Práctica 2: Visión artificial y aprendizaje

INTRODUCCIÓN

En esta práctica hemos llevado a cabo el desarrollo y la evaluación de clasificadores tanto binarios como multiclase para identificar el número representado en una imagen. Específicamente:

He desarrollado todo lo siguiente **from scratch:**

- He creado nuestro propio clasificador usando el algoritmo AdaBoost y hemos entrenado este modelo utilizando el conjunto de datos de imágenes MNIST.
- He desarrollado un clasificador ADABoost Multiclase para entrenar un clasificador ADABoost binario para cada una de las clases del 0 al 9 de MNIST
- He hecho una implementación from scratch del algoritmo PCA (Principal Component Analysis) para reducir el número de características a probar dentro del entrenamiento de nuestro ADABoost
- He implementado una versión de ADABoost binario apaz de detectar sobreentrenamiento.

He desarrollado todo lo siguiente haciendo uso de la librería **sklearn** y **keras:**

- He empleado la clase "AdaBoostClassifier" y “DecissionTreeClassifier” de la biblioteca scikit-learn para resolver nuestra tarea, comparándola con el clasificador que desarrollamos from scratch, probando con con árboles de decisión de profundidad = 1 y de profundidad > 1.
- He utilizado la biblioteca keras para realizar predicciones utilizando un perceptrón multicapa (MLP) y una red convolucional (CNN).

Junto a todo esto, para cada tarea se ha hecho un estudio de la precisión y del tiempo de ejecución obtenidos en base a diferentes combinaciones de parámetros. Una vez realizado todo esto para cada una de las tareas, se ha concluido la práctica con una evaluación comparativa del rendimiento tanto a nivel de tiempo como de precisión de todos los métodos empleados.

NOTA: Se recomienda tener abierto el archivo .ipynb al mismo tiempo que se lee esta memoria. Mi idea era el no llenar toda la memoria de imágenes innecesarias que ya se encuentran disponibles desde otro archivo.

Repositorio de Github de la práctica (acceder desde la rama "cris"):
https://github.com/cacs2-ua/SI-Recu-Prac2-cacs2.git

Tarea 1 A

En primer lugar, he implementado la función “balance_training_dataset” la cual se encarga de hacer que el conjunto de entrenamiento sea balanceado y uniforme. Esto quiere decir lo siguiente:

Consideremos que estamos entrenando un ADABoost Binario para el dígito 7, y que MNIST devuelve un conjunto de entrenamiento inicial con una cantidad de 7’s = 9000. Entonces, Hacer que el conjunto de entrenamiento esté balanceado implica hacer que el número de clases positivas (número de 7’s) sea igual que el número de clases negativas (cantidad de resto de números). De esta forma, hay que hacer que haya un total de 9000 instancias en total para el resto de dígitos. Finalmente, hacer que sea uniforme implica que, dentro de las 9000 instancias pertenecientes al resto de clases, para cada clase, debe de haber el mismo número de instancias (en este caso, debe de haber 900 ceros, 900 unos, 900 doses…)

A continuación, he pasado a definir la clase ADABoost propiamente dicha:

- En primer lugar, la clase DecisionStump define nuestros clasificadores débiles. Cada uno de estos clasificadores almacena un índice para las imágenes, un umbral y una polaridad, todos seleccionados de manera aleatoria. Durante la predicción, el resultado depende de la polaridad y de si el valor del índice supera el umbral establecido.

- Un aspecto importante a considerar se trata de que he elegido un índice de característica aleatorio con dos coordenadas, puesto que mi idea inicial era trabajar con las imágenes 2D de MNIS de forma directa. Más adelante, en la

tarea 1C me di cuenta de que esto no era lo óptimo y ya ahí utilicé el método reshape para aplanaar las imágenes de MNIST a imágenes 1D.

- A continuación, viene la declaración de la clase Adaboost, que representa nuestro clasificador binario principal. El proceso de entrenamiento implica almacenar en una lista "T" clasificadores débiles que se entrenan utilizando el conjunto de datos "X_train" e "Y_train".

- Se repite este proceso "A" veces, generando un nuevo clasificador débil en cada iteración y evaluando su rendimiento en el conjunto de entrenamiento.

- El clasificador que presenta el menor error entre los "A" clasificadores se guarda en nuestra lista de "T" clasificadores débiles, junto con su coeficiente "alfa", que refleja la importancia del clasificador según su tasa de error.

- Con esto completamos la preparación para realizar predicciones sobre el conjunto de imágenes "test". Este proceso implica calcular una lista que combina el peso de cada clasificador con el valor de su predicción para cada imagen.

- Finalmente, sumamos los pesos de cada clasificador para cada imagen y determinamos el signo del resultado para decidir si la imagen corresponde al número para el cual se entrenó el modelo.

- Resulta especialmente importante entender bien el concepto de polaridad para entender como funciona realmente las predicciones. De esta forma:

- Si la polaridad es 1, entonces aquellas características menores que el umbral serán clasificadas como incorrectas y las que estén por encima del umbral serán clasificadas como correctas

```
if self.polarity == 1: # Si la polaridad es 1
    predictions[X_column < self.threshold] = -1 # S
```

- Por otro lado, si la polaridad es -1, AdaBoost invierte las predicciones para que el error siga siendo menor que 0.5 (para que siempre sea mejor que elegir al azar). De esta forma, ahora las muestras con valor de columna menores que el umbral serán clasificadas como correctas y las que estén por encima, como incorrectas.

Una vez implementadas las clases DecissionStump y ADABoost, me dispuse a crear las funciones run_adaboost_on_mnist para entrenar un dígito por separado y run_adaboost_for_all_digits para mostrar los resultados de entrenar dígito por dígito. Dentro de esta última función incluí que se mostrase por pantalla la matriz de confusión una vez terminase el entrenamiento del dígito correspondiente.

Así, por ejemplo para el dígito 6:

```
Accuracy for digit 6: 0.9512
Confusion Matrix for digit 6:
[[8606  436]
 [  52  906]]
```

La matriz de confusión de la imagen significa lo siguiente:

- 8606 instancias fueron correctamente predichas como diferentes de 6 (Verdaderos Negativos)
- 436 instancias fueron incorrectamente predichas como iguales a 6 (Falsos Positivos)

- 52 instancias fueron incorrectamente predichas como diferentes de 6 (Falsos Negativos)
- 906 fueron correctamente predichas como iguales a 6 (Verdaderos positivos)

Con esta versión de ADABoost, se han obtenido las siguientes precisiones con T=50 y A=20:

Accuracies for all digits: {0: 0.9565, 1: 0.9624, 2: 0.8992, 3: 0.8887, 4: 0.9116, 5: 0.8734, 6: 0.9458, 7: 0.9464, 8: 0.8717, 9: 0.8623}

Tarea 1B

Observando las gráficas correspondientes del archivo .ipynb, observamos que, tanto aumentar como aumentar T llevan a obtener precisiones mayores. Esto nos indica que no existe sobreentrenamiento para valores de A y T menores de 150 (cuando uno de los dos se mantiene fijado). Como veremos más adelante, para un determinado A fijado menor que 100, el sobreentrenamiento comienza a manifestarse a partir de los 200 clasificadores débiles.

Una observación relevante es que la variable T parece tener el mayor impacto en los resultados. Se observan casos donde una prueba con una T alta y una A baja ha superado en rendimiento a otra prueba con una T menor pero una A considerablemente mayor. En conclusión la ganancia obtenida al utilizar una combinación de T y A muy alta en comparación con una A más pequeña no justifica el aumento en el tiempo de entrenamiento. En los valores más altos, encontramos diferencias mínimas en precisión pero con costos temporales significativamente diferentes.

Asimismo, es **importante** notar que, para cada resultado obtenido en la experimentación, la ejecución asociada se repite un número parametrizado de veces y se promedia a través del parámetro “repetitions”. Este hecho es

imprescindible debido a que si no se promediase en lo absoluto no se estaría obteniendo una verdadera relevancia estadística.

Volviendo con T y A, si no queremos sobrepasar la condición de que T*A < 3600, la combinación ganadora en relación precisión tiempo seria T=70 y A = 50, puesto que con un tiempo de ejecución de menos de 20 segundos ya se obtiene una precisión de 0.96.

Ahora bien, si queremos afinar aún más en la precisión para ganar algunas décimas y sobrepasando T*A < 3600, una combinación más curtida sería T = 140 y A = 100, con la cual obtenemos una precisión alrededor de 0.965.

Tarea 1C

La implementación del clasificador multiclase me resultó directa, aprovechando la experiencia previa de haber programado desde cero el ADABoost Binario anteriormente mencionado.

A nivel de implementación, creamos la clase AdaboostMulticlase que contiene las variables T y A, y ahora se añade una lista para almacenar cada uno de los 10 clasificadores binarios correspondientes a cada uno de los diez dígitos de MNIST.

Implementar el proceso de entrenamiento también fue fácil: iteramos 10 veces, una por cada dígito,para crear y entrenar un clasificador binario que se guarda en

la lista correspondiente. Al final, tenemos nuestra lista completa de clasificadores para los 10 dígitos.

Para las predicciones multiclase, cada imagen se evalúa con todos los clasificadores binarios disponibles.

Para que el clasificador multiclase funcionase de forma apropiada, tuve que cambiar un pequeño detalle (pero fundamental) de la implementación implementación del ADABoost binario, de forma que ahora este devuelve las predicciones **tal cual** y no solo el signo (ahora una clasificación por ejemplo de 0.21 se devuelve tal cual como 0.21 y no como +1). Esto permite al ADABoost multiclase determinar la clasificación más acertada para tomar así la predicción con mayor precisión según el dígito correspondiente.

Para T=50 y A=50, se obtiene una precisión de 0.86 en un tiempo de ejecución de alrededor de 20 segundos.

Una vez se termina de ejecutar el ADABoost Multiclase, se muestra por pantalla la matriz de confusión correspondiente, por ejemplo:

```
Multiclass Accuracy: 0.8601
Overall Multiclass Confusion Matrix:
[[ 9359     0    36    36     9   135   126    18    45    36]
 [    0 10873   108    36     0    45    36     0   252     0]
 [  144   297  8124   324   279    72   243   216   585    36]
 [   99    90   243  8408    45   567    45   162   279   162]
 [   45    45    99    27  8353   126   117    63   207   738]
 [  234   144   108   711   261  6544   207   189   270   252]
 [  135    45   162    27   189   189  8689    18   117     9]
 [   36   225   198    36   207    63     9  8921    18   567]
 [  126   153   108   450    99   261   108   126  8048   261]
 [   99    72    54   117   702   171     9   441   225  8200]]
```

Cada elemento (i,j) de la matriz de confusión indica cuantas veces un dígito i (verdadero) fue clasificado como un dígito j (predicho) por el modelo

Así, la tercera fila proporciona la siguiente información:

-. El dígito 2 fue predicho 144 veces como dígito 0

-. El dígito 2 fue predicho 297 veces como dígito 1

-. El dígito 2 fue predicho 8124 veces como dígito 2

-. El dígito 2 fue predicho 324 veces como dígito 3

Y así con el resto.

En cuanto a la experimentación, observando las gráficas correspondientes, vemos como nuevamente se obtienen mayores valores de precisión tanto cuando se aumenta A como cuando se aumenta T.

Un aspecto clave a destacar es que, dado que el ADABoost multiclase entrena 10 clasificadores binarios, ahora el tiempo de ejecución es considerablemente más elevado, aunque sigue siendo muy asumible (el tiempo más grande de ejecución que aparece en la gráfica es de alrededor de 38 segundos).

También se observa nuevamente que T es la variable que mayor impacto genera en cuanto a la obtención de la precisión (lo cual es obvio, ya que el ADABoost multiclase hace uso directo de la implementación de nuestro ADABoost binario)

Asimismo, recordemos que la lista de precisiones para nuestro ADABoost binario con A=50 y T=50 fue de:

Accuracies for all digits: {0: 0.9565, 1: 0.9624, 2: 0.8992, 3: 0.8887, 4: 0.9116, 5: 0.8734, 6: 0.9458, 7: 0.9464, 8: 0.8717, 9: 0.8623}

El promedio de estas 10 precisiones es de 0.9100.

Como se observa en las gráficas, el valor de la precisión alcanza 0.90 a partir de 145 clasifficadores aproximadamente (de hecho, con T = 180 y A = 50 se alcanza una precisión de 0.9160). Esto nos indica que el ADABoost multiclase tiende a predecir de forma más precisa las clases predichas con menor precisión por un ADABoost binario individual (como el 5, el 9, el 8…)

Finalmente, como recomendación óptima, elegiría precisamente T = 180 y A =50, ya que consigue una precisión alta dentro de un tiempo de ejecución corto. Aumentar en mayor medida T, y sobre todo A, llevaría a aumentar cada vez más el tiempo de ejecución a cambio de solo unas pocas centésimas extras de precisión, como se ve en la gráfica la cual empieza a allanarse en torno a 0.90 (y si se siguiese aumentando aún más, se incurriría en un sobreentrenamiento).

Tarea 1D

Para esta tarea, he realizado dos cambios fundamentales a la hora de la eficacia y precisión de los experimentos realizados:

-. En primer lugar, he cambiado la clase DecissionStump para que ahora tenga un feature_index de una sola coordenada. Con este cambio, he conseguido aplanar las imágenes de entrada de MNIST, reduciendo así la dimensión de las imágenes de 2D a 1D gracias al método reshape. Aún así, este cambio solo ha dado algunas centésimas extras de precisión y ha agilizado un poco el tiempo de ejecución.

-. El cambio más significativo ha sido la implementación del PCA (Principal Component Analyisis).

Definición

La técnica de PCA (Análisis de Componentes Principales) es un método de reducción de dimensionalidad que permite transformar un conjunto de datos de alta dimensión en uno de menor dimensión mientras conserva la mayor cantidad posible de la varianza (el error ponderado) original. PCA logra esto mediante la identificación y selección de los vectores principales (a partir de eigenvalues y eigenvectors) que mejorse adecuan a la variabilidad de los datos.

Para la implementación **from scratch** del PCA, me base en los enlaces que aparecen en el correspondiente apartado de la Bibliografía

Aplicando PCA sobre el conjunto de entrenamiento logré mejorar la precisión y el rendimiento gracias a que ahora se capturan las características que capturan la mayoría de variabilidad en los datos, reduciendo así el número total de características a probar. Al haber un menor número de píxeles ha considerar, el rendimiento va a ser obviamente menor. Y por otro lado, al capturar los datos con mayor impacto en la predicción de cada clase, se logra reducir el **ruido** en los datos de entrenamiento, lo cual, por definición, también conlleva a una mayor precisión

El nuevo parámetro añadido, **n_components** se utiliza para especificar el número de componentes principales que se deben retener.

En cuanto a las gráficas, observamos una mejora en torno a una décima en base a la tarea 1B, así como una mejora notable en el tiempo de ejecución (por las razones anteriormente mencionadas)

Vemos que el número óptimo para **n_components** es de 30 para T=50 y A=50. Esto es así debido a que, si se aumenta **n_componentes** sin criterio alguno, se puede reducir la eficacia de la **eliminación de ruido**

En cuanto a T y A (T nuevamente vuelve a tener un impacto mucho mayor que A), vemos como la precisión alcanza su máximo y empieza a allanar para T = 100.

De esta forma, una combinación óptima para obtener la mejor proporción de precisión/tiempo_ejecución sería T = 150, A = 50 y n_components = 30, lo cual nos daría una precisión muy cercana a 0.980

Nota: Estos cambios también han disminuido de forma notable el tiempo de ejecución para el ADABoost multiclase, pero la precisión sigue siendo básicamente la misma. Esto es así por lo siguiente: aunque el clasificador binario que predice la clase correspondiente se haya vuelto más preciso, el resto de clasificadores binarios también se han vuelto más precisos, con lo que, en proporción, el ADABoost multiclase va a estar en las mismas que en la tarea 1C a la hora de elegir el clasificador binario más acertado, puesto que **las diferencias de precisiones** entre el clasificador que predice la clase y las de aquellos que no la predicen se van a mantener prácticamente igual.

Tarea 1E

En la tarea anterior, hemos observado como la precisión del ADABoost va aumentando, y de la misma manera, como las gráficas se van allanando cada vez más. Esto es un claro indicio de que nos estamos acercando al sobreentrenamiento.

La implementación de un ADABoost Binario que pare el entrenamiento cuando detecte overfitting ha cosnsitido en lo siguiente:

-. En primer lugar, ahora se considerarán dos conjuntos de entrenamiento diferentes: el de **entrenamiento verdadero**, el cual se usa para entrenar a cada Tocón de decisión; y por otro lado, el conjunto de **validación**, el cual se utiliza para comprobar si al añadir el nuevo DecissionStump a la lista de clasificadores débiles supone una mejora o no de la mejor precisión obtenida hasta el momento. En el momento en que añadir el nuevo clasificador débil implique un empeoramiento **real** de la precisión, dicho clasificador débil se eliminará de la lista y se devolverá la mejor precisión encontrada antes de añadir dicho clasificador.

-. El número de muestras asignadas al conjunto de **entrenamiento verdadero** y al conjunto de **validación** viene determinado por el parámetro **split_proportion.** De esta forma, para **Split_proportion** = 0.90 se estaría asignando un 90% de datos al conjunto de **entrenamiento verdadero** y un 10% de datos para el **conjunto de validación.** Es importante no asignar un número elevado de muestras al **conjunto de validación** para así evitar tiempos de ejecución exorbitantemente elevados

-. Esta nueva lógica implica inicializar el número de clasificadores débiles a 0, ya que ahora lo que se busa es obtener el "**número óptimo de clasisifcadores débiles**"

-. Para comprobar la precisión al añadir el nuevo clasificador débil sobre el conjunto de **validación**, se promedia el método **predict** de la clase ADABoost sobre el propio conjunto de validación **un número parametrizado de veces**. Este número viene determinado por el parámetro **iter_number**. Es necesario realizar este promedio para así evitar desviaciones estadísticas debido al indeterminismo intrínsecco de ADABoost

-. Una vez obtenida la precisión promediada, se compara esta precisión con la mejor precisión encontrada hasta el momento. Ambas precisiones se redondean a

un número parametrizado de cifras significativas (así se puede preseleccionar desde el código cliente con que nivel de precisión se quiere obtener la propia precisión del ADABoost). Este parámetro viene determinado por el parámetro **round2**

-. Si la nueva precisión es menor que la mejor precisión (ambas redondeadas a **round2** cifras signifficaativas), entonces se saca el nuevo clasificador débil de la lista y se detiene el entrenamiento (ya que se ha detectado sobreentrenaamiento).

-. Finalmente, para añadir un mayor grado de practicdad y para evitar incurrir en bucles infinitos, se han añadido los siguientes parámetros de ejecución respectivamente: **bestAccuracyBreak** y **practicalAccuracyBreak**. La lógica de ambos es la siguiente:

-. **bestAccuracyBreak** se encarga de parar el entrenamiento una vez se haya alcanzado una precisión sobre el conjunto de validación mayor o igual que **bestAccuracyBreak**

-. practicalAccuracyBreak se encarga de comprobar si la mejor precisión encontrada hasta el momento (redondeada a **round2** cifras) no ha cambiado tras un número = **practicalAccuracyBreak** de iteraciones. Esto sirve por lo siguiente. Supongamos que round2=3. Podría pasar que en cierto punto la precisión al añadir un nuevo clasificador se quede estancada en un intervalo de forma infinita, como por ejemplo, entre 0.980 y 0.981. Si la precisión se queda estancada en dicho intervalo, la condición **if round(newAccuracy, round2) < round(bestAccuracy, round2)** nunca se alcanzaría, con lo que se acabaría incurriendo en un bucle infinito y el entrenamiento no terminaría nunca

-. Finalmente, mencionar que a la condición if round(newAccuracy, round2) < round(bestAccuracy, round2) se le añade la comprobación and self.T > 10:

Esto se debe a que, para determinadas clases, durante los 10 primeros clasificadores débiles existen decrementos tempranos de la precisión. Por ejemplo, para el dígito 5 con A=200 y Split_proportion = 0.90, la precisión en T= 6 baja bastante con respecto a T=5, pero luego vuelve a subir en T=7 (con lo que este caso no estaría representando un sobreentrenamiento real). Con este añadido, nos aseguramos de que siempre se detecte un sobreentrenamiento **verdadero**.

-. Si el parámetro de verbose se pone a True, se mostrá cada clasificador que se va añadiendo con el siguiente formato:

Classifier 8: error = 0.3564595213090047, alpha = 0.295382120094494, newAccuracy = 0.9333333333333325, bestAccuracy = 0.9324894514767927

Donde newAcurracy y bestAcccuracy hacen referencia al conjunto de validación.

Con todo esto, al ejecutar, por ejemplo:

```
accuracy = run_adaboost_for_one_digit_detecting_overfitting(digit=0, A=200,
                                                            verboseParam=True,
                                                            split_proportion=0.90, iter_number=50,
                                                            round1 = 3,
                                                            round2 = 3,
                                                            bestAccuracyBreak = 0.999,
                                                            practicalAccuracyBreak = 500)  # Ejecutamos AdaBoost
```

Obtenemos lo siguiente:

```
Se ha detectado sobreentrenamiento. El número óptimo de clasificadores débiles es: 218
El número óptimo de clasificadores débiles es: 218
La precisión obtenida en el entrenamiento ha sido de: 0.981
Accuracy for digit 0: 0.9758
```

Se obtiene una precisión del 0.9758, lo cual es totalmente congruente con lo observado en las gráficas de la tarea 1B, donde la gráfica se empezaba a aplanar un poco antes de legar a 0.97 (y como ya sabemos, que una gráfica se comience a aplanar es un claro indicio de acercamiento al sobreentrenamiento)

También es importante destacar que la precisión obtenida en el entrenamiento del conjunto de validación no es la misma que la precisión final debido a que en el conjunto de validación solo se han considerado un 10% de las muestras. Para que ambas fuesen prácticamente idénticas, **Split_proportion** tendría que ser igual a 0.50. Pero ya hemos visto que esto último no tendría sentido, puesto que llevaría a tiempos de ejecución excesivamente elevados

En cuanto a los resultados de la experimentación, aquí lo que interesa no es ver cual es la precisión máxima que podemos llegar a obtener, sino determinar **como afectan** en la precisión final obtenida los parámetros A y split_proportion.

Por esta razón, para cada experimentación se ejecutará:

```
accuracy = run_adaboost_for_one_digit_detecting_overfitting(
    digit, A=A, verboseParam=False, split_proportion=split_proportion, iter_number=T_fixed, graph_param=True, bestAccuracyBreak= 0.95
)
```

Observamos como fijamos **bestAccuracyBreak = 0.95** para poder así acelerar la ejecución de la experimentación (cuando se alcance una precisión de 0.95 sobre el **conjunto de validación** entonces e detendrá el entrenamiento)

-.Notemos que variar los valores de A no ocasiona cambios significaativos en la precisión obtenida (para A distinto de 30, todos los valores de precisión están entre 0.940 y 0.948 para una **split_proportion** de 0.9)

-. En cuanto al valor de **split_proportion,** observamos nuevamente que no varía apenas para valores comprendidos entre 0.70 y 0.90. Asimismo, observamos como los tiempos de ejecución para estos valores son rápidos

Por lo tanto, una combinación óptima de estos valores sería la que hemos mostrado de **A=200** y **split_proportion = 0.90,** con la que se obtiene una precisión en el conjunto de validación del 0.981 y de 0.9758 de precisión en el conjunto de test

A modo de conclusión, esta versión de ADABoost es mucho más conveniente puesto que obtiene de forma dinámica el mejor valor posible para **T** dado un **A** fijado y al mismo tiempo se minimiza en muy gran medida el tiempo de ejecución gracias a la combinación con **split_proportion**. También cabe destacar que este ADABoost obtiene precisiones más bajas que el de la tarea1C ya que, aunque se detecte el sobreentrenamiento, esto no lleva a que se reduzca la dimensionalidad ni preseleccionando índices de características representativas. Ambos son conceptos diferentes

Tarea 2A

Tras haber tenido que crear todo desde cero en las aanteriores tareas, poder usar por fin alguna librería externa como sklearn y keras ha sido un gran alivio.

Los parámetros que puede recibir la clase DecissionTreeClassifier son los siguientes:

Criterion: Este parámetro define la función que mide la calidad de hacer un split

Splitter: Determina la estrategia para elegir la división en cada nodo.

Max_depth: Indica La profundidad máxima del árbol.

min_samples_split: El número mínimo de muestras que debe tener un nodo antes de que se pueda dividir. Este parámetro ayuda a controlar el sobreentrenamiento

min_samples_leaf: El número mínimo de muestras que debe tener una hoja.

Min_weight_fraction_leaf: Indica La fracción mínima ponderada del total de pesos de las muestras de entrada requerida para estar en una hoja del nodo

max_features: Indica El número de características a considerar cuando se busca la mejor división.

Random_state: Controla la aleatoriedad del estimador. Sirve para reproducir los mismos resultados en múltiples ejecuciones, dándole un valor numérico fijo.

Max_leaf_nodes: El número máximo de nodos hoja.

min_impurity_decrease: Un nodo se dividirá si esta división induce una disminución de la impureza mayor o igual a este valor.

class_weight: Permite ponderar las clases si el conjunto de datos es desbalanceado.

Cpp_alpha: Parámetro para la poda de complejidad mínima

Dado que en esta tarea, tenemos que trabajar con un tocón de decisión (es decir, solo dos nodos hoja), los parámetros que nos interesan son los siguientes:

Criterion, splitter, max_depth (que será igual a 1), min_samples_leaf, min_weight_fraction_leaf, max_features (que es el equivalente al parámetro A) y random state

Con profundidad = 0, determinados parámetros no tenían sentido de ser aplicados:

-. min_samples_split: Este parámetro especifica el número mínimo de muestras necesarias para dividir un nodo interno. En un árbol de profundidad 1, solo hay una división (la raíz), y este parámetro no influiría más allá de esta división inicial.

-. max_leaf_nodes: Dado que un árbol de profundidad 1 tendrá como máximo dos nodos hoja (correspondientes a las dos clases resultantes de la única división), establecer un límite en el número máximo de nodos hoja no tiene efecto.

-. min_impurity_decrease: Este parámetro controla cuánto debe disminuir la impureza para justificar una división adicional. En un árbol de un solo nivel, solo se realiza una división (la del nodo raíz), y este parámetro no influirá en decisiones adicionales de división.

De la clase **ADABoostClassifier,** los parámetros más relevantes son los siguientes:

Base_estimator: Este parámetro define el modelo base que AdaBoost utilizará para el proceso de boosting. Este parámetro tomará el valor que hayamos construido para el DecissionTreeClassifier

n_estimators: Especifica el número de estimadores que se van a utilizar (es equivalente al parámetro **T)**

algorithm: Este parámetro determina el algoritmo de boosting que se empleará. 'SAMME' es una versión multiclase del algoritmo AdaBoost original

random_state: igual que el descrito para el DecissionTreeClassifier

De esta forma, ya hemos descrito que cambios hay que hacer para que esta implementación se comporte de igual forma que la nuestra **from scratch** en base a **T** y **A**: hacer que max_features=A en el constructor de DecissionTreeClassifier y hacer que n_estimators = T en el constructor de AdaBoostClassifier.

Finalmente, para la experimentación, los parámetros seleccionados han sido los siguientes:

max_depth, min_samples_leaf, min_weight_fraction_leaf, max_features y random state.

Destacar lo siguiente:

-. **T** y **A** se comportan de forma análoga a como lo hacían en la implementación from scratch

-. **Num_samples_leaf_values**: al aumentar su valor, la precisión no cambia, pero sí que se reduceun poco el tiempo de ejecución

-. Min_weight_fraction_leaf_values: vemos como al aumentar su valor, la precisión se mantiene estable, hasta que cae en picado cuando se toma el valor de 0.45. Esto se debe a que **Min_weight_fraction_leaf_values** solo puede tomar valores reales entre 0.0 y 0.5, de forma que cuando se está muy próximo a 0.5, la precisión tiende a disminuir de forma muy notable.

Teniendo en cuenta todo esto, una combinación óptima de estos parámetros sería:

(T=140, A = 100, Num_samples_leaf_values = 30,
Min_weight_fraction_leaf_values = 0.3)

Con esta combinación se obtendría una precisión de alrededor de 0.985 (la cual supera a nuestra versión de ADABoost del 1C, pero por poco. Se ve que lo implementamos bien). Y todo esto con un tiempo muy inferior a nuestro ADABoost Multiclase

Tarea 2B

En este caso, al tener un árbol de decisión con profundidad mayor que 0, los parámetros considerar serían los siguientes:

Criterion, splitter, max_depth, min_samples_split, min_samples_leaf, min_weight_fraction_leaf, max_features (que es el equivalente al parámetro A) random state, max_leaf_nodes, min_impurity_decrease, max_depth

Al tener ahora un DecissionTreeClassifier con profundidad mayor que 1, ahora **SÍ** que tiene sentido aplicar todos estos parámetros.

En cuanto a la experimentación:

-. Los parámetros **T, A, Num_samples_leaf_values Min_weight_fraction_leaf_values** se comportan de forma análoga que en el apartado anterior

-. Min_samples_split: podemos observar que la variación en este parámetro no ha generado ninguna variación en la precisión obtenida. De igual forma, el tiempo de ejecución apenas cambia para las variaciones hechas en este parámetro

-. Min_samples_leaf: exactamente igual que el parámetro anterior

-. Max_leaf_nodes: exactamente igual que el parámetro anterior ☹

-. Min_impurity_decrease: mirando la gráfica, se observa que este parámetro tiende a disminuir la precisión cuando va aumentando su valor. Por otra parte, me resulta curioso ver el pico que hay en el tiempo de ejecución cuando toma el valor 0.150. Seguramente se deba a una desviación estadística, la cual se evitaría promediando cada resultado un número mayor a 5 veces (número con el que se han estado promediandl todas las gráficas hasta el momento)

-. Max_depth: este ha sido el parámetro que mayor impacto ha tenido en la mejora de la precisión. Contra más aumentaba, mayor era la precisión. Y además, cuando aumentaba, el tiempo de ejecución solo se incrementaba unas pocas décimas para cada nuevo valor de **max_depth.**

Con todo esto, una combinación óptima de parámetros sería:

(T=140, A = 100, Num_samples_leaf_values = 30,
Min_weight_fraction_leaf_values = 0.3, min_samples_split = 20,
min_samples_leaf = 5, min_leaf_nodes = 30, min_impurity_decrease = 0.050,
max_depth = 12)

Con todo esto, se obtiene una precisión en torno a 0.997 (ahora sí que ha superado a nuestro ADABoost propio). Y todo esto con un tiempo muy inferior a nuestro ADABoost Multiclase

Tarea 2C

Para implementar el perceptrón multicapa, me he basado principalmente en el código que aparecía en las transparencias de teoría

Comencemos un con una breve definición acerca de qué es un MLP:

Definición

Un perceptrón multicapa (**MLP)** es un tipo de red neuronal artificial compuesta por una capa de entrada, una o más capas ocultas, y una capa de salida. Funciona de la siguiente manera:

-. Capa de Entrada: Recibe los datos de entrada.

-. Capas Ocultas: Procesan la información de entrada mediante neuronas conectadas. Cada neurona aplica una función de activación a la suma ponderada de sus entradas, permitiendo capturar patrones complejos.

-. Capa de Salida: Produce la salida final.

Parámetros más relevantes

Número de capas:

-. Añaden profundidad al modelo, permitiendo que aprenda características más complejas.

-. Más capas pueden captar más detalles, pero pueden hacer el modelo más complejo y propenso al sobreentrenamiento.

Neuronas por capa:

-. Determinan la capacidad de cada capa para aprender patrones.

-. Más neuronas pueden mejorar el aprendizaje, pero también aumentan el riesgo de sobreajuste y el costo computacional.

Organización de las capas:

-. Define la estructura del modelo, cómo las capas están dispuestas y conectadas.

-. La correcta organización mejora la eficiencia del aprendizaje y la capacidad del modelo para generalizar.

Batch_size:

-. Cantidad de muestras que el modelo procesa antes de actualizar los parámetros.

-. Tamaños más grandes pueden hacer el entrenamiento más estable pero más lento; tamaños más pequeños permiten actualizaciones más frecuentes pero menos estables.

Learning_rate:

-. Controla el tamaño de los pasos de actualización durante el entrenamiento.

-. Learning rates altos pueden hacer que el modelo converja rápidamente pero con riesgo de no encontrar el mínimo óptimo; learning rates bajos aseguran convergencia pero pueden hacer el entrenamiento muy lento.

Algoritmo de optimización (Adam):

-. Actualiza los pesos del modelo basándose en los gradientes calculados durante el backpropagation.

-. Afecta la velocidad y la calidad de la convergencia del modelo. Adam combina las ventajas de otros dos optimizadores, Adagrad y RMSProp.

Función de activación (ReLU y softmax):

-. ReLU introduce no linealidad en las capas ocultas, permitiendo aprender funciones complejas. Softmax convierte las salidas en probabilidades.

-. ReLU ayuda a evitar el problema de desvanecimiento de gradientes, mientras que softmax facilita la interpretación de las salidas del modelo en tareas de clasificación.

En cuanto a la experimentación:

-. Number of layers: vemos que la precision apenas varía de unos valores a otros. Lo que sí varía es el tiempo de ejecución (y mucho) cuando se aumenta el número de capas

-. Number of Neurons: vemos como alcanza su punto máximo de precisión para un valor de 1000, y a partir de ahí empieza a descender. En cuanto al tiempo de ejecución, vemos que se mantiene bastante caótico.

-. Batch Size: Observamos como la precisión se mantiene prácticamente sin cambios. Lo interesante aquí es ver como el tiempo de ejecución disminuye de forma drástica cuando se va aumentando el **Batch Size**

-. Learning Rate: Observamos como resulta ser una pésima idea aumentar el valor de este parámetro, puesto que la precisión cae en picado hacia lo más bajo.

-. Epoch values: número total de veces que el algoritmo de aprendizaje trabajará a través del conjunto completo de datos. Cada epoch representa una iteración completa sobre todos los datos de entrenamiento, permitiendo al modelo ajustar sus pesos para mejorar su precisión.

Con todo esto, una combinación óptima para estos parámetros sería:

(number_of_layers = 1, number_of_neurons = 1000, batch_size = 1000, learning_rate = 0.0001)

Con todo esto, se obtiene una precisión en torno a 0.983 en un tiempo de ejecución en torno a 40 segundos

Tarea 2D

Vamos a comenzar con una breve definición acerca de las redes **CNN**

Definición

-. Las redes neuronales convolucionales (CNN) son una clase de redes neuronales profundas, especializadas para procesar datos con una estructura en forma de cuadrícula, como imágenes.

-. Las redes neuronales convolucionales (CNN) funcionan aplicando filtros en capas convolucionales para extraer características visuales de imágenes. Estas características se simplifican en capas de pooling, se normalizan y se activan para mejorar la eficiencia del aprendizaje. Finalmente, se aplanan y se procesan en capas densas que clasifican o predicen resultados basados en las características detectadas. Las CNN ajustan automáticamente sus filtros y parámetros durante el entrenamiento para optimizar el rendimiento en tareas específicas.

Parámetros y funciones más relevantes

-. num_conv_layers: Número de capas convolucionales en la red. Cada capa convolucional puede ayudar a capturar características más complejas de la imagen.

-. num_dense_layers: Número de capas densas (o completamente conectadas) que siguen a las capas convolucionales. Estas capas combinan las características aprendidas por las capas convolucionales para realizar la clasificación final.

-. BatchNormalization: Técnica para estabilizar y acelerar el aprendizaje al normalizar la entrada de cada capa dentro de una mini-batch. Ayuda a combatir el problema de los cambios en la distribución de las entradas durante el entrenamiento (problema de cambio de covariable).

-. LeakyReLU: Variante de la función de activación ReLU que permite un pequeño gradiente cuando la unidad no está activa (es decir, permite que los valores negativos tengan un gradiente distinto de cero). Ayuda a evitar el problema de las neuronas "muertas" en una red.

-. MaxPooling2D: Operación que reduce la dimensionalidad espacial de las entradas (altura y anchura), lo cual es útil para disminuir la cantidad de parámetros y la carga computacional, y también para extraer características dominantes manteniendo la invariancia espacial.

-. Dropout: Técnica de regularización donde aleatoriamente algunas unidades de la red se "apagan" (es decir, se les asigna un peso de cero) durante el entrenamiento. Esto ayuda a evitar el sobreajuste al forzar a la red a aprender patrones redundantes.

- . Flatten: Transforma la matriz de características multidimensionales a un vector de una sola dimensión, permitiendo que los datos puedan ser procesados en capas densas.

-. loss='categorical_crossentropy': Función de pérdida usada para problemas de clasificación multiclase, que compara la distribución de las predicciones de la red con la distribución real de las etiquetas (en formato de one-hot encoding) y mide el desempeño de la red en términos de probabilidad.

-. Epoch values: número total de veces que el algoritmo de aprendizaje trabajará a través del conjunto completo de datos. Cada epoch representa una iteración completa sobre todos los datos de entrenamiento, permitiendo al modelo ajustar sus pesos para mejorar su precisión.

En cuanto a la experimentación, no he podido realizar la función que muestre las gráficas correspondientes debido a falta de tiempo.

No obstante, ejecutando:

```
model = build_cnn_model(num_conv_layers=2, num_dense_layers=1, conv_filters=[32, 64], dense_units=[128])
```

He obtenido una precisión de 0.9941 tras haber promediado 5 veces. Al igual que lo que ocurría con el perceptrón multicapa, parece ser que la variación en los valores de los parámetros llevarían a lo sumo en unas pocas décimas extras de precisión.

Cabe destacar que las redes CNN se comportan tremendamente bien con las imágenes MNIST debido a que están **especializadas** para reconocer patrones espaciales y jerarquías de características en datos con una estructura en forma de **cuadrícula** (como las propias imágenes MNIST)

Tarea 2E

Finalmente, tras todo lo explicado y tras todas las experimentaciones realizadas. Podemos concluir la siguiente ordenación de peor a mejor clasificador

Top 7: Tarea 1C

El clasificador multiclase es útil para clasificar con mayor precisión aquellas clases que tienden a ser mal clasificadas por un **ADABoost Binario** individual. No obstante, la precisión de este clasificador (en torno a 0.90) es bastante inferior al resto de clasificadores

Top 6: Tarea 1E

Gracias a la implementación de la detección del sobreentrenamiento, se pudo crear un ADABoostBinario que calculase de forma dinámica el valor óptimo para el parámetro T. Esto mejoró con creces la automatización en la búsqueda de la mejor combinación de parámetros. No obstante, el mero hecho de detectar sobreentrenamiento no hace que se pueda alcanzar una precisión "superior" (Con esta versión se alzanca una precisión en torno a 0.975 en el mejor caso. Es por eso eso que la tarea 1D está en el siguiente puesto

Top 5: Tarea 1D

Al haber usado la función reshape para trabajar con imágenes 1D en lugar de 2D y, sobre todo, tras haber implementado PCA para preseleccionar los índices de características más relevante; gracias a esto se consiguió mejorar la precisión de nuestro ADABoost binario (llegando a 0.981 en el mejor de los casos), sino que también se llegó a acelerar por bastante el tiempo de ejecución, gracias a que ahora había que considerar menos características en el conjunto de entrennamiento

Top 2: Tarea 2C

Gracias a la utilización de un MLP, logramos, en el mejor de los casos, conseguir una precisión en torno a 0.983 con un tiempo de ejecución de unos 40 segundos

Top 3: Tarea 2 A

Gracias a la utilización de la librería sklearn, la cual está completamente optimizada, se pudo obtener una precisión en torno a 0.986 en el mejor caso, en unos 35 segundos de ejecución El problema de esta versión es que la restricción de que la profundidad máxima fuese de 1 limitaba mucho su máximo potencial

Top 2: Tarea 2D

Aprovechando que las redes CNN están **especializadas** para reconocer patrones espaciales y jerarquías de características en datos con una estructura en forma de **cuadrícula** (justo como nuestras queridas imágenes MNIST), se logró obtener una asombrosa precisión de 0.9924 en un tiempo aproximado de unos 2 minutos

Top 1: Tarea 2B

Y finalmente, el clasificador más eficiente y eficaz para resolver el problema de reconocer los dígitos del 0 al9 de MNIST ha sido **el clasificador ADABoost Multiclase usando sklearn con DecissionTreeClassifier.max_depth > 1.** Al romper la resticción de la max_depth, se lofró una increíble precisión del **0.9974** en alrededor de un minuto medio, superando así a las redes CNN no solo en eficacia, sino también en **eficiencia**. Por lo tanto, concluimos con que este es el clasificador **más óptimo** para resolver el problema propuesto

BIBLIOGRAFÍA

En primer lugar, mencionar que he utilizado dos IAs en las siguientes partes:

-. En primer lugar, he usado **Github Copilot** para poder cmentar de forma rápida **todas** las líneas de código implementadas por mí misma. La razón por la que hice esto fue debido a que fui realizando esta práctica de forma alternada, debido a que tenía bastante trabajo también con otras asignaturas. De este modo, cuando volvía a ponerme con la práctica, era capaz de volver a comprender el concepto de mi propia implementación en sollo unos instantes. He de decir que tuve que cambiar muchos de los comentarios que me daba **GitHub Copilot** ya que había varios que no eran correctos (por ejemplo, confundía el parámetro **T** con el parámetro **A**).

-. En segundo lugar, he usado ChatGPT 4 para poder realizar las funciones de mostrar las gráficas de la experimentación de cada tarea de un modo mucho más rápido. La razón de esto es que **TODAS** las funciones de experimentación siguen la misma estructura. De esta manera, al tratarse de tareas repetitivas, ChatGPT me ayudó a agilizar mucho este proceso, y al tratarse de código sencillo, casi siempre hacía bien estas funciones a la primera (salvo en la experimentación del **MLP,** en la cual tuve que ajustar por mi cuenta muchas cosas, y en la de las redes **CNN**, las cuales ya no me dieron tiempo a realizar su experimentación)

Para el resto de tareas, adjunto a continuación los enlaces de los sitios webs de los cuales he utilizado o buscado información

TAREA 1A

(Apuntes de Teoría)

https://www.geeksforgeeks.org/implementing-the-adaboost-algorithm-from-scratch/

https://towardsdatascience.com/adaboost-from-scratch-37a936da3d50

https://www.kdnuggets.com/2020/12/implementing-adaboost-algorithm-from-scratch.html

TAREA 1D

https://www.geeksforgeeks.org/principal-component-analysis-pca/

https://medium.com/accel-ai/pca-algorithm-tutorial-in-python-93ff19212026

https://www.cs.cmu.edu/~mgormley/courses/10601-s18/slides/lecture30-pca-adaboost.pdf

https://www.researchgate.net/publication/225125789_PCA_Enhanced_Training_Data_for_Adaboost

TAREA 2A Y 2B

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

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

https://scikit-learn.org/stable/modules/tree.html

TAREA 2C

(Apuntes de Teoría)

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

https://medium.com/@artjovianprojects/deep-learning-project-multilayer-perceptron-e34017941918

TAREA 2D

(Apuntes de Teoría)

https://keras.io/api/layers/convolution_layers/

https://www.tensorflow.org/tutorials/images/cnn