CS 5150: Game Artificial Intelligence
Super Mario Bros. bot

Introduction

Super Mario Bros. is an arcade platform video game developed and published by Nintendo. Players control Mario, or his brother Luigi in the multiplayer mode, as they travel the Mushroom Kingdom to rescue Princess ToadStool from the antagonist, Bowser. We will try to apply deep Q learning, which provides right directions to Mario to complete the level while gaining points by collecting coins and dodging enemies. It also has special power-ups which give Mario superpowers to kill the enemies. We will also try to implement a PyTorch based neural network to achieve the same task.

Keras Model

We have implemented a deep Q learning network which gets the state from the env.step() function and pass it to MobileNetV2, a famous deep learning network which comes with keras.applications to extract features out of the state. We pass these features to a dense neural network with 3 layers to generate Q value for twelve actions which constitute the COMPLEX_MOVEMENT we pass to the environment. We select the action with highest value and simulate that key press.

Rewards are given based on how further right mario goes. The reward given for an action which takes mario further right is the original reward returned by the env.step() function multiplied 20 times. The reward for moving to the left is original reward times 2. The death penalty is -25.

We trained it for various amounts of time and different number of episodes and were getting decent results. We also made sure that the results were not just a result of random actions and decayed the probability (epsilon_decay) with which the controller selects the actions. As time passes on it starts using model generated actions and reduces dependence on random actions. We also used feature extraction using Deep neural network as opposed to passing the whole image to the neural network.

We also tried training mario using neural network written by us as opposed to Experience Replay logic that is generally used in Deep Q Learning, but it did not give us as good results as the aforementioned keras model even after training it for same number of episodes and time, so we continued experimenting with experience replay for the rest of the duration of the project.

Links to various training models are attached in zip file under Videos Links folder.

Topics

Milestone 1 - Environment setup and testing on our local computer.
Milestone 2 - We will make our own Q learning model and basic neural network run for the problem.

Team
Ojas Patwardhan - patwardhan.o@husky.neu.edu
Soumyadeep Sinha- sinha.sou@husky.neu.edu

Instructions

Prerequisites -
Python
Libraries - tflearn, keras, cv2, tensorflow, gym-super-mario-bros, numpy

Open Pycharm, click on create new project, setup virtual environment, copy and paste KerasNN and TflearnNN folders in the project directory.

Steps to run keras -

1. Download the zip file submitted.
2. Extract it in a folder
3. Install the necessary python dependencies
4. Modify the LR rate, Number of episodes and other variables to required value.
5. Run train.py file using python
6. Wait for it to load necessary files. Training is paused initially.
7. Emulator should automatically start and you can watch the training.

Steps to run tflearn (tensorflow) -

1. Download the zip file submitted.
2. Extract it in a folder
3. Install the necessary python dependencies
4. Modify the LR rate, Number of episodes and other variables to required value.
5. Run main.py file using python

Systems

Keras

In our Keras implementation, we passed the state returned by the env.step() function to feature_extraction.py. In this file, we ran MobileNetV2 which is a more efficient version of VGG16 to extract the required features since the original state returned was too big and was considerably slowing down our emulator during training. We also tried feature

extraction using VGG16 but still the system was very slow. For feature extraction, we re-sized the image which is nothing but the state returned by the function. We then performed interpolation on this and then predicted the important features and returned that image. This new image/ state was then passed to act method which then chooses either a random action or predicts the value of the action based on the value of epsilon. Our neural network consists of 3 layers and outputs the the values for 12 actions which Mario can perform. We then select the best action with the maximum value and perform it in the game. After every episode we made sure that we decayed the value of epsilon by a certain amount in order to reduce the chances of selecting a random move as the training progressed. We are also saving the weights of the model after the completion of every episode. All the parameters like current state, next state, action, reward are being stored during training for the purpose of experienced replay. After every episode we are running the replay function on a minibatch of size 32 which basically picks random samples from memory and trains the model and saves the new weights. We experimented using Simple as well as Complex movements for Mario. Simple movements consist of 7 possible actions and Complex movements have 12 actions.

TFLearn

Our first successful attempt to create a neural network to train Maro to complete level was done using TFLearn which is a modular and transparent deep learning library built on top of Tensorflow. It was designed to provide a higher-level API to TensorFlow in order to facilitate and speed-up experimentations, while remaining fully transparent and compatible with it.

Our Neural Network had 6 layers with input Layer taking in the raw rgb pixel data state of the Game. The inner layers with 64, 128,128,64 Neurons at each level respectively and the initial drop out rate as 0.8 per layer. The outer output Layer has 6 Nodes. We trained the said Neural Network using the State received from the initial game plays with a minimum score requirement for certain number of episodes and Learning rate as 1e-3. We saved these states and corresponding one hot encoded actions associated with that state in a list called training data. Once the training data is generated, it is then reshaped and fed to the neural network and output of which is the model which is then fitted. Once the Model is fitted and saved, we load the model and predict the moves using the model in the actual game.

This was our initial attempt to code the neural network which had many flaws such as more number of neurons, higher drop-out rates, non-inclusion of proper rewards and feeding of raw rgb pixel data as state to name a few, which we rectified in our project later on.

Failures and Setbacks

We faced quite a lot of setbacks while implementing the project:

We initially started with developing and implementing our neural network using PyTorch. However, we quickly realized that Tensorflow and its abstraction TfLearn had better learning curve, documentation and tutorials online. Hence, we started developing our neural network using TfLearn. Being new to Tensorflow and not having any previous experience in machine learning, we created a neural network with many neurons and layers which made the process really slow. Having added the drop-out rate for every layer of our network to 0.8 meant that 80% of our nodes were not getting trained at all!. However, we had few successes using this primitive neural network, after training the model for 3000 episodes, in that we were able to dodge the enemies and cross small and medium pipes.

Our major setback here apart from all the setbacks mentioned above was our character “mario” was not able to do “High jumps”. We tried to add rewards for high jumps to our model but to no avail. Hence, we decided to move on and start implementing our entire project in keras which is another abstraction of TensorFlow hoping we might be able to overcome our previous setbacks.

Now, with the better understanding of Neural networks and how nodes and other factor affect the model we started implementing our project keeping in mind our previous setbacks. The Neural Network we ended up using has just 3 layers, less number of nodes and no drop out rates. We were able to achieve the same result now with training mario for just 50 episodes. Now, we were in the same stage as we did before which meant that our “Mario” was still not “high jumping”. So we set out to solve our more critical problem of training the system faster.

Our main objective of making the neural network train faster was to train “Mario” for longer episodes as quickly as possible. Initially, we were feeding the raw rgb pixel data to our neural network which made it really slow to train the network. So we decided to extract the major features from the raw data and perform feature extraction by MobileNetV2 and then feed this reduced state with only major features to our neural network. This made the training process much simpler and faster.

Our Next objective was to tackle the “high jump” problem. To tackle the high jump problem we changed the movements from “SIMPLE MOVEMENTS” to “COMPLEX MOVEMENTS” and increased the rewards for the long jumps. Once Our model learnt how to do high jumps we were successfully able to add rewards and penalty for death and moving right. We also gave a penalty equivalent to death penalty if mario stood in the same place for a long time.

Successes

We tried several experiments on our reward system like increasing the original reward given to Mario for moving to the right. Similarly we also tried punishing mario more when he died or moved to the left. We successfully made the whole pipeline that took the state, extracted features from it, calculated rewards and took actions for 50 and 100 training episodes possible for our model. We have two major models, implemented with this setup. Since sometimes during training, we observed that mario was not able to cross the pipes, we gave him a penalty equivalent to death penalty so that as to prevent him from staying in the same place for a long period of time.

Evaluation

For evaluation, we check the score as well as the x-position of Mario returned by the step function. We have explained the various evaluation experiments we made to our reward system in the experiment section.

Lessons learned

1. Use tools and softwares with better documentation, online help and learning curve.
2. Increasing the number of neuron and layers doesn't necessarily translate to better performance.
3. Be careful of the drop-out rates.
4. Proper rewards are necessary for better performance
5. Optimization such as giving penalty for staying at the same place for a longer time and Feature extraction to reduce the state can help in making the training faster

Experiments

Parameters:

Number of actions - 7 (Simple - NOOP, right, right + a, right + b, right + a + b, a, left)
Number of actions - 12 (Complex - NOOP, right, right + a, right + b, right + a + b, a, left, left + a, left + b, left + a + b, down, up)

Experiment 1:

TFLearn NN Vs Keras NN

Keras and Tflearn are deep learning libraries written in Python. In Keras NN, we implemented feature extraction whereas in Tflearn NN we did not perform feature extraction. Our Keras NN had only 3 layers while the Tflearn implementation had 6 layers. Since we made the NN in Keras smaller than Tflearn, there were 60 nodes as compared to 356 in Tflearn NN, we decided to remove the dropout rate in our Keras NN. The rewards returned by env.step() in Tflearn NN were considered as they were while training the model whereas in Keras NN, we increased/ decreased the rewards based on the state of Mario on top of the rewards returned by env.step(). For example, since it was difficult for Mario initially to do a long jump we increased the reward for mario to select the long jump when possible. Similarly we gave penalties if the character is going to the left of the screen. There were other modifications and optimizations which we implemented in our Keras NN such as feature extraction and time out penalties which we did not do in our Tflearn NN.
We believe that the modifications in the Keras can also be replicated in Tflearn. We did conduct experiments by changing the number of neurons in neural network, dropout rate and learning rate in our TFLearn NN which led to significant progress in the amount of training required to achieve the same result than our previous version of the same neural network. However, we are yet to implement the feature extraction techniques and adding penalties and rewards to the NN which have planned as a future enhancement to the system.

Experiment 2:

Using Simple and Complex movements and variations in penalties and rewards

We experimented using Simple and Complex movements for Mario in Keras NN. While using simple movements the character could not go far because it was not getting a high reward for actions like high jump. However, complex movement offers a much wider range of actions (12) and at one point during training, Mario went down the tunnel and completed the level. The other experiment which we performed was by varying the default death penalty and reward ranges in smb_env.py. We observed that this did not affect the performance much. We then tried changing the reward returned by env.step() and Mario started learning that going to the right meant it was better. We also gave a high reward whenever the action taken was a high jump or a super jump. For every time Mario died, we gave a punishment. We noticed that sometimes in training Mario would keep performing the normal jump action and was not able to cross the pipe. To teach him that this is wrong, we also gave punishment equivalent to death penalty.

Experiment 3:

Different Level

We trained our Mario in level one however, we tried to use the model trained in level 1 in level 2 to see how effective the model was. In Level 2, there is a narrow gap through which mario will have to pass but there are few approaching enemies which makes it extremely difficult because the only way mario can cross this is by moving to left but we have assigned high negative penalty when moved to left. This make the character move right and collide with the enemy. Hence, we conclude that we might have to train longer and tweak the rewards and penalties such that it can cross level 2.

Experiment 4:

Varying batch sizes

In this experiment, we tried varying the batch sizes used for training the model. After every episode, we were training the model by taking random samples from the memory. We found that a batch size of 32 gave the most optimal results. When we overfitted the model using batch sizes of 64 and 40 respectively, Mario was not able to cross the third pipe and kept going to the left. A batch size of 30 produced somewhat better results after about 5 episodes of training.

Results and Videos

https://youtu.be/BdpB-nFpD-Q
https://youtu.be/SnpCrMlz3BM
https://youtu.be/mf-KlkYyelU
https://youtu.be/BbJEDAxWD6E
https://youtu.be/WlQBs6GdRqQ
https://youtu.be/Y2FoFzfN9to
https://youtu.be/7ieGVpJadW4
https://youtu.be/OuzdLPOfk1A
https://youtu.be/muruQMOJlJk
https://youtu.be/bkVrrkNFS9g
https://youtu.be/eAp4cUSfgfc