Escuela de ingeniería
Ingeniería civil en modelamiento matemático de datos

Aplicación de aprendizaje reforzado en la detección de fuente de contaminación en red de agua superficial

Jaime Rodriguez
Profesor guía: David Salas

Investigación de título para optar al título de ingeniería civil en modelamiento matemático de datos

Rancagua, Chile
Diciembre de 2023

Dedicatoria

A mi familia en general, y en especial a mi madre, Sofía, cuyo apoyo constante ha sido mi motor a lo largo de este viaje académico. A ella le dedico este trabajo en agradecimiento por inculcarme la disciplina de levantarme temprano, por su incansable motivación y por enseñarme el valor del esfuerzo necesario para alcanzar mis sueños. A mi sobrino Jorge, su mera existencia ha sido mi fuente inagotable de inspiración. Este trabajo está dedicado a él, con la esperanza de verlo cumplir sus sueños y ser testigo de la realización de los míos. Gracias por ser mi constante recordatorio de que cada esfuerzo vale la pena.

Agradecimientos

Agradezco sinceramente a todos los profesores que desempeñaron un papel fundamental en mi formación académica y que contribuyeron de manera significativa a la realización de esta memoria. Quiero expresar mi gratitud de manera especial a los profesores David Salas y Anton Svensson, cuya orientación, apoyo y valiosos comentarios fueron fundamentales para el desarrollo de este trabajo. Su dedicación y conocimientos han dejado una marca indeleble en mi aprendizaje, y estoy agradecido por la inspiración que me brindaron a lo largo de este proceso. Sus contribuciones han sido invaluables y han enriquecido mi experiencia académica de manera excepcional.

Quisiera agradecer además al profesor Diego Muñoz por introducirme en el mundo de las bibliotecas Gym y Stable Baselines. Su capacidad para transmitir conocimientos complejos de una manera accesible y práctica ha sido esencial en el desarrollo de esta memoria.

Índice

Resumen

Este trabajo de titulación se enfoca en el modelamiento de una red de distribución de agua superficial, representada como un grafo con un nodo contaminante. La investigación se centra en el entrenamiento de una política cuyo objetivo es identificar el nodo contaminante tomando la menor cantidad de muestras posibles. Una muestra permite saber si el nodo contaminante está aguas arriba o no. Con esta información, una política consiste en crear un árbol de decisión que permite seleccionar el siguiente punto de muestreo usando los resultados de las muestras anteriores.

Para crear políticas con aprendizaje reforzado primero se debe probar y testear si las políticas creadas realmente aprenden y así corroborar que funciona la implementación establecida, una buena opción es entrenar políticas para juegos conocidos, donde se conoce su política óptima, por ello se aplicaron técnicas avanzadas de aprendizaje reforzado en el ámbito de juegos estratégicos de dos jugadores, utilizando el juego de tres en línea como escenario de prueba. Durante esta fase experimental, se exploraron diversos algoritmos de aprendizaje reforzado. Se destacó la viabilidad de entrenar una política efectiva para un jugador, mientras se modelaba al segundo jugador como una parte intrínseca del entorno, es decir, el segundo jugador no se entrenó con aprendizaje reforzado, sino que su política se definió previamente. Se logró crear un agente lo suficientemente competente como para ganar en el 99% de las interacciones.

Luego se procedió a entrenar políticas en diferentes tipos de entornos con diferentes reglas. Primeramente, se crearon grafos simples, y posteriormente, se avanzó hacia la generación de grafos dirigidos acíclicos aleatorios con diferentes dimensiones, donde una dimensión en un grafo dirigido acíclico viene dada por la cantidad de nodos que tiene el grafo. El entrenamiento de políticas en estos grafos proporcionó valiosas percepciones sobre la escalabilidad de las técnicas de aprendizaje reforzado en contextos más complejos. Finalmente se llevó a cabo un estudio utilizando datos reales de una distribución de agua superficial en la región del Biobío. El grafo dirigido acíclico consta de 127 nodos con grado máximo 5, y la política desarrollada logra encontrar el nodo contaminante en promedio después de 6.4 muestras siendo significativamente

mejor que algoritmos de búsquedas en arboles actuales. Este enfoque progresivo desde juegos estratégicos hasta entornos de red de agua demuestra una metodología sistemática y rigurosa.

Palabras clave: aprendizaje reforzado, búsqueda en grafos, redes neuronales artificiales, calidad de agua superficial

Introducción

El control de calidad de aguas superficiales es un aspecto crítico de la gestión del recurso hídrico. La identificación de puntos de contaminación en estas redes es un desafío fundamental para prevenir riesgos para la salud pública. Numerosos estudios previos han abordado esta problemática desde diversas perspectivas como usar sistemas de información geográfica, redes de monitoreo o hasta la utilización de macroinvertebrados acuáticos como bioindicadores de contaminación . (Torres Beristaín et al., 2013; Gutiérrez Pérez, 2021; Madera, 2016).

La teoría de grafos ha sido instrumental para representar la topología de las redes de distribución de agua, permitiendo analizar la conectividad y las interrelaciones entre los nodos. (Ashaw Muñoz, Navarro, y Gil Sánchez, 2020). En este contexto, la centralidad de un nodo se convierte en un indicador crucial, ya que nodos particularmente importantes, como puntos de conexión críticos o zonas vulnerables, pueden desempeñar un papel significativo en la propagación de contaminantes.

Una línea de investigación importante consiste en la búsqueda de nodos en un grafo, considerando alguna regla de propagación. (Goodrich y Tamassia, 2013; Cormen et al., 2009; Fayyad et al., 2002; Liu et al., 2009; Barrat et al., 2008). En particular se ha estudiado en profundidad la búsqueda de nodos en árboles, que son un tipo específico de grafos. En estos casos es posible encontrar políticas de búsqueda donde a complejidad temporal de la búsqueda en árboles con grado máximo Δ es $O(\Delta \log(n))$ (Cormen et al., 2009, p. 278). Recientemente, algoritmos de búsqueda en grafos han sido utilizados para detectar zonas de contaminación de COVID-19 muestreando aguas de cañerías. (Xu et al., 2022; Wang et al., 2022).

Los resultados destacan que la política desarrollada logra identificar el nodo contaminante con un promedio de 6.4 búsquedas en un grafo dirigido acíclico de grado máximo 5 con 127 nodos, mostrando una eficiencia notable en comparación con enfoques convencionales. Estos hallazgos refuerzan la relevancia de la aplicación de técnicas de aprendizaje reforzado en la gestión de la calidad del agua y sugieren su potencial para mejorar la resiliencia y capacidad de respuesta ante eventos de contaminación en redes de distribución de agua superficial.

En la presente investigación, se propone una política entrenada con aprendizaje reforzado para la identificación óptima de nodos contaminantes en una red de agua superficial. Los algoritmos clásicos de búsqueda en árboles no son aplicables en este contexto, pues estas redes antes mencionadas no poseen estructura de árbol, sino más bien de grafo dirigido acíclico. La elección de un grafo dirigido acíclico para modelar la red y la implementación de esta política en un escenario real, utilizando datos de la región del Biobío, aportan una perspectiva novedosa y aplicada a la investigación existente.

Hipótesis

El uso de técnicas de aprendizaje reforzado en el modelamiento de una red de distribución de agua superficial, específicamente para la identificación de nodos contaminantes, permitirá desarrollar una política eficiente que logre identificar el nodo contaminante con la menor cantidad de búsquedas posibles. Se espera que esta aproximación sea efectiva incluso en grafos dirigidos acíclicos de gran tamaño, como el caso de la red de distribución de agua en la región del Biobío, y que los resultados obtenidos demuestren mejoras significativas en comparación con enfoques tradicionales de búsqueda.

Preguntas de investigación

- ¿Cuál es la diferencia de una política entrenada con un entorno determinista a uno mixto?, donde para esta investigación se define como un entorno determinista el que cambia de estado usando una función determinista y el entorno mixto es un entorno estocástico que con cierta probabilidad cambia usando una función determinista y con otra probabilidad cambia aleatoriamente con una distribución uniforme, ¿Qué política es mejor?
- ¿Qué algoritmo de aprendizaje reforzado es el indicado para entrenar una política en un entorno de dimensionalidad escalable?, ¿Qué tipos de algoritmos no son los adecuados?
- A comparación con el número de pasos promedio de demora en búsqueda de árboles O ($\triangle$ log(n)), ¿Cuánto mejor o peor es la política entrenada con aprendizaje reforzado?
- De forma empírica, ¿Cuántos pasos de entrenamiento se necesitan para obtener una política entrenada con algún algoritmo de aprendizaje reforzado dada la dimensionalidad del entorno?

Objetivo general

Creación de una política con aprendizaje reforzado que detecte fuente contaminante dentro de un grafo dirigido acíclico que representa una red de agua superficial real de la región del Biobío con regla de propagación en la menor cantidad de muestras posibles.

Objetivos específicos

- Creación de políticas de agente que juegan tres en línea con algoritmo de aprendizaje Q-learning y A2C.
- Comparar y monitorear las políticas entrenadas para agentes que juegan tres en línea en diferentes casos:
    - Política entrenada en entorno aleatorio contra un entorno aleatorio, determinista y mixto.
    - Política entrenada en entorno determinista contra un entorno aleatorio, determinista y mixto.
    - Política entrenada en entorno mixto contra un entorno aleatorio, determinista y mixto.
- Creación de políticas entrenadas para agentes que están en entorno de búsqueda en grafo línea, DAG aleatorio y DAG de la región del Biobío.
- Ver los resultados de las políticas y compararlas con el número promedio de búsqueda de algoritmos actuales.

Marco teórico y revisión de literatura

1. Grafos:

A continuación, se definirán conceptos de la teoría de grafos:

- Grafo: Un grafo G se define formalmente como un par ordenado (V, E), donde: $V := \{v_1, v_2, \ldots, v_n\}$ conjunto de vértices , $\mathrm{E} := \{e | e = \{v_i, v_j\}, v_i, v_j \in V\}$ conjunto de aristas .
- Grafo dirigido: Un grafo dirigido G es un grafo donde cada arista dirigida $e \in E$ se representa como un par ordenado (v_i, v_j), donde $(v_i, v_j) \neq (v_j, v_i)$ con $v_i \neq v_j$.
- Camino en un grafo: Dado un grafo dirigido $G = (V, E)$, un camino desde v_1 hasta v_k está dado por una secuencia de nodos $(v_1, v_2, \ldots, v_k)$, donde para cada i desde 1 hasta $k - 1$, $(v_i, v_{i+1}) \in E$. Y $v_i \neq v_{i+1}$ para cada i desde 1 hasta $k - 1$.
- Apuntar: Para un grafo dirigido $G: (V, E)$, un nodo v apunta a un nodo u si la arista $(u, v) \in E$.
- Recibir: Para un grafo dirigido $G: (V, E)$, un nodo v recibe a un nodo u si la arista $(v, u) \in E$.
- Grafo dirigido acíclico: Un grafo dirigido acíclico G es un grafo dirigido donde no existe un camino $P_{(\mathrm{v,v})}$ para todo $\mathrm{v} \in V$.
- Predecesores: Para un grafo dirigido acíclico G, se les llama predecesores de un nodo v al conjunto: $Pred(v) := \{u \in V | \exists \text{ un camino de } u \text{ a } v\}$.
- Sucesores: Para un grafo dirigido acíclico G se les llama Sucesores de un nodo v al conjunto: $Suc(v) := \{u \in V | \exists \text{ un camino de } u \text{ a } v\}$.

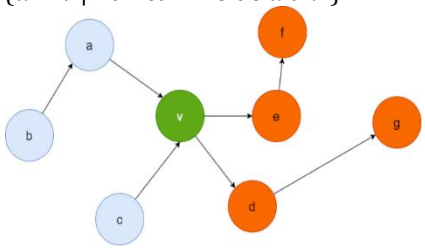


Ilustración 1 Ejemplo de grafo dirigido acíclico, donde se muestran los predecesores de v mostrados en azul y sus sucesores en naranja.

- Grado saliente y entrante de un nodo: Para un grafo dirigido acíclico G el grado saliente de un nodo v es el número de aristas que tienen a v como origen, matemáticamente se expresa como $d_{saliente}(v) = |\{(v,w) \in E : w \in V\}|$. Análogamente el grado entrante de un nodo v es el número de aristas que tienen a v como destino, matemáticamente $d_{entrante}(v) = |\{(w,v) \in E : w \in V\}|$.

- Grado total de un nodo: Para un grafo dirigido acíclico G el grado total de un nodo v es la suma del grado saliente y entrante de ese nodo, es decir, $d_{total}(v) = d_{entrante}(v) + d_{saliente}(v)$.

- Grado máximo de un grafo: Para un grafo dirigido acíclico G, el grado máximo es el valor del grado total de un nodo en el grafo que tiene el valor más alto, matemáticamente se expresa como $\max_{v \in V} d_{total}(v)$ y tiende a notarse como Δ.
- Dimensionalidad de un grafo: Viene dada por la cantidad de nodos que tiene el grafo omitiendo el número de aristas que contiene. (En el contexto de esta investigación, las aristas no influyen en la dimensionalidad del problema, ya que, las aristas son parte del entorno y no el conjunto de posible acciones como se definirá más adelante con los nodos).

2. Redes neuronales artificiales:

Un componente primordial en esta investigación de título son las redes neuronales artificiales (RNA). En palabras simples, una red neuronal artificial es un sistema de computación que imita el funcionamiento del cerebro humano para realizar tareas específicas. Está compuesta por "neuronas" artificiales que se organizan en capas, y estas neuronas trabajan juntas para procesar información. Durante el entrenamiento, la red aprende a realizar una tarea específica ajustando sus conexiones internas. La idea es que, al igual que el cerebro, la red pueda generalizar y aplicar lo que ha aprendido a nuevas situaciones. Las redes neuronales artificiales se utilizan en una variedad de aplicaciones, como reconocimiento de imágenes, procesamiento de lenguaje natural y toma de decisiones.

En esta sección se definirá matemáticamente lo que es una neurona artificial (NA) y una red neuronal artificial (RNA), además se nombrará en palabras simples como es el proceso de entrenamiento y la función objetivo que busca optimizar.

Definición 1 (NA). Una NA es una entidad matemática definida por una función $f: A \subset \mathbb{R}^n \to B \subset \mathbb{R}$, la cual se expresa de la siguiente manera: $f(x) = s(w^T x + b)$, donde $s: \mathbb{R} \to B$, $w \in \mathbb{R}^n$, y $b \in \mathbb{R}$. La función $s(\cdot)$ también es conocida como función de activación.

Algunos tipos de NA conocidos:

- Perceptrón: Funciona de manera similar a una puerta lógica, siguiendo la nomenclatura, la neurona perceptrón define los siguientes aspectos: $A \equiv \{0,1\}^n$, $B \equiv \{0,1\}$ y

$$s(z) = p(z) = \begin{cases} 0 & \text{si } z \leq 0, \\ 1 & \text{si } z > 0 \end{cases} \text{ con } z = (w^T x + b)$$

- Sigmoidea: La estructura de la función de activación es continua y diferenciable, definiendo $A \equiv [0,1]^n$, $B \equiv [0,1]$ y

$$s(z) = \sigma(z) = \frac{1}{1+e^{-z}} \text{ con } z = (w^T x + b)$$

Definición 2 (RNA). Una RNA es una colección de neuronas artificiales organizadas en m+1 capas, etiquetadas desde la capa L_0 a L_m. La capa i=0 corresponde a la capa de entrada, las capas i∈{1,…,m−1} son capas ocultas, y la capa i=m es la capa de salida. Para i∈{1,…,m}, las salidas de las neuronas de la capa L_{i-1} son ocupadas como las entradas de las neuronas en la capa L_i. Finalmente, la RNA es una función que recibe un vector $x \in \mathbb{R}^{L_0}$ (que corresponde a los valores de las salidas de la capa de entrada) y que entrega una salida $y \in \mathbb{R}^{L_m}$, (correspondiente a las salidas de la capa de salida).

Existe una formulación matemática de una RNA que sirve para facilitar cálculos y un proceso llamado propagación hacia adelante, donde los datos de entrada se pasan a través de la RNA capa por capa, y se aplican operaciones matriciales y funciones de activación para obtener la salida de la red, entender como funciona la formulación matemática ayuda a entender cómo

trabaja la RNA, pero no es primordial para este estudio, por esto, la formulación se encuentra anexada.

2.1 Entrenamiento de una RNA:

Una RNA se puede entender como un método de aprendizaje supervisado. A continuación, se definen conceptos claves para el entrenamiento de una RNA pensándola como un método de aprendizaje supervisado.

-Dominio: Un dominio X corresponde al conjunto de señales o características, que desean ser etiquetados por la RNA.

-Etiquetas: Corresponde al conjunto Y de etiquetas, que se usará para la tarea de aprendizaje.

-Conjunto de entrenamiento: corresponde a un conjunto $T \subseteq$ XxY. Cada punto $t \in T$ se conoce como ejemplo de entrenamiento.

-Pesos y sesgos de una RNA: Dado una RNA R entonces $W := \{w: w \in W^i \ \forall i \in \{0, \dots, m\}\}$ y $B := \{b: b \in B^i \ \forall i \in \{0, \dots, m\}\}$ son los conjuntos de todos los pesos y sesgos de R, donde m representa el número de capas sin contar la capa de entrada. Nos referimos al vector de todos los elementos de W como w, el vector de pesos de R, análogamente nos referimos al vector de todos los elementos de B como b, el vector de sesgos de R. Los elementos de w se ordenan jerárquicamente por capas y posición del sesgo en la RNA, es decir, $w = (w_1^1, \dots, w_{L_m L_{m-1}}^m)$, análogamente el vector $b = (b_1^1, \dots, b_{L_m}^m)$.

-Salida de la RNA $h(\cdot)$: Sean S^i, W^i y B^i la salida, los pesos y sesgos de la capa i-ésima de una RNA R y sea $F^i(\cdot)$ una función de mapeo de vectores de la capa i-ésima, donde para cualquier vector $V = (v_1, \dots, v_{L_i})^T$, $F^i(\mathrm{V}) = (f_1^i(v_1), \dots, f_{L_i}^i(v_{L_i}))$, con f_j^i la función de activación de la neurona j en la capa i. La salida de una RNA es una función $h: X \to Y$ que permite etiquetar los elementos del dominio $h(x) = S^m = F^m(W^m S^{m-1} + B^m)$ donde, $S^i = F^i(W^i S^{i-1} + B^i)$ con $S^0 = x$. Fijarse que la estructura de la salida de una RNA depende de los pesos y sesgos, por este motivo también se usará la notación $h_{w,b}(x)$ para referirse a la salida de una RNA con pesos w y sesgos b.

-Error cuadrático medio: Dado un conjunto de entrenamiento $T := \{(x_1, y_1), \ldots, (x_n, y_n)\}$ y salida de una RNA R, $h(\cdot)$, el error cuadratico medio es la función:

$$MSE(h) = \frac{1}{2n}\left(\sum_{i=1}^{n} || h(x_i) - y_i||^2\right)$$

El proceso de entrenamiento tiene el objetivo de minimizar una función de costo derivada del error cuadrático medio que fue realizada por un conjunto de entrenamiento $T := \{(x_1, y_1), \ldots, (x_n, y_n)\}$, donde los que anteriormente eran parámetros de peso y sesgo de la RNA pasarán a ser variables del problema de optimización:

$$min_{(w,b)} \;\; C_T(w, b) = \frac{1}{2n}\left(\sum_{i=1}^{n} || h_{(w,b)}(x_i) - y_i||^2\right)$$

En resumen, la minimización de la función de costo busca encontrar los pesos y sesgos de la RNA que generen como salida una etiqueta lo más parecida posible a la etiqueta del conjunto de entrenamiento.

El algoritmo usado para encontrar un mínimo ajustando los parámetros de pesos y sesgos de manera iterativa es el descenso del gradiente que se muestra en el Algoritmo 1 anexado. En palabras simples, trata de ajustar los pesos de la red de manera incremental y sistemática para reducir la pérdida durante el entrenamiento. El gradiente indica la dirección y la magnitud del cambio más rápido en la función de pérdida, y el descenso de gradiente se mueve en esa dirección para encontrar los valores de peso que minimizan la pérdida. Es esencialmente un proceso matemático iterativo que ajusta los parámetros de la red para mejorar su rendimiento en una tarea específica. El algoritmo descenso del gradiente demuestra un rendimiento eficiente cuando el tamaño total de datos de entrenamiento T, es relativamente pequeño. Sin embargo, su aplicabilidad se ve limitada cuando T es grande, ya que el cálculo completo del gradiente de la función de error puede resultar computacionalmente costoso. Dado este desafío, han surgido variantes del algoritmo del descenso del gradiente para mejorar el tiempo de aprendizaje. Entre

ellas, dos enfoques notables son el Descenso del Gradiente Estocástico y el Descenso del Gradiente por Mini Lotes.

3. Proceso de decisión de Markov.

Un proceso de Markov es una secuencia de variables aleatorias $\{X_t\}_{t\in T}$, donde $T \subset \mathbb{R}$, (tiempo puede ser continuo o discreto), que satisfacen la propiedad de Markov $P(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1}, \dots, X_0 = x_0) = P(X_{t+1} = x_{t+1}|X_t = x_t)$.

Definición 4 (Proceso de Decisión de Markov (MDP)). Un Proceso de Decisión de Markov (MDP) es una tupla $\langle S, A, T, R\rangle$, donde:

$-S$ es un conjunto finito o infinito de estados que representa las situaciones posibles en las que se encuentra un agente.

$-A$ es un conjunto finito de acciones que el agente puede tomar en cada estado.

$-T(s, a, s')$ es la función de transición que describe la probabilidad de transición desde un estado s al tomar una acción a y llegar a un estado s'. $T(s, a, s')$ representa $P(S_{t+1} = s'|S_t = s, A_t = a)$.

$-R(s, a, s')$ es la función de recompensa que asigna una recompensa numérica al agente por tomar una acción a en el estado s y llegar al estado s'. $R(s, a, s')$ representa la recompensa inmediata recibida por el agente.

El termino MDP, se utilizará más adelante para representar un concepto de aprendizaje reforzado.

4. Aprendizaje Reforzado:

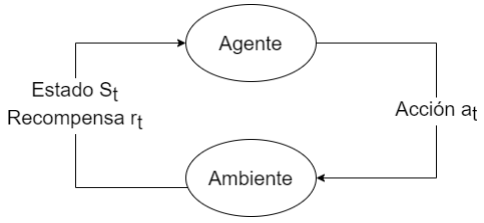


Ilustración 2. Ilustración básica aprendizaje reforzado

El aprendizaje reforzado se basa en la creación de un agente que aprende a tomar decisiones secuenciales en un entorno para maximizar una señal de recompensa acumulativa. En

este enfoque, el agente toma acciones en un entorno y recibe una retroalimentación en forma de recompensas o penalizaciones, lo que le permite aprender qué acciones son más beneficiosas en diferentes situaciones.

La "gracia" del aprendizaje reforzado radica en su capacidad para abordar problemas complejos y de toma de decisiones secuenciales. A través de la interacción continua con el entorno, el agente aprende a tomar decisiones óptimas para maximizar las recompensas a lo largo del tiempo

El objetivo es crear políticas para agentes que se desenvuelven en un entorno de tal manera que obtengan el mayor beneficio posible. A continuación, se detalla lo que es una política en el contexto de aprendizaje reforzado y sus posibles tipos:

I. Política general: Dado un MDP definido por la tupla (S, A, P, R), una política π se define como una función que mapea estados a acciones: $\pi: S \to A$.

II. Política determinista: Dado un MDP, un estado $s_t \in S$, se define como política determinista a la función: $\mu: S \to A$, que determina la acción del agente $a_t = \mu(s_t)$

III. Política estocástica: Dado un MDP, una política estocástica se define como una función $\pi: S \times A \to [0,1]$ que asigna una probabilidad $\pi(a|s)$ a cada acción a en cada estado s tal que para cada estado s, $\sum_{a \in A} \pi(a|s) = 1, \quad \forall s \in S$.

Visto lo que es una política, se dará lugar a conceptos importantes del aprendizaje reforzado. Desde ahora un entorno se refiere a un MDP, esto para facilitar la comprensión de la lectura.

4.1 Conceptos claves.

-Espacio de Observación (O): El espacio de observación es un conjunto O, donde cada elemento $o \in$O representa una observación posible que el agente puede percibir del entorno. Dependiendo del problema, O puede ser un espacio discreto o un espacio continuo.

-Espacio de Estados (S): El espacio de estados es un conjunto S, donde cada elemento $s \in$S es un estado posible del entorno. Similar al espacio de observación, puede ser discreto o continuo.

-Espacio de Acciones (A): El espacio de acciones se define como un conjunto A, donde cada elemento $a \in$A es una acción que el agente puede realizar. Este espacio por lo general es discreto.

-Episodio: Dado un entorno, se define como episodio a una secuencia de estados $s_t \in S$ y acciones $a_t \in A$ de la forma $\tau = (s_0, a_0, s_1, a_1, \dots)$ con, $s_0 \sim \rho_0(\cdot)$ donde ρ_0 es la distribución del estado inicial.

-Transición de estados determinista: Es una función que depende del estado y la acción actual, la cual usa para concretar el siguiente estado $s_{t+1} = T(s_t, a_t), d$onde $T: S \times A \to S$

-Transición de estados estocástica: Es una función que asigna un valor probabilístico al siguiente estado $P(s_{t+1}|s_t, a) = T(s_t, a, s_{t+1})$ donde $P(s_{t+1}|s_t, a_t)$ es la probabilidad del siguiente estado dado el estado y la acción actual; y $T: S \times A \times S \to [0,1]$.

-Paso de un episodio: Nos referimos a paso de un episodio a un procedimiento, método, función o algoritmo que define la recompensa, observación e información de término, (si el episodio ha terminado o no), dado el estado y la acción actual del episodio.

-Recompensa: Dado un entorno MDP, la recompensa es una función que asigna una valoración numérica a una acción $a_t \in A$ en un estado $s_t \in S$. Matemáticamente, se expresa como $R(s_t, a_t) = \mathbb{E}[r|s_t, a_t]$ donde $R: S \times A \to \mathbb{R}$ es la función de recompensa y r representa la recompensa instantánea al tomar la acción a_t en el estado s_t

-Funciones de valor:

- Función de Valor de Estado ($V^\pi(s)$): Representa el valor esperado acumulado de recompensas al comenzar en el estado s y seguir la política π a partir de ese punto. Matemáticamente, se define como $V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R\left(s_t, \pi(s_t)\right) \mid s_0 = s\right]$
- Función de Valor de Acción ($Q^\pi(s, a)$): Representa el valor esperado acumulado de recompensas al comenzar en el estado s, tomar la acción a y luego seguir la política π.

Matemáticamente, se define como $Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R\left(s_t, \pi(s_t)\right) \mid s_0 = s, a_0 = a\right]$, donde γ es el factor de descuento que modela la importancia de las recompensas futuras.

4.2 Entrenamiento de políticas con aprendizaje reforzado:

El método para entrenar políticas con aprendizaje reforzado depende de muchos factores como el entorno, la dimensionalidad, la escalabilidad que tiene el problema, entre otros. En esta parte se nombrarán algoritmos importantes de aprendizaje reforzado que se utilizan para entrenar políticas y que serán ocupados posteriormente.

4.2.2 Q-learning:

Q-learning es un algoritmo de aprendizaje por reforzamiento que se utiliza para entrenar políticas en entornos donde un agente toma decisiones secuenciales para maximizar una recompensa acumulada. Aquí hay una explicación paso a paso de cómo funciona el algoritmo Q-learning:

4.2.2.1 Definición del Problema: El problema se modela como un entorno MDP y el agente toma acciones en el entorno y recibe retroalimentación en forma de recompensas.

4.2.2.2 Inicialización de la Tabla Q: Se crea una tabla Q que asocia pares estado-acción con un valor Q, que representa la utilidad esperada de tomar una acción en un estado específico.

4.2.2.3 Elección de Acciones: El agente elige una acción (a) en el estado actual (s) utilizando una estrategia de exploración y explotación. Comúnmente, se utiliza ϵ-greedy, donde se elige la acción con mayor valor Q con probabilidad $1-\epsilon$ y una acción aleatoria con probabilidad ϵ, con $\epsilon \in [0,1]$.

4.2.2.4 Interacción con el Entorno: El agente realiza la acción elegida en el entorno y observa el nuevo estado (s') y la recompensa (r) asociada.

4.2.2.5 Actualización de la Tabla Q: Se actualiza el valor Q para el par estado-acción actual utilizando la fórmula de actualización de Q: $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [r + \gamma \cdot \max Q(s',a')]$, donde α es la tasa de aprendizaje que

controla la magnitud de la actualización y γ es el factor de descuento que pondera las recompensas futuras.

4.2.2.6 Iteración: El agente repite los pasos 3,4,5 hasta que alcanza un criterio de finalización predefinido, como un número fijo de iteraciones.

4.2.2.7 Política creada: La política creada se obtiene eligiendo, en cada estado, la acción con el mayor valor Q.

Q-learning es especialmente eficaz en problemas donde el agente toma decisiones secuenciales y la retroalimentación se proporciona de manera esparcida a lo largo del tiempo, sin embargo, La gestión de exploración y explotación puede ser difícil, y en entornos con grandes espacios de estados o acciones, la memoria y la computación pueden ser costosas. El algoritmo se encuentra en el algoritmo 3 anexado.

4.2.3 A2C:

El algoritmo A2C (Advantage Actor-Critic) es un método de aprendizaje por refuerzo que combina elementos de la política (Actor) y la función de valor (Critic) para entrenar agentes en entornos de aprendizaje reforzado. A continuación, una explicación de cómo funciona el algoritmo:

4.2.3.1 Inicialización: Se inicializan los parámetros de la RNA del actor (θ_π), la RNA del critic (θ_v) y se inicializa el entorno y se observa el estado inicial s_0.

4.2.3.2 Bucle de Episodios: Para cada episodio se inicializan los gradientes de la política ($d\theta_\pi$) y del valor ($d\theta_v$), además se observa el estado inicial s.

4.2.3.3 Bucle de Pasos del Episodio: Para cada paso del episodio se elige una acción a siguiendo la política actual $\pi(a|s, \theta_\pi)$. Se ejecuta la acción a y se observa la recompensa r y el nuevo estado s'. Luego se calcula la ventaja ($A(s,a)$) como la diferencia entre la recompensa inmediata y la diferencia entre los valores estimados del estado siguiente y el estado actual. Se actualiza el gradiente de la política ($d\theta_\pi$) sumando el gradiente del logaritmo de la probabilidad de la acción ponderado por la ventaja: $\nabla_{\theta_\pi} \log \pi(a|s, \theta_\pi) A(s,a)$. Se actualiza el gradiente del valor ($d\theta_v$) sumando el gradiente del error cuadrático entre la

recompensa y la diferencia de valores estimados: $\nabla_{\theta_\pi}(r + \gamma V(s', \theta_v) - V(s, \theta_v))^2$. Finalmente se actualiza el estado s con s'.

4.2.3.4 **Actualización de Parámetros:** Después de completar un episodio, se actualizan los parámetros de la red del actor (θ_π) y del critic (θ_v) utilizando los gradientes acumulados. Para la actualización de la red del actor: $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi d\theta_\pi$, donde α_π es la tasa de aprendizaje para el actor. Para la actualización de la red del critic: $\theta_v \leftarrow \theta_v + \alpha_v d\theta_v$, donde α_v es la tasa de aprendizaje para el critic.

Este proceso se repite a lo largo de varios episodios para mejorar tanto la política del actor como la estimación del valor del critic. La tasa de descuento γ y las tasas de aprendizaje (α_π y α_v) son hiperparámetros que afectan el rendimiento del algoritmo. En resumen, A2C busca mejorar la política del actor y la estimación del valor simultáneamente para aprender de manera más eficiente en entornos de aprendizaje por refuerzo.

5 OpenAI Gym:

OpenAI Gym es un entorno de desarrollo de código abierto diseñado para facilitar la creación y la experimentación con algoritmos de aprendizaje por refuerzo. Aquí hay una descripción de funciones clave de OpenAI Gym para la creación de entornos personalizados:

- init: Inicializa el entorno. Se definen las propiedades específicas del entorno, como el tamaño del espacio de observación, el espacio de acción, y cualquier otra configuración necesaria.
- reset: Reinicia el entorno a un estado inicial y devuelve la observación inicial. Es común realizar tareas como inicializar variables de estado en este método.
- step: Este método realiza una acción en el entorno y devuelve la observación resultante, la recompensa, un indicador de finalización del episodio y cualquier información adicional relevante.
- render: Este método se utiliza para la renderización del entorno, permitiendo visualizar la ejecución del entorno.

6 Stable Baselines:

Stable Baselines es una biblioteca de Python diseñada para facilitar la implementación y el desarrollo de algoritmos de aprendizaje por refuerzo. Esta biblioteca proporciona implementaciones eficientes y estables de varios algoritmos de RL, lo que permite a los usuarios centrarse en diseñar y experimentar con políticas de agente y entornos sin tener que preocuparse demasiado por los detalles de implementación.

7 Algoritmo minimax para juego tres en línea:

El algoritmo Minimax explora recursivamente el árbol de posibles movimientos y utiliza una función de evaluación para asignar un valor a cada estado del juego. Luego, elige el movimiento que maximiza o minimiza este valor, dependiendo del turno del jugador. El algoritmo se especifica en el Algoritmo 5 anexado.

Ejemplos: Se realizaron variados ejemplos para los diferentes temas expuestos en el marco teórico. Estos son relevantes para la comprensión del trabajo y la revisión de la literatura.

Marco metodológico

Este trabajo se centra en la implementación de métodos de aprendizaje reforzado para resolver problemas prácticos en sistemas de redes, específicamente en la gestión de redes de agua superficial. La Región del Biobío en Chile, conocida por su diversa topografía y sistemas hídricos, presenta un escenario ideal para aplicar estas técnicas avanzadas.

Como punto de partida y base experimental, se ha elegido el juego de tres en línea para entrenar una política. La simplicidad y la naturaleza bien definida de este juego lo convierten en un campo de pruebas ideal para desarrollar y ajustar algoritmos de aprendizaje reforzado. Al dominar las estrategias en este entorno controlado, el agente desarrolla habilidades fundamentales en la toma de decisiones y el reconocimiento de patrones, que son cruciales para abordar problemas más complejos.

1.Tres en línea:

1.1Definición de juego tres en línea:

Es un juego de mesa representado por un tablero de 3×3. Cada casilla del tablero puede estar en uno de los siguientes estados:

0 :Casilla vacía
1 :Ficha del Jugador 1 (O)
2 :Ficha del Jugador 2 (X)

El juego se juega entre dos jugadores, Jugador 1 (O) y Jugador 2 (X). El objetivo es completar una línea de tres fichas del mismo jugador en fila, columna o diagonal, o lograr un empate llenando todo el tablero sin que ningún jugador logre una línea ganadora.

1.2 Realización de funciones generales del juego tres en línea en Python:

Se crearon varias funciones que corresponden a la naturaleza del juego tres en línea que se ocuparan posteriormente para entrenar su política. Aquí solamente se describirá a grandes rasgos lo que hacen las funciones más comunes en la implementación, para más detalle se recomienda observar el código en Python.

- Chequeo de ganador: Recibe un tablero de juego y arroja quien ganó el juego.
- Movimientos posibles: Recibe un tablero de juego y arroja los movimientos posibles que hay en ese tablero.

1.3 Creación de política para agentes de Q-learning en juego tres en línea con Python.

A continuación, se explicará la lógica detrás de la creación de la política para el agente de Q-learning, para más detalle acerca del funcionamiento se recomienda ver el código. En particular existen tres partes importantes para la creación de la política:

1.3.1 Creación del entorno en Python:

1.3.1.1 Inicialización: Este entorno, representado por la clase 'Tictactoe', inicia con un tablero 3x3 y variables para el jugador actual y el posible ganador.

1.3.1.2 Paso de entrenamiento: La función 'step' avanza el juego al realizar una acción, devolviendo el estado actual, el siguiente estado, la recompensa y si el juego ha terminado.

1.3.1.3 Reinicio del entorno: 'Reset' reinicia el juego, incluyendo el tablero y el jugador actual.

1.3.2 Creación del agente de Q-learning: Se define un agente de aprendizaje por refuerzo que implementa el algoritmo Q-learning. Al iniciar, se configuran parámetros como la tasa de exploración, la tasa de aprendizaje, y el factor de descuento. Se inicializa las recompensas de las estimaciones de valor Q para cada estado-acción. La implementación actualiza los valores Q según la ecuación de Q-learning. El agente aprende interactuando con el entorno y ajusta sus estimaciones de valor Q en función de las recompensas recibidas. La estrategia permite al agente descubrir nuevas acciones mientras se enfoca en las acciones más prometedoras.

1.3.3 Creación de todos los estados posibles de juego: Se crea un archivo que para cada estado s existe una lista $Q(s,\cdot)$ de largo 9 que representa el valor Q de cada acción para el estado s, este archivo representa la tabla Q. Al inicio todos los valores Q son 0.5, es decir, en un inicio $\forall s \in S, \forall a \in A, Q(s,\mathrm{a}) = 0.5$

1.3.4 Proceso de entrenamiento: Se inicia un proceso de entrenamiento para dos jugadores en el juego del tres en línea. Durante cada episodio de entrenamiento, los jugadores interactúan en un bucle, eligiendo acciones y actualizando sus estrategias según las recompensas obtenidas.

En cada turno, un jugador elige una acción en el tablero utilizando una estrategia llamada epsilon-greedy, está elige con cierta probabilidad una acción aleatoriamente o con otra probabilidad elige la acción mediante la tabla Q de ese momento. Luego, aprende de las

recompensas ajustando sus expectativas sobre la calidad de sus acciones. La actualización de las expectativas se realiza mediante la fórmula del algoritmo Q-learning, donde se considera la recompensa obtenida y el valor máximo esperado para las acciones futuras.

El juego continúa alternando entre los jugadores hasta que se complete el episodio. La recompensa depende del resultado de la jugada: 1 si el jugador gana, -1 si pierde, y 0 si empata o el juego no ha terminado. Este proceso se repite para varios episodios de entrenamiento, determinados previamente. El objetivo es que ambos jugadores mejoren sus estrategias de juego a lo largo del entrenamiento.

1.4 Creación de políticas para agente entrenado con algoritmo A2C en juego tres en línea con Python junto con las librerías Gym y Stable Baselines.

Se entrenará una para política para el jugador 1, donde el jugador 2 es parte intrínseca del entorno, es decir, jugador 2 no se entrenará.

1.4.1 Política 'optima' del entorno para juego tres en línea:

Definimos un agente con política óptima para el juego tres en línea como un agente que siempre gana o empata. En la literatura, un agente con política 'óptima' del juego tres en línea se puede obtener gracias al algoritmo minimax. Una desventaja importante del algoritmo radica en su consumo elevado de memoria al intentar determinar la acción óptima en un determinado estado del juego. Para reducir este tiempo en el proceso de entrenamiento, se creó una matriz $Q(s,a)$ con ayuda del algoritmo minimax. Primeramente, se obtienen todos los estados posibles, luego para cada estado se usa minimax para determinar la mejor acción y finalmente se guarda en la matriz Q, con $\mathrm{Q}(\mathrm{s},\cdot) = \{\mathrm{q}_1^{\mathrm{s}},\ldots,\mathrm{q}_9^{\mathrm{s}}\}$, donde $\mathrm{q}_{\mathrm{i}}^{\mathrm{s}}$, es el valor Q de la acción i en el estado s y $\mathrm{q}_{\mathrm{i}}^{\mathrm{s}} = 1$ si i es la acción óptima y 0 si no. Como minimax entrega siempre una acción para un posible estado entonces $\sum_{\mathrm{i}=1}^{9} \mathrm{q}_{\mathrm{i}} = 1$.

La matriz Q definida previamente, da lugar a la política $\pi^{det}(s)$ que se usará posteriormente para entrenar al jugador 1. La política $\pi^{det}(s)$ escoge una acción mediante la matriz Q, es decir,

$\pi^{det}: S \rightarrow A$, donde dado un estado s, escoge la acción tomando el índice del elemento de Q (s,) que tiene un uno.

1.4.2 Definición de entorno y agente en Python usando librería Gym para juego tres en línea:

- 1.4.2.1 Inicialización: Se crea una instancia de clase que es la que representa el entorno en el problema. Se inicializa el espacio de acciones como $A := \{a_1, \ldots, a_9\}$, representando las 9 posiciones del tablero y el espacio de observación como $O := \{o | \ o = (o_1, \ldots, o_9) \ \forall o_i \in \{0,1,2\}\}$, representando todos los posibles tableros que puedan existir calculando las combinaciones posibles de una tupla de nueve elementos, donde cada elemento puede ser 0,1 o 2.

- 1.4.2.2 Paso de entrenamiento: La función que se encarga de arrojar la observación actual, recompensa y estado de juego es la llamada 'step', esta función nos entrega esa información en cada paso de un episodio. El agente obtiene la recompensa de 1 si gana, 0 si empata o si el juego aún no termina y −1 si el agente pierde o si realiza una acción inválida.

2. Búsqueda en grafos:

En esta sección se abordará el problema de búsqueda de contaminantes en un caudal como si fuese un juego, la definición es la siguiente:

2.1 Definición del juego:

- El juego en el cual se desenvuelve el jugador es un grafo dirigido acíclico (DAG).

- Cada nodo i tiene un valor $f_i \in$ {Limpio,Contaminado,Fuente} asociado. El nodo fuente v_f se etiqueta aleatoriamente por el entorno mediante una distribución D; los demás nodos se etiquetan como sigue:

  - Los nodos sucesores de v_f son etiquetados como Contaminado

  - Los nodos que no están etiquetados se etiquetan como Limpio

El grafo representa una red de agua superficial, los nodos del grafo representan puntos de muestreo de agua, es decir, son puntos factibles para sacar una muestra y medir la calidad de agua. En el grafo solo existe un nodo contaminante, y el objetivo del juego es encontrar ese nodo contaminante en la menor cantidad de muestreos posibles.

Ilustración 3 La vista aérea de cerca del paisaje fluvial desembocando en el mar, generada por DALL·E el 18-11-2023. La imagen representa un rio con un caudal de una forma peculiar con el objetivo de dar mayor entendimiento al juego de búsqueda en grafos

En la ilustración 3 se muestra un posible caudal de río en el cual se puede ilustrar el juego. Si al caudal de rio de la ilustración anterior se le agregan puntos de muestreo de calidad de agua, podríamos obtener un grafo que represente el caudal tal como se observa en la ilustración 4.

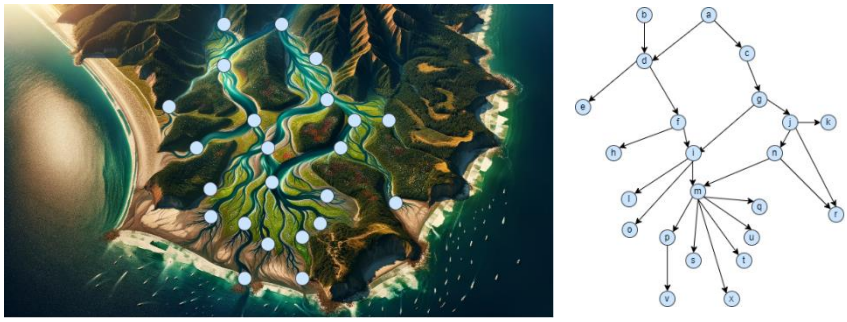


Ilustración 4 Caudal de río con puntos de muestreo a la izquierda y el grafo dirigido acíclico asociado a su derecha.

En un juego, el jugador solo puede observar el grafo, pero no puede observar las etiquetas de los nodos. En la ilustración 5, se muestran las etiquetas de cada nodo, suponiendo que el nodo fuente es el 'f'.

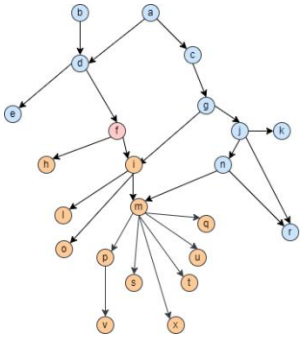


Ilustración 5 Grafo de un caudal de río con etiquetas, donde los nodos limpios son representados en azul, los contaminados son los representados en naranja y el nodo fuente en rojo.

Como se puede ver en la ilustración 5, los nodos sucesores al nodo fuente están contaminados, y el resto están limpios. Algo no tan intuitivo es entender por qué si el jugador, realiza una muestra en el nodo ‘g’ del grafo, entonces por lógica no debería realizar muestras en los nodos ‘a’, ‘c’ y ‘g’ en jugadas posteriores. La lógica detrás del descarte de nodos en cada paso de juego viene dado por el hecho de que, si reviso un nodo limpio, esto quiere decir que es imposible que los nodos predecesores al nodo limpio estén contaminados, de forma análoga si reviso un nodo contaminado quiere decir que su fuente está en los predecesores de ese nodo. En la ilustración 6 se muestra como un jugador encuentra el nodo fuente realizando una muestra en el nodo ‘g’, (limpio), luego al nodo ‘m’, (contaminado), después al nodo ‘i’, (contaminado), para finalmente dar con el nodo fuente ‘f’, mostrando el descarte lógico de nodos para cada jugada.

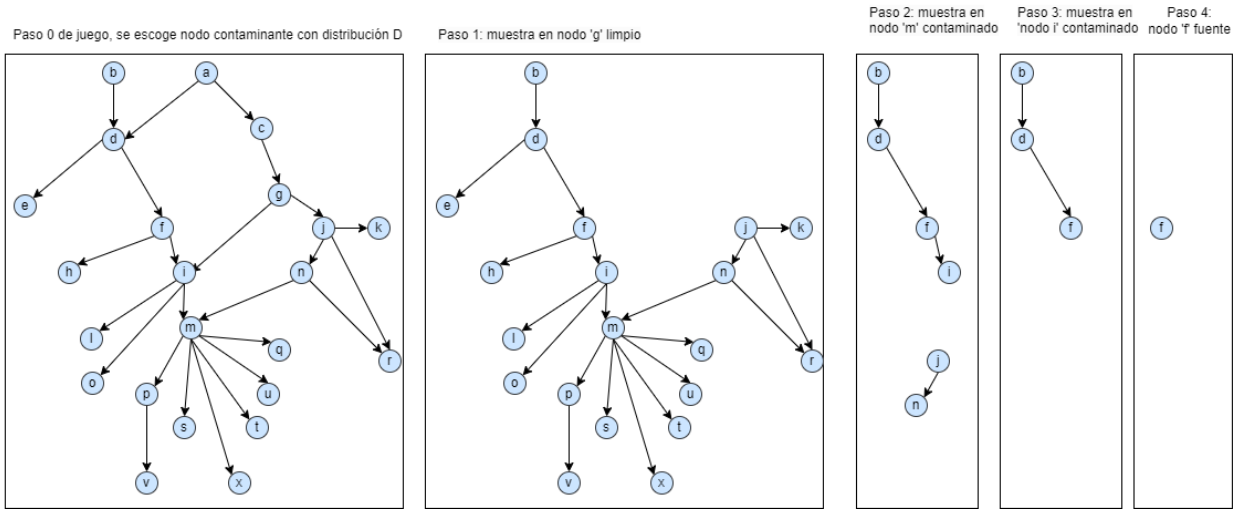


Ilustración 6 Ejemplo de juego, donde el jugador realiza muestreo en los nodos 'g', 'm', 'i' y ‘f', realizando el descarte lógico dado por las etiquetas.

2.2 Definición de entorno de juego búsqueda en grafos en Python con librería Gym.

En esta sección se abordará y detallará como se realizó el entorno de juego en Python, para ello primeramente debemos dar lugar a las funciones claves que se utilizarán para definir el entorno y las decisiones del agente en este.

2.2.1 Funciones de entorno generales:

2.2.1.1 Predecesores y sucesores: Se creó una función que obtiene los predecesores o sucesores de un nodo en un grafo, la función se basa en la búsqueda en profundidad. Recibe un conjunto de aristas, un vector, una lista vacía y un valor de verdad, si es falso se obtienen los predecesores, de lo contrario se obtienen los sucesores. El detalle se puede ver en el Algoritmo 6 anexado.

2.2.1.2 Eliminador de nodos: La función recibe un grafo, un nodo y un booleano, (verdadero si el nodo ingresado es contaminado o falso si el nodo es limpio, no puede recibir un nodo v fuente); y arroja un conjunto de nodos factibles para el siguiente paso en el episodio. El detalle se encuentra en el Algoritmo 7 anexado.

2.2.1.3 Etiquetas f_i: La función que recibe un grafo y un nodo que es el nodo fuente escogido previamente por una distribución D, y arroja una lista que contiene los valores f_i para cada nodo i. El detalle se encuentra en el Algoritmo 8 anexado.

2.2.2 Definición de entorno para juego de búsqueda en grafo en Python con librería Gym:

Definimos el espacio de observación del agente como el espacio de estados del entorno, donde una observación es un vector que representa las acciones válidas en un paso de entrenamiento. Y se define el espacio de acciones como el conjunto de vectores del grafo. Notar que un agente puede cometer acciones inválidas, ya que, para todo estado $s \in S$ el agente puede escoger un $v \in V$. El agente a priori no sabe realizar acciones válidas. Matemáticamente:

$$O := S := \{o \in \{0,1\}^{|V|} \mid o_i = 1 \text{ si el nodo i es una acción válida y 0 si no } \forall\ i \in V\} \text{ y } A := \{v \in V\}$$

A continuación, se detallará el funcionamiento del entorno en Python, para más detalle se recomienda ver el código subido en el repositorio de GitHub.

2.2.2.1 Inicialización: Se definen las aristas y los vértices del grafo. Luego se inicializa las observaciones y el nodo contaminante del entorno. Finalmente se inicializan el espacio de acción y el espacio de observación.

2.2.2.2 Reinicio: se escoge un nuevo nodo fuente y se reinicia el vector de observación.

2.2.2.3 Paso del episodio: Arroja la observación del siguiente paso, la recompensa para el paso actual e información del estado de termino de juego. Para mayor entendimiento del proceso se recomienda ver la ilustración 7.

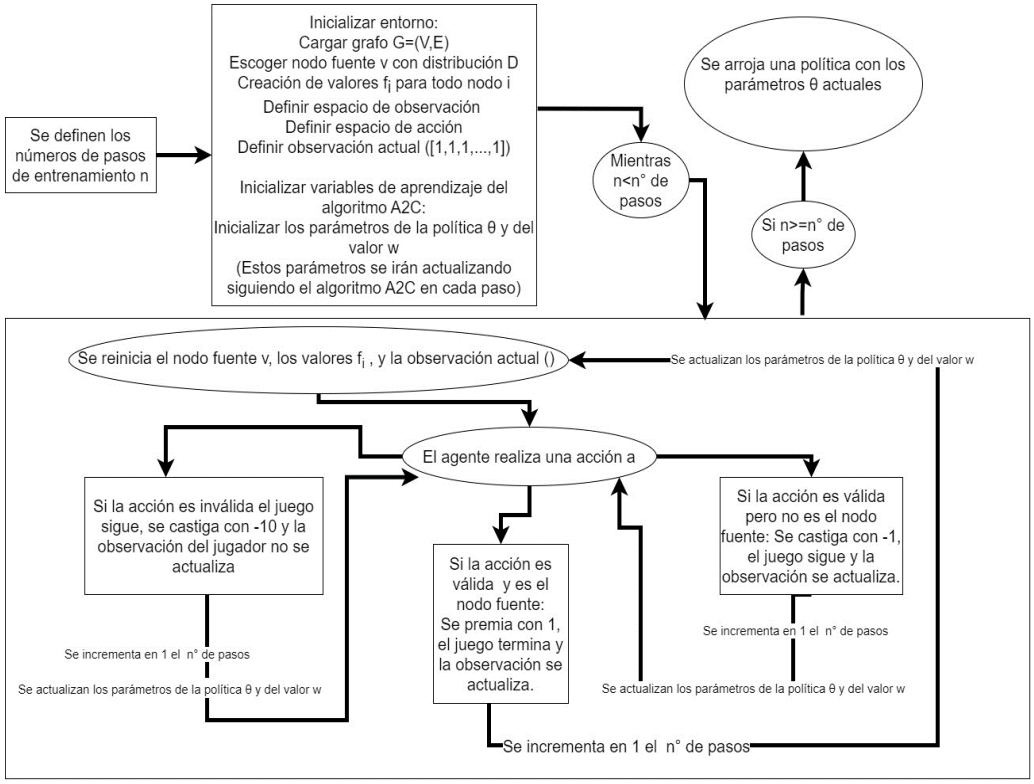


Ilustración 7. Diagrama simple del funcionamiento de la creación de una política para el juego de búsqueda en grafo con algoritmo A2C

2.3 Definiciones y suposiciones:

A continuación, se nombrarán definiciones y suposiciones que se realizaron en el estudio.

2.3.1 Objetivo de política del agente: El objetivo que busca el entrenamiento de la política se expresa como $\min_{\pi} E_D[Pasos_\pi]$, es decir, se busca que, en promedio, la política se demore la cantidad mínima de pasos en el episodio.

2.3.2 Definición grafo línea: Se definió un tipo de grafo con el propósito de probar a nuestro agente en un entorno simple y así monitorear su aprendizaje. El grafo línea es un DAG, donde dado su vector de nodos $V = \{v_1, v_2, v_3, \dots, v_n\}$, entonces sus aristas son de la forma $E = \{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)\}$.

2.3.3 Definición de supuesta política óptima para un tipo de grafo línea: Se supone que una política óptima para un grafo línea con n$= 2^k - 1, \forall k \in N$ nodos y para un entorno en donde el nodo contaminante se esconde con una política de distribución uniforme, es la que dado un conjunto de vértices $V = \{v_1, v_2, v_3, \dots, v_{2^k-1}\}$, siempre escoge el vértice que está a la mitad $v_{\lfloor (2^k-1)/2 \rfloor}$. La política óptima se deriva de la búsqueda binaria. Para más detalle ver el ejemplo 5 anexado.

2.3.4 Representación de políticas determinista: Una política determinista tiene la ventaja de que para cualquier sitio en el que se esconda el nodo contaminante, entonces solo existe un único conjunto de vértices de muestreo que te llevan a él. La política será representada con llaves y tendrá de largo la cantidad de nodos del grafo principal, donde cada elemento de la política tendrá un valor que representa el número de muestreos que se necesitaron para llegar al índice que representa el nodo. Por ejemplo, si tenemos un grafo línea con 7 nodos, entonces la supuesta política óptima es representada como {3,2,3,1,3,2,3}.

2.3.5 Demora promedio de una política: Para obtener el número promedio de pasos de demora que tiene la supuesta política óptima determinista en un grafo línea de largo $n = 2^k - 1$ con $k \in N$, se puede realizar un cálculo donde su resultado es $\frac{2^k k - 2^k + 1}{n}$. El Desarrollo se puede ver en el Cálculo 1 anexado.

2.3.6 Definición de grafos dirigidos acíclicos aleatorios: Para probar la eficacia del entrenamiento de las políticas en búsqueda en grafo, se realizaron DAG aleatorios y se entrenaron políticas en esos ambientes. Para realizar los DAG aleatorios, se creó una función que recibe un número que representa la cantidad de nodos que queremos que tenga el grafo y arroja un DAG, esto lo hace conectando cada nodo con un número aleatorio de nodos anteriores, (entre 1 y 3), asegurando

que el grafo sea acíclico. Esto se logra generando conexiones de manera aleatoria entre los nodos del grafo.

2.3.7 Datos de red superficial de agua de la región del Biobío: La base de datos fue entregada por el profesor de la universidad de O'Higgins Anton Svensson, la base otorgaba información variada de la red superficial de agua, sin embargo, estos datos se limpiaron y solo se obtuvieron de ellos los puntos de muestreo y sus interconexiones para formar el grafo que representa la red. El grafo consta con 125 nodos y 131 aristas, donde su grado máximo es de 5. Además, es un grafo no conexo que tiene dos componentes conexas. El gráfico del grafo se ve en la Ilustración 8.

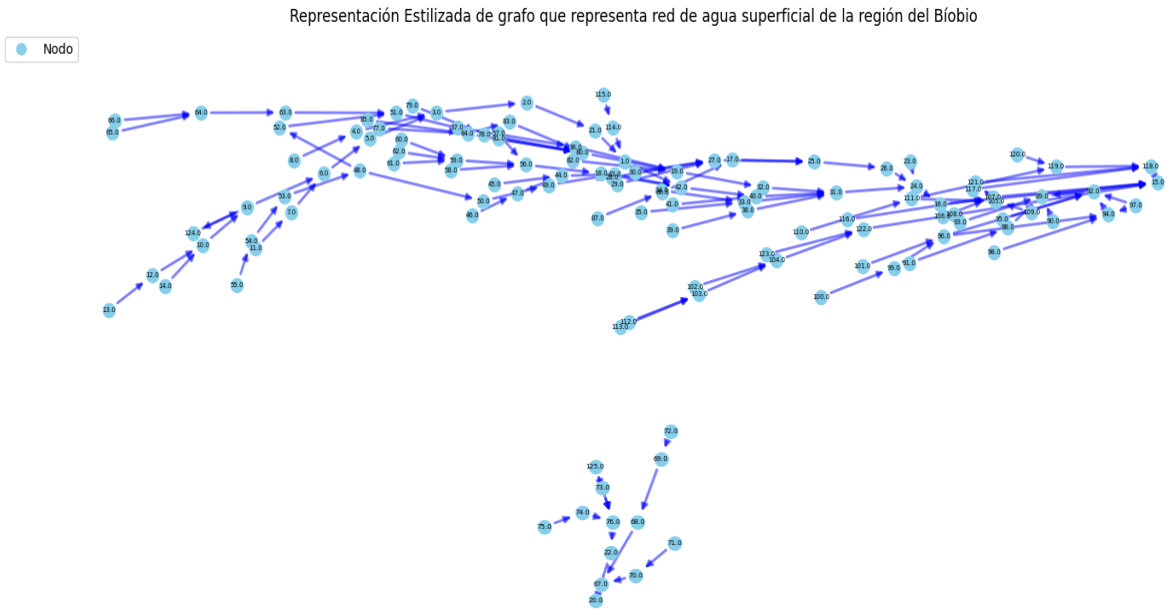


Ilustración 8. Gráfico de grafo que representa la red de agua superficial de la región del Biobío hecho en Python.

Los códigos de esta sección se pueden ver en el siguiente enlace: https://github.com/JaimeRodriguezCh/Reinforcement-Learning.

Resultados y discusión

1. Resultados de políticas entrenadas con algoritmo de Q-learning para juego tres en línea:

Dos políticas de agentes de Q-learning fueron entrenadas a lo largo de 100,000 episodios con resultados mixtos. Entre las ventajas, se destaca que los agentes aprendieron a realizar el último

movimiento ganador cuando era posible y que el algoritmo generó a los agentes de manera eficiente. Sin embargo, se identificaron desventajas, incluyendo la necesidad de un alto número de episodios para que los agentes aprendan y el hecho de que su aprendizaje se limita principalmente a realizar el movimiento ganador, sin alcanzar una comprensión más profunda del juego ni desarrollar una política óptima. Se señala también que la formación requiere de suficiente aleatoriedad para cubrir diversos escenarios y formar así un agente competente.
La explicación a que el agente no aprendiera a jugar se debe a la del algoritmo de Q-learning.
Para ejemplificar, definamos $\alpha = 0.1$, $\gamma = 0.9$, e inicialicemos $Q(a,s) = 0.5\ \forall a \in A, s \in S$.
Fijémonos en el siguiente extracto de un episodio que pasa del estado s al estado s' con la acción a_4 donde el agente protagonista es 1:

$$\begin{bmatrix} 1 & 2 & 0 \\ 2 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix} \overset{a_4=(3,2)}{\rightarrow} \begin{bmatrix} 1 & 2 & 0 \\ 2 & 2 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Su recompensa es $R = 0$, por que aún no termina el juego, esto quiere decir que su matriz $Q(s,\cdot)$ hizo la siguiente transición:

$$\begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \overset{R=0}{\rightarrow} \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & 0.495 & 0.5 \end{bmatrix}$$

Esto viene dado por:
$Q(s,a_4) = (1-\alpha)\cdot Q(s,a) + \alpha \cdot \left(R + \gamma \cdot \max Q(s',a')\right) = (1-0.1)\cdot 0.5 + 0.1 \cdot (0 + 0.9 \cdot 0.5) = 4.95.$
Esto es perjudicial si en el episodio el jugador 1 hubiese ganado, ya que, aprendió a no realizar una acción que lo llevo a la victoria.

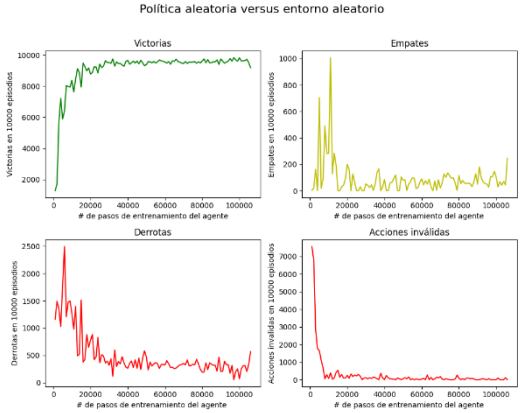
Resultados de políticas de agente entrenado con algoritmo A2C para juego tres en línea en Python con las librerías Gym y Stable Baselines:

Hay tres políticas a tratar, la política $\pi^{al}: S \to A$, que dado un tablero s, escoge aleatoriamente y de manera uniforme una acción válida dentro del tablero, $\pi^{det}: S \to A$ definida en el marco teórico y la política $\pi^{pse}: S \to A$, que con un 10% escoge una acción con la política π^{al} y con un 90% escoge una acción con la política $\pi^{det}: S \to A$. La política π^{pse} es una política estocástica referida como política mixta. Se entrenaron 300 políticas, estas se agrupan en grupos de 100, 100 de ellas se entrenaron en un entorno que realiza las acciones con la política π^{al}, otras 100 políticas se entrenaron con π^{det} y las últimas 100 se entrenaron con π^{pse}, lo único que difiere cada política es la cantidad de pasos de entrenamiento con las cuales se entrenaron, por ejemplo la primera política que se entrenó con π^{al} se entrenó con 1000 pasos y la última con 100000 pasos, donde todos son procesos diferentes. Para discutir y visualizar el rendimiento se realizaron gráficos que visualizan los resultados que obtuvieron las políticas entrenadas, estos gráficos

constan del número de empates, victorias, derrotas y acciones invalidas que obtuvieron las políticas por cada 10000 episodios.

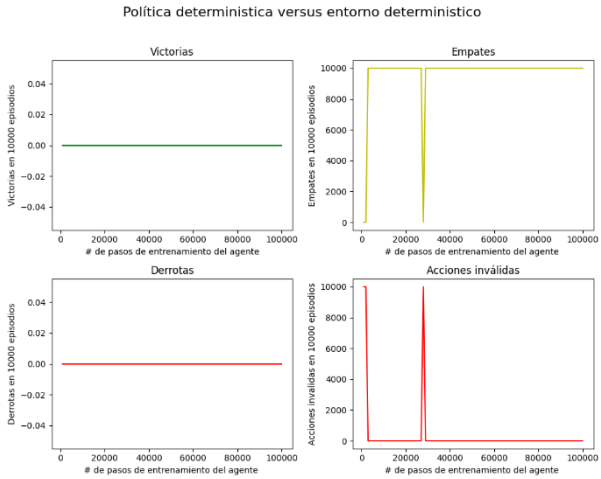
Para ordenar esta sección se decidieron agrupar los gráficos para cada grupo de 100 políticas entrenadas de la siguiente manera:

- Grupo de 100 políticas entrenadas con π^{al} testeadas en un entorno con política π^{al}.



Comentarios: Se logra ver como las políticas a medida que aumentan sus pasos de entrenamiento obtienen mayor cantidad de victorias y menor cantidad de derrotas y empates. Además, pareciera que se requiere de pocos pasos de entrenamiento para que la política aprenda a no realizar acciones inválidas.

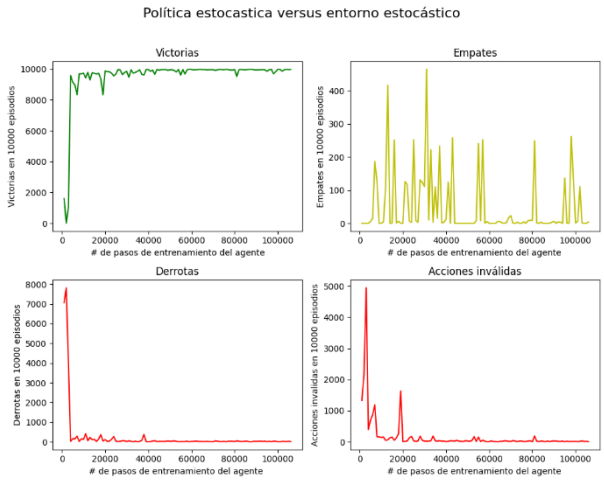
- Grupo de 100 políticas entrenadas con π^{det} testeadas en un entorno con política π^{det}.



Comentarios: Se logra apreciar como para un entorno en el cual es imposible ganar, las políticas entrenadas con pocos pasos de entrenamiento, (como por ejemplo 1000), logran aprender rápido

a empatar. Esto se puede deber a que, al ser un entorno determinista, el agente tiene un menor conjunto de estados que aprender. En la política que se entrenó con 27000 pasos de entrenamiento, no aprendió a realizar acciones válidas cosa que políticas entrenadas con pasos menores si aprendieron.

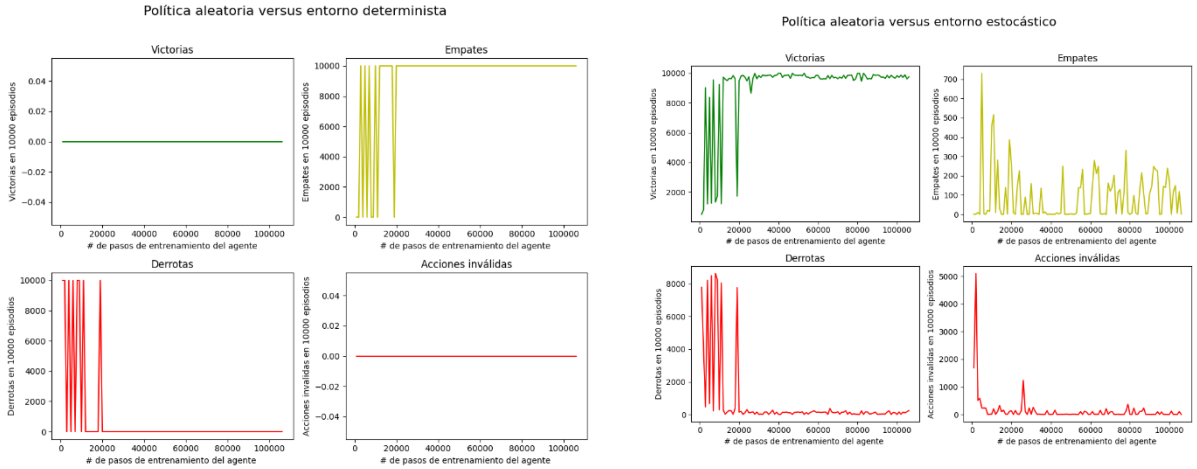
- Grupo de 100 políticas entrenadas con π^{pse} testeadas en un entorno con política π^{pse}.



Comentarios: Se logra apreciar que en este caso se puede concluir algo similar al primer experimento, sin embargo, aquí la cantidad de empates versus la cantidad de pasos pareciera ser más versátil. Esto último, solo es un detalle de perspectiva, ya que, la cantidad de empates no sobrepasa en ningún momento la cantidad de 500 en 10000 episodios. Se puede concluir que aprende a no realizar acciones inválidas, a realizar victorias y a no obtener derrotas, en políticas entrenadas con pocos pasos de entrenamiento. Este entorno es el más complejo y el más parecido al comportamiento humano, por lo que los resultados son prometedores. La mejor política en este experimento obtuvo un 99.6% de victorias.

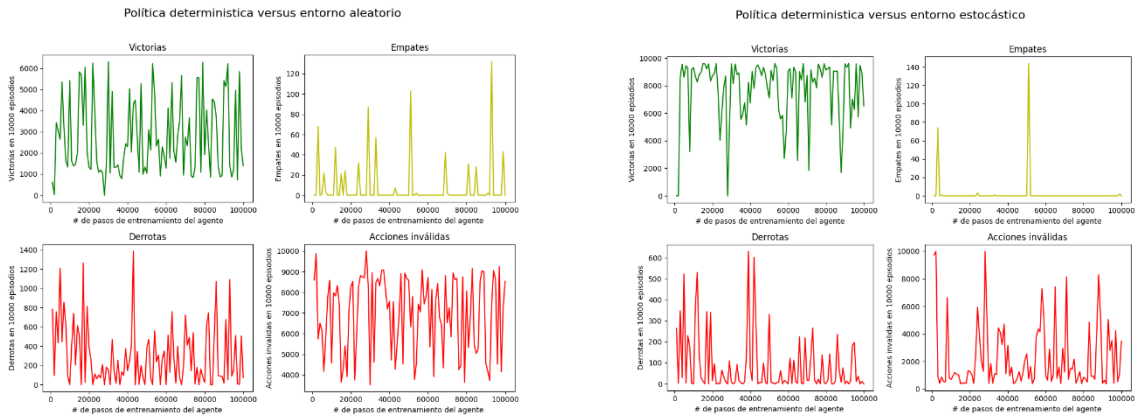
Ahora bien, puede resultar interesante observar lo que ocurre cuando las políticas entrenadas en un entorno se desenvuelven a un entorno diferente. A continuación, se presentarán los resultados de estos experimentos:

- Grupo de 100 políticas entrenadas con π^{al} testeadas en un entorno con política π^{det} a la izquierda y testeadas en un entorno con política π^{pse} a la derecha.



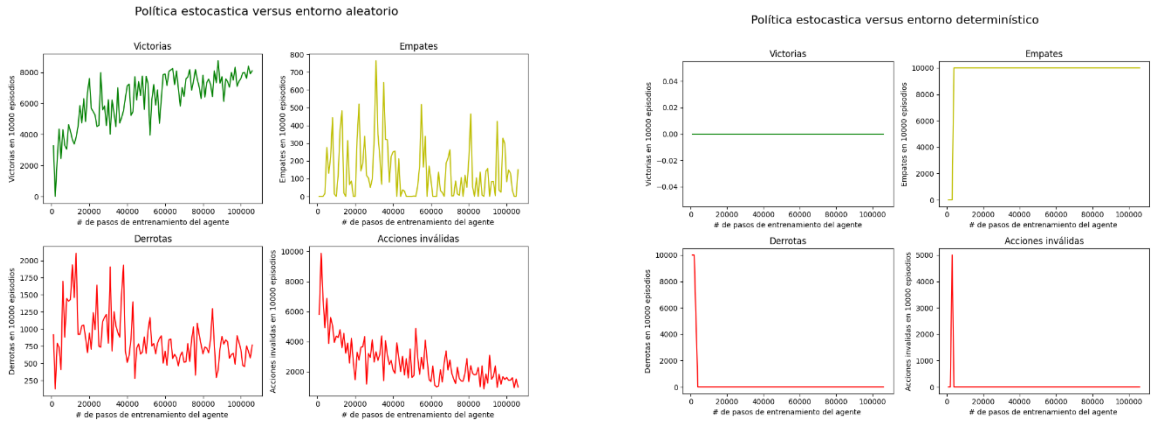
Comentarios: En ambos casos las políticas se desenvuelven bien en un entorno en el cual no han sido entrenados. Es claro ver que en ambos casos se necesita un número mayor de pasos de entrenamiento para obtener buenos resultados, sin embargo, se obtuvieron políticas similares a las obtenidas por las políticas que se entrenaron en su respectivo ambiente. La política entrenada con π^{al} que obtuvo mejor resultado en un entorno π^{det} es exactamente la misma que se obtuvo con la política entrenada en un entorno π^{det}, y la política entrenada que obtuvo mejor resultado en un entorno π^{pse} obtuvo un 99.4% de victorias, muy similar a los 99.6% de victorias que obtuvo la política entrenada en el entorno π^{pse}. Esto entra en discusión si es realmente necesario entrenar al jugador con una política predefinida y compleja como lo es π^{pse}, quizá solo basta con la estructura de la aleatoriedad para que el agente aprenda, aunque talvez esto se deba a la estructura estocástica del proceso de entrenamiento y lo simple del juego tres en línea.

- Grupo de 100 políticas entrenadas con π^{det} testeadas en un entorno con política π^{al} a la izquierda y testeadas en un entorno con política π^{pse} a la derecha.



Comentarios: Las políticas entrenadas en un entorno determinista ,es decir, en un entorno con política π^{det}, pareciera no desenvolverse bien en otros entornos no deterministas por lo versátil de los gráficos. Una explicación razonable es que las políticas entrenadas en un entorno determinista solamente aprenden una serie de acciones que lo llevan a empatar y hacen siempre esas mismas acciones, esto porque, el entorno determinista siempre realizará una única acción para un tablero de juego. Por ello cuando se desenvuelve en un entorno con un grado de aleatoriedad como π^{al} o π^{pse} este no conoce otros movimientos clave para poder realizar una victoria.

- Grupo de 100 políticas entrenadas con π^{pse} testeadas en un entorno con política π^{al} a la izquierda y testeadas en un entorno con política π^{det} a la derecha.



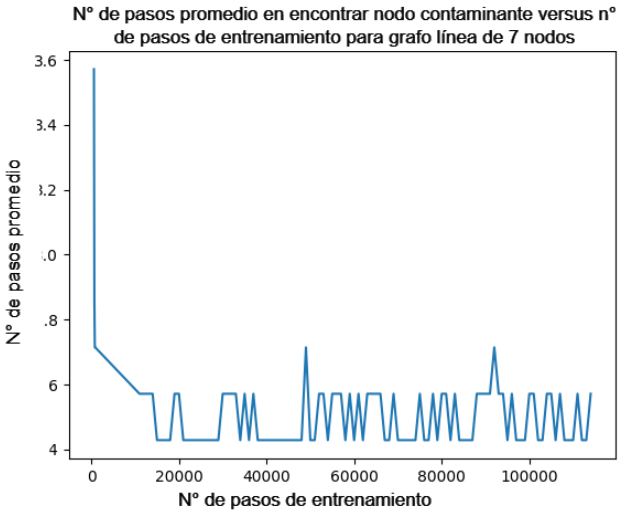
Comentarios: Al igual que para las políticas entrenadas en un entorno aleatorio, (entorno con política π^{al}), parece desenvolverse bien en ambos entornos, sin embargo, estos resultados son más versátiles y pareciera que se requiere de políticas entrenadas con gran cantidad de pasos para obtener buenos resultados en entornos diferentes al entorno de entrenamiento.

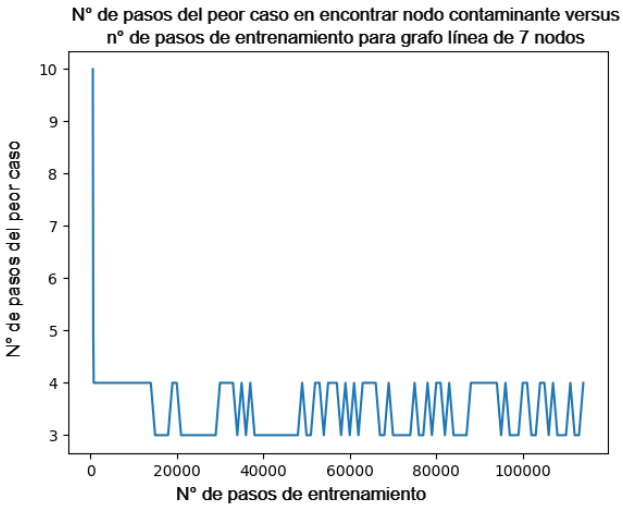
Discusión: Aunque las políticas entrenadas con la política mixta π^{pse} obtuvieron la mejor política, tiene una desventaja y es que, para poder entrenar una buena política, depende de la creación de una política similar, (la política del entorno π^{pse}), en cambio, las políticas entrenadas en un entorno completamente aleatorio lograron desenvolverse bien en un entorno diferente y más complejo como lo es el entorno con la política π^{pse}, obteniendo un resultado muy similar a las políticas que se entrenaron justamente en ese entorno y teniendo la ventaja de que no es

necesario crear previamente una política compleja. Las políticas entrenadas en un entorno determinístico, no se logran desenvolver bien en entornos diferentes. Todos los agentes se desenvolvieron bien en su propio entorno.

Resultados de políticas de agente entrenado con algoritmo A2C en entorno de búsqueda en grafos programado en Python con las librerías Gym y Stable Baselines:

Grafo línea de 7 nodos: Se entrenaron 108 políticas en el mismo entorno y en lo único que difieren es en los pasos de entrenamiento, indicados en el eje horizontal, esto con el objetivo de ver cómo va evolucionando el aprendizaje del agente a medida que los pasos de entrenamiento aumentan.



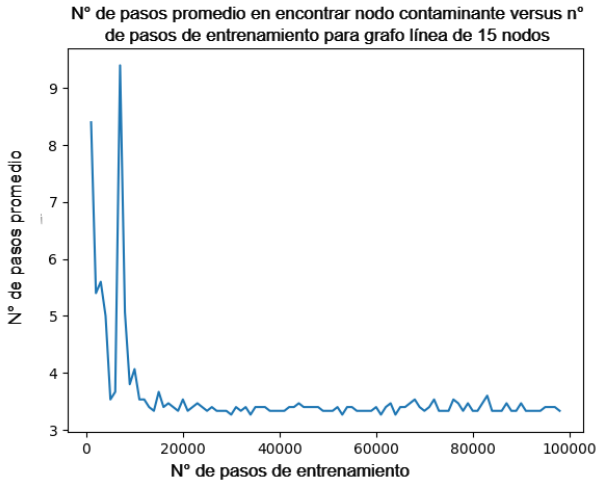


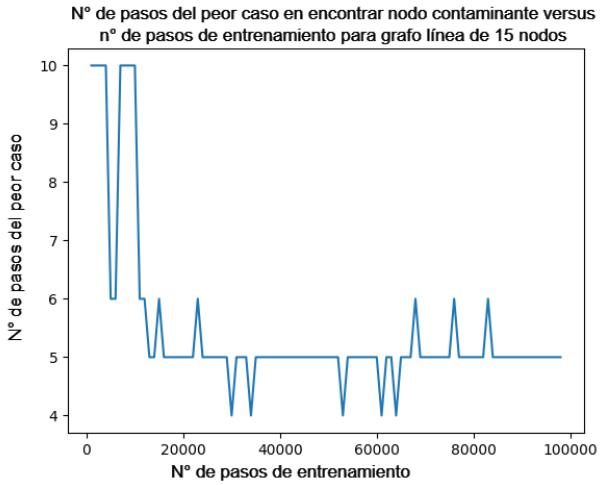
Comentarios: El proceso de entrenamiento logró aprender la política óptima representada como {3,2,3,1,3,2,3}, sin embargo, no se puede asegurar que al incrementar los pasos de entrenamiento la política converja, inclusive se puede apreciar como las políticas iteran en aprender dos políticas, la primera antes mencionada y la segunda representada como {3,2,3,4,1,3,2}. Aunque la diferencia entre el número promedio de búsqueda de cada política es un valor pequeño, (0.142), este valor podría ser mayor en un problema con otra dimensionalidad, por lo que no es recomendable realizar solo un modelo con un número grande, como por ejemplo 1.000.000 de pasos de entrenamiento para todos los casos, ya que, esto varía dependiendo de la dimensionalidad del grafo. Las políticas con un número de búsquedas 10 en el peor caso,

significan que no aprendieron a buscar en algún nodo, donde 'no aprender a buscar ciertos nodos', se refiere a que existen nodos en donde el nodo contaminante se esconde y la política resultante nunca busca, por ejemplo, la primera política se representa como $\{3,2,1,2,3,4,\infty\}$, es decir, si en esta política el nodo contaminante se esconde en el nodo 7, el número de pasos promedio en encontrar el nodo contaminante sería ∞, ya que, la política nunca busca en ese nodo y el peor caso también, pero esto es un problema a la hora de graficar, por ello se definió que si se demora más de 10 o 10, en encontrar algún nodo, en este caso, significa que la política no aprendió a buscar en ese nodo. Por ello la primera política, en el gráfico, tiene un número de búsqueda promedio de 3.57, el resultado del promedio $\frac{3+2+1+2+3+4+10}{7}$.

De este experimento nos quedaremos con la mejor política que en este caso calza con la supuesta política óptima {3,2,3,1,3,2,3}, y con el número de pasos mínimo que se usaron para obtener esta mejor política, que en este caso fueron 1000.

Grafo línea de 15 nodos : Se entrenaron 99 políticas realizando el mismo experimento anterior.

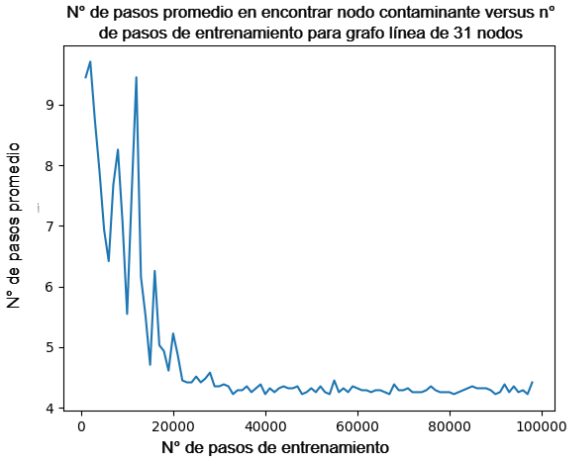


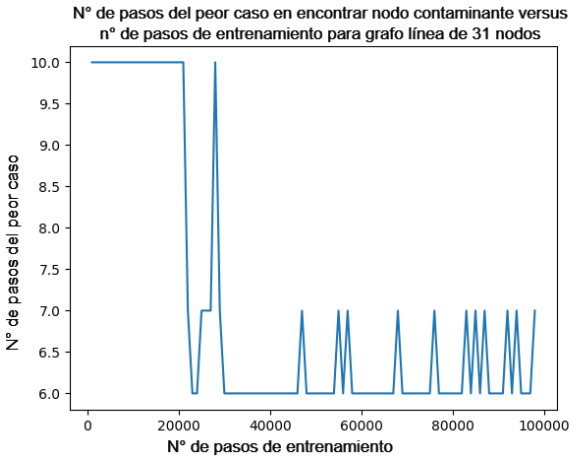


Comentarios: A diferencia de los resultados del grafo línea de 7 nodos, en este caso no es tan notorio distinguir cual es la mejor política en el gráfico de número de pasos promedio, a partir de las políticas creadas con 20.000 pasos de entrenamiento, pareciera que tienen resultados similares. La mejor política encontrada en este caso se demoró 3.266, donde su peor caso se demoró 4 muestreos en encontrar el nodo contaminante. La política encontrada se representa

como {4,3,4,2,4,3,4,1,4,3,4,2,4,3,4}, calzando con la política óptima. La mejor política se entrenó con 11000 pasos.

Grafo línea de 31 nodos : Se entrenaron 99 políticas realizando el mismo experimento anterior.

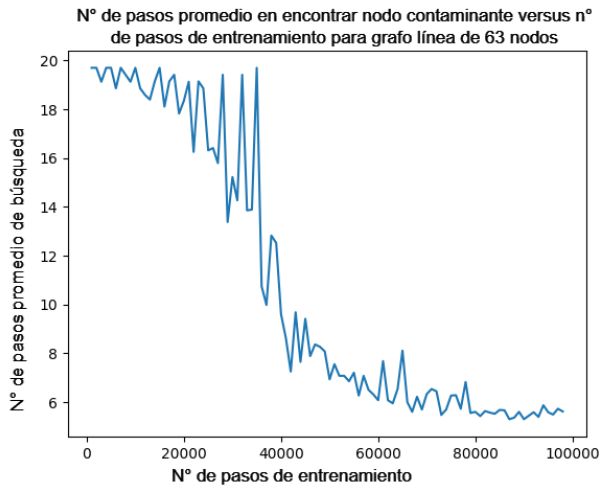


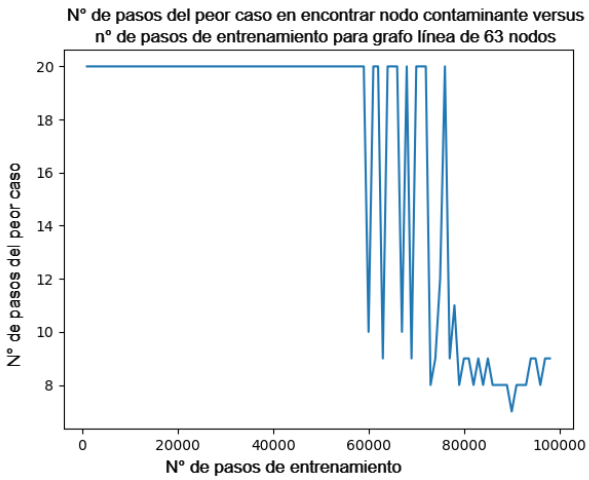


Comentarios: A medida que la dimensionalidad del grafo aumenta, se requieren más pasos de entrenamiento para obtener una buena política. Una buena política es una política con un número de pasos promedio similar al número de pasos promedio de la política óptima. Gracias al segundo gráfico donde se muestra el número de pasos que se obtiene en el peor caso, podemos ver que las primeras políticas no lograban aprender a buscar en ciertos nodos. Todas las políticas que en el peor caso obtuvieron 10 significa que en algún punto se demoran infinitamente en encontrar un nodo contaminado. Gracias al primer gráfico se puede observar como a medida que las políticas aumentan sus pasos de entrenamiento, los nodos aprendidos aumentan. Por ejemplo, observemos que al principio las políticas solo buscaban en un solo nodo y no buscaban en ningún otro nodo, por eso el mayor número de búsqueda promedio visto en el gráfico es 9.709 por el cálculo $\frac{10 \cdot 30+1}{31}$, esto significa que en 30 nodos se demoró más de 10 pasos en encontrarlo y en este caso una demora de 10 significa que nunca encuentra ese nodo o que se demora infinitamente en encontrar en ese nodo, (esto fue corroborado en el código, solo se supone esto para el mayor entendimiento de los gráficos).

La mejor política obtuvo un número de búsqueda promedio de 4.2258, siendo un poco mayor al número de búsqueda promedio de la política óptima 4.16. En el peor caso se demora 6 búsquedas. La política se entrenó con 32000 pasos.

Grafo línea de 63 nodos : Se entrenaron 99 políticas realizando el mismo experimento anterior.

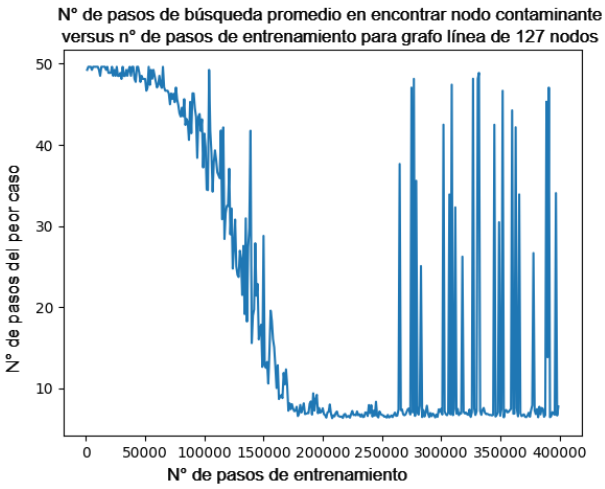


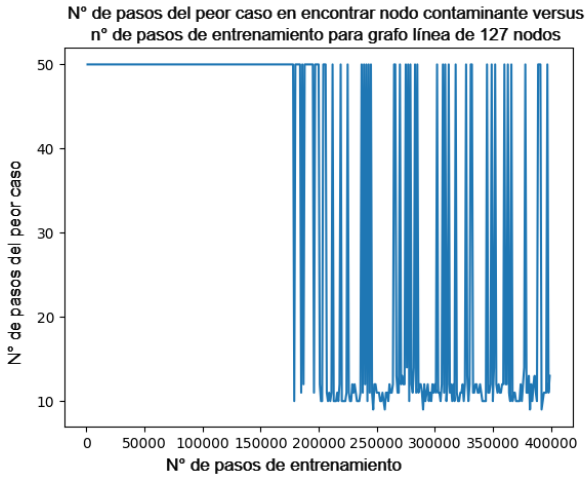


Comentarios: En este caso, es más notorio observar como se requieren más pasos de entrenamiento en un grafo con mayor dimensionalidad para obtener una buena política y también se logra observar como en más de la mitad de las políticas estas no aprenden a revisar todos los nodos. Esto último puede significar que como el entorno tiene más estados, se hace más complicado realizar políticas que revisen todos los nodos y que tengan un número de búsqueda promedio similar a la de la política óptima. De igual manera, gracias al gráfico de la izquierda, se puede ver como a medida que los pasos de entrenamiento de las políticas aumentan, estas gradualmente aprenden a buscar en todos los nodos. En este experimento, si el número de búsqueda del peor caso es mayor o igual a 20, significa que se demora infinitamente en buscar algún nodo. Al final del gráfico derecho, se logra observar cómo algunas políticas aprenden a revisar todos los nodos y como olvidan este aprendizaje, esto podría ser preocupante para un juego de mayor dimensionalidad y recalca que es mala idea entrenar solo una política, ya que, inclusive si se entrena con un número grande de pasos como 1.000.000, esta puede no aprender a revisar todos los nodos.

La mejor política obtuvo un número de búsqueda promedio en encontrar el nodo contaminante de 5.3, mayor al número de búsqueda promedio de la política óptima 5.09. La mejor política se entrenó con 86.000 pasos de entrenamiento y el número de búsqueda del peor caso fue de 8.

Grafo línea de 127 nodos : Se entrenaron 400 políticas realizando el mismo experimento anterior.





Comentarios: Se requirieron políticas entrenadas con un número de pasos de entrenamiento mayor que los experimentos anteriores para obtener buenas políticas. Lo más notorio en este gráfico es la versatilidad que se muestra, aquí, si el número de pasos de entrenamiento del peor caso es mayor o igual a 50 significa que la política no aprendió a revisar algún nodo. En el experimento se encontraron buenas políticas, que tienen un número promedio de búsqueda muy similar a la política óptima, sin embargo, independiente de la cantidad de pasos de entrenamiento, de igual forma parecieran olvidar a revisar ciertos nodos, (razón por la cual el último gráfico parece un código de barra). Esto puede deberse a la naturaleza estocástica del proceso de entrenamiento sumado la magnitud de la dimensión del juego.

En el cuadro 1 se encuentran resumido los resultados de los entornos grafo línea para 7, 15, 31, 63 y 127 nodos, donde se incluye el número de pasos promedio en encontrar el nodo contaminante de la mejor política obtenida, el número de pasos que se requirieron para obtener esa política y el número de pasos promedio de la política óptima π^*.

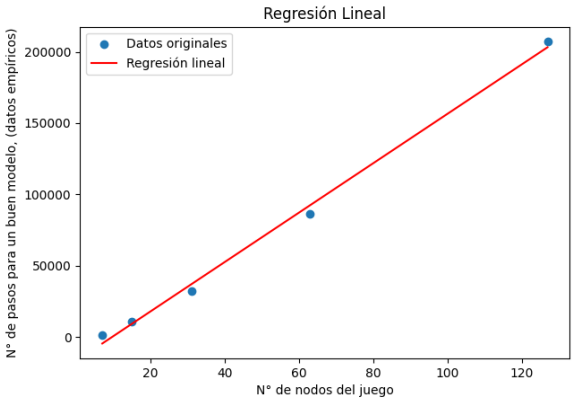
N° de nodos	T. promedio	T. peor caso	Pasos de entrenamiento	T. promedio π^*
7	2.428	3	1000	2.428
15	3.266	4	11000	3.266
31	4.2258	6	32000	4.16
63	5.3	8	86000	5.09
127	6.32	9	207000	6.055

Cuadro 1: Resumen de mejor política encontrada para los 5 experimentos

Estimado empírico de número de pasos de entrenamiento que se necesitan para realizar un buen modelo:

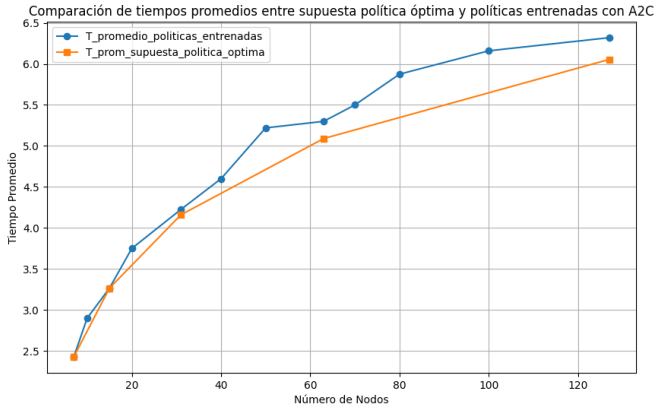
Un tema importante es el tiempo de ejecución que demora entrenar un agente, este tiene varios factores como la cantidad de pasos con la cual se entrenó el agente y también la dimensionalidad del grafo. Como se mostró en el cuadro 1, se puede apreciar como a medida que el número de nodos del grafo incrementa, el número de pasos de entrenamiento para obtener un buen modelo también lo hace. Para obtener una función aproximada que nos entregue un aproximado del número de pasos para obtener un buen modelo, utilizaremos los datos del cuadro 1 y se realizará una regresión lineal.

La regresión lineal arrojó una función de la forma $y = 1731.8x - 16768.1$.

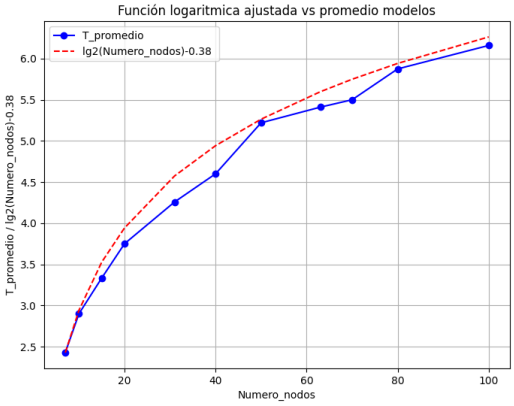


Número de pasos promedio en encontrar el nodo contaminante de una política entrenada en entorno búsqueda en grafo línea de carácter logarítmico:

En el marco teórico se logró obtener una fórmula del número de pasos promedio en encontrar el nodo contaminante de la supuesta política óptima, donde está depende del largo del grafo línea que tiene que ser de la forma $\mathrm{n} = 2^k - 1$. La fórmula es la siguiente $\frac{2^k k - 2^k + 1}{2^k - 1}$. Ahora bien, se realizaron varios entornos con grafos líneas de diferentes largos $n \in \{7,10,15,20,31,40,50,63,70,80,100,127\}$, se crearon políticas en esos entornos y se midió el número de pasos promedio en encontrar el nodo contaminante.

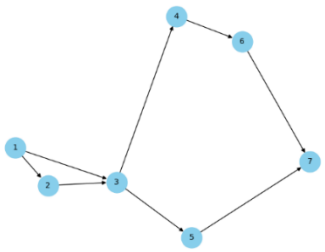


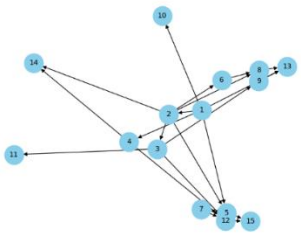
Observando el comportamiento del gráfico anterior, se puede apreciar como el número de pasos promedio en encontrar el nodo contaminante tiene una similitud a una función de carácter logarítmica. Para mostrar esto se hizo un gráfico ajustando la función logarítmica de base 2 restándole una constante. Esta constante es el resultado de la resta entre el primer número de pasos promedio en encontrar el nodo contaminante de la política óptima cuando el grafo tiene 7 nodos (2.428) y $log_2(7)$.

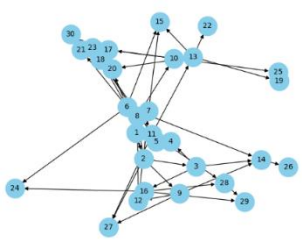


Grafos dirigidos acíclicos aleatorios: Para ver si el entrenamiento era efectivo para cualquier grafo y no solamente para grafos líneas, se realizaron políticas para entornos de búsqueda en DAG creados de manera aleatoria. Se crearon 3 DAG de 7,15 y 30 nodos respectivamente. Para cada DAG se entrenaron 99 políticas, en lo único que difieren es en la cantidad de pasos de entrenamiento, donde la primera política se entrenó con 1000 pasos, a segunda con 2000 y así sucesivamente hasta la política que se entrenó con 99000 pasos, esto con el objetivo de ver cómo va evolucionando el aprendizaje del agente a medida que los pasos de entrenamiento aumentan.

Grafo dirigido acíclico de 7 nodos.	Grafo dirigido acíclico de 15 nodos.	Grafo dirigido acíclico de 30 nodos

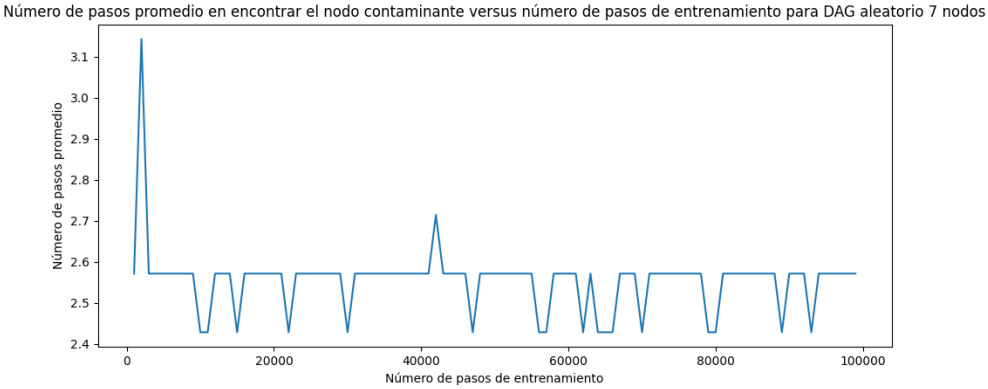


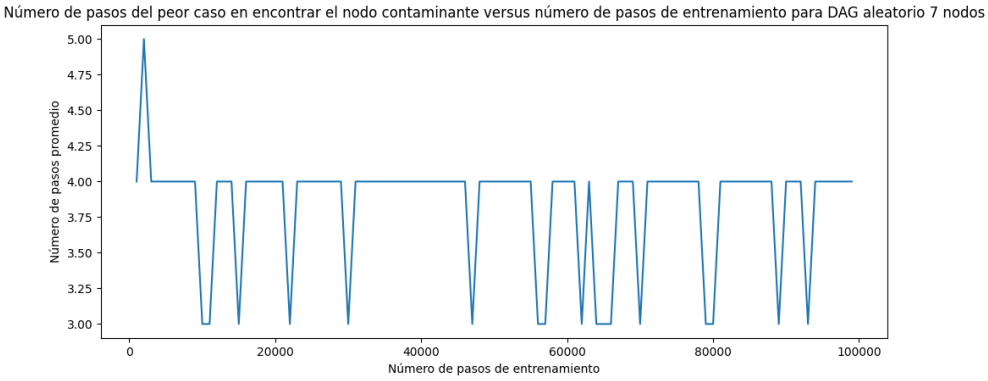




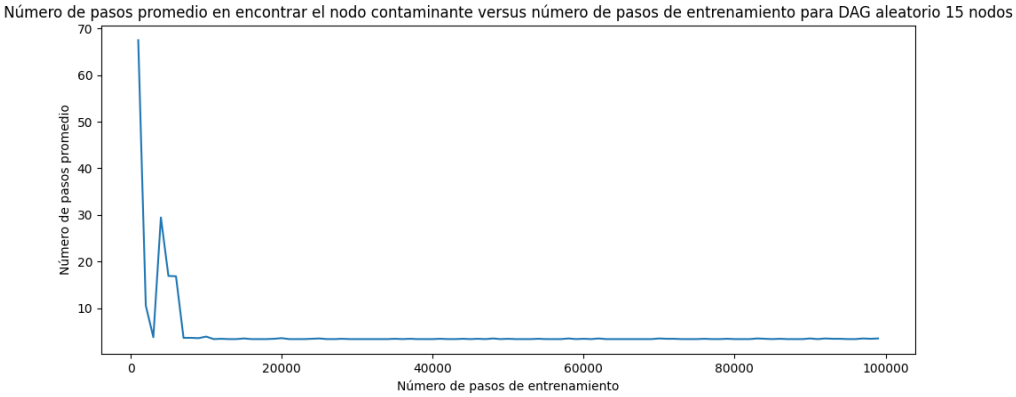
Los grafos tienen como grado máximo 2,3 y 3 respectivamente. Se rbealizaron los mismos experimentos que los grafos línea, solo para ver si realmente se obtiene una buena política. Sus gráficos son los siguientes:

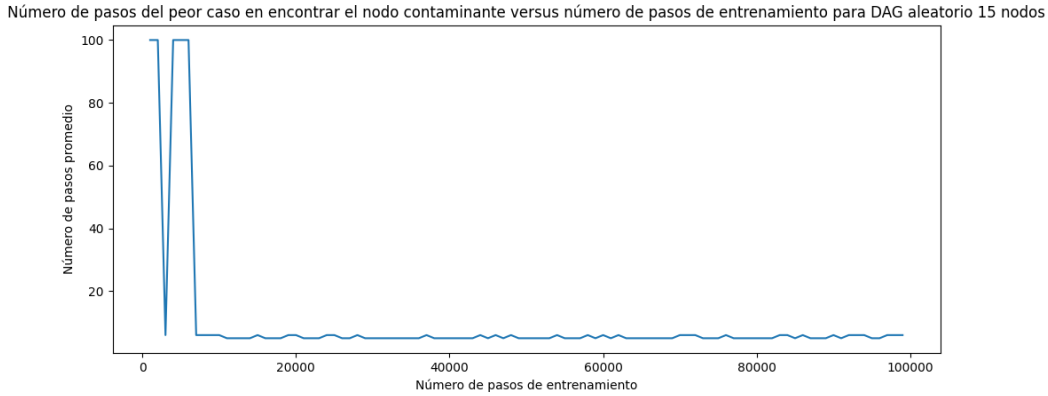
- Gráficos DAG aleatorio de 7 nodos:



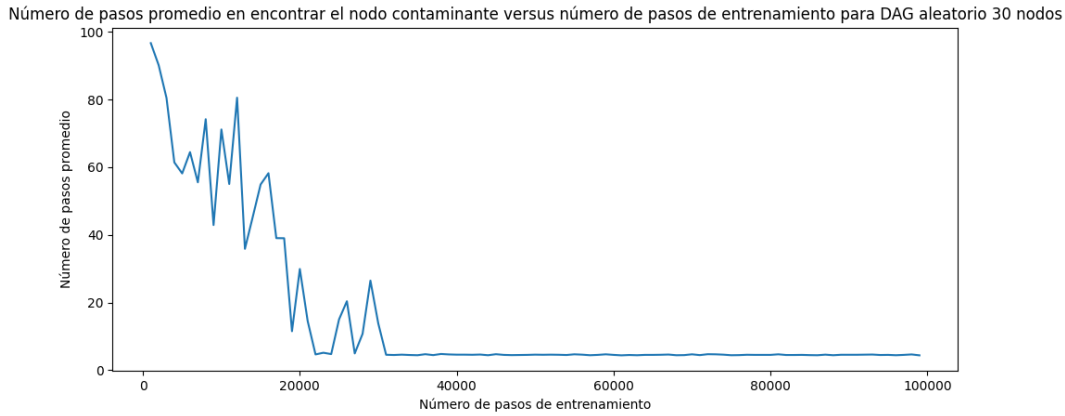


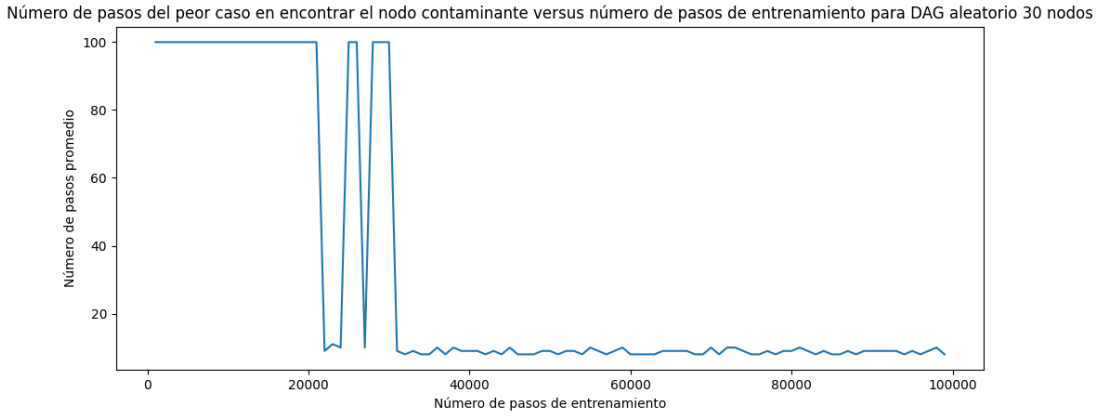
Gráficos DAG aleatorio 15 nodos:





- Gráficos DAG aleatorio 30 nodos:





Comentarios: Se puede observar que, para los tres ejemplos, a medida que los pasos de entrenamiento crecen, entonces las políticas se demoran menos en promedio en encontrar al nodo contaminante. Se logra notar que al igual que los resultados de las políticas entrenadas en

grafo línea, a medida que el entorno aumenta de dimensionalidad, más se demora en encontrar una buena política.

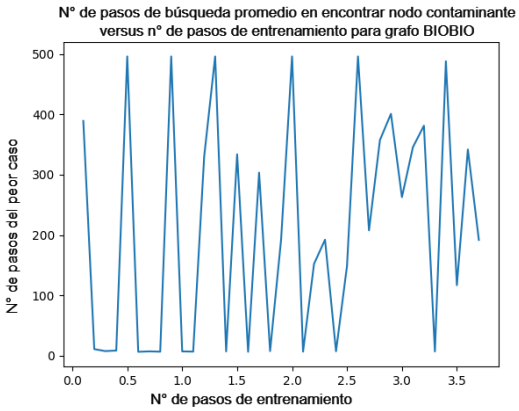
A continuación, se muestra una tabla donde se presenta el número de pasos promedio en encontrar el nodo contaminante de la mejor política para cada grafo, el número de búsquedas del peor caso, el número de pasos de entrenamiento que se tiene para cada política y el tiempo de demora de algoritmos conocidos para búsqueda de nodos en árboles que es de $O(\Delta\, log(n))$, donde n es el número de nodos del árbol y Δ es el grado máximo.

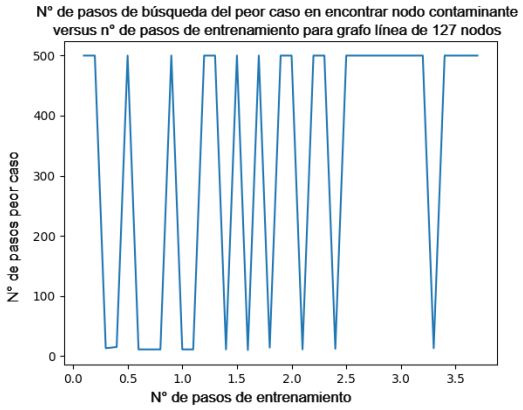
N° nodos	Tiempo promedio de mejor política.	Tiempo promedio algoritmo literatura	Número de pasos de entrenamiento.
7	2.428	1.69	9000
15	3.4	3.53	10000
30	4.366	4.431	60000

Comentarios: De la tabla se puede notar nuevamente como a medida que la dimensionalidad del juego incrementa, entonces el número mínimo para obtener un buen modelo de forma empírica también. Lo más notable de la tabla es que el tiempo promedio de las políticas entrenadas son mejores a excepción del primer caso de 7 nodos. Lo más probable es que se deba a la pequeña dimensionalidad del grafo. Con esto confirmamos que el entorno está bien definido y puede crear políticas iguales o mejores a la literatura actual en la mayoría de los casos.

Grafo dirigido acíclico red de distribución de agua superficial región del Biobío:

Se entrenaron 37 políticas, donde la primera consta con 100000 pasos de entrenamiento, la segunda con 200000 y así sucesivamente hasta la última política que consta de 3700000 pasos de entrenamiento.





Comentarios: Las políticas en este entorno más complicado le hace más difícil al agente aprender una buena política, como se puede notar en los gráficos, la versatilidad de estos se debe a la gran dimensionalidad del problema. Es similar al fenómeno que ocurrió con el grafo línea de 127 nodos. La mejor política encontrada se demora en promedio 6.46, es decir, en promedio se demora entre 6 a 7 muestreos para encontrar el nodo contaminante. En el peor caso, el nodo se demora 9 muestreos en encontrar al nodo contaminante. Recordemos que el grafo es de grado 5 y tiene 127 nodos por lo tanto según métodos convencionales con algoritmos actuales un buscador de nodos se demora $5\ log(127) = 10.5$ en promedio.

Conclusión

Este estudio ha abordado la relevancia y la eficacia del uso de técnicas de aprendizaje reforzado en el modelamiento de redes de agua superficial para la identificación de nodos contaminantes. A través de un enfoque sistemático que combina teoría y aplicaciones prácticas, esta investigación ha contribuido significativamente al campo del aprendizaje reforzado y la gestión de recursos hídricos.

El trabajo inició con la hipótesis de que el aprendizaje reforzado podría mejorar la eficiencia en la identificación de nodos contaminantes en redes de agua, incluso en estructuras complejas como los grafos dirigidos acíclicos de la región del Biobío. Dicha hipótesis se basó en la premisa de que las técnicas de aprendizaje reforzado, aplicadas en un contexto de dimensionalidad escalable, podrían superar a los métodos tradicionales de búsqueda. La

relevancia de esta investigación radica en su potencial para ofrecer soluciones más eficientes y precisas en la gestión de recursos hídricos.

Respecto a los objetivos específicos, se ha logrado:

1. Desarrollar políticas de agentes utilizando algoritmos Q-learning y A2C en el juego de tres en línea, proporcionando una base sólida para el entrenamiento de políticas en contextos más complejos.
2. Comparar eficazmente políticas entrenadas en entornos aleatorios, determinista y mixtos, demostrando la adaptabilidad y robustez de las políticas desarrolladas.
3. Crear y evaluar políticas en entornos de grafos dirigidos acíclicos, incluyendo el caso específico de la red de agua de la región del Biobío, confirmando la aplicabilidad de estas técnicas en contextos reales y complejos.
4. Comparar el rendimiento de las políticas entrenadas con algoritmos actuales, destacando mejoras en eficiencia y precisión.

Entre las fortalezas de esta investigación se encuentra la innovadora aplicación de técnicas de aprendizaje reforzado en un área práctica y crucial. Además, el estudio ha demostrado la escalabilidad y adaptabilidad de estas técnicas en diversos entornos. Sin embargo, también se reconocen limitaciones, como la dependencia de una cantidad significativa de datos para el entrenamiento efectivo y posibles desafíos en la interpretación y generalización de los resultados. Comparativamente, esta investigación ha mostrado avances respecto a otros estudios en el área, especialmente en términos de eficiencia y adaptabilidad de las políticas desarrolladas. En conclusión, este trabajo representa un avance significativo en la aplicación de aprendizaje reforzado para la identificación eficiente de contaminantes en redes de agua, con potencial para impactar positivamente en la gestión de recursos hídricos y la sostenibilidad ambiental.

Referencias

Torres Beristáin, B., González López, G., Rustrián Portilla, E., & Houbron, E. (2013). Enfoque de cuenca para la identificación de fuentes de contaminación y evaluación de la calidad de un río, Veracruz, México. Revista Internacional de Contaminación Ambiental, 29(3), 135-146. http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S0188-49992013000300001&lng=es&tlng=es

Gutiérrez Pérez, J. A. (2021). Monitorización, detección y estimación de estados de fallo en la calidad del agua de redes de distribución urbanas [Tesis doctoral, Universitat Politècnica de València]. http://hdl.handle.net/10251/169363

Madera, Lisbeth C, Angulo, Luis C, Díaz, Luis C, & Rojano, Roberto. (2016). Evaluación de la Calidad del Agua en Algunos Puntos Afluentes del río Cesar (Colombia) utilizando Macroinvertebrados Acuáticos como Bioindicadores de Contaminación. *Información tecnológica*, *27*(4), 103-110. https://dx.doi.org/10.4067/S0718-07642016000400011

Ashaw Muñoz, M. I., Navarro, D., & Gil Sánchez, E. (2020). Modelación matemática aplicada a la red de distribución de agua potable del distrito de Penonomé, República de Panamá. *Tecnociencia, 22*(2), 45-67. https://doi.org/10.48204/j.tecno.v22n2a3

Goodrich, M. T., & Tamassia, R. (2013). Graph algorithms: Theory and practice. Pearson.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Algorithms on graphs. MIT Press.

Fayyad, U., Yaghini, R. S., & Agarwal, A. (2002). Label propagation: A powerful tool for graph analysis. IEEE Intelligent Systems, 17(3), 33-38. doi:10.1109/MIS.2002.1006702

Liu, T.-Y., Zhou, M.-X., & Zhang, Y.-C. (2009). A survey on the detection of communities in graphs. ACM Computing Surveys (CSUR), 41(4), 16. doi:10.1145/1572997.1573001

Barrat, A., Barthélemy, M., & Vespignani, A. (2008). The spread of epidemics on networks. Cambridge University Press. doi:10.1017/CBO9780511750932

Nielsen, M. (2019). Neural Networks and Deep Learning. Recuperado de http://neuralnetworksanddeeplearning.com/index.html

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: MIT Press.

Xu, J., Wang, Z., Zhang, X., Li, X., & Zhang, Y. (2022). Using graph search to detect COVID-19 contaminated water networks. Water Research, 188, 116868.

Wang, W., Wang, H., Wang, Y., & Li, W. (2022). A graph-based approach for COVID-19 contaminated water detection. Water, 14, 10264.

Anexos

Los códigos de esta sección se pueden ver en el siguiente enlace:

https://github.com/JaimeRodriguezCh/Reinforcement-Learning.

Ejemplos:

Ejemplo 1. Computar la compuerta NAND con una neurona perceptrón.

Sea x_1 y x_2, variables lógicas, entonces la tabla de verdad lógica para la proposición NAND es:

x_1	x_2	NAND
1	1	0
1	0	1
0	1	1
0	0	1

Para crear una NA perceptrón que compute la proposición NAND primero debemos darnos cuenta de que es una NA que recibe dos señales o entradas y que debe arrojar un $p(w^T x + b) \in \{0,1\}$ tal que:

$w_1 + w_2 + b \leq 0$, $w_1 + b > 0$, $w_2 + b > 0$, $b > 0$.

Fijarse que con $w_1 = w_2 = -2$ y $b = 3$, se puede lograr crear una NA perceptrón que compute la proposición NAND.

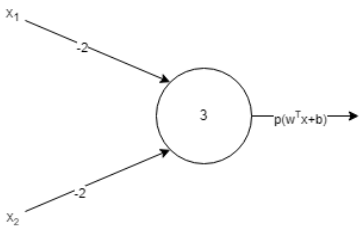


Ilustración 9. Neurona perceptrón que computa compuerta NAND.

Ejemplo 2. Formulación vectorial de una RNA.

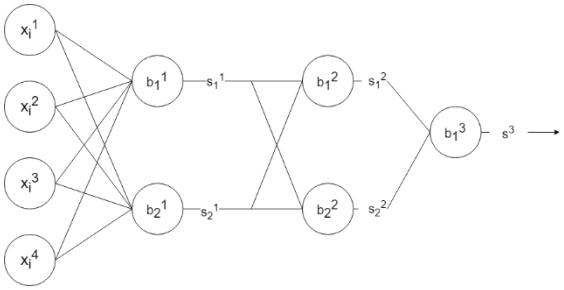


Ilustración 10. Red neuronal artificial de 2 capas ocultas.

La ilustración 2 nos muestra una RNA con una capa de entrada con 4 NA, con 3 conjuntos de pesos, 3 conjuntos de sesgos y 4 conjuntos de salidas, vectorialmente se define:

Pesos: $W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 & w_{14}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 & w_{24}^1 \end{bmatrix}$ $W^2 = \begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{bmatrix}$ $W^3 = \begin{bmatrix} w_{11}^3 \\ w_{21}^3 \end{bmatrix}$

Sesgos: $B^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$ $B^2 = \begin{bmatrix} b_1^2 \\ b_2^2 \end{bmatrix}$ $B^3 = \begin{bmatrix} b_1^3 \end{bmatrix}$

salidas capa de entrada: $S^0 = \begin{bmatrix} x_i^1 \\ x_i^2 \\ x_i^3 \\ x_i^4 \end{bmatrix}$

salidas primera capa oculta: $S^1 = f(W^1 S^0 + B^1) = \begin{bmatrix} f(w_{11}^1 x_i^1 + w_{12}^1 x_i^2 + w_{13}^1 x_i^3 + w_{14}^1 x_i^4 + b_1^1) \\ f(w_{21}^1 x_i^1 + w_{22}^1 x_i^2 + w_{23}^1 x_i^3 + w_{24}^1 x_i^4 + b_2^1) \end{bmatrix} = \begin{bmatrix} s_1^1 \\ s_2^1 \end{bmatrix}$

salidas segunda capa oculta: $S^2 = f(W^2 S^1 + B^2) = \begin{bmatrix} f(w_{11}^2 s_1^1 + w_{12}^2 s_2^1 + b_1^2) \\ f(w_{21}^2 s_1^1 + w_{22}^2 s_2^1 + b_2^2) \end{bmatrix} = \begin{bmatrix} s_1^2 \\ s_2^2 \end{bmatrix}$

salidas capa de salida: $S^3 = f(W^3 S^2 + B^3) = f(w_{11}^3 s_1^2 + w_{21}^3 s_2^2 + b_1^3) = s_1^3$

Ejemplo 3. Entrenamiento con algoritmo descenso del gradiente de NA sigmoidea que computa compuerta NAND en Python.

Observar que para entrenar una NA sigmoidea que compute la compuerta NAND el conjunto de entrenamiento es $T = \{((0,1),1);((1,0),1);((1,1),0)\}$, donde su largo es $n = 4$, los pesos y sesgos de la NA son $w = (w_1, w_2)$ y $b = b$ y la función de costo es $C_T(w,b) = \frac{1}{2n}\left[\left(\frac{1}{1+e^{-(w_1x_{11}+w_2x_{12}+b)}} - y_1\right)^2 + \left(\frac{1}{1+e^{-(w_1x_{21}+w_2x_{22}+b)}} - y_2\right)^2 + \left(\frac{1}{1+e^{-(w_1x_{31}+w_2x_{32}+b)}} - y_3\right)^2 + \left(\frac{1}{1+e^{-(w_1x_{41}+w_2x_{42}+b)}} - y_4\right)^2\right]$. El código en Python del ejemplo es el visto en el Código 1, fijarse que para calcular $\frac{\partial C_T}{\partial w_1}$, $\frac{\partial C_T}{\partial w_2}$ y $\frac{\partial C_T}{\partial b}$, se ocupó la librería Sympy.

El criterio de parada se limita a encontrar pesos y sesgos que evaluados en $C_T(w,b)$ den un valor máximo de 0.01. A medida que las iteraciones crecen, la función de costo decrece. El algoritmo se detuvo en la iteración número 588039, se obtuvieron los pesos y sesgos $w_1 = -3.21$, $w_2 = -3.21$, $b = 4.92$ y la función de costo obtuvo el valor de $C_T(w,b) = 0.009$. Las salidas de la NA entrenada fueron $h(0,0) = 0.99$, $h(0,1) = 0.84$, $h(1,0) = 0.84$, $h(1,1) = 0.18$, donde si nos fijamos, son valores similares a las etiquetas del conjunto de entrenamiento. En la ilustración 20 se muestra un gráfico que evidencia el valor que da la función de costo cuando la NA estaba entrenada con 0, 10, 100, 1000, 10000, 100000 y 588039 iteraciones respectivamente. Se logra observar el decrecimiento de la función de error a medida que las iteraciones del algoritmo del descenso del gradiente crecen.

Código 1: Ejemplo 3 NA sigmoidea que computa NAND entrenada con descenso del gradiente

```
#inicializacion de librerias
from math import exp
import sympy as sp
from sympy.utilities import lambdify
import matplotlib.pyplot as plt

# Inicializacion de parametros
x11,x12,x21,x32,y4=0
y1,x22,y2,x31,y3,x41,x42=1
n=4

# Inicializacion de variables
w1,w2,b=0.5

# Definicion de funcion de costo y variables
w1, w2, b= sp.symbols('w1 w2 b')
C = (1/(2*n))*(((1 / (1 + sp.exp(-(w1*x11+w2*x12+b))))-y1)**2+
((1 / (1 + sp.exp(-(w1*x21+w2*x22+b))))-y2)**2+
((1 / (1 + sp.exp(-(w1*x31+w2*x32+b))))-y3)**2+
((1 / (1 + sp.exp(-(w1*x41+w2*x42+b))))-y4)**2)
C_eval = lambdify((w1,w2,b), C, "numpy")

# Definicion de derivadas parciales
derivada_w1 = sp.diff(C, w1)
derivada_w1_eval = lambdify((w1,w2,b), derivada_w1, "numpy")
derivada_w2 = sp.diff(C, w2)
derivada_w2_eval = lambdify((w1,w2,b), derivada_w2, "numpy")
derivada_b = sp.diff(C, b)
derivada_b_eval = lambdify((w1,w2,b), derivada_b, "numpy")

# Algoritmo descenso del gradiente
while C_eval(W1,W2,B)>0.01 :
    W1=W1-eta*(derivada_w1_eval(W1,W2,B))
    W2=W2-eta*(derivada_w2_eval(W1,W2,B))
    B=B-eta*(derivada_b_eval(W1,W2,B))
```

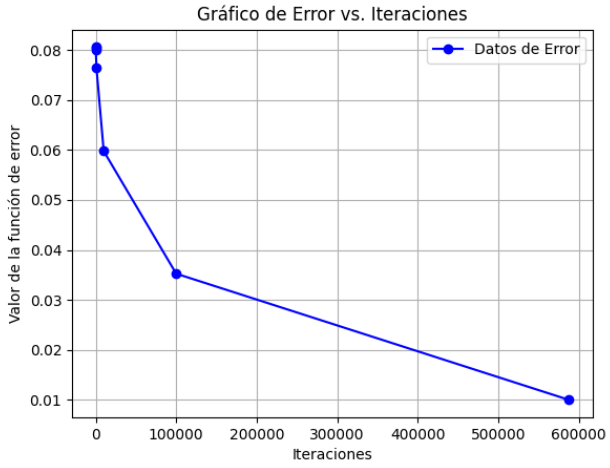


Ilustración 11. Valor de la función de costo en las iteraciones 0, 10, 100, 1000, 10000, 100000, 588039.

Ejemplo 4 (En repositorio de GitHub). Red neuronal artificial (RNA) para el reconocimiento de dígitos escritos a mano utilizando la base de datos MNIST.

La base de datos MNIST contiene imágenes de dígitos manuscritos del 0 al 9. Cada imagen, en escala de grises, tiene una matriz asociada de tamaño 28x28 píxeles que representa un solo dígito, con elementos de la matriz en el rango [0,1].

La RNA consta de una capa de entrada de longitud 784, que representa la cantidad de píxeles en una imagen 28x28. Además, cuenta con una única capa oculta de 30 neuronas y una capa de salida de 10 neuronas, que representa la etiqueta del dígito ingresado como imagen

El código, compuesto por 74 líneas, es una actualización del código original de Michael Nielsen, disponible en su libro "Neural Networks and Deep Learning" (2019, Capítulo 1) [http://neuralnetworksanddeeplearning.com/chap1]. En el código, se implementó el descenso del gradiente por mini lotes para entrenar la RNA, utilizando el método de retro propagación para calcular derivadas parciales de la función de costo. Se recomienda revisar el código directamente para obtener más detalles. El enlace al proyecto actualizado de Michael Nielsen se encuentra en el repositorio de GitHub mencionado al inicio de esta sección, la actualización se hizo porque el código original no funciona para librerías de Python actuales. El resultado es una RNA con una precisión del 95.1% en el reconocimiento de dígitos escritos a mano.

Ejemplo 6 (Grafo línea de 7 nodos y política determinista óptima). Para un grafo línea de 7 nodos, $V := \{1,2,3,4,5,6,7\}$, la política determinista revisa primero en el nodo 4. En lo que sigue hay tres opciones:

- Si el nodo 4 es el nodo fuente termina el juego y la política se demoró 1 paso
- Si el nodo 4 está contaminado entonces los nuevos nodos válidos donde puede estar el nodo fuente son $\{1,2,3\}$ por ser los predecesores de 4. En este punto la política revisa en el nodo 2 y nuevamente hay tres opciones:
  - Si el nodo 2 es el nodo fuente, el juego termina y la política se demora 2 pasos.
  - Si el nodo 2 está contaminado, entonces por lógica del juego el nodo fuente es el 1 ,ya que, es el único predecesor, lo que concluye con una demora de 3 pasos.
  - Si el nodo 2 está limpio, entonces por lógica del juego el nodo fuente es el 3 ,ya que, es el único sucesor, lo que concluye con una demora de 3 pasos.
- Si el nodo 4 está limpio entonces los nuevos nodos válidos donde puede estar el nodo fuente son $\{5,6,7\}$ por ser los sucesores de 4. En este punto la política revisa en el nodo 6 y nuevamente hay tres opciones:
  - Si el nodo 6 es el nodo fuente, el juego termina y la política se demora 2 pasos.
  - Si el nodo 6 está contaminado, entonces por lógica del juego el nodo fuente es el 5 ,ya que, es el único predecesor, lo que concluye con una demora de 3 pasos.
  - Si el nodo 6 está limpio, entonces por lógica del juego el nodo fuente es el 7 ,ya que, es el único sucesor, lo que concluye con una demora de 3 pasos.

Para obtener un tiempo promedio de demora, lo que se hace es observar cuanto se demora una política en todos los casos posibles y dividirlo en la cantidad de casos posibles, en este ejemplo, si el nodo fuente está en el nodo 1 la política se demora 3 pasos/tiempos, si esta en el 2, el tiempo de demora o pasos es de 2, y así sucesivamente. Si sumamos todos los tiempos de demora

daría el valor de $3+2+3+1+3+2+3=17$, concluyendo un tiempo promedio de $\frac{17}{7}=2.428$.

También es importante observar el peor caso, que en este ejemplo es 3.

En la Ilustración 21 se puede ver cómo funciona la supuesta política óptima en el ejemplo:

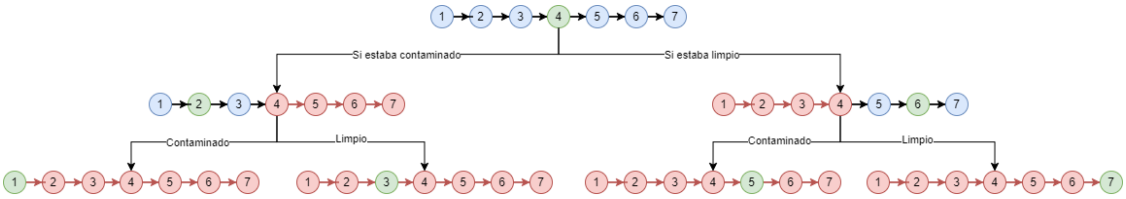


Ilustración 12.Representación de política óptima para un grafo línea de 7 nodos

Algoritmos y funciones:

Algoritmo 1 Descenso del Gradiente para una RNA

function DESCENSODELGRADIENTE($C_T(\cdot,\cdot)$, η, R, w, b)
  while $C_T(w,b) \neq 0$ o se cumpla algún criterio de parada **do**
    for $i = 1$ hasta $|w|$ **do**
      $w_i \leftarrow w_i - \eta \frac{\partial C_T}{\partial w_i}$
    end for
    for $j = 1$ hasta $|b|$ **do**
      $b_j \leftarrow b_j - \eta \frac{\partial C_T}{\partial b_j}$
    end for
  end while
  return w, b
end function

Algoritmo 2 Algoritmo Q-learning

Inicializar la tabla Q con valores arbitrarios
Inicializar el estado actual s
for cada episodio **do**
  while El juego aún no termina **do**
    Elegir una acción a basada en una política de exploración, (con cierta probabilidad escoge una acción aleatoria o escoge la mejor opción según la tabla Q actual).
    Tomar la acción a y observar la recompensa r y el nuevo estado s'
    Actualizar la tabla Q utilizando la ecuación de actualización Q:

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a')]$$

    Actualizar el estado actual s al nuevo estado s'
  end while
end for

Algoritmo 3 Temporal Difference (TD)

Inicializar la función de valor $V(s)$ para cada estado s
for cada paso **do**
    Calcular el error temporal (TD error):

$$\text{TD error} = r + \gamma \cdot V(s') - V(s)$$

    Actualizar la función de valor:

Algoritmo 4 Actor-Critic con Ventaja (A2C)

Inicializar los parámetros de la red del actor θ_π y la red del critic θ_v
Inicializar el entorno y observar el estado inicial s_0
for cada episodio **do**
    Inicializar el gradiente de la política: $d\theta_\pi \leftarrow 0$
    Inicializar el gradiente del valor: $d\theta_v \leftarrow 0$
    Observar el estado inicial s
    for cada paso del episodio **do**
        Escoger acción $a \sim \pi(a|s, \theta_\pi)$
        Ejecutar acción a y observar recompensa r y nuevo estado s'
        Calcular $A(s, a) = r + \gamma V(s', \theta_v) - V(s, \theta_v)$
        Actualizar $d\theta_\pi \leftarrow d\theta_\pi + \nabla_{\theta_\pi} \log \pi(a|s, \theta_\pi) A(s, a)$
        Actualizar $d\theta_v \leftarrow d\theta_v + \nabla_{\theta_v}(r + \gamma V(s', \theta_v) - V(s, \theta_v))^2$
        $s \leftarrow s'$
    end for
    Actualizar los parámetros de la red del actor: $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi d\theta_\pi$
    Actualizar los parámetros de la red del critic: $\theta_v \leftarrow \theta_v + \alpha_v d\theta_v$
end for

Algoritmo 5 Minimax para Tres en Línea

function MINIMAX(*tablero*, *profundidad*, *esMaximizador*)
    puntuacion $\leftarrow$ *Evaluar*(*tablero*)
    if *puntuacion* $= 10$ **then**
        return *puntuacion*
    end if
    if *puntuacion* $= -10$ **then**
        return *puntuacion*
    end if
    if No hay movimientos restantes y no hay ganadores **then**
        return 0
    end if
    if *esMaximizador* **then**
        mejor $\leftarrow -\infty$
        for cada celda vacía en *tablero* **do**
            Realizar movimiento en celda
            mejor $\leftarrow \max(mejor, \text{MINIMAX}(tablero, profundidad + 1, \text{false}))$
            Deshacer movimiento
        end for
        return *mejor*
    else
        mejor $\leftarrow \infty$
        for cada celda vacía en *tablero* **do**
            Realizar movimiento en celda
            mejor $\leftarrow \min(mejor, \text{MINIMAX}(tablero, profundidad + 1, \text{true}))$
            Deshacer movimiento
        end for
        return *mejor*
    end if
end function

Algoritmo 6 Obtención de predecesores o sucesores de v con dfs

```
1: function DFS(E, v, L, SoP)
2:     for u ∈ N⁻(E, v) si SoP == False de lo contrario u ∈ N⁺(E, v) do
3:         if u ∈ V then continue
4:         else
5:             L = L ∪ u
6:             L = dfs(E, u, L, SoP)
7:         end if
8:     end for
9:     return L
10: end function
```

Conjuntos de algoritmo 6: $N^+(E,v) = \{(v,u) \mid (v,u) \in E\}$ y $N^-(E,v) = \{(u,v) \mid (u,v) \in E\}$.

Algoritmo 7 Eliminador de nodos, (obtención de nodos factibles en cada paso)

```
1: function ELIMINADORDENODOS(V, E, v, CoL))
2:     NodosFactibles = dfs(E,v,{},False)
3:     if CoL == True then
4:         return NodosFactibles
5:     else
6:         NodosFactibles = NodosFactibles ∪ v
7:         NodosFactibles = {u ∈ V | u ∉ NodosFactibles}
8:         return NodosFactibles
9:     end if
10: end function
```

Algoritmo 8 Valores f_i para cada nodo i

```
1: function FUNCIONCLF(E.V, v))
2:     Sucesoresv= dfs(E,v,{}, True)
3:     f = (f_1, ..., f_|V|)
4:     for nodo ∈ V do
5:         if nodo ∈ Sucesores and nodo ≠ v then
6:             f_índice(nodo) = Contaminado
7:         else
8:             f_índice(nodo) = Limpio
9:         end if
10:    end for
11:    f_índice(v) = Fuente
12: end function
```

Obtención del gradiente mediante propagación hacia atrás en una RNA:

Sea T un conjunto de entrenamiento, R una RNA, w, b los vectores de pesos y sesgos de R, $f(\cdot)$ la función de activación de las NA en R, $C_T(w, b)$ una función de costo. El siguiente cálculo para obtener las derivadas parciales de la función C_T es el procedimiento de la propagación hacia atrás:

Para un peso w_{jk}^i:

$$\frac{\partial C_T}{\partial w_{jk}^i} = \frac{\partial C}{\partial z_j^i}\frac{\partial z_j^i}{\partial w_{jk}^i} \text{ Por regla de la cadena}$$

$$z_j^i = \sum_{k=1}^{L_{i-1}} w_{jk}^i s_k^{i-1} + b_j^i \text{ Por definición}$$

$$\frac{\partial z_j^i}{\partial w_{jk}^i} = s_k^{i-1} \text{ Cálculo de la derivada}$$

$$\frac{\partial C}{\partial w_{jk}^i} = \frac{\partial C}{\partial z_j^i} s_k^{i-1} \text{ Resultado final del cálculo}$$

Análogamente el cálculo para un sesgo b_j^i:

$$\frac{\partial C_T}{\partial b_j^i} = \frac{\partial C}{\partial z_j^i}\frac{\partial z_j^i}{\partial b_j^i} \text{ Por regla de la cadena}$$

$$\frac{\partial z_j^i}{\partial b_j^i} = 1 \text{ Cálculo de la derivada}$$

$$\frac{\partial C}{\partial b_j^i} = \frac{\partial C}{\partial z_j^i} 1 \text{ Resultado final del cálculo}$$

Entornos que no funcionaron y que vale la pena mencionar:

Se crearon más entornos que no lograron aprender a jugar y otros que se suponen son análogos. Estos tenían la misma estructura, solo cambiaban la recompensa del agente, a excepción de 1 que difiere de la manera de escoger el nodo contaminante. Las versiones predecesoras las mencionare en este espacio:

- Castigar al final: El primer entorno castigaba con la cantidad de búsquedas al final de una partida, es decir, cuando el agente encontraba el nodo contaminante obtenía una recompensa de $-contador\_de\_búsquedas$, y en los pasos intermedios, (cuando no encontraba el nodo fuente), obtenía recompensas de 0. La lógica detrás era que se pensaba que el agente iba a tratar de encontrar el nodo en la menor cantidad de pasos posibles, ya que, mientras más rápido encontraba el nodo, el castigo al total del episodio

era menor, por ejemplo, si encontraba el nodo fuente a la primera el castigo total es de -1, en cambio si lo encontraba en $|V|$ pasos el castigo era de $-|V|$. El resultado fue un agente que aprendió a nunca ganar, es decir, si encontraba un nodo que no era el nodo fuente, era su única acción infinitamente. Esto se podía arreglar de varias maneras, pero se decidió castigarlo paso a paso en lugar de castigarlo únicamente al final. El entorno actual castiga con -1 si el nodo revisado no es la fuente, y es análogo al castigo total que se tenía en esta primera implementación.

- Entornos análogos: Experimentando con las recompensas para cada entorno, se logró ver, que las siguientes recompensas eran análogas al modelo definido anteriormente:
  - Si se castiga con 0 si el nodo revisado no es la fuente y con 1 si lo es.
  - Si se castiga con $-contador\_de\_búsquedas$ si el nodo revisado no es la fuente y con 1 si lo es.
  - Si se castiga con -1 si el nodo revisado no es la fuente, se premia con 10 si se encontro el nodo fuente en menos de $log(|V|)$ pasos de búsqueda, con 5 si se demora menos de $|V|$ y con 1 si lo encuentra en cualquier otro caso.
- Manera de escoger el nodo fuente: Primeramente, el nodo fuente se escogía de manera aleatoria utilizando la librería random de python, escogiendo un nodo al azar del conjunto de nodos V en cada episodio. Aunque la máquina logra aprender a jugar, se logro notar una deventaja, y es que es probable que, en el entrenamiento del agente, puede que el nodo contaminante no se esconda en todos los nodos, lo que deja al agente sin saber que acción realizar en ciertos escenarios. Por ello, se implemento la creación de un vector copia del vector de nodos V pero con los indices desordenados, así el nodo fuente se esconde aleatoriamente, pero con la ventaja de visitar todos los nodos.

Cálculo 1, Formula de tiempo promedio para supuesta política óptima:

Para efectos prácticos, para cualquier política π se representará como una tupla donde el valor del nodo de la tupla representa el tiempo de demora en el cual se revisa ese nodo, por ejemplo, si tenemos un grafo con $V = \{1,2,3,4,5,6,7\}$, y una representación de una política en donde se revisa primero el 7 luego el 6 y así sucesivamente hasta llegar al 1 sería: $\{7,6,5,4,3,2,1\}$, de igual modo para la política π^* sería $\{3,2,3,1,3,2,3\}$.

A continuación, se mostrará el comportamiento de la política π^* en diversos escenarios que difieren en la cantidad de nodos

- Si la cantidad de nodos es $2^1 - 1 = 1$, entonces la política es trivial y el tiempo promedio es de 1 y la política se representará como $\{1\}$
- Si la cantidad de nodos es $2^2 - 1 = 3$, entonces la política se representa $\{2,1,2\}$ y el tiempo promedio es de $\frac{1 \cdot 2 + 3}{3} = \frac{5}{3}$
- Si la cantidad de nodos es $2^3 - 1 = 7$, entonces la política se representa $\{3,2,3,1,3,2,3\}$ y el tiempo promedio es de $\frac{5 \cdot 2 + 7}{3} = \frac{17}{7}$
- Si la cantidad de nodos es $2^4 - 1 = 15$, entonces la política se representa $\{4,3,4,2,4,3,4,1,4,3,4,2,4,3,4\}$ y el tiempo promedio es de $\frac{17 \cdot 2 + 15}{15} = \frac{49}{15}$
- …
- Si la cantidad de nodos es $n = 2^k - 1$, el tiempo promedio es de $\frac{1}{n}\sum_{i=1}^{k} i\, 2^{i-1}$, con $n = 2^k - 1$

Veamos que para el termino i-ésimo tenemos que:

$$\sum_{i=1}^{k} i\, 2^{i-1} = \sum_{i=1}^{k}\sum_{j=1}^{i} 2^{i-1} = \sum_{j=1}^{k}\sum_{i=j}^{k} 2^{i-1} = \sum_{j=1}^{k}\sum_{i=0}^{k-j} 2^{i+j-1} = \sum_{j=1}^{k} 2^{j-1}\sum_{i=0}^{k-j} 2^{i}$$

$$\sum_{j=1}^{k} 2^{j-1}\frac{2^{k-j+1}-1}{2-1} = \sum_{j=1}^{k} 2^{j-1}\, 2^{k-j+1} - \sum_{j=1}^{k} 2^{j-1} = \sum_{j=1}^{k} 2^{k} - \sum_{j=0}^{k-1} 2^{j}$$

$$= k2^k - \frac{2^k - 1}{2-1} = (k-1)2^k + 1$$

Esto da lugar a una fórmula general para obtener el tiempo promedio de demora que tiene nuestra supuesta política óptima determinista en un grafo línea de largo $n = 2^k - 1$ con $k \in N$

$$\left(\frac{1}{n}\right)\left((k-1)2^k\right)+1\Big) = \frac{2^k k - 2^k + 1}{n}$$

Formulación vectorial de una RNA:

Sea R una RNA y un vector $x \in \mathbb{R}^{L_0}$. La formulación vectorial de la primera capa es:

$$S^0 = [x^1 \quad x^2 \quad \cdots \quad x^{L_0}]^T$$

Sea L_i el número de NAs que hay en la capa i-ésima, Sea $b_j^i \in \mathbb{R}$ el sesgo de la NA j en la capa i, entonces la formulación vectorial de los sesgos de la capa i-esima viene dada por el vector:

$$B^i = \begin{bmatrix} b_1^i & b_2^i & \cdots & b_{L_i}^i \end{bmatrix}^T$$

Sea L_i el número de NA que hay en la capa i-ésima, Sea $w_{jk}^i \in \mathbb{R}$ el peso de la capa i-ésima, tal que k es la NA de la capa anterior $i-1$ que apunta a la NA j de la capa i, entonces la formulación matricial de los pesos de la capa i viene dada por la matriz:

$$W^i = \begin{bmatrix} w_{11}^i & w_{12}^i & \cdots & w_{1L_{i-1}}^i \\ w_{21}^i & w_{22}^i & \cdots & w_{2L_{i-1}}^i \\ \vdots & \ddots & \ddots & \vdots \\ w_{L_i1}^i & w_{L_i2}^i & \cdots & w_{L_iL_{i-1}}^i \end{bmatrix}$$

La formulación matemática de la salida de la capa i-ésima, viene dada por el vector

$$S^i = F^i(Z^i) \text{ Donde, } Z^i = W^i S^{i-1} + B^i \forall i \geq 1$$

Donde $S^i \in \mathbb{R}^{L_i}$, es decir es un vector del estilo $S^i = \begin{bmatrix} s_1^i & \cdots & s_{L_i}^i \end{bmatrix}^T$ y $F^i(\cdot)$ es la función que mapea vectores visto en el marco teórico.