Two Headed Snake: Reinforcement Learning Snake

Syed Araib Karim, Fawadul Haq,
Luke Hillard, Michael Nutt,
and Hannah Helgesen
Dept. of Computer Science and Engineering
University of North Texas, Denton, USA
Lukehillard@my.unt.edu - Snake AI Trainer, michaelnutt2@my.unt.edu,
hannahhelgesen@my.unt.edu, syedaraibkarim@my.unt.edu,
fawadhaq@my.unt.edu

***Abstract*—Many breakthroughs in artificial intelligence (AI) come from combining multiple models together into an altogether new idea. Experimenting with new combinations will likely lead to many breakthroughs in the field of AI are altogether necessary to create machines that can navigate the world in it's entirety.**

The goal of this project is to further develop our understanding of reinforcement learning practices by training an AI to play the game Snake competitively against a human player. For those unfamiliar with how the game Snake works, the player controls a snake with the goal of eating bits on the screen to increase the size of the snake, with the challenge being avoiding hitting the wall or the snakes own body. In our two player implementation, the playing field will be larger with the goal being to avoid hitting the walls, your own snake, or the other players snake.

Our AI training will be taken a step further with sentiment analysis on the current score of each player, allowing the AI player to 'taunt' the human player based on difference in scores to provide a realistic simulation of playing with another human player. We hypothesise that our model will demonstrate how the outputs of one model can be used as the inputs for another successfully in a real-time scenario. By leveraging many existing tools we will also show that the field of AI research is ready to begin combining models in novel ways.

I. Workflow

II. Data Specifications

One of the goals of our project was to make the snake agent taunt the opponent at where as stage of the game play. The states are:

1) When the agent is ahead.
2) When the agent is behind.
3) When there is stalemate.

We created a dataset of over 100 taunts Google Drive[1] which is being used by the agent to generate text. Each taunt is given a value of 1,-1,0 depending on the above states respectively.

III. Project Design

Our completed project[2] uses the OpenAI Gym to train the agents on playing snake with PyGame working as the interface for the human player and the display of what is happening in the game.

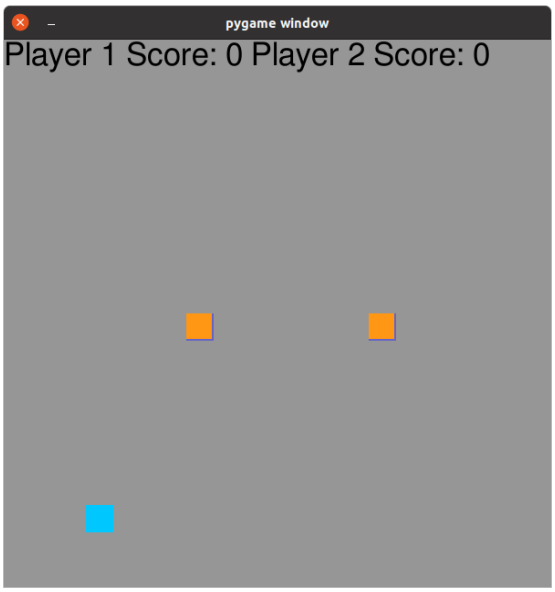

Fig. 1. Game Interface, orange dots are starts of snakes and blue dot the apple.

A. The Game

The game was made using the PyGame library of Python, based on the tutorial series[1]. The tutorial series built Snake as a single player game, we then modified it to work with multiple players, both being humans, one human one AI, or both AIs. We also modified the size of the game display to match what the agents were trained on, going with a 600x600 board with each block being 30x30, creating a 20x20 grid.

The game is over if either snake runs into either a wall, itself, or the other snake. If the snake collides with the apple block it will increase in size by 1. The length of the snake is the score for that player. The code runs by creating two Snake objects

The main game loop is reading inputs from the snakes and updating the positions of the snakes that made the moves, based on if the inputs were the wasd keys or arrow keys.

[1]https://bit.ly/3bJzuHl
[2]https://github.com/michaelnutt2/RLSnakeGame

1) Game Methods:

- `Class Snake`
  - The init function of the class takes in a boolean variable for if it is an AI or not, and a list of the keys that it will use to move the snake. For player 1 that is wasd, for player 2 the arrow keys.
- `game_loop`
  - The main game loop that runs the game. In this function the `player_one` and `player_two` objects are defined based on user input, 1, 2 or no players. It creates the snake starting locations as one third and two thirds into the grid for player 1 and 2, then loops through looking for events indicating key presses to determine where to move each snake.
- `update_move`
  - Takes in a player, event, and boolean variable for if human or not, then returns the change in coordinates based on the event.
- `the_score`
  - Renders the score text on the display based on the length of each snake.
- `snake`
  - Renders the snake blocks based on head location and body segment locations.
- `message`
  - Renders any message displayed, used for AI taunts and for game over screen.

B. Gym Snake

We used, gym-snake, an OpenAI, reinforcement learning gym environment developed by Grant Satchel as a base for our model [2]. The environment provides a snake environment that observes the whole grid, receives a +1 reward upon eating an apple, and receives a -1 reward upon death. We edited the environment for the snake to be able to observe the surrounding coordinates around its head and the euclidean distance between its head and the food. Among the surrounding spaces, it is able to distinguish between collision obstacle (wall or snake body), empty space, and food. We also changed the reward system to grant a reward equal to the reciprocal of the distance to the food (to incentivize it moving towards the food). The reward for eating the food is 10, however, which makes it more favorable to reach the food as opposed to skirting around it. Conversely, the reward for dying is -10 now.

C. Deep Q-Learning

We use a Keras Deep Q-Learning approach for teaching an agent to play Atari Breakout [3]. Deep Q-Learning works by recording the actions in batches then updating the weights of our model. We also predict what our future rewards are, as is normal for Q learning, but instead of a state-action table, we update our models' weights using a mask and gradient to do the same update function in basic Q-Learning. Our γ (gamma) for time discounting is 0.99 and our ϵ (epsilon) for doing a random move at minimum is 0.1 or 10% chance per action. The process for generating our agent is as follows:

1) We generate our models for each snake with the same parameters. They are neural networks of 5 dense layers of 30, 60, 60, 30, then 4 nodes. This is to mimic the network made in [3].
2) Each snake then does random actions for the first 5000 frames.
3) Each snake takes less random actions for the next about 1000000 frames.
4) From there once a snake reaches about 10 food for the last 100 rounds of snake, we consider the model trained.

Once the model is trained, we save the ".keras" file of the model of each snake. Outside of training, when the actual game is run, we allow the agent to cheat a little as it randomly dying due to a random move would create many short games or one sided games. So we allow it to avoid dying randomly to moving into walls and itself and other snakes if the option is there. We also allow it to greedily grab food if the snake's head is in a space next to food.

D. Snake Model Observations

Our model has a few quirks to it. Firstly the model when another snake other than itself is active, will generally meander away from the food until the other snake dies or the food spawns near it. Additionally once a snake dies, it will suddenly look for food much faster than before and explore the grid more. This only happens sometimes. Sometimes after the other snake dies, it finds a loop for it to repeat until it randomly moved out of the loop. The final observation is that if a food spawns on the edge of the grid, the snake has issues going towards it as it avoids edges because it learnt that going near an edge generally means death due to random movement.

E. Text Generation

Our goal was to train the agent to generate taunts based on the dataset we created. We tried several different ways of achieving the task with the limited time resources.

1) Markov Chain: Markov Chain is one of the earliest algorithms used for text generation. It is a stochastic model, meaning that it's based on random probability distribution. Markov Chain models the future state (in case of text generation, the next word) solely based on the previous state (previous word or sequence).

However, there are some disadvantages on using Markov Chains to build text generator:

1) The generated text is as good as the input corpus (garbage in garbage out)
2) Need to create multiple n-gram Markov Chains (high order model) to capture the context

2) Bidirectional LSTM: LSTM (Long Short-Term Memory) neural networks, thanks to their capability to learn long-term dependencies, are successfully used in classification, translation and text generation. The steps we followed are mentioned below:

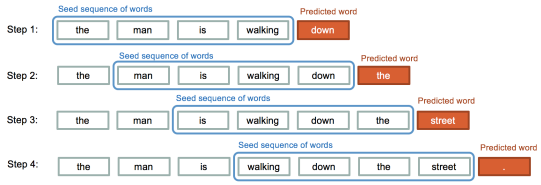

Fig. 2. LSTM: steps to generate sentences.

1) read the data (the novels we want to use as inputs),
2) create the dictionary of words,
3) create the list of sentences, which are the inputs of our neural network,
4) create the neural network,
5) train the neural network, generate new sentences.

Bidirectional LSTMs can be used to train two sides, instead of one side of the input sequence. First from left to right on the input sequence and the second in reversed order of the input sequence.

We added a dropout layer to our Bidirectional LSTM avoid overfitting, one more LSTM layer and, one more dense layer with activation as Relu, and a regularizer to avoid over-fitting again are then added. The output layer has softmax so as to get the probability of the word to be predicted next. To perform the training we initialized the model and the optimizer. Once the model is trained , we saved the weights of the neural network to later use them to generate text. The result of the training loss is showw on Fig 2.

Unfortunately the generated text was not up to our requirements as the model was not able to generate random sentences and always outputted the same text depending on the seed text.

3) GAN: GAN is as a generator network that is trained to produce realistic samples by introducing an adversary i.e. the discriminator network, whose job is to detect if a given sample is “real” or “fake”. GAN have been used successful to generate music and images but its runs into issues with text generation.

Consider the RNN-based generator to be the generator network in a GAN, instead of training the RNN to minimize cross-entropy loss with respect to target one-hot vectors, we will be training it to increase the probability of the discriminator network classifying the sentence as “real”. It’s an issue because, in order to train the generator, we need to feed the output of the generator to the discriminator and back-propagate the corresponding loss of the discriminator. For these gradients to reach the generator, they have to go through the non-differentiable “picking” operation at the output of the generator. This is problematic as back-propagation relies on the differentiability of all the layers in the network.

There are various solutions to the above problem such as:

1) The REINFORCE algorithm and policy gradients (Reinforcement Learning-based solutions)
2) The Gumbel-Softmax approximation (A continuous approximation of the softmax function)

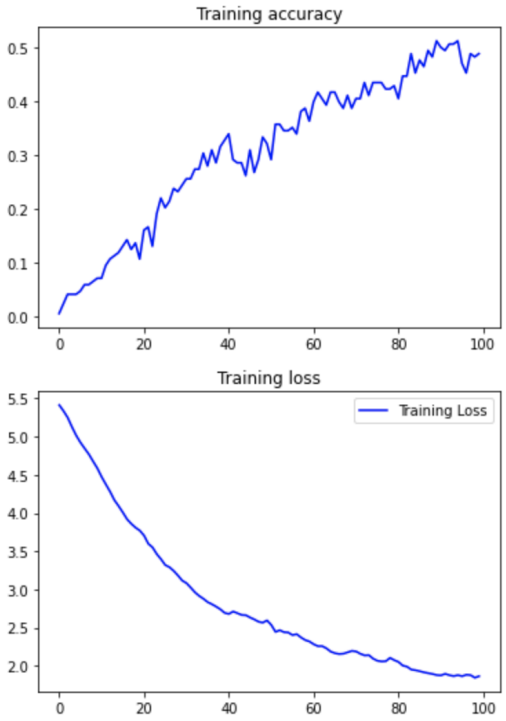

Fig. 3. Bi directional LSTM Training accuracy and loss.

3) Avoiding discrete spaces altogether by working with the continuous output of the generator

F. GPT/Word Matrix

The machine learning models, GPT, GPT-2, and GPT-3, are text transformer models from OpenAI which have been trained on on huge corpuses of internet text. Because of the incredibly generalized datasets used for their training, GPT models could not be used to generate new taunts on the fly, similarly to some of the other text generation models we tried. We attempted to tokenize our hand-made taunt data and use it to fine-tune the parameters of GPT-2, as it was the strongest model we could get access to, however this was not possible do to the small size of our dataset.

Remedying the problem by duplicating values led to overfitting and output taunts so similar to the ones we wrote, it was akin to an implementation wherein we select options from our own hand-made datasets randomly instead of generating new options. Do the time constraints, we ultimately implemented this simple option. GPT’s pretrained transformer model would work if we had the time to come up with a few hundred more taunts to prime it, and it would even make a fun next step to this project.

G. Milestones

1) Acquiring Gym Snake and PyGame assets
2) Setting up a PyGame environment
3) Setting up a Gym Snake environment

4) Edited gym snake's algorithm: update observation with head's surrounding coordinates and food distance, incentivize moving towards food
5) Integrating Deep-Q Learning for 2 player Snake
6) Training and modifying model
7) Creating data of taunts
8) implementing multiple models for Text generation.
    a) Markov chain
    b) Bidirectional LSTM.
    c) GAN.
    d) GPT/Word Matrix

REFERENCES

[1] Yousef Metwally. *Create a Snake Game in Python with PyGame*. https://www.coursera.org/projects/snakegame-python-pygame.

[2] Grant Satchel. *gym-snake*. https://github.com/grantsrb/Gym-Snake.git. 2021.

[3] Mathias Lechner Jacob Chapman. *Deep Q-Learning for Atari Breakout*. https://keras.io/examples/rl/deep_q_network_breakout/. 2020.