eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Science

Computation

End of Degree Project

World Models for Text Based Agents

Author

Jon Perez Etxebarria

Directors

Aitor Soroa and Gorka Azkune

Contents

List of Figures

List of Tables

1. CHAPTER

Introduction

Since the dawn of first and oldest Abrahamic religion, Judaism, humanity has strived towards the aspiration that is the creation of an intelligent automaton capable of obeying their very command. This first automaton took the name of 'golem' and in the very early texts, even Adam was intended as a golem remarking the scope that this supposes. In a more modern era we have substituted this esoteric names and concepts by a new science called Artificial Intelligence tasked with the creation of 'agents', the spiritual successor of the golems.

Over the years we developed different approaches, some that are rule based and whose goal is to mimic human reasoning. The agents created under this approach are called Knowledge Based Systems [Akerkar and Sajja, 2009]. Another one of this families, is Machine Learning, algorithms based on either statistical or mathematical concepts which are able to be trained on data and learn the patterns that lie within. In order to predict future instances of similar data. Finally, we arrive at the most ambitious of the attempts to create the modern age golem, neural networks.

Based, as the name implies, in our very own cognitive system, in which a neuron is activated if met with a particular signal coming from a particular input. In the same way this approach tries to mimic this system in as much as technology and computational power allows it. There are many different architectures of neural networks, and even in this we are tied to our own nervous system, since each architecture of neural networks its specialized in a particular task or input format.

In the many attempts to create the modern golem we have mimicked many of our own senses and human capabilities. When we developed Convolutional neural networks based on the mathematical concept of the convolution instead of the matrix multiplication. We found a neural architecture that outperformed the previous ones in solving problems that had to deal with images or two dimensional data [Krizhevsky et al., 2012]. This field tasked with making agents able to work with a visual input would end up being called computer vision.

Perhaps the most impressive human quality is our reasoning. Some studies have suggested that human reasoning is intrinsically tied to our ability to speak, [Klein, 2017] so naturally we started to build neural models with the objective to understand and replicate our language. And so, neural networks came into the field of Natural Language Processing. It is to this particular field that this project intends to contribute.

The human brain has a natural talent for information abstraction and extrapolation. This allows us to generate rather simple models and maps that represent very complex actions and situations. This can be easily seen with any action that requires a high reaction speed. If you ask a boxer how they were able to dodge so many punches with such precision and speed, you will not receive an analytical answer that decomposes every part of the situation. Rather than that, you will most likely receive an answer that references their training. We humans can be trained to build an intuition and model of the situation that surrounds us, the questions at hand is: could a machine do it?

In 2018 David Ha and Jürgen Schmidhuber tried to answer this question using computer vision. In a project that took the name of 'World Models' [Ha and Schmidhuber, 2018] they were able to train a deep neural network that was able to sample and select the most likely future scenario. This helped it beat the different challenges presented by adding this memory or intuition capability to a quite simple model. The question remains if this ability can be extended to an agent capable of understanding human language. In this project I intend to present the first approach towards building an agent that is able to follow text instructions and complete those task by making use of its pre-trained conception of the environment that surrounds it . This new added capability to imagine or predict the future should lead to the agent taking better and more informed decisions since it would able to infer what could happen based on hat action it takes and its actual state.

One of the main problems any NLP researcher finds himself with, is the lack of good and appropriate data to train the models with. In 2018 Microsoft Research published a new environment with the goal of building a suitable set up to train and evaluate reinforcement learning agents on a text based environment. Therefore, providing a potential solution to the aforementioned problem.

TextWorld [Côté et al., 2018] is a sandbox learning environment, specifically designed for text based models using reinforcement learning to train themselves. It enables the creation of games with specific task as the objective. TextWorld allows us to create controlled environments parameterized to our liking. Both the description of the surrounding environment and the actions available in the current step are in text format which means in order to create a successful model said model will have to develop some basic understanding of the human language.

Since the model will need to be able to understand human language, we need to be able to put sentences in a manner that is able to be fed to a neural network. This is where embeddings come in, one of the pillars of NLP. Embeddings are fixed sized vectors that encapsulate the semantic information of a body of text, be it words or sentences. In this paper we decanted for the Universal Sentence Encoder, or USE for short, [Cer et al., 2018], developed by Google. USE provides a solution that allows us to use the textual inputs and compare how semantically similar two sentences are. The Universal Sentence Encoder is a model capable of encoding sentences into numerical vectors that retain semantical information from the original sentence. We can check how similar two sentences are by checking the two embeddings' semantic similarity. This is done by first of all computing the embeddings of said sentences, after that, we check for the cosine similarity of those two embeddings, the closer the answer is to one, the more semantically similar the two sentences are.

In this project I intend to present our attempt to build an agent with an intuition based on human language. This intuition will be built by the agent by interacting with the world in a controlled environment. Our goal is to create an agent that by playing in text games created via TextWorld it is able to form an intuition of the world surrounding it. Using this intuition it should try to outperform other models lacking this capability.

2. CHAPTER

Objectives

In this project I set out to create a text based agent with the ability to predict the future inside a controlled environment as my main goal. This project was inspired by the project that David Ha and Jürgen Schmidhuber developed 2018 under the name World Models [Ha and Schmidhuber, 2018]. In said project they present a complex agent whose core concept is the ability to generate a preconception of the world that surrounds it and make decisions based on its intuition.

Since said goal cannot be achieved in a vacuum, many tangential goals raised while deciding on the tools and framework. First, I decided to use TextWorld. TextWorld is a text based environment designed to facilitate reinforcement learning training for text based agents. This made it a prime candidate to develop this project. Since TextWorld is meant to use reinforcement learning this topic also needed to be researched. With the environment decided upon I moved towards understanding David Has and Jürgen Schmidhubers work in the world models project. Understanding their work was key in order to be able to adapt it to text. Putting it all together:

- Researching the TextWorld environment 3
- Develop an understanding of Reinforcement Learning 3.2
- Studying and researching the work done in the World Models project 3.3
- Adapt the World Models agent to work in text based environments 4

- Train and test the agent in TextWorld 6

With all of this in mind the sole objective of the project could be summarized as: understanding the the key topics and state of the art related to them in order to adapt and use the acquired knowledge to build a text based agent able to model the world that surrounds it in a control environment. With the final result being the implementation of said agent.

2.1 Reach

The reach of this project starts by the understanding of all the concepts mentioned at the beginning of this chapter, continuing through the implementation of the different components. With the end goal of having a functioning agent at the end of the project.

2.2 Risks

There are many moving pieces in this project and every moving piece represents a risk.

2.2.1 Research

Starting with the research, this was one of the most solid parts since I parted from already finished projects and research. The biggest risks involved in research are not being able to understand the topics at hand or not understanding how to translate the theory into an actual implementation of the solution.

2.2.2 Implementation

Continuing with the implementation, this part probably represents the most amount of risks. One of the potential problems is that the solution presented is that some of the models were implemented using TensorFlow (TF) , a framework to build neural networks developed by Google, and even in the use of TF, two very different versions of this framework were needed. On top of that, some of the other models were implemented using PyTorch, a framework for the implementation of neural networks by Facebook.

Another risk related to the implementation is the change from computer vision and the field of images to natural language processing and text. This risk in not only regarding the models and their architecture this risk expands to data. Since I changed the environment to TextWorld, the data needed to come from within. This supposed an increase of the maximum number of actions from 2-3 to more than 400 unique actions. Staying on the topic of actions some other change that could cause problems came into play, in the original 2018 World Models [Ha and Schmidhuber, 2018] project all the actions are available to the agent at all times. In this project only a small part of those 454 actions are available at any given moment, numbers may vary between four and fifteen available actions at any given time.

The last risk I will mention regarding the implementation is the fact that there was always a risk of not being able to code a solution that worked. Or not being able to piece it all together because of the different dependencies.

2.2.3 Results

Finally, I arrive at the results, as with any project that has within its purview the development of a product or in this an agent with a goal in mind, there is always the risk of said agent not meeting the expected results.

2.3 Schedule

The hole project lasted for ten entire months. During the beginning of the project the main objective was the research of the already implemented state of the art technologies that could be applied for the project. After the research was done the implementation part took place with the great majority of this time being spent of the memory component since it was the most complex of all the components and required extra detail and attention. This is explained to further detail in the Gantt diagram in Figure 2.1.

2.3.1 Gantt Diagram

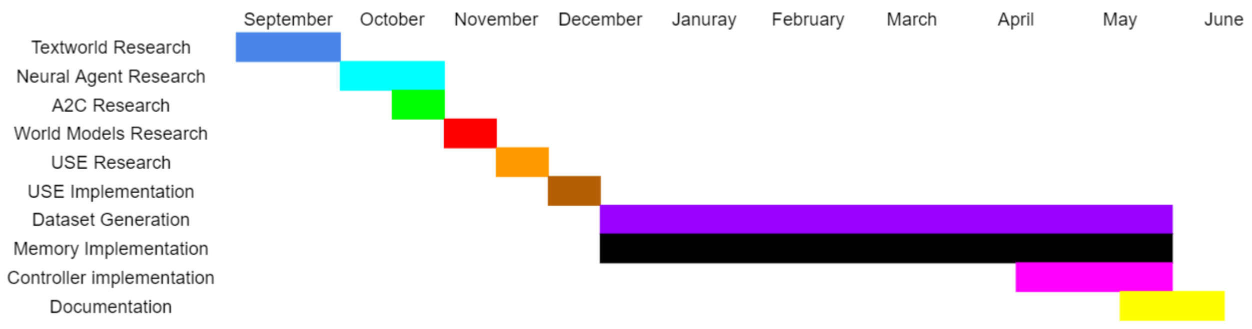


Figure 2.1: Gantt diagram

3. CHAPTER

Background and Sate of the Art

In the first two chapters I have made plenty references to previous works and projects. In this chapter I intend to take a more in depth and detailed look into this references.

3.1 TextWorld

TextWorld [Côté et al., 2018] is an extensible Python framework for generating text games. Text based games are games in which the game environment and the player interact solely in text. The environment gives the player a description of what surrounds him. The player in turn will input a text action, changing the state of the world. A couple of classical text games are the Zork saga [Lebling et al., 1979] or colossal cave adventure [Sharma, 2008]. The main idea behind TextWorld is to create an environment where researchers can train and test reinforcement learning models in skills such as language understanding.

3.1.1 Interacting with TextWorld

Despite TextWorld being a python package it can be launched from the terminal without any mention of python. Nonetheless, in this project I specifically want to play and work in those environments using python. TextWorld offers and interface to interact with Gym

[Brockman et al., 2016], which is a reinforcement learning toolkit developed by OpenAI. Using this toolkit, I can load TextWorld maps and games and interact with them. TextWorld will give us an observation in string format as well as a list of available commands that I can choose us out as actions. After every action the environment will decide if the action deserves a reward and the case that it does the agent receives a numerical reward.

3.1.2 Games types

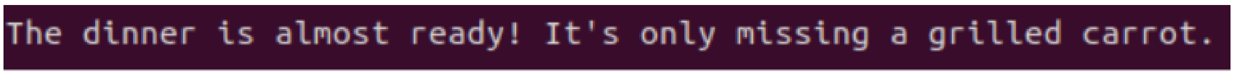
There are many different types of games in TextWorld. In this section I will explain the concepts behind them.

Coin Collector : This is the simplest game mode inside of TextWorld. The game consists of a set of rooms with nothing in them. The goal is to collect a single coin that spawns in a room with no other distractions.

Treasure Hunter: It is a similar concept to the coin collector. in this game the agent is tasked with the collection of a particular items that is located in a random location. Other items are spread through the rooms acting a decoys. If the agent picks a decoy it loses the challenge.

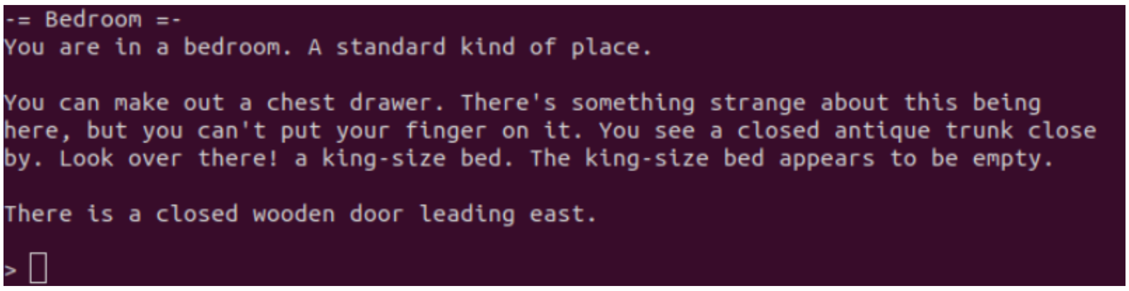
Simple Game: Despite the name this is the most complex game type in the game. You are giving a food name and a way of cooking it in a bigger and more complex house environment. The agent needs to find the object it needs to cook and the cook it in the appropriate way in order to complete the task. The house is composed of many different room type that contain locked doors, open doors, vaults even keys that unlock doors.

One example of the initial objective description you may get when starting this type of game of this types can be seen in Figure 3.1, followed by an observation of your surroundings in Figure 3.2. This was the type of game I used to train and test my final agent.



```
The dinner is almost ready! It's only missing a grilled carrot.
```

Figure 3.1: Objective description in TextWorld

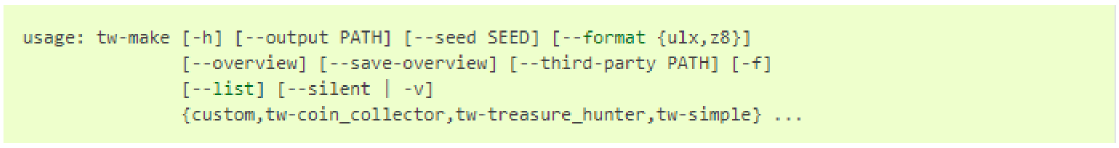


```
-= Bedroom =-
You are in a bedroom. A standard kind of place.

You can make out a chest drawer. There's something strange about this being
here, but you can't put your finger on it. You see a closed antique trunk close
by. Look over there! a king-size bed. The king-size bed appears to be empty.

There is a closed wooden door leading east.

>
```

Figure 3.2: Observation given by TextWorld

Map Creation

The map creation in TextWorld is done with the 'tw-make' command followed by a series of parameters as showcased in Figure 3.3.



```
usage: tw-make [-h] [--output PATH] [--seed SEED] [--format {ulx,z8}]
               [--overview] [--save-overview] [--third-party PATH] [-f]
               [--list] [--silent | -v]
               {custom,tw-coin_collector,tw-treasure_hunter,tw-simple} ...
```

Figure 3.3: tw-make commands

Below the explanation to some of the main parameters can be seen.

subcommand: Specifies the game type, the options are tw-simple, tw-treasure_hunter and tw-coin_collector .

–regards: the frequency of the rewards. There are three options, dense, balanced and sparse.

–output: Path to the directory where to store the map.

–seed: It conditions the map creation. Random by default.

–list: It prints the to-do list.

overview: It prints an overview of the map.

–blend-description: The descriptions will be blended in different parts of the given observation. False by default.

–ambiguous-descriptions: It gives broader descriptions of the objects instead of being specific.

–nb-parallel-quests: Number of parallel quest and agent can have at the same time.

–quest-length: Number of minimum required steps to complete the task.

–quest-(min/max)-depth: Minimum or maximum number of actions required.

–level: Difficulty parameter.

3.2 Neural Agent

In this research project I made use of the basic neural agent provided as an example in the TextWorld notebooks. This agent uses the A2C reinforcement learning algorithm for training and has two main components to it, one being the actor and the other the critic. This agent is used as an educational demonstration on how to build a reinforcement learning agent using the TextWorld interface. In the context of this project it will act as the baseline to test my agent against. It will also serve as the mold to build my final agent out of.

3.2.1 A2C

Actor Critic [Rosenstein et al., 2004] is a type of reinforcement learning algorithm that has two main components. The Critic which is the component that estimates the value function, the value function returns the value of an objective function in a maximization problem, in this particular scenario, I want to maximize the reward, as defined by the parameters of the problem, which in this case, those would be the possible actions I can take. On the other hand, I have the Actor, this part updates the policy gradients in the direction suggested by the Critic. This two functions are parameterized with neural networks.

A2C is a policy gradient algorithm. The agent will interact with the environment and collect a set of state transitions, these transitions in this case are different observations and rewards given by TextWorld. After n steps or at the end of an episode the critic calculates the required updates to the weights and applies them.

In A2C or Advantage Actor Critic, a baseline function is used. This baseline function sets an expectation for the reward given the current situation. The reason this function is important is because it will be used to subtract it from the actual reward the agent receives. By this subtraction I make the final reward the agent receives smaller and thus making the changes to the weights smaller. By subtracting the baseline, I compute the value of how much better is to take a certain action compared to the average move. This is called the advantage function $A(s_t, a_t)$, which can be seen below.

$$A(s_t, a_t) = Q_w(st, at) - V_v(st, at)$$

Where V is the baseline function and Q is the reward received.

Architecture

In this section I would like to dissect the architecture or the neural agent into its three main components. As has been explained in the TextWorld section 3, the environment gives an observation to the player and the player inputs an action back. The available commands are encoded by an embedding layer and a Gated Recurrent Unit [Chung et al., 2014], which is a type of recurrent neural network. on the other side the observations are encoded by a different embedding layer and two GRUs. The critic will infer the estimated value from this second GRU. Now I arrive to the third component, the linear ‘phase. I create command-observation pairs were each individual command is concatenated with the observation. This matrix is run through an attention layer and different activation functions to end up being sample with a multinomial layer to get the index of the command that the agent is going to select. In Figure 3.4 an abstraction of the architecture is represented.

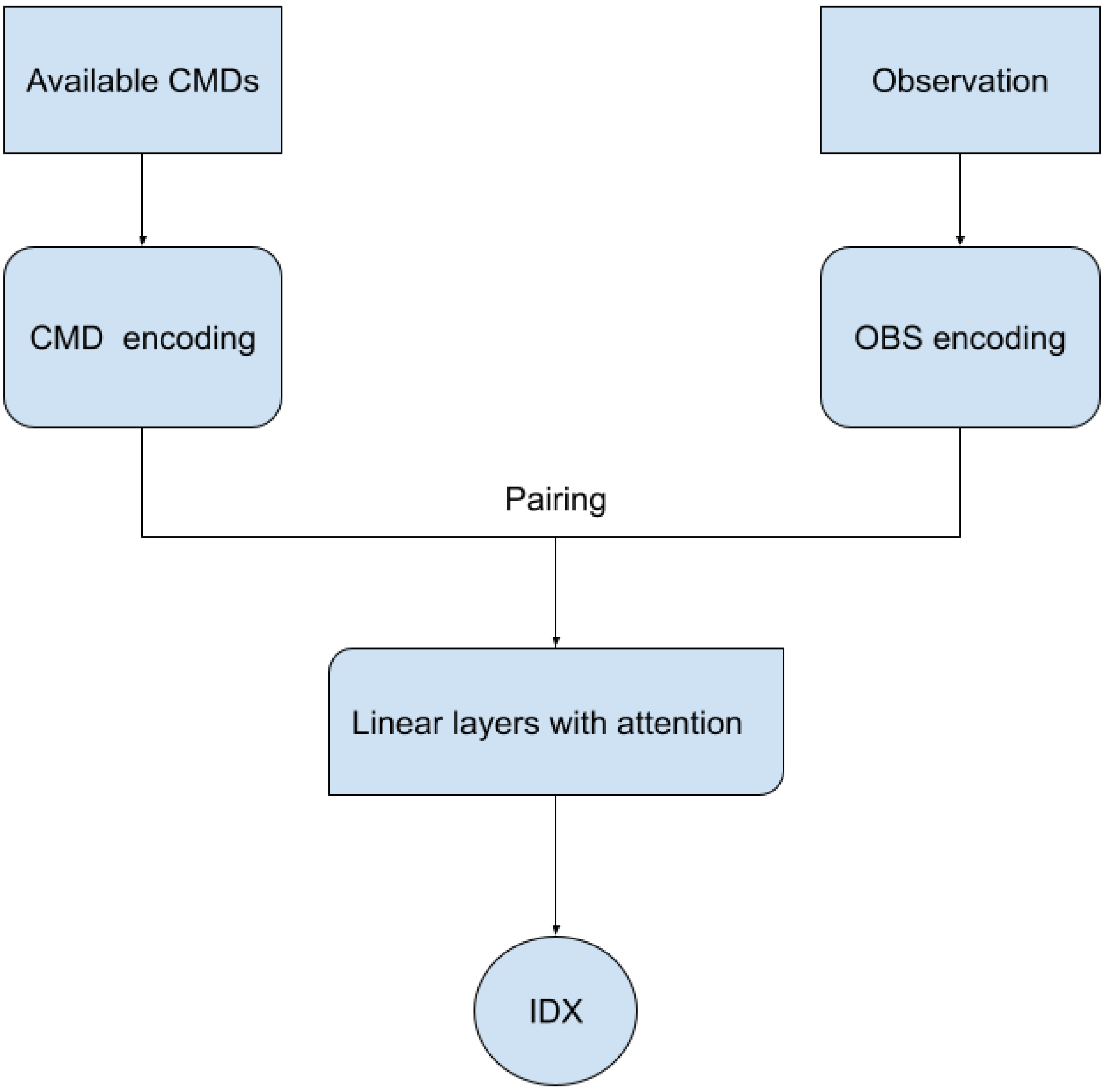


Figure 3.4: Neural Agent Architecture

3.3 World Models

The main inspiration behind the experiments behind this project is the work published in 2018 by David Ha and Jürgen Schmidhuber [Ha and Schmidhuber, 2018]. The motivation behind David Has and Jürgen Schmidhubers work was to imitate human modeling capabilities. They go ahead to explain that us humans do not keep an exact representation of the world that surrounds us, I keep a rough estimate of it. So they set forward to try and add a similar capability to an agent. The agent should be able to generate a model of the surrounding world in order to predict the future. If it is able to keep a representation of what will happen in the future it should be able to make better decisions to solve the problem in front of it.

The agent that resulted from this goal was tested in two environments inside OpenAI Gym [Brockman et al., 2016]. Gym is a toolkit for developing and comparing reinforcement learning algorithms. The two environments where, a car racing problem in which the agent needs to drive a car in a two dimensional world and a vizdoom , were the agent needs to dodge incoming projectiles.

Since World Models agent is directly inspired by the human cognitive system, it has three components. The first part is the Visual sensory component. It compresses visual data into a smaller representative numerical vector of fixed a size, this component is called V. There is also a Memory component whose task is to make predictions about the future codes that V is going to generate, this part will be called M. Finally, I have a decision-making component, the Controller or C. C decides which action to take based solely on the memory and the visual representation generated by the previously mentioned components. All these components have a different and unique training procedure.

In Figure 3.5 I can see the whole structure of the agent. The arrows represent the connections between the components. The very first input is the image that the environment generates. That image is encoded by V and said encoding is passed to both M and C. M, that has been previously trained infer information about the future, will output a vector with said information. And finally C with those two inputs provided by M and V will take an action.

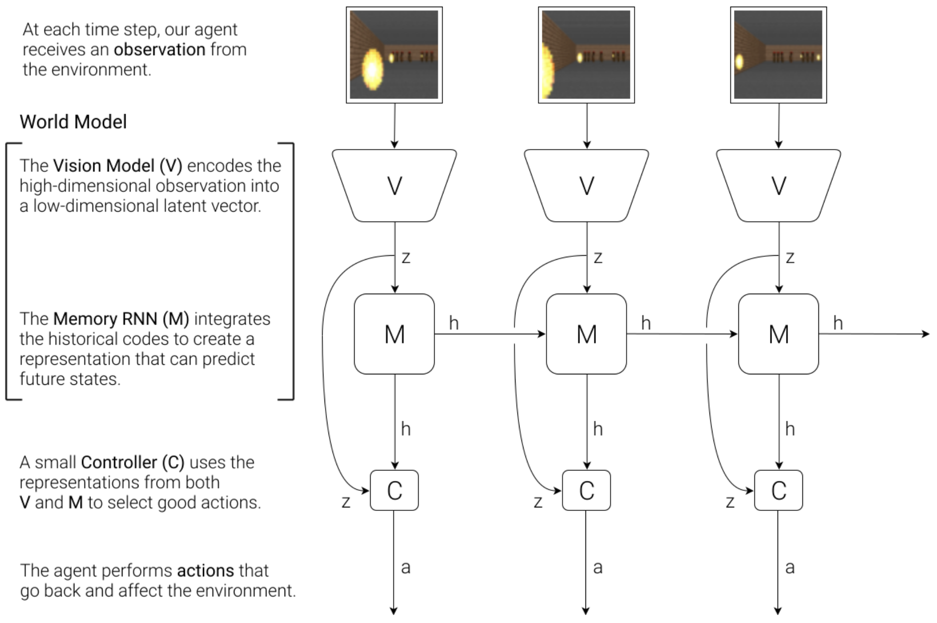

At each time step, our agent receives an **observation** from the environment.
World Model
The **Vision Model (V)** encodes the high-dimensional observation into a low-dimensional latent vector.
The **Memory RNN (M)** integrates the historical codes to create a representation that can predict future states.
A small **Controller (C)** uses the representations from both **V** and **M** to select good actions.
The agent performs **actions** that go back and affect the environment.
V
z
M
h
C
a

Figure 3.5: Overview of the World Models agent. Source: [Ha and Schmidhuber, 2018]

3.3.1 Visual Component (V)

The Visual components, or Vs, job is first to receive information from the environment. In the World Models project case that information will take the form of images. The second part of its job is to translate this images into a compressed vector that both the Memory and the Controller can understand. In doing the translation V should try to keep as much relevant information as possible from the original data. To perform this translation, they used a Convolutional Variational Autoencoder.

Variational Autoencoders

Variational autoencoders are type of neural network characterized by two encoder/decoder architecture. These two components are trained together. The encoders job is to compressed the input into a fixed sized vector, while the decoder takes that vector as an input and tries to recompose the original input.

The encoder is a neural network that. It must learn an efficient compression of the data into a lower dimensional space. It is important to note that the lower dimensional spaces of the encoder are stochastic. The encoder output parameter $q_\theta(z|x)$ are defined as a Gaussian probability density. I can sample from this distribution the representation of z.

The decoder is also a neural network. Taking the z vector as an input it must learn to reconstruct the original data. It outputs the probability distribution of the data $p_\phi(x|z)$.

As a loss function for using the negative log-likelihood with a regularizer is common in variational autoencoders. Since there is no global representation shared for all data points, I decompose the loss function into a single data point l_i. The total loss will end up being a summatory of all the data points: $\sum_{i=1}^{N} l_i$.

In most cases, this networks are trained to compress the original data into a lower dimensional vector. This means that after the training is done the decoder is discarded and I keep only the encoder to generate the latent z vectors.

The visual component is a convolutional variational autoencoder with four layers in the encoder and another four in the decoder. All this layers use the ReLu activation function. [Kingma and Welling, 2014].

In the Figure 3.6 I can see a basic schema of how V works. An image is fed to the encoder. This image gets compressed into a latent z vector, then decoder will try to reconstruct the original image. In this particular case the used a convolutional variational autoencoder which means that instead of having linear layers the layers are convolutional [Krizhevsky et al., 2012].

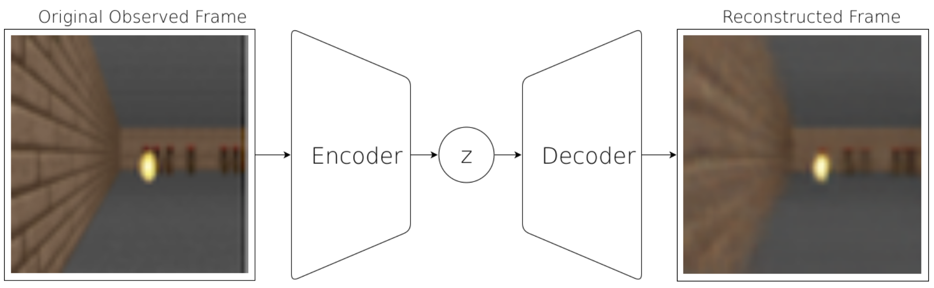


Figure 3.6: Representation of the Visual components work. Source [Ha and Schmidhuber, 2018]

Training

In this section I will give an overview on how the V was trained. One of the key factors factor on the training of V, is that a Gaussian prior was enforced in the latent vector z. This limits the visual components capacity for compressing each frame. In exchange for that lack of capacity, the VAE becomes more robust against unrealistic world scenarios.

The loss function is also different from the one mentioned in the previous subsection 3.3.1. They used two different loss functions L^2 loss which measures the distance between the reconstructed image and the original one and KL loss, which measures the divergence of a Dirac delta label and a Gaussian prediction.

The data was not collected by a random policy agent playing in the environment. They stored the frames from the played games to build a static dataset. Then they trained the visual component for a single epoch in said data.

3.3.2 Memory (M)

The Memory, or M for short, is in many ways the cornerstone of the whole project. This component is the one the one in charge of building an intuition for the world that surrounds the complete agent. Using that trained intuition M will use it to give information about the future to the controller. In the same sense as the visual components job is to compress the data in each frame, the memories role is to compress what happens over time. To accomplish this, M will serve as a predictive model for the future latent z vectors that the visual component will produce.

In figure 3.7 it is depicted the overall structured of the memory component. the RNN receives a z_{t-1} vector, an action and a temperature component. The z_{t-1} vector is the vector generated by V in the previous step. The action, a_{t-1}, is the one taking by the agent also the previous step. The temperature (τ), is a random component that is use to randomize the final output. M will finally output a prediction on what would be Vs next latent z_t vector. In order words M is outputting what it believes the future would look like.

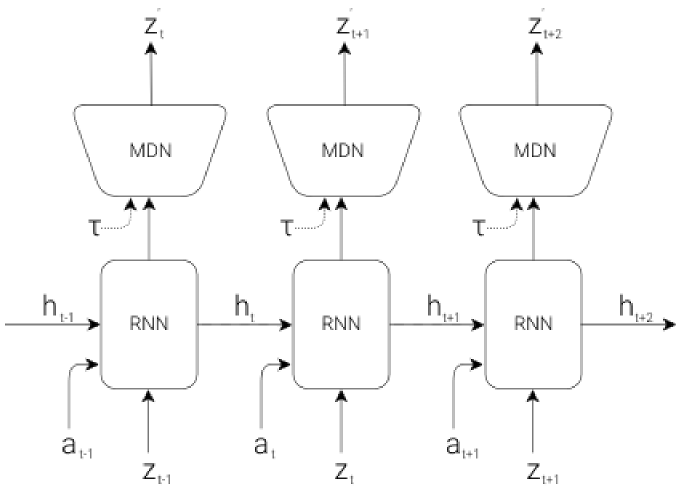


Figure 3.7: Overview of the Memory. Source [Ha and Schmidhuber, 2018]

How it works

As can be seen in Figure 3.3 there are two components forming the Memory. The first component is a Recurrent Neural Network, specifically and Long Short-Term Memory (LSTM) l[Hochreiter and Schmidhuber, 1997]. This is the component that will receive the input. This input has two parts, first a latent z_t vector generated by V and with it comes the action a_t that the controller took in the previous step. The RNN will not try to output z_{t+1} directly, its job is to model $P(z_{t+1}|a_t, z_t, h_t)$ where, a_t is the action the agent took at time t, z_t is the encoding V generated at time t and h_t is the hidden state of the RNN at time t.

The second component is *Mixture Density Network* [Bishop, 1994] (MDN). This MDN will take as an input the probability distribution $P(z_{t+1}|a_t, z_t, h_t)$ generated by the RNN. Here is where τ is used to increase or decrease the uncertainty, the higher the values the higher the uncertainty. This mixture density network will sample the probability distribution generated by the RNN. The output of this component will be a prediction of the latent vector z_{t+1} that V would generate after the action is taken in the actual environment.

These two components will form an MDN-RNN and work together to predict the future on the environment they are in.

Recurrent Neural Networks

Recurrent Neural Networks, RNN for short, are a type of neural networks specifically designed to work with sequences. In the world models scenario, there is a sequence of action-observation pairs. There are different architectures of recurrent neural networks, but I will specifically talk about the Long Short-Term Memory ones. LSTMs were created with the idea to allow recurrent neural networks to work with longer sequences, and therefore allowing this networks to link causes and effects remotely.

LSTM store information beyond the normal workflow of an RNN in a gated cell. Information can be stored, read and written in this cell in a similar way as data can be stored in the memory of a computer. This is where then name comes from. This actions are managed by element wise multiplication with the hidden layer weights and using the sigmoid activation function. The later gives us an output between 0-1 which makes it suitable for controlling the gates.

One of the key features of the architecture can be observed in Figure 3.8 are the skip predictions to all the hidden layers and outputs This not only makes the network faster to train by reducing the amount of steps but also mitigates the 'vanishing gradient', a problem where the computed gradients becomes increasingly smaller to the point which it become small enough that it doesn't affect the weights.

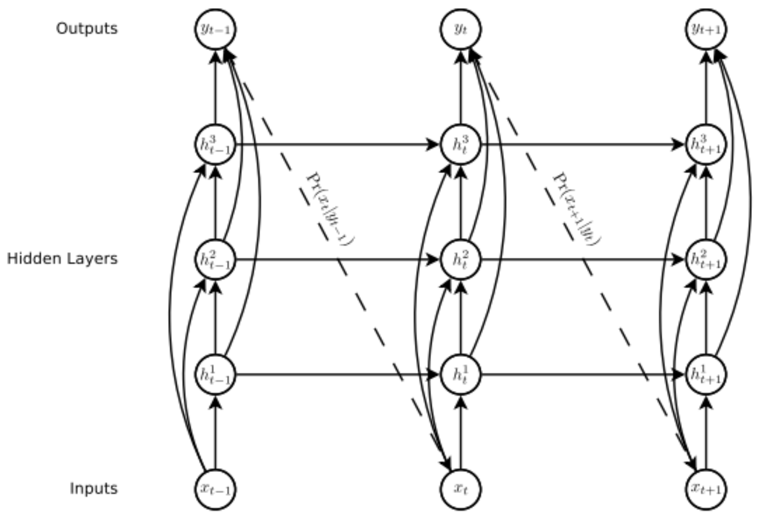


Figure 3.8: Structure of the LSTM. Source [Ha and Schmidhuber, 2018]

I can define the output ecuation of the LSTM as follows:

$$\hat{y} = b_y + \textstyle\sum_{n=1}^{N} W_{yh^n} h_t^n$$

$$y_t = \Upsilon(\hat{y})$$

where the W terms denote the weight matrices ($e.g$ W_{ih^n} is the weight matrix connecting the inputs to the n^{th} hidden layer. The b terms denote the bias vectors ($i.e$ b_y denotes the bias in the output vector). And finally H is the hidden layer function. Υ represents the output function. This networks define a function, parameterized by the weight matrices, from input sequences $x_{1:t}$ to the output vector y_t.

This output vectors are used to parameterize the predictive distribution $Pr(x_{t+1}|y_t)$ for the next input. The probability that the network gives to the x sequence is:

$$Pr(x) = \textstyle\prod_{t=1}^{T} Pr(x_{t+1}|y_t)$$

Mixture Density Network

Switching the focus to the second component of the memory, the *Mixture Density Network* [Bishop, 1994]. In default neural networks a set of inputs x is directly map to the target variables y. In this type of networks, the inputs are used to learn the parameters of predefined distribution. The particular distribution used for the World Models project [Ha and Schmidhuber, 2018] is a mixture of Gaussians. This particular case I have an RNN-MDN which mean the inputs I will use to parameterize the mixture of Gaussians will be the output of RNN.

A subset of the outputs is used to define the mixture of weights, while the remaining outputs are used parameterize the individual mixture components. The mixture is scaled in the following way:

$$x_i = e^{(x_i - Max(X))} \Rightarrow x_i \in (0,1]$$

Where X is the the mixture of Gaussians.

The RNN will output $Pr(x_{t+1}|y_t)$ as well as the mean, μ, and the standard deviation σ. Then it subtracts the logarithmic exponential sum from the mixture in the following way, while keeping the dimensions:

$$Pr(x) = Pr(x) - \log(\sum_{i=1}^{n} Pr(x_i)[1])$$

From said probability density distribution, it will output a diagonal covariance matrix of the factored Gaussian distribution. In order to achieve this diagonal sampling, a random number, $x \in (0,1)$, is generated. then I traverse the components of the distribution accumulating their values until their sum is greater than x. It returns the index that tipped the value above that of x. Since for every component of the mixture I select one index I can say I diagonally sampling. This indexes will be used to select the values of the mean, μ, and the standard deviation, σ. from the samples the RNN gave as an output.

Once the mixture has been scaled and the mean, μ, and standard deviation, σ have been sampled the final prediction for z_{t+1} can be generated the following way:

$$z_{t+1} = \mu + e^{\sigma}$$

As can be understood by the previous explanation the Purpose of the Mixture Density Network is to learn to parameterize a distribution. In this case a Gaussian distribution. As can be expected a single normal distribution is lacks the required complexity to predict the future in complex environments. For this reason, they will model their prediction from a sum of 5 Gaussian distributions. By using a mixture of 5 distributions the model does a better job at modeling the environment.

Training

The MDN-RNN was trained on a similar way to the visual component. The data was collected by a random policy agent playing in the environment. They trained for a total of 20 epochs in this dataset. They had a different version for each of the problem. The RNN had a total of 256 hidden units for the car racing game and 512 for the vizdoom game. Both version used 5 Gaussian mixtures.

When training the MDN-RNN teacher forcing from the recorded data was used. From the pre-computed data set of means, μ_t, and standard deviations, σ_t, was stored for each of the frames, and sample an input $z_t \sim N(\mu_t, \sigma_t^2 I) z \sim N(\mu_t, \sigma_t^2 I)$ each time a training batch is constructed, to prevent overfitting the MDN-RNN to a specific sampled z_t.

The loss function, $L(x)$, used in this particular scenario is computed in the following way:

$L(x) = -\sum_{t=1}^{T} \log Pr(x_{t+1}|y_t)$

In this approach M models the probability distribution for the next frame. Which mean if it does a poor job at this it has encountered a part of the world that has not yet learned how to properly model. This is the reason why the loss function is negated. This encourages the model to move towards areas it has not properly learned yet.

3.3.3 Controller (C)

The controller is the part of the agent responsible for deciding which action should be taken in order to maximize the cumulative reward. This reward is given by the environment after each action. C is made as simple and as small as possible. The Controller can be simple thanks to it making the decision based on the other two components, V and M.

C is a single layer linear model that maps z_t and h_t, the vector generated by V and the last hidden layer of the RNN respectively, directly into an action a_t. The model is simple: $a = W_c \; [\; z_t \; ; h_t \;] + b_c$, where W_c is the weight matrix, b_c is the bias vector, and $[\; z_t \; ; h_t \;]$ is the concatenation of the latent vector z_t generated by V and h_t being the hidden state of the RNN component of M.

This is the decision making component in charge of sending the action to the environment. The Controller agent will make use of both of the previous models, V and M. The agent will use the previously trained V in order to get the compressed version of the input. Since M is trained for one specific purpose which is predicting Z_{t+1} and this prediction is always done using the RNN's hidden state h_t at time t, this vector is a great candidate to act as an input for the controller. The vector h_t since it has all the information required to make the prediction about the future. Meaning that the input or the Controller will be a concatenation of said hidden state and the compressed observation produced by V.

Since the RNN is an LSTM it has the cell state, represented by C and the hidden state represented by H [Hochreiter and Schmidhuber, 1997]. This two states are concatenated with each other and the z_t vector generated by V. This will be the input of the controller which with a simple single layer neural network that takes the action. The hidden state of the RNN gets updated by inputting z_t and the selected action into the RNN.

Combining all these 3 components together I get the following schema as seen in Figure 3.10.

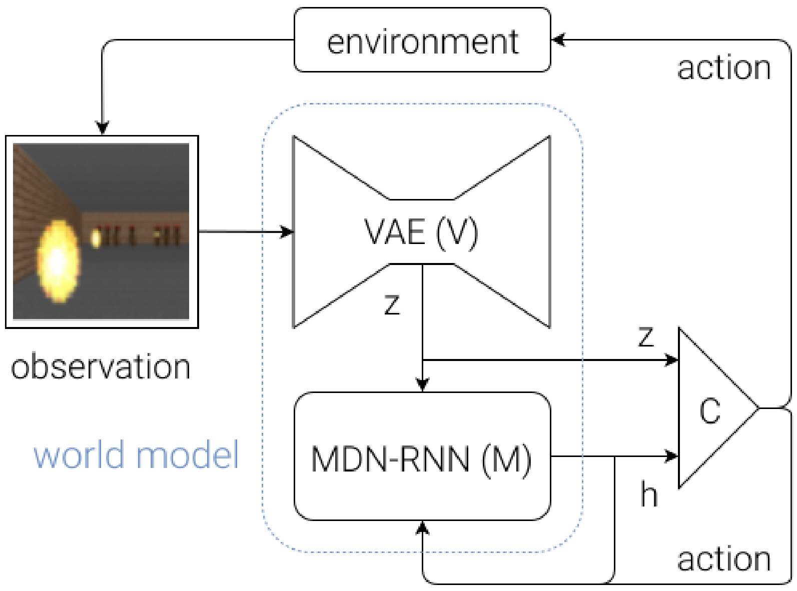


Figure 3.9: Schematics of the world models agent taken from their paper [Ha and Schmidhuber, 2018].

Training using the Covariance-Matrix Adaptation Evolution Strategy

To evolve the weights of the controller it trains used the $Covariance-Matrix\ Adaptation\ Evolution\ Strategy$ (CMA-ES) [Whiteson, 2012], Evolution strategies (ES) are stochastic, derivative-free methods for numerical optimization of non-linear or non-convex continuous optimization problems.

In Figure 3.10, the performance of the controller can be seen as it evolves using CMA-ES. In orange it can be seen the cumulative reward obtained by the worst performer of the generated population in that step. In green it is the performance of its counterpart, the best performer. The red line represents the average reward of the best agent. Finally, the blue line represents the cumulative mean reward of the entire population.

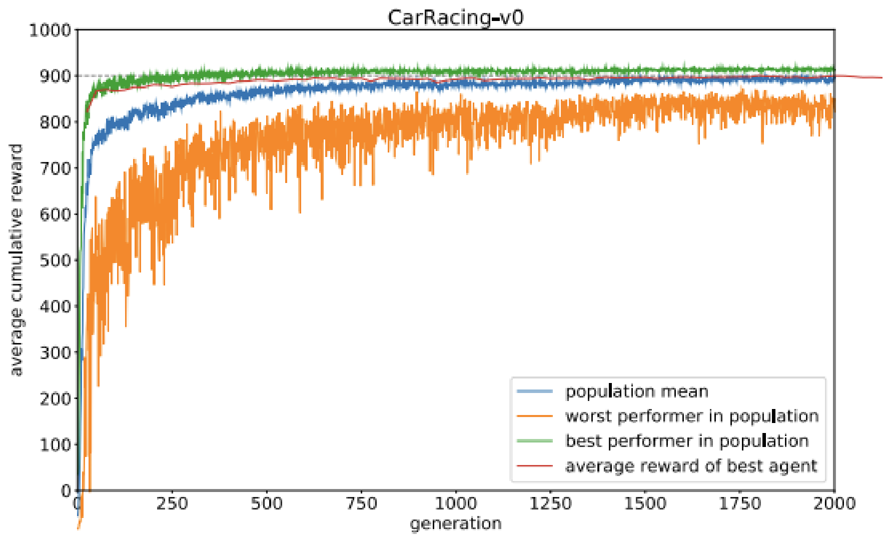


Figure 3.10: Evolution of the agents scores [Ha and Schmidhuber, 2018]

The requirement of the environment challenge was to achieve an average score of over 900 over 100 random rollouts so every iteration the best performing agent at the end of every 25 rollouts is selected. That agent was then tested over another 1024 rollouts to record the average red line. After 1800 generations the agent managed to score an average of 900.46 mean score after 1024 random rollouts and thus beating the challenge.

3.4 Universal Sentence Encoder

The Universal Sentence Encoder [Cer et al., 2018] or USE for short, is a model developed by Google Research to specifically target transfer learning to other NLP tasks. It generates sentence level embeddings which are able to summarize the semantic meaning of a sentence into a vector of fixed size. The are two versions model with trade-offs in speed and accuracy. For both variants they investigated and reported the relationship between the model complexity, resource consumption, the availability of transfer task learning data and task performance. They compared the model with baselines that used word level transfer learning as well as baselines that did not implement transfer learning. They found out that sentence level embeddings tend to outperform word level embeddings.

Limited amounts of training data is available for many NLP task. This represents a challenge for data hungry methods. Since annotating supervised training data is very costly large training sets are not widely available. Many models address this problem using limited transfer learning with trained models such as the ones produced by word2vec. However recent work has demonstrated strong transfer learning performance using pre-trained sentence level embeddings. In the paper cited in the beginning of this section, two models are presented, a transformer model and a deep averaging network model. In this project the model I used was the transformer model.

In this project I decided to use the Universal Sentence Encoder because of its performance in transfer learning tasks. Meaning that I could most likely directly import this model and use it to encode the observations given by TextWorld.

Sentence embeddings

An embedding is a fixed sizes numerical vector that attempts to encode some semantic meaning of the word or sentence it is encoding. The distributional hypothesis is usually the concept behind most embeddings. This hypothesis states that words which often have the same neighboring words tend to be semantically similar. For example, if 'football' and 'basketball' usually appear close the word 'play' I assume that they will be semantically similar. An algorithm that is based on this concept is Word2Vec [Church, 2017]. Word2Vec trained a neural network to distinguish co-occurring groups of words from randomly grouped words.

Moving to sentence level embeddings. There are different approaches to sentence embeddings. One of them is to average the word embeddings inside the sentence and use that average as the representation of the whole sentence. USE takes a very similar approach to this. the deep averaging network, DAN [Chen et al., 2018], creates embeddings by averaging together word and bi-gram level embeddings. Sentence embeddings are then obtained by passing the averaged representation through a feedforward deep neural network (DNN). The transformer sentence encoding model constructs sentence embeddings by using the encoding sub-graph of the transformer architecture. The encoder uses attention to compute context aware representations of words in a sentence. This attention that takes into account both, the ordering and identity of other words. The context aware word representations are averaged together to obtain a sentence-level embedding. Both models generated embeddings of size 512.

To compute the semantic similarity between two sentences, u and v, I measure the cosine similarity between the two sentence embeddings. The formula is as follows:

$$sim(u,v) = \frac{u \cdot v}{||u||||v||}$$

The results can be changed to an angular distance by applying the negative *arcos* to the previous formula:

$$sim(u,v) = -arcos(\frac{u \cdot v}{||u||||v||})$$

In Figure 3.11 a heat map of semantic similarity between sentences can be observed. The brigther the red the more semantically similar two sentence are.

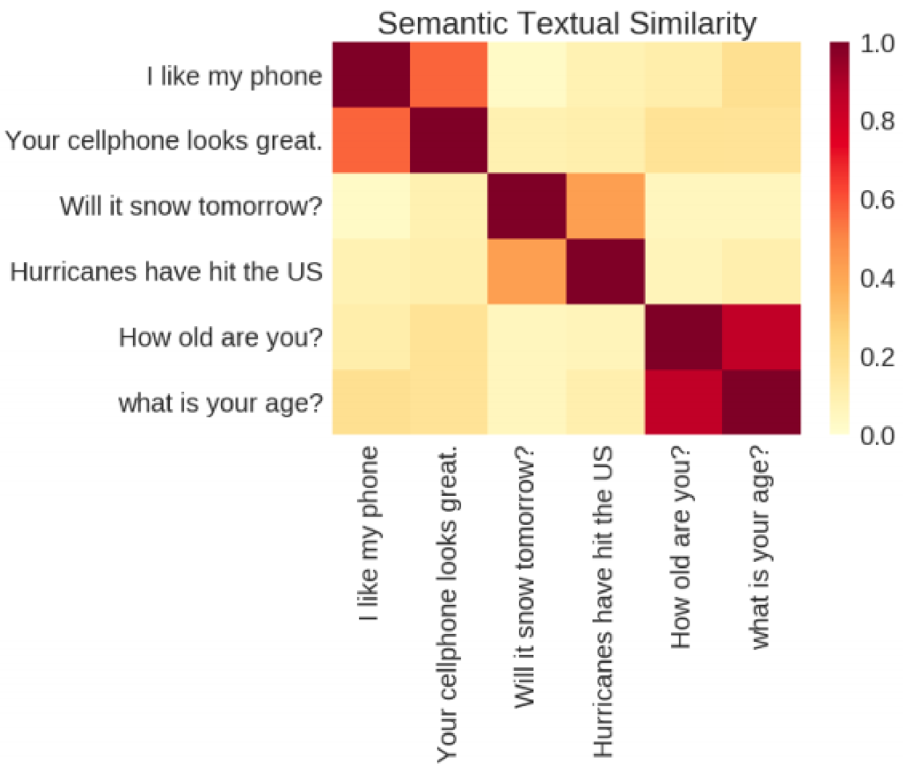


Figure 3.11: Semantic Textual Similarity. Source [Cer et al., 2018]

4. CHAPTER

Our method

In this chapter I intend to present and explain my approach to adapt the World Models project [Ha and Schmidhuber, 2018] to work in a text based environment, TextWorld [Côté et al., 2018] specifically. I kept the same 3 component architecture that David Ha and Jürgen Schmidhuber used in their project. I used the Universal Sentence Encoder, USE, as out text interpreter component. The memory was adapted to accept the new data but keeping its function intact. For the Controller I adapted the Neural Agent 3.2 to use the memory and USE in the same fashion as the original controller did in the World Models paper. Although I tried to maintain the integrity of the original idea as much as possible some changes needed to be made in order to be able to move to text data. A Representation of the whole procedure can be seen in Figure 4.1.

4.1 Universal Sentence Encoder (USE)

The Universal Sentence Encoder will take the place V had in the original World Models project. Since I are working in a text based environment I discarded the Visual component used in the original project. To take its place I decided to use the Universal Sentence Encoder [Cer et al., 2018]. The goal of the original V was to compress as much information as possible from an image into a lower dimensional vector. Since I changed to a text environment this approach is no longer feasible. To encode the observations given by the environment I will generate sentence level embeddings. The embeddings have a fixed size

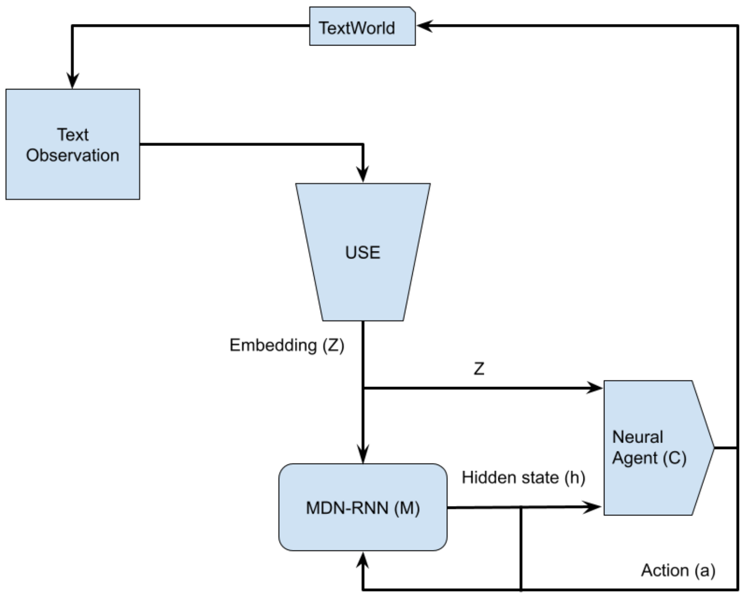


Figure 4.1: Schema of out Text Based Agent

and keep semantic information from the original sentence, making USE a perfect analog to the original visual component.

I gave USE two different uses. First I used it to generate a dataset in order to use to train my memory component. In a similar approach to the original paper I set a random policy agent to play 1000 roll-outs in the game environment. Using the Universal Sentence Encoder, I encoded the observations that TextWorld returned and saved them into a file with the action taken. This method allowed us to generated a consistent dataset.

The second use was to substitute the embedding layers of the Neural Agent so I can keep it consistent with the way the memory is trained. In substituting the previous embedding layers of the Neural Agent for USE I will make my controller more in line with the type of information M received.

Since the Universal Sentence Encoder comes pretrained, there was no need for training. Furthermore, given the fact that it was trained with transfer learning in mind there was no need to fine tune the model for this particular task.

4.2 Memory Component (M)

The Memory component will be the part of my agent in charge of developing an understanding and intuition of the world that surrounds it. The type understanding it needs to build is not precise to the very detail, but more like an intuition. It will develop this intuition by training on observation-action pairs trying to predict what the next observation would look like. M is based on the original MDN-RNN 3.3.1, and will perform the same role.

Operating in the same way as the original did, M will receive an encoded information and an action. Based on that information Ms role is to model the probability density function $P(z_{t+1}|a_t, z_t, h_t)$.

Where:

a_t is the action taken at time t.

z_t is the USE embedding of the observation at time t.

h_t is the hidden state of the RNN at time t.

z_{t+1} is the embedding USE will produce after the action is taken.

The RNN will generate a probability density function that is the passed to MDN. The MDN will diagonally sample the mean, μ, and the standard deviation, σ. Following the same procedure as it was done in the previous project I sampled z_{t+1} in the following way:

$z_{t+1} = \mu + e^{\sigma}$

Adaptation to text

When adapting the memory to text the first thing that should be mentioned is the change of the representation. The observations why used were one dimensional vectors of size

512 generated by USE. This embeddings are larger than the ones that were used in the original one. On top of that the complexity of the problem and the information to keep has also increased. M now has to understand a global task and the relative locations of the agent, the interactable elements and which of them will help completing said task.

However, there are more factors to take into account than just the representation of the observations. The action space has also gotten more complex. In the previous scenarios there were 2 to 3 actions in total, all of them available at any given time step. In the TextWorld environment I extracted a total of 454 unique actions, with usually only 6 to 12 being available at any point. On top of that there were actions in the test and validation datasets that did not appear on train, which adds another layer of complexity. To solve this problems the actions where codified into one hot encoded arrays and the actions that did not appear on train were ignored.

4.2.1 Training

I took two different approaches to training the M. The first approach was an online training approach; the model is trained batch by batch. This batches are randomly selected from the dataset. Every training step a set of x random indices are selected from the training data, from which x sequences are formed, this sequences are used to train the model. The process gets repeated for a predefined number of batches.

For the online method I trained the model for a total of up to 2000 batches, from this number of batches onward a very stagnant loss function was very noticeable and I decided not to extend the training. A Stochastic Gradient Descent style training was also tried for 4 epochs on a dataset of 600,000 situations, but there was no noticeable difference.

In both the approaches teacher forcing is used to avoid a bad prediction deviating the following ones and in consequence making that sequence useless training wise. In order to apply teacher forcing for this scenario, since I do not have a set of pre-computed means, μ, and standard deviations, σ, I used the pre-computed embeddings that are already part of the sequence.

From every training session two models are selected, the model that performed best in the validation set in terms of cost and the model that performed best in that same validation set in terms of semantic similarity.

The loss function for the memory is computed via the following formula:

$$L = -(\phi + (-0.5(\frac{(y-\mu)}{e^{\sigma}})^2 - \sigma - \sqrt{2\pi}))$$

Where:

ϕ represents the mixture.

y is the gold standard.

μ is the population mean.

σ is standard deviation.

As can be seen the loss function is negated, this will encourage the agent to explore previous unexplored areas to get a better overall intuition for its entire surroundings. Another insight that can be derived from that loss function is that in order to train the model, there is no need to explicitly generate the z_{t+1} vector. The generation and sampling of this vectors is done in a separate sampling function after the models training has finished. To test for semantic similarity during training, at the same time the validation is done, I sample some the z_{t+1} vectors and compute the cosine similarity in the same way I explained in section 3.4

4.2.2 Sampling

Sampling is used to generate the final prediction of the latent vector z_{t+1}, and thus predict the future. The prediction is done by sampling the mean and standard deviation from the predicted mixtures, $P(z_{t+1}|a_t, z_t, h_t)$, given by RNN.

4.3 Controller (C)

The controller is the component that interact with the environment. It integrates USE and M in its structure to make decisions and is tasked to take make the best decisions in order to maximize the reward function. Following the guideline in the World Models project [Ha and Schmidhuber, 2018], C will be as simple as possible.

The base for the controller is taken from the original Neural Agent that is used as a baseline. This is for various reasons, but the most important one is the difference in environments. In both of the games, carracing and vizdoom 3.3, that the world models agent played, the actions are available at all points. This is not the case in any of TextWorlds games. The agent has a total of 454 possible actions, but only around 6-12 are available at each time step. This complicates the job of the M and the design on the architecture of C. In order to work around this problem, I decided to work using the Neural Agent as a baseline since it already handled this problem.

Adding USE and M to the architecture of the Neural Agent means that most of its complexity disappears. All the embedding layers and GRUs are substituted by the Universal Sentence Encoder. Working on the decision making and as part of the critic I will add the Memories RNNs hidden states. The linear part of the process I will have the embedding of every command with the observation embedding concatenated with the hidden states. Once the commands and the observation are ready the process is quite similar passing through an attention layer to a softmax and finally a multinomial will select the best action to take in this process.

This does leave the agent with quite a limited amount of trainable components which may end up becoming a problem.

In figure 4.2 I can see one of the architectures of my controller after adding the Universal Sentence Encoder [Cer et al., 2018] and the Memory model. looking at the right hand side I can see that both the GRUs and the embedding layer have been taking out and substituted by USE and M will contribute its hidden state concatenated to the embedding generated by USE. The critic will make its evaluation based on those two vectors. On the other side, I used the Universal Sentence Encoder in order to get the embeddings of the commands two which in the same fashion as explained in figure **??**. Where each command is concatenated to the observation and the hidden state of the Memory and an attention layer with softmax and a multinomial will sample the index of the commands with the best expected reward.

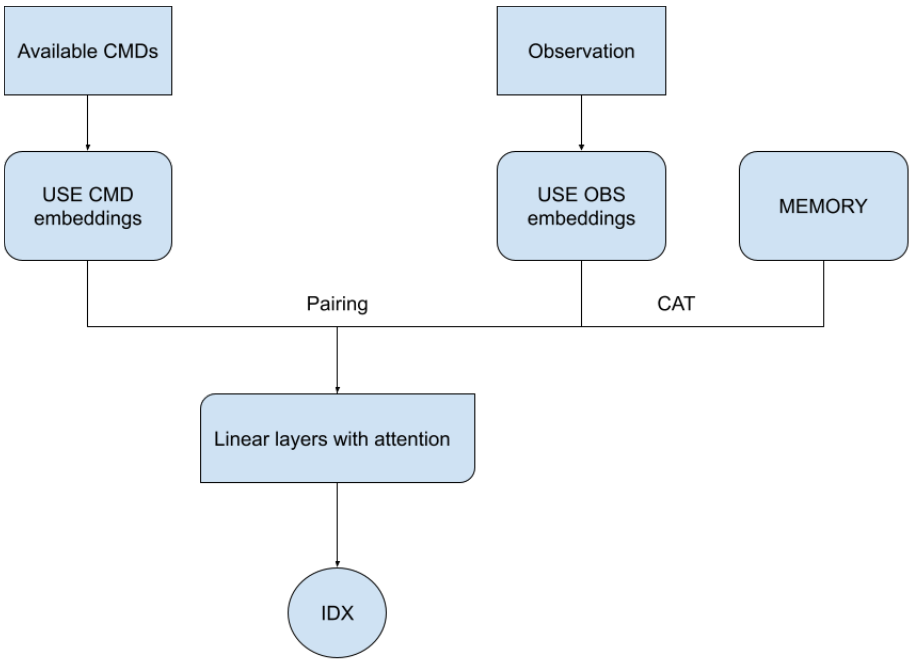


Figure 4.2: Architecture of the modified Neural Agent

5. CHAPTER

Experimental Settings

In this section I would like to discuss in depth the experimental settings throughout the course of this experiment.

5.1 Dataset Generation

When setting forward to train M it became apparent that there was no suitable dataset for the task. This meant that a dataset needed to be generated. I decided to tackle this tasks in a similar fashion as it was done in the original paper.

M takes a latent vector z_t generated by USE and $a_t - 1$ the previous action that, once folloId, yielded z_t. I set a random agent to play 1000 games of TextWorld, with a maximum total steps of 1000. This agent takes all of the decisions at random ensuring that I get an ample sample size and many different combinations of the action-observation pair, since the actions would follow no inherent logic. At every step I use the Universal Sentence Encoder to generate the embedding of each observation. During the generation of the dataset the action is kept in the string format that TextWorld delivers it in. This pairs are ultimately stored in a pickle file. Three sets Ire generated, 1 million rows for the training set, and 350,000 for each, test and validation.

5.1.1 Data Handling

In this section I intend to explain how this data was handled before it was fed to the MDN-RNN.

Remembering how the pairs are ordered in each row of the data, will I have z_{t+1} and a_t. This means that in order to correctly set the information up to be fed to the Memory I need to pair each action with its corresponding observation.

In Figure 5.1 I can see how each action is concatenated with the previous observation. This is due to the fact the action is taken and I cannot see the resulting observation until the next time step. In order to work around this the input is comprised of z_t and a_{t+1} and the expected output is z_{t+1}

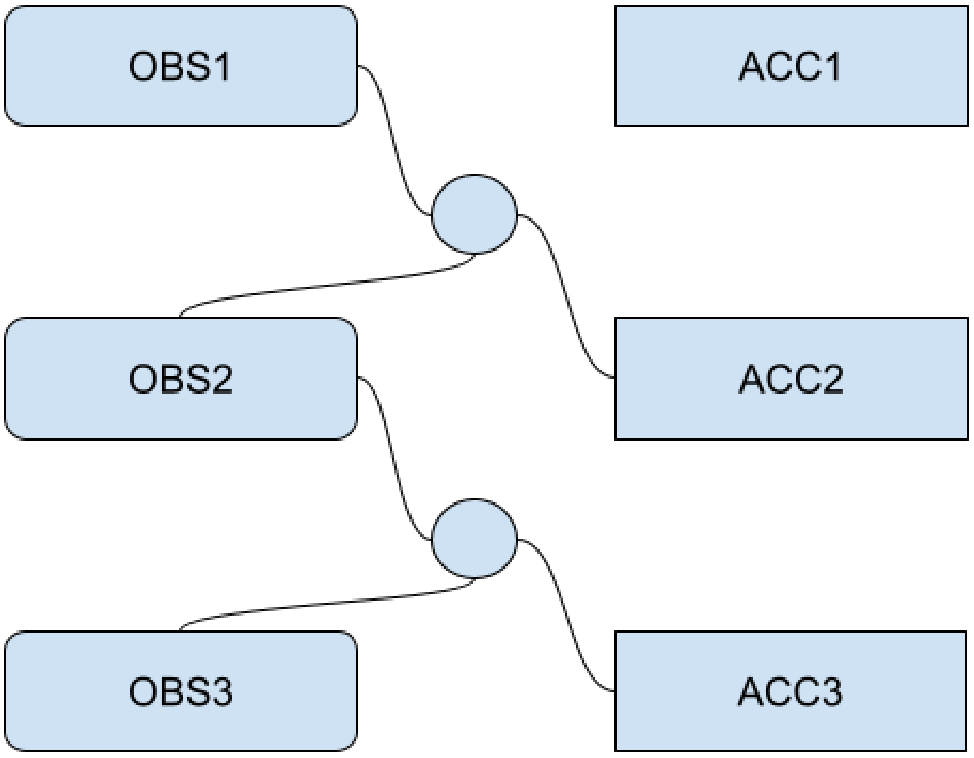


Figure 5.1: Schema picturing how the inputs and outputs are created from the dataset.

The latent vector z_t is a size 512 embedding previously generated by USE, but the action still needs to be prepossessed. TextWorld present the player with a set of available actions. Once an action is taken the new observation is generated. In order to encode the actions, I One Hot Encoded them. Since only one action can be taken at a time ohe is a suitable encoding for this solutions. The input for the Memory will be a concatenation of the embedding generated by USE and the encoding of the action.

An issue that needed to be addressed is what happens with the actions that are in testing and validation sets that do not appear in the training data. Since the environment is so complex when generating the the validation and test datasets, some actions appeared in this two that did not appear on the training set. since they Ire around 10 actions I decided to ignore those action observation pairs.

5.2 MDN-RNN settings

Hyperparameters of the Memory:

- RNN size: 512
- the number of Gaussian mixtures: 5
- Input size: 966
- Learning rate: 0.001
- Batch size: 30
- Sequence Length: 5
- Recurrent dropout: 0.8

The MDN-RNN was originally trained in 4000 batches but after a more indepth analisis I noticed that beyond the batch number 2000 the change of performance was minimal. After this was noticed the number of batches was reduced to 2000. Some models where trained with a bigger RNN size but this supposed no mayor impact in the performance of the network. Three different sequence lengths where tried, 5,7 and 10 and with very similar results in cost and in the semantic similarity department.

One observation that was noticed is that the training and validation performance where almost mirrored and Int hand in hand. The results did also remain similar when using the stochastic gradient descent method.

5.3 Controller

Many different version of the controller where implemented.

5.3.1 Different Controller versions

I trained many different version of the controller in order to see how certain changes on the design would impact the performance in the environment. The most important one are listed below.

- **RNN of size 512 and an active memory** was the standard version since it replicate closer the behavior of the memory in the original paper.

- **RNN of size 512 and a locked memory** this test came up after considering the action space might prove too complex to be updated at every instance. The main difference with the previous agent is the fact that the hidden state does not update after taking an action and is kept in the same state as it left the training.

- I also tested and agent with the **Memory trained using SGD** since I wanted to see if there would any difference betIen the online method proposed in the original paper and a more standard approach to training models.

- Finally, I arrive at the biggest deviation, this model I have called Z_{t+1} model, since instead of using the hidden state same as all the previous version this model radically alters the approach. In previous agents the Memory component of the input is constant for all the commands, this time I sample the prediction that the memory makes of the future given the current observation and the commands available. In this agent every command is paired with the z_{t+1} latent vector that M predicts will be the result of taking that action given the actual observation.

5.3.2 Ablation

The Controller has two very distinct components, the Universal Sentence Encoder and M. In order to properly asses the effects of the memory I made an ablation test where I used on only the universal sentence encoder to encode the current observation and the embeddings without using the memory to make decisions. this experiment alloId us to gain more insight into the MDN-RNN [Graves, 2013] since the loss function and the semantic similarity Ire not clear metrics on their own.

The Controller was tested in the same map the MDN-RNN was trained on so the impact of the memory component could be better appreciated. Regarding the actor critic (A2C) reinforcement learning function, it was left untouched since there no significant changes Ire needed. The agent was trained for 100 episodes with a maximum number of steps of 100 too.

6. CHAPTER

Results

In this chapter I will review the results obtained by the Memory component during its training and validation as Ill as the results obtained by the complete agent playing inside the TextWorld environment. For all the graphs the blue line will represent the metric related to the train data and the orange will be the metric related to the test data.

6.1 Memory Results

I performed many tests in order to get the best possible understanding of the memories results. The analysis of results for M was focused around two mayor metrics. First I looked at the cost during training and validation. Second I looked at the semantic similarity between the sampled $\hat{z}_{t+1}$ and the gold standard z_{t+1}.

In Figure 6.1 I can see the cost represented by the Y axis and the batch number in the X axis. I am trying get the cost as low as possible. M was trained for 2000 epochs and the validated every 50.

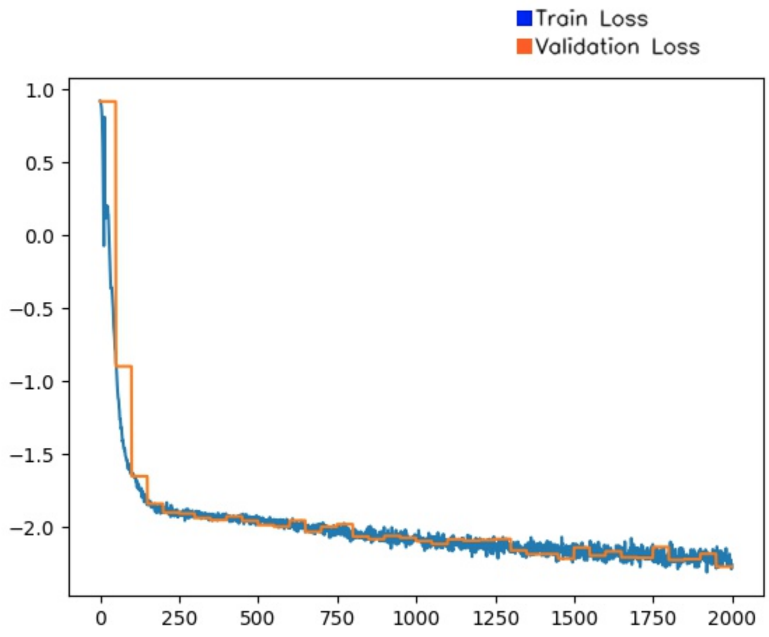


Figure 6.1: Cost function for M

In Figure 6.2 I can see the semantic similarity 3.4 represented by the Y axis and the batch number by the X axis. I are trying to achieve the highest value possible for the similarity. M was trained for 2000 epochs and the validated every 50.

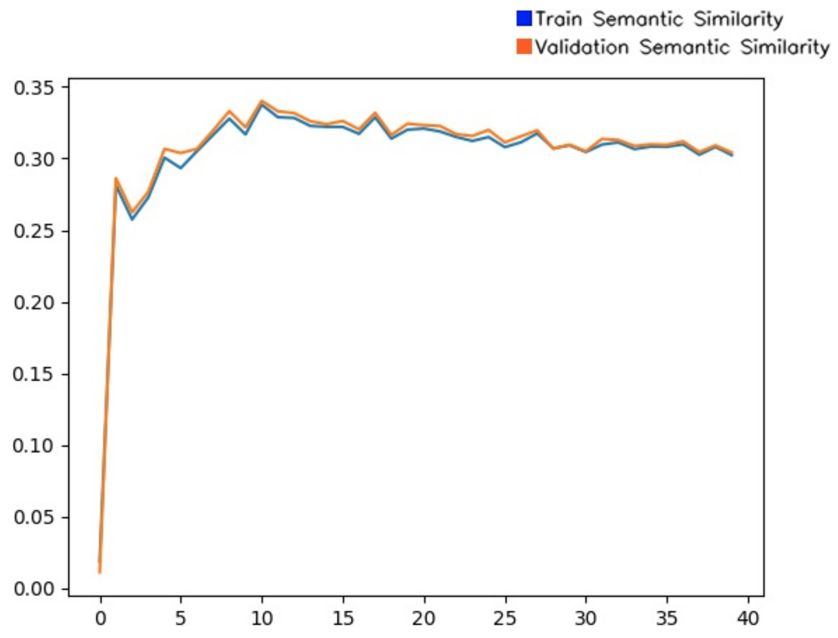


Figure 6.2: Semantic similarity between $\hat{z}_{t+1}$ and z_{t+1}

Test were also made for heavier versions of the MDN-RNN. The size of the RNN was increased to 2048 hidden units and the number of batches to 4000.

In Figure 6.3 I can see the cost represented by the Y axis and the batch number in the X. I are trying to get the cost as low as possible. M was trained for 4000 epochs and the validated every 50. As can be seen there is noticeable improvement compared to the lighter version of M.

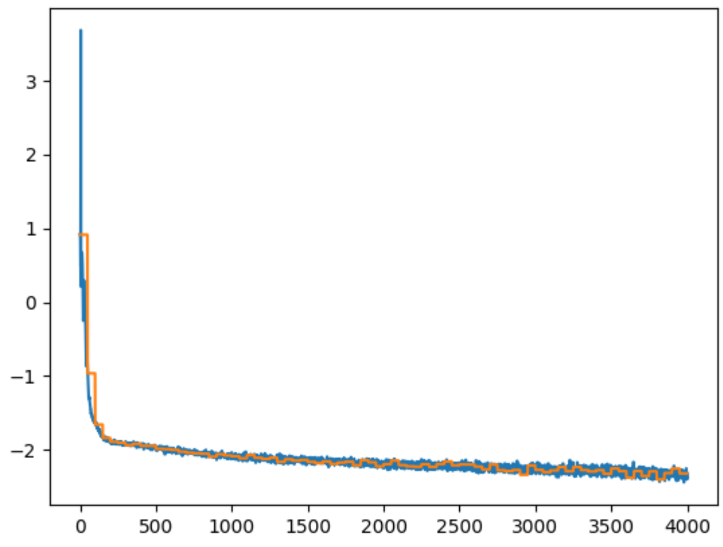


Figure 6.3: Cost function for the heavy version of M.

In Figure 6.4 I can see the cost represented by the Y axis and the batch number in the X. I are trying to achieve the highest similarity possible. M was trained for 4000 epochs and the validated every 50. Once again there is noticeable improvement compared to the lighter version of M.

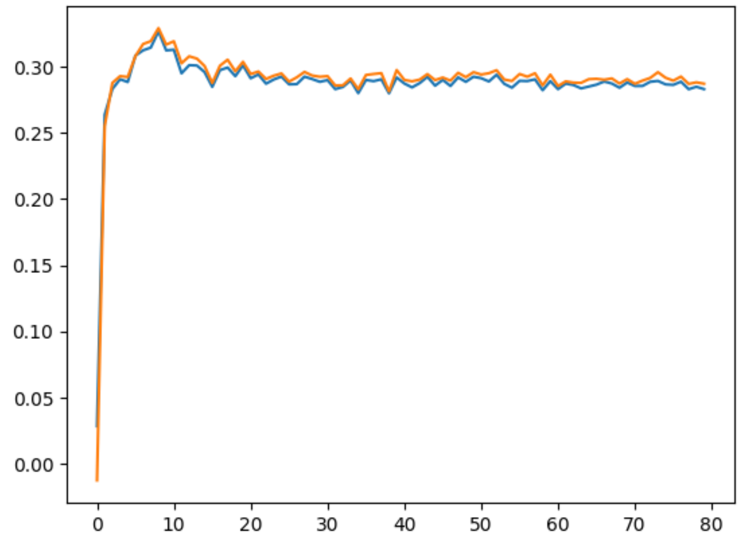


Figure 6.4: Semantic similarity betIen $\hat{z}_{t+1}$ and z_{t+1} for the heavy version of M

6.2 Text Agent

In this section I will present the results obtained by my complete agent. In order to put into context my agents' performance I took the standard Neural Agent as a baseline. The performance of various versions of the agent can be seen in the Table **??**. All measurements where done in the same map the memory was trained in. A simple game style map in TextWorld with the rewards parameter set to dense 3.1.2. The performance was measured by their mean score out of 10 after playing in the map for 100 times with a maximum of 100 steps per game. Every time the agent performs an action the world will return a reward that can be either 1 or 0. A positive reward means the action put the payer closer to completing the task while any other action will receive the reward of 0.

As can be seen in Table 6.1 there are two versions of every type of agent. One for the memory that scored the best in cost during the validation of M. The other one the memory that managed to get the best scores in semantic similarity when validating.

The best performing model was the standard Neural Agent folloId by the agents that had the blocked memory. This means the memory was static and was not updating after every

Model	Score
Baseline	
Neural Agent	4.6
Models with a static memory	
blocked_model_cost	4.2
blocked_model_simm	4.2
Models with a dynamic memory	
SGD_score_model	4
SGD_cost_model	4
Z_{t+1}_cost_model	4.1
Z_{t+1}_simm_model	4
Ablated_model	4

Table 6.1: Model performance in the training environment

action.

SGD is the prefix to denote that the model was trained using the Stochastic Gradient Descent method instead of the online random batch method which was the standard set by David Ha and Jürgen Schmidhuber.

$\mathbf{Z}_{t+1}$ is used to denote the agents where I concatenated each command with the prediction made by the memory for the pair of the current observation with said command.

Finally the **Ablated** model made use only of the Universal Sentence Encoder without using the memory.

6.3 Discussing the Results

Unfortunately, the agents I built Ire not able to beat the baseline I set up. In spite of that there are still come positive takeaways.

I can see that the models that where trained using the online mode with the blocked memory made better decisions than the ones their counterparts, that used SGD and had an unblocked memory. This may be due to the fact that the environment is rather complex and has too many possibilities. More importantly the changes in the environment are more radical compared to the more continuous environment in the original World Models project [Ha and Schmidhuber, 2018]. This may imply that having the memory intact from the more generalized training, might help compared to constantly trying to update it for the actual scenario.

On the other hand, I can see that the models that Ire trained using the Stochastic Gradient Descent did not manage to give the controller agent as much information, giving validity to the original training method used in the World Model papers.

One of the most important deviations from the original implementation is in the agents with the prefix z_{t+1}. In the original implementation the agent uses the hidden states of the RNN in order to get information related to the possible futures. In this implementation instead, I used each of the commands paired with the actual observation so sample z_{t+1}. This predicted z_{t+1} vectors take the place of said hidden states. Unfortunately, they did not achieve he expected results.

The ablated model is as expected among the worst performers although not as far behind as it could be expected.

Despite not being able to beat the baseline, there is still a good takeaway in the fact the model with the trained memory intact was able to help the agent perform better than the rest.

One thing that raised suspicion about the memory component is the fact that despite the change in the size of the RNN and the number of mixtures in the MDN the results pretty much stayed similar, both in the sharp rise to the maximum/minimum and the stability after the quick ascension or decline.

7. CHAPTER

Conclusions

When this project began, I set to build a text based agent that was able to model the world around it in the same abstract fashion as us humans. I used the Universal sentence encoder in order to use its embeddings as our representation of the text observation. On the other hand, I adapted the memory model in order to make it work with the new type of data. The Neural Agent was completely changed keeping only the A2C reinforcement training function so it could accommodate for the USE embeddings and the memory component.

The result of the experiment where not as good as expected. I would like to outline the differences between the projects that most likely contributed to this results

The first reason is the change in problem type. The task of Natural Language Processing is in it shelf. Compiling and gathering all the semantic relationship between words in a sentence, coupled with the complexity of understanding what the task is and all the objects that relate to it, is in itself a way more complex task than turning left or right.

On top of that I would also like to outline the environment difference. In the original project [Ha and Schmidhuber, 2018] the model updates each frame, which means that between updates the changes are minimum. In comparison, the changes in TextWorld completely alter the world state and observation. The difference is akin to the difference between discrete and continuous variables. In TextWorld it appears to be harder for the memory to abstract knowledge compared to the mostly similar roads with the green grass sides in the original project.

Moving onto the decision space of both problems. The original World Models [Whiteson, 2012] the agent had 3 possible actions in the car racing problem and 2 possible actions in the vizdoom 3.3 problem, all of them available at all times. In this scenario there are over 400 total actions and at every given moment only 6-12 of them are available at one time. This once again complicates the problem by making action selection more complex.

Overall it is a difficult task and still has a lot of work ahead of it.

7.1 Future Work

There are still plenty of possible avenues to better understand and implement the problem. One of them would be a to build a DNN that is able to extract which location of the environment the memory located the prediction. The place described by the memory could play a major role in the decision making of the agent. The agent could understand which rooms it is more likely to find certain objects and path towards them.

Another different training procedure that could be interesting to test out, would be substituting the One Hot Encoding approach for embeddings of the actions generated by USE. This way the memory could derive some semantic meaning from the commands that is encoded in the same way as the observation it is receiving.

One more approach that could reveal more information about the memory component, would be putting the agent into a simpler environment where the changes every step are not as radical. This could yield information that helps us understand how well the RNN-MDN abstract information on continues environments compared to more discrete ones.

Appendices

Bibliography

[Akerkar and Sajja, 2009] Akerkar, R. and Sajja, P. (2009). *Knowledge-based systems*. Jones & Bartlett Publishers.

[Bishop, 1994] Bishop, C. M. (1994). Mixture density networks.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Cer et al., 2018] Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Céspedes, M., Yuan, S., Tar, C., et al. (2018). Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.

[Chen et al., 2018] Chen, X., Sun, Y., Athiwaratkun, B., Cardie, C., and Weinberger, K. (2018). Adversarial deep averaging networks for cross-lingual sentiment classification. *Transactions of the Association for Computational Linguistics*, 6:557–570.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[Church, 2017] Church, K. W. (2017). Word2vec. *Natural Language Engineering*, 23(1):155–162.

[Côté et al., 2018] Côté, M.-A., Kádár, Á., Yuan, X., Kybartas, B., Barnes, T., Fine, E., Moore, J., Hausknecht, M., El Asri, L., Adada, M., et al. (2018). Textworld: A learning environment for text-based games. In *Workshop on Computer Games*, pages 41–75. Springer.

[Graves, 2013] Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

[Ha and Schmidhuber, 2018] Ha, D. and Schmidhuber, J. (2018). World models. *arXiv preprint arXiv:1803.10122*.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Kingma and Welling, 2014] Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes.

[Klein, 2017] Klein, R. G. (2017). Language and human evolution. *Journal of Neurolinguistics*, 43:204–221. Language Evolution: On the Origin of Lexical and Syntactic Structures.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105.

[Lebling et al., 1979] Lebling, P. D., Blank, M. S., and Anderson, T. A. (1979). Special feature zork: a computerized fantasy simulation game. *Computer*, 12(04):51–59.

[Rosenstein et al., 2004] Rosenstein, M. T., Barto, A. G., Si, J., Barto, A., and Powell, W. (2004). Supervised actor-critic reinforcement learning. *Learning and Approximate Dynamic Programming: Scaling Up to the Real World*, pages 359–380.

[Sharma, 2008] Sharma, L. (2008). *" Colossal Cave Adventure–Hindi Mein!": An Experimental Approach to Second Language Acquisition Through Computer Games*. PhD thesis, University of California, Irvine.

[Whiteson, 2012] Whiteson, S. (2012). Evolutionary computation for reinforcement learning. *Reinforcement learning*, pages 325–355.