Project II - World Models

Leonel Flores

May 9, 2021

1 Part I

1.1 Introduction

When interacting with the real world, we might find ourselves in situations where we have to make a series of decisions in order to attain the best possible outcome. In control theory, this decision making process can be modeled by a ***Markov decision process (MDP)***. Specifically, an MDP $\mathcal{M}$ is defined as the collection of possible states, possible actions, possible rewards, and a policy for making decisions using this knowledge. MDPs are well understood, and we have techniques and tools for working with them. However, MDPs depicting the real world tend to be computationally expensive, or even intractable. ***Reinforcement learning (RL)*** attempts to address the pitfalls of MDPs. Specifically, RL concerns itself with allowing agents to perform optimally with large MDPs, or even unknown MDPs. This is done by dropping in a (deep) neural network so that our model actually learns the way we need to make these decisions.

Simple RL algorithms have gotten us far. However, these methods have not been robust enough to encompass solving all problems. Moreover, the more we apply RL to real-world problems, we gain perspective on how "learning" really works (or, at least how it should work). The authors of this paper have, as a result, proposed a new approach for learning in an RL setting. At a very high level , this new technique blends elements of spatial awareness and past experiences. We now describe the model in its entirety.

1.2 Agent Model

There are three main components to this model: the vision model, the memory model, and the controller. When considering both the vision and memory models, we call it the ***World model.*** The components are described as follows:

- ***Vision:*** The vision model takes up the form of a variational autoencoder. Specifically, what this does is it takes an input image at time t and creates a latent vector z_t through a learned encoder. This latent vector is meant to encode all of the relevant information of the picture, somewhat of a compression. Each entry in z_t represents a probability distribution of the presence of said latent feature. Then, with this latent vector, we learn a decoder that takes z_t and reconstructs the original input image. As we can see, the loss to be optimized is the absolute difference between the original input image and the reconstruction.

- ***Memory:*** The memory takes up the architecture of an MDN-RNN, which needs to be unpacked a bit. Recall that a recurrent neural network (RNN) is a NN architecture that, at each time t, we have

$$h_t = f(h_{t-1}, x_{t-1}; \theta).$$

  One way to think about this is that h_t will have to encode all of the information/knowledge of all past occurrences (x_{t-i}) and of the previous state (h_{t-j}). Whenever we're ready to infer using our model, rather than getting a deterministic answer $\hat{y}$, the MDN portion of this architecture will give us $p(\hat{y})$, effectively a probability distribution for what the outcome could be. These two aspects comprise the memory model of the agent.

- ***Controller:*** The controller portion of the model is where the agent actually learns to make the decisions that will maximize the expected cumulative rewards. Note that the bulk of the complexity of the agent is in the World model (perceiving what's around us in a learned way is the hardest part). As a result, this lets the controller be a simple linear single-layer neural network:

$$\text{Action}_t(z_t, h_t) = W_C \cdot [z_t, h_t] + b_C.$$

  One important thing to note about the controller model is that the authors make it a point to use covariance-matrix adaptation evolution strategy as the controller's optimizer.

This model comes together exactly how you'd think.

1. First, the vision model produces a latent vector (in this paper, the length of z_t is 32) from an observation made (i.e., an image) of the environment.

2. The vector z_t is then fed into our memory model, outputting a hidden state h_t.

3. Both z_t, h_t get fed into our controller C, and the resulting action a_t gets broadcasted back to the environment and the memory model. In code, this looks something like this:

```
z = vision.encode(observation)             # vision is our VAE
action = controller(z, h)                  # controller is our controller model
reward, observation = env.step(action)     # environment evolves according to action
cumulative_rewards += reward               # obtained reward added to running count
h = memory(action, z, h)                   # updates our memory model (RNN)
```

and this occurs for every time step t until done.

The authors test two situations with this model and report their findings: with a top-down car racing simulator, and a video game. What the authors find are state-of-the-art results for these situations.

2 Part II

In this section, we review the process of attempting to reproduce the original results of the World Models paper.

2.1 Computing Environment

The first that needs to be addressed is the computing environment. For this project, we were granted credits for AWS, the cloud computing platform by Amazon. Specifically, we were allotted a certain number of resources for training the model. The instance that this repository was tested and trained on was the Deep Learning AMI that sat on top of 100 gigabytes of disk space, 16 gigabytes of memory, and 4 virtual CPU cores running at approximately 2.3 GHz. Before even beginning the training, it is apparent that reproducing the exact results on this environment would take a significant amount of time.

The first bottleneck that was apparent to me was that my EC2 instance did not come with a GPU mounted. While this does not prevent any actual training from happening, it does hold us back in terms of peak training speed. Even more so, the GPU having its own memory would have helped the model train on larger batch sizes as these batches need to be in actual memory for any efficient computation. Overall, the lack of GPU (and what felt like the inability to attain one) does affect training speed and accuracy.

The next bottleneck of this computing environment was the amount of memory available. While 16 gigabytes is plenty for small batches and not-too-complicated models, in this instance, it was a drawback. Training the VAE requires episodes of observations, where each observation is an image of the world that is being learned. For this case, each image has a dimension of 64 x 64 x 3. When there are hundreds of observations in each series and hundreds of episodes per batch, this adds up. Thus, it becomes apparent that, typically, the more memory available, the more efficient the training. With a size of 16 gigabytes, I had trouble fitting more than 100 episodes in memory for training. Quite often, Tensorflow had issues being able to allocate all of the necessary space to hold the episodes after each epoch, and would kill the training. This led to many frustrations, and so I eventually settled using 50 episodes for each training epoch (with the standard 300 timesteps) for about 10 epochs. Clearly, with only so much data available for training the VAE, this is an area where training performance was severely affected.

The last bottleneck of the computing environment that was apparent to me was the availability of only 4 virtual CPU cores. Tensorflow is robust enough to allow efficient use of parallelism when it comes to training. For training the controller, we had the choice of number of worker threads for training our model. However, with only four virtual cores, it becomes a dicey game instantiating more threads than number of cores since plenty of overhead is incurred with all of the context switching of the threads. This is another area where performance was affected.

2.2 Results

Given all of the background about the computing environment, we can be sure that, before even looking at the data, it would be a Sisyphean task to completely reproduce the original results of the paper. I will go through each trainings and results.

2.2.1 VAE

The first model that was trained was the vision model. This has to be the case since the latent vectors for each of the observations are required by the other two models. After input data was generated using the script 01_generate_data.sh, the vision model was trained using python 02_train_vae.py –N 100. The results were checked using the jupyter notebook also present in the repository, 02_check_vae.ipynb.

From the notebook, it seems like the VAE was mildly successful in learning a suitable encoder and decoder. From the notebook, we see that the inputs and reconstructed outputs are generally pretty similar, at least for the images that were present in the notebook. This led me to feel comfortable about the training process for the VAE. Please refer to the notebook for the training's results.

2.2.2 RNN

The memory model is what is trained after the vision model, since we need to have a fully trained memory model before training our controller. The training of this model was straightforward, and I was able to use the default options for training the model. This was perhaps the least problematic model to train in my situation. The results were checked using the jupyter notebook present in the repository, 04_check_rnn.ipynb.

The results of this model were harder to understand. While the idea of training the model to predict what the next observation will be based off of memory, the statistics reported in the checking notebook were a bit obscure. As a result, please look at the notebook for results.

2.2.3 Controller

The controller is perhaps the most time consuming model to train in my case. Even more so, I was not able to attain scores even remotely close to what the paper had originally produced. My speculation is that this is because this is the most computationally intensive model to train. With the bottlenecks discussed earlier in this section, it becomes apparent that this would be an area where training performance would be drastically affected. As a result, the agent does not perform anywhere near as expected when compared to the original paper.

3 Part III

In this part, we were asked to integrate GANs into the World Models technique. Recall that a GAN has two components: a generator and a discriminator. In this architecture, we have a generator who learns to actually produce an output learned from data. At the same time, the discriminator learns to tell the difference between what the generator creates and actual real inputs. When these two things are paired together, what we end up with is something like a game between these two things: the generator learns to create more authentic works from the knowledge that the discriminator. With this game of cat and mouse, we hope to end up with a generator who is really good at creating things that resemble the original inputs, and the a discriminator who is really good at telling whether an image is authentic or not.

This idea can be integrated into the World Models technique. Rather than having just a VAE with an encoder and a decoder that learns from comparing the original input and the reconstructed output, we replace the decoder with a GAN. Now we would have three components to our vision model: an encoder to produce a latent vector, a generator that will produce reconstructions based on said latent vector, and a discriminator that will be able to tell the difference between the reconstruction and the actual input image. Having the three components, we can expect to have a model that will reconstruct the original input image with a high degree of accuracy.

3.1 Implementation

Specifically, I attempted to follow this implementation of VAEGANs: https://github.com/leoHeidel/vae-gan-tf2. Although I was able to somewhat shoehorn the implementation into the already-existing code of the VAE, I had a hard time getting the model to run and train. As a result, there are no results to be shared or discussed. However, I can speculate on the accuracy and training process for this approach. First, GANs can be computationally expensive to train. This is because we are now pitting against each other two NNs of comparable size. In this approach, we go from training two components (encoder and decoder) to training three (encoder, generator, and discriminator). From this alone, one can expect for more computations to be carried out for training. Second, I expect that, in the long run, the VAEGAN will produce better latent vector representations of images. However, it becomes a matter of accepting the trade-offs.