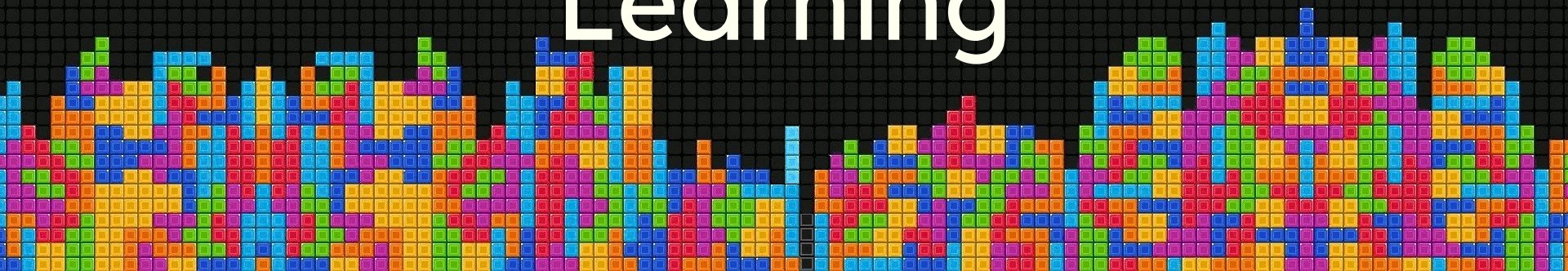




# Deep Reinforcement Learning



## Team Members

1. Emilio Kiryakos
2. Dev Patel
3. Markus Perez
4. Michael Klimek
5. Christopher Bejar

# Project Objective

3

## Primary Objective

- Create base agent that can reach 10,000 pts in under 30,000 episodes
- Use different reward functions, benchmarking and comparing the reward function designs for Tetris

## Secondary Objective

- Create an environment for Tetris using either Tetris Gymnasium or open-source Tetris programs and collect data
- Lay groundwork for future offline reinforcement learning research on Tetris

### Reinforcement Learning with Online Interactions



### Offline Reinforcement Learning





# Refresher on Reinforcement Learning

# Reinforcement Learning

5

## Reinforcement Learning

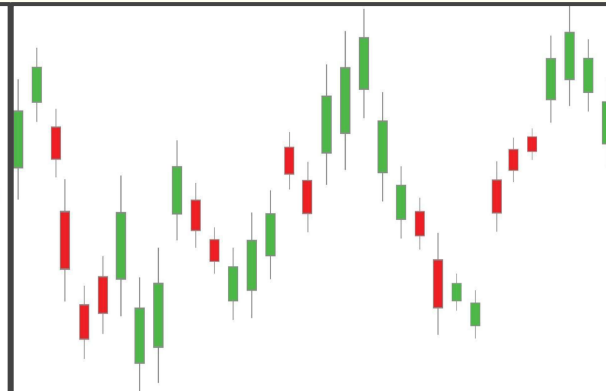
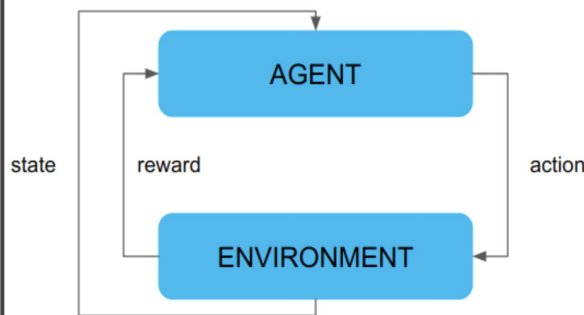
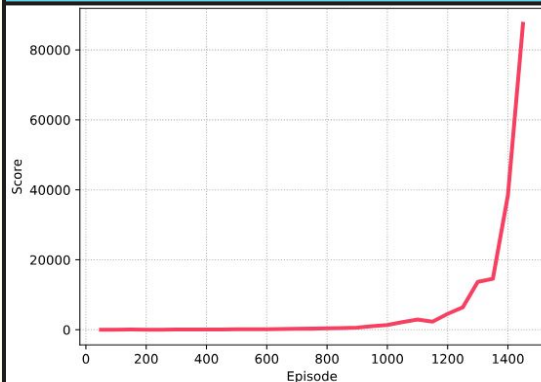
- Focuses on the study of intelligent agent behavior with the goal of reward maximization
- Rewards are scores accumulated from a reward function that rates different behavior

## Steps

1. Create Agent & Rewards for Agent
2. Let Agent use the environment and learn from its Rewards
3. Fine tune to get desired results

# Reinforcement Learning

6



## Goal-Oriented

Reinforcement Learning focuses on improving cumulative rewards while training. As the agent improves, the reward values increase exponentially

## Feedback Driven

Reinforcement Learning agents have an action space that includes all of choices it could make. It decides on which decision to make through learning done on previous actions and their results

## Real World Uses

There are many uses for reinforcement learning among vast industries. Some examples include NLP, autonomous vehicles, financial trading, or personalized recommending.

# Traditional Q-Learning

7

## Overview of traditional Q-Learning algorithm

Maintains a table of state-action value estimates (Q-values) and a list of action rewards

Updates Q-values using temporal difference learning or Bellman's equation

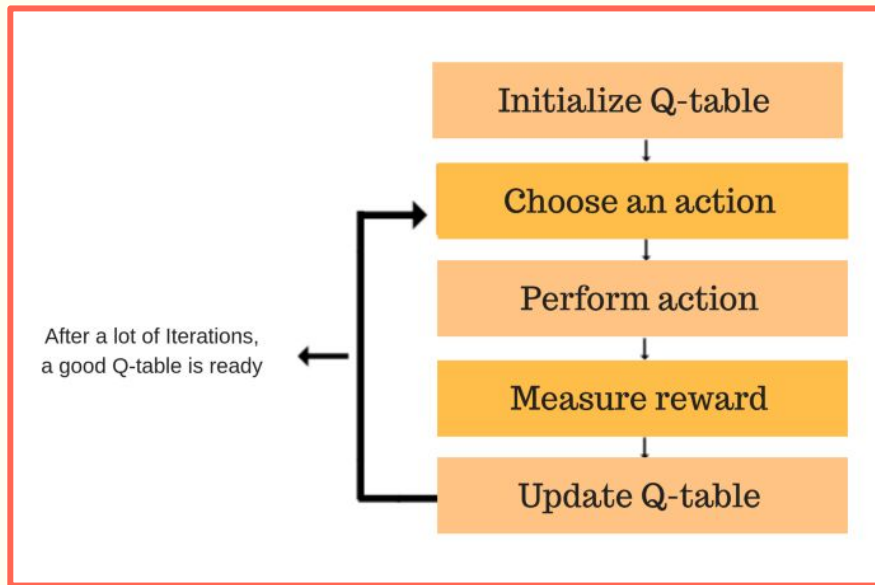
Selects actions using an exploration-exploitation strategy (e.g. epsilon-greedy)

## Limitations of traditional Q-Learning

Unable to scale to large state spaces due to the curse of dimensionality

Requires discretization of the state and action spaces

May struggle to generalize to unseen states



$$Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma Q(S', A'))$$

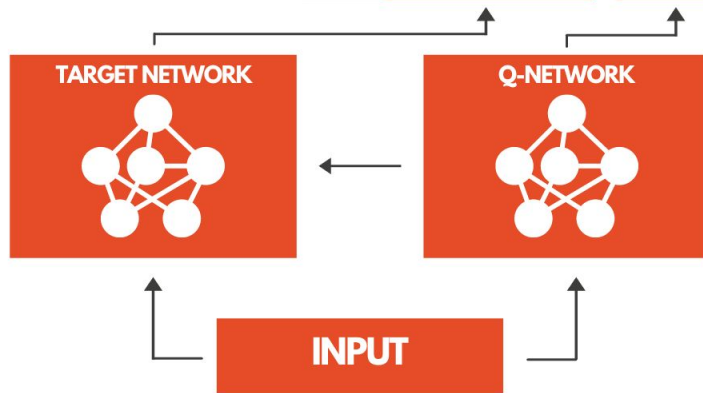
# Deep Q-Learning

8

## Deep Q-Learning Algorithm

- Utilizes Deep Neural Networks as function approximation of aforementioned Q table
- Takes advantage of NNs but is not a model itself

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$





# Technical Approach: Deep Q-Learning

9

## Core Components

- Deep Neural Networks
- $\epsilon$ -Greedy Action Selection

## Environment

- Gymnasium
- Open-Source Tetris Game
- PyTorch for Deep Learning

## Key Data Metrics

- Loss between predicted and actual Q-values
- Reward, per episode and cumulative

# Project Timeline

10

## Setup Phase

- VM Setup
- Environment Configuration
- Initial Testing
- Research

## Development Phase

- Implement DQN
- Training Algorithms
- Experiment with different network architectures
- Debug issues

## Final Phase

- Multiple Training Runs
- Tune parameters
- Data Analysis
- Results Compilation

# DeepQNetwork.py

11

## DQN Layering

Our Deep Q-Learning network follows a FCNN architecture. Meaning it processes data using three fully connected neural layers.

This architecture works for Tetris because of its ability to comprehend and focus on a small amount of metrics, like height of stacks, total points, and lines cleared.

```
def __init__(self, state_dim, action_dim):
    super(DQN, self).__init__()
    self.model = nn.Sequential(
        #input
        nn.Linear(state_dim, state_dim//2),
        nn.ReLU(),

        #hidden layer 1
        nn.Linear(state_dim//2, state_dim//4),
        nn.ReLU(),

        #hidden layer 2
        nn.Linear(state_dim//4, state_dim//8),
        nn.ReLU(),

        #output layer
        nn.Linear(state_dim//8, action_dim)
    )
```

## Implementation

The network inputs a dimension of 944 and eventually outputs 8 Q-values from which the model chooses from when decision-making.

The network uses a ReLu function for activation. Since it helps simplify the more complex aspects of tetris and is computationally efficient.

# main.py *Training*

12

## Epsilon-Greedy Exploration

The model balances and tries exploration (random actions) and exploitation (Q-value based actions) using epsilon-greedy decay.

## Batch Training

The most recent samples from memory are used to update the model by minimizing the difference between predicted and target Q-values through training.

## Buffers and Run Time

Our agent trained for a set amount of games. We set this number to be 4000 attempts. The agent uses the first 600 episodes as a buffer to collect data and begins training after.

```
while episode < self.episodes:
    # get all possible actions and states
    next_moves = env.get_next_states()
    next_actions, next_states = zip(*next_moves.items())
    next_states = torch.stack(next_states)

    # update epsilon
    epsilon = max(self.epsilon_min, self.epsilon - (self.epsilon_decay * episode))
    epsilons.append(epsilon)

    # predict the q-values for the next states
    model.eval()
    with torch.no_grad():
        predictions = model(next_states)[: , 0]

    # choose an action using epsilon-greedy
    if random() < epsilon:
        index = next_actions.index(sample(next_actions, 1)[0]) # Random action
    else:
        index = torch.argmax(predictions).item() # Greedy action

    # take the action
    action = next_actions[index]
    next_state = next_states[index, :]
    reward, done = env.step(action, render=True)
    rewards.append(reward)
    cum_reward += reward

    # Store the experience in memory
    memory.append([state, reward, next_state, done])
```

# Training Progress and Results

13

## Challenges During Training

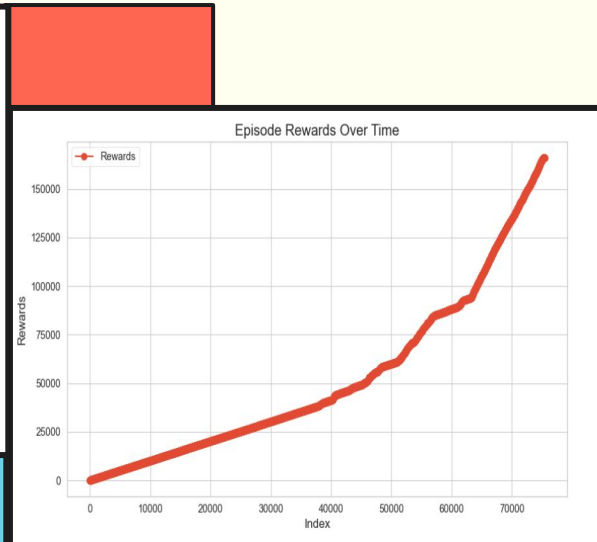
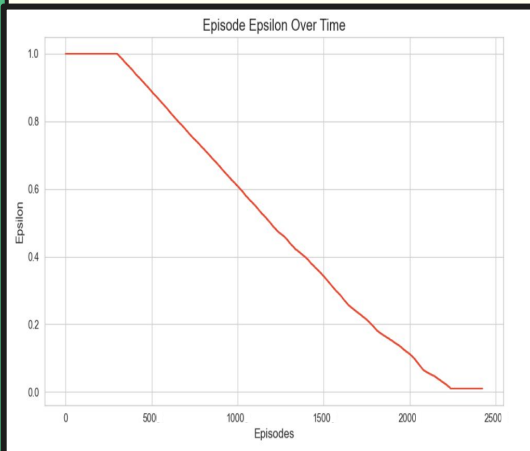
After an initial lack of progress, we experimented with tuning hyperparameters like batch sizes, learning rate, epsilon-decay, memory size, and length to limited success.

## Environment Experiments

The reward system of the Tetris gymnasium environment was preventing successful training, the model was switched to a custom reinforcement learning Tetris program.

## Training Success

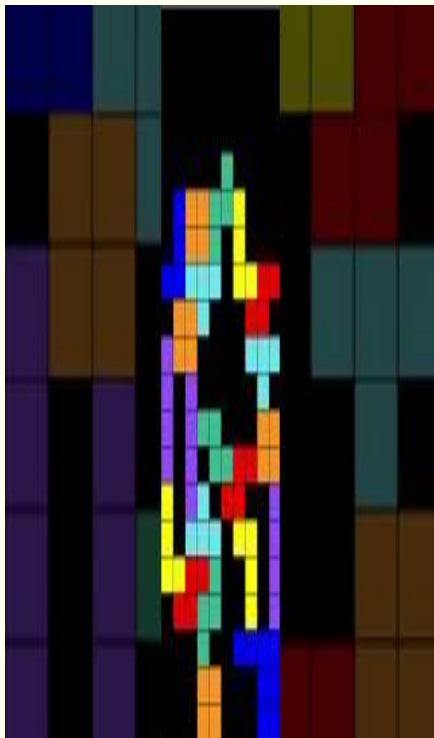
Eventually after tuning and structural changes, we were able to successfully train the model to play Tetris. The model's record being over 16000 points and 800 lines cleared and took only ~2200 episodes.



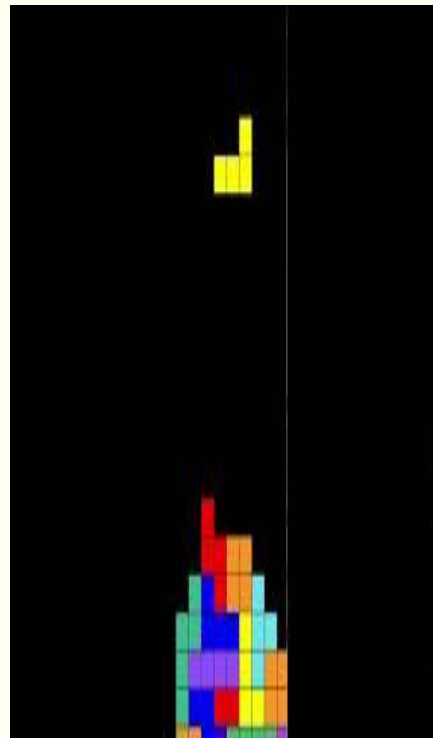
# Project Results

14

Before Training



After Training



- ❑ **Alignment with Goal:**
  - ❑ Our base agent was able to learn an effective Tetris playing strategy and achieve higher scores than a random agent or average human
  - ❑ Comparison of reward functions is in-progress
- ❑ **Future Improvements:** Explore alternative network architectures, hyperparameter tuning, and incorporation of more advanced RL techniques like double DQN or dueling DQN

# Conclusion

16

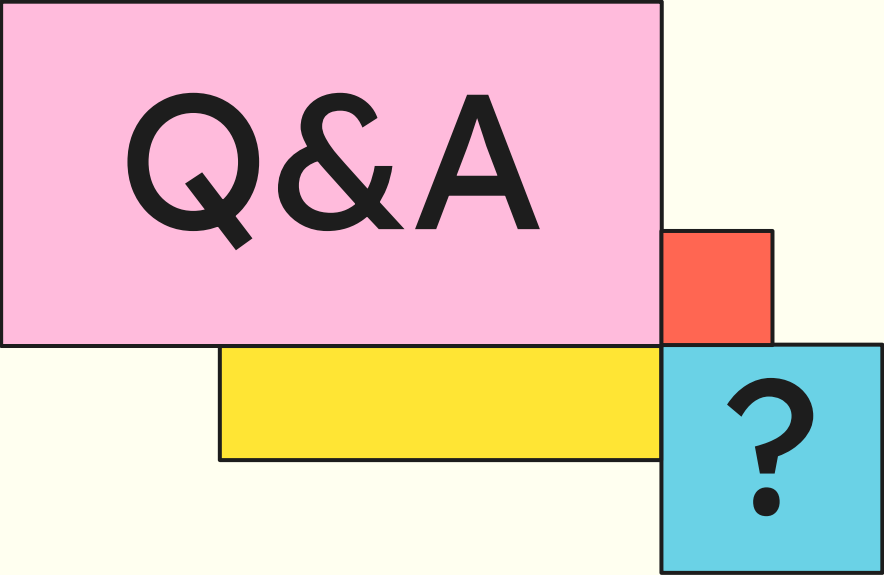
## **Key Takeaways:**

- Demonstrated the effectiveness of deep reinforcement learning for solving complex sequential decision-making problems
- Learned that careful feature engineering and network design are crucial for achieving good performance

## **Next Steps:**

- Complete study of reward functions
- Adapting the model to different games/applications.
- Implement early stopping to prevent overfitting.
- Record state, action, and rewards for others' future study





Q&A

?