An Exploration of Neural Networks Playing Video Games

Joshua J Luo  Follow
May 21, 2019 · 13 min read ⭐

A brief look at a curiosity-driven model playing Mortal Kombat 3 for the Sega Genesis. The curiosity-driven model is the player on the left.

Thank you to my team members: Benjamin Guo, Christian Han, Cooper Shearer, Justin Qu, and Kylar Osborne.

Introduction

Video games aren’t just fun. They provide a platform for neural networks to learn how to interact with dynamic environments and solve complex problems, just like in real life. Video games have been utilized for decades to evaluate artificial intelligence performance and have been recently featured in the news with companies like DeepMind successfully developing AI systems that are able to defeat top e-sports professionals.
Our team wanted to explore how various machine learning algorithms would perform playing some of our favorite, classic video games. Our goal was not to create the best model specific to a particular game, but to, instead, observe their behavioral characteristics and their suitability for different game tasks.

**Background**

Before we start diving into the details of our project, let’s take a look at how these models work on a basic level and how we interfaced with video games.

**Reinforcement Learning**

Reinforcement learning is an area of machine learning. It is unlike supervised learning because labelled inputs and outputs are not needed. Instead, an agent performs some action in an environment, and the environment returns an observation and a reward. Based on this reward, the agent can learn to continue to exhibit its current behavior or avoid it. This feedback loop continues as the agent attempts to maximize its reward.

To put this in a more concrete context, let’s imagine a game of Mortal Kombat where the agent is our fighter and the environment consists of the opponent and the stage. If our fighter punches and is able to lower the opponent’s health, then we may be rewarded for our action. On the other hand, if our fighter runs and the opponent is able to kick us and lower our health, then we may be penalized for our action.

**Interfacing with Video Games**

In order to interface machine learning algorithms with video games, we utilized Gym and Gym Retro which are reinforcement learning tools developed by OpenAI. Gym allows us to create the aforementioned agent-environment system, and Gym Retro
allows us to emulate and run retro games from consoles like the Atari, Nintendo Entertainment System, and Sega Genesis by giving us access to things like in-game variables and save states.

The neural network has access to predefined inputs and each input causes an agent to perform an action. For example, for Mortal Kombat 3 on the Sega Genesis console, a neural network would only be able to choose from the six buttons and the directional arrows on the original Sega Genesis controller. Each button is linked to a specific action (e.g. the “A” button corresponds to a Low Kick move). By using these inputs to interact with the environment, the neural network is able to learn from the observations and rewards; eventually, it is able to learn how to correctly respond to situations and win the game.

**Input Preprocessing**

The algorithms were given each frame of the video game as an image input. A single level played may consist of tens of thousands of frames, and each individual frame contains a lot of information already. Because of this, one of the major aspects to this project was preprocessing these image inputs in order to balance the time it took to train with the neural network’s performance. The computation-reducing preprocessing methods worked, but some of the more advanced techniques (excluding edge detection) that would increase performance did not seem to help our models.

**Scaling Inputs and Limiting Colors**

In order to reduce computation time while maintaining the essential features of the data, we converted many of our images down in size and removed the color. For example, the Sega Genesis emulator has an original size of 320x224 pixels. We scaled this down to about half the size and removed the color.
Reducing input image size by half and removing colors

Limiting Inputs
For each game, there were particular actions and buttons we did not want the agents to take. For example, in Sonic The Hedgehog, using the “Up Arrow” would allow Sonic to look upwards, and while training, we found that some instances would only look up instead of progressing through the level. We limited what inputs were accessible in order to speed up the training process.

Sonic really likes looking at those clouds...

Frame Skipping
We actually ignore some of the frames given as input because not all of the frames contain important information. For example, take a look at the 4 frame sequence below. It is difficult to differentiate between the first two frames.
By skipping some frames, we are able to retain enough data to discern important features like movement while reducing computation and training time.

Sequence of frames after applying frame skipping. Frames in yellow are used for input while frames that are crossed out are ignored.

**Frame Stacking**

By looking at this single image, it is almost impossible to tell which way the jumping character is moving.

By frame stacking, we can combine multiple frames into a single input for the model. Our model is actually choosing actions based on a series of frames that are “stacked
together” instead of just a single frame. This allows the model to discern things like which way the character is moving or its velocity. Unlike our other methods of preprocessing, this technique actually increases the computational load, but the information that can be gained by frame stacking outweighs the cost.

![Sequence of frames](image)

Background Removal

While much of our preprocessing was aimed at reducing information to decrease computational complexity, the idea behind background removal is to reduce noise. We found that some of our trained AIs exhibited uncharacteristic behavior when they reached a new level with a different background.

We tried many different background removal algorithms from the OpenCV library; however, there currently does not exist an algorithm for removing dynamic backgrounds. An alternative to removing the background would be to train the AI on various levels with different backgrounds.

![Background removal](image)

The background removal seems to be effective until the background begins scrolling.
Object Tracking

After our lack of success with background removal, we considered simply tracking the player object and opponent in Mortal Kombat 3. Unfortunately, the existing trackers in the OpenCV library either increased the processing time of each frame by a factor of 20x or were not able to maintain a lock on the desired target. Additionally, they required an initial starting box to be placed on the object, which had to be selected manually or required some object detection software.

In addition to these drawbacks, object tracking would only be applicable to Mortal Kombat 3 where there are always only 2 characters and no other relevant objects. In Sonic The Hedgehog and Super Mario Bros., simply tracking the main character would remove relevant environmental information such as Goombas or powerups.

Object Detection

In order to combat the drawbacks of object trackers, we decided to try using an object detection library called YOLOv3. There were several versions of the YOLOv3, each a different size of a pre-trained neural net. While the baseline version worked well at identifying the humanoid figures in Mortal Kombat, it took about 5 seconds to run on a single frame, making training nearly impossible on our current hardware. (To put that into perspective, we were currently training at over 60 frames per second.) We also tried the “tiny” version of YOLOv3, which included a smaller neural network, and this proved to be fast enough; however, it was not able to consistently identify the characters likely due to the lower resolution.
In addition to the lower accuracy, it was completely unable to detect anything useful in Super Mario Bros and Sonic The Hedgehog because it was not trained to identify the various enemies and items in either of those games. Due to a lack of time, we were unable to implement some sort of object detection preprocessing because it would have required our team to not only retrain the models, but to also find and make a training set of all the important game objects.

**Edge Detection**

Another attempt to reduce noise was through edge detection. The edge map converts the grayscale frames into binary frames, further reducing the input complexity. However, generating the edge map did add a small overhead to each step. Initially, our team was concerned that too much detail was lost and that the neural network would not be able to discern itself from its opponent in Mortal Kombat, but brief testing proved that this was not the case. When combined with a model, this preprocessing technique allowed us to achieve our best results on Mortal Kombat 3.

![Edge detection](image)

*Edge detection proved to be much more effective than we initially believed*

**Neural Networks**

To observe how machine learning algorithms performed while playing video games, we used three models: NeuroEvolution of Augmenting Topologies (NEAT) from NEAT-Python, Proximal Policy Optimization (PPO) from OpenAI’s Baselines, and a curiosity-driven neural network from researchers from UC Berkeley, University of Edinburgh, and OpenAI.

We trained all of these models on Mortal Kombat 3 for the Sega Genesis Console. Additionally, we were able to train NEAT on Sonic The Hedgehog for the Sega Genesis,
and we were able to train the curiosity-driven neural network on both Sonic The Hedgehog for the Sega Genesis as well as Super Mario Bros. for the Nintendo Entertainment System.

**NEAT (NeuroEvolution of Augmenting Topologies)**

NEAT is a feed-forward network describing individual networks using genomes. The algorithm evolves the network through generations using a fitness function. During training, the subsequent generations are produced through reproduction between high-scoring genomes along with some mutations. A fast implementation of adding nodes and connections allows for genome networks to adapt and grow in structure which provides an advantage over fixed size networks.

We used a Python implementation of NEAT to train on Mortal Kombat 3 and Sonic The Hedgehog. NEAT is initialized with a configuration file which defines constants such as number of inputs/outputs, probabilities for mutation, and parameters for weights. The input to the feed-forward network is the image of the game screen, and the outputs are the button presses. A reward function accumulates to determine the fitness of each genome.

We successfully trained a NEAT network to beat the first level of Sonic The Hedgehog. We also trained it on Mortal Kombat 3, but were unable to recover the video file due to technical difficulties.
Lucas Thompson’s video on NEAT helped us write our code and also greatly aided our ability to train effectively.

**Baselines**

The OpenAI Baselines library implements a diverse range of reinforcement learning algorithms, and we chose to use the PPO2 model based on the success we had seen in related projects.

While working with Baselines, we found that it was important to come up with a reasonable reward function as early as possible. We ended up having to revise our reward function multiple times during the project which made it hard to compare the performance of models when we wanted to test new approaches because the scaling of the rewards would be inconsistent. We found that the best reward function was a near equal balance of rewards and penalties and that the rewards should be given immediately, rather than at the end of an episode. The current reward function monitors the health and match wins of both fighters, and when a change is detected, a reward or penalty is immediately given.

After getting the reward function working, we wanted to focus on preprocessing the inputs, which we discussed earlier. The goal of a lot of the preprocessing was to reduce the size of the model in GPU memory. We trained this model on a video-card with 8 GB of on-board RAM, and it was surprisingly easy to overrun the available memory. We needed to train the models in parallel to get training times to be reasonable, usually with 16 environments, so any changes to the size of the model were magnified 16-fold when it came to GPU memory. Scaling and gray-scaling the images cut the input size substantially, but those gains are mostly lost when you increase the image size again through frame stacking. The performance increase of frame stacking more than justified the space, so we began skipping frames to bring the memory footprint back down. The other factors that impact the model size are the \textit{nsteps} variable, which controls the horizon that the model optimizes over, and the \textit{nminibatches} variable which we believe takes the entire horizon and breaks it into smaller chunks that are used for gradient descent to update the policy. Increasing \textit{nsteps} increases the size of the model in
memory, but increasing `n minibatches` decreases the model in memory; we believe this is because the frames are no longer stored in memory once the minibatch has been processed.

Once the model was able to fit in memory, we began to optimize hyper-parameters. We found that a learning rate of 0.00005 worked well. We tried different values up to 0.00025 which caused those models to initially reduce their error quickly, but they slowed down considerably after around 3 million time steps and were overtaken by the models with smaller learning rates after around 10 million time steps.

Of the optimizations we attempted, the most surprising was the performance increase due to the edge detection. To our eyes, the edge detector appeared to have removed too much detail, but the model trained with the edge detector beat all of our previous attempts.

We successfully trained our Baselines network to beat the first four levels in Mortal Kombat 3.

Full gameplay for Baselines on Mortal Kombat 3. The model is the Kabal character on the left.
To see the code for our PPO2 model, check out the GitLab. This code was based on a project by AurelianTactics where he used PPO to play Contra 3.

Curiosity

Rather than learning by using a reward function that must be constructed by hand and then tested, which can be a tedious and time-consuming process, curiosity-driven neural networks learn by being able to correctly predict what will happen as a consequence of its actions given its current state. These models are intrinsically motivated; they always try to minimize their own prediction error. This in contrast to extrinsically motivated models which try to maximize some user-defined reward. For example, consider the case of a curious agent playing Super Mario Bros. The agent will naturally want to explore the entire level in order to minimize its error in predicting what will happen by moving and jumping around, and exploring levels is the same as completing them in Mario.

We had to train the curiosity-driven neural network on the games to make sure the agent knew what the correct inputs were. We also had to add support for certain consoles’ display resolutions. There are various options in the code for how features can be learnt: pixels, random features, variational autoencoders, inverse dynamic features. An interesting result in the research paper was that using just the pixels as features often yielded the worst performance. In fact, random features turned out to be surprisingly effective in comparison and is the default feature-learning method. Due to time and computation restraints, we chose to go with this default method for each of the games we trained on. It would be interesting, in the future, to use the other feature-learning methods and compare the performance of the resulting agents.

The curiosity-driven neural network performed best with Super Mario Bros., making it to the third level. On the other hand, an agent playing Sonic the Hedgehog would always get stuck on the first level. We believe this is due to the very dynamic backgrounds in the game (e.g., animated waterfalls, shimmering water, rotating flowers, scrolling clouds). In comparison, the backgrounds in Super Mario Bros. are largely static. However, this result is not too surprising. The research paper briefly discusses this particular limitation. Mortal Kombat 3, despite being trained for the least amount of time, also has relatively static backgrounds, and the agents playing this game could often get to the second level.
A curious agent playing Sonic the Hedgehog becomes easily distracted by the busy background. Here, the agent most likely thinks the animation of the waterfall is a result of its own actions. It then gets stuck trying to minimize its error in predicting how the waterfall will change as a result of running and jumping around.

We successfully trained a curiosity-driven network to reach the second level of Mortal Kombat 3 and the third level of Super Mario Bros., but we were unable to beat any levels in Sonic The Hedgehog.

Full gameplay for Curiosity on Mortal Kombat 3. The model is the Shang Tsung character on the left.
Our code for the curiosity-driven neural network was mostly from the original implementation found on the creator’s GitHub. Video tutorials by Lucas Thompson were instrumental in our understanding of the code base.

Conclusion
Ultimately, the combination of preprocessing, edge detection, and Baselines resulted in the best results on Mortal Kombat 3, allowing us to beat the first four levels. NEAT was the easiest to use; after we got it set up, all we really needed to do was tune hyper-parameters to improve performance. Baselines was much more difficult because designing an effective reward function was much more challenging than we expected. Finally, Curiosity was difficult to use due to lack of documentation, but it was simple to train after that since no extrinsic rewards were necessary. Curiosity’s performance in terms of efficiency was negatively impacted by dynamic backgrounds, but it was still able to make progress through the level.

<table>
<thead>
<tr>
<th></th>
<th>Mortal Kombat 3</th>
<th>Sonic The Hedgehog</th>
<th>Super Mario Bros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neat</td>
<td>Level 2</td>
<td>Level 1</td>
<td>N/A</td>
</tr>
<tr>
<td>Baselines</td>
<td>Level 4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Curiosity</td>
<td>Level 2</td>
<td>Level 1</td>
<td>Level 3</td>
</tr>
</tbody>
</table>

Results for furthest level reached for each of the models on each game

We touched on some of the challenges we ran into in some of the earlier sections, but throughout the project, training time was our biggest challenge. It was a bottleneck for this project, even with our preprocessing efforts.

For this project, we were able to use Mortal Kombat 3 as our control game, but we’d like to expand this in the future by including a greater number of games as well as a greater diversity of game types. We’d also like to see how general these models can be; in other words, would it be possible to have a single model that can excel at many different games?

Though not quite as fun as playing the video games ourselves, we had a great time learning, failing, and succeeding throughout this project.