FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Data Augmentation for Reinforcement Learning

Martin Nykodem

Department of Knowledge Engineering

Supervisor: MSc. Juan Pablo Maldonado Lopez, Ph.D.

May 13, 2019

Acknowledgements

First, I would like to thank my thesis advisor MSc. Juan Pablo Maldonado Lopez, Ph.D., for his expert advice and patient guidance through thesis writing.

Also, I am very grateful to Ing. Eliška Šestáková for providing valuable advice and corrections on the thesis stylistics.

Finally, I must express my very profound gratitude to my family and friends for supporting me throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Declaration

In Prague on May 13, 2019 . . . .

Citation of this thesis

Nykodem, Martin. *Data Augmentation for Reinforcement Learning.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

V této práci je implementován nedávno představený framework pro posilované učení výzkumníků Ha a Schmidhubera nazvaný *World Models*. Ti přichází s originální myšlenkou naučit se svět z mnoha aspektů a ne jen ze zkušeností. Aby toho dosáhli, rozdělili jejich algoritmus do tří hlavních částí – zrak, paměť a řízení. Tento způsob vnímání světa je blíže tomu, jak to dělají lidé, či zvířata.

Implementace tohoto přístupu přináší různé výzvy, jelikož se nedá přímo převést na nové prostředí.

Výsledky jsou srovnatelné s metodami nevyužívající model, ale pro jejich dosažení bylo potřeba mnohem méně intarakcí s prostředím. Tato technika má tedy dopad na dlouho trvající problém zvětšování dat, který je zásadní pro nasazení systémů posilovaného učení do reálného světa.

V teoretické části práce jsou poskytnuty informace potřebné k pochopení fungování jednotlivých částí *World Models*, úvod do posilovaného učení a další přístupy pro zvětšování dat pro posilované učení.

Klíčová slova strojové učení, posilované učení, model-based, World Models

Abstract

A recently introduced framework for reinforcement learning, called *World models* by Ha and Schmidhuber, has been implemented for this thesis. They came with a novel idea to learn the world from many channels, not only experience. To simulate this approach, they split the algorithm into three main components – vision, memory, and control. This appears to be closer to what animals and humans do.

The implementation of this approach has its challenges and does not translate directly into a new environment.

The results are comparable with model-free methods, but with much fewer queries to the environment. This technique has implication for the long-standing problem of data augmentation, which is crucial to the deployment of real-life reinforcement learning systems.

The theoretical part of the thesis provides a general background for understanding the components of the *World Models* along with a brief introduction to reinforcement learning and the underlying ideas behind other data augmentation techniques for reinforcement learning.

Keywords machine learning, reinforcement learning, model-based, World Models

Contents

List of Figures

List of Tables

List of Listings

Introduction

In recent years, reinforcement learning has made some substantial achievements, and it becomes a fast developing dynamic field of machine learning, where new methods are being developed consistently. However, a lot of them often suffer from being data inefficient. This limits the applicability of these methods to complex domains such as robotics. Several approaches for data efficient reinforcement learning have been proposed like PILCO, MBVE or *World Models* (see [1, 2, 3]). The connecting idea is to replicate the environment of the agent (by creating a model or learning the distribution of the states given actions and previous states).

The model-based data augmentation framework called *World Models* from the recent paper of Ha and Schmidhuber 2018 comes with a novel idea to learn the world from many channels, not only from experience. The inspiration for the framework was the predictive model of our brains. Humans develop an abstract model of the world based on what they perceive with their senses and use the inner predictions of the model for decision making. These predictions are not just about the future in general, but also about the future sensory data. Although this idea was always present, this is the first time it was successfully applied to solve reinforcement learning problems.

It was achieved by creating a world model consisting of a vision and memory module, which learns an abstract representation of spatial and temporal aspects of the environment in an unsupervised manner. The controller provided with features extracted from the world model has fewer parameters, as most of the complexity is left to the world model. A small controller allows to use a special training algorithm for dealing with sparse rewards – currently one of the biggest problems in reinforcement learning.

For the implementation, Ha and Schmidhuber utilized the power of deep neural networks specifically in the form of variational autoencoder for compressing the observation, LSTM (long short-term memory) networks with predictions based on the probabilistic distribution for learning the dynamics of the model and evolution strategies for finding the correct parameters of the controller's network.

Our goal is to apply the mentioned framework to the OpenAI gym Atari emulated environment called Skiing-V0, analyze the performance of the achieved result, and consider possible approaches to improve the framework, especially on the controller part. Our implementation is written in Python with the support of machine learning frameworks *TensorFlow* and *PyTorch*.

This text is intended for readers who want to have a better understanding of the basics of reinforcement learning and grasp the concepts, which are used in the implementation of the *World Models* paper.

I chose this topic since reinforcement learning has been showing a lot of success recently, and it also seems to be the closest to the way we learn. In this thesis, I intend to explore one of the latest approaches.

The rest of this thesis is organized as follows. Chapter 1 provides a brief introduction to the types of machine learning techniques with a particular focus on reinforcement learning. Next, we explain the necessary theoretical background needed for understanding the implementation. At the end of the chapter, the underlying ideas behind various data augmentation techniques are outlined. At the beginning of Chapter 2, the tools used during the implementation are mentioned. Afterward, we describe the architecture, training process, optimization and improvements of the individual parts for the *World Models* framework application on the Skiing environment. The results of the training are presented in Chapter 3. Finally, the contributions of this thesis are summarized in Chapter 4, where possible improvements for future work are also suggested.

Chapter 1

State-of-the-art

In this chapter, we provide a brief introduction to machine learning followed by reinforcement learning basics, terminology and types. Afterward, we describe the concepts needed to understand the implementation such as neural networks types, and its specifics or evolution strategies. In the end, we reveal some ideas behind various data augmentation techniques for reinforcement learning.

1.1 Machine Learning

Machine learning (ML) is a part of computer science, where a computer can perform tasks without the need for specific instructions. To gain this ability the computer needs at first to learn a mathematical model from the training data. [4]

The machine learning techniques can be divided into three main categories by way of training the model:

Supervised learning is primarily used for classification and regression. The model is provided with data and its labels (desired solution) as training examples.

Unsupervised learning is mainly utilized for clustering, data visualization, and dimensionality reduction. The model is learning without a teacher, so the algorithms are left to find the structure of data. The data are self-organized to clusters with similar properties.

Reinforcement learning is very different from the previous two and will be discussed in detail in Section 1.2 since it is the basis of the thesis implementation.

1.2 Reinforcement Learning

When we ask ourselves what is the way we learn, we probably think of learning from interaction with our surroundings. As from our birth, we had to learn a lot of things, and there was no teacher to tell us how to do it. Nor we had capabilities to understand it. The repeated actions in the environment gave us enough information about the environment functioning and causality. With this knowledge, we could choose the right actions to achieve our desired goal. Even in our adult life, the interaction with the environment still plays a major part in our learning as we have to constantly modify our behavior by choosing new actions based on changes in our surrounding.

The reinforcement learning (RL) framework is inspired by the idea of learning from the interactions with the environment. In this case, the model is called an agent and can observe an environment, select and perform actions and get in return rewards or penalties [5]. This process is visualized in Figure 1.1. Its goal is to maximize the reward received from the environment. Therefore, it learns to map the right action to the received states. This is not an easy task as in some cases the reward is delayed.

RL is mainly used for teaching computer or robots to perform well in a variety of simulation or even the real world. Its capabilities were recently demonstrated when an algorithm AlphaGo from Google's Deep Mind defeated world-class players in the board game Go, or algorithm AlphaStar from the same team defeated professionals in an online cooperative game StarCraft II. [6, 7]

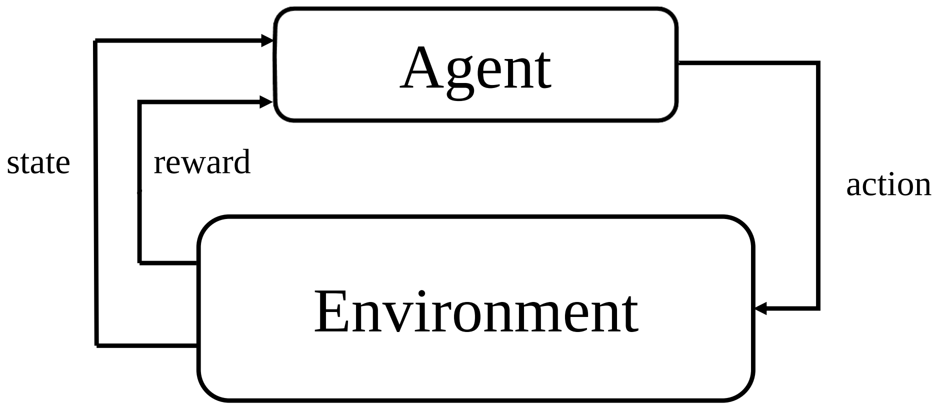


Figure 1.1: Visualization of interaction between environment and agent.

1.2.1 Terminology

In this section, we set a definition for some basic RL terms.

State and Observation of the environment the state S_t is a complete description of the state of the world at time step t. An observation is a partial description of a state, which may omit some information. Although frequently the notation S is put instead of O when actually is speaking about observation. In the thesis, we will use this approach as it should be clear from the context which case is meant.

Typically the observation is represented by a real-valued matrix, e.g., RGB matrix of pixel values, current data loaded in the RAM, values from robot's sensors. [8]

Observation space defines the structure of the observation.

Action space determines the range of possible actions available to the agent. In *discrete* action space, action can be represented by only a finite number of values. In *continuous*, actions are real-valued vectors.

This distinction has consequences for choosing the learning method. Some families of algorithms can only be applied in one case and would have to be substantially reworked for the other. [8]

Done is a flag received by the agent from the environment when it is no longer possible to take actions, e.g., the player died in the game, the time limit is up, or agent completed the goal.

Reward is in RL a part of the feedback from the environment. When an agent interacts with the environment, he can observe the changes in the state and reward. This helps him to choose the right action because the goal for the agent is to maximize cumulative reward (sum of all rewards) received during the series of steps in the environment.

The reward at time step t is defined as:

$$R_t = r(S_t, A_t),$$

where A_t is action performed by an agent at time step t and r is the reward defining function.

Policy is a rule used by an agent to decide what actions to take.

Policies can be separated into two categories:

Deterministic is usually denoted by μ:

$$A_t = \mu(S_t).$$

Stochastic is usually denoted by π:

$$A_t = \pi(\cdot|S_t).$$

In deep RL policies depend on a set of parameters (e.g., the weights and biases of a neural network) which are adjusted by some optimization algorithm. To highlight the connection is a θ parameter commonly added as a subscript on the policy symbol.

Because the policy is essentially the agent's decision mechanism, it is not uncommon to substitute the word "policy" for "agent". [8]

1.2.2 Types of Reinforcement Learning Techniques

In this section, we show the different approaches of training agent in RL and compare their benefits and trade-offs. There is not a clear way to group the RL methods. One of the main branching points could be if the model uses the model of the environment (a function which predicts state transitions and rewards).

Methods which do not use the model are called *model-free*, and they tend to be easier to implement and tune. Popular methods for model-free learning are Q-learning [9], A2C/A3C [10]. On the other hand, methods using the model predictive function are called *model-based*.

The main advantage of having the model is the possibility to include the prediction for the next states to the received observation; therefore, having a better ability to plan for the future. This looks like an obvious way to go. However, in most cases, the model of the environment is not available to us. Learning the model is usually harder to implement and creates a new problem. The learned model is not perfect and the agent uses its imperfection to hack its reward function and perform well in it, but in the actual environment performs poorly. [8]

1.3 Artifical Neural Networks

One of the reasons why machine learning became so powerful and capable of learning various complex tasks such as natural language processing, computer vision, playing games is an invention of artificial neural networks (NN). The model of NN is inspired by a biological brain. It consists of many interconnected units called artificial neurons or simply *neurons*. The connections between them are weighted, and single neuron can have multiple value inputs from other neurons, but only one value output. [5]

Typically, the neurons are aggregated into layers, which perform a specific transformation on their inputs. The layers are separated into three types:

Input layer has the number of neurons equal to the size of the input to the network. The neurons in this layer are usually used only for loading the data and does not make any modification to it.

Hidden layer contains neurons, which by adjusting weights can compute complex functions.

Output layer produces the result of the neural network.

In Figure 1.2, we illustrate an architecture of the feed-forward neural network with two hidden layers. In the feed-forward NN, the information flows from the input layer to the output. In other words, the network does not contain any loops.

Although the *Universal approximation theorem* [1] states that feed-forward neural network with one hidden layer containing a finite number of neurons can approximate any continuous functions on compact subsets of $\mathbb{R}^n$. [11] It is certainly not a practical way to do. For learning complex functions, neural networks with more hidden layers (called “deep networks”) are used, since their hierarchical structure makes them well adapted to learn the hierarchies of knowledge. We provide an example of such network in Section 1.4.

Theorem 1 (Universal approximation theorem) *Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote compact subset of $\mathbb{R}^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$.*

Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$

as an approximate realization of the function f that is,

$|F(x) - f(x)| < \varepsilon$ *for all* $x \in I_m$.

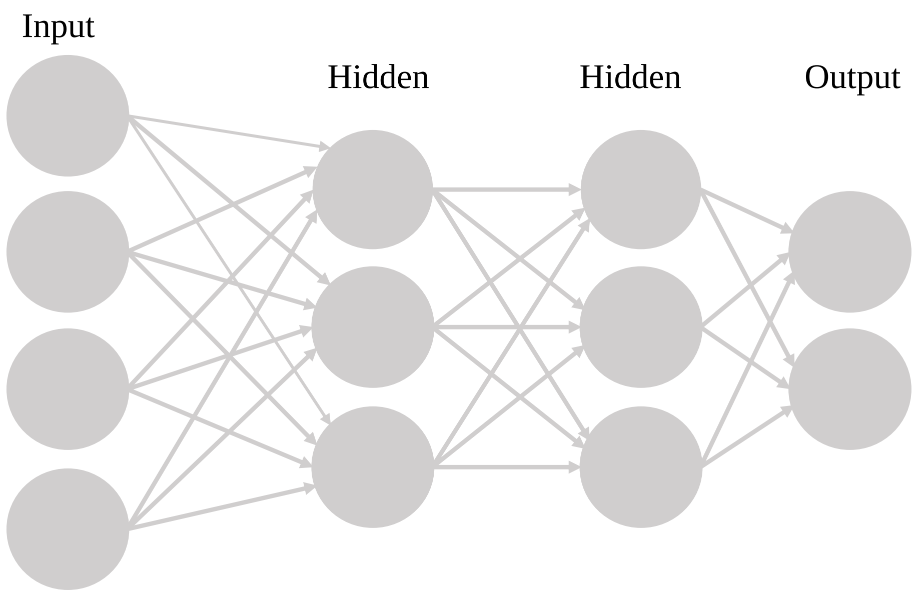


Figure 1.2: Feed-forward network

1.3.1 Activation Function

The neuron without an activation function works as a linear separator. In the real world, most functions are not linearly separable. Which is why there was added an activation function. This enables the non-linear transformation of the output, therefore, having the ability to encode complex patterns of the data. [12]

Its input is a sum of the weighted sum of neuron's input and bias, which behaves like a constant, that shifts the activation function to the left or right. Mathematically defined as:

$$f(\sum x_i w_i + b),$$

where x_i and w_i is the i-th input and weight of the neuron, b is the neuron's bias, and f is an activation function. We must denote that some works use different notation, which describes the computation of the whole layer output and is written as matrices operation:

$$g(Wx + B),$$

where W, x, B represents the weight, input and bias matrix and g is an activation applied to each value of the resulting vector.

The process of calculating the neuron's output value is visualized in Figure 1.3

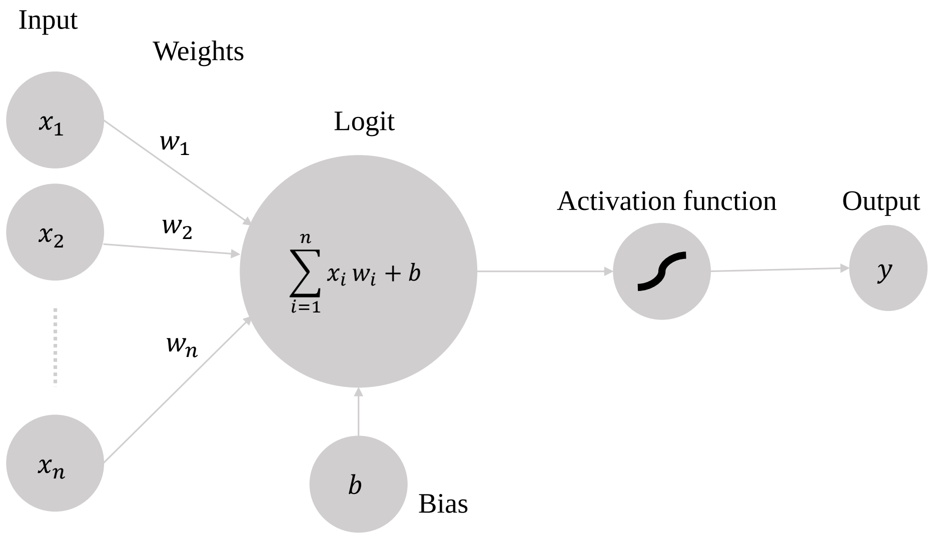


Figure 1.3: Neuron's output value computation

There are many activation functions, each with its benefit. Here we show the commonly used ones. Comparison of their function graphs is shown in Figure 1.4

Sigmoid function is defined as:

$$f(x) = \frac{1}{1+e^{-x}}.$$

From the function's graph in Figure 1.4a it is visible that it maps the values between 0 and 1 and that on the curve between X values -2 to 2, Y values are very steep. This means any small changes in the values of X in that region will cause values of Y to change significantly. So the sigmoid function tends to bring the Y values to either end of the curve.

Due to this property, it can make a clear distinction on the output and making it suitable for classification tasks.

Tanh function is defined as:

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

This function has similar characteristics to sigmoid that we discussed above (in fact, it is a scaled sigmoid function). It maps the outputs between -1 and 1 and its derivative is steeper, which causes gradient (described in Section 1.3.3) to be stronger. [13]

ReLu function is defined as:

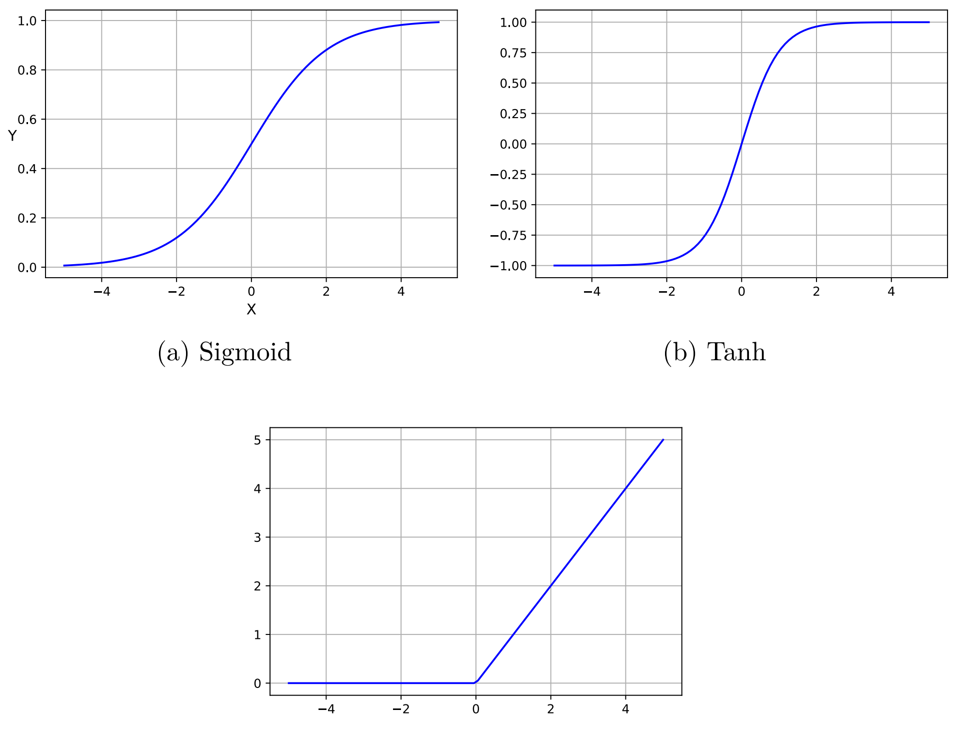
$$f(x) = max(0, x).$$

This means it gives an output x if x is positive and 0 otherwise. ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. However, the property of having all negative values become zero may cause the neural network to stop training.

Softmax is a special type of an activation function, which uses logits (raw scores output by the last layer of a neural network before activation takes place) to compute neurons output. It is defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}},$$

where x_j are the logits of the layer. The sum of the layer output adds to one. For this property it is commonly used for probabilistic classification.



(a) Sigmoid

(b) Tanh

(c) ReLu

Figure 1.4: Comparison of activation functions' graphs

1.3.2 Loss Function

The loss function is the mathematical formula, which helps to determine how far the model is from the ground truth.

The counted error is then used to update the weights between neurons using the chain-rule of backpropagation. This will not be fully described as it is not needed for understanding the thesis. Information about the algorithm can be found in [14].

Specific loss functions are used for different tasks. Here we show the frequently used ones:

Cross entropy

To understand the cross-entropy we at first need to know what an **entropy** is.

Claude Shannon, the founder of the entropy concept, defined it in his paper “A Mathematical Theory of Communication” as the smallest possible average size of lossless encoding of the messages sent from the source to the destination. It is based on combining the knowledge of the probability distribution of messages types (for this example encoded in bits) and knowing the minimum size for their encoding.

In general, for encoding N different messages (values) expressed in bits we need at most $\log_2 N$ bits, but also we do not need more than that. If we know the probability P of the message, which is always in range between 0 and 1 we can express the same as $-\log_2 P$.

So the the minimum average encoding size – *entropy* $H(P)$ is defined as:

$$H(P) = -\textstyle\sum_i P(i) \log_2 P(i),$$

where $P(i)$ is the probability of i-th message. [15]

Cross entropy is used to measure how close is the true probability distribution to the predicted probability distribution. This quality is making it suitable for multi-class classification tasks.

Its mathematical formula is:

$$H(P,Q) = -\textstyle\sum_i P(i) \log_2 Q(i),$$

where P is the known distribution and Q is the estimated one.

Kullback-Leibler divergence

This is not commonly used as a loss function. However, it is a part of a loss function for the variational autoencoders, which will be mentioned in Section 1.6. The **Kullback-Leibler (KL) divergence** tells us how well the probability distribution Q approximates the probability distribution P by calculating the cross-entropy minus the entropy by a simple formula:

$$D_{KL}(P||Q) = H(P, Q) - H(P).$$

Mean squared error (MSE)

It is commonly used for regression problems where the goal is to predict n values. In principle MSE punishes large mistakes much more than small mistakes. [16] With that said, it is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{t=1}^{n} e_t^2,$$

where n is the number of outputs, and e is the difference between the original and the predicted value.

1.3.3 Gradient Descent

To optimize the weights and biases in the neural network, we commonly choose an algorithm called *gradient descent.*

A gradient is a direction on which a function grows. Using calculus, we know that the slope of a function at value is the derivative of the function with respect to that value.

It is usually written as:

$$\nabla f(\Theta),$$

where ∇ is denoting the gradient of the function f at values Θ.

Our goal is to minimize the function f parameterized by the network parameters θ. This is achieved by iteratively moving in the direction of the negative of the gradient.

The simplest update rule is:

$$\theta \leftarrow \theta - \alpha \nabla f(\theta),$$

where α is the step size and is determining the size of the update.

This rule, however, fails in many practical problems (for instance, in neural networks with many parameters) so in practice, different heuristics for the step size and the update schedule are used.

[17, 18] There are multiple ways to calculate gradient unusually based on the type of the problem. However, two of them is usually used for most of the application:

Mini-batch gradient descent performs an update for every mini-batch of size n, which can lead to more stable convergence.

Adam (Adaptive moment estimation) based on the information about a certain amount of previous updates it increases updates for dimension facing in the same direction. Also, the algorithm is adapting the learning rate to the parameters.

When the neural network consists of many hidden layers a problem with gradient might arise:

Exploding gradient is a situation where some weights are assigned wrongly too high importance.

Vanishing gradient is a situation, where the weights of the model are too small to learn efficiently or eventually disappear.

1.4 Convolutional Neural Networks

Convolution neural nets (CNNs) are deep artificial NN that play a key role in the computer vision. More specifically they address problems with image classification, optical character recognition and object recognition within scenes.

The CNN was also inspired by functions of our brains – especially the visual cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area. Moreover, some neurons react only to images with horizontal lines and some only to lines with different orientation. [19]

An essential feature of the CNN is the ability to reduce the images into a form which is easier to process, without losing features which are critical for getting the right prediction. So how do they do it?

CNN typically consist of an input layer, multiple convolutional layers, pooling layers, fully connected layers, and the output layer.

Input Layer

When used for computer vision on the input is an image, which can be represented as a matrix with a dimension of image width, height and number of picture's channels, e.g., three channels in case of RGB and data representing the value of channel value at a given pixel.

Convolutional Layer

The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. It is done by using a stack of *Kernel/Filters*, which shifts through the image and use the matrix multiplication operation between the kernel and the portion of the image the kernel is currently at. The result or sum of matrix multiplication over the channels is an input for the next layer. The size of the shift in each dimension is given by the *stride* parameter. Another important feature for the CNN is *padding*. With value "valid" it means no padding. Value "same" results in padding the input such that the output has the same length as the original input. [20]

Pooling layer

The goal of the pooling layer is to subsample or shrink the input image in order to reduce memory usage, the number of computational operations and number of parameters which also party helps against overfitting.

As is convolutional layers, each neuron in the pooling layer is connected to the limited number of neurons in the previous layer. The only difference is it has no weights connecting them. Instead, it only aggregates data using either the maximum or average of the selected kernel.

Fully connected layer

The fully connected layer is a usually selected approach to learn non-linear combinations of the high-level features represented by the output of the convolutional layer.

1.5 Recurrent Neural Networks

In Section 1.3, we mentioned the feed-forward networks where the data flows straight from the input layer to the output layer. The RNN works in a similar way except it also has connection backward. The difference between them is demonstrated in Figure 1.5. These loops are allowing them to persist information from previous runs. When RNN makes a decision, it takes into consideration current input and a hidden state where the sequential information from previous inputs is preserved. Therefore, RNNs are well-suited to make predictions based on time series data.

In Section 1.3.1, we showed how the feed-forward neural network's output of the layer is computed by applying a weight matrix to its inputs. The RNNs apply weights to the current as well as to the information from previous input stored in the hidden state.

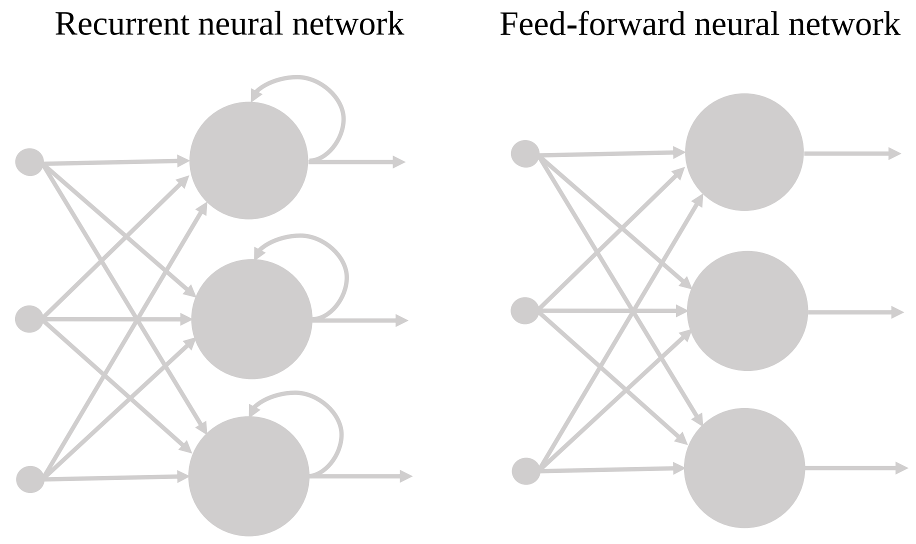


Figure 1.5: Difference between the recurrent neural network and the feed-forward neural network.

The computations of RRN's output vector y_t at time step t is defined as:

$$y_t = f(W_y h_t + b_y),$$

where f is an activation function, W_y is the weight matrix, b_y is bias and h_t is the hidden state at time step t, which is computed as:

$$h_t = g(W_h x_t + U h_{t-1} + b_h),$$

where the to the input x_t modified by a weight matrix W_h is added layer's bias b_h and the hidden state of the previous time step h_{t-1} multiplied by its own hidden-state-to-hidden-state matrix U. The result is again transformed by an activation function g. The weight matrices are filters that determine how much importance to accord to both the present input and the former hidden state. [21]

In fact, the RNN loops can be unrolled to a sequence of neural networks which are then trained one after another with backpropagation. As a consequence, RNN with large hidden states behaves like a neural network with many hidden layers. These, as we know from Section 1.3.3, often struggle with vanishing or exploding gradient.

The exploding gradient was solved by gradient clipping where the predefined threshold for the gradients keeps them from getting too large, but their direction stays the same.

Long Short-Term Memory Network (LSTM)

The solution to vanishing gradients was solved by modification of recurrent net with adding so-called "long short-term memory units" by Hochreiter and Schmidhuber[1]. [22]

The long short-term memory units also enable RNNs to remember their inputs over a long time. This is because LSTMs contain their information in a memory that is much like the memory of a computer since the LSTM can read, write and delete information from its memory. [23]

1.6 Autoencoders

An autoencoder is a special type of a network, which is capable of learning a way to compress input data into smaller representation without any supervision. Except for dimensionality reduction, it is also used for denoising the data, detecting features and generating new data similar to training data sets.

The way they do it is by taking an input (e.g., image or some vector) and run it through the first part of a network called an *encoder*. In the encoder, the data is compressed by several stacked convolutional or fully connected layers and outputted to the bottleneck layer, which size is a smaller dimension than the input. From the bottleneck vector (usually called z) the second part of the network called *decoder* takes the data and reconstructs them to the original form. The architecture is illustrated in Figure 1.6. The loss function for the vanilla autoencoder is usually just MSE between the original input and the reconstructed data.

The compression done by the autoencoders is data specific, as the weights of the network are tuned for data of particular characteristics, and will not work well on unseen datasets.

1.6.1 Variational Autoencoder

Variational autoencoder (VAE) is a special category of autoencoders introduced in 2014 by Kingma and Welling [24]. Its only difference from the vanilla autoencoders is that instead of mapping the input to the fixed vector z it produces a mean coding μ and standard deviation σ and the z is then sampled randomly from the Gaussian distribution with mean μ and standard deviation σ.

[1]If the name Schmidhuber sound familiar, it is because it was already mentioned as an author of the World Models paper.

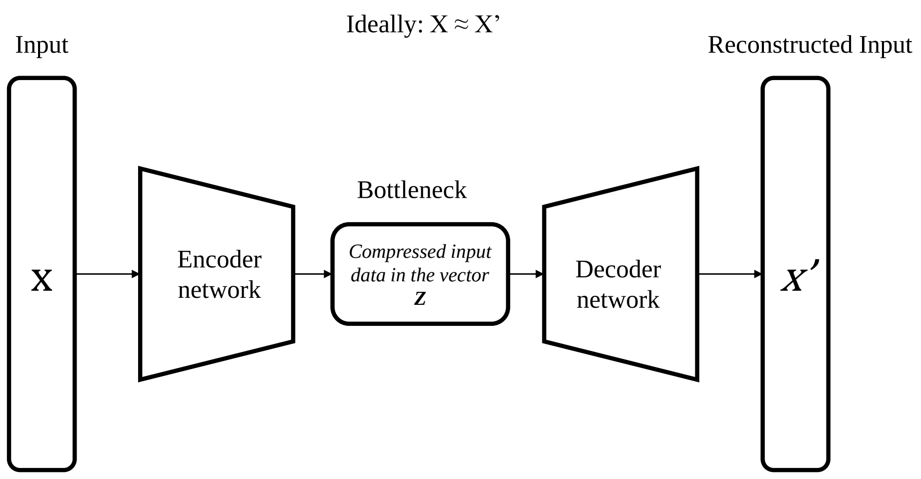


Figure 1.6: Autoencoder model architecture

The **loss function** pushes the codings to gradually migrate within the coding space (also called latent space) to occupy roughly (hyper)spherical region and consists of two parts: one is simply a reconstruction loss as in vanilla autoencoder, which forces autoencoder to correctly reproduce input data, except the expectation operator, because VAE reconstructs data from a distribution. The second is based on the KL divergence, that punishes the network for having the learned distribution too far from the normal distribution. As a consequence, it is possible after the training to take a random coding from the Gaussian distribution and decode it to generate new data similar to the trained one.

1.7 Mixture Density Network

Mixture density network (MDN) models are generally used to estimate the real distribution of some data, typically by assuming that each data entry has some probability of being associated with a certain Gaussian distribution.

So for each input is produced a set of mean, standard deviation parameters, and a set of probabilities that the output point would fall into those Gaussian distributions.

This sort of model can be useful if combined with neural networks, where the outputs of the neural network are the parameters of the mixture model, rather than a direct prediction of the data label. Also is commonly applied in cases where the target variable cannot be easily approximated by a single standard probability distribution. [25]

1.8 Evolution Strategies

In this section, we provide an introduction to the black-box NN weights optimization strategy based on a blog post [26]. In reinforcement learning, where NN is used for a decision making, it is not always easy to estimate the suitable gradient of reward signals given to the agent in the future to an action performed by the agent right now. For this kind of problems, it might be better to use black-box optimization algorithms such as evolution strategy (ES) instead of a possibly meaningless gradient.

The ES strategy can be defined as an algorithm, which at first randomly creates a population of candidates and then evaluates their performance by an objective function. Based on the candidates' evaluation the algorithm produces the next population, which should be able to perform better than the previous one. This process goes iteratively until the ending condition is met, e.g., performance is good enough or stopped progressing.

1.9 Data Augmentation Algorithms in Reinforcement Learning

As in many other ML fields, reinforcement learning suffers from being data inefficient and requires many trials or highly powerful computational resources to solve a relatively simple task. Hence it is still unusable or highly impractical to train ML tasks on real robots as the hardware tends to wear out after thousands of trials. To address this sort of problems the researchers are developing data augmentation methods.

In this chapter, we present underlying ideas behind different approaches. As it can be done by extracting knowledge from a few trials (PILCO); using learned dynamics of the model in model-free RL to reduce the sample complexity of learning (MBVE), or a way to learn the dynamics of the model so well it can be used as a training environment instead of running in a computationally expensive real environment (World Models).

1.9.1 PILCO

For model-based RL technique is hard to create a working model of the environment from a few observation as they tend to create its model bias (assuming that the learned model accurately resembles the real environment). The PILCO [2] (probabilistic inference for learning control) is getting around this problem by incorporating the model uncertainty into the model planning and control as indicated in Figure 1.7.

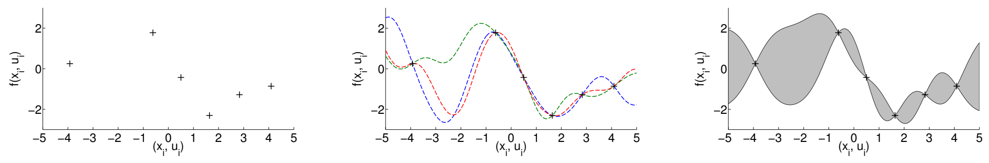


Figure 1.7: Small data set of observed transitions (left), multiple plausible deterministic function approximators (center), probabilistic function approximator (right). The probabilistic approximator models uncertainty about the latent function. Source: [2]

From data collected from the environment, the PILCO uses a Gaussian process (GP) to learn the dynamics of the model. After it is trained the GP sample many possible trajectories to train a controller, this approach is highly suitable for robotics tasks where the robot cannot be trained in a simulated environment. However not for all of them as it only works well on low dimensional data and GP computational complexity makes it difficult to model on high dimensional observation.

One of the great examples of PILCO's practical model-based policy search method is learning to swing a pendulum on a cart pole to the inverted position. To learn this task, it took the algorithm only 17.5 seconds, which is currently the best-achieved result.

1.9.2 MBVE

In the paper MBVE [3], (model-based value expansion) published in 2018, UC Berkeley team present a design which takes the information from the dynamic of the model uses it to reduce a sample complexity for calculating the value function in a model-free algorithm. Its logic stands behind the fact that model-free algorithm has the capacity of rich value function approximator to learn complex tasks, but needs a lot of iteration to do so. Moreover, model-based algorithms, with knowledge of the environment, can quickly arrive to near-optimal policy. Although for the complex tasks, the inaccuracy of the learned model fails to provide accurate predictions for the controller to learn a good policy.

This hybrid algorithm takes dynamic of the model to simulate the short-term horizon and Q-learning for estimating the long-term gains behind the simulated horizon. This provides higher-quality target values for training the Q-function. By splitting the estimates into distinct intervals and using the decoupled interface, no special algorithm modification needs to be done. The splitting horizon is determined by the level of trust in believing the model can make accurate estimates.

1.9.3 World Models

This paper published by Ha and Schmidhuber in May 2018 [1] explores the usage of generative neural networks to understand the environment and then by using features extracted from the learned model (or as they call it world model) as inputs to the controller. This allows to use fairly simple policy to solve the required task. In the second part of the paper is even demonstrated the ability to train the agent entirely inside of the dream environment, which is generated by the world model, and then transfers this policy back into the actual environment.

To tackle this task divide-and-conquer tactic was used and *World Models* framework is split it into three parts: visual model, memory model, and control.

The visual model takes as input a frame from the game and uses a VAE to produce a more abstract, and compressed representation of the frame as its latent vector z. In the next step, such encoded sequences concatenated with actions from a random game policy generated rollouts are fed into memory model (implemented by MDN-RNN), which learns how the environment tends to behave and is able to make predictions on the next state of the world – essentially learning a model of the world. Lastly evolution strategies are used for the controller, which only decides next action a at timestep t as $a_t = W_c * [z_t, h_t] + b_c$, where W_c is the weight matrix; b_c the bias vector and $[z_t, h_t]$ is the concatenated VAE's latent vector and the MDN-RNN's hidden state at timestep t. Using a controller that is relatively simple and keeping the complexity to the world model allows using interesting techniques to train the controller.

The experiments were done on two environments called *Car Racing* and *VizDoom*. The advantage of having a model for predicting was demonstrated on the *Car Racing* experiment. The ability to train in the dreamed environment was shown in the *VizDoom* experiment.

Chapter 2

Implementation

In this chapter, we are going to explain what was our approach to implement the *World Models* framework from the paper of Ha and Schmidhuber. [1]

At first, we describe the tools used during the implementation. Afterward, we present the used environment and our applied modification and improvements to it. In the following sections, we discuss in detail the individual modules, their architecture, and training process. For the controller, we introduce various approaches to decision making. In the last section, we display the complexity of individual modules and illustrate how they interact together.

2.1 Used Tools

In this section, we describe the tools and frameworks used for the implementation.

Python is an interpreted, high-level, programming language developed under open source license with the support of thousands third-party modules. It is also the most popular programming language used for Machine Learning. [27, 28]

Jupyter Notebook is an open-source web application that allows to create and share documents that contain live code, equations, visualizations, and narrative text. [29]

Collaboratory is a free to use Jupyter notebook environment provided by Google allowing to utilize the computational power of Tesla K80 GPU, two CPUs at 2.2GHz and 12 GB of RAM. The computing is unfortunately limited to a maximum of 12 hours of continuous run-time. If the computation needs to be prolonged, its state must be saved and restored again. [30]

Tensorflow is an open source library developed by Google Brain team, which is suited for machine learning tasks. It also comes with a great tool for visualizing the learning progress and many other applications called *TensorBoard*. [31].

PyTorch is an optimized open source deep learning library for Python. [32]

Pycma is a Python implementation of an evolution strategy called CMA-ES and a few related optimization tools. [33]

OpenCV is a library of Python bindings designed to solve computer vision problems. [34]

Gym is a Python package developed by the Open AI, which provides various environments for developing and comparing reinforcement learning algorithms. [35]

2.2 Environment

We chose one of the environments of Atari emulated games provided by the Open AI gym called Skiing-V0.

For better interaction and repairing some of its flaws (described below), we created a wrapper class, where we added or re-written a few functions. The example of the wrapper class along with main environment functions and description of their behavior is shown in Listing 1.

```python
from gym.envs.atari import AtariEnv

class SkiingWrapper(AtariEnv):
  def __init__(self, interpolation=cv2.INTER_CUBIC, env_reward=False):
    super(SkiingWrapper, self).__init__(game='skiing', obs_type='image')

  def reset(self):
    # ... Resets to a new game and returns an observation

  def step(self, action):
    # ... Returns observation, reward, done flag and diagnostic information

  ...
```

Listing 1: Environment wrapper and main functions

2.2.1 Reward

Unfortunately, the reward function is not described in the documentation and from our findings, the reward is assigned at the end of the run. For the controller's training optimization, we created our self-defined reward function, which is detecting states of going through the poles or crashing by looking for specific changes in the frames using OpenCV.

The **rewards** are:

$+100$	is given for going through the slopes
-5	is given when the skier crashes
0	else

2.2.2 Observation

The observation of the environment returned from the OpenAI gym was not optimal for the training, so some preprocessing actions were done before returning:

Cropping – the observation returned by the environment was too large, so it was cropped from unnecessary information.

Resizing – the observation after cropping was still too large, so we resized it using OpenCV. We found that the resize method `INTER_CUBIC` works best for the variational autoencoder even though it ends up a bit blurry it contains more useful information for the VAE, than for example sharp edge observation resized by `INTER_NEAREST` method.

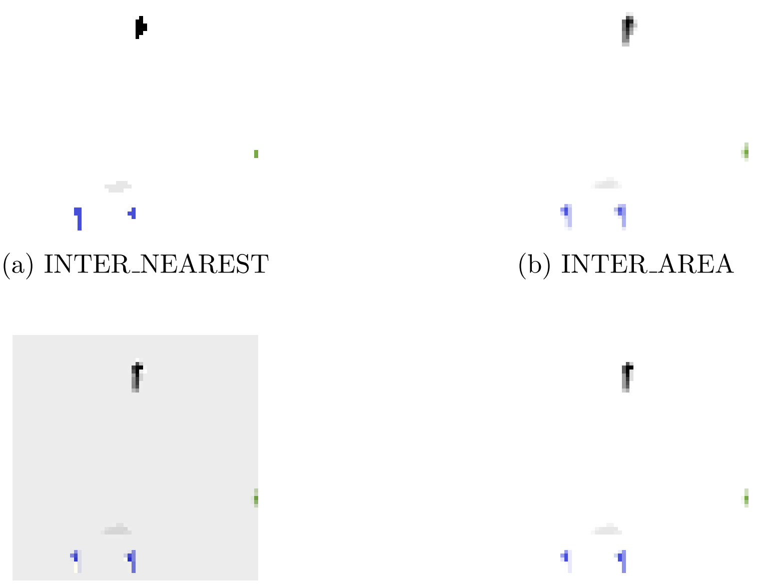
Figure 2.1 shows a comparison of the tested resizing methods.

Recolorization of the skier body – the skier original color on the RGB scale was too close to the white background, so the VAE decided not to tackle this problem, but it was more focused on getting the trees and poles right. This trick led the training much quicker to the satisfactory results.

Normalization – improves the convergence speed and accuracy of the NN. In our case we scaled the image data in range of $[0; 255]$ to $[0; 1]$.

In the end, we lowered our the **observation space** from (250, 160, 3) to (128, 128, 3).

In Figure 2.2 we show the comparison between the original observation and the preprocessed one.



(a) INTER_NEAREST (b) INTER_AREA

(c) INTER_CUBIC (d) INTER_LINEAR

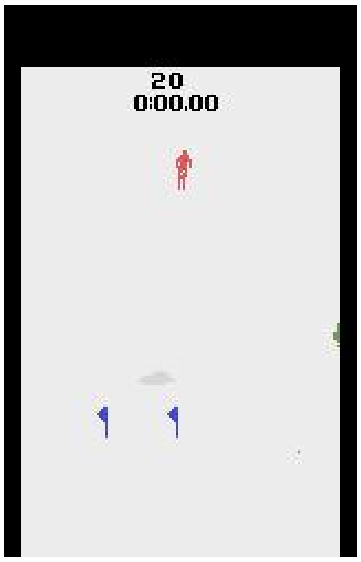
Figure 2.1: Illustration of various resize methods

2.2.3 Actions

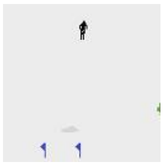
The actions for the Skiing-V0 environment are from discrete action space of size three defined by a single integer, where:

- 1 – turning left
- 2 – turning right
- 0 – continuing in the current direction

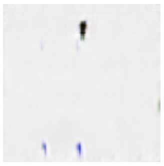
Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from $\{2, 3, 4\}$.



(a) Original frame



(b) Preprocessed frame



(c) Reconstructed frame

Figure 2.2: Frames comparison

2.3 Summary of the Training Procedure

The training was done in a series of steps as follows:

1. Save 16 rollouts of the environment with a random-policy agent.
2. On the frames train the vision module implemented by a VAE.
3. Save the action a from random-policy agent rollout along with the encoded frames in the latent vector z from trained VAE from 40 rollouts.
4. Train the memory module implemented by MDN-LSTM to be able to predict the next state.
5. Use the evolution strategies to find suitable weights and biases, with the provided values of the hidden states from MDN-LSTM and vector z, for the controller NN.

2.4 Vision Module

The vision module was implemented by variational autoencoder as Ha and Schimdhuber did in the original paper. The main reason for using VAE is it compresses high-dimensional space into smaller representation. In our case latent vector $z \in \mathbb{R}^{256}$. The probabilistic design of VAE makes the model more resistant to the unrealistic vectors produced by the memory model. During the implementation, we tried using vectors of a smaller size, but they stopped improving at the point where the reconstructions were not satisfactory.

In the experiments of *World Models* paper they used latent $z \in \mathbb{R}^{32}$ for Car Racing and $z \in \mathbb{R}^{64}$ for VizDoom. This was probably on account of the observation size of the environments were smaller and in case of Car Racing the observation was relatively simple and did not vary too much in the rollouts.

2.4.1 Architecture

For the encoder, we used four convolutional and for decoder five deconvolutional layers with strides of size (2,2) and valid padding. The implementation of this network was done in pure Tensorflow. Detailed VAE architecture with the number of trainable parameters is shown in Table 2.1.

Layer type	Filters #	Kernel shape	Output shape	Param #
placeholder	-	-	(128, 128, 3)	0
conv2d	32	(4, 4)	(63, 63, 32)	1568
conv2d	64	(4, 4)	(30, 30, 64)	32832
conv2d	128	(4, 4)	(14, 14, 128)	131200
conv2d	256	(4, 4)	(6, 6, 256)	524544
dense	-	-	(256)	2359552
dense	-	-	(256)	2359552
dense	-	-	(1, 1, 1024)	263168
conv2d_transpose	128	(5, 5)	(5, 5, 128)	3276928
conv2d_transpose	64	(5, 5)	(13, 13, 64)	204864
conv2d_transpose	32	(5, 5)	(29, 29, 32)	51232
conv2d_transpose	16	(6, 6)	(62, 62, 16)	18448
conv2d_transpose	3	(6, 6)	(128, 128, 3)	1731

Table 2.1: Architecture of VAE

2.4.2 Training Process

Generating Data

For training the agent, we generated 16 runs of random policy rollouts and 8 roll-outs for validation. In the *World Models*, they used 10,000 rollouts, but that seems unnecessarily too large. See in Figure 2.3, where is demonstrated loss value during the training on the training data and validation data.

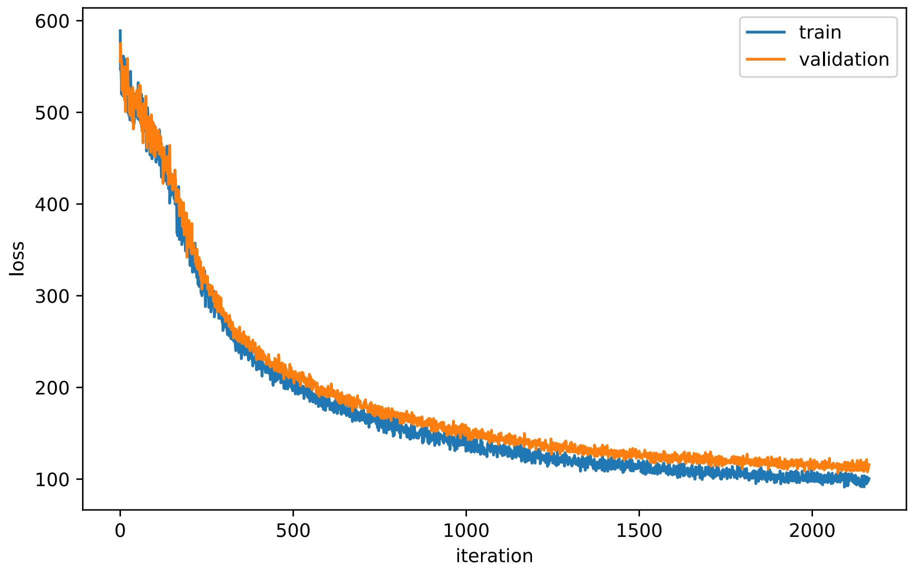


Figure 2.3: Validation and training loss during the VAE training

Training Procedure

For loading the data, we used data iterator, which loaded frames from the saved random policy rollouts and returned the batch of size 64. For optimizing the weights by minimizing the VAE loss, the Adam optimizer with a learning rate of 10^{-4} achieved the best results.

Evaluation of Training

The computing was performed on GPU hardware accelerator, but the time of training for VAE was not possible to capture precisely in the Google Collaboratory as a time of closing the session and computing resources are not granted and stable. But it should take less than a day to retrain the VAE from scratch.

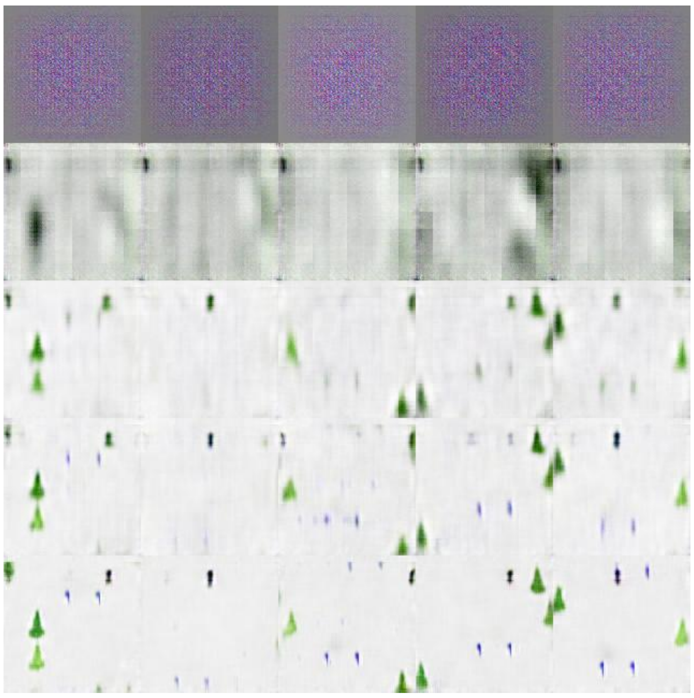


Figure 2.4: VAE reconstruction progress for specific frames (in the columns). In the rows are sampled reconstruction during the training from the first reconstruction to the last.

Figure 2.4 shows the reconstructions progress during the training to the from the first iteration to the final.

2.5 Memory Module

For the memory module, we used an LSTM network with a mixture of Gaussians as an output, which predicts the probability distribution for the encoded vector of the next frame z_{t+1}. More specifically the LSTM is predicting $P(z_{t+1} \mid a_t, h_t, z_t)$, where a_t is the action taken at time t and h_t is the hidden state of the LSTM at time t and z_t is the latent vector at time t.

The reason for using the probability distribution is that we will never learn the full model of the environment and for unknown states, it is better to model multiple possible scenarios with corresponding probability rather than single prediction.

2.5.1 Architecture

For the implementation, we modified the memory module of Djian implementation Sonic Retro Contest done in PyTorch [36].

The MDN-LSTM contains 1024 hidden units and outputs 5 Gaussians mixtures. This is larger than 512 hidden units used in the original implementation, but it could not predict well with the hidden state of that size. On the other hand, we had to shorten the sequence length from 1000 to 100 as the training process would take too long. This might reflect on the performance since in the original paper the sequence length was the same as one play and they did not have to reset the hidden state during the rollout.

2.5.2 Training Process

Generating Data

For the training, we generated another 40 random policy rollouts, where we saved action a, encoded observation in latent vector z. Which is again fairly low against 10,000 rollouts used in the original, but enough to sufficiently learn the model dynamics. We can see in Figure 2.6 that some causality is learned.

Training Procedure

On the input, the memory module was given a vector made by a sequence of 100 contacted action and latent z. The target vector was the same sequence of latent vectors shifted to the left, so it starts with the second frame, and to the end was duplicated the last frame.

For the optimizer, we used Adam with a learning rate of 10^{-4}.

Evaluation of the Training

The MDN-LSTM took it about a day till the performance stopped improving. In Figure 2.5 is plotted the decrease of the loss value saved during the training and the reaching of some local minima.

In Figure 2.6, the trained memory module "dreaming" for nine frames is visualized. The hidden state is updated after each prediction. The input for the prediction z_t is in this case previous predicted vector z_{t-1} and action "down". There is visible, how the memory module is predicting the future in the believable matter for the first few frames, but later it starts to dream an utterly different environment.

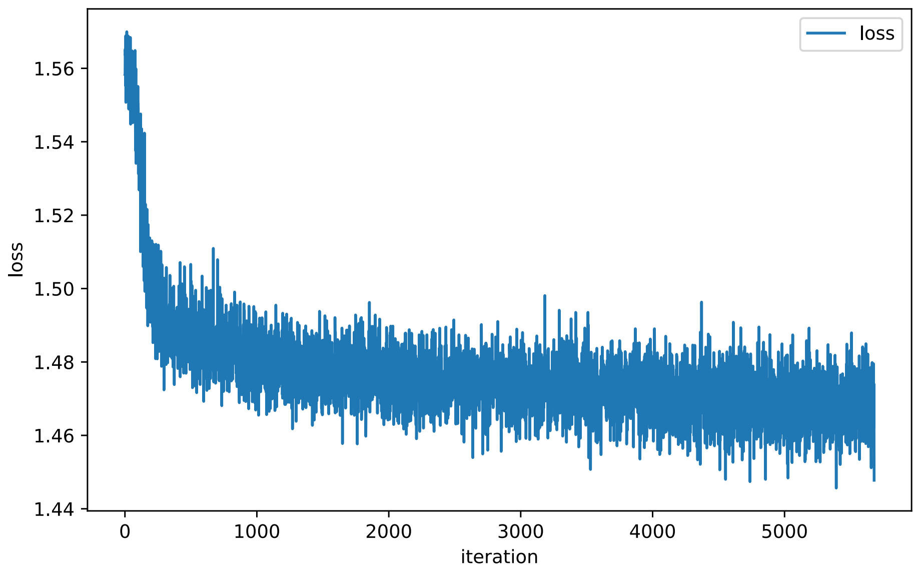


Figure 2.5: Memory module training loss

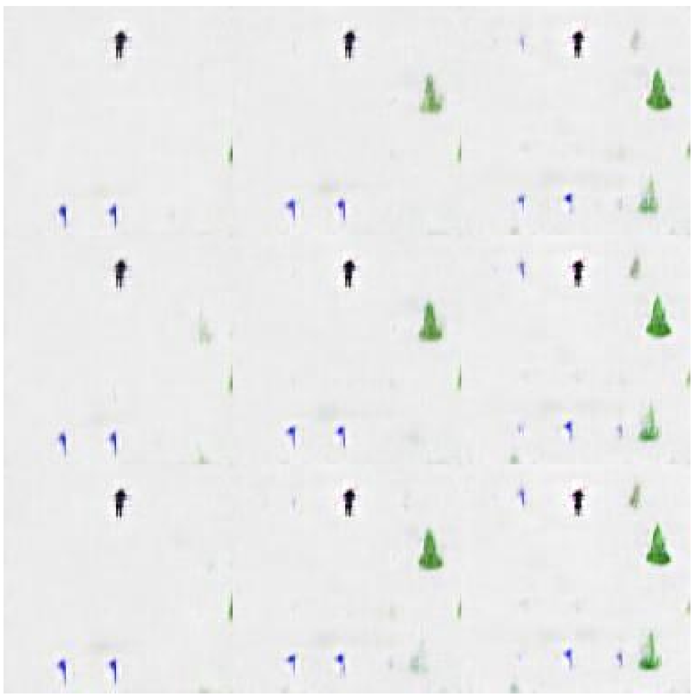


Figure 2.6: Visualization of MDN-LSTM dreaming. The first frame is in the left corner. The following are in the direction down, continuing on the next right column in the same manner.

2.6 Controller

The controller is choosing which actions to take for maximization of the cumulative reward. In this section, we show some possible optimization for the controller's decision mechanism.

2.6.1 Architecture

Because most of the complexity was left on the memory and vision module, the controller can be implemented as a simple neural network where a single linear layer is used. It maps the concatenation of the current latent representation of the frame z_t and the hidden state and cell of the LSTM h_t to an action a_t at each time step t:

$$a_t = W_c\ [z_t\ h_t]\ +\ b_c,$$

where the W_c and b_c are weights and biases of the controller network.

2.6.2 Decision Approaches

The controller must output an action, which is a single integer and values can be only $0, 1, 2$. To accomplish this, we tried three different approaches:

Softmax

In this approach, the neural network outputs three values, which are then converted using *softmax* operation to a vector that represents the probability distributions of potential action. The action is then chosen based on its probability.

Argmax

For the deterministic *argmax* approach, the neural network also outputs three values and as the name suggests – the controller chooses an action, which corresponds to an index with the highest value.

Single value output (SVO)

In this situation, is at first an interval $[-1; 1]$ split into three same length parts – each representing one action. The neural network outputs only single value, which is scaled using *tanh* activation function to $[-1; 1]$ and the action with the corresponding interval is then chosen.

2.6.3 Training

The small number of parameters of the controller model allows us to use the evolution strategy for its training. In this case, it is highly convenient as in this environment the rewards are fairly sparse. For optimization of the parameters, we used Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) from python package pycma since it works well for up to 10,000 parameters.

Weights Optimization Procedure

The evolution strategy from pycma package proposes some initial solutions (controller's weights and biases) of the population size, which are evaluated by the fitness function. In our case, the fitness function returns the controller's average cumulative reward from rollouts. The evaluated fitness list is then used for proposing a better solution.

This process is illustrateted in Lising 2.

```
while not es.stop():

  solutions = es.ask()

  fitness_list = list(es.popsize)

  for i in range(es.popsize):
    fitness_list[i] = play(solutions[i])

  # Based on the returned fitness
  # prepares a new population.
  es.tell(fitness_list)
```

Listing 2: Weights tuning process using evolution strategy

Computation Time Optimization

For the training the controller, we had to make some optimization and concessions to fasten the process as the Collaboratory resources are highly inconvenient for this type of task[2].

One of the slowing issues we had to address was that first solutions proposed by CMA-ES took incredibly long to finish as the action left or right slowing the speed of the skier, and in some cases, it predicted only one direction, so the skier almost stopped moving. This was solved by shortening the length of one play for solution evaluation to the final number of 500 steps. However after a while, we saw the controller was good at not crashing, but it did not collect any rewards for going through the slalom. The reason for that was that going through the poles was so infrequent, our original reward +10 was dissolved

[2]Training the controller for Car Racing experiment took about three days on 64-core CPU instance with 220GB RAM. Collaboratory provides 2-core CPU and 12GB RAM.

in the punishments for crashes. Consequently, instead of going down, the controller found that slowing an agent to minimal speed will cause minimum crashes, therefore, higher cumulative reward. Due to this fact, we increased the slalom reward to be more recognizable in the final result.

The second concession, which might contribute to unstable results of the best solution, was a reduction of plays per solution to only 3. In the *World Models*, the solution evaluation was averaging of 16 plays. Also, we reduced the population size from 64 to 30.

Training Progress

We trained the controller for 28 generations on the three techniques mentioned earlier and also on their variation, where the hidden state is not used for the prediction. This process took about four days of intermittent training to compute. Even though the computations were run in parallel as for each approach was created a single session.

The decision approaches performance during the generations is plotted in Figure 2.7.

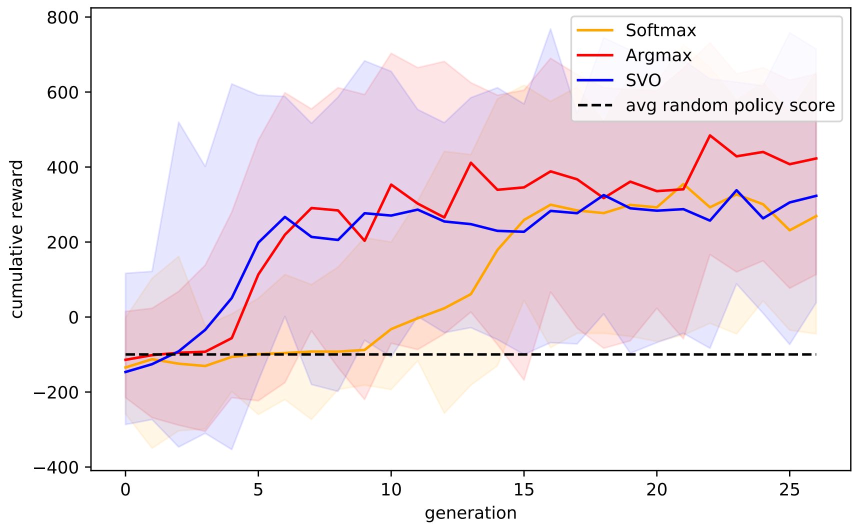


Figure 2.7: Visualization of decision approaches progress through the generations. The thick line represents average generation score and the shadow illustrates the minimal and maximal achieved score.

2.7 All Models Together

The flow diagram of the modules interacting with the environment is illustrated in Figure 2.8.

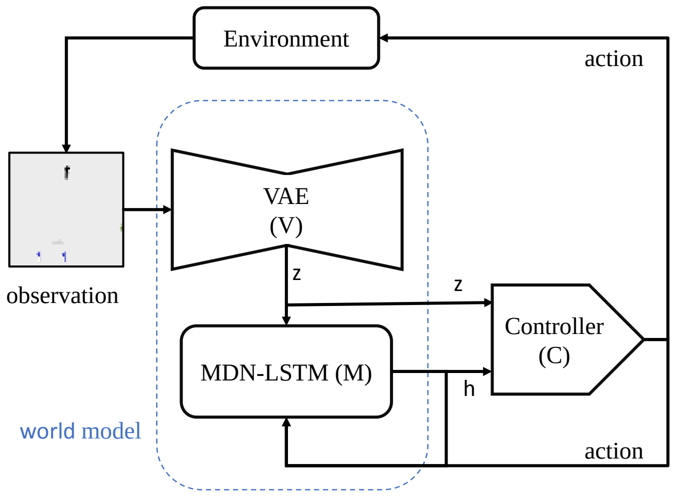


Figure 2.8: Flow diagram of World Models modules. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t. C will then output an action a_t for direction control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time step $t+1$.

Number of individual modules' trainable parameters is shown in Table 2.2.

Model	Parameter Count
VAE	9,254,477
MDN-RNN	12,596,992
Controller	6915

Table 2.2: Modules parameter count

Results

In this chapter, we analyze and compare the performance of various approaches for controller's decision architecture. In the first section, we show results in the environment with modified reward and shortened rollout time. In the second section, we compare the best performing decision approach on the original environment with other known results.

Performance of Various Decision Approaches

For testing the performance, we evaluated each decision approach on the weights proposed as the best by pycma module. For the proof on the advantage of having the predictions made by the memory module we also tested their variants, where the controller does not have access to the MDN-LSTM's hidden state h. In Table 2.3 is reported their average performance over 20 tries.

Decision approach	Score (z and h)	Score (only z)
softmax	276.2	**94.00**
argmax	**399.25**	5.75
SVO	356.75	-280.50
random policy score: -99.75		

Table 2.3: Decision approaches performance evaluation in the environment with modified rewards. The best performance is highlighted in bold.

From the results, we can see that in cases where the model has access to the full world model a deterministic approach is a better choice, however, for cases where the hidden state is not accessible should be applied probabilistic approach. This might be on the fact the world model already learned the right predictions, and the probabilistic action choosing is counterproductive.

On the other hand, it is convenient for unsure predictions from the controller with access only to latent vector z.

Performance in the Original Environment

For the performance comparison of the *World Models* framework we use some results of some other published RL methods applied to the Skiing environment. The results are demonstrated in Table 2.4.

Method	Average Score	Frames Count
A3C LSTM	$-14,863.8$	200M
A3C FF, 1 day	$-13,700.0$	200M
DQN	$-13,062.3$	200M
DDQN	$-9,021.8$	200M
GA	$-5,541.0$	6B
Vision model only, z input	$-15,209.0$	224,000
Full World Model, z and h	$-11,623.5$	224,000
Random policy	$-17,098.1$	
Human	$-4,336.9$	

Table 2.4: Methods performance comparison on the Skiing environment. A3C scores taken from [37]. DQN, DDQN, human and random policy scores are from [38]. GA scores (currently the best known published result) are taken from [39]. Our results (in bold) are average of 20 plays. For Vision model only we used sofmax decision method. For full world model we used argmax decision method.

The *World Models* framework outperformed some of the known methods but is far behind the current best performance. We assume the achieved score might be a bit better if we did not have to make some optimization due to computation resources we had.

However, the main essence of the framework – data augmentation proved to be working well. As for achieving this result, the *World Models* algorithm needed only about 224,000 generated frames which are equal to the number of interactions with the environment whereas A3C's and DQN's methods were trained on 200M frames and GA on 6B frames.

Conclusion

The goal of the thesis was to provide a general survey of data efficient methods for reinforcement learning and apply the model-based data augmentation framework called *World Models* of the recent influential paper by Ha and Schmidhuber to a different OpenAI gym environment. Apart from that, do a performance analysis of the achieved result and consider possible approaches to improve it, especially on the controller part.

Here is described how the mentioned goals were fulfilled, and future work extensions and improvements are considered.

At first, we provided a brief theoretical background needed for understanding the implementation and survey of data efficient methods for reinforcement learning.

In the practical part of the thesis, we went through the implementation of the aforementioned framework on the gym Skiing environment. We also gave experimental advice on optimization of the *World Models* by testing different methods for preprocessing the frames and presenting different decision mechanisms for the controller module. In the performance analysis, we found our results are competitive with recent work but on a significantly smaller computational budget (over 10x improvement).

Although the results of this approach are promising, there is still ample room for improvements both on theoretical and practical aspects of the problem. Some of such improvements include closer investigation on the influence of hyperparameters and compression ration on the performance or testing different optimization algorithms on the controller.

In an extension of the *World Models* paper, an idea for using an iterative training for creating an artificial curiosity and intrinsic motivation was proposed. By flipping the sign of memory module's loss function in the actual environment, the agent will be encouraged to explore parts of the world that it is not familiar with.

The above-mentioned extension and improvements might be further explored in the future Master's thesis.

Bibliography

1. HA; SCHMIDHUBER. Recurrent World Models Facilitate Policy Evolution. In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 2450–2462.
2. DEISENROTH; RASMUSSEN. PILCO: A Model-based and Data-efficient Approach to Policy Search. In: Bellevue, Washington, USA: Omnipress, 2011.
3. FEINBERG; WAN; STOICA; JORDAN; GONZALEZ; LEVINE. Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning. *CoRR*. 2018.
4. Definition – Machine Learning (ML). *SearchentErpriseAI* [online] [visited on 2019-04-14]. Available from: https : / / searchenterpriseai . techtarget.com/definition/machine-learning-ML.
5. AURÉLIEN. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. 2nd New edition. Sebastopol, United States: O'Reilly Media, 2019.
6. The story of AlphaGo. *Deep Mind* [online] [visited on 2019-04-14]. Available from: https://deepmind.com/research/alphago/.
7. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. *Deep Mind* [online] [visited on 2019-04-14]. Available from: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii.
8. Key Concepts in RL. *Open AI – Spinning Up* [online] [visited on 2019-04-14]. Available from: https : / / spinningup . openai . com / en / latest / spinningup/rl_intro.html.
9. MNIH; KAVUKCUOGLU; SILVER; GRAVES; ANTONOGLOU; WIERSTRA; RIEDMILLER. Playing Atari with Deep Reinforcement Learning. *CoRR*. 2013.
11. CYBENKO. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*. 1989, vol. 2, no. 4.

12. Fundamentals of Deep Learning — Activation Functions and When to Use Them? *Analytics Vidhya* [online] [visited on 2019-04-14]. Available from: https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/.
13. SHARMA. Understanding Activation Functions in Neural Networks [online] [visited on 2019-04-14]. Available from: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.
14. NIELSEN, Michael. *"Neural Networks and Deep Learning.* Determination Press, 2015.
15. SHIBUYA. Demystifying Entropy [online] [visited on 2019-04-14]. Available from: https://towardsdatascience.com/demystifying-entropy-f2c3221e2550?gi=e13b75cb121b.
16. How to select the Right Evaluation Metric for Machine Learning Models: Regression Metrics. *Towards Data Science* [online] [visited on 2019-04-14]. Available from: https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0.
17. Gradient Descent in a Nutshell. *Towards Data Science* [online] [visited on 2019-04-14]. Available from: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0.
18. Adam–latest trends in deep learning optimization. *Towards Data Science* [online] [visited on 2019-04-14]. Available from: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.
19. A Comprehensive Guide to Convolutional Neural Networks. *Towards Data Science* [online] [visited on 2019-03-28]. Available from: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.
20. Convolutional Layers Keras Documentation [online] [visited on 2019-04-14]. Available from: https://keras.io/layers/convolutional/.
21. A Beginner's Guide to LSTMs and Recurrent Neural Networks [online] [visited on 2019-04-14]. Available from: https://skymind.ai/wiki/lstm.
22. HOCHREITER; SCHMIDHUBER. Long Short-Term Memory. *Neural Computation.* 1997.
23. Recurrent Neural Networks and LSTM [online] [visited on 2019-04-14]. Available from: https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5.
24. KINGMA; WELLING. *Auto-Encoding Variational Bayes.* 2013.
25. Mixture Density Networks. *Otoro* [online] [visited on 2019-03-28]. Available from: http://blog.otoro.net/2015/06/14/mixture-density-networks/.

26. A Visual Guide to Evolution Strategies. *Otoro* [online] [visited on 2019-03-28]. Available from: http://blog.otoro.net/2017/10/29/visual-evolution-strategies/.
27. *Python info* [online]. Python Software Foundation, 2019 [visited on 2019-04-14]. Available from: https://www.python.org/about/.
28. Programming Languages Most Used and Recommended by Data Scientists. *Business Over Broadway* [online] [visited on 2019-04-14]. Available from: http://businessoverbroadway.com/2019/01/13/programming-languages-most-used-and-recommended-by-data-scientists/.
29. *The Jupyter Notebook* [online]. Project Jupyter, 2019 [visited on 2019-04-14]. Available from: https://jupyter.org/.
30. Welcome to Colaboratory. In: [online]. 2017 [visited on 2019-04-14]. Available from: https://colab.research.google.com/notebooks/welcome.ipynb.
31. Introduction to Tensorflow. *Tensorflow* [online] [visited on 2019-04-02]. Available from: ttps://www.tensorflow.org/guide/low_level_intro.
32. Key features & capabilities. *PyTorch* [online] [visited on 2019-04-02]. Available from: https://pytorch.org/.
33. pycma. In: *GitHub* [online] [visited on 2019-04-27]. Available from: https://github.com/CMA-ES/pycma.
34. Introduction to OpenCV-Python Tutorials. *OpenCV* [online] [visited on 2019-04-02]. Available from: https://docs.opencv.org/3.4/d0/de3/tutorial_py_intro.html.
35. Gym. *OpenAI* [online] [visited on 2019-04-02]. Available from: https://gym.openai.com/.
36. retro-contest-sonic. In: *GitHub* [online] [visited on 2019-04-27]. Available from: https://github.com/dylandjian/retro-contest-sonic.

Appendix **A**

Acronyms

CNN Convolutional neural nets

CMA-ES Covariance-Matrix Adaptation Evolution Strategy

GP Gaussian process

CPU Central processing unit

GPU Graphical processing unit

ES Evolution strategy

LSTM Long Short Term Memory nework

KL Kullback-Leibler divergence

NN (Artificial) neural network

MDN Mixture density network

ML Machine learning

MSE Mean squared error

RAM Random-access memory

RGB Additive color model in which red, green and blue light are added together to reproduce a broad array of colors

RL Reinforcement Learning

RNN Reccurent neural network

VAE Variational autoencoder

Appendix B

Contents of enclosed CD

`readme.txt`	file with CD contents description
`thesis.pdf`	thesis text in PDF format
`thesis_src`	directory of LaTeX source codes of the thesis
`world_models`	directory with implementation