Machine Learning Engineer Nanodegree

UDACITY

Capstone Proposal

Prepared by: Apoorv Malik

10 August 2019

PROPOSAL

Domain Background

In the field of Reinforcement Learning, numerous researches are ongoing. The prevalent trend in Artificial Intelligence is teaching AI agents to play video games. In 2014, Google acquired a company called Deep Mind. It created a neural network that learns how to play video games in a fashion similar to that of humans. The company made headlines in 2016 after its AlphaGo program beat a human professional Go player Lee Sedol, the world champion. It created an AI program that achieved outstanding performance on many different Atari 2600 games.

Research in this field is progressing at a swift pace. New techniques and algorithms are being discovered every year. It would be a great learning experience to implement the Deep Q-Learning algorithm and to design an AI agent that can beat a video game level on its own. Note that the DQN Agent is given only raw pixel data of what's happening on the game's screen.

One very famous game is Atari Breakout. The goal of this capstone project would be to develop a model using deep neural networks (using Deep Q-Learning algorithm) to train an AI agent to score as high as possible in the game.

My motivation for investigating this problem is that I am very fond of designing and testing AI agents to solve Reinforcement Learning tasks. Moreover, now that I understand how the Deep-Q Learning algorithm works, I ventured to implement it myself and to watch an agent get better and better through interaction.

Problem Statement

The main objective of this project is to design an AI agent that can score as high as possible in Atari Breakout. In Breakout, a layer of bricks lines the top third of the screen, and the goal is to destroy them all. A ball moves straight around the screen, bouncing off the top and two sides of the screen. When a brick is hit, the ball bounces back, and the brick is destroyed. The agent receives a reward of +1 for every brick destroyed. The agent loses a turn when the ball touches the bottom of the screen; to prevent this from happening, the agent has a horizontally movable paddle to bounce the ball upward, keeping it in play. There is a total of five turns for the agent. The episode terminates when all five turns are lost. The action space consists of four total actions: 'NOOP', 'FIRE', 'RIGHT', 'LEFT'. Given the state of the game in the form of raw pixel data, the agent needs to find the most appropriate action that will maximise the reward it will receive from the environment. The environment is completely solved when all the bricks are destroyed.

Datasets and Inputs

For this project, we will use the OpenAI's Gym library. OpenAI is a company created by Elon Musk that has been researching deep reinforcement learning. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. The open-source gym library gives us access to a standardised set of environments. It makes no assumptions about the structure of the agent and is compatible with any numerical computation library, such as TensorFlow or Theano. The Gym library is a collection of test problems — **environments** — that we can use to work out our reinforcement learning algorithms. These environments have a shared interface, allowing us to write general algorithms, that facilitate similar environments.

There are two primary components of reinforcement learning, the environment, and the agent. The agent interacts with the environment by choosing some action (from the set of all possible actions), and the environment responds to the agent by changing its state(observation) and giving out rewards(score) to the agent.

The dataset for this project comes from the OpenAI Gym's Atari game environment. For this project, we will use the Atari Breakout environment. We will use the modified version of the original Breakout environment, designed by Google DeepMind. This new version is called "BreakoutDeterministic-v4". In this version, the agent sees and selects actions on every 4th frame instead of every frame, and its last action repeats on the skipped frames. The action space of this environment consists of four possible actions, 'NOOP', 'RIGHT', 'LEFT', 'FIRE'. The definitions of these actions are as follows:

'NOOP': No Operation, no *real action* is executed when this is selected.
'RIGHT': This action moves the horizontal paddle to the right.
'LEFT': This action moves the horizontal paddle to the left.
'FIRE': This action bounces the ball upward, only executable once and at the start of a new episode.

The input that we will give to our Deep Learning Model will be the raw pixel data of the game's screen. The Atari frames are 210×160 pixel images with a 128 color palette. Providing these raw pixel data can be computationally challenging, and the algorithm may converge very slow due to high dimensional and complex input. So we need to apply an essential preprocessing step, the raw frames are preprocessed by first converting their RGB representation to a grey-scale representation, and then they are down-sampled to an 84×84 image. The last four frames (before the current frame) are preprocessed and then stacked together to produce a tensor of shape (1, 84, 84, 4) which will be the input to the DQN model. This stacking of previous frames allows the model to infer the velocity of the ball and make better decisions while choosing actions.

Solution Statement

The Deep Learning model is expected to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. For this, we require convolutional layers that can extract information from the image data, and then we need fully connected layers to form relations between states and actions. The output layer is a fully-connected linear layer with a single output for each of the valid action. The output of the model will be a vector consisting of four values (for four valid actions) with each value corresponding to the probability of taking that action With the use of both convolutional and fully connected neural network layers, the model can estimate the policy function. We will use Deep Q-learning algorithm to train our neural network. Some essential features of the DQN algorithm are explained below.

Experience Replay

The neural network's weights will not be updated after every episode. We will use *Experience Replay* to store the various experiences encountered in the game. Experience replay will store the observation tuple (state, action, reward, next state, done) that are seen during the gameplay, in a memory. After a fixed number of episodes, the model will sample a batch (of fixed size) of random experiences from the memory and use them to perform gradient descent and update step on the network. The reason why experience replay is helpful has to do with the fact that in reinforcement learning, successive states are highly similar. This means that there is a significant risk that the network will completely forget about what it's like to be in a state that it hasn't seen in a while. Replaying experience prevents this by still showing old frames to the network. It will also fix the issue of correlation between consecutive experiences, that can make the model incline towards choosing one kind of action. Hence, experience replay will ensure that the model has a stable learning rate and will prevent it from diverging.

Target Network

The Target network is another neural network with the same architecture as the main one. We update this network, after every n episodes. Recall, that the Q-values are estimated by the following formula:

$$Q(s, a) = r + \gamma \max_{a'}(Q(s', a'))$$

We use the target network to predict the targets for improving our model, more specifically, we use it to predict the future Q-values, i.e Q(s', a'). And use the main network for predicting the current Q-values. Therefore, the loss function is computed by taking the difference of the predictions of the two networks. This is shown below.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

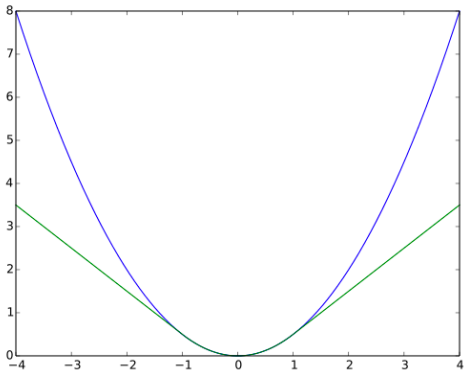
Here, θ_i^- is the target network and θ_i is the main network, we synchronise both the networks after every n episodes.

The use of target network makes the algorithm more stable compared to the standard Q-learning, where an update that increases $Q(s_t, a_t)$ often also increase $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between two consecutive Q function updates. This assures that the target y_j is stable, making divergence or oscillations much more unlikely. Eventually, we will converge the two networks so that they are the same, but we want the network that we use to predict future Q-values to be more stable than the network that we actively update every single step.

Huber Loss

We will use Huber Loss as a loss function for our Neural Network.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$



Green is the Huber loss and blue is the quadratic

This function is quadratic for small values of 'a', and linear for large values, with equal values and slopes of the different sections at the two points where $|a| = \delta$. The variable **a** often refers to the residuals, that is to the difference between the observed and predicted values, so the former expression can be expanded to:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Here, **y - f (x)** is the difference between the true value and the predicted value. The introduction of Huber loss allows less dramatic changes which often hurt RL.

Algorithm

We can now write the pseudocode for the algorithm that we will use to train our deep neural network.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Benchmark Model

For the benchmark, we will use the following scores achieved by various algorithms and human-level performance. The DQN model is expected to beat all these scores by a significant margin. The highest score in these benchmarks is that of a human. This score would be our threshold for measuring the agent's performance. The table below shows the result of various systems that played the game.

PLANNER	MODEL	SCORE
Random Policy	**N/A**	**1.2**
Human	**N/A**	**31**
Policy Gradients	**N/A**	**10.5**
MCTS Depth 10, 50 Samples	**Videoless**	**2.7**
MCTS Depth 10, 150 Samples	**Videoless**	**3.0**
Wt. MCTS Depth 10, 150 Samp.	**Videoless**	**6.1**

Table 1: Scores achieved by various methods for the game Breakout.

Evaluation Metrics

The standard evaluation metric for Atari game playing is the score returned by the game itself. The score is incremented on every successful hit on the brick by the ball. There are six rows of bricks. The color of a brick determines the points scored when the ball hits it. These are as follows:

Red - 7 points Orange - 7 points Yellow - 4 points

Green - 4 points Aqua - 1 point Blue - 1 point

The player has five total turns, and the game gets over if all the five turns are over (A turn gets lost if the ball hits the bottom of the screen). The total score is calculated at the end of the game by adding all the points scored during the game. No score penalty is applicable in this environment.

The reward/score system for the DQN agent will follow different score criteria. Moreover, the real performance and evaluation of the agent will be measured on the basis of original score criteria mentioned above. The reward/score system for the agent is as follows:

The rewards received by the agent (after hitting each block) is scaled between +1 and -1 (there is no negative reward in this environment). Since the scale of scores varies significantly from game to game, the positive rewards are fixed to be 1 and all negative rewards to be –1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude. The episode ends after a single turn is lost. However, the game resets only on true game over. This is done by DeepMind in their original solution since it helps value estimation. The total reward obtained by the agent at the end of the episode will be the total number of bricks hit (irrespective of their points), while the total points scored by the agent follow the original Breakout game's score criteria.

Project Design

The approach for making this project will include four main steps shown below.

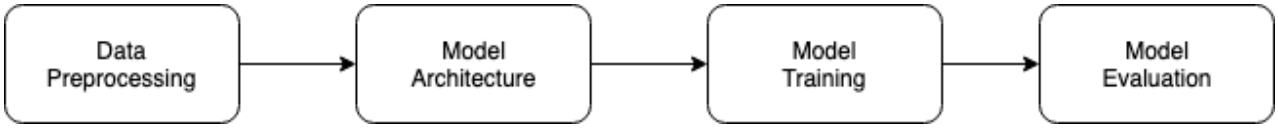


Figure 1: Project Workflow

Data Preprocessing:

The DQN agent is given the pixel data of the game's screen as input.

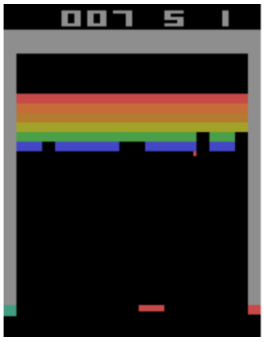


Figure 1: Single frame from Breakout

As explained earlier, the raw pixel data of the game's screen needs to be preprocessed first before supplied as input to the DQN model. The raw frames are preprocessed by first converting their RGB representation to a grey-scale representation, and then they are down-sampled to an 84×84 image.

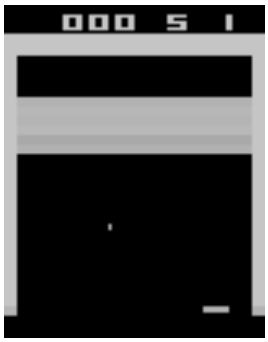


Figure 2: Single preprocessed frame from Breakout

The last four frames (before the current frame) are preprocessed and then stacked together to produce a tensor of shape (1, 84, 84, 4) which will be the input to the DQN model. This stacking of previous frames allows the model to infer the velocity of the ball and make better decisions while choosing actions.

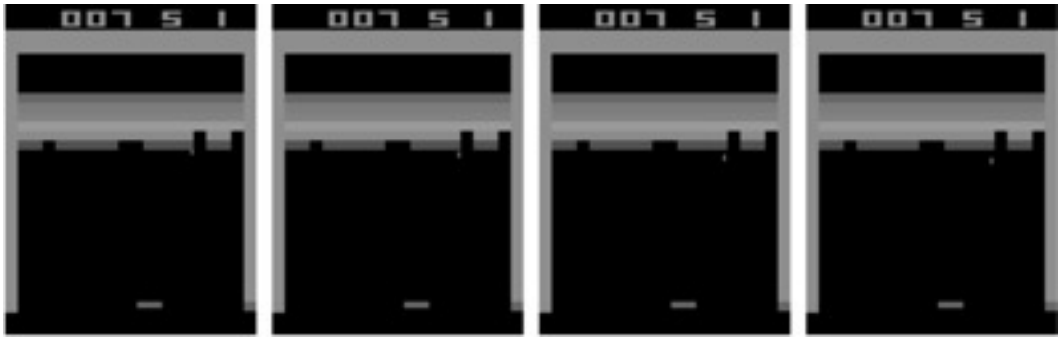


Figure 3: Four consecutive preprocessed frames from Breakout

Model Architecture:

To train the Deep Neural Network model, we will use the DQN algorithm. The network consists of convolutional layers that can extract information from the image data, followed by fully-connected layers to form relations between states and actions. With the use of both convolutional and fully-connected neural network layers, the model can estimate the policy function. The output layer is a fully-connected linear layer with a single output for each of the valid action. The output of the model will be a vector consisting of four values (for four valid actions) with each value corresponding to the probability of taking that action. A proposed structure for the architecture of the deep neural network is shown below. Note that the final architecture of the model may be slightly different.

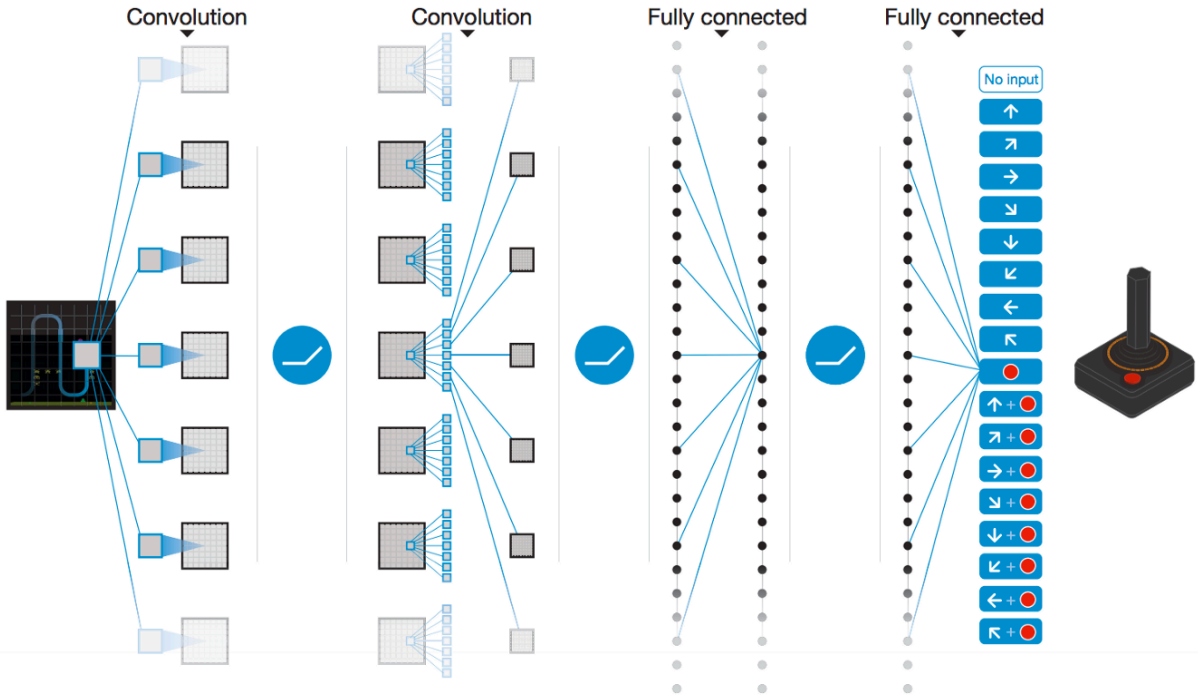


Figure 3: A visual representation of the network.

Model Training:

The model will be trained on my personal computer with tensorflow-gpu library that will train the model using Nvidia GTX 1080 GPU and 16gb of ram. The model is expected to train for 100,000-150,000 episodes. Also, the estimated time to train the model will be between 3 to 10 days.

Model Evaluation:

After the training is completed, the DQN agent is expected to perform better than all the benchmarks previously stated. I expect it to score at least 100 points in a single game. A higher score may be achieved by improving the hyper-parameters and amending the deep neural network architecture.

Plots and visualisations of multiple training sessions will be provided. It will be seen that the score increases with the episodes. All the comparisons with the benchmarks will be clearly visualised and plotted.
Finally, we will see a fully trained agent perform at the game.

References

- Justin Fu and Irving Hsu, Model-Based Reinforcement Learning for Playing Atari Games, http://cs231n.stanford.edu/reports/2016/pdfs/116_Report.pdf.
- Atari Breakout game manual, https://atariage.com/manual_html_page.php?SoftwareID=889.
- DeepMind, Human-level control through deep reinforcement learning, https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf
- Adrein Lucas Ecoffet, Beat Atari with Deep Reinforcement Learning, https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26
- https://en.wikipedia.org/wiki/Huber_loss