Pac-Man AI

Final Project by Team Randomized:
Noah Stroehle, Kyle Slater, Ivan Parrales, Victor Duarte

Abstract

We created a progressive web app (PWA) running a Pac-Man game that can be played as a human or by an Artificial Intelligence. Using different approaches of machine learning & artificial intelligence, we created a few bots that hook into the "agent interface" of our game. A user/player can either chose to play the game by himself or pick one of the provided bots to watch it play autonomously.

1 Goals

As a team we decided that we wanted to create a game-playing AI. We chose Pac-Man as it appeared to us as a fun thing to do, as well as not too hard so that it would be doable in our limited time.

Our bot/agent should be capable of playing Pac-Man by taking the place of the human player and being in direct control of Pac-Man. The goal of the agent is to maximize it's score and being able to beat random Pac-Man levels. To accomplish this, we wanted to use some form of (deep) reinforcement learning (e.g. Q-learning / actor critic learning).

As a basis for our project we chose to use an existing (reverse-engineered) Pac-Man game, so our project can focus on creating the actual AI, not recreating the game itself.

2 Design & Implementation

The project is done in JavaScript using TensorFlow.js for deep learning tasks. While the game is run through the browser, we used Node.js for training to get increased performance.

2.1 The Game

The game is a modularized version of *Shaun Williams' browser-based Pac-Man remake*. It is designed to be as close to the original Pac-Man as possible.

The "main loop" of the game lives in the file `executive.js`. It updates the current `state`, calls the `renderer` to redraw the canvas and then reschedules itself using the JavaScript function `requestAnimationFrame()` (falling back to `setTimeout()` if needed). Using the game's state system (`states.js`) and the UI / menu system, the game is able to display menus and switch to different user interfaces when the user chose an action. When the player starts a game, the state system is used to change the current state to `playState`.

In the `playState` a `Game` object is used to hold all data relevant to the current gameplay (player, ghosts, map, etc.). Each time the "main loop" is executed the game is advanced one step, player and ghost positions are updated, current ghost behaviour is updated (scatter or chase, recompute targets), pellets are consumed, and so forth.

Each time the player takes a step (basically moves a pixel), which can occur up to 2 times per frame, it is able to change it's direction. During normal play the `frontend` is able to set `player.inputDir`, which is then used before each step to update the actual movement direction (if it is a valid direction, not facing a wall). If the user however selected an AI agent to use in the `AI SETTINGS` screen, `player.agent` is set and will be used to choose the next action. This allows easy addition of bots by simply implementing the "agent interface" and setting it as `player.agent`. If the user wants to take over he has the possibility to override the AI's decision, and the AI will only take over again at the next intersection. This makes the AI a real interactive experience.

2.2 The Bots

Q learning

The first bot we created is bot using classical AI, a simple Q learning implementation with Q table look-ups. Starting out with an empty Q table (empty JS object "{}"). Each iteration during training an episode is played (an episode ends when Pac-Man dies or clears the level). For each frame the agent then chooses an action using the Q table.

The state is therefore encoded into a feature object containing the direction to the nearest pellet, as well as if a ghost is within one tile. Actions are simply represented by a numerical value (0-3 for up, left, down, right). An action is chosen using it's UCB value calculated from the Q table entries for the current state. Initially we used an ϵ-greedy policy, but it turned out that UCB was able to learn quicker, have better exploration/exploitation trade-off, and converge better on a single action – ϵ-greedy policy often started to jitter, switching between multiple directions each step –.

$$UCB_{max} = max_{a \in A} \left[Q(s,a) + \sqrt{\frac{2\ lnN_s}{n_{s,a}}} \right] \Rightarrow a^*$$

At the end of each episode, the `replayBuffer` is used to update the Q table entries accoring to a variant of the Bellman equation suitable for use with UCB:

$$R_{tot,i} = r_i + 0.8 * R_{tot,i+1}$$
$$Q(s_i, a_i)' = Q(s_i, a_i) + R_{tot,i}$$
$$n'_{s_i,a_i} = n_{s_i,a_i} + 1 \text{ and } N'_{s_i} = N_{s_i} + 1$$

Results: Bot navigates maze without any problems, but it is mostly not able to clear the level as it often gets cornered by ghosts, and sometime runs into them. This is due the limited amount of data the state encodes. Ghosts are only registered by the agent when they are on an adjacent tile. Also the bot has no time like perception, so there is no way of registering changes over time, like movement direction. Average score: 1000-1500, Best score: 5800

Q learning with human designed Q table

This agent is the same as the basic Q learning agent, but the Q table is not initialized empty, but specific values designed by us to result in the best performance. This was possible due to the simple state encoding, which drastically limits the size of the Q table.

Results: Bot is now able to clear levels ($\approx 15 - 20\%$), but still has trouble with ghosts entering a tile in the same frame as Pac-Man. Average score: around 2000, Best score: 6340

Policy network

This was our first attempt of using neural networks to create a Pac-Man bot. This agent uses a simple policy network, consisting of 1 hidden layer with 512 neurons (using ReLU as activation function) and a softmax output layer: `FC => RELU => FC => SOFTMAX`. It uses basically the same features as the Q learning bots.

Results: Bot is about as good as the Q learning bot, but took a lot longer to train. Average score: 1000-1200, Best score: 4320

Extended policy network

An extended policy network with an additional hidden layer with 1024 neurons. As well as additional features: Distance to the closest food, Manhattan distance to the ghosts & history of the last 3 frames (for 4 frames in total). Was added on the last day, so we couldn't train it for long enough to get meaningful results.

Deep AC learning

We tried using deep convolution neural networks in combination with Q/AC learning, but had to realize that it takes a very long time to train these types of models (see Problems section below). After a training session of 8 hours only showed little improvement and research told us at least 24 hours of training would be required for "usable" results, we shifted our focus to the other faster approaches. The remnants of these "experiments" can be found in `data/cnn/`.

3 Problems (& Solutions)

Modularization & API

We started out using the Pac-Man remake as is, trying to create a bot for it. It turned out this wasn't the optimal approach, as that remake project was never meant to be used for AI purposes. It could not be used in Node.js, which we wanted to use for training, and it did not expose a way of easily integrating and switching between different AI agents.

Solution: Rewrite the game ourselves in a modular fashion that is extensible, works well with Node.js, and exposes a way of easily adding new AI agents. This rewrite / refactor took us a considerable amount of time we hadn't accounted for in our planning.

Training time

As it turns out, training a Pac-Man bot using (deep) reinforcement learning is computationally very expensive and takes A LOT of time we sadly didn't have. *Online sources* mention that the standard approach in DQN literature is a (final) training time of 200 million frames (roughly 1 million games & two weeks of time). But to get at least some good results 10 million frames (approximately 24 hours) should be sufficient.

Solution: Just train for longer, or optimize the hyper-parameters to speed up training.

No concept of time & movement

As most of our models only receive a static snapshot of a single frame as input, the model has no way to develop a concept of time as well as movement. This decreases the performance of the agents as they, for example, cannot tell if a ghost is moving toward or away from tem.

Solution: Give the model access to more than one frame at once. This slows down training but will increase performance drastically in the long run.

Too simplistic encoding of the current game state

Most of our agents only get a "subset" of the complete current game state. This is due to the high complexity of the game, as well as our limited resources, especially training time (test with CNNs did take ages to make even small improvements).

Solution: Give the model access to a more complete representation of the game state, e.g. by using deep convolution neural networks.

High complexity

While the game of Go (one of the most complex board games) is played on a 19x19 board, Pac-Man uses maps of 36x28 tiles. Additionally Pac-Man is a system that includes multiple agents (Pac-Man & 4 ghosts) and evolves & reacts to actions in real time in a not completely deterministic way (there is some randomness involved). This shows that Pac-Man playing is highly complex tasks, that is even challenging for most humans.

4 Features

4.1 The Game

- Pac-Man & all 4 ghosts
- Pellets & power pellets
- Different ghost behaviours
- Random map generation
- Debug mode: Invincible Pac-Man, visible ghost targets, grid lines, additional logging
- As close as possible to the original arcade game (timings, speeds, scores)

4.2 The Bots

- Q-learning ("`Q LEARNING`")
- Q-learning with human designed Q table ("`Q LEARNING (HUMAN)`")
- Policy network ("`POLICY NETWORK`")
- Extended policy network ("`POLICY NETWORK (EX)`")

4.3 Web App

- Responsive & mobile capable (touch controls)
- Progressive Web App:
  - Runnable on any web-capable device
  - Can be downloaded and installed on your phones homescreen (Android)
  - Immersive fullscreen user experience after install
    → feels like a normal app
  - Offline mode
- Build system using npm with rollup & terser to pack and minify the JS code for "production"/distribution

5 Conclusion

Using methods of classical AI as well as deep learning we were able to create multiple different AI agents. While the bots we created show reasonable behaviour, they do not overcome human play (at least not in any significant way). We kind of underestimated the complexity of Pac-Man & the amount of work required. Sadly, we weren't able to use deep convolutional neural networks as originally planed due to the reasons set out above (the complexity vs. our limited time because of the finals and other projects). Given more time, these approaches could be revisited & bot performance could be improved significantly through much longer training sessions.

All in all it was a fun project & we are happy with the results, especially that the bot is sometimes able to clear the level (or gets very close to clearing it).

6 Future Directions

In future "iterations" it would be possible to further improve the bots performance by tackling the aforementioned problems. Maybe other algorithms / approaches could be tried, e.g. similar to Google DeepMinds Atari AI. Other possibilities include improving on the game itself (fruits, elroy mode, Ms. Pac-Man, Crazy Otto, Cookie Man, audio).

Additionally one could investigate adding additional game modes:

- Human vs Bot: side by side, playing simultaneously trying to reach a higher score than the opponent
- Human vs Bot: playing as a ghost you try to catch the bot-controlled Pac-Man
- Bot vs Bot: training two adversarial bots (one for Pac-Man, one for the ghosts)