Variations of Intelligent Agents to Solve Pacman

Stephen Hung
Computer Science
California Polytechnic State University
San Luis Obispo, United States
syhung@calpoly.edu

Sherry Lin
Computer Science
California Polytechnic State University
San Luis Obispo, United States
slin35@calpoly.edu

Tamim Fatahi
Computer Science
California Polytechnic State University
San Luis Obispo, United States
tfatahi@calpoly.edu

Wesly Ortega
Computer Science
California Polytechnic State University
San Luis Obispo, United States
wortega@calpoly.edu

Tantum Nilkaew
Computer Science
California Polytechnic State University
San Luis Obispo, United States
tnilkaew@calpoly.edu

I. Description

The goal of this project is to develop AIs playing the game PacMan autonomously. We have developed three Pacman models by applying different AI techniques including Q-learning, Perceptron-based classification, and Neuroevolution of Augmenting Topologies (NEAT) algorithm.

A. Features

Our project consists of three Pacman agents that uses AI in order to solve the game. To reduce repeated work, the UI and environment for our Pacman game are based off of the University of California: Berkeley's CS188 Pacman project [1]. The environment for our Pacman game consists of the following entities:

- **Pacman:** The agent that is either computer or manually controlled and aims to collect all of the dots while avoiding Ghosts.
- **Ghosts:** Agents that are computer controlled and exist only to eat the Pacman Agent and thereby end the game.
- **Collectable points (dots):** Little dots that give Pacman Agents points and there are big dots which allow the Pacman Agent to eat Ghosts.
- **Walls:** Solid blocks that prevent Ghosts or Pacman Agents from walking through.

Our Agents have the ability to know the locations of all entities but will have to learn the consequences and results of interaction with entities. Consequences and results that the Pacman Agent logs include points earned from eating dots, time spent playing the game, and dying from ghosts.

B. Relevance for AI

This project used different AI techniques to develop models playing Pac-Man. With Q-learning, the Pacman agent is able to learn the optimal policies for choosing the next action from its past experiences. With a modified Perceptron-based classifier, the Pacman agent is capable of learning from some pre-trained game data and apply the acquired action selection mechanism to new, similar situations. With NEAT algorithm, a series of Pacman agents are free to learn how to move through multiple generations in an evolution-like network, and only the ones outperform reaching the goals survive.

II. Background and Design

Since we have three different agents we have three different designs for the agents and will introduce the different designs and their backgrounds in the following sections.

A. Q-Learning Agent

Q-Learning is a values-based reinforcement learning algorithm to find the best action to take given the current state. The Q in Q-Learning stands for quality where quality is how useful an action is in gaining a future reward. Reinforcement Learning algorithms can be either model-based or model-free algorithms where model-based algorithms use a transition and reward function to estimate the optimal policy while model-free algorithms use an algorithm that estimates the optimal policy without a transition or reward function [3].

Contrary to some different reinforcement algorithms, Q-Learning finds the optimal policy independently of any actions that the agents takes. For Q-Learning, a Q-Table is built to calculate the maximum expected future rewards for each action at a given state thereby helping the algorithm identify which action given a state is optimal. In order to build this Q-table, the Bellman equation, seen in Fig. 1, is used [6].

The Bellman equation looks at possible rewards for each action at a given state and given a certain learning rate and the previous state, maximizes the possible value.

A Q-Table is m x n where n is the number of actions and m is the number of states. The algorithm takes each action possible for a given state and performs one of the actions based on the Q-Table. However when starting the algorithm, all Q-Values are 0 and thus a random action will be taken.

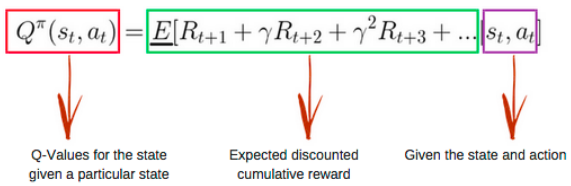

Fig. 1. The Bellman Equation [5]

Later on when Q-Values are non-zero, agents may still take random actions as part of the epsilon greedy strategy where for some probability $0 < \epsilon < 1$, the agent performs the random action. The usage of the epsilon greedy strategy allows the agent to explore the environment and become more confident in estimating Q-Values for given actions and states.

Upon performing the action, the agent observes the outcome and reward such as any dots eaten or if it has been eaten by a ghost and acts to update the Q-Table. As seen in Fig. 2, the Q-Value for a given state and action taken returns the expected future reward of that action.

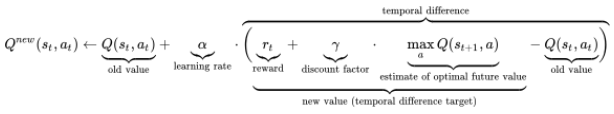

Fig. 2. Updating the Q-Value for a given State and Action [2]

This process is repeatedly done in a training cycle of games that allows the Q-Table to be fully confident in its Q-Values. Upon finishing the training, the epsilon greedy strategy and learning rate, α, are turned off which allows the Q-Learning algorithm to fully utilize the trained Q-Table to theoretically choose correct optimal actions.

1) Approximate Q-Learning: While Q-Learning is able to solve certain games with a small amount of states and actions like Tic-Tac-Toe, given enough states and actions, computational resources become unwieldy and accuracy is dropped significantly for given states and actions. In order to provide a solution to this, we can use a reward function as a value approximation which allows Pacman to generalize for previously unknown states. Our reward function is learned as a linear combination of features which then allows Q-learning to be updated based on these features and a value estimation over a sum of the state's features. The features we will use as part of our approximate Q-Learning are as follows

- Is a ghost one step away?
- Will the Pacman collide with a Ghost in the next step?
- Will food be eaten in the next step?
- How far is the nearest food?

B. Perceptron-Based Classification

Perceptron is the most basic type of neural network. It is easy to understand the decision-making process in perceptron network with linear function that defines the perceptron output. The network architecture for a single-layer perceptron is shown in Fig. 3. The perceptron takes in a vector of numerical inputs and a vector of numerical weights. Each input and weight pair are multiplied together, and the summation of these products is then passed to an activation function outputting true or false values. Usually, a non-negative value is classified as a true value and vice versa.

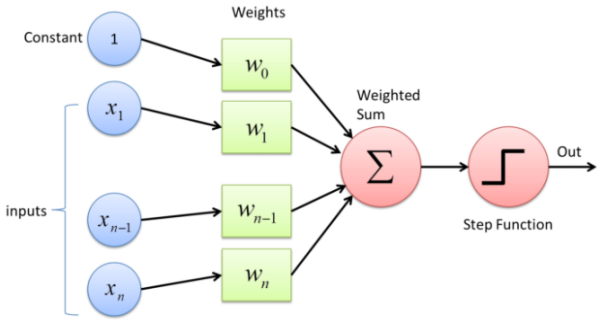

Fig. 3. Perceptron Neural Network Unit [4].

The multi-class perceptron is regarded as a direct extension of the binary perceptron. It keeps a weight vector for each class in the classification categories. The weighted sum is computed for each class accordingly; prediction with the highest weighted sum is taken as the result.

In a basic multi-class perceptron, given a feature list f and a class y, each class can be scored with:

$$score(f, y) = \sum_i f_i w_i^y \tag{1}$$

Then the class the with highest score is chosen as the predicted label for input data.

As for the learning, once the predicted label y' is found, we compare y' to the true label y. If the predicted label does not match the true label, it means that w^y should have scored f higher, and $w^{y'}$ should have scored f lower, in order to prevent the wrong predicted label occurring in the future. The weight vectors is then updated accordingly:

$$\begin{aligned} w^y &= w^y + f \\ w^{y'} &= w^{y'} - f \end{aligned} \tag{2}$$

C. Neuroevolution of Augmenting Topologies (NEAT)

NEAT is a genetic algorithm that attempts to replicate the process of evolution by simulating various agents in a population and are then ranked based on their performance at the given task. Agents with higher performance scores have a higher likelihood of passing down their genes to coming generations, which allows agents to improve over time at the given task.

Agents start with a simple neural network composed of only input neurons, output neurons, and randomly assigned weights and biases for each neuron. At the end of each simulation, called a generation, the best agents are picked and have their networks randomly mutated and passed onto the next generation. Additionally, the original networks are also

passed down in order to prevent making changes that would lead to a reduction of performance in following generations.

III. Implementation

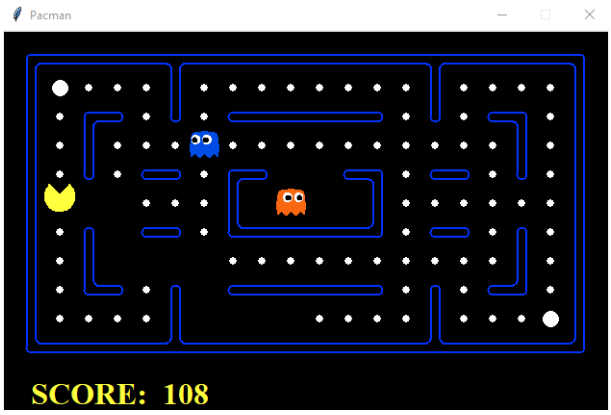

Fig. 4. Pac-Man UI.

We worked with an existing Pac-Man UI from Berkeley's Pacman project [1]. The Pac-Man graphics and the models we developed are written in Python. There are six components in the UI: one Pacman agent, two ghost agents, collectable food (dots), power pellets (energizer), walls, and current score (lower left corner) as shown in Fig. 4. Ghosts would be chasing after the Pacman or fleeing when Pacman has eaten the power pellets. The goal of Pacman is set to consuming as many dots as possible and as soon as possible while staying alive. Provided by the Berkeley's Pacman project is a range of layouts that we used for testing edge cases, for instance $openClassic$ is a wide open map with no internal walls and has both dots and Ghosts.

A. Q-Learning

Using the preexisting UI for Pacman, we created an agent called $ApproximateQAgent$ which would follow our Design for an Approximate Q-Learning Agent. Data about the environment is sent as a GameState object to our Agent and contains information about the state, possible legal actions, and locations of all entities in the Pacman game. Given the GameState, we can save a simplified version of the state to our Q-Table along with the given legal actions. The diagram for our Q-Learning update is as seen in Fig. 5:

In order to run the Approximate Q-Learning Agent, run the following command

"python pacman.py -p ApproximateQAgent -x X -n N -l L"

Where X is the number of training runs, N is the total number of runs (i.e training + test runs), and L is the layout you want to run the Pacman Agent in.

B. Perceptron-Based Classification

In the classification model, the perceptron-based classifier takes in a list of features extracted with a method we will describe in details later in the section, a weight vector of each

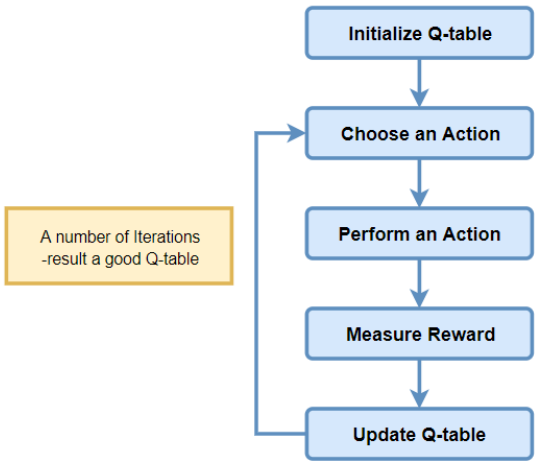

Fig. 5. Q-Learning Flow Chart [2]

possible action for Pacman labeled as "stop", "west", "east", "north", and "south", and labels from pre-trained data.

Before the training, all weights are initialized to 0. Given a Pacman's current state, for each possible next action, a score is computed as:

$$score(s, a) = w * f(s, a) \tag{3}$$

Then the predicted label a' is retrieved from the one with the highest score:

$$a' = arg \max_{a''} score(s, a'') \tag{4}$$

Similar to the standard multi-class perceptron classifier, the weights are updated for both predicted and actual labels when the two do not match. Instead of modifying two separate weight vectors on each update, both updates are done on the shared weights as shown in (2) with a and a'.

When creating the feature extractor, it's critical to consider each input and its calculated weight, as shown in Fig. 3. Four features have ranks based on their priority and distance: ghost, wall, energizer, and collectible points. Each element has a hard-coded constant that is then multiplied by its distance away from Pacman. The ghost will receive the highest priority of all components due to being the only one to end the game. Walls are second since they can cause Pacman to be immobile in specific directions. The third priority is the energizer. If the ghosts are closer to Pacman and both the energizer and collectible points are equidistant, Pacman will opt to go for the energizer to be safe. Collectible points are the last priority. Even though food is needed to win the game, they cause no harm to Pacman and, therefore, will be weighted less.

Using equation (3) with a, the weight and input multiplied together calculates each entity's score. That will then translate to the summation function (1), which will sum all the entity scores into one area. Finally, the activation function (4) will retrieve the highest calculated score of all inputs and identify which path Pacman will take.

C. Neuroevolution of Augmenting Topologies (NEAT)

The model was implemented using the NEAT-py library, which allows for the implementation of the genetic algorithm without the need of building everything from scratch.

Agents start with a fully connected neural network composed of 81 input neurons and 4 output neurons. The input neurons represent a 9x9 grid around the Pacman, which is used for Pacman to sense what is happening around him. Every object has a unique weight that is passed in as input to the network.

In order increase the rate at which Pacman learns how to play the game, inputs are passed in into the network in an order that is relative to direction in which Pacman is facing. Cells that are in the direction in which Pacman is facing are passed in first, while cells that are behind Pacman are passed in last.

The 4 output neurons represent the four directions in which Pacman can turn. Since the input grid is rotated as Pacman turns, the directions in which Pacman can turn were also changed. Instead of having Pacman decide between four absolute directions (North, South, East, or West), Pacman is left to choose between going forward, turning back, turning left, and turning right.

The performance of each agent is measured depending on the amount of food that it eats. At first, Pacman were also given points for moving around in an attempt to incentivize moving around the map, but it ended up creating an exploit in which Pacman would move back and forth between the same two cells trying to maximize the points it would get. Scores were also not dependant on the time they took to complete the level. While it makes sense for Pacman that finish a level faster to have higher scores, during training, it just created an extra hurdle that agents could not overcome during training.

IV. Testing

A. Q-Learning

To test our Approximate Q-Learning agent, we ran the agent through a series of different layouts to test which cases worked and which didn't. Each layout addresses a different edge case and we test each with a different amount of training runs to identify the number of training runs needed or if it is impossible for the Q-Learning Agent to solve. The data for our testing can be seen in Appendix A.

For 50 training runs, the Q-Learning Agent was able to solve 3 out of 13 layouts which were

- **contestClassic:** A small grid with 4 dots and 1 ghost
- **openClassic:** A large open map with 1 ghost and no walls.
- **originalClassic:** The original Pacman arcade map

and sometimes performed well in **mediumClassic**, which is a medium sized version of **originalClassic**

For 100 training runs, the Q-Learning Agent was able to solve 5 out of 13 layouts which added the new layout, **trickyClassic**, a map similar to **mediumClassic** but 2 power dots in the ghost spawn. At 100 runs, the Agent was able to perform consistently well in **mediumClassic**.

For 1000 training runs, the Q-Learning Agent was able to solve 7 out of 13 layouts which added the following new layouts:

- **mediumGrid:** A medium sized regular layout in a grid form.
- **smallClassic:** A smaller version of the original classic map.
- **testClassic:** A small vertical oriented map with 8 dots and 1 ghost.

contestClassic: did not perform well in this run but may be due to extenuating factors when training.

For 1500 training runs, the Q-Learning was able to solve 9 out of 13 layouts which added the following new layouts:

- **powerClassic:** A small-medium sized map with alot of power pellets.

For 2000 training runs, the Q-Learning was able to solve the same amount as 1500 (9 out of 13 layouts) but was able to perform better. For instance, on **smallClassic**, the average score for 1500 training runs is 372.6 but at 2000 training runs, the average score was 975.3.

We tested 5000 training runs for layouts which continuously had a 0% winning rate and found that the Pacman Agent was able to solve 1 out of 4 of these harder layouts. The 3 layouts failed are the following:

- **minimaxClassic:** A very small grid with 2 dots and 3 ghosts.
- **smallGrid:** A small map with 2 dots and 1 ghost
- **trappedClassic:** A map with only one hallway, 2 ghosts (one in front of and another behind the Pacman), and 4 dots.

The layout that was solved was **capsuleClassic**, which is a small map with alot of dots, and 3 ghosts.

B. Perceptron-Based Classification

There are 15 recorded games for the training data, 10 recorded games for validation data, and 10 for testing data. All recorded games are properly formatted and labeled to fit the classification model shown in Fig. 6. Have training, validation, and testing datasets are essential for machine learning. Each one partakes in the learning process and are described as follows:

- **Training Set:** Sample dataset used to fit the model.
- **Validation Set:** Sample dataset used to provide an unbiased assessment of a model fit on the training dataset when modifying the hyperparameters. The review will become increasingly bias as the parameters from the validation dataset becomes involved in the model.
- **Testing Set:** Sample dataset used to produce an unbiased evaluation for the final model fit on the training dataset.

To replicate Fig. 6, the command will be: "python dataClassifier.py -c perceptron -d pacman -f -g ContestAgent -t 1000 -s 1000", where $-t$ is the size of the training set and $-s$ is the amount of test data to use.

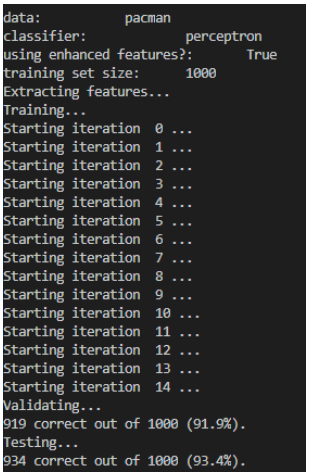

Fig. 6. Validation and testing with 15 iterations of training. The results yielded a 91.9% in validating and 93.4% in testing.

One of the challenges that arose was the limited number of training, validation, and testing datasets. Due to training the perceptron locally, it wasn't easy to get large sums of training sets. If we produced more datasets, the Pacman would have been trained better to work in multiple edge cases rather than the few tested.

C. Neuroevolution of Augmenting Topologies (NEAT)

Even after extensive training, it was not possible to create a Pacman that could both eat all the food while escaping from the ghosts. Agents would randomly turn into the ghosts while running away from them or would ignore them completely. Because of this, agents were then tested without ghosts to see how they performed in a scenario where they didn't have to worry about avoiding ghosts. Agents were tested in layouts of various sizes to see if the developed networks could be applied to different scenarios. Because of their limited vision (a 9x9 grid around them), agents were able to perform better in smaller maps since they would keep track of most of the things around them. In bigger maps, however, agents often lost track of food once they moved too far away. They sometimes were able to wander back to where the food was, but it was not something that occured consistently.

V. Analysis

A. Q-Learning

From the testing results, there was a trend with the Approximate Q-Learning Agent to only complete certain types of layouts. All of the large and most of the medium maps were able to be solved after a large amount of training runs were introduced. However, small maps or maps that force Ghosts and the Pacman Agent to be very close together are impossible for the Q-Learning to solve. Our number of layouts solved improves from 3 out of 13 to 10 out of 13 after moving from 50 to 5000 training runs. From this we can tell that the Approximate Q-Learning agent is able to solve relatively normal maps given enough training runs but trickier maps that require excellent timing and dodging of Ghosts are almost impossible for the Pacman Agent to solve.

This can be seen in **capsuleClassic** which is a small map with a lot of dots and 3 ghosts (seen in Figure 7)

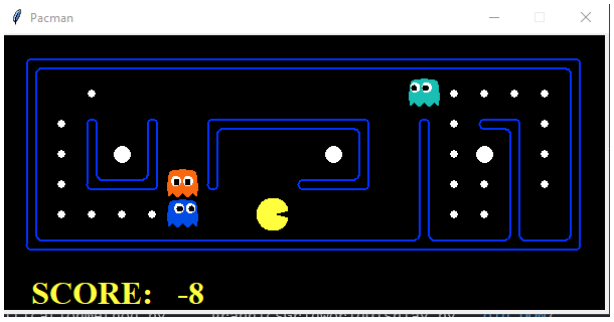

Fig. 7. Screenshot of capsuleClassic layout

This layout is very tricky since some areas are impossible for the Pacman to escape from, such as the rightmost side where there is only one entrance and exit. However, given enough luck and training the Pacman may be able to dodge the Ghost by waiting for the Ghost to walk away.

An example of an impossible layout is **minimaxClassic** which is a very small grid packed with ghosts (seen in Figure 8)

Fig. 8. Screenshot of minimaxClassic layout

As we can see, the Pacman Agent isn't given enough time to solve these layouts since the ghosts are too close-by and eat the Pacman before it can learn a correct route.

1) Difficulties and Future Work: Difficulties that the Q-Learning agent faced include include dealing with the Ghosts better and dealing with smaller and sparser maps. Furthermore, the Pacman doesn't understand that when eating the power dots, the agent can eat the Ghosts and thus doesn't need to dodge them. Future work that we can look into include creating and analyzing different features such as using the power dots.

B. Perceptron-Based Classification

Based on the results from Fig. 6, the goal set to have an accuracy above 90% was achieved. Moving forward, additional training, testing, and validation sets should be created to test for multiple instances of Pacman games. Adopting an environment in Google Collaboratory will allow us to bypass using local memory for the perceptron training that limited the

number of sets created. It will also allow for easier portability for others to utilize since it will be in a controlled environment.

C. Neuroevolution of Augmenting Topologies (NEAT)

Given the low efficacy of the agent, more testing is required in order to generate a model that can successfully complete the game when ghosts are present. Playing around with the weights used to represent each object may be enough, but it is likely that more tweaking will be needed. Moreover, lower performance in bigger maps is something that can be fixed by increasing the vision range. This will also increase the number of generations needed to generate a network that is able to complete the level since the size of the input layer has a quadratic growth, but, given enough time, it should be possible to generate such network.

VI. Conclusion

In this paper, we developed three different agents to solve the Pacman Arcade game and found that the Agents were able to perform well on a decent amount of maps.

References

[1] UC Berkeley CS188 Intro to AI - Materials.

[2] Mar 2021. Page Version ID: 1012624611.

[3] H. Nguyen and H. La. Review of deep reinforcement learning for robot manipulation. pages 590–595, 02 2019.

[4] S. Sharma. What the hell is perceptron?, Oct 2019.

[5] C. Shyalika. A beginners guide to q-learning, Nov 2019.

[6] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Appendix A
Testing Data for Q-Learning Agent

Layout	50 Training Runs		100 Training Runs		1000 Training Runs		1500 Training Runs		2000 Training Runs		5000 Training Runs		Map Description
	Average Score	Win Rate	Average Score	Win Rate	Average Score	Win Rate	Average Score	Win Rate	Average Score	Win Rate	Average Score	Win Rate	
mediumClassic	1334.1 or -429.3	0.9 or 0.0	886.8	0.7	1062.6	0.8	1211.1	0.9	1344.6	1	-	-	Medium version of original map
capsuleClassic	-528	0	-469.8	0	-478.5	0	-519.2	0	-426.3	0	476	0.8	small map, lots of dots, 3 ghosts
contestClassic	892	0.8	536.4	0.6	-318.6	0	920.1	0.8	986.5	0.9	-	-	4 dots, small grid, 1 ghost
mediumGrid	-511	0	-416.3	0.1	527.8	1	526.6	1	526.6	1	-	-	medium sized regular grid
minimaxClassic	-395.5	0.1	-495.8	0	-496.4	0	-548.4	0	-496.3	0	-500.9	0	2 dots, 3 ghosts, very small grid
openClassic	1240.8	1	1241	1	1240.9	1	-	-	-	-	-	-	Large open map, 1 ghost, no walls,
originalClassic	2232.5	0.9	1948.5	0.7	1866.8	0.6	2447.4	1	-	-	-	-	Original Map
powerClassic	-486	0	-471.8	0	-259.9	0	864.1	0.9	483	0.6			Small-medium sized map, lots of power points
smallClassic	-416.6	0	-436.4	0	972.8	1	372.6	0.5	975.3	1			Small version of original map
smallGrid	-510.6	0	-507.5	0	-508.3	0	-508.2	0	-507.9	0	-509.9	0	small map, 2 dots, 1 ghost
testClassic	-514.1	0	-504.7	0	561.8	1	562.6	1	562.6	1			vertical small map, 8 dots, 1 ghost
trappedClassic	-502.2	0	-501.7	0	-501.6	0	-502.2	0	-502.2	0			single route, 2 ghosts (front and back), 4 dots
trickyClassic	-452	0	1149.6	0.8	511.5	0.5	953.6	0.7	1361	0.9			similar to medium classic but 2 power dots in ghost spawn