UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Aprendizaje de representaciones simbólicas a partir de imágenes para problemas de planificación

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: Granados Bañuls, Alejandro

Tutor/a: Onaindia de la Rivaherrera, Eva

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Aprendizaje de representaciones simbólicas a partir de imágenes para problemas de planificación

TRABAJO FIN DE MÁSTER

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Alejandro Granados Bañuls
Director: Eva Onaindía De La Rivaherrera

Curso 2021-2022

Resumen

Para resolver un problema de planificación se requiere la especificación de un modelo simbólico del dominio de aplicación y la instancia a resolver por parte de un experto del dominio. Esto supone un obstáculo en el proceso de adquisición del conocimiento. Por otro lado, las técnicas de Aprendizaje Profundo aprenden representaciones que no son compatibles con los sistemas simbólicos que requieren los planificadores. En este trabajo se presenta una aproximación en la que se utilizan autocodificadores para aprender representaciones vectoriales de imágenes de un problema de planificación de modo que el resultado del autocodificador sea una representación simbólica del problema. Particularmente, se diseñará un autocodificador para aprender una representación de los estados del problema de planificación y otros autocodificadores para aprender las transiciones del sistema. La arquitectura diseñada se aplicará en la resolución de problemas en dos dominios, el juego del 8-puzzle y lights out, a partir de imágenes sin etiquetar que representen y definan mediante pares las transiciones de ambos dominios.

Palabras clave: Inteligencia Artifical, Planificación automática, Representación simbólica, Aprendizaje automático, Autocodificador

Abstract

Solving a planning problem requires the specification of a symbolic model of the application domain and the instance to be solved by a domain expert. This is an obstacle in the knowledge acquisition process. On the other hand, Deep Learning techniques learn representations that are not compatible with the symbolic systems required by planners. In this paper we present an approach in which autoencoders are used to learn vector representations of images of a planning problem so that the output of the autoencoder is a symbolic representation of the problem. Particularly, one autoencoder will be designed to learn a representation of the states of the planning problem and other autoencoders to learn the transitions of the system. The designed architecture will be applied to solve problems in two domains, the 8-puzzle game and the lights out, from unlabeled images that represent and define by pairs the transitions of both domains.

Key words: Artificial Intelligence, Symbolic representation, Automated Planning, Machine Learning, Autoencoder

Índice

Índice de figuras

CAPÍTULO 1
Introducción

El campo de la planificación inteligente intenta encontrar estrategias basadas en acciones para resolver problemas de un dominio particular. Para ello se utilizan lenguajes lógicos que permiten realizar una representación simbólica de las acciones y los estados de un problema. La necesidad de generar manualmente la codificación de un dominio supone un cuello de botella respecto al conocimiento porque sólo un experto en el dominio y en el lenguaje lógico puede codificar problemas de forma adecuada.

Para codificar problemas de planificación se requiere una inversión de tiempo importante. Primeramente, se debe especificar los predicados con los que se definirá el dominio y que serán los componentes constituyentes en forma de hechos de los estados del problema, tanto del estado inicial del problema como del estado que se desea alcanzar, llamado *estado objetivo*, como cualquiera de los estados que se generan durante el proceso de búsqueda y resolución del problema.

Las acciones se definen en términos de precondiciones y objetivos. Las precondiciones son restricciones que deben satisfacerse en un estado para que la acción sea aplicable. Una acción que puede ejecutarse en un estado conlleva una serie de efectos. Estos efectos implican un cambio de estado del problema que se definen mediante la adición y eliminación de hechos del estado. De esta forma el estado de un problema cambia con cada acción.

Como se puede ver, la codificación de los estados y acciones de un dominio puede llevar mucho tiempo. Esto es lo que se llama *cuello de botella del conocimiento*, ya que sólo un experto en el dominio y en el lenguaje simbólico puede realizar esta actividad (y no siempre se puede hacer de manera rápida).

Gracias a la evolución del campo del *machine learning*, más específicamente en la rama de las redes neuronales, han surgido muchos modelos para resolver diferentes tareas. Una de ellas es la capacidad de transformar datos (texto, imágenes, datos tabulares) a representaciones vectoriales. Estas representaciones vectoriales contienen las características más importantes de los datos proyectados por lo que se pueden usar para diferentes tareas. Una de estas tareas es la reconstrucción de las representaciones vectoriales a los datos originales. Por ejemplo, transformar una imagen a representación vectorial y después transformar esta representación vectorial otra vez a la imagen original.

Una técnica común para generar representaciones vectoriales es la conocida como *encoder-decoder*. Esta técnica no sólo es usada por el campo del *machine-learning* para generar representaciones vectoriales, sino que es usada en otros ámbitos (la codificación *jpg* por ejemplo). Con el *encoder-decoder* se codifican los datos usando el *encoder*. El *encoder* genera las representaciones vectoriales que serán usadas por el *decoder* para reconstruir los datos.

El presente proyecto pretende resolver el cuello de botella mencionado usando un *encoder-deocder* que genere representaciones vectoriales de los estados de un problema de planificación. Estas representaciones vectoriales podrán utilizarse posteriormente para generar codificaciones en un lenguaje lógico y así evitar este cuello de botella.

1.1 Motivación

La idea del presente proyecto surge a partir del artículo *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary* [3]. En este trabajo se presenta un modelo basado en redes neuronales para resolver problemas de planificación sin necesidad de utilizar una codificación simbólica del problema. Para ello se usa una arquitectura de redes neuronales tipo *encoder-decoder* (*Variational Autoencoders* para ser exactos) para generar representaciones vectoriales de los estados del problema, las cuales se definen dentro de lo que se conoce cómo espacio latente [30]. En el citado artículo el espacio latente es denominado espacio subsimbólico, pues dentro de ese espacio latente se representan vectorialmente los estados de un problema. Esa es la parte en la que se centra el presente proyecto, en generar un espacio subsimbólico (espacio latente) para los estados de un problema.

Las representaciones vectoriales de los estados de un problema podrían ser usadas después por otros modelos. Por ejemplo, se podría entrenar un modelo de *machine learning* para decidir el movimiento que se puede aplicar a un estado de un problema que está representado vectorialmente. Otro posible uso de las representaciones vectoriales de un estado del problema podría ser la transformación de dicha representación a una representación simbólica. Esto último resolvería parte del problema del cuello de botella, ya que no sería necesario que un experto codificara manualmente un dominio desde cero. En este caso, la codificación la realizaría el modelo aprendido, reduciendo así el tiempo invertido en la representación de un problema.

Primeramente se pensó en construir un modelo que se encargaría de generar representaciones vectoriales de estados de un problema en formato imagen y después codificar las representaciones vectoriales de dichos estados en un lenguaje lógico (es decir, a una representación simbólica), pero después de un tiempo se vio que esto se salía del alcance del proyecto. Después se decidió que en lugar de realizar esa codificación se realizaría un modelo basado en redes neuronales para aplicar acciones a estados de un problema. Este modelo se encargaría de codificar un estado a una representación vectorial y después transformaría la representación vectorial a otro estado en formato imagen. El nuevo estado será el resultado de aplicar una acción al estado que ha sido proyectado a representación vectorial.

1.2 Objetivos

Con lo anterior definido, se puede proceder a exponer los objetivos del presente proyecto:

- Mostrar que los estados de un problema en formato imagen se pueden representar vectorialmente conservando sus características más importantes.
- Construir un modelo basado en redes neuronales que sea capaz de procesar imágenes que representen estados de un problema de planificación. El procesamiento consistirá en dos pasos: proyectar la imagen que representa un estado a representación vectorial y después transformar esta representación vectorial a otra imagen.

La imagen generada por el modelo representará un nuevo estado del problema, el cual será el resultado de aplicar un movimiento válido al estado representado en la imagen de entrada. Además, se deberá implementar un módulo adicional capaz de indicar que movimiento se ha aplicado en el estado representado en la imagen de salida.

- Generar los datos necesarios para entrenar el modelo construido. Estos datos serán generados en base a dos problemas de planificación. Los datos se utilizarán para comprobar si el modelo es capaz de realizar una representación vectorial de forma correcta.

  Se utilizarán las imágenes generadas por el modelo así como los movimientos elegidos por el módulo adicional para determinar la correctitud de la representación vectorial. Se comprobará si las imágenes contienen estados correctos y si esos estados se han generado a partir de aplicar un movimiento al estado representado en la imagen de entrada. Además se comprobará si el movimiento elegido por el módulo adicional mencionado anteriormente se corresponde con el movimiento aplicado que resulta en la imagen de salida.

CAPÍTULO 2
Antecedentes

En este capítulo se presentará unas nociones generales sobre la Planificación Automática así cómo su relación con el presente proyecto. Acto seguido se procederá a proporcionar información sobre *Machine Learning*, más específicamente el campo del *Deep Learning*. Se hablará sobre como funcionan las redes neuronales con un ejemplo básico y se mostrará varios ejemplos de arquitecturas. Finalmente se mencionará la arquitectura usada en el presente proyecto, que será una red tipo *U-Net* pero con algunas modificaciones.

2.1 Planificación

En la mayor parte de tareas que conllevan una complejidad media o alta se suele definir un primer paso de planificación para poder abordarlas de forma razonable. Por ejemplo, si se intenta resolver una tarea compleja como la construcción de un edificio, sin una correcta planificación el resultado puede acabar siendo, con altas probabilidades, un fracaso. Por esta razón la planificación es una de las partes cruciales de un proyecto o de la resolución de un problema complejo.

La resolución de un problema de planificación requiere definir previamente el dominio de actuación donde se especifican las **acciones** que pueden realizarse en dicho dominio. La aplicación de una acción conlleva un cambio en el **estado actual** del problema, que dependerá del cumplimiento de una serie de **condiciones** en el estado y tendrá una serie de **efectos** que dará lugar a un nuevo estado.

Un problema de planificación es un problema de toma de decisiones en la que el planificador o resolutor debe decidir la acción a ejecutar en un estado del problema entre varias acciones aplicables en el estado con el fin de alcanzar un objetivo particular. Si para un estado de un problema hay varias acciones aplicables porque ese estado cumple las condiciones de varias acciones, ¿cuál es la mejor acción a tomar?. En el caso de elegir una acción, ¿Qué métricas se pueden usar para elegir la mejor acción?. Una posible solución es elegir aquellas acciones que generen el resultado del problema los más rápido posible, o aquellas que usen la menor cantidad de recursos, o tal vez una media de las dos.

La Planificación Automática engloba una serie de técnicas para poder resolver un problema de planificación partiendo de un estado inicial y con el fin de llegar a un estado final donde se cumplan los objetivos que se persiguen [13]. Los elementos que intervienen en un problema de planificación son:

1. Definición de las acciones del dominio.

2. Definición del estado inicial del problema.

3. Algoritmo usado para resolver el problema.
4. Definición del estado objetivo del problema.
5. La función heurística que utilizará el planificador para guiar la búsqueda y que puede ser óptima o no

Uno de los lenguajes más populares para definir una tarea de planificación es el *Planning Domain Definition Language* (**PDDL**) [25, 11]. Con este lenguaje se definen los predicados y las acciones del dominio de planificación. Para definir una instancia de un problema se debe definir el estado inicial del mismo mediante la instanciación de los **predicados** que da lugar al conjunto de **hechos** que compondrán al estado inicial así como los hechos que se quieren alcanzar en el estado final del problema (estado objetivo). Por ejemplo, supongamos que hay un contenedor llamado *contenedor 1* en un barco y queremos depositar este contenedor en un camión. Uno de los posibles predicados que definiría el estado inicial sería:

$$(esta\_en\ ?contenedor\ ?lugar\_del\_contenedor)$$

Siendo $?contenedor$ y $?lugar\_del\_contenedor$ dos variables que representan un contenedor y el lugar donde puede estar el contenedor, respectivamente. A partir de este predicado se puede definir un hecho, que es la instanciación del predicado con objetos definidos en la instancia del problema:

$$(esta\_en\ contenedor1\ barco)$$

Y uno de los posibles hechos del estado objetivo seria:

$$(esta\_en\ contenedor1\ camion)$$

Existen múltiples paradigmas para resolver un problema de planificación, esto es, para encontrar una secuencia de acciones o plan que aplicado en el estado inicial del problema permita llegar a un estado final donde se cumplan los objetivos. Entre estos paradigmas podemos citar la Planificación de Orden Parcial (POP) [20], planificación heurística [18] o algoritmos basados en grafos [6].

La especificación de un problema de planificación requiere de un experto del dominio para describir el conocimiento del problema y su correspondiente modelización en PDDL, lo que conlleva definir los predicados, las acciones, las condiciones y efectos de las acciones, etc. Precisamente, en este trabajo se ofrece un mecanismo para solventar este **cuello de botella de conocimiento**, buscando una solución capaz de interpretar el conocimiento de un problema a partir de imágenes de los estados del problema.

2.1.1. Definición formal

A continuación se presenta la definición formal de una tarea de planificación. Se define primeramente el concepto de dominio y posteriormente el una instancia de problema de un dominio de planificación.

Un dominio de planificación se define como una 2-tupla $D = (P, A)$ donde P es el conjunto de predicados que se utilizan para modelar las propiedades del dominio y A representa el conjunto de posibles acciones que se pueden realizar dentro del dominio. Una acción $a \in A$ se define como una 3-tupla $a = (Pre, Add, Del)$ donde Pre es el conjunto

de condiciones que debe cumplirse para que la acción sea aplicable en un estado, *Add* es el conjunto de efectos positivos, es decir los nuevos hechos que se añaden al estado resultante de ejecutar la acción, y *Del* es el conjunto de efectos negativos (hechos que se eliminan en el estado resultante).

Un cambio de estado se define como una función $f : S \times A \rightarrow S$, donde S es el conjunto de posibles estados del problema. Una instancia de un problema es una 3-tupla $Pr = (D, I, G)$, donde D es el dominio de planificación, I es un conjunto de hechos que representa el estado inicial del problema y G es un conjunto de hechos que representa el estado objetivo que se quiere alcanzar. Una solución a un problema Pr es una secuencia de acciones o plan $\langle a_1, a_2, \ldots, a_n \rangle$, $a_i \in A \ \forall i$

2.1.2. Cuello de botella de conocimiento

Para resolver un problema de planificación se requiere previamente la definición de un modelo del dominio por parte de un experto. Este es un proceso costoso en el que hay que definir predicados, acciones, relaciones entre objetos, etc. Y es precisamente en este punto donde las técnicas de *machine learning* pueden aportar sus beneficios. En el presente proyecto se va a estudiar la capacidad de una red neuronal de aportar a este **cuello de botella de conocimiento** que hasta ahora lo debía realizar una persona con información y conocimiento del problema.

En concreto, el planteamiento que se va a realizar es el siguiente: a partir de una imagen que representa un estado de un problema de planificación, analizar si una red neuronal es capaz de generar otro estado que sea válido mediante una representación vectorial del estado original, es decir, generar un estado que sea resultado de la aplicación de una acción en el estado de la imagen. Por ejemplo, a partir de una imagen de un tablero de ajedrez con las fichas posicionadas en sus respectivas celdas, analizar si una red neuronal es capaz de generar un movimiento válido de una ficha en una nueva imagen.

Con todo lo anterior dicho se puede afirmar que la red neuronal va a aprender a modelar el conjunto de acciones A y predicados P de forma implícita, aliviando así el cuello de botella antes definido. El objetivo no es que la red neuronal aprenda a hacer movimientos óptimos o que lleven de forma óptima a la solución de problema, sino que simplemente aprenda a realizar movimientos válidos.

2.2 Redes neuronales

El área conocida como *machine learning* está tomando una gran importancia en la sociedad y panorama académico actual, especialmente la tecnología de Redes Neuronales. Cada pocos meses se presenta una red neuronal que bate récords en muchos de los campos de la Inteligencia Artificial. Una muestra de ello son las recientes publicaciones sobre modelos de difusión que generan imágenes en el campo de la visión por computador [7], los modelos de traducción automática en el campo del tratamiento del lenguaje natural [34], o aquellos que se dedican a discernir entre peticiones reales o malignas a un servidor web en el campo de detección de anomalías [8].

Las técnicas de *machine learning* tratan sobre la definición de un modelo que sea capaz de adaptarse aprendiendo a partir de ejemplos para resolver una determinada tarea usando un algoritmo de optimización. Dentro de éste ámbito se puede encontrar la inferencia gramatical, que trata sobre el aprendizaje de las gramáticas formales a partir de una serie de muestras, modelos de máxima entropía, aquellos basado en agrupación o *clustering*, como el algoritmo k-medias o las mixturas de gaussianas. En el presente tra-

bajo se van a usar las redes neuronales para resolver el cuello de botella de conocimiento mencionado en el punto 2.2.

Una red neuronal se puede definir cómo una serie de capas en las que se realizan una serie de operaciones en unas unidades de cómputo llamadas neuronas sobre los datos que se proporcionan a las capaa. Matemáticamente, una red neuronal se puede definir cómo una función F parametrizada por unos pesos Θ a la que se le proporciona una entrada x y devuelve una salida y':

$$y' = F(x; \Theta)$$

Un punto muy importante dentro de las redes neuronales es la definición de la topología del grafo que compone la conectividad de las capas. Cada capa puede tener una configuración diferente, con diferentes unidades de cómputo (las llamadas neuronas) y funciones de activación diferente.

En el siguiente punto se explicará un ejemplo de modelo de red neuronal llamado *MultiLayer Perceptron* (*MLP*). Este modelo ejemplifica de forma sencilla qué es una red neuronal, una neurona y el significado de conectividad entre capas.

2.2.1. Multi-Layer Perceptron

El *Multi Layer Perceptron* (*MLP*) [15] es uno de los modelos más básicos de red neuronal. Este modelo es un tipo de red neuronal que se caracteriza por tener una capa de entrada, una o más capas ocultas y una capa de salida. Cada una de estas capas está compuesta por neuronas que se conectan con las neuronas de la capa siguiente. Un ejemplo de esto se puede ver en la Figura 2.1.

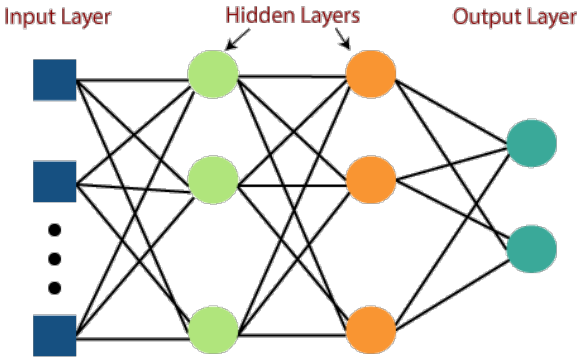

Figura 2.1: Ejemplo de *MLP*

Una neurona es un punto en la Figura 2.1 y una capa es una columna de neuronas. Una neurona es un elemento que recibe una serie de entradas, las cuales multiplica por un peso y les suma un número llamado *bias*. Esto se puede interpretar cómo una suma ponderada. Después de aplicar esta suma ponderada se aplica una función f al resultado, que suele ser una función no lineal (cómo una *sigmoid* o *relu*). La misión de la red neuronal es aprender los pesos y *bias* de cada neurona para que, cuando se le introduzcan unas entradas, la salida sea la deseada. Un ejemplo de neurona se puede ver en la Figura 2.2.

Como se puede ver la conexión de las capas definen un grafo, que en el caso del *MLP* es simplemente una secuencia de capas, aunque se pueden definir otros tipos de topologías cómo se verá más adelante.

Una vez definida la topología se procede con la fase de entrenamiento. En esta fase se introduce una serie de entradas en la red neuronal y se dice cual es la salida deseada. La

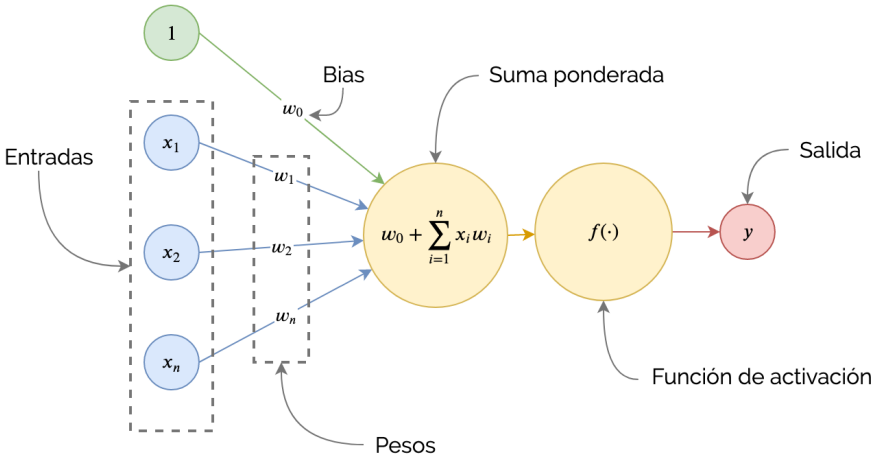

Figura 2.2: Nuerona de un MLP

red neuronal calcula la salida que obtiene y la compara con la salida deseada. A partir de esta comparación se calcula el error y se actualizan los pesos y *bias* de cada neurona. Este proceso se repite hasta que el error sea lo suficientemente pequeño o se acaben los ciclos de entrenamiento también llamados *epochs*.

En cada ciclo de entrenamiento se introducen todos los datos de entrenamiento en paquetes, llamados *batches*, en la red neuronal. Esto facilita el entrenamiento de la red neuronal ya que no es necesario cargar todos los datos a la vez en memoria, sino que se van cargando los *batches* de datos a medida que se van necesitando. Esto es especialmente útil cuando se trabaja con grandes cantidades de datos.

La forma en la que se calcula el error y se actualizan los pesos y *bias* de cada neurona es mediante una función de coste f y una función de optimización. La función de coste calcula el error entre la salida de la red neuronal y la salida deseada. Por ejemplo, una función de coste muy común es el error cuadrático medio (*MSE* por sus siglas en inglés), que se define como:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

, donde y_i es la salida deseada y $\hat{y}_i$ es la salida de la red neuronal. La función de optimización se encarga de actualizar los pesos y *bias* de cada neurona para minimizar la función de coste. Un ejemplo de función de optimización muy usado es el descenso del gradiente, que se define como:

$$w_{t+1} = w_t - \alpha\frac{\partial f}{\partial w}$$

, donde w son los pesos de la neurona, α es el ratio de aprendizaje o *learning rate* y f es la función de coste. A partir de las funciones de coste y optimización se puede usar el algoritmo de *backpropagation* para calcular los gradientes de cada neurona y actualizar los pesos y *bias* de cada neurona.

Todas las redes neuronales se basan en los mismos principios, pero cada una de ellas tiene sus propias particularidades.

2.2.2. Arquitectura *Encoder-Decoder*

En este proyecto se utilizarán redes que se basan en la arquitectura ***encoder-decoder***, razón por la cual se dedica una sección a explicar este modelo.

El *encoder-decoder* es un tipo de arquitectura que se usa en una gran cantidad de aplicaciones como codificación de imágenes (*jpg*), traducción automática, visión por computador, etc. Este tipo de arquitectura se basa en dos partes, una llamada *encoder* y otra llamada *decoder*. El *encoder* se encarga de codificar la entrada en una representación interna, mientras que el *decoder* se encarga de decodificar esta representación interna en la salida deseada. En el caso de codificación de imágenes [12], el *encoder* codifica la imagen en una representación interna, mientras que el *decoder* decodifica esta representación interna en una imagen. Un ejemplo de esto se puede ver en la Figura 2.3:

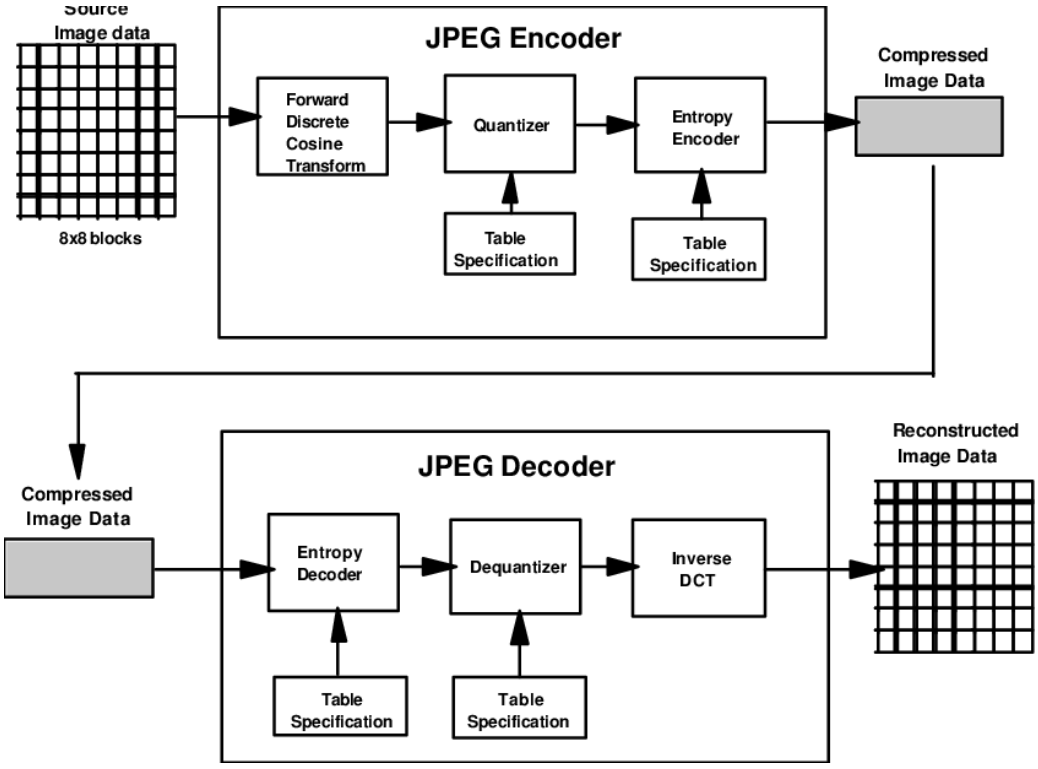

Figura 2.3: jpg como *encoder-decoder*

En el caso de las redes neuronales, el *encoder* y el *decoder* están compuestos por capas de neuronas. El *encoder* se encarga de codificar la entrada en una representación interna, mientras que el *decoder* se encarga de decodificar esta representación interna en la salida deseada. En el caso de codificación de imágenes, el *encoder* codifica la imagen en una representación interna, mientras que el *decoder* decodifica esta representación interna en una imagen. Esta representación interna se le suele llamar espacio latente, *hidden state* o *embedding*. Un ejemplo de arquitectura *encoder-decoder* se puede ver en la Figura 2.4.

Una gran cantidad de modelos se basan en esta arquitectura, como por ejemplo las redes neuronales recurrentes (*RNN* por sus siglas en inglés) aplicadas a la traducción automática o las redes neuronales convolucionales (*CNN* por sus siglas en inglés) cómo el *Variational Autoencoder* (*VAE*) o las *U-Net*.

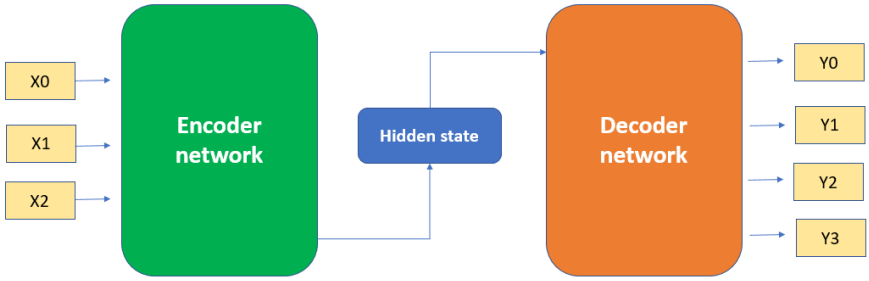

Figura 2.4: Arquitectura *encoder-decoder*

2.2.3. *U-Net*

La *U-Net* es una arquitectura de redes neuronales convolucionales que se empezó a utilizar para segmentación de imágenes biomédicas [28]. Esta arquitectura se basa en la arquitectura *encoder-decoder*, pero con una serie de modificaciones. La arquitectura clásica *U-Net* se puede ver en la Figura 2.5:

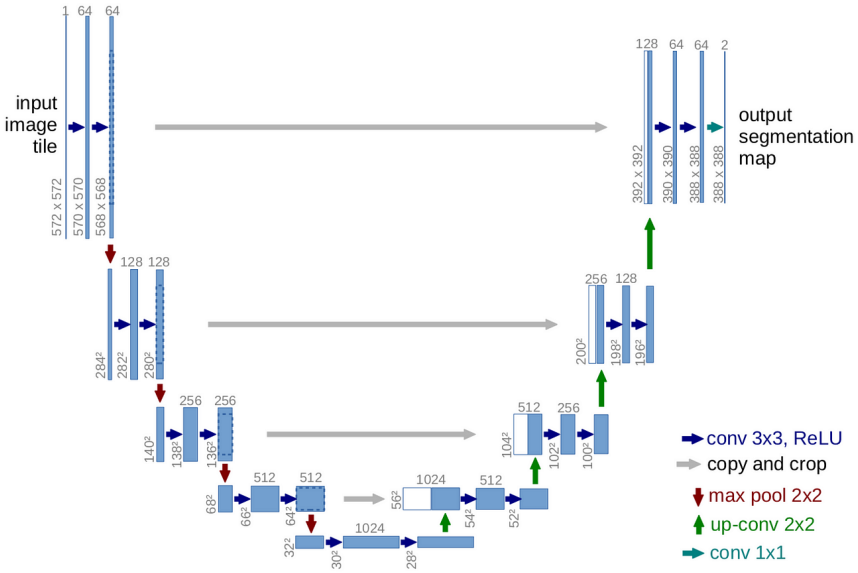

Figura 2.5: Arquitectura *U-Net*

En la Figura 2.5 se puede apreciar la parte *encoder* y *decoder* de la arquitectura, y también la peculiaridad de esta arquitectura, la llamada *skip connection*, que consiste en conectar la capa i-ésima del *encoder* con la capa n-ésima del *decoder*. En la Figura 2.5 se puede observar las fechas horizontales que conectan las capas del *encoder* con las capas del *decoder*, es decir, las *skip-connections*. Esto permite que la información de la capa i-ésima del *encoder* se pueda usar en la capa n-ésima del *decoder*. Se ha decidido usar este modelo ya que para la generación de una imagen que represente el resultado de la aplicación de una acción a un estado representado como una imagen será necesaria la **representación la-**

tente que contenga la representación característica de la entrada así cómo la **información contextual** de la imagen para poder generar el resultado y que este sea válido.

CAPÍTULO 3
Datasets utilizados

En este capítulo se presenta los *datasets* de imágenes utilizados para el desarrollo del trabajo. Particularmente, el modelo se ha aplicado en dos *datasets*: el problema del *8-puzzle* y *lights-out*.

3.1 Generación de los *datasets*

Las muestras que se utilizarán para los experimentos tienen el siguiente formato:

$$(x, y, m) : x \in D^{i \times i}, m \in \mathbb{B}^{n \times (n+1)}, y \in D^{n \times i \times i}$$

donde:

- i es el tamaño en píxeles de un lado de la imagen que representa un estado.
- D se define como el conjunto:

  $$D = \{\frac{k}{255} : 0 \leq k \leq 255\}$$

  que son los valores de la matriz que representa una imagen de un sólo canal (escala de grises). Las imágenes son matrices compuestas por valores de 0 a 255, pero necesitamos normalizar cada elemento y pasarlo a valores entre 0 y 1 (para optimizar el entrenamiento de la red neuronal), por lo que dividimos entre 255 cada elemento de la imagen. Se ha elegido esta forma de normalización en lugar de utilizar la media y la varianza porque es el mecanismo más extendido en los proyectos de Visión por Computador mediante redes neuronales.
- $\mathbb{B}$ Se define como el conjunto de valores binarios, por lo que $\mathbb{B} = \{0, 1\}$.
- x es la imagen de entrada que representa un estado del problema y se representa como una matriz de $i \times i$ cuyos valores son números reales.
- m es un tensor de valores binarios en el que se especifica el movimiento asociado a una imagen, siendo n el máximo número de movimientos que puede realizarse en el problema. Un movimiento se define cómo un vector en formato *one-hot* y se representa como un tensor unidimensional en el que cada posición indica el movimiento aplicado, siendo la última posición el indicador de movimiento no posible. Por ejemplo, si hay 4 movimientos entonces una representación de uno de ellos será $(0\,0\,0\,1\,0)$ (aquí indica que se ha realizado cuarto movimiento), donde se ve que hay

5 dimensiones, 4 que indican el movimiento realizado y una última que indica si el movimiento no es posible.

Para cada x se especifica cuantos movimientos se pueden realizar (posibles y no posibles). Este número de movimientos n será el mismo para todas las muestras. Con todo lo explicado, se puede decir que el tensor m tendrá una dimensionalidad de $n \times (n+1)$, esto es, n movimientos a aplicar en x. Cada movimiento se representa usando el tensor *one-hot* de dimensionalidad $n+1$.

Por ejemplo, para un problema donde se define $n = 2$ (como máximo se pueden realizar 2 movimientos), el tensor m tendrá una dimensionalidad de $2 \times (2+1)$. Supongamos que para una imagen determinada el primer movimiento no es posible pero el segundo sí. El tensor m tendría la siguiente forma: $m = ((0\,0\,1)(0\,1\,0))$.

- y es un tensor con n imágenes, una por cada movimiento definido en m. Cada imagen representa un nuevo estado cuyo origen es la imagen representada en x y sobre el cuál se ha aplicado una acción o movimiento especificado en el problema (ver definición de m). Como en el caso de x, cada imagen en y se compone de $i \times i$ píxeles, por lo que el tensor y tendrá la siguiente dimensionalidad: $n \times i \times i$.

Las muestras se generan de manera *online*, es decir, en cada paso de entrenamiento se generan las imágenes y los movimientos que se van a usar. El proceso consiste en generar una secuencia de movimientos aleatorios a partir de un tablero ordenado según un orden específico. Una vez generada esta secuencia, se escogen de forma aleatoria los movimientos que se van a usar para entrenar la red.

El tamaño de la secuencia de movimientos generada es el tamaño de *batch* elegido en el entrenamiento multiplicado por 4. Esto se realiza así para no restringir las imágenes con las que se va a entrenar la red al tamaño del *batch*. Por ejemplo, si se va a utilizar un *batch* de 32 elementos para entrenar la red, se generarán $32 \cdot 4 = 128$ movimientos de los cuáles se escogerán sólo 32 de manera aleatoria.

A la hora de explicar cada *dataset* se obviará en la explicación la dimensión no posible del tensor m, pero en los ejemplos si que mostrará de manera adecuada.

3.2 *Dataset* 1: *8-puzzle*

El *dataset* 1 contiene muestras del juego del *8-puzzle*. Este problema consta de un tablero dividido en nueve celdas, ocho de las cuales contienen los números del 1 al 8. En el tablero siempre hay una celda vacía (que de ahora en adelante se representará como el número 0). La casilla 0 puede intercambiar su posición con los números que tenga arriba, abajo, a la derecha o a la izquierda si la posición de las casillas respectivas en el tablero lo permite. El problema consiste en conseguir una configuración determinada del *puzzle* mediante movimientos de intercambio de la casilla 0 con la casilla a su derecha, izquierda, arriba o abajo (siempre que se satisfagan las restricciones de los límites del tablero) . Un ejemplo de una configuración final del problema se puede ver en la Figura 3.1.

El tablero se compone de una cuadrícula de 9 celdas divididas en tres filas y tres columnas. Cada número del tablero será representado cómo una imagen del *dataset MNIST* [9] (incluido el 0 como casilla vacía). El dataset *MNIST* contiene imágenes de números manuscritos y se utiliza para tareas de clasificación. Concretamente, *MNIST* contiene 60.000 imágenes de entrenamiento y 10.000 imágenes de test. Una imagen se compone de 28×28 píxeles en escala de grises y está etiquetada con el número que representa.

Como el tablero se compone de 3×3 celdas y cada celda debe contener una imagen de MNIST cuyas dimensiones son 28×28 píxeles, se puede definir que la imagen que

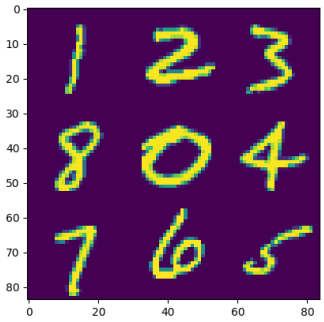

Figura 3.1: Un estado del *dataset 8-puzzle*

representa el tablero estará compuesta de 28 píxeles × 3 filas × 28 píxeles × 3 columnas = 84 × 84 píxeles.

La imagen de salida será la misma que la de entrada, pero con el 0 desplazado a la posición indicada por el movimiento. Un ejemplo de una imagen de entrada y una imagen de salida se puede ver en la Figura 3.2.

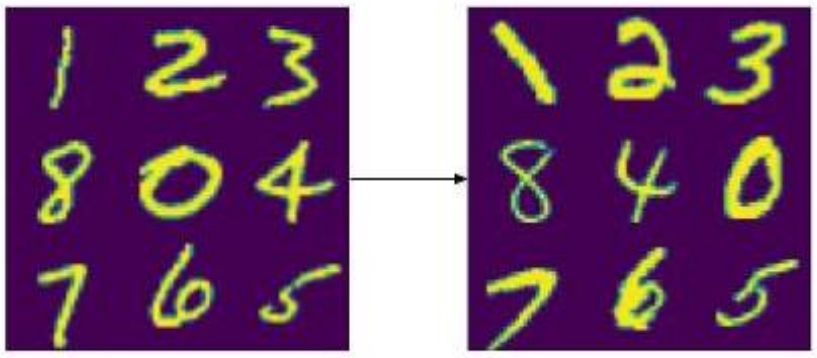

Figura 3.2: Movimiento *8-puzzle*

3.2.1. Representación de un estado

Un estado del juego del *8-puzzle* se puede representar de varias formas; por ejemplo, cómo un vector de 9 elementos:

$$v = (1\,2\,3\,8\,0\,4\,7\,6\,5)$$

o una matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{pmatrix}$$

En este proyecto se va a utilizar el vector por un mejor uso de la memoria principal e indexación.

3.2.2. Movimientos

Todo movimiento aplicable en una configuración del tablero involucra la celda vacía (el 0). Estos movimientos se definen como el intercambio de una celda adyacente al 0 con la celda del 0. Con esta información se puede observar que hay un máximo de 4 movimientos posibles (arriba, abajo, a la izquierda y a la derecha) y que varios de estos movimientos pueden no ser válidos si involucran mover el 0 fuera del tablero.

Cada movimiento válido representa una permutación entre dos elementos del vector v que representa el tablero. Un ejemplo de movimiento a la derecha se puede ver en la Figura 3.2, el cual produce una permutación entre las celdas 0 y 4 que hace que el vector:

$$v = (1\,2\,3\,8\,0\,4\,7\,6\,5)$$

pase a ser:

$$v' = (1\,2\,3\,8\,4\,0\,7\,6\,5)$$

en el que se han permutado los valores 4 y 0. Con esto se puede ver que aplicar un movimiento a la celda en la matriz A consiste en modificar la posición del 0 en el vector v. Esto se traduce en sumar o restar 1 en los movimientos hacia la derecha o izquierda, respectivamente, y sumar o restar 3 en los movimientos abajo o arriba respectivamente. El número que ocupaba el nuevo índice del 0 pasará a estar en el lugar en el que estaba el 0.

Usando el ejemplo anterior, el 0 está en la posición 5 de v. Al aplicar un movimiento a la derecha, se suma 1 a la posición del 0, lo que resulta en la posición 6. En la posición 6 de v está el número 4, que ahora estará en la posición original del 0, es decir la posición 5. El vector resultante final v' sería:

$$v' = (1\,2\,3\,8\,4\,0\,7\,6\,5)$$

Como se ha definido anteriormente, hay 4 posibles movimientos que definen una permutación entre dos elementos del vector, concretamente el 0 y un número en una casilla adyacente a él. Esto se puede expresar sumando o sustrayendo al índice en el que esté alojado el 0 una determinada cantidad:

- Mover hacia abajo: Sumar 3 al índice del 0 y su representación cómo *one-hot* será de 1000.
- Mover hacia arriba: Restar 3 al índice del 0 y su representación cómo *one-hot* será de 0100.
- Mover hacia la derecha: Sumar 1 al índice del 0 y su representación cómo *one-hot* será de 0010.
- Mover hacia la izquierda: Restar 1 al índice del 0 y su representación cómo *one-hot* será de 0001.

Cada movimiento se representará como la operación aplicada al 0 (+3 -3 +1 -1) y su codificación *one-hot* (1000 0100 0010 0001). Un ejemplo de los movimientos posibles se puede ver en la Figura 3.3.

A partir de las definiciones anteriores se puede definir cuando un movimiento es válido o no usando los límites del tablero y los números modulares:

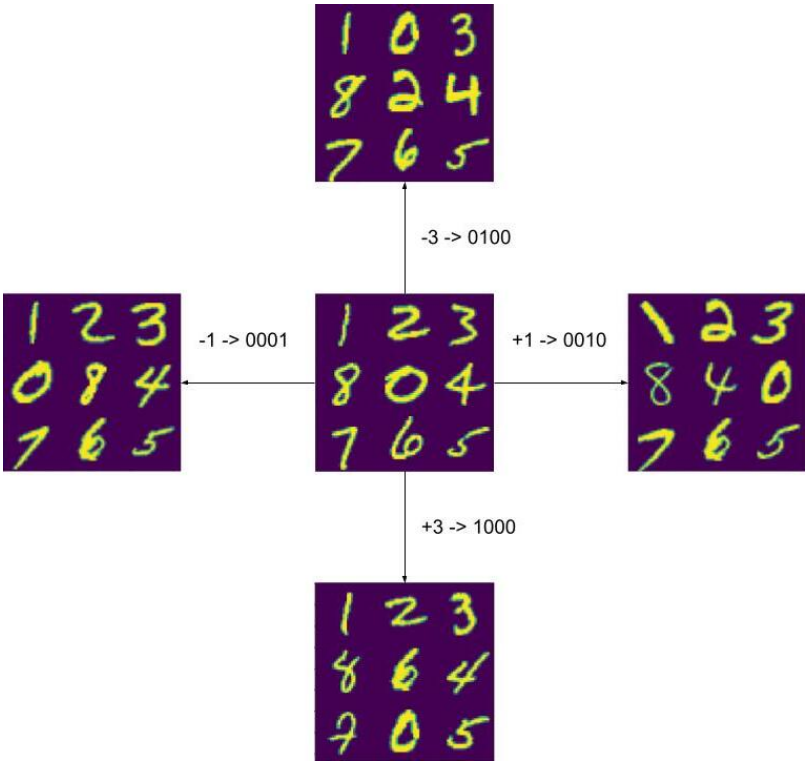

Figura 3.3: Posibles movimientos *8-puzzle*

- Si la acción de un movimiento hacia arriba da como resultado un número negativo el movimiento es inválido.
- Si la acción de un movimiento hacia abajo da como resultado un número mayor que 8 el movimiento es inválido.
- Si el resultado módulo 3 de un movimiento a izquierda es 2 el movimiento es inválido.
- Si el resultado módulo 3 de un movimiento a derecha es 0 el movimiento es inválido.

Por ejemplo, si el 0 está en el índice 2, en la esquina superior derecha del tablero, y se quiere hacer un movimiento a la derecha, este movimiento no sería válido porque se saldría del tablero. Esto significaría que el resultado de la siguiente operación sería 0:

$$(2+1) \mod 3 = 0$$

Lo que indicaría que el movimiento a la derecha es inválido.

3.2.3. Ejemplo

En la Figura 3.4 se puede ver un ejemplo del *dataset 8-puzzle* en formato imagen. En la figura hay 5 imágenes. La primera de ellas es la que aparece en la parte superior y es la que está asociada a la imagen de entrada x. Las 4 imágenes de abajo son las que están asociadas a y y en sus títulos se puede ver su codificación *one-hot* y su significado. Como el máximo número de movimientos que se puede hacer en un estado del puzzle son 4, n sería 4. Si se deseara comprobar las dimensionalidades de los tensores x, y y m en este ejemplo serian las siguientes:

- x: 84×84
- y: $4 \times 84 \times 84$
- m: 4×5

En la Figura 3.4 se puede ver que hay tres movimientos posibles: derecha, abajo y arriba. También se observa que la imagen en la que se debería encontrar el movimiento *izquierda* aparece vacía y en el movimiento codificado como *one-hot* el elemento a 1 es el último (es decir, no posible). Esta es la forma en la que se indica que un movimiento es no posible, dejando la imagen en negro (es decir, dejando todos sus valores a 0, que en la imagen viene representado como el color azul para una mejor visualización) e indicándolo en el tensor m poniendo el último elemento del tensor *one-hot* a 1.

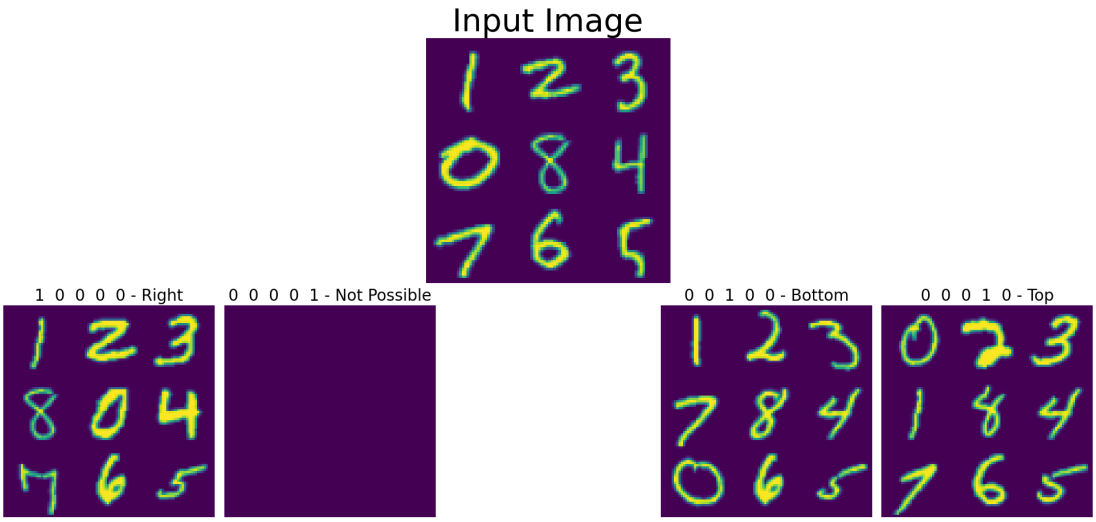

Figura 3.4: Movimientos posibles y no posibles aplicados a un estado del *dataset 8-puzzle*

3.3 *Dataset* 2: *lights-out*

El *dataset* 2 contiene muestras del juego *lights-out*. Este problema consiste en un tablero de $n = l \times l$ celdas donde cada celda representa una luz que puede estar en uno de dos estados, apagado o encendido. Una celda puede cambiar de estado apagado a encendido o de encendido a apagado. Cuando una celda cambia de estado afecta a las celdas circundantes (a las que están arriba, abajo, a la derecha, a la izquierda y en sus diagonales). El problema consiste en apagar todas las luces (celdas). Un ejemplo de un estado del problema con $l = 5$ se puede ver en la Figura 3.5, donde las celdas azules representan una luz que está apagada, las celdas amarillas una luz encendida y las partes verdes los bordes entre celdas.

El tablero se compone de una cuadrícula de n celdas divididas en l filas y l columnas, donde cada celda puede estar en estado encendido o apagado. La imagen de una celda que representa una luz encendida tiene todos sus píxeles con un valor de 1 (en la Figura 3.5 es el color amarillo). La imagen de una celda que representa una luz apagada tiene todos sus píxeles con un valor de 0 (en la Figura 3.5 es el color azul).

Una celda está formada por $t \times t$ píxeles. Cada celda tiene 4 bordes (arriba, abajo, izquierda y derecha), donde cada borde tiene una anchura de b píxeles y una longitud de t píxeles. Un ejemplo de ello se puede ver en la Figura 3.6, donde también se observa que

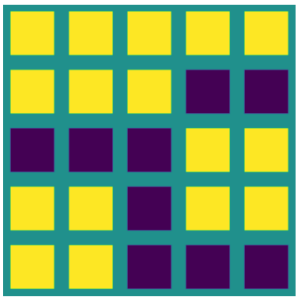

Figura 3.5: Imagen de un estado del *dataset lights-out*

las celdas no comparten bordes, es decir, cada celda tiene su propio borde. El valor de un píxel del borde es 0.5 (el color verde en las Figuras 3.5 y 3.6).

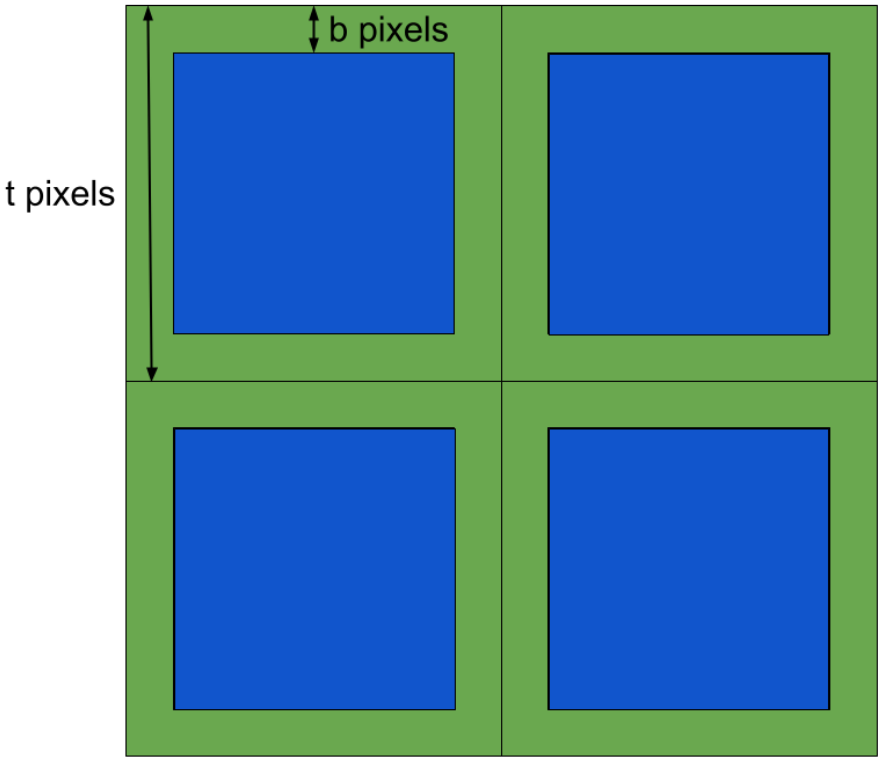

Figura 3.6: Composición una celda en base a sus píxeles

El número total de píxeles de una celda que estarán a 1 o a 0 se puede calcular a partir del cuadrado interior de la celda. Este cuadrado se puede ver en la Figura 3.6, siendo el cuadrado de color azul. Las celdas son cuadradas. Cada lado de la celda mide t píxeles, pero dentro estos t píxeles hay dos bordes de tamaño b. Descontando los dos bordes a la longitud t de la celda se obtiene el lado del cuadrado azul: $t - 2 \cdot b$. Sabiendo esto se puede calcular el área del cuadrado que representará el estado de la celda, que será de $(t - 2 \cdot b)^2$ píxeles, es decir, el total de píxeles de una celda menos los píxeles de los bordes.

El tamaño total del tablero se puede definir a partir de las celdas, por lo que si el tablero tiene $l \times l$ celdas, el tamaño total del tablero es $l \cdot t \times l \cdot t$ píxeles. En el presente

proyecto se ha usado un tablero de $l = 5$ celdas y un tamaño de celda t = 32 píxeles por lo que el tamaño total del tablero será de $l \cdot t \times l \cdot t = 160 \times 160$ píxeles.

3.3.1. Representación de un estado

Un estado del juego de *lights-out* se puede representar de varias formas; por ejemplo, cómo un vector binario (o booleano) de 9 elementos para un $l = 3$:

$$v = (0\,1\,1\,0\,0\,1\,1\,1\,0)$$

o una matriz:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Es importante recalcar que la negación de un elemento de una matriz consiste en:

$$neg(element) = \begin{cases} 1 & \text{if } element = 0 \\ 0 & \text{if } element = 1 \end{cases}$$

En este proyecto se va a utilizar el formato vector por un mejor uso de la memoria principal e indexación.

3.3.2. Movimientos

En este juego no hay movimientos no posibles por lo que no existen restricciones a la hora de realizar un movimiento. En total hay un movimiento por celda, o sea $n = l \cdot l$ movimientos. El movimiento consiste en cambiar el estado de una celda y sus celdas circundantes. Esto se puede representar como la negación de la celda y sus celdas circundantes. Por ejemplo, para el vector v anterior, si se apaga la luz de la segunda celda el resultado sería:

$$move(v, 1) = (1\,0\,0\,1\,1\,0\,1\,1\,0)$$

Para una visualización más sencilla, se muestra la misma operación en el elemento (0,1) en la matriz A:

$$move(A, 0, 1) = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Como se puede observar, la celda cambiada es la segunda (cuyo índice es 1 ya que se empieza a contar desde el 0), que cambia de estado. Este movimiento también afecta a las celdas circundantes, que también cambian de estado (las celdas $(0\,2\,3\,4\,5)$ en el vector). Una representación en formato imagen del movimiento aplicado se puede ver en la Figura 3.7.

El algoritmo que permite calcular un movimiento aplicado a un estado se compone de varios pasos. El primero de ellos es calcular los índices de las celdas colindantes a la celda de índice a (incluida ella misma):

$$col(a) = \{(a-l-1),(a-l),(a-l+1),(a-1),a,(a+1),(a+l-1),(a+l),(a+l+1)\}$$

Después se filtran aquellas celdas cuyos índices están fuera del tablero:

$$filter(a) = \{a' : 0 \leq a' < l \cdot l, a' \in col(a)\}$$

Finalmente se aplica la negación a las celdas que estén en $filter(a)$ y las demás se dejan como estaban:

$$negCell(v_u, u, a) = \begin{cases} neg(v_u) & \text{if } u \in filter(a) \\ v_u & \text{otherwise} \end{cases}$$

Siendo u el índice de un elemento dentro del vector que representa un estado y v_u el estado del elemento u en el vector v. Con las funciones anteriores se puede proceder a definir la función que aplica un movimiento a la celda con índice a en el vector v:

$$move(v, a) = (negCell(v_u, u, a) : 0 \leq u < |v|)$$

Usando el ejemplo anterior, cada paso ejecutado en el algoritmo daría los siguientes resultados:

$$v = (0\,1\,1\,0\,0\,1\,1\,1\,0)$$
$$a = 1$$
$$col(a) = (-3\ -2\ -1\,0\,1\,2\,3\,4\,5)$$
$$filter(col(a)) = (0\,1\,2\,3\,4\,5)$$

$$move(v, a) =$$
$$= (neg(0)\ neg(1)\ neg(1)\ neg(0)\ neg(0)\ neg(1)\ 1\,1\,0\,)$$
$$= (1\,0\,0\,1\,1\,0\,1\,1\,0)$$

Que en formato matriz sería:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Este movimiento se puede ver en la imagen de la Figura 3.7. Cada movimiento se representa cómo un vector *one-hot*. En total hay $n = l \cdot l$ movimientos a aplicar, por lo que cada movimiento se representará con un vector de dimensionalidad $n + 1$, siendo el último elemento el indicador de movimiento no posible aunque en este *dataset* nunca se vaya a usar ese indicador. Siguiendo con el ejemplo anterior, la representación del movimiento aplicado será:

$$g = (0\,0\,1\,0\,0\,0\,0\,0\,0\,0)$$

Aunque en este *dataset* no hayan movimientos no posibles las codificaciones *one-hot* siguen teniendo $n + 1$ elementos para mantener una consistencia para todos los *datasets*. Es por esto por lo que el último elemento del identificador en formato *one-hot* jamás estará activado.

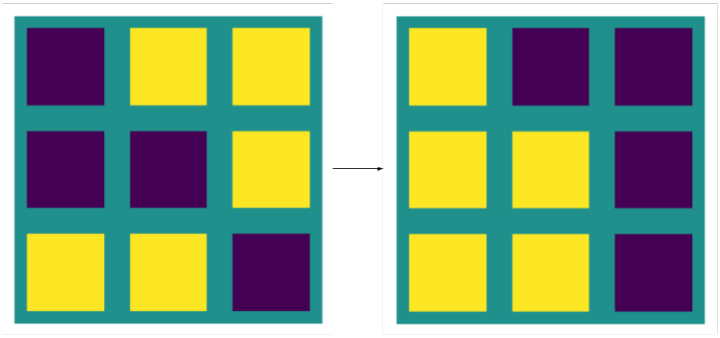

Figura 3.7: Ejemplo de movimiento del *dataset lights-out*

3.3.3. Ejemplo

En la Figura 3.8 se puede ver un ejemplo del *dataset lights-out* para un tablero de $l = 3$ y $t = 32$ píxeles, por lo que el tamaño de la imagen que representa un estado es de $32 \cdot 3 \times 32 \cdot 3 = 96 \times 96$ píxeles. En la figura hay 10 imágenes. La primera de ellas es la que aparece en la parte superior y es la que está asociada a la imagen de entrada x. Las 9 imágenes de abajo son las que están asociadas a y y en sus títulos se puede ver su codificación *one-hot* y su significado.

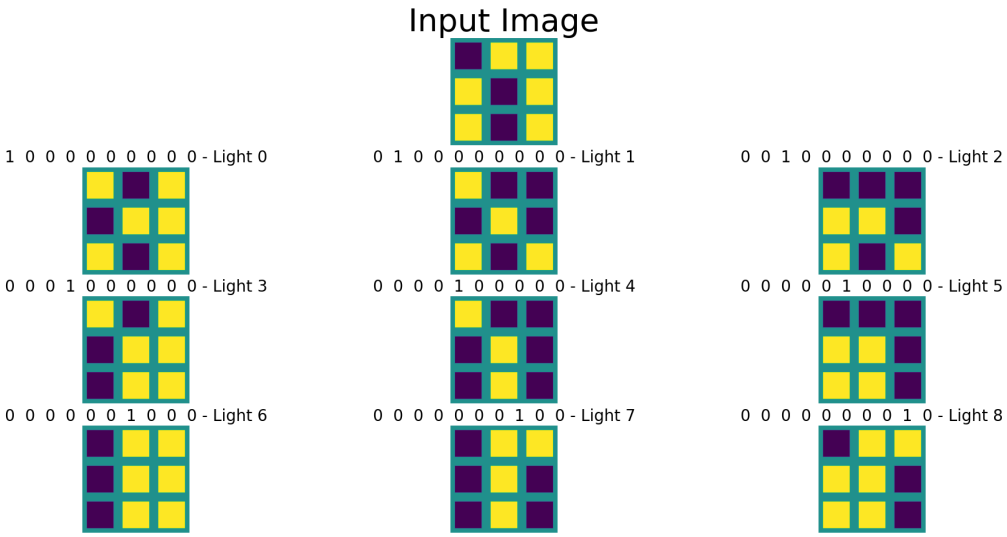

Figura 3.8: Todos los movimientos aplicables a un estado del *dataset 8-puzzle*

Como el máximo número de movimientos que se puede hacer en un estado son 9, n sería tendría nueve movimientos, siendo cada movimiento un vector *one-hot* de 10 elementos. as dimensionalidades de los tensores x, y y m en este ejemplo serian las siguientes:

- x: 96×96
- y: $9 \times 96 \times 96$
- m: 9×10

CAPÍTULO 4
Diseño y elección del modelo

En este trabajo se utilizará una arquitectura de tipo *Encoder-Decoder* pues es la más adecuada para la tarea en cuestión ya que se quiere que la red aprenda a realizar un movimiento a partir de una representación vectorial. Para ello se debe enseñar a la red a cambiar de estado a partir de una imagen de entrada. Así, con una arquitectura *Encoder-Decoder* se puede utilizar la parte de codificación para extraer características del estado que representa la imagen (es decir, proyectar a representación vectorial) y usar el *decoder* para que sea capaz de reconstruir un nuevo estado y su imagen.

Más específicamente se ha utilizado una arquitectura de tipo *U-Net* por su buen rendimiento en tareas de segmentación de imágenes médica. Una red tipo *U-Net* entiende donde está la zona de interés de la imagen y puede así aprender la zona donde se debe aplicar un movimiento. Además, las características aprendidas en el *encoder* permiten generar una representación simple (*one-hot*) de los movimientos a aplicar.

Primeramente se plantea usar una modelo basado en la arquitectura de un *Variational Autoencoder* (*VAE* por sus siglas en inglés). Después de un análisis se llega a la conclusión que otros modelos podían dar mejores resultados. Esto es así porque la arquitectura *VAE* se basa en autocodificar la entrada, es decir, que la imagen de entrada sea la misma que la de salida. En el proyecto se plantea crear un modelo que sea capaz de generar una imagen distinta de la imagen de entrada. Esta nueva imagen será el resultado de aplicar una acción al estado que esté representado en la imagen de entrada. Por esta razón se descarta utilizar arquitectura *VAE* y se decide utilizar la arquitectura *U-Net*, a la que se aplicará una serie de modificaciones para mejorar su rendimiento en la tarea en cuestión.

4.1 Definición de términos

A continuación se explican los términos que se emplearán en este capítulo ya que muchos de ellos se usan en la literatura de las redes neuronales y están en inglés:

- ***feature vector***: Un *feature vector* [1] es un tensor de $dim(z) = (z_0, ..., z_a)$ que representan una proyección de unos datos discretos en una representación continua.
- ***embedding***: Un *embedding* [22] es un *feature vector* en el que en teoría están contenidas las características más importantes de unos datos. Se puede definir cómo una proyección de los datos a un espacio latente continuo (representación vectorial) en el que los componentes del tensor contendrán aquellas características de los datos de entrada que son más representativas.
- ***fully-conected***: Este término hace referencia a una capa totalmente conectada donde todas las neuronas de entrada y salida están conectadas, como las de un *MLP* [15].

- **Convolución**: Una convolución [32] en el ámbito de las redes neuronales es una matriz de $k \times k$ elementos que se aplica a un *feature vector* para transformarlo en otro *feature vector*. Un ejemplo de ello se puede ver en la Figura 4.1.
- ***max-pool***: Una capa *max-pool* es una operación de agrupación donde se calcula el valor máximo para un conjunto de componentes de un *feature vector* y en el que devuelve sólo la parte máxima. La agrupación viene definida por aquellos componentes de un *feature vector* que estén dentro de una ventana de $q \times q$ elementos. Un ejemplo de ello se puede ver en la Figura 4.2.
- ***flatten***: Esto es una capa en el que se tiene de entrada un tensor de dimensiones $(z_0, ..., z_a)$ y se devuelve un tensor unidimensional cuyo tamaño es $\sum_{n=0}^{a} z_n$. Un ejemplo de ello se puede ver en la Figura 4.3.
- **Configuración de una red neuronal**: Cuando se habla de configuración de red neuronal se hace referencia a un conjunto de parámetros bien definidos de una red nueronal. Una diferencia entre una configuración de la red neuronal *X* y un modelo de la red neuronal *X* es que dentro del modelo se encuentran todas las posibles configuraciones de esa red neuronal. Por ejemplo, un modelo de red neuronal puede ser un *MLP*, y una configuración de un *MLP* un conjunto de hiperparámetros: 4 capas *fully connected* con 16 neuronas cada una.
- ***Batch norm***: Una capa *batch norm* es una capa que se encarga de normalizar los valores de un *feature-vector* [19].

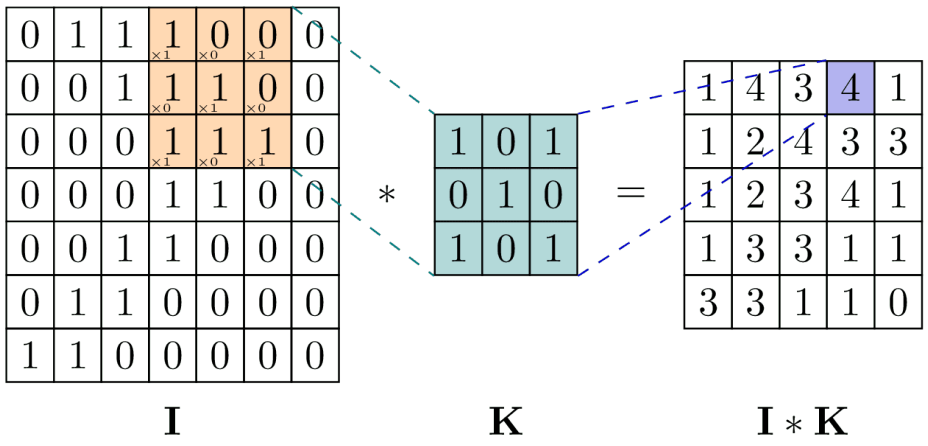

Figura 4.1: Convolución

4.2 Modelo básico de arquitectura *U-Net*

La configuración de una arquitectura tipo *U-Net* puede variar según las necesidades de la tarea. En este proyecto se decidió probar diferentes configuraciones de arquitecturas *U-Net* aunque todas ellas comparten la misma estructura básica. Además, todas las arquitecturas tienen una subred para predecir el movimiento a aplicar sobre la imagen de entrada que se conecta a la parte de codificación.

La Figura 4.4 muestra los componentes que constituyen las arquitecturas *U-Net* implementadas en este proyecto. En el diagrama se observan 4 módulos:

- ***Encoder Module***: Se encarga codificar la imagen de entrada y extraer sus características más importantes.

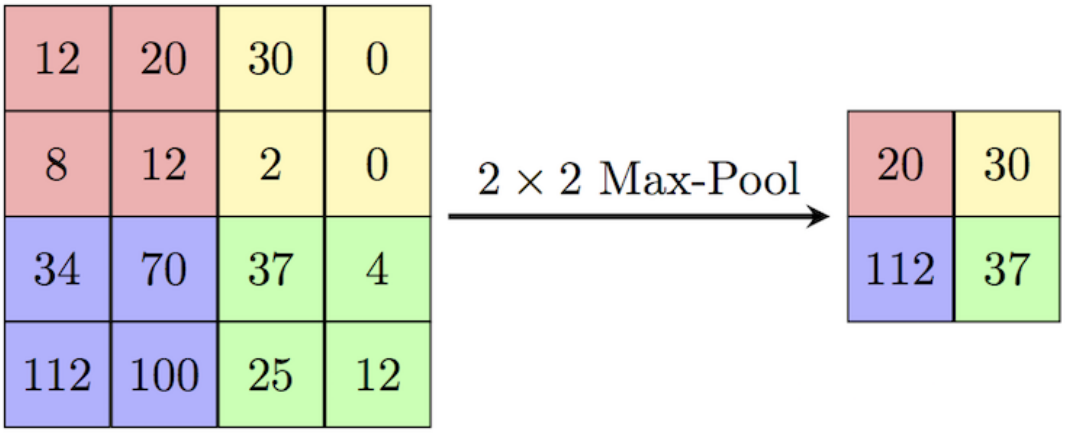

Figura 4.2: Operación *max-pool*

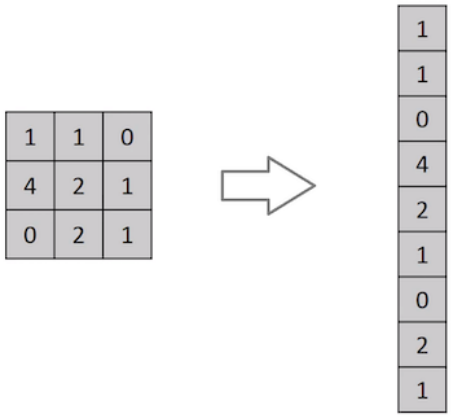

Figura 4.3: Operación *flatten*

- ***Embedding Module***: Genera el *embedding* de la imagen de entrada.
- ***Movements Module***: Predice el movimiento aplicado a la imagen de entrada.
- ***Decoder Module***: Reconstruye la imagen de entrada con un movimiento aplicado a partir del *embedding*.

Cada módulo esta formado por bloques excepto el módulo de *movements* que sólo posee capas convolucionales y una capa densa o *fully connected* [4] para predecir el movimiento a aplicar. Todos los bloques son iguales y representan una capa estilo convolución al que se le proporcionan unos filtros de entrada y unos filtros de salida. Todos los módulos poseen los mismos bloques excepto el módulo de *movements*.

Un bloque se entiende como una capa a la cual se proporciona un tensor de características y devuelve otro tensor de características. Un bloque se interpreta cómo una capa convolucional (el funcionamiento es el mismo), aunque dentro del bloque se pueden usar una o varias capas con diferentes interconexiones (cómo convoluciones [23], conexiones residuales [16], capas *inception* [29], etc.).

En definitiva, el objetivo de una capa convolucional es aplicar una convolución a un tensor de características, y la capa se define por los filtros utilizados para procesar dicho tensor. En cambio, un bloque estilo convolucional es una agrupación de capas convolucionales que pueden estar conectadas entre si de diversas formas.

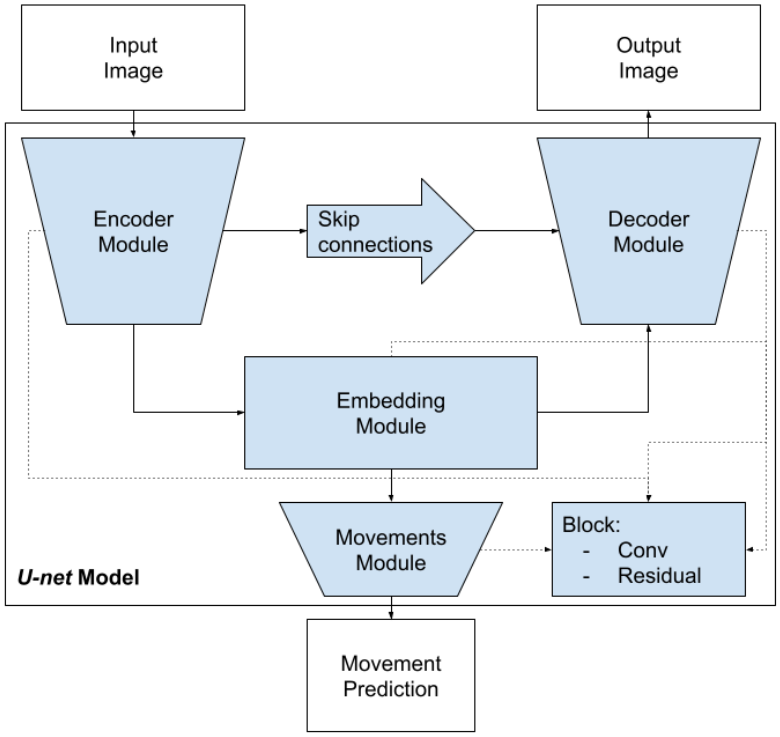

Figura 4.4: Base de la estructura de los modelos *U-Net* usados

4.2.1. *Encoder-Decoder*

El *encoder* esta formado por una serie de bloques de forma secuencial como se ve en la Figura 4.5. Estos bloques vienen definidos por filtros de entrada y salida tal y como se expone en el punto anterior. Los filtros de salida de un bloque son los filtros de entrada del siguiente bloque. El número de bloques del que se compone el *encoder* viene definido por una lista en el que se estipula los filtros de salida de cada bloque. Cada bloque tiene dos salidas, la que se proporciona al siguiente bloque y la que se usará cómo *skip connection*.

En la Figura 4.5 se observa un ejemplo de ello. Se define una lista de 3 filtros de salida $(16\ 32\ 64)$. Estos filtros son los que se usan para construir los bloques, que son los rectángulos verdes en la figura.

En el caso del *decoder* sería exactamente lo mismo pero inviritendo el orden de la lista de bloques y teniendo en cuenta las *skip-onnections* proporcionadas por el *encoder*. En el *decoder* cada bloque verde sólo tiene una salida que se conecta con el siguiente bloque, pues no es necesario usar *skip-connections*.

4.2.2. *Embedding*

Este módulo se compone de dos bloques que se utilizan para generar un *embedding* de la imagen de entrada y una convolución transpuesta que es la que proporciona la entrada del *decoder*. El módulo *embedding* será usado tanto por el *decoder* como por el móduo *movements*. Asimismo, una vez entrenada la red, el *embedding* podría también utilizarse para otro tipo de tareas en las que se necesite los *embeddings* de los estados representados en las imágenes de entrada.

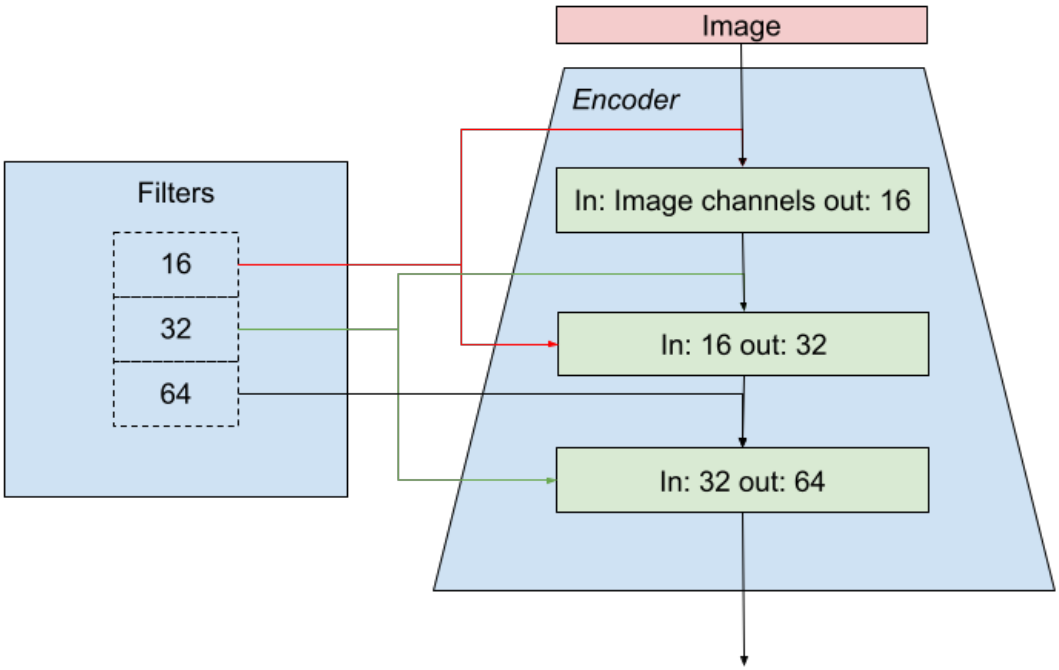

Figura 4.5: Estructura del *encoder*

Como se puede observar en la Figura 4.4, el módulo *embedding* conecta la salida del *encoder* con el *decoder*, de modo que se genera un espacio ***subsimbólico*** en el que el *embedding* representaría las características principales del estado de entrada. Esta sería una forma de realizar un *bypass* y evitar el cuello de botella del conocimiento. Es decir, se podrían utilizar los *embeddings* como entrada de otra red que se encargaría de aprender la representación *PDDL* del estado de entrada y de esta forma ahorrar el tiempo invertido en codificar cada estado del problema a una representación simbólica *PDDL*.

Aún así con este enfoque seguiría existiendo el problema del cuello de botella, pues habría que definir los predicados y tener muestras etiquetadas para entrenar la segunda red, pero se ahorraría mucho tiempo en la codificación de nuevos estados. En el capítulo final se mostrarán algunos ejemplos para poder evitar el cuello de botella usando una aproximación basada en el presente proyecto.

4.2.3. *Movements*

El módulo que se usa para la predicción del movimiento se basa en el mismo proceso que el *encoder*, pero en lugar de bloques tiene capas convolucionales. En las últimas capas de este módulo se realiza la operación *flatten* y se añade una capa *fully-connected* con una activación *softmax* para predecir el movimento elegido por la red.

4.3 Entrenamiento

Como se ha comentado en puntos anteriores, las muestras etiquetadas del *dataset* proporcionan los movimientos que se pueden aplicar a un estado (posibles y no posibles si los hay). En el entrenamiento sólo se puede elegir un movimiento para realizar el *back-propagation*, por lo que es necesario definir cómo se va a elegir el movimiento a realizar a partir de las posibilidades existentes. Un primer filtro aplicado a las muestras del *dataset*

es elegir solamente los movimientos válidos, de esta forma se previene enseñar al modelo movimientos que no son realizables.

Seguidamente se pensó en elegir de forma aleatoria el movimiento a aplicar, pero después de varias pruebas se observó que distintas configuraciones de redes no eran capaces de aprender a generar movimientos ni a predecir un movimiento válido. Después de analizar porque pasaba esto se llegó a la conclusión que era porque no había un patrón definido detrás de la elección de cada movimiento, sino que la distribución que generaba los movimientos era una distribución uniforme.

Esto explica por qué la red no era capaz de aplicar un movimiento a partir de una entrada, ya que no había un patrón que pudiera explicar cuál era el movimiento aplicado. Con esto en mente se procedió a cambiar la forma en la que se elegía el movimiento a aplicar. Este nuevo método consiste en elegir el movimiento cuya función de pérdida es menor:

$$movementToBackprop = argmin_j(\{loss(y_j, y', m_j, m') : 0 <= j < n\})$$

siendo $loss(y_j, y', m_j, m')$ una función que suma los resultados de las funciones de pérdida asociadas al *decoder* y el *movements*:

$$loss(y_j, y', m_j, m') = decoderLoss(y_j, y') + movementsLoss(m_j, m')$$

Y siendo (y_j, m_j) la imagen y la identificación en formato *one-hot* del j-ésimo movimiento que se puede realizar (posible o no posible) respectivamente y (y', m') las predicciones de la red neuronal del movimiento a aplicar:

$$y_j \in y, m_j \in m, (y', m') = UNet(x), n = |m|$$

La función de pérdida elegida para el *decoder* es la conocida como error de mínimos cuadrados (MSE por sus siglas en inglés) [2] y para el *movements* es la conocida cómo entropía categórica (*cross entropy* en inglés) [14]. Explicar en que consiste cada una se sale del alcance del presente proyecto por lo que se recomienda visitar las citaciones para una mejor comprensión de ellas.

Con esta forma de seleccionar la muestra con la que se realiza el *backpropagation*, la red no sólo aprende a realizar movimientos, sino que es capaz de establecer un orden de prioridad de movimientos. Esto se verá en el punto 4.5.

Para la elección del mejor modelo y configuración de arquitectura U-Net para los problemas presentados en el capítulo anterior, se realizaron una serie de experimentos, cuyos resultados se muestran en detalle en las siguientes secciones. Se probaron dos modelos U-Net, convolucional (sección 4.4) y residual (sección 4.5) y se eligieron aleatoriamente 30 configuraciones diferentes para los dos modelos.

Se ejecutaron 10 *epochs* en cada configuración para visualizar la evolución de ciertas métricas y estimar la configuración con más posibilidades de obtener mejores resultados. Se utilizaron varias técnicas de regularización, cómo la anteriormente mencionada *dropout*, *gradient clipping* [26], *weight decay* [33], *learning rate decay* [35] y *data augmentation* (usando la librería de *Python torchvision*) que es aplicar transformaciones aleatorias a las imágenes de entrada de la red neuronal. El optimizador usado en el proceso de entrenamiento es *AdamW* [21].

La experimentación para encontrar la mejor arquitectura se realizó utilizando el *dataset 8-puzzle*. El *dataset lights-out* se usará para mostrar que la red elegida es realmente una buena opción para el problema en cuestión.

Para el entrenamiento y las experimentaciones se utilizaron las siguientes librerías de *Python*:

- *Pytorch* [27]: Para la creación de la red neuronal y la generación de los tensores usados por el entrenamiento en base a imágenes.
- *Pytorch Lightning* [10]: Para el entrenamiento de las redes neuronales.
- *Wandb* [5]: Para la visualización de métricas y la comparación de modelos entrenados.
- *Ray-Tune* [24]: Para la selección de los hiperparámetros de cada red.

4.4 *U-Net* - Convolucionales

El primer modelo de *U-Net* de la experimentación es una red que contiene bloques convolucionales. Un bloque convolucional se compone de 3 elementos básicos:

- Una capa convolucional llamada *input* que transforma el *feature vector* de entrada con unos filtros de entrada w a otro *feature vector* con otros filtros de salida r.
- Una capa convolucional llamada *mid* cuyos filtros de entrada y salida son los mismos que la salida de la capa *input*, es decir, r filtros de entrada y r filtros de salida.
- Una capa *max-pool* en el caso del *encoder* y una convolución transpuesta en el caso del *decoder* y del *embedding*

Y cada capa convolucional se compone de tres sub-capas:

- Una capa convolucional simple.
- Una capa *batch norm* [19].
- Una capa de activación *Gelu* [17].

A continuación se muestran una serie de figuras con los resultados de los experimentos realizados con este modelo de *U-Net*. Los resultados se han obtenido del entrenamiento de 30 configuraciones con diferentes hiperparámetros (*learning rate*, *weight decay*, bloques de filtros de *encoder-decoder*, bloques de filtros de *movements module*). Las figuras muestran la media (como una linea azul) y la desviación estándar (como un área de un azul más claro que la media) de diferentes métricas respecto a los resultados obtenidos en el eje Y. En el eje X se muestran el avance del entrenamiento. Esto se representa mediante *pasos*. Cada 100 pasos se completa un *epoch*, por lo que en el paso 100 se habrán completado un *epoch*, en el 500 5 *epochs* y en el 1000 los 10 *epochs*.

No se explicará en detalle todas las configuración probadas puesto que el objetivo es investigar el comportamiento general del modelo en base a las 30 configuraciones. En el punto 4.6 se mostrará cuál es la configuración elegida así como los hiperparámetros utilizados en el entrenamiento.

La Figura 4.6 muestra la media de la precisión obtenida y su desviación estándar respecto a los movimientos elegidos por la red y los movimientos del *dataset*. Como se observa, la precisión nunca sube más allá de un 30 % y suele rondar el 25 % de precisión.

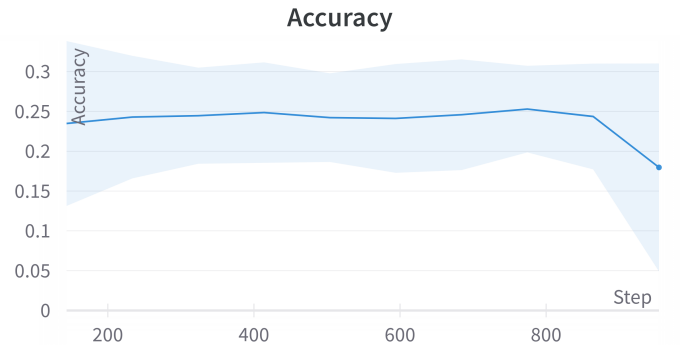

Figura 4.6: Media y desviación estándar de la precisión de los entrenamientos en la *U-Net*

Esto es así porque la red tiende a etiquetar todo cómo un mismo movimiento, por lo que cuando el movimiento no sea válido la red se estará equivocando.

El mismo problema se observa en la Figura 4.7 que muestra la media y la desviación estándar de los resultados de la función de pérdida $movementsLoss(m_j, m'_j)$. Se puede ver cómo la pérdida tiende a quedarse estática alrededor de 0.19.

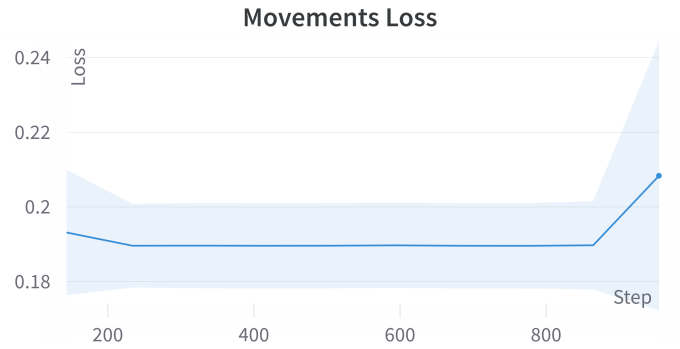

Figura 4.7: Media y desviación estándar de la pérdida del *movements* de los entrenamientos en la *U-Net*

En la Figura 4.8 se muestra la media y la desviación estándar de la función de pérdida $decoderLoss(y_j, y'_j)$, que es la que se encarga de estimar la similitud entre la salida del *decoder* (que es una imagen) y la imagen que representa un estado con un movimiento aplicado. Aquí se puede ver que el modelo aprende a generar las imágenes de forma correcta (pues la función de pérdida tiende a bajar) según el criterio establecido en el punto 4.3 de usar sólo las muestras que proporcionen mínima pérdida para realizar el algoritmo de *backprop*.

4.5 *U-Net+* - Residuales

El siguiente modelo de *U-Net* con la que se ha experimentado es una red que contiene bloques residuales. Un bloque residual se compone de tres elementos básicos:

- Una capa residual llamada *residual-input* que transforma el *feature vector* de entrada con unos filtros de entrada w a otro *feature vector* con otros filtros de salida r.

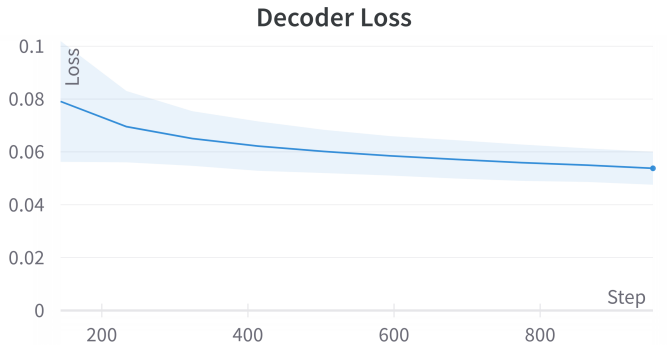

Figura 4.8: Media y desviación estándar de la pérdida del *decoder* de los entrenamientos en la *U-Net*

- Una capa residual llamada *residual-mid* cuyos filtros de entrada y salida son los mismos que la salida de la capa *residual-input*, es decir, r filtros de entrada y r filtros de salida.
- Un *max-pool* en el caso del *encoder* y una convolución transpuesta en el caso del *decoder* y del *embedding*

Y cada capa residual se compone de 4 sub-capas:

- Una capa convolucional denominada *input* con w filtros de entrada y r de salida.
- Una segunda capa convolucional denominada *mid* con r filtros de entrada y salida.
- Una capa convolucional denominada *auxiliar* que transforma el *feature vector* de entrada del bloque con w filtros a otro *feature vector* con los mismos filtros r que la capa convolucional *mid* si $w \neq r$. Si $w = r$ esta capa sólo aplica la función identidad.
- Una capa que suma las salidas de *mid* y *auxiliar*.

Las capas convolucionales mencionadas son las mismas capas que las definidas en el punto anterior. Las figuras que se muestran a continuación son las mismas que en el punto 4.4 pero usando el modelo *U-Net+*.

A continuación se muestran una serie de figuras con los resultados de los experimentos realizados con este modelo de *U-Net+*. Los resultados se han obtenido del entrenamiento de 30 configuraciones con diferentes hiperparámetros (*learning rate*, *weight decay*, bloques de filtros de *encoder-decoder*, bloques de filtros de *movements module*). El eje X de las figuras que se van a mostrar representan lo mismo que las figuras mostradas en el punto 4.4.

En la Figura 4.9 se representa la media de la precisión obtenida y su desviación estándar respecto a los movimientos elegidos por la red y los movimientos del *dataset*. En la figura donde se observa un aumento importante respecto a la mostrada en la sección anterior. Se pasa de tener una media de un 25 % a una media de un 75 %.

No obstante, en la Figura 4.9 se ve que hay mucha más desviación estándar. Esta variación se observa en el área azul de la figura, ya que el máximo punto del área azul esta más distante que el mínimo punto que en la Figura 4.6, es decir, el máximo valor del área azul en el paso 800 de la figura es de 1 (que representa un 100 % de precisión) y el mínimo es de 0.6 (que representa un 60 % de precisión). Esto se debe a que en algunas

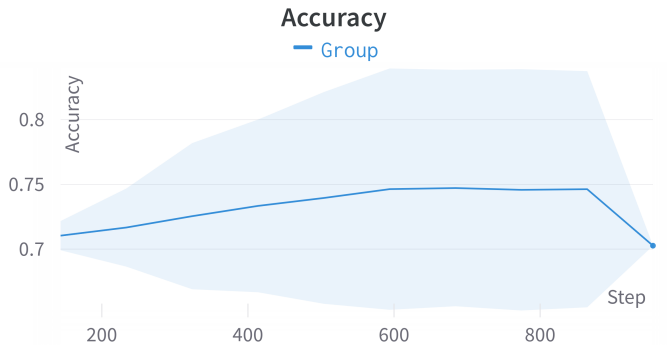

Figura 4.9: Media y desviación estándar de la precisión de los entrenamientos en la *U-Net+*

configuraciones se elegía siempre el mismo movimiento y en otras se elegían de manera correcta los movimientos atendiendo a cuando era posible aplicar un movimiento o no.

Por las restricciones en el problema del *8-puzzle*, aplicar el mismo movimiento no siempre es posible. Por ejemplo, no se puede elegir siempre el movimiento de derecha, porque en las casillas del borde derecho del tablero no se puede aplicar el movimiento derecha. Esto mismo es aplicable para los demás movimientos, si el 0 está en el borde superior del tablero no se puede realizar el movimiento hacia arriba, lo mismo hacia abajo con las casillas del borde inferior y lo mismo para el movimiento izquierda.

Siguiendo con ese ejemplo, el movimiento a la derecha no es aplicable si el 0 está en las casillas que estén en el borde derecho. Cómo en el borde derecho hay 3 casillas, esto se traduce a que el movimiento no se podrá realizar en el $\frac{3}{9}$ % de los casos, o dicho de otra forma, ese movimiento se podrá realizar sólo en las otras 6 casillas, es decir, en el $\frac{6}{9}$ % de las veces. Esto explica que la desviación estándar que está debajo de la media en la Figura 4.9 ronde el 60 %, porque un movimiento (por ejemplo derecha) será aplicable en 6 de cada 9 casillas. Teniendo en cuenta que entre las 30 configuraciones entrenadas muchas siempre elegían el mismo movimiento (derecha por ejemplo) y que la imagen que representa un estado al que se le quiere aplicar un movimiento se genera de manera aleatoria, implica que la probabilidad de que el 0 esté en una casilla en la que se puede mover (por ejemplo a la derecha) será de un $\frac{6}{9}$ % = 66,66 %.

Aunque esto ocurría en la mayoría de las 30 configuraciones entrenadas, algunas configuraciones específicas conseguían aprender que había movimientos no válidos, por lo que elegían movimientos dependiendo de si se podían aplicar o no. Es por esto que la desviación estándar que está por encima de la media llega al 100 %, porque en varias ocasiones la red elegía los movimientos de manera correcta.

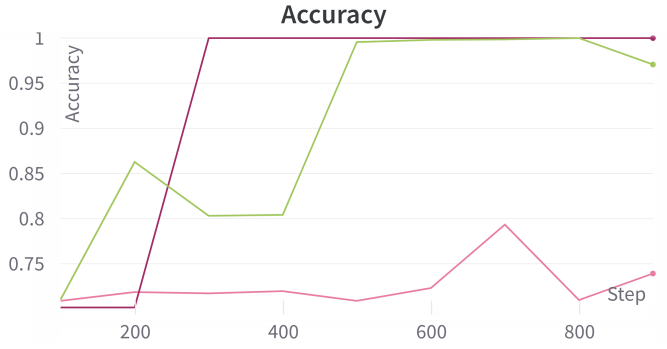

Figura 4.10: La precisión de las tres mejores configuraciones de la *U-Net+*

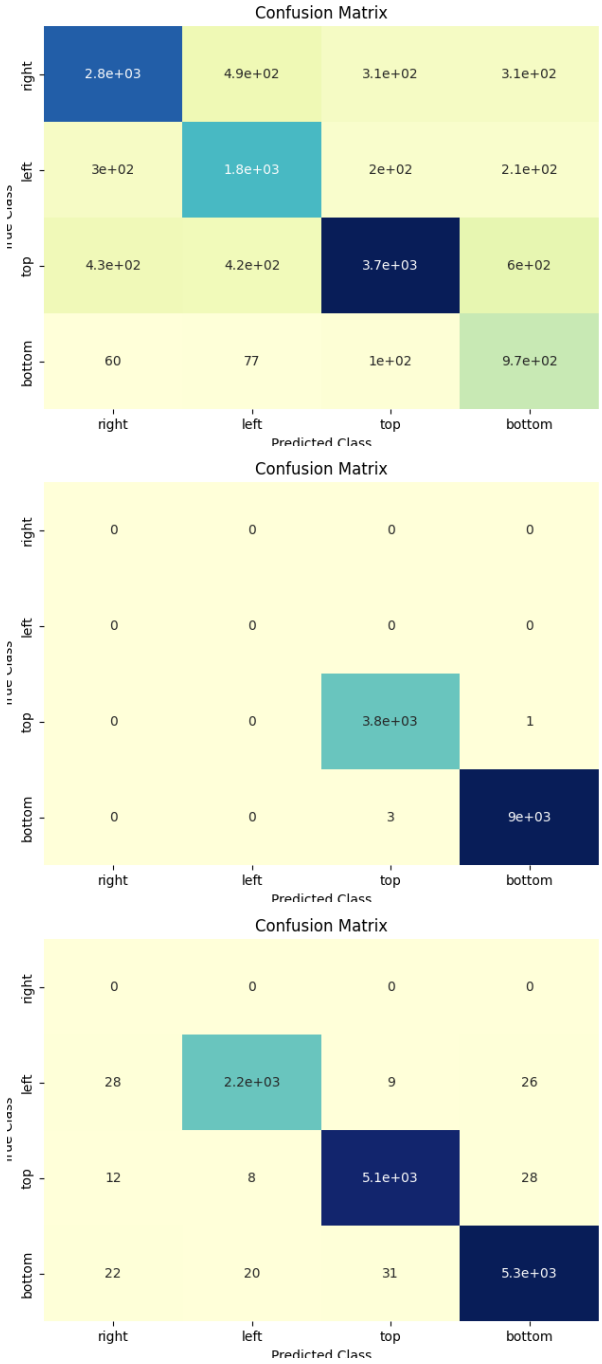

Figura 4.11: Las matrices de confusión de las tres mejores configuraciones de la *U-Net+*

En la Figura 4.10 se muestra la precisión obtenida respecto a los movimientos elegidos por la red y los movimientos del *dataset* de las 3 configuraciones con mayor precisión y en la Figura 4.11 se muestran sus matrices de confusión. En la Figura 4.11 se observa cómo la red tiende establecer un 'orden de prioridad' entre movimientos. En la figura, cuanto más oscura es una celda de la matriz de confusión más probabilidad hay de que ese movimiento sea el elegido, por lo que el 'orden de prioridad' se define a partir de estos colores. Cuanto mas oscuro es un movimiento en la matriz de confusión más prioridad tiene. Aunque se hable de prioridad realmente el color define la probabilidad de que un movimiento sea elegido, pero se ha decidido nombrarlo 'orden de prioridad' para poder explicarlo de forma más sencilla.

En la primera imagen de la Figura 4.11 se puede ver cómo el movimiento más probable es el tercero (arriba). Como este movimiento no se puede realizar siempre, el siguiente con más prioridad tiende a ser a el primero (a la derecha). Si ninguno de los dos movimientos es posible, entonces tiende a elegir el segundo (hacia la izquierda) con más probabilidad. Un razonamiento parecido es aplicable también a la tercera imagen de la Figura 4.11, pero la diferencia es que parece que la red ha aprendido que con tres movimientos tiene suficiente, no necesita aplicar los 4 para asegurar que siempre sea posible realizar un movimiento.

En cambio, en la segunda imagen de la Figura 4.11 ocurre algo curioso. El movimiento con más probabilidad es abajo y el segundo con más probabilidad es arriba. En cambio los demás movimientos aparecen amarillos, lo que indica que no se van a a elegir. Aprendiendo a mover hacia abajo y hacia arriba a la red no le hace falta aprender nada más, pues si no se puede realizar un movimiento hacia arriba se puede hacer un movimiento hacia abajo, por lo que no es necesario aprender a mover a izquierda y derecha, que es lo que indica la matriz de confusión de la segunda imagen de la Figura 4.11.

Con esto se observa que la red es capaz de aprender lo que es un estado y de seleccionar otro estado con un movimiento aplicado al estado inicial y que ese movimiento sea válido. En la Figura 4.12 se muestra la función de pérdida $movementsLoss(m_j, m'_j)$. Ahí se observa que la pérdida tiende a bajar, por lo se puede asumir que con suficientes *epochs* las imágenes generadas por el *decoder* serán las correctas.

Figura 4.12: La pérdida del *movements* de las tres mejores configuraciones de la *U-Net+*

La Figura 4.13 muestra la función de pérdida $decoderLoss(y_j, y'_j)$. Ahí se puede ver que la tendencia respecto a la generación de imágenes es parecida a la tendencia de la Figura 4.8, pero estos valores son incluso mejores. Con todo lo anterior se puede proceder a establecer el modelo *U-Net+* como el modelo elegido para resolver los problemas del presente proyecto.

4.6 Configuración elegida

En base a los resultados del punto 4.5 se procedió a seleccionar el mejor modelo para generar movimientos en el *dataset* del *8-puzzle*. La elección de la configuración se realizó en base a las 3 mejores configuraciones mostradas en el punto 4.5. Las precisiones de estas 3 configuraciones son las mostradas en la Figura 4.10. Al intentar probar varias ejecuciones con estas configuraciones, se vio que la elección de los movimientos así como la capacidad de llegar a un orden de prioridad correcto y no etiquetar todo como un sólo movimiento era muy dependiente de la inicialización de la red. Después de varios intentos se observó una tendencia entre configuraciones:

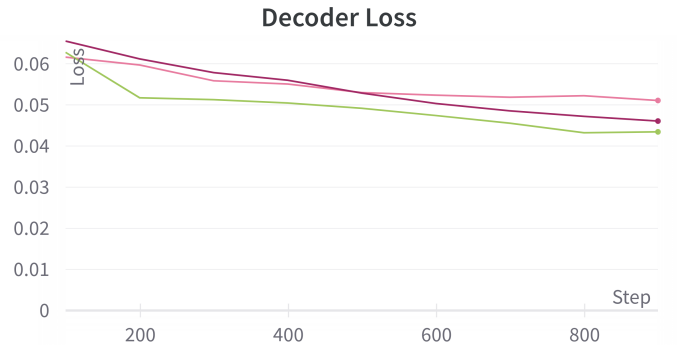

Figura 4.13: La pérdida del *decoder* de las tres mejores configuraciones de la *U-Net+*

- La lista de filtros del *encoder-decoder* debía ser de cuatro elementos, más específicamente de $[8, 16, 32, 64]$ filtros.
- La lista de filtros del *movements* sólo debía ser de un elemento, pues sino la red tendía a sobreaprender. Los filtros elegidos fueron 16.
- El *learning rate* debía estar en el intervalo de $[0{,}0004 - 0{,}0006]$.
- El *weight decay* no tenía nignún impacto en las configuraciones probadas.
- El mejor tamaño de *batch* debía ser de 64 elementos.

Con todo esto en cuenta, se procedió a entrenar la red que aprendería a generar movimientos para los dos *datasets* usados en el presente proyecto. Esta red tendría la siguiente configuración:

- Lista de filtros del *encoder-decoder*: $[8, 16, 32, 64]$.
- Lista de filtros del *movements* $[16]$.
- *learning rate*: 0,0005779322403055052.
- El *weight decay*: 0,0000003215826495926.
- Tamaño de *batch*: 64.

En la Figura 4.14 se pueden ver la precisión y las pérdidas del *movements* y el *decoder* respectivamente a lo largo de 100 *epochs*. Viendo la segunda imagen se observa que al inicio del entrenamiento, la pérdida de *movements* tendía a permanecer estática, pero que al cabo de varios *epochs* esta pérdida bajó drásticamente, lo que tiene relación directa con el crecimiento drástico de la precisión en la primera imagen.

Esto se debe a que durante los primeros *epochs* la red tiende a etiquetar siempre con el mismo movimiento pero tras varios *epochs* la red aprende que esto no debe ser así (gracias a un movimiento afortunado en el gradiente) y crea el orden de prioridad que se puede ver en la Figura 4.15. Ahí se ve que el primer movimiento elegido tiende a ser hacia la derecha, después hacia arriba y finalmente hacía abajo.

Esta configuración es la que se ha usado para entrenar las redes usadas en los *datasets* de *8-puzzle* y *lights-out*.

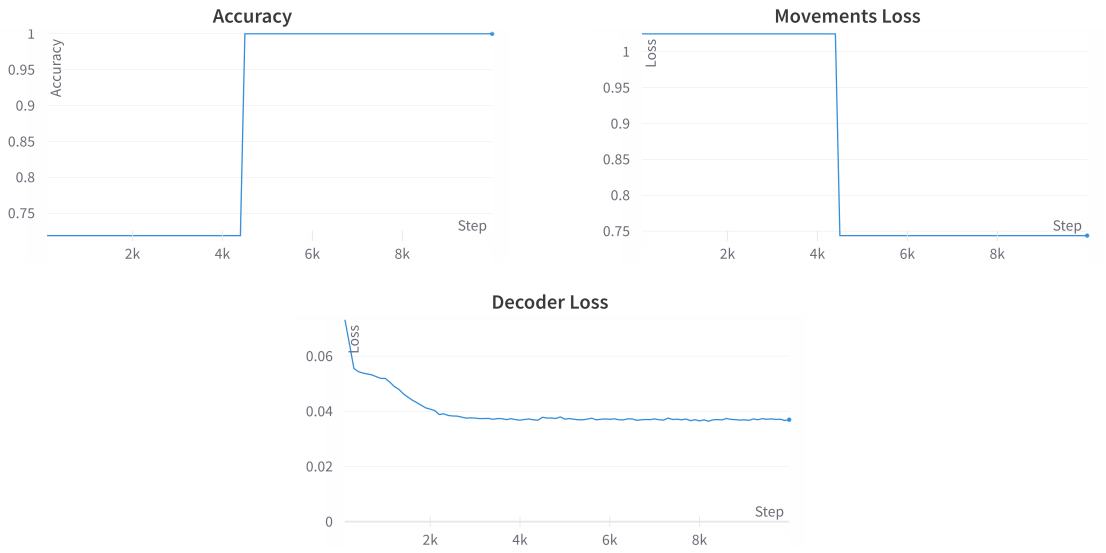

Figura 4.14: La evolución de la precisión, las pérdidas del *movements* y del *decoder* de la mejor configuración encontrada

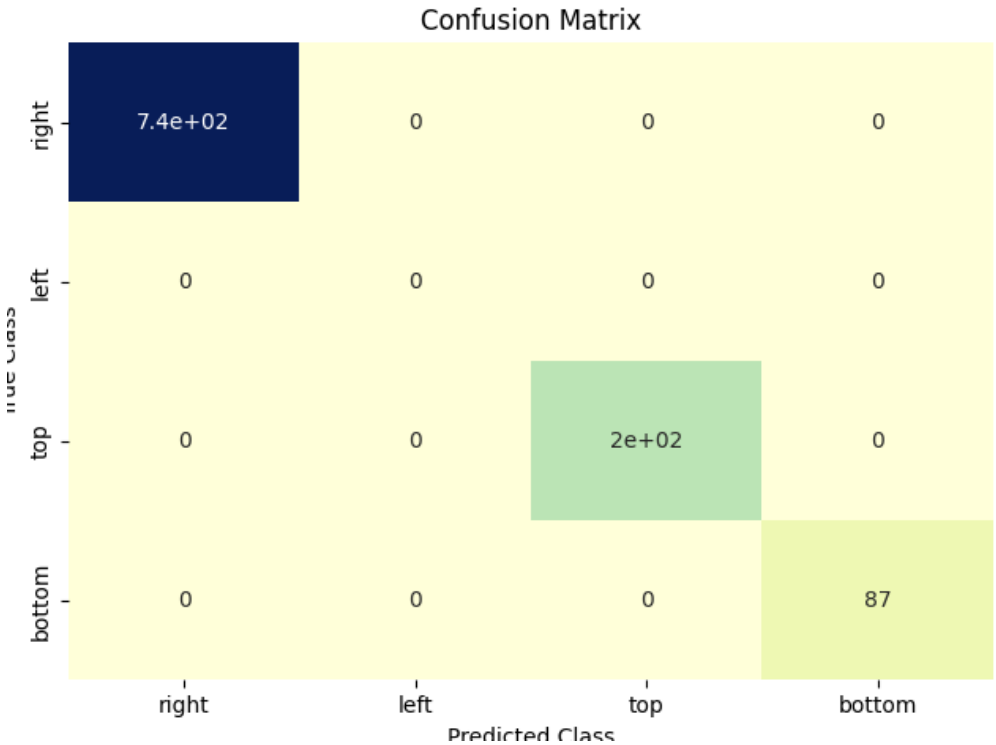

Figura 4.15: Matriz de confusión de la mejor configuración encontrada

CAPÍTULO 5
Resultados

A continuación se van a mostrar los resultados obtenidos con la red entrenada que se ha explicado en el punto 4.6. El modelo va a procesar una serie de imágenes y devolverá que movimiento se le puede aplicar (la salida del módulo *movements*) y también devolvera una imagen que represente la imagen de entrada de la red con un movimiento aplicado (la salida del módulo *decoder*).

Más adelante se entrenara la misma red usando el *dataset lights out*, sobre el que se realizarán las mismas pruebas. Con esto se comprobará si la configuración elegida es adecuada y proporciona buenos resultados.

5.1 Resultados *8-puzzle*

A continuación se mostrarán varias imágenes de entrada de la red junto con las imágenes generadas a partir de las entradas. También se indicará el movimiento elegido por la red para esas imágenes.

En la Figura 5.1 se muestran las predicciones realizadas por el modelo entrenado. Las entradas del modelo se pueden ver la izquierda de la figura, las imágenes generadas por la red a la derecha y los movimientos predichos por la red en las flechas que unen las figuras de la izquierda con las figuras de la derecha.

Cómo se puede observar las imágenes generadas aplican un movimiento correctamente. Además, ese movimiento está etiquetado de manera correcta (no hay discordancia entre la imagen generada y el movimiento predicho). Con esto se puede confirmar que el *embedding* generado contiene la información necesaria para codificar el estado de entrada y que a partir de esta información se puede generar otras representaciones, ya sea a imágenes con movimientos aplicados o a vectores *one-hot* que indiquen que movimiento se puede aplicar.

Viendo los movimientos aplicados, se ejemplifica el *orden de prioridad* establecido por la red neuronal. En las imágenes de la Figura 5.1 en las que mover a la derecha es posible se mueve a la derecha. En cambio, en la tercera imagen de la Figura 5.1 el movimiento a la derecha no es posible (ni tampoco el movimiento hacía abajo), por lo que la red decide mover hacia arriba. Esto concuerda con el orden de prioridad mostrado en el punto 4.6 en la Figura 4.15.

En algunos casos el modelo entra en bucle, es decir, para una entrada dice que hay que mover hacia arriba y después, si enviamos esta salida otra vez como entrada, dice que hay que mover hacia abajo. Con esto se entra en un bucle que, aunque se apliquen movimientos válidos, no tiene mucho sentido de cara a la planificación.

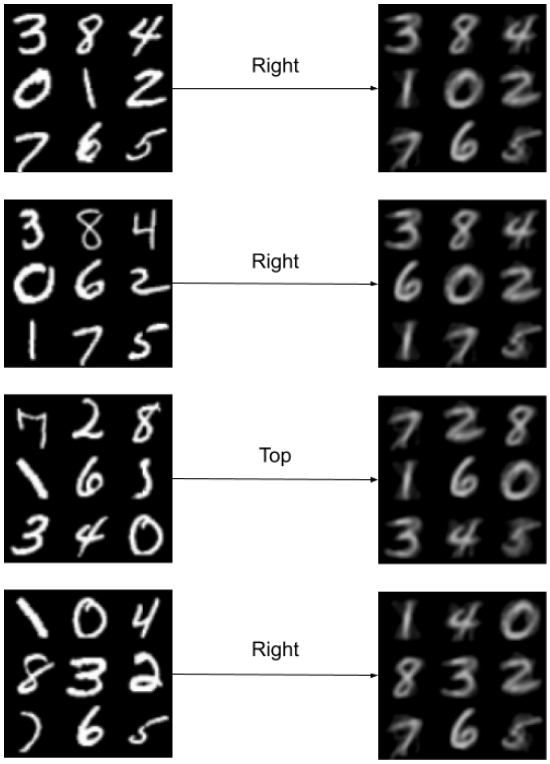

Figura 5.1: Entrada y salidas del modelo entrenado para generar movimientos del *dataset 8-puzzle*

Respecto a los números del tablero generados por la red en cada imagen de salida, en la Figura 5.1 se observa que cada entrada tiene los números dibujados de manera diferente. Es decir, el número 7 es diferente en cada imagen de la Figura 5.1, así como el número 3 (y los demás). Sin embargo, en las imágenes de salida generadas por el modelo cada número tiene la misma representación en todas las imágenes. Por ejemplo, se puede ver que el número 7 tiene la misma forma en todas las imágenes de la derecha en la Figura 5.1, al igual que con el 8 y los demás números.

Esto puede dar lugar a pensar que el *embedding* está aprendiendo a codificar cada número de manera similar, aunque el número pueda estar escrito de diferentes formas. De este modo, el *decoder* puede extraer dicha información y asignar una representación similar a cada número. Cómo las representaciones de cada número serán parecidas o iguales sin importar como este dibujado en la imagen de entrada, el *decoder* siempre devolverá la misma representación de ese número, que es lo que se observa en la Figura 5.1.

En algunos casos el *decoder* no fue capaz de generar una imagen correctamente y solo generaba una imagen en negro. En cambio, el módulo de *movements* si que seleccionaba un movimiento correcto. Esto ocurría el 20 % de las veces, por lo que aunque la red funciona bien se podrían seguir realizando mejoras en el *encoder-decoder* para reducir esa probabilidad.

Finalmente se van a mostrar 15 pares de imágenes en la Figura 5.2. Cada par contiene una imagen proporcionada al modelo y una imagen generada por el modelo a partir de la imagen de entrada. En la Figura 5.2 no sólo se muestran imágenes correctas, sino que también se muestran ejemplos de cuando una imagen no se genera de forma correcta.

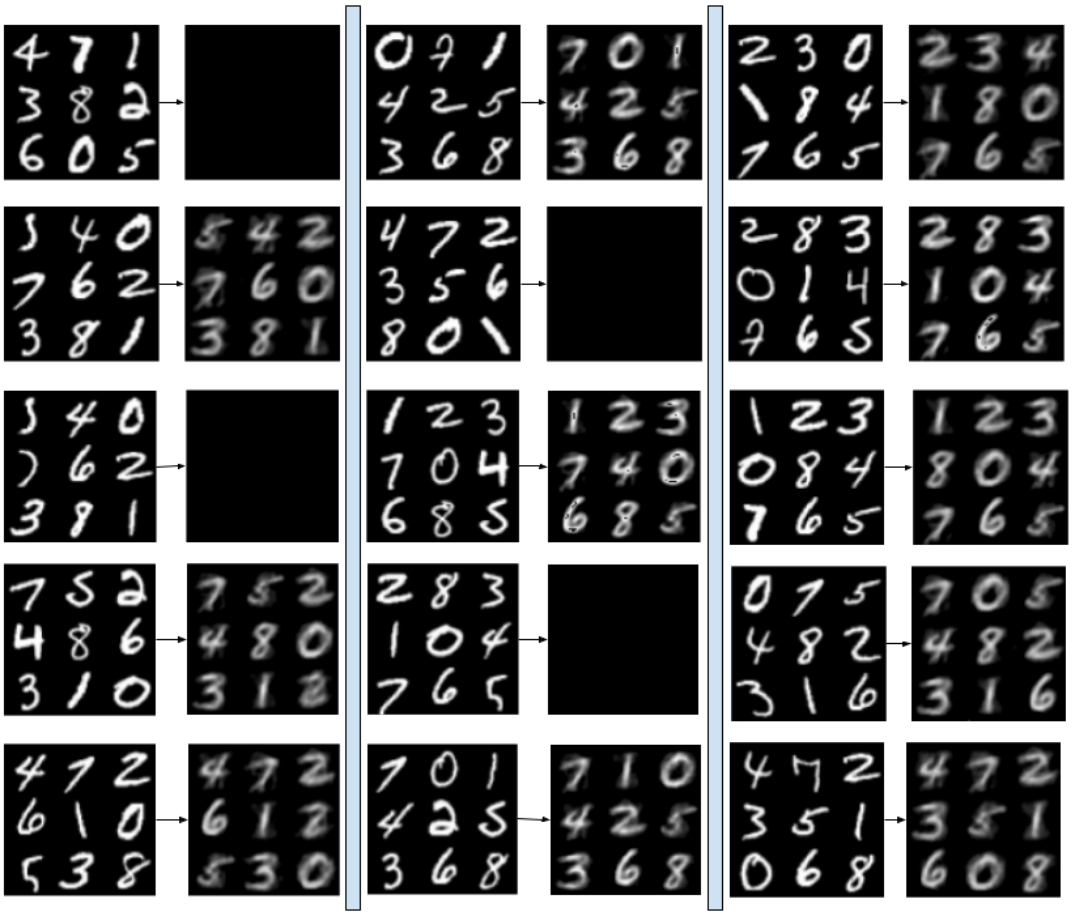

Figura 5.2: 15 Entradas y salidas del modelo entrenado para generar movimientos del *dataset 8-puzzle*

5.2 Entrenamiento del modelo con el *dataset lights-out*

Para confirmar que la configuración elegida en el punto 4.6 es adecuada para los problemas del presente proyecto se procedió a entrenar otro modelo usando el *dataset lights-out* y ver si los resultados eran buenos o no. Este *dataset* tenía una peculiaridad que lo diferencia del *8-puzzle*, y es que aquí no hay movimientos no posibles. Con esto en mente se podría asumir que al poder hacer siempre un mismo movimiento y que siempre fuera posible hacer ese movimiento ante cualquier situación el orden de prioridad tendería a elegir siempre el mismo movimiento.

A continuación se van a mostrar en figuras las evoluciones de la precisión al elegir movimiento, la evolución de la función de pérdida $decoderLoss(y_j, y'_j)$ y la evolución de la función de pérdida $movementsLoss(m_j, m'_j)$ en los ejes Y y los pasos de entrenamiento en el eje X. En este caso con 10 *epochs* fue suficiente para obtener muy buenos resultados.

En la primera imagen de la Figura 5.3 se puede ver que la precisión es 100 % casi desde el principio del entrenamiento. Esto se confirma con la segunda imagen de la Figura 5.3 que muestra la función de pérdida $movementsLoss(y_j, y'_j)$. En esa imagen la pérdida cae al principio del entrenamiento y después se mantiene estable. Respecto a la tercera imagen de la Figura 5.3 que muestra la función de pérdida $decoderLoss(y_j, y'_j)$, se observa que la tendencia es parecida a la expuesta en el punto 4.6 en la tercera imagen de la Figura 4.14.

La explicación de por qué la precisión es tan alta casi desde el principio del entrenamiento se debe a que siempre se elige el mismo movimiento. Esto tiene sentido, al no

Figura 5.3: La evolución de la precisión, las pérdidas del *movements* y del *decoder* de la mejor configuración encontrada para el *dataset lights-out*

tener restricciones el modelo siempre elige el mismo movimiento, que es lo más sencillo de aprender. Aquí sigue existiendo el orden de prioridad, pero todos los movimientos tiene probabilidad 0 de ser elegidos excepto uno que tiene una probabilidad del 100 %.

Para ver si se elegía siempre el mismo movimiento se decidieron realizar dos entrenamientos más con diferentes inicializaciones pero con la misma configuración del modelo. Las métricas de estos entrenamientos eran parecidas a las mostradas en la Figura 5.3. En el caso de los movimientos elegidos, sigue ocurriendo lo mismo, sólo se realiza un movimiento. La úncia diferencia es que el movimiento a aplicar es diferente con cada inicialización.

Con lo anterior expuesto se puede pensar que el orden de prioridad generado por el modelo depende de la inicialización realizada y de las restricciones del problema a la hora de realizar un movimiento. Si el *dataset* tiene restricciones el modelo generará un orden de prioridad, sino siempre elegirá el mismo movimiento.

5.3 Resultados *lights-out*

A continuación se van a mostrar varias imágenes de entrada de la red junto con las imágenes generadas a partir de las entradas. También se indicará el movimiento elegido por la red para esas imágenes.

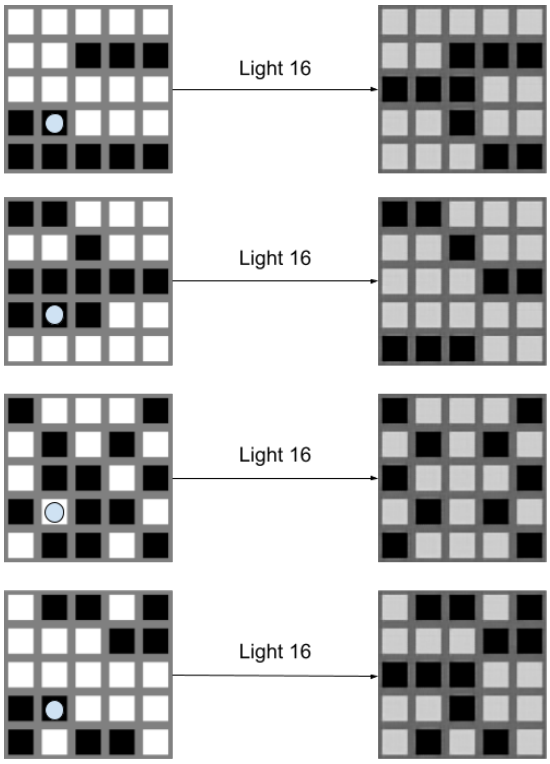

Figura 5.4: Entrada y salidas del modelo entrenado para generar movimientos del *dataset lights-out*

En la Figura 5.4 se muestran las predicciones realizadas por el modelo entrenado. Las entradas del modelo se pueden ver a la izquierda junto a un punto en la celda en la que se va a aplicar un movimiento (la luz 16 en todos los ejemplos) para una mayor facilidad de visualización. Las imágenes de la derecha son las generadas por la red y los movimientos predichos por la red aparecen encima de las flechas que unen las figuras de la izquierda con las figuras de la derecha.

Cómo se puede observar las imágenes generadas aplican un movimiento correctamente, cambiando de estado la luz 16 y las celdas circundantes a la celda 16 y dejando las demás celdas en el mismo estado. Además este movimiento está etiquetado de manera correcta (no hay discordancia entre la imagen generada y el movimiento aplicado).

Para este problema se pueden realizar las mismas afirmaciones que en el punto 5.1 respecto a los *embeddings*. El modelo ha aprendido a generar unos *emeddings* de la imagen de entrada que permiten al *decoder* y al *movements* generar movimientos correctos y en concordancia. También se observa que el modelo es capaz de generar la estructura del tablero de forma correcta, tanto los bordes como las celdas y sus estados.

En este problema ocurre lo mismo que en el *8-puzzle*, en algunos casos el *decoder* produce imágenes en negro. Aún así el módulo *movements* sigue seleccionando movimientos correctamente. La probabilidad es algo mayor para este problema pues llega hasta el 25 % de los casos.

Además hay algunas veces que al *decoder* le cuesta generar correctamente las imágenes. En la Figura 5.5 se muestran algunas imágenes en las que las celdas con luces encendidas no están generadas correctamente. Se puede observar la diferencia respecto a la Figura 5.4. En la Figura 5.4 las luces encendidas están pintadas completamente en blanco. En cambio en la Figura 5.5 aparecen con partes en negro, cómo si las luces tuvieran manchas.

Aunque no se hayan generado de forma correcta, las celdas con luces encendidas son las correctas, por lo que el movimiento se está realizando correctamente. Esto podría arreglarse empleando más *epochs* de entrenamiento ya que el modelo generado sólo se entrenó con 10 *epochs* a diferencia del modelo del *8-puzzle* que se entrenó con 100 *epochs*.

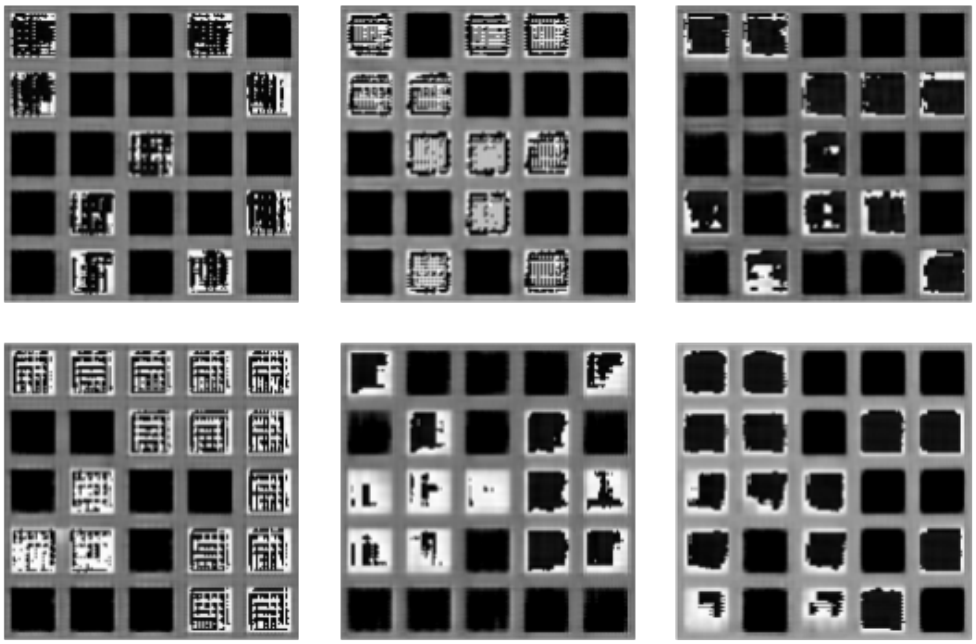

Figura 5.5: Resultados no totalmente correctos del *dataset lights-out*

Finalmente se van a mostrar 15 pares de imágenes en la Figura 5.6. Cada par contiene una imagen proporcionada al modelo y una imagen generada por el modelo a partir de la imagen de entrada. En la Figura 5.6 no sólo se muestran imágenes correctas, sino que también se muestran ejemplos de cuando una imagen no se genera de forma correcta.

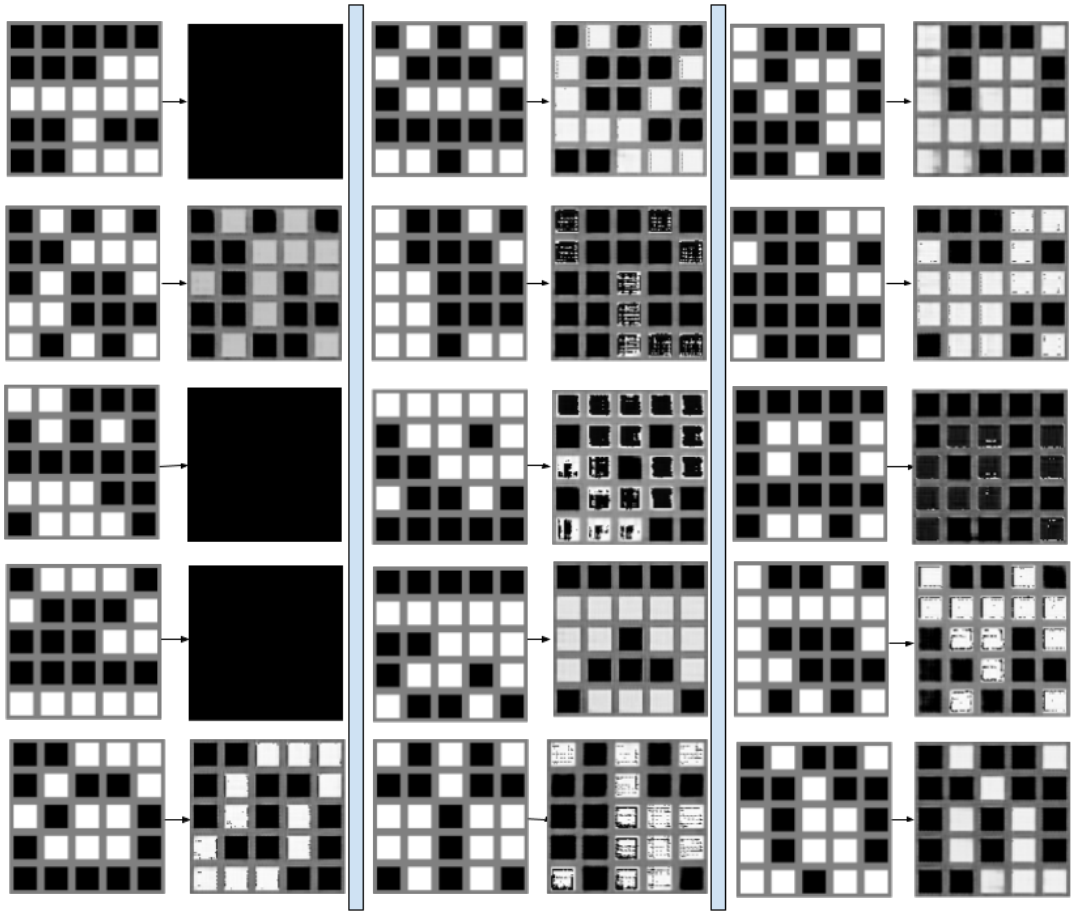

Figura 5.6: 15 Entradas y salidas del modelo entrenado para generar movimientos del *dataset lights-out*

CAPÍTULO 6
Conclusiones y Trabajo Futuro

En el presente proyecto se ha estudiado la generación de *embeddings* a partir de imágenes que representan el estado de un problema usando una arquitectura *U-Net* con ligeras modificaciones (usando capas residuales). Después se han usado dos módulos (*decoder* y *movements*) para aplicar movimientos a los estados representados en los *embeddings* y se ha aplicado a dos problemas, *datasets 8-puzzle* y *lights-out*.

En este capítulo se presentan las conclusiones del trabajo realizado y de los resultados obtenidos con los dos modelos entrenados. Finalmente se sugerirán trabajos futuros en base a los resultados obtenidos.

6.1 Conclusiones

Una vez analizados los resultados de los modelos entrenados para los problemas *8-puzzle* y *lights-out* se puede afirmar que los *embeddings* aprendidos tienen la información necesaria para codificar un estado y permitir a los módulos *decoder* y *movements* generar movimientos a partir de estos *embeddings*.

Además, viendo los resultados del capítulo 5 se puede afirmar que el modelo *U-Net+* es un buen modelo para resolver ambos problemas. Es un modelo capaz de generar unos *embeddings* apropiados que proporcionan las características suficientes para permitir a los demás módulos realizar sus tareas.

Aunque en la mayoría de casos los modelos entrenados sobre el *8-puzzle* y *lights-out* funcionan correctamente hay casos en los que tienen problemas a la hora de generar imágenes, tal y como se vio en el capítulo 5.

Además existe un problema respecto a la generación de movimientos, especialmente en el modelo entrenado sobre *lights-out*, y es que no se elige un movimiento que lleve a resolver el problema. Esto hace que los módulos *decoder* y *movements* carezcan de utilidad practica.

Es por esto que aunque el modelo *U-Net+* funciona bien hay que seguir realizando mejoras para resolver los problemas expuestos en el capítulo 5.

6.2 Trabajo futuro

Cómo trabajo futuro se podría:

- Estudiar como afectaría entrenar los modelos con muestras que contienen movimientos elegidos por un planificador inteligente. Esto podría aportar mejoras al

problema de la elección del movimiento por la red, pues se cambiaría la forma en la que se generan los *datasets*. Esto también implicaría que la forma en la que se gestiona la función de pérdida del modelo debería cambiar.

Haciendo que los movimientos que se usan para entrenar la red fueran generados por un planificador inteligente en lugar de forma aleatoria podría hacer que la red detectara patrones detrás de estas muestras y estableciera de manera correcta que movimientos elegir para llegar a una solución.

- Cambiar el módulo de *movements* por otro llamado *PDDL-decoder*. Este nuevo módulo se encargaría de transformar los *embeddings* a *PDDL*, por lo que se estaría pasando de una representación subsimbólica (*embedding*) a otra simbólica (*PDDL*). Esto proporcionaría una gran ventaja, pues la persona encargada de generar representaciones simbólicas en *PDDL* de un estado no necesitaría codificar manualmente el problema, sino que podría usar la red con el módulo *PDDL-decoder* para generarlo. Esto no se ha abordado en este trabajo por exceder el alcance del proyecto.

- Realizar modificaciones sobre la *U-Net+* para resolver los problemas explicados en el capítulo 5, por ejemplo, probando otro tipo de bloques que no sean residuales.

- Cambiar parte del modelo para usar los modelos conocidos cómo *transformers* [31], ya que están aportando muy buenos resultados en una gran variedad de tareas (clasificación de imágenes, generación de imágenes, similitud entre textos, resumen de textos de forma automática...).

Bibliografía

[1] Feature vector. https://brilliant.org/wiki/feature-vector/. Accessed: 2023-7-9.

[2] *Mean Squared Error*, pages 337–339. Springer New York, New York, NY, 2008.

[3] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary, 2017.

[4] S.H. Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. Impact of fully connected layers on performance of convolutional neural networks for image classification. *Neurocomputing*, 378:112–119, feb 2020.

[5] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[6] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

[7] Florinel-Alin Croitoru, Vlad Hondru, Radu Tudor Ionescu, and Mubarak Shah. Diffusion models in vision: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–20, 2023.

[8] Zahra Zamanzadeh Darban, Geoffrey I. Webb, Shirui Pan, Charu C. Aggarwal, and Mahsa Salehi. Deep learning for time series anomaly detection: A survey, 2022.

[9] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[10] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019.

[11] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[12] Borko Furht. Image presentation and compression. 01 1999.

[13] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004.

[14] Elliott Gordon-Rodriguez, Gabriel Loaiza-Ganem, Geoff Pleiss, and John P. Cunningham. Uses and abuses of the cross-entropy loss: Case studies in modern deep learning, 2020.

[15] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[17] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[18] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[20] David Joslin and Martha E. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2*, pages 1004–1009. AAAI Press / The MIT Press, 1994.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[22] Will Koehrsen. Neural network embeddings explained, October 2018.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[25] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl–the planning domain definition language–version 1.2. *Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165*, 1998.

[26] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[28] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[30] Ekin Tiu. Understanding latent space in machine learning, February 2020.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[32] Wikipedia. Red neuronal convolucional — Wikipedia, the free encyclopedia. http://es.wikipedia.org/w/index.php?title=Red%20neuronal%20convolucional&oldid=148733539, 2023. [Online; accessed 09-July-2023].

[33] Zeke Xie, Issei Sato, and Masashi Sugiyama. Understanding and scheduling weight decay, 2021.

[34] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. A survey of deep learning techniques for neural machine translation, 2020.

[35] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. How does learning rate decay help modern neural networks?, 2019.