Automatic game playing with DL

Abdullah Al Forkan
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
Abdullah-al.forkan@stud.hshl.de

***Abstract*—In this paper, I have presented an implementation-based deep learning project of automatic snake game playing. First, I introduced some basic insights into deep learning that are related to game development. Then, I formalized deep reinforcement learning focusing on the implementation with deep Q-learning. Following that, I divided the project into three main parts making the game, agent, and modeling. Here, I deeply illustrated the whole development showing some important parts of the Python code. Though Structuring the implementation was quite confusing, I could represent the whole working process sequentially. Finally, I projected the results I got from my deep experiments and concluded the paper.**

I. Introduction

Artificial intelligence is the biggest innovation happened in the recent decade, which changed the way of living in the world by introducing an artificial brain implemented in any system. Machine learning is a subset of AI that learns from data using algorithms and Deep learning is a subset of Machine learning that uses large amounts of data showing exceptional performance in such as image and speech recognition, game development, Automation, Natural Language Processing (NLP), etc. In this article, I mainly focused on deep learning game development. Though the first innovation started in the 1950s with the ‘Ferranti Mark 1’ machine that learned to play Checkers and Chess, the proper deep learning implementation came to us with DeepMind’s Atari 2600 games developed by Deep Q-Network in 2013. It shocked us when DeepMind’s AlphaGo beat the world champion Go player in 2016, showing the power of deep neural networks in strategy games after that deep learning became unstoppable defeating professional players in various video games such as Dota-2 in 2018. [1] There are various reasons developers started using deep learning algorithms to develop games. Designers could not always anticipate possible situations in the game environment, agents and constantly changing gameplay. It is also very difficult to program a realistic gaming experience and every possible requirement for user satisfaction. So, deep learning came to play its role in deploying the neural network which works like our brain cells. It opened the gate to create realistic and immersive game environments through procedural content generation. Developing non-player character behavior became easier and more advanced making them more intelligent in learning evolving player interactions. Natural Language Processing of deep learning is currently used everywhere to create more realistic in-game dialogues and player interactions with virtual guides. Through convolutional neural networks (CNNs) and recurrent neural networks (RNNs), gesture control and object detection can be configured inside a game. Applying deep learning in simulator games has opened another direction for engineering in quality assurance. Like games, real-world machines can be trained and tested in any environment without any delay or risk. Hence, bug detection and identification of issues are now done more efficiently. Manual content creation has come to an ease with the help of Generative Adversarial Networks (GANs) which generate textures, assets, and even the whole game levels. Real-world physics is a difficult development part of the game and deep neural network makes it easy to create object physics such as water flow, wind, realistic fire, and different types of object behaviors. There is always some maths working behind the strategic games like chess, go, tic-tac-toe, etc. Deep learning can handle that very efficiently which even humans could not do before. [2] [3]

II. Deep Learning and Algorithms for games

According to the famous book ’Dive Into Deep Learning’, Deep learning is a subfield of machine learning that focuses on the training of neural networks to perform tasks without explicit programming. It involves the use of artificial neural networks with multiple layers (deep neural networks) to learn patterns and representations from data [4]. Deep learning includes some fundamental key concepts like Deep Neural Networks, Training and Learning, Activation Functions, Loss Functions and Optimization Algorithms [4]. Now I will discuss about some of the algorithms that are used in game development.

A. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of deep neural network architecture designed for processing and analyzing structured grid data, such as images and video. They have proven highly effective in tasks like image classification, object detection, and image recognition [4]. In video games, CNN is used for object detection, image and video analysis, and predicting customer lifetime value (CLV). CLV is based on player behavior and game interaction.

B. Recurrent Neural Networks (RNNs)

Recurrent Neural Network is a class of neural networks designed for sequential data processing. RNNs are particularly well-suited for tasks involving sequences, such as natural language processing and time series analysis [4]. It plays a pivotal role in enhancing video games by enabling more realistic behaviors, adapting gameplay, generating content, storytelling, gesture recognition in interactive environments, preventing cheating, etc.

C. *Long Short Term Memory Networks (LSTMs)*

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture designed to overcome the limitations of traditional RNNs in capturing long-range dependencies in sequential data. LSTMs are particularly effective in tasks involving time series prediction, natural language processing, and other sequence-related problems [4]. So, it adds up more efficiency over RNNs in game development.

D. *Generative Adversarial Networks (GANs)*

GANs were introduced by Ian Goodfellow and his colleagues in 2014 and have since become widely used for generating realistic and high-quality synthetic data. It consists of two neural networks a generator and a discriminator that are trained concurrently through adversarial training. They are broadly used in reinforcement learning, semi-supervised learning, and fully supervised learning [5]. In video games, GANs have been used for various purposes, including image generation and editing, procedural content generation, and AI agent generation. [6]

E. *Deep Belief Networks (DBNs)*

DBNs are stochastic neural networks that can extract rich internal representations of the environment from the sensory data. It can be used in developing the game agent and the environment. [7]

F. *Autoencoders*

Autoencoders are a type of neural network that are trained to encode input data into a lower-dimensional representation and then decode that representation back to the original input format [5]. It can be used for procedural content generation, which is the process of generating game content algorithmically rather than manually [8].

III. Deep Reinforcement Learning

Deep reinforcement learning is a combination of reinforcement learning and Q-learning where deep neural networks are implemented. Reinforcement learning is a machine learning algorithm where an agent makes sequential decisions by interacting with an environment and receiving rewards or penalties as feedback. The feedback is calculated based on its actions and goals to follow an algorithm that increases the overall reward over time. Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state and handle problems with stochastic transitions getting rewards without requiring adaptation. Deep reinforcement learning is in between supervised and unsupervised learning as it is not quite dependent on large amounts of data. This algorithm learns by solving multi-layer problems by trial and error training itself by taking the sequence of decisions and receiving rewards or penalties towards the goal of maximum reward. [9]

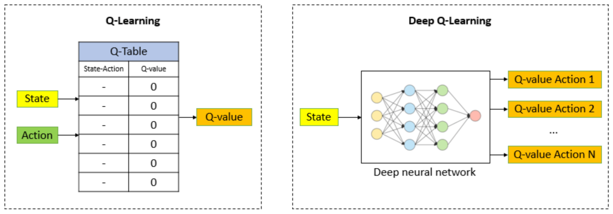


Fig. 1. Q-Learning and DQ-Learning [10]

1) Q-learning: It is a model-free, value-based, off-policy algorithm that is used to find the optimal policy for an agent in a given environment. The algorithm determines the best series of actions to take based on the agent's current state. The "Q" in Q-learning stands for quality, representing how valuable action maximizes future rewards. Q-learning gives us a Q-value and can be calculated through the Bellman Equation. In our snake game, the Q-value is initially predicted by the state of the model. Then for every new Q-value, we pass it through the Bellman Equation. On the right side of the Figure, we can observe the process of how the Q-value is calculated from the table of states and actions. [10]

2) Deep Q-learning: Deep Q-Learning is introduced when Q-Learning fails to calculate a large number of states and actions which makes Q-Table continuously growing. For example, assume a game table with more than a thousand states and actions, which creates a huge table that is very critical for using just Q-Learning. For some games like Chess and Alpha Go, it is even unthinkable. So, deep Q-learning solved this problem by combining Q-learning and deep neural networks. It receives the states as input and provides the Q-values for all possible actions as output. [10]

Deep Q-Learning process [11]:

1. Initialize Q-values: We do not have any Q-value at the start of the game. So, randomly predict a Q-value value and start our training process.

2. Choose action a for state s (Best Q-value) Here comes the Exploration part. In the beginning, it is really necessary to train the model, so that it can create enough training samples and remembers the parameters by keeping samples. The more training steps we get, the more we reduce the random exploration and use exploration instead. I explained the implementation in the agent section.

3. Perform action a, new state s'.

4. Measure reward R.

5. Update Q with Bellman Equation,

$$\underbrace{NewQ(s,a)}_{\text{New Q value for that state and that action}} = \underbrace{Q(s,a)}_{\text{Current Q value}} + \underbrace{\alpha}_{\text{Learning Rate}}[\underbrace{R(s,a)}_{\text{Reward for taking that action at that state}} + \underbrace{\gamma}_{\text{Discount rate}} \underbrace{max\,Q'(s',a')}_{\text{Maximum expected future reward given the new s' and all possible actions at that new state}} - Q(s,a)]$$

Fig. 2. Bellman Equation [11]

6. Update weights of the Neural Network

In my case, I used the Bellman equation step by step to find the new Q-value and updated our neural net weights to reduce the error. This makes the main equation the following form [11]:

$$Q_{new} = r + \gamma \times max_{a'} Q(s', a')$$

Characteristics:

r = Immediate reward

γ= Discount factor

Q(s, a) = Predicted Q-value of the next state.

The max function returns the maximum predicted Q-value of all possible actions in the next state.

The new Q-value Q_{new} is then used to update the Q-value of the selected action.

IV. Implementing the snake game

In our game, we used Q-value-based Deep Learning which is also called Deep Q Learning that extends reinforcement learning by using a neural network to predict the actions. At first, I created a snake game using Python programming in Visual Studio with Anaconda prompt, where I imported 'Pygame', 'Randon', 'Enum', and 'namedtuple'. Block size and speed both are set to 20, which can be manipulated. Then, I denoted five RGB colors randomly for the components and included a text font named 'Arial.tff'.

A. Basic Components Construction:

1) Display: To make the game environment, I created a display of size (640*480) with a caption and default Pygame clock.

2) Player/Snake: I constructed the block as Figure:x starting with the snake position. Then took the head of the snake and went left left twice in x direction by negative block size to make the whole snake body which contains three blocks.

```
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)]
```

3) Food: For the food block construction, I created a place food function that generates two random values of x and y between 0 and the width and height of the game display. Then the values of x and y are rounded to the nearest multiple of block size. Finally, it creates the food block and assigns it to the food. If the food block is in the snake, the place food function is called again to place a new food.

```
def _place_food(self):
x = random.randint(0, (self.w-BLOCK_SIZE )
//BLOCK_SIZE)*BLOCK_SIZE
y = random.randint(0, (self.h-BLOCK_SIZE )
//BLOCK_SIZE)*BLOCK_SIZE
self.food = Point(x, y)
if self.food in self.snake:
   self._place_food()
```

B. Play Steps

1) Direction: Our snake gets the following binary values to go in a certain direction.

- Straight – [1, 0, 0]
- Right turn – [0, 1, 0]
- Left turn – [0, 0, 1]

We have 4 actions that our snake can perform which are up, down, left, and right. But if we design it like this then for example what can happen is we might take the action left and then because of two directions up and down which results in a 180-degree turn, we can immediately die. So, the better decision is to design three actions as shown above. Here, [1, 0, 0] means the current direction which means if we go right, we stay right. If we go left, we stay left. If we are going right, by taking another right [0, 1, 0], we go down, and by taking a left [0, 0, 1] turn we can go up. So, the direction changes according to the direction of the snake head.

2) Movement: It is nothing but our current snake block position in the display. The following representation will occur if the snake gets input for direction from the AI model.

- if direction == Direction.RIGHT: x += BLOCK SIZE
- elif direction == Direction.LEFT: x -= BLOCK SIZE
- elif direction == Direction.DOWN: y += BLOCK SIZE
- elif direction == Direction.UP: y -= BLOCK SIZE

here you can notice for down and up directions I increased and decreased blocks respectively, because our display starts at [0,0] at the top.

3) Reward and Score: If the snake wanders around, there is no reward, and the score does not change. Every time the snake successfully eats food that means 'snake-head = food', then a reward of +10 will be added to the score. If there is a collision happened the reward is -10 and the game will be over following a reset operation. The negative reward is only given as a penalty for the wrong move

4) Game Over: There are two kinds of collision that lead to game over state. a. Boundary Collision: When in the X-direction or Y-direction following 2 cases happen, the snake hits the right/left wall and up/down wall respectively.

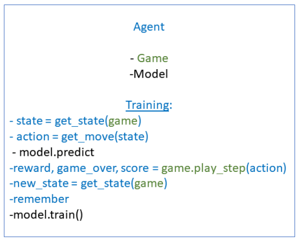
- head.x is smaller than 0 and bigger than (Width - Block size)
- head.y is smaller than 0 and bigger than (Height - Block size)

b. Self-Collision: When our snake-head is in self.snake[1:], it hits itself. Here [1:] is called slicing and represents the full length of the snake. Each kind of collision leads to the game being over, and the score is returned. Following that, the game will be reset. Reset means the score will be set to 0, the snake will start from the initial condition, and the food function will put a new food randomly.

All the components and play steps will be updated to the UI that will display the game as programmed. [12]

V. Implementing the agent

In the definition of AI games, an agent is a computer-controlled player that interacts with the environment and makes ideal decisions based on the states using learning algorithms. In our implementation A snake learns to eat food following the rules and learns from previous experiences using a deep learning algorithm called deep reinforcement learning. Here, the agent development process includes the game development and modeling components and trains the snake accordingly. The following figure* shows the major characteristics of the agent.



Agent

- Game
-Model

Training:
- 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()

Fig. 3. Agent

To implement the deep Q learning algorithm in Python, I imported the PyTorch framework with some other modules like random, NumPy, and deque and included the previously developed game file and modeling file for training.

A. Initials

First, we set some required parameters that will make our calculations. Maximum memory = 100000 Batch size = 100 Learning rates = 0.001 Epsilon = 0 Discount rate: Gamma = 0.9

B. Direction setup:

To get the states, we need to set the standard direction of the game environment.

- Point left = Point(head.x - 20, head.y)
- Point right = Point (head.x + 20, head.y)
- Point Up = Point(head.y – 20, head.x)
- Point Down = Point(head.y + 20, head.x)

here Point Left means the negative of snake-head block in the X-direction. So, a block size of 20 is negated and the Y-direction does not change. Similarly, for point right it is positive. For point up, it goes negative of y and for point down, it is positive of y direction block by block.

C. Get States

We have 11 states in total.

- Move direction states = [left, right, up, down]
- Danger zones = [danger straight, danger right, danger left]
- Food direction = [left, right, up down]

The move direction states show the possible actions the snake can perform and are guided by the danger and food states. Danger states are dependent on the current direction of the head of the snake. Thus, these states are not as equal as regular direction points. Danger straight is true when,

- snake direction is right, and collision is right or
- snake direction is left, and collision is left or
- snake direction is up, and collision is up or
- snake direction is down, and collision is down

Danger right is true when,

- snake direction is up, and collision is right or
- snake direction is down, and collision is left or
- snake direction is left, and collision is up or
- snake direction is right, and collision is down

Danger left is true when,

- snake direction is down, and collision is right or
- snake direction is up, and collision is left or
- snake direction is right, and collision is up or
- snake direction is left, and collision is down

After the inspection of these three states, we get an array of three values in binary. Food states denote the location of the food and return an array of 4 values in binary.

- Food left is true when food.x <head.x,
- Food left is true when food.x >head.x,
- Food up is true when food.y <head.y,
- Food down is true when food.y >head.y,

So, these are our 11 states, which we will set as input of the deep reinforcement learning model.

D. Scenarios

The first scenario (Figure 2):

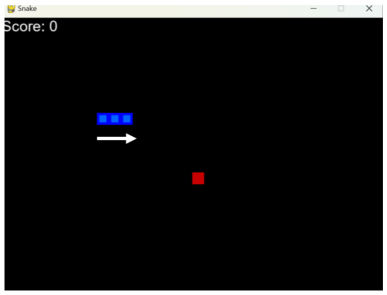


Fig. 4. First Scenario

According to the binary state representation,

- Danger zones = [danger straight, danger right, danger left] = [0, 0, 0]
- Direction states = [left, right, up, down] = [0, 1, 0, 0]
- Food direction = [left, right, up, down] = [0, 1, 0, 0]

As our snake is ahead of the food and going straight there is still no danger. Thus all the danger states are 0.

The direction state right is set to 1, as we have to go right after a certain distance.

Now our food is located on the right side of the snake, so the food right state is true.

Second scenario(Figure-3):

- Danger zones = [danger straight, danger right, danger left] = [1, 0, 0]
- Direction states = [left, right, up, down] = [0, 1, 0, 1]
- Food direction = [left, right, up, down] = [0, 1, 0, 1]

Here our danger straight is true. Our direction will be right and to get the food we have to go down and right.

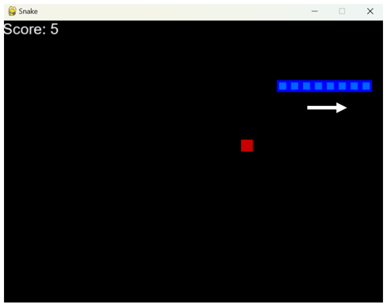


Fig. 5. Second Scenario

E. Remember

This function is to remember (state, action, reward, next state, done). This will append the values of the parameters until maximum memory is reached.

F. Train short memory

Each game step is stored as a short memory and can store up to the maximum memory size which we set at 100,000.

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

That means, 1 short memory contains 1 single set of tuples. Which are state, reward, next state, and done.

G. Train long memory

All short memories are stored in the long memory in a 'Batch size' of 100 memories. Each batch size of 100 is a 'Mini Sample' of long memory. The trick to make the sample is by a simple if condition. If the memory is bigger than the batch size then keeps making mini samples, otherwise it waits until that.

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

Finally, we will Zip the mini sample which contains a list of tuples, and train them. It can be noticed that the parameters are not singular anymore.

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

H. Get action

This is an exciting function of the agent, where a trade-off between exploration and exploitation happens based on the state. In the beginning, we want to make sure some random exploration happens. The better our Agent gets by acquiring values from memories, moves will be done through exploitation. This condition will be controlled by an epsilon parameter set to 80 and can be changed. It has a negative relation with the number of games. Thus, the bigger the number of games becomes, the smaller our Epsilon gets.

```
   self.epsilon = 80 - self.n_game.
```

1) Random exploration: Now, we set a randomizer limit [0 to 200]. If the randomizer value is smaller than our epsilon, then the final move array will get a random direction between the index (0 to 2).

```
   self.epsilon = 80 - self.n_games
final_move = [0,0,0]
if random.randint(0, 200) < self.epsilon:
   move = random.randint(0, 2)
   final_move[move] = 1
```

2) Exploitation: If the number of games is so big that our epsilon becomes smaller than 0, else statement will occur. Then the action will be predicted based on a state from the PyTorch deep reinforcement model.

```
else:
state0 = torch.tensor(state,
         dtype=torch.float)
prediction = self.model(state0)
move = torch.argmax(prediction).item()
final_move[move] = 1
return final_move
```

For example, we get a value of [6, 3.1, 0.2] from our model output where we get a maximum value at index 0. Then the index of maximum value will be chosen, and the final move will be 1 in the exact index 0. So, the movement array will be [1, 0, 0] directing the snake to go straight. Eventually, we get a final move in return in this function either randomly or by prediction algorithm. We will go into the detailed calculation of the model prediction value in the modeling section.

I. Train

This global train function contains the main running loop of the agent development which performs the automatic actions based on the algorithm. The loop starts initially with an old state by exploration and performs a random move. This gives us the reward, score, and done values and generates a new state. Based on that short memory is trained. Then the task is remembering all the values of the parameters, which are the old state, final move, reward, new state and done. Now we add a condition, if done or game over occurs, we reset the game, increase the number of games by 1, and store it in the long memory. If we also get a score that is bigger than our previous record, it will be saved as a new record in our model. Finally, it is the right moment to print the number of games, score, and highest record. Some plotting variables are also included to understand the results better. [12]

VI. Implimenting the model

A. Model Implementation

For modeling purposes, I imported some modeling tools torch, torch.nn, torch.optim, torch.nn.functional, and os. I designed the modeling mainly with two operations. One is Linear QNet and the other one is QTrainer.

B. Linear QNet using Neural Network

1) Model Construction: A linear Q-network is a type of Deep Q-network (DQN) that uses a linear function approximator to estimate the Q-values for each action in a given state(1). The network takes the state as input and outputs a tensor of Q-values for each action (1). The Q-values are then used to select the best action to take in the current state. we will use a feed-forward neural network with an input layer, a hidden layer, and an output layer. It can be extended or improved, but our implementation will work fine with this specification. to create the model, we need to specify the size of (input layer, hidden layer, and output layer) = (11, 256, 3). As we have 11 states, the input layer has to be 11. The output layer size will be 3, as we need 3 values straight, right, or left and the hidden layer size can be manipulated. Here we set it at 256. Now we create two linear relations and connect them using the PyTorch neural network model (nn). First, Linear1 has an input layer as input and a hidden layer as output. Similarly, Linear2 has a hidden layer as input and an output layer as output. Which is programmed as, self.linear1 = nn.Linear(input size, hidden size) self.linear2 = nn.Linear(hidden size, output size)

2) Forward function: The forward function in a linear Q-network is responsible for computing the Q-values for each action given the current state of the environment 1. It takes the current state as input and returns a tensor of Q-values for each action (1). The Q-values are then used to select the best action to take in the current state. For PyTorch, we always have to implement the forward function. It takes the state as input and passes it through a series of fully connected layers with non-linear activation functions such as ReLU. The output of the final layer is a tensor of Q-values for each action. I used the x variable to connect both linear models and used the activation function ReLU only in linear1.

```
x = F.relu(self.linear1(x))
x = self.linear2(x)
```

Finally, we save the model indicating a path by using os module we imported.

C. QTrainer:

It contains the model, learning rate, and discount rate gamma. During training, the weights of the network are updated using a variant of stochastic gradient descent called Adam optimizer (1). The optimizer estimates the first and second moments of the gradients to adaptively adjust the learning rate of each weight parameter (1). After implementing the optimizer, I used a loss function to update the weights of the network. The loss function is a criterion used for Mean Square Error which is nothing but,

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

1) Train step: The main purpose here is to convert all of our training to PyTorch tensor. The training parameters are state, action, reward, next state, and done. In the conversion process, we just pass each parameter with 'torch.tensor()' with the data type. Like, state = torch.tensor(state, dtype=torch.float) These values eventually get larger and become the (n, x) shape. In PyTorch, the unsqueeze function is used to add a new dimension to a tensor at the specified position (3). Here, the unsqueeze function is used to add a new dimension of size 1 to the tensors state, next state, action, reward, and done at position 0. This is done to ensure that the tensors have the correct shape for further processing in the DQN model.

```
if len(state.shape) == 1:
state = torch.unsqueeze(state, 0)
```

Similarly, for other parameters.

D. Q Value and Bellman Equation

The predicted Q-Value with the current state is,

```
pred = self.model(state)
target = pred.clone()
```

Now we apply the Bellman equation in our Python programming and run it in the model. As we already the equation is, $Q_{new} = r + \gamma * max(next_predictedQvalue)$

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

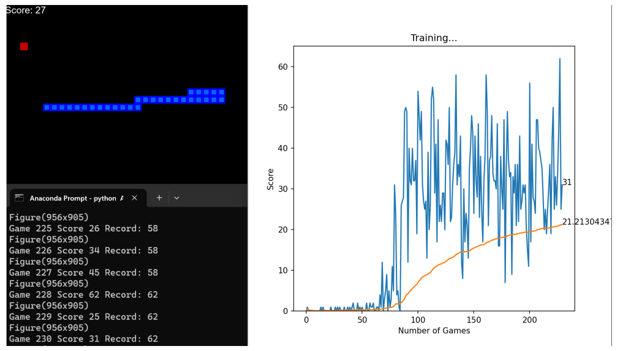
Here, a copy of the predicted Q-values pred is created and assigned it to the variable target. It then iterates over each sample in the batch using a for loop. For each sample, it calculates the new Q-value 'Qnew' and as the immediate reward[idx]. Now, I put the condition, If the episode is not done, it calculates the maximum Q-value of all possible actions in the next state using the torch.max part and adds it to the immediate reward multiplied by the discount factor gamma. Finally, it updates the Q-value of the selected action using the last part of the code. In PyTorch, we use an optimizer to empty the gradient. We have the predicted Q-value as 'Pred' and the Qnew value as 'Target'. So, we calculate the loss function using, $Loss = (Q_{new} - Q)^2$. In the end, I applied back-propagation and updated the step.

```
loss = self.criterion(target, pred)
loss.backward()
self.optimizer.step()
```

[12]

VII. Training and Testing

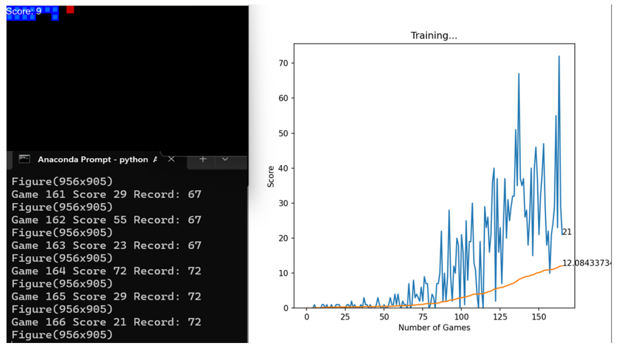
[After the first training, I got the following result in Figure 7.



```
Figure(956x905)
Game 225 Score 26 Record: 58
Figure(956x905)
Game 226 Score 34 Record: 58
Figure(956x905)
Game 227 Score 45 Record: 58
Figure(956x905)
Game 228 Score 62 Record: 62
Figure(956x905)
Game 229 Score 25 Record: 62
Figure(956x905)
Game 230 Score 31 Record: 62
```

Fig. 6. 1st Training

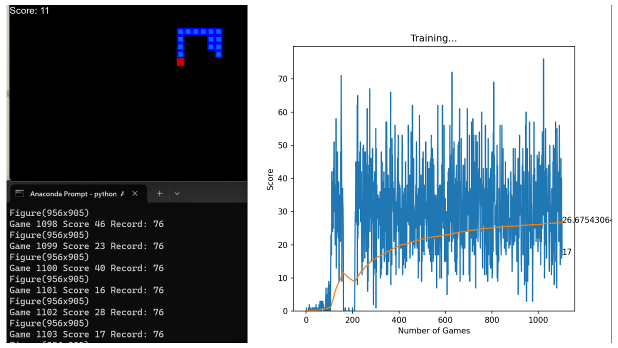
In Figure 7, I implemented another trick by adding another condition to avoid self-collision, which gave me great results. The highest score I achieved here is 72 by training only 164 games. Finally, in Figure 8, I made the model more greedy by



```
Figure(956x905)
Game 161 Score 29 Record: 67
Figure(956x905)
Game 162 Score 55 Record: 67
Figure(956x905)
Game 163 Score 23 Record: 67
Figure(956x905)
Game 164 Score 72 Record: 72
Figure(956x905)
Game 165 Score 29 Record: 72
Figure(956x905)
Game 166 Score 21 Record: 72
Figure(956x905)
```

Fig. 7. 2nd Training

implementing rewards, if it goes near the food and a Euclidean distance formula to avoid self-collision. So, I was able to get the highest score possible in my environment after a lot of trials which is 76.



```
Figure(956x905)
Game 1098 Score 46 Record: 76
Figure(956x905)
Game 1099 Score 23 Record: 76
Figure(956x905)
Game 1100 Score 40 Record: 76
Figure(956x905)
Game 1101 Score 16 Record: 76
Figure(956x905)
Game 1102 Score 28 Record: 76
Figure(956x905)
Game 1103 Score 17 Record: 76
```

Fig. 8. 3rd Traing

Though I got the highest score on the third test, the average score was not stable. So, I considered keeping the implementation of the second test(Fig. 7) which gives stable scores throughout the testing.]

Conclusions

The implementation of the automatic snake game using deep learning and Python represented a significant advancement in the field of AI game development. By utilizing the deep reinforcement learning algorithm, the automatic snake game demonstrated the ability to create intelligent and adaptive gameplay. Through this implementation, I understood that any kind of game can be solved. To solve the automatic snake gameplay, my first main goal was to make the environment and policies. Then I fed them together in the deep learning model model with a formula called the Bellman equation. When the training started, my deep Q-learning model started to collect the successful batches of trial states and kept on following the best Q-values. There were two kinds of challenges I faced during this implementation. The first face was to make the actual game. Thanks to the advanced data sets available on the internet which helped me solve every error step by step. Then the second face was quite hard. Here my approach was to make the snake avoid collisions and increase the score as high as possible. I applied various tricks and tested the game day after day to increase even for 1 point. Though looking over the snake playing at 100 speed was quite hallucinating, what I discovered was incredible. Deep learning implementation in game development was started in the 1950s and never stopped. Nowadays DL algorithms are very common in every game to solve unthinkable problems which made it possible for the game industries to compete with other tech companies.

References

[1] H. Sarmah, "Timeline of games mastered by artificial intelligence," *Analytics India Magazine*, 12/07/2019. [Online]. Available: https://analyticsindiamag.com/timeline-of-games-mastered-by-artificial-intelligence/
[2] Rokas Vaivada, "How deep learning is revolutionizing the gaming industry — linkedin," 2023. [Online]. Available: https://www.linkedin.com/pulse/how-deep-learning-revolutionizing-gaming-industry-rokas-vaivada/
[3] Andrey Kurenkov's Web World, "A 'brief' history of game ai up to alphago," 21/06/2022. [Online]. Available: https://www.andreykurenkov.com/writing/ai/a-brief-history-of-game-ai/
[4] A. Zhang, *Dive into deep learning*. Cambridge, UK: Cambridge University Press, 2023.
[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, ser. Adaptive computation and machine learning. Cambridge Massachusetts: The MIT Press, 2016.
[6] Sciforce, "What's next for gans: Latest techniques and applications," *Sciforce*, 12/01/2022. [Online]. Available: https://medium.com/sciforce/whats-next-for-gans-latest-techniques-and-applications-3be06a7e5ab9
[7] M. Zambra, A. Testolin, and M. Zorzi, "A developmental approach for training deep belief networks," *Cognitive Computation*, vol. 15, no. 1, pp. 103–120, 2023.
[8] A. Sarkar, Z. Yang, and S. Cooper, "Controllable level blending between games using variational autoencoders." [Online]. Available: https://arxiv.org/pdf/2002.11869.pdf
[9] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018. [Online]. Available: https://arxiv.org/pdf/1811.12560.pdf
[10] Baeldung on Computer Science, "Q-learning vs. deep q-learning vs. deep q-network — baeldung on computer science," 2023. [Online]. Available: https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network

[11] freeCodeCamp.org, “Diving deeper into reinforcement learning with q-learning,” *freeCodeCamp.org*, 10/04/2018. [Online]. Available: https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/

[12] Patrick Loeber, “Python + pytorch + pygame reinforcement learning – train an ai to play snake,” 2022. [Online]. Available: https://www.youtube.com/watch?v=L8ypSXwyBds