COLOMBIA POTENCIA DE LA VIDA

TIC

Actividad 3

Algoritmos de Aprendizaje por Refuerzo

TALENTO TECH

AZ | PROYECTOS EDUCATIVOS

UTP Universidad Tecnológica de Pereira

Algoritmos de Aprendizaje por Refuerzo

El módulo de Algoritmos de Aprendizaje por Refuerzo introduce a los estudiantes a los conceptos fundamentales de la Inteligencia Artificial orientada a la toma de decisiones. Comienza con una visión general de los principales algoritmos de Reinforcement Learning (RL), incluyendo Q-Learning, Sarsa y la Política de Gradiente de Montecarlo. Se explora en detalle el funcionamiento de Q-Learning, centrándose en cómo el algoritmo aprende la función Q-valor y toma decisiones basadas en ella. Luego, se profundiza en Sarsa, otro algoritmo popular de RL, comparándolo con Q-Learning y destacando sus diferencias clave. Por último, se introduce la Política de Gradiente de Montecarlo como una técnica de optimización basada en gradientes que busca mejorar el rendimiento de los agentes de RL en entornos complejos. Este módulo proporciona una base sólida para que los estudiantes comprendan y apliquen los principios y técnicas del Aprendizaje por Refuerzo en una variedad de aplicaciones prácticas.

Algoritmos de Aprendizaje por Refuerzo

El aprendizaje por refuerzo (RL) es una rama del aprendizaje automático donde los agentes aprenden a tomar decisiones óptimas a través de la interacción con un entorno. Este conjunto de técnicas es fundamental para situaciones en las que un agente toma acciones en un ambiente para maximizar una recompensa acumulativa a lo largo del tiempo.

01 02 03 04

Política de gradiente de Montecarlo

La política de gradiente de Montecarlo es una técnica de optimización basada en gradientes que se utiliza para aprender directamente políticas en lugar de funciones de valor. Utiliza un enfoque de gradiente ascendente para actualizar los parámetros de la política con el objetivo de maximizar la recompensa esperada.

Sarsa

Sarsa es otro algoritmo de RL que sigue una estrategia de aprendizaje on-policy, lo que significa que toma decisiones basadas en una política actualizada en cada paso. A diferencia de Q-Learning, Sarsa actualiza su función de valor de acción basada en la política seguida, lo que lo hace más conservador y adecuado para entornos donde la exploración puede ser costosa.

Q-Learning

Q-Learning es uno de los algoritmos de RL más conocidos y utilizados. Se basa en la idea de aprender una función de valor de acción, llamada Q-valor, que estima la recompensa acumulada esperada al realizar una acción en un estado dado. A través de la exploración y la explotación, el agente actualiza iterativamente sus estimaciones de Q-valor utilizando la ecuación de Bellman.

Introducción a los principales algoritmos de RL

Los algoritmos de RL se basan en el concepto de aprender de la experiencia mediante la interacción con un entorno, buscando maximizar la recompensa acumulada a largo plazo. Estos algoritmos son esenciales en la toma de decisiones secuenciales, como juegos, robótica, control de procesos, entre otros.

Ejercicios de Aprendizaje por Refuerzo en Python

```python
# Ejercicio 1: Introducción a los principales algoritmos de RL

# Define el entorno del juego

class Environment:

      def __init__(self):

      self.state_space = [0, 1, 2, 3]  # Estados posibles

      self.action_space = [0, 1]         # Acciones posibles

      self.rewards = {0: -1, 1: -1, 2: -1, 3: 10}  # Recompensas por
estado


# Crea una instancia del entorno

env = Environment()


# Muestra información del entorno

print("Estados:", env.state_space)

print("Acciones:", env.action_space)

print("Recompensas:", env.rewards)
```

1. Introducción a los principales algoritmos de RL:

Genera un entorno de juego simple donde un agente debe aprender a navegar y recolectar objetos. Define estados, acciones y recompensas para el agente.

Ejercicios de Aprendizaje por Refuerzo en Python

2. Q-Learning:

Implementa el algoritmo Q-Learning para que un agente aprenda a navegar y recolectar objetos en el entorno definido. Muestra cómo se actualiza la función Q-valor.

```
# Ejercicio 2: Q-Learning
import numpy as np
# Inicializa la tabla Q con valores arbitrarios
Q = np.zeros((len(env.state_space), len(env.action_space)))

# Define los parámetros del algoritmo
alpha = 0.1  # Tasa de aprendizaje
gamma = 0.9  # Factor de descuento

# Entrena el agente utilizando Q-Learning
for _ in range(1000):
      state = np.random.choice(env.state_space)  # Estado inicial
aleatorio
      while state != 3:  # Hasta llegar al estado objetivo
      action = np.random.choice(env.action_space)  # Selecciona una
acción aleatoria
      next_state = state + action
      reward = env.rewards[next_state]
      Q[state, action] = Q[state, action] + alpha * (reward + gamma *
np.max(Q[next_state]) - Q[state, action])
      state = next_state

# Muestra la función Q-valor aprendida
print("Función Q-Valor aprendida:")
print(Q)
```

Ejercicios de Aprendizaje por Refuerzo en Python

3. Sarsa:

Implementa el algoritmo Sarsa para compararlo con Q-Learning en el mismo entorno. Muestra cómo se actualiza la función Q-valor y compara los resultados.

```
# Ejercicio 3: Sarsa
# Reinicializa la tabla Q con valores arbitrarios
Q = np.zeros((len(env.state_space), len(env.action_space)))

# Entrena el agente utilizando Sarsa
for _ in range(1000):
    state = np.random.choice(env.state_space)  # Estado inicial aleatorio
    action = np.random.choice(env.action_space)  # Selecciona una acción aleatoria
    while state != 3:  # Hasta llegar al estado objetivo
    next_state = state + action
    next_action = np.random.choice(env.action_space)  # Selecciona una acción aleatoria
    reward = env.rewards[next_state]
    Q[state, action] = Q[state, action] + alpha * (reward + gamma * Q[next_state, next_action] - Q[state, action])
    state = next_state
    action = next_action

# Muestra la función Q-valor aprendida con Sarsa
print("Función Q-Valor aprendida con Sarsa:")
print(Q)
```

Ejercicios de Aprendizaje por Refuerzo en Python

4. Política de Gradiente de Montecarlo:

Implementa la técnica de optimización basada en gradientes para aprender una política en el mismo entorno. Muestra cómo se actualizan los parámetros de la política utilizando el gradiente ascendente.

```
# Ejercicio 4: Política de Gradiente de Montecarlo
# Inicializa la política con probabilidades uniformes
policy = np.ones((len(env.state_space), len(env.action_space))) /
len(env.action_space)

# Define la función de recompensa promedio
def average_reward(Q):
    return np.mean([Q[state, np.argmax(policy[state])] for state in
env.state_space])

# Entrena la política utilizando Gradiente de Montecarlo
for _ in range(1000):
    state = np.random.choice(env.state_space)  # Estado inicial
aleatorio
    while state != 3:  # Hasta llegar al estado objetivo
    action = np.random.choice(env.action_space, p=policy[state])  #
Selecciona una acción con la política actual
    next_state = state + action
    reward = env.rewards[next_state]
    gradient = np.zeros_like(policy[state])
    gradient[action] = 1
    alpha = 0.01  # Tasa de aprendizaje
    policy[state] += alpha * gradient * (reward - average_reward(Q))
    state = next_state

# Muestra la política aprendida
print("Política aprendida con Gradiente de Montecarlo:")
print(policy)
```

COLOMBIA POTENCIA DE LA VIDA

TIC

TALENTO TECH

AZ | PROYECTOS EDUCATIVOS

UTP Universidad Tecnológica de Pereira