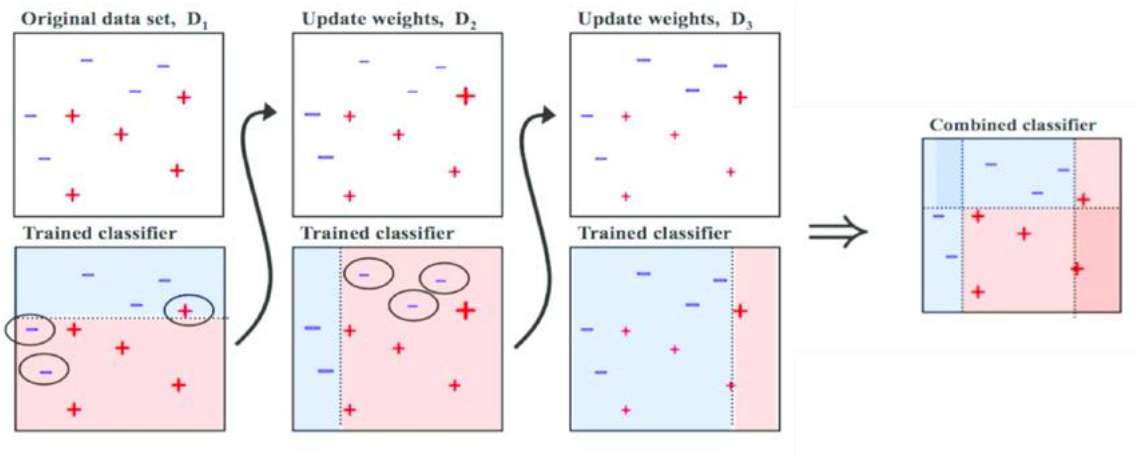
SISTEMAS INTELIGENTES

Práctica 2: Visión artificial y aprendizaje



Paula Gálvez Romero de Ávila 50509119S

Tabla de contenidos

Introducción

En esta práctica, nos adentramos en el mundo del aprendizaje supervisado, una rama clave del aprendizaje automático, aplicándolo al desafiante y conocido problema del reconocimiento automático de números manuscritos. Nuestro enfoque se centra en las cifras del 0 al 9, utilizando la reconocida base de datos MNIST, un punto de referencia estándar en el campo del aprendizaje automático.

Este proyecto no solo nos sumerge en el reconocimiento de patrones y el procesamiento de imágenes, sino que también nos ofrece la oportunidad de explorar el trabajo con píxeles en situaciones prácticas de dificultad moderada.

Para ello, emplearemos la librería de machine learning scikit-learn, una herramienta poderosa y versátil para configurar modelos de clasificación y regresión. Scikit-learn destaca por su amplia gama de técnicas en aprendizaje automático, permitiéndonos experimentar con diversos enfoques y métodos.

La práctica se estructura en dos partes fundamentales:

- Desarrollo de un Clasificador Multiclase Adaboost: En la primera parte, nos dedicaremos a programar en Python un clasificador multiclase Adaboost desde cero. Este enfoque nos permitirá obtener una comprensión profunda de los mecanismos internos y los desafíos del reconocimiento de cifras manuscritas.
- Implementación con Scikit-learn y Keras: La segunda parte del proyecto implica el uso de scikit-learn para implementar el clasificador multiclase Adaboost. Además, exploraremos el uso de un perceptrón multicapa (MLP) mediante la librería Keras de TensorFlow. Como tarea opcional también con Keras, de nuevo, se implementará un ejemplo de Red Neuronal Convencional (CNN). Este parte de la práctica nos permitirá comparar la efectividad de distintas técnicas y herramientas en la solución del mismo problema.

Al finalizar esta práctica, habremos no solo desarrollado habilidades técnicas valiosas en el campo del aprendizaje automático y el procesamiento de imágenes, sino también adquirido una comprensión más profunda de cómo se pueden aplicar estas técnicas en problemas reales y de actualidad.

Se pretende, sobre todo, no solo implementar lo comentado si no, experimentar en cada tarea todas las posibilidades para explotar de la mejor medida la eficacia de cada técnica para el contexto de esta práctica.

El Conjunto de Datos MNIST

El conjunto de datos MNIST (Modified National Institute of Standards and Technology) es fundamental en el campo del aprendizaje automático y el reconocimiento de patrones.

Compuesto por 70,000 imágenes en blanco y negro de dígitos manuscritos (60,000 para entrenamiento y 10,000 para pruebas), MNIST sirve como un banco de pruebas estándar para algoritmos de aprendizaje automático.

Características Clave del MNIST:

Dimensiones de las Imágenes: Cada imagen en MNIST tiene dimensiones de 28x28 píxeles, proporcionando una resolución suficiente para el reconocimiento de patrones mientras se mantiene la eficiencia computacional.

Etiquetado: Cada imagen está etiquetada con el dígito correspondiente (0-9), lo que facilita su uso en aprendizaje supervisado.

Diversidad y Realismo: Las imágenes en MNIST provienen de una variedad de escrituras manuscritas, reflejando así una gama realista de variaciones estilísticas y dificultades en el reconocimiento.

Importancia del MNIST en el Aprendizaje Automático:

Punto de Referencia: MNIST ha servido como un punto de referencia estándar para evaluar y comparar el rendimiento de los algoritmos de aprendizaje automático, particularmente en el reconocimiento de imágenes.

Accesibilidad: La simplicidad y accesibilidad del conjunto de datos lo han convertido en una herramienta educativa esencial para aquellos que se inician en el aprendizaje automático.

Innovación y Desarrollo: A lo largo de los años, MNIST ha impulsado la innovación y ha sido un catalizador para el desarrollo de técnicas avanzadas de procesamiento de imágenes y reconocimiento de patrones.

En resumen, el MNIST no es solo un conjunto de datos, sino una piedra angular en el aprendizaje automático, ofreciendo un terreno fértil para la experimentación, el aprendizaje y el avance tecnológico en el campo del reconocimiento de patrones y la inteligencia artificial.

Organización de la práctica

A lo largo de esta práctica he cambiado varias veces la organización de las tareas.

Comencé haciendo una función para cada tarea. Conforme fui avanzando vi que podía reutilizar código, llamando dentro de unas tareas a otras, pero me gustaría explicar como ha sido la entrega final para quedarme tranquila y evitar confusiones.

En el main he dejado líneas comentadas, para que se viese como sería llamar a las funciones de manera individual, y hago inciso de que no recomiendo hacer todo el main con todo descomentado porque habría repetividad de código y tardaría un tiempo en ejecutar toda la práctica.

Orden del main:

1. tarea1C, para visualizar la gráfica de cada dígito tras su entrenamiento (tarea1A_y_1B) bajo varios valores de T y de A.
2. tarea2B() para realizar la comparación que se solicita, invoca a las tareas 1D, 2A y 2C, salen las trazas de cada una y compara sus resultados devolviéndolos en una gráfica. Esta me chirriaba un poco, pero quería incorporar a la gráfica comparativa los resultados de la 2C, ya que me parecía muy interesante hacerlo, por lo que yo las habría cambiado de orden. Igualmente creo que lo he explicado bien en la memoria, y en el código se ve claro.
3. tarea2D, el preceptrón multicapa (MLP)
4. tarea 2E, la red neuronal convuncional (CNN)

A mi parecer, como ya he dicho, tras muchos cambios, ha sido la mejor manera de estructurar la práctica para estudiar y analizar las soluciones.

Recomiendo ir comentando las tareas y ejecutándolas 1 a 1 (las del main)

Apunte: He notado que si se compila toda la práctica de golpe hay alguna tarea en concreto la del MLP y la CNN, que pueden tardar más supongo que será por desgaste computacional. Yo la memoria como iba experimentando cosas, iba compilando tarea a tarea, me gustaría aclararlo para evitar equivocaciones.

Tarea 1A: Implementación de las clases Adaboost y DecisionStump.

– Explicación del Algoritmo de Adaboost

Primero explicaré el algoritmo AdaBoost en general, y luego detallaré cómo lo he implementado en las clases DecisionStump y Adaboost.

Algoritmo AdaBoost

AdaBoost (Adaptive Boosting) es un algoritmo de aprendizaje automático que se utiliza para mejorar el rendimiento de otros sistemas de aprendizaje.

Es un método de ensamblaje, lo que significa que combina múltiples clasificadores "débiles" para crear un clasificador "fuerte". Cada clasificador débil se entrena de manera secuencial.

Los clasificadores débiles son modelos simples que hacen predicciones ligeramente mejores que adivinar al azar. Por ejemplo, un clasificador débil podría ser un pequeño árbol de decisión. Los clasificadores débiles son usualmente simples y tienen un rendimiento apenas mejor que el azar.

AdaBoost ajusta iterativamente los pesos de las observaciones en los datos de entrenamiento, dándole más importancia a las observaciones que fueron clasificadas incorrectamente en las iteraciones anteriores. La idea es mejorar continuamente el modelo, enfocándose en los casos más difíciles.

Voy a destacar el concepto de peso ya que a mí me creo confusión. No es lo mismo el peso de los datos que el peso de los clasificadores:

El peso de los datos: Durante el entrenamiento si un clasificador comete errores, los datos que fueron clasificados incorrectamente reciben un mayor peso en la siguiente iteración, lo que significa que el próximo clasificador se enfocará más en esos datos difíciles, intentando corregir esos errores.

El peso de los clasificadores: Una vez que un clasificador ha sido entrenado y evaluado, se le asigna un peso basado en su precisión. Aquí, los clasificadores más precisos reciben pesos más altos. Esto significa que sus predicciones tendrán más influencia en la decisión final del modelo combinado.

¿Cómo Funciona AdaBoost?

Inicialización: A cada observación del conjunto de datos del entrenamiento se le asigna con un peso inicial igual. Esto significa que, al principio, todos los datos son igualmente importantes para el algoritmo.

Entrenamiento Iterativo de Clasificadores:

En cada iteración, se entrena un nuevo clasificador débil (DecisionStump).

Este clasificador se enfoca en los datos que fueron mal clasificados en las rondas anteriores, ajustando sus pesos para darles más importancia.

El clasificador débil se añade al modelo final, con un peso que depende de su precisión. Los clasificadores más precisos tienen más peso.

Actualización de Pesos:

Después de cada iteración, AdaBoost aumenta los pesos de los datos mal clasificados. Esto significa que en la siguiente iteración, el nuevo clasificador se centrará más en esos datos difíciles.

Por otro lado, reduce los pesos de los datos correctamente clasificados.

Combinación de Clasificadores:

El proceso continúa durante un número predefinido de iteraciones o hasta que se alcanza un nivel de precisión deseado.

Al final, AdaBoost combina todos los clasificadores débiles, cada uno con su peso correspondiente, para formar el modelo final.

En resumen, AdaBoost es un algoritmo potente que mejora sistemáticamente la precisión de los modelos de aprendizaje automático al enfocarse en los datos más difíciles y combinar múltiples clasificadores débiles (que pueden ser de diferentes tipos) en un modelo robusto y preciso, lo que lo hace ventajoso para la mejora del rendimiento.

Pseudocódigo

Algorithm 1 Adaboost

procedure ADABOOST(X, Y)
  $D_1(i) = 1/N$ ▷ Indica como de difcil es de clasificar cada punto i
  for $t = 1 \rightarrow T$ **do** ▷ T es el nmero de clasificadores dbiles a usar
    Entrenar h_t teniendo en cuenta D_t
    Start
      for $k = 1 \rightarrow A$ **do** ▷ A = num. de pruebas aleatorias
        F_p= generaClasificadorDébilAlAzar()
        $\epsilon_t = P_{D_t}(h_t(x_i) \neq y_i) \rightarrow \epsilon_{t,k} = \sum_{i=1}^{N} D_t(i) \cdot (F_k(x) \neq y(x))$
      return $< F_p | \min(\epsilon_{t,k}) >$
    End
    Del h_t anterior obtener su valor de confianza $\alpha_t \in \mathbb{R}$
    Start
      $\alpha_t = 1/2 \log_2 \left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
    End
    Actualizar D_{t+1}
    Start
      $D_{t+1} = \frac{D_t(i) \cdot e^{-\alpha_t \cdot y_i h_t(x_i)}}{Z_t}$
      $Z_t = \sum_i D_t(i)$
    End
  return $H(x) = sign(\sum_t \alpha_t \cdot h_t(x))$

Aunque tengo que decir que lo que más me ha ayudado a saber hacerlo es la descripción de los comentarios del enunciado de la práctica, ya que a partir de ahí he podido seguir la "plantilla" y los comentarios, y he podido sacar bien el código.

Implementación del Código

Clase DecisionStump

La clase DecisionStump es una implementación de un clasificador débil. En AdaBoost, un "stump" es un árbol de decisión muy simple, que consta de un único nodo de decisión y dos hojas. Este clasificador toma una sola característica y realiza una predicción basada en un umbral.

- Inicialización: Elige aleatoriamente una característica, un umbral y una polaridad que puede ser 1 o -1.
- Predicción: Clasifica los datos en base a si están por encima o por debajo del umbral, usando la polaridad para decidir cómo asignar las etiquetas.

Clase Adaboost

La clase Adaboost implementa el algoritmo AdaBoost.

- Inicialización: Recibe por parámetro el número de clasificadores débiles (T) a entrenar y el número de intentos (A) y los asigna a la clase.
- Método fit:
  - Inicializa los pesos uniformemente.
  - Itera sobre el número de clasificadores a entrenar.
  - Para cada iteración, crea y entrena múltiples DecisionStumps, seleccionando el mejor basado en el error mínimo.
  - Calcula el valor de alpha para el mejor DecisionStump.
  - Actualiza los pesos de las muestras, dando más peso a las mal clasificadas.
  - Añade el mejor DecisionStump a la lista de clasificadores.
- Método predict:
- Combina las predicciones de todos los DecisionStumps entrenados, ponderados por sus respectivos valores de alpha.
  La predicción final es el signo de la suma ponderada de las predicciones, aunque ya se mencionará más adelante que esto solo se devolverá en el caso del adaboost binario, cuando se quiera implementar o a invocar el multiclase, devolverá directamente las predicciones sin hacer el np.sign.

Tarea 1B: Mostrar resultados del clasificador Adaboost

En esta tarea se pretende probar el rendimiento del clasificador AdaBoost entrenándolo para distinguir un dígito específico (0-9) del conjunto de datos MNIST del resto.

Parámetros:

- digito: El dígito específico (0-9) que se quiere clasificar.
- T: El número de clasificadores débiles (DecisionStump) a utilizar en AdaBoost.
- A: El número de intentos para encontrar el mejor DecisionStump en cada iteración de AdaBoost.
- aux: Es solamente una variable auxiliar, para indicar que según dónde se invoque a esta tarea, se retornará una cosa u otra, para simplificar código.
- verbose: Parámetro opcional que, si se establece en True, imprime información adicional durante el entrenamiento.

Implementación:

Carga de Datos de MNIST: Utiliza mnist.load_data() para cargar los conjuntos de entrenamiento y prueba de MNIST.

Preprocesamiento: Aplana las imágenes de MNIST (de 2D a 1D) y normaliza los valores de píxeles (dividiéndolos por 255), es decir, los valores de los píxeles suelen estar en el rango de 0 a 255, sobretodo para imágenes en escala de grises donde 0 representa el negro y 255 representa el blanco. Normalizar estos valores implica dividir cada valor de píxel por 255 (el valor máximo posible). Esto convierte todos los píxeles en valores entre 0 y 1.

Filtrado de Datos: Modifica las etiquetas (y_train y y_test) para reflejar si cada imagen corresponde (1) o no (-1) al dígito seleccionado.

Balancear los datos y equilibrar el peso: Este paso hay varias formas de hacerlo, pero yo he utilizado la función SMOTE de la librería imblearn.over_sampling.

¿Por qué SMOTE()?

Conozco el uso de este método para balancear datos, debido a que recientemente hice un cursillo de un día en la empresa de Teralco, en el que se centró en el análisis de datos y su uso para modelos de entrenamiento en relación con la Inteligencia Artificial.

Este curso fue breve, solamente se estudió un poco el análisis de datos, el uso las gráficas y otros métodos gráficos y otros modelos de entrenamiento como el RandomForestClassifier.

SMOTE es una técnica de balanceo de clases utilizada en el procesamiento de datos para abordar el problema del desequilibrio de clases. Funciona generando ejemplos sintéticos de la clase minoritaria en lugar de simplemente duplicar los ejemplos existentes.

1. Selecciona una muestra de la clase minoritaria.
2. Encuentra sus k vecinos más cercanos (k-NN) en el conjunto de datos.
3. Selecciona aleatoriamente uno de estos vecinos y utiliza una combinación lineal entre la muestra y el vecino seleccionado para crear una nueva muestra sintética.

Esta técnica ayuda a crear un conjunto de datos más equilibrado, lo que puede mejorar el rendimiento del modelo en problemas de clasificación desbalanceada.

Inicialización del Clasificador AdaBoost: Crea una instancia de Adaboost con los parámetros T y A.

Entrenamiento del Clasificador: Entrena el clasificador AdaBoost con X_train y y_train_digito y mide el tiempo de entrenamiento para análisis posterior.

Predicción y Evaluación: Utiliza el clasificador entrenado para predecir etiquetas tanto en el conjunto de entrenamiento como en el de prueba, y calcula la precisión (tasa de acierto) en ambos conjuntos.

Mostrar la matriz de confusión: herramienta utilizada en aprendizaje automático para evaluar el rendimiento de un modelo de clasificación. Consiste en una tabla que muestra la comparación entre las etiquetas reales y las predicciones hechas por el modelo. La matriz de confusión típicamente incluye:

- Verdaderos Positivos (VP): Casos correctamente clasificados como positivos.
- Falsos Positivos (FP): Casos incorrectamente clasificados como positivos.
- Verdaderos Negativos (VN): Casos correctamente clasificados como negativos.
- Falsos Negativos (FN): Casos incorrectamente clasificados como negativos.

La matriz de confusión permite calcular métricas de rendimiento como precisión, recall, F1-score, entre otras, y proporciona una visión clara de en qué aspectos el modelo es fuerte o débil.

Impresión de Resultados: Muestra por pantalla la precisión obtenida en los conjuntos de entrenamiento y prueba, y el tiempo total de entrenamiento. Si verbose es True, imprime detalles adicionales como ya se había mencionado sobre cada clasificador en lista_clasificadores.

Retorno de Resultados: La función devuelve la precisión de entrenamiento, la precisión de prueba y el tiempo total de entrenamiento, para el análisis de la siguiente tarea.

Traza de ejecución - tarea1B(3, 10, 20, aux=True, verbose=True)

```
Entrenando clasificador Adaboost para el dígito 3, T=10, A=5
Tasas acierto (train, test) y tiempo: 71.19%, 63.23%, 0.099 s
Matriz de Confusión:
[[5539 3451]
 [ 226  784]]
Entrenando clasificador Adaboost para el dígito 3, T=20, A=10
Tasas acierto (train, test) y tiempo: 86.16%, 83.82%, 0.434 s
Matriz de Confusión:
[[7517 1473]
 [ 145  865]]
Entrenando clasificador Adaboost para el dígito 3, T=30, A=20
Tasas acierto (train, test) y tiempo: 89.25%, 88.05%, 1.090 s
Matriz de Confusión:
[[7919 1071]
 [ 124  886]]
Entrenando clasificador Adaboost para el dígito 3, T=40, A=30
Tasas acierto (train, test) y tiempo: 89.92%, 88.49%, 2.139 s
Matriz de Confusión:
[[7926 1064]
 [  87  923]]
Para el digito 3
El modelo tiene una precision de entrenamiento de media0.841325716089031
El modelo tiene una precision de test de media 0.808975
En el tiempo medio de 0.9407176971435547
```

Tarea 1C: Ajuste óptimo de T y A

El objetivo es determinar cómo los parámetros T y A afectan la tasa de acierto y el tiempo de entrenamiento del clasificador AdaBoost en el conjunto de datos MNIST, para cada dígito, y encontrar una combinación de estos parámetros que maximice la precisión manteniendo un producto T x A igual o inferior a 900.

Esta tarea se realiza para un solo dígito, pero a pesar de que el concepto multiclase se realiza en la siguiente tarea, a mi me ha parecido buena idea para analizar bien cada dígito hacer interesante esta tarea.

Para cada dígito, esta tarea tenía que hacer para varios T X A entrenar el modelo. La forma de entrenar consiste en llamar para cada iteración, para cada T x A (recorriendo los dos vectores con la útil función de Python zip()), invocar a la función tarea_1A_y_1B(digito, T, A, aux=True), de esta manera, a cada dígito lo estudiaré según su entrenamiento pasando por diferentes valores de T y de A.

Aux=True, porque a partir de este entrenamiento genero una gráfica para cada dígito en una misma figura, por lo al hacer aux=True, la tarea_1A_y_1B() me devolverá en este caso, la tasa de acierto de entrenamiento, la tasa de entrenamiento del test y el tiempo, y a partir de los datos recogidos generó la figura con gráficas ya mencionada.

Metodología

Realizar experimentos variando los valores de T y A para observar cómo afectan la precisión y el tiempo de entrenamiento.

Generación de Gráficas: Utilizar Matplotlib para crear gráficas que muestren la relación entre la tasa de acierto y el tiempo de entrenamiento para diferentes valores de T y A.

Diseño de Gráficas:

- Eje horizontal: Producto de T x A.
- Eje vertical izquierdo: Tasa de acierto (precisión), representada en colores rojo y naranja para entrenamiento y prueba, respectivamente.
- Eje vertical derecho: Tiempo de entrenamiento, representado en color azul.

Analizar los resultados para deducir los efectos de variar T y A.

Consideraciones para la Elección de T y A

- T (Número de Clasificadores): Un valor más alto de T generalmente mejora la precisión pero aumenta el tiempo de entrenamiento.
- A (Número de Intentos): Un valor más alto de A aumenta las posibilidades de encontrar un DecisionStump óptimo en cada iteración, lo que puede mejorar la precisión pero también aumenta el costo computacional del entrenamiento.

Un error que yo cometí y el cual encuentro interesante es que al principio entrenaba con los valores T = [10, 10, 10, 10] y A = [20, 30, 5, 15] lo que sobreentrenaba el modelo, lo cual es erróneo.

Otro error que cometí sobre todo en otras tareas fue el poner un valor para A excesivamente alto, por ejemplo, experimenté poner T=100 y A=50, pero a pesar de que pueda obtener una mejor precisión, el tiempo puede ser excesivamente largo, pudiendo tardar en solo esta tarea más de 5 minutos.

Implementación

La función tarea1C genera las gráficas deseadas utilizando los datos proporcionados de la tarea_1A_y_1B (precisión y tiempos de entrenamiento para diferentes valores de T y A).

Parámetros de Entrada:

- T_values: Lista de valores de T.
- A_values: Lista de valores de A.
- train_accuracies: Precisión en el conjunto de entrenamiento.
- test_accuracies: Precisión en el conjunto de prueba.
- training_times: Tiempos de entrenamiento.

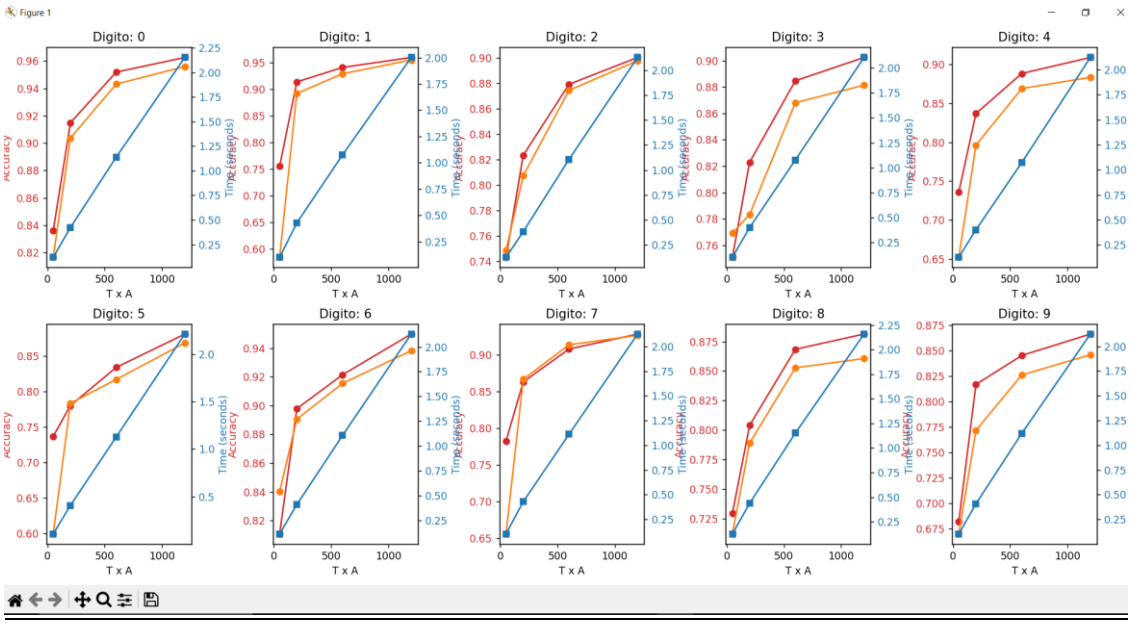
Gráfica Generada: Muestra dos curvas en un solo gráfico, una para la precisión y otra para el tiempo de entrenamiento,(la linea recta azul es el paso del tiempo de ejecución), facilitando la comparación visual entre estos dos aspectos críticos.

Ejemplo de ejecución

(Destacar que en cada ejecución no sale exactamente la misma gráfica, por lo que voy explicarla la ecución en base a la siguiente gráfica)

T_values = [10, 20, 30, 40]

A_values = [5, 10, 20, 30]



- La línea roja es la precisión de entrenamiento.
- La línea naranja es la precisión de los test.
- La línea azul es el tiempo de ejecución.

En general, la precisión de entrenamiento mejora con el aumento del número de clasificadores. Esto indica que el ensamble se beneficia de un mayor número de clasificadores débiles para capturar la complejidad de los datos.

Para muchos dígitos, la precisión de prueba mejora inicialmente con más clasificadores y luego se estabiliza o incluso disminuye, lo que sugiere que hay un punto de rendimiento óptimo después del cual se produce sobreajuste o se obtienen ganancias marginales.

El número de intentos de ajuste por clasificador (A) también influye en la precisión, pero debe equilibrarse cuidadosamente para evitar el sobreajuste. No siempre un mayor número de intentos conduce a mejores resultados en los datos de prueba, lo que indica la importancia de la validación cruzada para encontrar el mejor valor de A.

El tiempo de ejecución se incrementa con más clasificadores e intentos, lo que puede ser un factor limitante en entornos con recursos computacionales restringidos o donde se requiere una respuesta rápida.

Conclusiones de algunos dígitos:

Dígito 0: Este dígito muestra una buena mejora en la precisión de prueba con el aumento de T y A, pero el tiempo de ejecución también aumenta. Un equilibrio entre precisión y tiempo de ejecución podría encontrarse en un valor intermedio de T y A.

Dígito 1: La precisión de prueba para el dígito 1 es bastante alta, incluso con valores bajos de T y A, y no parece mejorar significativamente con valores más altos. Esto sugiere que el dígito 1 es más fácil de clasificar y que podríamos evitar el uso de modelos excesivamente complejos.

Dígito 2: Observamos una mejora constante en la precisión de prueba con un número creciente de clasificadores, pero también un aumento en el tiempo de ejecución. Podría ser útil explorar un punto medio de T que mantenga una alta precisión sin incurrir en costos de tiempo excesivos.

Dígito 3: Parece haber un punto óptimo de T y A donde la precisión de prueba es alta y el tiempo de ejecución no es prohibitivo. Valores más altos de T y A no mejoran significativamente la precisión y aumentan el tiempo de ejecución.

Dígito 5: La precisión de prueba aumenta con más clasificadores hasta cierto punto antes de que comience a estabilizarse, lo cual puede indicar sobreajuste si se sigue incrementando el número de clasificadores. El tiempo de ejecución también aumenta con más clasificadores e intentos. Un balance entre precisión y eficiencia se encontraría en un número moderado de clasificadores.

Dígito 7: Este dígito muestra una tendencia similar al dígito 3, con un punto de eficiencia donde se logra una buena precisión de prueba sin tiempos de ejecución excesivos.

Dígito 9: La precisión de prueba también mejora con más clasificadores, pero hay que tener cuidado con el sobreajuste, como se indica por la estabilización de la precisión de prueba en números más altos de clasificadores. El tiempo de ejecución sigue la misma tendencia de aumento que con el dígito 5, lo que sugiere que un número demasiado alto de clasificadores puede no ser eficiente.

En resumen, la eficacia del entrenamiento de AdaBoost para la clasificación de dígitos del conjunto de datos MNIST varía según el dígito. Algunos dígitos son intrínsecamente más fáciles de clasificar y requieren menos clasificadores y menos intentos de ajuste. Para otros, la precisión mejora con más clasificadores hasta un punto después del cual se obtienen rendimientos decrecientes y un posible sobreajuste. El tiempo de ejecución siempre debe considerarse, ya que los modelos más complejos no siempre justifican el costo computacional adicional. La optimización de hiperparámetros, por lo tanto, debe hacerse de manera cuidadosa y posiblemente de manera individualizada para cada dígito, buscando el equilibrio entre la precisión, la generalización y la eficiencia.

Traza

PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py

Tarea 1C : Gráficas de investigación sobre entrenar para varios T y A

Entrenando clasificador Adaboost para el dígito 0, T=10, A=20

Tasas acierto (train, test) y tiempo: 85.71%, 84.67%, 0.415 s

Matriz de Confusión:

[[7630 1390]

[143 837]]

Entrenando clasificador Adaboost para el dígito 0, T=20, A=30

Tasas acierto (train, test) y tiempo: 94.29%, 93.23%, 1.190 s

Matriz de Confusión:

[[8380 640]

[37 943]]

Entrenando clasificador Adaboost para el dígito 0, T=40, A=20

Tasas acierto (train, test) y tiempo: 95.73%, 94.93%, 1.533 s

Matriz de Confusión:

[[8548 472]

[35 945]]

Para el digito 0

El modelo tiene una precision de entrenamiento de media0.9190814332649123

El modelo tiene una precision de test de media 0.9094333333333333

En el tiempo medio de 1.046230713526408

Entrenando clasificador Adaboost para el dígito 1, T=10, A=20

Tasas acierto (train, test) y tiempo: 90.55%, 88.76%, 0.371 s

Matriz de Confusión:

[[7805 1060]

[64 1071]]

Entrenando clasificador Adaboost para el dígito 1, T=20, A=30

Tasas acierto (train, test) y tiempo: 94.11%, 92.92%, 1.090 s

Matriz de Confusión:

[[8196 669]

[39 1096]]

Entrenando clasificador Adaboost para el dígito 1, T=40, A=20

Tasas acierto (train, test) y tiempo: 96.33%, 95.77%, 1.559 s

Matriz de Confusión:

[[8473 392]

 [31 1104]]

Para el digito 1

El modelo tiene una precision de entrenamiento de media0.93661672111858

El modelo tiene una precision de test de media 0.9248333333333333

En el tiempo medio de 1.006890853246053

Entrenando clasificador Adaboost para el dígito 2, T=10, A=20

Tasas acierto (train, test) y tiempo: 81.74%, 79.44%, 0.375 s

Matriz de Confusión:

[[7081 1887]

 [169 863]]

Entrenando clasificador Adaboost para el dígito 2, T=20, A=30

Tasas acierto (train, test) y tiempo: 88.34%, 88.28%, 1.116 s

Matriz de Confusión:

[[7929 1039]

 [133 899]]

Entrenando clasificador Adaboost para el dígito 2, T=40, A=20

Tasas acierto (train, test) y tiempo: 89.46%, 89.26%, 1.501 s

Matriz de Confusión:

[[8022 946]

 [128 904]]

Para el digito 2

El modelo tiene una precision de entrenamiento de media0.8651542627339229

El modelo tiene una precision de test de media 0.8565999999999999

En el tiempo medio de 0.9975654284159342

Entrenando clasificador Adaboost para el dígito 3, T=10, A=20

Tasas acierto (train, test) y tiempo: 82.81%, 80.39%, 0.354 s

Matriz de Confusión:

[[7231 1759]

[202 808]]

Entrenando clasificador Adaboost para el dígito 3, T=20, A=30

Tasas acierto (train, test) y tiempo: 87.06%, 84.21%, 1.088 s

Matriz de Confusión:

[[7528 1462]

[117 893]]

Entrenando clasificador Adaboost para el dígito 3, T=40, A=20

Tasas acierto (train, test) y tiempo: 88.94%, 85.97%, 1.489 s

Matriz de Confusión:

[[7685 1305]

[98 912]]

Para el digito 3

El modelo tiene una precision de entrenamiento de media0.8626946852549704

El modelo tiene una precision de test de media 0.8352333333333334

En el tiempo medio de 0.9772059917449951

Entrenando clasificador Adaboost para el dígito 4, T=10, A=20

Tasas acierto (train, test) y tiempo: 78.01%, 71.60%, 0.446 s

Matriz de Confusión:

[[6363 2655]

[185 797]]

Entrenando clasificador Adaboost para el dígito 4, T=20, A=30

Tasas acierto (train, test) y tiempo: 88.54%, 86.63%, 1.090 s

Matriz de Confusión:

[[7829 1189]

[148 834]]

Entrenando clasificador Adaboost para el dígito 4, T=40, A=20

Tasas acierto (train, test) y tiempo: 89.01%, 87.51%, 1.481 s

Matriz de Confusión:

[[7851 1167]

[82 900]]

Para el digito 4

El modelo tiene una precision de entrenamiento de media0.851877838915765

El modelo tiene una precision de test de media 0.8191333333333333

En el tiempo medio de 1.0058977603912354

Entrenando clasificador Adaboost para el dígito 5, T=10, A=20

Tasas acierto (train, test) y tiempo: 75.06%, 71.50%, 0.442 s

Matriz de Confusión:

[[6474 2634]

[216 676]]

Entrenando clasificador Adaboost para el dígito 5, T=20, A=30

Tasas acierto (train, test) y tiempo: 83.80%, 81.68%, 1.139 s

Matriz de Confusión:

[[7392 1716]

[116 776]]

Entrenando clasificador Adaboost para el dígito 5, T=40, A=20

Tasas acierto (train, test) y tiempo: 86.61%, 84.83%, 1.510 s

Matriz de Confusión:

[[7718 1390]

[127 765]]

Para el digito 5

El modelo tiene una precision de entrenamiento de media0.8182359515564595

El modelo tiene una precision de test de media 0.7933666666666667

En el tiempo medio de 1.0305670897165935

Entrenando clasificador Adaboost para el dígito 6, T=10, A=20

Tasas acierto (train, test) y tiempo: 89.97%, 90.35%, 0.450 s

Matriz de Confusión:

[[8227 815]

[150 808]]

Entrenando clasificador Adaboost para el dígito 6, T=20, A=30

Tasas acierto (train, test) y tiempo: 93.09%, 91.73%, 1.155 s

Matriz de Confusión:

[[8304 738]

[89 869]]

Entrenando clasificador Adaboost para el dígito 6, T=40, A=20

Tasas acierto (train, test) y tiempo: 94.93%, 94.70%, 1.615 s

Matriz de Confusión:

[[8599 443]

[87 871]]

Para el digito 6

El modelo tiene una precision de entrenamiento de media0.9266052784043982

El modelo tiene una precision de test de media 0.9226

En el tiempo medio de 1.07357390721639

Entrenando clasificador Adaboost para el dígito 7, T=10, A=20

Tasas acierto (train, test) y tiempo: 82.96%, 84.95%, 0.351 s

Matriz de Confusión:

[[7686 1286]

[219 809]]

Entrenando clasificador Adaboost para el dígito 7, T=20, A=30

Tasas acierto (train, test) y tiempo: 91.34%, 88.50%, 1.106 s

Matriz de Confusión:

[[7906 1066]

[84 944]]

Entrenando clasificador Adaboost para el dígito 7, T=40, A=20

Tasas acierto (train, test) y tiempo: 92.96%, 92.38%, 1.748 s

Matriz de Confusión:

[[8298 674]

[88 940]]

Para el digito 7

El modelo tiene una precision de entrenamiento de media0.8908718712198752

El modelo tiene una precision de test de media 0.8861

En el tiempo medio de 1.0685651302337646

Entrenando clasificador Adaboost para el dígito 8, T=10, A=20

Tasas acierto (train, test) y tiempo: 80.10%, 77.97%, 0.416 s

Matriz de Confusión:

[[7043 1983]

[220 754]]

Entrenando clasificador Adaboost para el dígito 8, T=20, A=30

Tasas acierto (train, test) y tiempo: 85.30%, 83.37%, 1.156 s

Matriz de Confusión:

[[7519 1507]

[156 818]]

Entrenando clasificador Adaboost para el dígito 8, T=40, A=20

Tasas acierto (train, test) y tiempo: 88.17%, 86.67%, 1.526 s

Matriz de Confusión:

[[7868 1158]

[175 799]]

Para el digito 8

El modelo tiene una precision de entrenamiento de media0.8452202872321434

El modelo tiene una precision de test de media 0.8267000000000001

En el tiempo medio de 1.0328954855600994

Entrenando clasificador Adaboost para el dígito 9, T=10, A=20

Tasas acierto (train, test) y tiempo: 78.57%, 73.42%, 0.367 s

Matriz de Confusión:

[[6491 2500]

[158 851]]

Entrenando clasificador Adaboost para el dígito 9, T=20, A=30

Tasas acierto (train, test) y tiempo: 81.27%, 80.72%, 1.107 s

Matriz de Confusión:

[[7235 1756]

[172 837]]

Entrenando clasificador Adaboost para el dígito 9, T=40, A=20

Tasas acierto (train, test) y tiempo: 89.35%, 87.29%, 1.546 s

[[7856 1135]

[136 873]]

Para el digito 9

El modelo tiene una precision de entrenamiento de media0.8306321807182107

El modelo tiene una precision de test de media 0.8047666666666666

En el tiempo medio de 1.0068879127502441

Conclusiones

Es importante encontrar un equilibrio entre la precisión y el tiempo de entrenamiento. Un modelo muy preciso que tarda mucho en entrenarse puede no ser práctico, mientras que un modelo rápido pero inexacto no será útil.

La optimización de T y A depende del contexto específico y los requisitos de rendimiento. La tarea 1C ayuda a entender cómo ajustar estos parámetros para obtener los mejores resultados en MNIST, respetando la restricción de $T \times A \leq 900$.

Tarea 1D: Clasificador Multiclase

El objetivo principal es transformar el clasificador binario AdaBoost en un clasificador multiclase capaz de identificar cada uno de los dígitos del 0 al 9 en el conjunto de datos MNIST. Esto se logra mediante la implementación de 10 clasificadores binarios independientes, cada uno especializado en identificar un dígito específico.

> Como ya he explicado antes, se podría decir que yo he implementado el binario en la tarea1C.
>
> La versión que programe al principio, era que en la implementación de la tarea1D(), para cada dígito, llamaba a la función tarea1C(), la que ya se encargaba de entrenar cada dígito y obtener resultados. Sin embargo, la tarea1C está más orientada al estudio y representación del entrenamiento para diferentes valores de T y de A, por lo que para tareas futuras, en concreto la tarea2B(), me vi obligada a hacerlo por separado, pero igualmente me pareció interesante dejar como ya he mencionado, en la tarea1C, el estudio para todos los dígitos en vez de ser llamada solo para 1.
>
> Si se quisiera hacer que la tarea1C() no hiciera un estudio como un adaboost multiclase, simplemente se eliminaría el for que recorre todos los dígitos, se reestructuraría el código y se recorrería igualmente los valores T x A para solo un dígito predeterminado.

Dado que más de un clasificador binario puede clasificar una imagen como positiva, es necesario un mecanismo para seleccionar la clase más probable. En lugar de utilizar solo el signo de la predicción (1 o -1), se deben considerar los valores reales de las predicciones para determinar la certeza de cada clasificación.

Metodología**:** Implementar 10 clasificadores binarios AdaBoost, cada uno entrenado para distinguir un dígito específico del resto.

Implementación: Pruebas con Diferentes Dígitos

El bucle principal itera sobre cada dígito (0-9), utilizando los clasificadores binarios para realizar la clasificación, es decir para cada dígito se creará un clasificador Adaboost que se entrenará en la Tarea 1B, y después se guardan esos clasificadores para realizar una precisión final.

Un punto a resaltar que se ha llevado a cabo en esta práctica es que para pasar de hacer el Adaboost Multiclase, en el predict del adaboost en vez de devolver etiquetas binarias que se generaban gracias a np.sing(y_pred), tengo un booleano que según si lo que se pretende invocar es el predict del adaboost binario, etiquetas_binarias = True y se devolverán np.sign(y_pred), o si es el predict del adaboost multiclase, etiquetas_binarias=False y se devolverán directamente el valor de las predicciones, escogiendo más adelante el clasificador que tenga un valor de predicción mayor y evaluándolo, como se mencionó en clase.

Ejemplo de ejecución

T = 100

A = 5

PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py

Tarea 1D : Clasificador multiclase

Adaboost Multiclase T = 100, A = 5

Entrenando clasificador Adaboost para el dígito 0, T=100, A=5

Tasas acierto (train, test) y tiempo: 96.33%, 96.20%, 1.098 s

Matriz de Confusión:

[[8668 352]

[28 952]]

Entrenando clasificador Adaboost para el dígito 1, T=100, A=5

Tasas acierto (train, test) y tiempo: 95.09%, 94.12%, 1.070 s

Matriz de Confusión:

[[8315 550]

[38 1097]]

Entrenando clasificador Adaboost para el dígito 2, T=100, A=5

Tasas acierto (train, test) y tiempo: 89.84%, 90.23%, 1.140 s

Matriz de Confusión:

[[8100 868]

[109 923]]

Entrenando clasificador Adaboost para el dígito 3, T=100, A=5

Tasas acierto (train, test) y tiempo: 90.30%, 89.52%, 1.201 s

Matriz de Confusión:

[[8054 936]

[112 898]]

Entrenando clasificador Adaboost para el dígito 4, T=100, A=5

Tasas acierto (train, test) y tiempo: 90.97%, 88.66%, 1.200 s

Matriz de Confusión:

[[7991 1027]

[107 875]]

Entrenando clasificador Adaboost para el dígito 5, T=100, A=5

Tasas acierto (train, test) y tiempo: 85.73%, 85.08%, 1.235 s

Matriz de Confusión:

[[7766 1342]

[150 742]]

Entrenando clasificador Adaboost para el dígito 6, T=100, A=5

Tasas acierto (train, test) y tiempo: 95.36%, 94.31%, 1.202 s

Matriz de Confusión:

[[8544 498]

[71 887]]

Entrenando clasificador Adaboost para el dígito 7, T=100, A=5

Tasas acierto (train, test) y tiempo: 91.67%, 91.74%, 1.193 s

Matriz de Confusión:

[[8225 747]

[79 949]]

Entrenando clasificador Adaboost para el dígito 8, T=100, A=5

Tasas acierto (train, test) y tiempo: 88.20%, 86.81%, 1.232 s

Matriz de Confusión:

[[7870 1156]

[163 811]]

Entrenando clasificador Adaboost para el dígito 9, T=100, A=5

Tasas acierto (train, test) y tiempo: 88.40%, 86.15%, 1.180 s

Matriz de Confusión:

[[7761 1230]

[155 854]]

Matriz de Confusión para el clasificador multiclase:

[[917 1 11 1 3 22 13 4 5 3]

[0 1070 23 11 1 4 3 1 22 0]

[19 61 794 32 24 6 35 25 22 14]

[14 17 20 804 0 65 11 25 39 15]

[3 9 7 6 802 17 27 23 21 67]

[29 27 5 104 36 578 23 28 39 23]

[10 16 16 3 40 27 827 7 11 1]

[8 41 31 14 15 1 0 869 7 42]

[10 12 14 64 24 63 16 15 711 45]

[13 10 14 14 100 21 5 100 30 702]]

Precisión del clasificador multiclase: 80.74%

He escogido un valor tan alto de T con un número tan pequeño de A, ya que el Adaboost multiclase me daba una tasa relativamente baja, de un 64% si T = 10 y A = 20, que es lo normal debido a la complejidad de distinguir entre 10 clases diferentes y las limitaciones de los clasificadores binarios individuales al trabajar juntos en un enfoque "one-vs-all", además de la elección aleatoria de características y umbrales en los "Decision Stumps" puede no ser óptima, lo que puede afectar la precisión general del modelo. Como decía, gracias a las conclusiones sacadas de la tarea1C(), Si pongo de T un numero alto y un numero de intentos bajo, conseguiré buena precisión, o al menos una precisión aceptable en un tiempo relativamente corto.

Obviamente esto se puede seguir experimentando, variando los valores de T y A, pero es el experimento más interesante y el que he querido comentar.

Tarea 1E (OPTATIVA): Mejoras creativas

Esta estrategia se enfoca en optimizar el proceso de ensamblaje de AdaBoost para mejorar la precisión y eficiencia.

Para esta tarea he probado tres posibles mejoras, y aunque solo me quedé con la última recalco las otras dos descartadas ya que me parecieron muy interesantes pero en nuestro caso no sirvieron:

- Una era la selección dinámica de características. En la implementación actual, cada DecisionStump selecciona una característica (un píxel en el contexto de las imágenes MNIST) al azar. Sin embargo, no todos los píxeles contribuyen igualmente a la precisión del clasificador. Una selección más informada puede mejorar significativamente el rendimiento. Todo esto se hace a través del estudio de la varianza.
  Antes del entrenamiento, calculaba la varianza de cada característica (píxel) en el conjunto de entrenamiento. Las características con varianza más alta son más propensas a ser informativas.
  Sin embargo, esta mejora fue descartada ya que no aumentaba la precisión, incluso la reducía un poco y el tiempo igual se incrementaba en unidades, pero se incrementaba, por lo que decidí descartarla.

- La otra era la integración de características derivadas junto con la selección de características aleatoria. Esto implica generar nuevas características basadas en las existentes, lo que podría ayudar a mejorar la capacidad de discriminación de los clasificadores débiles.
  En lugar de considerar solo píxeles individuales, podríamos crear características derivadas que capturen información sobre las relaciones entre píxeles adyacentes. Por ejemplo, podríamos calcular la diferencia o la suma de píxeles adyacentes, lo que podría ayudar a capturar bordes, esquinas y otras estructuras importantes en las imágenes de dígitos.
  Esta mejora también fue descartada porque incrementaba el tiempo en unidades y no había cambio en la precisión.

Por lo que finalmente me quedé con la siguiente propuesta:

Mejora Propuesta: **Ajuste Dinámico de T y A**

En lugar de usar un número fijo de clasificadores débiles (T) y de intentos (A), podríamos ajustar estos valores dinámicamente durante el entrenamiento. Si el modelo está mejorando rápidamente, podemos reducir T para acelerar el entrenamiento. Si el modelo se estanca o mejora lentamente, podemos aumentar A para buscar clasificadores débiles más efectivos.

Esta estrategia busca equilibrar la exploración de nuevos clasificadores con la eficiencia del entrenamiento. Ajustar dinámicamente T y A puede ayudar a evitar un entrenamiento innecesariamente largo y encontrar un buen conjunto de clasificadores más rápidamente, sin embargo, ajustar dinámicamente T y A puede mejorar el rendimiento, pero aumentar el riesgo de sobreajuste, especialmente si se aumenta demasiado A. Es crucial monitorear el rendimiento no solo en el conjunto de entrenamiento sino también en un conjunto de validación o prueba.

<u>Explicación de la Mejora:</u>

- umbral_mejora: Es un valor que determina qué tan grande debe ser la mejora en el error para considerar que un clasificador está contribuyendo significativamente.

- incremento_A: Si el clasificador no mejora el error más allá del umbral, se incrementa el valor de A para explorar más clasificadores en las siguientes rondas.
- min_iteraciones y early_stopping: Estos parámetros permiten detener el entrenamiento de manera anticipada si no se observan mejoras significativas, lo que puede evitar el sobreajuste y reducir el tiempo de entrenamiento.

Implementación

Durante el entrenamiento, evalúa cuánto mejora cada nuevo clasificador débil. Si un clasificador no mejora significativamente el modelo, considera detener el entrenamiento temprano o aumentar A para la siguiente iteración.

Implementa un criterio de detención temprana basado en la mejora observada. Por ejemplo, si la tasa de error no disminuye significativamente después de cierto número de iteraciones, detenemos el entrenamiento.

Si los clasificadores seleccionados no están mejorando el modelo, aumenta A para explorar más opciones en las siguientes rondas.

Tras experimentar con diferentes valores para los umbrales de mejora, incrementos de A, y condiciones de detención temprana deje los valores:

```
ass Adaboost:
  def __init__(self, T=5, A=20):
      self.T = T
      self.A = A
      self.umbral_mejora = 0.01
      self.incremento_A = 5
      self.min_iteraciones = 10
      self.early_stopping = True
      self.lista_clasificadores = []
```

Esta mejora se centra en hacer que el proceso de entrenamiento de AdaBoost sea más inteligente y adaptable, potencialmente llevando a un mejor equilibrio entre precisión y eficiencia.

Ahora vamos a comparar las dos versiones, la original y la nueva:

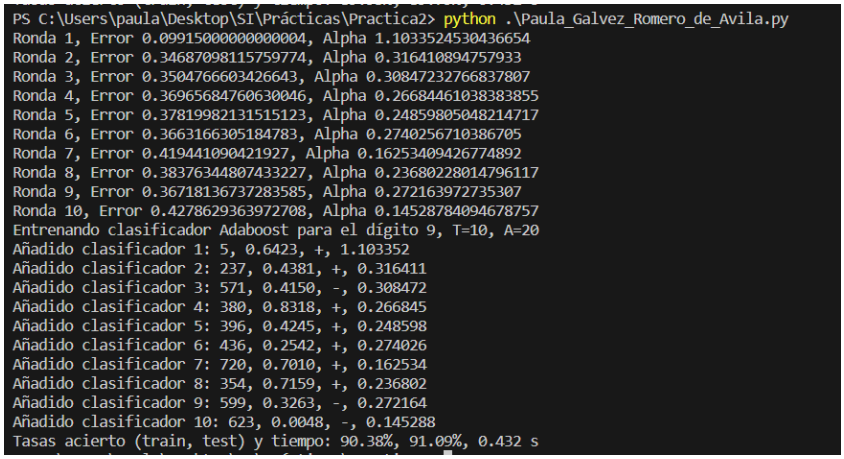
Realizando el entrenamiento para el dígito 9 por ejemplo donde T = 10 y A = 20 con verbose = True, ejecutamos 3 veces:

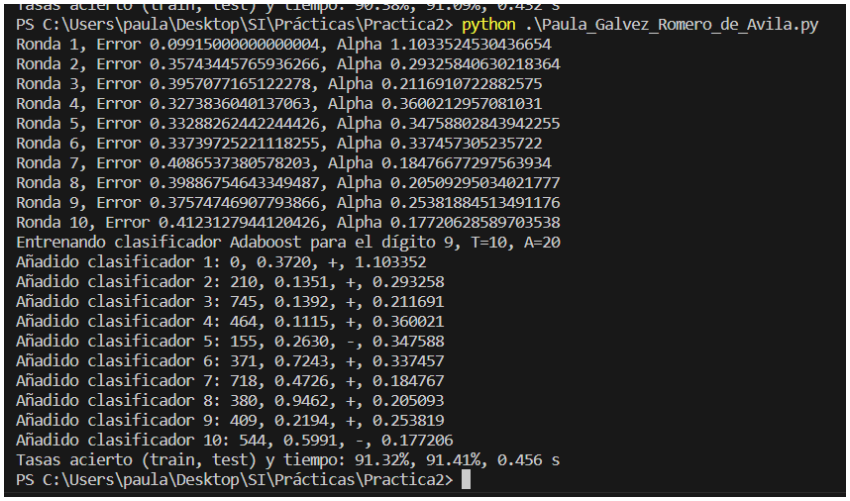
```
PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py
Ronda 1, Caracteristica 721, Umbral 0.9402869183560714, Polaridad 1, Error 0.0985000000000003, Alpha 1.10700174672505
Ronda 2, Caracteristica 685, Umbral 0.0024132456147515358, Polaridad 1, Error 0.41795165599088, Alpha 0.16559386951914015
Ronda 3, Caracteristica 495, Umbral 0.9602272326224887, Polaridad -1, Error 0.4339484361281964, Alpha 0.13287972989643165
Ronda 4, Caracteristica 572, Umbral 0.18662359437200038, Polaridad -1, Error 0.37629340524591887, Alpha 0.2526554423104406
Ronda 5, Caracteristica 715, Umbral 0.8067334964266772, Polaridad 1, Error 0.4153512188342914, Alpha 0.170943409931343
Ronda 6, Caracteristica 629, Umbral 0.5520497673134688, Polaridad -1, Error 0.4100545467653624, Alpha 0.18186994424846162
Ronda 7, Caracteristica 291, Umbral 0.4551532630206786, Polaridad 1, Error 0.3839018089165774 4, Alpha 0.23650976886825378
Ronda 8, Caracteristica 566, Umbral 0.19903800710869057, Polaridad -1, Error 0.41858140760965973, Alpha 0.16429978632641848
Ronda 9, Caracteristica 373, Umbral 0.5438073873660172, Polaridad 1, Error 0.3930845467493984, Alpha 0.21718238722571037
Ronda 10, Caracteristica 87, Umbral 0.6844681207889599, Polaridad 1, Error 0.44170622029893797, Alpha 0.11712015469629015
Entrenando clasificador Adaboost para el dígito 9, T=10, A=20
Añadido clasificador 1: 721, 0.9403, +, 1.107002
Añadido clasificador 2: 685, 0.0024, +, 0.165594
Añadido clasificador 3: 495, 0.9602, -, 0.132880
Añadido clasificador 4: 572, 0.1866, -, 0.252655
Añadido clasificador 5: 715, 0.8067, +, 0.170943
Añadido clasificador 6: 629, 0.5520, -, 0.181870
Añadido clasificador 7: 291, 0.4552, +, 0.236510
Añadido clasificador 8: 566, 0.1990, -, 0.164300
Añadido clasificador 9: 373, 0.5438, +, 0.217182
Añadido clasificador 10: 87, 0.6845, +, 0.117120
Tasas acierto (train, test) y tiempo: 90.16%, 89.95%, 0.225 s
```

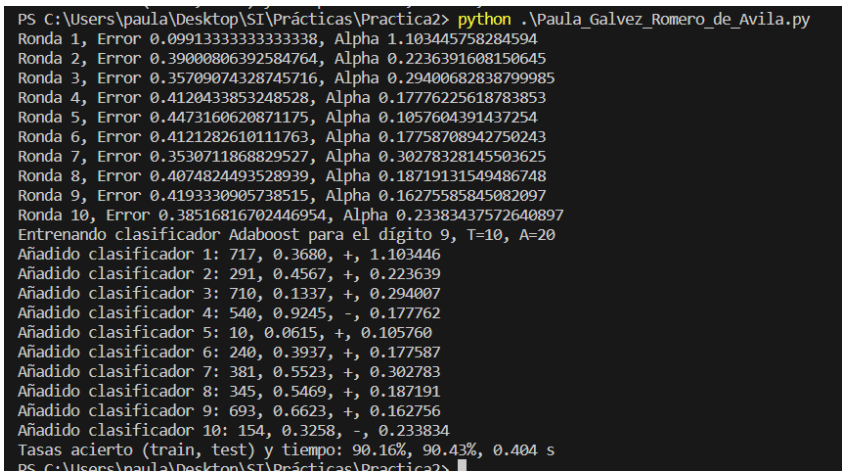
```
PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py
Ronda 1, Caracteristica 169, Umbral 0.22059236561619944, Polaridad 1, Error 0.09915000000000004, Alpha 1.1033524530436654
Ronda 2, Caracteristica 373, Umbral 0.47376729508384885, Polaridad 1, Error 0.3687581235266678, Alpha 0.2687740869841217
Ronda 3, Caracteristica 464, Umbral 0.6146971498339711, Polaridad 1, Error 0.3498073860230466, Alpha 0.3099429852760091
Ronda 4, Caracteristica 286, Umbral 0.25912067550654505, Polaridad 1, Error 0.46606851024614815, Alpha 0.06796744657932632
Ronda 5, Caracteristica 483, Umbral 0.1676856117474309, Polaridad -1, Error 0.4363740283602794, Alpha 0.1279455608079674
Ronda 6, Caracteristica 746, Umbral 0.4864217504501577, Polaridad 1, Error 0.45547524202679807, Alpha 0.0892860242422411
Ronda 7, Caracteristica 680, Umbral 0.8360892051664321, Polaridad -1, Error 0.48434854823060375, Alpha 0.03131313372802493
Ronda 8, Caracteristica 371, Umbral 0.5144033495895451, Polaridad 1, Error 0.40481832249733646, Alpha 0.1927141594985614
Ronda 9, Caracteristica 571, Umbral 0.33117311020221396, Polaridad -1, Error 0.32384633740226304, Alpha 0.3680756127114349
Ronda 10, Caracteristica 154, Umbral 0.6312353286669627, Polaridad -1, Error 0.41552148243096365, Alpha 0.17059285545899686
Entrenando clasificador Adaboost para el dígito 9, T=10, A=20
Añadido clasificador 1: 169, 0.2206, +, 1.103352
Añadido clasificador 2: 373, 0.4738, +, 0.268774
Añadido clasificador 3: 464, 0.6147, +, 0.309943
Añadido clasificador 4: 286, 0.2591, +, 0.067967
Añadido clasificador 5: 483, 0.1677, -, 0.127946
Añadido clasificador 6: 746, 0.4864, +, 0.089286
Añadido clasificador 7: 680, 0.8361, -, 0.031313
Añadido clasificador 8: 371, 0.5144, +, 0.192714
Añadido clasificador 9: 571, 0.3312, -, 0.368076
Añadido clasificador 10: 154, 0.6312, -, 0.170593
Tasas acierto (train, test) y tiempo: 89.04%, 88.73%, 0.231 s
PS C:\Users\paula\Desktop\SI\Prácticas\Practica2>
```

```
PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py
Ronda 1, Caracteristica 307, Umbral 0.754288616586432, Polaridad 1, Error 0.09916666666666671, Alpha 1.1032591617596375
Ronda 2, Caracteristica 409, Umbral 0.34721766964949663, Polaridad 1, Error 0.31634504320623963, Alpha 0.38530990706436674
Ronda 3, Caracteristica 598, Umbral 0.29526718297887267, Polaridad -1, Error 0.33775065149879224, Alpha 0.33666711915679315
Ronda 4, Caracteristica 224, Umbral 0.8389893940025842, Polaridad 1, Error 0.33772597214021216, Alpha 0.3367222880044183
Ronda 5, Caracteristica 566, Umbral 0.09642171130741728, Polaridad -1, Error 0.41715928537388575, Alpha 0.1672229019191563
Ronda 6, Caracteristica 570, Umbral 0.34346250698021397, Polaridad -1, Error 0.4256378149048854, Alpha 0.14983569834654542
Ronda 7, Caracteristica 4, Umbral 0.7966461033847748, Polaridad 1, Error 0.3858719503510807, Alpha 0.23234893823719424
Ronda 8, Caracteristica 630, Umbral 0.38326557677554796, Polaridad -1, Error 0.4363962280768837, Alpha 0.12790043083752795
Ronda 9, Caracteristica 326, Umbral 0.8333028802281272, Polaridad 1, Error 0.42128457698382954, Alpha 0.15875114948551766
Ronda 10, Caracteristica 264, Umbral 0.8532243722742169, Polaridad 1, Error 0.43220728051326807, Alpha 0.13642556611659357
Añadido clasificador 1: 307, 0.7543, +, 1.103259
Añadido clasificador 2: 409, 0.3472, +, 0.385310
Añadido clasificador 3: 598, 0.2953, -, 0.336667
Añadido clasificador 4: 224, 0.8390, +, 0.336722
Añadido clasificador 5: 566, 0.0964, -, 0.167223
Añadido clasificador 6: 570, 0.3435, -, 0.149836
Añadido clasificador 7: 4, 0.7966, +, 0.232349
Añadido clasificador 8: 630, 0.3833, -, 0.127900
Añadido clasificador 9: 326, 0.8333, +, 0.158751
Añadido clasificador 10: 264, 0.8532, +, 0.136426
Tasas acierto (train, test) y tiempo: 90.08%, 89.90%, 0.226 s
```

Y tras hacer la tarea 1E, ejecutamos 3 veces:







Aunque incremente un poquito el tiempo, la precisión suele ser igual o mejor. Hay veces que conviene el equilibrio y aunque haya un poco más de coste computacional y tarde un poco más el programa, puede haber mejores resultados.

He optado por esta mejora ya que encima como ya he mencionado, si variamos los parámetros y experimentamos con ellos, se pueden obtener resultados distintos, pero en este caso, esta ‘configuración’, produce resultados de precisión igual o mejores.

Tarea 2A: Modela el clasificador Adaboost con scikit-learn

Scikit-learn es una biblioteca para aprendizaje automático de software libre para el lenguaje de programación Python, que ofrece una amplia variedad de herramientas eficientes para análisis predictivo y modelado de datos.Incluye varios algoritmos de clasificación, regresión y análisis de grupos entre los cuales están máquinas de vectores de soporte, bosques aleatorios, Gradient boosting, K-means y DBSCAN. Está diseñada para interoperar con las bibliotecas numéricas y científicas NumPy y SciPy y es reconocida por su facilidad de uso, rendimiento y versatilidad para realizar tareas de aprendizaje automático.

Esta tarea solicita la implementación de AdaBoostClassifier que es una implementación del algoritmo AdaBoost (Adaptive Boosting) en scikit-learn. Es un algoritmo de ensamblaje que combina múltiples clasificadores débiles para crear un clasificador fuerte. AdaBoost ajusta iterativamente los pesos de los datos de entrenamiento basándose en las predicciones de los clasificadores débiles, dando más peso a las instancias mal clasificadas en iteraciones anteriores, como ya se ha explicado antes.

Implementación

Se utiliza fetch_openml para cargar el conjunto de datos MNIST.

Las etiquetas se convierten a enteros para un manejo más eficiente.

Los datos se dividen en conjuntos de entrenamiento y prueba usando train_test_split.

Normalizo los valores a píxeles.

Se instancia AdaBoostClassifier con n estimadores lo que se identifica con el parámetro T, y una semilla para la generación de números aleatorios (random_state=42) para asegurar la reproducibilidad.

> Cada uno de estos estimadores (n_estimators/T) se entrena solo una vez, pero con un conjunto de pesos de entrenamiento que se actualiza en cada iteración. Por lo tanto, no hay un parámetro para "intentos por clasificador", es decir lo que sería A, porque cada clasificador se construye solo una vez, por lo que, A podría ser el número de características a considerar en un clasificador de árbol de decisión, como en DecisionTreeClassifier. Esto podría ser una estrategia para mejorar la diversidad entre los árboles en un ensamble como Adaboost.
>
> Por defecto, AdaBoostClassifier utiliza DecisionTreeClassifier como el clasificador base, por lo que experimentaremos con sus parámetros en la tarea 2C().

Se entrena el modelo con fit usando los conjuntos de entrenamiento.

Se evalúa el modelo con el conjunto de prueba, calculando la precisión y generando un informe de clasificación que incluye métricas como precisión, recuperación y puntuación.

Parámetros de AdaBoostClassifier

- estimator : El estimador base a partir del cual se construye el conjunto impulsado. Se requiere soporte para la ponderación de la muestra, así como también classes_ y n_classes_ atributos. El estimador base por defecto es DecisionTreeClassifier inicializado con max_depth=1.
- n_estimators (T): Número de clasificadores débiles a entrenar. Un número mayor puede mejorar el rendimiento, pero también aumenta el riesgo de sobreajuste y el tiempo de computación. Por defecto es 50.
- learning_rate: Pondera la contribución de cada clasificador. Un valor más bajo puede requerir más estimadores, pero puede mejorar el rendimiento general. Por defecto es 1.0.
- algorithm: Puede ser ‘SAMME’ o ‘SAMME.R’. ‘SAMME.R’ utiliza la probabilidad de clase y suele ser más efectiva, y también es la opción por defecto.
- base_estimator: Permite especificar un clasificador base diferente. Por defecto, es un árbol de decisión de un solo nivel.

En nuestro caso habrá que experimentar, pero inicialmente solo creamos el AdaboostClassifier con el parámetro n_estimators = igual a n, ya que a la función le pasamos por parámetro el valor de n.

En cuanto a la experimentación de parámetros:

Utilizando el n_estimators por defecto (con valor 50), consigo una tasa relativamente baja -------, por lo que tras estar experimentando he visto viable pasarle un valor de n que sea 100, ya que aunque tarde un poco más consigo una mejor precisión.

Tarea 2B: Compara tu versión de adabost con la de scikit-learn.

En esta tarea se quiere comparar mi adaboost multiclase con el adaboost implementado de sklearn, para varios valores de T y de n_estimators en el adaboost multiclase. Yo he generado una gráfica en la que comparo para cada valor la precisión de mi Adaboost Multiclase, del AdaboostClaassifier de sklearn básico de la tarea 2A, y para abreviar código he querido ostrar también la ejecución de la tarea2C Del AdaboostClassifier parametrizando el estimador DecisiónTreeClassifier.

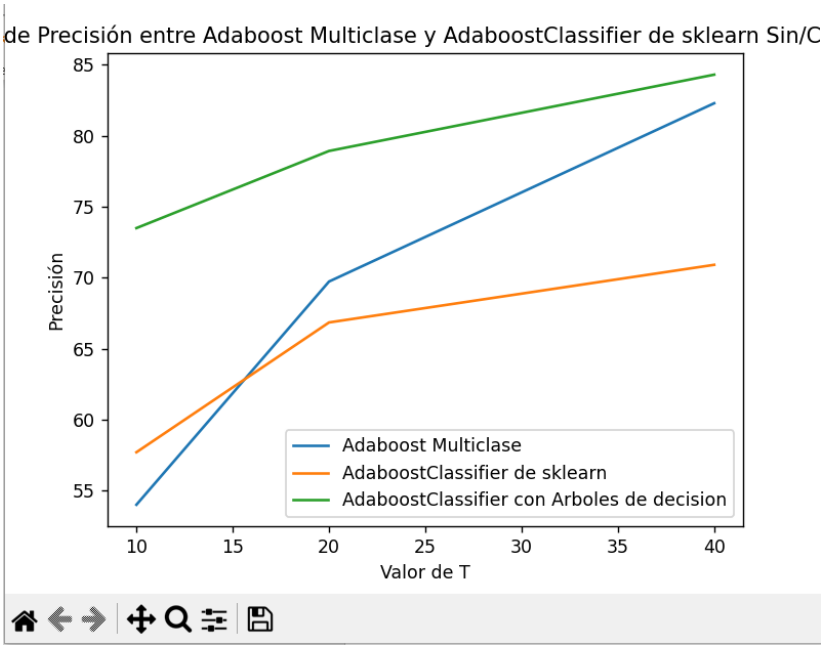
Dado que en el AdaboostClassifier con n_estimators = 50, daba una precisión un poco baja, al principio puse valores altos para realizar la comparación y que fuera acertada, como [10, 50, 100, 150], pero la ejecución del programa llegaba a los 20 minutos de ejecución, por lo que cogi los valores por defecto que te indican en el enunciado de T = [10,20,40] y A = [10,20,40], siendo para mi valores bajo para el AdaboostClassifier pero dado que lo que se pretende es comparar que para unos mismos valores, como es de eficaz cada técnica, da lo mismo, ya que cuando haya que tomar una elección se cogerá la mejor técnica, es decir la más eficaz.

Imagen de la terminal: Ejemplo de lo que estaba tardando el AdaboostClassifier

```
              4       0.83      0.85      0.84      1295
              5       0.86      0.80      0.83      1273
              6       0.93      0.95      0.94      1396
              7       0.89      0.83      0.86      1503
              8       0.77      0.86      0.81      1357
              9       0.74      0.79      0.76      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 173.122 s
La precisión del modelo es del 89.19%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1343
           1       0.98      0.98      0.98      1600
           2       0.87      0.88      0.87      1380
           3       0.87      0.83      0.85      1433
           4       0.87      0.86      0.87      1295
           5       0.87      0.85      0.86      1273
           6       0.95      0.96      0.95      1396
           7       0.91      0.88      0.89      1503
           8       0.82      0.90      0.86      1357
           9       0.82      0.81      0.82      1420

    accuracy                           0.89     14000
   macro avg       0.89      0.89      0.89     14000
weighted avg       0.89      0.89      0.89     14000

Tiempo: 345.887 s
Traceback (most recent call last):
```

Implementación:

Ejecuto para cada valor de T (y A en caso de que se necesite), las tareas 1D(Adaboost Multiclase), 2A(AdaboostClassifier de sklearn) y la 2C(AdaboostClassifier de sklearn con DecisionTreeClassifier).



Por lo que podemos observar el Adaboost Multiclase comienza con una precisión menor al Adaboostclassifier de sklearn, pero conforme va creciendo la T, el multiclase rápidamente incrementa su precisión notablemente frente al de sklearn, aunque es inevitable pensar que a números mayores como n=100, el de sklearn poblabemente podría obtener una mejor precisión pero sin embargo se juzgaría su eficacia debido al excesivo tiempo de ejecución que tarda, por lo que podemos sacar la conclusión de que el multiclase es más eficaz que el de sklearn.

Sin embargo, se ha obviado la reflexión sobre la gran precisión de AdaboostClassifier con DecisiónTreeClassifier, que parte desde el inicio con una precisión mucho más alta que la de las otras dos técnicas y en un tiempo que se considera poco. En la siguiente tarea se explicará este último y el por qué de sus resultados.

Traza

PS C:\Users\paula\Desktop\SI\Prácticas\Practica2> python .\Paula_Galvez_Romero_de_Avila.py

Tarea 2B: Comparando el adaboost multiclase con los dos adaboost classifier implementado o no el DecisionTree

Adaboost Multiclase T = 10, A = 10

Entrenando clasificador Adaboost para el dígito 0, T=10, A=10

Tasas acierto (train, test) y tiempo: 86.85%, 82.90%, 0.320 s

Matriz de Confusión:

[[7442 1578]

[132 848]]

Entrenando clasificador Adaboost para el dígito 1, T=10, A=10

Tasas acierto (train, test) y tiempo: 85.90%, 82.63%, 0.204 s

Matriz de Confusión:

[[7204 1661]

[76 1059]]

Entrenando clasificador Adaboost para el dígito 2, T=10, A=10

Tasas acierto (train, test) y tiempo: 79.04%, 76.43%, 0.320 s

Matriz de Confusión:

[[6792 2176]

[181 851]]

Entrenando clasificador Adaboost para el dígito 3, T=10, A=10

Tasas acierto (train, test) y tiempo: 79.10%, 82.52%, 0.220 s

Matriz de Confusión:

[[7484 1506]

[242 768]]

Entrenando clasificador Adaboost para el dígito 4, T=10, A=10

Tasas acierto (train, test) y tiempo: 82.63%, 81.63%, 0.228 s

Matriz de Confusión:

[[7339 1679]

[158 824]]

Entrenando clasificador Adaboost para el dígito 5, T=10, A=10

Tasas acierto (train, test) y tiempo: 70.95%, 71.16%, 0.204 s

Matriz de Confusión:

[[6499 2609]

[275 617]]

Entrenando clasificador Adaboost para el dígito 6, T=10, A=10

Tasas acierto (train, test) y tiempo: 86.98%, 87.35%, 0.268 s

Matriz de Confusión:

[[7911 1131]

[134 824]]

Entrenando clasificador Adaboost para el dígito 7, T=10, A=10

Tasas acierto (train, test) y tiempo: 81.45%, 76.25%, 0.296 s

Matriz de Confusión:

[[6713 2259]

[116 912]]

Entrenando clasificador Adaboost para el dígito 8, T=10, A=10

Tasas acierto (train, test) y tiempo: 72.63%, 67.53%, 0.204 s

Matriz de Confusión:

[[6013 3013]

[234 740]]

Entrenando clasificador Adaboost para el dígito 9, T=10, A=10

Tasas acierto (train, test) y tiempo: 69.92%, 71.30%, 0.348 s

Matriz de Confusión:

[[6428 2563]

[307 702]]

Matriz de Confusión para el clasificador multiclase:

[[768 25 80 5 5 26 45 12 13 1]

[3 1040 2 40 0 5 1 11 16 17]

[87 92 482 85 32 13 87 85 63 6]

[80 74 54 593 12 21 16 27 101 32]

[7 97 16 23 428 21 97 52 37 204]

[131 159 27 154 37 203 31 26 55 69]

[82 56 112 22 41 2 613 12 6 12]

[14 74 7 22 42 31 5 649 4 180]

[108 85 32 164 22 34 84 74 328 43]

[22 84 4 63 355 18 16 87 62 298]]

Precisión del clasificador multiclase: 54.02%

Adaboost Multiclase T = 20, A = 20

Entrenando clasificador Adaboost para el dígito 0, T=20, A=20

Tasas acierto (train, test) y tiempo: 94.21%, 92.82%, 0.896 s

Matriz de Confusión:

[[8339 681]

[37 943]]

Entrenando clasificador Adaboost para el dígito 1, T=20, A=20

Tasas acierto (train, test) y tiempo: 91.88%, 89.20%, 0.835 s

Matriz de Confusión:

[[7822 1043]

[37 1098]]

Entrenando clasificador Adaboost para el dígito 2, T=20, A=20

Tasas acierto (train, test) y tiempo: 86.16%, 85.21%, 0.768 s

Matriz de Confusión:

[[7617 1351]

[128 904]]

Entrenando clasificador Adaboost para el dígito 3, T=20, A=20

Tasas acierto (train, test) y tiempo: 83.38%, 80.68%, 0.713 s

Matriz de Confusión:

[[7194 1796]

 [136 874]]

Entrenando clasificador Adaboost para el dígito 4, T=20, A=20

Tasas acierto (train, test) y tiempo: 85.83%, 84.79%, 0.908 s

Matriz de Confusión:

[[7655 1363]

 [158 824]]

Entrenando clasificador Adaboost para el dígito 5, T=20, A=20

Tasas acierto (train, test) y tiempo: 75.98%, 75.67%, 0.924 s

Matriz de Confusión:

[[6925 2183]

 [250 642]]

Entrenando clasificador Adaboost para el dígito 6, T=20, A=20

Tasas acierto (train, test) y tiempo: 89.94%, 90.62%, 0.866 s

Matriz de Confusión:

[[8266 776]

 [162 796]]

Entrenando clasificador Adaboost para el dígito 7, T=20, A=20

Tasas acierto (train, test) y tiempo: 88.12%, 87.98%, 0.857 s

Matriz de Confusión:

[[7899 1073]

 [129 899]]

Entrenando clasificador Adaboost para el dígito 8, T=20, A=20

Tasas acierto (train, test) y tiempo: 82.18%, 79.65%, 0.879 s

Matriz de Confusión:

[[7150 1876]

 [159 815]]

Entrenando clasificador Adaboost para el dígito 9, T=20, A=20

Tasas acierto (train, test) y tiempo: 81.84%, 78.83%, 0.771 s

Matriz de Confusión:

[[7011 1980]

 [137 872]]

Matriz de Confusión para el clasificador multiclase:

[[870 1 9 8 6 52 19 9 6 0]

[0 1033 23 4 0 0 7 21 47 0]

[26 91 700 45 22 13 29 31 63 12]

[17 24 20 764 7 61 11 29 59 18]

[7 32 6 12 540 60 26 21 19 259]

[31 29 10 129 73 371 38 37 97 77]

[66 26 120 7 54 26 618 9 29 3]

[5 54 36 22 13 11 4 811 9 63]

[14 64 31 94 34 39 14 24 615 45]

[26 23 14 40 100 26 3 98 28 651]]

Precisión del clasificador multiclase: 69.73%

Adaboost Multiclase T = 40, A = 40

Entrenando clasificador Adaboost para el dígito 0, T=40, A=40

Tasas acierto (train, test) y tiempo: 96.12%, 96.05%, 2.812 s

Matriz de Confusión:

[[8664 356]

[39 941]]

Entrenando clasificador Adaboost para el dígito 1, T=40, A=40

Tasas acierto (train, test) y tiempo: 96.20%, 95.32%, 2.655 s

Matriz de Confusión:

[[8415 450]

[18 1117]]

Entrenando clasificador Adaboost para el dígito 2, T=40, A=40

Tasas acierto (train, test) y tiempo: 91.71%, 91.02%, 2.918 s

Matriz de Confusión:

[[8176 792]

[106 926]]

Entrenando clasificador Adaboost para el dígito 3, T=40, A=40

Tasas acierto (train, test) y tiempo: 91.12%, 89.77%, 2.929 s

Matriz de Confusión:

[[8059 931]

[92 918]]

Entrenando clasificador Adaboost para el dígito 4, T=40, A=40

Tasas acierto (train, test) y tiempo: 90.82%, 89.11%, 2.883 s

Matriz de Confusión:

[[8027 991]

 [98 884]]

Entrenando clasificador Adaboost para el dígito 5, T=40, A=40

Tasas acierto (train, test) y tiempo: 87.91%, 88.23%, 2.959 s

Matriz de Confusión:

[[8090 1018]

 [159 733]]

Entrenando clasificador Adaboost para el dígito 6, T=40, A=40

Tasas acierto (train, test) y tiempo: 94.63%, 93.39%, 2.848 s

Matriz de Confusión:

[[8446 596]

 [65 893]]

Entrenando clasificador Adaboost para el dígito 7, T=40, A=40

Tasas acierto (train, test) y tiempo: 94.14%, 93.90%, 2.742 s

Matriz de Confusión:

[[8440 532]

 [78 950]]

Entrenando clasificador Adaboost para el dígito 8, T=40, A=40

Tasas acierto (train, test) y tiempo: 89.31%, 87.82%, 2.875 s

Matriz de Confusión:

[[7944 1082]

 [136 838]]

Entrenando clasificador Adaboost para el dígito 9, T=40, A=40

Tasas acierto (train, test) y tiempo: 86.83%, 84.01%, 2.897 s

Matriz de Confusión:

[[7512 1479]

 [120 889]]

Matriz de Confusión para el clasificador multiclase:

[[914   1   9   5   0   16   14   11   4   6]

 [0 1111   7   2   3   0   4   1   7   0]

 [19   46 793   33   23   5   25   21   56   11]

 [17   14   23 799   8   43   16   21   42   27]

 [4   7   3   6 798   9   36   6   7 106]

 [24   18   8   80   32 594   20   26   50   40]

 [22   10   27   4   24   27 829   2   11   2]

[4 31 32 9 25 1 1 871 3 51]

[11 26 14 34 15 39 20 10 755 50]

[9 14 8 14 90 13 2 74 21 764]]

Precisión del clasificador multiclase: 82.28%

Adaboostclassifier con n_estimators = 10

Matriz de Confusión para el clasificador multiclase:

[[1157 0 22 9 1 72 41 3 27 11]

[0 1021 46 452 4 2 2 60 11 2]

[43 101 911 46 23 13 65 36 131 11]

[15 83 47 986 7 88 30 71 69 37]

[6 1 42 34 536 12 88 287 43 246]

[42 32 30 351 24 402 111 135 72 74]

[61 45 193 21 74 32 901 15 47 7]

[9 34 46 22 48 121 35 1020 23 145]

[28 134 60 203 26 31 54 35 754 32]

[4 19 58 69 340 17 58 418 45 392]]

La precisión del modelo es del 57.71%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.85	0.86	0.85	1343
1	0.69	0.64	0.67	1600
2	0.63	0.66	0.64	1380
3	0.45	0.69	0.54	1433
4	0.49	0.41	0.45	1295
5	0.51	0.32	0.39	1273
6	0.65	0.65	0.65	1396
7	0.49	0.68	0.57	1503
8	0.62	0.56	0.58	1357
9	0.41	0.28	0.33	1420
accuracy			0.58	14000
macro avg	0.58	0.57	0.57	14000

weighted avg 0.58 0.58 0.57 14000

Tiempo: 8.695 s

Adaboostclassifier con n_estimators = 20

Matriz de Confusión para el clasificador multiclase:

[[1103 0 56 7 1 106 33 10 22 5]

[0 1440 60 28 4 6 1 49 11 1]

[31 101 927 30 28 22 87 50 91 13]

[40 48 58 943 18 108 15 76 89 38]

[1 2 41 34 720 27 32 100 42 296]

[42 25 25 261 38 686 48 30 79 39]

[42 32 185 23 115 38 903 5 51 2]

[13 21 35 22 43 29 0 1198 20 122]

[21 90 76 169 47 45 26 24 831 28]

[3 18 66 82 318 12 3 248 62 608]]

La precisión del modelo es del 66.85%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.85	0.82	0.84	1343
1	0.81	0.90	0.85	1600
2	0.61	0.67	0.64	1380
3	0.59	0.66	0.62	1433
4	0.54	0.56	0.55	1295
5	0.64	0.54	0.58	1273
6	0.79	0.65	0.71	1396
7	0.67	0.80	0.73	1503
8	0.64	0.61	0.63	1357
9	0.53	0.43	0.47	1420
accuracy			0.67	14000
macro avg	0.67	0.66	0.66	14000
weighted avg	0.67	0.67	0.67	14000

Tiempo: 18.276 s

Adaboostclassifier con n_estimators = 40

Matriz de Confusión para el clasificador multiclase:

[[1154 0 78 5 4 42 38 5 13 4]
[0 1485 19 28 3 8 5 37 14 1]
[29 53 723 57 24 11 370 44 55 14]
[40 62 20 918 11 92 55 73 113 49]
[9 5 41 18 744 28 17 139 44 250]
[46 38 23 181 30 731 57 34 82 51]
[30 33 105 16 26 43 1115 1 24 3]
[12 16 16 16 22 11 0 1229 16 165]
[17 103 40 145 18 53 39 21 878 43]
[6 21 31 31 115 20 0 188 58 950]]

La precisión del modelo es del 70.91%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.86	0.86	0.86	1343
1	0.82	0.93	0.87	1600
2	0.66	0.52	0.58	1380
3	0.65	0.64	0.64	1433
4	0.75	0.57	0.65	1295
5	0.70	0.57	0.63	1273
6	0.66	0.80	0.72	1396
7	0.69	0.82	0.75	1503
8	0.68	0.65	0.66	1357
9	0.62	0.67	0.64	1420
accuracy			0.71	14000
macro avg	0.71	0.70	0.70	14000
weighted avg	0.71	0.71	0.70	14000

Tiempo: 34.908 s

Adaboostclassifier ajustando DecisionTree con n_estimators = 10 y n_features = 10

Matriz de Confusión para el clasificador multiclase:

[[1033 1 45 7 16 184 33 3 11 10]

[0 1506 10 12 7 5 1 25 27 7]

[13 40 935 101 33 82 65 31 71 9]

[6 28 65 1007 8 92 13 53 119 42]

[9 5 14 11 858 30 44 57 27 240]

[9 11 17 187 41 815 29 11 106 47]

[37 15 78 10 92 41 1095 2 13 13]

[2 19 33 21 66 10 3 1210 20 119]

[16 37 24 117 24 103 17 26 894 99]

[7 16 15 19 220 8 3 158 38 936]]

La precisión del modelo es del 73.49%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.91	0.77	0.83	1343
1	0.90	0.94	0.92	1600
2	0.76	0.68	0.71	1380
3	0.67	0.70	0.69	1433
4	0.63	0.66	0.65	1295
5	0.59	0.64	0.62	1273
6	0.84	0.78	0.81	1396
7	0.77	0.81	0.79	1503
8	0.67	0.66	0.67	1357
9	0.61	0.66	0.64	1420
accuracy			0.73	14000
macro avg	0.74	0.73	0.73	14000
weighted avg	0.74	0.73	0.74	14000

Tiempo: 3.492 s

Adaboostclassifier ajustando DecisionTree con n_estimators = 20 y n_features = 20

Matriz de Confusión para el clasificador multiclase:

[[1078 0 55 10 4 123 43 2 24 4]

[0 1533 7 17 4 3 7 5 22 2]

[5 15 1114 58 40 22 31 19 71 5]

[8 5 54 1125 9 118 5 16 53 40]

[2 1 38 7 988 18 4 27 21 189]

[5 6 31 232 21 866 11 5 63 33]

[10 5 141 18 36 47 1096 2 40 1]

[3 7 18 13 68 12 0 1191 18 173]

[8 13 62 105 16 58 7 16 1020 52]

[4 6 39 31 172 10 0 91 29 1038]]

La precisión del modelo es del 78.92%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.96	0.80	0.87	1343
1	0.96	0.96	0.96	1600
2	0.71	0.81	0.76	1380
3	0.70	0.79	0.74	1433
4	0.73	0.76	0.74	1295
5	0.68	0.68	0.68	1273
6	0.91	0.79	0.84	1396
7	0.87	0.79	0.83	1503
8	0.75	0.75	0.75	1357
9	0.68	0.73	0.70	1420
accuracy			0.79	14000
macro avg	0.79	0.79	0.79	14000
weighted avg	0.80	0.79	0.79	14000

Tiempo: 8.046 s

Adaboostclassifier ajustando DecisionTree con n_estimators = 40 y n_features = 40

Matriz de Confusión para el clasificador multiclase:

[[1250 0 22 3 2 31 10 0 20 5]

[0 1509 7 23 3 7 0 4 43 4]

```
[   7    4 1134   49   20   21   25    8  107    5]
[   4   10   54 1084    3  159    2   12   68   37]
[   2    0   36   16  985   15    8   26   20  187]
[   7    4   10  107    9 1038    8    1   67   22]
[   4    2   43    1   20   38 1255    2   27    4]
[   8    6   27   28   22    4    1 1192   13  202]
[   3   14   27   50   13   36   12    4 1173   25]
[   6    9   20   33   57    8    0   86   20 1181]]
```

La precisión del modelo es del 84.29%

El informe de clasificación es:

	precision	recall	f1-score	support
0	0.97	0.93	0.95	1343
1	0.97	0.94	0.96	1600
2	0.82	0.82	0.82	1380
3	0.78	0.76	0.77	1433
4	0.87	0.76	0.81	1295
5	0.76	0.82	0.79	1273
6	0.95	0.90	0.92	1396
7	0.89	0.79	0.84	1503
8	0.75	0.86	0.80	1357
9	0.71	0.83	0.76	1420
accuracy			0.84	14000
macro avg	0.85	0.84	0.84	14000
weighted avg	0.85	0.84	0.84	14000

Tiempo: 19.111 s

Tarea 2C: Sustituye el clasificador por árboles de decisión

La Tarea 2C implica la integración de un clasificador basado en árboles de decisión (DecisionTreeClassifier) dentro de AdaBoostClassifier de scikit-learn, aplicado al conjunto de datos MNIST. Esta tarea permite explorar las capacidades de los árboles de decisión más allá de su rol habitual como clasificadores débiles en algoritmos de ensamblaje.

Árboles de Decisión: Son modelos predictivos formados por nodos de decisión, ramas y nodos hoja. Cada nodo de decisión representa una característica del dato, la rama representa la decisión tomada, y el nodo hoja representa el resultado o la predicción.

En el proceso de decisión, cada nodo se basa en el valor de una característica. El árbol se bifurca en varias ramas hasta llegar a un nodo hoja. En tareas de clasificación, cada nodo hoja representa una clase.

Adaptación de los árboles de decisión en el AdaBoostClassifier

Se utiliza DecisionTreeClassifier como clasificador base por defecto en AdaBoostClassifier, con el parámetro por defecto max_depth=1, que significa la máxima profundidad, de manera que, un árbol más profundo puede capturar más complejidad, pero puede aumentar el riesgo de sobreajuste.

dt_clf = DecisionTreeClassifier(max_depth=1): Elige un árbol de decisión con una profundidad máxima de 1 (un árbol de un solo nivel), lo que implica que cada árbol base es bastante simple, esto lo podemos modificar para experimentar y observar los posibles cambios en el rendimiento.

Otras configuraciones en los parámetros, como min_samples_split o min_samples_leaf, también pueden ser ajustadas para experimentar con la complejidad del árbol.

AdaBoostClassifier(estimator=dt_clf, n_estimators=n, random_state=42): Se configura AdaBoost para utilizar el árbol de decisión definido como clasificador base, con n estimadores y una semilla aleatoria para la reproducibilidad, como ya se ha experimentado antes en la tarea 2A.

Evaluación del Modelo:

Se mide la precisión del modelo y se genera un informe de clasificación.

Se registra el tiempo de entrenamiento, que es crucial para evaluar la eficiencia del modelo.

La Tarea 2C demuestra cómo la integración de un DecisionTreeClassifier más complejo en AdaBoostClassifier puede afectar el rendimiento en una tarea de clasificación de dígitos manuscritos, y hablo de un DecisionTreeClassifier más complejo porque el AdaBoostClassifier, ya utiliza por defecto los árboles de decisión, “Si *None*, entonces el estimador base es DecisionTreeClassifier inicializado con max_depth=1.”, es decir, la versión más básica un árbol de decisión de máxima profundidad de 1.

Esta tarea sirve para ‘trastear’, experimentar entre los valores de todos los parámetros que se les puede indicar según este contexto. La experimentación con los parámetros del árbol de decisión ofrece una visión valiosa sobre cómo la complejidad del clasificador base influye en el rendimiento general del modelo de ensamblaje.

Experimentación

- Primero vamos a experimentar solamente variando la máxima profundidad

Experimento 1 Max_depth = 1 (La de por defecto)

Podemos observar que tarda relativamente poco, pero tiene una precisión no muy alta.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 71.65%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.85      0.90      0.87      1343
           1       0.85      0.94      0.89      1600
           2       0.71      0.55      0.62      1380
           3       0.67      0.65      0.66      1433
           4       0.74      0.58      0.65      1295
           5       0.68      0.59      0.63      1273
           6       0.69      0.82      0.75      1396
           7       0.72      0.70      0.71      1503
           8       0.70      0.69      0.69      1357
           9       0.56      0.70      0.62      1420

    accuracy                           0.72     14000
   macro avg       0.72      0.71      0.71     14000
weighted avg       0.72      0.72      0.71     14000

Tiempo: 43.408 s
```

Experimento 2 Max_ depth = 5

Tarda más o menos 3 minutos, 4 veces más que con max_depth = 1, aquí se cuestiona si merece la pena tener una precisión del 86% si va a tardar tanto.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 86.21%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.95      0.94      0.95      1343
           1       0.96      0.97      0.96      1600
           2       0.86      0.84      0.85      1380
           3       0.83      0.79      0.81      1433
           4       0.83      0.85      0.84      1295
           5       0.86      0.80      0.83      1273
           6       0.93      0.95      0.94      1396
           7       0.89      0.83      0.86      1503
           8       0.77      0.86      0.81      1357
           9       0.74      0.79      0.76      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 172.835 s
```

Experimento 3 Max_depth = 8

Tarda 5 minutos con aprox 31s, es decir tarda bastante, esto ya descarta totalmente lo eficaz a mi parecer, al menos en este caso, ya que tarda demasiado.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 92.94%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      1343
           1       0.98      0.98      0.98      1600
           2       0.93      0.94      0.93      1380
           3       0.91      0.89      0.90      1433
           4       0.91      0.92      0.92      1295
           5       0.91      0.91      0.91      1273
           6       0.97      0.96      0.97      1396
           7       0.94      0.92      0.93      1503
           8       0.88      0.92      0.90      1357
           9       0.87      0.89      0.88      1420

    accuracy                           0.93     14000
   macro avg       0.93      0.93      0.93     14000
weighted avg       0.93      0.93      0.93     14000

Tiempo: 318.630 s
```

Experimento 4 Max_depth = 10

Por curiosidad lo he probado con 10, 7 minutos con28s.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 95.21%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      1343
           1       0.99      0.98      0.98      1600
           2       0.93      0.95      0.94      1380
           3       0.93      0.93      0.93      1433
           4       0.95      0.95      0.95      1295
           5       0.95      0.94      0.95      1273
           6       0.98      0.97      0.98      1396
           7       0.96      0.94      0.95      1503
           8       0.92      0.94      0.93      1357
           9       0.92      0.93      0.93      1420

    accuracy                           0.95     14000
   macro avg       0.95      0.95      0.95     14000
weighted avg       0.95      0.95      0.95     14000

Tiempo: 436.780 s
```

Es inevitable sacar la conclusión de que cuanto mayor sea el valor de max_depth, mayor precisión tendrá, pero a que precio de tiempo, por lo que de momento vamos a escoger el valor de profundidad 5, ya que, aunque tarde 3 minutos, se consigue cierta precisión y se puede experimentar el uso de más parámetros para igual reducir el tiempo.

Sin embargo vamos a seguir experimentando por sus parámetros dejando de momento max_depth=5.

- Ahora vamos a añadir más parámetros

Experimento 1

Primeramente, ya que hay muchos parámetros y muchos posibles de valores y combinaciones, para evitar sobreajustes o malos resultados, he estado mirando por internet pero he terminado consultando el chatgpt, para preguntarle que combinación me recomendaba para este contexto.

Combinación / Configuración proporcionada por el chatGPT

```
dt_clf = DecisionTreeClassifier(
        criterion='entropy', # Uso de entropía para una mejor división
        max_depth=5,         # Profundidad máxima
        min_samples_split=4, # Mínimo de muestras para dividir
        min_samples_leaf=2,  # Mínimo de muestras en nodo hoja
        random_state=42
    )
```

Me ha comentado que para el contexto de la práctica es interesante utilizar y experimentar estos parámetros, variando los valores que tienen por defecto.

El tiempo dejecución ha sido un total de 3 minutos 58 s – 4 minutos, con una precisión del 90%

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 90.24%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1343
           1       0.98      0.97      0.98      1600
           2       0.89      0.89      0.89      1380
           3       0.87      0.87      0.87      1433
           4       0.91      0.89      0.90      1295
           5       0.87      0.86      0.87      1273
           6       0.97      0.92      0.95      1396
           7       0.94      0.88      0.91      1503
           8       0.81      0.90      0.86      1357
           9       0.82      0.88      0.85      1420

    accuracy                           0.90     14000
   macro avg       0.90      0.90      0.90     14000
weighted avg       0.90      0.90      0.90     14000

Tiempo: 215.256 s
```

Experimento2

```
dt_clf = DecisionTreeClassifier(
        criterion='entropy', # Uso de entropía para una mejor división
        max_depth=5,         # Profundidad máxima
        random_state=42
    )
```

Ya que el valor por defecto de ‘criterion’ es ‘gini’, vamos a probar con su otro posible valor ‘entropy’.

Criterion: Define la función para medir la calidad de una división. Las opciones incluyen "gini" para la impureza de Gini, y "entropy" para la ganancia de información de Shannon.

‘entropy’ a menudo es preferido para tareas de clasificación.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 86.21%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.95      0.94      0.95      1343
           1       0.96      0.97      0.96      1600
           2       0.86      0.84      0.85      1380
           3       0.83      0.79      0.81      1433
           4       0.83      0.85      0.84      1295
           5       0.86      0.80      0.83      1273
           6       0.93      0.95      0.94      1396
           7       0.89      0.83      0.86      1503
           8       0.77      0.86      0.81      1357
           9       0.74      0.79      0.76      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 172.835 s
```

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 90.15%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      1343
           1       0.98      0.97      0.97      1600
           2       0.90      0.89      0.89      1380
           3       0.88      0.86      0.87      1433
           4       0.90      0.89      0.89      1295
           5       0.86      0.87      0.86      1273
           6       0.98      0.93      0.95      1396
           7       0.92      0.88      0.90      1503
           8       0.82      0.91      0.86      1357
           9       0.82      0.86      0.84      1420

    accuracy                           0.90     14000
   macro avg       0.90      0.90      0.90     14000
weighted avg       0.90      0.90      0.90     14000

Tiempo: 213.521 s
```

La imagen de la izquierda es con criterion = ‘gini’, y la de la derecha criterion = ‘entrophy’.

Aunque la segunda imagen tenga más precisión, tarda demasiado tiempo por lo que por la poca diferencia en la precisión yo escogería seguir utilizando la de por defecto, osea ‘gini’

Experimento 3

```python
dt_clf = DecisionTreeClassifier(
        criterion='gini', # Uso de entropía para una mejor división
        max_depth=5,         # Profundidad máxima
        min_samples_split=4, # Mínimo de muestras para dividir
        min_samples_leaf=2,  # Mínimo de muestras en nodo hoja
        random_state=42
    )
```

Por defecto el mínimo de muestras para dividir es de 2, y mínimo por nodo hoja es de 1. Vamos a probar a duplicarlo.

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 86.21%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.95      0.94      0.95      1343
           1       0.96      0.97      0.96      1600
           2       0.86      0.84      0.85      1380
           3       0.83      0.79      0.81      1433
           4       0.83      0.85      0.84      1295
           5       0.86      0.80      0.83      1273
           6       0.93      0.95      0.94      1396
           7       0.89      0.83      0.86      1503
           8       0.77      0.86      0.81      1357
           9       0.74      0.79      0.76      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 172.835 s
```

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 86.11%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.93      0.93      0.93      1343
           1       0.97      0.96      0.96      1600
           2       0.83      0.84      0.84      1380
           3       0.81      0.80      0.81      1433
           4       0.86      0.81      0.83      1295
           5       0.85      0.81      0.83      1273
           6       0.95      0.93      0.94      1396
           7       0.91      0.83      0.87      1503
           8       0.79      0.84      0.82      1357
           9       0.72      0.83      0.77      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 174.269 s
```

La imagen de la izquierda son los por defecto, y la de la derecha los resultados del experimento.

Sorprendentemente para mí la diferencia es mínima, ya que la precisión baja en un 0,11%, y tarda dos segundos más en ejecutarse solo. Igualmente sigo considerando mejor los valores por defecto.

Experimento 4

Igualmente, ya que hemos obtenido resultados bastante similares vamos a probar a reducir la máxima profundidad a 3.

```
dt_clf = DecisionTreeClassifier(
        criterion='gini', # Uso de entropía para una mejor división
        max_depth=3,         # Profundidad máxima
        min_samples_split=4, # Mínimo de muestras para dividir
        min_samples_leaf=2,  # Mínimo de muestras en nodo hoja
        random_state=42
    )
```

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 86.21%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.95      0.94      0.95      1343
           1       0.96      0.97      0.96      1600
           2       0.86      0.84      0.85      1380
           3       0.83      0.79      0.81      1433
           4       0.83      0.85      0.84      1295
           5       0.86      0.80      0.83      1273
           6       0.93      0.95      0.94      1396
           7       0.89      0.83      0.86      1503
           8       0.77      0.86      0.81      1357
           9       0.74      0.79      0.76      1420

    accuracy                           0.86     14000
   macro avg       0.86      0.86      0.86     14000
weighted avg       0.86      0.86      0.86     14000

Tiempo: 172.835 s
```

```
Tarea 2C: Adaboostclassifier con DecisionTree
La precisión del modelo es del 81.31%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.92      0.87      0.89      1343
           1       0.93      0.96      0.94      1600
           2       0.79      0.62      0.69      1380
           3       0.79      0.74      0.76      1433
           4       0.85      0.84      0.84      1295
           5       0.78      0.77      0.78      1273
           6       0.82      0.91      0.86      1396
           7       0.89      0.76      0.82      1503
           8       0.68      0.79      0.73      1357
           9       0.72      0.86      0.78      1420

    accuracy                           0.81     14000
   macro avg       0.82      0.81      0.81     14000
weighted avg       0.82      0.81      0.81     14000

Tiempo: 102.809 s
```

De nuevo, la imagen de la izquierda, son los valores por defecto (min_samples_split=2, min_samples_leaf=1 y criterion='gini') y max_depth=5, y la imagen de la derecha los resultados del experimento.

Como podemos ver, este experimento se realiza en menos tiempo pero se consigue yn 5% menos de precisión.

Experimento 5

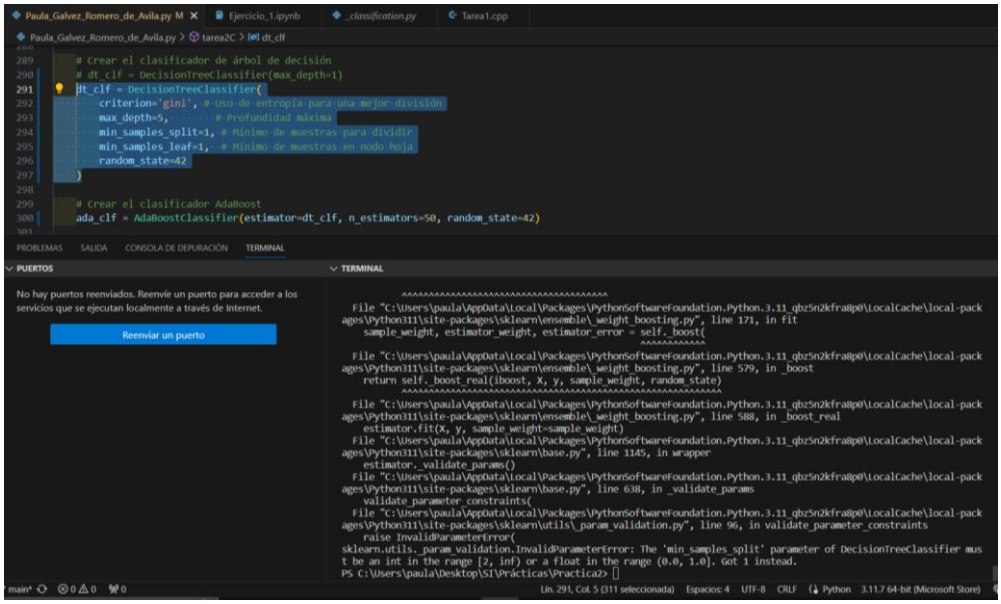
¿Qué pasa si el mínimo de muestras es un decimal? -> ERROR, porque no se puede tener media muestra en un nodo hoja

```
dt_clf = DecisionTreeClassifier(
        criterion='gini', # Uso de entropía para una mejor división
        max_depth=5,         # Profundidad máxima
        min_samples_split=1, # Mínimo de muestras para dividir
        min_samples_leaf=0.5,  # Mínimo de muestras en nodo hoja
        random_state=42
    )
```

Experimento 6

¿Y que pasa si el mínimo de muestras para dividir y el mínimo de muestras en cada nodo hoja es el mismo? -> ERROR

```
dt_clf = DecisionTreeClassifier(
        criterion='gini', # Uso de entropía para una mejor división
        max_depth=5,        # Profundidad máxima
        min_samples_split=1, # Mínimo de muestras para dividir
        min_samples_leaf=1,  # Mínimo de muestras en nodo hoja
        random_state=42
    )
```



Conclusión hasta este apunto: Debido a que los resultados de los experimentos han sido iguales o peores, considero dejar todos los valores por defecto menos max_depth a la que le asignaré el valor 5.

- Experimentación con el parámetro max_features

Este parámetro indica el número de características a considerar al buscar la mejor división:

- Si int, entonces considere max_features características en cada división.
- Si floqqat, entonces max_features es una fracción y las características se consideran en cada división.max(1, int(max_features * n_features_in_))
- Si “sqrt”, entonces max_features=sqrt(n_features).
- Si “log2”, entonces max_features=log2(n_features).
- Si Ninguno, entonces max_features=n_features.

La búsqueda de una división no se detiene hasta al menos una se encuentra una partición válida de las muestras de nodo, incluso si es necesario inspeccionar efectivamente más que max_features características.

¿Que pasa si a max_features le damos el valor de lo que sería la A?.

Experimento n_estimators = 20 y en el árbol de decisión, max_features=A=10.

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 20 y n_features = 10
Matriz de Confusión para el clasificador multiclase:
[[ 986    0   86    9    4  188   40    6   17    7]
 [   0 1511    8   13    0    4    2   30   29    3]
 [   5   22 1059   54   23   47   49   17   99    5]
 [   5   16   57  929   10  184    6   31  148   47]
 [   3    6   31   17  863   27   14   35   27  272]
 [   1    3   18   92   19  988   24    6   97   25]
 [   7    8  151    4   33   45 1124    1   20    3]
 [   3    9   29   15   55   18    0 1220    9  145]
 [   5   17   26   73   19   64   10   18 1038   87]
 [   3    5   20   21  182    5    1  141   24 1018]]

La precisión del modelo es del 76.69%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.97      0.73      0.84      1343
           1       0.95      0.94      0.95      1600
           2       0.71      0.77      0.74      1380
           3       0.76      0.65      0.70      1433
           4       0.71      0.67      0.69      1295
           5       0.63      0.78      0.70      1273
           6       0.89      0.81      0.84      1396
           7       0.81      0.81      0.81      1503
           8       0.69      0.76      0.72      1357
           9       0.63      0.72      0.67      1420

    accuracy                           0.77     14000
   macro avg       0.77      0.76      0.77     14000
weighted avg       0.78      0.77      0.77     14000

Tiempo: 6.850 s
```

Da una precisión relativamente baja pero un buen tiempo de ejecución.

Experimento n_estimators = 50 y en el árbol de decisión, max_features=A=10.

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 50 y n_features = 10
Matriz de Confusión para el clasificador multiclase:
[[1130    0   35    4    4  124   22    1   19    4]
 [   0 1538    7   11    4    2    2    6   28    2]
 [   8   11 1159   55   14   24   35   12   57    5]
 [  18   12   40 1124    6   79    2   18   92   42]
 [   2    2   29    6  996   16   14   10   16  204]
 [  10    3   11   49   12 1001   13    0  137   37]
 [   5    4   79    4   13   37 1235    2   17    0]
 [   1    4   25   15   39   10    0 1287   10  112]
 [   8   12   17   77   13   59   13   11 1098   49]
 [   5    4   14   14  125    6    0   75   27 1150]]

La precisión del modelo es del 83.7%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.95      0.84      0.89      1343
           1       0.97      0.96      0.96      1600
           2       0.82      0.84      0.83      1380
           3       0.83      0.78      0.81      1433
           4       0.81      0.77      0.79      1295
           5       0.74      0.79      0.76      1273
           6       0.92      0.88      0.90      1396
           7       0.91      0.86      0.88      1503
           9       0.72      0.81      0.76      1420

    accuracy                           0.84     14000
   macro avg       0.84      0.83      0.84     14000
weighted avg       0.84      0.84      0.84     14000

Tiempo: 17.005 s
```

La precisión ha subido, pero el tiempo también, y aún no lo podemos considerar buen resultado.

Experimento n_estimators = 50 y en el árbol de decisión, max_features=A=20.

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 50 y n_features = 20
Matriz de Confusión para el clasificador multiclase:
[[1223    0   20    2    1   49   22    2   15    9]
 [   0 1532    6   13    2    4    1    9   27    6]
 [   5    9 1186   33   16   17   40   17   48    9]
 [  11    4   64 1081    6  134    3   13   84   33]
 [   4    0   37    6  969   14    7   19   19  220]
 [   9    2   22   85   10 1035    7    5   69   29]
 [  12    5  113    4   10   38 1198    3   13    0]
 [   1    2   21   14   58    4    0 1167    9  227]
 [   2    7   40   70   11   51    2    6 1126   42]
 [   2    6   31   21  208    5    1   53   18 1075]]

La precisión del modelo es del 82.8%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.96      0.91      0.94      1343
           1       0.98      0.96      0.97      1600
           2       0.77      0.86      0.81      1380
           3       0.81      0.75      0.78      1433
           4       0.75      0.75      0.75      1295
           5       0.77      0.81      0.79      1273
           6       0.94      0.86      0.90      1396
           7       0.90      0.78      0.83      1503
           8       0.79      0.83      0.81      1357
           9       0.65      0.76      0.70      1420

    accuracy                           0.83     14000
   macro avg       0.83      0.83      0.83     14000
weighted avg       0.84      0.83      0.83     14000

Tiempo: 19.019 s
```

La precisión se ha reducido un poco y el tiempo ha crecido.

Experimento n_estimators = 50 y en el árbol de decisión, max_features=A=5.

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 50 y n_features = 5
Matriz de Confusión para el clasificador multiclase:
[[1162    0   29   14    6   74   35    2   13    8]
 [   0 1474   15   14    3    1    7    3   75    8]
 [  10   11 1089   44   17   26   46   26  105    6]
 [  16   11   61 1113    7   45    8   21  100   51]
 [   2    1   43    3 1077   13    6   22   20  108]
 [  19    5   26  122   19  930   38    8   89   17]
 [  10    3   55    0   27   35 1248    2   16    0]
 [   1    5   14   17   54   13    1 1110   16  272]
 [  10   17   44   75   27   72   21   17 1020   54]
 [   8    8   15   16  185   11    2   88   30 1057]]

La precisión del modelo es del 80.57%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.94      0.87      0.90      1343
           1       0.96      0.92      0.94      1600
           2       0.78      0.79      0.79      1380
           3       0.78      0.78      0.78      1433
           4       0.76      0.83      0.79      1295
           5       0.76      0.73      0.75      1273
           6       0.88      0.89      0.89      1396
           7       0.85      0.74      0.79      1503
           8       0.69      0.75      0.72      1357
           9       0.67      0.74      0.70      1420

    accuracy                           0.81     14000
   macro avg       0.81      0.80      0.81     14000
weighted avg       0.81      0.81      0.81     14000

Tiempo: 15.821 s
```

El tiempo ha bajado pero la precisión también.

¿Pero que pasa si incrementamos ahora la máxima profundidad a 10?

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 50 y n_features = 5
Matriz de Confusión para el clasificador multiclase:
[[1300    0    6    4    4   10    4    2   12    1]
 [   0 1558    8    7    4    3    1    4   13    2]
 [   1    4 1288    9   10    1    9    7   48    3]
 [   2    0   25 1296    2   35    2   11   38   22]
 [   1    0    5    3 1176    2    5    4    7   92]
 [   9    2    6   62    4 1133    7    1   39   10]
 [   6    0    6    1   10   15 1348    0   10    0]
 [   1    3   16    4   20    1    0 1363   12   83]
 [   2    4   14   34    6   26    2    3 1257    9]
 [   5    2    7   13   40    5    1   27   28 1292]]

La precisión del modelo es del 92.94%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.98      0.97      0.97      1343
           1       0.99      0.97      0.98      1600
           2       0.93      0.93      0.93      1380
           3       0.90      0.90      0.90      1433
           4       0.92      0.91      0.91      1295
           5       0.92      0.89      0.90      1273
           6       0.98      0.97      0.97      1396
           7       0.96      0.91      0.93      1503
           8       0.86      0.93      0.89      1357
           9       0.85      0.91      0.88      1420

    accuracy                           0.93     14000
   macro avg       0.93      0.93      0.93     14000
weighted avg       0.93      0.93      0.93     14000

Tiempo: 18.037 s
```

Podemos ver que en un tiempo considerado bueno, por el que ya se rondaba, se ha conseguido una muy buena precisión.

Volvemos a hacer n_estimators = 20 y n_features = 10, ya que daba una precisión baja para ver que sucede.

```
Adaboostclassifier ajustando DecisionTree con n_estimators = 20 y n_features = 10
Matriz de Confusión para el clasificador multiclase:
[[1282    0   11    4    1   21    7    2   12    3]
 [   0 1545   11   12    3    3    1    2   22    1]
 [   6    4 1250   20   12    8   11   14   50    5]
 [   5    5   35 1223    2   64    5   14   59   21]
 [   1    0   19    1 1104    4    4   11   12  139]
 [   6    1    7  104    5 1085    5    0   48   12]
 [   5    0   17    0    7   20 1336    0   10    1]
 [   2    3   22    2   22    6    0 1302   15  129]
 [   3    3   19   55    7   33   12    8 1192   25]
 [   3    5    8   16  117    8    1   40   21 1201]]

La precisión del modelo es del 89.43%
El informe de clasificación es:
              precision    recall  f1-score   support

           0       0.98      0.95      0.97      1343
           1       0.99      0.97      0.98      1600
           2       0.89      0.91      0.90      1380
           3       0.85      0.85      0.85      1433
           4       0.86      0.85      0.86      1295
           5       0.87      0.85      0.86      1273
           6       0.97      0.96      0.96      1396
           7       0.93      0.87      0.90      1503
           8       0.83      0.88      0.85      1357
           9       0.78      0.85      0.81      1420

    accuracy                           0.89     14000
   macro avg       0.89      0.89      0.89     14000
weighted avg       0.90      0.89      0.89     14000

Tiempo: 8.558 s
```

Definitivamente son buenos resultados, ya que en poco tiempo se ha conseguido una buena precisión.

Por lo que con esto podemos concluir que la experimentación en los parámetros, en concreto en el incremento de max_depth y max_features=A, dan muy buenos resultados y se puede considerar una técnica fiable, como ya se vio en la tarea anterior, ya que son importantes para controlar la complejidad de los árboles individuales en el ensamble de AdaBoost. Un equilibrio adecuado en estos parámetros puede llevar a un modelo que no solo aprende las características generales de los dígitos, sino que también es robusto frente a nuevas imágenes, mejorando así la eficacia del modelo.

Tarea 2D

La Tarea 2D implica modelar un clasificador de Perceptrón Multicapa (MLP) para el conjunto de datos MNIST utilizando la biblioteca Keras en Python. Esta tarea se centra en implementar, entrenar y evaluar un MLP, que es una red neuronal artificial compuesta por múltiples capas linealmente apiladas.

Como se ha visto en clase, un perceptrón es una neurona artificial que toma una serie de entradas x y produce una salida, es decir, toma una decisión(salida) ponderando (w) una serie de factores(x). Podemos representar las entradas y pesos como tuplas, pero hay que destacar que para la toma de decisión se tiene en cuenta el bias (b), que nos indica la facilidad de que un perceptrón se dispare y b = -umbral.

$$\text{salida} = \begin{cases} 0 & \text{si } w \cdot x + b \leq 0 \\ 1 & \text{si } w \cdot x + b > 0 \end{cases}$$

Podemos combinar varios perceptrones conectados entre si para implementa funciones más complejas, de ahí viene el perceptrón multicapa (MLP), que es una red neuronal básica compuesta por una capa de entrada, varias capas ocultas y una capa de salida. Cada capa contiene un número definido de neuronas (o nodos), y cada neurona en una capa está conectada a todas las neuronas de la capa siguiente, por lo que se podría decir que los MLP utilizan una combinación de transformaciones lineales y funciones de activación no lineales para aprender patrones complejos en los datos.

A mayor número de capas, podrá tomar decisiones más complejas.

Como se ha solicitado en la práctica, este año no hace falta implementar a fondo un perceptrón multicapa, si no que vamos a utilizar la librería Keras para utilizar su implementación.

Estructura de la red neuronal:

- Modelo Secuencial
  - Capa de Entrada: Determinada por las dimensiones de los datos de entrada. Para MNIST, esto es 784 (28x28 píxeles aplanados).
  - Capas Ocultas: Se puede empezar con una sola capa oculta, como va a ser nuestro caso. La cantidad de neuronas y el número de capas ocultas pueden variar según la complejidad del problema, y como vamos a empezar con solo una capa oculta, vamos a especificarle a través de Dense con activación ReLu.
  - Capa de Salida: Tiene 10 neuronas para MNIST, correspondientes a las 10 clases de dígitos (0-9). Usa la función de activación softmax para la clasificación multiclase.

Funciones de Activación:

ReLU: Utilizada en las capas ocultas para introducir no linealidad.

Softmax: En la capa de salida para obtener probabilidades de las clases.

Compilación del Modelo:

Optimizador: Adam, conocido por su eficiencia y ajuste automático del learning rate.

Función de Pérdida(loss): sparse_categorical_crossentropy, adecuada para clasificación multiclase.

Métricas: Precisión (accuracy), para evaluar el rendimiento del modelo.

Entrenamiento del Modelo:

Epochs: Número de veces que el modelo verá todo el conjunto de datos de entrenamiento.

Batch Size: Número de muestras procesadas antes de actualizar el modelo.

Validation Split: Fracción de los datos para validar el modelo durante el entrenamiento.

Ejecución

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:11:00.978064: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized t
o use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlo
w with the appropriate compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2726 - accuracy: 0.9211 - val_loss: 0.1585 - val_ac
curacy: 0.9490
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1158 - accuracy: 0.9663 - val_loss: 0.1165 - val_ac
curacy: 0.9642
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0751 - accuracy: 0.9777 - val_loss: 0.1069 - val_ac
curacy: 0.9690
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0541 - accuracy: 0.9835 - val_loss: 0.0903 - val_ac
curacy: 0.9747
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0377 - accuracy: 0.9894 - val_loss: 0.0932 - val_ac
curacy: 0.9718
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0296 - accuracy: 0.9911 - val_loss: 0.0881 - val_ac
curacy: 0.9733
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0209 - accuracy: 0.9940 - val_loss: 0.0919 - val_ac
curacy: 0.9736
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0181 - accuracy: 0.9943 - val_loss: 0.0840 - val_ac
curacy: 0.9759
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0157 - accuracy: 0.9952 - val_loss: 0.0958 - val_accuracy: 0.9742
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0117 - accuracy: 0.9964 - val_loss: 0.0903 - val_accuracy: 0.9771
313/313 [==============================] - 1s 2ms/step - loss: 0.0842 - accuracy: 0.9792
Test Accuracy: 0.979200005531311
Test Loss: 0.0841858759522438
Tiempo: 39.306 s
```

Experimentación y Ajuste de Parámetros

Número de Neuronas y Capas: Ajustar estos parámetros puede impactar en la capacidad del modelo para aprender patrones complejos.

Learning Rate del Optimizador: Ajustar el learning rate puede influir en la convergencia y el rendimiento del modelo.

Para mejorar la eficacia de un Perceptrón Multicapa (MLP) en una tarea de clasificación de dígitos utilizando el conjunto de datos MNIST, se peude experimentar con varios parámetros y técnicas.

- Ajustar el Número de Capas y Neuronas:

Añadir más capas ocultas o cambiar el número de neuronas en cada capa. Esto puede ayudar al modelo a aprender representaciones más complejas, pero también puede aumentar el riesgo de sobreajuste. En este caso vamos a dejarlo con solo dos Dense, ya que la ejecución tarda mucho y da una precisión bastante alta.

- Modificar el Dropout:

Ajustar la tasa de dropout puede ayudar a regularizar el modelo. Un dropout demasiado alto puede impedir el aprendizaje efectivo, mientras que uno demasiado bajo puede llevar a sobreajuste. He adaptado un Dropout del 50% para evitar el sobreajuste.

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:16:17.491819: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.3799 - accuracy: 0.8888 - val_loss: 0.1799 - val_accuracy: 0.9474
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1988 - accuracy: 0.9414 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1537 - accuracy: 0.9535 - val_loss: 0.1082 - val_accuracy: 0.9674
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1321 - accuracy: 0.9596 - val_loss: 0.0956 - val_accuracy: 0.9716
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1157 - accuracy: 0.9647 - val_loss: 0.0889 - val_accuracy: 0.9722
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1016 - accuracy: 0.9678 - val_loss: 0.0882 - val_accuracy: 0.9719
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0948 - accuracy: 0.9708 - val_loss: 0.0838 - val_accuracy: 0.9752
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0880 - accuracy: 0.9716 - val_loss: 0.0804 - val_accuracy: 0.9749
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0823 - accuracy: 0.9749 - val_loss: 0.0803 - val_accuracy: 0.9739
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0750 - accuracy: 0.9760 - val_loss: 0.0775 - val_accuracy: 0.9761
313/313 [==============================] - 1s 2ms/step - loss: 0.0774 - accuracy: 0.9784
Test Accuracy: 0.9783999919891357
Test Loss: 0.07742641866207123
Tiempo: 39.542 s
```

Dando una precisión por muy muy poco más alta en u poquito más de tiempo, lo cual se puede considerar o no prescindible.

Si ponemos DropOut = 20%

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:19:13.811900: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.3006 - accuracy: 0.9131 - val_loss: 0.1555 - val_accuracy: 0.9532
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1387 - accuracy: 0.9588 - val_loss: 0.1177 - val_accuracy: 0.9663
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0966 - accuracy: 0.9706 - val_loss: 0.0967 - val_accuracy: 0.9704
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0753 - accuracy: 0.9772 - val_loss: 0.0855 - val_accuracy: 0.9729
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0614 - accuracy: 0.9807 - val_loss: 0.0911 - val_accuracy: 0.9702
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0509 - accuracy: 0.9834 - val_loss: 0.0819 - val_accuracy: 0.9742
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0415 - accuracy: 0.9869 - val_loss: 0.0798 - val_accuracy: 0.9753
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0339 - accuracy: 0.9893 - val_loss: 0.0781 - val_accuracy: 0.9757
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0312 - accuracy: 0.9896 - val_loss: 0.0856 - val_accuracy: 0.9746
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.0278 - accuracy: 0.9904 - val_loss: 0.0846 - val_accuracy: 0.9754
313/313 [==============================] - 1s 2ms/step - loss: 0.0785 - accuracy: 0.9783
Test Accuracy: 0.9782999753952026
Test Loss: 0.07854561507701874
Tiempo: 39.060 s
```

El cambio vuelve a ser mínimo.

Dropout = 70%

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:20:50.445035: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.4915 - accuracy: 0.8502 - val_loss: 0.2078 - val_accuracy: 0.9408
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2880 - accuracy: 0.9136 - val_loss: 0.1595 - val_accuracy: 0.9531
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2413 - accuracy: 0.9249 - val_loss: 0.1395 - val_accuracy: 0.9581
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2162 - accuracy: 0.9355 - val_loss: 0.1229 - val_accuracy: 0.9628
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1946 - accuracy: 0.9415 - val_loss: 0.1193 - val_accuracy: 0.9644
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1861 - accuracy: 0.9448 - val_loss: 0.1079 - val_accuracy: 0.9682
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1786 - accuracy: 0.9455 - val_loss: 0.1045 - val_accuracy: 0.9698
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1648 - accuracy: 0.9500 - val_loss: 0.1041 - val_accuracy: 0.9699
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1615 - accuracy: 0.9504 - val_loss: 0.1019 - val_accuracy: 0.9697
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1524 - accuracy: 0.9533 - val_loss: 0.0986 - val_accuracy: 0.9716
313/313 [==============================] - 1s 2ms/step - loss: 0.0894 - accuracy: 0.9749
Test Accuracy: 0.9749000072479248
Test Loss: 0.08941996842622757
Tiempo: 38.593 s
```

La precisión baja un poco, pero ni se aprecia.

En mi conclusión por evitar el posible sobreajuste, voy a introducirlo al 50%.

- Cambiar la Función de Activación:

Aunque 'relu' es una buena elección general, experimentar con otras funciones de activación (como 'tanh' o 'elu') puede ofrecer resultados interesantes.

Si probamos con ‘tanh’

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:23:03.576919: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.4064 - accuracy: 0.8780 - val_loss: 0.2652 - val_accuracy: 0.9220
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2786 - accuracy: 0.9167 - val_loss: 0.2183 - val_accuracy: 0.9359
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2326 - accuracy: 0.9307 - val_loss: 0.1779 - val_accuracy: 0.9474
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2010 - accuracy: 0.9394 - val_loss: 0.1586 - val_accuracy: 0.9521
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1818 - accuracy: 0.9453 - val_loss: 0.1442 - val_accuracy: 0.9573
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1655 - accuracy: 0.9496 - val_loss: 0.1385 - val_accuracy: 0.9576
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1532 - accuracy: 0.9532 - val_loss: 0.1269 - val_accuracy: 0.9616
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1431 - accuracy: 0.9558 - val_loss: 0.1271 - val_accuracy: 0.9632
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1361 - accuracy: 0.9589 - val_loss: 0.1166 - val_accuracy: 0.9653
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1251 - accuracy: 0.9606 - val_loss: 0.1127 - val_accuracy: 0.9664
313/313 [==============================] - 1s 2ms/step - loss: 0.1078 - accuracy: 0.9674
Test Accuracy: 0.9674000144004822
Test Loss: 0.1077730730175972
Tiempo: 39.361 s
```

Vemos que la precisión baja un poco y el tiempo es más o menos el mismo.

Y ahora vamos a probarlo con ‘elu’

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:25:34.805249: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.4010 - accuracy: 0.8790 - val_loss: 0.2458 - val_accuracy: 0.9299
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2669 - accuracy: 0.9219 - val_loss: 0.1907 - val_accuracy: 0.9455
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2178 - accuracy: 0.9361 - val_loss: 0.1671 - val_accuracy: 0.9514
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1843 - accuracy: 0.9442 - val_loss: 0.1454 - val_accuracy: 0.9569
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1650 - accuracy: 0.9501 - val_loss: 0.1317 - val_accuracy: 0.9611
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1528 - accuracy: 0.9535 - val_loss: 0.1188 - val_accuracy: 0.9647
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1367 - accuracy: 0.9587 - val_loss: 0.1139 - val_accuracy: 0.9674
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1272 - accuracy: 0.9606 - val_loss: 0.1089 - val_accuracy: 0.9685
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1184 - accuracy: 0.9632 - val_loss: 0.1094 - val_accuracy: 0.9678
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1127 - accuracy: 0.9644 - val_loss: 0.1087 - val_accuracy: 0.9685
313/313 [==============================] - 1s 2ms/step - loss: 0.0972 - accuracy: 0.9719
Test Accuracy: 0.9718999862670898
Test Loss: 0.09724437445402145
Tiempo: 39.710 s
```

Vemos que la precisión sube un poco en comparación con tanh, pero es más baja que con relu, por lo que de los tres experimentos nos quedamos con el que teníamos, con ‘relu.’

- Optimizador y Tasa de Aprendizaje:

Puedes probar diferentes optimizadores como SGD, RMSprop, o Adam con distintas tasas de aprendizaje. Una tasa de aprendizaje adaptativa (como la que ofrece Adam) suele funcionar bien, pero en algunos casos, una tasa de aprendizaje fija o con decaimiento puede ser más efectiva.

El caso inicial era con el optimizador Adam()

Vamos a probar con SGD()

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:29:18.188894: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.9009 - accuracy: 0.7400 - val_loss: 0.4492 - val_accuracy: 0.8831
Epoch 2/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.4872 - accuracy: 0.8595 - val_loss: 0.3552 - val_accuracy: 0.8991
Epoch 3/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.4087 - accuracy: 0.8819 - val_loss: 0.3132 - val_accuracy: 0.9109
Epoch 4/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.3628 - accuracy: 0.8963 - val_loss: 0.2863 - val_accuracy: 0.9166
Epoch 5/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.3333 - accuracy: 0.9042 - val_loss: 0.2629 - val_accuracy: 0.9258
Epoch 6/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.3081 - accuracy: 0.9110 - val_loss: 0.2460 - val_accuracy: 0.9324
Epoch 7/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.2912 - accuracy: 0.9173 - val_loss: 0.2326 - val_accuracy: 0.9348
Epoch 8/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.2750 - accuracy: 0.9218 - val_loss: 0.2199 - val_accuracy: 0.9392
Epoch 9/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.2623 - accuracy: 0.9250 - val_loss: 0.2091 - val_accuracy: 0.9419
Epoch 10/10
1200/1200 [==============================] - 3s 3ms/step - loss: 0.2474 - accuracy: 0.9296 - val_loss: 0.1995 - val_accuracy: 0.9445
313/313 [==============================] - 1s 2ms/step - loss: 0.1920 - accuracy: 0.9453
Test Accuracy: 0.9452999830245972
Test Loss: 0.19197769463062286
Tiempo: 31.831 s
```

Tarda menos pero da un porcentaje de precisión menor.

Y si probamos con RMSprop()

```
Tarea 2D: Perceptron multicapa (MLP)
2023-12-24 16:30:29.766278: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU i
nstructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
Epoch 1/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.3750 - accuracy: 0.8886 - val_loss: 0.1837 - val_accuracy: 0.9465
Epoch 2/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.2117 - accuracy: 0.9389 - val_loss: 0.1439 - val_accuracy: 0.9572
Epoch 3/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1738 - accuracy: 0.9497 - val_loss: 0.1275 - val_accuracy: 0.9632
Epoch 4/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1505 - accuracy: 0.9572 - val_loss: 0.1186 - val_accuracy: 0.9673
Epoch 5/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1394 - accuracy: 0.9616 - val_loss: 0.1116 - val_accuracy: 0.9695
Epoch 6/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1280 - accuracy: 0.9650 - val_loss: 0.1114 - val_accuracy: 0.9704
Epoch 7/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1221 - accuracy: 0.9668 - val_loss: 0.1085 - val_accuracy: 0.9716
Epoch 8/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1209 - accuracy: 0.9678 - val_loss: 0.1061 - val_accuracy: 0.9716
Epoch 9/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1143 - accuracy: 0.9702 - val_loss: 0.1057 - val_accuracy: 0.9721
Epoch 10/10
1200/1200 [==============================] - 4s 3ms/step - loss: 0.1060 - accuracy: 0.9723 - val_loss: 0.1175 - val_accuracy: 0.9723
313/313 [==============================] - 1s 2ms/step - loss: 0.1070 - accuracy: 0.9749
Test Accuracy: 0.9749000072479248
Test Loss: 0.10704189538955688
Tiempo: 38.526 s
```

Tiene 1% de precisión menor, y tarda 1 segundo menos, por lo que se podría considerar el uso de RMSprop() en lugar de Adam() como optimizador.

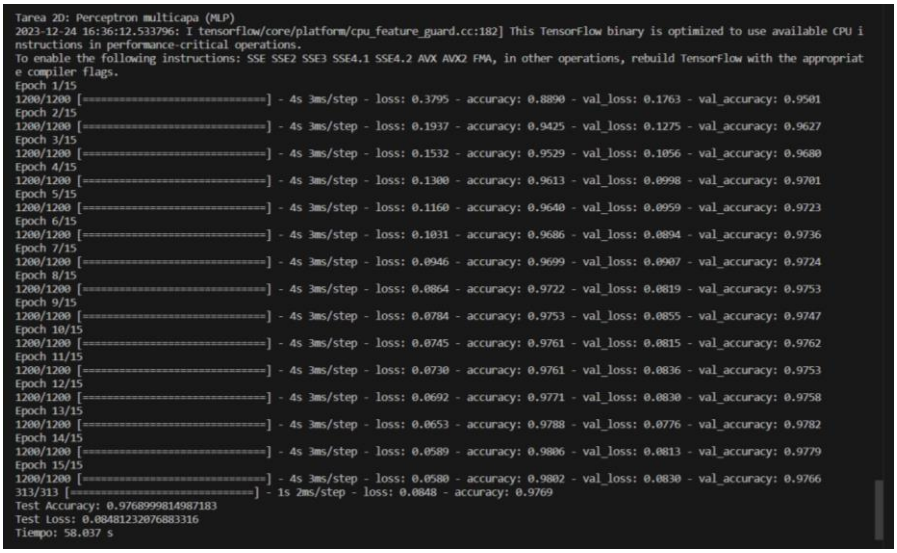
- Tamaño del Batch y Número de Epochs:

Experimentar con diferentes tamaños de batch puede afectar la velocidad y la calidad del entrenamiento. Del mismo modo, aumentar el número de epochs puede mejorar el rendimiento hasta cierto punto, pero también puede conducir a sobreajuste.

Bach_size inicialmente está en 32 y epochs=10

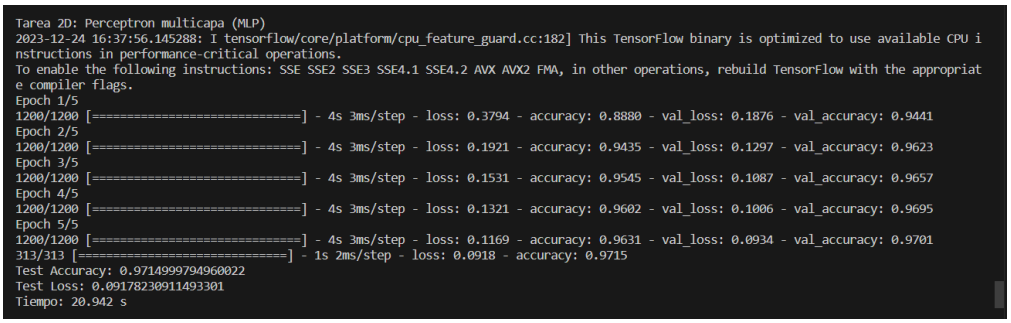
Vamos a experimentar.

Experimento 1 epochs=15



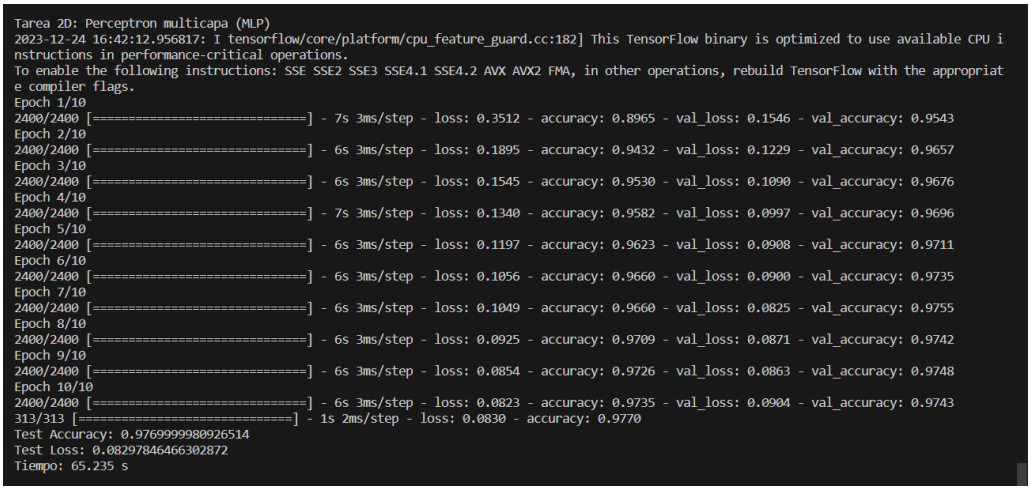
No merece la pena, poque ha bajado la precisión y ha tardado casi 1 minuto.

Experimento 2 epochs = 5



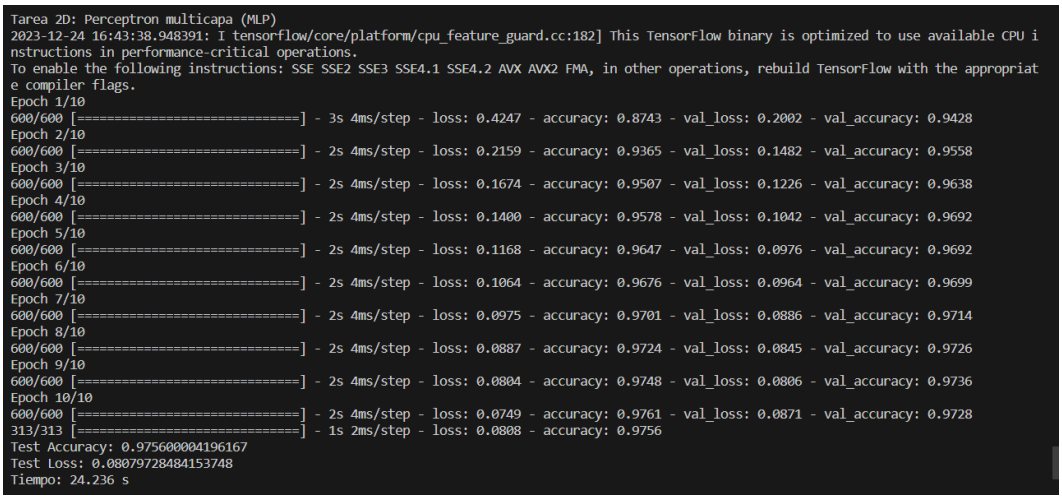
Tarda menos, pero la precisión ha bajado un poco, por lo que vamos a dejarlo en 10.

Experimento 3 batch_size = 16



La precisión es casi la misma, un poco menor, pero tarda poco más de un minuto.

Experimento 4 batch_size=64



El tiempo es menor que cuando batch_size=32, y la precisión es casi la misma por lo que se podría considerar utilizar 64 en vez de 32.

- Función de Pérdida:

Para la clasificación multiclase, 'sparse_categorical_crossentropy' es una buena elección.

- Normalización de Batch y Capas de Regularización:

La adición de capas de normalización de batch puede mejorar la estabilidad y el rendimiento del entrenamiento. También se puede experimentar con regularización L1 o L2 en las capas densas.

- Ajustar la Validación Split:

Variar la proporción de división de validación puede proporcionar diferentes vistas sobre cómo el modelo generaliza en datos no vistos.

- Experimentar con Técnicas de Aumento de Datos:

Aunque menos común en MLPs que en redes convolucionales, experimentar con técnicas de aumento de datos, como el cambio de escala o la rotación de imágenes, puede mejorar la robustez del modelo.

En conclusión, en la mayoría de los experimentos se encuentran resultados relativamente parecidos a los obtenidos con la configuración inicial, por lo que se podrían cambiar los valores de los parámetros, según el contexto que se requiera. Yo lo voy a dejar con la configuración inicial ya que ya de por sí da muy buenos resultados, y quiero evitar el sobreajuste o las equivocaciones.

Aclaración: Hay parámetros que he considerado pero he preferido experimentar porque me suenan de mencionarlos en clase de teoría sobre todo, y son los valores que se utilizan más comunes.

Evaluación del Modelo

Precisión y Pérdida: Medir la precisión y la pérdida en el conjunto de prueba para evaluar el rendimiento general del modelo.

Análisis del Historial de Entrenamiento: Observar las métricas de entrenamiento y validación a lo largo de las epochs para identificar signos de sobreajuste o convergencia. En esta práctica no lo he utilizado.

Conclusión

En la Tarea 2D, se utiliza Keras para implementar un MLP para clasificar dígitos manuscritos del conjunto de datos MNIST. La tarea implica una comprensión profunda de la arquitectura de las redes neuronales, la selección de parámetros adecuados y la evaluación del modelo. La flexibilidad de Keras permite experimentar con diferentes configuraciones de la red, lo cual es esencial para encontrar la mejor configuración en términos de precisión y eficiencia computacional.

Tarea 2E: Modela un clasificador mediante CNN para MNIST con Keras

Las CNN son un tipo de red neuronal artificial especializada en procesar datos con una topología en forma de cuadrícula, como imágenes, es decir, están diseñadas específicamente para procesar datos en forma de múltiples matrices (como imágenes), lo que las hace más eficientes para tareas de visión artificial. Por eso, a diferencia de los MLP que tratan los datos de entrada como vectores unidimensionales, las CNN conservan la estructura espacial de las imágenes, permitiendo que el modelo aprenda patrones como bordes, texturas y formas.

Las CNN son capaces de aprender jerarquías de patrones, desde características simples en las primeras capas hasta patrones más complejos en capas más profundas y son relativamente invariantes a la posición y orientación de los objetos en la imagen, lo que las hace robustas a variaciones en la ubicación de los objetos de interés.

Se caracterizan por su uso de capas convolucionales que aplican filtros o núcleos a los datos de entrada para extraer características espaciales y temporales. Esto se realiza mediante una operación matemática llamada convolución, dada en clase de teoría.

Además de las capas convolucionales, las CNN típicamente incluyen capas de pooling (o agrupamiento), que reducen las dimensiones espaciales (anchura y altura) de los datos de entrada para disminuir la cantidad de parámetros y el cómputo en la red.

Gracias a las operaciones de convolución y pooling, las CNN reducen el número de parámetros en comparación con los MLP, lo que disminuye el riesgo de sobreajuste y mejora la eficiencia computacional.

Implementación

Cargar el Conjunto de Datos MNIST:

(X, y), (X_test, y_test) = mnist.load_data(): Esta línea carga el conjunto de datos MNIST, que contiene imágenes de dígitos escritos a mano. X y y son los datos y etiquetas de entrenamiento, mientras que X_test y y_test son los datos y etiquetas de prueba.

Normalizar y Redimensionar los Datos:

X.reshape((-1, 28, 28, 1)).astype('float32') / 255: Las imágenes se redimensionan para tener una forma adecuada para una CNN (con un canal de color, en este caso, escala de grises) y se normalizan dividiéndolas por 255, transformándolas a un rango de 0 a 1.

Dividir en Conjuntos de Entrenamiento y Validación:

train_test_split(X, y, test_size=0.2, random_state=42): Divide los datos en conjuntos de entrenamiento y validación, con un 20% de los datos reservados para validación.

Construcción del Modelo CNN:

Sequential(): Define un modelo secuencial que es una forma eficiente y sencilla de construir redes neuronales para una amplia variedad de tareas de aprendizaje automático, siempre que la arquitectura de la red sea de naturaleza lineal.

Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)): Capa convolucional con 32 filtros y un tamaño de kernel de 3x3. Usa ReLU como función de activación.

MaxPooling2D(pool_size=(2, 2)): Capa de pooling para reducir la dimensionalidad espacial.

Flatten(): Aplana los mapas de características para convertirlos en un vector.

Dense(128, activation='relu'): Capa densa con 128 neuronas.

Dense(10, activation='softmax'): Capa de salida con 10 neuronas, una por cada clase de dígito, utilizando softmax para la clasificación multiclase.

Compilación del Modelo:

model.compile(optimizer=Adam(), loss='sparse_categorical_crossentropy', metrics=['accuracy']): Compila el modelo con el optimizador Adam, la función de pérdida sparse_categorical_crossentropy para la clasificación multiclase y monitorea la precisión.

Entrenamiento del Modelo:

model.fit(): Entrena el modelo con los datos de entrenamiento, usando una validación basada en el conjunto de validación.

Evaluación del Modelo:

model.evaluate(X_test, y_test): Evalúa el rendimiento del modelo en el conjunto de prueba.

Impresión de Resultados:

Imprime la precisión y la pérdida en el conjunto de prueba.

Modelo e Historial:

Para análisis posterior, se puede devolver el modelo y su historial de entrenamiento, pero como he explicado en la tarea anterior, es algo de lo que yo no voy a hacer uso.

Experimentación

No he experimentado mucho en esta tarea, debido a que es muy parecida a la anterior y los cambios eran más o menos los mismos.

Tarea 2F: Realiza una comparativa de los modelos implementados

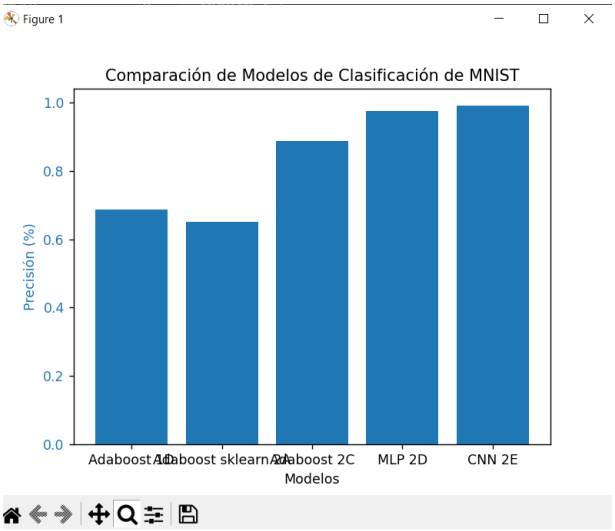
Adaboost Multiclase 1D

AdaboostClassifier sencillo de sklearn 2A

AdaboostClassifier incorporando DecisionTreeClassifier parametrizado 2C

MLP 2D

CNN 2F



Para esta práctica comparando las precisiones aproximadas obtenidas podemos deducir que la red neuronal convencional es la que mejor precisión obtiene, seguido por poco del Perceptrón Multicapa, pero me gustaría destacar que viendo las diferencias en los tiempos de ejecuciones, sería debatible mi uso de ellas, ya que he podido observar que ambas tareas no tardan lo mismo si las ejecuto solo 1 por 1 a si ejecuto el main entero, ya que si ejecuto el main entero estas en concreto se alargan más de lo que deben. Lo que me hace pensar que según que contexto, que uso, que costes computacionales hay de antes, igual a pesar de su precisión conviene utilizar otra técnica que no tarde tanto, ya que para mi si me dan a elegir entre varias técnicas y tengo otras con las que también obtengo mejores resultados, me puede convenir utilizar una que el equilibrio entre la precisión y el tiempo se consideren más eficaces que solamente mirar por *'Qué técnica me da mejores resultados'*.

Siguiendo, analizando la comparación vemos algo que ya comparamos en la tarea 2B:

Mi Adaboost Multiclase da mejores resultados que el AdaboostClassifier BASE de sklearn, pero el AdaboostClassifier de la tarea 2C habiéndolo implementado de manera más eficaz, da muchos mejores resultados que mi Adaboost Multiclase como ya se analizo en la tarea 2B.

Para la resolución del objetivo de está práctica de explorar diferentes técnicas para el reconocimiento de dígitos a través de imágenes por mnist el ranking literal según la precisión quedaría así:

1. Red Neuronal Convencional (CNN)
2. Perceptrón Multicapa (MLP)
3. AdaboostClassifier de sklearn 2C
4. Adaboost Multiclase 1D
5. AdaboostClassifier sencillo 2A

Sin embargo, si yo tuviera que utilizarlo para el día del mañana para un problema no muy complejo me guiaría por el AdaboostClassifier de sklearn 2C, ya que el equilibrio entre la precisión y el tiempo para esta práctica es el que más me ha convencido. Por otro lado, no descarto que la CNN o el MLP sean mucho más eficaces para otro tipo de problemas igual o más complejos, ya que me han parecido técnicas muy interesantes con las que tratar.

Bibliografía

Para esta práctica se ha hecho uso de los apuntes de la asignatura, a parte del enunciado entre otros el Tema 6 de teoría en el cual se explica el algoritmo de Adaboost y más conceptos y del Tema 7.

También se ha utilizado el Chat GPT 3.5 para la resolución de dudas sobre el funcionamiento o detección de errores, aunque no haya sido del todo eficaz.

Adicionalmente se han buscado código ejemplo en repositorios de Github o en otras páginas web pero las páginas que más me han servido han sido las siguientes:

https://keras.io/examples/vision/mnist_convnet/

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

https://www.tensorflow.org/guide/core/mlp_core