Snake Game Using Reinforcement Learning

Yashshree Patil
yashshree.patil@rutgers.edu

Abstract

To build an AI bot that can learn how to play the game Snake from scratch. To achieve this, I will be using the Deep Reinforcement Learning algorithm (a field of Unsupervised Machine Learning). The algorithm which will be used is Deep Q Learning, it will focus on giving the system parameters related to the state of the snake and a positive and negative reward based on its action. No rules about the game will be given to the bot and the bot has no information on what it needs to do at the start. The goal for the system is to figure it out and elaborate a strategy to maximize the score/ reward. Further, we can use different strategies such as Store Long term memory, Random actions, use pixels and Convolutional Neural Networks in the state space, etc. to improve the algorithm.

1. Motivation

There is no fun and engagement when the agents in our games outsmart human players. Take an example of a small tic-tac-toe game, the agent is aware of all the possible conditions and rules and Hence, the match is either going to end up in a draw or the agent will always win[1]. This takes out all the fun and uncertainty of the game. Therefore, we do not want to push the limit of our ML bot, as we usually do in different fields. Here the opponent needs to be imperfect, imitating a human-like behavior. One way to involve this human-like behavior in a computer agent is by Reinforcement Learning. Ever since I watched the Netflix Documentary AlphaGo, I have been Fascinated by Reinforcement Learning and as a beginner to Reinforcement Learning Algorithms, I wanted to start with a single-player game and snake being one of the popular games of our childhood I decided to start with the Snake game using Reinforcement Learning.

2. Introduction

In this implementation of the Snake game using Reinforcement Learning, I am going to use Deep Q Learning (a Combination of Deep Learning and Reinforcement Learning) instead of the Traditional supervised Machine Learning approach because traditional ML algorithm needs to be trained with an input state and target label but in this example, we do not know what is the best action to take at each step of the game. In Reinforcement Learning we pass a reward, positive or negative, and depending on the action the system took, the algorithm needs to learn what actions can maximize the reward, and which need to be avoided. By using the states as the input, values for actions as the output, and the rewards for adjusting the weights in the right direction, the agent learns to predict the best action for a given state.

3. Related Work

Alpha Go, The First computer program to defeat a professional human Go player. Go is known as the most challenging classical game for artificial intelligence because of its complexity. Standard AI methods, which test all possible moves and positions using a search tree, cannot handle the sheer number of possible Go moves or evaluate the strength of each possible board position. Hence, Reinforcement Learning was used, a computer program that combines advanced search trees with deep neural networks. These neural networks take the description of the board as an input and process it through several different network layers containing millions of neuron-like connections, Playing the GO Game thousands of times with itself, each time learning from its mistakes.[2]

ALE (Arcade Learning Environment) games using DRL. For control tasks like playing games, researchers have traditionally used handcrafted-features that require a lot of human effort. Hence, Neural networks are chosen over the traditional method to extract useful visual features from the high dimensional input, such as a video game screen, to learn near-optimal value function in such control tasks.[3]

Deep reinforcement Learning with various methods(DQL, UNREAL, IQN, Rainbow etc.)[4] in Video Games(Single-agent, Multi-agent)[5]

Other Problem statements that are closely related to the snake game problem are PAC-MAN, Atari, and Doom Game using Reinforcement Learning.

3.1. PAC MAN using Reinforcement Learning

Various reinforcement learning methods on the classical game Pacman; Studied and compared Q-learning, approximate Q-learning, and Deep Q-learning based on the total rewards and win-rate. While Q-learning has been proved to be quite effective on a small grid, it becomes inefficient to find the optimal policy in large grid-layouts. In approximate Q-learning, They handcraft 'intelligent' features to feed into the game. However, more powerfully, Deep Q-learning (DQL) can implicitly 'extract' important features, and interpolate Q-values of enormous state-action pairs without consulting large data tables. The main purpose of the project was to investigate the effectiveness of Deep Q-learning based on the context of the Pacman game having Q-learning and Approximate Q-learning as baselines [6].

3.2. DOOM using Deep Reinforcement Learning

DOOM game played using Deep-Q-Learning
The Deep Q Neural Network used in the DOOM game takes a stack of four frames as an input. These pass through its network, and output a vector of Q-values for each action possible in the given state. The biggest Q-value is taken of this vector to find the best action.
In the beginning, the agent does badly. But over time, it begins to associate frames (states) with the best actions to do.
2 processes happening in the algorithm are:
- Environment is sampled where the actions are performed and store the observed experiences tuples in a replay memory.
- Select the small batch of tuple random and learn from it using a gradient descent update step [7].
Q value for a given state and action is updated using the Bellman equation:

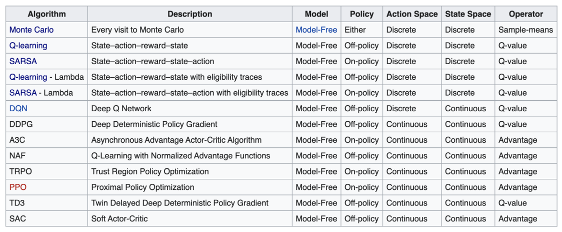
$$Q(s,a) = \sum_{s'} P_{ss'} \left[R_{ss'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

Bellman equation

3.3. ATARI games using Deep Reinforcement Learning

They Presented the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. They applied their method(model) to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. And found that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them[5].

Reinforcement learning algorithms



Algorithm	Description	Model	Policy	Action Space	State Space	Operator
Monte Carlo	Every visit to Monte Carlo	Model-Free	Either	Discrete	Discrete	Sample-means
Q-learning	State–action–reward–state	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA	State–action–reward–state–action	Model-Free	On-policy	Discrete	Discrete	Q-value
Q-learning - Lambda	State–action–reward–state with eligibility traces	Model-Free	Off-policy	Discrete	Discrete	Q-value
SARSA - Lambda	State–action–reward–state–action with eligibility traces	Model-Free	On-policy	Discrete	Discrete	Q-value
DQN	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	On-policy	Continuous	Continuous	Advantage
NAF	Q-Learning with Normalized Advantage Functions	Model-Free	Off-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Model-Free	Off-policy	Continuous	Continuous	Advantage

Reinforcement learning algorithms
A rough comparison of the algorithms can be drawn by the Model, Policy, Action Space, State Space, and Operator they use. Some of the Algorithms are:

1. Monte Carlo methods can be used in an algorithm that mimics policy iteration. Policy iteration consists of two steps: policy evaluation and policy improvement.

Monte Carlo is used in the policy evaluation step. In this step, given a stationary, deterministic policy π , the goal is to compute the function values $Q^{\pi}(s,a)$ (or a good approximation to them) for all state-action pairs (s,a) [4].

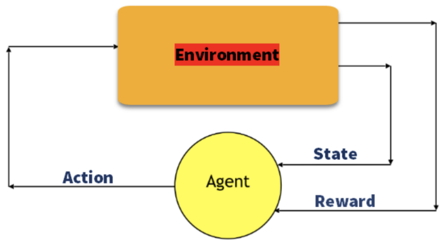
2. State–action–reward–state–action (SARSA) is an algorithm for learning a Markov decision process policy, used in the reinforcement learning area of machine learning. A SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm. The Q value for a state-action is updated by an error, adjusted by the learning rate alpha. Q values represent the possible reward received in the next time

step for taking action in a state s, plus the discounted future reward received from the next state-action observation [8].

3. DNQ(Deep Reinforcement Learning): This approach extends reinforcement learning by using a deep neural network and without explicitly designing the state space[4].

4. Technical details

Technical details of Snake game using Reinforcement Learning consists of State(Current Environment), Creating Environment and the agent, Reward and action, Deep Neural Network, Algorithm, GUI(Graphical User Interface) and Python modules used:



Reinforcement Learning Block Diagram

4.1. Model

The model used to solve the problem statement is Deep Q Learning System.
Deep Q Learning is a Combination of Deep Learning (Neural networks) and Reinforcement Learning.

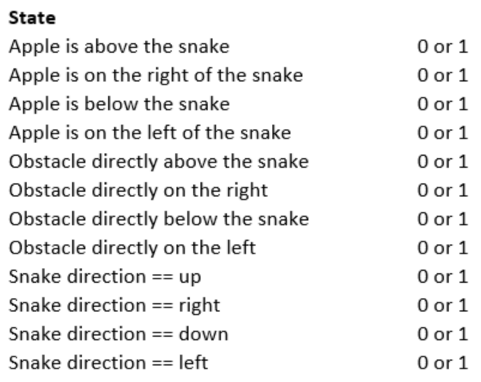
Why Neural Network method works for the snake game problem:

For the game of snake, I have used Replay experiences also known as memory. where all the state space of the game is stored. It is important to make memory storage for the agent to learn from it. When state and action at each step are stored in q-table it gets quite large. Q-table is also known as the state - action table.
When we have a very large number of state-action pairs, it is not feasible to store every Q-factor separately.
• Then, it makes sense to store the Q-factors for a given action within one neural network.
• When a Q-factor is needed, it is fetched from its neural network.
• When a Q-factor is to be updated, the new Q-factor is used to update the neural network itself.

While it's manageable to create and use a Q-table for simple environments, it's quite difficult with some real-life environments. The number of actions and states in a real-life environment can be thousands, making it extremely inefficient to manage Q-values in a table.
This is where we can use neural networks to predict q-values for actions in a given state instead of using a table. Instead of initializing and updating a q-table in the q-learning process, we'll initialize and train a neural network model.

4.2. State

For the Representation of state, an array containing 12 Boolean variables can be taken. Which takes into account:
- if there's an immediate danger in the snake's proximity (above, below, on the right, or the left of the snake).
- if the snake is moving up, down, left, or right.
- if the apple/food is above, below, on the left, or the right.



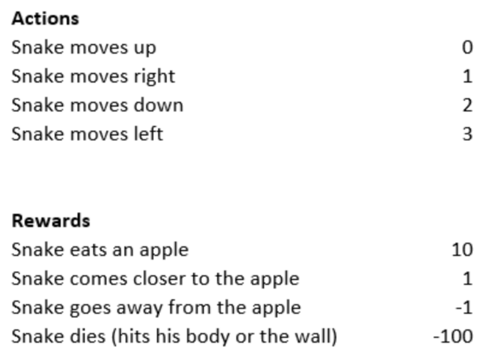
State	
Apple is above the snake	0 or 1
Apple is on the right of the snake	0 or 1
Apple is below the snake	0 or 1
Apple is on the left of the snake	0 or 1
Obstacle directly above the snake	0 or 1
Obstacle directly on the right	0 or 1
Obstacle directly below the snake	0 or 1
Obstacle directly on the left	0 or 1
Snake direction == up	0 or 1
Snake direction == right	0 or 1
Snake direction == down	0 or 1
Snake direction == left	0 or 1

Figure 1: State and its Boolean values 0 for False, 1 for True

4.3. Reward and action

Reward There are multiple solutions in defining the state space and rewards, and one will work better than the other. For now, let's try the following.
If the snake grabs an apple/food, give a reward of 10. If the snake dies, the reward is -100. To help the agent, give a reward of 1 if the snake comes closer to the apple, and a reward of -1 if the snake moves away from the apple. The agent will try to maximize the reward and will take action accordingly.

Action Defining actions is easy. The agent can choose between going up, right, down, or left. To maximize the reward, the agent will take optimal action.



Actions	
Snake moves up	0
Snake moves right	1
Snake moves down	2
Snake moves left	3

Rewards	
Snake eats an apple	10
Snake comes closer to the apple	1
Snake goes away from the apple	-1
Snake dies (hits his body or the wall)	-100

Figure 2: Actions and Rewards

4.4. Creating Environment and the agent

By adding some methods to the Snake program, it's possible to create a Reinforcement Learning environment.
The added methods are
- reset(self)
- step(self, action)
- get_state(self)
Besides this, it's necessary to calculate the reward every time the agent takes a step
- run_game(self)

The agent uses a Deep Q Network to find the best actions.
The Starting parameters are:

epsilon: sets the level of exploration and decreases(decays) over time
param['epsilon'] = 1
param['epsilon_min'] = .01
param['epsilon_decay'] = .995

gamma : value immediate (gamma=0) or future (gamma=1) rewards
param['gamma'] = .95

batch size: the batch size is needed for replaying previous experiences
param['batch_size'] = 500

4.5. Deep Neural Network

The Deep Neural Network exists of layers with nodes. The first layer is the input layer. Then the hidden layers transform the data with weights and activation functions. The last layer is the output layer, where the target is predicted. By adjusting the weights the network can learn patterns and improve its predictions. By using the states as the input, values for actions as the output, and the rewards for adjusting the weights in the right direction, the agent learns to predict the best action for a given state.

Neural network parameters :
param['learning_rate'] = 0.00025
param['layer_sizes'] = [128, 128, 128]

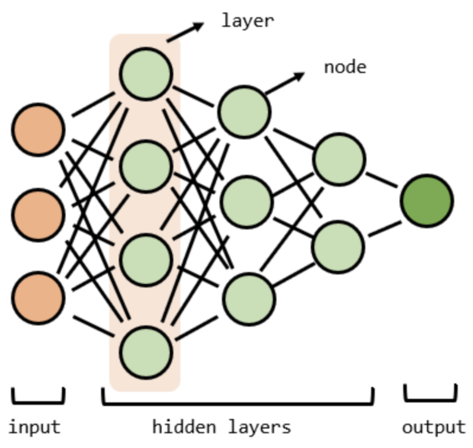


Figure 3: Deep neural network

4.6. Algorithm

On a general level, the algorithm works as follow:
1. The game starts with a random state. The system gets the current state parameters.
2. Based on the state, it executes an action, randomly or based on its neural network. During the first phase of the training, the system often chooses random actions to maximize exploration. Later on, the exploration rate (epsilon) decays and the system relies more and more on its neural network.
3. When the AI chooses and acts, the system updates the state, all states are stored in replay previous experiences are also known as memory. This data is later sampled to train the neural network.
4. Last two operations are repeated until a certain condition is met (If the snake dies by biting itself or colliding on the wall).

4.7. GUI

The game is built in Python with the turtle module (Feature of python) for Graphical User Interface.
Food/apple is circular with red color.
The snake's body is in black combined with multiple square block sequence.

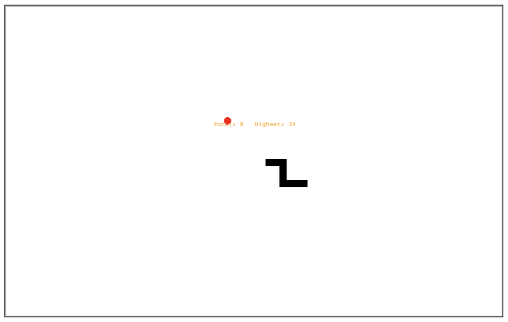


Figure 4: Snake and Food at a random instance of a game

4.8. Python modules required

Major python modules required are written as follow:
Keras: Used for working with neural networks, its an open-source library that provides a python interface for artificial neural networks
This module is used to create and predict outputs of neural networks. Sequential, dense, and adam are the key methods used from Keras module
Turtle: Used to build Graphics for programming
gym: a toolkit for developing and comparing Reinforcement Learning algorithms.

5. Results

In the first game, the agent has no clue what it's doing: it focuses more on exploring the environment, it easily collides on the wall than eat the apple.
Rewards are calculated at each state and the agent is in the learning process in the initial episodes(games).
Around episode 10-15:
The agent learns: it does not take the shortest path but finds his way to the apples.
Around episode 30-35: The agent avoids the body of the snake and finds a fast(shortest) way to the apples, after playing 30 games.

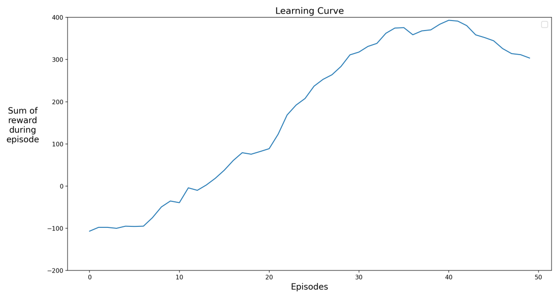


Figure 5: Graph of reward points vs Number of episodes(games played) (Batch size: 500, Total Number of episodes: 50)

From the graph, we can see that initially, the agent is scoring a negative reward as it focuses more on exploration and hence dies easily, but as the number of episodes increases the reward is increasing linearly with it.
Reward reaches its peak value around 40-42 episode (game) and later there is a short reduction/degradation in the reward points.
There is a problem with the solution
The agent does not learn to avoid enclosing. It only learns to avoid obstacles directly surrounding the snake's head, but it can't see the whole game. So the agent will enclose itself and die, especially when the snake is longer.
This can be because of the number of episodes which we have limited to 50 and the batch size of replay experiences which is 500.
Therefore, Let's look at the Evaluation techniques and whether we can achieve better performance in any other way?

6. Evaluation

6.1. Playing with the state space

To check how the state parameters affect the behavior of the agent: The agent learns to play snake (with experience replay), but maybe it's possible to change the state space and achieve similar or better performance.
1. State-space 'no direction': don't give the agent the direction the snake is going.
2. State-space 'coordinates': replace the location of the apple (up, right, down, and/or left) with the coordinates of the apple (x, y) and the snake (x, y). The coordinates are scaled between 0 and 1.
3. State-space 'direction 0 or 1': the original state space.
4. State-space 'only walls': don't tell the agent when the body is up, right, down, or left, only tell it if there's a wall.

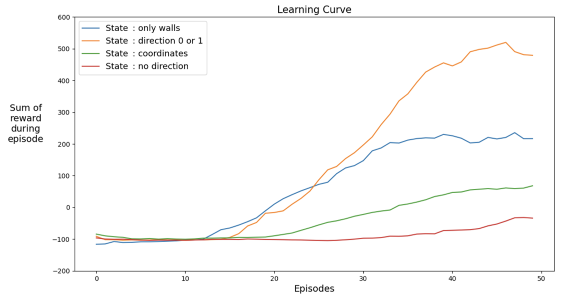


Figure 6: graph with the performance using the different state spaces

It is clear from the graph that using the state space with the directions (the original state space) learns fast and achieves the highest return. But the state space using the coordinates is improving, and maybe it can reach the same performance when it trains longer. A reason for the slow learning might be the number of possible states: 20^4*2^4*4 = 1,024,000 different states are possible (the snake canvas is 20*20 steps, there are 2^4 options for obstacles and 4 options for the current direction). For the original state space the number of possible states is equal to $3^2*2^4*4 = 576$ (3 options each for above/below and left/right: snake cannot move in backward direction hence, 3 options). 576 is more than 1,700 times smaller than 1,024,000. This influences the learning process.

6.2. Playing with the rewards

Is there a better way to program the Rewards?
we have current rewards as:
Snake eats the apple (10)
Snake gets closer to the apple (1)
Snake goes away from the apple (-1)
Snake dies (-100)

1. If we scrap the reward for snake getting closer and away from the apple:
Updated conditions
Snake eats the apple (10)
Snake dies (-100)
This way Snake takes more time to reach the food.

2. If we add a reward for surviving:
Updated conditions: Snake eats the apple (10)
Snake dies (-100)
Snake survives (2)
This way the snake will walk in circles without getting killed and score more rewards just by walking than by eating the apple.

3. If we increase the reward of Snake eats the apple to (100)
Updated Conditions:
Snake eats the apple (100)
Snake gets closer to the apple (1)
Snake goes away from the apple (-1)
Snake dies (-100)
This increases the chance of taking the shortest path and if the apple is body blocked by a snake it ends up biting itself and dies.

Experience Replay
The reason behind the learning of the agent (only needs 30 games) is experience replay. In experience replay the agent stores previous experiences and uses these experiences to learn faster. At every normal step, several replay steps (batch_size parameter) is performed. This works so well for Snake because, given the same state-action pair, there is low variance in reward and next state.

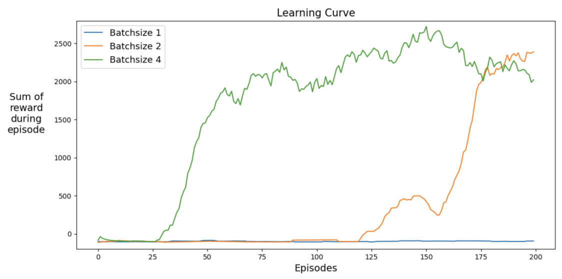


Figure 7: Train 200 Episodes/games with 3 different batch sizes.

It seems indeed that experience replay helps a lot, at least for these parameters, rewards, and this state space. How many replay steps per step are necessary? To answer this question we can play with the batch_size parameter (mentioned in the section Creating the Environment and the Agent). In the original experiment, the value of batch_size was 500.

Even with batch size 2, the agent learns to play the game. In the graph you can see the impact of increasing the batch size, the same performance is reached more than 100 games earlier if batch size 4 is used instead of batch size 2.

7. Conclusion

The agent learns to play snake and achieves a high score (number of apples eaten) between 30 and 40 after playing 50 games. That is comparatively better than a random agent. The maximum score for this game is 400. Why does not the agent achieve a score of more than that? As mentioned in the results section, the agent does not learn to avoid enclosing. The agent learns to avoid obstacles directly surrounding the snake's head, but it can't see the whole game. So the agent will enclose itself and die, especially when the snake is longer.

One way to solve this problem is to use pixels and Convolutional Neural Networks in the state space. Then the agent can see the whole game, instead of just nearby obstacles. It can learn to recognize the places it should go to avoid enclosing and get the maximum score.

The other way to improve performance/reward is to increase the number of total episodes/games played and reduce the batch size of the Experience Replay to its optimal value.
Hence, by running this program on multiple values of a total number of episodes and different batch sizes. Performance of the Reinforcement Learning Snake Program can be increased by a small margin.

We can see an increased reward in the following Figure which has a batch size of 50 and the total number of episodes as 100

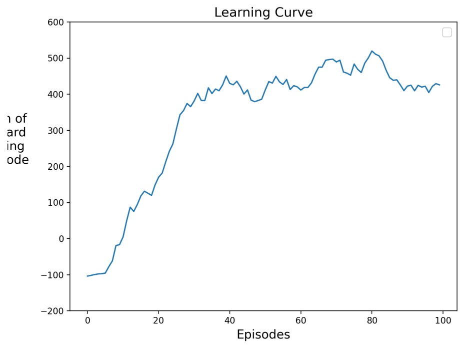


Figure 8: Graph of Reward vs Number of episodes (Batch size: 50 and Total Number of episodes: 100)

References

[1] Devanshi, “Tic Tac Toe AI Vs User.” See also URL https://medium.com/analytics-vidhya/tic-tac-toe-ai-vs-user-1a2903621737.

[2] D. Mind, “Alpha GO.” See also URL https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[3] A. V. Debidatta Dwibedi, “Playing Games with Deep Reinforcement Learning,” Master’s thesis. See also URL https://debidatta.github.io/assets/10701final.pdf.

[4] Wikipedia, “Reinforcement Learning.” See nalso URL https://en.wikipedia.org/wiki/Reinforcement$_learning$.

[5] Y. Z. Kun Shao, Zhentao Tang, “A Survey of Deep Reinforcement Learning in Video Games,” Master’s thesis. See also URL https://arxiv.org/pdf/1912.10944.pdf.

[6] J. A. Abeynaya Gnanasekaran, Jordi Feliu Faba, “Reinforcement Learning in Pacman.” See nalso URL http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf.

[7] T. Simonini, “An introduction to Deep Q-Learning: let’s play Doom.” See nalso URL https://www.freecodecamp.org/news/an-introduction-to-deep -q-learning-lets-play-doom-54d02d8017d8.

[8] Wikipedia, “SARSA.” See nalso URL https://en.wikipedia.org/wiki/State–action–reward–state–action.