hochschule mannheim

A new Adversarial Approach for Model Generation in both a Supervised and Reinforcement Learning Context

Christian Coenen

Master Thesis

for the acquisition of the academic degree Master of Science (M.Sc.)

Course of Studies: Computer Science

Department of Computer Science

University of Applied Sciences Mannheim

29.01.2021

Tutors

Prof. Jörn Fischer, University of Applied Sciences Mannheim

Prof. Ivo Wolf, University of Applied Sciences Mannheim

Coenen, Christian:
A new Adversarial Approach for Model Generation in both a Supervised and Reinforcement Learning Context / Christian Coenen. –
Master Thesis, Mannheim: University of Applied Sciences Mannheim, 2021. 66 pages.

Coenen, Christian:
Ein neuer Kontroverser Ansatz für die Modellgenerierung in dem Kontext des Überwachten und Bestärkenden Lernens / Christian Coenen. –
Master-Thesis, Mannheim: Hochschule Mannheim, 2021. 66 Seiten.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 29.01.2021

Christian Coenen

Acknowledgement

Throughout the writing of this thesis, I have received a great deal of support and assistance.

I would like to thank my supervisor, Professor Jörn Fischer, whose expertise was invaluable in formulating the research objectives. Your regular insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

In addition, I would like to thank my parents for their unconditional support, not only throughout this thesis but all my studies. I definitely would have had a harder time without it. Thank you for being always there for me.

Abstract

A new Adversarial Approach for Model Generation in both a Supervised and Reinforcement Learning Context

This research presents a new adversarial approach to generate models for two different machine learning problems. A concept is presented that shows how an Autoencoder can be combined with a Generative Adversarial Network to learn to generate new samples based on a given label. The combined network is trained using the MNIST dataset. Furthermore, the concept is extended to address learning a model of a maze environment in a reinforcement learning setup. A prototype is developed and used to conduct the experiments that verify the concept.

The experiments' analysis shows that the model learns to generate accurate new samples that show similarity to training samples of the given label. Experiments analyzing the extended concept show that an accurate model of the maze environment can be generated. The experiments also show the limitations of the concept: The inability to learn the environment's logic and physic. This results in inaccurate predictions for unknown state-action pairs.

Ein neuer Kontroverser Ansatz für die Modellgenerierung in dem Kontext des Überwachten und Bestärkenden Lernens

Diese Arbeit stellt einen neuen kontroversen Ansatz für die Modellgenerierung für zwei unterschiedliche Probleme des Maschinellen Lernens vor. Es wird ein Konzept präsentiert, welches zeigt wie ein Autoencoder mit einem Generative Adversarial Network kombiniert werden kann, um zu erlernen neue Datensätze basierend auf einem gegebenen Label zu generieren. Das kombinierte Netzwerk wird anhand des MNIST Datensatzes trainiert. Darüber hinaus wird das Konzept erweitert, um über das Verfahren des bestärkenden Lernens ein Modell einer Labyrinthumgebung zu erlernen. Um das Konzept zu verifizieren wird ein Prototyp entwickelt, welcher zur Durchführung der Experimente verwendet wird.

Die Analyse der Experimente zeigt, dass das Modell lernt präzise neue Datensätze zu generieren. Diese Datensätze weisen hierbei eine Ähnlichkeit zu den Trainingsdatensätzen des verwendeten Labels auf. Experimente, welche das erweiterte Konzept analysieren, zeigen, dass ein präzises Modell der Labyrinthumge-

bung erzeugt werden kann. Des Weiteren zeigen die Experimente die Grenzen des Konzeptes auf. Dies ist zum Beispiel die Unfähigkeit des Lernens der Logik und Physik der Umgebung, welches zu ungenauen Vorhersagen bei unbekannten Zustand-Aktions Paaren führt.

Contents

Chapter 1

Introduction

This chapter introduces the concept of using adversarial training to create generative models. When speaking about adversarial training, an often-used architecture is the so-called Generative Adversarial Network (GAN). It is introduced by Goodfellow, Pouget-Abadie, Mirza, *et al.* [1] and led to many impressive results like generating realistic-looking images of persons [2] or generating music [3]. Because a GAN can be trained without using label information, it can be used in an unsupervised training setup. This raises the question, whether the label information in a labeled dataset can be incorporated into the training process to generate samples that show similarity to samples of a given label (for example: generate a bold man; generate a woman with glasses). To separate this research from related work, a given sample should also be transformable into a new sample of a given label (for example: transform a bold man into a woman with glasses). Because this requires the input to be of the same dimensionality as the output, this research uses the autoencoder architecture to compress the problem space. Additionally, weight sharing is used between the encoder and decoder to prevent increased complexity and decreased performance compared to the related work. This results in an objective as follows:

> O1: Generating new samples given random inputs and a label, as well as encoding these samples, using the same weights.

The concept and findings to address O1 are then applied to a reinforcement learning setup to train a realistic environment model, which learns accurate predictions for unknown state-action pairs. Inspired by the dreaming process observed in the human brain, it is assumed that an adversarial approach can help learn the logic and physics of an environment. When dreaming, a person acts in his world model, which can create trajectories of observations that are interpreted as real observations

[4]. As shown by Ha and Schmidhuber [5], training a reinforcement learning agent in its environment model can lead to state-of-the-art results.

> O2: Applying the core idea of O1 to a reinforcement learning setup to generate trajectories of realistic new states, not present in the dataset, without using the environment.

This research's remainder is structured as follows: The second chapter provides a detailed introduction to machine learning fundamentals and specific architectures used throughout this research. It is followed by showing the prior conducted work that tries to address problems of similar or related nature to this research's objectives (third chapter). The fourth chapter analyses the objectives and presents a concept to address them, while the fifth chapter presents a prototypical implementation based on the concept. The prototype is then used in the sixth chapter to analyze how different approaches perform concerning the objectives. The information gained through the analysis is synthesized in the seventh chapter. The eighth and last chapter concludes the research and suggests paths that are worthwhile for future research activities.

Chapter 2

Fundamentals

This chapter covers the fundamentals behind the concepts used throughout this research. It starts with multiple sections that provide an overview of neural networks and how they function mathematically before the relevant neural network architectures are introduced. Then, two commonly used programming libraries to create neural networks are described. Last, multiple sections are provided that give the reader an introduction to the relevant topics in the context of reinforcement learning, which is required to address the second objective.

2.1. Artificial Neuron

As described by Krenker, Bester, and Kos [6], an artificial neuron—from now on specified as a neuron—is a mathematical function that receives one or more input(s) and produces one output. It is derived from the biological neuron that is found in all complex forms of life. Figure 2.1 shows how three inputs into an artificial neuron are processed to produce an output. The inputs 'Input 1' and 'Input 2' are multiplied with their respective weights (denoted with x in the figure), whereas the bias input is not multiplied. The three resulting values are then summed before a transfer function—also called activation function—is applied to normalize the output into a specific range. Activation functions are further discussed throughout the following sections.

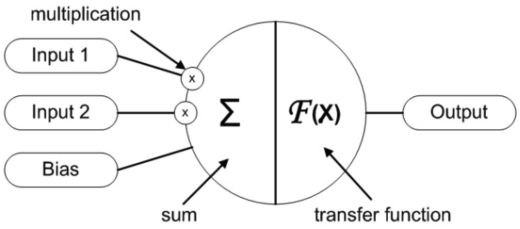

Figure 2.1.: The figure [6] shows how inputs are processed in an artificial neuron to produce an output. The inputs 'Input 1' and 'Input 2' are multiplied with their respective weights (denoted with x), whereas the bias input is not multiplied. The three resulting values are then summed before a transfer function—also called activation function—is applied to normalize the output into a specific range.

2.2. Artificial Neural Networks

As described by Sarle [7], an artificial neural network–from now on specified as a neural network or network–is a mathematical model inspired by the biological neural network. Figure 2.2 visualizes a simple, densely connected neural network consisting of four layers. The first layer is called the input layer and receives an input value for each neuron during training. The following two layers, called hidden layers, are used to increase the non-linearity and, therefore, the network's complexity. The last layer is called the output layer, and its neuron values represent the result of the neural network. Each layer has a specified number of neurons shown as circles in the figure. Between the layers are so-called weights that are used to connect two layers. Additionally, each layer has a unique neuron called the bias neuron. The difference between the bias neuron to a typical neuron is that the bias neuron is not connected to the previous layer.

The following two sections will go through a neural network's training process consisting of a step called forward propagation and a step called backpropagation that follows after the forward propagation step.

2.2.1. Forward Propagation

Forward propagation is a chain of mathematical calculations to transform an input vector into an output value or output vector. The input data has to have the same dimension as neurons present in the neural network's input layer to assign each neuron precisely one input value. The values are then forwarded to the neurons in the following layer $(l + 1)$ by applying the dot product between the neuron values of the current layer $(a^{(l)})$ and the weight matrix $W^{(l)}$ that connects layer l with layer

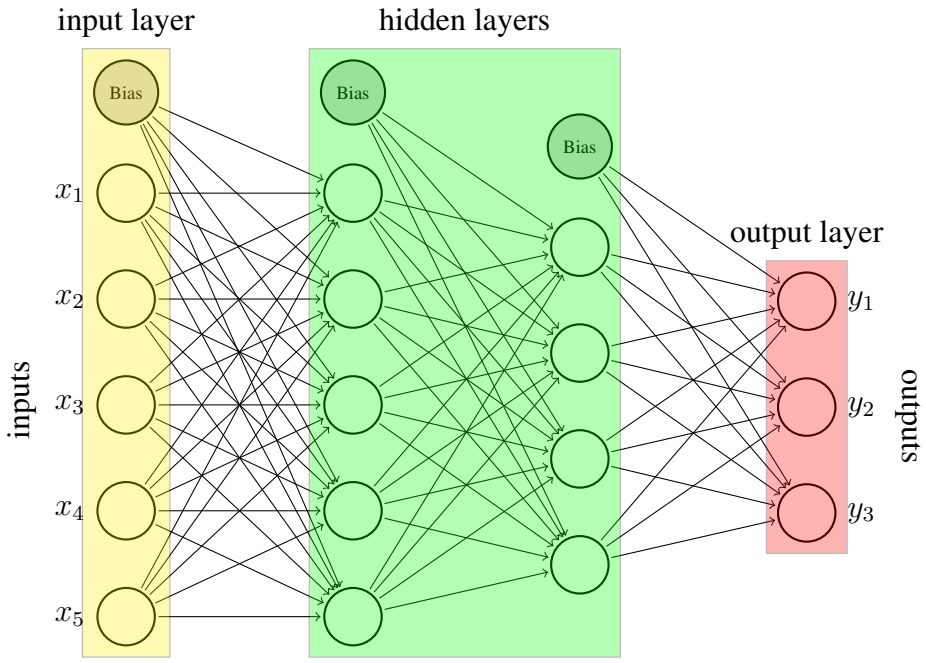

Figure 2.2.: The figure shows a densely connected neural network consisting of four layers.

$l + 1$. After that, the current layer's bias weight vector ($b^{(l)}$) gets added. Because the neuron value of the bias is always set to one, it is not present in the equation. The last step is to apply the result to a so-called activation function (σ), which will be discussed in the next paragraph. This process gets repeated till the output layer (L) is reached. The values in the neurons of the output layer are the results that the neural network predicts.

$$a^{(l+1)} = \sigma(b^{(l)} + a^{(l)} \cdot W^{(l)})$$

The activation function helps the network to learn complex patterns from the input values. It scales the result from the prior calculation into a range based on the function and can also apply non-linearity. A commonly used activation function is the sigmoid function shown in Figure 2.3. It is defined as follows:

$$s = \frac{1}{1 + e^{-x}}$$

The sigmoid function is just an example of an activation function that can be chosen. There are many activation functions suited for different machine learning problems, like the Hyperbolic Tangent (tanh) function and the Rectified Linear Unit (ReLU)

function [8], to name a few. Choosing the right activation function for a specific problem can differentiate between successful and non-successful network training.

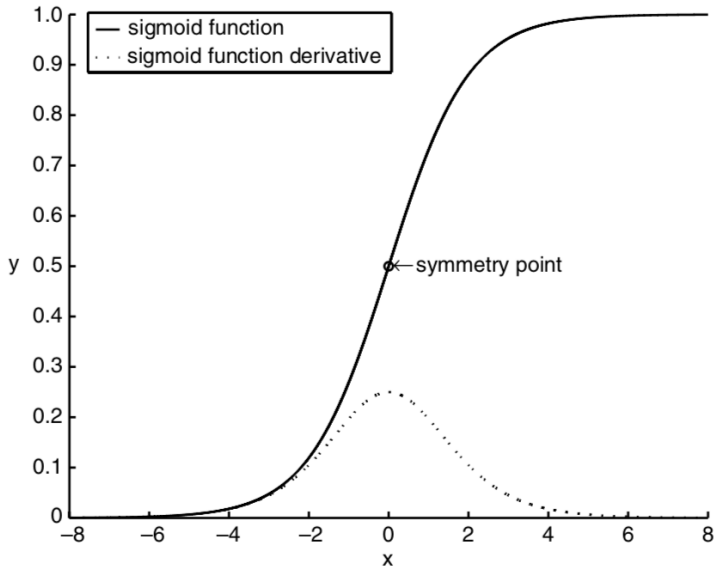

Figure 2.3.: The figure [9] shows the sigmoid function. It is a non-linear function that normalizes its input into a range of $(0, 1)$. Also shown is the function's derivative $\left[\frac{dy}{dx} = y * (1 - y)\right]$ that is needed for the backpropagation algorithm.

2.2.2. Error Calculation

In a supervised setup, the neural network's output needs to be rated, which is achieved by applying an error function. The error function compares the output with the label. A label can be thought of as the ground truth created for the input. The greater the difference between output and label, the greater the error value. As with the activation function, there are different error functions for various machine learning problems. Choosing the right error function for a specific problem can differentiate between successful and non-successful network training. A commonly used error function for regression problems is the Mean Squared Error (MSE). It calculates the difference between the output (Y) and corresponding label $(\hat{Y})$ and squares the result to avoid negative values. This procedure is repeated for n samples. The sum of squared errors is then multiplied by $\frac{1}{n}$ to receive the mean of the squared errors. Mathematically, it is defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

2.2.3. Backpropagation

Backpropagation is the standard algorithm used to train neural networks. First presented as 'reverse mode of automatic differentiation' by Linnainmaa [10], it is based on minimizing errors through gradient descent. Ten years later, it finds its first application in neural networks [11].

The idea is to propagate the error value back to each weight in the network based on its participation. The backpropagation is done by calculating derivatives with respect to the weights to receive the error gradients (δ) for a layer. Once all error gradients in a network are calculated, the weights are updated in the negative direction of their gradient (gradient descent). The following sections describe the mathematical steps that are required to apply the backpropagation algorithm.

Error Gradient Calculation (Output Layer)

The error gradients for the output neurons $(\delta^{(L)})$ can be determined by calculating the difference between the label (y) and the neuron values in the output layer $(a^{(L)})$.

$$\delta^{(L)} = y - a^{(L)}$$

Error Gradient Calculation (Hidden Layer(s))

The error gradients for the hidden neurons can be determined by calculating the derivative of the activation function used in the specific hidden layer during the forward propagation. If—for example—sigmoid is used as error function $(g(z))$, the derivative would look as follows ($\odot$ = element-wise multiplication):

$$g'(z) = g(z) \odot (1 - g(z))$$

With $g'(z)$, the error gradients for the neurons in layer l $(\delta^{(l)})$ can be calculated by first applying the dot product between the transposed weights $(W_{\mathsf{T}}^{(l)})$ and the error gradients of the neurons in layer $l + 1$ $(\delta^{(l+1)})$ and then taking the element-wise

product between the result and $g'^{(l)}(z^{(l)})$. The term $z^{(l)}$ refers to the neurons' output in layer l before an activation function is applied.

$$\delta^{(l)} = W_{\intercal}^{(l)} \cdot \delta^{(l+1)} \odot g'^{(l)}(z^{(l)})$$

Calculating the Jacobian Matrices

With the error gradients, the jacobian matrices $J_w^{(l)}$ and $J_b^{(l)}$–required to update the weights–can be created by using the following formulas:

$$J_w^{(l)} = \delta^{(l)} \cdot a_{\intercal}^{(l-1)}$$

$$J_b^{(l)} = \sum \delta^{(l)}$$

$J_w^{(l)}$ is a matrix of the same size as $W^{(l-1)}$ (the weights connecting layer $l-1$ with layer l) and contains the gradient for each weight in the weight matrix $W^{(l-1)}$.

$J_b^{(l)}$ is a vector of the same size as $b^{(l-1)}$ (the bias weights from layer $l-1$ to layer l) and contains the gradient for each weight in the weight vector $b^{(l-1)}$.

Updating The Weights In All Layers

The jacobian matrices represent the gradients for each incoming weight in the layer they represent and have to be multiplied by α (a hyperparameter in the range $0 < \alpha \leq 1$ that defines how strong the weights are updated) and negated (to descent instead of ascent the gradient).

$$W^{(l-1)} = W^{(l-1)} - \alpha * J_w^{(l)}$$

$$b^{(l-1)} = b^{(l-1)} - \alpha * J_b^{(l)}$$

2.2.4. Autoencoder

Described by Bank, Koenigstein, and Giryes [12] and shown in Figure 2.4, an Autoencoder (AE) is a network architecture consisting of an encoder and decoder network. Both networks are connected through the latent space, representing the

neurons' values in the layer between the encoder and decoder. If the latent space dimension is smaller than the input dimension, the layer can also be referred to as the bottleneck layer. While the encoder's task is to map the input into the latent space while losing the least amount of information, the decoder's task is to reconstruct the encoder's input—given the latent space values—with minimal loss. Commonly used loss functions for AEs focus on the reconstruction error between input and output for each value separately (for example, the MSE). While this can be the right choice for different data types, it can lead to blurriness when using image data. This problem can be addressed using loss functions based on the similarity score, like the Multi-Scale Structural-Similarity Score (MS-SSIM) [13], which reduces blurriness and improves the quality of reconstructed images.

An AE can also generate outputs by removing the encoder network and feeding the decoder with randomly generated latent space values. Because the latent space values are directly mapped to the decoder function without using a probability distribution, an AE is not defined as generative model [14].

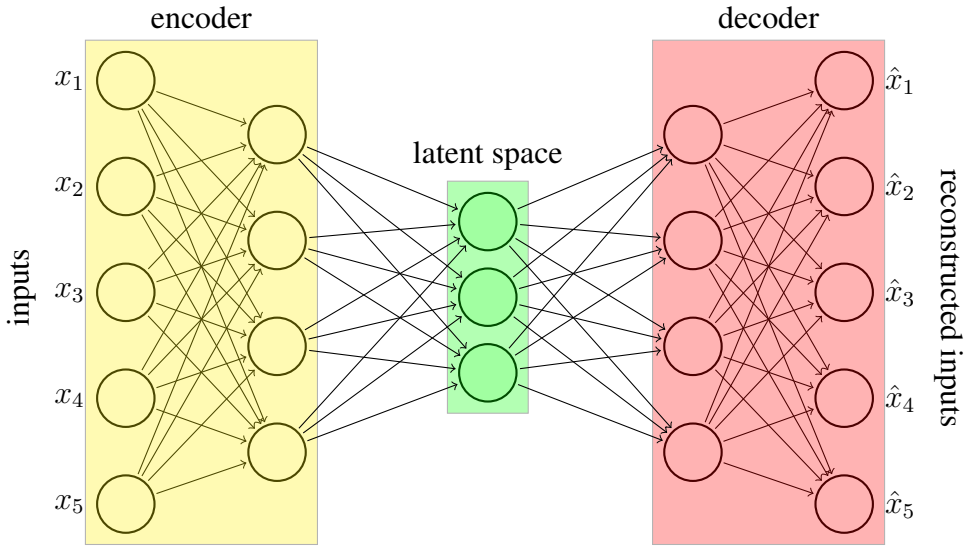

Figure 2.4.: The figure shows a typical Autoencoder architecture. As seen on the left, a five-dimensional input gets fed into an encoder network. The output of the last encoder layer serves as input to the latent space layer. That is the layer between the encoder and decoder (often also the layer with the fewest neurons). The latent space values are then fed as input into the decoder (having the encoder's reversed architecture) to reconstruct the original input as accurately as possible.

2.2.5. Weight Sharing

The idea of weight sharing is to use the same weights for different layers in a neural network. The idea is first presented for neural networks with the introduction of Restricted Boltzmann Machines (RBMs) by Hinton [15]. As seen in Figure 2.5 (left), the idea behind an RBM is to encode an input through a bottleneck layer and then reconstruct the input by passing it through the same layers backward. This approach enforces the use of shared weights between the encoding and reconstruction (decoding) step. An AE, as seen in Figure 2.5 (right), can be thought of as the unfolded version of an RBM that additionally allows the use of different weights between the encoder and decoder layers. Using shared weights, however, results in a 50% decrease in trainable weights, which yields an increased training speed and decreased risk of overfitting due to better generalization [16]. Because a linear AE with shared weights is equivalent to Principal Component Analysis (PCA) [17], shared weights can result in a better geometrical encoding of the input. The better geometrical encoding occurs because the decoder only performs a back-transformation from the encoded point, leaving the encoder with the task to precisely map the inputs into the low-dimensional space. The mentioned advantages come at the price of less trainable parameters, which means that the model cannot learn a function as complex as when not using shared weights.

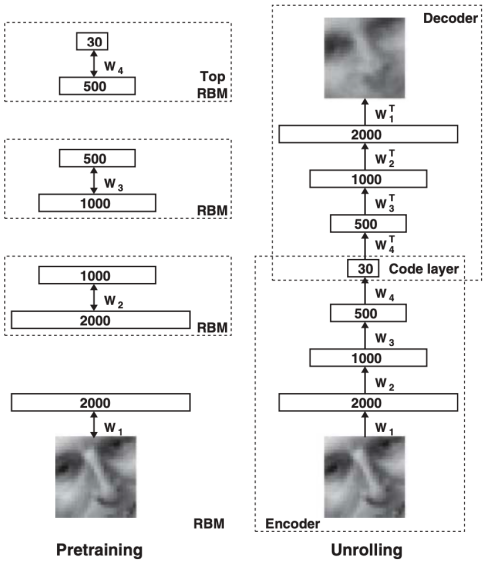

Figure 2.5.: The figure [15] shows a Restricted Boltzmann Machine used to pretrain weights (left) as well as an Autoencoder using the pretrained weights (right). While the weights trained by the Restricted Boltzmann Machine are shared between the encoding and decoding process, they can be different when using an Autoencoder. In the figure, the weights trained through the Restricted Boltzmann Machine are used to initialize the Autoencoder. Therefore, the weights of the Autoencoder are also shared at initialization.

2.3. Generative Models

This section goes through two commonly used architectures to create models with generative capabilities. As defined by Ng and Jordan [14], a generative model generates new samples from noise by either using a probability distribution or a discriminative network.

2.3.1. Variational Autoencoder

Introduced by Kingma and Welling [18] and seen in Figure 2.6, a Variational Autoencoder (VAE) is an AE that maps the inputs to a normal distribution instead of a fixed vector. This difference results in the latent space consisting of two different vectors. The first represents the mean of the distribution, and the second represents the standard deviation of the distribution. The decoder can then be used to reconstruct encoded images–or generate new ones–by sampling from the distribution. This allows the creation of new samples similar to the samples from the dataset on which the VAE is trained.

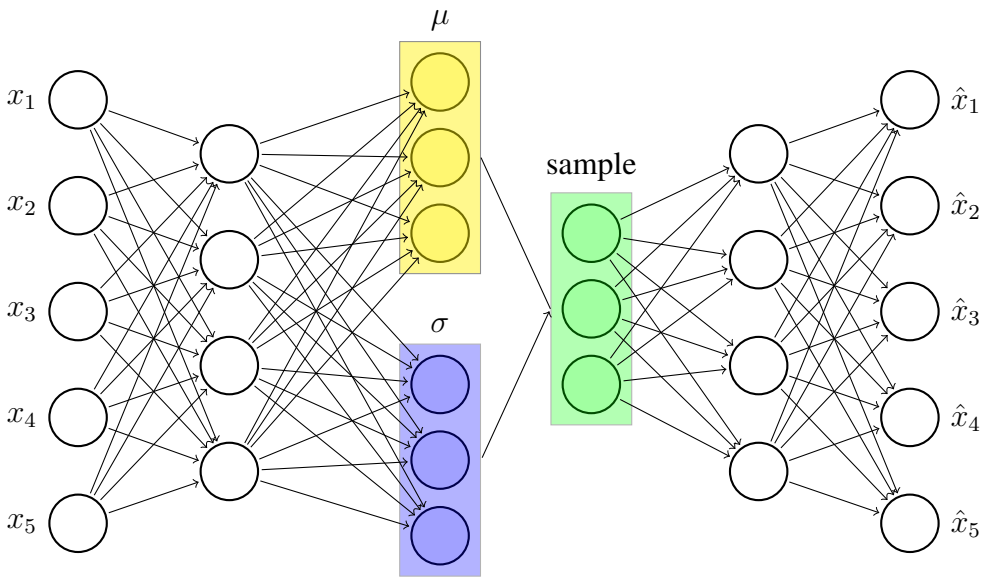

Figure 2.6.: The figure shows the architecture of a Variational Autoencoder. The differences towards a traditional Autoencoder are colored. That is, two layers (highlighted in yellow and blue) that are connected to the last encoder layer and one layer (highlighted in green) that is connected to the yellow and blue layer. All colored layers need to have the same amount of neurons because each neuron in the green sample layer (n_i) will sample from a normal distribution using μ_i and σ_i.

2.3.2. Generative Adversarial Network

The idea behind a GAN [1] is to train two networks with contrary goals. As visualized in Figure 2.7, the generator network aims to create new samples that are as indistinguishable as possible from the dataset. The discriminator network needs to learn to distinguish real samples from fake samples as accurately as possible. These opposed goals result in an adversarial training process where both networks can theoretically improve themselves to an optimum against each other. It has been shown that GANs can be trained to create new samples of similar quality to the dataset.

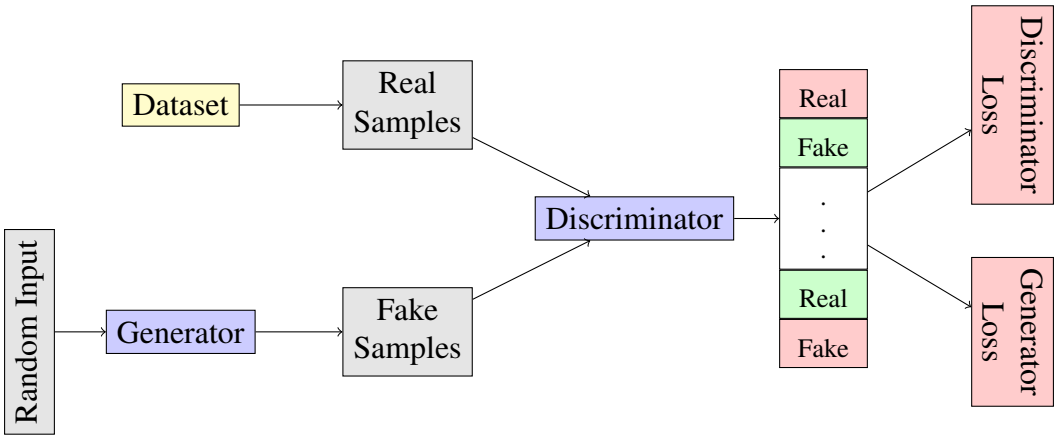

Figure 2.7.: The figure shows the architecture of a GAN. The generator network generates samples from random input, which are labeled as 'fake'. Additionally, 'real' labeled samples are drawn from the dataset. Both 'real' and 'fake' samples are fed into the discriminator network. By classifying the images as 'real' or 'fake', the discriminator and generator are receiving a loss value based on the discriminator's performance. If it is good, the discriminator receives a low loss value while the generator receives a high loss value and vice versa.

2.4. Tensorflow

Introduced by Abadi, Agarwal, Barham, *et al.*, Tensorflow [19] is an open-source platform for machine learning. It is available for multiple programming languages, including Python. With Tensorflow, a user can implement computational graphs—also called dataflow graphs—that can then be used on different platforms ranging from mobile devices to distributed large-scale training systems. Figure 2.8 visualizes a dataflow graph that can be used to perform forward propagation. The gradient calculation needed for many machine learning algorithms is also supported and can be automatically performed by Tensorflow through a method called 'gradients'.

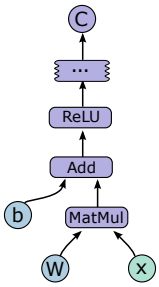

Figure 2.8.: The figure [19] shows the visualization of a dataflow graph as it can be created in Tensorflow to perform forward propagation. The graph consists of nodes (rectangles), edges (arrows), and variables (circles), with a node representing a mathematical operation that can receive variables and other operations through edges.

2.5. Keras

Introduced by Chollet *et al.*, Keras [20] is a deep-learning API written in Python that uses Tensorflow to create the underlying dataflow graphs. With Tensorflow 2.0, Keras became the official high-level API and is since then integrated into Tensorflow. Keras supports different layer architectures—for example, dense layers, convolutional layers, max-pooling layers—that can be combined to create machine learning architectures. Listing 2.1 and Figure 2.9 illustrate how a neural network consisting of an input layer, one hidden layer, and an output layer can be created using the Keras high-level API.

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(100, activation="relu", name="hidden_layer",
                                input_shape=(784,)))
model.add(tf.keras.layers.Dense(10, activation="relu", name="output_layer"))
model.build()
```

Listing 2.1: Python code to create a neural network model using Keras. A new layer can be added by calling the 'add' function and passing the required parameters like the number of neurons.

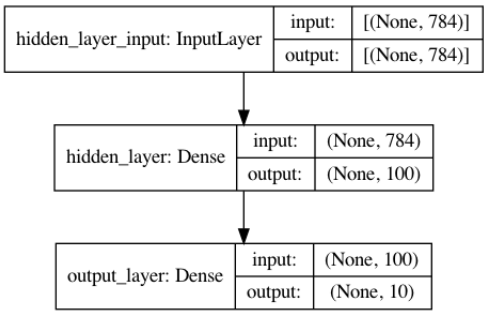

Figure 2.9.: The figure shows the architecture that gets created when running the code defined in Listing 2.1. The first dimension is set to 'None' because it will be set according to the defined batch size before training.

2.6. Reinforcement Learning

As described by Sutton and Barto [21] and visualized in Figure 2.10, Reinforcement Learning (RL) addresses the problem of self-learning without labeled data by letting an agent take an available action (A_t) in an environment state (S_t) to collect observations of the next state (S_{t+1}) and a reward signal (R_{t+1}). This cycle—mathematically defined as Markov Decision Process (MDP)—is repeated until a termination condition is met. Over time, the agent learns to prefer actions that yield a high reward. In most RL problems, maximizing the reward is the main objective for an agent and, therefore, a critical part of the training process.

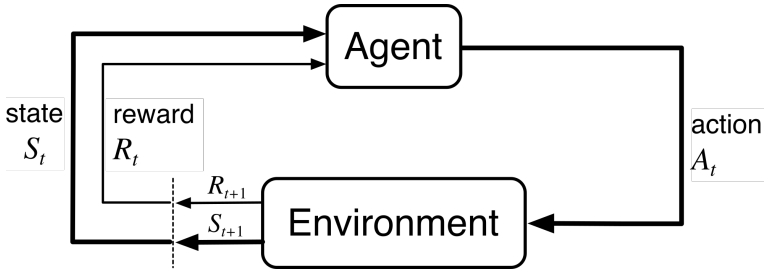

Figure 2.10.: The figure [21] shows a Markov Decision Process, which is common in reinforcement learning problems. An agent behaves in an environment by taking possible actions. An action triggers an update of the environment resulting in a new state and reward for the agent. The reward $(R \in \mathbb{R})$ is determined by the environment depending on the agent's new state.

2.6.1. Q-Learning

Q-learning [22] is an algorithm used to approximate the optimal action-value function for MDPs and is introduced by Watkins and Dayan in 1992. This algorithm is one of the early breakthroughs in RL [21]. It is defined as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * \Big[R_{t+1} + \gamma * \underset{a}{max} Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

At timestep $t+1$, the algorithm updates the Q-value for the state-action pair $Q(S_t, A_t)$ by taking the highest valued Q-value in the current state $(\underset{a}{max} Q(S_{t+1}, a))$, multiplying it with a discount factor (γ), subtracting the Q-value that is to be updated $(Q(S_t, A_t))$, adding the received reward (R_{t+1}) and multiplying the result by the learning rate (α). An essential element is that the reward is propagated back to prior states based on γ. A Q-value can, therefore, be thought of as the future estimated reward. To summarize, the algorithm updates its Q-values with respect to both the current and future estimated reward weighted through γ.

2.6.2. Model-Free and Model-Based

In terms of RL, a model-free approach trains an agent by using an algorithm to learn the Q-values for possible state-action pairs. When speaking about a model-based approach, the setup is extended to include a model of the environment. Sutton and Barto describe a model as follows:

> "By a model of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. [21]"

This definition means that a model-based approach can—if trained sufficiently—use its model to predict possible outcomes from a given state. Further, a model might infer accurate environment transitions for currently unseen states by combining previously learned characteristics of the environment.

2.6.3. Gym

Gym [23] is an RL toolkit developed by OpenAI and designed for researchers. It is written in Python and consists of a collection of environments that share a defined interface. These environments serve as a baseline to analyze how new algorithms compare to existing ones or how different algorithms perform in a new environment. The toolkit can also be used to register new environments and has no requirement for how the environment is created as long as it has implemented the defined interface. Figure 2.11 shows a few environments that are part of the gym collection.

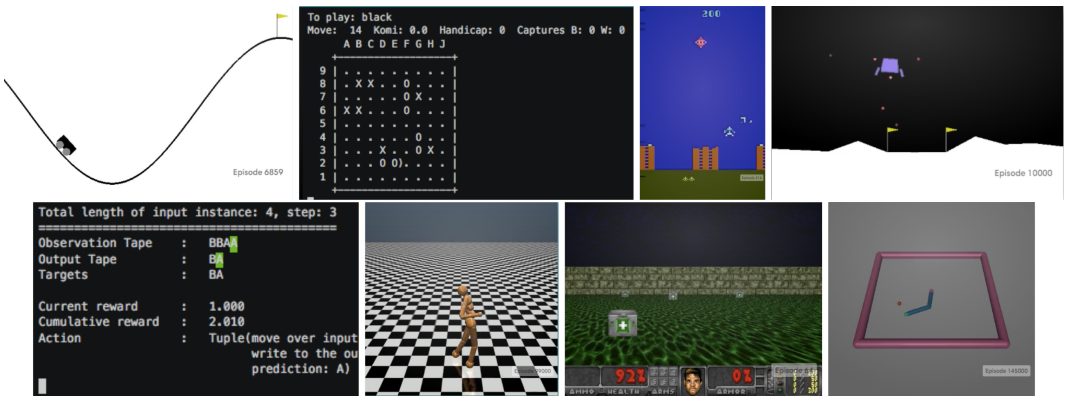

Figure 2.11.: The figure [23] shows some of the environments that can be used through *gym* by default. Researchers use those environments to compare different RL algorithms.

Chapter 3

Related Work

Machine Learning has become a popular research topic with the rise of computational power throughout the last decades. Therefore, it is expected that prior work related to the objectives of this research has been conducted. This chapter summarizes the related work for each objective.

3.1. Objective 1

This section describes work that is related to O1. It is divided into subsections, which means that the related work is described independently of each other. The objective is shown in the following for faster reference:

> Generating new samples given random inputs and a label, as well as encoding these samples, using the same weights.

3.1.1. Variational Autoencoder-Generative Adversarial Network

A VAE-GAN [24] is a combination of a VAE and a GAN network. The idea is to replace the often-used element-wise error functions like the mean-squared-error with a more complex error representation like a similarity metric learned by another neural network. As seen in Figure 3.1, this led to the VAE-GAN combination where the decoder in a VAE is used as a generator for the GAN while the discriminator network in the GAN is used as an error representation for the samples produced by the generator/decoder.

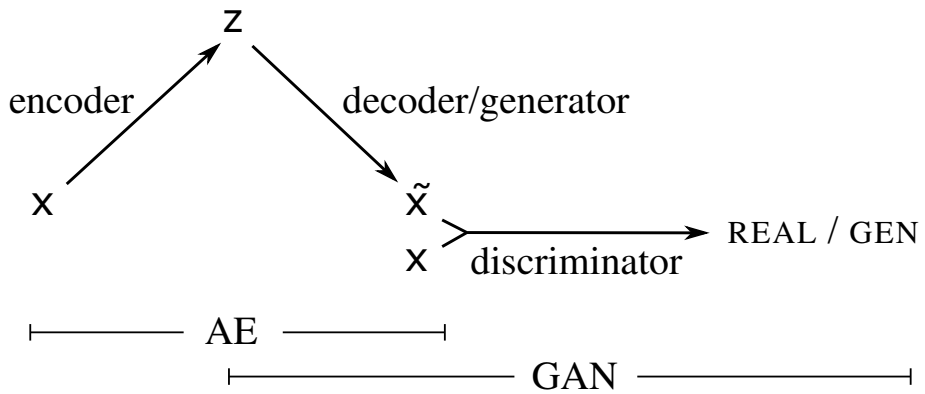

Figure 3.1.: The figure [24] shows the VAE-GAN architecture consisting of a VAE and GAN. The architecture uses the same network as a decoder and generator to decode and produce new samples $\tilde{x}$. The discriminator learns to distinguish real samples (x) from generated samples ($\tilde{x}$) and, therefore, train the generator.

3.1.2. Supervised Adversarial Autoencoder

In 2016, Makhzani, Shlens, Jaitly, *et al.* [25] introduced different approaches to using adversarial AEs. The architecture of an adversarial AE is nearly identical with the VAE-GAN architecture shown in Figure 3.1. The only difference is that an AE is used instead of a VAE. In their work, Makhzani, Shlens, Jaitly, *et al.* extend the adversarial AE to a supervised learning setup by adding the label as one-hot encoded vector y to the hidden code z. The latent space, therefore, not only consists of z but also y. Figure 3.2 shows that this leads to disentangled representations between the number (represented by y) and its style (represented by z).

Figure 3.2.: The figure [25] shows MNIST samples generated by a Supervised Adversarial Autoencoder. Each row activates each value in y once while using the same randomly drawn z values. It can be seen that training an adversarial Autoencoder in a supervised setup makes it possible to create a specific digit by setting the corresponding value in y. Additionally, the style of the digit can be changed by varying the values of z.

3.1.3. Semi-Supervised Generative Adversarial Network

Compared to the Supervised Adversarial AE, which contains label information in its latent space, the Semi-Supervised GAN (SGAN) [26] trains the discriminator to classify its input between $n + 1$ classes. To give an example, one assumes a dataset with the labels 0 to 9. Then SGAN will learn to classify samples into 0 to 9, plus an additional class labeled 'fake'. Using this architecture, Odena shows that it is possible to simultaneously train a generative model and a classifier, which results in both an improved classifier (compared to an isolated classifier) (Table 3.1) as well as an improved generator (compared to just using a GAN) (Figure 3.3).

Classifier Accuracy

EXAMPLES	CNN	SGAN
1000	0.965	0.964
100	0.895	0.928
50	0.859	0.883
25	0.750	0.802

Table 3.1.: The table [26] shows an accuracy comparison between SGAN and a Convolutional Neural Network (CNN) using different sizes of MNIST training examples.

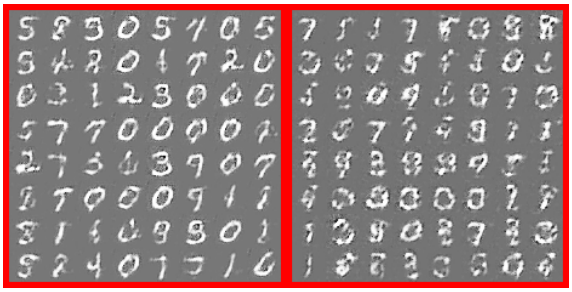

Figure 3.3.: The figure [26] shows generated samples after 2 MNIST epochs. Samples generated by SGAN are on the left half, while samples generated by the GAN are on the right half. It can be seen that an SGAN can generate clearer images compared to a GAN.

3.1.4. Conditional Generative Adversarial Network

The Conditional GAN (CGAN) [27] adds label information in both the generator and discriminator. The process of adding one-hot encoded label information to the generator is the same as done in a Supervised Adversarial AE. However, Mirza and Osindero extend this idea by additionally adding the one-hot encoded label information as input to the discriminator as visualized in Figure 3.4. Training this

architecture can lead to generated samples based on a condition (e.g., create a sample that gets classified as 5) as seen in Figure 3.5, where each row is conditioned on one label, and each column shows a different generated sample. The results, however, are outperformed by other approaches as stated by Mirza and Osindero:

> "The conditional adversarial net results that we present are comparable with some other network based, but are outperformed by several other approaches – including non-conditional adversarial nets. We present these results more as a proof-of-concept than as demonstration of efficacy, and believe that with further exploration of hyper-parameter space and architecture that the conditional model should match or exceed the non-conditional results. [27]"

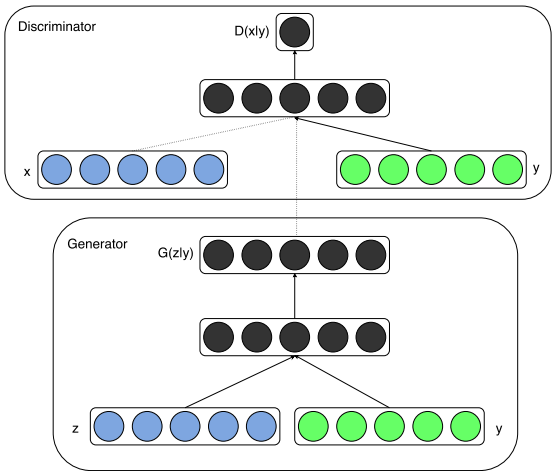

Figure 3.4.: The figure [27] shows the CGAN architecture consisting of a generator and a discriminator, both receiving a one-hot encoded label vector y. The architecture allows the generator to learn to generate samples based on the state of the label vector.

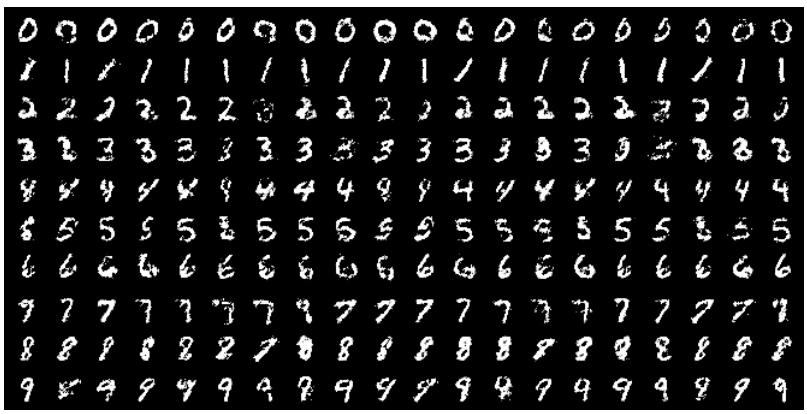

Figure 3.5.: The figure [27] shows generated samples by the generator in the CGAN. Each row is conditioned on one label, and each column shows a different generated sample.

3.1.5. Auxiliary Classifier Generative Adversarial Network

With the Auxiliary Classifier GAN (AC-GAN) [28], Odena, Olah, and Shlens present a similar approach to the CGAN. However, instead of using the label information as input to the discriminator, they task the discriminator to predict the label information alongside the classical 'real' or 'fake' prediction (Figure 3.6 (c)). This approach leads to disentangled latent space representations between the hidden space z and the label information c. They further show that this architecture can better discriminate images by artificially resizing them through the generator.

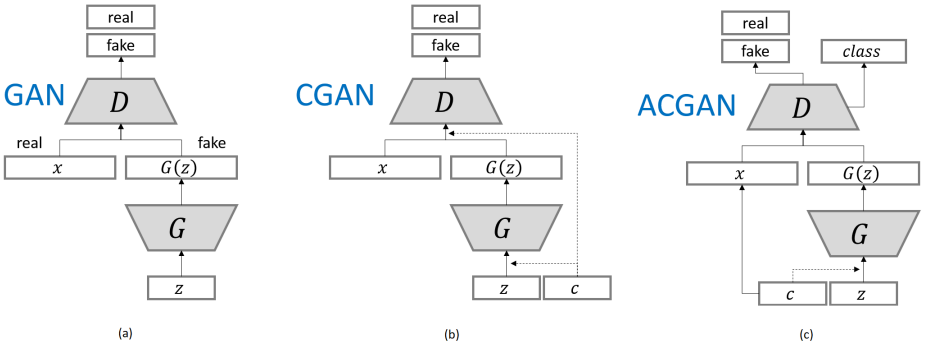

Figure 3.6.: The figure [29] shows the GAN (a), CGAN (b), and AC-GAN (c) architecture. It can be seen that the CGAN introduces the idea of encoding the label in both the generator and discriminator, while the AC-GAN changes the label from being an input into the discriminator to using it as a second objective for the discriminator.

3.2. Objective 2

This section describes work that is related to O2. The objective is shown in the following for faster reference:

> Applying the core idea of O1 to a reinforcement learning setup to generate trajectories of realistic new states, not present in the dataset, without using the environment.

3.2.1. World Models

In 2018, Ha and Schmidhuber [5] published a paper about a new approach to train an RL agent. As partly shown in Figure 3.7, the idea is to generate random samples from the environment, use those samples to train a VAE (denoted with V) which

learns to represent the input as distribution in its latent space, use the latent space to train a Recurrent Neural Network (RNN) (denoted with M) and then use both the latent space and hidden representations of the RNN as inputs for an agent (denoted with C) that evolves using an evolution strategy.

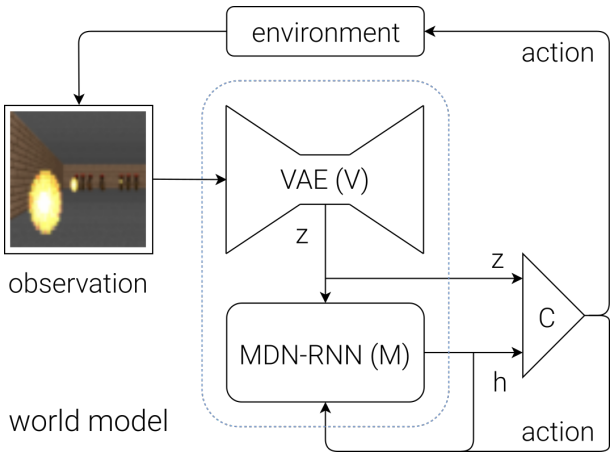

Figure 3.7.: The figure [5] shows the architecture to train a reinforcement learning agent through outputs from a Variational Autoencoder and a recurrent neural network. The approach led to state-of-the-art results in the field and is the first known to solve the CarRacing-v0 gym environment.

This approach is the first to train an agent that solves the CarRacing-v0 gym environment by obtaining an average score above 900. It also achieves the highest average score (1092) to date on the DoomTakeCover-v0 gym environment by only training the agent from trajectories generated by the environment's model (M).

One noticeable limitation of this approach is the training of V and M, which has to occur before training C by using the random policy. Training V and M requires an environment that can be thoroughly explored using the random policy.

Chapter 4

Concept

This chapter shows a theoretical approach to achieve the objectives which are—as defined in Chapter 1—shown in the following for faster reference:

> O1: Generating new samples given random inputs and a label, as well as encoding these samples, using the same weights.
>
> O2: Applying the core idea of O1 to a reinforcement learning setup to generate trajectories of realistic new states, not present in the dataset, without using the environment.

4.1. Analysis of the Objectives

O1 requires the use of a generative model to be able to generate new samples. As explained in Section 2.3, there are two types of models with generative capabilities—VAEs and GANs. Without modification, however, both models are not able to receive any label information. Additionally, incorporating the received label into the training process requires the model to perform a supervised classification task that is not present in a standard VAE and GAN. A thorough search of the relevant literature yielded one related article for VAEs and four related articles for GANs. Summaries about those model architectures are written in Chapter 3. Because a VAE-GAN can be seen as an extension of a regular GAN network by additionally using an encoder and a sampling layer, it is assumed that all GAN-related model architectures presented in Chapter 3 can be applied to the VAE-GAN model architecture.

The objective also states that the generative network should encode given samples using the same weights. This requirement can be fulfilled by using an AE-GAN architecture where the generator is used as a decoder together with the encoder to represent the AE and as a generator together with the discriminator to represent the GAN. Because the decoder model architecture is always a mirror of the encoder model architecture, all weights can be shared between both models using the transposed weight matrices for the decoder. A VAE can not be used because weight sharing is not possible due to asymmetric weights between the encoder and decoder at the sampling layers. The asymmetric weights can be seen when inspecting Figure 2.6.

To satisfy O2, O1 has to be applied to a reinforcement learning setup. Assuming that the same core principles can be applied, a few adjustments have to be made.

First, training in a reinforcement learning setup requires an environment with the following properties [30]:

1. returning a reward r for each possible state s
2. performing a state transition and returning the next state $s + 1$ for any given state s
3. for terminating environments:
   3.1. informing the agent if an end state s_e is reached
   3.2. resetting itself if an end state s_e is reached

Second, the agent interacting with the environment needs a model attached, representing the expected cumulative reward for a given state-action pair [30].

Third, the encoder and discriminator receive at least two different input arrays due to the use of state-action pairs. The support of multiple input arrays should be considered, especially for implementations, to avoid code refactoring later on.

4.2. Aspects to be Considered

Multiple aspects need to be considered to ensure successful training when trying to solve problems using neural networks:

Learning Rate The learning rate is a hyperparameter that determines the strength to which the gradients are applied to the weights. As stated by Buduma and

Locascio [31], picking the correct learning rate is a significant- and, at the same time, hard problem to solve. Setting the learning rate too low can result in a slow convergence towards the local or global minimum, while setting it too high might prevent the algorithm from reaching a minimum at all. Even with a carefully adjusted learning rate, multiple convergence problems can occur. Two methods used to stabilize and accelerate the training with respect to the learning rate are: Using a momentum term [32] and adaptive learning rates [33], [34]. A learning rate optimizer that is often used to manage and update the learning rates during training automatically is Adam [35]. It uses both a momentum term as well as adaptive learning rates.

Activation Functions As explained in Section 2.2.1, the activation function helps the network to learn intricate patterns from the input values. It also normalizes the neuron values into a specific range, which can be especially helpful or even required for the output layer.

Layer Types A layer type determines how incoming weights are processed. There are different types of layers with different strengths for specific problems. While a fully-connected layer is the most used and best-known layer type, it might be insufficient for complex tasks like image segmentation or language translation. Even though model architectures solving those tasks can contain fully-connected layers, they often require additional layer types like convolutional layers [36] or recurrent layers [11].

While the previous aspects apply for nearly all problems where neural networks are used, the following aspects are specifically important to consider when training with a discriminator network in an adversarial approach:

Dropout Introduced by Srivastava, Hinton, Krizhevsky, *et al.*, dropout [37] is a technique for randomly deactivating neurons in the network during training to prevent overfitting. According to [1], it is one reason for the striking successes behind the results produced by GANs. Therefore, dropout is often used in conjunction with the discriminator model training in a GAN.

Label Smoothing Introduced by Salimans, Goodfellow, Zaremba, *et al.*, label smoothing [38] is another technique that can improve the stability when training a GAN. The idea is to smooth the positive labels (e.g., labels annotating a real sample) with a fixed constant (c). To give an example: If c is 0.1, the label would be smoothed from 1.0 to 0.9 for all positive samples. This tech-

nique can prevent the discriminator from outperforming the generator due to the weight adjustments that occur even for correct predictions due to the smoothed labels.

4.3. Resulting Concept

This section discusses the resulting chapter based on the objectives and the previous sections of this chapter. The concept is split into two sections, one for each objective. The concept for O1 lies the foundation for the concept of O2, which can be seen as a modification of O1.

4.3.1. Objective 1

Because in machine learning, the architecture of a neural network is often dependent on the dataset, it should be the first decision. This research tries to address and achieve O1 using the MNIST dataset, which consists of 60000 labeled grayscale images of size 28x28 that show handwritten single-digit numbers. The dataset is used for the following reasons:

1. It is a well-known dataset in the research community
2. It has a relatively low complexity, which is useful for verifying new architectures
3. The results can be visualized and interpreted
4. It is a labeled dataset which is required when using a classifier

As discussed in the analysis of the objective, the architecture requires both an AE and a GAN. While this defines the general architecture that is used, many specific decisions have to be made in order to ensure a stable and successful training process that leads to achieving the objective. While some decisions require experimentation and trial and error, others can be defined based on best practices and findings from prior work. Those decisions—made for both the AE and GAN—are discussed in the following sections:

AE Architecture Decisions

The first architecture decision is how the classifier is added to the AE. When looking at the objective, it can be seen that new samples should be generated from both a given random input and specified classification value. If the classifier is added to the decoder output layer, a specified value can not be used as input for the generator (which is also the decoder). Therefore, the classifier has to be added to the latent space layer. It is a common practice to architect the classifier so that it works with one-hot encoded labels. Applied for the MNIST dataset, the classifier consists of 10 binary output neurons.

Second, the AE consists of Dense layers. While using convolutional layers might result in improved sample quality, it makes the architecture more complicated and harder to analyze. Therefore convolutional layers are not used in this concept but can be seen as a considerable extension.

Third, the decoder output is activated using a sigmoid function and uses a binary cross-entropy loss function. In contrast, the classifier output is activated using a softmax function and uses a categorical cross-entropy loss function. Cross-entropy loss functions are used because they use a logarithmic term that cancels out the exponent in the sigmoid and softmax function. Goodfellow, Bengio, and Courville [39] state that using a loss function without a logarithmic term can lead to loss-saturation, which could prevent gradient-based learning algorithms from making progress. More sophisticated loss functions like SSIM and MS-SSIM are not considered because the MNIST dataset consists of 28x28 images, which are too small and would result in blurred output images, as stated by Snell, Ridgeway, Liao, *et al.* [13].

Fourth, Leaky ReLU, the recommended activation function for neural networks [39], is used as an activation function for the hidden layers. It adds sparsity—when randomly initializing the weights around 0—and a reduced likelihood of vanishing gradients in deep neural networks due to its derivative being either 0 or 1. A problem, however, is the dying ReLU problem that can occur if too many neurons are activated to 0. It results in gradients of zero which in turn prevents any learning. The leaky ReLU function addresses the problem by introducing an α constant by which the input (I) is multiplied if $I < 0$. No research has been found that analyzes and suggests an optimal α value. Choosing this value is left to experimentation.

Fifth, Adam is used as learning rate optimizer with the default settings[1] proposed by Kingma and Ba [35]. Schaul, Antonoglou, and Silver [40] have shown that adaptive learning algorithms—which Adam is one of—are not as strongly affected by varying hyperparameters as non-adaptive algorithms are. It can be interpreted as prioritizing an adaptive learning algorithm to achieve more robustness. When it comes to the choice of which adaptive algorithm to use, there is currently to consent as it "... depends largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning)", stated by Goodfellow, Bengio, and Courville [39].

Other values that affect the training process, like the layers' sizes, the training epochs, and batch size—to name a few—have to be finetuned. Therefore, they are not specified in the concept and should be explored when performing the experiments.

GAN Architecture Decisions

Because the decoder from the AE is used as a generator, decisions in this section must only be made for the discriminator and combined GAN model.

As for the AE, the first decision is about the classifier. Experiments have to show whether adding a classifier to the discriminator helps in generating accurate samples. If used, the classifier must be added to the discriminator output layer to provide an additional loss to the generator.

Second, the discriminator also consists of Dense layers. It is assumed that making the discriminator architecture similar to the generator stabilizes the training. Therefore—except for dropout layers—only hyperparameter changes are used to prevent the discriminator from outperforming the generator.

Third, the discriminator output is activated using a sigmoid function and uses a binary cross-entropy loss function, while—if added—the classifier output is activated using a softmax function and uses a categorical loss function. It is the same as for the generator and should further stabilize the training.

Fourth, Leaky ReLU is used as an activation function for the hidden layers. As mentioned before, architecting the discriminator similar to the generator should stabilize the training.

[1] $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

Fifth, Adam is used as a learning rate optimizer[2]. Opposed to the AE, a smaller learning rate (α) and first exponential decay rate (β_1) are used to configure Adam[3]. Radford, Metz, and Chintala [41] describe that using this configuration helps them stabilize the training of a Deep Convolutional GAN (DCGAN). Even though the GAN defined in this section consists of Dense layers, it is assumed to help stabilize the training. It should, however, been seen as a recommendation that can be changed when experiencing otherwise.

Sixth, dropout layers and label smoothing should be introduced to the discriminator if the training is unstable. Unstable training often occurs when the discriminator is too strong and can be observed when the generator has a loss value that is several factors higher than the discriminator's loss.

As for the AE, other values that affect the training process have to be finetuned. Therefore they are not specified in the concept and should be explored when performing the experiments.

Architecture Conclusion

The previous sections specify a new architecture to address O1. While the resulting architecture consists of known architectures like an AE and GAN, as well as known concepts like weight sharing, it combines them in a unique way that is not found throughout related research. Therefore, the described architecture is defined as Shared Adversarial Classification Autoencoder (SA-CAE) throughout the remainder of this research.

Training

Most neural networks are trained by iterating through the dataset in batches for a defined number of epochs. While this also applies to training a GAN, its step execution differs. As described by Goodfellow, Pouget-Abadie, Mirza, *et al.* [1], each training step in a GAN consists of two training phases that are mentioned in the following:

[2]It should be noted that both the loss function and training optimizer can theoretically differ between the discriminator and GAN. Even though the GAN contains the discriminator, it is an independent model. The previously defined loss function and training optimizer, however, are used for both the discriminator and GAN

[3]$\alpha = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999, \epsilon = 10^{-8}$

1. Training the discriminator weights by using a batch of data that consists of randomly drawn samples from the dataset (labeled as 1) and samples generated by the generator (labeled as 0), with each of them making up half of the defined batch size.
2. Training the generator weights using a batch of samples generated by the generator that is discriminated by the discriminator. The discriminator weights are deactivated during this step.

The AE weights are updated after the discriminator update by using the same randomly drawn samples that are used to update the discriminator. Therefore, the AE is trained on half of the batch size compared to the discriminator / GAN training.

The following enumeration summarizes the training for further clarification:

1. The discriminator model is trained using its model, which has the same loss function and learning rate optimizer as the GAN. A training batch consists of 50% randomly drawn samples from the training dataset labeled with the value 1 and 50% samples generated by the generator using random inputs labeled with the value 0.
2. The AE model consists of the encoder and decoder model and is trained using the AE model with its defined loss function and learning rate optimizer. It is trained using the same randomly drawn samples from the training dataset that are used to train the discriminator model.
3. The generator model (which is also the decoder model) is trained using the GAN model, which has the same loss function and learning rate optimizer as the discriminator. It is trained using random inputs. The GAN model consists of the discriminator model and generator model. The discriminator is not trained during this step.

4.3.2. Objective 2

The following sections describe a setup to address the second objective. If a decision is not mentioned in the concept for O2, it is unchanged from the decision made for O1.

Environment

This section describes the environment used by an agent to gather the data required for training a model that addresses O2. The environment has to satisfy the properties mentioned in the analysis of the objective to be usable in a reinforcement learning setup. A maze environment that satisfies the properties is defined in the following. It is used for the same reasons that MNIST is used to address O1 (except the fourth reason).

1. The maze is block-based, meaning that a 3x3 maze consists of 9 blocks
2. The maze can be stepped block by block in the four cardinal directions
3. The maze consists of two types of blocks:

   Valid blocks that the agent can move to

   Invalid blocks (e.g., walls) that the agent can not move to
4. The size and structure is not determined and can be varied
5. It can be chosen whether the agent starts in the upper left corner or at a random valid position that is not the goal position
6. The goal is always in the lower right corner
7. The agent gets a negative reward of -0.04 for each valid step that results in not reaching the goal
8. The agent gets a positive reward of 1 for a step that results in reaching the goal
9. The agent stays on its current block when trying to move to an invalid block and gets a negative reward of -0.6
10. The environment informs the agent if the goal is reached and resets itself by placing the agent back to its start position
11. The environment can reset itself after a specified number of defined steps

To act in an environment, an agent needs a state representation of the environment at each timestep (S_t). The usage of a block-based maze with blocks that are either occupied by the agent or not leads to a binary block representation. Blocks that the agent is unable to reach, however, would always be zero and are therefore not included in the state representation. Figure 4.1 shows an example of a 3x3 environment with two walls and an agent occupying the upper left block. If an agent acts in

this environment, it receives an environment state of seven values. The blocks not occupied by the agent are valued as zero; the block occupied by the agent is valued as one. Assuming that the figure shows the environment state at timestep one (S_1) and that the agent moves one step to the right to observe the state representation (S_2), the state representations would look as follows:

$$S_1 = (1, 0, 0, 0, 0, 0, 0), S_2 = (0, 1, 0, 0, 0, 0, 0)$$

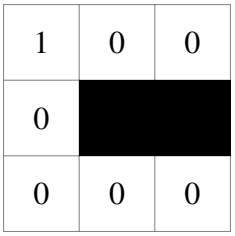

Figure 4.1.: The figure shows a 3x3 block-based maze environment with two walls and an agent occupying the upper left block. A block that can not be occupied is shown in black. It has no value and is not present in the state representation. A block's value is either zero—when it is not occupied by the agent—or one—when the agent occupies it.

Agent

This section describes the agent used to gather the data required for training a model that addresses O2. The agent's task is to act in an environment based on its state. Because the objective is to create a realistic model of the environment, the agent itself is kept simple with a linear Q-learning model that gets the environment state as input and outputs four Q-values—one for each possible action. Because each Q-value is associated with one action, the highest value is used to determine the action that gets executed to update the environment accordingly. The network weights are initialized to zero to ensure that the Q-values for all possible states are zero at the start of the training. While this prevents learning with loss functions like the MSE and cross-entropy, it does not prevent learning when using the Q-learning loss because of the reward term in the formula definition. Figure 4.2 shows the linear neural network representing the agent if the maze shown in Figure 4.1 is used.

It can be seen that the neural network has no bias neurons and no bias weights. The reason for that visualized in a video linked in Appendix A and explained in the following: Assuming the lower right block in Figure 4.1 is the goal position. As soon as the agent reaches the goal, it receives a positive reward associated with the action

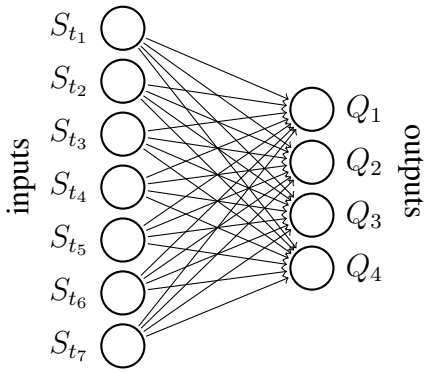

Figure 4.2.: The figure shows a densely connected linear neural network consisting of two layers. The neural network has neither a bias neuron nor bias weights.

that moves the agent to the right. When the maze resets, the agent is placed back to the block in the upper left corner with a preference of going right, which results in going to the upper right corner. When thinking about the problem of solving a maze, having a preferred direction seems counterproductive. Therefore, the neural network representing the agent does not use a bias neuron and bias weights in this concept.

Modified SA-CAE Architecture

This section describes the required changes to the SA-CAE architecture to address O2.

The objective states that the trajectories of realistic new states should be generated. Using the same approach used for O1 would result in generating state-action pairs given a random input observation and next state. The next state, however, is subject to be generated and is therefore not given, meaning to address the objective, the encoder has to receive a state-action pair that is not present in the dataset and outputs the expected next state and reward. Except for compressing the input, there is no need to use an AE when there is no additional information gain (e.g., using a classifier for O1).

Because compressed inputs are not required for a binary maze environment due to their already low complexity, the AE used for O1 is replaced by a regular feedforward neural network that is referred to as generator. It receives two inputs—the state (S_t) and action (A_t) and produces two outputs–the predicted next state (S_{t+1}) and predicted reward (R_{t+1}). The activation function for the first output is the softmax

function with the binary cross-entropy as the loss function. The softmax function is used because the next state should be only one neuron that gets activated. Softmax encourages this by normalizing the output so that the sum equals one. The activation function for the second output is the tanh function with the MSE as the loss function. The tanh function is used because it normalizes the input into the range (-1, 1), which suits the rewards defined for the maze environment. Figure 4.3 shows the generator's resulting architecture when the maze environment shown in Figure 4.1 is used. The number and size of the hidden layers are not defined and can be varied.

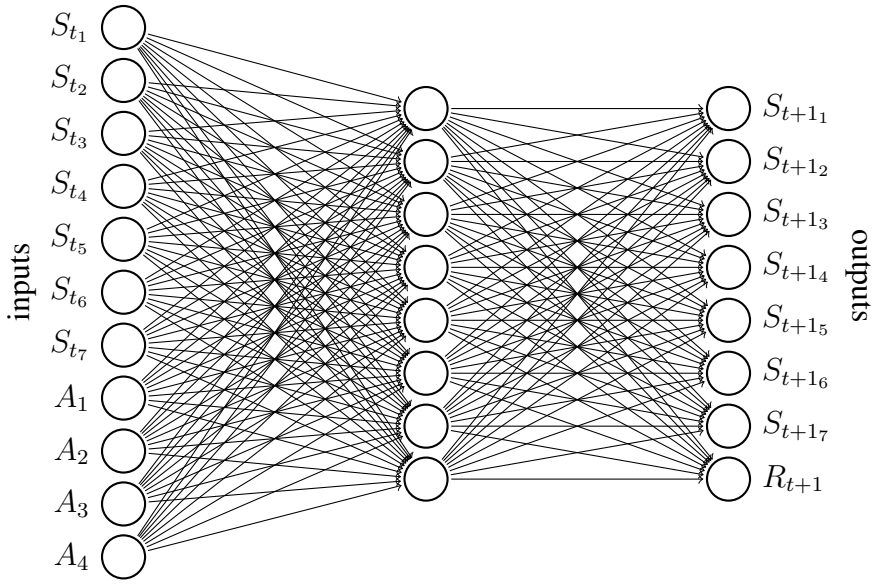

Figure 4.3.: The figure shows a densely connected neural network consisting of three layers. It is a valid network architecture for the generator according to the concept when the maze shown in Figure 4.1 is used as the environment.

The discriminator defined in O1 can be used with two modifications. First, to discriminate whether a new state and received reward are realistic, it needs to know the prior state and action. Both are available because they are used as input for the generator. Therefore, the discriminator input consists of S_t, A_t, S_{t+1}, R_{t+1}. Second, the classification output is removed because, as described in the previous paragraph, the next state is generated by the generator and, in turn, not available as a classification label. Therefore, the discriminator only predicts whether its inputs are real or fake (rof). Figure 4.4 shows the resulting architecture for the discriminator when the maze environment shown in Figure 4.1 is used. As for the generator, the number and size of the hidden layers are not defined and can be varied.

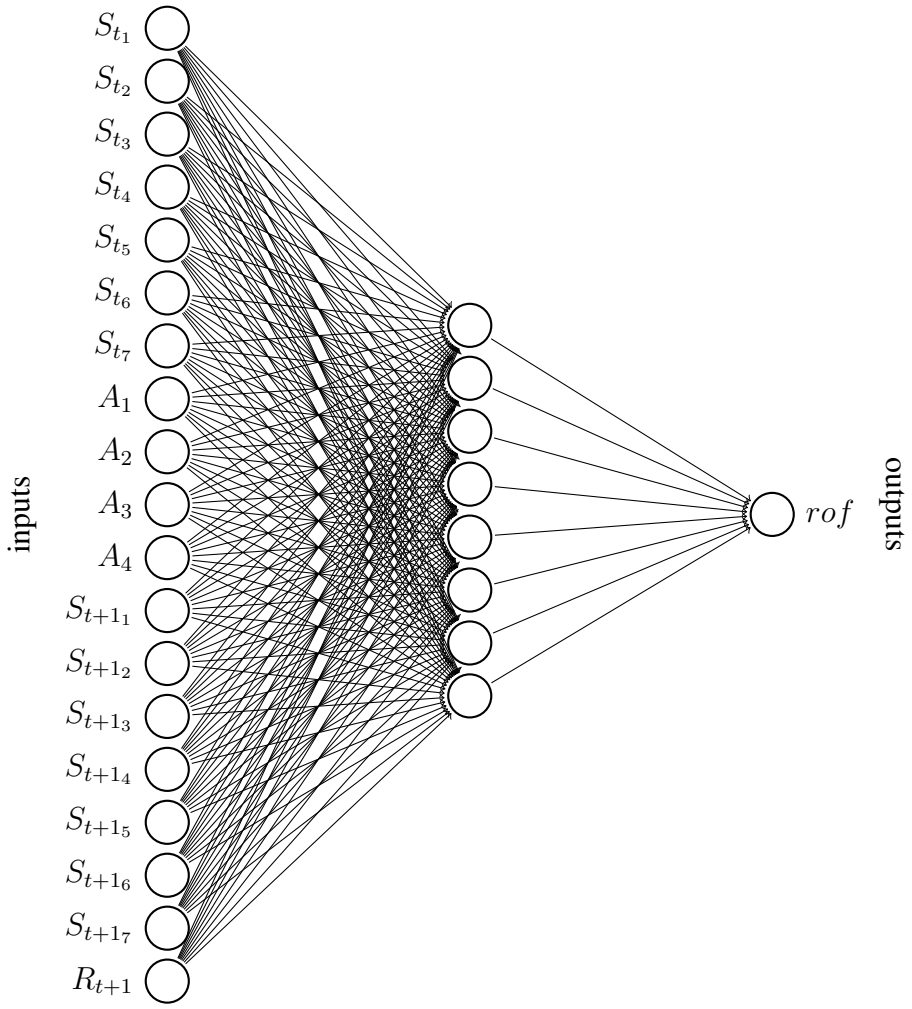

Figure 4.4.: The figure shows a densely connected neural network consisting of three layers. It is a valid network architecture for the discriminator according to the concept when the maze shown in Figure 4.1 is used as the environment.

Architecture Conclusion

The previous sections specify a new architecture to address O2. While it is inspired by the previously defined SA-CAE architecture, it has significant differences like multiple inputs and the usage of a normal feedforward network instead of an AE . The described architecture to address O2 is defined as World-Model-GAN throughout the remainder of this research.

Training

This section describes the differences in the training between the World-Model-GAN and the SA-CAE.

The first difference is that the data needs to be generated manually. To do this, the agent acts in the environment for a manually chosen duration to gather samples. A sample consists of the following information:

1. The state S_t
2. The performed action A_t in S_t
3. The next state S_{t+1}
4. The reward R_{t+1}
5. An information about whether the goal is reached or not

To ensure a balanced dataset, each encountered state-action pair (S_t, A_t) is only present once in the dataset to prevent a model during training from preferring specific state-action pairs because they are more dominant in the dataset.

A change to the model is that, instead of updating the AE weights (encoder and generator) after the discriminator update in a supervised setup, only the generator weights are updated. It is required because of the change from an AE to a regular feedforward neural network.

Another change is the training itself. It is left for experimentation whether the generator needs to be trained on the training dataset in a supervised setup or learn the model only through regular GAN training. If supervised training is performed, one has to experiment with whether all possible state-action pairs or only pairs present in the training dataset are used as inputs when training the generator through the GAN.

Chapter 5

Prototype

This chapter discusses a prototypical implementation of the SA-CAE and World-Model-GAN. The implementation is used to conduct the experiments in the following chapter. Because—intentionally—the concept does not specify all required parameters, the prototype is implemented in a generic way that allows modifying those parameters without the need to rewrite the underlying logic.

5.1. Architecture

As seen in Figure 5.1, the architecture of the prototype consists of multiple classes. The root class is named *Network* and contains the necessary logic to create the networks defined in the concept. To allow weight sharing, it uses the class *DenseTranspose* to create layers that are using references to given weight matrices. Based on *Network* are the two classes *NetworkSupervised* and *NetworkRL*. The former is used to create an SA-CAE, while the latter is used to create a World-Model-GAN. Both classes are designed to be customizable regarding their hyperparameters and model architecture. The customizability is necessary to test and compare different network styles in the experiments chapter. It can also be seen that both classes have a data parameter. While *NetworkSupervised* can use publicly available datasets like the MNIST database [42], *NetworkRL* relies on data generated by the *Trainer* class, which trains the Q-values of an agent in an environment through reinforcement learning. It also stores the trajectories encountered during the training. After training, the stored trajectories are saved to a file that can then be passed to *NetworkRL* during initialization.

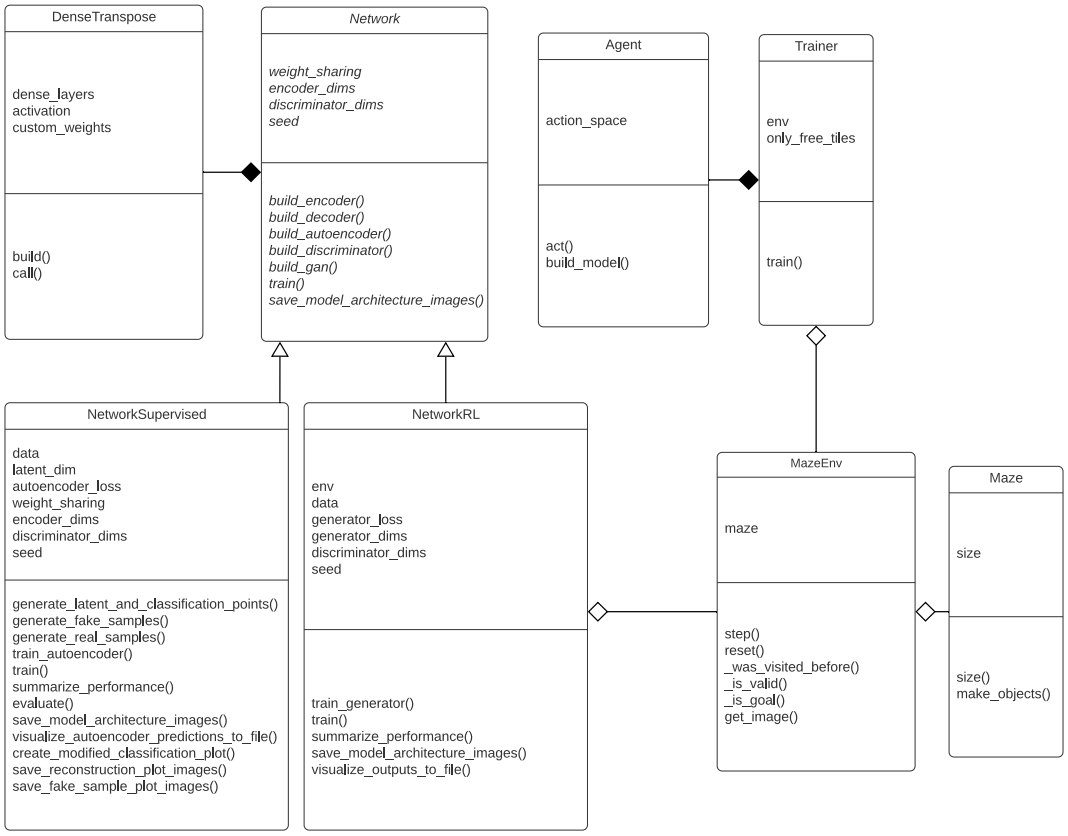

Figure 5.1.: The figure shows a class diagram for the prototype. It can be seen that different machine learning problems like supervised learning and reinforcement learning can be addressed by creating a class that inherits from the *Network* class. The reinforcement learning class *NetworkRL* relies on a maze environment for which a class named *Trainer* can train a reinforcement learner that is present in the class named *Agent*. *Trainer* is also used to store and save trajectories to a file, which can then be used to train a World-Model-GAN with the class *NetworkRL*.

5.2. Implementation

The prototype is implemented using the programming language Python. It relies on the Keras library (which is part of Tensorflow) to create the neural networks. *Trainer* requires an environment that is built as gym environment. Because *MazeEnv* inherits from *BaseEnv*, which itself inherits from *ENV* in the gym library, it can be built and used as gym environment. The prototype uses Black as code formatter to ensure a consistent code style and Poetry for dependency management. In the following, all implementation-wise topics that are worth mentioning are discussed.

5.2.1. Network Building Methods

Network consists of multiple building methods. The methods are designed to generalize enough so that they can be called from both a supervised and reinforcement learning setup. Additionally, each building method is similar with respect to its parameters and output. A method either relies on one or multiple models that need to be passed or a list of input tensors and output layers. The latter can be seen in Listing 5.1, which shows the signature for the method named *build_encoder*. Methods that combine multiple models into a new model have an additional parameter that defines which output layers from the passed models should be ignored in the combined model. A building method always returns a model object with its name set accordingly. It is also possible to change the name to a custom name through a parameter available for all building methods.

```
def build_encoder(
    self,
    input_tensor: List[Tensor],
    output_layers: List[Layer]
    model_name: str = "encoder"
) -> Model
```

Listing 5.1: The listing shows the signature of the build_encoder method in *Network*. It can be seen that the method has four parameters. The first is a reference to itself, resulting in the method's requirement to be called from an instance of *Network* or a class that inherits *Network*. The second requires a list of tensor objects defining the input tensors for the model created in the method. The third requires a list of layers that will be used as output layers in the model. The last one is not required and can be used to overwrite the default name for the returned model.

5.2.2. DenseTranspose Class

DenseTranspose is a self-written class that inherits from a class named *Layer* in the Keras library. It is used for all hidden decoder layers when the weight sharing variable in *Network* is set to the boolean value *true*. Figure 5.2 shows how neuron weights are shared when an AE is used with *DenseTranspose* layers. It can be seen that the neuron weights are mirrored onto the decoder. The mirroring occurs by using references that are then transposed to perform calculations. Therefore, the decoder does not copy the architecture since this would result in new weight connections. When creating a *DenseTranspose* layer, either an object of the Keras class *layers.Dense*, or an object of the TensorFlow class *Variable*, has to be passed to *DenseTranspose*. In both cases, the references to the passed object's neuron weights will be used to calculate the activations in the decoder layers. Because the bias sizes

differ between the encoder and decoder, bias weights are not shared. Instead, new bias weights are created for the decoder layers according to the following layer's size. The abundance of shared bias weights can also be seen in Figure 5.2, where the bias weights are asymmetric with sizes of four, three, four, and five.

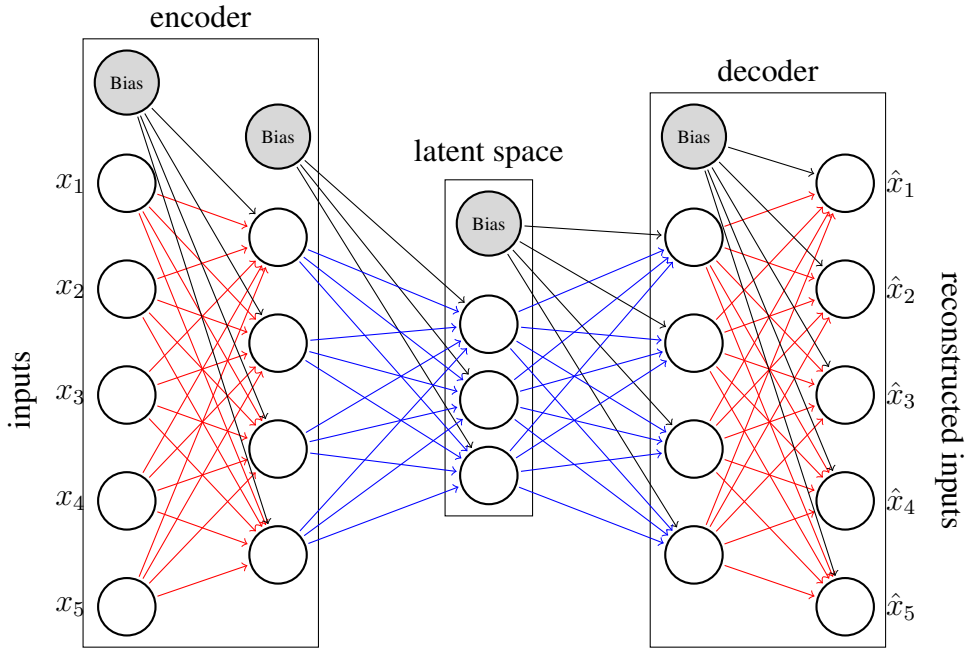

Figure 5.2.: The figure shows a typical Autoencoder architecture with shared neuron weights and unique bias weights. The neuron weights connecting the encoder input layer to the first hidden encoder layer are highlighted in red. In contrast, the neuron weights connecting the first hidden encoder layer with the latent space layer are highlighted in blue. It can be seen that the decoder mirrors the encoder's architecture and neuron weights. The mirroring occurs by using references that are then transposed to perform calculations. Therefore, the decoder does not copy the architecture since this would result in new weight connections. It can also be seen that it is impossible to share the bias weights using this architecture due to their asymmetric occurrence.

5.2.3. Visualizations

Visualizations are an essential part of understanding complex problems and can help to find and solve problems, especially if there is no similar way to express the underlying information [43]. This subsection will go through the different visualization techniques that are implemented in the prototype.

Model Architecture

Model architecture images can be saved as PNG images and SVG graphics by calling the *save_model_architecture_images* method that is implemented in *Network* and extended in both *NetworkSupervised* and *NetworkRL*. The images are created through a Keras function named *plot_model*. Figure 5.3 shows the architecture image for an AE that gets created when calling *plot_model*. However, the image has been adjusted after creation to reflect its underlying model implementation correctly. The adjustment is necessary due to a bug in *plot_model*, which might be addressed in a future version[1].

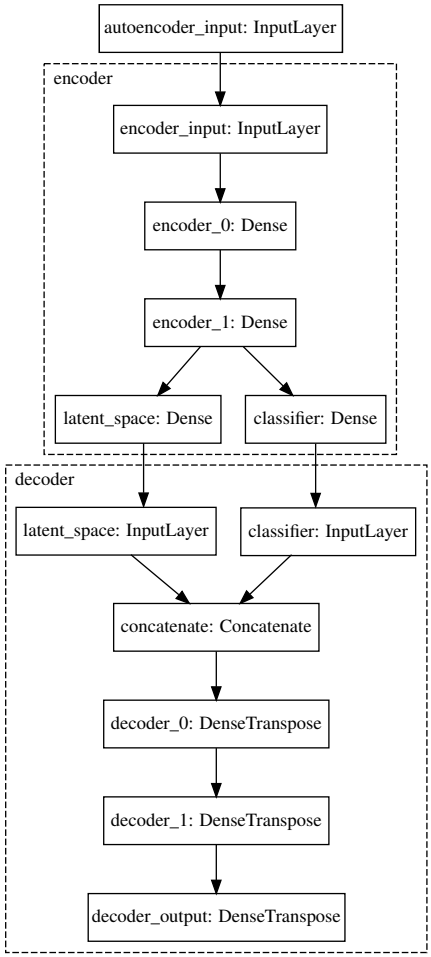

Figure 5.3.: The figure shows the architecture of an Autoencoder that can both reconstruct as well as classify inputs. It can be seen that the decoder uses *DenseTranspose* layers, which means that the decoder uses the same weights as the encoder. The graphic is created by calling the *plot_model* function in the Keras library given a model object.

[1] For more information, see https://github.com/tensorflow/tensorflow/issues/42101

AE Reconstructions

To analyze the reconstructions of a trained AE, *NetworkSupervised* implements a method named *save_reconstruction_plot_images* that works with two- and three-dimensional inputs. The method creates and saves an image that shows both reconstructions and the original input for a list of samples. Because the AE can classify the input, its predictions are also shown in the image. The reconstructions and classifications can be seen in Figure 5.4, where an AE is trained using the MNIST dataset. The upper image represents the original input, while the lower image represents the reconstruction. Shown between both images is the classification that is made by the AE.

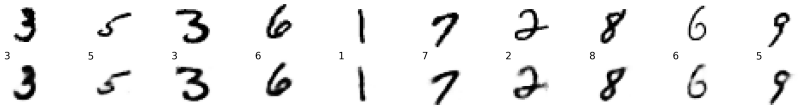

Figure 5.4.: The figure shows reconstructions and classifications for samples from the MNIST dataset using a trained Autoencoder. The upper number represents the original input, while the lower number represents the reconstruction. Between both images is the classification made by the Autoencoder. It can be seen that an Autoencoder can be used to both accurately reconstruct and classify MNIST samples.

Generated Samples

To analyze the training of a GAN, *NetworkSupervised* implements a method named *save_fake_sample_plot_images*. Given a number of samples to create, it generates random noise and a label that is used as input to the encoder/generator. The generated samples and their respective label are then plotted and saved to a file. Figure 5.5 shows an image generated by calling *save_fake_sample_plot_images*.

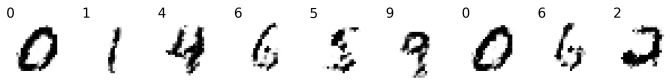

Figure 5.5.: The figure shows samples generated by a trained generator. It can be seen that the generator can produce specific numbers based on the given label (and random noise).

Reinforcement Learning

There are two visualizations implemented in the prototype to analyze and verify the World-Model-GAN. The first visualization is implemented in a method named *visualize_outputs_to_file* in *NetworkRL*. An image generated by the method can be seen in Figure 5.6.

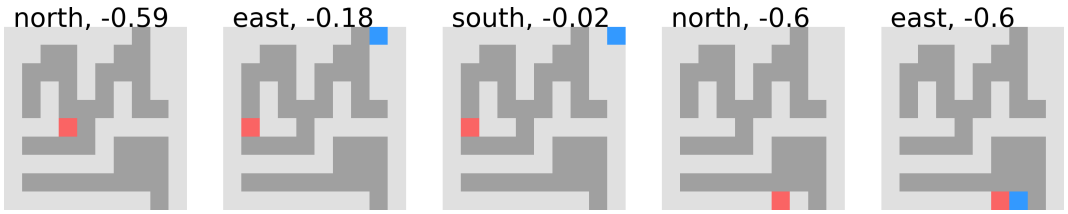

Figure 5.6.: The figure shows five different state-action pairs in a maze environment together with the model predictions. The maze itself represents the states, while the action and predicted reward are written above the maze. The blue block is the agent's state, while the red block is the predicted next state. The red block overshadows the blue block, which is the case if the predicted next state is the same as the initial state.

The second visualization is used to evaluate the Q-value network during training to ensure that a valid reinforcement learning agent acts in the environment. The evaluation ensures the creation of a realistic dataset during training. The training is visualized by two self-written classes named *NetworkMobjectSimple* and *RLScene*, which depend on a library called manim[2] (short for math animation). Executing *RLScene* through the manim module starts the visualized training process. When finished, the visualization is shown and saved as a video file. Because the two classes and manim itself are not directly linked to *NetworkRL* and are only used to ensure the Q-value network's correctness, they are not present in the class diagram. Figure 5.7 shows an image extracted from a video that is generated using *RLScene*. It can be seen that the environment and the network—including all weights—are visualized to verify the RL training process.

5.2.4. Requirements for a Gym Environment

While registering an environment as gym environment has its advantages, like the compatibility to a wide variety of algorithms, it also requires the environment to provide the following interfaces [23]:

[2]There is currently no paper related to manim. Manim is a python library to create mathematical animations using different shapes, latex, and standard text elements. Version 0.1.0 is used for this research and can be accessed through the following link: https://github.com/ManimCommunity/manim/tree/v0.1.0

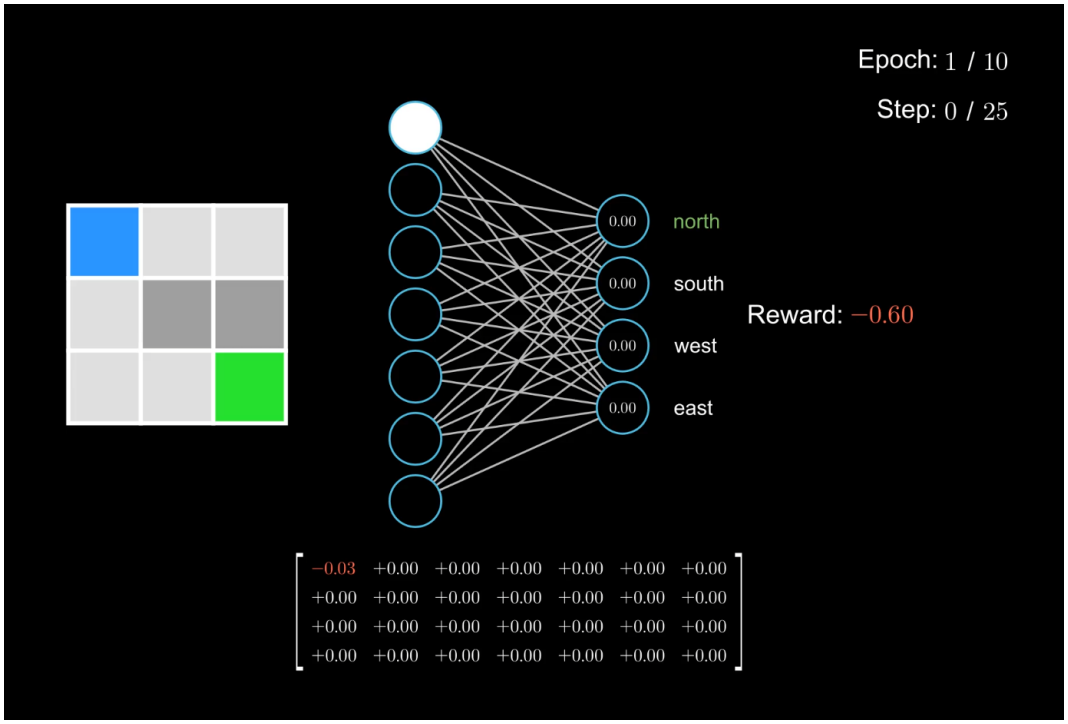

Figure 5.7.: The figure shows an image extracted from a video that is generated using the *RLScene* class. It can be seen that the environment and the network—including all weights—are visualized to verify the reinforcement learning training process. The underlying library to create and render the animation is called manim

1. The environment class needs to inherit from the *gym.Env* class
2. The environment class needs to define the action and observation space as *gym.spaces* objects during initialization
3. The environment class needs to provide a method named step (with a parameter that receives an action) to execute one timestep within the environment and to return an observation
4. The environment class needs to provide a reset method to reset the state of the environment to an initial state and to return an initial observation
5. The environment class needs to provide a render method to render the environment

The prototype contains a gym environment in the form of the *MazeEnv* class, which implements the mentioned interfaces. The environment itself is only required for the experiments regarding O2.

Chapter 6

Experiments

This chapter uses the self-developed prototype—discussed in the previous chapter—to verify the concept through experiments. It is split into two sections to separate the experiments for each objective.

6.1. Objective 1

This section covers the experiments regarding O1. Besides verifying the concept, the experiments are conducted in a way that shows the reader how the results derive from a simple architecture to the proposed SA-CAE architecture. To ensure reproducibility, Table 6.1 lists the default configuration. Any parameter that is not explicitly specified in the experiment's configuration table is defined by the value specified in the default configuration. Multiple comma-separated values mean that the experiment is conducted with different values of this parameter.

Table 6.1.: The table shows the default values for all parameters that are used throughout the experiments in this section

Model	Parameter Name	Parameter Value
AE	Decoder loss function	Binary cross-entropy
"	Classifier loss function	Categorical cross-entropy
"	Latent space dimension	20
"	Batch size	32
"	Training epochs	10
"	Layer type	Dense
"	Encoder / Decoder layer sizes	[500, 500]
"	Hidden activation function	Leaky ReLU ($\alpha = 0.2$)
"	Decoder output activation function	Sigmoid
"	Classifier output activation function	Softmax
"	Encoder / Decoder shared weights	True
"	Learning rate optimizer	Adam[1]
GAN	Discriminator loss function	Binary cross-entropy
"	Classifier loss function	Categorical cross-entropy
"	Training epochs	40
"	Batch size	32
"	Steps per epoch	500
"	Discriminator layer sizes	[500, 500]
"	Layer type	Dense
"	Hidden activation function	Leaky ReLU ($\alpha = 0.2$)
"	Discriminator output activation function	Sigmoid
"	Classifier output activation function	Softmax
"	Dropout	0.3
"	Label smoothing	0.1
"	Learning rate optimizer	Adam[2]
Data	Dataset name	MNIST
"	Training data size	45000
"	Validation data size	5000
"	Test data size	10000

[1] $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$
[2] $\alpha = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999, \epsilon = 10^{-8}$

6.1.1. AE with and without Classifier

ID 48692

This experiment compares results generated by a regular AE and an AE that can also classify the samples—referred to as Classification Autoencoder (CAE) in the following. The experiment tries to answer the question of whether an AE can:

1. Classify a sample during encoding
2. Use the classification and latent space to reconstruct the sample

Table 6.2 shows the configuration parameters that differ from the default configuration defined in Table 6.1.

Table 6.2.: Configuration for experiment ID 48692

Model	Parameter Name	Parameter Value
AE	Latent space dimension	5

As seen in Figures 6.1 and 6.2, it can be seen that adding classification neurons to the encoder does not affect the ability to reconstruct given samples even when using a low dimensional latent space. While using a CAE changes the setup from an unsupervised to a supervised setup, it adds the capability to classify given samples during the encoding process.

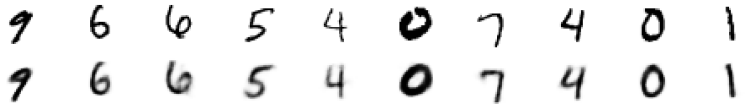

Figure 6.1.: The figure shows ten test samples in the first row and their associated reconstruction in the second row. The reconstructions are generated by the decoder model, which is a part of the Autoencoder model. It can be seen that the Autoencoder reconstructs samples accurately, showing only minor indifferences.

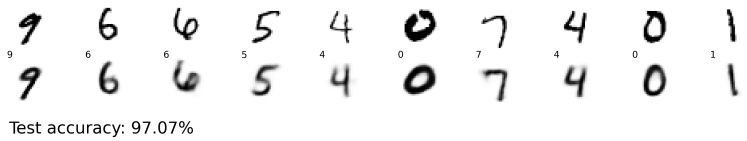

Figure 6.2.: The figure shows ten test samples in the first row and their associated reconstruction in the second row. Between both rows is the number predicted by the classifier given the test sample. The reconstructions are generated by the decoder model, which is a part of the Autoencoder model. Below both rows is the classification accuracy that the classifier achieved on the test set. It can be seen that the Autoencoder can reconstruct and classify the samples accurately by showing only minor reconstruction indifferences and classification errors.

6.1.2. CAE Generative Capabilities

ID 23573

This experiment analyzes whether the classification neurons in a CAE can influence the decoder's reconstructions in the direction of samples labeled by the classification value. The experiment is conducted using a test sample (Figure 6.3) and a sample consisting of randomly generated values (Figure 6.4). Besides, different latent space sizes are used to see the effects of the latent space neurons' dominance compared to the classification neurons.

Table 6.3 shows the configuration parameters that differ from the default configuration defined in Table 6.1.

Table 6.3.: Configuration for experiment ID 23573

Model	Parameter Name	Parameter Value
AE	Latent space dimension	5, 20

As seen in Figure 6.3, the classification neurons can enable transformative capabilities in an AE. It seems, however, that this applies just for the case where classification neurons dominate the latent space neurons. If the latent space neurons dominate the classification neurons, the generative capabilities are not present, and the AE produces an accurate reconstruction no matter which classification neuron is active.

Figure 6.4 tries to address the objective by using a random sample and different classifications to generate new samples that are similar to samples in the dataset with the given classification label. As seen in the figure, however, the CAE model is not able to achieve this objective. The problem seems to be that the difference between the training samples and the randomly generated samples is too strong, resulting in uncommon latent space values that are rarely experienced during training. Additionally, an AE's objective is to reconstruct a sample and not the generation of a new sample, even though both objectives should be similar in regard to their output values.

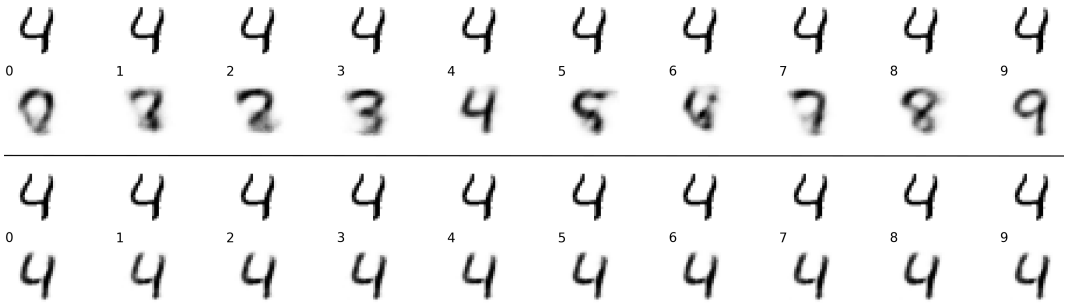

Figure 6.3.: The image is split into two sections. The upper section shows results with a latent space size of 5, while the lower section shows results with a latent space size of 20. Each section shows the same test sample ten times in the first row. The test sample is annotated by the classification value seen below the test sample image, which is used together with the latent space representation to generate the reconstructions seen in the second row. The latent space value is generated by the encoder given the test sample. Therefore it is the same for all columns. It can be seen that the decoder can transform the sample solely based on the classification value in the case where the classification neurons dominate the latent space neurons.

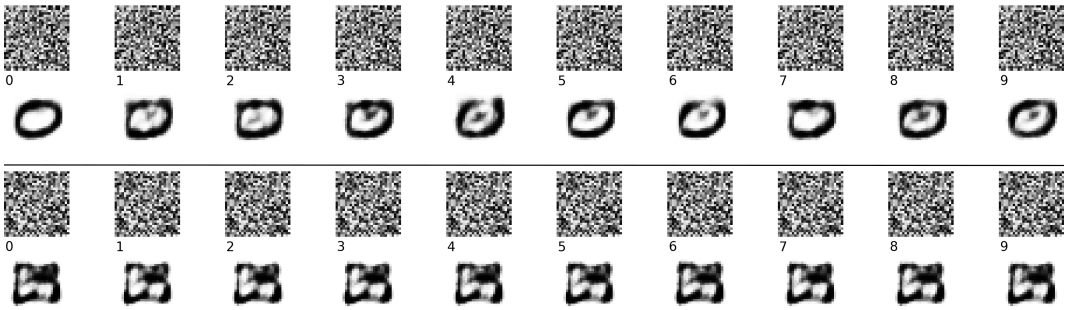

Figure 6.4.: In analogy to the image structure described in Figure 6.3, the decoder is tasked to create reconstructions from a random input. As seen in the figure, using this approach does not allow the decoder to generate new samples that reflect the classification value in the form of an image.

6.1.3. CAE-GAN with Random Input Samples

ID 93832

This experiment analyzes whether training the decoder by using both the GAN and the AE can help to achieve the objective. The idea is to alternately train the decoder as a generator in a GAN training step and as a standard decoder in an AE training step. Because the decoder and encoder are using the same weights, the encoder is also trained during the GAN training step. The experiment does not use a validation set, which means that 50000 MNIST samples are used to train the models while the remaining 10000 samples are used for testing purposes after the training process.

Table 6.4 shows the configuration parameters that differ from the default configuration defined in Table 6.1.

Table 6.4.: Configuration for experiment ID 93832

Model	Parameter Name	Parameter Value
AE	Encoder / Decoder layer sizes	[1024, 512, 256]
"	Training epochs	6
"	Batch size	16
"	Steps per epoch	500
"	Latent space dimension	20, 50
GAN	Discriminator layer sizes	[1024, 512, 256]
"	Training epochs	6
Data	Training data size	50000
"	Validation data size	0
"	Test data size	10000

As seen in Figures 6.5 and 6.6, adding a GAN to the training process helps transform and generate new samples based on the classification value. However, it can be seen that the quality of the samples is reduced due to the increased complexity that occurs when creating random samples. The stronger generalization also decreases the reconstruction quality, as seen in the fifth column in Figure 6.5. One can argue that the complexity is the same because the dataset samples have the same dimension. However, the n-dimensional space covered by the dataset is more compressed than the fully expanded space when creating random samples.

As seen in Figure 6.6, increasing the latent space seems to decrease the ability to generate numbers. This decreased ability is likely because the latent space neurons dominate the classification neurons by a factor of 5:1.

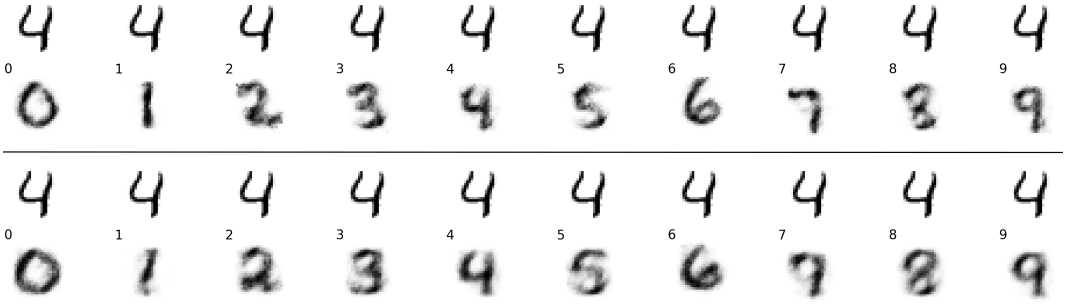

Figure 6.5.: In analogy to the image structure described in Figure 6.3—except that a latent space of 20 (upper half) and 50 (lower half) is used—, it can be seen that the decoder can transform the sample solely based on the classification value even for a relatively large latent space of 50.

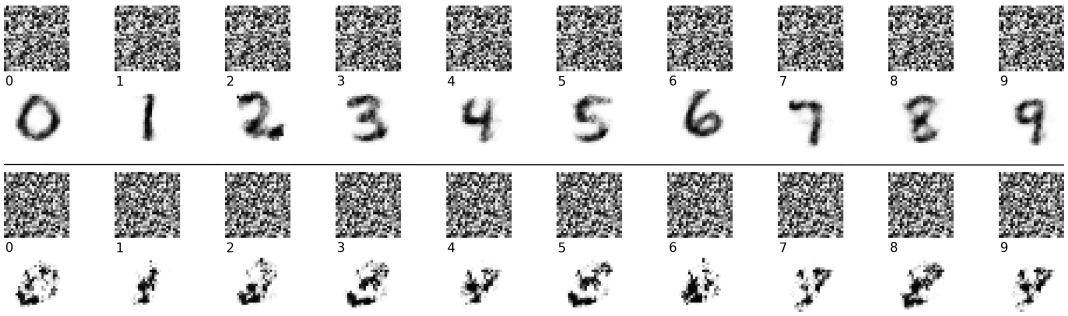

Figure 6.6.: In analogy to the image structure described in Figure 6.4—except that a latent space of 20 (upper half) and 50 (lower half) is used—, it can be seen that the decoder can generate new samples solely based on the classification value because the latent space is the same for all of the ten images. Increasing the latent space, however, reduces the quality of the generated samples.

6.1.4. AC-GAN with Random Input Samples

ID 79532

This experiment analyzes whether using an AC-GAN instead of a normal GAN helps to generate samples from random input even when the latent space neurons dominate the classification neurons in the AE. Adding a classifier to the discriminator forces the generator to fool the real or fake discrimination by the discriminator and create the number represented by the classification neurons so that the classifier in the discriminator can classify it correctly.

Table 6.5 shows the configuration parameters that differ from the default configuration defined in Table 6.1. Using an AC-GAN, the latent space dimension is the input dimension and the encoder/decoder layer sizes are the feedforward layer sizes.

Table 6.5.: Configuration for experiment ID 79532

Model	Parameter Name	Parameter Value
AE	Encoder / Decoder layer sizes	[1024, 512, 256]
"	Training epochs	6
"	Batch size	16
"	Steps per epoch	500
"	Latent space dimension	20, 50
GAN	Discriminator layer sizes	[1024, 512, 256]
"	Training epochs	6
Data	Training data size	50000
"	Validation data size	0
"	Test data size	10000

As seen in Figure 6.7, using the additional classifier in the GAN improves the generated samples even when the latent space neurons dominate the classification neurons.

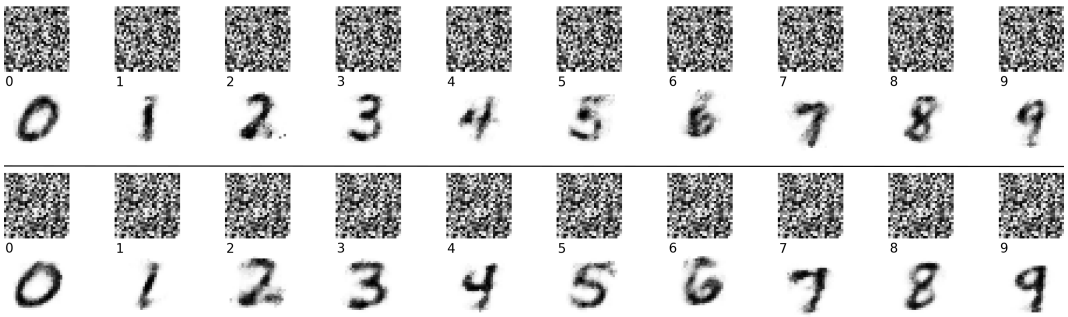

Figure 6.7.: In analogy to the image structure described in Figure 6.6, it can be seen that the decoder can generate new samples solely based on the classification value because the latent space is the same for all of the ten images. Opposed to Figure 6.6, however, the generated samples do not visibly decrease in quality when increasing the latent space to 50.

6.1.5. Comparison between SA-CAE and AC-GAN

ID 53784

This experiment compares the results between the SA-CAE and AC-GAN architecture. While the former receives random input samples and encodes them to generate new samples through the decoder/generator, the latter has no encoder and therefore receives random latent space values that are fed into the generator to generate new samples.

Table 6.5 shows the configuration parameters that differ from the default configuration defined in Table 6.1. The AC-GAN architecture uses the same parameters as the SA-CAE architecture except that the latent space dimension is the input dimension and the encoder/decoder layer sizes are the feedforward layer size.

Table 6.6.: Configuration for experiment ID 53784

Model	Parameter Name	Parameter Value
AE	Encoder / Decoder layer sizes	[1024, 512, 256]
"	Training epochs	6, 30
"	Batch size	16
"	Steps per epoch	500
"	Latent space dimension	50
GAN	Discriminator layer sizes	[1024, 512, 256]
"	Training epochs	6, 30
"	Steps per epoch	500
Data	Training data size	50000
"	Validation data size	0
"	Test data size	10000

Figure 6.8 shows the results for an AC-GAN with random latent space inputs; Figure 6.9 shows the results for an AC-GAN with latent space inputs sampled from a normal Gaussian distribution; Figure 6.10 shows the results for an SA-CAE with random encoder inputs. Each figure shows 50 samples on the left, generated after six epochs of training, and 50 samples on the right, which are generated after 30 epochs of training.

First, Figure 6.8 shows that using a random latent space results in a low sample quality where the label is hard to assign in some cases. Sampling the latent space from a normal Gaussian distribution (Figure 6.9) addresses this problem. It can be seen that the samples can be assigned to their respective label more comfortably.

Comparing Figure 6.9 and 6.10 shows that the sample quality between the sixth and thirtieth epoch increases for the AC-GAN while decreasing for the SA-CAE. Interestingly, the sample quality looks better after six epochs of SA-CAE training than thirty epochs of AC-GAN training. It can be seen, however, that the sample variance is more significant when using an AC-GAN compared to the SA-CAE, meaning that it can generate samples with the same label that differ more from each other. It is assumed that this is either because the encoder compresses the random input into similar latent space values, therefore just using a fraction of the latent space, or because of a too strong discriminator. Further research needs to be conducted to examine the exact cause of the small sample variety.

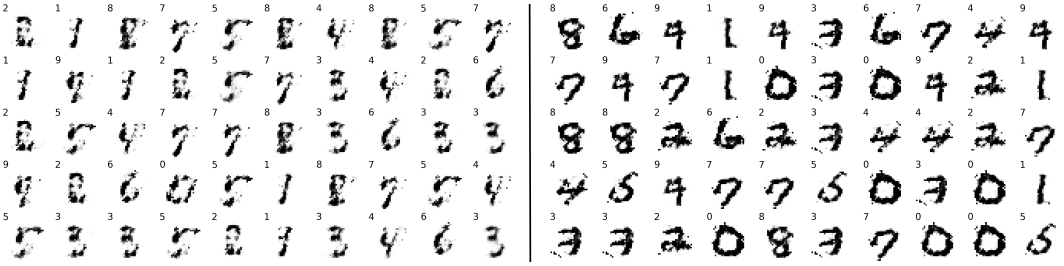

Figure 6.8.: The figure shows 100 generated samples. The left half contains 50 generated samples after training the AC-GAN for six epochs with random latent space inputs. The same applies to the right half after training for thirty epochs. The label of the value that the network is supposed to generate can be seen on each sample's upper left corner. When looking at the generated samples, it can be seen that the sample quality increases only slightly and is, for example, not as good as the samples generated when using a normal Gaussian distribution or an SA-CAE.

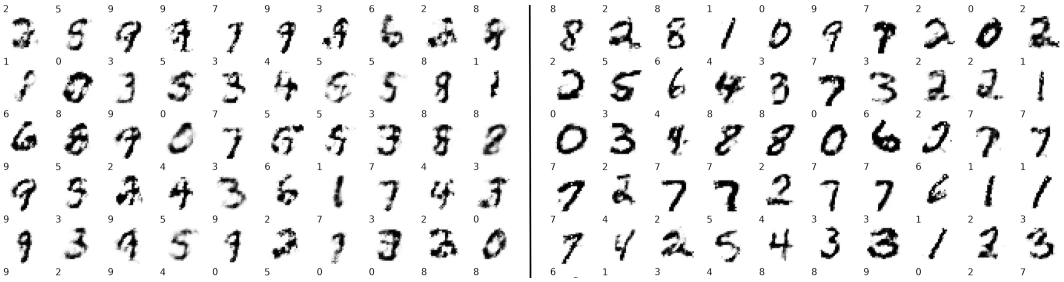

Figure 6.9.: In analogy to the image structure described in Figure 6.8, this figure shows generated samples for an AC-GAN using a normal Gaussian distribution. While the sample quality after six epochs is comparable to the samples shown in Figure 6.8, it increases after thirty epochs. It also shows a greater variety of generated samples with the same label.

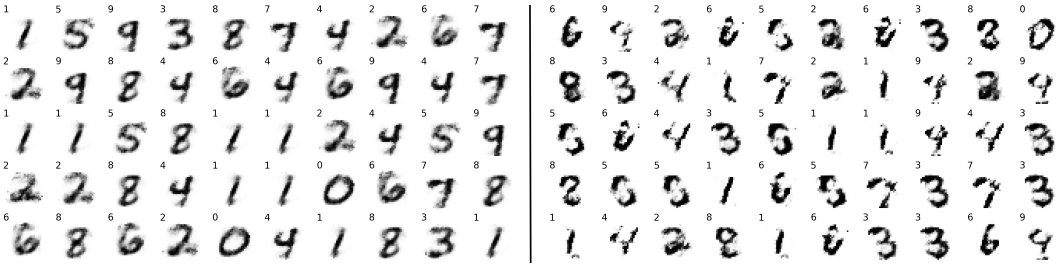

Figure 6.10.: In analogy to the image structure described in Figure 6.8, this figure shows generated samples for an SA-CAE using random samples as inputs. It can be seen that—after six epochs—the image quality is superior to both Figure 6.9 and 6.10. The sample variety of samples with the same label, however, is lower compared to Figure 6.9 and similar to Figure 6.8. It should also be noted that the quality decreases as seen when inspecting the right half (after thirty epochs) compared to the left half (after six epochs).

.

6.2. Objective 2

This section covers the experiments regarding O2. The experiments build upon the insights gained through the experiments of O1 and extend it to the World-Model-

GAN architecture. To ensure reproducibility, Table 6.7 lists the default configuration. Any parameter that is not explicitly specified in the experiment's configuration table is defined by the value specified in the default configuration. The generated dataset, which is split into training and testing data, always contains all possible state-action combinations for the environment used in the specific test.

Table 6.7.: The table shows the default values for all parameters that are used throughout the experiments in this section

Model	**Parameter Name**	**Parameter Value**
Generator	Next state loss function	Binary cross-entropy
"	Reward loss function	MSE
"	Training epochs	60
"	Batch size	2
"	Steps per epoch	200
"	Layer type	Dense
"	Layer sizes	[30, 50]
"	Hidden activation function	Leaky ReLU ($\alpha = 0.2$)
"	Next state activation function	Softmax
"	Reward activation function	Tanh
"	Learning rate optimizer	Adam[1]
Discriminator	Loss function	Binary cross-entropy
"	Layer sizes	[10]
"	Layer type	Dense
"	Hidden activation function	Leaky ReLU ($\alpha = 0.2$)
"	Output activation function	Sigmoid
Discriminator & GAN	Learning rate optimizer	Adam[2]
"	Training epochs	60
"	Batch size	4
"	Steps per epoch	200
Data	Dataset name	Maze (10x10)
"	Training data size	116
"	Test data size	116

[1] $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$
[2] $\alpha = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999, \epsilon = 10^{-8}$

6.2.1. Training Approaches

The experiments are similar to each other with the difference of using different training approaches. The approaches are summarized in the following:

1. The first training approach consists of three training steps. The first training step consists of updating the discriminator weights by using real trajectories from the training data and fake trajectories generated through the generator, which receives state-action inputs from the test data to complete the trajectory. The second training step consists of updating the generator weights in a supervised setup using real trajectories from the training data. The third training step consists of updating the generator through the GAN using state-action inputs from the test dataset.
2. The second training approach differs from the first in two aspects. First, the discriminator weights are updated using real trajectories from the training data and fake trajectories generated through the generator, which receives state-action inputs from the training or test data to complete the trajectory. Second, the generator is updated through the GAN by using state-action inputs from the training or test dataset.
3. The third training approach differs from the second approach by removing the generator's supervised training step.

6.2.2. World-Model-GAN using Training Approach 1

ID 78432

This experiment analyzes whether the generator can learn to create realistic next states based on unknown state-action pairs using the test dataset as inputs for the generator during GAN and discriminator training. The generator is additionally trained in a supervised setup using the training data. A dataset from a 10x10 maze environment is used throughout this experiment. There are no parameters that vary from the default configuration.

Figure 6.11 shows the results for training samples, while Figure 6.12 shows the results for test samples. It can be seen that the model learns to accurately predict the next states for state-action pairs present in the training dataset. It fails, however, to predict the next state on the test samples that are used for training the generator

in a GAN. The reward predictions on the training data show slight deviations from the reward labels in the training dataset, while the predictions on the testing data are accurate for some samples while being inaccurate for others.

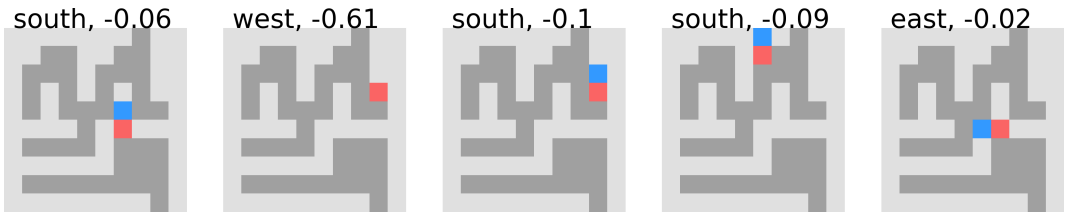

Figure 6.11.: The figure shows the predictions on train samples. The initial state is shown in blue; the action is shown on the left above the maze; the predicted reward is shown on the right above the maze; the predicted next state is shown in red. If there is no blue block, the predicted next state is the same as the initial state. It can be seen that the model can accurately predict the next state and only shows slight deviations for the predicted reward.

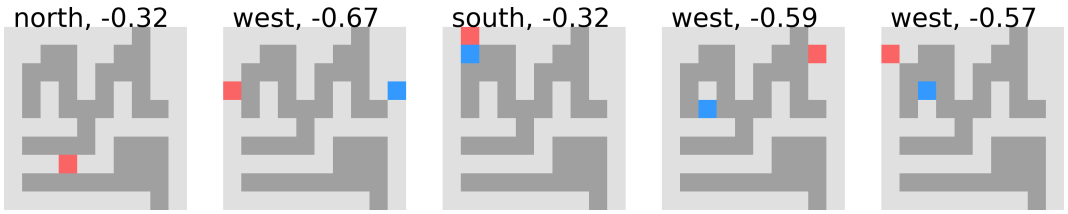

Figure 6.12.: This figure is analog to 6.11 except that test samples are used. It can be seen that the model is unable to predict the next state for all shown samples while being only partly accurate with the reward prediction.

6.2.3. World-Model-GAN using Training Approach 2

ID 38365

This experiment analyzes whether the generator can learn to create realistic next states based on unknown state-action pairs using the full dataset as possible inputs for the generator during GAN and discriminator training. The generator is additionally trained in a supervised setup using the training data. A dataset from a 10x10 maze environment is used throughout this experiment. There are no parameters that vary from the default configuration.

Figure 6.13 shows the results for training samples, while Figure 6.14 shows the results for test samples. It can be seen that using the full dataset during GAN and discriminator training does not improve the test sample outputs compared to the results achieved using the first training approach.

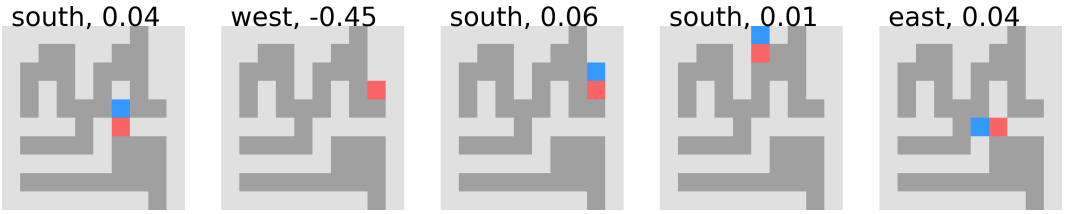

Figure 6.13.: The figure is analog to 6.11, except that the second training approach is used. As with the first training approach, it can be seen that the model can accurately predict the next state while showing only slight deviations for the predicted reward.

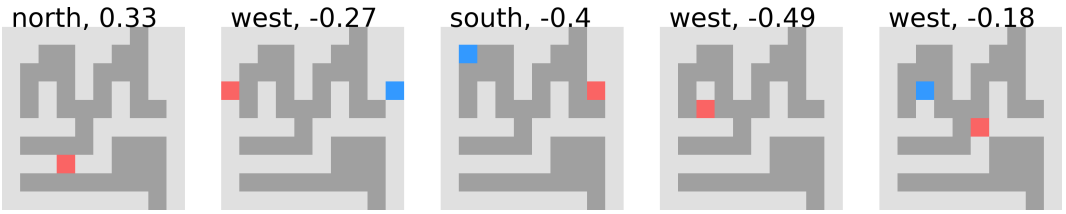

Figure 6.14.: This figure is analog to 6.12 except that the second training approach is used. As with the first training approach, it can be seen that the model is unable to predict the next state. The reward prediction shows even more inaccuracy compared to the first approach.

6.2.4. World-Model-GAN using Training Approach 3

ID 98256

This experiment analyzes whether the generator can learn to create realistic next states based on unknown state-action pairs using the full dataset as possible inputs for the generator during GAN and discriminator training. Opposed to the second approach, the generator is not trained in a supervised setup. A dataset from a 10x10 maze environment is used throughout this experiment. There are no parameters that vary from the default configuration.

Figure 6.15 shows the results for training samples, while Figure 6.16 shows the results for test samples. It can be seen that omitting the supervised training does not improve the test sample outputs compared to the results achieved using the other training approaches. Additionally, without the supervised training step, the generator is unable to accurately predict the next state and reward for state-action-pairs from the training data.

To summarize, the first two training approaches can be used to create an accurate model of the environment for known state-action pairs. In contrast, the third approach fails to learn an accurate model due to the missing supervised learning step. When it comes to learning the physics of the environment to predict realistic next states and rewards for new unknown state-action pairs, all three training approaches fail. Possible reasons and limitations for this are discussed in the next chapter.

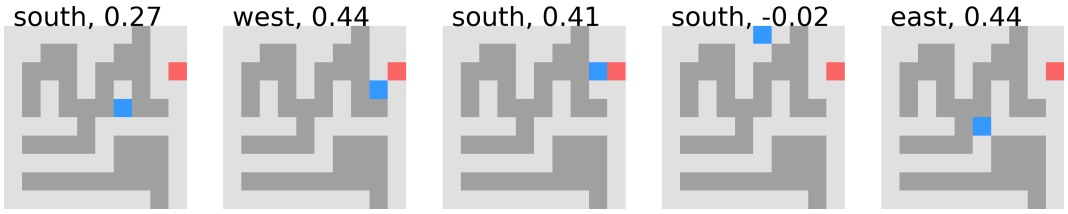

Figure 6.15.: The figure is analog to 6.11, except that the third training approach is used, which omits the supervised training step. As opposed to the other approaches, it can be seen that the model cannot make accurate predictions for the next state and reward.

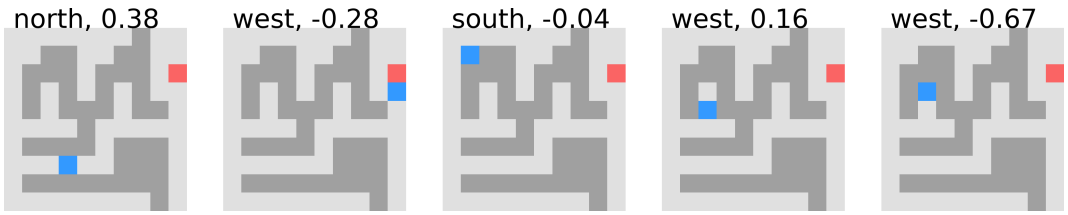

Figure 6.16.: This figure is analog to 6.12 except that the third training approach is used, which omits the supervised training step. As with the other approaches, it can be seen that the model is unable to predict the next state and makes inaccurate predictions for the reward.

Chapter 7

Results

This chapter summarizes the findings from the experiments that are conducted in the previous chapter. It is separated into two sections, with each section addressing one objective.

7.1. Objective 1

The experiments show that both an SA-CAE and AC-GAN can be used to generate samples from random inputs. While the former requires random input samples, the latter requires random latent space values. It is hard to argue that one approach is better than the other because they can not be used interchangeably. An SA-CAE, however, might be a suitable solution for a problem like reconstructing input samples where parts have been cut off—a problem for which an AC-GAN cannot be used. This means that the SA-CAE extends the usable problem setting compared to the AC-GAN without increasing the duration to perform backpropagation due to weight sharing. Only the forward propagation pass needs more computational calculations due to the additional encoder model. An essential factor that enables the generative capabilities is the GAN network. Only after adding the GAN, the AE shows that new samples can be generated from a random input vector.

While the first part of the objective—"Generating new samples given random inputs and a label..."—is addressed by both the SA-CAE and the AC-GAN, only the former addresses the second part, which states "... as well as encoding these samples, using the same weights". This is because the AC-GAN does not have an encoder model.

To summarize, the results gained from analyzing the experiments show that O1 can be achieved by using an SA-CAE as defined in the concept.

7.2. Objective 2

The experiments show that the proposed World-Model-GAN is not suitable to address the objective. While the concept can be used to learn a model for given state-action pairs, it can not learn the state transitions and reward predictions for unknown state-action pairs. The following paragraphs will discuss the possible reasons why the concept might be problematic for the objective.

The first problem is the use of binary neurons. Assuming the 3x3 maze state shown in Figure 4.1 is used under the premise that the action of going up and left is present in the training dataset, it can be explained as follows:

When the agent is training on the data, it always increases the weight connection(s) to the output neuron representing the activation of the same state ($S_{t+1} = S_t$). When the agent then encounters the same state with the action of going down or right in the test dataset, it cannot leave the state due to the strong association between the respective input- and output neuron. The discriminator is used to weaken the connection. To do so, it needs to predict *fake* when the agent does not change its state (even though the action is *down* or *right*) and *real* if it does. This requirement leads to the second problem.

The second problem is that the discriminator can learn which samples are fake and real through memorization. This memorization is possible because the initial state and action are passed to the discriminator, and omitting them is problematic because they are required to learn the state transitions. In the experiment, 116 training samples and 116 test samples are available. The discriminator can, therefore, directly learn that state-action pairs present in the training dataset are always associated with *real*. In contrast, state-action pairs from the test dataset are always associated with *fake*. It is supposed that this is the main problem that prevents the generator from learning the transitions. It does also explain that the same results occur with architecture and hyperparameter changes.

The second and third experiments use all 232 samples as possible inputs for the generator during GAN and discriminator training to prevent memorization. Therefore, each state-action pair can be labeled as *real* or *fake*, depending on whether the next

state and result are gathered from the dataset or generated through the generator. It can be seen, however, that using this approach does not improve the results compared to the results from the first experiment. It is assumed that the discriminator cannot learn the physics of the environment, which in turn prevents the generator from learning accurate predictions. Further research needs to be conducted to verify this statement and find possible solutions. Experiments that further decrease the discriminator size to increase the generalization have been conducted. Those experiments are not included in this research due to similar results.

Chapter 8

Conclusion

This research shows how generative models can be trained in an adversarial setup to create realistic new samples. Based on the insights gained through the experiments and result section, it can be concluded that the SA-CAE can be used to train a model that, in turn, can create realistic new samples. The World-Model-GAN, used to address O2, needs further work to address the limitation of being unable to learn state transitions for unknown states. It can, however, be used to learn an accurate model of the transitions present in the dataset.

8.1. Objective 1

The related work to O1 analyzes different approaches to generate realistic new samples from randomly drawn latent space values. In contrast, this research shows that the higher dimensional input can be randomized with achieving similar quality for the generated samples without requiring additional weights—due to weight sharing. The SA-CAE prototype allows further research to analyze how the presented concept performs on tasks like noise reduction and the reconstruction of distorted image segments compared to approaches that are not using adversarial training. Both concept and prototype can also be extended to use convolutional- instead of densely connected layers and a more sophisticated loss function for the AE like the MS-SSIM loss (requires the use of a different dataset). The usage of convolutional layers would further allow a comparison with the architectures described in the related work chapter.

Besides, this research shows a set of parameters that can be used to train the AC-GAN architecture on an MNIST dataset using densely connected layers. This approach differs from the use-case presented in the AC-GAN paper [28], where a convolutional AC-GAN architecture is used to increase image discriminability through upscaling.

An exciting direction for further research is whether the classifier in the discriminator can improve the classifier in the encoder through shared weights. While this enforces the discriminator to use the same architecture as the encoder, it is assumed that the classifier can be improved due to the increased sample variety produced by the generator.

8.2. Objective 2

This research further provides a concept and prototypical implementation of the World-Model-GAN, an architecture that allows learning a model of a given environment from collected data by training a generator network in an adversarial setup. While the World-Model-GAN can be used to learn an accurate model of the environment through supervised training, it cannot learn the physics of the environment, which results in inaccurate predictions for unseen state-action pairs. Further research is recommended to analyze how the discriminator can effectively learn the physics of the environment, which seems to be the limiting factor of this research.

Assuming the World-Model-GAN limitation is solved, one could adjust the concept to learn the model while training the agent in a cyclic training process. One could even integrate the Q-value network into the model network to create a model that can both predict realistic future states and the respective Q-values. This modification would allow the agent to sample future trajectories using its environment model to improve itself.

List of Abbreviations

MS-SSIM	Multi-Scale Structural-Similarity Score
SA-CAE	Shared Adversarial Classification Autoencoder
GAN	Generative Adversarial Network
AE	Autoencoder
VAE	Variational Autoencoder
ReLU	Rectified Linear Unit
tanh	Hyperbolic Tangent
MSE	Mean Squared Error
RBMs	Restricted Boltzmann Machines
SGAN	Semi-Supervised GAN
CNN	Convolutional Neural Network
CGAN	Conditional GAN
AC-GAN	Auxiliary Classifier GAN
RL	Reinforcement Learning
MDP	Markov Decision Process
RNN	Recurrent Neural Network
DCGAN	Deep Convolutional GAN
CAE	Classification Autoencoder
PCA	Principal Component Analysis

List of Tables

List of Figures

Listings

Bibliography

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative adversarial nets", in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680.

[2] Tero Karras, Samuli Laine, and Timo Aila, "A style-based generator architecture for generative adversarial networks", presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 4401–4410.

[3] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever, "Jukebox: A generative model for music", *arXiv:2005.00341 [eess.AS]*, 2020.

[4] Yuval Nir and Giulio Tononi, "Dreaming and the brain: From phenomenology to neurophysiology", *Trends in Cognitive Sciences*, vol. 14, no. 2, pp. 88–100, 2010. DOI: 10.1016/j.tics.2009.12.001.

[5] David Ha and Jürgen Schmidhuber, "Recurrent world models facilitate policy evolution", in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 2450–2462.

[6] Andrej Krenker, Janez Bester, and Andrej Kos, "Introduction to the artificial neural networks", in *Artificial neural networks-methodological advances and biomedical applications*, IntechOpen, 2011.

[7] Warren S Sarle, "Neural networks and statistical models", *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, pp. 1538–1550, 1994, CiteSeerX. DOI: 10.1.1.27.699.

[8] Vinod Nair and Geoffrey E. Hinton, "Rectified linear units improve restricted boltzmann machines", presented at the ICML, 2010.

[9] M.T. Tommiska, “Efficient digital implementation of the sigmoid function for reprogrammable logic”, *IEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 6, pp. 403–411, 2003. DOI: 10.1049/ip-cdt:20030965.

[10] Seppo Linnainmaa, “Taylor expansion of the accumulated rounding error”, *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976. DOI: 10.1007/BF01931367.

[11] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, “Learning representations by back-propagating errors”, *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, Nature Publishing Group. DOI: 10.1038/323533a0.

[12] Dor Bank, Noam Koenigstein, and Raja Giryes, “Autoencoders”, *arXiv:2003.05991 [cs.LG]*, 2020.

[13] Jake Snell, Karl Ridgeway, Renjie Liao, Brett D. Roads, Michael C. Mozer, and Richard S. Zemel, “Learning to generate images with perceptual similarity metrics”, *arXiv:1511.06409 [cs.LG]*, 2017.

[14] Andrew Y. Ng and Michael I. Jordan, “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes”, in *Advances in Neural Information Processing Systems 14*, MIT Press, 2002, pp. 841–848.

[15] G. E. Hinton, “Reducing the dimensionality of data with neural networks”, *Science*, vol. 313, no. 5786, pp. 504–507, 2006. DOI: 10.1126/science.1127647.

[16] Ping Li and Phan-Minh Nguyen, “On random deep weight-tied autoencoders: Exact asymptotic analysis, phase transitions, and implications to training”, in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HJx54i05tX.

[17] Pierre Baldi and Kurt Hornik, “Neural networks and principal component analysis: Learning from examples without local minima”, *Neural networks*, vol. 2, no. 1, pp. 53–58, 1989, Publisher: Elsevier.

[18] Diederik P. Kingma and Max Welling, “Auto-encoding variational bayes”, *arXiv:1312.6114 [stat.ML]*, 2014.

[19] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane,

Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems", *arXiv:1603.04467 [cs.DC]*, 2016.

[20] François Chollet *et al.*, *Keras*. 2015. [Online]. Available: https://keras.io (visited on 08/29/2020).

[21] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[22] Christopher JCH Watkins and Peter Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992, Springer.

[23] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba, "OpenAI gym", *arXiv:1606.01540 [cs.LG]*, 2016.

[24] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther, "Autoencoding beyond pixels using a learned similarity metric", *arXiv:1512.09300 [cs.LG]*, 2016.

[25] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey, "Adversarial autoencoders", *arXiv:1511.05644 [cs.LG]*, 2016.

[26] Augustus Odena, "Semi-supervised learning with generative adversarial networks", *arXiv:1606.01583 [stat.ML]*, 2016.

[27] Mehdi Mirza and Simon Osindero, "Conditional generative adversarial nets", *arXiv:1411.1784 [cs.LG]*, 2014.

[28] Augustus Odena, Christopher Olah, and Jonathon Shlens, "Conditional image synthesis with auxiliary classifier GANs", *arXiv:1610.09585 [stat.ML]*, 2017.

[29] Ajkel Mino and Gerasimos Spanakis, "LoGAN: Generating logos with a generative adversarial neural network conditioned on color", *arXiv:1810.10395 [cs.CV]*, 2018.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey", *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996. DOI: 10.1613/jair.301.

[31] Nikhil Buduma and Nicholas Locascio, *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, Inc., 2017, 298 pp.

[32] Ning Qian, "On the momentum term in gradient descent learning algorithms", *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999. DOI: 10.1016/S0893-6080(98)00116-6.

[33] John Duchi, Elad Hazan, and Yoram Singer, "Adaptive subgradient methods for online learning and stochastic optimization.", *Journal of machine learning research*, vol. 12, no. 61, pp. 2121–2159, 2011.

[34] Matthew D. Zeiler, "ADADELTA: An adaptive learning rate method", *arXiv:1212.5701 [cs.LG]*, 2012.

[35] Diederik P. Kingma and Jimmy Ba, "Adam: A method for stochastic optimization", *arXiv:1412.6980 [cs.LG]*, 2017.

[36] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio, "Object recognition with gradient-based learning", in *Shape, Contour and Grouping in Computer Vision*, Berlin, Heidelberg: Springer, 1999, pp. 319–345. DOI: 10.1007/3-540-46805-6_19.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *Journal of machine learning research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[38] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen, "Improved techniques for training GANs", in *Advances in Neural Information Processing Systems 29*, Curran Associates, Inc., 2016, pp. 2234–2242.

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016.

[40] Tom Schaul, Ioannis Antonoglou, and David Silver, "Unit tests for stochastic optimization", *arXiv:1312.6055 [cs.LG]*, 2014.

[41] Alec Radford, Luke Metz, and Soumith Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks", *arXiv:1511.06434 [cs.LG]*, 2016.

[42] Yann LeCun, Corinna Cortes, and CJ Burges, "MNIST handwritten digit database", *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2010. (visited on 10/17/2020).

[43] Jill H Larkin and Herbert A Simon, “Why a diagram is (sometimes) worth ten thousand words”, *Cognitive science*, vol. 11, no. 1, pp. 65–100, 1987, Wiley Online Library.

Appendix A

RLScene Video Attachment

The video files can be accessed through the following link:

https://drive.google.com/drive/folders/1Nks-8TpE4ui2KfxRbR05t63NZDdalYrk

If—for some reason—the link is not working, the files can be requested by sending an email to *coenen.christian@outlook.com*.