Reinforcement Learning with Deep Q-Networks for Classical Arcade Games

Gabriel Ewing
Department of Electrical Engineering
and Computer Science
Case Western Reserve University
Cleveland, OH
gre5@case.edu

***Abstract*—I implemented deep neural networks for playing Atari 2600 games using Q-Learning. We create Voronoi diagrams for two-dimensional maps with point obstacles. We explore the algebraic and geometric intricacies of the Sweepline algorithm used to build the diagrams. We review the concept of a k-Nearest-Neighbors classifier and use the completed Voronoi diagram to find nearest neighbors.**

I. Introduction

Over the past few decades, computers have been able to play a series of increasingly complex games at a high level, perhaps most notably in Deep Blue's defeat of reigning world chess champion Garry Kasparov in 1997 and AlphaGo's victory against top Go player Lee Sedol in 2016. Complexity, in this case, is mostly tied to the *branching factor* of the game in question; that is, the number of actions that are available to a player at a point in time. While chess and Go are both complex beyond the point of human understanding, chess was won much sooner by computers because it has a significantly lower branching factor than Go.

Despite their massive possible game trees, both of the aforementioned games are laughably simple compared to the real world in which we live every day. This is in part due to the fact that the state at any given time for both games can be entirely encoded by a relatively small number of bits, whereas the real world has meaningful state down to the atomic level and a functionally infinite branching factor. The low-dimensional game representation means that vision, one of the most elaborate functions of our brains, has little to no bearing on our ability to play either game.

The natural next targets for computational game-playing, then, are games that require some visual processing to play effectively. In 2013, Bellemare et al. published the Arcade Learning Environment (ALE) [1], which allows programmatic access to an emulator for video games on the Atari 2600 console.

Note that in this paper we mainly follow Mnih et al. [2], and omit some details from that paper for brevity, but attempt to expand on areas that were given less attention in that paper and we found interesting during our implementation.

A. Q-Learning

Parallel to the trend of increasing complexity, focus has shifted from explicitly teaching computers to play games to having the computers learn for themselves from experience. This is in part due to stronger general interest in machine learning, and in part due to the learning approach surpassing the teaching approach.

The subfield of machine learning most relevant to gameplay is called reinforcement learning. Reinforcment learning is the problem of solving Markov Decision Processes (MDPs). An MDP is a task consisting of a set of states, a set of actions, a set of transition probabilities from the set of (state, action) pairs to the set of outcome states, a set of starting states, and a reward function. At each step of the MDP, the agent observes a state, selects an action, and is given a reward signal. The reward signal is what the agent should try to optimize.

A prominent classical reinforcement learning approach is Q-Learning. In Q-Learning, the agent maintains a function that represents the value of a given (state, action) pair: that is, the expected cumulative reward received by taking an action in a state. If the agent has learned a good Q-function, it can then exploit that by taking the action with the maximal Q-value in a given state.

Usually, the Q-function is parameterized by a linear combination of the state features. Its weights are updated by the *Bellman equation*:

$$Q_{new}(s,a) \mathrel{+}= \alpha \left(R(s,a) + \gamma \max_{a'} Q(s^{'}, a^{'}) \right) \quad (1)$$

α is the *learning rate*, which controls how much the weights move at each update. $R(s,a)$ is the reward received after taking action a in state s. γ is the *discount factor*, representing how much less a reward at a future step is worth than a reward at the current step. $s^{'}$ is the state transitioned to after taking action a.

Rather than use a set of linear weights for the Q-function, Deep Q-Learning uses a represents the Q-function via a neural network. We will explore this concept further in Section II-A.

B. Arcade Learning Environment

The ALE provides an interface to game binaries for Python and Java (binaries must be acquired from third-party sites).

After a game is initialized, play proceeds in a loop of the agent sending an action, receiving a reward, and requesting the frame of the current game state.

Frames are given as 210×160 pixel images with 128 colors. To reduce the computational load, Mnih et al. [2] downsampled the frame to 110×84 and converted the colors to grayscale. We also adopted this approach. Conveniently, the ALE has an API call to get a grayscale frame instead of RGB.

The actions available to an agent generally include up, down, left, right, and game-specific actions such as jump or shoot. These are encoded as integers by the ALE, so the agent has no semantic knowledge of the actions before it starts playing the game.

II. Methods

A. Deep Q-Learning

As mentioned in Section I-A, Deep Q-Learning uses a neural network to represent the Q-function. This formulation has a couple benefits. First, deep neural networks are capable of learning extremely complex value functions, whereas linearly-parameterized functions are limited by the complexity of the input features. This is important for any game with a large state space, such as those we are exploring here. Second, deep neural networks have been the driving force of the recent revolutionary improvements in computer vision. For games with visually-represented states, we can take advantage of those improvements.

There are some differences between the use and updating of a deep Q-network and a linearly-parameterized Q-function.

1) Action Representation: One possible network architecture would be to have a separate Q-network for each possible action. We could then feed a state through each network and select the action with the highest Q-value. This approach would work, but it ignores some possible performance improvements and recent discoveries in neural networks.

Neural networks are computationally expensive to train, and are even more expensive when the number of weights in the network is high. This means that when possible, we should try to construct our architecture so that the number of weights that need to be updated is minimized. One way to do this is to only use one network for all of the actions, and have each action correspond to one of the network's output nodes.

Besides the computational improvements, this takes advantage of the fact that the early-layer processing for each action should be virtually identical, since they all require knowledge of the relevant portions of the game state derived from the visual input. By using shared early layers, we ensure that useful visual processing patterns that would have only been "discovered" by one network in a separate-action architecture are instead shared by all of the actions.

This is similar to the principal that we saw in class where early-layer weights for image classification tasks can usually be reused effectively, even for classification on images that would seem unrelated to those that produced the original weights.

2) Network Feedback: The Bellman equation, discussed in Section I-A, is a *prescriptive* update equation; that is, it gives the steps that should be taken to update the Q-function. When we have a linear set of weights, the Bellman equation is sufficient and complete because we know exactly how the outputs will change with any alteration of the weights.

In contrast, with a Q-network, we need a *descriptive* update formulation. Since the backpropagation equations for neural networks are established and widely available, we need to *tell the network what it should produce* and then train it via backpropagation to match that output.

Mnih et al. [2] found a way to derive an error value for backpropagation given a reward and a transition:

$$Err(r,s,a,s^{'}) = r + \gamma \max_{a'} Q(s^{'},a^{'}) - Q(s,a) \qquad (2)$$

This still uses the Bellman contraction from a reward and future Q-value to figure out how "wrong" the current Q-value is.

B. Network Architecture

We mostly follow the architecture presented in [2], with some minor alterations.

1) Downsampling: As mentioned previously, they downsampled the input frames to 110×84. Furthermore, due to their reliance on a specific GPU convolution implementation that required square inputs, they truncated the sides of each downsampled frame so that the resulting output was 84×84. With modern neural network libraries, this is unnecessary, so we avoid the truncation. However, to make the modular arithmetic work nicely, we downsample to 108×84 rather than 110×84.

2) Frame Stacking: To reduce the number of network operations per game step, they stacked four downsampled frames simultaneously and used that as the input to the network. This means that the network only is called to get a new action every four game steps, and the agent simply repeats the previous action in between. We adopt this strategy.

3) Convolutional Layers: There are two convolutional layers in the Q-network: first, a layer of 16 8×8 convolution patches, followed by a layer of 32 4×4 convolution patches. Each uses TensorFlow's rectified linear unit activation function (tf.nn.relu).

4) Fully-Connected: Following the convolutional layers is a fully-connected layer with 128 units, also using the rectified linear activation function. The fully-connected layer is connected to the output layer, which has a number of nodes corresponding to the number of actions available in the given game.

C. Training

1) Experience Replay: Classical Q-Learning updates the Q-function with the last transition seen by the MDP. This would probably work in the deep Q-Learning paradigm, but the authors in [2] found what is likely an improvement. Instead of throwing away the transitions after an update is performed, they store a large number of the most recent transitions. At each update step, they sample from this memory and update the network using the sampled transitions. The main benefit of this is that the network does not deviate too much when the state goes in one direction for an extended period of time.

We discovered that each recorded transition requires quite a bit of data: the prior frame stack, the next frame stack, the action taken, and the average reward. Just storing the Q-values is insufficient, because the network update is supposed to be done with the error from the current weights, not a previous set of weights. Additionally, we have to store the action taken, because the action that maximizes the Q-value may have changed since the transition was recorded.

D. Data Structures

There are two fundamental data structures used in the construction of the Voronoi diagram.

1) Beach Line: The first data structure represents the beach line. Optimally, this would be implemented as a balanced binary search tree such as an AVL tree or red-black tree. In this tree, leaf nodes represent obstacles and the internal nodes represent the intersections between arcs [3].

In the interest of ease of implemenation, we instead use a doubly-linked list to store the beach line. Nodes in the list contain obstacles, and two adjacent nodes have pointers to each other and to the growing Voronoi edge between them. Using a list instead of a search tree increases the complexity of locating the arc that a new obstacle falls under from $O(\log n)$ to $O(n)$.

In this setup, a node has two adjacent edges, each with an intersection at its endpoint. When a node is first encountered by the sweepline, its arc is a vertical line, and so only intersects the beach line once. However, the equation for intersections in Section **??** returns two copies of this intersection. This allows us to create the arcs without having to handle an additional case.

An obstacle can have multiple nodes associated with it. Each time the obstacle's parabola is split by an obstacle event, the single node containing that segment of the arc splits into two. This corresponds to the obstacle's Voronoi region gaining an incident edge.

When a circle event occurs, the corresponding obstacle's node is removed from the list and its neighbors are joined together, forming an intersection on the Voronoi diagram and a new edge leading out of it.

2) Event Queue: The second data structure is a priority queue containing the events, which we refer to as the *event queue*. We implement this using a binary heap, Python's *heapq*. Initially, the event queue contains all of the obstacles minus duplicate obstacles, beginning with the obstacle with maximal y-value. Ordering ties are broken by placing the event with the lower x-value first.

3) Interactions: There is some overlap between the responsibilities of the beach line and the event queue. Both of them keep track of circle events: each node in the beach line has an associated circle event, if any, and the event queue contains the active circle events in addition to the obstacle events. After each event, the nearby nodes in the beach line are examined for circle event updates. This includes both adding potential new circle events if a node was added to the beach line, and removing circle events if a new obstacle falls inside the circle. If a circle event is removed from the beach line, it also is removed from the event queue. This is a costly operation because lookup in a heap costs $O(n)$.

E. Nearest Neighbors

Given a complete and robust Voronoi diagram, KNN classification can be done in a similar manner to a breadth-first search. To classify a test point, the first step is to locate the Voronoi region that it resides in. The obstacle corresponding to that region is the nearest neighbor. If $k > 1$, the obstacles in regions adjacent to the nearest neighbor are added to a list. The second nearest neighbor will be the closest obstacle in the list. Additional neighbors can be added in a similar manner.

Classification in this manner requires that edges have knowledge of the obstacles on both sides of them, which our Voronoi implementation does not. We output a list of edges and a dictionary mapping each obstacle to the edges surrounding its region.

We modify the nearest-neighbor implementation as follows: after the nearest neighbor is found, it is removed from the list of obstacles and the Voronoi diagram is re-calculated. The second nearest neighbor is now the obstacle for the region that the test point resides in. This can be repeated for the k nearest neighbors. This impacts the computational complexity of the classification, as creating k Voronoi diagrams increases the runtime by a factor of k.

The standard Voronoi diagram has at its perimeter a number of edges that extend infinitely in the plane. It is difficult to efficiently calculate whether a point lies in one of these semi-bounded regions. Because of this, we take a post-processing step for the Voronoi diagram that imposes a bounding box around the obstacles; all the obstacles and intersections lie within the box, and any test points for classification must as well.

An additional post-processing step is to sort the Voronoi regions by the minimum value of y of any point in the region. This allows us to do a binary search to find the region with the highest y that is not entirely higher than the test point. From that point, we proceed linearly through the regions in order of decreasing highest y, checking whether the test point belongs in that region, until the correct region is found. This heuristic improves the runtime of the search for the region that the point falls into once we have the diagram, as all regions with higher minimum y can be ignored. The sorting step itself negates at least some of the runtime improvements, but in a better classification scheme (search within a diagram rather than re-calculating the diagram) it would likely be worth the one-time costs.

III. Results

A. Voronoi Diagrams

shows the results of creating a Voronoi diagram with fifty point obstacles with x and y coordinates randomly selected from a uniform distribution between 0 and 10. While we did not analytically verify the results, the output looks intuitively correct. Unfortunately, there is a latent bug in our implementation that causes an error with probability proportional to the number of obstacles, so this is close to the upper limit of the problem size that we can currently generate results for.

B. Nearest Neighbors

As a result of the aforementioned issues, testing the nearest-neighbors classification on any real dataset is not feasible; training generally requires significantly more than fifty input points. However, we are able to produce the k nearest neighbors to an input point, for values of k up to about 10. This means that the theoretical challenges are mostly handled. Creating a more robust classifier would be mostly a matter of scaling and some amount of debugging. As mentioned in Section II-E, the algorithm that we used to find the neighbors was fairly inefficient, but on such a small dataset this did not impose any noticeable runtime degradation.

IV. Discussion

Our results are generally correct, but we wish that we could have implemented the most efficient algorithms and data structures for the diagram construction and KNN classification. During construction, we would prefer to have used a balanced binary tree rather than a doubly-linked list for the beach line. In the classification step, theoretical complexity improvements would have to be taken from modified data structures created during the diagram construction. If the edges list had included the vertices at either end of the edge, we could have made improvements in the time to find the region that a point belongs to. If the edges list had included the obstacles for the regions incident to the edge, we could have done an exhaustive search starting from the nearest neighbor rather than re-creating the Voronoi diagram. These improvements are theoretically simple to implement but we did not consider them during implementation of the Sweepline algorithm and lacked sufficient time to modify it at the end.

V. Conclusion

We successfully implemented a Voronoi diagram algorithm for point obstacles in two dimensions. While there exist easier Voronoi algorithms to implement, the Sweepline algorithm was interesting geometrically and promises good performance results. We used the completed Voronoi diagram to create a k-Nearest-Neighbors classifier that runs correctly on toy datasets.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 06 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] A. Miu, "Voronoi diagrams," 2001, http://nms.csail.mit.edu/~aklmiu/6.838/L7.pdf.

[4] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, no. 1-4, pp. 153–174, 1987.

[5] K. Schaal, "HAVOC - here's another Voronoi code," 2013, diploma thesis. http://www.kmschaal.de/Diplomarbeit_KevinSchaal.pdf.

[6] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.