RED NEURONAL PYTHON PARTE 1

Optimización y Heurística

3º Curso de Ciencia e Ingeniería de Datos
Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

Aris Vazdekis Soria y Alejandra Ruiz de Adana Fleitas
13/11/2023

MEMORIA SOBRE EL PROYECTO DE RED NEURONAL

Este proyecto tiene como objetivo implementar una red neuronal para la clasificación de prendas de moda utilizando el conjunto de datos Fashion MNIST. Se emplea la biblioteca numpy para el manejo del programa con arrays, la scikit-learn, para la creación de la matriz de confusión, y herramientas de visualización como matplotlib para analizar el rendimiento del modelo.

Para el desarrollo hemos creado una red neuronal dotada de tres capas ocultas con tamaños de 64, 32 y 16 respectivamente. La capa de salida tiene 10 nodos correspondientes a las clases de prendas en Fashion MNIST. Se utiliza la función de activación ReLU y la inicialización de pesos con Xavier. Implementamos el algoritmo de optimización Adam para ajustar los pesos de la red ya que es el que mejor se ajusta a nuestro conjunto de datos. Se emplea la función de entropía cruzada para poder calcular la precisión del modelo durante el entrenamiento y formar la función de pérdida.

PROGRAMACIÓN

La **función ReLU** se define matematicamente como f(x) = max(0,x). Esto significa que devuelve 0 si el valor de entrada es negativo y devuelve el mismo valor de entrada cuando es positivo. Por otro lado, la **derivada de la función ReLU** es necesaria para el proceso de retropropagación en el entrenamiento de la neurona. Lo que nos dice es que si la derivada es positiva tendrá valor 1 y cuando sea negativa o sea 0 tendrá valor 0.

La **inicialización de Xavier** es una técnica para inicializar los pesos de una red neuronal de manera que se reduzca la varianza de las activaciones a lo largo de las capas. La fórmula de la inicialización de Xavier es:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

donde n_{in} es el número de unidades de la capa de entrada y n_{out} es el número de unidades de la capa salida. La idea detrás de esta inicialización es que al multiplicar los pesos inicializados por una distribución normal con media 0 y varianza $\frac{2}{n_{in} + n_{out}}$ se logre que la varianza de las activaciones sea aproximadamente la misma en todas las capas.
En la función “xavier_initialization” primero se genera una matriz de pesos con valores aleatorios extraídos de una distribución normal estándar y luego esta matriz se multiplica por $\sqrt{\frac{2}{n_{in} + n_{out}}}$, que es la fórmula de Xavier para escalar los valores.

Como **método de optimización** elegimos el método **ADAM.** Este método modifica las bases del algoritmo SGD al introducir conceptos como el momentum y la estabilidad de los promedios de los gradientes.

SGD Original:

$\theta_{t+1}^{SGD} = \theta_t - \alpha \nabla J(\theta_t)$

(donde α es la tasa de aprendizaje y $\nabla J(\theta_t)$ es el gradiente de la función de pérdida con respecto a los parámetros en el tiempo t).

Modificación con Momentum:

$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot \nabla J(\theta_t)$

$\theta_{t+1}^{Momentum} = \theta_t - \alpha \cdot m_t$

(donde m_t representa el momentum que incorpora promedios de los gradientes)

Incorporación de RMSprop en Adam:

$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla J(\theta_t))^2$

$\theta_{t+1}^{Adam} = \theta_t - \alpha \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$

Esta versión completa de Adam incluye el RMSprop en donde el v_t es el momento del segundo orden y tanto el v_t como el m_t son versiones sesgadas corregidas de los momentos. El término ϵ se añade para evitar la división por 0 rompiendo el funcionamiento. El método Adam combina la estabilidad de los promedios de los gradientes para que las actualizaciones tengan inercia y utiliza el historial de gradientes adaptativo del RMSprop para ayudarse a superar problemas donde la magnitud de los gradientes varia ampliamente entre parámetros, de manera que la tasa de aprendizaje evita que sea demasiado pequeña o demasiado grande.

Una **fórmula** que utilizamos para calcular la pérdida en el error fue la **entropía cruzada**. Esta medida se emplea para evaluar cuánto se ajustan las probabilidades predichas por el modelo a las etiquetas reales. La formula de la entropía cruzada es:

$$H(y,\widehat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} log(\widehat{y}_{ij})$$

N = Número total de muestras
C = Número de clases
y_{ij} = Es 1 si la muestra i pertenece a la clase j y 0 de lo contrario (Etiqueta real)
$\widehat{y}_{ij}$ = Es la probabilidad predicha por el modelo de que la muestra i pertenezca a la clase j.

En el programa la entropía cruzada se calcula con "*output_layer_output[range(len(y_train)), y_train]*", donde se selecciona las probabilidades predichas correspondientes a las clases reales de las muestras. Luego, se aplica el logaritmo negativo (−log) y se toma la media para obtener la pérdida promedio.

Explicación del backpropagation en el codigo

1) Cálculo de salidas
    - Se calcula las salidas de cada capa de la red neuronal utilizando las funciones de activación ReLU en las capas ocultas

```
hidden_layer_input1 = np.dot(X_train, weights_input_hidden1)
+ bias_hidden1
    hidden_layer_output1 = relu(hidden_layer_input1)
…
```

    - Calcula la salida final utilizando la función softmax para obtener probabilidades

```
output_layer_input = np.dot(hidden_layer_output3,
weights_hidden3_output) + bias_output
    exp_scores = np.exp(output_layer_input)
      output_layer_output = exp_scores / np.sum(exp_scores,
axis=1, keepdims=True)
```

Primero se calcula la entrada a la capa con “output_layer_input” y después se aplica la función softmax. En “exp_scores” se calcula el exponencial de cada elemento en la entrada de la capa salida y en “output_layer_output” se aplica la función softmax normalizando las exponenciales para obtener probabilidades.

2) Cálculo de la pérdida:
Calculamos la perdida usando la entropia cruzada

```
# Calcular la pérdida
        loss  =  -np.log(output_layer_output[range(len(y_train)),
y_train]).mean()
```

3) Backward pass:

```
# Backward pass
    error = output_layer_output
    error[range(len(y_train)), y_train] -= 1

    d_output = error
    error_hidden3 = d_output.dot(weights_hidden3_output.T)
                        d_hidden3      =      error_hidden3      *
relu_derivative(hidden_layer_output3)

    error_hidden2 = d_hidden3.dot(weights_hidden2_hidden3.T)
                        d_hidden2      =      error_hidden2      *
relu_derivative(hidden_layer_output2)

    error_hidden1 = d_hidden2.dot(weights_hidden1_hidden2.T)
                        d_hidden1      =      error_hidden1      *
relu_derivative(hidden_layer_output1)
```

- Calcula el gradiente de la pérdida con respecto a la salida final
  - “error” calcula el error en la salida final.
  - “d_output” almacena este error.

- Propagación hacia atrás a través de la red:
  - “d_hidden3”, “d_hidden2”, y “d_hidden1” calculan los errores en las capas ocultas utilizando la regla de la cadena y la derivada de la función de activación ReLU.

4) Actualización de pesos:

```
 # Actualiza los pesos utilizando el algoritmo Adam
     weights_hidden3_output, mt_hidden_output, vt_hidden_output =
adam_optimizer(weights_hidden3_output,
hidden_layer_output3.T.dot(d_output) * learning_rate,
 mt_hidden_output, vt_hidden_output, t=epoch+1
    )
…
```

Aquí, “adam_optimizer” toma los pesos, los gradientes con respecto a esos pesos, y los momentos de primer y segundo orden para actualizar los pesos utilizando el algoritmo Adam.

En tu código, los momentos de primer orden (“mt”) y de segundo orden (“vt”) se actualizan y se utilizan para adaptar la tasa de aprendizaje de manera eficiente.

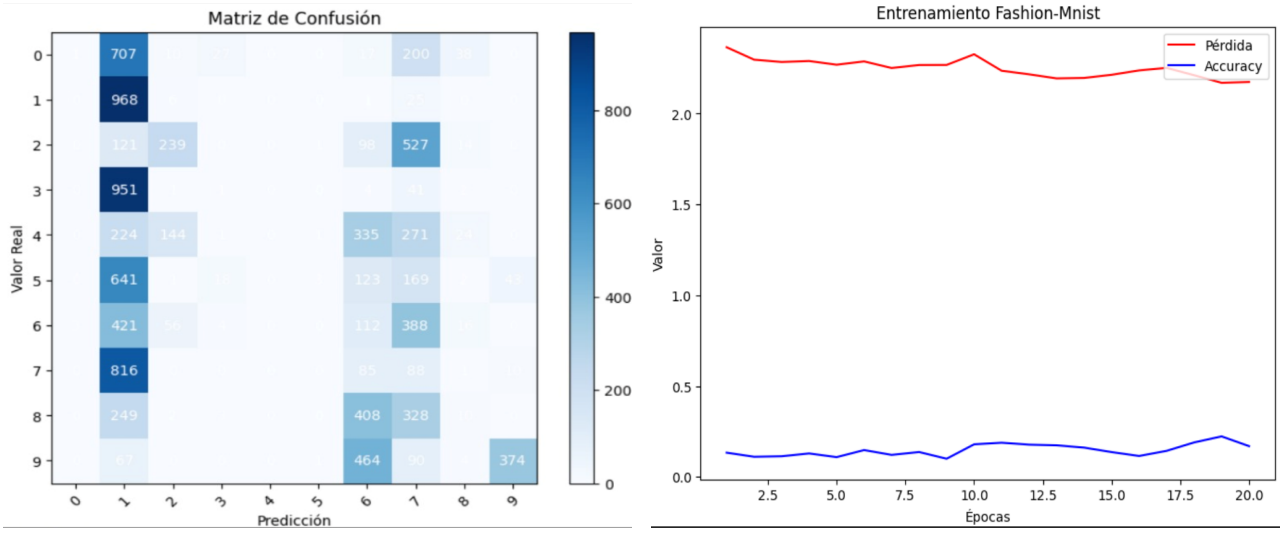
EXPERIMENTACIÓN

Los datos de entrenamiento y prueba se cargan desde archivos CSV y, como parte del preprocesamiento, normalizamos los valores de píxeles dividiéndolos por 255. Esta normalización tiene como objetivo escalar los valores de cada píxel al rango de 0 a 1. Este proceso es crucial, ya que ayuda al modelo a converger más rápidamente durante el entrenamiento al mantener los valores en una escala manejable.

La elección de una tasa de aprendizaje de 0.0001 se basa en experimentos previos. Tasas de aprendizaje más altas tendían a provocar que el modelo clasificara todas las imágenes en una sola clase, mientras que tasas más bajas resultaban en un sobreajuste. La tasa seleccionada busca un equilibrio que permita un aprendizaje efectivo sin perder generalización.

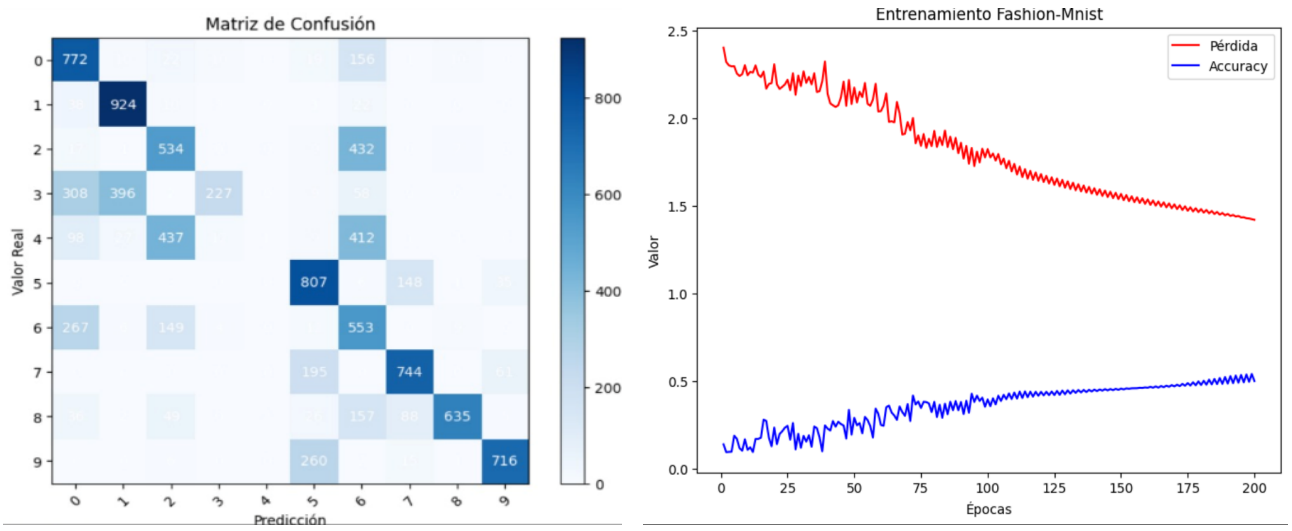
En la evaluación del modelo, hemos implementado la matriz de confusión para examinar su comportamiento en cada ejecución. Además, hemos incorporado una función de pérdida que nos proporciona información sobre la mejora potencial del modelo.

En las figuras de prueba 1, 2, 3 y 4, hemos explorado diferentes números de épocas (20, 200, 1000 y 2000, respectivamente). Estas pruebas buscan analizar cómo el modelo se desempeña con variaciones en el número de épocas, permitiéndonos evaluar su capacidad para aprender patrones a lo largo del tiempo.



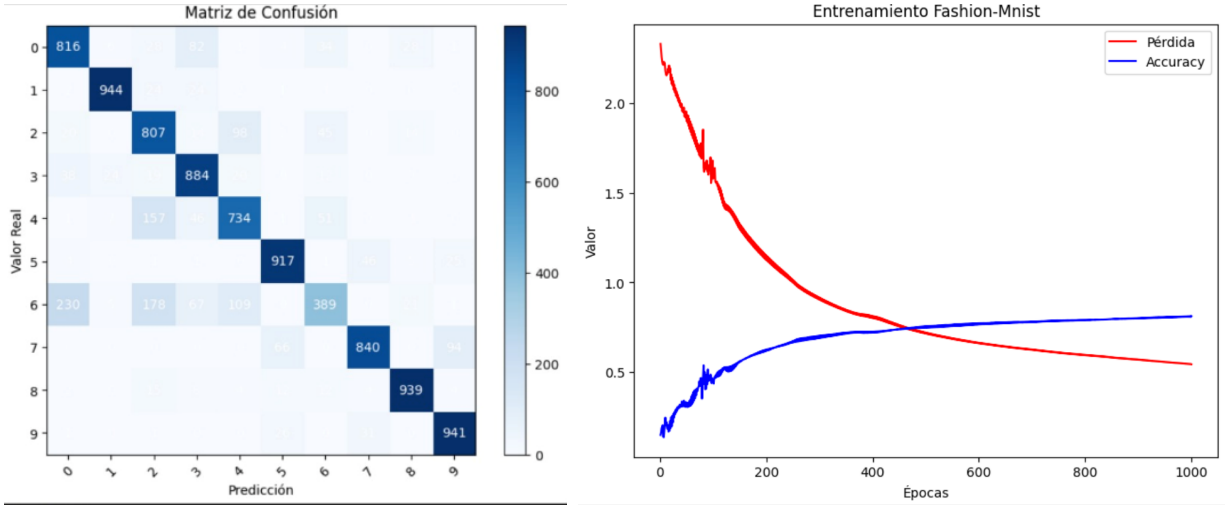
Figuras prueba 1.

Observamos como el modelo no es capaz de predecir correctamente en cada grupo de los 10 tipos de prendas del dataset fashion mnist. La función de pérdida de un valor muy elevado, por encima del 2.2,mientras que el accuracy nos da un valor muy bajo, por debajo del 0.5, lo que nos indica que el modelo carece de precisión. También vemos como la funcion de perdida no tiene una curvatura estable, lo que nos indica el factor de aleatoriedad provocado por una insuficiencia de entrenamiento.



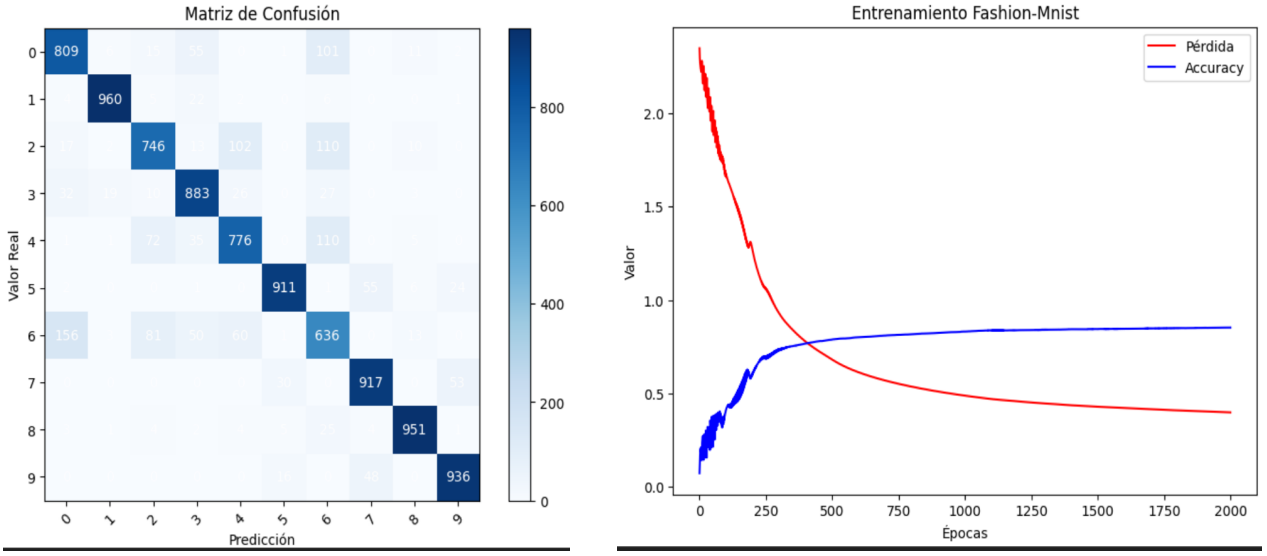
Figuras prueba 2.

En el segundo modelo vemos como mejora el modelo 1, puesto que ha realizado considerablemente más epoch en su entrenamiento. La mayoría de las clases se han clasificado correctamente exceptuando la clase 4 en la cual la tasa predictiva es pésima. Se ha reducido el valor de la función de pérdida por debajo de 1.2 y el accuracy ha aumentado por encima del 0.5 pero aun así sigue siendo un valor relativamente bajo.



Figuras prueba 3.

Vemos como con 1000 epoch el modelo clasifica con precisión cada imagen siendo una clara mejoría respecto al modelo 2. La función de pérdida se sitúa por encima del 0.5 y el accuracy se sitúa cerca del 1, lo que es un valor bastante aceptable. Vemos como la pendiente de la función de pérdida nos muestra como probablemente con un número mayor de epoch, el modelo podría disminuir su función de pérdida.



Figuras prueba 4.

Finalmente este último entrenamiento en 2000 epoch nos muestra como hay una ligera mejoría frente a las figuras número 4. La clasificación del modelo es lo suficientemente precisa como para dar el modelo como válido con una ligera mejora de clasificación de un 3% respecto a las 1000 epoch. La función de pérdida nos da un valor por debajo del 0.5 y el accuracy un valor rozando el 1, así que consideramos válido este entrenamiento.

CONCLUSIONES

El proyecto presentó un desafío en términos de implementación y comprensión de los algoritmos de aprendizaje profundo. En una primera versión utilizamos una inicialización de pesos con valores completamente aleatorios. Para evitar que los gradientes en cada capa no sean demasiado pequeños o demasiado grandes por la variación de los valores aleatorios, mejoramos la red con la inicialización de pesos con Xavier. La inicialización de pesos con Xavier consiste en establecer los pesos iniciales de manera que tengan una varianza apropiada. Esto ayuda a mantener la propagación del gradiente en un rango que facilita el entrenamiento.

Luego el modelo sufrió de una aleatorización de las predicciones, en donde el fallo radicaba en que no estamos aplicando una función de activación ReLu, utilizando en un principio una función sigmoidal y su derivada. Cuando se usa una función sigmoidal y su derivada en combinación con la inicialización de pesos aleatorios, nos enfrentamos a un problema de desvanecimiento del gradiente. Esto significa que para entradas muy grandes o muy pequeñas, la derivada se volvía casi cero. Esto hacía que los gradientes fueran demasiado pequeños para propagarse efectivamente a través de la red durante el entrenamiento.

A pesar de las dificultades iniciales, se logró entrenar un modelo capaz de clasificar el conjunto fashion mnist con bastante precisión.

LÍNEAS FUTURAS

Para futuras mejoras, se podría explorar la posibilidad de ajustar la arquitectura de la red, probar diferentes funciones de activación o implementar técnicas de regularización. También se deberían calcular las derivadas y el gradiente automáticamente. Además, se podría considerar la aplicación de técnicas de aumento de datos para mejorar la generalización del modelo, como la rotacion de las imagenes, cambios de color, etc.

RECURSOS UTILIZADOS

La estructura y contenido de la memoria se han desarrollado siguiendo las pautas del proyecto de red neuronal. Entre los recursos utilizados encontramos como entorno de desarrollo la aplicación Visual Studio Code. No se utilizó ninguna herramienta de control de versiones y como herramienta de documentación se utilizó Documentos de Google.