Centro Universitario de Ciencias Exactas e Ingenierías

Licenciatura en Ingeniería en Computación

Materia: Seminario de Solución de Problemas de Inteligencia Artificial I. Clave: I7039.

Profesor: Sencion Echauri Felipe

Estudiante: Silva Moya José Alejandro. Código: 213546894.

Estudiante: Plascencia Camarillo Moisés Rafael. Código: 216788171

Proyecto Final: Enseñando a una IA a jugar Cartpole mediante el uso de machine learning y redes neuronales con Deep Q Learning y OpenAI Gym.

Índice

Problemática seleccionada

En el presente proyecto se expone la solución de un problema sencillo mediante machine learning (con la rama conocida como Deep Q Learning) para enseñarle a un ente artificial a jugar y terminar el antiguo juego llamado “Catpole”, que consiste en la simulación de sostener en equilibrio una vara sobre una superficie plana.

Consideramos dicho juego como una excelente opción, ya que representa un ambiente controlado con una cantidad relativa de pocos factores, y habilidades de movimiento reducidas a dos: movimientos de izquierda a derecha por parte del agente para poder mantener la vara en equilibrio. Esto, si bien quizás no representa una problemática muy cotidiana, sí abre campo a las bases del aprendizaje máquina, que en mayores desarrollos podría sencillamente tener aplicaciones en todo tipo de industria y control de artefactos, por lo que su relevancia potencial es bastante alta.

Así pues, fue necesario comenzar por aprender desde las bases de la inteligencia artificial (visto en la materia durante el semestre), para pasar a redes neuronales y con ello al aprendizaje máquina.

Algoritmo elegido

Como se mencionó con anterioridad, el problema presentado en el actual proyecto fue desarrollado y resuelto con redes neuronales, que siguen el principio conocido como Deep Q Learning; que es una rama del machine learning.

Q Learning es una técnica de aprendizaje por refuerzo (Reinforcement Learning) utilizada en el aprendizaje automático o aprendizaje máquina. Su objetivo es aprender una serie de normas que le digan a un agente qué acciones debe tomar bajo ciertas circunstancias. Por estos principios, no requiere un modelo del entorno (conocimiento previo), y puede manejar problemas con transiciones estocásticas y recompensas sin requerir de adaptaciones.

Para cualquier proceso de decisión de Markov finito, Q Learning encuentra una política óptima en el sentido de que maximiza el valor esperado de la recompensa total sobre todos los pasos sucesivos, empezando desde el estado actual. Así, Q Learning puede identificar una norma de acción – selección óptima para cualquier proceso de decisión de Markov finito., dado un tiempo de exploración infinito y una norma parcialmente aleatoria. “Q” nombra la función que devuelve la recompensa que proporciona el refuerzo y representa la “calidad” de una acción tomada en un determinado estado.

¿Cómo se resolvió la problemática por medio del algoritmo seleccionado?

Considerando las bases que tenemos para la construcción de este programa, resulta completamente importante mencionar el uso de redes neuronales y aprendizaje de máquina como los principales exponentes de la solución al problema en cuestión.

Una vez aplicadas las redes neuronales y el agente dentro de un ambiente controlado, la solución se fue dando como en cualquier otro sistema básico de aprendizaje máquina: al agente se le proporcionaron herramientas y un sistema de recompensa y castigo, para que él mismo fuera tomando decisiones y corrigiendo las mismas conforma a los puntos obtenidos. Sin embargo, esto no es tan sencillo como el código que se expondrá (que consta de menos de 100 líneas de código y está desarrollado en Python), ya que, si bien es sencillo de entender dentro de lo que cabe, debemos tomar en cuenta que para entenderlo realmente, resulta completamente necesario tener conocimientos previos de inteligencia artificial, de las librerías utilizadas para el desarrollo del código, de redes neuronales y de machine learning.

Desarrollo del Programa

Comencemos por las librerías que el programa necesita para poder ser ejecutado.

Para que todo funcione de manera correcta, el programa debe ser ejecutado con Python 3.6.x o inferior, debido a que una de las librerías implementadas es Tensorflow, y al momento en que se realiza este proyecto y reporte, aún no existe una versión de la misma para Python 3.7. Si en el futuro se libera una versión adecuada, el programa debería poder funcionar entonces.

Así pues, los módulos que son necesarios instalar en Python para este programa son:

- Matplotlib
- Keras
- Tensorflow
- Gym
- Numpy

Observemos las primeras líneas del código.

```
1  import random
2  import gym
3  import numpy as np
4  from collections import deque
5  from keras.models import Sequential
6  from keras.layers import Dense
7  from keras.optimizers import Adam
```

En este caso lo que podemos ver son:

- El módulo aleatorio, para la aleatoriedad controlada que requiere normalmente cualquier inteligencia artificial.

- Gym, que proviene de OpenAI, como fuente del juego sin la inteligencia artificial, de manera que podemos manejar todo dentro de un ambiente pre-generado y mucho más controlado.
- Numpy para ciertos cálculos y trabajo con TDA's.
- Collections para trabajar más eficientemente con listas o arrays.
- En este caso se utiliza el modelo secuencial de keras para el individuo. Existen más modelos con diferentes características y pueden ser estudiados y entendidos en la documentación oficial de la librería.
- El tipo de capa que se utilizará para la red neuronal será secuencial; nuevamente, se puede estudiar a fondo en la documentación oficial y analizar más opciones.
- Finalmente, el tipo de optimizador a usar en este caso será Adam. Adam es uno de los modelos más eficientes y versátiles para estos casos, y como en los dos puntos anteriores, se puede estudiar más detenidamente en los documentos oficiales y comparar características.

```
 9   ENV_NAME = "CartPole-v1"
10
11   GAMMA = 0.95
12   LEARNING_RATE = 0.001
13
14   MEMORY_SIZE = 1000000
15   BATCH_SIZE = 20
16
17   EXPLORATION_MAX = 1.0
18   EXPLORATION_MIN = 0.01
19   EXPLORATION_DECAY = 0.995
```

A continuación, tenemos los principales factores a usar durante toda la ejecución del programa, como lo son el factor Gamma, necesario para el algoritmo de Q Learning, la proporción de aprendizaje, una memoria total y una memoria para cada iteración o momento de toma de decisión, y los límites de exploración.

```
22  class DQNSolver:
23
24      def __init__(self, observation_space, action_space):
25          self.exploration_rate = EXPLORATION_MAX
26
27          self.action_space = action_space
28          self.memory = deque(maxlen=MEMORY_SIZE)
29
30          self.model = Sequential()
31          self.model.add(Dense(24, input_shape=(observation_space,), activation="relu"))
32          self.model.add(Dense(24, activation="relu"))
33          self.model.add(Dense(self.action_space, activation="linear"))
34          self.model.compile(loss="mse", optimizer=Adam(lr=LEARNING_RATE))
```

Comencemos con el algoritmo propiamente.

Para la inicialización del proceso, comenzaremos creando toda la red neuronal. En este caso tenemos una red neuronal con espacio de acción, proporción de exploración y memoria ya predefinidos. Anteriormente dijimos que el modelo sería secuencial, y configuramos una red neuronal con las siguientes capas:

- Una capa de entrada que toma como entrada un arreglo de tamaño posible de 0 hasta “observation_space”, que en nuestro caso cuenta con 4 datos: límites de posición de agente, límites de velocidad de movimiento del agente, límites del ángulo de inclinación de la vara, y la velocidad de la vara en la punta de la misma. Además de esto, el 24 inicial indica que el arreglo resultante a retornar tendrá un tamaño posible desde cero hasta 24.
- La segunda capa es la capa de procesamiento de la inteligencia artificial, que tendrá el mismo valor de retorno, y cuyo valor de entrada no es necesario especificar, ya que se puso explícitamente en la primera capa. El activador de esta capa y de la anterior es “relu”. Para más información, revisar la documentación de Keras.
- Finalmente, una capa de resolución o salida de datos de activación lineal, con respuesta hacia el espacio de acción.
- Finalmente se compila la red neuronal para poder tener listo al agente. Loss, optimizador y learning rate pueden variar, y los primeros dos se pueden buscar en los documentos oficiales, mientras que el learning rate es manualmente editable.

```
36      def remember(self, state, action, reward, next_state, done):
37          self.memory.append((state, action, reward, next_state, done))
38
39      def act(self, state):
40          if np.random.rand() < self.exploration_rate:
41              return random.randrange(self.action_space)
42          q_values = self.model.predict(state)
43          return np.argmax(q_values[0])
```

Para que el agente pueda aprender y recordar sus mejores decisiones, es necesario configurar un método que haga que guarde todos estos datos cada que nosotros lo consideremos relevante.

Para que el agente lleve a cabo sus decisiones, si un número aleatorio es menor que nuestra proporción de exploración, entonces le decimos que aleatoriamente tome una opción de aquellas en su espacio de acción (2: moverse a la izquierda o a la derecha). Y en cada decisión, obtener la predicción de los valores de Q para el posterior mejoramiento del ente. Finalmente, siguiendo el paradigma de Q Learning, obtenemos el valor máximo de resolución obtenido hasta el momento (el mejor).

```
45      def experience_replay(self):
46          if len(self.memory) < BATCH_SIZE:
47              return
48          batch = random.sample(self.memory, BATCH_SIZE)
49          for state, action, reward, state_next, terminal in batch:
50              q_update = reward
51              if not terminal:
52                  q_update = (reward + GAMMA * np.amax(self.model.predict(state_next)[0]))
53              q_values = self.model.predict(state)
54              q_values[0][action] = q_update
55              self.model.fit(state, q_values, verbose=0)
56          self.exploration_rate *= EXPLORATION_DECAY
57          self.exploration_rate = max(EXPLORATION_MIN, self.exploration_rate)
```

Ahora nos encontramos frente a algo conocido como “experience_replay”, en donde procesaremos siempre de la manera más adecuada toda la experiencia (o Q) obtenida por el ente.

Primero necesitamos revisar que no hayamos llegado al límite de memoria asignada a la IA. Después, tomamos una muestra de memoria ya existente con datos, actualizamos la recompensa (ya sea buena o mala), y mientras el juego aún no haya terminado o no haya sido vencido, calcularemos una nueva recompensa mediante la el modelo de predicción de Q Learning, optimizaremos el modelo de la siguiente decisión, y haremos que el ente pueda decidir qué es lo mejor a hacer en el siguiente paso…y entonces aprende por sí mismo.

```
60  def cartpole():
61      env = gym.make(ENV_NAME)
62      observation_space = env.observation_space.shape[0]
63      action_space = env.action_space.n
64      dqn_solver = DQNSolver(observation_space, action_space)
65      run = 0
66      while True:
67          run += 1
68          state = env.reset()
69          state = np.reshape(state, [1, observation_space])
70          step = 0
71          while True:
72              step += 1
73              env.render()
74              action = dqn_solver.act(state)
75              state_next, reward, terminal, info = env.step(action)
76              reward = reward if not terminal else -reward
77              state_next = np.reshape(state_next, [1, observation_space])
78              dqn_solver.remember(state, action, reward, state_next, terminal)
79              state = state_next
80              if terminal:
81                  break
82              dqn_solver.experience_replay()
83
84
85  if __name__ == "__main__":
86      cartpole()
```

Finalmente tenemos en forma la clase del ente inteligente.

Para la inicialización del programa, necesitamos llamar al ambiente de juego desde Gym, que en este caso es Cartpole, inicializamos el espacio de observación y el de acción, e inicializamos la Deep Q Network con dichos datos.

Ahora, solamente ponemos el algoritmo a correr.

En cada iteración limpiamos los valores ya no utilizados de los estados, reajustamos el tamaño del arreglo de estados, activamos al ente mediante el método “render”, hacemos que determine la acción a tomar, analizamos el siguiente estado, la recompensa, si es o no el fin del juego, y el resto de información relevante que el algoritmo arroja en cada iteración.

Mantenemos la recompensa mientras el juego no haya acabado, y la invertimos a negativo en caso contrario.

Regeneramos el tamaño del arreglo del estado siguiente, actualizamos la memoria, y cambiamos de estado de acuerdo con el paso (decisión tomada) hecho. Si terminamos el juego, finalizamos el algoritmo; de otra manera volvemos a procesar todo en un nuevo paso.

Resultados obtenidos

Los resultados obtenidos se pueden observar con mayor facilidad al ejecutar el programa, dado que poner imágenes resultaría confuso, puesto que la ejecución y aprendizaje por parte del ente se realiza en tiempo real, y una captura de pantalla no representaría ninguna prueba clara de nada. Además, existe la ventaja de que, al ser un ambiente controlado con tan sencilla complejidad, al implementar este tipo de algoritmo y aprendizaje, los resultados más óptimos se pueden observar – en la mayoría de las ocasiones – en la meta de los 3 minutos o menos.

Conclusión

Pudimos obtener bastantes conocimientos a partir de este proyecto, y llevar nuestro aprendizaje relativo a la inteligencia artificial mucho más allá, al no quedarnos solo con lo visto en clases, sino propiciar algo mejor al integrar conocimientos más avanzados y lograr entregar un proyecto final funcional más completo.

La materia y todos los temas vistos resultaron ser bastante interesantes, al darle una vertiente bastante distinta a la programación que llevábamos haciendo durante tantos semestres. En lo personal pensamos que es una de las mejores aplicaciones que se le puede dar al conocimiento y trabajo de la programación, ya que los campos de aplicación son tantos, que representa un potencial mejoramiento de toda la calidad de vida, y resolución de todo tipo de problemas.

Con esto, pues, damos por concluida la materia, y esperamos poder desarrollar sistemas y entes mucho más completos, complejos e inteligentes en la siguiente materia de Inteligencia Artificial.

Bibliografía

- Tankala, A. (2019, Noviembre 14). Build your First AI game bot using OpenAI Gym, Keras, TensorFlow in Python. Consultado en Diciembre 8, 2019, de https://medium.com/coinmonks/build-your-first-ai-game-bot-using-openai-gym-keras-tensorflow-in-python-50a4d4296687.
- Janakiraman, H. (2017, Mayo 11). Day 22: How to build an AI Game Bot using OpenAI Gym and Universe. Consultado en Diciembre 8, 2019, de https://www.freecodecamp.org/news/how-to-build-an-ai-game-bot-using-openai-gym-and-universe-f2eb9bfbb40a/.
- Surma, G. (2019, Noviembre 10). Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning). Consultado en Diciembre 8, 2019, de https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288.
- Choudhary, A. (2019, Mayo 6). Introduction to Deep Q-Learning for Reinforcement Learning (in Python). Consultado en Diciembre 8, 2019, de https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.
- Comi, M. (2019, Marzo 24). How to teach an AI to play Games: Deep Reinforcement Learning. Consultado en Diciembre 8, 2019, de https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a.
- Kaleem, N. (2019, Febrero 25). Training an AI to Play OpenAI's Cartpole. Consultado en Diciembre 8, 2019, de https://medium.com/datadriveninvestor/training-an-ai-to-play-openais-cartpole-395e17db847f.
- Choudhary, A. (2019, Mayo 7). Reinforcement Learning: Model Based Planning using Dynamic Programming. Consultado en Diciembre 8, 2019, de https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming/.
- Choudhary, A. (2019, Septiembre 7). Solving the Multi-Armed Bandit Problem from Scratch in Python. Consultado en Diciembre 8, 2019, de https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/?utm_source=blog&utm_medium=introduction-deep-q-learning-python.
- Choudhary, A. (2019, Septiembre 7). Solving the Multi-Armed Bandit Problem from Scratch in Python. Consultado en Diciembre 8, 2019, de https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/?utm_source=blog&utm_medium=introduction-deep-q-learning-python.
- Choudhary, A. (2019, Mayo 6). Tutorial on Monte Carlo Tree Search - The Algorithm Behind AlphaGo. Consultado en Diciembre 8, 2019, de https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/.