enero, 2024

Memoria Trabajo Sistemas Conexionistas

Alberto García Martín

VNiVERSiDAD D SALAMANCA

Máster en Sistemas Inteligentes
Universidad de Salamanca

Índice

1. Introducción

En este trabajo de la asignatura de Computación Neuroborrosa, se pedía la realización de un trabajo aplicando redes neuronales para resolver un problema. Se proponía que el alumno utilizara las herramientas que considerara oportunas para resolver el problema, siempre y cuando se utilizara una red neuronal. En mi caso, opté por un enfoque desafiante pero educativo: implementar una red neuronal desde cero en C++, utilizando solo la biblioteca estándar de C++ y ninguna biblioteca externa.

El proceso comenzó con un estudio de los fundamentos teóricos de las redes neuronales. Esto incluyó el estudio de conceptos como neuronas artificiales, capas, pesos, funciones de activación y gradientes. Una vez adquirido este conocimiento teórico, el siguiente paso fue desarrollar una biblioteca simple y modular para definir y entrenar redes neuronales.

Esta biblioteca está formada por tres clases principales: `Neuron`, `Layer` y `MLP`. La clase `Neuron` representa una neurona artificial, con un conjunto de entradas y pesos. La clase `Layer` representa una capa de neuronas, con un conjunto de neuronas y una función de activación. La clase `MLP` representa una red neuronal multicapa, con un conjunto de capas.

Para verificar el correcto funcionamiento de la biblioteca, además de establecer pruebas unitarias, se implementó una red neuronal para resolver el problema clásico de clasificación del conjunto de datos *Iris* de Fisher[2]. Además, con el objetivo de probar la biblioteca en un problema más complejo, se implementó una red neuronal para resolver un problema de clasificación de actividades físicas empleando un conjunto de datos de *smartwatches* recopilado por Fuller et al.[3]

Por último, para asegurar que los resultados devueltos por las redes neuronales eran correctos, se implementaron las mismas redes neuronales utilizando la biblioteca *PyTorch* y se compararon los resultados. Los resultados obtenidos por ambas bibliotecas fueron similares, lo que indica que la biblioteca desarrollada funciona correctamente dentro de sus limitaciones, al no tener el amplio abanico de optimizaciones que tienen las bibliotecas de aprendizaje automático más populares.

Tanto el código fuente de la biblioteca desarrollada como el código las redes neuronales implementadas se encuentran adjuntadas a este informe en la carpeta `mlp.cpp`. Las pruebas de comparación con *PyTorch* se encuentran en un cuaderno de *Jupyter* en la carpeta `mlp.py`.

2. Desarrollo de la biblioteca

Esta sección detalla el proceso de desarrollo de la biblioteca de redes neuronales. Se desglosarán las tres clases principales que la componen, explicando su funcionalidad, estructura y cómo interactúan entre sí para formar una red neuronal cohesiva

y funcional. Se explicarán las decisiones de diseño, los desafíos encontrados y las soluciones adoptadas para construir una biblioteca robusta y eficiente en C++.

2.1. Clase de neurona

La clase `Neuron` representa una neurona artificial. Esta clase es la unidad básica de una red neuronal, y posee como atributos privados el valor de salida de la neurona, el valor del gradiente, su valor de sesgo, un vector de pesos y otro de entradas, cuya longitud será igual para ambos. Además contienen un valor booleano para determinar si la generación de los pesos es determinista o no, con el objetivo de poder reproducir los resultados de las pruebas unitarias.

Al instanciar una neurona, se crea tanto el vector de entradas como el de pesos, con la longitud especificada en el constructor. El vector de pesos se inicializa siguiendo la distribución propuesta por He et al.[4] para la inicialización de pesos en redes neuronales. Esta distribución se basa en la inicialización de los pesos con una distribución normal con media 0 y desviación estándar $\sqrt{2/n}$, donde n es el número de entradas. Esta distribución se utiliza para reducir la probabilidad de que los gradientes de las neuronas se desvanezcan o exploten durante el entrenamiento, problema encontrado habitualmente durante el desarrollo de este proyecto.

Con el objetivo de prevenir una explosión de gradientes, también se implementó una función de *clipping* para limitar el valor de los gradientes. Esta función se aplica a cada gradiente de la neurona, y limita su valor entre un valor mínimo y máximo. En este caso, tras prueba y error, se estableció el valor mínimo a -10 y el máximo a 10.

A la hora de calcular la salida de la neurona, en esta clase tan solo se calcula el valor antes de aplicar la función de activación, es decir el producto escalar entre el vector de entradas y el de pesos, sumándole el valor del sesgo, que por defecto es 1. A este valor posteriormente se le aplica la función de activación desde la clase de la capa, ya que se pretende que la función de activación sea la misma para todas las neuronas de una capa.

También se han implementado métodos para guardar y cargar los pesos de una neurona en un fichero. Estas funciones se repetirán en las otras clases con el objetivo de poder guardar el estado de la red neuronal en un fichero y poder cargarlo posteriormente. Esto es útil para poder guardar una red previamente entrada y poder cargarla posteriormente para realizar predicciones. Cabe destacar que estas funciones no guardan la información sobre la estructura de la red neuronal, por lo que es necesario conocer la estructura de la red neuronal para poder cargarla correctamente.

2.2. Clase de capa

La clase `Layer` representa una capa de neuronas dentro de una red neuronal. Funciona como un contenedor para múltiples instancias de la clase `Neuron`, gestionando la interacción entre ellas y proporcionando funcionalidades clave para el procesamiento de datos a través de la red. Como atributos privados, esta clase contiene un vector de neuronas, la función de activación de la capa y su derivada.

Además contiene un valor booleano para determinar si a la salida de la capa, antes del paso por la función de activación, se le aplica normalización o no, esto puede ayudar a evitar que los gradientes exploten en redes con un alto número de neuronas.[1]

Para mejorar el rendimiento de la red neuronal en problemas de clasificación multiclase, se ha implementado la función *softmax*. Esta función comprime un vector de valores en un vector de probabilidades, de forma que la suma de todos los valores del vector resultante es 1. Esta función normalmente se utiliza en la última capa de la red neuronal, con el objetivo de obtener un vector de probabilidades que represente la probabilidad de que la entrada pertenezca a cada una de las clases.

Se ha implementado también un método para conectar una capa con la anterior, de forma que se puedan conectar todas las capas de la red neuronal. Este método actualiza las entradas de cada neurona de la capa con los valores de salida de la capa anterior, e inicializa los pesos de cada neurona de la capa. Esta función es imprescindible para poder crear una red neuronal multicapa, donde cada capa tenga propiedades diferentes.

2.3. Clase de red neuronal multicapa

La clase `MLP` representa una red neuronal multicapa, esta clase es la encargada de gestionar la interacción entre las capas de la red neuronal, y contiene la lógica principal para entrenar la red neuronal. Los atributos privados de esta clase son un vector de capas, el valor de la tasa de aprendizaje y una bandera para indicar si a la última capa de salida se le aplica la función *softmax*descrita anteriormente.

El constructor de esta clase se encarga de inicializar la estructura de la red neuronal, según un vector de enteros que indicará el número de neuronas de cada capa. También se puede inicializar una red neuronal vacía a la que se le pueden añadir capas posteriormente.

En esta clase se definen los métodos de entrenamiento e inferencia de la red neuronal. En el proceso de entrenamiento para cada dato de entrada se realiza una propagación hacia adelante (*feedforward*) y una propagación hacia atrás *backpropagation*. Este proceso se repite tantas veces como se defina en el parámetro de épocas que se le pasa a la función. La función de inferencia, o predicción, simplemente realiza una propagación hacia adelante y devuelve los valores de salida de la última capa.

El proceso de *feedforward* consiste en propagar los valores de entrada a través de la red neuronal, calculando el valor de salida de cada neurona y aplicando la función de activación de la capa. Este proceso se realiza en cada capa de la red neuronal, comenzando por la primera capa de entrada y terminando en la última capa de salida. El resultado de este proceso es un vector de valores de salida, que se utilizará para calcular el error de la red neuronal, o para realizar predicciones.

La función de *backpropagation* propaga el error de la red neuronal hacia atrás, calculando el gradiente de cada neurona y actualizando los pesos de cada neurona. Este proceso se realiza en cada capa de la red neuronal, comenzando por la última capa de salida y terminando en la primera capa de entrada. El resultado de este proceso es una red neuronal con los pesos actualizados, que se utilizará para realizar la siguiente propagación hacia adelante.

Aunque inicialmente el concepto de retropropagación puede parecer sencillo, su implementación es compleja, y es donde se encuentra la mayor parte de la complejidad de la biblioteca. Además es donde se han producido la mayor parte de los errores durante el desarrollo. Esto es debido a que los gradientes de las neuronas pueden sufrir de dos problemas: desvanecimiento o explosión. El desvanecimiento de gradientes se produce cuando los gradientes de las neuronas se vuelven cada vez más pequeños a medida que se propagan hacia atrás, lo que hace que los pesos de las neuronas no se actualicen correctamente. Por otro lado, la explosión de gradientes se produce cuando los gradientes de las neuronas se vuelven cada vez más grandes a medida que se propagan hacia atrás. Lo que hace que los pesos de las neuronas se actualicen demasiado rápido y la red neuronal no converja, o que directamente escapen del rango de representación de los números de punto flotante.

3. Evaluación de la biblioteca

Para la evaluación de la biblioteca durante el desarrollo se han empleado pruebas unitarias, mientras que una vez completada se han implementado dos redes neuronales para resolver dos problemas de clasificación, uno sobre el conjunto de datos *Iris* y otro sobre un conjunto de datos más complejo. En estas implementaciones se ha hecho uso de las funciones de activación definidas en `utils.cpp` para no tener que reimplementarlas constantemente, aquí se han definido las funciones sigmoide, tangente hiperbólica, ReLU e identidad.

3.1. Pruebas unitarias

Para la evaluación de la biblioteca durante el desarrollo se han implementado pruebas unitarias para cada una de las clases. Estas pruebas tienen como objetivo comprobar que las funciones principales de cada clase funcionan correctamente, y que los resultados devueltos por cada clase son los esperados. Para ello, en el caso de las pruebas sobre la clase de la neurona como las de la capa, tanto los valores de entrada como los pesos han sido establecidos manualmente, de forma que se puedan predecir

los resultados esperados. En el caso de la prueba para la red neuronal completa se ha inicializado una red neuronal con pesos deterministas para que aprenda a resolver la función XOR, llegando a resolverla de forma aceptable. El resultado de esta prueba se puede ver en la tabla 1. Esta red neuronal se ha diseñado con 2 capas ocultas de 4 neuronas cada una, con una función de activación ReLU en las capas ocultas y una función de activación sigmoide en la capa de salida.

Entrada	Predicción	Objetivo
0 0	0.0288306	0
0 1	0.999449	1
1 0	0.729016	1
1 1	0.144376	0

Tabla 1: Resultados de la prueba de la función XOR.

3.2. Clasificación del conjunto de datos Iris

Para poder utilizar el conjunto de datos *Iris*, primero se han tenido que preprocesar los datos. Para ello la columna con la clase de cada flor se ha transformado en un valor numérico. Posteriormente se ha barajado el conjunto de datos, empleando una semilla constante para poder comparar resultados entre ejecuciones. A continuación se ha dividido el conjunto de datos en dos subconjuntos, uno de entrenamiento, con el 80 % de los datos, y otro de test, con el 20 % restante. Finalmente el conjunto de entrenamiento se ha dividido en dos vectores, uno con las características de entrada y otro con la clase de cada flor codificada en un vector de 3 elementos, donde cada elemento representa una clase.

Tras múltiples iteraciones se ha alcanzado una estructura de red neuronal que resuelve el problema de forma aceptable. Esta red neuronal tiene una capa de entrada con 4 neuronas, dos capas ocultas con 10 neuronas cada una y una capa de salida con 3 neuronas. El número de capas ocultas y el número de neuronas de cada capa se han determinado mediante prueba y error, con una capa oculta no llegaba a aprender el problema, mientras que con más capas los resultados eran inconsistentes entre ejecuciones. La función de activación utilizada en las capas ocultas ha sido la ReLU, ya que otras funciones de activación como la sigmoide en ocasiones sufrían problemas de explosión de gradientes y se quedaban atascados en los valores límite del gradiente, mientras que la ReLU no sufría este problema. En la capa de salida se ha utilizado la función de activación softmax, ya que se trata de un problema de clasificación multiclase. El entrenamiento se ha realizado con una tasa de aprendizaje de 0.00001 y 10000 épocas, esto daba mejores resultados que una tasa de aprendizaje más alta y menos épocas, y un número de épocas más alto no mejoraba los resultados.

Para evaluar los resultados de la red neuronal se ha utilizado la matriz de confusión, que permite visualizar los resultados de la clasificación de forma más clara, además se ha calculado la exactitud. En la tabla 2 se puede ver la matriz de confusión obtenida por la red neuronal. En esta matriz se puede ver que la red neuronal

ha clasificado correctamente todas las flores de la clase 0, mientras que ha clasificado incorrectamente una flor de la clase 2 como la clase 1. La exactitud obtenida por la red neuronal ha sido del 96.67 %, lo que indica que la red neuronal ha aprendido correctamente el problema. En otras ejecuciones la exactitud ha variado ligeramente, pero por lo general siempre se ha mantenido alta. Además siempre ha identificado claramente las flores de la clase 0 (setosas), mientras que ha tenido más problemas para diferenciar las flores de la clase 1 y 2, que son más parecidas entre sí.

Clase	**0**	**1**	**2**
0	7	0	0
1	0	14	0
2	0	1	8

Tabla 2: Matriz de confusión de la red neuronal sobre el conjunto de datos Iris.

3.3. Clasificación de actividades físicas

El preprocesamiento de este conjunto de datos ha sido similar al del conjunto de datos *Iris*. Primero se ha transformado la columna de la clase en un valor numérico, posteriormente se ha barajado el conjunto de datos y se ha dividido en dos subconjuntos, uno de entrenamiento con el 80 % de los datos y otro de test con el 20 % restante. Finalmente el conjunto de entrenamiento se ha dividido en dos vectores, uno con las características de entrada y otro con la clase de cada actividad codificada en un vector de 6 elementos, donde cada elemento representa una clase.

En este caso no se ha conseguido una estructura de red neuronal que resuelva el problema de forma aceptable, o que llegue a aprender el problema. Se han probado múltiples estructuras de red neuronal, con diferentes funciones de activación, tasas de aprendizaje y número de épocas, pero en todos los casos la red neuronal no era capaz de aprender nada. En la tabla 3 se puede ver la matriz de confusión obtenida por la red neuronal con la última estructura probada. Se puede ver que la red es incapaz de aprender y que siempre clasifica todas las actividades como la primera clase, o como la última clase, obteniendo una exactitud del 26.62 %.

Sin embargo, como se verá en la siguiente sección, la misma red neuronal implementada en *PyTorch* tampoco es capaz de resolver este problema de clasificación usando los mismos hiperparámetros, para lograr una red neuronal que resuelva el problema se necesitan algoritmos de optimización más avanzados, que no se han implementado en esta biblioteca.

Clase	**0**	**1**	**2**	**3**	**4**	**5**
0	99	0	0	0	0	31
1	66	0	0	0	0	12
2	69	0	0	0	0	1
3	68	0	0	0	0	3
4	69	0	0	0	0	17
5	47	0	0	0	0	40

Tabla 3: Matriz de confusión sobre el conjunto de datos de actividades físicas.

4. Comparación con PyTorch

Para asegurar que los resultados obtenidos por la biblioteca son correctos, se han implementado las mismas redes neuronales utilizando la biblioteca *PyTorch* y se han comparado los resultados. Para ambos conjuntos de datos se ha realizado el mismo preprocesamiento realizado en la implementación en C++, pero esta vez utilizando las bibliotecas *Pandas* y *scikit-learn* para facilitar el proceso.

La red neuronal implementada para el conjunto de datos *Iris* es la misma que la implementada en C++, con una capa de entrada con 4 neuronas, dos capas ocultas con 10 neuronas cada una, y función de activación ReLU y una capa de salida con 3 neuronas con función de activación *softmax*. El resultado obtenido en el conjunto de validación ha sido muy similar al obtenido en la implementación en C++, con una exactitud del 93.33 %, y una matriz de confusión que se puede ver en la tabla 4.

Clase	**0**	**1**	**2**
0	10	0	0
1	0	7	2
2	0	0	11

Tabla 4: Matriz de confusión de la red para Iris implementada en PyTorch.

La red neuronal inicial implementada para el conjunto de datos de actividades físicas también tiene la misma estructura que la implementada en C++, con una capa de entrada con 6 neuronas, dos capas ocultas con 10 neuronas cada una, y función de activación ReLU y una capa de salida con 6 neuronas con función de activación *softmax*. En este caso se está utilizando una función básica de optimización, el descenso de gradiente estocástico. Como ocurre en la implementación en C++, la red neuronal no es capaz de aprender el problema, obteniendo una exactitud del 26.62 %, y una matriz de confusión que se puede ver en la tabla 5.

Este problema tiene fácil solución en *PyTorch*, ya que se pueden utilizar algoritmos de optimización más avanzados, como el algoritmo Adam[5]. Tan solo aplicando este algoritmo de optimización y manteniendo la misma estructura de red e hipérparámetros, se ha conseguido una red neuronal que logra aprender el problema, con una exactitud del 67.43 %, y una matriz de confusión que se puede ver en la tabla 6.

Clase	0	1	2	3	4	5
0	107	0	0	0	0	0
1	87	0	0	0	0	0
2	82	0	0	0	0	0
3	105	0	0	0	0	0
4	62	0	0	0	0	0
5	79	0	0	0	0	0

Tabla 5: Red para actividades físicas implementada en PyTorch con función SGD.

Clase	0	1	2	3	4	5
0	75	8	5	14	5	0
1	10	65	6	6	0	0
2	6	4	62	10	0	0
3	3	0	5	97	0	0
4	0	2	2	5	53	0
5	31	0	7	18	23	0

Tabla 6: Red para actividades físicas implementada en PyTorch con función Adam.

Por lo tanto, tras observar estos resultados, se puede concluir que la biblioteca desarrollada funciona correctamente dentro de sus limitaciones, al no tener las optimizaciones que tienen las bibliotecas de aprendizaje automático como *PyTorch*.

5. Conclusión

En este trabajo, se han implementado redes neuronales mediante una biblioteca completamente implementada en C++, diseñada desde cero y sin bibliotecas externas. Al evaluar el rendimiento y la funcionalidad de esta biblioteca sobre dos conjuntos de datos distintos relacionados con problemas de clasificación, y al compararla con una de las bibliotecas de aprendizaje profundo más consolidadas, como *PyTorch*, se ha observado la eficacia de esta implementación.

Además, el aprendizaje obtenido durante este proyecto ha superado todas las expectativas. No solo se ha construido una biblioteca funcional de redes neuronales, sino que también se ha adquirido una comprensión profunda de los mecanismos y las complejidades del aprendizaje profundo. A través de la superación de problemas inesperados, como los relacionados con la explosión y el desvanecimiento de gradientes, se ha adquirido una comprensión más profunda de los conceptos teóricos de las redes neuronales.

En resumen, este proyecto no solo ha culminado en el desarrollo de una biblioteca de red neuronal funcional y eficaz, sino que también ha servido como un viaje de aprendizaje.

Referencias

[1] Jimmy Lei Ba, Jamie Ryan Kiros y Geoffrey E. Hinton. Layer Normalization, 2016. arXiv: 1607.06450 [stat.ML].

[2] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: 10.24432/C56C76.

[3] Daniel Fuller. Replication Data for: Using machine learning methods to predict physical activity types with Apple Watch and Fitbit data using indirect calorimetry as the criterion. Versión V1, 2020. DOI: 10.7910/DVN/ZS2Z2J.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. En *2015 IEEE International Conference on Computer Vision (ICCV)*, páginas 1026-1034, 2015. DOI: 10.1109/ICCV.2015.123.

[5] Diederik P. Kingma y Jimmy Ba. Adam: A Method for Stochastic Optimization. En Yoshua Bengio y Yann LeCun, edición, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.