NTNU
Norwegian University of
Science and Technology

Data Efficient Deep Reinforcement Learning through Model-Based Intrinsic Motivation

Mikkel Sannes Nylend

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Massimiliano Ruocco, IDI
Co-supervisor: Arjun Chandra, Telenor
Humberto Castejon, Telenor

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In the last few years we have experienced great advances in the field of reinforcement learning (RL), much thanks to deep learning. By introducing deep neural networks in RL it is possible to have agents learn complex behaviors by just observing a game screen, just like humans learn to play games. Even though this is great, there is one limitation that makes the transition to real world problems tough, and that is data efficiency.

One way to go about improving the data efficiency of RL is to approximate a model of the environment, called model-based RL. Even though model-based agents can be more data efficient, they are usually computationally heavy and often end up being too inaccurate. In this thesis, we explore the use of deep dynamics models (DDM) trained dynamically in environments with high-dimensional state representations. Furthermore, we study four different ways of calculating curiosity-based intrinsic motivation extracted from the DDM to achieve more efficient exploration.

Having the DDM made up of an autoencoder (AE) and a transition prediction model that operate in the latent space generated by the AE, we introduce the first intrinsic bonus as the AE reconstruction error. The second one is based on the prediction error from the DDM. The third bonus introduce a novel idea of using MC dropout, presented in (Gal & Ghahramani 2015), to extract the uncertainty of the DDM. The last type of intrinsic bonus extract the uncertainty by MC dropout from a bootstrapped DDM. Interestingly, the proposed bonus based on MC dropout outperforms the more commonly used bonus based on dynamics prediction errors in both data efficiency and final performance in the Atari 2600 domain. Additionally, we manage to have agents learn by only receiving intrinsic reward and no any extrinsic rewards from the environment.

Sammendrag

I de siste årene har vi opplevd stor fremgang i feltet forsterkende læring (RL), mye takket være dyp læring. Ved å introdusere dype nevrale nettverk i RL har det blitt mulig å trene agenter til å spille spill bare ved å observere skjermen. Dette kan sammenlignes med måten mennesker lærer ved å prøve og feile. Selv om dette høres lovende ut, så sliter fortsatt RL teknikker med å være for data ineffektiv til å løse viktige problemer i den virkelige verden.

En måte å forbedre data effektiviteten på er ved å trene en modell som kan tilnærme seg dynamikken i miljøet som agenten handler i. Dette kalles modellbasert RL. Selv om modellbaserte RL agenter kan være mer effektive, så blir de ofte for tunge beregningsmessig og for upresise til å kunne brukes. I denne oppgaven utforsker vi bruken av dype dynamikkmodeller trent dynamisk i miljøer med høy-dimensjonale observasjoner. Videre studerer vi fire forskjellige måter å beregne nysgjerrighetsbasert indre motivasjon hentet fra den dype dynamikkmodellen for å oppnå mer effektiv utforskning av miljøet.

Den dype dynamikkmodellen er sammensatt av en "autoencoder" (AE) og en overgangsprognosemodell som opererer i den komprimerte plassen generert av AEen. Basert på denne modellen så er den første indre motivasjonstypen basert på rekonstruksjonsfeilen til AEen. Den andre typen baserer seg på prediksjonsfeilen til dynamikkmodellen. Den tredje motivasjonsbonusen introduserer en ny idé om å bruke "MC dropout", presentert i (Gal & Ghahramani 2015), for å hente ut usikkerheten i dynamikkmodellen. Den siste typen indre motivasjon henter ut usikkerheten ved å bruke "MC dropout" i en "bootstrapped" dynamikkmodell. Den nye foreslåtte indre motivasjonsbonusen basert på "MC dropout" utkonkurrerer alle de andre motivasjonsbonusene når det kommer til data effektivitet og endelig poengsum i Atari 2600 domenet. I tillegg kan motivasjonsbonusen få agentene til å lære av seg selv uten å motta noen eksterne belønninger fra miljøet.

Preface

This thesis concludes my Master of Science in Computer Science for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU). The thesis was written the spring 2017 in cooperation with Telenor Research.

I would like to thank my supervisors Massimiliano Ruocco, Humberto N. Castejón and Arjun Chandra for valuable feedback and enlightening discussions. Additionally, I would like to thank Alfredo V. Clemente and Per Torgrim F. Thorbjørnsen for useful inputs in our meetings. A special thanks to Alfredo V. Clemente for sharing his excellent work on PAAC from (Clemente et al. 2017).

Table of contents

List of figures

List of tables

Notations and Abbreviations

Symbol	Description
t	Current time step
s_t	The current state observed in time step t
a_t	Selected action in time step t
r_t	Reward received from the environment after state transition between time step t-1 and t
R^+	Intrinsic reward bonus
π	Action policy for an agent
D_{KL}	Kullback–Leibler divergence (also given as KL divergence)

AE	Autoencoder
AL	Autoencoder loss (intrinsic reward bonus)
ALE	Arcade Learning Environment
BDU	Bootstrapped dynamics uncertainty (intrinsic reward bonus)
CD	Convolutional deconvolutional architecture
CMP	Convolutional max pooling architecture
CPU	Central processing unit
DDM	Deep dynamics model
DL	Dynamics loss (intrinsic reward bonus)
DNN	Deep neural network
DQN	Deep Q-network
DU	Dynamics uncertainty (intrinsic reward bonus)
FC	Fully connected architecture
GP	Gaussian process
GPU	Graphical processing unit
MB-PAAC	Model-based parallel advantage actor-critic algorithm
MDP	Markov decision process
MSE	Mean squared error

PAAC	Parallel advantage actor-critic algorithm (model-free)
RGB	Images with three color channels (red, green and blue)
RL	Reinforcement learning
SL	Supervised learning
TPM	Transition prediction model
UL	Unsupervised learning
VAE	Variational autoencoder
VCD	Variational convolutional deconvolutional architecture

1 Introduction

Artificial intelligence has in recent years experienced tremendous progress and success. One of the most important drivers for this success is the field of deep learning. By using multiple interconnected hidden neural layers trained with stochastic gradient descent, deep learning has provided solutions to problems that previously were unsolvable. Seeing such great progress opens a whole new set of applications, and that is why deep learning has become one of the greatest research priorities for many of the greatest tech companies in the world (e.g. Google, Facebook, Amazon). Most of these applications use a learning technique called supervised learning (SL) which learns by correcting the error between the predicted output of the network with a correct answer. Although, lately, we have seen deep learning being used in the two other categories as well; reinforcement learning (RL) and unsupervised learning (UL). UL tackles problems where there is no information about what to do with the data. Therefore, there are no correct answer to every input like in SL. Basically, the algorithm must make sense of the data all by itself. RL problems are sequential decision problems where the agent tries to learn an optimal behaviour based on a feedback signal from the environment. RL differs from SL and UL in that the agent can start without any data or knowledge about the problem. Hence, the agent must explore the environment to figure out how to reach an optimal behaviour, also called optimal policy.

Deep learning has changed these fields a lot in the recent years, and the big breakthrough for RL came with Deepmind's publication of (Mnih et al. 2013). A general agent capable of learning how to play a huge variety of games by just looking at the image generated from the game screen. The agent was controlled by a deep neural network (DNN) and achieved superhuman performance in many of the games from the Atari 2600 domain. A new era for general complex systems had begun. Compared to the handcrafted specialized program used to beat the world champion Gary Kasparov in chess in 1997 (Campbell et al. 2002), DQN was a steppingstone to the next big release from the same company, AlphaGo (Silver et al. 2016). AlphaGo used multiple general deep learning algorithms from both SL and RL to learn the game of Go[1]. In 2016 they beat one of the top Go players in the world, a decade

[1] The game of Go is an ancient Chinese two player board game where the goal is to surround more territory on the 19 by 19 grid than your opponent by placing stones strategically.

earlier than the experts had predicted. Even though the listed breakthroughs are connected to games, RL can also be used in important real world problems such as self-driving cars (Shalev-Shwartz et al. 2016), advanced robotics (Ng et al. 2006), recommendation systems (Shani et al. 2012), advertising (Cai et al. 2017), dynamic pricing (Raju et al. 2003), chatbots (Li et al. 2016) and power optimization[2].

1.1 Motivation

All the mentioned RL problems have one thing in common, which is that they need very many training steps to learn a decent policy. This is because the agent must explore the environment to learn what works and what does not work for every state-action pair. The number of pairs grows exponentially as the dimensions of states and actions increases. A reduction in the number of training samples could be beneficial in many ways. For instance, in real world problems we do not always have unlimited exploration steps like we have in a simulation. Therefore, the agent should carefully select the actions that maximizes the information about the environment to make efficient exploration. A more data efficient agent would also benefit RL problems that are fully simulated as well. By making smarter actions, the agent might be able to solve more difficult problems than an agent exploring by selecting random actions.

Smart exploration can be done in many ways, but this thesis focus on how an approximated dynamics model of the environment can be used as an information source for the RL agent to explore smarter. When the agent learns a simulation of the environment, the dynamics model, it is known as *model-based RL*. Model-based RL has shown great potential when it comes to data efficiency compared to more commonly used *model-free* RL. Model-free RL learns the policy directly from acting in the unknown environment. However, approximating a model of the environment dynamics is a very hard task, and inaccurate dynamics models may result in worse learning efficiency. As the state space increases to higher dimensional representations as images it becomes notoriously difficult to model changes in an accurate manner. For this thesis, all environment observations are images received from games in the Atari 2600 domain (Bellemare et al. 2015). A well-known idea is to use intrinsic reward bonuses to make the agent perform better. Intrinsic rewards can increase for more novel states and

[2] https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/

decrease for highly visited states, representing a metric for surprise. Using intrinsic motivation to guide the agents towards interesting and novel states in high-dimensional environments is well-known idea explored in multiple papers (Houthooft et al. 2016; Bellemare et al. 2016; Ostrovski et al. 2016; Stadie et al. 2015; Achiam & Sastry 2017). As the agent uses fewer time steps in the environment to learn the same optimized policy, real world problems like robotics and self-driving cars, where the number of exploring steps could be limited by wearing parts or other factors will benefit from this.

1.2 Goals and Research Questions

Before the main research were started some guideline requirements, goals and research questions were proposed. This made it easier to keep track of what to read and what to change during the study.

Goal 1: Explore the most efficient techniques for creating a data efficient model-based RL algorithm for high-dimensional environments.

To start on this problem, the state-of-the-art model-based techniques must be studied. Therefore, the first research question is formulated as:

Research question 1: What are the best working techniques and architectures for approximating a deep dynamics model in environments with high-dimensional state representations?

Answers and knowledge from the first research question can be used to generate a novel architecture with the best ideas from the state-of-the-art model-based RL techniques. The environments will be a comparison for how well the different ideas work. This thesis focus on using intrinsic motivation to improve data efficiency. Therefore, the second research question is written as:

Research question 2: How can different types of intrinsic motivation affect the data efficiency and overall performance of reinforcement learning agents in environments with high-dimensional state representations?

To test the solution, it should be tested and analyzed in similar high-dimensional environments as used in state-of-the-art research. This will make it possible to compare with the other model-based techniques working in high-dimensional state spaces. With the research questions established, we can work on the requirements guiding this research and setting the limits for the final solution. We define the requirements for the solution as:

Requirement 1: The proposed system must work in environments with high-dimensional state representations.

Requirement 2: The proposed system must work in different environments.

Requirement 3: The proposed system must work for environments with a discrete action space.

1.3 Structure

The rest of the thesis is structured as follows.

- Chapter 2 provides the necessary background information on deep learning and RL to follow the rest of the thesis.
- Chapter 0 describes the state-of-the-art methods closely related to this study. This includes effective exploration methods and well-functioning model-based architectures.
- Chapter 4 presents the conceptual concepts of the proposed architecture.
- Chapter 5 explains details for the main experiments performed, together with hyperparameters and implementation details.
- Chapter 6 presents and discusses the results of the main experiments.
- Chapter 7 concludes the work with some proposals for future work.

2 Background

Machine learning is a field within computer science that makes software programs learn behaviors without being specifically programmed to act in a specific way. The programs learn by reading and making sense of data. Before going deeper into machine learning and its methods, we must understand what is meant by the term *model*. A model can be defined as the resulting function after a training algorithm has been run on a training set. As described in chapter 1, machine learning can be classified into the three main categories of learning methods; supervised learning (SL), reinforcement learning (RL) and unsupervised learning (UL).

This chapter presents background information, terms, equations and algorithms used in this study. The first part will go through the basics of deep learning and how DNNs are used as machine learning models. Next, more specific information about deep reinforcement learning with some modern state-of-the-art architectures will be explained. Finally, we dive into the exploration-exploitation trade-off problem.

2.1 Deep Learning

Deep learning is a field within machine learning that uses so-called deep neural networks as function approximators. These function approximators are inspired by the human brain and how neurons in the brain are connected. The neural network consists of interconnected nodes, where each connection has an associated valued weight that modulates the signal being transferred between the two interconnected nodes. A simple representation of a network can be found in Figure 2.1. The nodes in DNNs are grouped into sets of layers where the first layer is the input layer and the last layer is the output layer. Between the input and output layers there are one or more hidden layers. To get an output from the network, we feed the input layer with a tensor which propagates the values through the weights where they are multiplied and summed for the receiving node. We can write the linear output of a single node like:

$$f(t, w, b) = \sum_{i=1}^{n} (t_i w_i + b_i) \tag{2.1}$$

Where t, w and b are the input tensor, input weight tensor and bias respectively. The bias is just a single weighted input of one and acts as a constant that can be used to shift the output in either directions. Another use for the biases is that it makes neurons able to fire even though the input tensor is filled with zeros. The summation is done for all inputs to the node, hence, i represents the current input to the node where n is the total number of inputs. Usually, the values outputted from the nodes are fed through a nonlinear activation function. This makes the network capable of approximating nonlinear functions, which is an important property. Some common nonlinear activation functions are Tanh and Rectified Linear Unit (ReLU). The activation functions can be written respectively as:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$ReLU(x) = \begin{cases} 0 & for\ x < 0 \\ x & for\ x \geq 0 \end{cases}$$

Where x is the linear output of the node following the equation (2.1). Another type of function that are commonly used for neural networks is the softmax function. This operates on the whole layer instead of single nodes and works by taking the outputs from the previous layer as input and then normalize each value on the sum of all outputs. The equation for all K individual values in the output vector $x_i \in x$ is defined as:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$$

Essentially, this gives us a normalized vector where all values sum up to one. Hence, a probability distribution of the values in x.

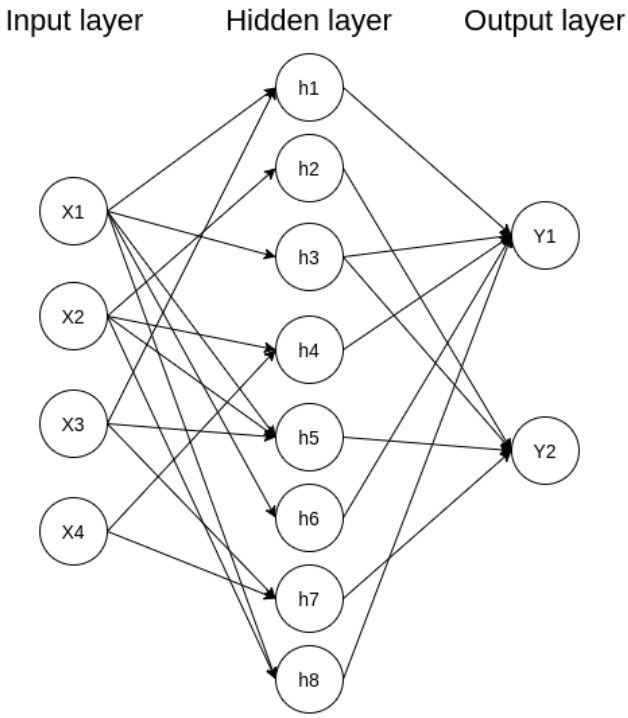


Figure 2.1: A partially connected feed-forward neural network with four input nodes (the circles) connected to a hidden layer with eight nodes, which further is connected to two output nodes. The arrows represent the data flow through the weights between the nodes.

2.1.1 Backpropagation

To approximate a function with a neural network, we gradually modify the weights of the network towards the wanted function. This technique is called backpropagation. To do this, we need to feed our input tensor, called x, through the network and get an output tensor, called y'. Then we can calculate the error based on a loss function, L. The loss is a representation of how different output y' is to the expected output y. There are many possible loss functions, but mean squared error (MSE) is the most common choice for neural networks. We can write the loss function for MSE by:

$$L_{MSE}(y, y') = \frac{1}{n}\sum_{i=1}^{n}(y'_i - y_i)^2 \tag{2.2}$$

Where n is the size of the output tensor. Another common loss function is to use the cross entropy, given by:

$$L_{CE}(y, y') = \sum_{i=1}^{n} y_i' \, log(y_i)$$

Cross entropy creates an indirect measure of how bad our prediction distribution y' is at representing the true distribution y. The goal is to minimize this loss, and thereby minimize the error between the output of the network and expected output. To minimize the loss, we calculate the change in weights, also called the gradients. The gradients are computed by the negative derivative to the loss. For example, using L_{MSE} from equation (2.1). To control the amount of learning and to gradually descend towards the minimum loss, the gradients are multiplied with a learning rate α. The learning rate is a constant in the range $0 < \alpha < 1$. To make the neural network generalize and not specialize on the given data set it assumes that the data set is independent and identically distributed. If this assumption is broken, bad side effects may occur. For instance, if we were to have a DNN do anomaly detection for a factory we would typically not have an equal amount of anomaly examples as normal examples. If the DNN were trained with this skewed dataset, it could learn that anomalies almost never occur and just learn to predict normal examples all the time.

2.1.2 Dropout

One typical problem of neural nets is overfitting on the training set. This problem arises when a too small dataset is used to train a network with too many nodes. This makes the network specialize on the training set, rather than generalize towards the wanted function. This problem can be addressed with *dropout*, proposed in (Hinton et al. 2012). Dropout is a computationally effective technique that skips to train a node with a probability p. This technique can be applied to one or more layers. Dropout makes the network regularize towards the wanted function and reduces the chance of overfitting. This is because the network must learn how to perform well even though some parts of the network are missing.

2.1.3 Convolutional Neural Networks

The nodes in a neural network can be interconnected in many ways to get different functionality. Until now, we have talked about regular feed-forward networks, which is often structured with fully connected layers where all nodes from one layer are connected to all the nodes of the next layer. Another layer architecture that has been used a lot lately to gain good

results in machine learning involving images is *convolutional neural networks* (CNN). These networks are a kind of feed-forward network, but the activation of a node depend on neighboring nodes in the previous layer instead of all the nodes as in fully connected. This makes CNNs excellent at finding lines, edges, shapes and local patterns in the data. For images, this makes each layer learn more and more abstract features the deeper into the network. Typically, the first layers would recognize lines and edges, while the last layers would recognize more abstract shapes and patterns. Like in the Figure 2.2, a CNN is often used together with other types of layers such as pooling layers and fully connected layers. Pooling layers are used to make a non-linear down sampling representation of the convolution. The most common pooling layer are max pooling which selects the maximum value of an area in the convolution.

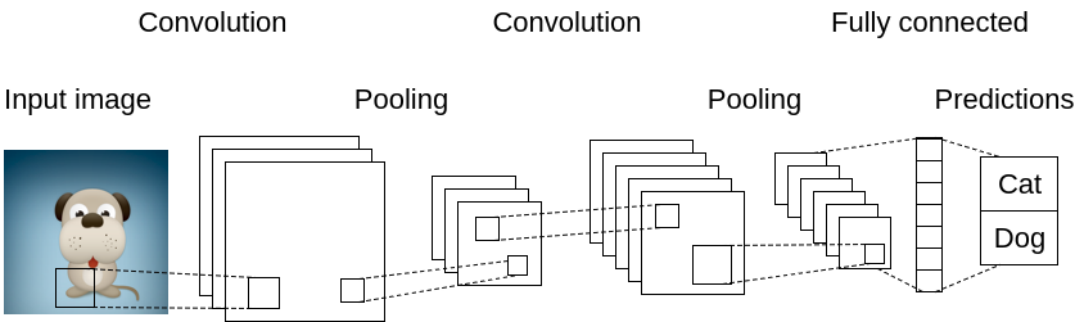


Figure 2.2: A common CNN architecture for image classification. The image is fed into a series of convolutional and pooling layers. It is common to add a fully connected non-linear layer at the end before printing the softmax probabilities over the different classes.

2.1.4 Recurrent Neural Networks

For more time dependent data streams, like sound, text or video, we have a structure called *recurrent neural networks* (RNN). RNNs have one or more layers which is connected to the next layer and to itself with a time step delay. This means that the layer get input from previous time steps, which makes it possible to learn patterns through sequences. Take a natural language problem for instance, if you say “I don’t”, the model would have a higher probability of predicting the next word to be “know” than for example “car”. This would be hard for a regular network to learn because it could never remember the previously observed words. Long short-term memory (LSTM) layers, as proposed in (Hochreiter & Schmidhuber 1997), are one of the most common type of recurrent layers.

2.1.5 Autoencoders

Another common structure is called *autoencoders* (AE). These networks are used to compress the input tensor into a more compressed representation. This is done by feeding the input through a network with gradually smaller layer sizes, the encoder, and then through gradually greater layer sizes, the decoder. The network can be trained in a semi-unsupervised fashion by setting the original input as target for the output. When optimized on the error between the reconstructed input x' and the original input x, the network learns a compressed latent representation of x called z. A simple illustration of an AE can be found in Figure 2.3. These structures can be very useful to get a more compact latent variable with useful features from a high-dimensional input tensor. AEs can be put together with regular fully connected layers, however, for high-dimensional inputs like images, it is common to use convolutional layers to capture better features in the latent space. Such AEs is often referred to as convolutional autoencoders. This type of AE is further explored in (Masci et al. 2011).

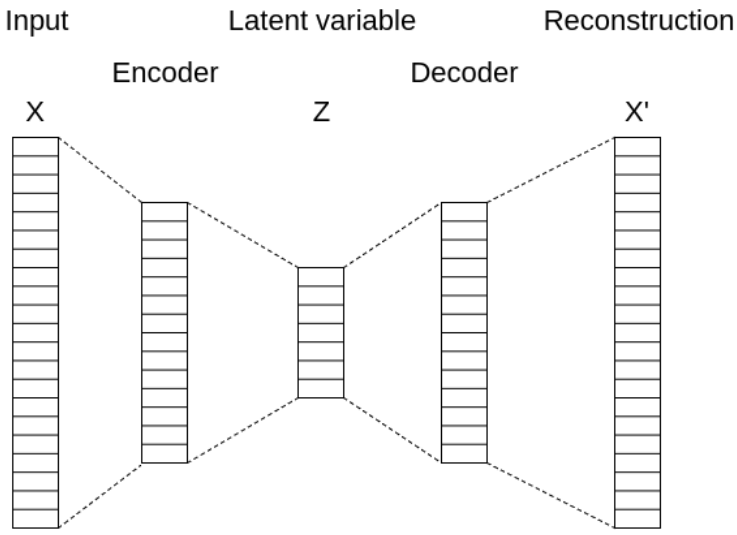


Figure 2.3: A representation of a deep AE that gets an input x, tries to encode (compress) this into a latent representation z, and then tries decode (decompress) it to represent the original input. This makes the network learn important features in the dataset to make good compressions.

Regular AEs are good at generating a latent space with important features extracted from the training samples. A downside to these AEs are that they are free to decide where to put each feature the latent space. Therefore, to generate a smoother latent space we introduce a constraint that makes the encoder generate latent variables that follows a Gaussian distribution. This makes it possible to sample latent vectors without a real input sample. AEs

containing this constraint are called variational autoencoders (VAE) and they are closely related to generative models. The difference from a regular AE is that the encoder in a VAE generates a point in a Gaussian distribution, a mean vector and a standard deviation vector. We can then sample a latent vector from this distribution. The sampled vector can further be sent to the decoder as in a regular AE. A simple representation of a VAE is shown in Figure 2.4. To train a VAE we must add the constraint to the loss function. The most common constraint is based on the Kullback-Leibler (KL) divergence of how close the latent variables fits in a unit Gaussian distribution. The KL divergence acts as a regularization term for the neural network and can measure how similar two distributions are. If we represent the encoder of the VAE as $q(z|x)$ and the decoder as $q(x|z)$, we can write the KL divergence as:

$$D_{KL}(q(z|x)||\mathcal{N}(0,1))$$

This shows the similarity between the normal distribution $\mathcal{N}(mean, variance)$ and the output distribution from the encoder $q(z|x)$. The KL divergence regularization term can be written as:

$$D_{KL}(z_m, z_{std}) = 0.5 \cdot \sum_{i=1}^{N}(z_{m,i}^2 + z_{std,i}^2 - log(z_{std,i}^2) - 1)$$

Where the z_m represents the latent mean vector and z_{std} represents the latent standard deviation vector from the neural network. The summation is done for all N variables in the vectors. The final loss for the VAE can be given as:

$$L_{VAE}(x, z_m, z_{std}) = L_{MSE}(x) + D_{KL}(z_m, z_{std})$$

The final loss, L_{VAE}, combines D_{KL} with the MSE reconstruction loss from equation (2.2). Other types of reconstruction loss, such as cross entropy, can be used as well.

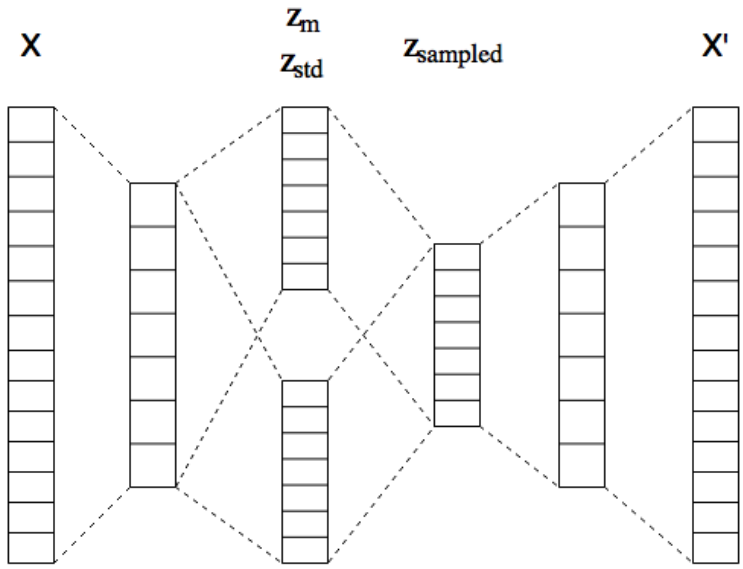


Figure 2.4: A representation of a deep variational AE where the data flow for inference is from left to right. The input x is encoded into an approximated Gaussian distribution with mean and standard deviation like z_m and z_{std} respectively. We can then sample a latent variable from the mean and standard deviation distributions represented as $z_{sampled}$ in the figure. The latent variable can be decoded to a reconstructed output x' in a similar way as with a regular AE.

2.2 Uncertainty in Deep Neural Networks

Managing to extract precise uncertainty from DNNs can be crucial for all types of systems using them. Understanding uncertainty in DNNs makes it possible to understand the weaknesses of the model. For classification tasks, the most common way of doing this is by using a softmax layer as the last layer of the DNN. This gives us a probability distribution over all possible classes. Even though this gives an uncertainty measure for the current input, the measure is not very precise. To understand why we must look how *Bayesian learning* works.

Bayesian learning is based on taking not one single possible solution for the observed dataset, but rather take many possible solutions. Optimally, we would have infinitely many solutions for the current dataset. By having many solutions, we can extract the variance in all those solutions and get a precise uncertainty value for each new data point. However, when using the softmax technique described above we are only using one set of weights. Hence, only one solution for the currently observed data is given. There exist many methods to overcome this

problem for DNNs, but only the two used in the thesis will be described. These methods share the property of having a low computational cost which is very important in RL. An experimental comparison of the two methods can be found in Appendix C.

The first one is *bootstrapped* techniques. These techniques are based on the statistical resampling methods proposed by (Efron 1982). However, these methods only describe bootstrap in a general way for extracting different properties in approximators. It was only recently proposed in (Osband et al. 2016) how bootstrapped methods could be used to extract uncertainty in DNNs. The authors modify the last layer of the network such that instead of having one output layer it has multiple equally sized output layers, called heads. This makes it possible to train each head on different parts of the data. Thereby, the variance from all heads for a new observation can be used as a more precise uncertainty value. This is because the different heads acts as different possible solutions for the current dataset, comparable to Bayesian learning. Optimally, all heads will with enough data converge towards the same solution.

The second method is the recently proposed *Monte Carlo (MC) dropout* from (Gal & Ghahramani 2015). This tackles the "multiple solution" problem by feeding the same input multiple times through the same network while using stochastic dropout masks. This is also referred to as stochastic inference and has similar properties to variational inference from Bayesian learning. When feeding the same input through the network while some of the nodes are dropped gives us a slightly different output for each feed forward. The number of how many times the same input is fed into the network is defined as T. MC dropout will give better uncertainty estimates as T increases due to having more solutions to the current data.

There are of course some negative aspects of using these methods. Bootstrap usually makes the convergence slower, hence, worse data efficiency. Also, the network size grows because we are forced to store the extra weights for each head compared to a regular DNN. MC dropout requires more computation during the inference which can be bad in many situations where the inference needs to be very computationally efficient.

2.3 Deep Reinforcement Learning

Reinforcement learning (RL) is based on the problem where you want to optimize behavior based on received feedbacks. Visualize how a baby tries something new and observes how well it did by getting a feedback from the parents. If the baby sees a smile, which indicates a positive feedback, it will try to do the same action again. In RL, the baby is called an agent, and the world is called an environment. The agent receives the feedback, which we call reward, from the environment after performing an action. The full RL loop is illustrated in Figure 2.5. Deep reinforcement learning is a small subterm within RL where the agent decides its actions based on one or more deep learning models. Hence, the agent learns the optimal action policy based on exploring the environment, receiving rewards and training the deep models to optimize the long-term reward.

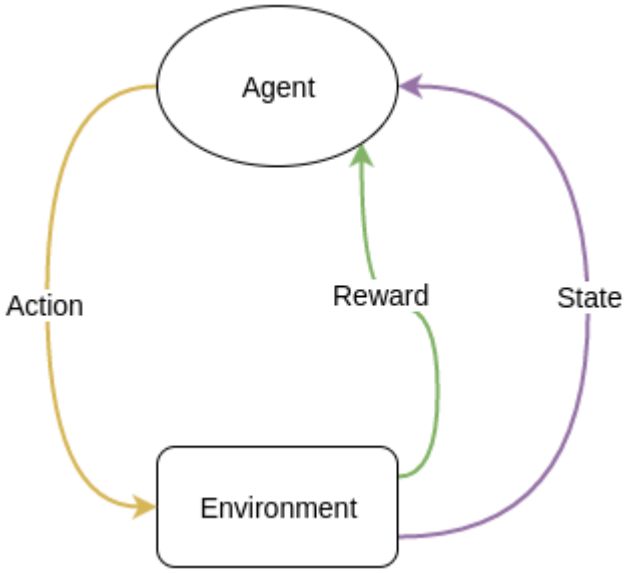


Figure 2.5: A flowchart of how the RL process. The agent observes a state, performs an action and receives a reward. The environment transition into the next state is based on what the agent performed.

2.3.1 Environments

The environments in RL can be represented as a wide range of things, from simple games like Tic-tac-toe to the real world. The observed state from the environment can be represented in many ways. Common representations are inputs from robotic sensors or pixels from an image. The environment can be fully or partially observable. Hence, the observed state may or may not be a full representation of the environment. For example, in Tic-tac-toe,

the agent would typically receive the whole board as a state, therefore, it is a fully observable environment. On the other hand, a robot moving through the real world would never be capable of mapping every variable into a vector for the real world, thus, it is partially observable. Partially observable environments are usually much harder to solve than fully observable environments.

An environment can be discrete or continuous. If an environment is discrete, there is a finite number of states in the state space of the environment. Tic-tac-toe, again, would be a good example for discrete environments. The total number of states in Tic-tac-toe's state space is $9! = 362880$ (9 possible moves when placing the first mark, 8 for the second and so on). The robot in the real world has infinite many states.

Environments can also be categorized by the property of being deterministic or stochastic. In a deterministic environment, the same state transition would happen for any equal state and action input. Let us take the first move of Tic-tac-toe. If we place the mark in the upper left corner, the mark will always end up in the upper left corner. There is no chance of the mark ending up in some other square on the board. However, the robot might see the exact same state, but even if the same action is performed it might not see the same outcome. Thus, there is some randomness in the environment our agent does not control.

In RL, environments can be described as a *Markov Decision Processes* (MDP). This representation is a map of all possible outcomes for all set of states and actions put together. One episode of a MDP is a finite sequence of states, actions, state transitions and rewards.

2.3.2 Agents

An agent consists of a model that tries to learn the optimal action policy for maximizing future rewards received from the environment. Actions can be discrete or continuous, where discrete actions represent a fixed number of possible actions, and continuous actions consists of an unlimited range of values for each possible action. The nine squares on the Tic-tac-toe board is an example of a discrete action space, and the robot's motor forces assessed with real values are an example for a continuous action space. Agents are classified into two main groups of models, *model-free* and *model-based*. Model-free agents try to learn the optimal behaviour directly from experience. Model-based agents try to optimize the behaviour as

well, but they do it by creating a model of the environment that can be used to predict the next state and reward based on previous experience. This prediction model is called a dynamics model and represents how the environment reacts to all different state-action pairs. For this thesis, we will group any technique that benefits from having a dynamics model as model-based RL.

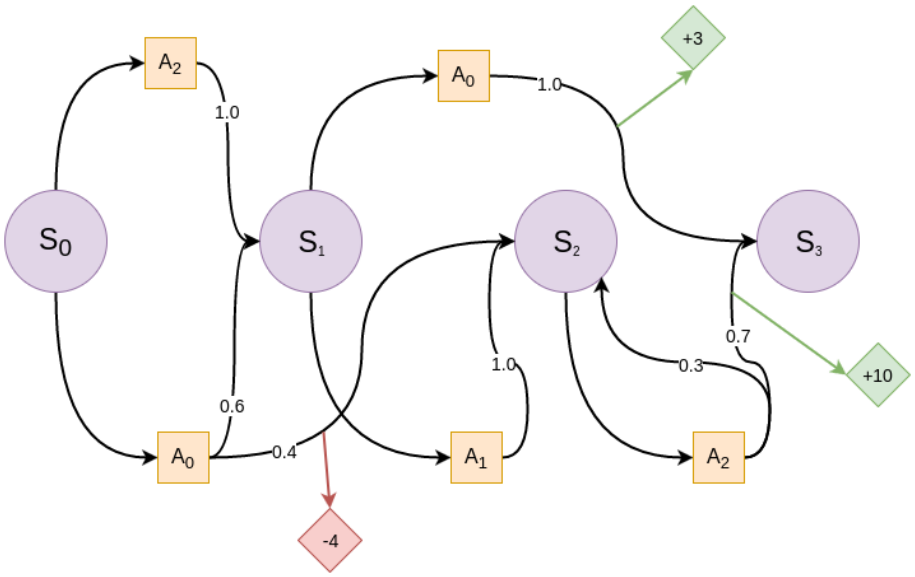


Figure 2.6: A representation of a Markov decision process, where S is the state, A is the selected action, and the diamond shaped boxes represent the received rewards. This environment is a nondeterministic environment because of the probabilities listed in each arrow after the action. These probabilities represent the transition function of the environment. For example, the environment has a chance of 0.6 to end in S_1 and a chance of 0.4 to end in S_2after the action A_0 is performed in S_0.

2.3.3 Rewards

The reward corresponds to feedback the agent receives from the environment after performing an action. It can be positive or negative. If there is no immediate feedback for the executed action the reward is zero. Problems where positive or negative rewards are received for each action are usually easier for the agent to learn. These types of environments are said to have dense rewards, and the opposite types of environments have sparse rewards. The reward function can be seen as the optimization goal for the agent's behavior.

2.3.4 Use Cases of Model-Based Reinforcement Learning

The main goal of using model-based RL instead of model-free RL is to utilize the approximated model of the environment to gain faster and more stable learning, and hopefully, a better final policy. In addition to making learning faster, model-based RL can be used to do planning. This is done by feeding the approximated dynamics model the current state and the following trajectory of actions. The future states can be analyzed and used to do smarter exploration or risk assessment. For instance, if one were supposed to learn a helicopter to fly, one could make an approximated dynamics model and use it to avoid big and expensive crashes. A similar experiment was done in (Kahn et al. 2017), where the uncertainty of the dynamics model was used as a measure for how high speed the robot could travel with. High certainty equals low percentage of crash which makes the robot able to go at a higher speed. Another possible feature of having an approximated dynamics model is that it can generate extra experience for the agent. In theory, with a close to perfect model in a deterministic environment, the agent would only have to observe every state once because the outcome of every action could be simulated for each observation. The focus for this thesis is to use the dynamics model for smart exploration. Smart exploration in model-based RL means to utilize some feature of the dynamics model to guide the agent towards novel states and better data efficiency.

2.4 Deep Reinforcement Learning Architectures

In this section, we describe the most commonly used deep RL architectures. We start off by explaining Q-learning and how it is converted to Deep Q-learning. Thereafter, we move to actor-critic methods, ending with the current state-of-the-art model-free Parallel Advantage Actor-Critic architecture.

2.4.1 Deep Q-learning

Q-learning is a model-free *value iteration* RL algorithm that uses a function, $Q(s_t, a_t)$, to represent the expected maximum discounted reward for the future when performing action a_t in the current state s_t. The value function can be written as:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma(r_{t+2} + \gamma^2(r_{t+3} + ...))|s_t, a_t]$$

This can be further compressed into:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) | s_t, a_t] \tag{2.3}$$

Here, r_{t+1} is the received reward for taking action a_t in state s_t. The γ is the discount factor for future rewards, and is set to a value between 0 and 1. The last representation, equation (2.3), has compressed the recursion occurring in the first equation. This is called the *Bellman optimality equation*. The Bellman optimality equation is what makes it possible to iteratively improve the action value estimates. The Q-values can be stored in different ways. For small state and action spaces it would be easy to store the values in a tabular way where each individual state and action pair has a Q-value represented in each the cell. This becomes problematic as soon as either the state space or the action space grows. Therefore, this can be solved by approximating the Q-function with a function approximator, like a DNN. Q-learning algorithms that has its Q-function represented by DNNs are called Deep Q-learning (DQN). This is reckoned to be one of the major algorithms for starting deep RL and was published by Deepmind in (Mnih et al. 2013). DQN managed to have one algorithm learn to play 57 different Atari 2600 games by feeding the agent the game screen and game reward. This method suddenly made it possible to perform RL on high-dimensional data like images. A representation of how the DQN is constructed can be seen in Figure 2.7.

The final action policy for Q-learning is determined by:

$$\pi(s_t) = argmax_a Q(s_t, a_t)$$

The Q-function can then be iteratively improved by updating $Q(s_t, a_t)$:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \; argmax_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

In this equation, α represents the learning rate, which is the rate of how much the processed step affects the Q-function. The learning rate is a number between 0 and 1. From the Bellman optimality equation previously described we take the maximum Q-value for the next state and action pair, hence, $max_a Q(s_{t+1}, a_{t+1})$. This means that if $\alpha = 1$, both terms cancel each other out and we are left with the Bellman optimality equation (2.3).

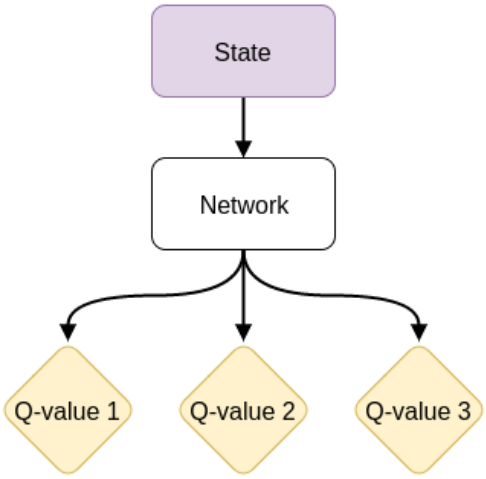


Figure 2.7: A representation of the DQN in a setting with three possible actions. Each of the Q-values represents the expected maximum discounted future reward for that specific action taken in state s_t.

2.4.2 Actor-Critic Methods

Even though DQN made a big breakthrough in RL, it had some negative aspects with it as well. The most important one, being a very computationally heavy and data inefficient algorithm. Trained on a high-end desktop computer, it could use two weeks to learn a good policy for an Atari environment. This made it hard to experiment with for smaller companies and private persons that did not have the same amount of computational power. An alternative to Q-learning is *actor-critic* algorithms. Actor-critic algorithms are divided into two parts; an actor which defines the action policy, and a critic which criticizes the actor's behaviour. Actor-critic is reckoned to be the best of both world, incorporating value-iteration (critic) and *policy gradient* (actor).

Policy gradient is another type of learning technique in RL that tries to learn the action policy directly, without having to learn each action value for every state. This makes it possible to learn stochastic policies. Policy gradient methods are considered to be more efficient than value-based methods for high-dimensional action-spaces. This is because it is easier to generalize towards knowing that one action is better than another rather than approximating the expected cumulative score from this current time step and on. Some of the main cons of policy methods are that they can easily get stuck a in local optima, and that they are more unstable during learning compared to value-based methods. For deep RL the policy methods

can be optimized by gradient ascent. One of the greatest problems is that it is intractable to compute the gradient. A solution is to estimate the gradient by log likelihood ratio. This updates the DNN such that high rewarding actions are more likely to be selected next time.

Back to actor-critic algorithms, which combine both a value-based method and a policy gradient method, the actor produces a stochastic policy $\pi(s_t; \theta)$ while the critic defines an estimate of the value for being in the current state $V(s_t; \theta_v)$. The two parameters θ and θ_v defines the parameters for the actor network and the critic network respectively. One of the benefits of using both value iteration and policy gradient is that the actor can be updated more efficiently by using the value estimates from the critic. One of the best performing methods in the Atari 2600 domain is an algorithm based on actor-critic. It is called Asynchronous Advantage Actor-Critic (A3C), proposed in (Mnih et al. 2016), and takes multiple asynchronously working actor-critic agents which shares the actor-critic networks to learn the same environment much faster and more stable.

2.4.3 Parallel Advantage Actor-Critic Architecture

RL has in the last four years improved on the DQN and regular actor-critic methods, and one of the most recently proposed algorithms is the Parallel Advantage Actor-Critic (PAAC) algorithm from (Clemente et al. 2017). Compared to DQN, PAAC can train an agent to super-human performance in the Atari domain in just 12 hours on a desktop computer with a GPU. PAAC combines multiple parallel n-step actor-critic learners, which was proposed by (Mnih et al. 2016). A representation of the architecture for PAAC can be seen in Figure 2.8. Each agent is called a worker. The number of workers are stated as n_w. Each worker can control n_e environments. As displayed in Figure 2.9, one single neural network generates the features for both actor and critic. The workers run synchronously in an effective manner for t_{max} time steps, where a master computes the actions for each environment every time step. This creates n_e observations every time step, where one observation can be described as the tuple $o_t = (s_t, a_t, r_t, b_t)$, containing the current state, action, reward and a Boolean flag to know when the episode is over. Every observation is stored in a small replay buffer with size $B_o = t_{max} \cdot n_e$. After t_{max} time steps are done, gradients are calculated and updated to the actor and critic.

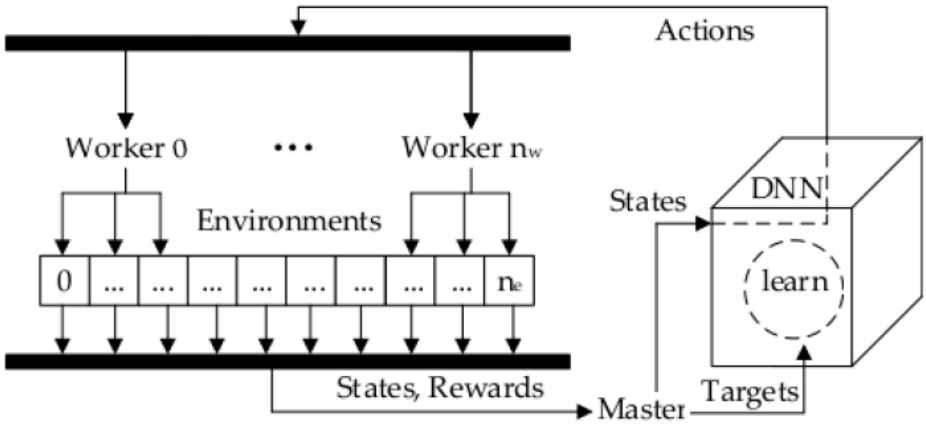


Figure 2.8: The generic architecture for PAAC. The figure is fetched from (Clemente et al. 2017).

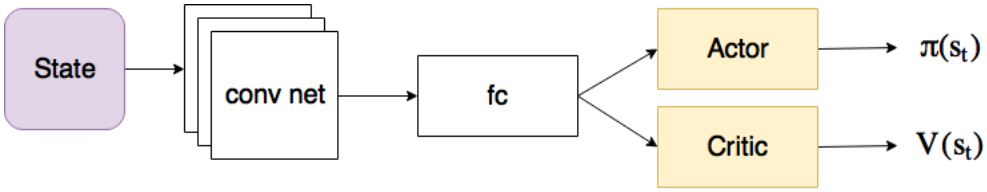


Figure 2.9: Deep neural network structure for PAAC.

To describe the updates in PAAC in more detail we must define some parameters. We first define the quality of a state-action pair as,

$$Q^{(n)}(s_t, a_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}; \theta_v)$$

where γ is a discount factor and $0 < n \leq t_{max}$. Secondly, we define the entropy of the policy as $H(\pi(s_{e,t}; \theta)$. This is only needed to keep the agent from premature convergence. Finally, the gradients used to update the networks for the actor and the critic are calculated as described in equation 10 and 11 in (Clemente et al. 2017):

$$\Delta_\pi \approx \frac{1}{t_{max} \cdot n_e} \sum_{e=1}^{n_e} \sum_{t=1}^{t_{max}} (Q^{(t_{max}-t+1)}(s_{e,t}, a_{e,t}) - V(s_{e,t})) \Delta_\theta log\pi(a_{e,t}|s_{e,t}) + \beta \Delta_\theta H(\pi(s_{e,t}))$$

$$\Delta_V \approx \Delta_{\theta_v} \frac{1}{t_{max} \cdot n_e} \sum_{e=1}^{n_e} \sum_{t=1}^{t_{max}} (Q^{(t_{max}-t+1)}(s_{e,t}, a_{e,t}) - V(s_{e,t}))^2$$

2.5 Exploration-Exploitation Trade-off Problem

One of the most fundamental problems in RL is the exploration-exploitation trade-off problem. The trade-off is based on the problem of selecting an action that might gather more information about the environment, exploration, or selecting the best action given the current information, exploitation. Even though following an exploitation heavy policy might yield higher rewards in near future, performing more exploration might give more even more rewards in long-term.

2.5.1 Naive Exploration Techniques

One of the easiest exploration technique to imagine would be to always select the best action given the current information. This is called the *greedy policy* and will always try to maximize the possibility of getting a reward for the next time step. Greedy policy can be written as,

$$\pi_{greedy}(s, a) = \underset{a' \in A}{argmax}\ Q_t(s, a')$$

where Q_t represents the value for taking action a in time step t. The greedy policy would work in an easy environment where the optimal long-term policy is the same as the optimal short-term policy. The greedy policy would never be able to solve harder environments where the agent must do an action that leads to no (or negative) reward to receive a bigger reward later. Thus, we must look for a more exploration-based policy to solve harder problems.

A more exploration-based policy is the ϵ-*greedy* policy which selects a random action with a probability ϵ and follows the greedy policy otherwise. Following this policy, the agent would never stop exploring. A slightly modified version of the ϵ-greedy is the *decaying* ϵ-*greedy* which does the same as regular ϵ-greedy, but with a decaying ϵ. This ensures that the agent utilizes more and more of the information gathered from the environment. The decaying ϵ-

greedy is one of the most used exploration techniques because of its simplicity and effectiveness.

Another heavily used exploration technique is the *softmax policy* which selects an action uniformly based on the current beliefs about each action. This means that actions which is expected to give higher rewards will have a higher probability of being selected. A common way of doing this is to make a Boltzmann distribution over the expected values for each action. Formalized as,

$$\pi_{softmax}(s,a) = \frac{e^{\frac{Q(s,a)}{\tau}}}{e^{\sum_{a' \in A} \frac{Q(s,a')}{\tau}}}$$

where τ is a temperature constant. When $\tau \to 0$, the softmax policy is the same as the greedy policy.

2.5.2 Optimism in the Face of Uncertainty

One key principle in the exploration-exploitation trade-off problem is: "*optimism in the face of uncertainty"*. By this we mean that the agent should explore based on how uncertain it is about a state-action pair. From Figure 2.10 we can see that even though action a2 has a higher mean than action a1, action a1 is the one to be selected because of having a greater potential. We see this as a win-win situation for the agent, where it either can be more certain that action a1 was worse than action a2, or that action a1 gave a higher reward than action a2 and thereby we could exploit that action. An exploration technique that follows this principle is the *Upper Confidence Bound* (UCB) policy which selects the action with highest possible value with some confidence bound added to each action. To calculate the upper confidence bound we use the algorithm for UCB1. The UCB1 policy can be formalized as:

$$\pi_{UCB1}(s,a) = \underset{a' \in A}{argmax}\ Q(s,a') + \sqrt{\frac{2\ log\ t}{N_t(s,a')}}$$

Where t is the current time step and N_t is the number of times this state-action pair has been seen before. This of course limits UCB1 to enumerable environments.

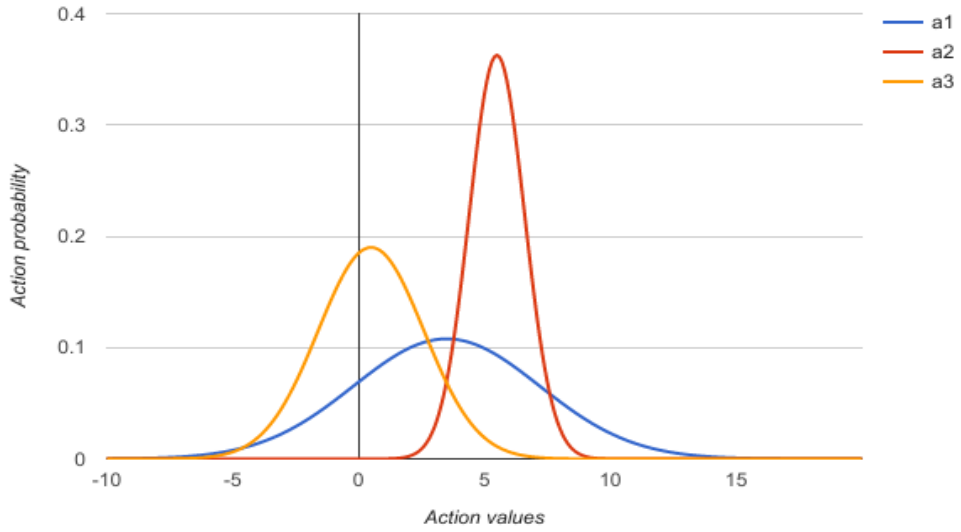


Figure 2.10: Plot describing the principle: "optimism in the face of uncertainty". Action a2 has the highest certainty and highest mean values, but action a1 is the one to be selected because of the potential to be better.

2.6 Measuring Exploration Efficiency

How efficient an agent explores an unknown environment can be measured in multiple ways. The most basic method looks at how much *regret* is gathered throughout the learning period. Regret is defined as the expected loss between the selected action and the optimal action for every time step. The sum of loss up to a certain time step can be defined as,

$$L_t = \mathbb{E}[\sum_{t'=1}^{t} V_{t'}^* - r_{t'}]$$

where V^* is the optimal policy and r is the received reward. We see that the goal for any agent is to learn the optimal action policy with the lowest L_t possible. From the plot in Figure 2.11, we can see the illustrated effect of having a decaying ϵ based on total regret. The problem of using regret on RL problems is that we usually never know the optimal value for every time step. Therefore, we need another way of measuring how efficient our algorithm explores the environment.

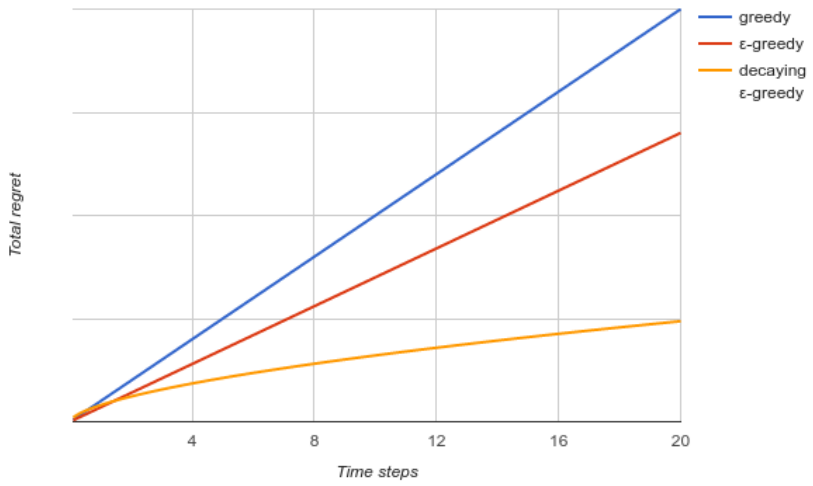


Figure 2.11: Plot of total regret for different naive exploration techniques.

A more commonly used way of testing efficient exploration, used in (Stadie et al. 2015; Osband et al. 2016), is to use Area Under Curve (AUC) where we approximate the AUC for the reward graph by for example using the trapezoidal rule. Again, because we do not know the optimal learning strategy for the environment, this can only be used for comparing efficiency for one agent against another.

3 Related Work

In this chapter, we will go into some of the state-of-the-art deep RL methods that focus on data efficiency. Additionally, we will look at the most successful ways of creating deep dynamics models (DDM) for high-dimensional environments. The described techniques are the inspiration to this study. Therefore, they are closely related in terms of architecture and methodology. First, in section 3.1 we describe different successful techniques for approximating a dynamics model in high-dimensional state spaces. After that, we move over to advanced exploration methods. We divide these techniques into model-free, section 3.2, and model-based, section 3.3, exploration techniques.

3.1 High-Dimensional Dynamics Models in RL

One important aspect of improving sample efficiency in this thesis is the use of an approximated dynamics model. Model-based methods tend to be more data efficient than model-free methods because of the extra knowledge of the environment stored in the dynamics model. This knowledge can be used in many ways to get better data efficiency. Although, with an inaccurate learnt model, the agent may not learn anything at all. This comes from the principle of propagating errors. A small prediction error will quickly escalate to a big error a few time steps in the future. Therefore, the hard part of model-based RL is to create precise and accurate models. This gets even harder as the state and action space grows. In this section, we elaborate on the current state-of-the-art techniques for creating a dynamics model.

3.1.1 Encoder-Decoder Architecture

In (Oh et al. 2015) a deep convolutional encoder-decoder structure is used to compress the high-dimensional state representation into a lower embedding. The embedding is then concatenated with an action and further fed into the decoder layers of the network. This encoder-decoder network is trained to predict the future image based on the current observation and action. This network represents the approximated dynamics model of the environment. The method is tested in the Atari environment where each observation is represented by four downscaled 84x84 grayscale images. The authors tested two different structures for handling the four images as input. The first version sends all four images,

concatenated depth wise, as input to the network. The second version sends one image at the time and captures the sequence of four images in a recurrent layer after the encoding layers. This is shown in the Figure 3.1. By using a curriculum learning where the network learns to predict future frames with gradually increasing time span, it becomes capable of predicting accurate frames up to 250 time steps in the future. Although, this was achieved using the full sized 210x160 RGB images.

The authors do also propose a method to increase exploration efficiency by using informed exploration. The informed exploration stores the last T trajectories in a memory and calculates a similarity between the predicted future frame and all stored memory. The agent selects the action that returns the least similar frame. This makes the agent explore in a more efficient manner than just selecting actions randomly. Informed exploration improves the final performance compared to the regular ϵ-greedy exploration in DQN for three of the five Atari games that were tested. The method achieves precise long-term predictions and further use the predictions in an RL setting to make actions that leads to less frequently visited states.

One of the biggest problems of this method is the stochasticity in the environment transitions. An example of stochasticity in transitions are the spawning fish in Seaquest, which quickly disappear from the future predictions. This is one of the hardest problems when it comes to planning many steps ahead. Another problem for this method is the vanishing of fast moving small objects like the gunshot in Seaquest. Placing such details in the latent space of an encoder-decoder network is important because it is often these objects that matter the most in the game. Also, because of an AE with many parameters, the dynamics model needs a lot of training time, making the process of training new agents very slow.

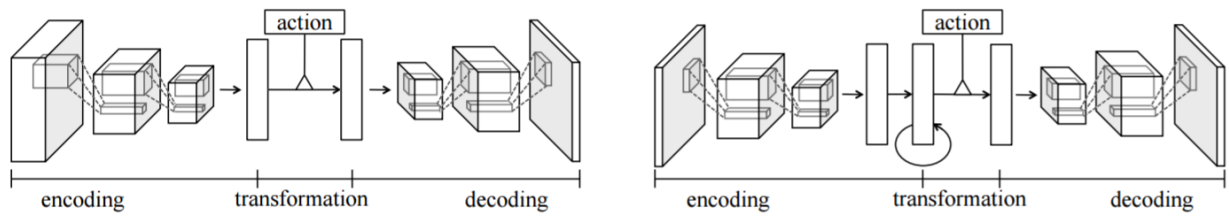


Figure 3.1: Figure from (Oh et al. 2015) showing two DDM architectures. The figure to the left gets four frames as input and have a non-recurrent transformation part. The rightmost figure gets only one frame as input, but models the time dependency in a recurrent layer in the transformation part of the network.

A similar network structure to (Oh et al. 2015) is the architecture proposed in (Wahlström et al. 2015), where the observation is compressed with a comparable encoder-decoder structure. The network consists of an encoder part that compresses the high-dimensional image observation into a lower latent representation, and a decoder part that decompresses the latent representation into a reconstructed image. Between the encoder and decoder parts of the network is the prediction layers that learns the transition rules of the environment. This prediction part takes the encoded latent variable and concatenates it with the action in the same way as in (Oh et al. 2015). However, one significant difference between (Oh et al. 2015) and (Wahlström et al. 2015) is the use of an n-step history of latent variables that is fed into the prediction part of the network. This differs from (Oh et al. 2015) where they tested two methods, one where four images as input to the encoder-decoder network and a second to replace one of the encoded fully connected layers with a recurrent layer.

The authors use the predictions from the learned dynamics model to make a very data efficient adaptive model-predictive-control algorithm for simple continuous control problems. The environments tested were CartPole problems where the agent controls a cart with a balancing pole in a 1-dimensional action space. Even though the proposed method achieves great sample efficiency and performance, it only works for continuous environments. Also, the environments tested were very simple environments where the observations were high contrast images containing no extra details other than the controllable parts of the environment (the cart and the pole). Simple observations make it much easier for the encoder-decoder network to compress and decompress the state into a useful latent representation. The use of fully connected AEs struggles with small details and would probably cause a problem for more detailed environments.

3.1.2 Variational Encoder-Decoder Architecture

The work from (Watter et al. 2015) tackles the creation of approximating a dynamics model in high-dimensional state spaces in a slightly different way than the previously explained methods. Instead of using a regular AE structure, a VAE is used. The model works by first encoding the image observation s_t into a latent space z. The latent representation z_t is then fed into an approximated transition function.

The authors try multiple different transition functions, but find that a simple linear transition function performed best. The predicted latent distribution $\hat{z}_{t+1}$ is inferred by the action a_t and the prediction from the transition prediction. When the true s_{t+1} is observed, the transition approximator is updated by the KL divergence between the true z_{t+1} and the predicted z'_{t+1}. The architecture of the E2C model can be seen in Figure 3.2. The transition model was tested on an image-based CartPole environment and got precise predictions up to 10 time steps ahead. The network architecture in this method solve one of the problems from (Wahlström et al. 2015) where the encoder-decoder structure was put together by fully connected layers. For this method, convolutional layers and pooling layers are used to create filters that generalize objects in the observation. Even though, the method is still only tested on the same types of simple environments as used in (Wahlström et al. 2015).

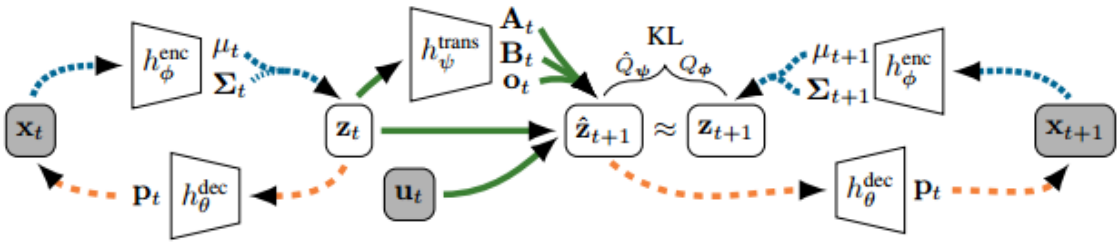


Figure 3.2: Figure from (Watter et al. 2015) of the architecture for the E2C model. The flow goes from left to right and starts with observation s_t (x_t in the figure) that gets encoded into the latent variable z_t. This latent variable is then used together with action a_t (u_t in the figure) to predict the next latent variable $\hat{z}_{t+1}$.

3.2 Model-Free Exploration Techniques

In this section, we describe advanced exploration techniques that do not require any approximated dynamics model. These methods are typically based on counting unique states or by extracting uncertainty properties in the model-free algorithms to perform effective exploration.

3.2.1 Optimal Exploration in Finite MDPs

Some of the earliest methods that tackled the exploration-exploitation tradeoff was the count-based algorithms. Count-based methods keep a counter for each individual state (and possibly state-action pair). This knowledge can be used by the agent to know how "known" the

current state is. Therefore, this counter would be used as a measure for how much the agent should follow an exploring action policy or an exploiting action policy. A well-working algorithm for handling this is the Explicit Explore or Exploit (E^3) algorithm from (Kearns 2002), which compares the counter for the current state to a threshold value to determine if this state is known. If the state is known, the action with the highest value is selected, and if the state is set to unknown, an exploring action is selected. Another well-known count-based exploration technique is the R-MAX algorithm from (Brafman & Tennenholtz 2001), which is based on the idea of optimistic initialization. Optimistic initialization means that all unknown state-action pairs are assumed to have the maximum possible reward. The reward for the unknown state-action pair is gradually lowered towards its real reward as it is visited more often. RMAX follows the famous exploration-exploitation principle "optimism in the face of uncertainty". By having all unknown state-action pairs at maximum reward, it means that the agent can act greedily all the time and still be able to perform exploration. These algorithms are reckoned to be close to optimal exploration policies in finite MDPs. However, when moving over to infinite MDPs, optimal exploration becomes much harder.

3.2.2 Approximating Visit Count in Infinite MDPs

For simple 2D-mazes, E^3 and R-MAX can generate good exploration policies, but as soon as the number of states in the state space becomes too big to keep track of, they fall short. One of the reasons for why they fall short is that for huge state spaces every unique state-action pair is rarely, if ever, visited twice. To apply count-based exploration to high-dimensional MDPs we must approximate the counter for the current state. A newly proposed method that manages to use count-based exploration on high-dimensional state representations is presented in (Bellemare et al. 2016). The method is based on computing a pseudo-count from a sequential density model over the state space in the high-dimensional Atari environments. The pseudo-count equation can be represented as:

$$\hat{N}(s_t) = \frac{p_n(s_t)(1 - p'_n(s_t))}{p'_n(s_t) - p_n(s_t)}$$

Here, s_t is the observation, $p_n(s_t)$ is the sequential density model over observation s_t, giving a probability distribution over s_t, and $p_n{}'(s_t)$ is the probability from the density model after observing a new s_t. This equation will give novel states a low score (close to zero) and well-

known states a high score. The score is further used as an intrinsic reward bonus, to shape the obtained reward from the environment. The intrinsic reward bonus is given by:

$$R^{+}(s_t, a_t) = \beta(\hat{N}(s_t) + 0.01)^{-\frac{1}{2}}$$

This equation defines the reward bonus that is added to the extrinsic reward received from the environment. The score is altered by the exploration constant $0 < \beta \leq 1$. This algorithm results in state of the art performance on multiple of the Atari games. It is one of the few methods that manages to learn a long-term policy and achieve a decent score in the notoriously hard environment Montezuma's Revenge. In this environment, effective exploration is a must to achieve a score at all, much because of sparse rewards. Even though this method seems to outperform most other methods when it comes to smart exploration it lacks the possibility to use a prediction model to do planning, extra simulation and all the other possibilities an approximated dynamics model provides. Another point is that this method does not make the agents much more data efficient, it only helps for exploring in hard environments. If we look at training plots for the first 100 million training frames the proposed method performs equal to the model-free A3C.

3.2.3 Dropout as a Bayesian Approximation

In section 2.2 we described a method for how dropout can be used to extract more accurate uncertainty of a DNN model than just using regular softmax. This exact method is described in (Gal & Ghahramani 2015) where they show the effectiveness of using MC dropout during inference with DNNs. Among one of their experiments they implement this form of uncertainty extraction in a regular deep Q-network. The uncertainty extracted from the DQN is used together with Thompson sampling to select the action with the greatest potential (i.e. select the action with the highest expected reward with respect to some uncertainty). This method is tested on a 2D-world where the agent has 9 sensors and 5 actions to control two wheels. The agent must learn how to control itself and to pick up green circles and avoid red circles. The Thompson sampling method is compared with naive decaying ϵ-greedy exploration. Thompson sampling reaches a reward of 1 much faster than ϵ-greedy. A downside to this MC dropout Thompson sampling is that it seems to suffer from premature convergence. Because of no fixed decay in exploration, the new proposed method achieves a lower final score compared to ϵ-greedy.

3.2.4 Bootstrapped DQN

A second way to extract uncertainty from DNNs, described in section 2.2, is by bootstrapping the network. By converting the regular DQN to a Bootstrapped DQN architecture, (Osband et al. 2016) achieves better final results compared to regular DQN on most of the Atari games. A representation of the architecture is displayed in Figure 3.3. The architecture is tested with different numbers of heads, but seems to perform better the more heads used. Of course, this leads to a tradeoff between better performance and computational time. Another important aspect with bootstrapped DQN is that each head observes different data than the other heads. This is because every episode is connected to a Bernoulli sampled mask that tells which head this episode belongs to. That is why it is called bootstrapping the network, each head only observes a part of a bigger dataset.

The authors propose that by making the agent learn with a randomly selected head each episode, the agent will follow slightly different policies and explore more efficiently. Therefore, the proposed method does not utilize the uncertainty possibilities directly, but rather make each head converge to slightly different Q-values and thus achieving slightly different policies each episode. Because the network is trained with an experience replay it is a fine balance of how much data should be shared between the heads. Too much information shared and there will be no difference between the estimates of each head, too little information shared will make the network converge at a slow rate. To find out how much data should be shared between the heads they tested with different probabilities. After the tests, they landed on using 10 heads and sharing all data between the heads as being the most effective configuration. Thompson sampling can be comparable to the proposed bootstrapped DQN, with the main difference being that bootstrapped DQN changes head every episode, while Thompson sampling changes every time step.

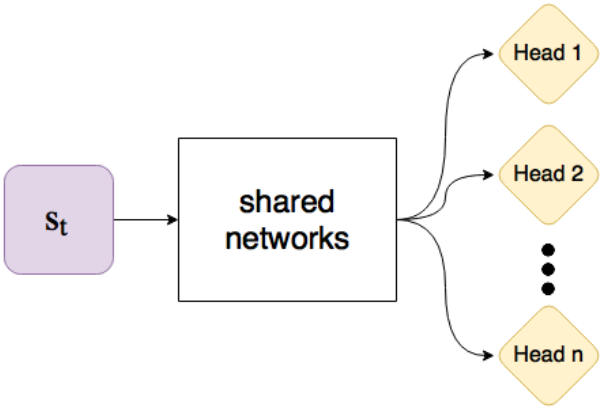


Figure 3.3: Architecture for the proposed bootstrapped DQN.

Bootstrapped DQN reaches human performance about 30% faster than DQN on average for the Atari games. One of the negative sides of this technique is that it is 20% slower than the already slow DQN. This is because of having a greater network (all the heads) and because of the longer training time needed for bootstrapped networks in general. This is observed in the experiments described in Appendix C. Such a slow algorithm is not usable for other than the ones with access to a great amount of computational power.

3.3 Model-Based Exploration Techniques

This section describes the other types of advanced exploration techniques that utilize a dynamics model in some way. The different methods exploit the dynamics model in many ways to create more data efficient agents. However, all methods use the dynamics model to extract some sort of novelty score for the current observation to select the action that will maximize the information about the environment. This guides the agents toward unseen parts of the environment space which leads to faster learning and better data efficiency.

3.3.1 PILCO

As discussed in section 2.2, Bayesian inference can be used to solve the exploration-exploitation trade off optimally by extracting uncertainty from models. Because computing the exact posterior distribution is computationally intractable, this must be approximated. In (Deisenroth & Rasmussen 2011), the proposed method called Probabilistic Inference for Learning COntrol (PILCO) uses a Gaussian Process (GP) to approximate the dynamics of the environment. A GP can be described as a statistical model where each observable point is

defined by its value together with a normally distributed random variable. Easily described as a distribution over functions for the current domain. The advantage of using a GP is that it automatically gives an uncertainty estimate for every new observation. If a new observation is different from previous observations the uncertainty will be higher. The learned dynamics model is used to do model-based policy search combined with gradient based policy improvement. This method achieves great data efficiency, and is one of the most data efficient algorithms for low dimensional robotics problems. Because GPs scales badly for higher dimensions, PILCO does not work for environments with higher dimensional state representations like images.

A newly proposed extension to the PILCO architecture is the Deep PILCO from (Gal et al. 2016), which uses a neural network instead of a GP for the learned dynamics model. This extension achieves the uncertainty estimates as in a GP by using MC dropout. The advantage of doing this is of course being able to model high-dimensional state representations which was one of the main problems with the regular GP PILCO. This means that even environments with states represented as images, such as Atari, can be modelled and explored more efficiently. Even though this makes PILCO usable for high-dimensional problems, (Gal et al. 2016) only puts this as future work.

3.3.2 Intrinsic Motivation with Neural Networks

Another way of affecting the exploration of the agent is by feeding intrinsic reward bonuses for actions that lead to good progress. This is done in (Stadie et al. 2015) by learning a DDM of the environment and giving reward bonuses based on how big the error is from the predicted next step to the real next step. A DDM is just a deep neural network approximating the environment dynamics. The intrinsic reward bonus is defined as,

$$R^{+}(s_t, a_t) = \beta N(s_t, a_t)$$

where β is an exploration constant that controls the amount of bonus added to the extrinsic reward. $N(s_t, a_t)$ is a novelty function for the state-action pair. This novelty function uses a normalized prediction error from the dynamics model to give a number between zero and one. The novelty function is given by,

$$N(s_t, a_t) = \frac{\bar{e}_t(s_t, a_t)}{t * C}$$

where $\bar{e}_t$ is the normalized error between the prediction and the real truth for the current time step t. The novelty function is decayed over time based on the current time step and a decay constant $C > 0$. The authors propose a dynamics model based on DNNs. To make it work for high-dimensional state representations, like images, the authors perform a dimensionality reduction by using an AE. The dynamics model that learns to predict the next state operates on the latent space generated by the AE. This makes it easier for the dynamics model to learn the transition function, and not having to "learn to see" before it can learn to predict. The algorithm gives better exploration results and higher scores early in the learning process than other naive exploration techniques in the Atari environment setting.

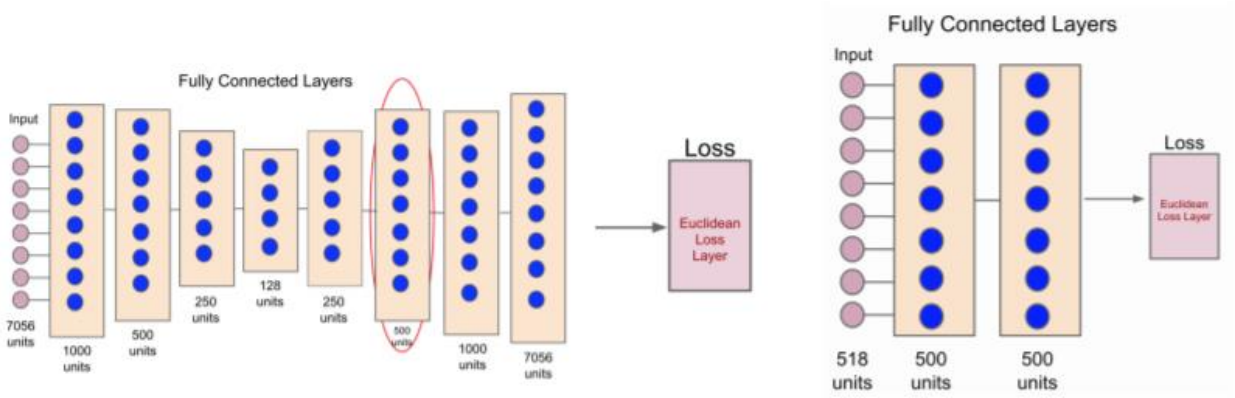


Figure 3.4: Figure from (Stadie et al. 2015) showing the architecture of their DDM . To the left is the AE that is fed with one flattened grayscale image (84x84x1) at the time. The right model is the dynamics model that learns to predict the next state based on the current observation and the selected action. The layer in the AE with the red circle is the latent vector chosen to be input to the dynamics model.

A disadvantage of using AEs in the dynamics model is that AEs tends to be rather data inefficient. This is especially true when it comes to fully connected AEs in combination with images, like the one used in the paper. In total, the proposed AE architecture needs about 15.5M variables to be tuned. This is both computationally heavy for the training phase and the inference phase that happens after every action selection. Because the AE is a computationally heavy model, the paper proposes two methods of training; one is a static

training method where a random policy samples a certain amount of data from the environment and trains the AE on this data once, the second one is a dynamic training schedule that trains on the gathered data after every nth episode. Another problem with this algorithm is that it does not incorporate any time dimension into the dynamics model. Hence, the AE takes one frame at the time, and there is no recurrence in the dynamics model. This means that any important information that requires a sequence of frames in the environment will be lost. The last concern with this method, which also is mentioned by the authors, is that the model has no way of separating the environment uncertainty and the model uncertainty. In other words, in a stochastic environment, the agent would be drawn towards stochastic states because they seem to be novel states (always yield high errors from the dynamics model). In truth, these states might not be as good as another more deterministic states. The best guidance selects the actions that maximizes the model uncertainty independent of what the environment uncertainty is.

3.3.3 Intrinsic Motivation with Bayesian Neural Networks

A similar method to (Stadie et al. 2015) is described in (Houthooft et al. 2016), where the reward bonus is described by the information gain by having taken action a_t in state s_t. This method is named Variational Information Maximizing Exploration (VIME). One major difference compared to (Stadie et al. 2015) is that the dynamics model is represented by a Bayesian neural network. The Bayesian neural network is a regular DNN trained with stochastic gradient variational Bayes, also known as Bayes by Backprop from (Blundell et al. 2015). This is a modified backpropagation optimization technique for approximating a distribution over all the weights. Hence, the representation of the environment is given by a distribution rather than a single point estimate. The information gain is calculated by taking the KL divergence of the posterior distribution before taking the action and the distribution after taking the action. This can be written as:

$$IG(s_{t+1}; \Theta|\phi_t) = \mathbb{E}_{s_{t+1}}[D_{KL}[p(\theta, \phi_{t+1})||p(\theta, \phi_t)]]$$

Where p is the variational distribution parameterized by the random variable Θ with the values $\theta \in \Theta$, while ϕ_t represents all history up to the current time step t. This is formalized as $\phi_t = \{s_0, a_0, s_1, a_1, \ldots, s_t\}$. The final intrinsic reward bonus is given by:

$$R^{+}(s,a) = \beta D_{KL}[p(\theta, \phi_{t+1})||p(\theta, \phi_t)]$$

Where β is an exploration constant that controls the amount of curiosity bonus fed to the agent. This exploration technique achieves results faster and in many cases, higher final performance on multiple low-dimensional continuous control tasks. Some of the environments tested are CartPole, MountainCar and Walker2D. A limitation of using this technique, with Bayesian neural network, is that these models do not scale very well. This is reflected in the low-dimensional state representations in all the tested environments. That is why Bayesian neural networks are hard to use with other high-dimensional state representations like images, as used in the Atari domain.

3.4 Summary

We see a trend towards using methods that extracts measures for how novel a state or transition is going to be for the agent. This measure can then be used to guide the agent towards novel states more often. The motivation is often done by feeding intrinsic reward bonuses to the agent. The different kinds of bonuses are based on uncertainty as seen in (Gal & Ghahramani 2015; Houthooft et al. 2016), state counting as in (Bellemare et al. 2016; Ostrovski et al. 2016) and dynamics prediction errors as in (Stadie et al. 2015). Another factor we see is that many of the explained methods use an approximated dynamics model to calculate the intrinsic reward. From this standpoint, we will look further into both aspects. The method proposed in this thesis will try to solve some of the problems with the current state-of-the-art methods by combining some of the best ideas.

4 Methodology

The proposed DDM is a generic module that fits with most types of RL techniques. For this thesis, all experiments and testing are done with PAAC (described in section 2.4.3). This is different from most of the work described in chapter 0 where DQN is the common algorithm choice. The advantage of using PAAC compared to DQN is much faster learning on a single computer together with higher final performance on many of the Atari environments. The proposed algorithm combines the described DDM from section 4.1 with PAAC. As described in section 2.3.4, there are many ways of using an approximated dynamics model. For this thesis, the DDM is used to generate different kinds of intrinsic reward bonuses, inspired by (Bellemare et al. 2016; Stadie et al. 2015; Houthooft et al. 2016). This model-based PAAC (MB-PAAC) algorithm is explained in section 4.2.

4.1 Deep Dynamics Model

One of the main parts of the proposed system is the DDM which is a DNN that tries to approximate the state transitions of the environment. The architecture is inspired by some of the key elements of the deep dynamics models described in chapter 0. The work is especially inspired by (Oh et al. 2015; Wahlström et al. 2015; Watter et al. 2015; Stadie et al. 2015). A figure of the proposed architecture is shown in Figure 4.1. A repeating pattern for most of the DDMs, particularly the ones handling high-dimensional state spaces, is the ability to compress the state representation into a denser representation. The idea behind this is that the dense representation is likely to contain important features of the observations. Thus, making the transitions easier to approximate. This follows an old saying in machine learning, “garbage in, garbage out”. Another benefit of using a denser representation is that many techniques are limited to smaller models, like Bayesian neural networks for example. These techniques can be used to learn the dynamics after the compression is done.

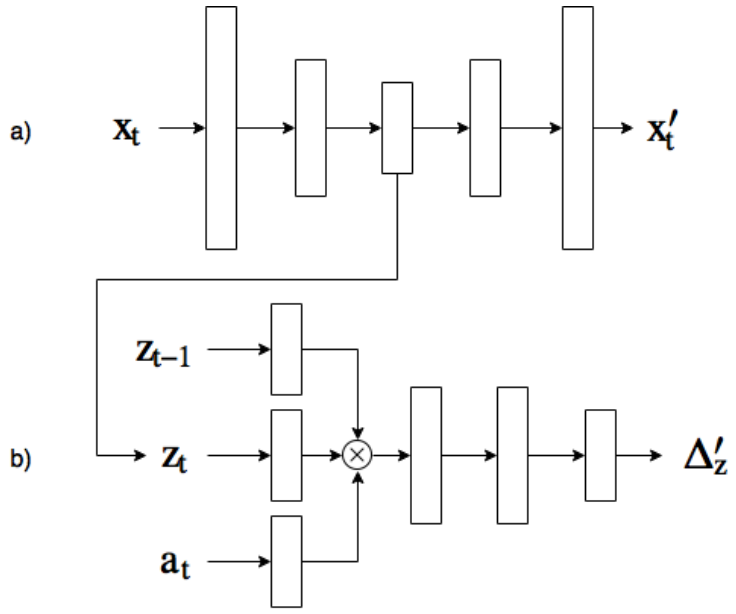


Figure 4.1: Architecture of the proposed DDM. a) The autoencoder, compresses a high-dimensional input x_t into a latent representation z_t that contains important features. To predict the state transition, the latent variable is concatenated with the selected action and the previous latent variable. b) The concatenation is then fed into a series of fully connected layers, the transition prediction model (TPM), to predict the difference between the next latent variable and the current latent variable. This prediction is given as Δ'_z.

4.1.1 Feature Extracting

For the proposed system, an AE is used to compress the observation into a latent space, similar to (Oh et al. 2015; Wahlström et al. 2015; Watter et al. 2015; Stadie et al. 2015). The output of the AE model is defined as $M_A(x_t) = (z_t, x'_t)$, where z_t is the compressed latent variable for input x_t. We define the input x_t to be a part of the full observation s_t. This means that if s_t contains multiple images, x_t might represent one of the images. For instance, in the Atari setting, x_t would typically be one of the four images in s_t. The use of a single image simplifies the AE. The reconstructed input is given as x'_t. When it comes to architecture selection, there is a tradeoff between computational effectiveness and reconstruction precision. It is important to keep the model relatively small to maintain a short training and inference time. On the other hand, to be an effective AE, it should be able to fill the latent space with good features and make proper reconstructions. Therefore, four different architectures were implemented and tested. See Table 4.1 for descriptions for each proposed architecture. More details about their architectures will be explained in chapter 5.

Table 4.1: Overview of the four proposed autoencoder architectures. More detailed architecture choices are described in section 5.3.1.

AE type	Description
Fully Connected (FC)	A series of fully connected layers with decreasing size in the encoder and increase in size for the decoder
Convolutional Max Pooling (CMP)	The encoder uses a combination of convolutional layers and max pooling layers. The middle layer is still fully connected. The decoder uses a combination of convolutional layers and upsampling layers.
Convolutional Deconvolutional (CD)	The encoder is a series of convolutional layers. The middle layer is still fully connected. The decoder uses a series of deconvolutional layers. Inspired by the architecture in (Oh et al. 2015).
Variational Convolutional Deconvolutional (VCD)	Same as CD, but converted into a variational autoencoder. This was inspired by (Oh et al. 2015; Watter et al. 2015).

4.1.2 Difference-Based Attention Loss

One of the most important factors to make this architecture to work is the feature extraction done by the AE. Because AEs in general are trained to minimize the reconstruction error, they often learn more precise features for objects that take up a greater part of the observation. Thus, smaller details will most likely be squished out from the latent space to make better reconstructions of the bigger parts. This is a good property of AEs when it comes to feature extraction for other problems like image classification. However, when it comes to RL, often the most important parts of an observation might not be the biggest part of the observation, but rather a small detail. This is discussed in (Sandven 2016) as being one of the biggest problems for AEs in the RL setting. Therefore, to force the AE to learn the small important details, we must penalize it for removing the small moving details from the reconstruction.

The advantage of being in an RL setting is that we have sequences of images. Because the DDM is trying to learn what the changes are from the current state to the next state, we can consider the parts of the image that change the most important to compress into the latent space. The proposed penalization uses the change from the current state to the next state to

see what the AE should be focusing on. This is done by increasing the AE loss for parts of the reconstruction that contained a change. This forces the AE to learn to map all moving parts of the observation to the latent space. More formally, we create a loss-matrix which defines how each pixel should be penalized. Parts of the image that do not change are set to μ, while changing parts are set to 2μ, where μ is a constant bigger than 0.5. We can formalize this loss multiplier as:

$$L^*(x_{t,i}, x_{t+1,i}) = \mu\lceil x_{t+1,i} - x_{t,i}\rceil + 1$$

Here, $x_{t,i}$ represents pixel i of observation x_t. If we combine MSE with this penalizing matrix as a multiplicative factor in the loss for each pixel we get:

$$L_{MSE^*}(x_t, x_{t+1}) = \frac{1}{n}\sum_{i=1}^{n} L^*(x_{t,i}, x_{t+1,i})((x'_{t,i} - x_{t,i})^2) \tag{4.1}$$

This finds the loss for all n pixels in the image and then calculates the mean for the whole image. The attention oriented loss makes the AE focus on the moving parts of the observation rather than on the biggest parts.

4.1.3 Transition Prediction Model

The main idea behind the proposed architecture is to do all the transition predictions in the latent space learnt by the AE. The Transition Prediction Model (TPM) predicts the next latent state directly. To do this it needs the latent variable for the current observation and the action selected at that time step. For this model, the two inputs are concatenated and fed into a regular fully connected DNN. The basic output of TPM can be written as $M_T(z_t, a_t) = z'_{t+1}$. With the inferred z'_{t+1} we can get the full sized reconstructed observation x'_{t+1} for the next time step by feeding it into the decoder part of the AE. Even though this is not directly needed for this study, it is a good property to have for observing and verifying the predictions done by the TPM.

As proposed in (Anderson et al. 2015) making a neural network learn to predict the next state directly may cause the network to learn the identity function instead of the transition

function. This is because the change between the current time step and the next time step is small. Therefore, a small change is done to the targets of TPM. Instead of predicting z'_{t+1}, it learns to predict $\Delta_z = z_{t+1} - z_t$. This forces the TPM to learn the actual changes of each action. We rewrite the output target for the TPM to be $M_T(z_t, a_t) = \Delta'_z$.

One of the drawbacks of (Stadie et al. 2015) was that it did not have any time dependency in the dynamics model. This is solved for the proposed dynamics model by adding the previous latent variable as an additional input to the TPM. Again, we can rewrite the TPM output to be $M_T(z_{t-1,}z_t, a_t) = \Delta'_z$. This final formalization of the TPM matches the representation in Figure 4.1.

The problem of time dependency could also have been solved by using a recurrent neural network. The main drawback of using a recurrent neural network is the bad computational performance compared to a regular feedforward neural network. Recurrent neural networks might be able to capture longer time sequences, but based on the results found in (Oh et al. 2015) there were no significant difference between using a recurrent architecture compared to a regular feedforward architecture. Therefore, the more efficient feedforward architecture with multiple time steps as input was selected.

4.1.4 Experience Replay

As stated in section 2.1, training DNNs with stochastic gradient descent assume an independent and identically distributed dataset. When employing gradient descent to train an RL agent that gets correlated and non-identically distributed data from the environment, it can end in a biased and poorly trained neural network. A solution for this problem was discussed in (Mnih et al. 2015) where they introduced an experience replay buffer to decorrelate the observations. Along with decorrelation, experience replay can also benefit from selecting distinct samples that are rarer. The proposed system does also benefit from using an experience replay. Even though on-policy RL techniques cannot handle an experience replay directly without modifications, the DDM using SL will benefit from having a decorrelated training buffer. Additionally, with the use of multiple agents in parallel on could gather data from different parts of the environments for the same time step. This could help decorrelate the data, as discussed in (Mnih et al. 2016; Clemente et al. 2017).

4.1.5 Training Schedule

As mentioned in (Stadie et al. 2015), the DDM can be trained both statically on a pre-gathered dataset or trained dynamically through the learning process of the agent. The static training schedule would work fine for environments that stay the same as the agent learns to optimize its policy. A common property for all RL environments where a static training schedule work, is that they are easy to explore. Often, a small dataset gathered from a random agent would give a decent representation of the full MDP. The dynamic schedule would be capable of learning more advanced environments that change as the agent learns. The downside to a dynamic training schedule is the problem of defining a well-working schedule. This is a problem because of the challenges that comes with convergence speed of the DDM compared to the policy and value network. Another problem is to keep the model from overfitting to the observations. This may easily occur when a DDM is trained with efficient SL updates together with inefficient RL updates.

4.2 Sample Efficient Model-Based PAAC with Intrinsic Motivation

Curiosity-based intrinsic reward bonuses can be generated in many ways. In this section, we propose four different ways of extracting them from the DDM. The first way of generating intrinsic reward bonuses is the reconstruction loss directly outputted from the AE. The idea behind using this reconstructions loss is that the agents will be motivated to observe novel states. This idea assumes that the AE will generate a higher loss for novel states compared to a lower loss for more common state. One advantage of this bonus is that it does not need the TPM model and will therefore have better computational efficiency. The second type of intrinsic reward bonus is similar to the bonus from (Stadie et al. 2015) where the bonus is the received prediction error from the DDM. This should guide the agents towards more novel state transitions. The third type of intrinsic reward bonus uses the uncertainty in the TPM for the current state and the selected action. This idea is related to (Houthooft et al. 2016) where they extract the surprise of the dynamics model with an approximated KL divergence of the true transition probabilities from a Bayesian neural network. However, in this work, the uncertainty is calculated in a more computationally efficient manner by getting the average variance of the output when using MC dropout. This is done by feeding the same input multiple times through the model with stochastic dropout masks. State-action pairs that generate a high variance in the output are assumed to have high uncertainty, thus, giving a high bonus. Because the TPM is just a small network, we do not have to perform MC dropout

on the entire DDM. This form of intrinsic reward is closely related to Bayesian learning where we have both a point in the space and the uncertainty about that current point. The last intrinsic reward bonus utilizes this way of calculating uncertainty, however, the TPM architecture is modified to have multiple output heads. This is inspired by the Bootstrapped DQN architecture from (Osband et al. 2016).

As stated by many of the methods explained in chapter 0, the intrinsic rewards should follow a decaying schedule such that the true extrinsic rewards received from the environment counts more and more during training. Below is a summary of the different kinds of intrinsic rewards and how the reward bonus is calculated. For all equations, e is represented as the current environment from PAAC. A representation of each bonus type can be found in Figure 4.2.

1. **Autoencoder Loss (AL):** By using the loss generated from the AE as an intrinsic reward bonus, the agents are guided towards novel states. The AE loss is converted to a reward bonus by normalizing it on a moving maximum of the AE loss L_{AE}. The bonus can be written as:

$$R_{AL}^{+}(s_{t,e}) = \frac{L_{AE}(s_{t,e})}{max_{L_{AE}}} \tag{4.2}$$

2. **Dynamics Loss (DL):** By using the loss generated from the dynamics model as intrinsic reward, the agents are guided towards novel state transitions. The bonus is calculated in the same way as AL, but by using the loss from the TMP, L_{TPM}. This is formalized as:

$$R_{DL}^{+}(z_{t-1,e}, z_{t,e}, a_{t,e}) = \frac{L_{TPM}(z_{t-1,e}, z_{t,e}, a_{t,e})}{max_{L_{TPM}}} \tag{4.3}$$

3. **Dynamics Uncertainty (DU):** By using the uncertainty, or surprise, extracted from the dynamics model as intrinsic reward, the agents are guided towards novel state transitions in the same way as with DL. However, the difference of using uncertainty instead of the calculated loss between an observation and a prediction is that the DL only represents the error from that current combination of weights. Because the uncertainty is extracted by MC dropout as proposed in (Gal & Ghahramani 2015) and described in section 2.2, we get a measure over many possible solutions to the observed data which should give a more accurate uncertainty value. The downside to

DU compared to AL and DL is of course a longer computational time. We define the output distribution over the T stochastic forward passes of the TPM model M as:

$$\Phi \sim \underset{i=1,2,\ldots T}{M(z_{t-1,e}, z_{t,e}, a_{t,e})_i}$$

We define the standard deviation of the output distribution Φ as σ_Φ. The uncertainty is converted to a reward bonus by taking the average over the standard deviation for each dimension. When the size of the latent space is given as $|z|$, this bonus is formalized:

$$R^+_{DU}(z_{t-1,e}, z_{t,e}, a_{t,e}) = \frac{1}{|z|}\sum_{i=1}^{|z|}\sigma_\Phi(z_{t-1,e}, z_{t,e}, a_{t,e})_i \tag{4.4}$$

4. **Bootstrapped Dynamics Uncertainty (BDU):** We combine MC dropout with bootstrap to hopefully achieve a more accurate uncertainty measure of the model. To use this bonus, we modify the network architecture for the TPM to include multiple output layers, or heads. The idea is that each head might learn the observed data in different ways and move towards the real truth from different angles in the solution space. See Figure C.4 in Appendix C for a visualization of bootstrap combined with MC dropout. The bonus is calculated as in DU, but with an average over all the heads. When h represents the number of output heads we can formalize this intrinsic reward bonus as:

$$R^+_{BDU}(z_{t-1,e}, z_{t,e}, a_{t,e}) = \frac{1}{h|z|}\sum_{k=1}^{h}\sum_{i=1}^{|z|}\sigma_\Phi(z_{t-1,e}, z_{t,e}, a_{t,e})_{i,k} \tag{4.5}$$

4.2.1 Overcoming Environment Uncertainty

As described in chapter 0, one of the greatest problems of using the loss or the uncertainty directly from the dynamics model is that we cannot differentiate between model uncertainty and environment uncertainty. This problem was mentioned in (Stadie et al. 2015) to be one of the limits of using the loss from the dynamics model. What we want is to extract the model uncertainty, not the environment uncertainty, for any state-action pair to guide the agent towards transitions that are novel to the agent and not just highly stochastic. To overcome

this problem, we introduce the loss from the AE as a metric describing how many times the agent has seen this state. This constant is based on the assumption that a frequently visited state will yield a small loss from the AE, while a novel state will give a high loss. This can be used to increase or decrease the reward bonus from the dynamics model such that the agent is guided towards novel transitions. We can write the following assumption: if the AE loss is small and the dynamics loss is high, we have a stochastic transition. Stochastic transitions should not be highly rewarded. The constant can be calculated in the same way as the AL bonus is calculated. Therefore, for stochastic environments, the reward bonuses from DL, DU and BDU can be multiplied with this adjustment constant to avoid some of the stochasticity in the environments.

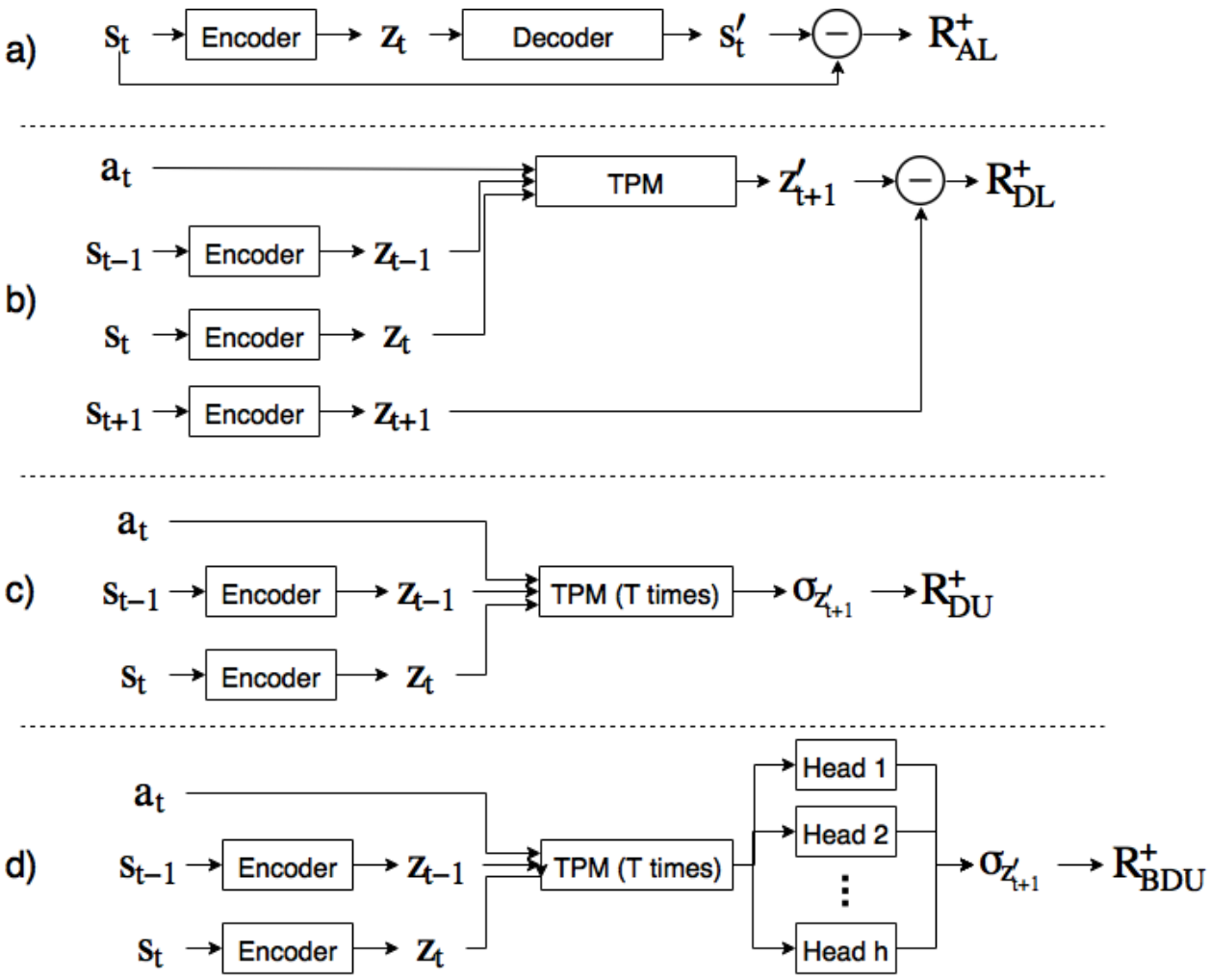


Figure 4.2: Detailed representation of the intrinsic reward bonuses and how they are calculated, starting from AL (a), DL (b), DU (c), BDU (d).

Algorithm 1: MB-PAAC with intrinsic motivation

Initialize PAAC as in Algorithm 1 from (Clemente et al. 2017)
Initialize DDM
Initialize empty shared replay memory buffer B
repeat
  for t=1 to t_{max} **do**
    Sample a_t from $\pi(a_t|s_t)$
    Calculate v_t from $V(s_t)$
    parallel for i=1 to n_e **do**
      Perform action $a_{t,i}$ in environment e_i
      Observe new state $s_{t+1,i}$ and reward $r_{t+1,i}$
      B $\leftarrow (s_{t,i}, s_{t+1,i}, a_t)$ // Store the transition data
    end parallel for
  end for
$R_{t_{max}+1} = \begin{cases} 0 & for\ terminal\ s_t \\ V(s_{t_{max}+1}) & for\ non-terminal\ s_t \end{cases}$
Calculate R_t^+ with one of the equations; (4.2), (4.3), (4.4) or (4.5)
for t=t_{max} down to 1 **do**
  $R_t = r_t + \gamma R_{t+1} + \beta R_t^+$
end for
Train PAAC by steps 16-19 in Algorithm 1 from (Clemente et al. 2017)
for j=1 to k **do**
  Sample random batch b_j from B
  Train AE with batch $b_{j,t}$
  Calculate latent variables $z_{j,t-1}$, $z_{j,t}$ and $z_{j,t+1}$ from b_j with AE
  Train TPM with $(z_{j,t-1}, z_{j,t}, a_{j,t})$ as input and Δ_z as target
end for

5 Experimental Setup

In this chapter, we describe the environments, detailed architecture settings and how the hyperparameters were selected for the final experiments. At first, in section 5.1, we give a brief explanation of the main experiments. Section 5.2 contains an introduction of the five selected Atari games and why they were selected. In section 5.3 the process of selecting the best architecture for the AE will be presented. Finally, in section 5.4, the important settings for the MB-PAAC will be elaborated.

5.1 Introduction

The main experiments for this thesis explore different types of intrinsic motivation for RL agents. As described in section 4.2, this thesis uses the newly proposed PAAC as the underlying model-free RL algorithm. To focus on the DDM and the intrinsic motivation, the code was modified from the original code from PAAC. This code uses Python[3] with Tensorflow[4] to compute the tensor operations for the DNNs in an efficient manner. The modified MB-PAAC implementation can be found under `https://github.com/mrminy/Master_Thesis` along with the code for the other experiments in this study.

5.2 Environments

The selected environments for the experiments are from the commonly used Atari 2600 domain. The environments are from the open sourced Arcade Learning Environment[5] (ALE) and are made specifically for training RL agents in high-dimensional episodic environments. In ALE, each observation is an RGB frame of size 210x160. It is common to use a series of four consecutive frames to determine the state (Mnih et al. 2013). As most RL algorithms handling such high-dimensional state representations, the images are preprocessed into 84x84 grayscale images. Therefore, each observation that is analyzed by the agent is of size 4x84x84. There are at most 18 possible actions for the agent to choose from (some of the environments utilize less than 18 actions). Only one action can be selected at a time. The

[3] https://www.python.org/
[4] https://www.tensorflow.org/
[5] http://www.arcadelearningenvironment.org/

action descriptions and ids are displayed in Table 5.1.One of the main reasons for why Atari 2600 was selected as the environment domain was its popularity. Most RL methods tested in high-dimensional environments use Atari as a benchmark. This makes it easier to compare to other methods. Another reason was that Atari contains many games with different MDP properties. However, because of time and computational constrictions, only a few of the environments were for the experiments. Table 5.2 gives an overview of the exploration characteristics for all the selected environments. A more detailed analysis of each selected environment can be found in Appendix A.

Table 5.1: The 18 possible actions in the Atari domain. Not all games utilize all 18 actions.

no-op	fire	up	right	left	Down
up-right	up-left	down-right	down-left	up-fire	right-fire
left-fire	down-fire	up-right-fire	up-left-fire	down-right-fire	down-left-fire

Table 5.2: Comparing the selected environments based on exploration properties.

Easy exploration	**Hard exploration**
Pong Breakout Seaquest	Q*Bert Montezuma's Revenge

5.3 Network Architectures

The MB-PAAC algorithm requires three neural networks; one for the regular model-free algorithm calculating the policy and advantage, a second one for the AE used to compress the observations into a latent vector and a third for the TPM approximating the transitions in the latent space generated from the AE. The selection of the AE architecture is described in section 5.3.1 below. For all following figures, the different network architectures are described with the following layer definitions:

- Fully connected(output) → fc
- 2D convolution(filters, cols, rows, stride, padding) → conv
- 2D max pooling(size) → maxpool
- 2D upsampling(size) → upsampling
- 2D deconvolution(filters, cols, rows, stride, padding) → deconv
- Softmax(output) → softmax

The two other networks have their architectures represented in Figure 5.1 (PAAC) and Figure 5.2 (TPM). ReLU was selected to be the activation function for most of the layer connections in the networks, except from TPM which was found to work best with the Tanh activation functions.

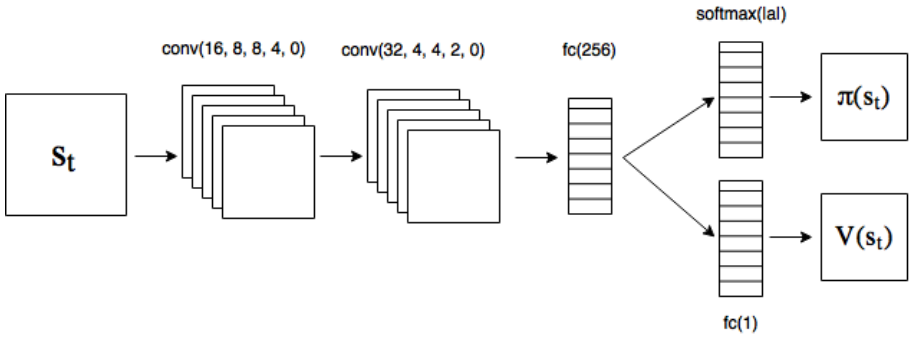


Figure 5.1: Architecture of the model-free PAAC network.

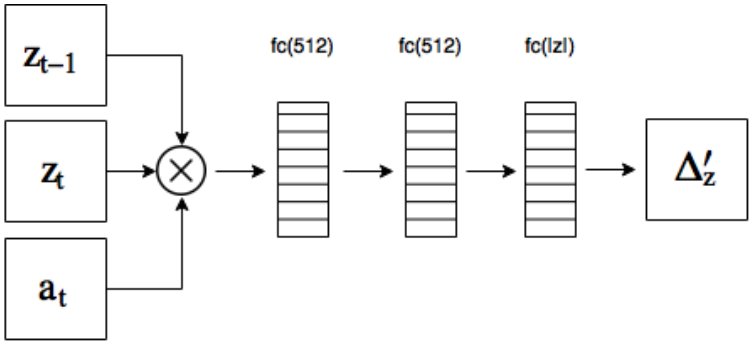


Figure 5.2: Architecture of the TPM network. The inputs are first concatenated and sent into the deep fully connected neural network. Dropout are performed on the two fully connected layers with output size of 512 during training and during inference for the MC dropout. When the network is bootstrapped, the last fully connected layer is copied multiple times in parallel, just like in figure Figure 3.3.

5.3.1 AE Architectures

As explained in 2.1.5, AEs can be structured in indefinitely many different layer combinations and sizes. Therefore, to find the best working architecture, the four architecture types described in 4.1.1 were compared with MB-PAAC in Breakout. Reconstruction images together with the original images were sampled at a fixed rate for all architectures through the training. Each agent ran for 10^7 global time steps. This way the reconstructions could be

compared at different points in time during training. Figure 5.3 contains representations of the four different AE architectures. All AEs were trained with L_{MSE^*} from equation (4.1), with $\mu = 4$, except for the VCD which use the binary cross entropy version of the loss functions for variational AEs as explained in section 2.1.5.

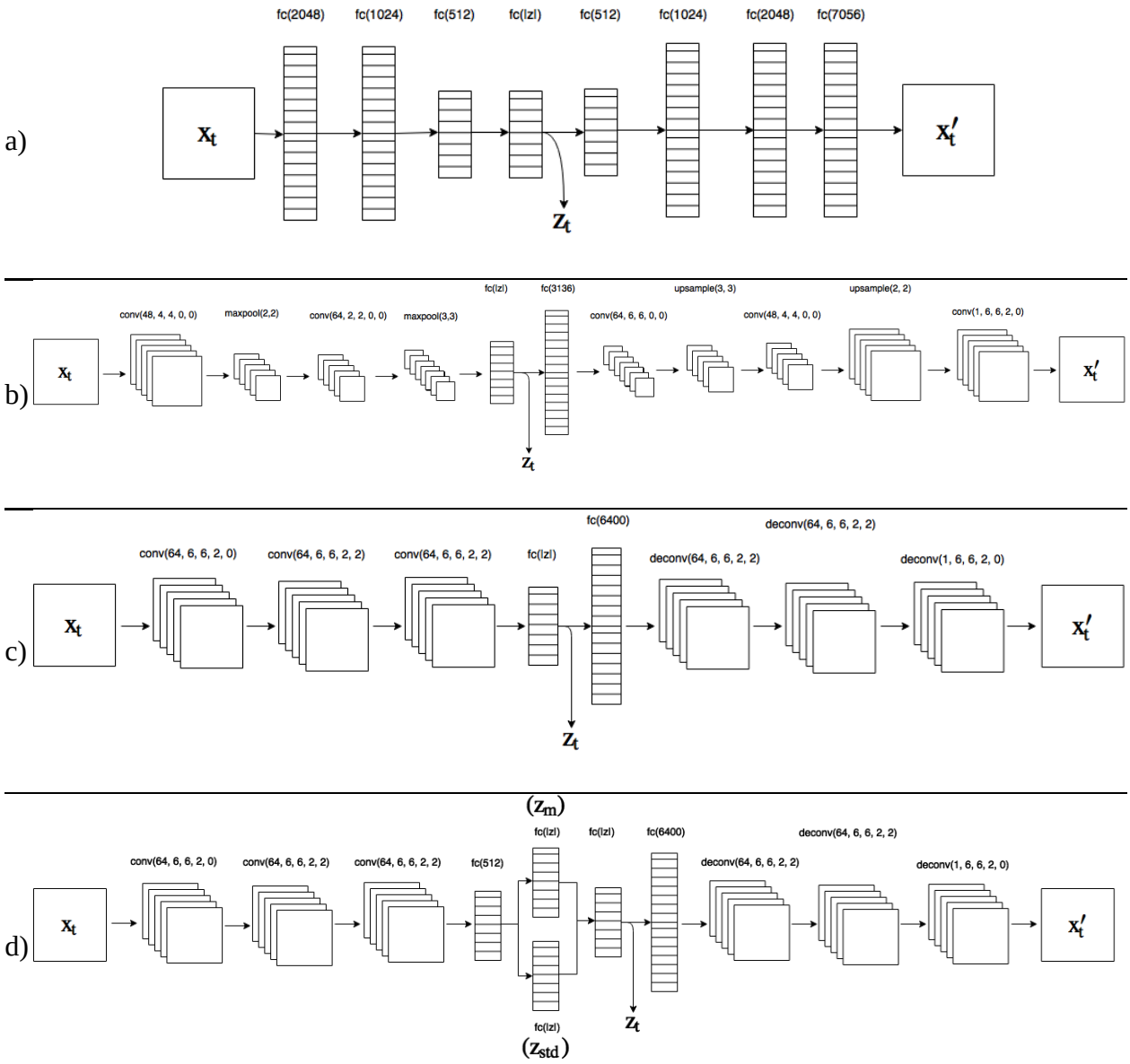


Figure 5.3: Visualization of the four different autoencoder architectures described in 4.1.1; Fully Connected (a), Convolutional Max Pooling (b), Convolutional Deconvolutional (c), Variational Convolutional Deconvolutional (d). All the hidden layers use ReLU as the nonlinear activation function except the two separated fully connected layers (z_m and z_{std}) in VCD that use a regular linear activation function.

5.4 Hyperparameters

For all experiments performed on the five environments, the same hyperparameters for the original PAAC were selected as described in Appendix B. Most of the final hyperparameters for the DDM were selected based on experimentation.

One of the most crucial hyperparameters was the tuning of the decaying schedule for the exploration constant which decides how much intrinsic reward is given. The final schedule was set to $\beta_{start} = 0.1$, $\beta_{end} = 0.0001$ with a linear decay over the full training period. Another important hyperparameter is the size of the latent variable which was chosen to be $|z| = 256$. The selected dynamic training schedule was training the DDM for 100 batches every 5120 ($t_{max} \cdot 1024 = 5120$) time steps where each batch contains 32 images. The experience replay size could have been set higher, but to limit the memory usage, it was set to be a first-in-first-out buffer with maximum 16k observations.

To keep the TPM from overfitting a dropout rate of 0.05 was added to the two fully connected layers. The same rate was used for the MC dropout inference. The number of heads were set to $h = 10$ for the BDU bonus. The different AEs were trained with an initial learning rate of $5{\times}10^{-4}$ using the Adam optimizer for backpropagation. Different learning rates were tested, but the DDM was found to be unstable and not performing well when getting out of the range $[10^{-5}, 10^{-3}]$. The TPM networks were trained with regular MSE, described in equation (2.2).

6 Results and Discussion

This chapter presents the results of the MB-PAAC tested in the high-dimensional Atari domain. Section 6.1 starts by showing how well the different AE architectures manage to capture the environments. Then, in section 6.2, the results of difference-based attention loss from section 4.1.2 are discussed. Furthermore, in section 6.3, we look at how well the full DDM manages to predict future. In section 6.4 we show the results from the main experiments, comparing the four types of intrinsic reward bonuses. In section 6.5 we explore the different computational performance of the different methods. We finish off in section 6.6 with a discussion of the proposed architecture.

6.1 Autoencoder Results

Figure 6.1 is a summary of the main results gathered from the comparison of the AE architectures. The figure lists the four architectures at different time steps during the training in Breakout. We can see that after 40k time steps (first column) no architecture manage to learn to integrate the moving paddle. The only differences are the quality of the reconstruction, where some of the outputs are grainier than others. Moving to 200k time steps (second column) we see that all architectures have managed to include the paddle and its position into the reconstructed image. One detail that differentiates the AEs is the scores listed at the top of the observations, where all but VCD manage to reconstruct the score numbers accurately. Still, the ball is not reconstructed by any of the architectures. It is when we move on to 2M time steps (third column) it gets interesting, because the only architecture that manages to reconstruct the ball is the CD architecture. Whereas all the other architectures manage well to reconstruct the paddle, the score and the destroyed bricks, they do not manage to reconstruct the small and fast moving ball. In fact, when we look at 10M time steps (fourth column) they never manage to place the ball into the latent space. The CD architecture prints out the crispest reconstruction as well. Based on these results, the CD architecture was selected for further experiments.

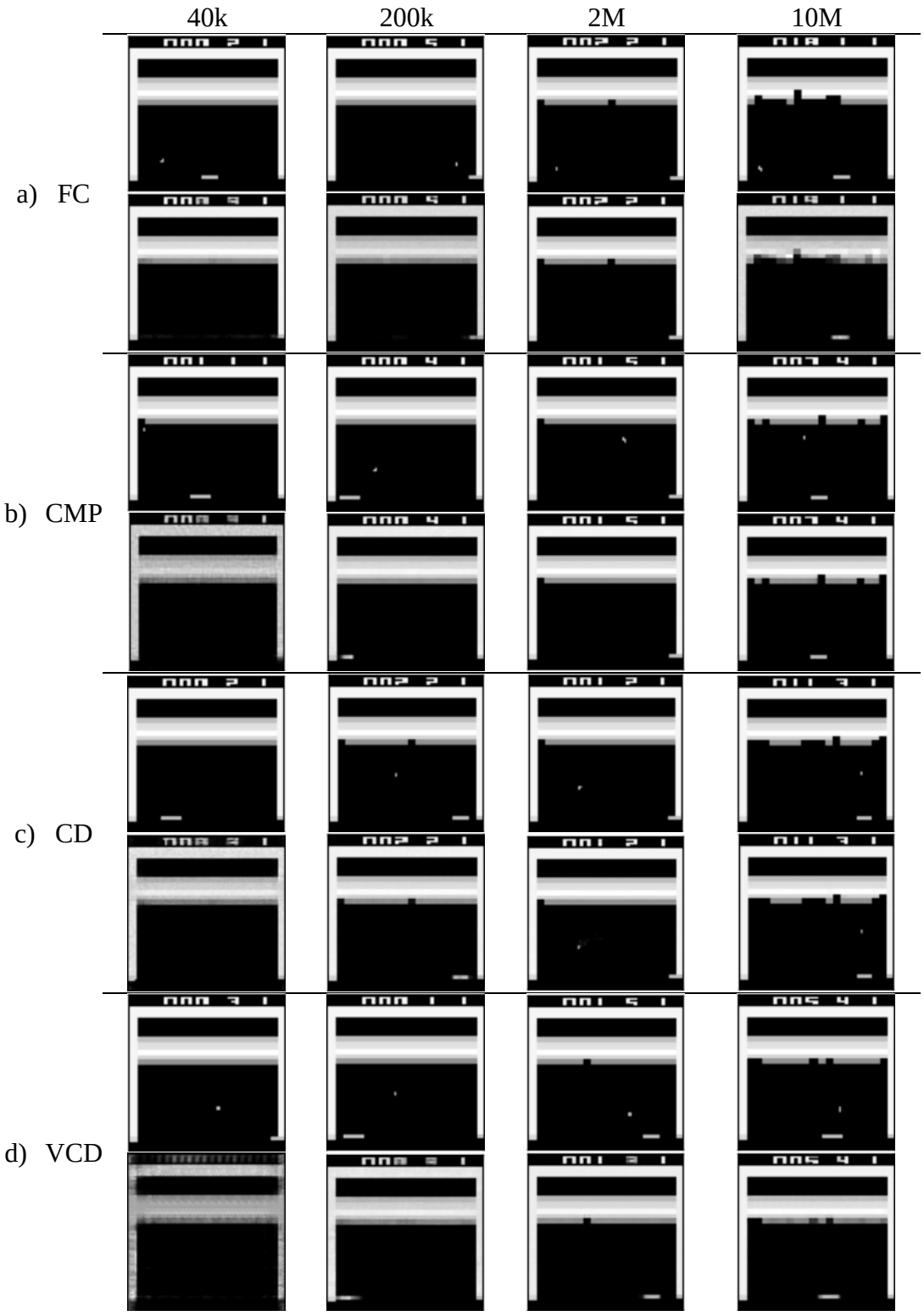


Figure 6.1: Comparing different autoencoder architectures during different time steps (listed at the top of the figure) of for the RL agent. Each architecture has two rows where the first row is the observed image and the second row is the reconstructed image.

6.2 Attention Loss for Autoencoder

Another experiment was performed to see what difference the difference-based attention loss makes. The difference-based attention loss should make the AE focus on parts of the image that change during the state transitions. Using the attention loss made the AE learn a better representation for the environments' changing parts much faster. For example, when used in environments containing small changing objects, like the ball in Pong or Breakout, it was able to learn reconstructions faster. This is shown in Figure 6.2 where attention loss is studied for Breakout. After 205k time steps the AE using the attention loss reconstructs a much clearer picture than the one without attention loss. It is even capable of reconstructing the changing score at the top of the screen perfectly. However, both AEs fail to reconstruct the ball at this level. The first reconstruction of the ball happens after 1884k time steps, but only for the AE using attention loss. The AE without attention loss manages to successfully reconstruct the ball after 2028k, almost 150k time steps after the one with attention loss. Therefore, the attention loss were used for all experiments.

6.3 DDM Predictions

We start by looking at how well the DDM manages to capture the different mechanics of the environments. If we look at Figure 6.3 we see that most of the important dynamics of the Breakout environment are captured by the model. This figure is fetched after training for 10M time steps. If we look at the leftmost column showing the predicted transition when selecting the "no-op" action. No-op should not make any adjustments to the environment other than what the environment controls by itself, which in this case, is the movement of the ball. As we see, the paddle is completely still, but the ball is reconstructed a tiny bit up to the left compared to the original image. The same story goes for the second action, fire, which should be the same as the no-op action when the ball is present on the screen. Finally, we see that the DDM has managed to understand what happens in the environment. When action right and action left is selected, the paddle is moved right and left respectively. This example shows that the dynamics model is capable of learning the dynamics for each action accurately. Other frames may cause the dynamics model to not model any change, even though it should have been one. The reason for this may be that there have not been enough examples of this current transition. Another reason may be that the AE does not reconstruct the image properly, which causes mistakes in the TPM prediction.

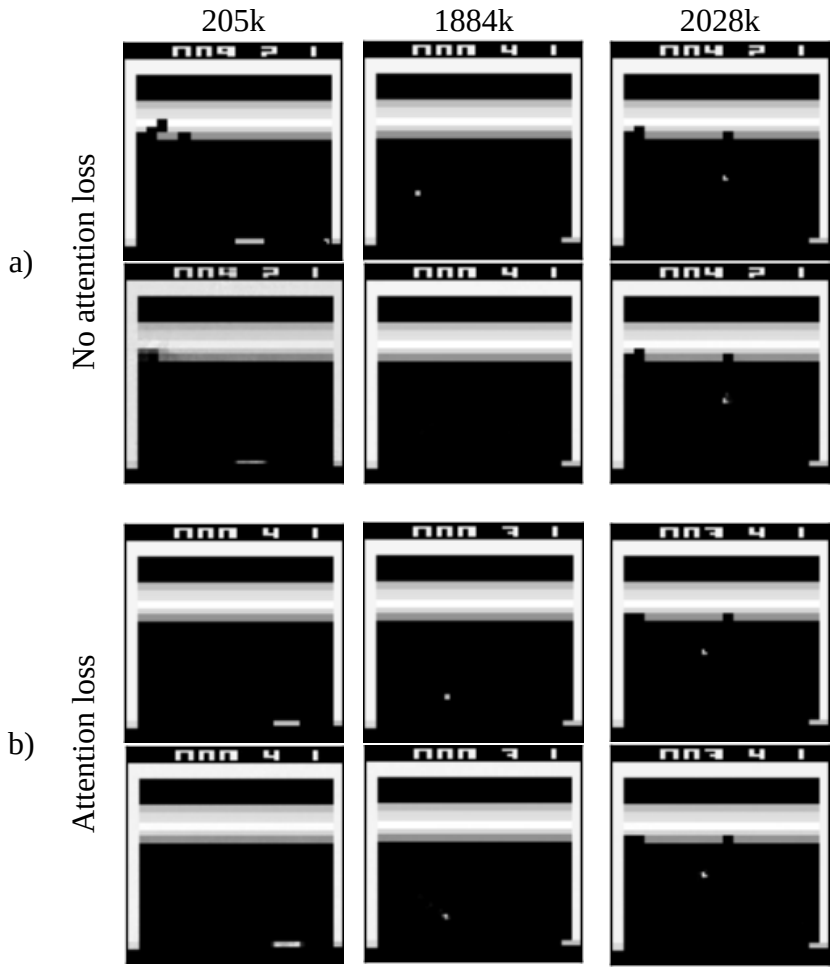


Figure 6.2: Results of using difference-based attention loss (b) versus not using it at all (a). The situations displayed in the figure emphasize that the attention loss helps the autoencoder learn the changeable parts of the observations much faster. The first rows from each pair represents the preprocessed image observed from the environment, while the second rows represent the reconstruction generated by the AE. At the top is the time step stamps for when the reconstructions were sampled.

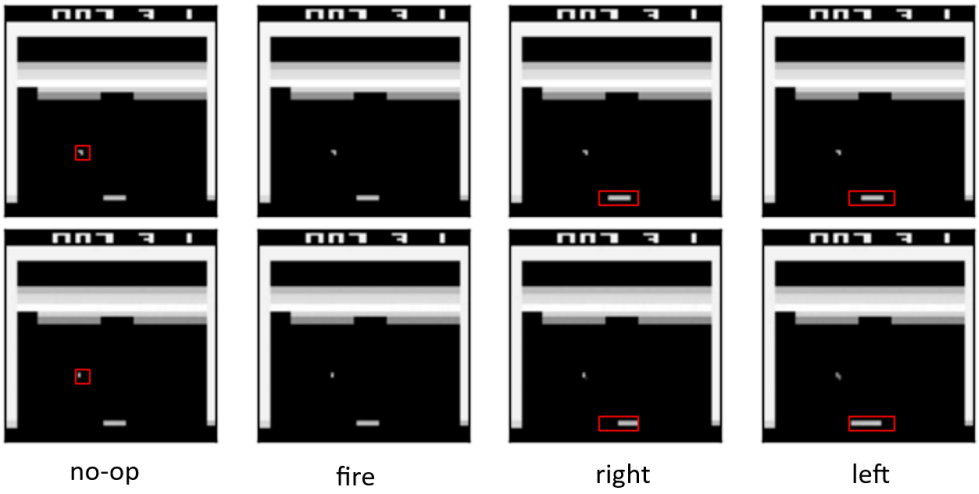


Figure 6.3: Predictions from the DDM in Breakout. The two rows represent two consecutive time steps where the first row is the original image and the second row is the next observation predicted by the DDM with all possible actions. The first row is the same image repeated four times to make the difference between the different actions easier to see. The red boxes are added manually to make it easier to see the differences between the simulated time step.

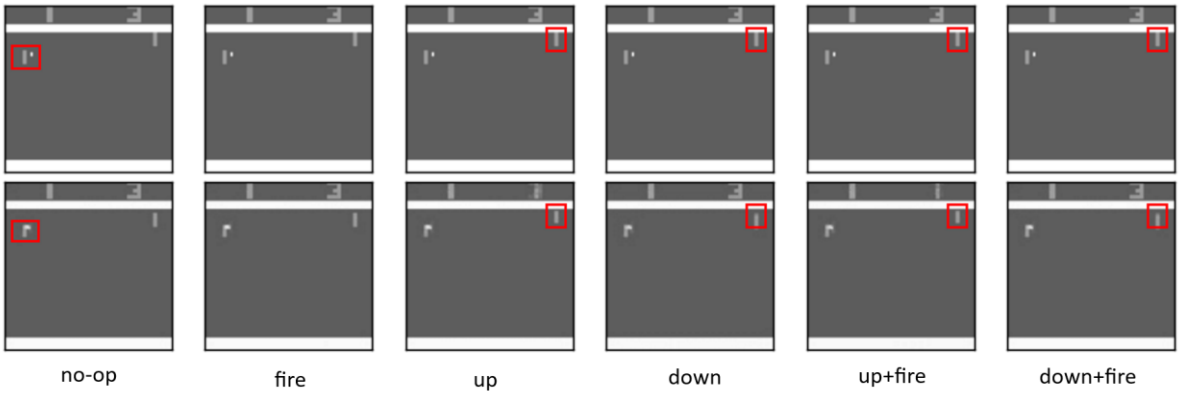


Figure 6.4: Predictions from the DDM in Pong. In this example the DDM did a good job of predicting the outcome of each transition. The top row is the same original image of the current time step and the bottom row is the predicted next observation for all actions.

Moving over to Pong, where the agent has two additional actions which makes it a bit harder for the DDM to understand how each action affects each state. In Figure 6.4 we see that the DDM manages to understand the dynamics of Pong in a precise way. The ball is predicted to hit the opponent's paddle (the leftmost paddle) and the agent's paddle is moved up and down based on the given action. It even understands that the paddle will not move further up because it has hit the top wall. However, an interesting observation is that this constraint

seems to modify the latent variable such that the score on top of the image is not precisely reconstructed.

Even though some of the most important dynamics were captured in Breakout and Pong, the DDM still had troubles with rare transitions. For instance, the fire action, which for both games should spawn the ball if the ball has gone out. This dynamic was never captured by the DDM and would have generated a critical impact if one were to simulate full episodes. Inaccuracies like this is what makes it hard to make a DDM that can predict many time steps into the future.

Another problem with the DDM was environments like Montezuma's Revenge and Q*Bert where each action does not perform the same transition every time. For example, if the agent is close to a wall in Montezuma's Revenge, the agent will not move through the wall. This happens a lot more often in Montezuma's Revenge than in Breakout and Pong. Another example is when the agent is airborne, then no action makes any difference to the transition. This creates another level of difficulty for the DDM to understand the dynamics.

A third factor for Montezuma's Revenge and Q*Bert is that the number of actions is increased to 18. An example of the DDM's predictions for the first six actions in Montezuma's Revenge can be seen in Figure 6.5. This confirms the hard challenge of making DDMs for high-dimensional environments. However, if we look close at the images, it is possible to see that the DDM has moved the agent one or two pixels upwards for the "up" action and similar distance downwards for the "down" action.

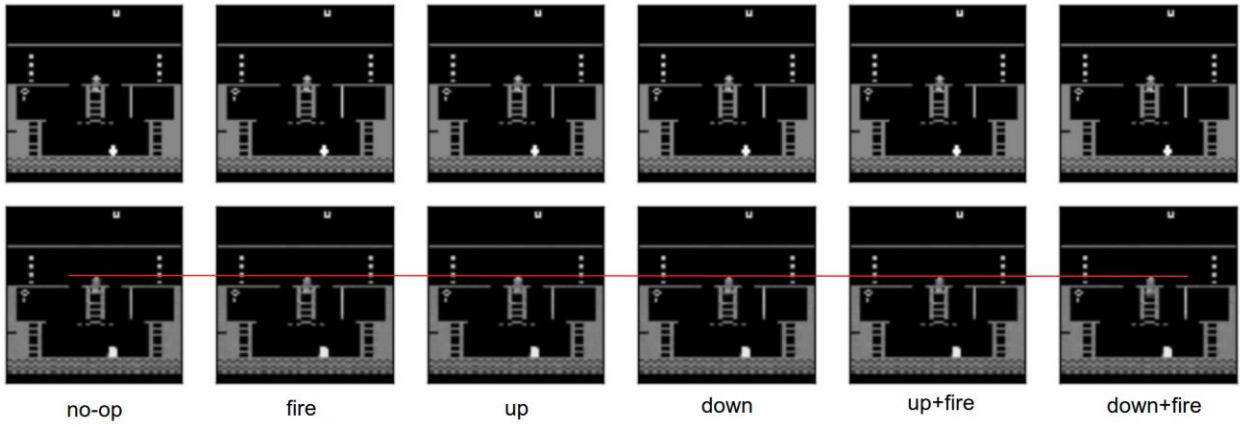


Figure 6.5: Predictions from the DDM in Montezuma's Revenge. The top row is the original frame from state s_t and the bottom row is a predicted frame for observation s_{t+1} based on the selected action (only the first six out of 18 actions are represented). The red line is added manually to make it easier to observe the changes to the agent placed on the ladder (might not be visible on paper).

6.4 Comparing Intrinsic Rewards

From Figure 6.6 we can see the results comparing each of the intrinsic motivation types to the model-free PAAC. The plots represent the moving average of the extrinsic reward (y-axis) during all the training time steps (x-axis). Looking at Pong, the plot Figure 6.6 (a), we see that all types of intrinsic motivation helped the agent at learning a good policy. Both, DL and DU, performs significantly better than the original PAAC.

Moving over to Breakout, Figure 6.6 (b), we see that very few of the intrinsic types manage to make the agent learn any faster. In fact, in many cases the intrinsic motivation made learning slower. This result is highly comparable to the one found in (Stadie et al. 2015) where intrinsic motivation was used in the same way as DL. When using a dynamic training schedule, like in this method, they achieved slight improvements in Pong, but performed much worse in Breakout. However, an interesting observation is our newly proposed intrinsic bonus DU, which performs much better in both environments. A pattern can be found for Seaquest, in Figure 6.6 (c). Again, DU seems to learn efficiently in a stable manner.

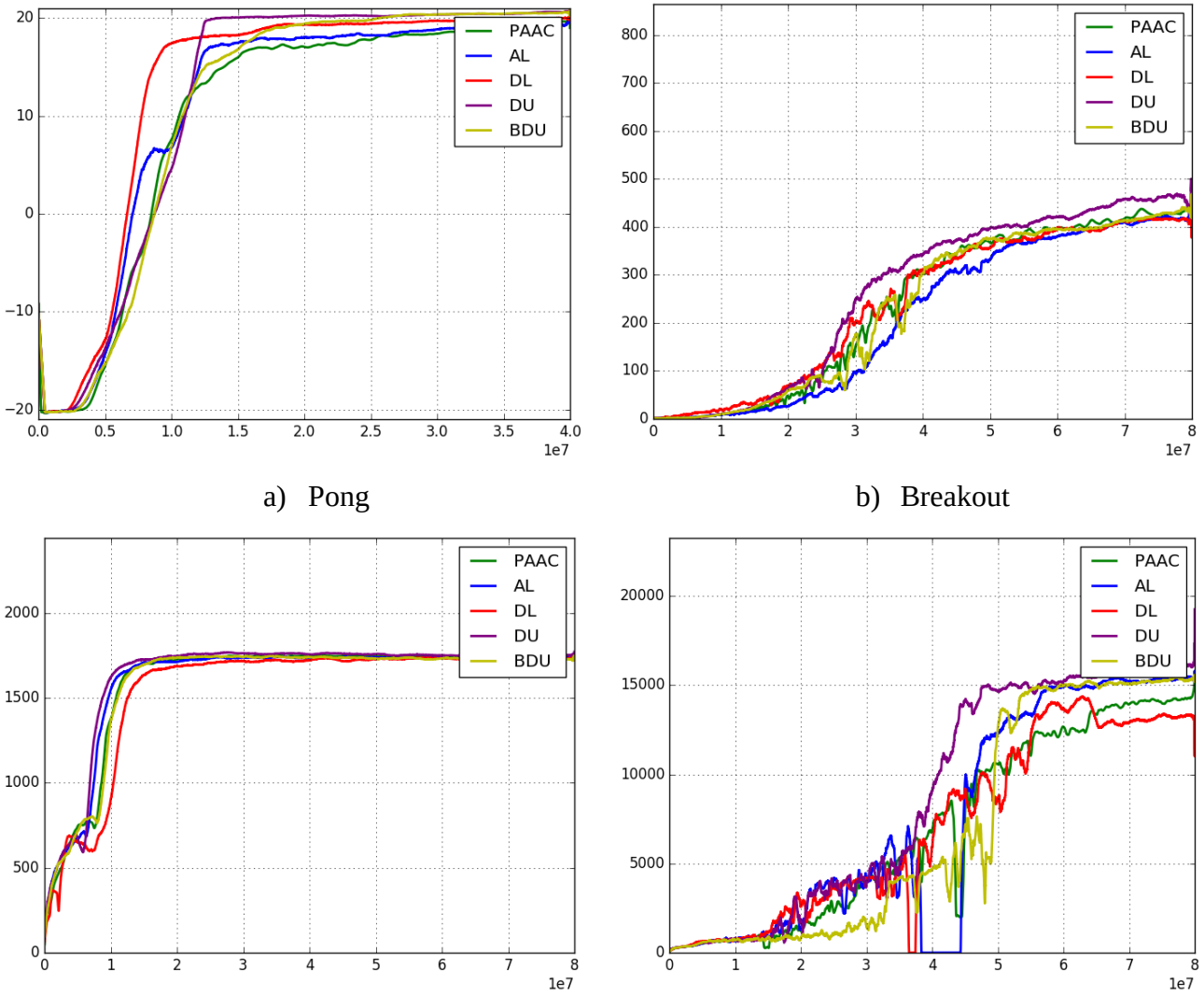
Looking at Q*Bert, Figure 6.6 (d), we see a greater spread in the results. In terms of stability during the training we see that the model-free PAAC and DU are stable all the way through the learning process. In the period from 35M to 50M time steps, when the increase in scores

are at their highest, every intrinsic motivation except DU experiences instabilities. With high instabilities during training, the agent will have worse data efficiency, and in the worst case it will stop learning and forget everything it has learned. Luckily, in these cases the agent got on the right track again. The same results were observed in (Clemente et al. 2017) where some of the environments cause instabilities during training. When comparing learning efficiency, the same pattern can be seen again. DU performs better than the other types of intrinsic rewards, and again, the results for DL match what was found in (Stadie et al. 2015), where intrinsic motivation sometimes made learning slower. Another interesting observation is that both AL and BDU perform better than the original PAAC.

Moving to the hardest environment of them all, Montezuma's Revenge, we unfortunately see the same results as in most of the other state-of-the-art methods. Neither of the intrinsic reward bonuses makes the agent explore efficiently enough. All agents manage to reach the key multiple times during training, but not often enough to learn a policy. Therefore, all rewards are basically zero through the whole training process. It seems like the DDM did not manage to learn the dynamics well enough to make the agent explore in an efficient manner. It is likely that with a more accurate dynamics model the intrinsic motivation could have guided the agent to more efficient exploration, thus, be able to learn a rewarding policy.

6.4.1 Data Efficiency

To look more closely at the data efficiency of each type of intrinsic reward we use the AUC score for the reward curves. These scores are normalized to the AUC score from the original PAAC and are displayed in Figure 6.7. Based on the scores we can see that the intrinsic reward bonuses do not improve the data efficiency for all types of environments, but rather improve in some scenarios and impairs the data efficiency in other cases. Looking at Table 6.1, we see that all intrinsic reward types boost the overall data efficiency. However, not all of them improve by a lot. If we look away from Montezuma's Revenge, where no agents managed to learn any decent policy, we see that the two most efficient types are DL and DU. The interesting part is that DU seems to be more stable and more effective than any of the other reward types, indicating that uncertainty from MC dropout might be a better choice for intrinsic motivation than just using the error from the dynamics model.



a) Pong b) Breakout

c) Seaquest d) Q*Bert

Figure 6.6: Plots of the reward curves for the different intrinsic rewards to the model-free PAAC rewards for the five selected environments. The graphs are labeled as Pong (a), Breakout (b), Seaquest (c) and Q*Bert (d).

Table 6.1: A comparison of the increase in data efficiency compared to the model-free PAAC. The scores are calculated by taking the AUC score for each intrinsic reward type and normalize it on the model-free PAAC AUC score.

	AL	DL	DU	BDU
All 5 games	1.021	1.032	1.111	1.034
All, except Montezuma's Revenge	1.018	1.074	1.14	1.028

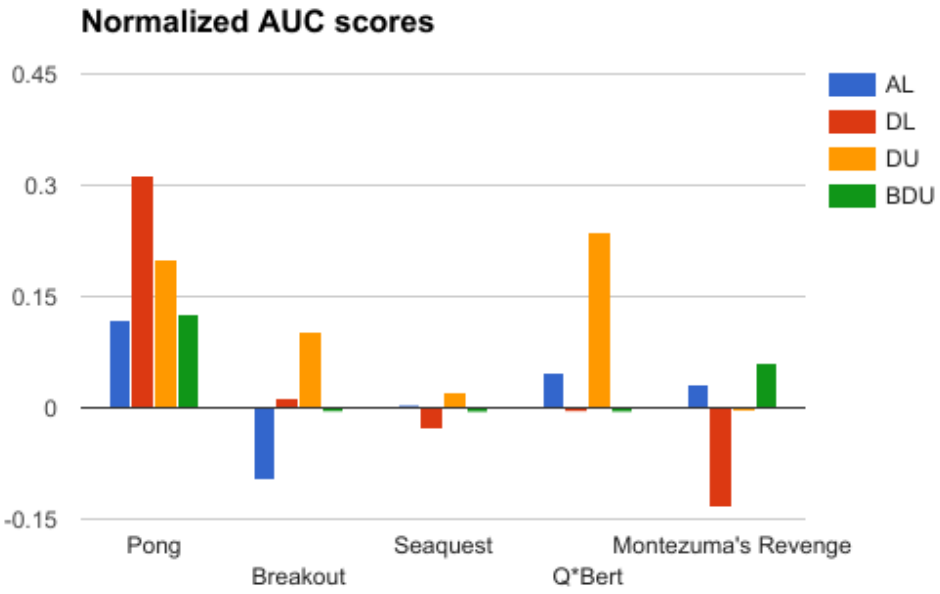


Figure 6.7: AUC scores for the average over two agents per category. The AUC scores are calculated by approximating the integral with the trapezoidal rule. Finally, they are normalized on the AUC score from the model-free PAAC, making everything above zero more data efficient.

An observation that seems to contradict that the uncertainty from MC dropout is a better choice than the direct loss is the performance of BDU. Based on the smaller experiments discussed in Appendix C, one should expect better uncertainty estimates from a combination of bootstrap and MC dropout compared to just using MC dropout alone. However, we see just a very small increase in data efficiency compared to the original PAAC, and in fact, much worse data efficiency when compared to DU. One of the main reasons for this could be the training of the DDM when the TPM architecture contains multiple heads. Looking at Figure 6.8 it becomes clear that adding multiple heads to the TPM makes the training process much more unstable. It also takes much longer time to converge, as confirmed by the smaller experiments discussed in Appendix C. Even after 80M time steps, the bootstrapped TPM has not converged to the same level as when not using multiple heads. These instabilities might cause horribly wrong uncertainty measures that only will distract the agent instead of motivating it towards novel transitions.

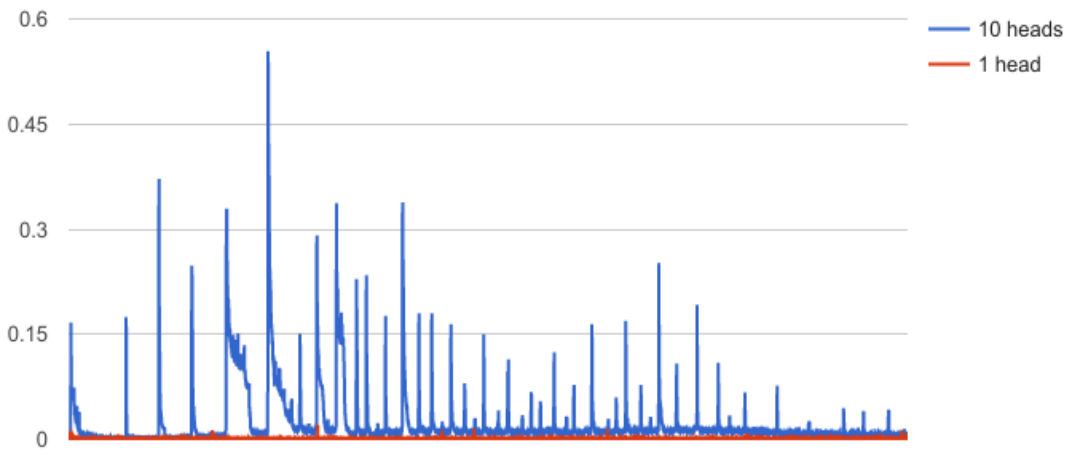


Figure 6.8: Training TPM with and without bootstrapping. Training with multiple heads were performed by selecting one of the heads randomly during training. The data was gathered over 80M time steps in Breakout.

6.4.2 Final Performance

We see the same pattern in the final performance after 80M time steps (40M time steps for Pong). This is shown in Table 6.2. DU is not only more data efficient than the original PAAC, but also manages to outperform all the other methods in all environments. Another observation is that AL and DL in fact generate premature convergence in multiple environments. This is also shown in (Gal & Ghahramani 2015) where their agents learn fast in the beginning, but are affected by premature convergence.

Table 6.2: Final scores comparing the model-free PAAC to the different kinds of intrinsic bonuses from MB-PAAC. The results are from the best out of two trained agents for each category when measured as an average over 30 episodes. The overall best results for each game are marked in bold. Random scores are fetched from (Mnih et al. 2015).

	Random	PAAC	AL	DL	DU	BDU
Pong	-20.7	20.4	19.7	20.0	**20.6**	20.5
Breakout	1.7	464.4	422.1	457.5	**473.4**	453.7
Seaquest	68.4	1756.0	1743.2	1733.5	**1758.7**	1741.3
Q*Bert	163.9	14970.0	15380.7	12578.2	**16580.6**	15279.2
Montezuma's Revenge	0.0	0.0	0.0	0.0	0.0	0.0

6.4.3 Learning without extrinsic rewards

Based on the results that intrinsic reward bonuses help exploration, one could assume that by only feeding the agent intrinsic rewards it would be able to learn some policy that is better than random. To test this hypothesis, we trained agents with the two best resulting intrinsic reward bonuses, only removing the extrinsic rewards received from the environments. The results are presented in Figure 6.9. As expected, the intrinsic rewards alone are not enough to achieve state-of-the-art performance in the five tested Atari games. However, we see that the agents learn policies much better than random even though no rewards are received from the environment[6]. This encourage the idea that using surprise as an inner motivation is a good thing for exploration.

6.5 Computational Performance

This section compares how much the introduction of a big dynamics model slows down the efficient PAAC algorithm. As expected, Figure 6.10 show that the introduction of a DDM will slow down training time of the agents by a fair amount. This is expected because the original PAAC algorithm only operates with a very small convolutional network, compared to the MB-PAAC architecture with the additional AE and the associated TPM. The performance experiments were performed on a computer with CPU i7 4790k and GPU Nvidia GTX 980.

The MB-PAAC with intrinsic motivation is in average 4.16 times slower than the model-free PAAC. This indicates that the introduction of the approximated dynamics model might not be worth it for the small performance and data efficiency gain. However, when looking at the utilization of the GPU resources, displayed in Figure 6.11, it is possible that one might get a great performance increase with a stronger GPU for the MB-PAAC architecture. The PAAC algorithm is bound by the bandwidth of transferring data from the CPU to the GPU, while the MB-PAAC is more bound by pure GPU computational power.

[6] A video of agents trained without any extrinsic rewards playing different Atari games can be found under https://www.youtube.com/watch?v=MKR8iH5gqW4&t.

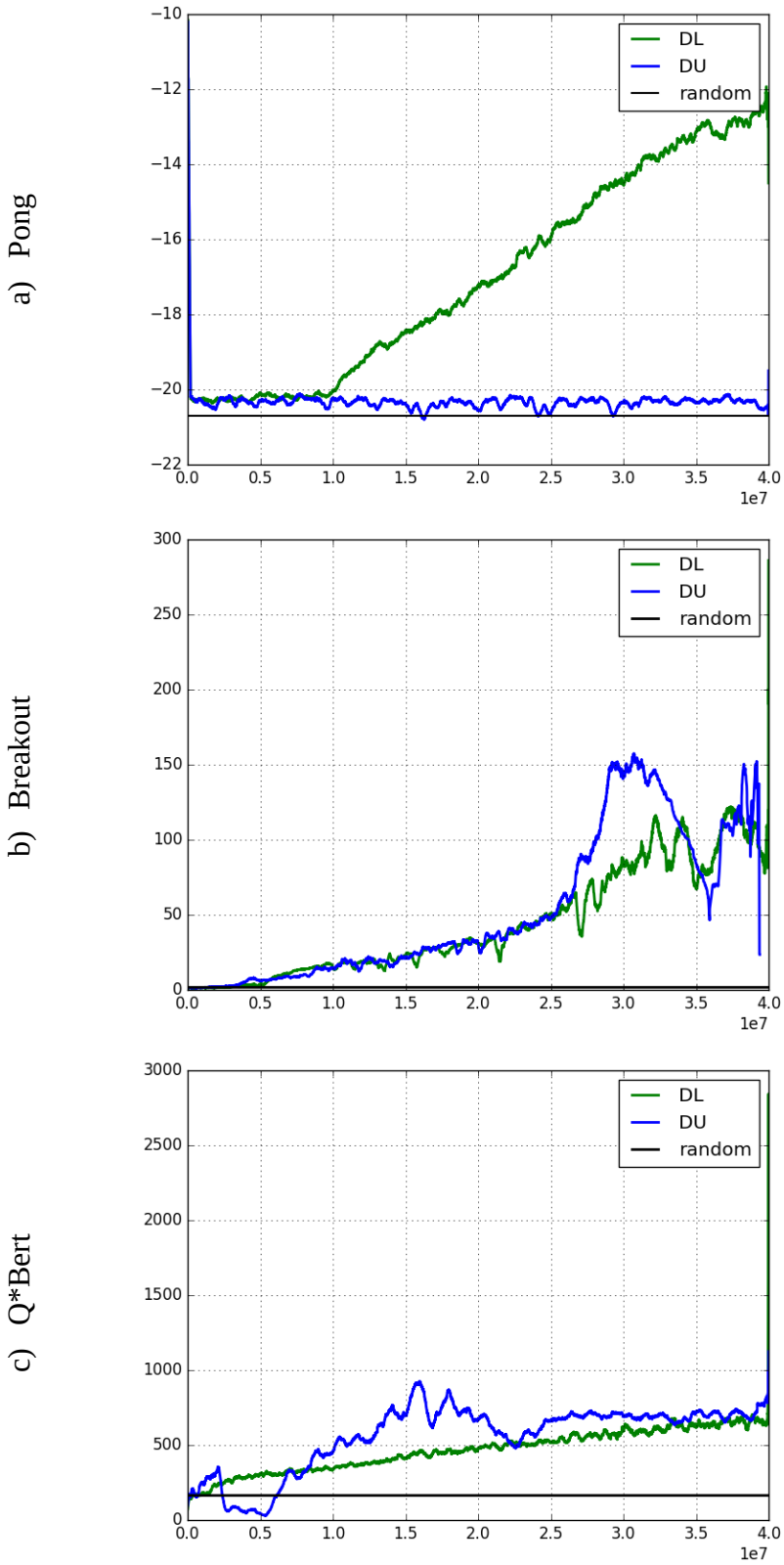


Figure 6.9: Learning by intrinsic rewards only in the Atari domain. The plots compare the intrinsic motivations; DL and DU. The x-axis represents time steps and y-axis represents episodic extrinsic rewards during training.

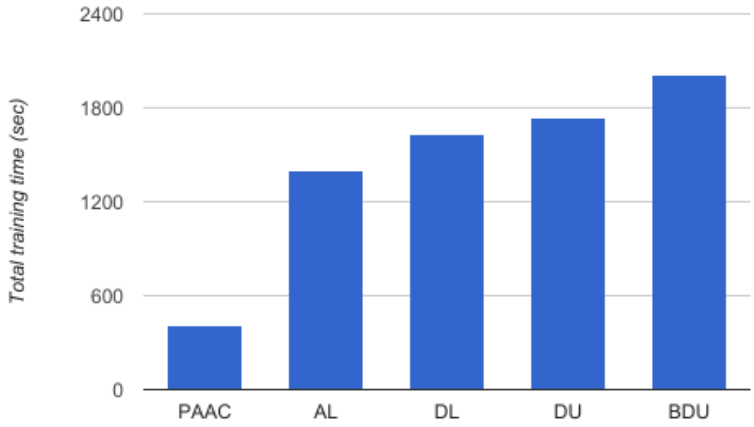


Figure 6.10: Training time for the five different settings, where PAAC represents the model-free baseline, and the four others are MB-PAAC with different intrinsic motivations. The results are from agents trained for 1M time steps in Breakout.

Comparing the computational efficiency between the different intrinsic reward bonuses we can see that AL is the most efficient one. This is as expected because the AL only need to feed data through the AE, hence, the TPM is never used. The three other intrinsic motivations use the TPM, but there are still some differences between them. For instance, the novel DU technique is 6.4% slower than the more common DL technique. However, DU scales worse than DL when the number of actions increase. The slowest of them all, again as expected, is the BDU technique which is the same as DU, only the TPM contains multiple output heads.

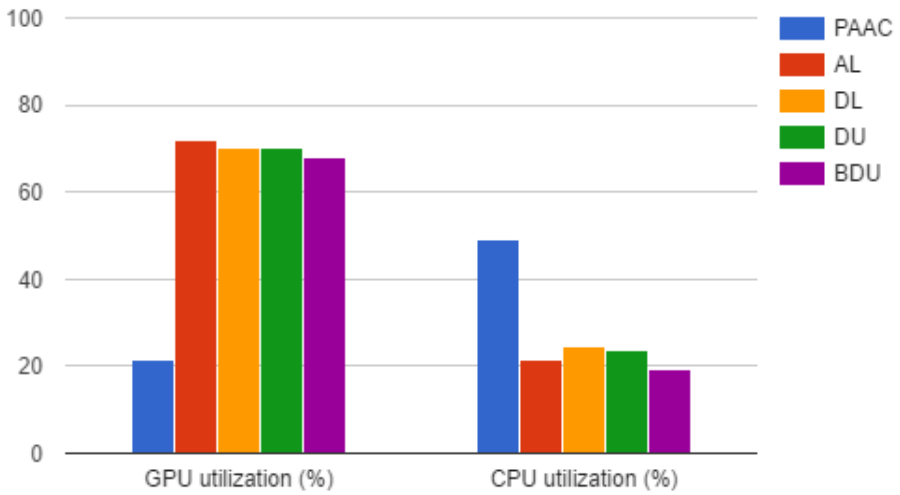


Figure 6.11: GPU and CPU utilization during training in Breakout for the five different settings. PAAC represents the model-free algorithm, while the four others are MB-PAAC with different intrinsic motivation.

6.6 Discussion of Solution

Even though creating an approximated dynamics model of high-dimensional environments is considered a hard problem, the proposed DDM manages to learn decent representations for most of the tested Atari games. Comparing the DDM to the current state-of-the-art it did not manage to learn as precise transitions as in (Oh et al. 2015). This might be because the architecture described in the paper was significantly bigger and slower. When looking at the different intrinsic reward bonuses we can see that all of them increased data efficiency on average of the five environments. Comparing the results to (Stadie et al. 2015) we see a matching pattern for the DL bonus.

The proposed system did not manage to learn any decent policy in the sparsely rewarded Montezuma's Revenge. Compared to (Bellemare et al. 2016) this implies that the intrinsic rewards based on an inaccurate DDM will not give good enough uncertainty measures to guide the agent towards novel states. When looking at how much an unstable DDM affected the performance of the BDU reward bonus, a more precise DDM might be enough to make an agent learn to achieve scores in Montezuma's Revenge.

7 Conclusion and Future Work

Section 7.1 will conclude the research, followed by a description of the main contributions in 7.2. Finally, we propose some thoughts on the possibilities for future work in section 7.3.

7.1 Conclusion

The study has shown that deep dynamics models (DDM) could be used to generate different kinds of intrinsic reward bonuses that can be used to improve exploration and data efficiency in deep reinforcement learning (RL). The proposed DDM was inspired by the state-of-the-art architectures described in chapter 0 where the state space is compressed into a latent space which then can be used to approximate the environment dynamics. The DDM consists of two parts; an autoencoder (AE) compressing the high-dimensional states into a latent space, and a Transition Prediction Model (TPM) predicting the dynamics in the generated latent space. The TPM takes in the latent representation for both the previous state and the current state together with the selected action to predict the latent variable for the next state. The DDM were used to extract different intrinsic reward bonuses for guiding the agent towards novel state transitions, thus, achieving more efficient exploration.

Together with the DDM, this study has presented a comparison of four different ways of computing intrinsic reward bonuses. The comparison included already existing techniques, such as using the prediction error from the DDM, to our proposed novel technique extracting the uncertainty of the DDM by MC dropout. The four bonus types compared are; autoencoder reconstruction error, dynamics prediction error, uncertainty by MC dropout and uncertainty by MC dropout with a bootstrapped TPM. All tested types of intrinsic reward bonus improved data efficiency compared to a model-free reinforcement architecture. However, the most efficient technique, outperforming the common way of using prediction error, was our newly proposed uncertainty measure extracted from the TPM by MC dropout. By using this new intrinsic motivation, the agent outperformed the model-free agent in four out of five tested Atari games, matching the performance in the last one. However, being more data efficient and ending with a better final policy comes at the computational cost of having to train a DDM. We also verified the results by showing that the agents can learn decent action policies without receiving any extrinsic rewards from the environment at all.

Looking back at the research questions posed in section 1.2, we can see that research question 1 is answered in chapter 0 where an overview of the current state-of-the-art is described. Research question 2 is answered by the techniques, experiments and results discussed in chapters 4, 5 and 6. All in all, we can conclude that intrinsic reward bonuses can increase both data efficiency and overall final performance of RL agents. It can even help agents learn without any extrinsic rewards. In light of these achievements, we can confirm that the task of creating accurate deep dynamics models in high-dimensional environments is a hard, but promising direction for more research.

7.2 Contribution

The main contribution of this study is a performance comparison of different types of intrinsic motivation extracted from an approximated dynamics model in the high-dimensional Atari 2600 domain. Ranging from the commonly used dynamics prediction error to a more novel way of using MC dropout to extract the uncertainty from the dynamics model. We have found that the more accurate uncertainty estimates from MC dropout can outperform the classical prediction error bonus. Secondly, we proposed a deep dynamics model inspired by many of the current state-of-the-art model-based architectures. The architecture of using uncertainty extracted from a deep dynamics model has shown to improve data efficiency even though the dynamics model is inaccurate.

7.3 Future Work

This last section provides an insight for possible future improvements to the work done in this study. The proposals will cover architectural changes for the current DDM, different combinations of intrinsic rewards, and more long-term possibilities with the model-based architecture together with intrinsic motivation.

7.3.1 Improvements to the DDM Architecture

The current DDM models time sequence by feeding the current observation concatenated with a past observation. As explained in chapter 0, the work in (Oh et al. 2015) propose two ways of modelling the time sequence. The first one uses a similar concatenating technique as used in the TPM, but the second architecture, which they call recurrent encoding, utilizes a recurrent layer for capturing the sequences. This idea of using a recurrent layer, for instance a

LSTM layer in the TPM, could be an interesting start for improving the current model. Even though recurrent layers are more computationally heavy, they have the capability of capturing longer time sequences. The introduction of a recurrent layer could be an improvement to the dynamics model, however, from the results described in (Oh et al. 2015) there is no significant difference between the recurrent architecture and the feedforward architecture. However, the use of a LSTM in the model-free A3C architecture from (Mnih et al. 2016) proved to give better results. For A3C, the regular feedforward version achieves 496.8% of human score in the Atari domain and the recurrent version achieves 623.0%.

Another feature that might increase the accuracy of the proposed DDM could be to upgrade the experience replay into a prioritized experienced replay, similar to the buffer in (Schaul et al. 2015). Prioritized experience replay is an experience buffer where the training samples are not sampled uniformly, but rather by how surprising the sample is for the model. By introducing this to the already existing experience replay, we can expect a potentially faster learning DDM with better accuracy. By having a DDM learn faster we can lower the training time for the model, thus getting better computational performance. The same principle could be used for creating an even smarter dynamical training schedule for the DDM. For example, by measuring the average decay in loss, we can select to train the DDM more often or rarer based on how well it performs. This can affect environments that change as the agent progresses through learning. For example, when the agent in Montezuma's Revenge suddenly manages to open the door from room one to room two, and much of what it knows must be relearned because of the differences between the rooms.

Currently, the proposed DDM only predicts one time step at the time, making planning for multiple time steps in the future infeasible. One way to overcome this problem is to change the learning schedule into predicting n steps forward in time, with a curriculum of gradually increasing n. This is what was done in (Oh et al. 2015) to make accurate predictions over 250 time steps into the future. However, the downside to this method is that it is policy dependent, which means that if the action policy changes the predictions will be terrible. This means that training the DDM simultaneously as training the policy may need a consideration of techniques like TRPO from (Schulman et al. 2015), where policy updates are constrained.

7.3.2 A New DDM Architecture

The very recently proposed DDM architecture from (Pathak et al. 2017) achieves remarkable results by using prediction error as intrinsic motivation in high-dimensional environments. Their new architecture is called a self-supervised inverse dynamics model. The architecture makes the model learn to predict the action taken in the transition between the previous state to the current state. This is called the inverse model. Further, they use the embedding generated by learning to predict the actions, to predict the embedding for the next observation. This is called the forward model. This technique is very similar to how the TPM works with the AE. However, it is different from regular models directly predicting the pixels for the next observation based on the selected action. See Figure 7.1 for an illustration of the full architecture.

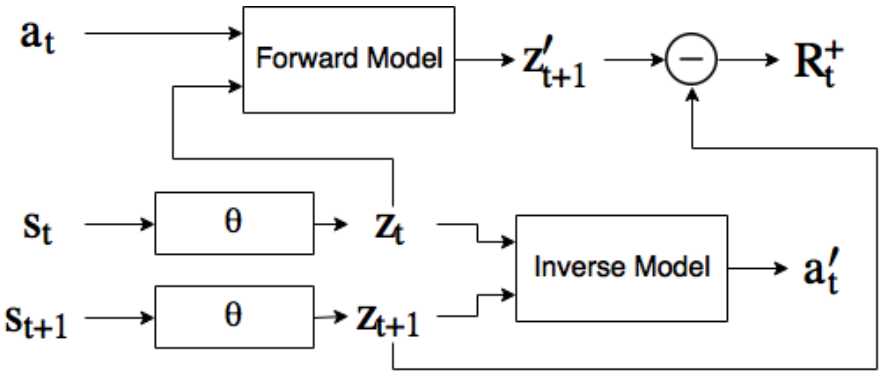


Figure 7.1: The architecture of the inverse model proposed in (Pathak et al. 2017) consisting of two main models; the forward model and the inverse model. The inverse model learns to predict the action taken in the current state s_t resulting in the next state s_{t+1}. This model generates a latent space that gives the embedding z_t and z_{t+1} for s_t and s_{t+1} respectively. The embedding z_t is then concatenated with the selected action a_t and fed into the forward model where the embedding for the next state is predicted. The intrinsic reward bonus R_t^+ is calculated by the difference between the prediction z'_{t+1} and the actual latent variable z_{t+1}.

This change in architecture avoids the many challenges of predicting next observations as pixels. It also benefits from having only the events affecting the agent embedded in the latent space, which in the end is what you want from intrinsic motivation. The authors give an explanatory example for this problem: *"Imagine a scenario where the agent is observing the movement of tree leaves in a breeze. Since it is inherently hard to model breeze, it is even harder to predict the pixel location of each leaf."* (Pathak et al. 2017).

The idea is that the if the agent would have used a model based on pixels, the agent would always have been curious about the leaves because of the random movements. The other point is that for another scenario where the leaves does not matter, but they appear in the observation, they will attract focus away from the important changes in the environment because of their “random” movements. If one were to change the DDM architecture in a way that give more accurate intrinsic motivation, this would probably be the way to go.

7.3.3 Combining Multiple Intrinsic Motivations

This study compare different kinds of intrinsic motivation for guiding the agent with curiosity as an analogy to human thinking. An interesting future research would be to combine multiple types of intrinsic motivation. For instance, by combining DL and DU one could motivate the agent in slightly different ways and together create a more accurate curiosity measure. Taking this a step further, one could use a function for dynamically adjust for what kind of bonus to use for different observations. However, both measures come from the same approximated dynamics model which gives a limited uncertainty measure. If one were to start combining intrinsic motivations one could use the uncertainty from the policy (and/or value) network, as proposed in (Gal & Ghahramani 2015), and join it with the uncertainty extracted from the DDM. This idea is somewhat comparable to the idea behind bootstrap and the use of multiple heads.

7.3.4 Long-term Views

Even though the main contribution and focus has been with intrinsic motivation from a DDM, the research has an underlying goal of driving model-based RL techniques further in high-dimensional environments. Model-based techniques have a great potential when it comes to data efficiency, which might be a necessary step for making RL fully available to real world problems. Effective exploration is only one part of the potentially greater data efficiency with model-based agents. With more accurate models, one could use planning multiple steps ahead, or indeed, use the model to simulate experiences. This could bring us closer to the ultimate goal of making intelligent agents utilize skills they have learned, to solve problems in the real world.

Bibliography

Achiam, J. & Sastry, S., 2017. Surprise-Based Intrinsic Motivation for Deep Reinforcement Learning. , pp.1–14.

Anderson, C.W., Lee, M. & Elliott, D.L., 2015. Faster reinforcement learning after pretraining deep networks to predict state dynamics. *Proceedings of the International Joint Conference on Neural Networks*, 2015–Septe.

Bellemare, M.G. et al., 2015. The arcade learning environment: An evaluation platform for general agents. *IJCAI International Joint Conference on Artificial Intelligence*, 2015–Janua, pp.4148–4152.

Bellemare, M.G. et al., 2016. Unifying Count-Based Exploration and Intrinsic Motivation. *arXiv*, (Im), pp.1–26. Available at: http://arxiv.org/abs/1606.01868 [Accessed August 31, 2016].

Blundell, C. et al., 2015. Weight Uncertainty in Neural Networks. *Icml*, 37, pp.1613–1622. Available at: http://arxiv.org/abs/1505.05424%5Cnhttp://www.arxiv.org/pdf/1505.05424.pdf.

Brafman, R.I. & Tennenholtz, M., 2001. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *IJCAI International Joint Conference on Artificial Intelligence*, 3, pp.953–958.

Cai, H., Ren, K. & Zhang, W., 2017. Real-Time Bidding with Reinforcement Learning in Display Advertising. , pp.1–16.

Campbell, M., Hoane Jr., a. J. & Hsu, F., 2002. Deep Blue. *Artificial Intelligence*, 134(1–2), pp.57–83. Available at: http://www.sciencedirect.com/science/article/pii/S0004370201001291.

Clemente, A. V, Chandra, A. & Humberto, C.N., 2017. Efficient Parallel Methods for Deep Reinforcement Learning. , p.9. Available at: https://arxiv.org/pdf/1705.04862v1.pdf.

Deisenroth, M.P. & Rasmussen, C.E., 2011. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. *Proceedings of the International Conference on Machine*

Learning, pp.465–472.

Efron, B., 1982. 5. The Bootstrap. In *The Jackknife, the Bootstrap and Other Resampling Plans*. pp. 27–36. Available at: http://epubs.siam.org/doi/abs/10.1137/1.9781611970319.ch5.

Gal, Y. & Ghahramani, Z., 2015. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. , 48. Available at: http://arxiv.org/abs/1506.02142.

Gal, Y., McAllister, R. & Rasmussen, C.E., 2016. Improving PILCO with Bayesian Neural Network Dynamics Models. In *Data-Efficient Machine Learning workshop, ICML*.

Hinton, G.E. et al., 2012. Improving neural networks by preventing co-adaptation of feature detectors. , pp.1–18. Available at: http://arxiv.org/abs/1207.0580.

Hochreiter, S. & Schmidhuber, J., 1997. Long Short-Term Memory. *Neural Computation*, 9(8), pp.1735–1780. Available at: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735.

Houthooft, R. et al., 2016. Variational Information Maximizing Exploration. Available at: http://arxiv.org/abs/1605.09674 [Accessed August 31, 2016].

Kahn, G. et al., 2017. Uncertainty-Aware Reinforcement Learning for Collision Avoidance. Available at: http://arxiv.org/abs/1702.01182.

Kearns, M., 2002. Near-Optimal Reinforcement Learning. , pp.209–232.

Li, J. et al., 2016. Deep Reinforcement Learning for Dialogue Generation. *arXiv*, 2(2), pp.1192–1202. Available at: http://arxiv.org/abs/1606.01541.

Masci, J. et al., 2011. Stacked convolutional auto-encoders for hierarchical feature extraction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6791 LNCS(PART 1), pp.52–59.

Mnih, V. et al., 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv*, 48, pp.1–28. Available at: http://arxiv.org/abs/1602.01783.

Mnih, V. et al., 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529–533. Available at: http://dx.doi.org/10.1038/nature14236.

Mnih, V. et al., 2013. Playing Atari with Deep Reinforcement Learning. , pp.1–9. Available at: http://arxiv.org/abs/1312.5602.

Ng, A.Y. et al., 2006. Autonomous inverted helicopter flight via reinforcement earning. *Springer Tracts in Advanced Robotics*, 21, pp.363–372.

Oh, J. et al., 2015. Action-Conditional Video Prediction using Deep Networks in Atari Games. *Nips*, p.9. Available at: http://arxiv.org/abs/1507.08750.

Osband, I. et al., 2016. Deep Exploration via Bootstrapped DQN. *arXiv*, 1602.04621, pp.1–18. Available at: http://arxiv.org/abs/1602.04621.

Ostrovski, G. et al., 2016. Count-Based Exploration with Neural Density Models.

Pathak, D. et al., 2017. Curiosity-driven Exploration by Self-supervised Prediction. Available at: http://arxiv.org/abs/1705.05363.

Raju, C.V.L., Narahari, Y. & Ravikumar, K., 2003. Reinforcement learning applications in dynamic pricing of retail markets. *E-Commerce, 2003. CEC …*. Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1210269.

Sandven, T., 2016. Visual Pretraining for Deep Q-Learning. , (June).

Schaul, T. et al., 2015. Prioritized Experience Replay. *arXiv*, pp.1–23. Available at: http://arxiv.org/abs/1511.05952.

Schulman, J. et al., 2015. Trust Region Policy Optimization. Available at: http://arxiv.org/abs/1502.05477.

Shalev-Shwartz, S. et al., 2016. Long-term Planning by Short-term Prediction. , pp.1–7. Available at: http://arxiv.org/abs/1602.01580.

Shani, G., Brafman, R.I. & Heckerman, D., 2012. An MDP-based Recommender System. *J. Mach. Learn. Res.*, 6, pp.1265–1295. Available at: http://arxiv.org/abs/1301.0600.

Silver, D. et al., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), pp.484–489. Available at: http://dx.doi.org/10.1038/nature16961.

Stadie, B.C., Levine, S. & Abbeel, P., 2015. Incentivizing Exploration In Reinforcement

Learning With Deep Predictive Models. Available at: http://arxiv.org/abs/1507.00814 [Accessed August 31, 2016].

Wahlström, N., Schön, T.B. & Deisenroth, M.P., 2015. From Pixels to Torques: Policy Learning with Deep Dynamical Models. *arXiv preprint arXiv:1502.02251*, (1), p.9. Available at: http://arxiv.org/abs/1502.02251.

Watter, M. et al., 2015. Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images. *arXiv*, pp.1–18. Available at: http://arxiv.org/abs/1506.07365.

Appendix A: Atari Environments

a) Pong | b) Breakout | c) Seaquest | d) Q*Bert | e) Montezuma's Revenge

Figure A.1: Images from two different time steps for the five environments. The first row being the original 210x160 RGB image and the second row being the preprocessed 84x84 grayscale image that is observed by the agent.

Pong

Pong is considered to be one of the easiest environments from the Atari domain. The goal of Pong is to move a paddle (right side of the game screen) up and down to hit a ball over to the other side such that the opponent's paddle does not reflect it. A positive point is received for each time the ball passes the opponent's paddle, and a negative point each time it passes the agent's paddle. The point equals the received reward for each time step. After 21 points is added or subtracted, the episode ends. Having only six actions in the action space makes it easier than the environments containing all 18 actions. Another property that makes Pong easy to learn is that the observations contain only important objects to solve the game. There are few details and no unnecessary moving parts. Also, because the environment stays the same all the way through the learning process makes it easy to optimize. Each transition is deterministic which makes it easier for the DDM to learn an approximation. An example observation can be seen in Figure A.1 (a). Pong was selected as an easy baseline to start with, where having a fast learning agent aids experimentation.

Breakout

Breakout is similar to Pong, but it contains some differences that makes it a bit harder to learn a decent policy. The goal in Breakout is to control a paddle (bottom of the screen) to reflect a ball at the bricks in the top of the screen. Points are rewarded when the ball hits and breaks the bricks. The environment is finished after the ball has passed the paddle five times. There are two main differences to Pong. The first one is that the agent can only choose between four possible actions instead of six. This property make Breakout easier than Pong. The second one is that the observations change as the agent gets further into the learning process. This is because as more bricks are destroyed, the ball can suddenly move through where there used to be a brick. This comes natural to us humans, but to a RL agent it is an important factor. The agent must learn what are walls and what are destroyable bricks. An example observation can be seen in Figure A.1 (b). Breakout was selected to be an easy environment containing changing factors as the agents learn.

Seaquest

Seaquest is a submarine game where the agent controls a submarine underwater. The goal is to shoot or avoid various enemies and pick up divers swimming around. Another critical aspect of Seaquest is the oxygen bar that represents how much oxygen is left in the submarine. To refill oxygen, the submarine must resurface. This is one of the challenging problems for this environment. The rewards are received for killing enemies and picking up divers. Because the enemies and divers spawn at random locations at either side of the screen, many of the transitions are stochastic. A screenshot from Seaquest can be seen in Figure A.1 (c). Seaquest was selected because it requires a bit more exploration than Breakout and Pong, but is still an easy game to achieve a decent policy. Additionally, it introduces stochastic transitions.

Q*Bert

In Q*Bert, a jumping agent is controlled in a stationary platform environment. The goal is to make the agent visit all the platforms without dying to any of the enemies that randomly spawns. A reward is received each time the agent visits a new platform or when a randomly spawned berry is picked up. When all platforms are visited, the agent gets a new set of platforms to visit, only this time, the platforms have changed color. It is the last point that makes Q*Bert a bit harder than, for instance, Seaquest. An observation from Q*Bert can be

seen in Figure A.1 (d). Q*Bert was chosen because of its dynamically changing nature (the colors on the platforms) and some stochastic transitions (spawning enemies and berries).

Montezuma's Revenge

The goal in Montezuma's Revenge is to control an agent in a platform environment with multiple rooms. Each room contain keys, locked doors and enemies. The agent receives rewards for picking up keys, opening doors and collecting diamonds. Montezuma's Revenge is one of the hardest environments of all Atari environments. With extremely sparse rewards and many ways to end the episode, it is notoriously hard for an RL agent to explore and find a decent policy. The observations do also change a lot as the agent progresses through the different rooms. After the first key is picked up and the first door is opened, a whole new scene is presented to the agent. This makes it hard for the agent to learn one hardcoded policy that works for one scenario. To succeed in Montezuma's Revenge, the agent would have to perform smart exploration by not visiting states that are already known more than a couple of times. Another property of Montezuma's Revenge is that all 18 actions are used. All these features combined gives a huge state-action space to explore with few rewarding signals received. One thing that could have made Montezuma's Revenge even harder is if it contained some stochastic transitions which is does not. An example observation from the first scene can be seen in Figure A.1 (e). Montezuma's Revenge was selected because of being notoriously hard, and because smart exploration is required to receive any significant amount of rewards to guide learning.

Appendix B: Hyperparameters for PAAC

Table B.1: Hyperparameters used for the model-free PAAC runs. These parameters were also used for the MB-PAAC runs.

Parameter (argument)	Value	Description
$\epsilon_{Optimizer}$ (-e)	0.1	Epsilon parameter for the RMSProp and Adam optimizers in Tensorflow
$\alpha_{Optimizer}$ (--alpha)	0.99	Discount factor for previous history in RMSProp and Adam optimizers in Tensorflow
Learning rate (-lr)	0.0224	The initial learning rate for the optimizers
Entropy (--entropy)	0.01	For the entropy regularization in the policy update
Gradient clipping (--clip_norm)	40.0	Calculated gradients will be clipped at this value
γ (--gamma)	0.99	Discount factor for the RL update
Emulator count (-ec)	32	The number of emulators ran in parallel
Emulator workers (-ew)	8	The number of workers ran in parallel per agent
Network architecture (--arch)	NIPS	The network architecture used. It consists of two convolutional layers followed by a fully connected layer.

Appendix C: Experiments with Bootstrap and MC Dropout

This experiment tests two different ways of extracting uncertainty from deep neural networks. The first one being MC dropout which work by feeding the same input multiple times through the network with stochastic dropout masks. The set of outputs generates a variance which can be used to calculate the uncertainty of the neural network. The second one is bootstrap which modifies the network to have multiple heads. The different heads are trained on different parts of the dataset. This makes each head capture different aspects of the dataset, thus the variance in output from all the heads can be used to calculate the uncertainty of the network. The experiment perform regression on a nonlinear function given in Appendix A in the paper (Osband et al. 2016). The function is written as,

$$y_i = x_i + sin(\alpha(x_i + w_i)) + sin(\beta(x_i + w_i)) + w_i$$

where x_i is drawn uniformly from $(0, 0.6)U(0.8, 1)$ and $w_i \sim N(\mu = 0, \sigma^2 = 0.03^2)$.

A fully connected deep neural network with two hidden layers of 60 nodes and a dropout rate of 0.03 is used. The dropout rate is low because of the small network size. In all experiments $T = 100$ stochastic forward passes were used for MC dropout. The first experiment tests how Tanh and ReLU acts in MC dropout. From Figure C.1 we clearly see that the uncertainty generated by Tanh gives a bad representation compared to ReLU which steadily increases as the predictions get further away from the dataset. This might be because Tanh is limited to the range of [-1, 1]. The second experiment compares how different numbers of heads in bootstrap affects the uncertainty. From Figure C.2 we see that four heads give a better uncertainty measure than using two heads. In the last experiment, we combine four headed bootstrap and MC dropout which gives the best uncertainty of all methods. This is shown in Figure C.3. The reason for this might be that bootstrap and MC dropout represent different solutions in the solution space for the observed data. The bootstrap starts off at different points in the weight space and gradually moves towards the final solution. The MC dropout represent an approximation over different functions of the currently observed data. Visually it would look something like what is presented in Figure C.4. However, this comes at the cost of being computationally heavy. Comparing four headed bootstrap to MC dropout computationally wise, MC dropout is faster.

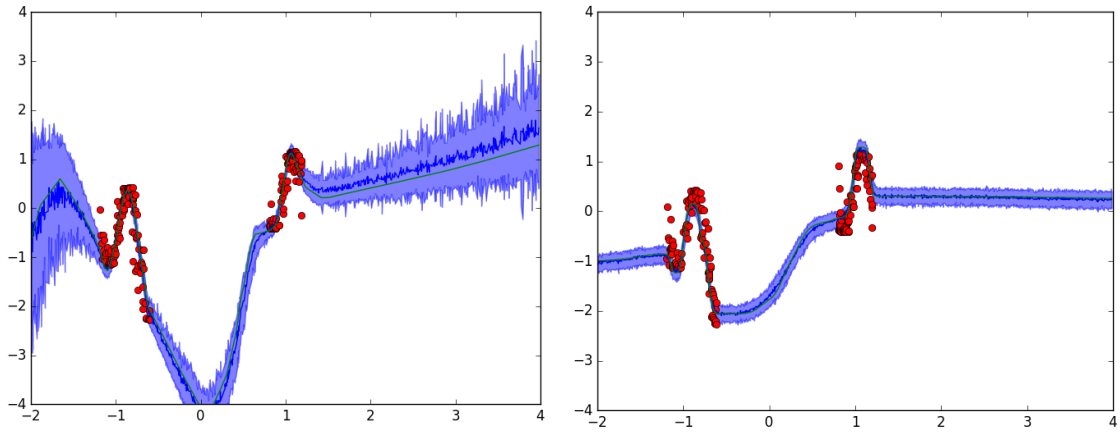


Figure C.1: Comparing ReLU versus Tanh for MC dropout. The plot on the left is represents ReLU and the one on the right represents Tanh. These results confirm the results found in (Gal & Ghahramani 2015). The red dots are the dataset presented to the networks. The dark blue line is the mean and the shaded area is two times standard deviation.

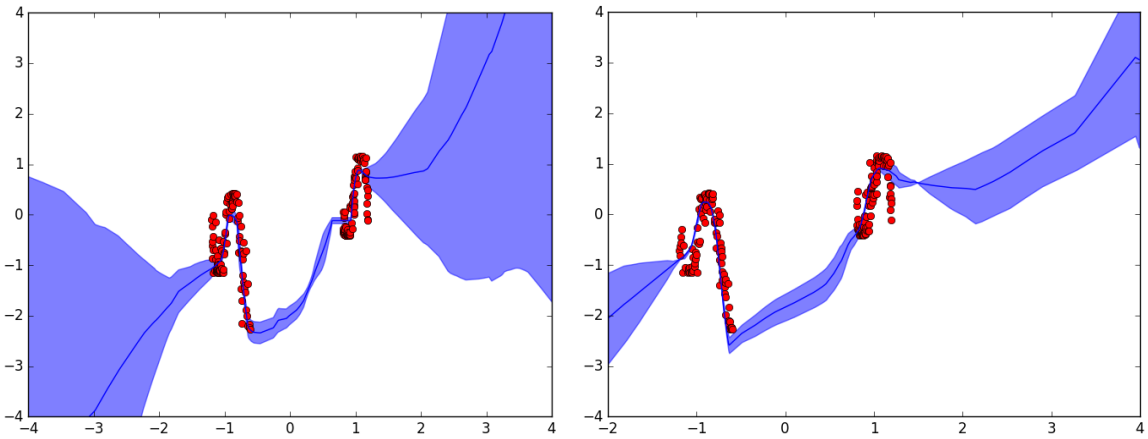


Figure C.2: Uncertainty for bootstrap with different numbers of heads. More heads results in higher and more precise uncertainty. Two heads (right) in this situation does underestimate the uncertainty, while the four heads (left) gives a more natural uncertainty estimate. Both models learn a decent representation of the real function.

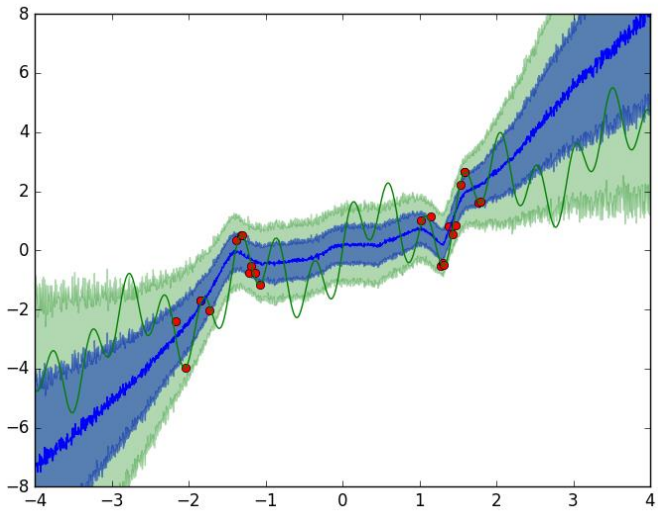


Figure C.3: Combining bootstrap with four heads and MC dropout gives the best uncertainties for this regression task, but is also the heaviest computationally wise. The red dots are the dataset, the green line is the true function, the blue line is the average of the output, the blue area is one times standard deviation and the green area is two times standard deviation.

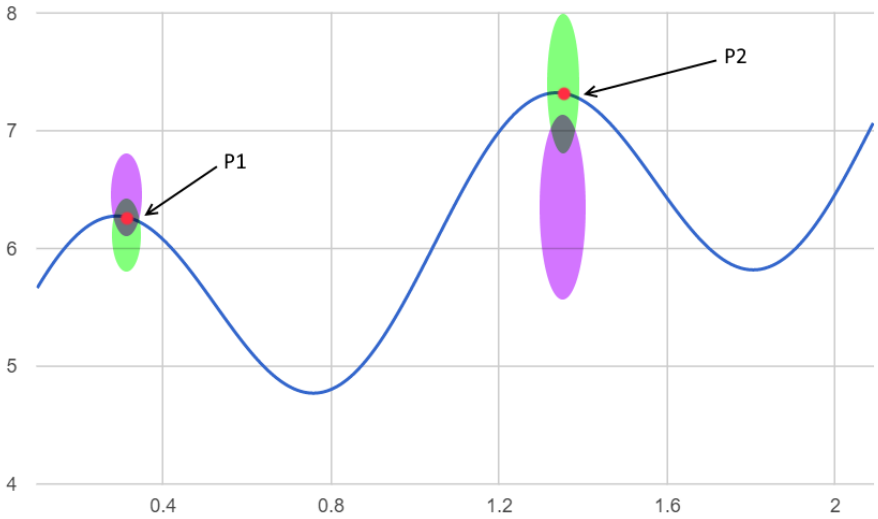


Figure C.4: Illustration of the combination of bootstrap and MC dropout. The plot represents the uncertainty in a regression task with a combination of bootstrap with two heads and MC dropout. The green fields represent the distribution of outputs from MC dropout for head 1 while the purple fields represent the distribution of outputs from MC dropout for head 2. We see that this gives a distribution of uncertainties instead of only giving single point outputs which would occur when only using bootstrap or MC dropout for themselves. P1 represents a data point that are often observed and P2 represents a data point which is rarely observed.