Reinforcement Learning Agent using Neural Networks for Geometry Friends

João Abreu
MSc student in Information Systems and Computer Engineering
Instituto Superior Técnico - Universidade de Lisboa
Lisbon, Portugal
joao.abreu@tecnico.ulisboa.pt

***Abstract*—A reinforcement learning based agent for the Single AI Circle track of the Geometry Friends AI competition is described. This agent uses value function approximation using neural networks.**

An overview of the choices made for state representation, reward function and action selection mechanisms is provided.

The implementation makes use of the Tensorflow framework in Python for training and the TensorflowSharp library for evaluation in the game agent.

I. Introduction

An agent was created for the Single AI Circle Track of the Geometry Friends AI Competition based on Reinforcement Learning using neural network function approximation.

This particular track is concerned primarily with controlling the movement of a single character instead of puzzles that require complex reasoning.

The Tensorflow framework was used to train and evaluate the feed-forward neural network that was used to approximate the value functions.

II. Background

As previously mentioned, the agent that was developed was based on Reinforcement Learning, in which the environment is modeled as a Markov Decision Process (MDP). This paradigm is concerned with agents that perform actions (a) while in a certain state of the environment (s) and receive an instantaneous reward (r_t) which is a measure of benefit.

The approach used is based on Q-Learning which learns to approximate functions ($Q(s,a)$, Equation (2)) which associate these state-action pairs with a value that is a measure of cumulative reward or long-term benefit for the agent (R, Equation (1)).

If these functions converge to their optimal solution, the optimal policy ($\pi^*(s)$) is that which selects action $\pi^*(s) = \arg\max_b Q(s,b)$.

$$R = \sum_{t=0}^{T} \gamma^t r_{t+1}, 0 < \gamma < 1 \tag{1}$$

$$Q(s,a) = E[R|s,a] \tag{2}$$

Among other ways, neural networks can be used to learn approximations for these functions based on training them with data experienced by the agent.

III. Choice of state space representation and reward function

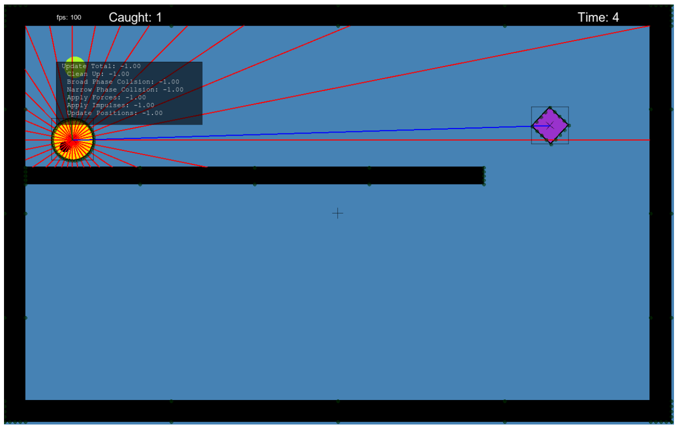
For the circle character, there are 4 available actions: doing nothing, rolling left, rolling right and jumping.

In this case, the network consisted of the input layer (explained next), three hidden layers (with 512, 256 and 128 hidden units each) and a linear output layer which represented the approximate Q-value for each possible action in the state used as input.

The inputs to the network were the following state representation in the form of an array of floating point values that consisted of (Figure 1):

- distances to obstacles in 32 directions that uniformly span 360º around the agent (red lines in the figure);
- the x and y velocities of the agent (green line in the figure);
- a vector between the agent and the two nearest collectibles (blue lines in the figure).

The choice for each of these components was motivated in allowing a representation of the immediate obstacles in the environment (analogous to laser range finders in robotics), current movement conditions of the agent and a potential direction towards progression, respectively. The constant sized floating point valued array also makes it easier to use the representation as an input to the neural network.



```
Update Total: -1.00
  Clean Up: -1.00
  Broad Phase Collsion: -1.00
  Narrow Phase Collsion: -1.00
  Apply Forces: -1.00
  Apply Impulses: -1.00
  Update Positions: -1.00
```

Figure 1. Screen-shot with debug information for the sensors of the agent.

For each level, the reward function was also defined with the intention of inducing a somewhat greedy strategy to obtain the collectibles, the choice was made to provide a reward of -1, to encourage faster completion of the levels, unless the agent got closer to any of the collectibles than ever before or the agent picked up a collectible, in which case the reward had a value of 100.

The capture of a collectible also determined that it was a terminal state, with its value fully determined by the immediate reward.

The choice of action was made using an ϵ-greedy strategy, which with $\epsilon = 10\%$ probability picked a random action and otherwise picked an action that maximized value.

The choice of random actions was made to be biased against the jumping action, since it usually has a more dramatic impact on the position of the agent and leads to a lack of control while the agent is not touching any of the platforms.

IV. Training

The additional complication of real-time decision in Geometry Friends and limitations in the C# API for Tensorflow motivated the choice of an off-line learning algorithm (Neural Fitted Q Iteration, [1]). With this algorithm only evaluation of the neural network was required in between action selection (optimization is performed between levels).

The approach alternated periods of training and data gathering.

The training consisted in optimizing the neural network to iteratively minimize the error of the predicted value functions through stochastic gradient descent.

Data gathering consisted in having the agent interact with the environment according to the ϵ-greedy policy using the neural network function approximation. The agent played a single level to completion (either completing the objectives or reaching the level-specific time-limit) and the experienced transitions, containing the observed state, performed action, reward obtained, following state and information on whether the state was terminal were stored.

The algorithm used is a slightly modified version of Neural Fitted Q iteration, that includes a fixed sized memory and a randomized order during training (Algorithm 1).

The optimizer used for minimizing the loss function was Tensorflow's Adam Optimizer with default parameters.

The agent was trained on the 5 public levels for the 2017 competition in the Circle AI track.

To make training faster, a speed-up parameter of 10 was provided to the game during the training. To ensure that the number of updates/transitions was constant during training and normal evaluation, this speed-up parameter is also used in the agent to scale the number of action selections by the agent per game simulation time-step appropriately (this value is read from a file).

V. Implementation

The described sensors, reward function and action selection mechanisms had to be implemented in the agent code in C#.

```
Initialize memory D with size N
Initialize network $Q_0(s, a)$
$k = 1$
**while** $k <= K$ **do**
    Play new game level and store transitions
      $T_t =< s_t, a_t, r_t, s_{t+1}, terminal_t >$ in D
    Randomize order of memory D
    $i = 0$
    **while** $i < I$ **do**
        Pick batch of transitions from memory
          $\{T_j \mid i \times B < j < (i+1) \times B\}$
        Compute targets:
        **if** $terminal_j$ **then**
            $target_j = r_j$
        **else**
            $target_j = r_j + \max_b Q_{k-1}(s_{j+1}, b)$
        **end**
        Minimize $Loss = \frac{1}{B} \sum_j (Q_k(s_j, a_j) - target_j)^2$
        $i = i + 1$
    **end**
    $k = k + 1$
**end**
```

Algorithm 1: Modified version of the Neural Fitted Q Iteration algorithm.

The neural network approximation was implemented and trained using Tensorflow in a Python script and loaded into the C# agent for action selection using the TensorflowSharp library.

To allow communication between the Python (training) and C# (agent) code, each of these serialized to disk the network (in protocol buffer format) and the observed transitions (in JSON format), respectively. The performance, while not ideal, proved adequate for the purposes.

VI. Conclusion and Future Developments

The main aspects of the design and development of the agent were presented.

The architecture of the agent is very simple and to tackle more complicated levels it would be interesting to incorporate a planning component, either through the reward function (reward shaping) or otherwise.

The state representation that was selected is incomplete unlike traditional MDPs, making the problem at hand a Partially Observable Markov Decision Process (POMDP). Other authors that have tackled similar problems have used a memory component in the form of recurrent neural networks [2].

VII. Files

- CircleAgent.cs - Main implementation of the circle agent
- nfq_main.py (in Agents folder) - Python script used for training, this requires the Tensorflow, Numpy and PyQtGraph packages to be installed. (note, running the script overwrites the saved network and the parameters file). It imports and calls:

- network.py - implementation in Tensorflow of the network, evaluation and optimization methods
- training.py - implements the memory that is used to store transitions during training.
- monitor.py - used to plot the progression of the training procedure.

Acknowledgment

The author would like to thank professor Rui Prada for proposing and guiding the development of the agent as well as his thesis supervisors, professors Francisco Melo and Manuel Lopes, for helpful comments and suggestions.

References

[1] Riedmiller, M.: Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3720 LNAI, 317–328 (2005)

[2] Lample, G., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. arXiv preprint arXiv:1609.05521 (2015) (2016), http://arxiv.org/abs/1609.05521