SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Department of Computer Science and Engineering

(Artificial Intelligence and Machine Learning)

Neural Networks And Deep Learning (21AI72)

Report on

Policy Gradients and Learning to Play Pacman using Deep Q-Learning

Prepared by

Anurag Poojary	4SF21AD008
Sathya Shravan	4SF21AD013
Rayson M fernandies	4SF21AD043
Shashank SK	4SF21AD048

Under the Guidance of

Dr. Gurusiddayya Hiremath

Associate Professor

Department of Computer Science and Engineering

(Artificial Intelligence and Machine Learning)

SCEM, Mangaluru

Academic Year: 2024-25

Abstract

This report explores the concepts of policy gradients and deep Q-learning, presenting their applications, advantages, and challenges in reinforcement learning. Special attention is given to their use in training an AI agent to play Pacman.

1 Introduction to Policy Gradients

Policy gradients are a class of reinforcement learning algorithms where the policy is optimized directly.

- **Definition**: Direct optimization of the policy function to maximize expected reward.
- **Applications**: Robotics, autonomous vehicles, game playing.
- **Advantages**: Effective in high-dimensional or continuous action spaces and allows direct optimization of performance metrics.

2 Policy Gradients - Key Concepts

- **Policy**: Determines the action to take in a given state.
- **Gradient**: Measures how the policy parameters influence performance.
- **Goal**: Maximize expected reward by adjusting policy parameters.
- **Algorithms**: Common methods include REINFORCE, Proximal Policy Optimization (PPO), and Actor-Critic Methods.

3 Steps in Policy Gradient Methods

1. Initialize policy parameters.
2. Generate trajectories by interacting with the environment.
3. Compute rewards for each trajectory.
4. Calculate the policy gradient.
5. Update policy parameters using gradient ascent.

4 Challenges in Policy Gradients

- High variance in gradient estimates.
- Sensitive to hyperparameters.
- Requires careful balancing of exploration and exploitation.

5 Learning to Play Pacman with Deep Q-Learning

- **Deep Q-Learning (DQL)**: Combines Q-Learning with deep neural networks to approximate Q-values.
- **Goal**: Train an AI agent to play Pacman by learning optimal actions for maximum rewards.
- **Key Idea**: Use a neural network to approximate the Q-function, which predicts the expected reward of actions.
- **Experience Replay**: Store and sample past experiences to stabilize training. This allows the model to learn from a diverse set of experiences.

6 Game Environment for Reinforcement Learning

To train an RL agent to play Pacman, we first need to define the game environment and implement the necessary game logic.

6.1 Choose a Game Engine

Popular options include:

- **Unity**: A widely used game engine suitable for both 2D and 3D games. Unity supports reinforcement learning integration through frameworks like ML-Agents.
- **Pygame**: A simpler, Python-based game engine suitable for creating 2D games. It is easy to implement and is often used for educational purposes.
- **Custom-built environments**: For more specific or tailored environments, one can build a custom game engine suited to the problem at hand.

6.2 Define the Game State

The game state can be represented by a set of variables that describe the key elements of the game:

- **Pacman's position**: Coordinates of Pacman on the grid.
- **Ghost positions**: Coordinates of the ghosts on the grid.
- **Pellet locations**: Positions of the remaining pellets or power pellets.
- **Score**: The current score based on the collected pellets and eaten ghosts.
- **Lives**: The number of remaining lives for the agent.

6.3 Implement Game Logic

The game logic defines how the environment behaves based on the actions taken by the agent:

- **Movement**: Define how Pacman moves in response to actions (up, down, left, right).
- **Collision Detection**: Determine when Pacman collides with walls, ghosts, or pellets.
- **Score Calculation**: Increase the score when Pacman collects pellets or eats ghosts.
- **Game Over Conditions**: The game ends when Pacman loses all lives or completes all levels.

7 Reinforcement Learning Algorithm Implementation

The key to training an AI agent to play Pacman is using a reinforcement learning algorithm like Policy Gradients or Deep Q-Learning.

7.1 Policy Gradients

- **Policy Network**: Define a neural network that outputs action probabilities based on the current state.
- **Training**: Use gradient ascent to adjust the network weights to maximize the expected reward. The network learns to take the best actions based on the environment's feedback.
- **Loss Function**: A loss function is used to measure the difference between expected rewards and actual rewards.

7.2 Deep Q-Learning

- **Q-Network**: Define a neural network that approximates the Q-function. This function predicts the expected future reward for each state-action pair.
- **Training**: Use a loss function to minimize the difference between the predicted Q-value and the target Q-value. The target Q-value is calculated using the Bellman equation.
- **Experience Replay**: Implement experience replay by storing past experiences and sampling them to update the Q-network. This helps stabilize training and improves sample efficiency.

In [ ]:

```python
import pygame
import tensorflow as tf

# Define the game state
class GameState:
    def __init__(self):
        # Initialize game variables
        self.pacman_pos = (10, 10)
        self.ghost_pos = [(20, 20), (30, 30)]
        # ...

    def update(self, action):
        # Update game state based on the action
        if action == "UP":
            self.pacman_pos = (self.pacman_pos[0], self.pacman_pos[1] - 1)
        # ...

# Define the policy network
model = tf.keras.Sequential([
    # Input layer
    tf.keras.layers.Dense(64, activation='relu'),
    # Hidden layers
    tf.keras.layers.Dense(32, activation='relu'),
    # Output layer (action probabilities)
    tf.keras.layers.Dense(4, activation='softmax')
])

# Define the training loop
def train(env, model, optimizer):
    for episode in range(1000):
        state = env.reset()
        done = False
        while not done:
            # Get action probabilities
            action_probs = model(tf.expand_dims(state, 0))
            # ... (sample action, take action, update state, calculate reward)
            # ... (update model parameters using gradient ascent)

# Main game loop
def main():
    pygame.init()
    # ... (initialize Pygame window, etc.)

    while True:
        # Get current state
        state = env.get_state()
        # Get action from model
        action = model.predict(state)
        # Update game state
        env.update(action)
        # Render the game
        pygame.display.flip()

if __name__ == "__main__":
    main()
```

Figure 1: code implementation

8 Conclusion

- **Policy Gradients**: Powerful for continuous action spaces and optimizing complex objectives.
- **Deep Q-Learning**: Effective for discrete action problems like Pacman.
- **Future Work**: Develop hybrid models combining policy gradients and value-based methods for enhanced performance.

References

- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

- Casas, N. "Deep deterministic policy gradient for urban traffic light control. arXiv 2017." arXiv preprint arXiv:1703.09035.