Galaxian Q-learning vs. Space Invaders DQN

Reilly Farrell, Kateryna Dzhyndzhyrysta, and George Wang

Introduction

This paper presents the comparison between utilizing Q-learning for Galaxian and a Deep-Q neural network for Space Invaders, both for the Atari 2600. The initial goal of the project was to use the Arcade Learning Environment to implement a Deep-Q neural network; however, the lack of support a custom environment has in AI Gym led to multiple complications in our initial plan. To circumvent this, our approach branched off into two separate entities: Galaxian agent driven by Q-learning and a Space Invaders agent driven by a neural network. The purpose of performing two separate experiments was to still go with Galaxian, our initial game, but to additionally experiment with Deep-Q neural networks, our intended approach.

Galaxian

For Galaxian, processing the raw, unoptimized screen without AI Gym was too resource intensive, so we instead went the route of Q-learning driven on score. This allows the agent to find a way to move that nets it the most points for the current state it is in. This movement begins to change based on how effective its current best option is, as there is only one continuously updating state rather than hundreds of extremely specific states. Additionally, the agent will not move the ship if there is a bullet to the left or right of it, to prevent collision. This paired with a specific area around the left edge and right edge of the screen, that results in a negative reward, allows the agent to dodge most enemy attacks. The negative edge reward is so the ship doesn't get trapped in the corners, the most vulnerable area for it to be. Overall this approach performs moderately well right out the gate but most likely has a bottleneck in performance that a deep q network can surpass given enough training. The average run is around 1,200 points and the best is around 3,000.

Space Invaders

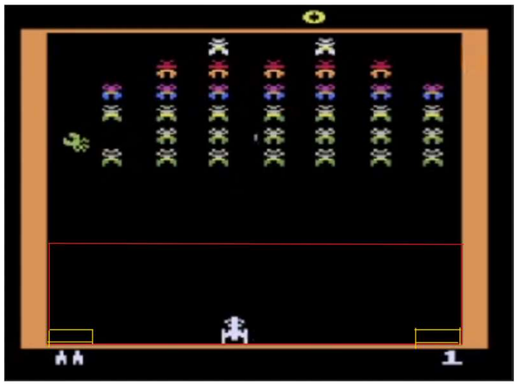
In our Deep Q network implementation of space invaders, we primarily utilized the Open Ai Gym toolkit as our environment and Keras Tensorflow to create our deep learning model and build and train our agent. The observation space we used was a RAM state space of 128 bytes so we did not have to use any convolutional layers but rather we opted to use dense layers with varying numbers of neurons in each different model we tested. Each model was tested with 2 millions steps whereas we tweaked parameters such as the number of neurons in each dense layer, the number of dense layers, epsilon greedy probability, and learning rate.The biggest changes in score per episode were a result of the number of neurons we used in each dense layer and our best model (3 dense layers; 512, 128 into 6 neurons) achieved a near double in score per episode compared to a model that performed random actions with only 2 million steps of training.

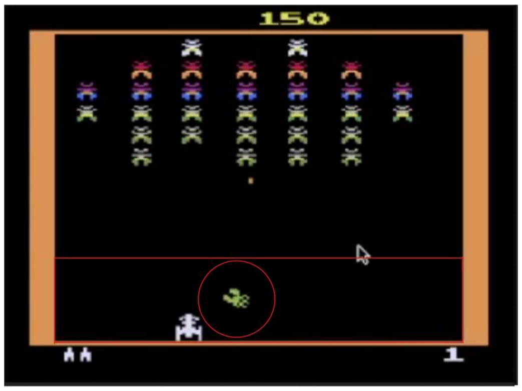
Galaxian Approach/Methods

For Galaxian the agent is driven by reinforcement learning utilizing two separate Q tables, each updated at the same time after an action is performed. This approach does not remember each different number of points as a unique state, but rather utilizes one continuously updating state with one set of Q tables. The reasoning behind this design choice is that there are far too many score states to remember and just because an agent has a certain number of points, doesn't mean it is always in the same x points state; there are many different factors that would cause 'similar states' to vary greatly. Such differences include ship placement, enemy ship placement, and enemy bullet placement. The first Q-table is for deciding which action the agent will take. While there is a moderately sized number of actions for the agent to perform on the Atari controller, in practice the only actions that really matter are moving left and shooting or moving right and shooting. These two actions belong to the action Q-table. Their respective Q-values are both initialized to 0. The second Q table is for distance, and it corresponds to an action. This determines how long (essentially how far) the agent will perform its chosen action. This table consists of the values [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50] and are all initialized at 0. Both the left shoot action and right shoot action have their own distance Q-table. The reasoning for having two separate distance Q tables, one for each action, is to allow the agent to perform well in states that have most enemy ships on one side, rather than being evenly distributed. For the specifics of this Q-learning implementation, we utilize epsilon greedy with an exploration rate of 20%. The learning rates for which direction to move is 0.35 while the learning rate for how far to move is 0.15. The reward for not increasing the score after moving is -1000 while the reward for not scoring while moving a certain distance is -300. Rewards for scoring are calculated the same for direction and distance as the number of points gained consequence of the action distance pair. The discount factor for both movement and distance are both 0.8. The key components of the Q-function are ensuring that the agent is greatly punished for not scoring, because we want it to adapt quickly, and fast learning if it is moving in the wrong direction, again for quick realignment/adaptation to ensure better scoring.

Danger Vector

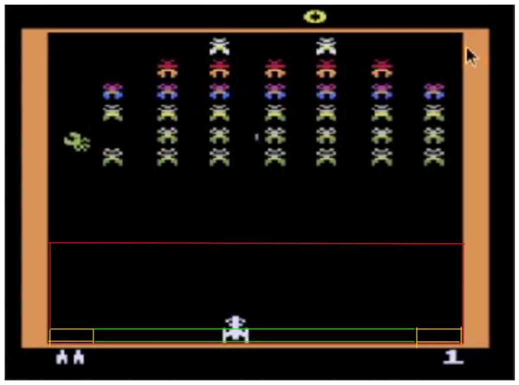
Successful implementation and trials led to the observation that the Q-learning approach, while able to score moderately well, cannot dodge enemy ships or attacks and thus fails quickly. To circumvent this the idea of a danger vector was introduced. Before every agent's call to act, the method ale.getScreen() can be used to get the grayscale representation of the screen as a pixel vector. As previously mentioned, this entire vector cannot be used repeatedly to drive learning due to resource constraints; however, a partially cropped section can influence decision making without drastically harming performance. This cropped pixel vector was deemed the danger vector.
The danger vector, as depicted to the right outlined in red, is the cropped area around the ship. Summing the grayscale pixel values of this vector will always result in the same number if only the ship is inside it, thus allowing us to detect if there is an enemy projectile or enemy ship in this 'danger zone'. With this information we can have the agent look in the danger zone before acting and decide to not move if there is a possibility of colliding with something. The justification for not moving is due to most of the agent's observed deaths being a consequence of lateral collision. This consideration of the danger vector improved the agent's overall performance roughly 20%. (See data section). With the agent suffering fewer lateral collisions, we noticed the new most frequent cause of death was being collided into while in the left or right corner.



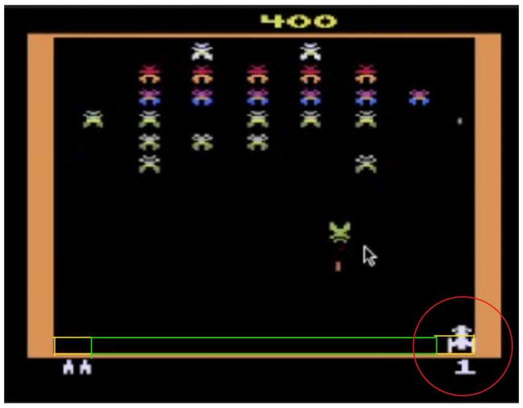


Edge Reward

To attempt resolving the edge collision problem, we first needed a way to track where the ship is. Attempting to track the ship by movement is far too inaccurate and extremely unreliable, thus a similar approach as the danger vector was considered. Summing the two cropped vectors that near the edge, near the bottom section of the



ship, as depicted in yellow, will always be divisible by 14, the pixel value of the ship's color, given the ship as at one of the edges. Using this we know if the ship is at one of the edges and can give a negative reward to movement direction, -10,000, appropriately. Using such a high negative reward ensures the ship will not dwindle in our predefined edge area and quickly move away. This addition along with the danger vector resulted in a 44% increase from only utilizing the danger vector, and a 72% increase from the original.



Galaxian Data and Experiments

Q-Table with Points

Average score of 100 episodes: 736.6

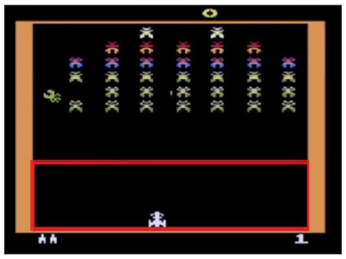
Max score: 2040

Q-Table with Points and Danger Vector

Average score of 100 episodes: 880.2

Max score: 2150

20% Increase from previous/initial version



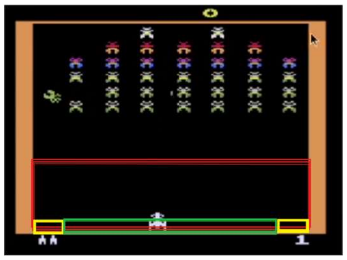
Q-Table with Points, Danger Vector, and Edge Rewards

Average score of 100 episodes: 1270.3

Max score: 3460

44% Increase from previous version

72% Increase from initial version



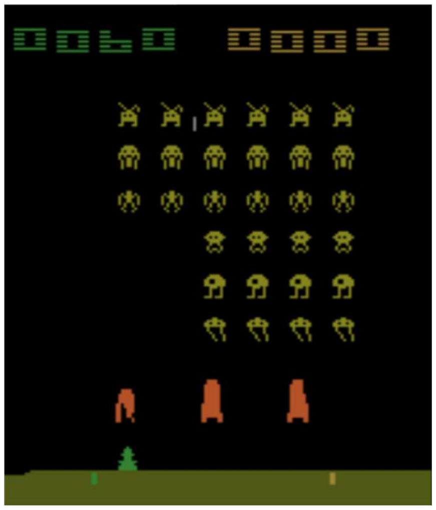
Overview/Future Adjustments

Overall, this implementation of Q-learning performs adequately without need for prior training. The major downside to this approach is that it heavily relies on a specific environment and has a clear ceiling in performance. Without the agent being able to understand or learn how it should react to it being in close proximity of danger (enemy ships or bullets), it can never be extremely great at staying alive. A future approach could entail breaking the screen up into quadrants, to track the different ship/bullet states, and then attempting implementation of a deep-q neural network using these states. Additionally, better understanding of the ram representation of the game could yield similar promising results.

Space Invaders Approach/Methods

Deep Q-Networks

Our initial approach for Space Invaders in the Open Ai Gym environment was to utilize keras tensorflow to build a Deep Q-Network. Through keras tensorflow, we are able to create sequential models to group linear stacks of different types of layers where we are also able to specify different features of each layer. These sequential models thus represent our main method for creating deep learning models with keras. Keras-rl then allows us to define a policy (explained below), provides an agent class (rl.agents.dqn.DQNAgent) and allows us to train our agent (explained below). Hyperparamters, Infrastructure of Neural Networks and Training are discussed later in this section.



When building the DQNAgent; model, memory, and policy should be specified in the parameters. The parameter enable_dueling_network is set to false (see next subsection for DDQN) and nb_steps_warmup set to specify how long we wait before we start doing experience replay. In our DQN model, nb_steps_warmup was set to 1000.

Dueling Deep Q-Networks

Without going into too much detail of how DDQN works, the main motivation of this approach was the ease of implementation and to test for improvements. A good quote from freecodecamp explains that “intuitively [our] DDQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state”. We hypothesized that space invaders followed a similar logic that could allow DDQN to thrive. For instance, moving right or left only matters if there is a risk of getting hit by a bullet. And, in most states, moving left or right has no effect on what happens. DQNAgent from the rl.agents library allows easy implementation of Dueling Deep Q-Networks with an additional “dueling_type” parameter.

Exploration Policy

An exploration policy helps the model learn from a wider range of states and actions. Through utilizing the LinearAnnealedPolicy() with the EpsGreedyQPolicy() function from the rl.policy library, the value of epsilon changes linearly throughout training the model. While testing, we use epsilon = 0.05 as our standard so the agent can still perform random actions and this also ensures that the agent does not get stuck.

Hyperparamters

The hyperparameters were set standard to epsilon greedy = 0.05, and learning rate = 0.0001 for most of the models tested. For one model trained, we did change the hyperparameters of the learning rate to 0.0003 to test its influences on the score per episode.

Infrastructure of Neural Networks

Keras Tensorflow was utilized to create the neural networks. Because we are using a RAM state space, convolutional layers are not needed but rather we only need to flatten the RAM vector and experiment with different dense layers. Three main different neural network infrastructures were tested. 3-layer fully connected network [512, 128, 6], [128, 32, 6] and 2-layer fully connected network [128, 6].

Training

For training, we used the optimizer that implements the Adam algorithm (variation of stochastic gradient descent method). Each model was trained for 2M steps with a learning rate affected by the LinearAnnealedPolicy() with the EpsGreedyQPolicy() function from the rl.policy library with the epsilon annealed linearly range being between 0.1 and 1.

Space Invaders Data and Experiments

Models were trained each with 2 millions steps and were tested for 50 episodes to calculate a consistent average score per episode.

Random actions agent: Agent that performs random actions from the available actions state space.
Average score per episode: 123.5
Max score: 260

DQ 3-layer fully connected network [512, 128, 6]: Deep Q-network with 3 dense hidden layers of 512, 128, and 6 neurons.
Average score per episode: 200.25
Max score: 560

DDQ 2-layer fully connected network [128, 6]: Dueling Deep Q-network with 2 dense hidden layers of 128 and 6 neurons.

Average score per episode: 220.6
Max score: 575

DDQ 3-layer fully connected network [128, 32, 6]: Dueling Deep Q-network with 3 dense hidden layers of 128, 32, and 6 neurons.
Average score per episode: 146.5
Max score: 470

DDQ 3-layer fully connected network [512, 128, 6]: Dueling Deep Q-network with 3 dense hidden layers of 512, 128, and 6 neurons.
Average score per episode: 238.75
Max score: 530

DDQ 3-layer fully connected network [512, 128, 6] (Learning Rate=3e-4):
Average score per episode: 190.5
Max score: 535

Overview/Future Adjustments

Through comparison of the average score per episode for each of the different models present in the above Space Invaders Data and Experiments section, it is quite prevalent that different parameters/attributes have a wide range of impacts on the Agent's performance. Though more extensive training/experiments are required to determine a more specific correlation between specific attributes of the model and average score per episode increase. Our experiments have shown the use of an agent trained on a neural network will always score on average higher than an agent that acts randomly. Neural networks trained with more neurons per dense layer ([512, 128, 6] vs. [128, 32, 6]) will also have a higher chance of scoring higher. One thing to note is that, we speculate that the avg score per episode vs. neurons used graph will reflect that of a polynomial where too many neurons could result in a decrease in performance. As we had predicted earlier, a Dueling Deep Q-network performed better than a regular Deep Q-network but the performance gain was not very significant (increase of 38.5 avg score per episode). Interestingly, using only 2 dense layers with [128, 6] neurons performed better than a 3 dense layer model with [128, 32, 6] neurons and only slightly worse than a 3 dense layer model with [512, 128, 6] neurons. Another interesting comparison is that slightly increasing the learning rate from 1e-4 to 3e-4 actually decreased the average score per episode by 48.25. Whilst training the agents, it could be observed that RAM space states having fewer parameters, trained very quickly and loss and mean_q would on average see a significant decrease in improvement at around 120k steps into training. No concrete results are available (problems with google colab) but while observing the training in action, it was interesting to see a logarithmic trend with loss and mean_q.

Our best model (DDQ 3-layer fully connected network [512, 128, 6]) performed quite well with a near double in average score per episode compared to an agent that performed random actions. There are quite a few future adjustments that are worth mentioning/exploring. Without going into too much detail, a few ideas for adjustments that could improve our models: more training steps, more testing of different parameters, frame stacking and frame composition implementation, preprocessing of ram bytes that do not prove useful, Dueling Double DQN implementation, 1D convolutional layer over RAM state space, LSTM layer for keeping state over time, and adding dropout.

Retrospective

Most of the duration of the project was spent researching, as we were relatively new to working on AI for Atari games. Our first discovery was that you can use either pixels or RAM state as observation space. We have figured out how to use both: pixels for Galaxian and RAM for Space Invaders. One of our problems that we had was that we could not track the ship's movement. We thought that by tracking the ship's position we could drastically improve AI and make it better at dodging bullets. A research paper related to the topic, “The Arcade Learning Environment: An Evaluation Platform for General Agents”, described the DISCO method, which allowed detecting objects within the Atari 2600 screen by using an algorithm to encode positions of objects. Using that method would have elevated our project and our approach to training would have gone in a different way. However, implementing this method was too complicated and would have required too much time.

Through Galaxian we learned of the necessity to optimize images prior to using them in learning. In a game like Galaxian, there are a countless number of states for the agent to interpret and without proper trimming of useless information along with optimization of useful information, such a model will not be possible to create using above average computing power. To circumvent this, we learned techniques in utilizing grayscale image vectors, cropping image vectors, and carefully assessing the minimal amount of information an agent can get away with to act appropriately. The agent doesn’t really need to know every possible state it has been in/could be in, but rather what to do if it’s ‘in danger’, or at an edge. While a robust neural network that learns the entire environment itself is impressive, in practice it always isn’t computationally feasible. Instead, letting the agent learn with a few training wheels such as “don’t move if you’re in danger” or “stay away from the edges” can be enough to create something that works reasonably well. The downside of this approach is the clear bottleneck in performance. Just because you can raise a monkey and child together, exactly the same, doesn’t mean the monkey will grow up like the child to become a human adult. But rather, at some point the monkey will hit a threshold where it simply can’t become any more human, because it’s a monkey. A monkey has mental limitations in learning and in this case, this Galaxian agent is the monkey. This metaphor highlights the appeal to creating a more robust and complex deep-q neural network, as we did for Space Invaders.

Since Galaxian was using pixels as observation space, we have decided to use RAM for Space Invaders, given that we have access to OpenAI Gym library. The main positive about utilizing RAM state space was the reduced input parameters and how preprocessing was not required. Researching on how to use RAM brought us to Keras Tensorflow, and we then found a way to make that work with RAM. For some time we were confused about dense layers and what parameters to tweak for our experiments, however going through multiple papers, we managed to figure out optimal parameters to experiment with in this project. We also read through the OpenAI gym library documentation and used Tensorflow to develop DQN. While developing DQN, we discovered Dueling Deep Q-Networks. We found the concept to be very interesting and did some research so that we could implement that for Space Invaders. There were many more concepts related to neural networks (mentioned above in Space Invaders Overview/Future Adjustments) that we found during our research but were a bit outside the scopes of the CS4100 course and proved difficult to implement but are very much worth exploring in any future experimentations.

During the installation process, we learned how to set up the Atari emulator and Arcade Learning Environment on PyCharm. For Space Invaders, we made the simulation work on Jupyter Notebook with visuals and Google Colab without visuals.

Appendix

Space Invaders (Google Colab)

To make Space Invaders DQN work on Google Colab, download an ipynb file named “Space_Invaders_Colab” from Canvas. First, go here to download roms for Atari 2600 games. Then unzip “Roms” folder, open that folder and unzip “ROMS” folder. The next step is to upload the “ROMS” folder to your Google Drive - this will allow you to use those files in Google Colab. Finally, open Space_Invaders_Colab.ipynb file in Google Colab and run the first three lines of code. Those lines of code are for installation needed dependencies: first is for installing tensorflow, second is to make it so that Google Colab can access to your Google Drive and third one is to import roms. After that you should be ready to run the rest of the code. All visualization parameters are set to False, as we could not figure out how to make visualization work on Google Colab.

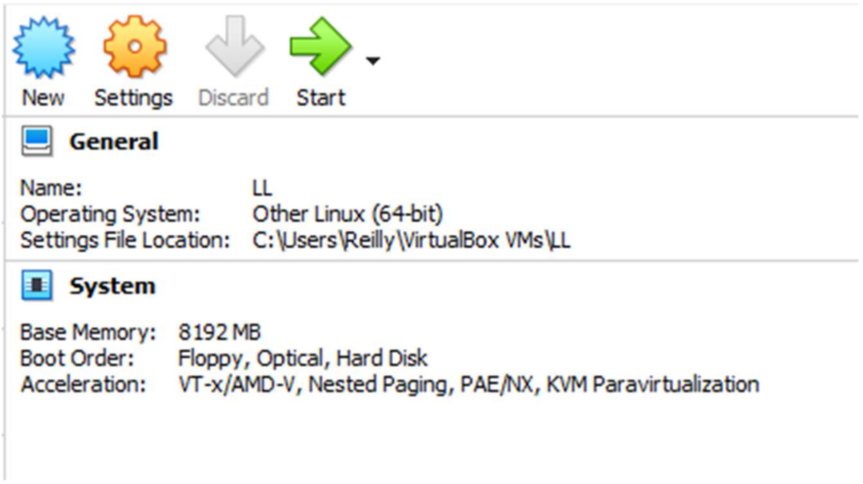
Space Invaders (Jupyter Notebook Python 3.8)

Download the Space_Invaders_Jupyter ipynb file from Canvas. Go here to download roms for Atari 2600 games. Then unzip "Roms" folder, open that folder and unzip "ROMS" folder. Follow the following resource if encountering difficulties with the ROM. https://github.com/openai/atari-py#roms The dependencies section in the Space_Invaders_Jupyter contains commands for all the pip installs needed for the project. Afterwards, the jupyter notebook should be set up to run through. Please note that if encountering errors in building the DQNAgent, deleting the model with the "del model" code box and rebuilding the model should resolve the issue.

Galaxian (Linux)

Download Linux Lite 5.4 here. Extract Ensure your system has VirtualBox, which can be downloaded here. Extract Linux Lite 5.4 (64bit).vdi to your desktop. Launch VirtualBox.

Towards the top middle click new.



General

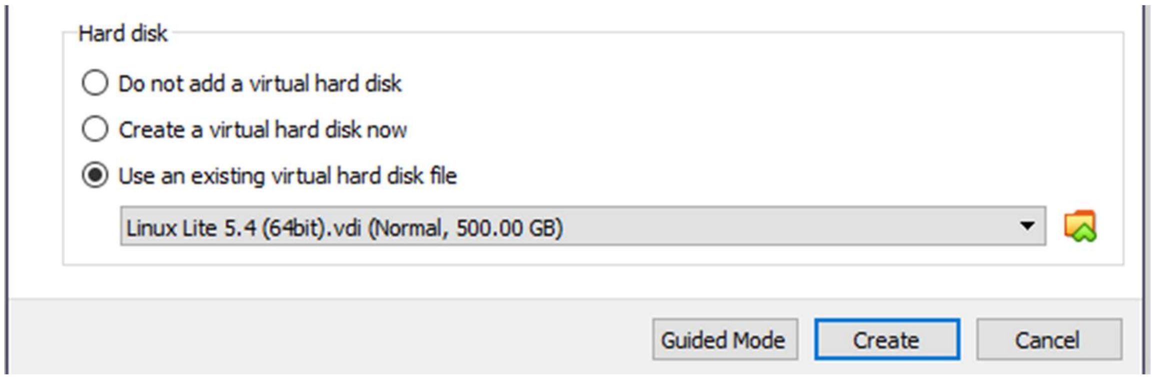
Name: LL
Operating System: Other Linux (64-bit)
Settings File Location: C:\Users\Reilly\VirtualBox VMs\LL

System

Base Memory: 8192 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

At the bottom of the newly opened window, click Expert Mode.

Select "Use an existing virtual hard disk file", click the folder, select add and navigate to the linux Lite VDI. Then press "Create"


Hard disk
Do not add a virtual hard disk
Create a virtual hard disk now
Guided Mode
Create
Cancel

Use an existing virtual hard disk file [✓]

Linux Lite 5.4 (64bit).vdi (Normal, 500.00 GB)

Double click on the newly created Virtual Machine to launch it.

Login with the password “osboxes.org”

Right click, select terminal, run the following commands:
(The password is: osboxes.org)

sudo apt install git

sudo apt install zlib1g

sudo apt-get install python3.8

sudo apt install python3-pip

sudo apt-get install libsdl1.2-dev

sudo apt-get install libsdl-image1.2-dev

sudo apt-get install libsdl-mixer1.2-dev

sudo apt-get install libsdl-ttf2.0-dev

sudo apt-get install libsdl-gfx1.2-dev

sudo apt-get update -y

sudo apt-get install -y stella

sudo apt-get update
sudo apt-get upgrade

sudo apt-get install g++

sudo apt-get -y install cmake

pip3 install numpy

pip3 install pynput

Download this Folder from here on your linux virtual machine: https://bit.ly/3A1vv32
(If the link isn't working try
https://drive.google.com/drive/folders/1se1nNYu9RyhlXNrs1h4EKNbF8kDFsdIT?usp=sharing)

(If neither work see the bottom)

Do the following commands:

cd Desktop

git clone https://github.com/mgbellemare/Arcade-Learning-Environment.git

Drag all three files download (Gal.a26, QLearner_Galaxian, setup.py) into the newly created folder from the above command (It will ask if you want to replace, select "yes")

Do the following commands:

cd ArcadeLearningEnvironment

pip3 install .
(There is a '.' after install in the above command)

######################
To run the program, python3 QLearner_Galaxian Gal.a26 (Number of episodes)
For example: python3 QLearner_Galaxian Gal.a26 10
Will run 10 episodes

Troubleshooting

Download and do the following if the folder link does not work:

GALAXIAN [USA] - Atari 2600 (800) rom download | WoWroms.com

QLearner_Galaxian is from the canvas submission

Edit setup.py and paste over the code so lines 48-55 look like this:

```
cmake_args = [
 f"-DCMAKE_BUILD_TYPE={config}",
 f"-DPython3_EXECUTABLE={sys.executable}",
 f"-DCMAKE_LIBRARY_OUTPUT_DIRECTORY={extdir}",
 "-DBUILD_CPP_LIB=OFF",
 "-DBUILD_PYTHON_LIB=ON"
]
```