Implementing a Variational Autoencoder and LSTM with Gaussian Mixture outputs with TensorFlow to learn a World Model of the CarRacing_v0 gym environment

Eva Kuth, Pia Stermann and Friedrich Heitzer

April 2021

Abstract

As our final project for the 'Implementing Artificial Neural Networks with tensorflow' course we attended during WS 2020/21 at the University of Osnabrück, we decided to attempt a reimplementation of the World Models approach by Jürgen Schmidhuber and David Ha from 2018 [HS18] to solve the CarRacing_v0 gym Reinforcement Learning environment. We made slight modifications to the implementation proposed by the authors, e.g. the use of PPO to train the Controller instead of genetic or evolutionary algorithms. In this document we describe mainly the motivation for, background and training (and training outcomes) of the World Model part of the agent architecture, namely the Vision and Memory module, which are implemented as a Variational Autoencoder and a Long-Short-Term-Memory with Mixture Density Network output layer, respectively.

Introduction

We chose to attempt a reimplementation of the "World Models" approach as presented by David Ha and Jürgen Schmidhuber in 2018 in the same-named paper [HS18], which uses different modules such as a Variational Autoencoder (VAE, the "Vision" module) to compress image frames (state observations of the environment), which are further used to train an (LSTM-)RNN with a Mixture Density Network output layer (together accounting for the "Memory" module) to make predictions about possible future frame compression codes, and a "Controller" module on top which is responsible for the RL-part (i.e. policy learning based on the other agent components' outputs).

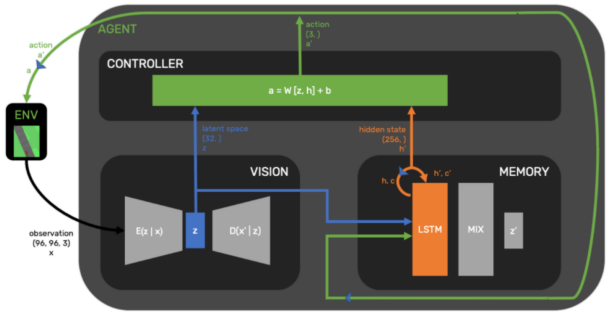

Figure 1: Structure of the World Models agent (image taken from this blog post by Adam Green

See figure 1 for a visualization of the structure and organization of the different component models that make up the RL agent. As one can see, the world models approach makes use of a variety of ANN architectures, concepts and techniques. This comes in handy, because we could well split the work of reimplementing it into two projects: For this first project we focus rather on the implementation and training of the Vision and Memory modules (as this is more related to what we learned in the IANNwTF course), while the second project deals with the implementation of the Controller and training thereof. There, we also elaborate more on how the different modules are combined to make up for a working RL algorithm and discuss it in the context of related research on RL algorithms. Furthermore, we will assess its performance (compared to existing alternative approaches), possible shortcomings or limitations, and make suggestions for future adaptations.

World Models for RL

In Reinforcement Learning one can distinguish between two major classes of algorithms, namely model-based and model-free algorithms.

Generally, model-based algorithms explicitly make use of and are informed by a model of the underlying Markov Decision Process (MDP) dynamics (either learned or known a priori). Whereas model-free approaches to RL do not require and use this additional information, or at least not explicitly. World models can generally be categorized as belonging to the family of model-based RL algorithms. However, the agent here does not have access to the full description of the MDP, but can only observe the environment via interactions and from this experience construct a learned model of its environment. This is typically achieved by an augmentation of the computational agent model's internal structure with a module that can predict future states based on previous experience from interactions with the MDP.

One of the authors of the World Models paper gave a really nice and compact (˜14 min) description of their approach, possible extensions, limitations and related work in a talk at the Neural Information Processing Sciences Conference in 2018.

Motivation

Firstly, in section 'Personal Interest', we state how we personally came to the topic, and what made it so appealing to us that we chose to try things out ourselves. Secondly, the general motivation for the World Models approach can be regarded as twofold: On the one hand, if implemented reasonably and applied to suitable problems, the approach has certain computational advantages over its model-free alternatives (in RL) and generally addresses some common problems of model-free and model based RL algorithms. This is what we will discuss further in section 'Computational Motivation'. On the other hand, it also seems to get close to how one might believe humans to learn, which we elaborate further on in section 'Cognitive Plausibility'.

Personal Interest

The topic of world models is dominated by two researchers, namely David Ha and Jürgen Schmidhuber. Schmidhuber mentioned the concept of world models for the first time in his report "Making the World Differentiable: On Using Self-Supervised Fully Recurrent Neural Networks for Dynamic Reinforcement Learning and Planning in Non-Stationary Environments" in the year 1990 [Sch]. His recent work with David Ha of Google (2018) [HS18] can be seen as probably the first or at least highly influential implementation of a World Models agent. We personally touched upon this topic in the last part of the course "Deep reinforcement learning" for the first time. From the very moment we saw the introduction video about this topic, we wanted to work on world models in our final project. How exactly is it possible to build software agents to behave like a human? Can computers build something like common sense or proper real-life knowledge? Furthermore, the video encourages thoughts about our own mental model, emphasizing its existence and trying to get to the bottom of the workload done internally in everyone's brain. Thinking about the vicinity this reinforcement learning approach has to real life makes it tangible but still unapproachable. The way three fairly simple networks are able to mimic this complex interaction is really fascinating.

Computational Motivation

World Models are an approach to tackle some major flaws which algorithms belonging to either of the two classes of RL algorithms mentioned above typically exhibit. Which problems these are specifically and how world models can help to overcome them is what we will elaborate on in the following.

Model-based problem: We do not have access to complete knowledge of the environment

On the one hand, traditional model-based RL algorithms - i.e. those that have access to a complete description of the environment dynamics - might exhibit extraordinary performance over the domain of problems where the environment dynamics can be described and accessed by a computational agent completely. However, they are not generally applicable to any given problem, i.e. when the models are too complex to define explicitly or simply not known. World Models can be regarded as belonging to the class of model-based RL algorithms, but instead of requiring complete a priori information about the environment dynamics, an environment dynamics model, the so called world model, is learned iteratively during or prior to training the RL agent as a part of the computational agent's architecture. The task of this world model - typically implemented via a neural network (NN) which incorporates some recurrent neural network (RNN) variant - is then to predict a representation of the future environment state(s) given an action taken by the agent at the current state of the world (or a compressed representation thereof) and the past state(s) of the RNN. Thereby, i.e. by introducing the task of constructing or approximating a proper model of the environment to the algorithm, the World Model approach is more generally applicable to RL problems than other traditional model-based approaches.

Model-free problem: We want the agent to learn quickly, i.e. generalize well from few real interactions with the environment

On the other hand, agent models for model-free RL algorithms such as PPO or Actor-Critic Methods do not contain structures which are intended to explicitly capture knowledge about the environment dynamics, making them at least as generally applicable as the (learned-world-)model-based alternatives. However, a major downside of these algorithms is that they usually require lots of interaction (trajectories) data from within the actual problem-specific environment (much more than humans would) in order to enable an agent to perform well over a given task (Kaiser et al., 2020 [Kai+20]). One possible reason for our human ability to learn so fast might be that we do not only make decisions about actions based on current and immediate observations of our surroundings, but more or less explicitly also use our memory of past experiences to

predict the outcomes of an action on our surroundings, thereby predicting observations we will likely make in the future (c.f. Hafner et al., 2018 [HS18]). In most of the approaches which make use of learned World Models, the problem of sampling-data inefficiency can be overcome, because World Models enable to capture state-action-transition probabilities over time (Weber et al., 2018). Thus, a World Model can be used to train the RL agent within a simulation of the real environment, just as humans might form decision making policies by imagination or when dreaming (cf. [Haf+20]). Simulation of the environment in World Model approaches is typically accomplished by using representations of future state observations (given an initial input observation) generated by the learned world model only (cf. [Haf+20],[HS18]). Hybrid approaches have also been proposed, in which model-free RL and learned world model networks are combined with model predictive control (MPC) strategies to increase learning efficiency, e.g. by Nagabandi et al. in 2017 [Nag+17].

General problem: Dealing with partially observable environments (given limited computational resources)

Another more general issue in RL comes with partially observable MDPs (POMDP). In POMDPs, a single observation of the environment does not capture all information about the actual environment state. According to Jürgen Schmidhuber, "realistic environments are not fully observable". Thus, according to Jürgen Schmidhuber, "General learning agents need an internal state to memorize important events in case of POMDPs". And exactly this is what the world model approach tries to accomplish. We know that we have only limited computational resources to train a RL agent to behave intelligently in an initially unknown environment (i.e. to learn a good policy), but we also know that the given task is a POMDP problem. Having an agent learn a policy as with standard policy-gradient based methods from - in this case - raw spatial data (state observations) and information from previous timesteps would require lots of computational resources due to large numbers of trainable parameters in and end-to-end training of the agent model. In the world models approach, learning a good compression of inputs over space and time (which might require a decent amount of parameters in the model to be tuned) is the main purpose of the world model module in the agent. By decoupling this task from learning the actual policy, the whole problem can be solved more efficiently (regarding the required number of sampled interactions of the agent with the real environment).

Cognitive Plausibility

Apart from computational aspects, world models are also of theoretical interest regarding their similarities to how we believe humans to make decisions in real life. As humans we are able to perceive our surroundings through many

different senses. By interacting with the environment, we further learn to predict the consequences of our actions. Everything we perceive from observations or through third parties is taken in, understood and reaches our brain. With that ability, we create a mental model which represents this environment, how it works and behaves in certain situations. Importantly, the mental model is based on beliefs and interests, not facts. It can and does get updated to simplify the complexity of the whole world into understandable chunks. Usually, all this happens within seconds and with the agent - i.e. ourselves - not even being aware of it. In addition, this mental model now can be used to learn from it. Because the mental model knows the behaviour and can predict several states, the agent does not need to perceive and consciously plan every action. Think about the following two examples. First, imagine standing on skis for the first time in your life. It will most likely feel really weird, unstable and just new for you. In that very moment, your mental model of skiing has just started to develop. The first couple of times when you get on your ski you will have to learn how the snow behaves in interaction with your ski, what will further happen when you ski at that specific speed and angle. During this process of learning, your mental model becomes clearer and more extensive. Finally, after some time you will not have to consciously think about what will happen in which specific moment. Often, there might not even be time to think things through, you will just have to react and interact with the environment. Now, this environment is fully represented within your mental model and you gained the ability to use that imagined model to learn your behaviour. The second example is concerned with artificial curiosity, which helps to learn how the real world works. To build an extensive mental model, humans need new challenges and new input, which is one of the goals of curiosity. However, while building a mental model other goals are present as well. Although one might be curious what will happen when touching a high voltage line and has never seen anybody do it, the already existing mental model will prevent doing it. One does not exactly know how it would feel, but at the same time does not want to know. The mental model already helps with predicting the consequences of certain actions and becomes more complete through curiosity. World models are now exactly this, including two major components. Firstly, the vision part, for instance the human's eye to perceive, encode and reduce complexity of the environment. And secondly, the memory part to integrate the knowledge and create the mental model, which could be represented by the human's brain. The internal world model is now followed by a controller to select the best action based on knowledge from the world model, in real-world applications the controller would be the human's body.

Implementation and Results

We will proceed by describing the proposed implementation of the world models approach (task, agent structure and training procedure) by Schmidhuber and Ha [HS18], and more specifically what we effectively did for our reimplementation. Firstly, we briefly describe what the task was for the agent. Afterwards, we will elaborate more on the agent structure with a focus on the World Model part (Vision and Memory Module), including a few visualizations of examplary outputs. We describe the Controller module and training thereof (using PPO) and discuss the final performance of the trained agent on the task against the background of related literature within the scope of our DRL project documentation.

The Task

The world models agent was originally tested on a (continuous control) Car Racing and Doom task, both of which are available as environments in OpenAI's gym framework [Ope]. For the scope and purpose of our project(s), we will only focus on the specific application of the approach to the CarRacing-v0 environment. In this environment, the agent has to steer a race car over a road as fast as possible within a fixed time. The tracks are randomly generated for each trial, and the agent is rewarded for visiting as many tiles as possible in the least amount of time. The agent controls three continuous actions: steering left/right, acceleration, and brake.

The Agent

Generally, the procedure of training an RL agent (as shown previously in figure 1 and in figure 2 with the world models approach is as follows:

1. Sample trajectories (i.e. raw spatial and temporal information) from random interactions with the environment.
2. Train the "Vision" module (a Variational Autoencoder) to compress image pixel-frames into latent vector representations.
3. Train the "Memory" module (a MDN with LSTM head) to learn environment dynamics and predict the next state's latent vector representation given the VAE-encoded state observations from the trajectory samples, and the corresponding action taken by the random agent.
4. Train the "Controller" module (originally using evolutionary algorithms). This can be done in the real or simulated environment using the Memory module to predict state-action transitions.

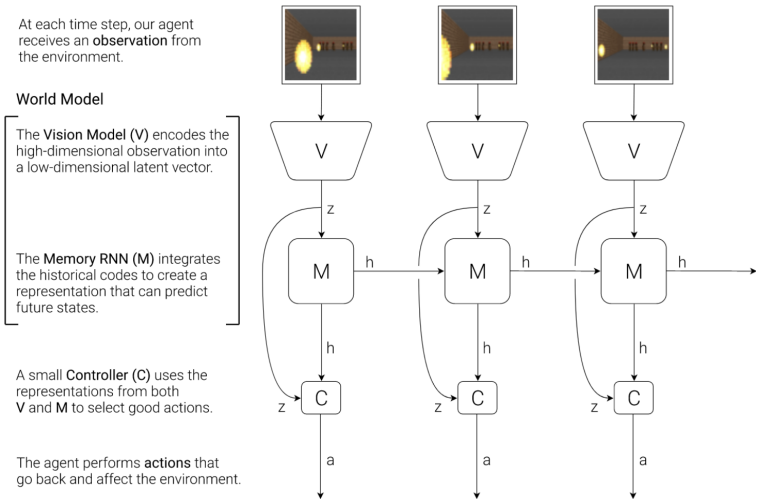

Figure 2: Training of the World Models agent (taken from [HS18])

We will describe how we dealt with the first three steps in more detail using the previously described car racing task in the following paragraphs. The final step again can be read up in the DRL project documentation.

Step 1: Generating Trajectories The goal is to here learn a good spatial and temporal representation of data that we can pass as input to the Controller (the policy-learning) module of the agent. To do so, we need to collect raw data of state-action-transition observations from interactions with the environment first. This data is used to train the world model's (=compression) modules (VAE and Memory) on. Therefore, the world models approach starts with having an initial agent repeatedly take random actions in the environment (until a total of 10000, these so-called "random-rollouts" of the environment are collected). Each raw state observation is a frame of 64x64 pixels, which is quite a lot of data to process at each single timestep. In fact, it is possibly more than effectively needed to capture the features of a state that are relevant or decisive for solving a given task. Passing them as input information to a policy-learning module might be overwhelming for the agent assuming it stays unfiltered. Recognizing that our agent cannot be trained on the full-sized input observations, we next need a method for spatial compression of the raw input observations (i.e. pixel frames).

Step 2: World Model - Vision The authors' method of choice here was to make use of a Variational Autoencoder (VAE) NN model architecture, which we got to know in the IANNwTF course as well. We adopted the implementa-

tion details about the specific encoder-decoder architecture structure as it was described in the paper (see figure 3).

The VAE/Vision/V model as the first part of the agent also constitutes the first part of what is called the World Model. A raw input observation enters the final agent model through V, which builds a compressed form of the multi dimensional input. As stated above, the necessity arises due to the agent not being able and usually not needing to cope with too high dimensionality. The VAE allows our agent (i.e. the memory and controller parts) to receive the lower dimensional, abstract and latent representation of the observation constructed by the VAE. Moreover, the V model never receives any rewards, thus learns only from observations of the environment and in interaction with the other parts of the agent.

Variational Autoencoders generally are a major part of world models, used to convert the images from the real environment into latent representation vectors.

In machine learning, Autoencoders are widely used to generate compressed representations of data, while still carrying most important information. These representations are commonly called embeddings, which capture the intrinsic dimensionality of the data. This dimensionality is generally beyond what humans are able to explore, Autoencoders are however able to discover and exploit patterns as well as nonlinear dependencies in the training data. But apart from dimensionality reduction and feature detection Autoencoders are capable of much more. Taking the idea further, variational Autoencoders can indeed be considered as generative models, as the decoder part is able to randomly generate new data which looks very similar to the training data. A classical Autoencoder always consists of an encoder and a decoder, connected by a bottleneck in between to constrain the amount of information handed over. The Encoder part of the model receives the input sample, which could be for instance an image or a sentence. It learns to recognize patterns and converts the input to an internal representation, the embedding vector. This latent space is usually of smaller dimension as the input but still needs to capture the important features of the training data which are sufficient for the upcoming approximate reconstruction. Second part of the Autoencoder is the decoder, which is responsible for reconstructing the input back from the latent space. The only input it receives is

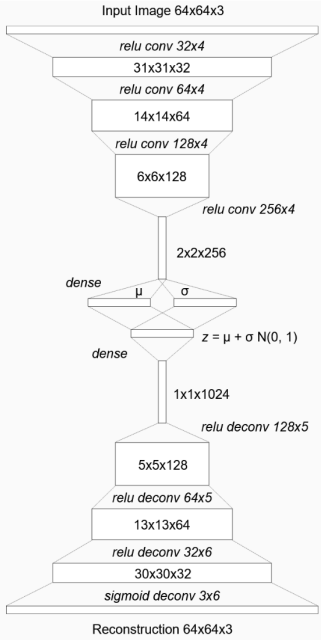

Figure 3: VAE model architecture (taken from [HS18])

the embedding vector, which was earlier created by the encoder. As output of the decoder, the reconstructed input is supposed to be a copy of the real input (see figure 4 for reconstructed input). Thus, Autoencoders are usually trained by minimizing the reconstruction loss, which captures the difference between original input and recreated output. We trained our VAE model for 42 epochs, over which the loss dropped from initially 300 to approximately 159.6, which is - to be honest - not really great compared to results from others, e.g. a final loss of 30 as reported by Adam Green [Gre19].

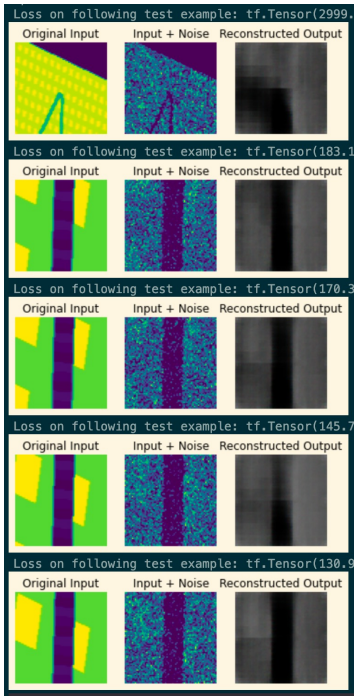

Figure 4: compared images, reconstructed by the VAE (can be found in the VAE_training.ipynb file)

In addition, there are variational Autoencoders (VAE). As mentioned before, those are able to work as generative models to create artificial samples similar to the ones from our training dataset. To realize this, the VAE learns a distribution for the vectors in the latent space. Their embedding space is by definition continuous and the latent variables entering the decoder are drawn from the probability distribution. The encoder of VAE takes the input from the environment and returns mean and standard deviation parameters for a probability density, for instance Gaussian. One vector sample is drawn from this gaussian distribution and fed into the decoder to receive an output image. The major difference to vanilla AE is that the latent representation vector is drawn from a distribution encoded earlier. Apart from that, we need to ensure a well trade-off between encodings which are close together while still being distinct, enabling smooth interpolation and the construction of samples not seen before. To achieve this trade-off, the distribution of the latent representation between encoder and decoder must resemble a Gaussian distribution. This is done by minimizing the Kullback-Leibler divergence, to have the probability distribution built by the encoder being similar to the target distribution. By creating this simple regularization term, similarity between the embedding distribution and the normal distribution is ensured.

Step 3: World Model - Memory Secondly, we want to incorporate data from the history of previous timesteps to account for the POMDP aspect of the problem. Because it would not be computationally and memory-wise feasible (and neither conceptually desirable) to keep all previous input observations in

memory and pass them as input to the "decision-making part" of the agent, aka Controller, at each timestep, we also want to compress the data along the time dimension.

Thus, the second component of the agent is the memory module (M), which consists of a recurrent neural network (RNN), more specifically an LSTM, with a mixture density network (MDN) output layer. During training, the RNN gets sequential input in the form of a compressed state observation (z) (sampled from the VAE training results) and a corresponding action a which had been taken by the agent in that state, and outputs a set of values that can further be passed to the MDN layer to produce the parameters of $|z|$=32 Gaussian Mixture probability distributions with 5 kernels each, from each of which (i.e. technically for each dimension of z) we can sample a single value to get a prediction for a possible next compressed state observation for each dimension of the next latent state representation (z').

The goal of M is thus to learn a function that maps from z and a given the previous observations and actions it has been presented with that can be used to predict future states, i.e. a probabilistic model capturing the environment dynamics, which should allow the agent to make better (informed) decisions. Specifically, it should learn to predict the consequences of the agent's actions on the environment and encode transition probabilities for given states and actions, given the sequence of previous states and actions in a trajectory. Accordingly, it emits the probability distribution over possible latent representation vectors in the next timestep. The hidden layer of the recurrent neural network should finally contain all relevant information about the current situation and many future steps. The second input to the Memory module as described in the paper is a temperature parameter, which is able to control the certainty of the network over the predicted future states by introducing noise to the network's predictions (see middle column in figure 4). Increasing the parameter when sampling the next latent representation leads to higher uncertainty, making the environment less predictable. However, sampling from the MDN output parameters might only be required later for training the controller but not for training the Memory itself.

Thus, instead of taking a concrete next state (z') prediction as output of our Memory model's `__call__()` function, we simply return the parameters for the Gaussian Mixtures. Those can afterwards still be used to sample a possible z' from the distribution they define, but also to calculate the loss function of the Memory model during training. The loss function is described as the negative log-likelihood of the next latent target vector as a prediction from Gaussian Mixtures, defined by the parameters returned after a call to the model with a given $[z,a]$ input and last state. We can still pass the temperature parameter to the sampling function used to generate concrete predictions of z' when training the controller, and even adjust it dynamically to vary the amount of stochasticity or noise in the predictions and avoid the Controller

to learn behavior that exploits imperfections of the World Model to get high rewards in the simulated environment.

More on MDNs

> "The average of several solutions is not necessarily itself a solution."
> Bishop, 1994

Bishop [94] initially introduced the notion of Mixture Density Networks in 1994 as an alternative to simple feedforward networks (FFNs) to approximate better solutions for inverse problems, i.e. problems where the roles of input and output variables are interchanged, and the mapping we want to learn from a single input value to output values can often be multi-valued for the output.

If we now use simple FFNs to output single values for each output unit, we can only learn a conditional average over the output which might be a poor fit for the possible target data distribution. Mixture Density networks are an attempt to solve this problem by learning the output values as a mixture of several identically but independently distributed random variables (RV). Therefore, we first define the number of kernels we want to use to approximate the data distribution function, and declaring the network's output values as parameters associated with a kernel function (i.e. one mu, sigma (assuming a diagonal covariance matrix), and a prior or so called "mixture weight" for each kernel or RV in the "mixture") for each element in the desired output vector. Figure 5 shows the kernel fits (means and sigmas) of an MLP with single kernel output layer against the learned kernel fits of an MLP with 10 kernel output layer for samples from an inverse sine function (for further details on this take a look at our corresponding `memory_development_and_testing|notebook`).

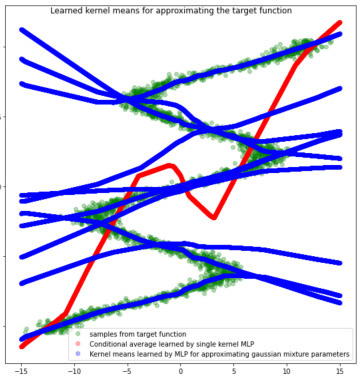

Figure 5: Single vs. 10 kernel MLP fits

There are several reasons why we might want to use a MDN also as an output layer in the case of the car racing problem instead of the deterministic statistics (conditional average and variance) we can obtain by a simple (i.e. single kernel output layer) MLP.

Firstly, we wish to predict continuous values for the next latent vector's elements.

Secondly, we have to make predictions under uncertainty. There are several possibilities for how the next frame might look like (and thereby be encoded in the latent space) after a given state observation is made and a specific action is taken. Thus, we would ideally like to model how uncertain we are about predictions (which can be encoded in the priors).

Furthermore, the predictions are conditioned on an input observation (here: the latent representation of the previous state's observation and corresponding action that was taken). Ideally, we would not like to obtain a single answer for the question of what might be the latent representation of the next state observation, but rather a range of answers to assess the probability of each individual answer. To put it in a nutshell: We are looking for a probability distribution over a range of answers given the input.

Originally conceived by Bishop, MDNs are a quite interesting topic beyond the presented work, and have recently been used for a variety of applications, e.g. to generate speech for Apple's Siri in iOS 11, or to generate artificial handwriting [Gra14] or possible continuations of sketches. There are multiple blog posts dedicated to implementations of MDNs using tensorflow (e.g. [Bor19], or [Bon]). We finally used the implementation of an MDN layer for use with tensorflow and keras by Charles Martin [Mar21] to set up a notebook ourselves to play around with implementing MLPs with MDN output layers and test them for the purpose of developing our Memory module.

The final Memory model and training code was then derived from our model definition in the development and testing notebook, the proposals from the original paper by Ha and Schmidhuber[HS18], and a well documented reimplementation of the world models approach by [Gre19]. We trained our memory model on the results from the sampled trajectories and the respective VAE outputs as described by Ha and Schmidhuber [HS18] and done by Green [Gre19].

Figure 6 shows some decoded state observation representations from predictions of future latent vectors for a single step into the future after a complete training epoch. We recognize that our model predictions seem quite imperfect when compared to the corresponding actual decoded state representations by the VAE but also seem somewhat reasonable as state observations.

Parameter Counts and Model Summaries

Finally, we present a summary of the number of parameters in our implementation of the VAE (see figure 7) and the Memory/MD-RNN (see figure 8) model and provide a comparison to the number of total number of each model's parameters in the original implementation for the car racing task in Table 1.

For the VAE model, the parameter counts are almost identical, whereas our

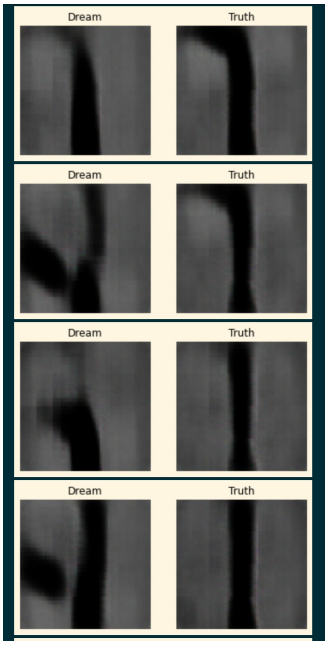

Figure 6: Memory visualizations

```
Model: "vae"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_1 (Sequential)    (None, 64)                754720
_________________________________________________________________
sequential_2 (Sequential)    (None, 64, 64, 3)         3592803
=================================================================
Total params: 4,347,523
Trainable params: 4,347,523
Non-trainable params: 0
_________________________________________________________________
```

Figure 7: VAE model summary

```
Model: "Memory"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_layer (LSTM)            multiple                  299008
_________________________________________________________________
td_mdn (TimeDistributed)     multiple                  83525
=================================================================
Total params: 382,533
Trainable params: 382,533
Non-trainable params: 0
_________________________________________________________________
```

Figure 8: Memory model summary

Model	original	our reimplementation
VAE	4,348,547	4,347,523
Memory (MD-RNN)	422,368	382,533

Table 1: Model Parameter Counts

Memory model has substantially less parameters than the model used in the original paper. We are not completely sure from where this difference arises, since the exact architecture was not described in the paper, i.e. there might be an additional dense layer in the memory model's original implementation (presumably between the LSTM layer and the MDN output module). However, since we had only limited computational resources to train our networks, and our model seemed to be powerful enough to generate somewhat reasonable predictions and turned out to be suitable to use for training the Controller afterwards, we did not add complexity to it just to get a closer matching to the original parameter counts.

References

[94] "Mixture Density Networks". In: Neural Computing Research Group Report (Feb. 1994), p. 25. URL: https://publications.aston.ac.uk/id/eprint/373/1/NCRG_94_004.pdf (visited on 03/20/2021).

[Gra14] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *arXiv:1308.0850 [cs]* (June 5, 2014). arXiv: 1308.0850. URL: http://arxiv.org/abs/1308.0850 (visited on 04/17/2021).

[Nag+17] Anusha Nagabandi et al. "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning". In: *arXiv:1708.02596 [cs]* (Dec. 1, 2017). arXiv: 1708.02596. URL: http://arxiv.org/abs/1708.02596 (visited on 04/03/2021).

[HS18] David Ha and Jürgen Schmidhuber. "World Models". In: *arXiv:1803.10122 [cs, stat]* (Mar. 28, 2018). DOI: 10.5281/zenodo.1207631. arXiv: 1803.10122. URL: http://arxiv.org/abs/1803.10122 (visited on 03/02/2021).

[Bor19] Dr Oliver Borchers. *A Hitchhiker's Guide to Mixture Density Networks*. Medium. Feb. 15, 2019. URL: https://towardsdatascience.com/a-hitchhikers-guide-to-mixture-density-networks-76b435826cca (visited on 04/02/2021).

[Gre19] Adam Green. *World Models (the long version)*. The intersection of energy & machine learning. Dec. 21, 2019. URL: https://adgefficiency.com/world-models/ (visited on 04/17/2021).

[Haf+20] Danijar Hafner et al. "Dream to Control: Learning Behaviors by Latent Imagination". In: *arXiv:1912.01603 [cs]* (Mar. 17, 2020). arXiv: 1912.01603. URL: http://arxiv.org/abs/1912.01603 (visited on 03/15/2021).

[Kai+20] Lukasz Kaiser et al. "Model-Based Reinforcement Learning for Atari". In: *arXiv:1903.00374 [cs, stat]* (Feb. 19, 2020). arXiv: 1903.00374. URL: http://arxiv.org/abs/1903.00374 (visited on 03/15/2021).

[Mar21] Charles Martin. *cpmpercussion/keras-mdn-layer*. original-date: 2018-06-16T14:13:36Z. Apr. 16, 2021. URL: https://github.com/cpmpercussion/keras-mdn-layer (visited on 04/17/2021).

[Bon] Christopher Bonnett. *Mixture Density Networks with Edward, Keras and TensorFlow — Adventures in Machine Learning*. URL: http://cbonnett.github.io/MDN_EDWARD_KERAS_TF.html (visited on 04/17/2021).

[Ope] OpenAI. *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com (visited on 04/16/2021).

[Sch] Jiirgen Schmidhuber. "Making the World Differentiable: On Using Self-Supervised Fully Recurrent Neural Networks for Dynamic Reinforcement Learning and Planning in Non-Stationary Environm˜nts". In: (), p. 28.