CS221 Fall 2016 Project Progress
Learning to Ski

SUNet ID: smiel Name: Shayne Miel

December 16, 2016

Abstract

Due to the constrained environment and well-defined inputs and outputs, video games are a ripe testing ground for reinforcement learning algorithms. In this project, we develop an agent for the simulated environment of the Atari game Skiing, using a fitted Q-iteration method with experience replay, reward shaping, and hand-engineered features to approximate an optimal policy for the near-continuous state space and discrete action space of the underlying Markov Decision Process model. We show that this method outperforms a reasonable baseline and provide suggestions for future improvements.

1 Model

Our model is the standard RL model, in which we have an agent who observes an environment with a (possibly infinite) set of states, $S \subseteq \mathbb{R}^{D_S}$, where $D_S \in \mathbb{N}$ is the dimensionality of the state space. The agent must make decisions about actions, $A \subseteq \mathbb{N}$, to take in order to maximize the cumulative reward, $R \in \mathbb{R}$, generated by the transitions between states. Our goal is to learn a policy, $\pi(s_t) \rightarrow a_t$, with $s_t \in S, a_t \in A$, that will maximize the total reward, R.

More generally, the environment is modeled as a Markov Decision Process (MDP), in which the transition between states is non-deterministic based on the actions, and neither the transition function nor the reward function is known.

1.1 States

The Atari simulator[1] provides the agent with observations of the game screen at discrete time steps. These screen shots are given as 3-dimensional matrices of shape $(250, 160, 3)$ which represent the RGB values of the screen's pixels. See Appendix A for a screen shot of the game.

From this information, we extract several measurements that we store as the state of the the MDP. For the state at step t, we calculate:

[1] OpenAI Gym

- $score_t$: the set of pixels that display the number of slaloms the skier has gone through at step t. This section of the screen is rows 30 to 39, and columns 65 to 82.
- $skier_x^t$: the horizontal pixel offset between the left edge of the screen and the middle of the skier.
- $skier_y^t$: the vertical pixel offset between the top of the skier's head at step t and the top of the skier's head at the beginning of the episode[2].
- $slalom_{x,y}^t$: the pixel location of the midpoint between the two flags of the nearest slalom whose y value is greater than $skier_y^t$ (i.e. the next slalom for the skier to go through).
- $time_t$: the number of centiseconds elapsed on the game clock between the beginning of the episode and step t.
- $motion_{x,y}^t$: $[skier_x^t - skier_x^{t-2}, skier_y^t - skier_y^{t-2}]$. We use a two-state gap because the discrete nature of pixels causes the one-state change to sometimes be 0 when the skier is moving slowly along one axis or the other.
- $speed_{x,y}^t$: $\frac{motion_{x,y}^t}{time_t - time_{t-2}}$
- cos_t: $\cos(motion_{x,y}^t, [1,0])$[3]

1.2 Actions

The action space is much smaller than the state space. The agent's options are to do nothing (noop), turn left (left), or turn right (right), indicated by 0, 1, and 2, respectively. Note that turning left or right is relative to the skier's orientation, which is opposite the player's orientation.

1.3 Transitions

When an agent chooses an action to perform, the simulation repeats that action for k frames, where k is sampled from $\{2,3,4\}$ with uniform probability. The next observation does not come until after those k frames.

1.4 Rewards

All rewards in the game are negative (i.e. costs). At each step of the simulation, the agent recieves between -3 and -7 points, depending on how many frames

[2]In the simulation, the skier does not change her vertical location on the screen. Rather, the hill scrolls upwards past her. We use the equivalent but more natural language of the skier moving down the hill.

[3]If $motion_{x,y}^t = [0,0]$ then this value is undefined. In that case, we use the convention that $cos_t = cos_{t-1}$.

were randomly skipped in the transition. This value is equal to the number of centiseconds passed on the game clock. At the end of the episode, the agent recieves -500 points for every slalom (out of 20) that was missed. Once the skier passes the 20th slalom, whether or not they go through it, the episode ends and the rewards begin again from 0. The episode also ends if the game clock reaches 5 minutes, in which case, no points are deducted for having missed slaloms. This means that the minimum possible reward is -39999 (cross the finish line 1 centisecond before the 5 minute timeout, having missed all 20 slaloms). The maximum reward is unknown but less than 0, because it takes a certain amount of time to go through all 20 slaloms.

2 Related Work

A lot of work has been done recently applying reinforcement learning techniques to Atari games. Most notably, the researchers at Deep Mind developed DQN, a deep neural network approach to Q-Learning.[4] They followed up that work by exploring asynchronous versions of Q-learning, Sarsa, and Actor-Critic.[5] Jaderberg, at al. expand on the Asynchronous Actor-Critic work by adding auxiliary control tasks via unsupervised psuedo-rewards in their UNREAL algorithm.[1] To our knowledge, none of these methods have been applied to the Atari Skiing game, in particular.

3 Method

The methods chosen to learn an optimal policy were driven by both the unique challenges of the task, as well as a desire to make the solution as simple as possible to aid in debugging and understanding. Because of the massive state space and the random frame skipping in every transition, it was important to use a method that accumulated multiple data points per learning cycle. On the other hand, episodes in this simulation are quite long; taking thousands of actions before any meaningful reward is given. Furthermore, the main signal in the rewards (i.e. the costs of missing slaloms) depend on events that happen hundreds or thousands of actions before the reward is actually given. To address these issues, we implement fitted Q-iteration with experience replay and reward shaping (or, more accurately, pseudo-rewards) to learn a Q-function approximator.[2, 3, 6, 7] Each of these components of the overall method will be described below.

During training, we follow an ϵ-greedy policy, which randomly samples an action from a uniform distribution. With probability $1 - \epsilon$, we instead take a weighted random sample from the softmax function $\sigma(z)$ where

$$z = \{Q_i(state_t, action) : action \in Actions\}$$

.

3.1 Fitted Q-Iteration

Fitted Q-Iteration, as described by Ernst, et al., is the batch mode extension of the standard Q-Learning method, in which we wish to learn the optimal Q-function,

$$Q_{opt} = \sum_{a \in A} T(s, a, s')(R(s, a, s') + \gamma \max_{a' \in A} Q_{opt}(s', a')$$

This recursion requires knowing the transition and reward functions for every state and action, which is not feasible with very large state spaces like the Skiing game. Instead, we use data sampled from a non-optimal policy to train a Q-function approximator in batches of (s, a, r, s') tuples. We iteratively improve the Q-function approximator over a number of data collection and training steps. Assume at step t we have collected n of these (s, a, r, s') tuples, generated by following the policy given by Q_{i-1}. We then construct a standard supervised learning task, in which the inputs, I and outputs O are

$$I_i = \phi(s_i, a_i)$$

$$O_i = r_i + \gamma \max_{a \in A} Q_{i-1}(s'_i, a')$$

We can then learn Q_i by training a regression algorithm to predict $\{I \to O\}$, and learn Q_{opt} by repeating this process until convergence.

3.2 Q-Function Approximation

To approximate the Q-function itself, we use L2-regularized linear regression, with a minibatch stochasitc gradient descent optimizer. Each iteration of the algorithm given above deines a minibatch, with one epoch per minibatch. We use randomly initialized weights (uniformly sampled from $[-0.5, 0.5]$) for the first iteration, and the previously learned weights as the starting point for each successive iteration. We also center and scale the features before training, and use the transformation learned on the tuples from iteration $t - 1$ to center and scale the features going into Q_{i-1} while collecting data for iteration i.

3.3 Experience Replay

Experience replay is closely related to the idea of batch mode learning. In Fitted Q-iteration, one updates the Q-function approximator every n steps with the (s, a, r, s') tuples collected over those n steps. We extend that algorithm by maintaining an M-length FIFO queue of (s, a, r, s') tuples, where $M >> n$. At each update of the Q-function, we append the most recent n tuples to the queue, regenerate the input/output pairs, and train on all M instances. This allows us to disconnect the frequency of training from the information content available at each iteration.

3.4 Reward Shaping / Pseudo-Rewards

The reward given at each step of the simulation is the time, in centiseconds, elapsed on the game clock. While the total number of elapsed centiseconds is an important metric to reward, at the step level this value is based only on the number of frames randomly skipped by the simulation, and therefore not indicative of good or bad performance on the part of the skier. To complicate matters, the largest part of the final reward - the negative value for missing slaloms - is not provided until the end of the episode.

In order to more closely align the transition of passing through a slalom with the relevant reward, and so that we can iterate on the Q-function more often than once an episode, we have developed some pseudo-rewards with which to train the agent. In fact, the true rewards are discarded, and the agent only sees these pseudo-rewards (although the pseudo-rewards are meant to approximate the true final reward value).

These rewards are not calculated until the n states have been collected for the next iteration of Fitted Q-Iteration. They are then added to the states and actions to make the (s, a, r, s') tuples in the experience replay memory.

Given a list of states collected at steps $[t_1, \ldots, t_n]$, the reward function is

$$R'(s_{t_i}, a_{t_i}, s_{t_{i+1}}) = 500 \cdot \mathbb{I}[score_{t_i} \neq score_{t_{i+1}}] - \frac{time_{t_n} - time_{t_1}}{skier_y^{t_n} - skier_y^{t_1} + \epsilon}$$

In English, we give a large positive[4] pseudo-reward for going through a slalom, and a cost that is inversely proportional to how quickly the skier moves down the hill. ϵ in the above equation is to prevent divide by zero errors when the skier is not moving down the hill.

3.5 Feature Engineering

In contrast to most of the recent deep neural network approaches in the reinforcement learning literature, we chose to develop a set of handcrafted features for the Q-function approximation. Due to the linear nature of our regression algorithm, we needed to inject some amount of non-linearity in the features themselves. In addition, we needed a clever way to combine the action under consideration with the features of the current state.

To do this, we created two helper functions:

- An action transformation, $f(action, speed_{x,y})$
- A feature/action convolution, $g(feature, threshold, f(action, speed_{x,y})$

We briefly explain these two functions below.

[4]As an implementation detail, we've found that multiplying this reward by -1 and setting $V_{opt} = \min_{a' \in A} Q_{opt}(s', a')$ leads to better performance, so that is how our code is written. The formulations are equivalent though.

3.5.1 Action Transformations

At every step, the skier is presented with the choice of 3 possible actions: do nothing (noop), turn left (left), and turn right (right). Regardless of the choice of action, the forces of gravity and momentum continue to affect the skier as she turns or stays straight. The effects of turning to the left or right are relatively similar across the states and can work with or against those forces of gravity and momentum. Choosing to do nothing, on the other hand, means that you are choosing explicitly to allow gravity and momentum to rule the skier's motion. That is, choosing to do a "noop" when you are gliding to the right is fundamentally different from a "noop" when you are gliding left or at a stand-still.

$f(action, speed_{x,y})$ sub-categorizes the proposed "noop" action, based on the speed information extracted from the state. More formally,

$$f(action, speed_{x,y}) = \begin{cases} left & action = left \\ right & action = right \\ noop_{LD} & (action = noop) \wedge (speed_x < 0) \wedge (speed_y > 0) \\ noop_{D} & (action = noop) \wedge (speed_x = 0) \wedge (speed_y > 0) \\ noop_{RD} & (action = noop) \wedge (speed_x > 0) \wedge (speed_y > 0) \\ noop_{LR} & (action = noop) \wedge (|speed_x| > 0) \wedge (speed_y = 0) \\ noop_{NONE} & (action = noop) \wedge (speed_x = 0) \wedge (speed_y = 0) \end{cases}$$

3.5.2 Feature/Action Convolution

To motivate the need for the following function, consider a state with a feature whose value is saying that the skier is 20 pixels to the left[5] of the slalom. Let us also assume that the weight of this feature in the regression model is positive. We would like to transform the feature so that when the skier is considering turning left[6] (i.e. towards the slalom), the resulting feature is positive, and when the skier is considering turning right (i.e. away from the slalom), the resulting feature is negative.

$g(feature, threshold, f(action, speed_{x,y}))$ takes a hand-engineered feature (a measurement of the state) and modifies it based on a threshold and the transformed action so that a linear algorithm can model it. It does this by mutliplying the raw feature by a constant factor, based on the feature's value with respect to the threshold and the transformed proposed action. Table 1 lists the scaling factors.

3.5.3 Features

With those two functions in hand, we can now describe the three features used in our approach. Below we give the intuition for each feature, as well as the

[5] in screen orientation

[6] in the skier's orientation

$f(action, speed)$	$feat < -thresh$	$-thresh \leq feat \leq thresh$	$thresh < feat$
$left$	2	-2	-2
$right$	-2	-2	2
$noop_{LD}$	-1	0	1
$noop_{D}$	0	2	0
$noop_{RD}$	1	0	-1
$noop_{LR}$	-3	-3	-3
$noop_{NONE}$	0	2	0

Table 1: Scaling factor for feature (feat) based on transformed proposed action and threshold (thresh)

formulation.

- Head towards the next slalom
  - $g(skier_x^t - slalom_x^t, \frac{\text{slalom width}}{2}, f(action, speed_{x,y}^t))$
- Stay away from the edge of the screen
  - $g(skier_x^t - \frac{\text{screen width}}{2}, \frac{\text{screen width}}{2} - 20, f(action, speed_{x,y}^t))$
- Head down hill
  - $g(cos_t, 0.44, f(action, speed_{x,y}^t))$

4 Results

In order to analyze the performance of our method, we compare it to two static baselines and an oracle. The first baseline chooses actions at random (*Random*). This policy leads the skier through a small number of slaloms accidentally, but takes a very long time to make its way down the hill. The second baseline always chooses "noop", which causes the skier to head straight down the hill (*Straight*). This is a much faster policy, in terms of total elapsed game time per episode, and because of the placement of the slaloms towards the center of the slope, it leads the skier through almost half of the slaloms.

The upper limit of the true per-episode reward is unknown. To approximate it, we implemented a policy that waits for human input before choosing each action (*Human*). The reward achieved by this method is higher than a typical human player of the arcade game, because the simulation pauses while the player decides what action to request next.

We compare these 3 policies with our method (*L2S*), using a 50 episode burn-in and averaging the true per-episode reward of the following 100 episodes. For the human oracle, the reward is averaged over only 3 episodes. The results are listed in Table 2.

The results in Table 2 were run with Q-Iteration every 75 steps and an experience replay memory length of 3000 steps. Because we have disconnected

Method	Reward	Due to Elapsed Time	# Slaloms Missed
Random	-16389.17	-8204.17	16.37
Straight	-9013.00	-3513.00	11.00
L2S	-7016.09	-5521.09	2.99
Human	-3662.33	-3662.33	0.0

Table 2: 100-episode average reward

Memory Length	Reward	Due to Elapsed Time	# Slaloms Missed
500	-7605.09	-6005.09	3.20
1000	-7956.27	-6076.27	3.76
2500	-7479.71	-5649.71	3.66
5000	-7192.96	-5537.96	3.31
10000	-7268.47	-5428.47	3.68

Table 3: 100-episode average reward for L2S with training every 75 steps and varying memory lengths

Memory Length	Reward	Due to Elapsed Time	# Slaloms Missed
10	-8628.98	-6383.98	4.49
30	-7584.33	-5764.33	3.64
60	-7393.69	-5653.69	3.48
90	-7341.78	-5701.78	3.28
120	-7332.87	-5662.87	3.34

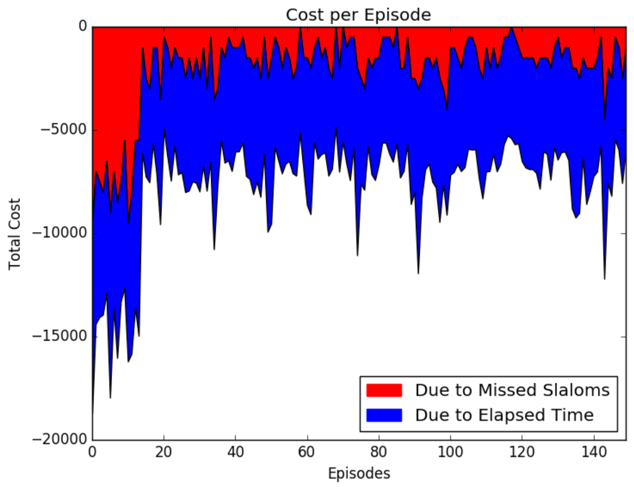
Table 4: 100-episode average reward for L2S with memory length of 3000 and varying training frequencies

the frequency of training from the size of the memory, it is interesting to look at how performance varies as we tune those parameters. Those results are shown in Table 3 and Table 4.

5 Analysis

If you compare the results of the L2S algorithm and the human oracle, you can see that there is still a decent gap in performance between the two. Why is this? Empirically, it is because the skier is getting “stuck”. As she moves down the hill, the skier will occasionally turn all the way to the left or right, stopping her downhill momentum. She will remain in this position, trying to turn further in that direction, until something changes. You can see this behavior in the simulation[7] as well as in Figure 1. Notice how every so many episodes, the cost due to elapsed time spikes. These are instances where the skier is getting stuck instead of heading down the hill as desired.

Figure 1: Components of per-episode reward



Our suspicion is that this behavior is due to some combination of noisy state measurements, the small feature set, and the feature/action convolution step. It is easy to picture a case where the skier has gone through a slalom at a shallow angle, and subsequently learns that horizontal motion is more valuable than downhill motion. It is also likely that, as the skier slows down, the measurement of her speed becomes less accurate. Unfortunately, our attempts to debug the issue have not been successful. We mention this here for future work. Perhaps using less complex features and a more complex regression algorithm will alleviate the problems.

[7]Use the ‘-r’ flag in the eval.py script to get live video rendering of the game

Speaking of debugging, we have included notebook.pdf (in data.zip), a Jupyter notebook created while trying to fix an issue with our measurement of the slalom location. This may be of more interest to the practioner than the researcher.

6 Future Work

The current state of the art for reinforcement learning algorithms on Atari games is to use deep neural networks to model whichever portion of the policy is being learned.[5, 1] Deep networks are notoriously difficult to interpret, although Zahavy, et al. make a valient effort for DQN.[8] In order to make the system less of a black box (mostly for debugging purposes), we chose to use a simple L2-regularized regression with a minimal number of hand-crafted features for the Q-function approximation. We consider this a base upon which further research can be built.

In particular, making the features less complex, coupled with a more sophisitcated non-linear algorithm for the Q-function approximation, might alleviate the issues keeping us from achieving oracle-level performance.

References

[1] Jaderberg, Max, et al. ”Reinforcement learning with unsupervised auxiliary tasks.” *arXiv preprint arXiv:1611.05397* (2016).

[2] Ernst, Damien, Pierre Geurts, and Louis Wehenkel. ”Tree-based batch mode reinforcement learning.” *Journal of Machine Learning Research* 6.Apr (2005): 503-556.

[3] Lin, Long-Ji. ”Self-improving reactive agents based on reinforcement learning, planning and teaching.” *Machine learning* 8.3-4 (1992): 293-321.

[4] Mnih, Volodymyr, et al. ”Human-level control through deep reinforcement learning.” *Nature* 518.7540 (2015): 529-533.

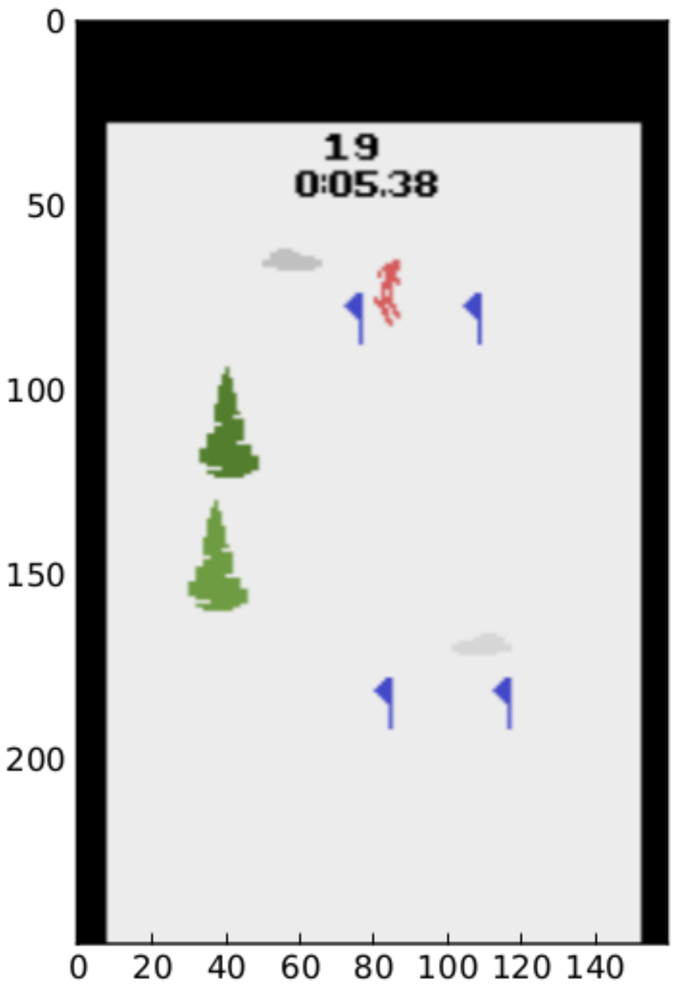
[5] Mnih, Volodymyr, et al. ”Asynchronous methods for deep reinforcement learning.” *arXiv preprint arXiv:1602.01783* (2016).

[6] Ng, Andrew Y., Daishi Harada, and Stuart Russell. ”Policy invariance under reward transformations: Theory and application to reward shaping.” *ICML.* Vol. 99. 1999.

[7] Riedmiller, Martin. ”Neural fitted Q iterationfirst experiences with a data efficient neural reinforcement learning method.” *European Conference on Machine Learning.* Springer Berlin Heidelberg, 2005.

[8] Zahavy, Tom, Nir Ben Zrihem, and Shie Mannor. ”Graying the black box: Understanding DQNs.” *arXiv preprint arXiv:1602.02658* (2016).

A Screenshot of Skiing Simulator


19
0:05.38
0
50
100
150
200
0
20
40
60
80
100
120
140