Trabajo práctico final Sistemas de Soporte para Celdas Producción Flexible

Deep Q-Learning Flappy Bird

Manuel Luis Llauro. Padrón 95736

Abstract. Este documento relata una síntesis de cómo se construyó el trabajo práctico final de la materia. El mismo consistió de una red neuronal que aprendió a jugar al flappy bird

Keywords: deep Q-learning. "greedy strategy". QTarget. Loss

1 Introducción

El objetivo de este trabajo práctico es construir un sistema con redes de aprendizaje profundo, en TensorFlow y particularmente con la "técnica" conocida como Deep Q-Learning, que sea capaz de ganar una versión hecha en Python del juego Flappy Bird.

2 Formación de la red

Entrada. La red neuronal recibe como parámetros de entrada un llamado 'state' o estado puntual del juego. Este mismo está compuesto de 4 sucesivas imágenes del juego para que la red entienda el movimiento que está teniendo el parajo para un estado determinado.

Estas imágenes fueron previamente procesadas antes de entrar en la red. Las mismas son cambiadas de tamaño a 84 x 84 píxeles, se cambian los colores por una escala de grises y finalmente se normalizan los valores de la imagen.

Modelo. El modelo básicamente está separado en 4 grandes partes. Las primeras tres partes del modelo se construyeron con estructuras similares, donde se tienen una capa convolucional, seguida de una "batch normalization" para aumentar la velocidad de entrenamiento de la red y por último una capa elu como función de activación de siguientes neuronas.

$$\text{elu:} \quad (\mathbf{0})$$

$$e \wedge x - 1 \text{ if } x < 0$$

$$x \text{ if } x >= 0$$

Esta estructura se repite 3 veces para luego tener una capa densa “fully connected”, que finalmente estará conectada con la salida de nuestra red. En nuestro caso, solo tenemos 2 posibles acciones y por lo tanto se estima el valor de Q-óptimo para cada acción. Es decir que se obtienen solo 2 posibles salidas, saltar o no saltar y finalmente se elige hacer la acción que otorga el mayor Q o retorno.

3 Entrenamiento

Pre-entrenamiento. Antes de arrancar el entrenamiento, es necesario que la replay memory tenga al menos 64 experiencias a modo de inicialización. Las mismas son realizadas haciendo acciones de manera aleatoria y guardando los resultados. Cada experiencia está formada como la siguiente tupla: (state, action, reward, nextState, isDead), donde state es el estado inicial de la experiencia, action es la acción que se tomó, reward es la recompensa obtenida al realizar la acción, nextState representa el estado inmediatamente siguiente en el que se encuentra y isDead es un booleano que indica si murió o no el agente.

Entrenamiento. Lo utilizado para entrenar el modelo fue una “greedy strategy” donde a cada paso decrecía con un determinado epsilon la probabilidad de realizar una acción de manera aleatoria. En esta versión del modelo se utilizó un epsilon de 0,0001. De esta manera, la probabilidad de realizar una acción aleatoria decae de manera exponencial siguiendo la siguiente ecuación.

$$\text{exploreProbability} = \text{exploreStop} + (\text{exploreStart} - \text{exploreStop}) * \text{np.exp}(-\text{decayRate} * \text{self.decayStep} * \text{fastDecay}). \quad (\mathbf{1})$$

Esto se hace a modo de que el modelo al principio explore todas las posibilidades de manera aleatoria y luego con el tiempo vaya eligiendo lo que cree que es la mejor acción a seguir.

Luego de realizar la acción elegida, ya sea por el modelo o de manera aleatoria, se procede calculando el QTarget que es el valor de la función action-value, la cual determina la recompensa del siguiente estado habiendo tomado la acción en el estado determinado.

$$QTarget = r + gamma*maxQ(s', a') \quad (2)$$

Una vez calculado el QTarget, procedemos a calcular la función Loss en nuestro modelo siendo la misma:

$$Loss = (Qtarget - Q)^2 \quad (3)$$

El cálculo de la función Loss es lo que utilizamos para ajustar los valores de la red para que la misma pueda aprender. Lo que queremos es minimizar el valor de Loss, siendo idealmente 0. Es importante hacer un seguimiento de los valores de Loss para entender si la red está aprendiendo o desaprendiendo en un momento dado. El algoritmo utilizado como optimizador para minimizar el valor Loss fue el "RMSProp".

4 Descripción del algoritmo

Para comenzar el entrenamiento correctamente, como ya se dijo, se debe inicializar la replay memory con experiencias que se obtienen de ejecutar acciones aleatoriamente.

Luego, lo primero que se hace es armar el state inicial del juego, consistente de 4 frames. Dados estos frames, se genera una predicción y se ejecuta la acción predicha. Dada la acción, se obtiene la recompensa de la misma y se verifica si el agente murió o no. Luego se crea el siguiente estado agregando al stack de frames la imagen del resultado obtenida desde el juego. Ahora que se obtuvo toda la información necesaria, se arma la experiencia y se guarda en la replay memory.

Finalmente viene la parte de aprendizaje, donde se calcula el QTarget y el valor de Loss para ajustar los valores de la red.

5 Resultados

Luego de 7.410.860 de ciclos de entrenamiento, la red llegó a un máximo puntaje de 248. Esto si bien aún no es perfecto, demuestra que el modelo logró aprender a jugar eficientemente al juego.

En los primeros ciclos de entrenamiento, el agente moría por muchos movimientos random que terminaba haciendo. Luego con el tiempo fue aprendiendo que obtenía mejores recompensas al estar moviéndose por el pasillo que hay entre los tubos, por más que no se esté dentro de los mismos.

Fue de una manera muy progresiva el aprendizaje. Después de pasar el primer millón de ciclos de entrenamiento el agente logró superar los 10 puntos. Para llegar

hasta los 25 tuvo que llegar hasta los 5 millones de ciclos de entrenamiento. Y finalmente a los 7 millones de ciclos logró llegar hasta el resultado final.