pdfpagewidth 8.5in pdfpageheight 11in

Intelligent Agents in a Probabilitsic Game : Proximity

Matthias Denu and **Donald Hamnett**
Northeastern University
College of Computer and Information Science
440 Huntington Ave #202, Boston, MA 02115

Abstract

The aim of this project was to implement intelligent agents to defeat a greedy agent in the game of Proximity.

Introduction

Background

Solving for an optimal policy in the game of Proximity is an example of a non-deterministic search problem due to the inherent randomness of the value a player will be able to place in a given move. In order to solve this problem from a pure expectimax search, it would require searching the entire state-space of the game, which is exponential in the number of discrete values and size of the board. For example, with 2 players, 6 values, and the blank tile, the number of possible values for a tile is $(2 \times 6) + 1 = 13$, and with an 8×10 grid there are 80 assignments, making the state-space $O(13^{80})$ ($O(41^{80})$ for a game with 20 possible values), which is a computationally intractable search problem. One approach to try to optimize your score is to take a greedy policy, maximizing your immediate score in a given turn. However, this proved non-optimal through our anecdotal experiences playing against a greedy agent, which Matthias was able to defeat regularly. This alludes to the fact that there does exist an optimal policy to Proximity when playing against an opponent with a fixed, non-optimal policy. The goal of our research was to determine and implement this policy against the greedy agent.

Markov Decision Processes [1] Many non-deterministic search problems can be represented as a Markov Decision Process (MDP). This is, they exhibit the Markov Property : *given the present state, the future and past are independent*. Or from a probabilistic perspective, for a state S, time-step t and action a, an MDP has the form:

$$P(S_{t+1} = s' | S_t = s, A_t = a_t, S_{t-1} = s_{t-1}, \ldots, S_0 = s_0)$$
$$= P(S_{t+1} = s' | S_t = s, A_t = a_t)$$

Figure 1: Characteristic Equation of a MDP

Though few phenomena in reality truly exhibit this property, several do exhibit it closely enough for this construction to be a useful model. To elaborate on MDPs, they defined by the following:

A set of **states**	$s \in S$
A set of **actions**	$a \in A$
A **transition function**	$T(s, a, s') = P(s'\|s, a)$
A **reward function**	$R(s, a, s')$
A **start state**	
and (sometimes) a **terminal state**	

Several strategies have emerged in order to solve for policies in MDPs, most of which are based off of Richard Bellman's pivotal work.

The Bellman Equation We need some more notation in order to formalize the Bellman Equation. Under an optimal policy he defines for a state s:

- $V^*(s)$: the value / utility of a state
  the expected utility starting in s and acting optimally
- $Q^*(s, a)$: the value of a q-state (state-action pair)
  The expected utility having taken action$a$ from s and thereafter acting optimally
- $\pi^*(s)$: the optimal action from s
- γ : discount factor, a measure of the tradeoff between immediate and future reward, which can take values $\in [0, 1]$

Following these definitions, the Bellman Equation defines the utility of a state:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \big[R(s, a, s') + \gamma V^*(s') \big]$$

Figure 2: The Bellman Equation

Bellman used this equation and his development of dynamic programming to produce the value iteration, where he initializes the zero utility to each state, and iterates over the values until convergence. This is an effective process assuming the transition probabilities are known, but can take many steps to converge.

procedure VALUE ITERATION(S, A, T, R)
  for $\forall s \in S$ **do**
    $V_0(s) \leftarrow 0$
  $t \leftarrow 0$
  repeat$\forall s \in S$
    $V_{t+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\big[R(s,a,s') + \gamma V_t^*(s')\big]$
    $t \leftarrow t+1$
  until convergence
return $V*(s) : s \in S$

There are other solutions to finding the optimal policy such as policy iteration and Monte Carlo methods, but our focus is on one of the most widely-used varieties, Q-Learning.

Q-Learning One aspect of the behavior of value iteration which was apparent was that the actual optimal policy converged before the values did. This made for an opportunity to converge to the optimal policy via a transformation of the Bellman equation to one represented by Q-values.

$$V^*(s) = \max_a Q^*(s,a)$$
$$Q^*(s,a) = \sum_{s'} T(s,a,s')\big[R(s,a,s') + \gamma V^*(s')\big]$$
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\big[R(s,a,s') + \gamma V^*(s')\big]$$

Figure 3: $V^*(s)$ in terms of $Q^*(s,a)$

Here is the Q-Learning algorithm which followed from this decomposition.

procedure Q-LEARNING(S, A, R, α =learning-rate)
  Initialize $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(terminal-state,) = 0$
  for each episode **do**
    Initialize s
    repeat(for each step of episode):
      Choose a from s using policy derived from Q (e.g.,ϵ-greedy)
      Take action a, observe r, s'
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)]$
      $s \leftarrow s'$
    until s is terminal
  return $\pi \approx \pi^*$

In order to estimate our Q-function in such a large state space, we attempted to use Artificial Neural Networks (ANNs).

Artificial Neural Networks Artificial Neural Networks have their origins in 1943, with the McCollough-Pitts Threshold Logic Unit (TLU), a mathematical interpretation how formal logic is carried out in the human brain[2]. The TLU representation of a neuron, is as a unit that takes in a set of inputs and calculates a weighted sum. This weighted sum would then be processed as:

$$f(\mathbf{x}) = \begin{cases} 0, & \big(\sum_i w_i x_i\big) + b < \text{threshhold} \\ 1, & \text{Otherwise} \end{cases}$$

Figure 4: TLU Activation

Any discrete binary task could be represented by these units, but their usefulness was simply as a model of the brain and offered no deductive improvements upon formal logic.

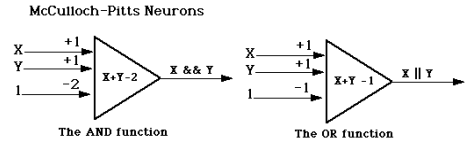

Figure 5: The McCollough-Pitts Neuron[3]

In 1958, the precursor of the modern-day neural network was born with Rosenblatt's development of the "perceptron." Expanding on the MuCollough-Pitts representation of a neuron, his perceptron was a unit that similarly took in a set of inputs and calculated a weighted sum, activating with the sign function.

$$sgn(\mathbf{x}) = \begin{cases} -1, & \big(\sum_i w_i x_i\big) + b < 0 \\ +1, & \text{Otherwise} \end{cases}$$

Figure 6: The Sign Function

The key insight of Rosenblatt was that opposed to weights being fixed, they were adjustable parameters with positive and negative weights, and could hence be learned, however, his perceptron algorithm[4] was eventually criticized for being capable of charactering solely linearly-separable decision boundaries and was most infamously incapable of computing the exclusive-or function.

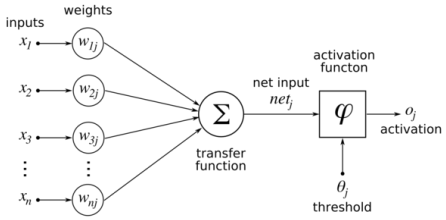

Figure 7: The Rosenblatt Perceptron[5]

The full history of perceptrons is outside the scope of this report, but after a period of falling out of fashion, the perceptron has made a massive resurgence in recent years due to the development of the ANN, where several perceptrons are combined in parallel and/or in layers to compute complex predictive models that can be applied to non-linearly separable decision boundaries. While the ANN was first developed in the 1960s, notably ADALINE and MADELINE[6], it was not until the recent increases in both computational power

and available data that their widespread use was adopted. Since then, these networks have had success in a vast collection of problems, ranging from natural language processing with recurrent neural networks, to computer vision with convolutional neural networks (CNNs).

A convolutional neural network is a popular tool used mainly in image classification and computer vision, but has been applied in recent years to a growing number of use cases. The basis of the CNN, and what makes it well suited to computer vision, is the convolutional layers. A convolution in this sense is a kernel with trainable weights which has the same depth of the image being convolved (i.e. and RGB image would have a depth of 3). However, the other dimensions of this kernel, or "filter," are usually smaller than on the other axes, which allows it to stride across windows of the image. The output of a single stride is a scalar, the sum of the element-wise product of the kernel with the input image. Each of these scalars is placed in a new grid, where spacial relativity of the original image and the kernel outputs are preserved. By using several filters and layers of convolutions (and a lot of training data), individual kernels can begin to recognize features such as edges and foreground, which when combined can lead to meaningful pattern recognition of the images.

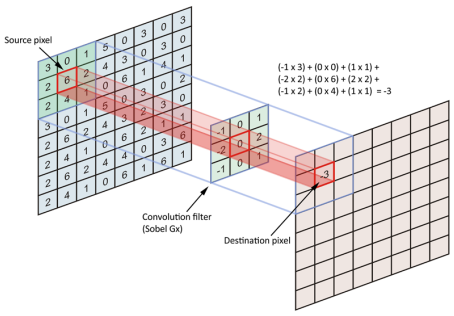

Figure 8: A convolution [7]

Deep Q-Learning One of the methods we attempted to deploy to create an intelligent agent for Proximity was Deep Q-Learning, which combines the aforementioned techniques of Q-Learning and CNNs. The motivation for this the parallel of the game board to an RGB image. It has three "depths," one each for a player's captured territory, and a discrete set of "pixel intensities," the value of the given territory, or -1 if the territory is claimed by an opponent, where the unclaimed channel's intensity is an indicator $\in \{-1, +1\}$ based upon whether or not it has been played. The idea was to train a network to glean generalize Q-values based on a convolution of the game state, in addition to the current score, the current value of the tile to play, and the remaining move values of each player (in the discrete move value version of the game).

Serving as a additional motivators were the results of Mnih, et al. in their paper "Human-level control through deep reinforcement learning"[8]. In this paper, they were able to train agents through Deep Q-Learning through the following algorithm, which we imitated.

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 $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 9: Deep Q-Learning

Project Description

Using the Keras deep learning library in Python with its functional API made it incredibly easy to build an ANN. Below is the network architecture.

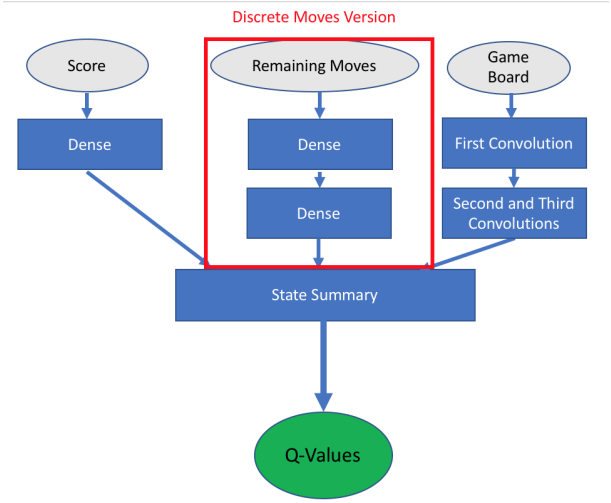

Figure 10: Model Architecture

We used the rectified linear unit (ReLU) for the hidden layer activations, and merged the layers as a linear combination for the output layer.

$$ReLU(x) = \max(0, x) \tag{1}$$

Figure 11: The ReLU function

Below are the relevant parameters of the networks:

Parameters

- Optimizer : Adam [9]
  - Learning Rate : 0.001
  - β_1 : 0.9

  - β_2 : 0.999
- Loss : Mean Squared Error
- Normalization : Batch Normalization on the Input Layer
- Dropout : Rate of 0.5 on all Hidden, Non-Convolutional Layers
- Batch Size : Range $[32, 512]$
- Epochs : Range $[10, 1000]$
- All other parameters can be found in *qnetwork.py*

Algorithm We used sample code to help implement the agent, making adjustments to suit the task at hand[10]. Her is the training pseudocode.

```
procedure TRAIN Q-LEARNING(terminalReward)
    wins ← 0
    for each experiment do
        learner ← DeepQAgent(previousWeights)
        for each episode do
            reward wins ← 0
            game ← Proximity
            while True do
                val ← next value
                state ← State(game, learner, val)
                action ← learnerAction(game, state)
                makeLearnerMove
                makeOpponentMove
                reward ← learnerScore - opponentScore
                if gameOver then
                    if learnerScore > opponentScore then
                        reward ← terminalScore
                    else if learnerScore > opponentScore then
                        reward ← terminalScore
                    nextState ← State(game, learner, val)
                    memoryReplay
(state, action, reward, nextState, Terminal)
                    break
                val ← next value
                nextState ← State(game, learner, val)
                memoryReplay
(state, action, reward, nextState, NonTerminal)
        save trained weights
```

Results

Unfortunately, we ran into some major roadblocks in our first foray into Deep Reinforcement Learning. These included, but were not limited to:

-

References

[1] Peter Norving and Stuart Russel. *Artificial intelligence: a modern approach*. Pearson Education Limited, 2013.

[2] Warren S. Mcculloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/bf02478259.

[3] George F. Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson Addison-Wesley, 2009.

[4] "Perceptron Algorithm, 1959; Rosenblatt". In: *SpringerReference* (). DOI: 10.1007/springerreference_57806.

[5] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[6] B. Widrow and M. E. Hoff. "Adaptive Switching Circuits". In: (1960). DOI: 10.21236/ad0241531.

[7] Pjreddie. *pjreddie/cnn-primer*. URL: https://github.com/pjreddie/cnn-primer/tree/master/1.

[8] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. DOI: 10.1038/nature14236.

[9] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[10] Keon. *keon/deep-q-learning*. URL: https://github.com/keon/deep-q-learning/blob/master/dqn.py.