learning

Reinforcement Learning is a branch of Machine Learning distinct from the Supervised and Unsupervised paradigms, that deals with the control of an *agent* that has a well-defined state, can take *actions* to change this state, and receives a *reward* signal that it seeks to maximise.

Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). A Markov decision process is a 5-tuple $(S, A, P.(\cdot,\cdot), R.(\cdot,\cdot), \gamma)$, where

- S is a finite set of states,
- A is a finite set of actions (alternatively, A_s is the finite set of actions available from state s s)
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

(Note: The theory of Markov decision processes does not state that S S or A are finite, but the basic algorithms below assume that they are finite.)

A standard environment is one where the agent receives a reward after taking an action a_t at time t.
Further in this thesis, we define a Map-like environment, an appropriate simplification, which provides the agent information about adjacent states, i.e their rewards and transition probabilities.

4.2 Q-learning

Q-learning is one of the most basic algorithms used in RL. It's off-policy nature (the agent can learn from experience that was generated while following a different policy) gives it a distinct advantage that has allowed Q-learning to be applied to a wide variety of applications.
In this thesis, Q-learning serves as the base algorithm for Deep Q-networks, which are used as a part of the ACPNN framework to incentivise exploration.

Definition 4.2.1 ***Q-value*** *The Q-value for a state, action pair (for an episodic task) is defined as the expected discounted return if an agent takes that action from that state and follows the same policy for the rest of the episode.*

$$Q(s,a) = E(\sum_{i}^{\infty} \gamma^i (R(s_i, \pi(a_i))))$$

In Q-learning, the Q-values are calculated from a set of transitions (s_t, a_t, r_t, s_{t+1}) using the following equations:

$$Q(s_t, a_t) = r_t + \gamma \cdot max_a(Q(s_t, a))$$

4.3 Value Iteration

Value Iteration is a technique that utilizes the properties of the Bellman equations that recursively define the value function $v(s)$.

By applying the Bellman equation repeatedly, the vaule function $v_t(s)$ eventually converges to the correct values. The exact number of iterations depend on the structure of the state space. However, there are plenty of heuristic bounds that are useful in determining the terminal point.

Algorithm 1 Pseudo-code that demonstrates the process of projecting a 3D scene onto a 2D representation

procedure VALUEITERATION(p_x, p_y)
    $v(s) \leftarrow 0 \forall s$
    while do $v_t(s) - v_{t+1}(s) < threshold$
        $v_{t+1}(s_n) = \sum_a \sum_{s_m} [r_t(s_m, a) + \gamma . v_t(s_m) . p(s_m, a, s_n)]$

4.4 (Neural) Fitted Q-Iteration

Fitted Q iteration, typically used with a neural network (but can be used with other models too), is a simple method developed to stabilise the neural network used for approximating the Q-value. Instead of tabular Q-learning's on-line approach that updates each Q-value individually, Fitted Q-iteration gathers all the samples of $(s_t, a_t, r_{t+1}, s_{t+1})$ and then applies supervised learning to train the network on the targets. The target for iteration t is defined as (17):

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \gamma . max_a Q_t(s_{t+1}, a)$$

4.5 Gaussian-Binary Restricted Boltzmann Machines

RBMs have been used widely to learn energy models over an input distribution $p(\mathbf{X})$. RBM is an undirected, complete bipartite, probabilistic graphical model with N_h hidden units, H, and N_v visible units, V. In Gaussian-Binary RBMs, hidden units are binary units(Bernoulli distribution) capable of representing a total of 2^{N_h} combinations, while the visible units use the Gaussian distribution. The network is parametrized by edge weights matrix $\mathbf{W}$ between each node of V and H, and bias vectors $\mathbf{a}$ and $\mathbf{b}$ for V and H respectively. Given a visible state $\mathbf{v}$,
the hidden state, $\mathbf{h}$, is obtained by sampling the posterior given by

$$p(\mathbf{h}|\mathbf{v}) = \frac{1}{1 + \exp(\mathbf{W}^T\mathbf{v} + \mathbf{b})}$$

Given a hidden state $\mathbf{h}$, visible state $\mathbf{v}$ is obtained by sampling the posterior given by

$$p(\mathbf{v}|\mathbf{h}) = \mathcal{N}(\mathbf{W}^T\mathbf{h} + \mathbf{a}, \mathbf{\Sigma})$$

Since RBMs model conditional distributions, conditional distributions($p(\mathbf{v}|\mathbf{h})$ and $p(\mathbf{h}|\mathbf{v})$) have a closed form while marginal and joint distributions($p(\mathbf{v})$, $p(\mathbf{h})$ and $p(\mathbf{h}, \mathbf{v})$) are impossible to compute without explicit summation over all combinations.

Parameters are learnt using contrastive divergence (4). Learning $\mathbf{\Sigma}$, however, proved to be unstable (4) and hence, we treat σ as a hyperparameter and use $\mathbf{\Sigma} = \sigma * \mathbb{I}_{N_v}$.

4.6 Variational Auto-encoders

Variational Auto Encoders(VAE)(10) attempt to learn the distribution that generated the data $\mathbf{X}$, $p(\mathbf{X})$. VAEs, like standard autoencoders have an encoder, $\mathbf{z} = f_e(\mathbf{x})$, and a decoder $\mathbf{y} = f_d(\mathbf{z})$ component. Generative models that attempt to estimate $p(\mathbf{X})$ use a likelihood objective function, $p_\theta(\mathbf{X})$ or $\log p_\theta(\mathbf{X})$. More formally, the objective function

can be written as

$$p(\mathbf{x}) = \int_{\mathbf{Z}} p(\mathbf{x}|\mathbf{z};\theta)p(\mathbf{z}) \approx \sum_{z\sim\mathbf{Z}} p(\mathbf{x}|\mathbf{z};\theta)p(\mathbf{z})$$
$$\hat{p}(\mathbf{x}) \approx \sum_{z\in\mathbf{D}} p(\mathbf{x}|\mathbf{z};\theta)p(\mathbf{z}) \text{ where } D = \{\mathbf{z_1}, \mathbf{z_2} \ldots, \mathbf{z_m}\} \text{ and } \mathbf{z}_k \sim \mathbf{Z}\, \forall k$$

where $p(\mathbf{x}|\mathbf{z};\theta)$ is defined to be $\mathcal{N}(f(\mathbf{z};\theta), \sigma^2\mathcal{I})$.

Gradient-motivated learning requires approximation of the integral with samples. In high-dimensional $\mathbf{z}$-space, this could lead to large estimation errors as $p(\mathbf{x}|\mathbf{z})$ is likely to be concentrated around a few select $\mathbf{z}$s and it would take an infeasible number of samples to get proper estimate. VAEs circumvent this problem by introducing a new distribution $\mathcal{Z} \sim \mathcal{N}(\mu_\phi(\mathbf{z}), \sigma_\phi(\mathbf{z}))$ to sample D from. To reduce parameters, we use $\sigma_\phi(\mathbf{z}) = c \cdot \mathbb{I}$. These two functions are approximated with a deep network and form the *encoder* component of the VAE. $p_\theta(\mathbf{X})$ is represented using the sampling function $f(z;\theta)$ where $z \sim \mathcal{N}(0,1)$ and $f(z)$ forms the *decoder* component of the VAE. After some mathematical sleight of hand to account for $\mathcal{Z}$ in the learning equations provides an intuitive understanding of these equations), we obtain the following formulation of the loss function

$$E = \mathcal{D}_{\mathcal{KL}}(\mathcal{N}(\mu(\mathbf{X}), \sigma(\mathbf{X})), \mathcal{N}(0,1)) + \sum_{k=1}^{N} \| \frac{(\mathbf{y}_i - \mathbf{x}_i)}{\sigma}^2 \|$$

where the KL-divergence term exists to adjust for importance sampling $\mathbf{z}$ from $\mathcal{Z}$ instead of $\mathcal{N}(0,1)$.

4.7 3D Scene Generation

Classical Computer Graphics typically deals with two broad areas: Projection and Pixel-Shading.

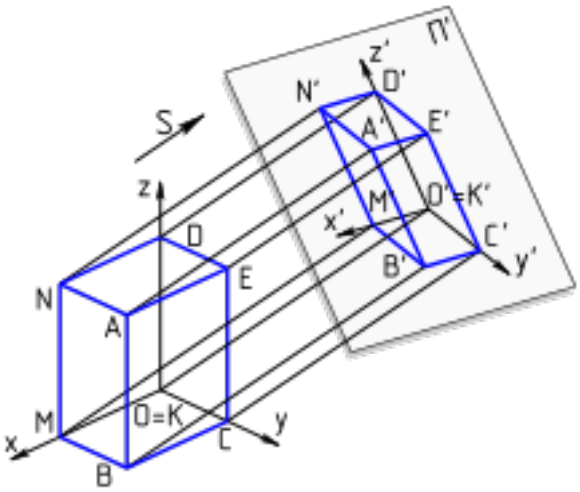
The former, "*projection*", deals with transforming the visible 3D scene into the viewport of the camera (aka, a 2D representation). In an abstract sense, projection simply reduces the dimensionality by performing an irreversible transformation to a simpler space (13). The latter, *pixel-shading*, deals with calculating the color of each pixel in the final scene. For the purposes of this thesis, we ignore this part (all surfaces in our world have a

distinct, solid color) as it's irrelevant to our objective.

Projection, in practice, is mostly done in two different modes

1. Orthographic Projection: Orthographic projection (or parallel projection) involves

Figure 4.1: Orthographic Projection



directly projecting every vertex along the perpendicular onto the target plane. This projection maintains the relative distances between points, but for practical purposes produces an extremely unnatural image of the scene. the fact that perspective distance to the plane does not affect the projection of a vertex makes it impossible to pick up depth cues through parallax.
These reasons, combined with the fact that both video games and real-life cameras are represented by the perspective projection, justify why this thesis focuses mostly on the perspective projection.
However, the demonstrated architecture can just as easily be used for any projection with certain properties (which will be stated soon).

2. Perspective Projection:

   A simple form of the perspective projection simply transforms the 3D point into the tangent of the angle made by the point to the plane along X and Y axes.
   The following diagram presents a simple definition of Weak Perspective Projection:

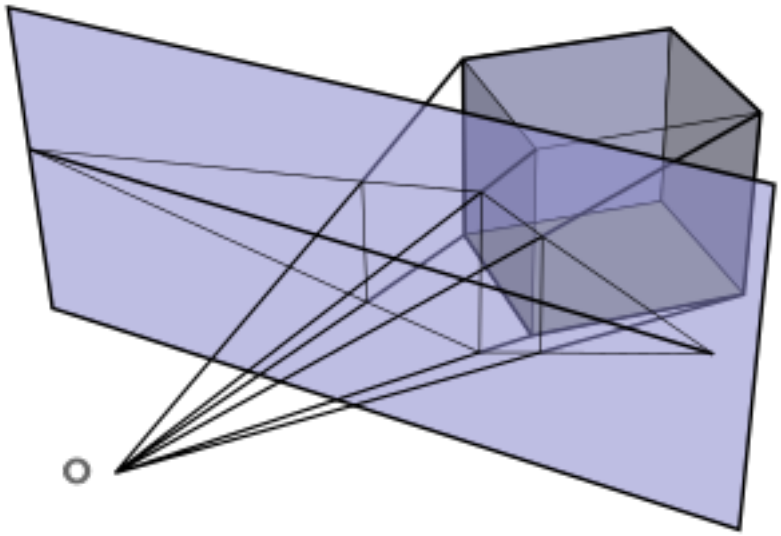
   The following equations outline the transformation from camera-space 3D coordinates (X_c, Y_c, Z_c) to pixel space (with the vanishing point as the origin):

$$P_x = \frac{\frac{X_a}{Z_a} + 1}{2}.W$$

$$P_y = \frac{-\frac{Y_a}{Z_a} + 1}{2}.H$$

   If the origin $(0, 0, 0)$ and the Z-axis are not the camera position and direction respectively, an additional linear transformation is required to get the world-space

Figure 4.2: (Weak) Perspective Projection



coordinates into camera-space (commonly referred to as the view matrix)

$$(X_c, Y_c, Z_c) = M_v.(X_w, Y_w, Z_w)$$

A third, more important projection is the (Strong) Perspective projection. Although we find the strong perspective projection widely used in games, for simplicity, this thesis limits itself to the weak perspective projection.

The Strong Perspective projection maps a *finite frustum* onto screen coordinates. Unlike *Weak Perspective*, *Strong Perspective* makes the additional assumption that points beyond a certain distance from the screen plane and points within a certain distance do not appear on the screen. The method accomplishes this by ensuring that the transformed screen space z coordinates of the point in the range $[0, 1]$ are valid, while others are clipped (omitted) from the screen.
While *Strong Perspective* is a more efficient and realistic model used by most 3D games, it is also unnecessarily complicated for the purposes of this thesis. To prove that this technique offers promising results, only the *Weak Perspective* transform is enough.

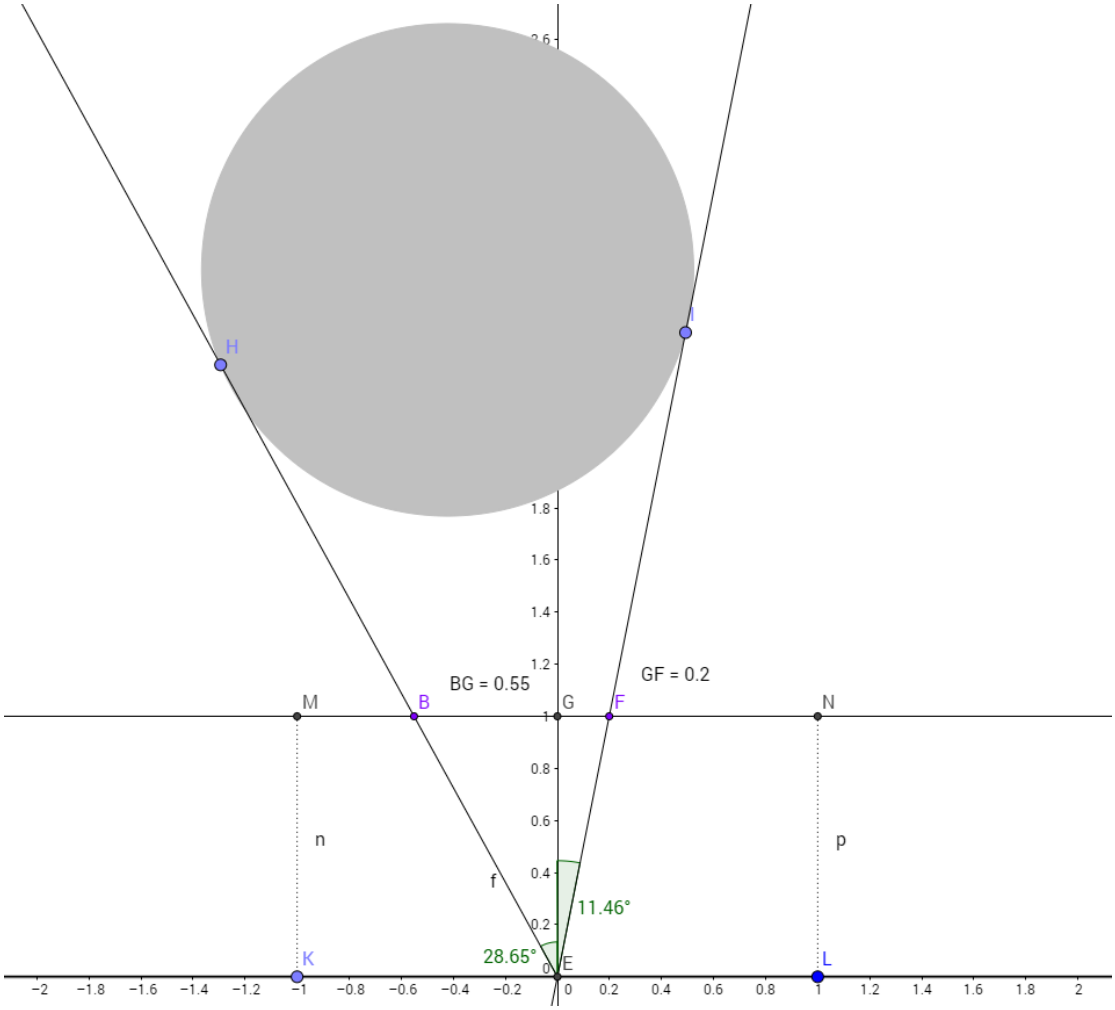


Figure 4.3: **Illustration of Perspective Projection** $FOV = 90^o$
For a fixed vanishing point, E, the weak perspective projection of points H and I is simply the intersection of the line HE and IE with the plane MN perpendicular to the camera direction and at a distance of *unity* from the camera (B and F). Note M and N (at $x = -1$ and $x = +1$ respectively) represent the far ends of the projected image and that B and F represent the location of H and I on the projected image with respect to M and N

4.8 Ray Tracing

Ray tracing (or more simply, ray casting) is a technique where a ray corresponding to each of the pixels of the final image is traced through a 3D scene and is tested for intersection with an object.

Every pixel in an image corresponds to a ray in 3D space which depends on the projection mechanism used for this particular system.

The algorithm to form the ray uses the inverse of the perspective projection algorithm.

The general one-step ray tracing algorithm works as follows:

Algorithm 2 Pseudo-code that demonstrates the process of projecting a 3D scene onto a 2D representation

procedure RENDERPIXEL(p_x, p_y)
    ray $\leftarrow GetRay(pixel\_x, pixel\_y)$
    $curr\_depth \leftarrow \infty$
    $curr\_intersection \leftarrow$ **null**
    for doobj in $objects$
        intersection $\leftarrow RayObjectIntersect(ray, object)$
        if $intersection.depth < curr\_depth$ **then**
            $curr\_depth = intersection.depth$
            $curr\_intersecion = intersection$
        if $curr_intersection \neq$ **null then**
            return $curr\_intersection.color$

Algorithm 3 Pseudocode that demonstrates the process of producing rays in the context of *perspective* projection

procedure GETRAY(p_x, p_y)
    $h_x \leftarrow 2.\frac{p_x}{W} - 1$
    $h_y \leftarrow -2.\frac{p_y}{H} + 1$
    return (h_x, h_y)

CHAPTER 5

FULLY-OBSERVABLE SPATIAL REASONING

5.1 RBM Model

5.1.1 Problem Statement

The first iteration of the concept of spatial reasoning involved a simple problem statement. The initial test problem used the special L-world which was a set of 2 binary 2x2 fields where the non-black cells were obstacles.

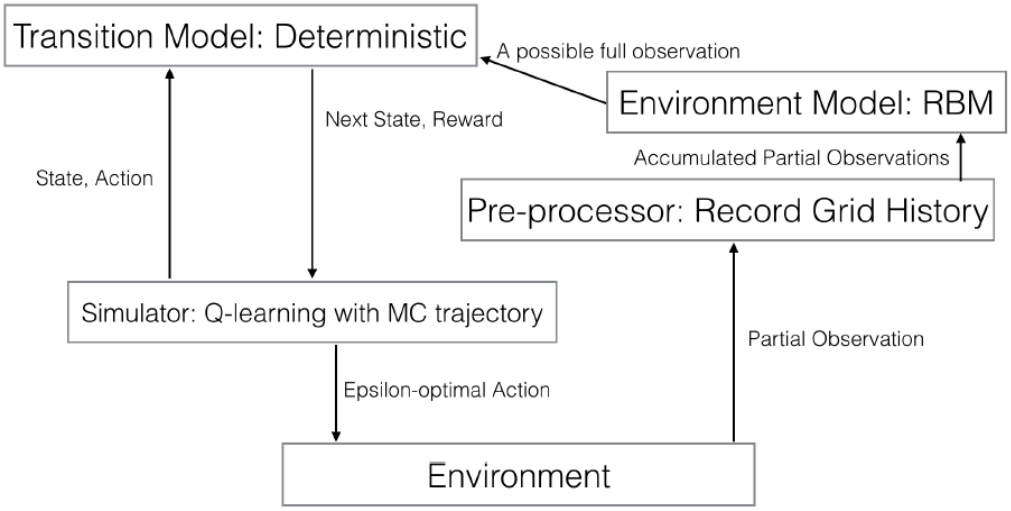
Figure 5.1: **L-world** - *Red* cells denote open space while *Green* cells denote obstacles. In both cases, the source is the top-left while the destination is the bottom-right. The agent can only see a 3x3 square area around it

The agent in the L-world can only see a small 3x3 field around itself at any point of time and it progressively explored the world as it moves through it.
For each episode, the actual world that the agent played was randomly chosen from the set of 2 possibilities in the L-world. Since the policy followed for each of the two worlds are completely different, standard methods like Q-learning, which work without context, fail in this context.

5.1.2 Approach

In order to ensure the available data is used completely, a probability based approach is used to reason about the unseen parts of the world using information about the parts that *have* been observed.
The framework used by this approach (and also by the Deep Generative Model) is explained in Figure 5.2

Figure 5.2: **Spatial Reasoning Framework** for Map-like environments

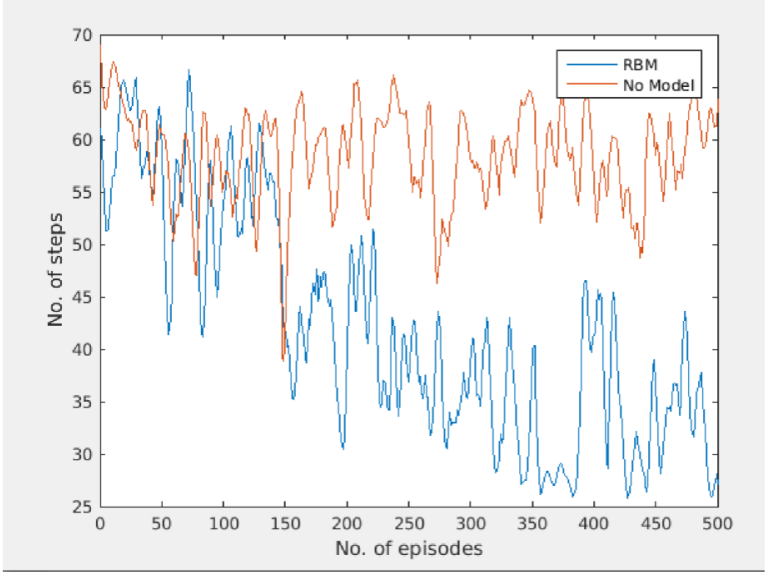


In this abstract architecture, the model used here is a *binary* RBM (more powerful models can be used for more complex environments). The binary RBM spans the entire grid and each cell in the grid is treated as a separate unit in the output of the RBM.
At each stage, the RBM is sampled using the currently visible portion of the environment. (For the invisible parts, the state is decided by sampling using only the bias of each cell).

5.1.3 Results

The RBM model tested against a Q-learning agent with no model as a baseline, to test for the gains by having a model.

Figure 5.3: **RBM vs. No-Model**



5.2 Deep Generative Model

5.2.1 Encoding

Let us consider the nature of our inputs. We have assumed that the agents observed surroundings are embedded on a map as an image X. A mask M is a binary image, of the same dimensions, with $m_i = \mathbf{1}$ if its corresponding state has been observed by the agent. We denote the i^{th} pixel and mask be denoted by x_i and m_i respectively.

In most episodes, the agent will not visit the entire grid-world, hence $m_i = 0$ for some $x_i \in X$. Since there can be several views of the same ground-truth MDP, we need to be able to reconstruct the ground-truth MDP from multiple observations of the MDP over several episodes. For Single-Task RL, this can be done in a tabular fashion. In MTRL, however, we have potentially infinite possible MDPs and it becomes hard to build association between different views of the same MDP.

Our method uses deep convolutional VAEs to infer the association between different views of the same MDP and use it with a low dimensional energy model to sample MDPs given the observations. The fact that the world has continuous local features (like continuous walls/obstacles) implies that the use of a convolutional network greatly boosts performance (by improving the ability to generalise). Figure 5.4 shows our setup to learn the associations and to infer ground truth MDP given observations. We use one

setup to train the model and another to allow back sampling of MDPs. We call these the *train* and *query* models.
Our method can be scaled to large state spaces because of the use of a *deep* auto-encoder.

Given this setup, for the learning phase, we modify the VAE loss function to account for unobserved states, $x_i \in X$ with $m_i = 0$. The new loss function is given by

$$E = \mathcal{D}_{\mathcal{KL}}(\mathcal{N}(\mu(\mathbf{X}), \sigma(\mathbf{X})), \mathcal{N}(0, 1)) + \sum_{k=1}^{N} m_i \cdot \frac{(\mathbf{y}_i - \mathbf{x}_i)^2}{\sigma}$$

Inclusion of m_i in the loss function is quite intuitive and works well on the sets that we tested on, since it removes any penalty for unseen x_i and allows the VAE to project its knowledge onto the unseen states.

5.2.2 Sampling

Given a partial observation, X, we sample for the posterior to obtain K MDP samples. If X doesn't have enough evidence to skew the posterior in favour of one single MDP, then the encoding produced by VAE, $\mathbf{z}$, is far from encodings of ground-truth MDPs, $\mathbf{z}_{in}$ in $\mathbf{z}$-space. We obtain an MDP that is a mixture of MDPs if we sample from this posterior. Solving this MDP could result in the agent following a policy unsuitable for any of the component MDPs in isolation.
One way to circumvent this problem is to train a probability distribution over the MDP embeddings, $\mathbf{z}_{in}$. For our 2-MDP environments, we use a Gaussian-Boltzmann RBM to cluster inputs with fixed-variance gaussians. We then use Algorithm 4 to sample from these gaussians.

Algorithm 4 Sample MDPs given $\mathbf{x}$

K MDPs sampled from model posterior Compute $\mathbf{z} = f_e(\mathbf{x})$
Sample $\mathbf{K}$ hidden RBM states $\mathbf{h^{(i)}} \in H$, $i \in \{1 \dots K\}$ from the posterior $p(\mathbf{h}|\mathbf{z})$
Calculate MAP estimate $\mathbf{z^{(i)}} = \arg\max_{\mathbf{z}} p(\mathbf{z}|\mathbf{h^{(i)}})$
Decode map estimates $\mathbf{z^{(i)}}$ to get MDP samples $\mathbf{y^{(i)}}$

5.2.3 Value function

Given K samples from model posterior, $p(\mathbf{y}|\mathbf{x})$, we perform action selection using an aggregate value function over the K samples. We define, for each state s, an aggregate value function $\bar{V}(s)$ as

$$\bar{V}(s) = \mathbb{E}_{\mathbf{m}\sim p(\mathbf{y}|\mathbf{x})}\big[V_{\mathbf{m}}(s)\big] \approx \frac{\sum_{k=0}^{K} V_{\mathbf{m}_k}(s)}{K}$$

where $\mathbf{m}$ is an MDP and $V_{\mathbf{m}}(s)$ denotes the value function for state s under MDP $\mathbf{m}$. $V_{\mathbf{m}}(s)$ can be obtained using any standard planning algorithms and we use value iteration(with γ=0.95, 40 iterations). Action selection is done using ϵ-greedy mechanism with $\epsilon = 0.1$. Since recomputing value functions at each step is computationally infeasible, each selected action persists for $\tau = 3$ steps.

We note that value functions used need not be exact, but can be approximate as they are only used for τ steps. A quicker estimate can be obtained using Monte-Carlo methods when the state-space is large.

5.3 Exploration Bonuses

5.3.1 Jacobian Exploration Bonus

To incentivize the agent to visit decisive pixels/locations, we introduce a bonus based on the change in the embedding Z. Intuitively, the embedding Z has the highest change when the VAE detects changes that are relevant to the distribution $p(X)$ that it is modelling. The bonus can be summarised as follows:

$$B_\alpha(s) = \alpha \cdot tanh\left(\epsilon + \frac{\partial \mathbf{z}}{\partial \mathbf{x_s}}\right)$$

where $\mathbf{x}_s$ denotes the list of observations made at state s. We use a $tanh(\cdot)$ transfer function to bound activations produced by the Jacobian, thus mainitaining numerical stability. This bonus can be used in two ways - as a pseudo reward,

$$R_s = R_s(\mathbf{x}) + B_\alpha(s)$$

or to replace the actual reward.

$$R_s = max(R_s(\mathbf{x}), B_\alpha(s))$$

where $R_s(\mathbf{x})$ is the actual reward deduced by the agent. While both methods showed improvement, the latter worked better since total reward for states which already gave a high reward was not further increased. Since B_α changes drastically with new observations, B_α is recomputed every time the $\hat{V}_t(s)$ is to be recomputed. B_α is also memory-less i.e. it doesn't carry over any information from one episode to the next.

5.3.2 Other Exploration Bonuses

Utility

This bonus incentivizes the agent to visit pixels/locations that are positively correlated winning behaviour.

Formally, the update rule using a grid-world trajectory, $\mathcal{T} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_T\}$, is given by

$$\beta^0(\mathbf{x}_t) = c$$

$$\beta^{n+1}(\mathbf{x}_t) = \begin{cases} (1-\delta) \cdot \beta^n(\mathbf{x}_t) + \delta \cdot \gamma^{T-t} R_n & \text{if } \mathbf{not}\ updated^n(\mathbf{x}_t) \\ \beta^{n+1}(\mathbf{x}_t) & \text{otherwise} \end{cases}$$

where R_n is the reward for completing the n^{th} episode and $updated^n(\mathbf{x}_t)$ is $true$ if $\beta(\mathbf{x}_t)$ has already received an update for episode n. The updates are performed in the order $\{\mathbf{x}_T, \mathbf{x}_{T-1}, \ldots \mathbf{x}_1\}$. Single update to $\beta(\mathbf{x}_t)$ was found to be numerically more stable compared to update in every step when the same location was seen at different time steps.

The agent starts out with uniform exploration bonus for every location in the grid-world. Episodes that result in a failure ($R_k = -1$) reduces the exploration bonus for the locations on the path taken. Episodes that result in a success ($R_k = +1$) increment the exploration bonus as this path is positively correlated with successes.

Updates to this bonus are made only at the end of each episode and they carry over to

the next episode.

Jacobian-Utility Hybrid

One issue with plain B_α is that it incentivizes the agent to visit decisive locations regardless of how useful they actually are(for instance, if all the worlds have the same reward structure but different observations).
B_β incentivizes the agent to visit locations based on how useful they are regardless of how decisive they could be. Intuitively, combining the two bonuses to give a $B_{\beta\alpha}$ bonus will incentivize the agent to only visit those locations that are decisive and positively correlated with winning behaviour.
Formally, we define this hybrid bonus $B_{\beta\alpha}$ as

$$B_{\beta\alpha}(\mathbf{x}) = \beta(\mathbf{x})(\epsilon + (1-\epsilon)tanh\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

$tanh$ acts as an activation function maxing out at 1 for those states that are decisive according to the Jacobian of the VAE.

Since $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ changes for every step, this bonus is step-wise adaptive. Also, since $\beta(\mathbf{x})$ is only updated at the end of an episode, this changes from episode to episode, reflecting the exploration done by the agent.

5.4 Testbed

We have implemented the following algorithms.

- Value Iteration, referred to as STRL
- Multi-task RL with VAE, RBM without exploration bonus, referred to as MTRL-0
- Multi-task RL with VAE, RBM, and Jacobian Bonus, referred to as MTRL-α

We have tested the above algorithms on 2 environments.

- Back World (Easy) [BW-E] - Goal location alternates depending on marker location color, marker location is fixed and is in most paths from start to goal. This domain demonstrates the advantage gained using a probabilistic model over the MDPs.

- Back World (Hard) [BW-H] - Same setting as BW-H, but marker location is not on most paths from start to goal. This domain demonstrates the advantage provided by the Jacobian exploration bonus and our generative model.

For STRL, using only visible portions of the environment was very unstable and hence, we had to add a pseudo reward. For each unseen location, we provide a pseudo reward, ε_n for n^{th} step (with $\varepsilon_0 = 0.3$), that is annealed by a factor of $\kappa = 0.9$. Each episode was terminated at 200 steps if the agent hadn't reached the goal. Using this pseudo reward, the agent was forcefully terminated fewer times. These worlds become challenging due to partial visibility. We use a 5x5 kernel with clipped corners and the agent is always assumed to be at the center. At each step, the environment tracks the locations that the agent has seen and presents it to the agent before an action is taken. For our experiments, we consider the average number of steps to goal as a measure of loss and average reward as a measure of performance.

5.5 Results

Table 5.1 gives average reward for each agent. Table 5.2 gives average episode length. We also impose forceful termination at 200 steps if episode has not yet completed. From the results, we infer the following.

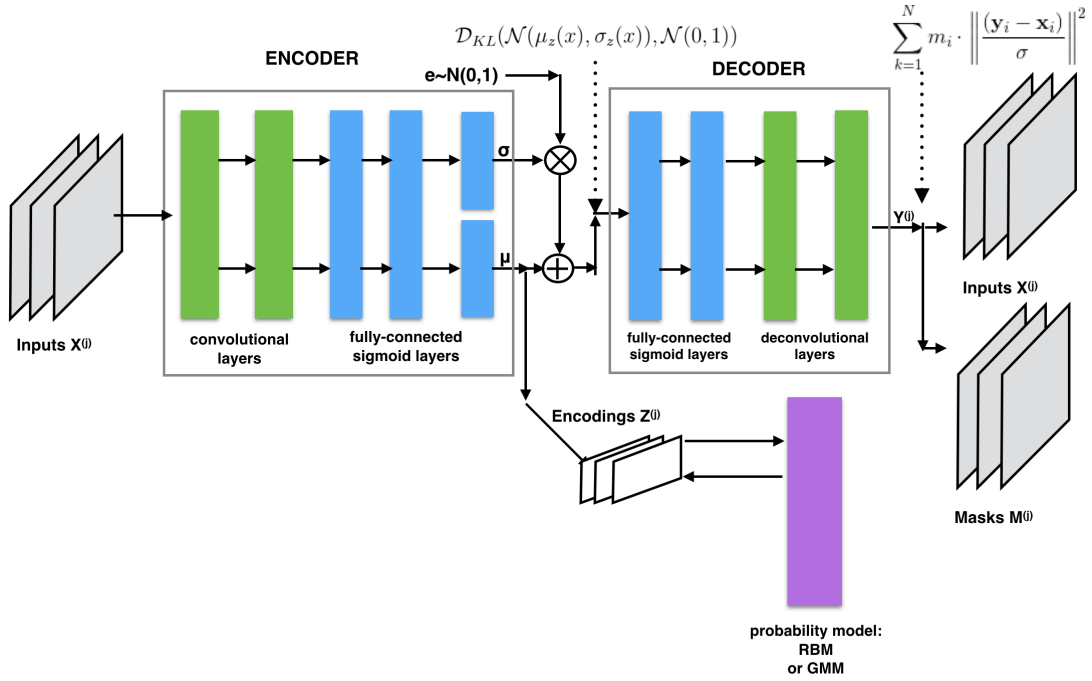
- STRL using value iteration does poorly as it as no way of deducing MDPs.
- MTRL-0 solves both BW-E and BW-H environments and does almost as good as MTRL-α. This improvement can be attributed to the use of our deep generative model.
- MTRL-α shows better results on BW-H. This was expected as MTRL-0 makes no attempt to visit marker locations. MTRL-α is motivated by the Jacobian Bonus to visit marker locations, thereby deducing the MDP.
- MTRL-0 performs as good as MTRL-α in BW-E as marker locations lie on most paths to the goal. However, since it fails to understand the significance of the marker locations and markers in BW-H are not on most paths to the goal, it results in longer episodes and lower reward.

Table 5.1: Average Reward

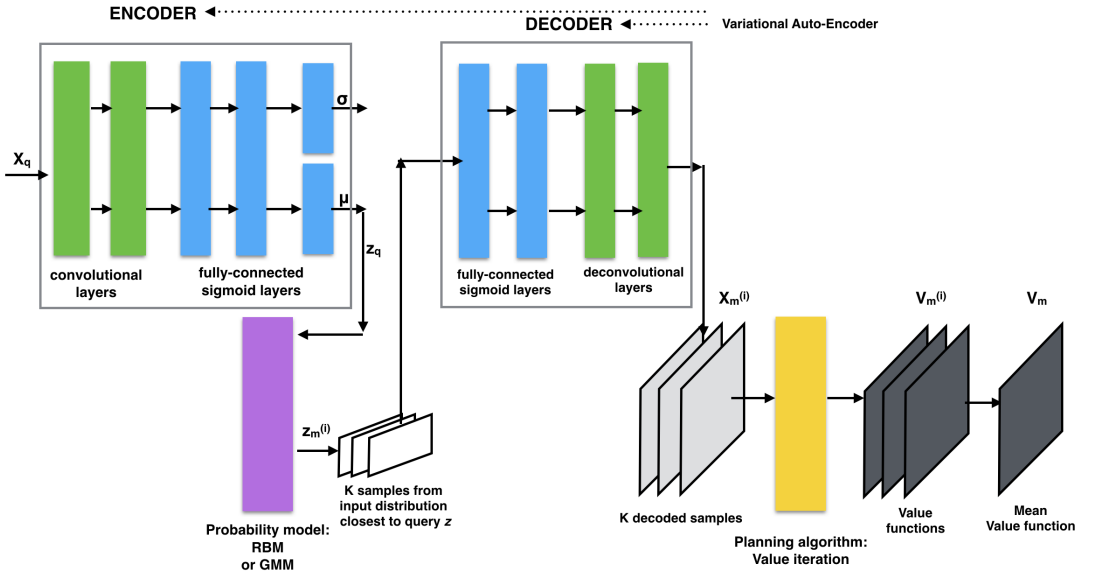
World	STRL	MTRL-0	MTRL-α
BW-E	0.21	**0.99**	**0.99**
BW-H	0.23	0.92	**0.99**

Table 5.2: Average Episode Length

World	STRL	MTRL-0	MTRL-α
BW-E	184.19	**46.20**	46.29
BW-H	183.64	54.0	**45.8**



(a) Train Model



(b) Query Model

Figure 5.4: **Deep Generative Model** - Train model requires mask inputs to account for missing observations. Query model involves value iteration to determine best action over sampled MDPs.

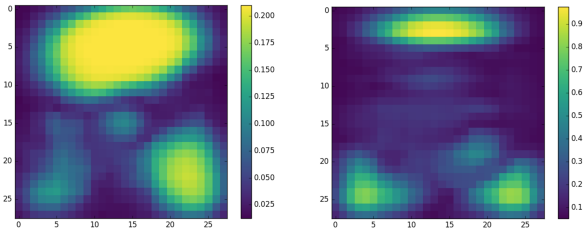
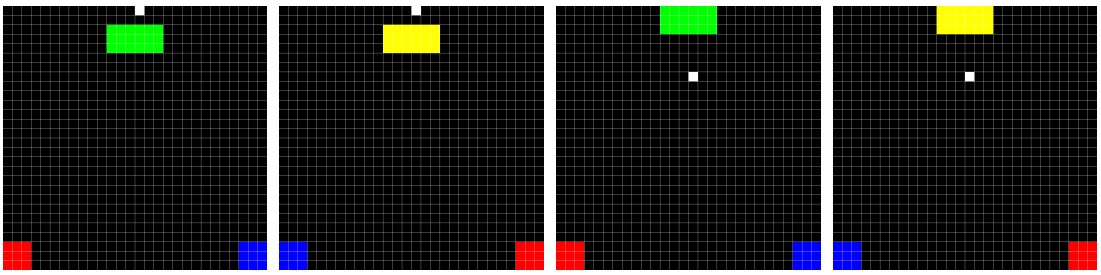
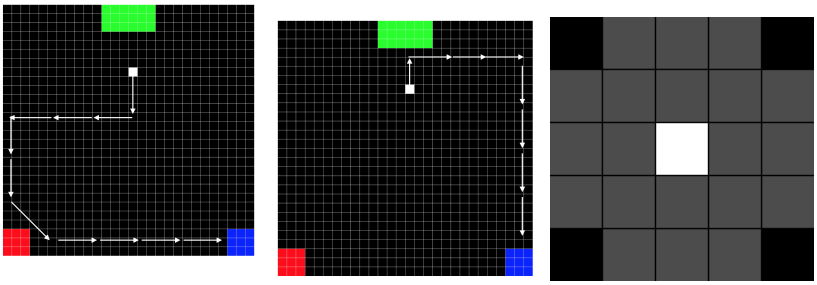


Figure 5.5: **Final Jacobian Bonus for BW-E and BW-H** - Locations in yellow-green are identified by the agent as being most helpful in deducing the MDP being solved.



(a) BW-E A (b) BW-E B (c) BW-H A (d) BW-H B



(e) Suboptimal path[BW-H] (f) Optimal path[BW-H] (g) Visibility Kernel

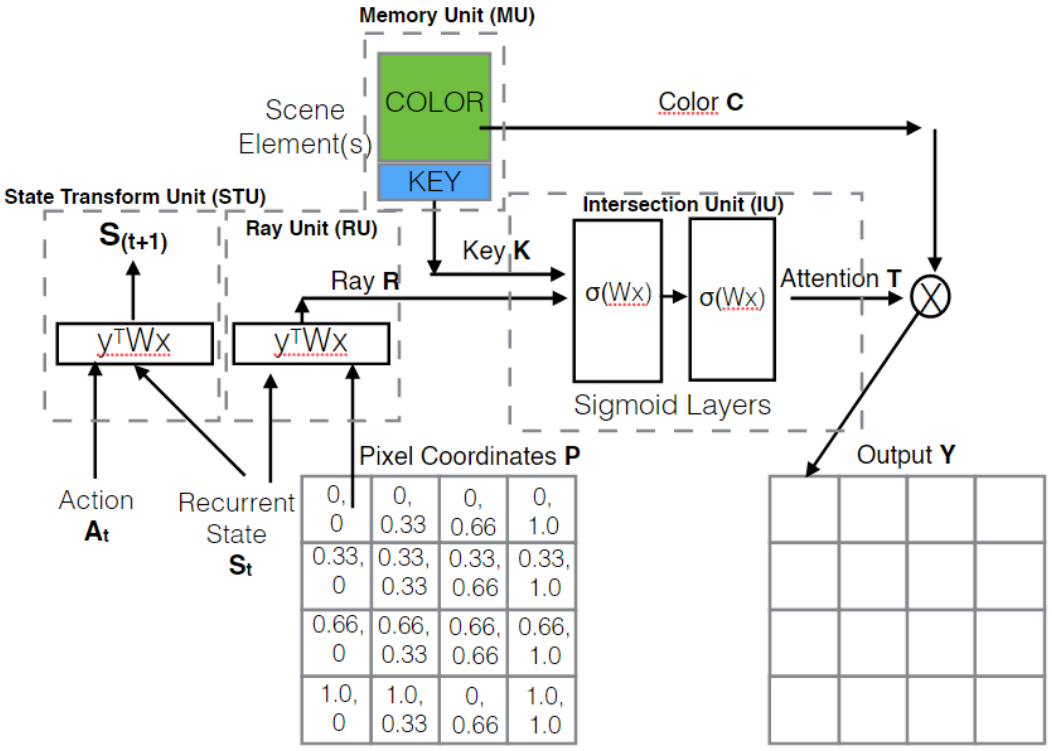
Figure 5.6: **28x28 worlds used in our experiments** - White indicates start position of agent. Green and Yellow are marker locations. Red locations are failures. Blue locations are all successes. Gray areas in kernel are visible to the agent. White cell in kernel is the agents position. Shown optimal path considers MDP deduction as a sub-problem.

CHAPTER 6

PARTIALLY-OBSERVABLE SPATIAL REASONING

6.1 Action Conditional Projection Neural Network Architecture

Figure 6.1: Action Conditional Projection Neural Network Architecture illustration



As explained in the *Motivation* section, the final *output* architecture of the ACPNN is shown in Figure 6.1.

The equations for the ACPNN structure can be laid out as follows:

$$s_0 = 0$$

$$s_{t+1} = s_t \cdot W_t^T \cdot a_t$$

$$r_t = s_t \cdot W_n^B \cdot p_{x,y}$$

$$h_t = \{relu(W_{l1}.[r_t, m_i]) | \forall m_i \in M\}$$

$$o_t = W_{l2}.h_t$$

$$a_t = softmax(h_t)$$

Some of the design choices are presented below, with appropriate justification

6.1.1 State Transform Unit

The State Transform Unit (STU) is primarily this assignment:

$$s_{t+1} = s_t \cdot W_t^T \cdot a_t$$

This structure also follows our assumption that the internal state transforms either *linearly* (Full free motion). If the motion is *piecewise* linear a different formulation is used (Covered in the section on *ACPNN-ReLU*).

A very similar transformation unit has been used in the Action-Conditional Video Prediction paper (1), where a linear transformation of the state vector S is used to introduce the effects of taking an action on the output.

Like the case of Action Conditional Video Prediction network, the ACPNN also has to learn the implicit notion of a state from scratch although this is harder in the latter scenario since the motion is very complicated.

6.1.2 Ray Unit

The Ray Unit (RU) is in-charge of combining the agent's current state s_t and the pixel coordinates $p_{x,y}$ to create an intermediate representation $r_{t,x,y}$(which we refer to as the ray element, because it's equivalent to a ray in *ray tracing*).

In order to maximize the potential for generalizing, the unit (like the STU) uses a simple

linear transform.

$$r_{t,x,y} = s_t.W_n^T.p_{x,y}$$

This is an intuitive decision, since the pixel coordinates $p_{x,y}$ vary uniformly from $-1to1$, using a linear unit ensures the ray elements are also linear in the pixel number.

6.1.3 Memory Unit

The Memory Unit (MU) is used to store the colors and (potentially) patterns that occur in the *scene*. Note the stress on the word *scene*. The memory unit is independent of the agent's current state or pixel coordinate.
In our simple implementation, the Memory Unit stores a set of solid colors C along with a representation vector k for each color. The vector intuitively represents the location of that color in the scene (this is like the attention-based *key* used in most *memory networks*). Currently, all our implementations learn a static set of colors C and representation vectors k for each environment (they are treated as regular parameters).

6.1.4 Intersection Unit

The IU is represented by the following transformation:

$$h_t = \{relu(W_{l1}.[r_t, m_i])|\forall m_i \in M\}$$

$$o_t = W_{l2}.h_t$$

The Intersection Unit (IU) is the most important part of the ACPNN, since it's target is generally *not* linear (unlike the other units). Under the assumption that the Ray Unit is *linear*, no matter what representation is used to represent the *object* (for simplicity, assume a *straight line segment*), the function to test for intersection is necessarily *quadratic*.

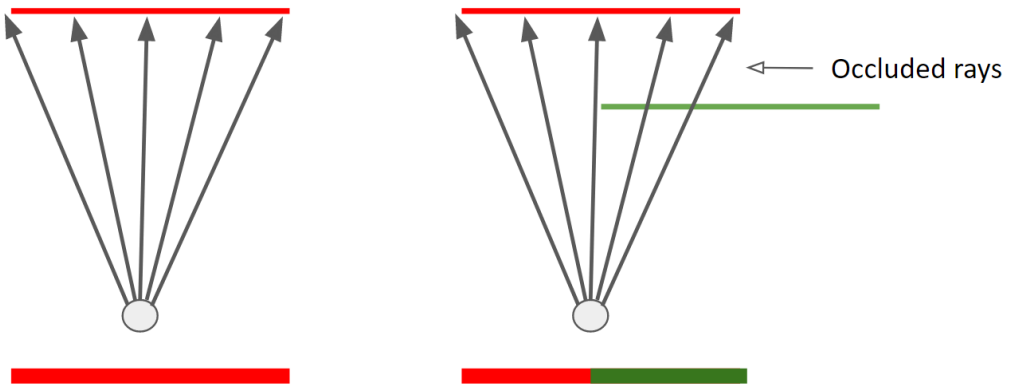
6.1.5 Competitive Attention

Instead of a standard sigmoid layer at the end of the intersection unit (IU), a softmax with variable temperature (competitiveness) was used. This notion, at first, seems out of place in the narrative of ray tracing, because intersection of one scene element is independent of the others.

$$o_t = W_l 2.h_t$$

$$a_t = softmax(h_t)$$

Figure 6.2: Illustration of the concept of occlusion



However, the consideration of *occlusion* makes things more complex, as the intersection equation for the occluded scene element SE_o should remain the same, but the element in front SE_f should have a much higher attention.

The softmax function theoretically allows this to happen, by making sure the final layer output is inversely proportional to the *depth* of the intersection.

The main problem here is that, as the softmax values get sharper, the learning gets slower and if the error manifold is incredibly complicated (in this case, it is), unlearning

a wrong set of weights is much harder.
The assumption that the values of $o_t(.)$ are such that the softmax distribution is sharp automatically means that it has to deal with this treacherous situation.

In order to avoid this, the softmax function can be modified slightly to allow the output layer to stay within reasonable values.
This works by splitting the values into two steps: When calculating the *gradient*:

$$a_t = softmax(h_t)$$

When calculating the *error function*:

$$a_t = \frac{h_t - max(h_t)}{h_t - max(h_t)} + 1$$

which is the sharp maximum value.
This prevents saturation of the softmax values by nullifying the gradients as soon as the correct order of attention values has been learnt.

6.1.6 Controlling Overflow

The softmax function used above has numerical stability issues: Since o_t is unbounded, the values can get unnecessarily large, often causing a floating point overflow (and sometimes NaNs).
To get around this, the ACPNN uses a modified softmax function that does not affect the final outcome but ensures that values are small enough to prevent floating point overflow.
Instead of this,

$$a_t = \frac{e^{h_t^i}}{\sum_i e^{h_t^i}}$$

the model uses this,

$$a_t = \frac{e^{h_t^i - max_i(h_t^i)}}{\sum_i e^{h_t^i - max_i(h_t^i)}}$$

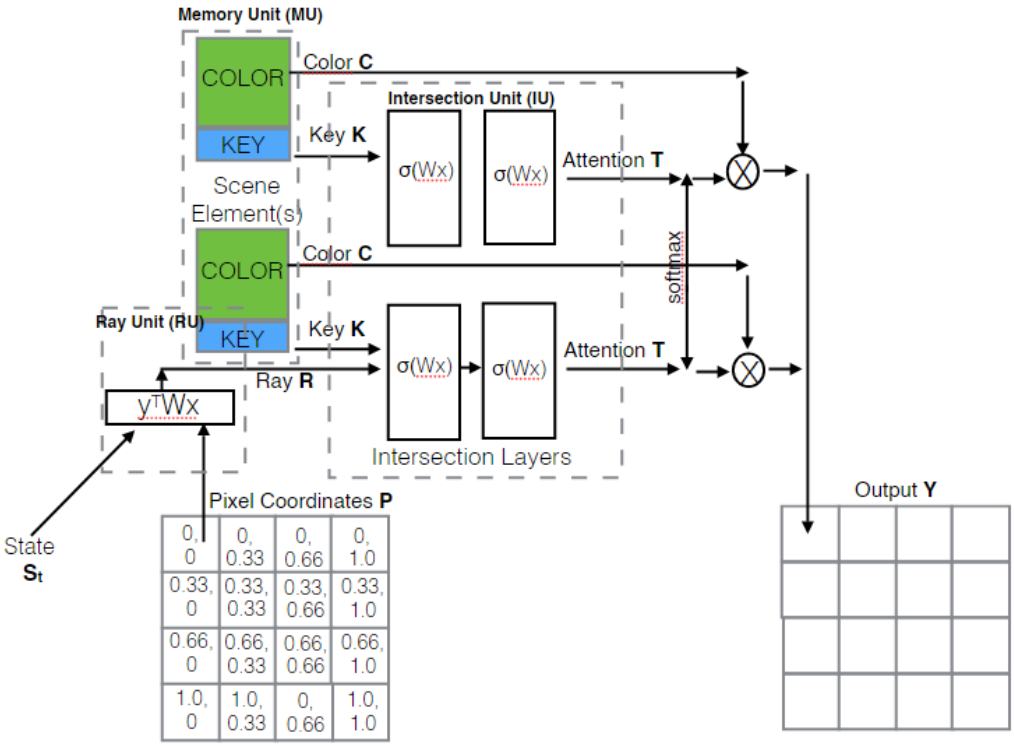
Note that the numbers are now either in the normal range, or too *small* (their exponent vanishes), which restores numerical stability without compromising on accuracy.

6.2 StateFul Projection Neural Network

The StateFul Projection Neural Network (SFPNN) is a *spin-off* of the ACPNN, but without the state transform unit (STU).

The network, in this case, is used to gauge the performance boost given by the STU and the extent to which learning the dynamics of the environment slows down the overall learning.

Figure 6.3: StateFul Projection Network illustration (Only the STU has been changed)



As shown in the figure above, the network does not have an STU. The state s is provided as an input. The state used is the vectorised form of the location and orientation i.e., the orientation is represented as the components of the normalised vector in that direction (for instance, along Y axis is $0, 1$). This is important, as orientation is a special type of parameter that is circular, and using a non-continuous form like degrees/radians will likely lead to bad generalisation as it's not obvious that 360^o is the same as 0^o.

6.3 ACPNN(ReLU)

To provide a solid basis for using ReLU DNNs for the State Transform Unit (STU), we look at the results provided by R. Arora (7). ReLU DNNs are capable of exactly fitting piecewise linear functions. In most games, the state transformation is inherently piece-wise iinear. ReLU DNNs are therefore likely to learn such a system faster than conventional DNNs.

As stated in (7), the following section provides the definition for Piece-wise linear functions (PWL) and Piece-wise linear motion.

6.3.1 Piece-wise Linear Motion

Definition 6.3.1 ***Piece-wise Linear Functions*** *We say a function* $\mathbf{f} : \mathbf{R^n} \mapsto \mathbf{R}$ *is continuous piece-wise linear (PWL) if there exists a finite set of closed sets whose union is* $\mathbf{R^n}$*, and* $\mathbf{f}$ *is affine linear over each set (note that the definition automatically implies continuity of the function). The number of pieces of* $\mathbf{f}$ *is the number of maximal connected subsets of* $\mathbf{R^n}$ *over which* $\mathbf{f}$ *is affine linear.*

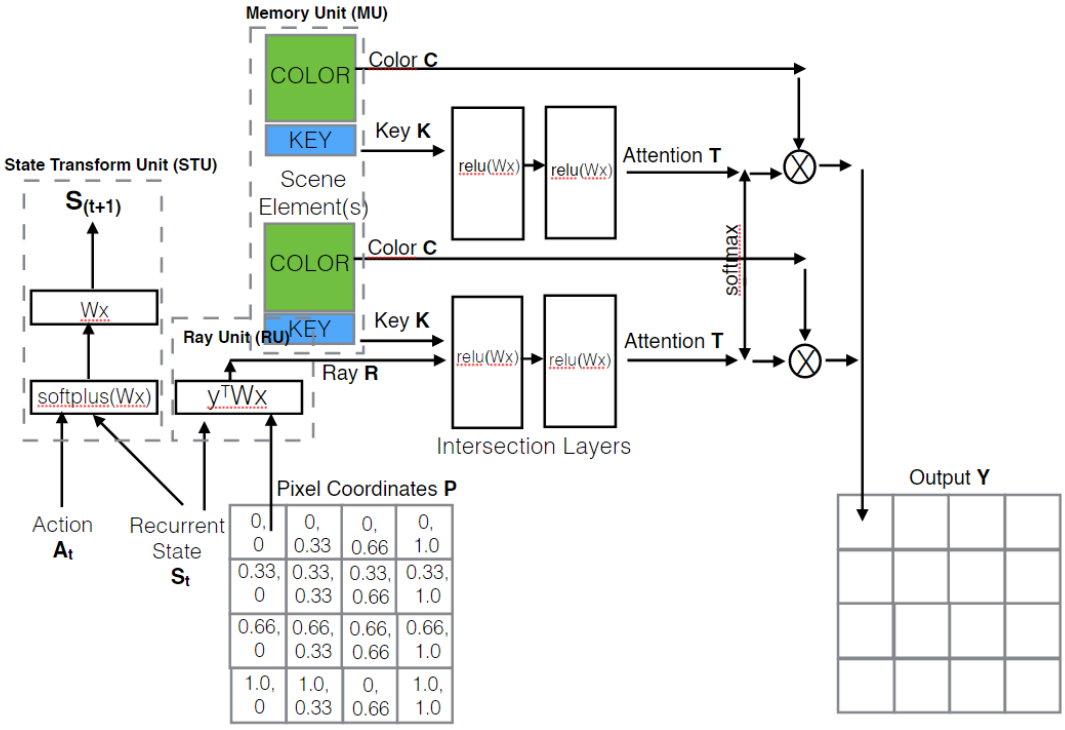
Definition 6.3.2 ***Piece-wise Linear Motion*** *Any motion simulator needs to transform an input state in* $\mathbf{R^n}$ *to an output state in the same* $\mathbf{R^n}$*. by extending the definition for PWL to the case where the function is a multi-dimensional mapping* $\mathbf{f} : \mathbf{R^n} \mapsto \mathbf{R^n}$*, we get piecewise linear motion. Note that, in this case, the function* $\mathbf{f}$ *is PWL in each of the* n *output dimensions.*

The problem of an agent constrained in a box follows the piece-wise linear motion.

6.3.2 Architecture

The main problem with the linear STU used in the original ACPNN form was that it couldn't efficiently model complex dynamics like hitting walls. Upon trying to move through wall, the internal state shouldn't change, but the linear operator cannot model this behaviour without also affecting the case where the agent *hasn't* hit a wall.
Thus, like the dynamics simulator used in (3), the STU unit was changed to use a simple

Figure 6.4: ACPNN with a ReLU-based STU and IU



neural network with ReLUs. Since ReLU-based deep neural networks are essentially piecewise *linear* (7) in nature, modelling a piece-wise linear transformation is a naturally easy task for a ReLU network.

6.4 ACPNN(ReLU)-DQN

6.4.1 Random Paths

To train the ACPNN, the strategy so far was to observe several randomly sampled paths (limited by a maximum number of steps). This strategy has obvious shortcomings, in that certain corner cases may be erroneously modelled, but these cases may not be observed often enough.

Since the current problem statement doesn't have a environmental reward signal because the goal here is to predict the frames, we need to devise an internal reward notion to allow the model to explore the environment better.

6.4.2 Prediction Error

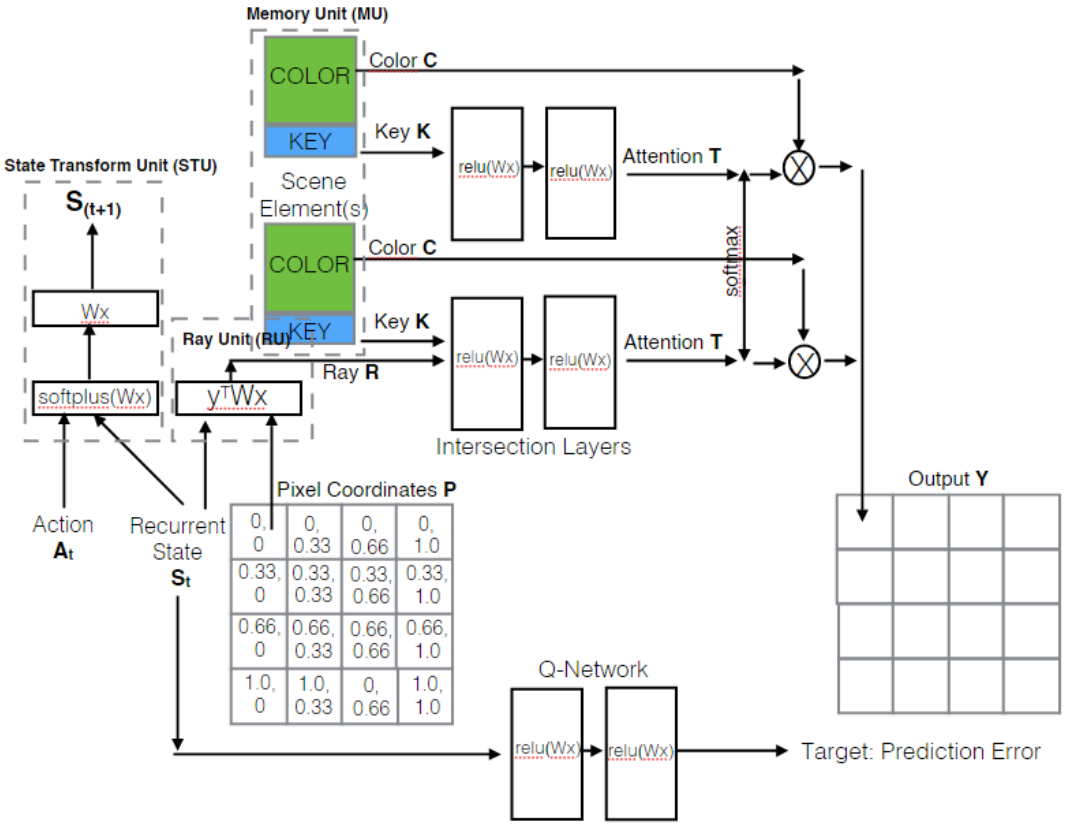
The ACPNN-DQN model takes inspiration from the paper on *Incentivising Exploration*. The *prediction error* in this case is the sigmoid of the difference between the expected output frame and the actual frame observed from the environment.

$$R(s_t) = sigmoid((o(s_t) - y_t)^2)$$

The sigmoid is necessary to bound the error terms and to prevent instability in the DQN, which, in it's original form, is known to poorly handle rewards with a large range.

6.4.3 Architecture

Figure 6.5: Spatial Reasoning Framework



The ACPNN-DQN is a version that uses a Q-network to model the *prediction error*

in the ACPNN's predictions.

The Q-network takes the internal state s_t as the input, and the target is the prediction error $R(s_t)$ (as detailed in the above section).

The training procedure that we followed to stabilise the training is as follows:

Algorithm 5 Pseudo-code that illustrates the algorithm used by ACPNN-DQN

1:
2: **procedure** TRAINACPNN-DQN
3: Initialize weights randomly for DQN $D()$
4: Initialize weights randomly for ACPNN $P()$
5: **for do**i from 1 to k:
6: Sample a set of trajectories T following $D()$
7: Train A on T for 500 epochs (just enough for some reasonable state embedding)
8: Train D for 10,000 epochs with internal state embedding S as the input and the prediction error $R(S)$ as the target (if necessary, fully unlearn the previous Q network weights)

The following approximate embedding shows the Q-values at various locations. The Q-values were summed over all *orientations* of the agent, for each location.

6.4.4 Instability Issues

Dealing with instability was a major concern while building the ACPNN-DQN.

The first concern is the sampling mechanism. Two alternatives are available:

1. ϵ-greedy: Sampling using the ϵ-greedy strategy is one way to ensure the network didn't get eternally trapped in one particular action (because of instability). Unfortunately, the greedy action is sometimes very close (in Q-value) to plenty of other actions which got overruled too often to be of considerable use.

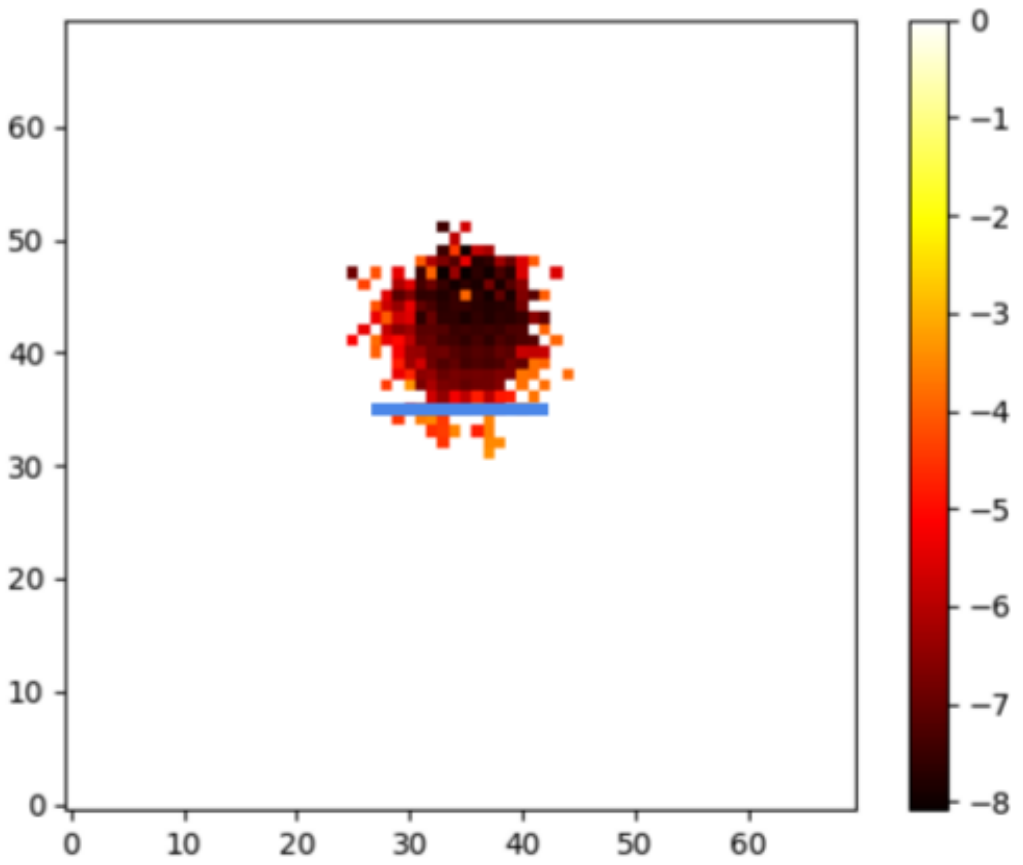
2. Boltzmann sampling: Boltzmann sampling simply uses the softmax function as the probability distribution. The main problem with this is that when the network becomes unstable and diverges, every action is the exact same, since that action has a value so large, it over-rules every other action.

Ultimately, Boltzmann sampling makes more sense, since the source of instability isn't the sampling, but rather the quality of the state embedding.

The better the quality of the state embedding (more training epochs for the ACPNN), the more stable the Q-network is.

The reason for running the DQN part for a very large number of epochs is because a

Figure 6.6: Sample Q-value field (white values are states that are never visited). The blue line represents the *red-line* object in the Redline world.



regular Q-network faces only a non-stationary target, while the network in ACPNN-DQN also faces a non-stationary *input* (ever-changing state embeddings). Thus, in case the state embeddings change drastically, the Q-network has to catastrophically unlearn everything.

6.5 Input-ACPNN for Representation Learning

6.5.1 Representation Learning

The challenge in representing an entire scene (and it's dynamics) with a single vector is that the entire scene is not visible at any one point and that the dynamics of the system have to be explicitly observed and modelled.

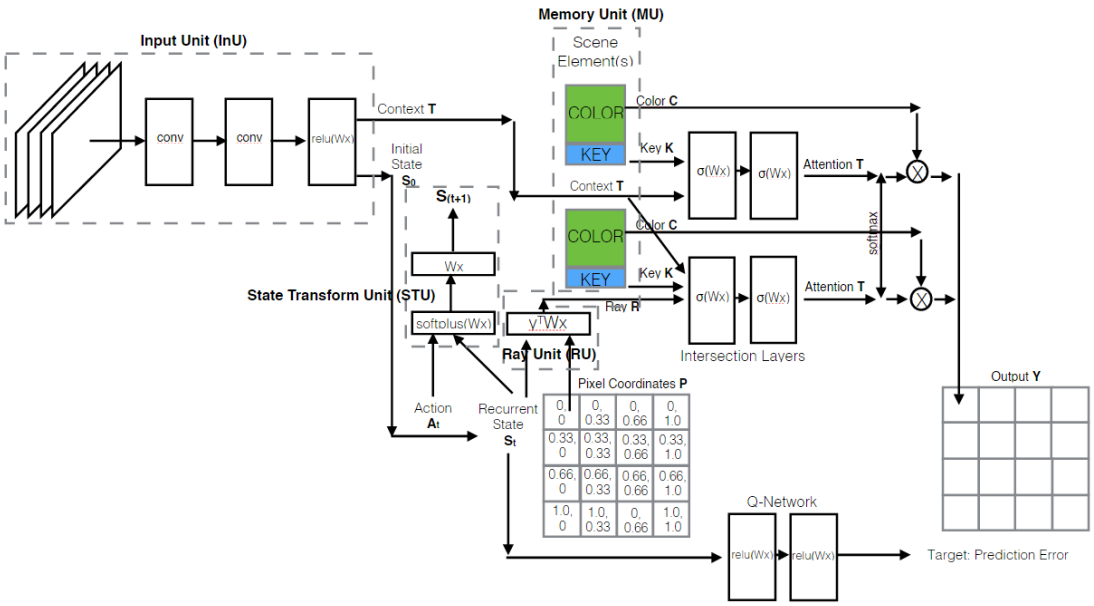
Therefore, to enforce representation learning, we use a convolutional network to process the first K frames of the agent and generate two vectors (initial state s_0 and scene context

t) out of which the latter, scene context t is the vector representation of the scene. The former value, initial state s_0 is used by the STU as the initial position of the agent. Note that the convolutional network has to identify the scene *and* identify the location of the agent within the the scene. Unlike an autoencoder, the architecture is not a simple one. The architecture presented below makes the most of both worlds by introducing fusing the first half (input part) of a convolutional auto-encoder and the ACPNN.

6.5.2 Architecture

The architecture shown below achieves all the requirements stated above. Since this is the first iteration that includes the input too, the problem statement has to be slightly changed. The agent now takes n steps from a random initial state s_0 to generate K frames as the input. The task for the Input-ACPNN is to predict the next $n_\prime$ frames.

Figure 6.7: Input ACPNN



The Input-ACPNN can not only be used to create complete model of a scene (and it's dynamics), but it can also be used in multi-task (MTRL) scenarios, where there are multiple environments and it's the agent's job to figure it out from the first n frames, and predict the next n'.

This can be applied to scenarios such as the I-world (6) where the agent can figure out the identity of the environment from the color of the tiles and the apparent distance from the end of the corridor.
The corridor problem in the I-world is interesting as it tests the generalisation ability of the agent. The use of a vector intermediate representation means that the Input-ACPNN can achieve this level of generalisation.

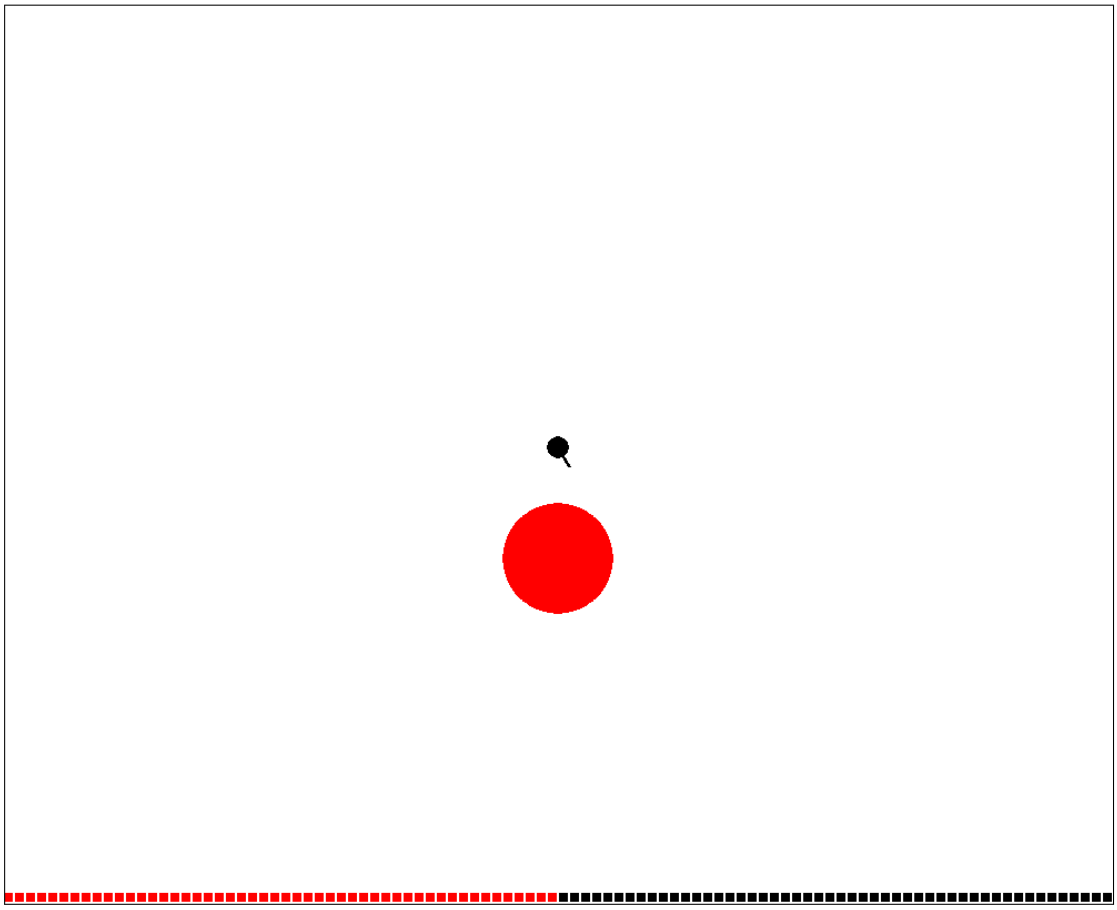
6.6 Testbed

For the testbed, currently the world used is a 2D one (with a 1D projection).
This is because when moving from 2D to 3D worlds, the ACPNN, SFPNN, ACPNN(ReLU) and ACPNN(ReLU)-DQN architectures all do not change their structure at all (except for an increase in the size of pixel coordinates P), unlike convolutional networks, where the structure of the filters changes a lot.

The use of pixel coordinates instead of convolutional filters means that extending ACPNN to 3D worlds simply takes longer to train rather than any increase/decrease in performance.
In practice, the 2D worlds tend to be harder to learn because of much less data per trajectory.
To demonstrate usefulness, one of the test cases used is a simple 3D world.

6.6.1 Red Circle World

Figure 6.8: **Red Circle World**
The agent's position is represented by the little black dot and the agent's view direction is represented by the black line on the dot.
The set of boxes denote the **100x1** image observed by the agent (Black represents no object in that direction)
The red circle is the 2D object whose 1D projection the agent witnesses



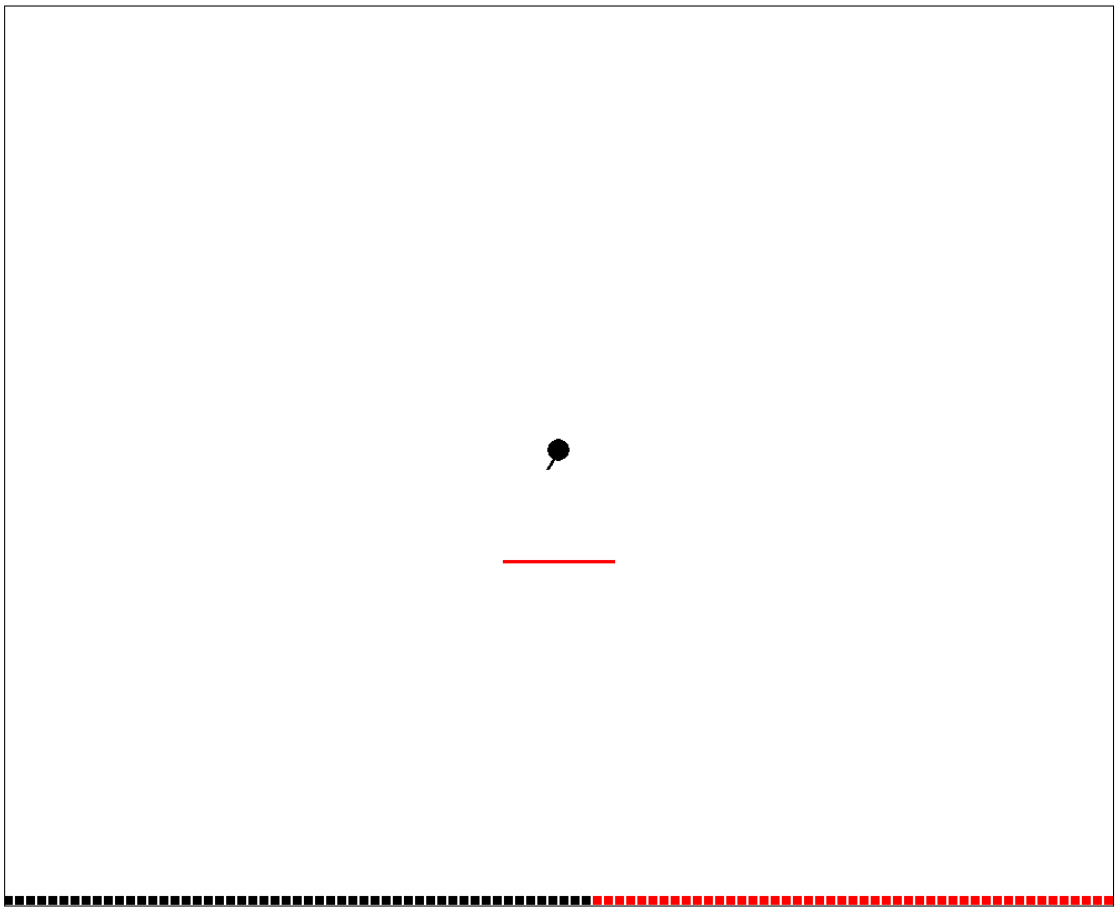
This world contains a single red circle at the center, with the agent having full, unrestricted motion.

Intuitively, all the intersection unit (IU) has to do is test for whether the ray was at a given distance from the center point and test it against a threshold value to determine the attention for that pixel.

Unfortunately, if we follow the linearity principle of keeping the intersection tests piecewise linear (to offer the neural network an easy, generalizable target), the red circle test does not qualify (the distance of a line from a point is *not* linear in the ray element R). The network thus has a harder time generalizing to it.

6.6.2 Red Line World

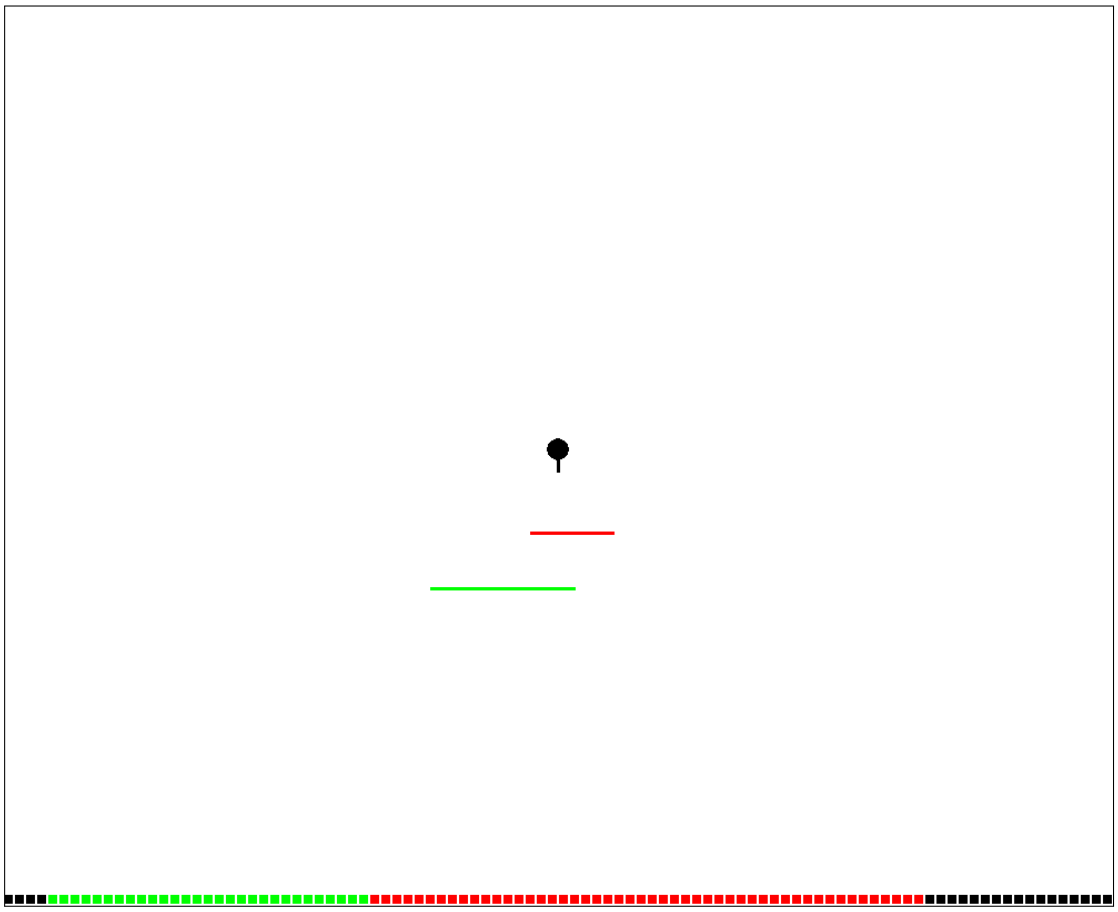
Figure 6.9: **Red Line World**
For details, refer to description for Figure 6.8



As described above, the Red Line World contains a single red line at the center, with the agent always starting at the same point (full freedom of movement).

6.6.3 Double Line World

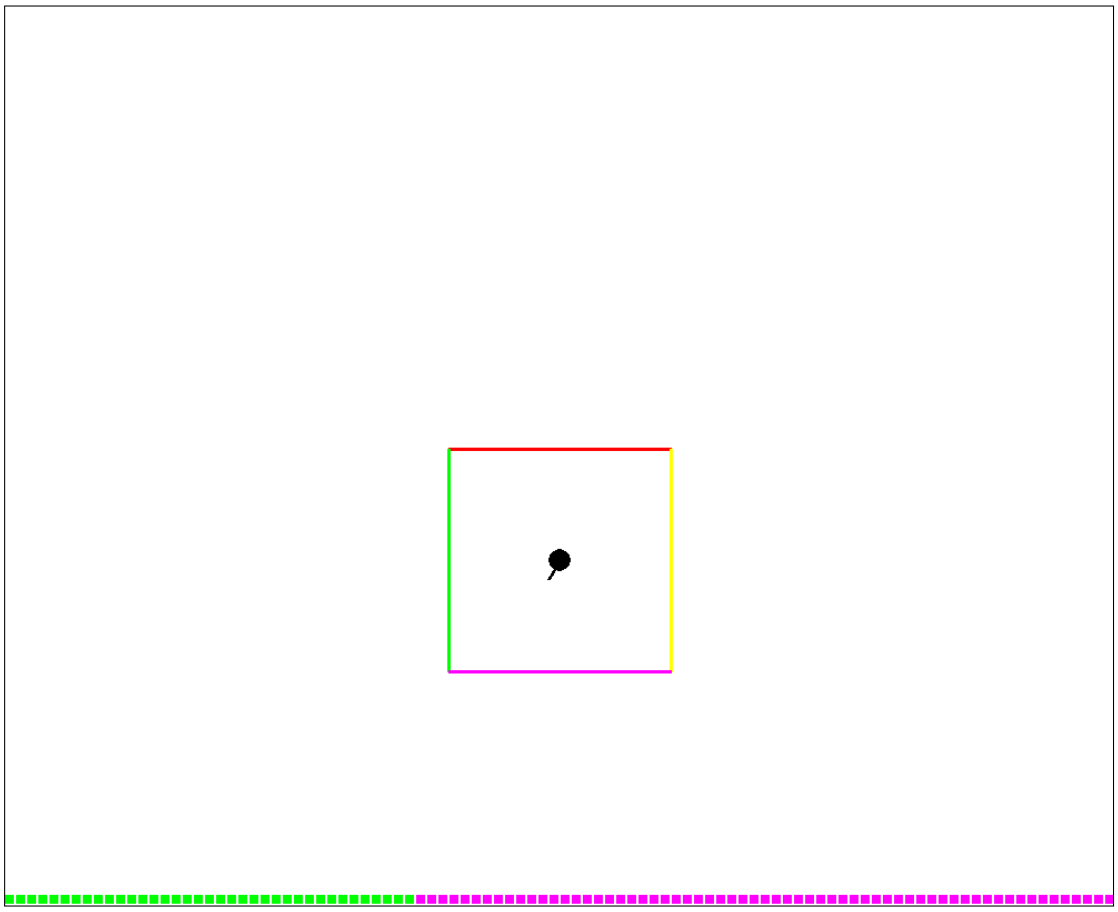
Figure 6.10: **Double Line World**
For details, refer to description for Figure 6.8



The double line world was developed to test the agent's ability to learn occlusion and use the softmax layer to ensure depth sensitivity. It consists of one red line and one green line.

6.6.4 Box World

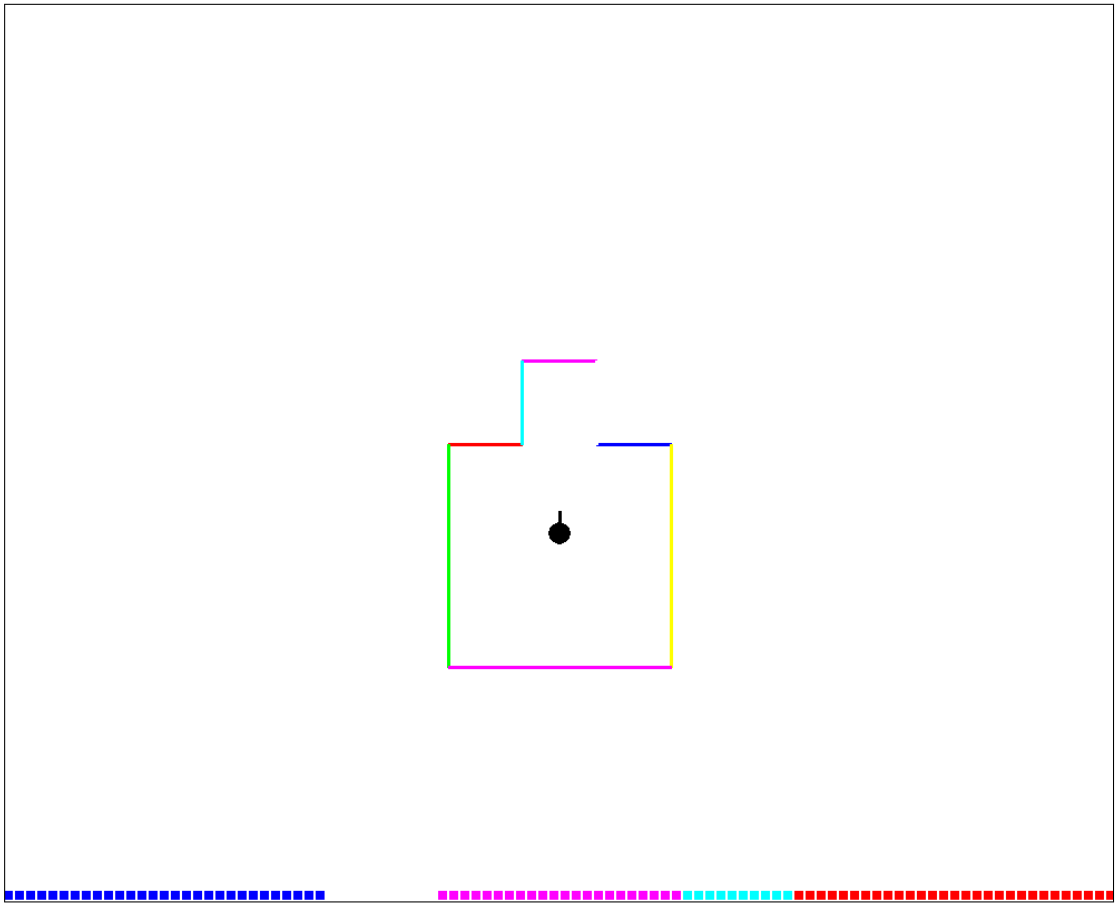
Figure 6.11: **Box World**
For details, refer to description for Figure 6.8



The box world was developed to test the effectiveness of the ACPNN(ReLU)'s State Transform Unit when applied to a 4-wall box world where the agent cannot move through walls. The agent's motion is constrained to within the 4 walls in this case.

6.6.5 Complex Box World

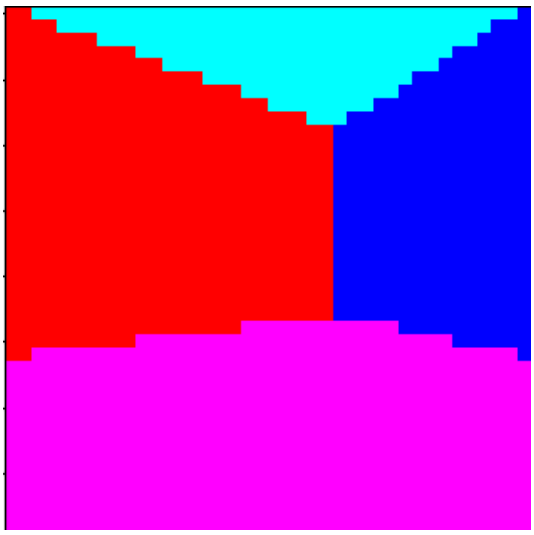
Figure 6.12: **Complex Box World**
For details, refer to description for Figure 6.8



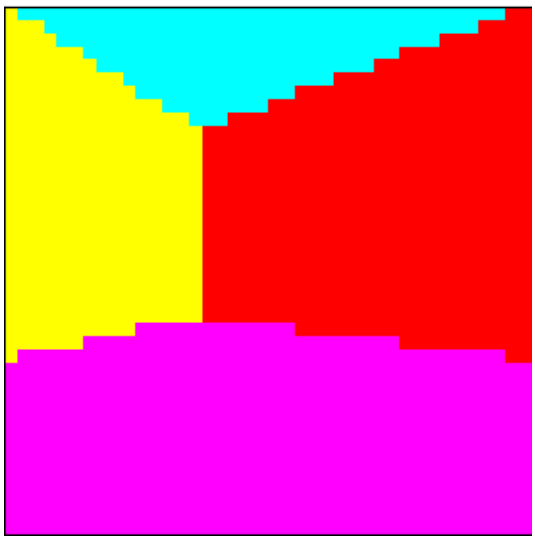
This world is a combination of the Box World and the Double Line world in that it tests for both occlusion and the ability to handle constrained motion.

6.6.6 3D world

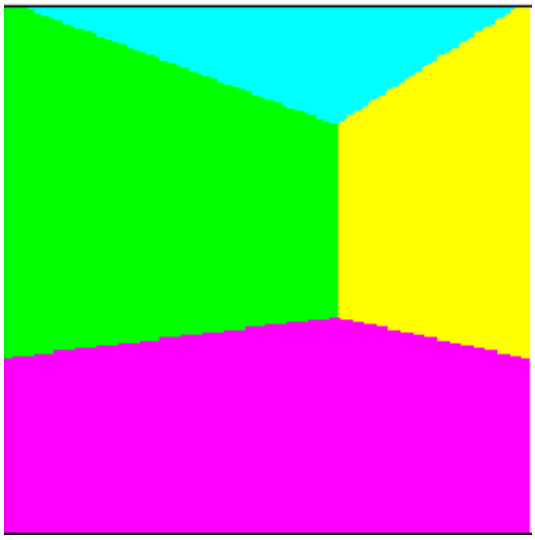
Figure 6.13: **3D World**: Various views of the 3D world



(a) **3D World Perspective Projection 40x40**



(b) **3D World Perspective Projection 40x40**



(c) **3D World Perspective Projection (High Resolution) 200x200** - The high resolution image is for clarity only. For performance reasons, the 40x40 down-scaled images are the images used for training

In order to show good performance in more realistic scenarios, we use a simple 3D world with 90^o FOV persepctive projection. Constrained motion still applies, with the agent unable to cross the walls. Currently, the 3D world is a simple closed box and the agent cannot look vertically up or down nor can it move in the vertical direction.

6.7 Results

For each test, the learning is performed through Stochastic Gradient Descent, by *randomly* sampling 200 data points for each gradient update. The optimizer used for computing adaptive gradients is *"Adam"* (22).

Figure 6.14: **ACPNN performance comparison**: *% Adjusted Error* versus *number of epochs* - Tests were performed with constant set of hyper-parameters over various environments

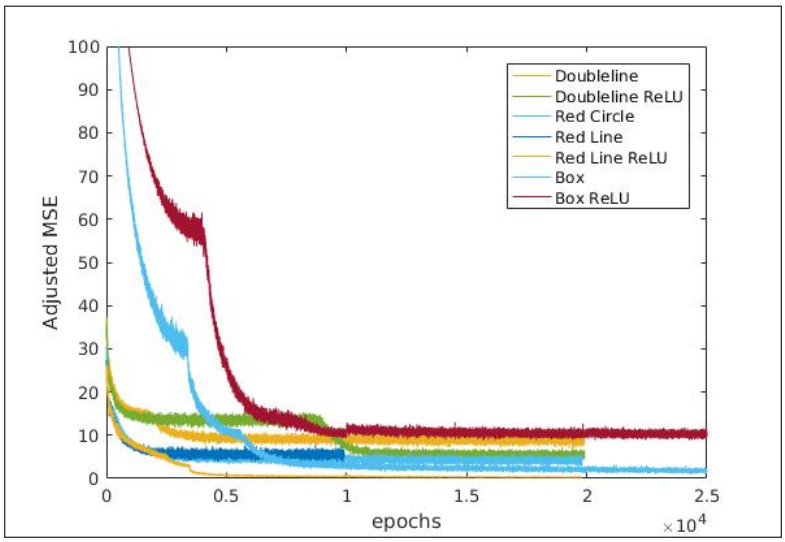
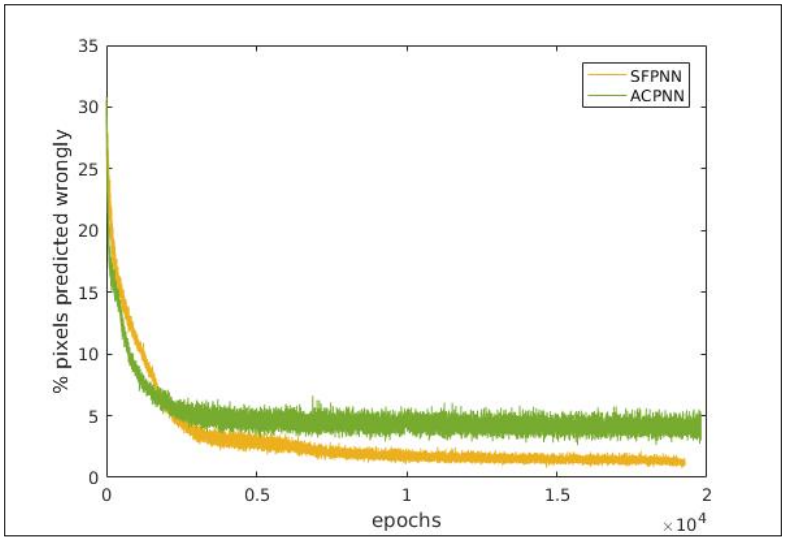


Fig. **??** shows the loss function of the DQN part of the ACPNN-DQN architecture. Note the sudden crest at the beginning followed by gradual stabilization. We noticed that this behaviour is quite common with most of the environments that were used for training.

The comparison between the performance of ACPNN (vanilla variant) on various environments is shown in Fig. 6.14.

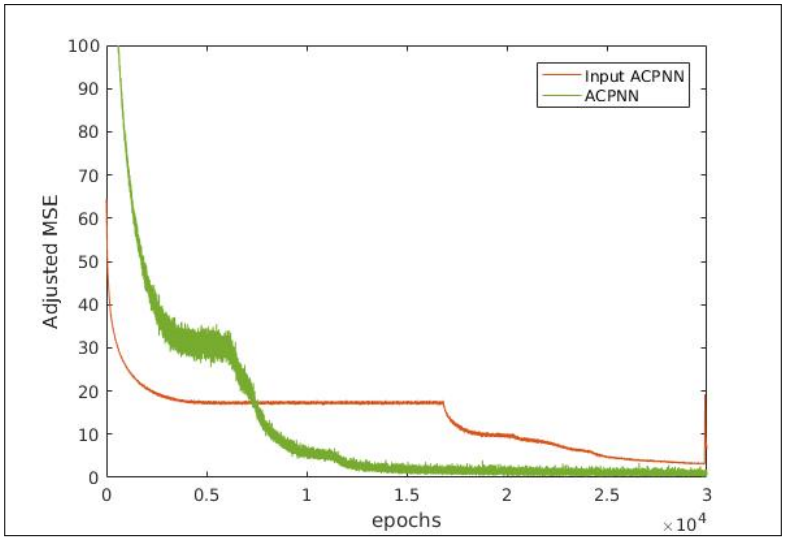
Figure 6.15: **SPNN performance**: *% q value* versus *number of epochs* - Note the inherent instability of the Q network. This is because both the targets *and* the inputs are *non-stationary*



In order to test the relative difficulty of learning the State Transformation Unit and the rest of the architecture, we compare the performance of the Stateless Projection Neural Network (SPNN) with the regular ACPNN.

The results in Fig. 6.15 show a definite improvement in the performance of the network when the internal state of the agent is available.

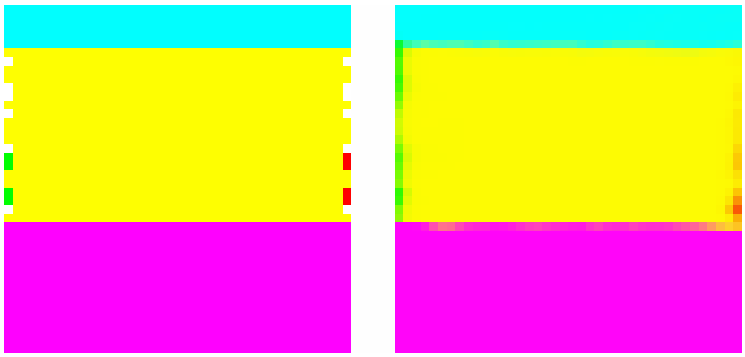
Figure 6.16: **InputACPNN performance**: A comparison between vanilla ACPNN versus the Input ACPNN architecture on the same 3D Box World. Note that, in the latter case, the agent starts off in a completely random orientation and the first frame is presented to the agent before the first action is taken



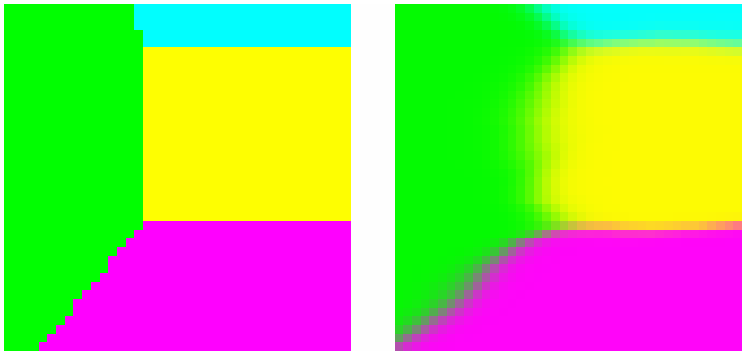
We tested the performance of Input-ACPNN versus the regular ACPNN to test the

effect of having to infer the starting location from the initial frame. Fig. 6.16 shows this effect and the extent of the additional complexity the Input-ACPNN faces when compared to the vanilla ACPNN.

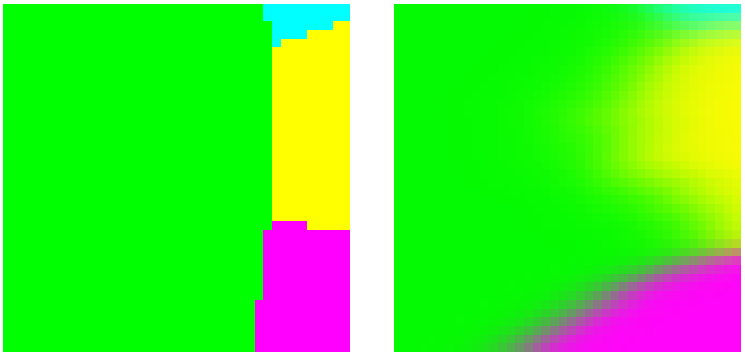
6.7.1 Visualizations



(a) **View reconstruction of the 3D Box World after 1 step** The reconstruction is on the *right* while the actual observation (reference) is on the *left*



(b) **View reconstruction of the 3D Box World after 5 steps** The reconstruction is on the *right* while the actual observation (reference) is on the *left*



(c) **View reconstruction of the 3D Box World after 11 steps** The reconstruction is on the *right* while the actual observation (reference) is on the *left*

On the most complex world, Fig. 6.17a, Fig. 6.17b and Fig. 6.17c shows the reconstructions made by the ACPNN architecture.

CHAPTER 7

CONCLUSION

In this thesis, we have presented a set of novel approaches to modern Reinforcement Learning. We have first proposed a deep generative model to understand and differentiate a set of tasks in the fully observable multi-task reinforcement learning scenario and a set of exploration bonuses that all the agent to take advantage of the model. At this time, when reinforcement learning agents attempt to tackle 3D partially observable worlds, we have proposed another novel method, inspired by computer graphics, to interpret the sequence of images generated by projecting the scenes onto a plane.
While our deep generative model showed excellent potential to work well on *Map-like environments*, it's application is still limited to agents in *fully observable environments*. This is the main reason for the shift in focus to the ACPNN architecture.

As far as we are aware, there is no other architecture that uses a model specifically built to generalize spatial projections. Although the model, in its current state, is slower than existing methods, this thesis shows potential for the method to generalize to a greater extent compared to convolutional networks. One of the model's drawbacks was the difficulty in stabilizing learning. Instead of a smooth, consistent curve, the model often had long idle periods followed by a sudden moment of learning.

In the future, it is likely that deep convolutional networks are not the *be-all-end-all* of machine learning, and specialized architectures will be developed for every domain. This thesis makes the case that a ray-tracing inspired neural network, with some more refinement, will be the state-of-the art in interpreting 2D images of spatial 3D scenes.

REFERENCES

[1] Junhyuk Oh Xiaoxiao Guo Honglak Lee Richard L. Lewis Satinder Singh *Action-Conditional Video Prediction using Deep Networks in Atari Games*. NIPS Advances in Neural Information Processing Systems 28 pp. 2863-2871, 2015

[2] Aaron Wilson Alan Fern Soumya Ray Prasad Tadepalli *Multi-Task Reinforcement Learning: A Hierarchical Bayesian Approach*. ICML Proceedings of the 24th international conference on Machine learning pp. 1015-1022, 2007

[3] Bradly C. Stadie, Sergey Levine, Pieter Abbeel *Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models*. ICLR 2016

[4] Geoffrey Hinton *A Practical Guide to Training Restricted Boltzmann Machines*. Neural Networks: Tricks of the Trade (2nd ed.) pp. 599-619, 2012

[5] Sutton, R. S. Barto, A. G. *Reinforcement learning: an introduction.* . Cambridge, MA: MIT Press. 1998

[6] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, Honglak Lee *Control of Memory, Active Perception, and Action in Minecraft*. ICML Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, pp. 2790-2799, 2016

[7] Raman Arora, Amitabh Basu, Poorya Mianjy, Anirbit Mukherjee *Understanding Deep Neural Networks with Rectified Linear Units*. arxiv.com preprint Nov. 2016

[8] Sai Praveen B, JS Suhas, Balaraman Ravindran *Exploration for Multi-task Reinforcement Learning with Deep Generative Models* NIPS DRL Workshop 2016

[9] Aviv Tamar, Sergey Levine, Pieter Abbeel *Value Iteration Networks* arxiv.com preprint 2016

[10] Diederik P Kingma, Max Welling *Auto-encoding Variational Bayes* arxiv.com preprint 2013

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller *Playing Atari games with Deep Reinforcement Learning* arxiv.com preprint 2013

[12] Alessandro Lazaric, Mohammad Ghavamzadeh. *Bayesian multi-task reinforcement learning* 27th ICML Omnipress pp. 599– 606. 2010

[13] Taco S. Cohen, Max Welling *Transformation Properties of Learned Visual Representations* ICLR 2015

[14] Arthur Appel *Some techniques for shading machine renderings of solids* AFIPS '68 (Spring) Proceedings of the spring joint computer conference, pp. 37-45, 1968

[15] Ingrid Carlbom, Joseph Paciorek *Planar Geometric Projections and Viewing Transformations* ACM Computing Surveys 10 (4): pp. 465–502, 1978

[16] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver *Prioritized Experience Replay* ICLR 2016

[17] Martin Riedmiller *Neural Fitted Q-Iteration* ECML Proceedings of the 16th European conference on Machine Learning, pp. 317-328, 2005

[18] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, Rob Fergus *Deconvolutional Networks* IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp.2528-2535, 2010

[19] Robert Gens, Pedro Domingos *Deep Symmetry Networks* NIPS Proceedings of the 27th International Conference on Neural Information Processing Systems, pp. 2537-2545, 2014

[20] Geoffrey E. Hinton Alex Krizhevsky Sida D. Wang *Transformation Auto-encoders* ICANN Proceedings of the 21th international conference on Artificial neural networks, pp. 44-51, 201‘

[21] Emilio Parisotto, Jimmy Lei Ba, Ruslan Salakhutdinov *Actor-Mimic: Deep Multi-task and Transfer Reinforcement Learning* ICLR 2016

[22] Diederik P. Kingma, Jimmy Ba *Adam: A Method for Stochastic Optimization* ICLR 2015

Papers in Workshops

1. Exploration for Multi-task Reinforcement Learning with Deep Generative Models NIPS DRL 2016
   Sai Praveen B, JS Suhas, Balaraman Ravindran