

AI Plays Snake via Reinforcement Learning

by Yesenia Pérez Ibarra

Introduction

In this project we can teach an AI to play Snake, creating the game with Pygame and the Deep Learning algorithm with PyTorch. We release the agent into the environment with zero knowledge about how to play the game and implement a reward system in order to teach it how to keep increasing the top score.

Basic Concepts

- *Reinforcement Learning* is a way of teaching a software agent how to behave in an environment by telling it how good it's doing based on a reward.
- In this case, the *agent* is the computer player, the *environment* is the game.
- To train the agent the approach of *Deep Q Learning* is used, which extends RL by using a deep neural network to predict the actions.

Important Variables

Reward

Whenever the snake eats the red dot, the reward is +10, when its game over -10, anything else is 0.

Action

Determines the next move. Logically there are four different actions to take in this game (left, right, up, down) but if assigned like this we can, for example, go right, then left and as a result we immediately die, so we have to avoid 180 degree turns. Because of that only 3 actions dependent of the current state are allowed:

- *Straight* [1, 0, 0]: stay in the current direction, if the snake is going right it remains going right.
- *Right Turn* [0, 1, 0]: right turn depending on the current direction, if you are going right and this action is implemented then you are going down.
- *Left Turn* [0, 0, 1]: by now you probably got it, if you are going right and then implement this action you are going up.

State

We have to tell the snake some information about its environment. There are 11 different boolean values:

- *Danger*: straight, right, left
- *Direction*: left, right, up down
- *Food*: left, right, up down

For example we have this state:

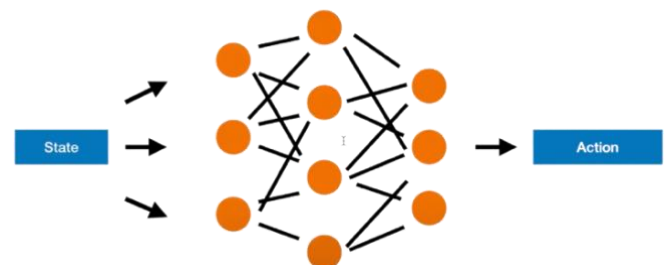


In which we are going right, and the red dot is right and down from the snake so the array would end up like this:

```
[ 0, 0, 0, // there is no immediate
danger
0, 1, 0, 0, // the snake is going right
0, 1, 0, 1] // the red dot is right and
down from the snake's current path
```

Model Overview

We have a feedforward neural network with an input layer (state), a hidden layer and an output layer (action). For the output we have 3 different numbers from which we take the highest and convert it into 1 and the other values 0. Say we have as an output [5, 3, 2] the values would be taken as [1, 0, 0] then proceed to the corresponding action which is straight.



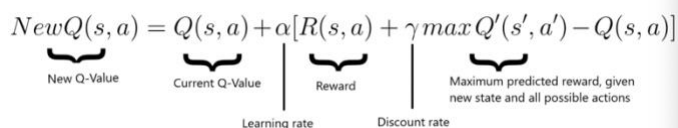
Training the model

The model is trained via Deep Q Learning, in which the Q value stands for the quality of the action so each action should improve the quality of the snake. First, we initialize the Q value (model) with random parameters. Then, it chooses the action with `model.predict()` getting the value of the current state or with random moves (at the beginning when we don't know a lot about the game). When the action is chosen, the code does it and later, the value of the reward comes updating the Q value to train the model.

```
loop(
    1. Choose action: model.predict(state)
      or random
    2. Perform action
    3. Measure reward
    4. Update Q value
)
```

Loss Function

For this model the Q values are updated according to the Bellman Equation, that roughly looks like this:

$$NewQ(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q'(s', a') - Q(s, a)]$$


Looking at the entire equation might be confusing but we can simplify the whole thing and it looks like this:

$$Q = model.predict(state_0)$$

$$Q_{new} = R + \gamma \cdot \max(Q(state_1))$$

Here we have the Q predicting with the old state, and the Q_{new} summing the reward with the maximum value of the Q's new state times gamma. Furthermore, we have:

$$loss = (Q_{new} - Q)^2$$

Ending up with a Mean Squared Error simple formula.

Code Overview

Implementing the game (Pygame)

The game has to be assigned as a loop, with each loop the `play_step` function gets an *action* therefore moving the snake. After moving it returns the corresponding reward, if the game is over or not and the current score.

```
play_step(action)
    return reward, game_over, score
```

Implementing the agent

Here everything comes together (game and model). We implement a training loop in which we receive the information from the game to calculate a state, and from the state we calculate the next action with `model.predict()`. With the new action we enter it in `play_step()` getting the reward, `game_over` and `score` information to calculate a new state. Then we remember the new state, old state, `game_over` and `action` current states to finally train the model.

For a better understanding of the training process its corresponding pseudocode is:

```
state = get_state(game)
action = get_move(state):
    model.predict()
reward, game_over, score =
game.play_step(action)
new_state = get_state(game)
remember
model.train()
```

Implementing the model (PyTorch)

The model is stored in a function called `LinearQNet` it is a feedforward neural network with linear layers storing the new state and old state values from the agent in order to train the model, calling `model.predict()` to execute the next action.

```
LinearQNet (DQN)
    model.predict(state)
    return action
```

Code Implementation

First, we have to install all packages needed to run the program, open the terminal (I'm using MacOS) and enter:

Create an environment named `pygame_env`

```
conda create -n pygame_env
```

Activate the environment

```
conda activate pygame_env
```

Install pygame

```
pip install pygame
```

Install PyTorch from <https://pytorch.org/get-started/locally/>

- PyTorch build: Stable 1.8.1
- OS: Mac
- Package: Conda
- Language: Python
- Compute Platform: CPU

```
conda install pytorch torchvision -c pytorch
```

Install Matplotlib and iPython for plotting

```
pip install matplotlib ipython
```

1. Implementing the game

The source code needed to run the snake game can be found on a Github repository: [python-engineer/python-fun/snake-pygame/snake_game.py](#) from there we are going to modify this file to be able to function without a manual input, instead an agent it's going to be implemented.

Opening this file, we need to change a few things:

- Add a reset function, so after each game the agent should be able to restart another game.
- Implement the reward.
- Change play function so it takes action and computes the direction.
- Keep track of the game iteration.
- Change the `is_collision` function.

To start, change the class name to *SnakeGameAI*

```
25 BLOCK_SIZE = 20
26 SPEED = 20
27
28 class SnakeGame:
29
```

```
33 BLOCK_SIZE = 20
34 SPEED = 20
35
36 class SnakeGameAI:
```

Create a new function called `reset` and add it to the `initializer`

```
def __init__(self, width, height):
    self.w = width
    self.h = height
    # init display
    self.display = pygame.display.set_mode((self.w, self.h))
    pygame.display.set_caption('Snake')
    self.clock = pygame.time.Clock()

    # init game state
    self.direction = Direction.RIGHT
    self.head = Point(self.w/2, self.h/2)
    self.snake = [self.head,
                  Point(self.head.x-BLOCK_SIZE, self.head.y),
                  Point(self.head.x-2*BLOCK_SIZE, self.head.y)]
    self.score = 0
    self.food = None
    self._place_food()

    self.frame_iteration = 0

def reset(self):
    self.direction = Direction.RIGHT
    self.head = Point(self.w/2, self.h/2)
    self.snake = [self.head,
                  Point(self.head.x-BLOCK_SIZE, self.head.y),
                  Point(self.head.x-2*BLOCK_SIZE, self.head.y)]
    self.score = 0
    self.food = None
    self._place_food()
    self.frame_iteration = 0
```

Get rid of the user input and add the action parameter into the `play_step` function

```
def play_step(self):
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_LEFT:
            self.direction = Direction.LEFT
        elif event.key == pygame.K_RIGHT:
            self.direction = Direction.RIGHT
        elif event.key == pygame.K_UP:
            self.direction = Direction.UP
        elif event.key == pygame.K_DOWN:
            self.direction = Direction.DOWN

    # 2. move
    self._move(self.direction) # update the head
    self.snake.insert(0, self.head)

    # 3. check if game over
    game_over = False
    if self._is_collision() or self.frame_iteration > 100000:
        game_over = True
        return game_over, self.score

    # 4. place new food or just move
    if self.head == self.food:
        self.score += 1
        self._place_food()
    else:
        self.snake.pop()

    # 5. update ui and clock
    self._update_ui()
    self.clock.tick(SPEED)

    # 6. return game over and score
    return game_over, self.score

def play_step(self, action):
    # 1. collect user input
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    # 2. move
    self._move(action) # update the head
    self.snake.insert(0, self.head)
```

Check the move function to change the `self.direction` input into the action variable

```
def _move(self, direction):
    self.direction = direction
    self.head = Point(self.head.x + direction.x * BLOCK_SIZE, self.head.y + direction.y * BLOCK_SIZE)
    self.snake.insert(0, self.head)

def _is_collision(self):
    # hits boundary
    if self.head.x > self.w - BLOCK_SIZE or self.head.x < 0 or self.head.y > self.h - BLOCK_SIZE or self.head.y < 0:
        return True
    # hits itself
    for segment in self.snake[1:]:
        if self.head.x == segment.x and self.head.y == segment.y:
            return True
    return False

def _is_collision(self, pt):
    # hits boundary
    if pt.x > self.w - BLOCK_SIZE or pt.x < 0 or pt.y > self.h - BLOCK_SIZE or pt.y < 0:
        return True
    # hits itself
    for segment in self.snake[1:]:
        if pt.x == segment.x and pt.y == segment.y:
            return True
    return False
```

For the `is_collision` function add the `pt` parameter and change `self.head` for it

```
def _is_collision(self, pt):
    # hits boundary
    if pt.x > self.w - BLOCK_SIZE or pt.x < 0 or pt.y > self.h - BLOCK_SIZE or pt.y < 0:
        return True
    # hits itself
    for segment in self.snake[1:]:
        if pt.x == segment.x and pt.y == segment.y:
            return True
    return False
```

For the move function, based on the action we want to determine the next move and define all the possible clockwise directions. Note: *to use these commands, we need to import NumPy as np.*

```

150 _move(self, direction):
151     _move(self, action)
152     #
153     #
154     #
155     #
156     #
157     #
158     #
159     #
160     #
161     #
162     #
163     #
164     #
165     #
166     #
167     #
168     #
169     #
170     #
171     #
172     #
173     #
174     #
175     #
176     #
177     #
178     #
179     #
180     #
181     #
182     #
183     #
184     #
185     #
186     #
187     #
188     #
189     #
190     #
191     #
192     #
193     #
194     #
195     #
196     #
197     #
198     #
199     #
200     #
201     #
202     #
203     #
204     #
205     #
206     #
207     #
208     #
209     #
210     #
211     #
212     #
213     #
214     #
215     #
216     #
217     #
218     #
219     #
220     #
221     #
222     #
223     #
224     #
225     #
226     #
227     #
228     #
229     #
230     #
231     #
232     #
233     #
234     #
235     #
236     #
237     #
238     #
239     #
240     #
241     #
242     #
243     #
244     #
245     #
246     #
247     #
248     #
249     #
250     #
251     #
252     #
253     #
254     #
255     #
256     #
257     #
258     #
259     #
260     #
261     #
262     #
263     #
264     #
265     #
266     #
267     #
268     #
269     #
270     #
271     #
272     #
273     #
274     #
275     #
276     #
277     #
278     #
279     #
280     #
281     #
282     #
283     #
284     #
285     #
286     #
287     #
288     #
289     #
290     #
291     #
292     #
293     #
294     #
295     #
296     #
297     #
298     #
299     #
300     #
301     #
302     #
303     #
304     #
305     #
306     #
307     #
308     #
309     #
310     #
311     #
312     #
313     #
314     #
315     #
316     #
317     #
318     #
319     #
320     #
321     #
322     #
323     #
324     #
325     #
326     #
327     #
328     #
329     #
330     #
331     #
332     #
333     #
334     #
335     #
336     #
337     #
338     #
339     #
340     #
341     #
342     #
343     #
344     #
345     #
346     #
347     #
348     #
349     #
350     #
351     #
352     #
353     #
354     #
355     #
356     #
357     #
358     #
359     #
360     #
361     #
362     #
363     #
364     #
365     #
366     #
367     #
368     #
369     #
370     #
371     #
372     #
373     #
374     #
375     #
376     #
377     #
378     #
379     #
380     #
381     #
382     #
383     #
384     #
385     #
386     #
387     #
388     #
389     #
390     #
391     #
392     #
393     #
394     #
395     #
396     #
397     #
398     #
399     #
400     #
401     #
402     #
403     #
404     #
405     #
406     #
407     #
408     #
409     #
410     #
411     #
412     #
413     #
414     #
415     #
416     #
417     #
418     #
419     #
420     #
421     #
422     #
423     #
424     #
425     #
426     #
427     #
428     #
429     #
430     #
431     #
432     #
433     #
434     #
435     #
436     #
437     #
438     #
439     #
440     #
441     #
442     #
443     #
444     #
445     #
446     #
447     #
448     #
449     #
450     #
451     #
452     #
453     #
454     #
455     #
456     #
457     #
458     #
459     #
460     #
461     #
462     #
463     #
464     #
465     #
466     #
467     #
468     #
469     #
470     #
471     #
472     #
473     #
474     #
475     #
476     #
477     #
478     #
479     #
480     #
481     #
482     #
483     #
484     #
485     #
486     #
487     #
488     #
489     #
490     #
491     #
492     #
493     #
494     #
495     #
496     #
497     #
498     #
499     #
500     #
501     #
502     #
503     #
504     #
505     #
506     #
507     #
508     #
509     #
510     #
511     #
512     #
513     #
514     #
515     #
516     #
517     #
518     #
519     #
520     #
521     #
522     #
523     #
524     #
525     #
526     #
527     #
528     #
529     #
530     #
531     #
532     #
533     #
534     #
535     #
536     #
537     #
538     #
539     #
540     #
541     #
542     #
543     #
544     #
545     #
546     #
547     #
548     #
549     #
550     #
551     #
552     #
553     #
554     #
555     #
556     #
557     #
558     #
559     #
560     #
561     #
562     #
563     #
564     #
565     #
566     #
567     #
568     #
569     #
570     #
571     #
572     #
573     #
574     #
575     #
576     #
577     #
578     #
579     #
580     #
581     #
582     #
583     #
584     #
585     #
586     #
587     #
588     #
589     #
590     #
591     #
592     #
593     #
594     #
595     #
596     #
597     #
598     #
599     #
600     #
601     #
602     #
603     #
604     #
605     #
606     #
607     #
608     #
609     #
610     #
611     #
612     #
613     #
614     #
615     #
616     #
617     #
618     #
619     #
620     #
621     #
622     #
623     #
624     #
625     #
626     #
627     #
628     #
629     #
630     #
631     #
632     #
633     #
634     #
635     #
636     #
637     #
638     #
639     #
640     #
641     #
642     #
643     #
644     #
645     #
646     #
647     #
648     #
649     #
650     #
651     #
652     #
653     #
654     #
655     #
656     #
657     #
658     #
659     #
660     #
661     #
662     #
663     #
664     #
665     #
666     #
667     #
668     #
669     #
670     #
671     #
672     #
673     #
674     #
675     #
676     #
677     #
678     #
679     #
680     #
681     #
682     #
683     #
684     #
685     #
686     #
687     #
688     #
689     #
690     #
691     #
692     #
693     #
694     #
695     #
696     #
697     #
698     #
699     #
700     #
701     #
702     #
703     #
704     #
705     #
706     #
707     #
708     #
709     #
710     #
711     #
712     #
713     #
714     #
715     #
716     #
717     #
718     #
719     #
720     #
721     #
722     #
723     #
724     #
725     #
726     #
727     #
728     #
729     #
730     #
731     #
732     #
733     #
734     #
735     #
736     #
737     #
738     #
739     #
740     #
741     #
742     #
743     #
744     #
745     #
746     #
747     #
748     #
749     #
750     #
751     #
752     #
753     #
754     #
755     #
756     #
757     #
758     #
759     #
760     #
761     #
762     #
763     #
764     #
765     #
766     #
767     #
768     #
769     #
770     #
771     #
772     #
773     #
774     #
775     #
776     #
777     #
778     #
779     #
780     #
781     #
782     #
783     #
784     #
785     #
786     #
787     #
788     #
789     #
790     #
791     #
792     #
793     #
794     #
795     #
796     #
797     #
798     #
799     #
800     #
801     #
802     #
803     #
804     #
805     #
806     #
807     #
808     #
809     #
810     #
811     #
812     #
813     #
814     #
815     #
816     #
817     #
818     #
819     #
820     #
821     #
822     #
823     #
824     #
825     #
826     #
827     #
828     #
829     #
830    
```

As we don't have user input, we can delete this if

```
if __name__ == '__main__':  
    game = SnakeGame()  
  
    # game loop  
    while True:  
        game_over, score = game.play_step()  
  
        if game_over == True:  
            break  
  
    print('Final Score', score)  
  
pygame.quit()
```

Rename the file to `game.py`

2. Implementing the agent

Create a new file named `agent.py`

Import the libraries

```
from numpy.core.fromnumeric import mean
import torch
import random
import numpy as np
from collections import deque
```

Import the classes from the game, model and helper files

```
from game import SnakeGameAI, Direction, Point
from model import LinearQnet, QTrainer
from helper import plot
```

Create the agent's parameters (they can vary)

MAX_MEMORY = 100_000

BATCH_SIZE = **1000**

LR = 0.001

Create a class (Agent) which contains:

- Init function

```
def __init__(self):
    self.n_games = 0
    self.epsilon = 0 # parameter to control randomness
    self.gamma = 0.9 # discount rate, must be  $x < 1$ 
    self.memory = deque(maxlen=MAX_MEMORY) # popleft()
    self.model = LinearQNet(11, 256, 3) # [size of the states, hidden
    self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)
```

- The training functions

- Train_long_memory

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE) # list of tuples
    else:
        mini_sample = self.memory

    states, actions, rewards, next_states, dones = zip(*mini_sample)
    self.trainer.train_step(states, actions, rewards, next_states, dones)
```

- Train_short_memory

```
def train_short_memory(self, state, action, reward, next_state, done):
    self.trainer.train_step(state, action, reward, next_state, done)
```

- Remember

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
```

- Get action

```
def get_action(self, state):
    #random moves: tradeoff exploration/exploitation
    self.epsilon = 80 - self.n_games # + games, - epsilon
    final_move = [0,0,0]
    if random.randint(0,200) < self.epsilon: # - epsilon, -frequent
        move = random.randint(0,2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float) #raw value
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1
    return final_move
```


- Get_state

```
def get_state(self, game):
    head = game.snake[0]
    point_l = Point(head.x - 20, head.y)
    point_r = Point(head.x + 20, head.y)
    point_u = Point(head.x, head.y - 20)
    point_d = Point(head.x, head.y + 20)

    dir_l = game.direction == Direction.LEFT
    dir_r = game.direction == Direction.RIGHT
    dir_u = game.direction == Direction.UP
    dir_d = game.direction == Direction.DOWN

    state = [
        # Danger straight
        (dir_r and game.is_collision(point_r)) or
        (dir_l and game.is_collision(point_l)) or
        (dir_u and game.is_collision(point_u)) or
        (dir_d and game.is_collision(point_d)),

        # Danger right
        (dir_u and game.is_collision(point_r)) or
        (dir_d and game.is_collision(point_l)) or
        (dir_l and game.is_collision(point_u)) or
        (dir_r and game.is_collision(point_d)),

        # Danger left
        (dir_d and game.is_collision(point_r)) or
        (dir_u and game.is_collision(point_l)) or
        (dir_r and game.is_collision(point_u)) or
        (dir_l and game.is_collision(point_d)),

        # Move direction
        dir_l,
        dir_r,
        dir_u,
        dir_d,

        # Food location
        game.food.x < game.head.x, # food left
        game.food.x > game.head.x, # food right
        game.food.y < game.head.y, # food up
        game.food.y > game.head.y, # food down
    ]

    return np.array(state, dtype=int)
```

If

```
if __name__ == '__main__':
    train()
```

Global function called train

```
def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    while True:
        # get old state
        state_old = agent.get_state(game)

        # get move
        final_move = agent.get_action(state_old)

        # perform move and get new state
        reward, done, score = game.play_step(final_move)
        state_new = agent.get_state(game)

        # train short memory
        agent.train_short_memory(state_old, final_move, reward, state_new, do

        # remember
        agent.remember(state_old, final_move, reward, state_new, done)

        if done:
            # train long memory, plot result
            game.reset()
            agent.n_games += 1
            agent.train_long_memory()

            if score > record:
                record = score
                agent.model.save()

            print('Game', agent.n_games, 'Score', score, 'Record:', record)

            plot_scores.append(score)
            total_score += score
            mean_score = total_score / agent.n_games
            plot_mean_scores.append(mean_score)
            plot(plot_scores, plot_mean_scores)
```

3. Implementing the model

Create a new file called model.py

Import the libraries

```
import torch
from torch._C import dtype
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import os
```

Create class LinearQNet

```
class LinearQNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

    def save(self, file_name='model.pth'):
        model_folder_path = './model'
        if not os.path.exists(model_folder_path):
            os.makedirs(model_folder_path)

        file_name = os.path.join(model_folder_path, file_name)
        torch.save(self.state_dict(), file_name)
```

Create class QTrainer

```
class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()

    def train_step(self, state, action, reward, next_state, done):
        state = torch.tensor(state, dtype=torch.float)
        next_state = torch.tensor(next_state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.long)
        reward = torch.tensor(reward, dtype=torch.float)
        # (n,x)

        if len(state.shape) == 1:
            # (1,x)
            state = torch.unsqueeze(state, 0)
            next_state = torch.unsqueeze(next_state, 0)
            action = torch.unsqueeze(action, 0)
            reward = torch.unsqueeze(reward, 0)
            done = (done, )

        # 1: predicted Q values w current state
        pred = self.model(state)

        target = pred.clone()
        for idx in range(len(done)):
            Q_new = reward[idx]
            if not done[idx]:
                Q_new = reward[idx] + self.gamma * torch.max(self.model(next_

            target[idx][torch.argmax(action).item()] = Q_new

        # 2: Q_new = r + y * max(next_predicted Q value)
        # pred.clone()
        # preds[argmax(action)] = Q_new
        self.optimizer.zero_grad()
        loss = self.criterion(target, pred)
        loss.backward()

        self.optimizer.step()
```

4. Create a helper to plot the output of the game

Create a new file named helper.py. With the help of matplotlib it will plot scores vs number of games played, displaying two graphs:

- The mean of the scores
- The value of the scores

```
class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()

    def train_step(self, state, action, reward, next_state, done):
        state = torch.tensor(state, dtype=torch.float)
        next_state = torch.tensor(next_state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.long)
        reward = torch.tensor(reward, dtype=torch.float)
        # (n,x)

        if len(state.shape) == 1:
            # (1,x)
            state = torch.unsqueeze(state, 0)
            next_state = torch.unsqueeze(next_state, 0)
            action = torch.unsqueeze(action, 0)
            reward = torch.unsqueeze(reward, 0)
            done = (done, )

        # 1: predicted Q values w current state
        pred = self.model(state)

        target = pred.clone()
        for idx in range(len(done)):
            Q_new = reward[idx]
            if not done[idx]:
                Q_new = reward[idx] + self.gamma * torch.max(self.model(next_

            target[idx][torch.argmax(action).item()] = Q_new

        # 2: Q_new = r + y * max(next_predicted Q value)
        # pred.clone()
        # preds[argmax(action)] = Q_new
        self.optimizer.zero_grad()
        loss = self.criterion(target, pred)
        loss.backward()

        self.optimizer.step()
```

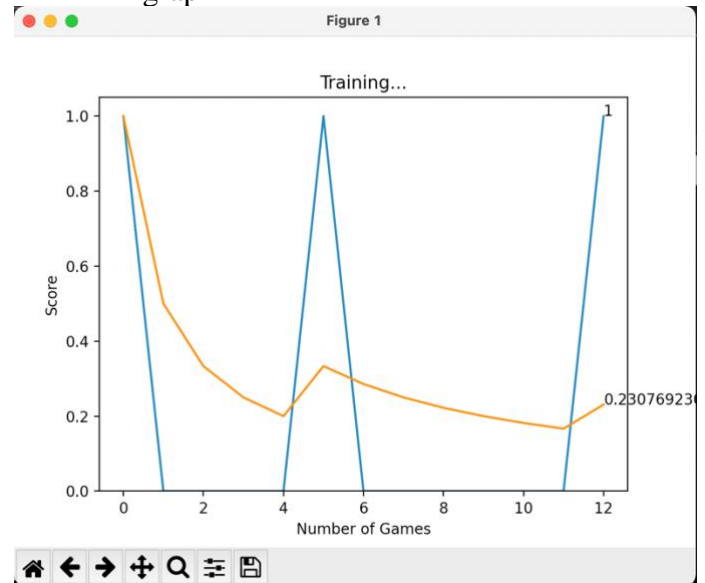
Running the program

Enter in the terminal

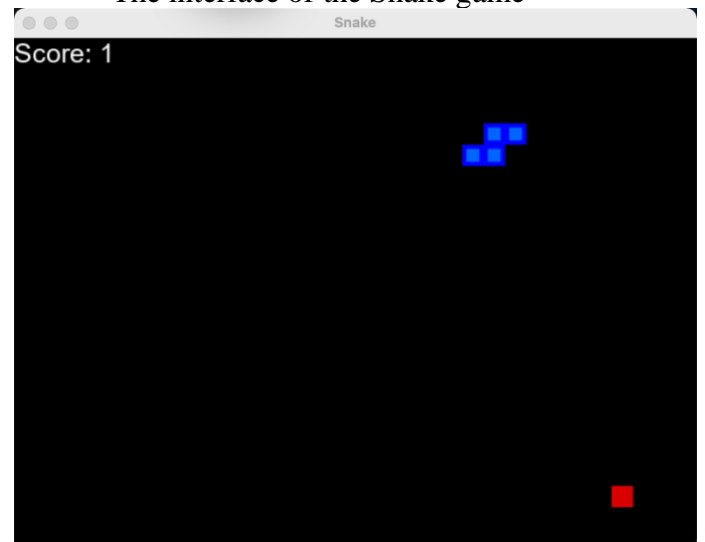
```
python agent.py
```

Then two windows pop out:

- A graph

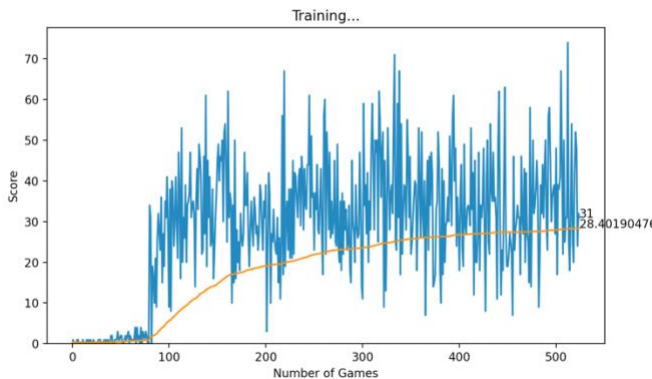


- The interface of the Snake game



Results

To conclude with this project, we may note that as we run the program, in fact, it gets better at playing Snake. It takes the agent between 80-100 games to get higher scores than 1 through 5. At first it seems completely lost as we may notice for this graph

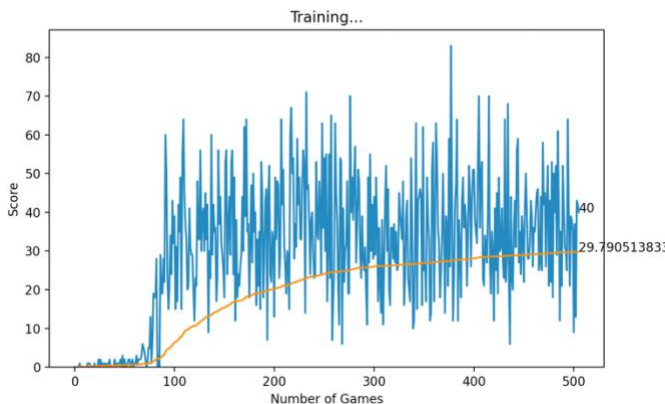


This is the graph for the first run roughly between 0-80 the scores are low, but they increase exponentially when it hits the 70-80 mark maintaining those high scores and the mean score rising with it.

```
Figure(1280x960)
Game 79 Score 0 Record: 4
Figure(1280x960)
Game 80 Score 0 Record: 4
Figure(1280x960)
Game 81 Score 34 Record: 34
Figure(1280x960)
Game 82 Score 31 Record: 34
Figure(1280x960)
```

Here we have a snippet from the terminal information which confirms the leap from 4 to 34 in game #81.

For the second run we have similar behavior:



```
Game 76 Score 5 Record: 6
Figure(1402x648)
Game 77 Score 13 Record: 13
Figure(1402x648)
```

Although the significant leap here is done between game #76 and #77. We can also note a difference of approximately 1.39 between mean scores.

Sources

- [1] <https://github.com/python-engineer/snake-ai-pytorch>
- [2] <https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a>
- [3] Tutorial for installing Anaconda:
<https://youtu.be/9nEh-OXVaNI>