Max Kelsen

hello@maxkelsen.com

OpenAI Retro Contest Writeup

Contents

Introduction

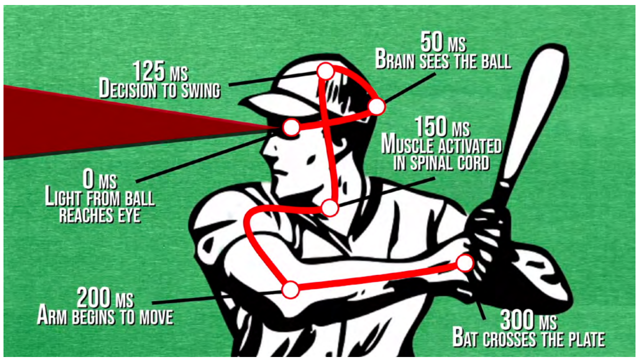
Thinking is hard. Ask someone how much time they spent thinking through their decisions and their answer would likely allocate far more time than what was true. There's a lot going on in the world, so much we've had to start outsourcing some of our thoughts.

Humans are experts at solving a wide range of challenging tasks and we're also great machines for looking backwards. Much of our problem solving involves retrieving past experiences from our memories which form the basis of our subsequent actions. This kind of problem-solving is efficient. Until recent history, conserving energy and making efficient decisions was of paramount importance.

Consider the scenario involving two of our ancestors trying to avoid a predator. Hidden in reeds of grass, a lion watches from afar, whilst our ancestors creep naively towards prey of their own. The lion decides it's feeding time and goes to pounce upon our distracted ancestors. One of them decides to stop and think through all of the available options to escape consumption. The other, despite never encountering a lion attack before, doesn't stop to think and instead acts upon the memories she's accumulated over years of storytelling around the fire. Who is more likely to end up as lunch?

For a more modern example, imagine a professional baseball player. If a batter was to think through all of the swing options they had before striking a ball, they wouldn't spend much longer on the team. Instead, much like our still-breathing ancestor, they act based off their accumulated memories. In essence, swinging without knowing exactly where the ball will go, but knowing where the ball might go.

Making decisions like these with the utmost of efficiency does come with a trade-off: humans are experts at self-delusion.



A baseball batter doesn't have time to fully calculate which is the best shot to take. Instead, the batter relies on memories from years of practice to make the decision subconsciously.
Source: World Models Explained by Siraj Raval.

"The first principle is that you must not fool yourself and you are the easiest person to fool." - Richard Feynman

Feynman knew how easy it was to get tripped up when our own version of the world clashes with reality.

Our bias towards efficiency makes it hard to replicate our problem-solving abilities. We rely on others being able to generalise as well as us and we get frustrated when they don't see the world through our eyes. We trick ourselves into believing our own model of the world is unique at the same time as believing it's the same as that of those around us.

Since being granted the privilege to spend long periods of time to think and the tools to outsource some of our thoughts to machines, it has been a dream of many to create a machine capable of thinking like us. But here we meet the self-delusion issue; we're great history machines but as soon as we try to explain why we are, our arguments break down almost immediately. Parallel to a knife being incapable of cutting itself or a fire not being able to burn itself, we haven't yet been able to explain how we think our thoughts.

Learning how to learn

The possibility of a thinking machine has been around for almost two centuries (potentially longer). Recent advancements in computing technology and the crossover of different fields such as neuroscience, philosophy and mathematics have made this possibility seem far more real.

The definition of artificial intelligence differs depending on who you talk to. But many would agree that an **agent** capable of solving a variety of challenges without being explicitly taught how to do so could be considered intelligent. For the rest of this write-up, we will take this as our overarching definition of what it means to be intelligent.

If learning is the ultimate meta-skill, how does an agent learn how to learn?

This is where the concept of World Models comes into play. To further the work of Ha and associates, we have attempted to adapt their World Models architecture to build a game-playing agent which is capable of playing unseen levels of Sonic The Hedgehog™.

The World Model of a Hedgehog

On April 5 2018, OpenAI announced a transfer learning competition. Participating teams had two months to build reinforcement learning (RL) algorithms capable of transferring knowledge acquired from playing levels of Sonic The Hedgehog™ 1, 2 and Sonic 3 & Knuckles™, to previously unseen levels.

Original cover photos of the Sonic SEGA Genesis series.
Source: Steam

Why bother with games?

Games are the perfect test-bed for building and rapidly testing RL agents — agents which may eventually be able to be used within different fields outside the world of gaming. Games offer repeatable states on demand, can be easily shared amongst groups of researchers and don't require vast amounts of hardware to get started.

How does our Sonic playing agent learn to think?

As described in the World Models paper, our agent creates a World Model of the game **environment**, learns to play in this World Model and then brings this knowledge back into the real environment.

But wait, what's a World Model?

Remember our ancestor friend from before? The one who managed to survive. Her World Model could be considered as being the accumulated memories she had built up after listening to years of stories about lions around the fire. She imagined what it would be like if one day she did come face to face with a lion, and at the same time, ran through various escape scenarios.

Humans are very good at this. Another example would be crossing the road. You don't have to be hit by a car to imagine what it's like being hit by a car. We don't even have to see someone being hit, we can create a scenario in our head of what it's like being hit by a car, imagine what might lead to this scenario and subsequently avoid those actions. This ability to imagine potential future scenarios is what separates us from other species.

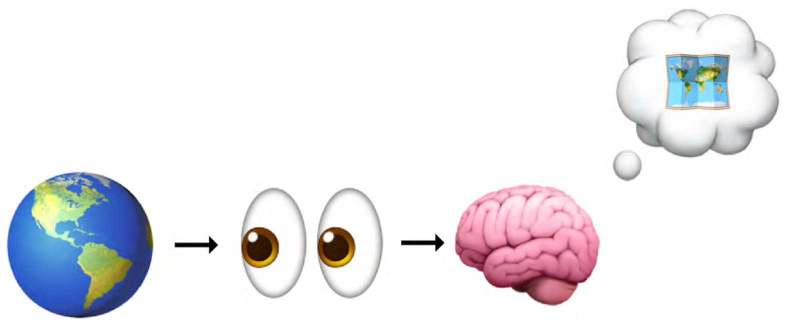
Imagining future states and how actions would affect them can be considered the fundamental concept of a World Model.

Remember our ancestor friend from before? The one who managed to survive. Her World Model could be considered as being the accumulated memories she had built up after listening to years of stories about lions around the fire. She imagined what it would be like if one day she did come face to face with a lion, and at the same time, ran through various escape scenarios.

Humans are very good at this. Another example would be crossing the road. You don't have to be hit by a car to imagine what it's like being hit by a car. We don't even have to see someone being hit, we can create a scenario in our head of what it's like being hit by a car, imagine what might lead to this scenario and subsequently avoid those actions. This ability to imagine potential future scenarios is what separates us from other species.

Imagining future states and how actions would affect them can be considered the fundamental concept of a World Model.

To create a World Model of the Sonic game environment, our agent first visualises human players successfully completing levels of the game through a Vision model (V). It records and remembers what it sees in a Memory model (M) and makes decisions on what action to take based off what it has seen in the past through a Controller model (C). We will explain each of these in more depth later on.



Observing the world.

Artificial Intelligence = Reinforcement Learning + Deep Learning

David Silver, co-founder of DeepMind has stated true artificial intelligence (AI) or artificial general intelligence (AGI) is the combination of reinforcement learning and deep learning (DL). As you learn more about the crossover of these two techniques, his statement becomes more valid.

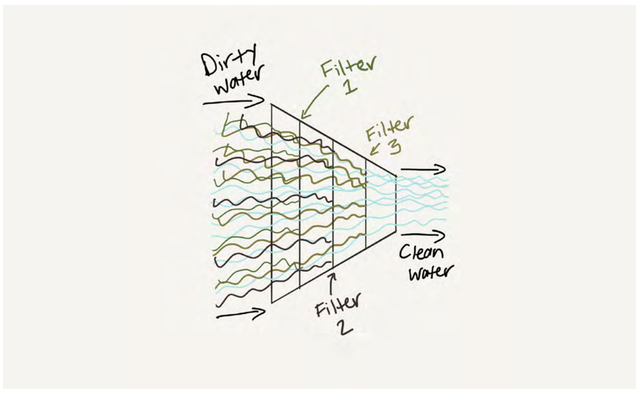
Reinforcement learning

Reinforcement learning encourages learning through feedback in the form of rewards and punishments (or lack of reward). For example, if your dog goes to the bathroom inside, you probably won't offer it a reward, hopefully reinforcing the opposite behaviour. If it goes to the bathroom outside you might reward it to encourage that particular behaviour.

How one might use reinforcement learning to train their dog to go to the bathroom outside (method not guaranteed).

Deep learning

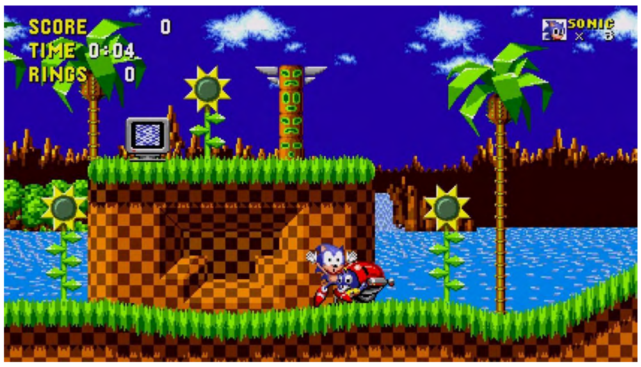
Deep learning could also be called feature learning. Consider a funnel with a series of filters throughout. Through the top of the funnel, you feed through dirty river water. At each subsequent filter, a particular kind of dirt is removed from the water. This would eventually lead to you finding the most consistent underlying pattern throughout the original dirty water input, i.e., clean water. Each filter abstracts away a feature of the dirty water.



Note that we can't directly decide/know which type of dirt (feature) gets filtered out at each layer. The network (funnel) learns this on its own.

For a Sonic game playing agent, it would probably be wise to reward a behaviour such as completing a level. As for what features are important to learn, it's much harder to say.

There's a score, there are some edges (ground/walls), there are enemies. All of these may contribute to the agent's World Model.



When you look at the Sonic game window, what's important to you?

Putting together our own World Model

Starting the competition we spent a few days getting acquainted with what OpenAI released as introductory information.

Having minimal experience with RL we turned to some fundamental resources, namely David Silver's brilliant course on YouTube. We also spent a lot of time in the OpenAI Discord chat asking questions and reading the troubleshooting steps of fellow competitors.

We began by running through the setup steps provided by OpenAI on the competition details page. The steps were straightforward but, due to some dependency issues, took the better part of a day.

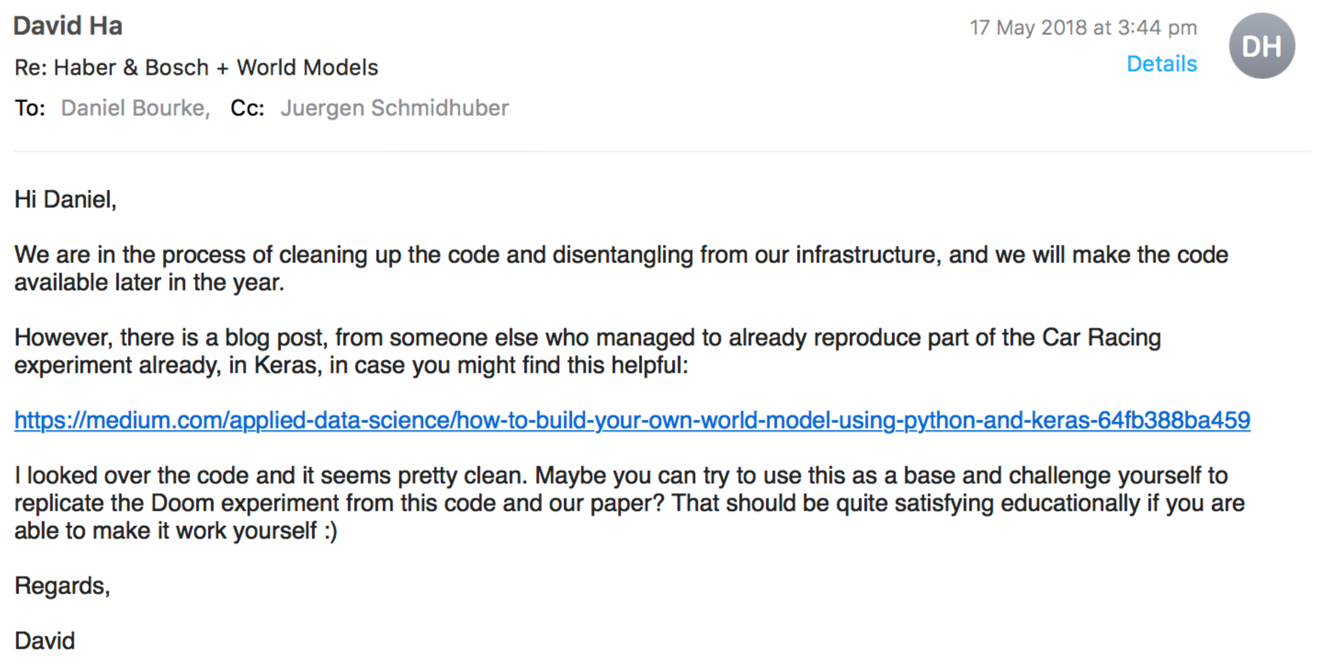
OpenAI released some baseline algorithms such as JERK, PPO and Rainbow DQN as starting points to add our own takes and expand upon. After a couple of attempts, we never managed to get a baseline algorithm running properly enough for submission to the leaderboard.

Later, we decided to reattempt implementing the Rainbow DQN algorithm when a video from 2-minute papers was sent through to the team on Slack. The video turned out to be a distillation of the previously mentioned World Models paper.

The timing couldn't have been better. An AI learning from its dreams? What? We had no choice but to try and apply it to Sonic.

Having three weeks left to complete the competition, we reached out to the authors of the paper asking if they had released any example code from the project.

David Ha, one of the co-authors, replied promptly with a link to a Medium article where David Foster from Applied Data Science posted his implementation of the paper. Thanks, Davids!


David Ha
Re: Haber & Bosch + World Models
To: Daniel Bourke, Cc: Juergen Schmidhuber

17 May 2018 at 3:44 pm
Details

DH

Hi Daniel,

We are in the process of cleaning up the code and disentangling from our infrastructure, and we will make the code available later in the year.

However, there is a blog post, from someone else who managed to already reproduce part of the Car Racing experiment already, in Keras, in case you might find this helpful:

https://medium.com/applied-data-science/how-to-build-your-own-world-model-using-python-and-keras-64fb388ba459

I looked over the code and it seems pretty clean. Maybe you can try to use this as a base and challenge yourself to replicate the Doom experiment from this code and our paper? That should be quite satisfying educationally if you are able to make it work yourself :)

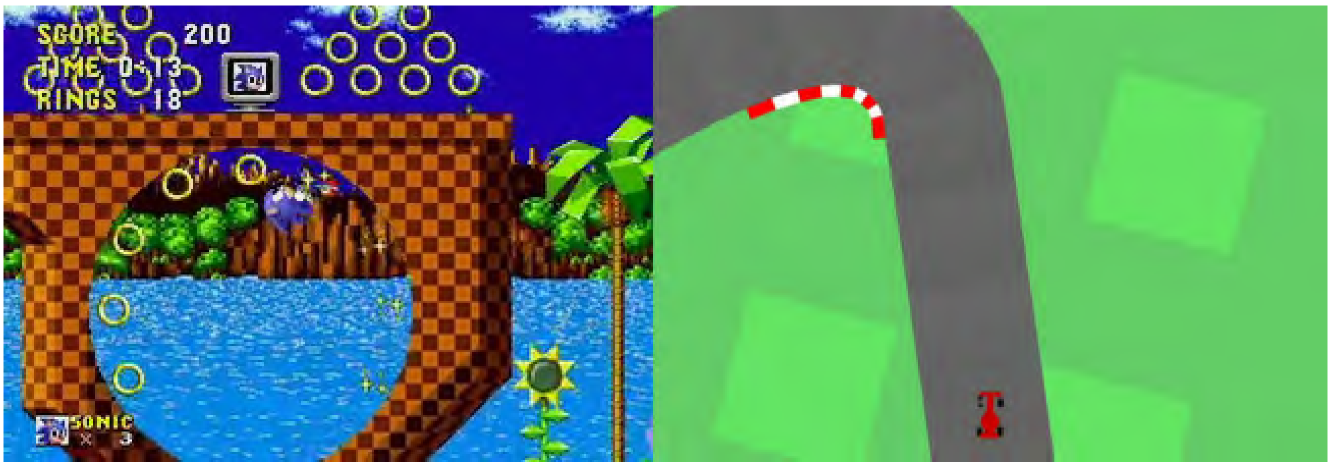
Regards,

David

Going straight to the source.

Although an example of the code was available, it was for the Car Racing environment used in the original paper. We had a lot of work to get it running on Sonic. Even for someone completely unfamiliar with both games, you can see straight away how much more complex Sonic is compared to the Car Racing game.

What mattered is we had begun to formulate our own World Model of the project ahead of us. Like the agent imagining its own scenarios in the World Models paper, we were imagining how to create our own game playing agent.

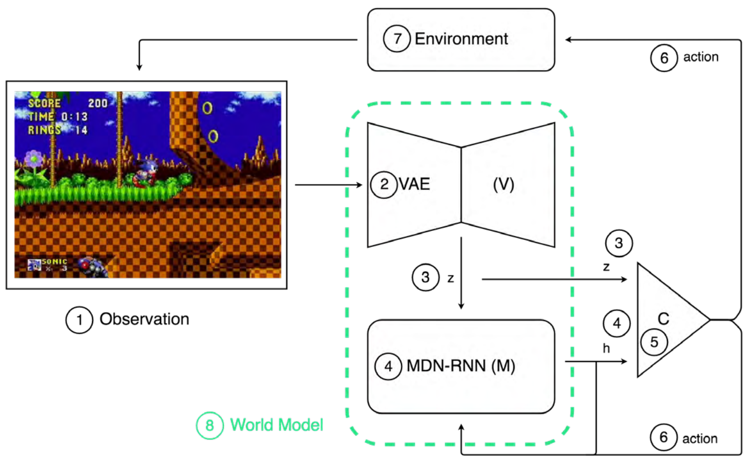


A battle of complexity: Sonic vs. Car Racing. Sonic: 1, Car Racing: 0.

Observe, Remember, Act, Repeat

It was time to teach our Sonic how to play, or better yet, let it learn how to play.

Much like the original paper, our model is comprised of three smaller models. We took the foundations laid by David Foster and adapted them to work with the Sonic environment.



An overview of the World Models architecture described in the original World Models paper.

The following process is repeated at each time step:

1. The OpenAI Gym environment wraps around the Sonic game making useful parameters such as pixel data observable.

2. Vision model (Convolutional **Variational Autoencoder**) accepts pixel data (224x320x3) from the observation as input and starts encoding it.

3. The encoded pixel data is exported as z (a **latent vector**) and passed to the Memory model and Controller (single layer linear model).

4. Memory model (**Recurrent Neural Network** with **Mixture Density Network** as output) accepts *z*, hidden state, *h* (from itself), and *actions* as input.

5. The Controller takes in z and h and outputs actions for the agent to take based on these parameters.

6. *Actions* decided by the Controller are fed back into the Memory model and to the environment.

7. The environment wrapper translates actions from the Controller into the game where further observations take place.

8. Most of the information regarding the environment and previous steps taken is contained within the Vision and Memory models. Because of this, a World Model can be constructed from these two. *Note: In our approach, we have yet to utilise this feature.*

Vision Model (Variational Autoencoder - VAE)

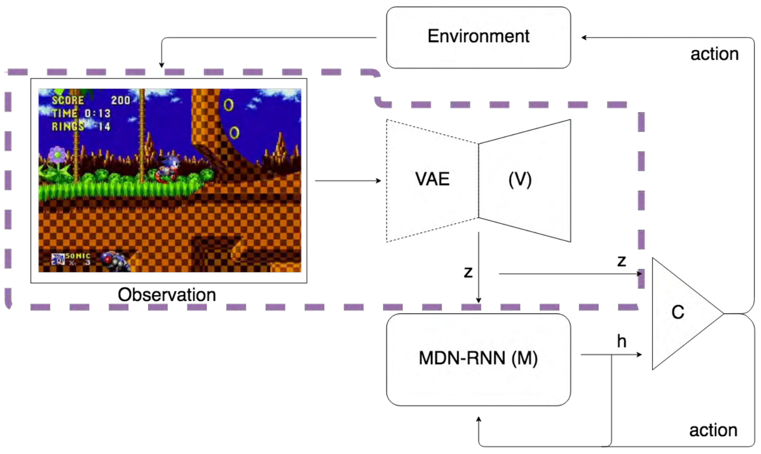
In the World Models implementation by David Foster, data for the Car Racing environment is generated by taking observations of randomly chosen actions across a series of time steps. In our experience, replicating this at the scale done in the article was a highly time-consuming step. And this is only with a simple game environment.

We realised it would take far too long to do this step for Sonic. Our hope of implementing a World Models equivalent for Sonic hit its first roadblock. Until, in another twist of fate, OpenAI released a dataset of recordings of their team successfully completing each level of the entire Sonic trilogy.

One of our team members discovered this and stayed up into the early hours of the morning mapping the recordings to the implementation of the variational autoencoder (VAE).

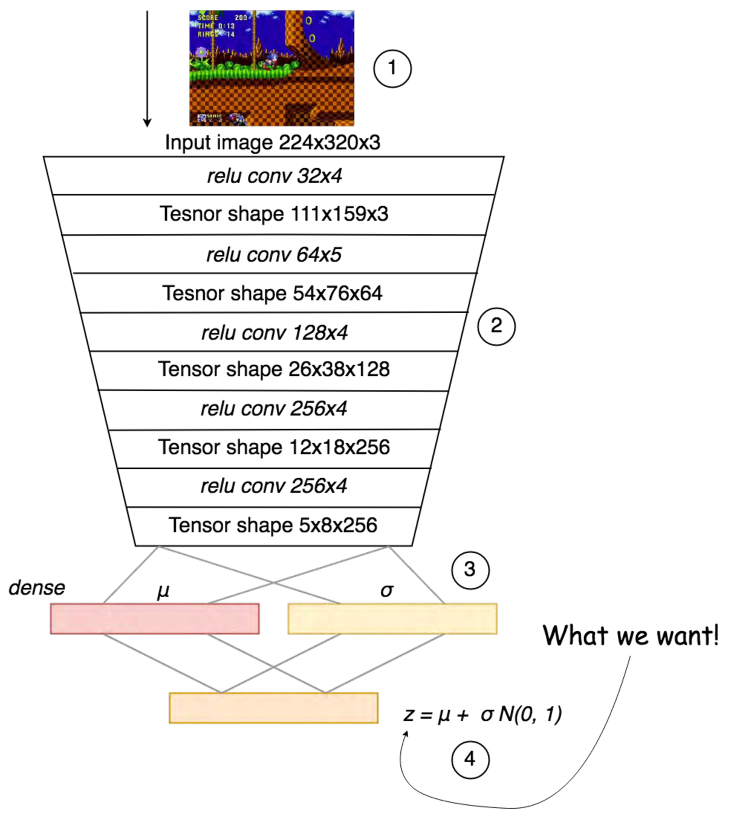
Suddenly, our VAE was encoding the high-dimensional series of 2D images of the OpenAI researchers playing Sonic.

Researchers from the original paper used a length 32 latent vector, z, for the Car Racing game and a length 64 latent vector, z, for Doom (a slightly more graphically intense game the researchers adapted their World Model architecture to). Knowing this, and due to the increase in complexity of Sonic compared to the Car Racing game and Doom, we decided to use a length 128 latent vector for our Sonic edition of the model.



Vision model portion of the World Model. The first half of the VAE architecture (dotted black line) is described in the figure below.

The latent vector, z, is created by passing the high-dimensional pixel data (224x320x3 pixels = 215,040 data points) from the game space through the first half of the VAE (the encoder portion).



The encoder portion of the VAE. See the supporting materials for the full VAE.

1. Pixel data from the game space is fed as the input to the model (224 high, 320 wide and 3 deep for the red, green and blue colour channels).

2. The input data is transformed through a series of five convolutional layers, depicted as Activation-type Output Channels x Filter Size.

3. Layer 5 is encoded into two low-dimensional vectors, **μ** (mean) and **σ** (standard deviation), each of size 128Nz.

4. The latent vector z is sampled from the **Gaussian prior**, N(μ,σI), and is also of size 128.

Compressing the pixel data into *z* is useful because it yields a mathematical representation of the image which requires far less computing power to perform calculations on. It is similar to us visualising a scenario in our mind. It may not be the entire picture but often, the most important details are there.

The second half of the VAE (the decoder portion) takes *z* and reconstructs it into a full resolution image. But because it's a variational autoencoder, the reconstruction is done based on *z*. Thus, a reconstructed image coming out of a VAE will always be slightly different than the original input, as *z* is sampled from a probability distribution over the original image. In the paper, the authors used a combination of these deconstructed images to help their agent learn within its dreams.

Now our agent can see the world but its memory is worse than a goldfish. It forgets what it sees at each time step, almost immediately. We can fix this by attaching a memory model.

Vision model in a nutshell:
Looking at the game space, what is the most important information and how can it be represented in a more condensed manner, z, to learn from?

[0.382, 0.229, 0.745, 0.197, 0.665... x 128]
VAE
z

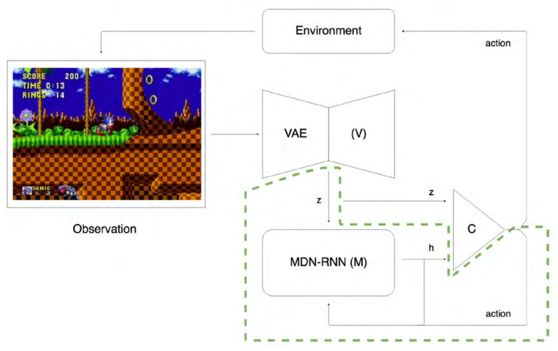
z is essentially a list of numbers which represents a 2D image of the Sonic game space.

How is the Vision model trained? Using the backpropagation algorithm.

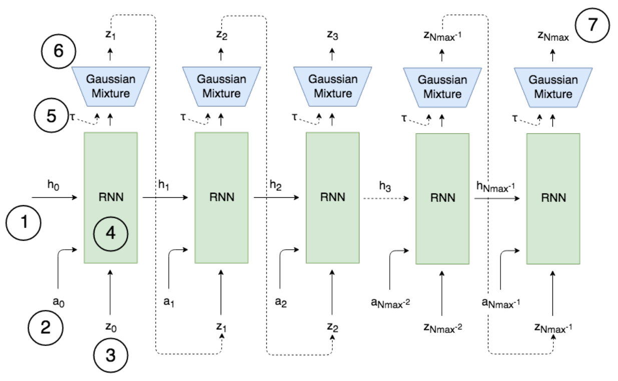
Memory Model (Mixture Density Network + Recurrent Neural Network or MDN-RNN)

If you were to start playing Sonic from scratch, after a short while, you'd start to build an idea of how the physics work and the underlying structure of each level. With this foundation of knowledge, you could start to predict what may happen next in the game and prepare for it.

The prediction of what may come next is the role of the Memory model. A Mixture Density Network attached to a Recurrent Neural Network (MDN-RNN), specifically a Long Short-Term Memory Network (LSTM) with 1024 hidden units.



The Memory model section of the architecture is highlighted by the green dashed line and is expanded in the figure below.



1. The hidden state, *h0*, of the RNN is fed to the first LSTM cell (green) at time 0.

2. The *action0* chosen by the controller is given to the LSTM at time 0.

3. The z0 latent vector from the Vision model is passed to the first LSTM cell at time 0.

4. Given the inputs, the LSTM performs the required calculations on the inputs and pushes its hidden state, *h1*, to the next LSTM cell and *z1* to the Mixture Density Network (MDN) output layer (blue).

5. A temperature parameter, τ, is fed to the MDN to control model uncertainty. Note: this step has not yet been replicated in our work.

6. The probability of the next *zt+1* is chosen from a range of likely possibilities based on the previous inputs *at*, *zt*, and *ht*.

7. This process is repeated for *N* (length of z) times.

Similar to the Vision model compressing what the agent sees, the role of the Memory model is to compress what happens over time with the goal of eventually being able to predict the future. Like a human playing Sonic predicting what the next scene might look like, the Memory model attempts to predict what the next *z* vector will be.

An MDN is used as an output layer to account for the randomness of the game space. This means, instead of outputting a deterministic z value, the Memory model outputs a range of possibilities rather than only one. In other words, a range the next *z* values should fall within.

Memory model in a nutshell: Given the latest observation from the Vision model, *zt*, the last action the agent took from the Controller model, at, and the previous hidden state of the RNN, *ht*, what will the next observation, *zt+1* likely look like?

How is the Memory model trained? Using the backpropagation algorithm.

Controller Model (CMA-ES Feed Forward Neural Network)

Okay, so you've been watching the game and you've developed an understanding of how your actions may influence Sonic, what do you do next? You keep playing the game and make decisions based off what you've seen.

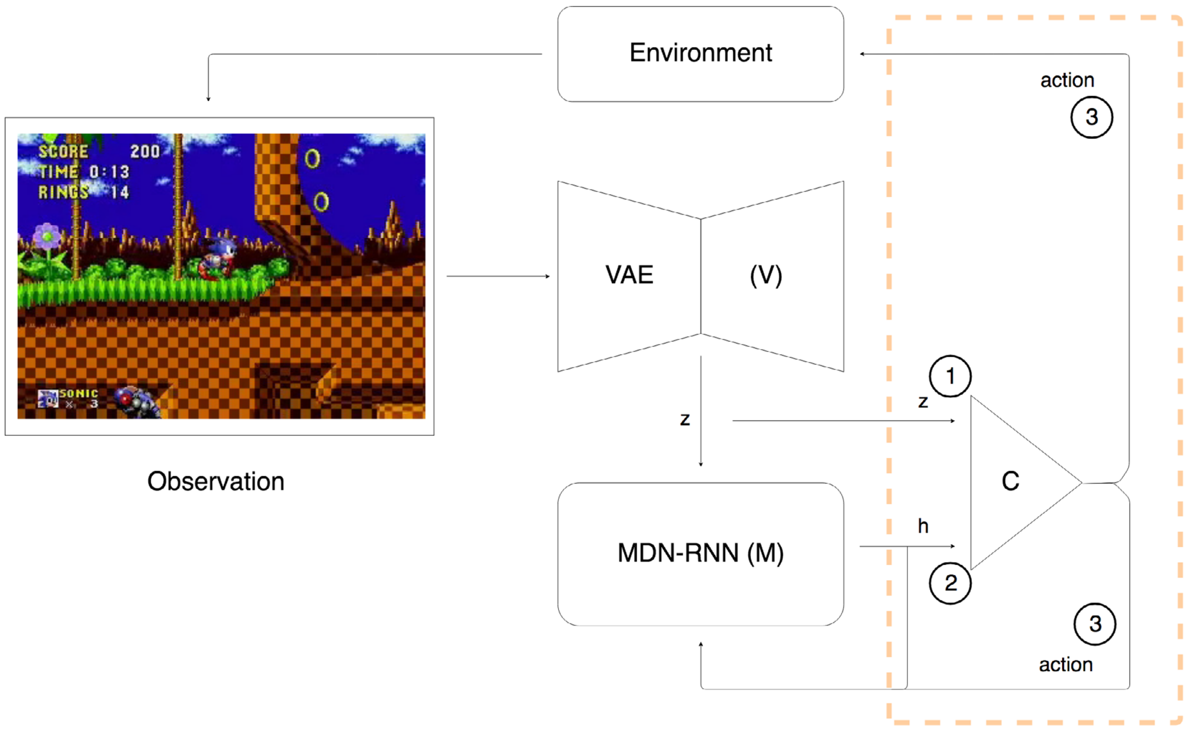
Easy, right?

$$a_t = W_c \left[z_t \; h_t \right] \; + \; b_c$$

The single layer linear function to calculate the most rewarding action at each time step.

This is where the Controller model comes in. Remember how reinforcement learning occurs through rewarding good actions and not rewarding bad actions? The role of the Controller is to maximise rewards.

The authors of the World Models paper deliberately made the Controller a simple single layer linear model which maps the Vision model observation (*zt*) and the Memory model hidden state (*ht*) to an action (*at*) at each time step.



Highlighted portion of the Controller within the World Models architecture.

1. Latent vector, *z*, of the encoded observation data is passed fromå the Vision model to the Controller.

2. The hidden state, *h*, of the Memory model is passed to the Controller.

3. Based on the inputs, the Controller outputs actions which maximise rewards and passes them to the Memory model and to the environment for Sonic to execute.

To find the optimal parameters (best W and b) of the Controller, a Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) algorithm was used. This means, several variants of the Controller were trialled across multiple CPUs at the same time and similar to natural selection, the best values were kept and the rest were discarded.

Controller model in a nutshell: Based on the observation received from the Vision model, *z*, and the hidden state, h, of the Memory model what is the best action for the agent to take at this time step?

How is the Controller model trained? Using a CMA-ES algorithm.

Three models walk into a bar (putting it all together)

Vision: Guys I can't stop watching this blue hedgehog run around on a screen, I don't have time to explain the whole story (full 224x320x3 game space) but here are the most important parts (*z*, length 128 vector).

Vision passes the latent vector, z, to Memory and Controller.

Memory: Wow, that's a lot to take in. Let me start writing all of this down (*h*), so the hedgehog went up in the first step (*a1, z1*) and then left in the second step (*a2, z2*). I wonder how the hedgehog decided what to do?

Memory passes the hidden state of the RNN, h, to Controller.

Controller: I can help there. Memory, pass me your notes (*h*) and Vision, I'll need you to keep your story going, tell me as it happens (*z*). Okay, I think I get what's happening, Memory, what do think of this action (a)?

Controller passes the action to Memory and back to the screen.

Memory: Well based on what I've seen before, I think you're on the right path, Vision, what are you seeing?

Vision checks out the effects of the action.

Vision: The numbers on the screen are going up! Keep doing what you're doing!

Vision sends a new z back to Memory.

Memory: This is easy! Vision, keep updating me on what you're seeing (*z*) and Controller tell me what action (*a*) is best, I'll remember it all (*h*).

Memory keeps note of what Vision has seen in the past and how new actions affect z.

Controller: And I'll decide what's best as long as Vision keeps updating me with what's on the screen, and you keep passing those secret notes you're keeping.

Memory continues to pass what it remembers (h) to Controller and likewise with Vision and what it sees (z). Controller then makes a judgement of what the best action (a) is to take next, based on the agent's memories of the environment and what it sees within the environment.

How does our agent play?

All of this sounds great but how does our agent actually play?

In the Car Racing example, the researchers were able to solve the game, achieving an average score of over 900 points using their World Models approach. Knowing this, we were hopeful to get our agent up to a level comparable with the baseline algorithms released by OpenAI. With this goal, our agent would have to finish with a score of 3000 or more as judged by OpenAI's scoring criteria.

To summarise our Sonic playing agent, the following steps are taken:

1. Generate data based on human players.

2. Train the vision model (VAE) on this data to represent the game space in *z* (length 128 latent vector).

3. Train the RNN to model the probability of zt+1 given ht, at and zt.

4. Define the controller as *at = Wc[zt, ht] + bc.*

5. Use CMA-ES to solve for *Wc* and *bc* that maximises the reward.

Our work-in-progress repo is available on GitHub.

Step 1 - Generating data

After watching OpenAI researchers play, our agent had gathered 448GB of numpy arrays filled with pixel data.

However, training directly on this data would require computing resources not yet reasonably available.

Like our brains filtering out excessive information taken in by our eyes and focusing on what's important, the role of the Vision model would be to encode this pixel data into a more manageable variable.

Step 2 - Training the Vision model

The Vision model was passed the 448GB worth of pixel data and compressed it into a series of 128 latent variables (581.8MB - a ~1000-fold decrease in size) for each time step, zt, to be fed into the Memory model.

What's important to note is it's unknown what each of the latent variables represents. The authors of the original paper have a great representation of what each latent variable encodes on their blog post under the VAE model section.

For example, one of the 128 latent variables may encode for the colour blue, another may represent the edges of the obstacles. Each one contributing a small part to the overall image.

It will be future work for us to replicate this ability to visualise what each z value represents.

Step 3 - Training the Memory model

Our Memory model takes in the encoded pixel data from watching the researchers play, as well as actions performed by the agent at each time step thanks to observations gathered by the environment.

Using the *.make()* and *.step()* methods from the OpenAI Gym Environment, information such as observation and reward at each time step are recorded.

Our Memory model was then trained to learn the relationship between each action and the pixel data at each time step.

Step 4 - Defining the Controller and evolving weight and bias parameters

As mentioned in the Gotta Learn Fast paper, there are 12 potential buttons which can be pressed in the Sonic environment, compared to 3 within the Car Racing game.

[B, A, MODE, START, UP, DOWN, LEFT, RIGHT, C, Y, X, Z]

Possible button presses with the SEGA Genesis controller.

In the Sonic environment, a 1 was used to define a button being pressed and a 0 to indicate the opposite.

So a ‘B’ button press would look like:

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

For multiple button presses at the same time, more than one number would have a 1 value.

It’s the role of the Controller to figure out the right combination of 0’s and 1’s (*actions*) to take at each time step.

Step 5 - Observing the agent play

After 12 hours of training on 72 cores we were excited to finally watch our agent play, however, our excitement was short lived.

Our agent struggled to make it past the first spring pad on Green Hill Zone Act 3.

http://bit.ly/mkagentfailing

Our Sonic was stuck jumping and spinning around in circles at the start of the level.

In an attempt to debug, we set print statement to show what action our agent was choosing at each time step.

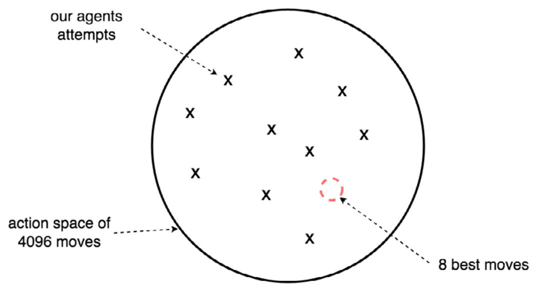
It turns out, our agent was randomly selecting combinations of button presses, trying to figure out which were best. Of course, this led to little or no progress.

Digging deeper, we found there to only be eight button combinations which were of high value in Sonic, the rest didn’t really matter.

Instead of focusing on the eight combinations which mattered and learning from those, our agent was wildly exploring a search space of 4096 options (212), a ~500-fold increase.

[NONE, LEFT, RIGHT, (LEFT, DOWN), (RIGHT, DOWN), DOWN, (DOWN, B), B]

With these button combinations, one can successfully complete every level of Sonic.



Instead of trying to explore the dotted red circle of the 8 best moves, our agent was searching for the right move in a pool of a possible 4096.

We discovered this 8 hours before we were supposed to submit our algorithm to the competition. Due to previous commitments, we were unable to implement our changes in time for submission.

Next steps

After a couple of weeks tinkering with the problem of getting an agent learning to play Sonic, we ended up with an agent whose results were comparable to that of someone giving the SEGA controller a massage.

Despite our failure to produce something comparable to the baseline algorithms, we were left with a long list of next steps and learnings which can be applied to future works.

How could we improve?

The following are approaches we can try for fixing our World Models implementation, as well as for future competitions.

1. Reducing the search space

Instead of trying to find the right actions amongst a haystack of 4096 possible options, we could reduce it to 8. This massive decrease in options is likely our next move.

2. Additional RNN inputs

For the Doom task in the World Models paper, the Controller took the hidden state, h, as well as the cell vector, c, from the RNN as input, while our model only takes h.

3. Generating random rollout data

Our agent learns from data generated from videos of OpenAI researchers playing Sonic. However, this was only one run through of each level. Future works may look into creating random rollout data of Sonic exploring the game space and learning from this.

4. Rewriting our agent from scratch

In an attempt to get a head start, we relied heavily on David Foster’s implementation of World Models. Taking this shortcut ended up in a lot of dependency issues and us ultimately having to try and reverse engineer much of the code. Starting from scratch may have taken longer but would result in more control.

5. Spending more time getting acquainted with the problem

we faced our own exploration versus exploitation dilemma. There will always be new ways to look at a problem but for now, applying our own version of the World Models technique seems like a great next step.

A new world

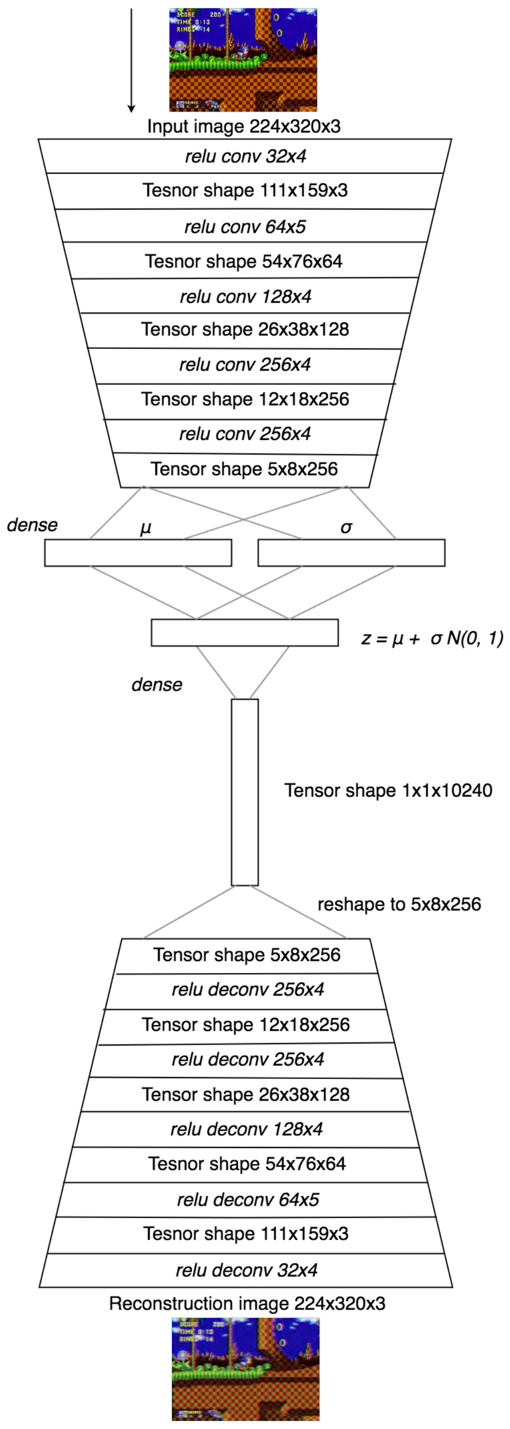
We’re all explorers at heart. This is why deep reinforcement learning is such an exciting field. From what we know so far, it’s the closest we’ve come to replicating how we think.

Although our Sonic-playing agents World Model didn’t improve much, our World Model of RL has come leaps and bounds.

For now, we better get back to learning. There’s still plenty of work to do.

As a final note, the authors would like to devote a special thank you to Max Kelsen for providing the resources to explore this problem, OpenAI for creating an incredibly motivating competition and all of the participants in the Discord chat for offering their support and advice.

Appendix



Our full VAE model (encoder and decoder layers).

Key terms

Environment - a defined space where an artificial agent can explore and gather information.

Agent - an autonomous entity which observes an environment through sensors and acts based on these observations.

Latent variable/vector - a variable not derived from direct observation but from probabilistic means.

μ - symbol for mean.

σ - symbol for standard deviation.

Gaussian - another term for normal distribution.

RNN - recurrent neural network.

VAE - variational autoencoder.

MDN - mixture density network.

Resources

The first sign of thinking machines: http://www.scienceclarified.com/scitech/Artificial-Intelligence/The-First-Thinking-Machines.html

World Models Paper: https://arxiv.org/abs/1803.10122

OpenAI Retro Contest Blog Post: https://blog.openai.com/retro-contest/

World Models Blog Post: https://worldmodels.github.io/

David Silver's RL course on YouTube: https://youtu.be/2pWv7GOvuf0

OpenAI Discord chat - https://discord.gg/chU7Zwa

OpenAI Baseline Algorithms - retro-baselines/agents at master · openai/retro-baselines

Recordings of OpenAI researchers playing Sonic - https://github.com/openai/retro-movies

Backpropagation Algorithm - https://en.wikipedia.org/wiki/Backpropagation

Covariance-Matrix Adaptation Strategy - CMA-ES - Wikipedia

Siraj Raval Video Explaining World Models - World Models Explained - YouTube

Steam store

Our unfinished code on GitHub - https://github.com/MaxKelsen/WorldModels

Big data. Big ideas.