CS 499 Artifact 2: Maze Game Development Narrative

By Alex Frankel

This artifact is a Q-Learning module trained to complete a simple maze game. It was created in October 2024. I selected this artifact for the category of Data Structures and Algorithms because it is a demonstration of a Machine Learning algorithm in my course work, particularly the Q algorithm and the configuration of the deep learning neural network. This artifact also defines a custom episodic memory data structure that records game data. I have improved this artifact to include different training parameters side by side to determine the most optimal configuration.

The following course outcomes are covered by this project:

“Design and evaluate computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution, while managing the trade-offs involved in design choices”

**This is demonstrated through my Q-learning agent**

“Employ strategies for building collaborative environments that enable diverse audiences to support organizational decision making in the field of computer science”

**This will be demonstrated through my parameter comparisons later in this document.**

**This is a planned and ongoing optimization process that contributes to the larger field of Computer Science.**

My biggest challenge to update this artifact was actually getting it running properly. That is, the school-provided Virtual Machine that it originally ran on was no longer available to me. The issue is that the course is **extremely outdated**. The deep network parameters were deprecated in the newer versions of Keras and Tensorflow. I fiddled around with reconfiguring the layers but to no avail as the algorithm refused to train or update epochs correctly. I scoured the net to find the exact versions of these libraries were used in the project and I found a fellow classmate that had the same issue but was able to rectify it. By running my own VM with the **extremely outdated** Python 3.7, I was able to access the exact outdated versions of Keras and Tensorflow that were required for this deep neural configuration to run at all. These versions are listed in the Jupyter Notebook for the project.

A benefit of running this on my machine rather than the provided VM was that the training that used to take 48 minutes or more now runs in 5 minutes.

At least my first run did.

My results have been inconsistent each time. Sometimes the model itself won’t evaluate! I was trying to figure out why it wouldn’t ever complete training. The end condition is a 100% winrate for the past 32 or so games. This is a very strict end condition and based on our random exploration variable of epsilon could change the results wildly. The epsilon value starts at 0.1 and decays to 0.05 after the win rate reaches 90%. This means there’s a chance of not winning despite how complete the model is, therefore making the win condition unreachable. I have seen the model run for 5 minutes, 9 minutes, or indefinitely. I could potentially change the end condition to make better progress. This will be necessary to evaluate other changes to the model.

This could also be due to a number of other factors.

As it turns out, the method call to model.fit had certain parameters that were making the training process very slow or never complete…

#updates model

history = model.fit(inputs, targets, epochs=10, batch\_size=24, verbose = 0)

the epoch and batch sizes could greatly affect the training time. I find that these can only really be determined experimentally but different sources online have told me differently. The following is what I ended up using

history = model.fit(inputs, targets, epochs=4, batch\_size=16, verbose = 0)

From what I understand, the epoch number determines the number of complete passes through the dataset. The higher it is, the slower it will be.

**My goal in this enhancement** is to demonstrate the differences in training based on different parameters. I was originally going to juxtapose them within the module itself but it has proven too complicated to run the model multiple times within a single module. It is a much better idea to run them in parallel and make a note of my findings, documenting and releasing any that improve the model’s performance. I could submit these as multiple notebook files. This way, the format of the files and the results contained within them are preserved.

Example:

A screenshot of a computer

AI-generated content may be incorrect.

One successful round of training. This is the path the agent chose:

A black and white squares

AI-generated content may be incorrect.

A second run took 20 more seconds and 53 more epochs

A screenshot of a computer

AI-generated content may be incorrect.

And the path IT chose

A grey and black crossword puzzle

AI-generated content may be incorrect.

Here is the epoch that the original model crashed after

A screenshot of a computer

AI-generated content may be incorrect.

Here is the path the file from the original model chose:

A black and white crossword puzzle

AI-generated content may be incorrect.

Notice here that none of these paths are entirely optimal and potentially differ each run. The agent isn’t tasked with perfection but rather using its best guess. In fact, despite the drastically increased performance it appears the updated model takes an even longer route! Since updating the epoch and batch sizes I can reliably run the model to completion in around 10 minutes and I am ready to run more tests on the parameters.

With a epsilon that decays to .0125, it only takes 8.48 minutes

A screenshot of a computer

AI-generated content may be incorrect.

Here is the path it took

A black and white squares

AI-generated content may be incorrect.

Finally, with epsilon decay entirely removed:

A screenshot of a computer

AI-generated content may be incorrect.

A black and white squares

AI-generated content may be incorrect.

It took slightly longer but took the shortest path available! It is exploring slightly more and thus learning ‘better’. After running this version multiple times, we get the exact same results. I can safely say that out of all three implementations, the one without any epsilon decay was the most effective.