PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

TAREA 5:

Aprendizaje Reforzado y

Redes Neuronales

Anna Ramon Hinojosa

aramonh@uc.cl

11/12/2023

DCCasas

1.1. Comprendiendo los datos

1.1.1.

Las redes neuronales reciben como entrada una gran cantidad de datos con el objetivo de aprender una tarea determinada y poder llegar a generalizarla. Son capaces de extraer las features más relevantes para dicha tarea, así como ir ajustando los pesos asignados a cada neurona para ajustar el mejor modelo posible.

Para que el entrenamiento se desarrolle de la mejor manera posible es importante que los datos estén normalizados, se hayan tratado los missing values, eliminado los valores atípicos, entre otros aspectos. Este proceso de preparación de los datos para ser insertados en la red neuronal se conoce como el preprocesado. Las redes neuronales son sensibles a la escala de los datos, si no se normalizan, puede entender que los atributos con un rango de valor superior a otros son más relevantes a la hora de realizar la tarea deseada. Para evitar esta confusión en la importancia de features es crucial mantener los atributos normalizados. Por otro lado, es importante tratar los outliers en el preprocesado de datos, ya que los valores atípicos pueden confundir a la red neuronal y no ser capaz de captar bien el comportamiento de los datos en cuestión. En resumen, no preprocesar los datos supondría una potencial mala generalización del comportamiento de la red.

1.1.2.

Las redes neuronales funcionan muy bien si se les entrega una gran cantidad de datos, ya que como más instancias haya mayor es el potencial para aprender a generalizar los features que contribuyen altamente en el desarrollo de la tarea de interés. Sin embargo, no siempre se poseen muchos datos. En el caso de instancias de imágenes hay distintas técnicas que permiten aumentar el conjunto de datos.

- Data Augmentation. Consiste en la modificación artificial de las imágenes de entrada permitiendo obtener un conjunto de datos mayor. Las modificaciones son la ingestión de ruido en la imagen (eliminando o distorsionando el valor de algunos pixels), modificación del contraste de la imagen, cambio de los colores, hacerla un poco borrosa…
- Random cropping. Realizar cortes aleatorios en las imágenes para generar nuevas con algunas distorsiones.
- Uso de GANs, Redes Generativas Adversarias. La parte de la red generativa permite la creación de imágenes sintéticas 'falsas' que podrían utilizarse para ampliar el conjunto de datos de entrada.

Información extraída de Image and Video Processing Group, TSC, UPC. Machine Learning II. Practical Aspects. Methodology.

1.2. Explorando los Multilayer Perceptron (MLP)

1.2.1.

Aumentar el número de capas y neuronas de una red le entrega mayor potencial para captar patrones complejos. El número de neuronas por capa podría ser análogo al número de features de estudio, a más neuronas, más capacidad de captar el comportamiento de los datos. El número de capas podría ser comparado con la relevancia a lo largo del tiempo de las 'features' captadas en las capas anteriores y la jerarquía que se representa. Sin embargo, utilizar muchas neuronas y muchas capas puede desarrollar un modelo sobreajustado, además de destacar que el tiempo de entrenamiento aumenta exponencialmente al aumentar estos parámetros.

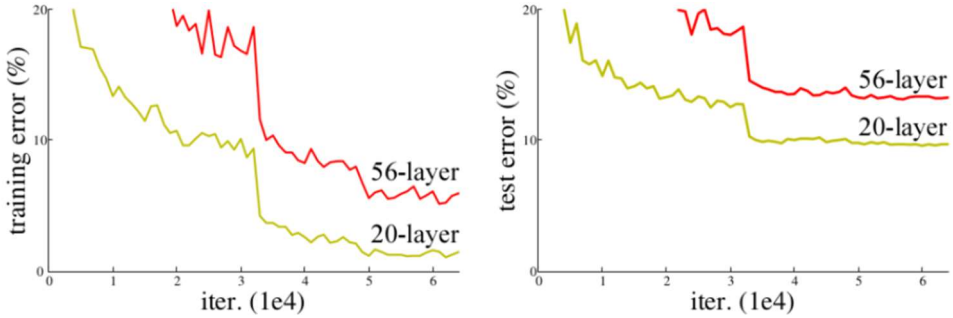

Figura extraída de: https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/

Debido al aumento de parametros a entrenar asociado a tener un número de neuronas y capas alto, la red puede perder su capacidad de generalización ya que se produce una degradación del problema. En resumen, se busca alcanzar un trade-off entre la complejidad del modelo y la capacidad de capturar patrones correctamente, evitando modelos sobreajustados (muchos parametros) y infra ajustados (pocas neuronas y capas).

1.2.2.

La función de activación se aplica a la salida de cada neurona por cada capa y determina si la neurona debe activarse o no, es decir, si el comportamiento que estaría capturando esa neurona es relevante para la determinación de la tarea en cuestión. Las funciones de activación no lineales permiten la generalización del modelo, facilitando la adaptación a diferentes tipos de datos. Dos funciones de activación no lineales comúnmente utilizadas son la ReLU y la Sigmoid:

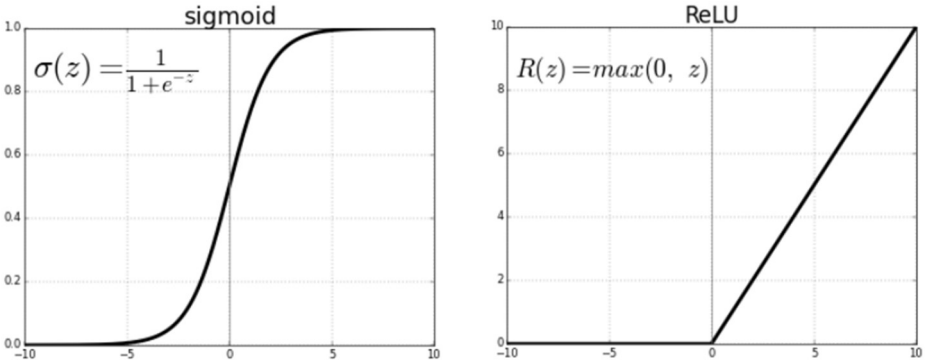

Figura extraída de: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

- ReLU: Si el resultado de combinar las entradas ponderándolas con los pesos asociados a la neurona es negativo, esa neurona será determinada como no activa. Todos los valores negativos son mapeados a 0 por la función ReLU. Es una función que se utiliza en las capas ocultas, ya que permite aprender patrones no lineales en los datos (dada su condición de no linealidad). Si las entradas son positivas, mapea cada valor a él mismo. Esto permite un flujo de gradiente más eficiente, evitando el problema de reducción del gradiente durante la backpropagataion que tienen alguna funciones de activación que se saturan.

- Sigmoid: Retorna valores entre 0 y 1, de modo que es especialmente utilizada para predecir probabilidades. Si el objetivo de la red es aprender a clasificar los datos de entrada en 2 clases, parece razonable utilizar una sigmoid en la última capa de la red, determinando la probabilidad de la instancia en pertenecer a la clase 1 o a la clase 2. Esta función tiende a saturarse en los extremos pues la derivada de la función se aproxima mucho a cero en los extremos. El problema que se desencadena es el desvanecimiento del gradiente de modo que al actualizar los pesos con la backpropagation, las actualizaciones resultan insignificantes.

1.2.3.

El objetivo es entrenar una red neuronal que generalice a nuevos datos de entrada. El problema del overfitting se da cuando una red se adapta demasiado a los datos de entrenamiento, resultando con un modelo demasiado complejo. Para prevenir el overfitting se pueden desarrollar distintas técnicas:

- Early stopping. Ir comparando las métricas de evaluación del test de train y el de validación. En el momento que el error de validación empiece a aumentar y el de train siga bajando, implica que el modelo se está sobre ajustando. En ese momento aturar el entrenamiento.
- Regularization. Controlar la capacidad de la red, el entrenamiento de muchos parámetros puede tender al overfitting. La asignación de pesos altos a las neuronas de la red puede conllevar transiciones bruscas, limitar el valor de los parámetros en la red.
- Normalization. Normalizar los valores de entrada en cada iteración con el objetivo de reducir la correlación.
- Dropout. En cada iteración de entrenamiento, aleatoriamente borrar algunas neuronas. Es decir, en el proceso de backpropagation, eliminar algunas neuronas y no propagar el error en esas. De este modo, las neuronas permanecen más insensibles a las ponderaciones de otras neuronas, proporcionando una red mas robusta.
- Data Augmentation. Explicado anteriormente.

1.2.4.

Un método de optimización utilizado en el entrenamiento de MLP es el Momentum o el Nesterov Momentum combinados con métodos para adaptar el learning rate como adaGrad o RMSProp.

El método del descenso del gradiente puede ser muy lento y estancarse en puntos críticos diferentes del punto de interés, el mínimo global. Como alternativa se presenta el método del Momentum que contempla la trayectoria del descenso del gradiente seguida hasta el punto actual, sería la concatenación ponderada de los gradientes hasta el momento. Se determina la variable velocidad almacenando el promedio de los gradientes vistos y ponderados según su antigüedad. Para cada iteración se computa:

$v^{k+1} = \gamma v^k - \alpha \nabla_\theta \mathcal{L} \quad |_{\theta^k}$

$\theta^{k+1} = \theta^k + v^{k+1}$

Este método es muy útil cuando la superficie a minimizar presenta derivadas heterogéneas en direcciones diferentes, pues la utilización del método del descenso de gradiente iría iterando de un lado a otro siendo muy lenta la convergencia. Utilizar el método del momentum permitiría guiar la búsqueda sin ir de un lado a otro, contemplando la inercia que lleva el descenso. Otro parámetro que contemplar es la α que indica que tanto se considera el valor del gradiente anterior. El método AdaGrad adatpa el learning rate para cada parámetro de θ basándose en sus previas actualizaciones siguiendo la siguiente política: a un valor de gradiente pequeño se le asocia una α grande, empujar el valor del parámetro hacia el gradiente. Mientras que a un valor de gradiente grande se asocia un learning rate pequeño, pues un gradiente grande implicaría un cambio muy grande en el valor del parámetro y no queremos que se actualice tanto.

1.2.5.

La detección de transacciones fraudulentas es un problema que puede ser afrontado con MLP. Una red neuronal puede ser entrenada para detectar patrones poco usuales en un conjunto de datos que almacena diferentes características sobre el comportamiento de usuarios. El objetivo es sacar provecho de la capacidad de detección de patrones que proporcionan las redes neuronales. En el momento que la red identifique un comportamiento atípico o propio de las instancias determinadas como fraudulentas en su entrenamiento, se clasificará dicho usuario asociado como potencialmente fraudulento.

Al tratarse de una red neuronal, para obtener un buen rendimiento es necesario entrenarla con una gran cantidad de datos. Un banco que quiere detectar potenciales acciones fraudulentas, puede permitirse este sistema, pues almacena una gran cantidad de información sobre las transacciones de sus clientes. Si la información contiene una etiqueta sobre si la acción fue fraudulenta o no, podemos aplicar un MLP basándose en aprendizaje supervisado. Si no se contiene la etiqueta, se puede aplicar un MLP que se especialice en la extracción de patrones y acabe clasificando aquellas instancias similares (fraude vs no fraude).

En este caso se dispone de la etiqueta asociada a cada transacción. Se le entrega a la red las características de la transacción: quién la realiza, a quién, la cantidad de dinero entregada, el día y la hora y la etiqueta asociada. La clasificación es un proceso iterativo donde los pesos de las neuronas van actualizándose conforme el algoritmo determinado (Momentum con AdaGrad) aplicando el mecanismo de propagación del error con backpropagation. El output de la red sería la clasificación binaria de la instancia, fraude / no fraude. Una buena función de activación en la última capa podría ser una sigmoide, asociando la probabilidad de pertenecer a una clase u otra.

1.3. Implementación de un MLP

Para entrenar el MLP partimos del conjunto X_mlp que contiene atributos numéricos y categóricos. Inicialmente se convierten las variables categóricas *Street, city* en variables numéricas aplicando un label encoder. En segundo lugar, dividimos el conjunto de datos (12518 instancias) en train (un 80%) y en test (un 20%). Para entrenar adecuadamente un MLP y evitar que de más peso a unos atributos que a otros por el simple hecho de tener un rango de valores superiores y no por ser realmente más relevantes, se escalan los datos. En hacer fit del modelo con los datos de train escalados, destinamos un 20% para el set de validación. Al tener tantas instancias en el datset, tiene sentido destinar un 20% a instancias de testeo, pues el dataset de train ya será bastante grande (buena representación de la realidad) y las instancias utilizadas para test también pueden ser una buena muestra representativa de la realidad. A más instancias, más fiel a la realidad puede ser el comportamiento que capture nuestro modelo, con lo que nos interesa entrenar con el mayor número de instancias posibles, pero manteniendo un set de test que también sea bastante representativo. En caso contrario, el modelo resultante no será capaz de generalizar y/o las métricas de test no serán representativas.

El objetivo de partida es analizar el efecto de la profundidad en una red, el número de neuronas, el tipo de optimización empleada y las funciones de activación. El método de optimización Adam se adapta a las necesidades de cada parámetro a optimizar, es decir, en función de su gradiente decide disminuir o aumentar el learning rate. Parece un método optimo y es por este motivo que será aplicado en los diferentes modelos MLP entrenados. Se utilizan 50 épocas y un batch de dimensión 32 para los entrenamientos.

Nº capas ocultas	**Neuronas por capa**	**Función de activación**	**MSE TRAIN**	**MSE TEST**
2	64 – 32 – 1	ReLU x2	$2.79 * 10^{11}$	$2.84 * 10^{11}$
3	64 – 32 – 32 – 1	ReLU x3	$8.76 * 10^{10}$	$9.35 * 10^{10}$
3	64 – 32 – 32 – 1	ReLU x2 + Sigmoide	$5.93 * 10^{11}$	$6.06 * 10^{11}$
4	64 – 32 – 32 – 16 – 1	ReLU x4	$8.21 * 10^{10}$	$8.81 * 10^{11}$
4	64 – 32 – 32 – 32 – 1	ReLU x4	$8.16 * 10^{10}$	$8.82 * 10^{10}$
4	128 – 64 – 32 – 16 – 1	ReLU x4	$8.16 * 10^{10}$	$8.79 * 10^{10}$
4 (200 epochs)	128 – 64 – 32 – 16 – 1	ReLU x4	$8.08 * 10^{10}$	$8.76 * 10^{10}$

Conclusiones:

- Dar profundidad a la red resulta con un MSE más pequeño, tanto en train como en test. Por lo tanto, parece que añadir más capas está permitiendo a la red entender mejor el comportamiento de los datos, mejorando su rendimiento y favoreciendo a la generalización. Hay que tener en cuenta que llega un punto que dar más profundidad puede implicar un aumento del error debido a estar sobre ajustando. De momento con 4 capas no sobre ajusta, pues el rendimiento mejora en comparación con 2 capas ocultas.

- Utilizar la función de activación no lineal sigmoide empeora los resultados, obteniendo una red que siempre acaba prediciendo un mismo valor. El MSE empeora mucho. Este hecho podría atribuirse a la saturación de la red.
- Aumentar el número de neuronas por capa puede provocar una mejoría en los resultados, en especial si se utiliza un numero de neuronas alto en las primeras capas, pero puede empeorar los resultados si se utilizan muchas neuronas en las últimas capas. En los dos últimos experimentos, ambos con 4 capas ocultas, realizamos que utilizar el doble de neuronas en la última capa (16-32) puede reducir el error de TRAIN y empeorar el de TEST, lo cual indicaría una tendencia a sobre ajustar.
- Aumentar el número de epochs supone que el modelo pase por todo el conjunto de entrenamiento más veces durante el proceso de optimización. En este caso pasar de 100 a 200 epochs parece que mejora el rendimiento del modelo, permitiendo un mejor ajuste a los datos. Sin embargo, este proceso tiene un potencial riesgo al sobre ajuste.

Las características que parecen tener un impacto más significativo en el rendimiento del modelo son el número de capas, neuronas y las funciones de activación utilizadas. La profundidad y amplitud del modelo permiten que este sea capaz de detectar patrones. Las funciones de activación no lineales contribuyen en la posibilidad de captar los patrones más complejos, aumentando la capacidad del modelo de una simple transformación lineal a modelos más complejos que pueden ajustarse muy bien a los datos.

Los modelos entrenados no parecen tener muy buen rendimiento, el valor de MSE es muy alto. Sin embargo, sí se puede apreciar el impacto en el MSE de cada parámetro estudiado.

Un MLP tiene un buen rendimiento en problemas supervisados, en especial en clasificación, tanto binaria (uso de una sigmoid al final de la red que determina la probabilidad de la instancia en pertenecer a una clase u otra) como multiclase (uso de una softmax que determina la probabilidad de cada clase). MLP permite reconocer patrones complejos, por lo tanto, en problemas de detección de comportamientos en un conjunto de datos, también pueden aportar mucho valor.

1.4. Introducción y teoría de las CNNs

1.4.1

En el contexto de procesamiento de señales, la convolución es la función resultante de la combinación entre dos funciones representando la interacción entre ellas. En las CNNs se entiende la operación de convolución como el resultado de multiplicar elemento a elemento los valores de la matriz de entrada (condición esencial, que los datos estén cuadriculados) con una matriz kernel, filtro, que va desplazándose por cada posición de la matriz input. A continuación se muestra una imagen para entender mejor la operación de convolución con stride=2 (moviendo los outputs 2 posiciones). Los valores representados en los cuadros azules son el resultado de multiplicar 3 por cada valor del filtro. El 0, 4, 2 de la casilla (2+6) son los valores asociados a multiplicar 2 por el kernel. Las casillas con valores solapados se suman.

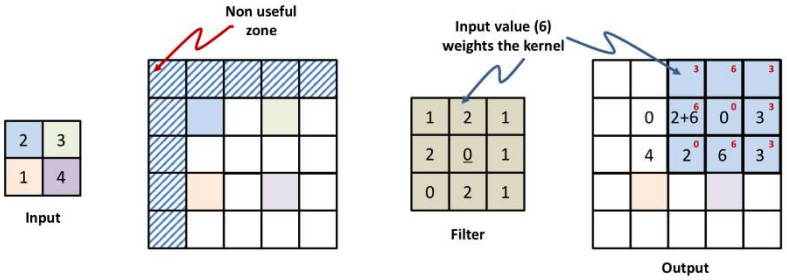

Figura extraída de: Dumoulin et aI, A guide to convolutional arithmetic for Deep learning

La idea es aplicar diferentes filtros que se ha demostrado que son capaces de capturar patrones en concreto. Por ejemplo, aplicar un filtro Sobel, resulta con la captación de los contornos verticales de la imagen.
Cada capa convolucional es capaz de extraer unas características determinadas. Cada filtro (kernel) aplicado en una capa se centra en detectar patrones específicos.

1.4.2.

Una CNN sin funciones de activación tiene la capacidad limitada ya que es equivalente a una transformación lineal, imposibilitando la capacidad del modelo para captar patrones complejos y no lineales en los datos. Cada capa aplicaría una transformación lineal de la entrada y la composición de capas resultaría siendo una función lineal. Por lo tanto, las funciones de activación no lineales tienen un rol muy importante tanto en las CNN como en las redes neuronales, permitiendo al modelo la posibilidad de aprender patrones complejos de los datos.
El rol de las capas max pooling es reducir la dimensionalidad de los datos, quedándonos únicamente con los detalles más relevantes. En contexto de imágenes, aplicar este método suele resultar con la imagen original más resaltada (sharped). Reducir la dimensionalidad de los datos implica reducir el número de parámetros, favoreciendo la reducción del sobre ajuste, pues los datos que descartamos son los menos relevantes.

Información extraída de: Introducción a Deep Learning. Inteligencia Artificial. Pontificia Universidad Católia de Chile. Hans Löbel.

1.4.3.

La flatten layer se encarga de aplanar los datos de entrada de modo que se obtenga un vector. En las CNN el resultado de aplicar una capa, a unos datos de entrada, es un tensor: por cada filtro que tenga la capa se obtiene una matriz, estas matrices se apilan y forman el tensor de salida. Al final de la red CNN se aplica un conjunto de capas fully-connected que no puede recibir un tensor como intput, por lo que previamente se aplana el tensor con una Flatten layer.

La función softmax permite calcular la probabilidad que tiene el conjunto de datos de entrada de pertenecer en cada una de las posibles clases, al coger el máximo de dichas probabilidades, se determina que cada instancia pertenece a la clase con mayor probabilidad asociada. Esta función suele aplicarse a la última capa del conjunto fully-connected en un problema de clasificación multiclase.

1.4.4.

Tanto las CNNs como las MLPs son dos arquitecturas diferentes de redes neuronales. En las MLPs las neuronas de cada capa están completamente conectadas con las neuronas de la capa siguiente. En el caso de las CNNs se centran en capturar patrones locales y jerarquías de características con lo que no todas las neuronas están conectadas entre ellas.

Las CNN suelen utilizarse para tratar con datos que son imágenes. Son resistentes al cambio de pixeles en una misma imagen, resultando más consistentes en este aspecto que un MLP. Es muy útil el uso de CNN para tareas de visión por computadora, análisis de imágenes. Procesar una imagen con un MLP, donde cada neurona está fully connected con las neuronas de la capa anterior , resulta con un número de parámetros de la red exponencialmente grande. La red resultante seria muy compleja de entrenar y muy propensa a ser sobre ajustada, al tener una red neuronal con tantos parámetros puede saturarse. En cambio, las CNNs son muy buenas para procesar datos con una estructura conocida ya que son capaces de explotar su estructura y reducir el número de parámetros de la red.

1.5. Creando y evaluando una CNN

Intenté implementar la arquitectura VGG-16 basada en CNN (en el archivo de código hay la celda donde se define la arquitectura), esta red se compone de 16 capas de profundidad lo cual supone un número de parámetros a entrenar altísimo. Además, al tratarse de un dataset bastante grande, 12518 instancias, la cantidad de parametros y variables a almacenar es muy alta. Al tratar de definir y entrenar el modelo VGG-16 me saltaba un error debido a haber ocupado todo el espacio disponible de la CPU. Como solución, implemente un modelo baseline muy básico, con dos capas convolucionales con un filtro/kernel 3x3 y una capa de max pooling al final de 2x2. Acto seguido se aplica un Flatten para poder aplicar tres capas fully-connected, dos hidden layers de 256 neuronas y una última capa que permite la predicción del precio por instancia que simboliza las características de una casa.

Para comparar la performance de este modelo básico con una opción más compleja, decido añadir dos capas convolucionales más de 128 y con los mismos kernels 3x3 y una capa max pooling al final. Se hace un flatten que permite introducir la segunda parte de la arquitectura de interés, la parte fully-connected: dos capas de 256 neuronas y una que permite hacer la predicción del precio de la casa correspondiente a la instancia de estudio.

En este apartado utilice los conjuntos de train y test definidos en uno de los chunks que aparecían en este apartado. En el enunciado se explicaba que para esta parte se debía utilizar X_cnn, pero encontré que el preprocesado que se aplicaba en este apartado permitía optimizar el tiempo de entrenamiento, pues el tamaño de las imágenes era menor. Otro aspecto a comentar es que el accuracy que se propone para la tarea:

$$accuracy = \frac{|precio\ real - prediccion\ de\ precio|}{precio\ real} * 100$$

No era representativa, pues si la predicción era muy precisa: $precio\ real = prediccion\ de\ precio$ el accuracy era 0 y realmente quería todo lo contrario, por lo que modifiqué ligeramente la métrica a:

$$accuracy\ customizado = 100 - accuracy$$

Resultados de los modelos entrenados:

MODEL	TR LOSS	TE LOSS	ACC TRAIN	ACC TEST
BaseLine	$1.53 * 10^{10}$	$1.49 * 10^{11}$	35.36	43.04
Complejo	$1.97 * 10^{10}$	$1.57 * 10^{11}$	34.75	43.28

El modelo Baseline con 133012033 parámetros entrenables parece que obtiene mejores métri cas que el modelo un poco más complejo (con 2 capas convolucionales y una max pooling m ás que el Baseline) y 66779969 parámetros entrenables. En ocasiones, añadir profundidad al modelo no aporta una mejora en los resultados.

1.6. Comparación de modelos

A continuación, se comparan las métricas de los mejores modelos que se han encontrado con MLP y utilizando CNN. Los valores de Loss obtenidos son muy grandes en ambas arquitecturas, lo que indica que las predicciones no son muy precisas. En el modelo de MLP se observa que de partida la loss es muy grande, pero con pocas iteraciones el valor disminuye drásticamente, sin embargo, queda estancado en el resto de épocas. En el caso del modelo CNN se observa como el valor de la loss en train disminuye a lo largo del tiempo, mientras que el valor de loss en el set de validación oscila bastante al principio y tiende a un comportamiento opuesto al del set de train, parece disminuir al principio, pero aumenta al final. Mostrando una tendencia al overfitting.

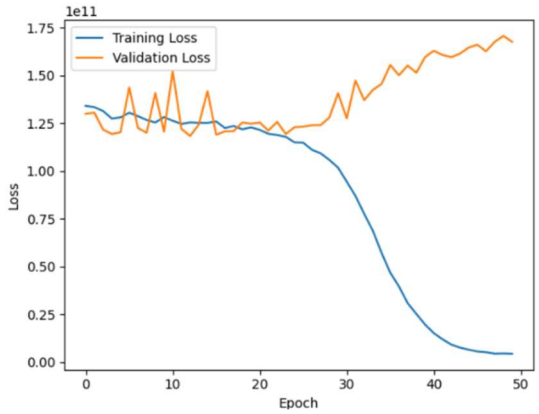

Figura: Loss del Modelo CNN complejo

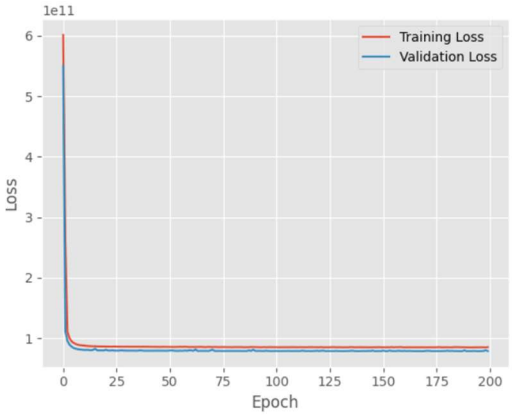

Figura: Loss del mejor modelo MLP definido

La loss más baja se obtiene con la arquitectura CNN. Parece que las imágenes están aportando más información para predecir el precio de una casa que los parámetros *Street, city, bedrooms, bathrooms, sqtf* que utiliza la MLP para predecir el precio de las viviendas. En cuanto a las arquitecturas basadas en CNN utilizadas, parece que el modelo más básico es el que obtiene mejores métricas, destacando que ambos parecen tener una tendencia al overfitting, pues la loss de test es superior a la de train y comparando los sets de train y validación se ve un escalado del valor de la loss en validación.

DCCanario

Actividad 1

La función del *exploration rate* es encontrar el punto óptimo ente explorar caminos nuevos (opciones nuevas que se desconoce la recompensa asociada) y explotar los caminos que ya se conoce que son buenos (tienen una alta recompensa asociada). Un *exploration rate* mínimo con un valor bajo implica mayor explotación, tomar aquellas decisiones que proporcionan una mayor recompensa esperada. El mecanismo consiste en explotar si al escoger un número aleatorio entre [0,1] su valor es superior a el valor de exploration rate, si es menor, se aplica una acción random (explorar). Un *exploration decay rate* alto implica que, en cada iteración, el valor de *exploration rate* disminuya, marcando una clara tendencia a la explotación. En resumen, un *exploration rate* mínimo con un valor muy bajo y un *exploration decay rate* con un valor muy alto puede llevar a que el agente decida no explorar lo suficiente, pues desde el inicio tiene una clara tendencia a la explotación de las altas recompensas conocidas. Llevando al agente a una situación donde quede atrapado en un conjunto limitado de acciones (las ya conocidas).

Si el *exploration decay rate* es demasiado bajo, el valor del *exploration rate* no se actualizará mucho. Inicialmente, interesa tener un exploration rate alto para fomentar la exploración de soluciones, pero a medida que van transcurriendo iteraciones (ya se conoce el valor de recompensa esperado de distintas acciones y estados) interesa aplicar más explotación que exploración. Esto se regula con el *exploration decay rate,* si es bajo, no se actualizará *exploration rate* y puede que el agente siga explorando a una tasa elevada durante muchas iteraciones. Una exploración excesiva resultaría con un algoritmo que funciona casi de manera aleatoria, siendo muy alto el tiempo hasta convergir.

En el juego del *Flappy Bird* nos interesaría partir con un *exploration rate* alto, que permita explorar qué acciones parecen tener mejores recompensas esperadas. Una vez hemos explorado suficiente (se conoce cómo afecta cada uno de los 5 parámetros de get_state() para la toma de decisión saltar o no y la recompensa asociada) interesa disminuir el *exploration rate* para explotar las mejores acciones. El *exploration rate decay* se encarga de control el trade-off entre una exploración inicial e ir aumentando la explotación a medida que se va ganando conocimiento (transcurso del tiempo). Contextualizando la situación de un *exploration rate* mínimo con un valor muy bajo y un *exploration decay rate* muy alto con el juego del Flappy Bird, se podría dar la situación que el Bird siempre muere en el mismo sitio. Estaría explotando al máximo las recompensas asociadas a los estados que conoce, pero al no haber explorado suficiente otras posibles acciones, no podría llegar a conocer el mejor comportamiento para superar todas las tuberías.

Actividad 2

Para implementar el algoritmo Q-learning he seguido las instrucciones indicadas en el código. A destacar sería la forma de definir la q-table, siendo un diccionario:

- KEY: tupla de 5 valores representando cada una de las combinaciones posibles determinadas por get_state() = 159936.
- VALUE: list de dos valores representando el q-value asociado al estado KEY en función de la acción que se decide ejecutar, la posición 0 corresponde al de no moverse y la posición 1 a moverse

Para el get_action() me basé en la definición del algoritmo q-learning que busca un trade-off de exploración y explotación, determinando un thershold: por encima de este se explota y por debajo de explora. Tras cada acción se actualiza la q-table y en terminar una partida se actualiza el *exploration rate* según la definición que dicta el q-learning algoritmo. Para determinar el fin del entrenamiento se puede limitar el número de episodios a realizar o bien detener el entrenamiento cuando la q-table haya convergido. En este caso la q-table consta de muchos estados: 159936 con lo que la convergencia puede ser muy larga. Decido limitar el número de episodios de entrenamiento, probando el rendimiento con 300 y 1000 episodios.

Actividad 3

Con los parámetros iniciales:

LR = 0.3

NUM_EPISODES = 10_000_000

DISCOUNT_RATE = 0

MAX_EXPLORATION_RATE = 1

MIN_EXPLORATION_RATE = 0.0002

EXPLORATION_DECAY_RATE = 0.00015

Mi modelo era incapaz de aprender, en todas las partidas el pájaro se chocaba con la parte superior del juego, terminando la partida. El problema es que utilizar un *discount_rate = 0* imposibilita la maximización de la recompensa dado un estado de partida y una acción a ejecutar. El agente se estaría enfocando más en las recompensas inmediatas, mientras que el objetivo es que sea capaz de captar que tan cerca se encuentra la tubería, para evitar chocar con ella. Aumentando el valor del *discount_rate* el agente (pájaro) rápidamente aprende a no chocar con las tuberías ni con la parte superior del entorno.

El LR controla la rapidez con la que el agente actualiza las estimaciones de los q-values en función de las nuevas experiencias. Un LR bajo prioriza la resistencia a la actualización de valores, manteniendo el valor calculado hasta el momento, mientras que un LR alto favorece la actualización de los q-values. Por default se nos entrega un LR de 0.3, lo cual indicaría una tendencia hacia la resistencia de la actualización de los q-value, manteniendo principalmente el q-value anterior y agregándole parte de la información capturada en la iteración del momento. Experimentando con distintos valores para el LR (0.001, 0.2, 0.3) se observa que el agente aprende más rápido como debe moverse en el espacio parar ‘ganar’ el juego si el LR es bajo. Con

un LR=0.01 el agente parece ser capaz de capturar muy bien el comportamiento a seguir pero consultando los valores q-value asociados a cada estado y acción vemos que son muy bajos y muy parecidos entre las dos posibles acciones distintas a realizar. Esto es debido a que un LR tan abjo fomenta un aprendizaje lento, con lo que se necesitarían muchos episodios para acabar obteniendo unos resultados buenos. Con LR=0.2, que sigue siendo bajo, permite no perder de vista el q-value histórico y añadirle poco a poco información de las acciones actuales.

Finalmente, se utiliza un LR=0.2 y un DISCOUNT_RATE=0.9. Un LR bajo implica un aprendizaje más lento, ya que el agente es cauteloso al cambiar sus estimaciones, pero proporciona un aprendizaje más estable. Un DISCOUNT_RATE cercano a 1 implica que el agente considere las recompensas futuras en mayor medida, permitiéndole al agente comprender y aprender a evitar el choque con las tuberías y los bordes del entorno. La combinación de un LR bajo y un DISCOUNT_RATE alto proporciona un balance entre la estabilidad del aprendizaje y la capacidad del agente para considerar las consecuencias a largo plazo de la toma de sus acciones.

(Adjunto las tres tablas asociadas a los q-values, las de LR=0.001, 0.3 son con 300 episodios y la LR=0.2 es con 1000 episodios)

Actividad 4

En las prácticas de aprendizaje supervisado, el agente aprende a cómo comportarse en función de la política de recompensa definida. El objetivo del agente es maximizar la recompensa que obtiene y en base a esta decide tomar una acción u otra. Por lo tanto, es muy importante la definición de la política de recompensa.

Para diseñar una nueva forma de recompensar al agente es relevante explotar el comportamiento deseado: realizar acciones que conduzcan al agente a través de las tuberías, sin chocar con ellas ni con los bordes del entorno, y alinear este comportamiento con la función de recompensa. También es importante incentivar al agente a la superación de obstáculos, con lo que se debe ajustar la recompensa según la distancia del pájaro al siguiente conjunto de tuberías.

- Asignar una recompensa alta por pasar exitosamente la tubería. +50puntos
- Asignar una penalización por chocar con la tubería o bordes del entorno. -1punto.
- Asignar una recompensa si el agente se acerca hacia el agujero de la tubería. +1punto.
- Asignar una penalización si el agente se aleja del agujero de la tubería. -1punto.

Estas situaciones ya se contemplan en la política actual. Además de estas, se podrían añadir las siguientes consideraciones:

- Recompensar al agente por volar justo en el centro del agujero. Esta política incentivaría al agente a precisar sus movimientos, evitando el choque con la tubería y encontrándose en una altura óptima para poder anticipar sus movimientos futuros. +10 puntos.
- Penalización por cambios de altura innecesarios. En ocasiones puede que el agente decida realizar una acción random, lo cual podría resultar con un salto innecesario. Imponer una penalización por cambios de altura que no estén relacionados con la posición de las tuberías. Ej. Hacer dos saltos: el primero ponía el agente a la altura del agujero de la tubería, con lo que se esperaba que el siguiente movimiento no fuese saltar, sino quedarse quieto. -1 punto.

En resumen, mi propuesta de política consistiría en añadir algunas normas más a la política ya existente. El objetivo es enfocar al máximo el agente para que capte el comportamiento deseado. Añadiendo las 2 reglas que he propuesto, se está premiando la precisión de los movimientos y evitando los movimientos innecesarios. Los puntajes determinados como penalización no pueden ser muy altos, deben ser lo suficientemente fuertes como para desincentivar comportamientos no deseados, pero no tan altos como para imposibilitar la exploración del agente. Por este motivo, la recompensa por un movimiento preciso es superior a la penalización por un movimiento innecesario.