CS221 Project Progress Report: Breakout Bot

Priyanka Rao, Reid Pryzant, Vincent-Pierre Berges

Introduction

Breakout has a rich state space and is aptly suited for a reinforcement learning (RL) approach. By applying variants of the well-known Q-learning algorithm, we seek to deepen our understanding of the problem and solution space. Since our project proposal, we have completed a custom implementation of the Breakout game. Furthermore we have decided how to model the game as a semi-deterministic MDP, and have begun experimenting with various reinforcement learning algorithms, function approximators, and feature sets.

Model

States and Actions

Our breakout game can be modeled as a non-deterministic MDP. The state-space of the game is the set of possible screens that could occur during gameplay. The action space is the set of possible moves that the paddle can take: {`left, right`}. Most state-action transitions have probability 1 and are deterministic, with the exception of the initial trajectory of the ball, which is random.

Unlike Google Deepmind, we chose not to model the game's state-space as a grid of pixels. While we may use a neural network function approximator, we don't plan on using a CNN feature extractor because we want the focus of our project to be on algorithms. Furthermore, since we implemented the game ourselves, we have direct access to whatever features the CNN could tell us.

On each frame, the game engine presents a vector of raw, untreated state attributes to the game-playing agent. This vector consists of every instance variable belonging to the game: the ball, its location and velocity, positions of all remaining bricks, position of the paddle, game state (yet to start, underway, game over), etc. This is the "raw state" of the game. Gameplay is modeled as the modification of these state variables over time.

Features

Our featurizers map state vectors into a reduced feature space. This serves two purposes: first, the reduced feature space is more amenable to learning. Second, some of our function approximators require special representations that the raw state cannot accommodate, like discrete features for traditional Q-learning. Feature sets that we have implemented thus far include:

- One-hot encodings of discretized ball position, speed, direction, and paddle position. This feature set is used by an implementation of Q-learning without function approximation.
- Difference between the ball's X coordinate and the paddle's X coordinate. This feature set is used by a simplistic agent we tried to teach to follow the ball.

- Ball angle, velocity, and paddle/ball differences in X coordinate. This feature set is used by simple continuous function appropriators.
- The complete, raw state. This feature set is used by a deep neural network function appropriator that we hope will learn useful higher-level representations.

Though feature vectors are technically a function of state and action, we have not implemented any action-specific features yet because individual left/right movements have negligible impact on game state. We do, however, map (state, action) tuples to each feature vector.

Algorithms

We will restrict ourselves to model-free, off-policy algorithms for learning and control. We choose model-free because our environment may be known (we implemented the game so we know all its rules), but is too large & ungainly to model directly. On-policy methods like $SARSA(\lambda)$ offer a tantalizing middle ground between the high variance model-free MC estimate and biased bootstrap estimates like $SARSA$. However, off-policy algorithms give us the ability to reuse experience generated by old policies. The literature has shown that this level of data efficiency is invaluable when trying to perform RL in a problem space as large and complex as our own. Furthermore, many on-policy methods are prohibitively slow in our setting. MC and $SARSA(\lambda)$ (with large λ) require that we run out long stretches of the game before updating our estimates. We prefer the speed and convenience of stochastic, online updates.

We have implemented several variations of Q-learning with function approximation using an ϵ-greedy acting policy. This is a model-free, off-policy approach where the estimated utility of being and a state s and taking an action a is updated according to the following implied objective:

$$min_w \sum_{s,a,r,s'} (\hat{Q}_{opt}(s,a;w) - (r + max_{a' \in actions(s')} \hat{Q}_{opt}(s',a';w)))^2$$

where $\phi(s,a)$ is a mapping from a (state, action) tuple to a feature vector (discussed above). Minimizing this function corresponds to following the following stochastic gradient descent (SGD) update on each (s,a,r,s'):

$$w' = w - \eta\Big(\hat{Q}_{opt}(s,a;w) - (r + \gamma \cdot max_{a'} \hat{Q}_{opt}(s',a';w))\Big)\phi(s,a)$$

So far we have implemented three function approximators: simple linear regression and logistic regression. We intend on implementing neural networks next. Because these are all differentiable, we may use SGD to update the parameters of this model on every step.

There remain two problems with Q-learning with function approximation that we have yet to fix.

1. If we apply SGD at every step than we will have a serious problem with multicolinearity in our "training data". Trajectories through a are highly correlated, so our experiences will not be i.i.d. Additionally, we throw away experience as soon after we visit it once. We hope to correct for this by making use of experience replay. This involves caching a record of agent experience, sampling from this experience at each time step, and applying SGD towards the Q-learning target of this sampled record. This will decouple game trajectories and give the learner algorithm iid training data. It also lets us revisit and relearn from our experience, one of the big draws of off-policy learning in the first place.

2. Q-learning is a bootstrap procedure. In its current definition we the freshest weights to calculate an estimated target. This means that our target estimates (and thus our updates) will be highly unstable since the underlying function changes with every step. We aim to correct for this by incorporating fixed Q targets. We periodically freeze an auxiliary set of weights, and use these for calculating the targets.

Implementation

Our implementation is structured into the following abstract base classes and sub-classes

- `Breakout`. The Breakout game engine is a base class containing all the Breakout logic. It is implemented with `pygame`, and is a heavily modified version of the `pygame` breakout tutorial program. 65 lines of the original Breakout implementation remain (12.5% of the game engine code). The game engine is capable of initializing a Breakout game, handling input, and executing individual game turns. All subclasses override an abstract `run()` method, which repeatedly provides input and executes game turns.
  - `HumanControlledBreakout`. This is a game object which takes input from the keyboard in its `run()` method.
  - `BotControlledBreakout`. This is a game object which contains an `Agent` as an instance variable. On each turn it presents the current state and reward to the agent. The agent updates the Q-values or weights, takes the decision for the next action using the Q-values, updates the past experience. The game then decides whether to explore (using a ϵ-greedy acting policy) or follow this optimal movement.
  - `OracleControlledBreakout`. This is a game object which breaks the rules of the game to win.
- `Agent`. This is a base class that game-playing agents inherit from. Agents need to be given a function approximator. This can be linear (weights are updated as the agent learns) but it can be a lot more complex.
  - `DiscreteQLearningAgent`. This is an implementation of the vanilla Q-learning algorithm. All the positions, angles and speed of the ball and paddle are transformed into binary variables (Is the ball x position between 0 and 8 pixels for instance) by the `SimpleDiscreteFeatureExtractor`. In this configuration, there is no function approximation, simple Q-learning is used.
  - `FuncApproxQLearningAgent`. This agent has a function approximator as an instance variable. It uses this function approximator to process states and compute Q-values.
- `FunctionApproximator`. This is the base class that function approximators inherit from. Each function approximator contains a scoring function and a training step function.
  - `LinearFunctionApproximator`. Linear regression:

$$\hat{Q} = \mathbf{w}^\intercal \phi(s, a)$$

  - `LogisticRegression`. Logistic regression:

$$\hat{Q} = \frac{1}{1 + e^{\mathbf{w}^\intercal \phi(s,a)}}$$

- `FeatureExtractor`. This is the base class that feature extractors inherit from. See the modeling section for a complete description of each feature set.
  - `SimpleDiscreteFeatureExtractor`
  - `ContinuousFeaturesWithInteractions`

A game has an agent which has a function approximator which has a feature extractor. These inheritance relations took time and effort to implement properly but will undoubtedly prove useful as we continue developing our solution. The learning algorithm, function approximator, and feature set are all independent modules that can be swapped in and out with ease.

Preliminary Results

We evaluated the performance of a simple Q learning algorithm without function approximation and two linear function approximators, one with a discrete feature set and one with a continuous feature set. *Table 1* shows a comparison of the average score achieved over 20k games. Note that the game awards 3 points for each broken brick.

Both of our function approximators outperform a baseline agent that acts randomly. This is a promising result because it shows that our agents are learning something. However, it is evident that there is much room for improvement: on average, the function approximator with continuous features breaks two bricks before dying.

For some more preliminary results, we tested each agent on a single 20k run with varying exploration probabilities (*table* 1). In general, it is evident that more exploration helps the agent. This makes sense, as the state-space of the game is quite large.

We tried to better understand the mechanism behind learning by visualizing the score for each game in a single 20,000 game run across multiple learners. (*Figure 2*). While simple Q learning had the lowest scores on average, it surprisingly had the highest score overall for a single game at 117 points. This can likely be attributed to random chance - the ball's trajectories for that game happened to pair well with the system's current Q-values.

In other preliminary experiments, we tried to find the optimal values of the grid step size, angle step size, and ball speed step size used in the game-playing engine. However, none of the values we tested significantly affected the average game score. In future experiments, we hope to test different combinations of values for these hyperparameters over many more games.

Appendix

Discrete Q-Learning	Function Approximation: discrete features	Function Approximation: continuous features	Random Agent (baseline)
3.927	4.637	6.150	3.404

Table 1: Mean score over 20,000 games (averaged over 5 batch-epochs) for each implementation with $\epsilon = 0.6$.

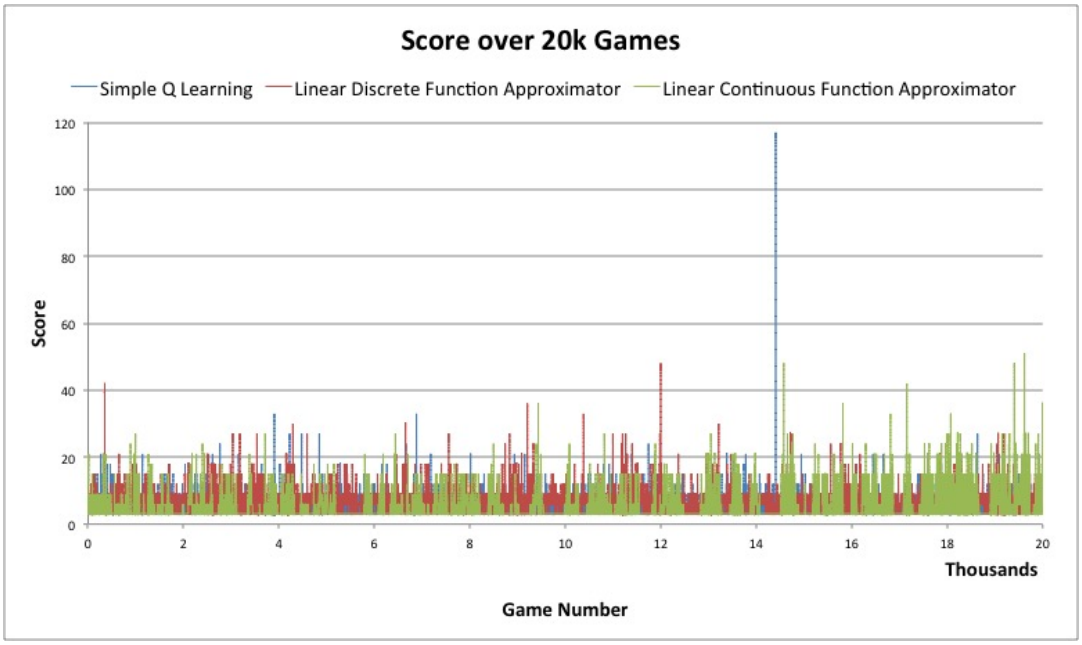

Figure 1: Score over 20k games for Breakout, with epsilon = .9 for simple Q learning and epsilon = .4 for the two function approximators. For the discrete function approximator, ball angle was discretized into 8 angles, speed step size was 3, and grid step size was 7.

Epsilon	Discrete Q-Learning	Function Approximation: discrete features	Function Approximation: continuous features
0.25	4.35	4.30	4.03
0.5	3.97	4.10	4.47
0.75	4.01	4.60	4.72

Table 2: Score over 20,000 Breakout games, with varying exploration probability and all other factors (grid step size, angle step size, ball speed step size) held constant.