AI PROJECT

Abstract

Artificial Intelligence faces more problems than it can currently solve and one such problem is learning to handle an environment with no data sets.We are taking 2013 publication by DeepMind titled ‘Playing Atari with Deep Reinforcement Learning’ introduced a new deep learning model on similar lines for reinforcement learning, and demonstrated its ability to master difficult control policies for Atari 2600 computer games, using only raw pixels
as input. My project was inspired by a few implementations of this paper.

Reinforcement Learning(RL)

Reinforcement Learning(RL) is how our brain still works. A reward system is a basis for any RL algorithm. If we take the example of a child's walk, a positive reward would be a clap from parents or ability to reach a candy and a negative reward would be say no candy. The child then first learns to stand up before starting to walk. In terms of Artificial Intelligence, the main aim for an agent, in our case the Dino , is to maximize a certain numeric reward by performing a particular sequence of actions in the environment. The biggest challenge in RL is the absence of supervision (labeled data) to guide the agent. It must explore and learn on its own. The agent starts by randomly performing actions and observing the rewards each action brings and learns to predict the best possible action when faced with a similar state of the environment.

We use Q-learning, a technique of RL, where we try to approximate a special function which drives the action-selection policy for any sequence of environment states
Q-learning is a model-less implementation of Reinforcement Learning where a table of Q values is maintained against each state, action taken and the resulting reward. A sample Q-table

should give us the idea how the data is structured. In our case, the states are game screenshots and action jumps and do nothing.

Introduction of ε

Absence of labeled data makes the training using RL very unstable. To create our own data,we let the model play a game randomly for a few thousand steps and we record each state, action and reward. We train our model on batches randomly chosen from these experience replays. Exploration vs Exploitation problem arises when our model tends to stick to same actions while learning, in our case the model might learn that jumping gives better reward rather than doing nothing and in turn apply an always jump policy. However, we would like our model to try out random actions while learning which can give better reward. We introduce ε, which decides the randomness of actions. We gradually decay its value to reduce the randomness as we progress and then exploit rewarding actions.

Introduction of γ

Credit Assignment problems can confuse the model to judge which past action was responsible for the current reward. Dino cannot jump again while mid-air and might crash into a cactus, however, our model might have predicted a jump. So the negative reward was in fact a result of previously taken wrong jump and not the current action. We introduce Discount Factor γ, which decides how far into the future our model looks while taking an action. Thus, γ solves the credit assignment problem indirectly. In our case the model learned that stray jumps will inhibit its ability to jump in the future when we set $\gamma=0.99$.

Building Interface

Our model is written in python and the game is built in JavaScript, we need some interfacing tools for them to communicate with each other.
Selenium, a popular browser automation tool, was used to send actions to the browser and get different game parameters like current score.

Now that we have an interface to send actions to the game, we need a mechanism to capture the game screen. Turns out selenium is capable of capturing screenshots but is very slow. A single frame took around 1sec for capture and processing.

Now that we have an interface to send actions to the game, we need a mechanism to capture the game screen. Turns out selenium is capable of capturing screenshots but is very slow. A single frame took around 1sec for capture and processing.
The PIL and OpenCV gave best performance for screen capture and pre-processing of the images respectively, achieving a descent frame-rate of 5 fps.

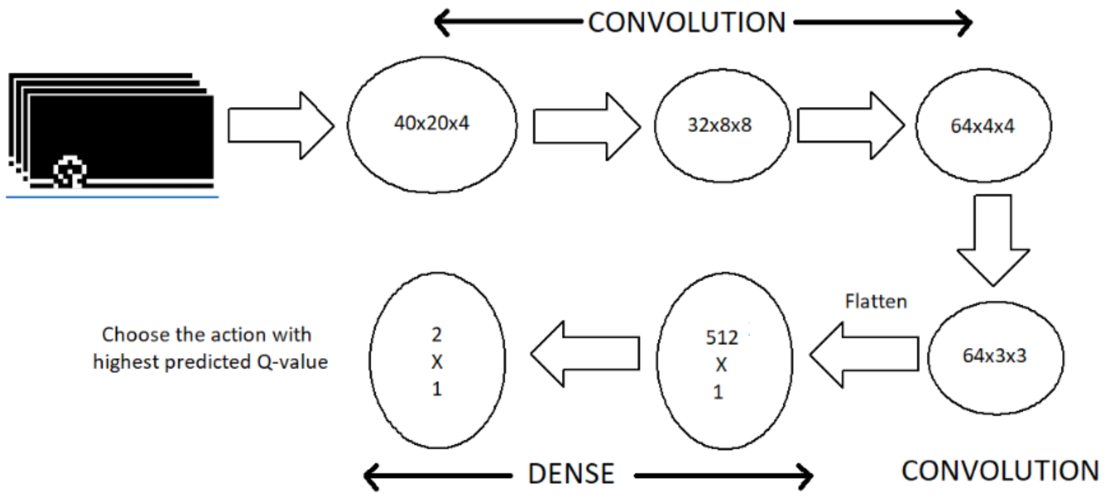
PRE-PROCESSING

Game Modifications
The original game has many features like the Dino, variable game-speed, obstacle types, clouds, stars, ground textures, etc. Learning all of them at once would consume a lot of time and might even introduce unwanted noise during training. I modified the game’s source code to get rid of few visual elements including high score and game-over panel and clouds. I limited the obstacles to a single type of cactus, and kept the speed of the runner constant.

Original Game Play

Image Processing

The raw image captured has a resolution of around 1200 x 300 with 3 channels. We intend to use 4 consecutive screenshots as a single input to the model. That makes our single input of dimensions 1200x300x3x4. The final processed input was of just 40x20 pixels, single channel and only edges highlighted using Canny edge detection.



Network Architecture with Layers

Then we can stack 4 images to create a single input. The final input dimensions are 40x20x4. Note that we have cropped the agent out because we don't need to learn the agent's features but only the obstacles and the distance from the edge.

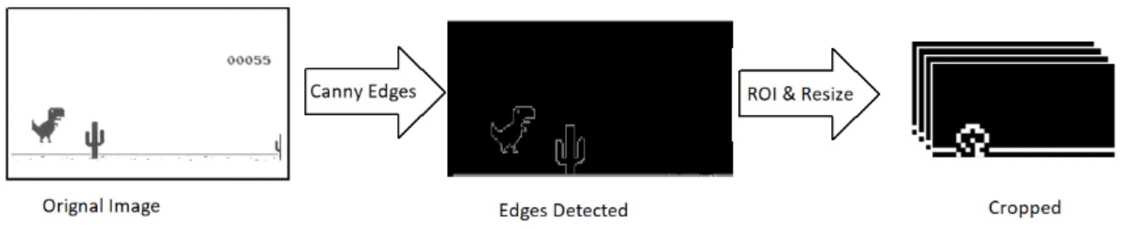


Image transformation to show only obstacles

. . .

So we got the input and a way to utilize the output of the model to play the game, now let’s look at the model architecture.

We use a series of three Convolutional layers which are flattened onto a Dense layer of 512 neurons. Don’t be surprised by missing pooling layers. They are really useful in image classification problems like ImageNet where we need the network to be insensitive to the location of objects. In our case, however, we care about the position of obstacles.

Our output has a shape equal to the number of possible actions. The model predicts a Q-value ,also known as discounted future reward, for both the actions and we choose the one with highest value. The method below returns a model built using Keras with tensorflow as back-end.

LET THE TRAINING BEGIN
The real magic happens here

This module is the main training loop . The code is self explanatory with comments. Here are few things that are happening in the game

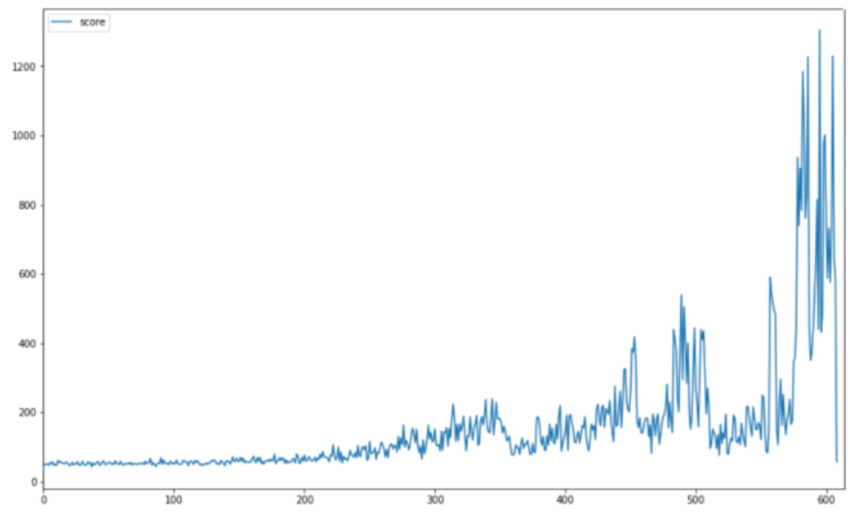
- Start with no action and get initial state(s_t)
- Observe game-play for OBSERVATION number of steps
- Predict and perform an action
- Store experience in Replay Memory
- Choose a batch randomly from Replay Memory and train model on it during training phase
- Restart if game over

We use the model on batches randomly chosen from Replay Memory

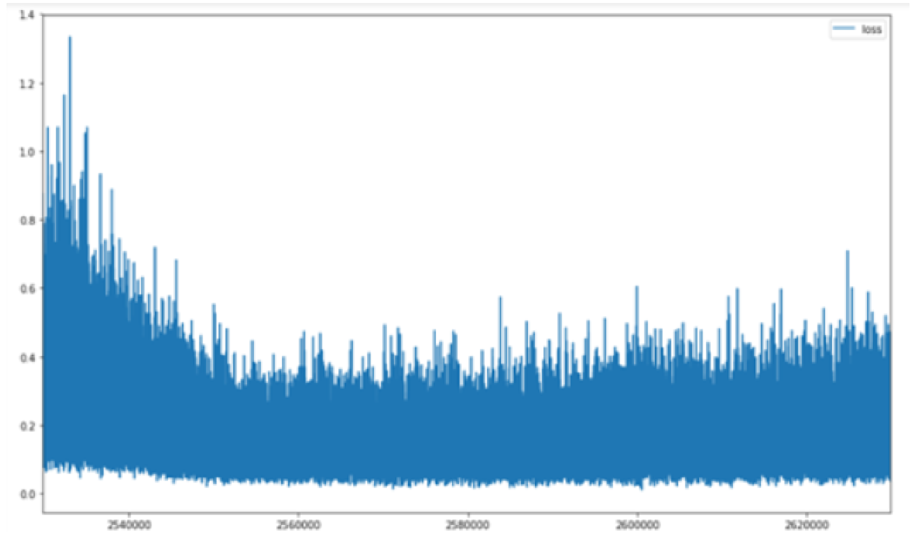
Results

We should be able to get good results by using this architecture. The GPU has significantly improved the results which can be validated with the improvement in the average scores. The plot below shows the average scores from the start of the training. The average score per 10 games stays well above 1000 at the end of training session.

Scores



Loss(MSE)


loss