25-26

Práctica 2 – Sistemas Inteligentes

INGENIERÍA INFORMÁTICA – UNIVERSIDAD DE ALICANTE

LUIS BERMÚDEZ FRANCO

MLP1: Lo primero que tenemos que hacer es procesar los datos de CIFAR-10 para MLP, para ello declaro el módulo **cargar_preprocesar_cifar10()** donde cargo CIFAR-10 mediante **keras.datasets.cifar10.load_data()**, luego normalizo los valores de los píxeles del rango [0,255] al rango [0,1] para evitar cálculos mayores y evitar desbordamiento. Ahora también aplano las imágenes a un vector de 3072 características como se indica en el enunciado y convierto las etiquetas a formato one-hot para representar cada clase como un vector de 10 elementos donde la posición del valor 1 me indique el tipo de imagen.

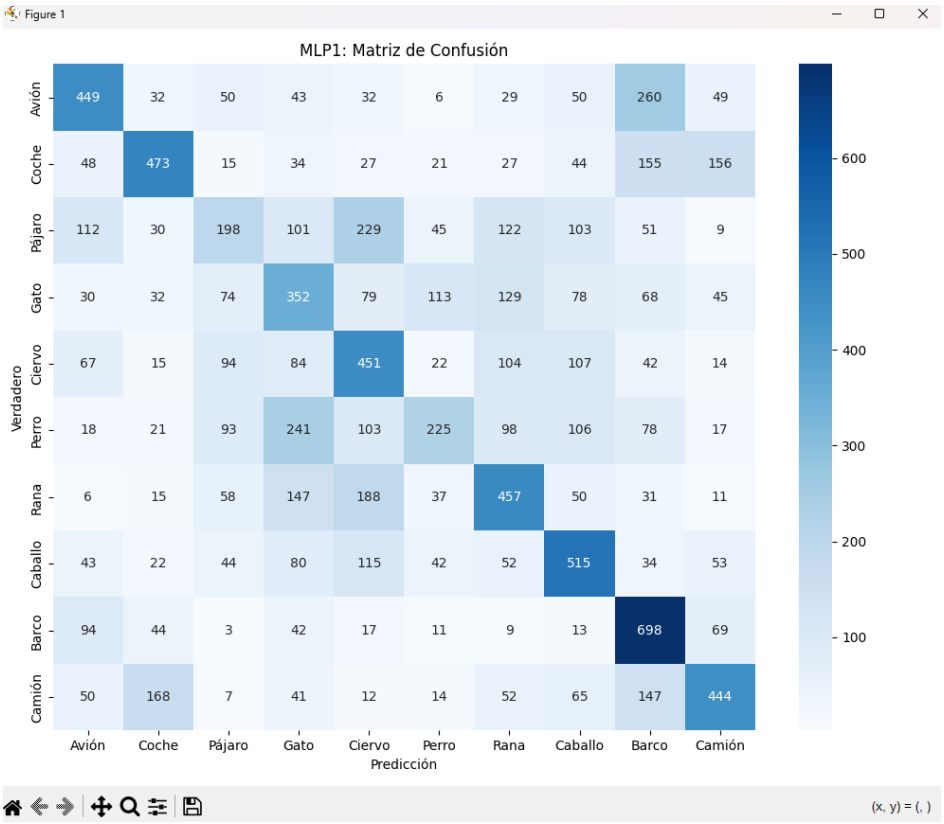
A continuación, tengo que definir la arquitectura del MLP mediante **Sequential** de Keras donde especifico la cantidad de neuronas de la capa oculta, en este caso 48, función de activación **sigmoid** que transforma cualquier número real en un valor entre 0 y 1, y le indicamos la entrada de 3072 características. también defino la capa de salida con 10 neuronas, 1 por cada tipo, modo de activación **softmax** que convierte el vector en otro de probabilidades de cada clase. Se pide que utilice el optimizador Adam, la función de pérdida **categorical_crossentropy** y la métrica accuracy, todo ello lo defino con **model.compile** indicando en *optimizer*, *loss* y *metrics* cada uno de estos valores.

Ahora toca entrenar la modelo usando la función **fit** con los valores indicados en el enunciado para cada uno de los parámetros. Finalmente, evaluó el modelo utilizando **evaluate** y genero gráficas para visualizar los resultados mediante los módulos auxiliares **plot_evolucion_entrenamiento** y **plot_matriz_confusion.**

El módulo plot_matriz_confusion me sirve para analizar los errores de clasificación ya que se encarga de analizar los resultados finales por clases para así poder entender que clase es la que más se equivoca nuestro modelo en identificar.

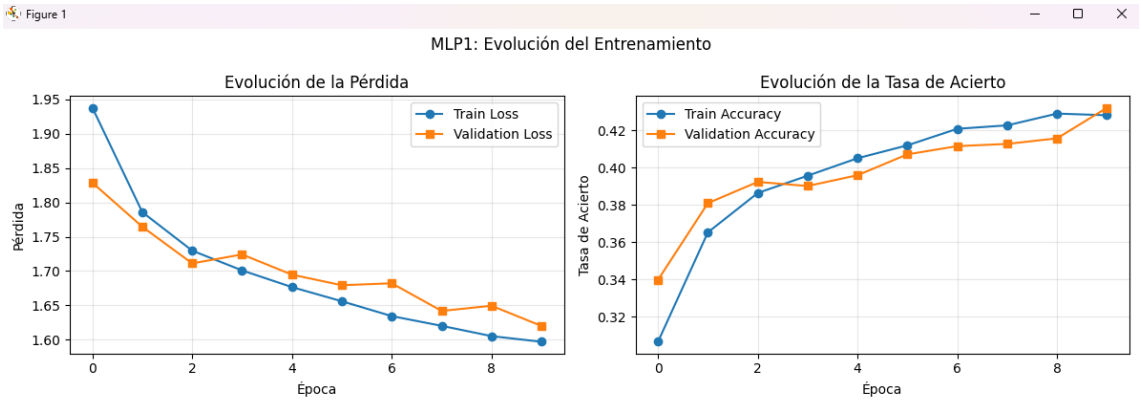
Recibe las etiquetas reales y las predicciones y calcula la matriz de confusión mediante la función **confusion_matrix**

El resultado que muestra es un mapa de calor con las clases de CIFAR-10, donde los valores de la diagonal son los aciertos por cada clase, y todos los valores que están fuera de esta diagonal son en los casos que nuestro modelo se ha equivocado



El módulo plot_evolucion_entrenamiento muestra la evolución del modelo durante el entrenamiento, es decir, recibe el objeto history generado por la función fit y que guarda las métricas calculadas, y crea dos gráficas.

Una donde se muestra la evolución de la pérdida y otra la evolución de la accuracy. Esto nos permite saber si el modelo está mejorando con el tiempo y si hay sobre entrenamiento.



Cuando realizamos el entrenamiento de nuestro modelo se nos muestran los siguientes datos:

Resumen del modelo:
Model: "sequential"

Layer (type)	Output Shape	Param #
capa_oculta (Dense)	(None, 48)	147,504
capa_salida (Dense)	(None, 10)	490

Total params: 147,994 (578.10 KB)
Trainable params: 147,994 (578.10 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.3066 - loss: 1.9379 - val_accuracy: 0.3398 - val_loss: 1.8287
Epoch 2/10
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 1s 967us/step - accuracy: 0.3653 - loss: 1.7858 - val_accuracy: 0.3810 - val_loss: 1.7645
Epoch 3/10
1407/1407 ━━━━━━━━━━━━━━━━━━━━ 1s 982us/step - accuracy: 0.3865 - loss: 1.7297 - val_accuracy: 0.3924 - val_loss: 1.7111
```

De estos podemos deducir que el modelo sigue aprendiendo ya que el accuracy sigue subiendo en cada época, y que se ha entrenado con 147994 parámetros.

```
Tiempo de entrenamiento: 14.50 segundos

Evaluando modelo con conjunto de test...
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.4262 - loss: 1.6095

Resultados en conjunto de test:
  Pérdida: 1.6095
  Tasa de acierto: 0.4262 (42.62%)

✅ Tarea MLP1 completada.
```

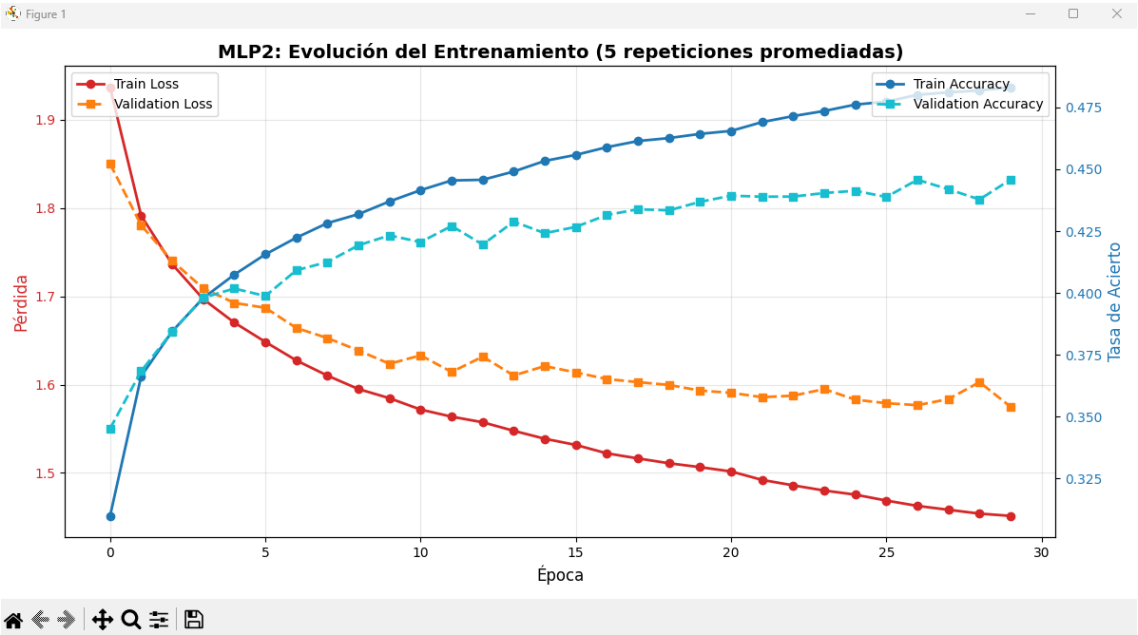
Se pude observar que todavía le falta entrenamiento para mejorar la tasa de aciertos y que todavía nuestro modelo tiene un margen de mejora.

MLP2: Se pide analizar la evolución del entrenamiento del modelo para detectar si ha sido entrenado lo suficiente o si sufre de sobre entrenamiento. Para esto el enunciado nos indica que hay que ajustar el parámetro **epochs**, es decir las épocas, para poder determinar el número optimo de estas que nos proporciona un buen resultado del modelo y detectar cuando se produce sobre entrenamiento.

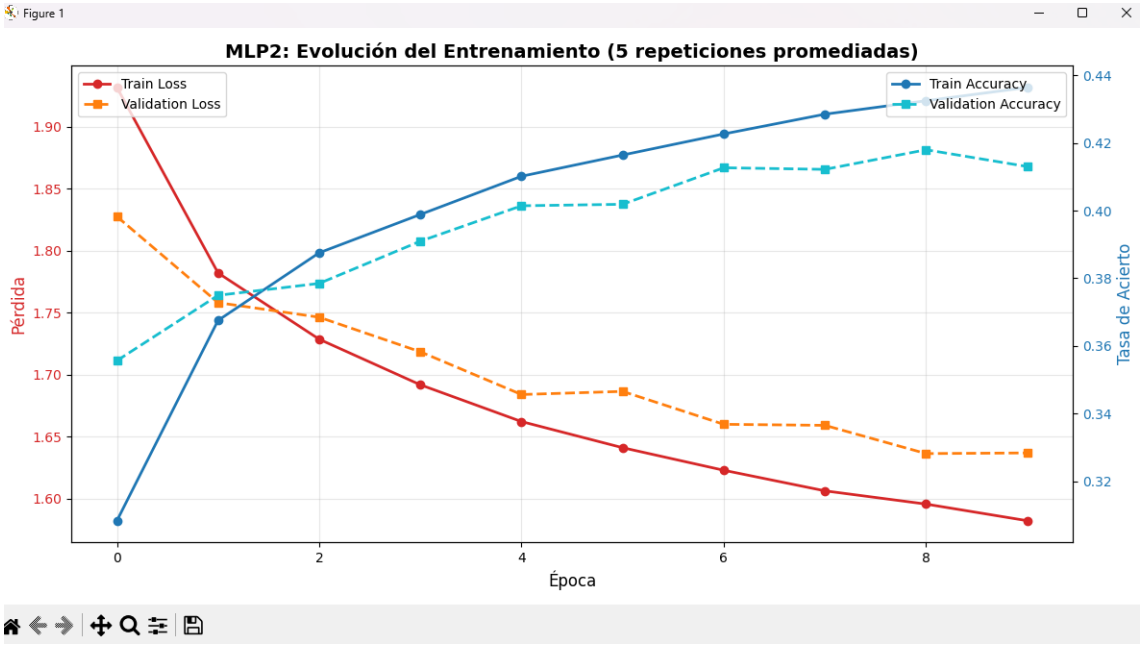
Para analizar los resultados, la practicas nos dice que realicemos 5 repeticiones independientes del mismo entrenamiento, ya que cada una parte de una inicialización aleatoria. Necesito promediar los resultados de las repeticiones para obtener mejores curvas de aprendizaje e identificar mejor los resultados. Para ello implemento la función **promediar_historias()** que calcula el promedio de cada métrica, en este caso pérdida y precisión, mediante las repeticiones para cada época.

Tengo que generar la gráfica para las curvas *train_accuracy, train_loss, validation_accuracy* y *validation_loss*, para ello uso mi función **plot_evolucion_mlp2()**. He realizado 3 casos con 30, 10 y 6 épocas.

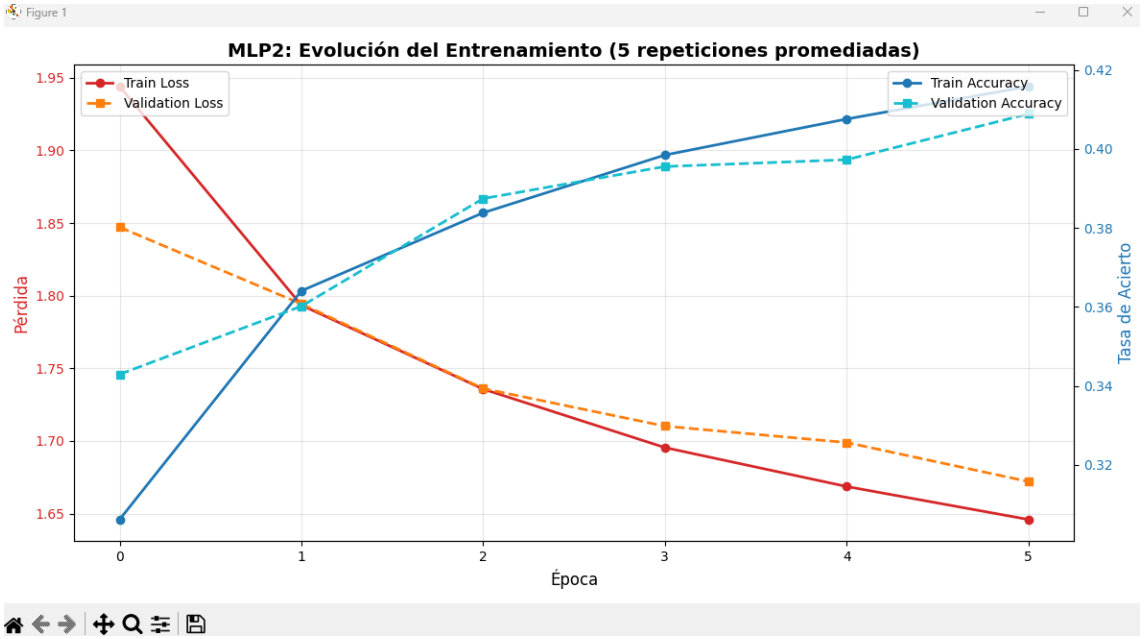
30 épocas



10 épocas:



6 épocas:



Como se puede observar, la curva en la gráfica de 30 épocas es más suave lo que permite un análisis más sencillo. En esa misma gráfica se puede apreciar como train loss va descendiendo durante el transcurso de las épocas, en cambio, validation loss desciende, llegando a su mínimo en la época 21 aprox, pero a partir de esa época vuelve a ascender como pasa en la 29. Podemos concluir que ocurre un sobre entrenamiento, ya que train loss sigue bajando mientras que validation loss no le acompaña, esto sucede a partir de la época 10.

Para la precisión nos fijamos en train accuracy y validation accuracy. Se aprecia que validation accuracy va ascendiendo durante las épocas, y train accuracy también, aunque sufre algunas pequeñas caídas.

En conclusión, se puede decir que la época optima esta entre la 21 y la 24 según validation loss y accuracy, por lo que se recomienda parar ahí el entrenamiento para evitar sobrentrenarlo.

Para ello implemento **EarlyStopping** que me permite automatizar en que época debe de parar el entrenamiento. Lo que hace este modulo es monitorizar las métricas durante el proceso de entrenamiento, y si detecta que no mejoran tras el numero de épocas indicado en su parámetro **patience** detiene el entrenamiento.

Para mejorar este proceso he hecho uso de **restore_best_weights = True** que me permite mantener siempre la época con mejor resultado a pesar de que se procesen más épocas debido al valor de patience.

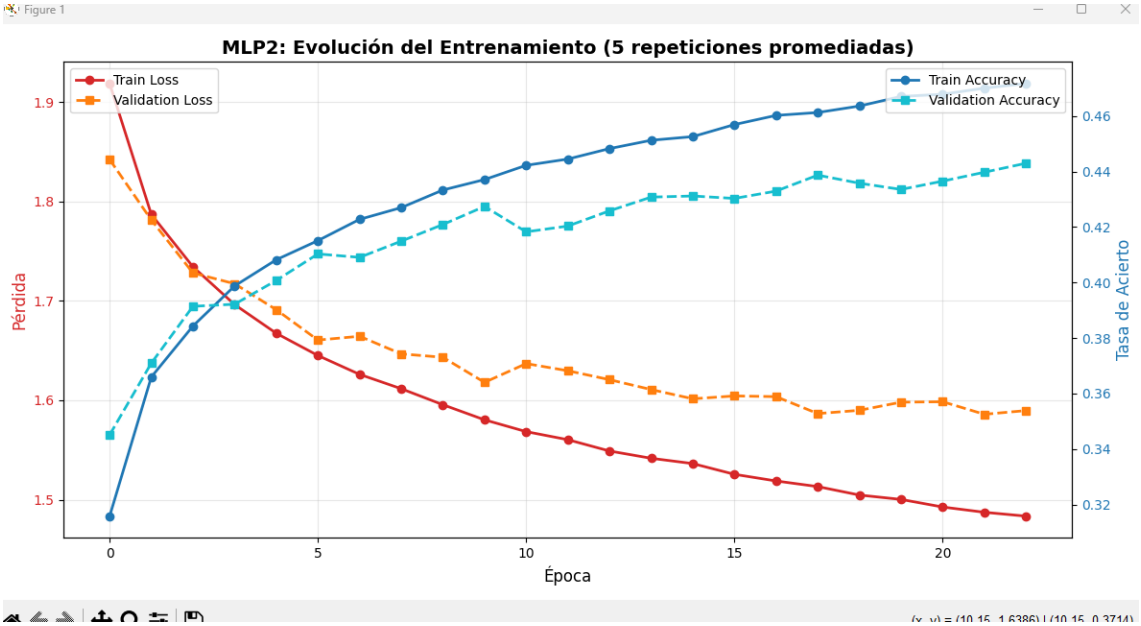
```
[MLP2] Análisis de resultados promediados:
  - Mejor val_accuracy: 0.4458 en época 30
  - Mejor val_loss: 1.5750 en época 30
  - Tiempo promedio por repetición: 66.74s

[MLP2] Probando EarlyStopping con diferentes configuraciones...

[MLP2] EarlyStopping Config 1: monitor=val_accuracy, patience=3
Epoch 9: early stopping
Restoring model weights from the end of the best epoch: 6.
  [MLP2] Entrenamiento detenido en época 9
  [MLP2] Test accuracy: 0.4204 (42.04%)
  [MLP2] Tiempo: 14.15s

[MLP2] EarlyStopping Config 2: monitor=val_loss, patience=5
Epoch 15: early stopping
Restoring model weights from the end of the best epoch: 10.
  [MLP2] Entrenamiento detenido en época 15
  [MLP2] Test accuracy: 0.4316 (43.16%)
  [MLP2] Tiempo: 20.66s

[MLP2] EarlyStopping Config 3: monitor=val_accuracy, patience=5
Epoch 28: early stopping
Restoring model weights from the end of the best epoch: 23.
  [MLP2] Entrenamiento detenido en época 28
  [MLP2] Test accuracy: 0.4339 (43.39%)
  [MLP2] Tiempo: 36.92s

[MLP2] Resumen de configuraciones EarlyStopping:
  Config 1: 9 épocas, test_acc=0.4204, tiempo=14.15s
  Config 2: 15 épocas, test_acc=0.4316, tiempo=20.66s
  Config 3: 28 épocas, test_acc=0.4339, tiempo=36.92s

✅ Tarea MLP2 completada.
[MLP2] Recomendación: Usar 30 épocas (mejor val_accuracy: 0.4458)
```

En la ejecución podemos ver como la mejor configuración es la número 3 donde patience tiene el valor 5 y para en la época 28, aunque se queda con el valor de la

época 23 ya que ha sido la que mejor tasa de acierto ha tenido. Las otras dos configuraciones no se tienen en cuenta ya que la primera al tener patience con valor 3 es demasiado estricta, parando en la época 9 sin permitir a nuestro modelo llegar a su mejor rendimiento. La configuración 2 es válida, pero obtenemos mejor tasa de acierto en la 3.

Configuración	Monitor	Patience	Época (Parar)	Test Accuracy	Tiempo (s)
1	val_accuracy	3	9	42.04%	14.15
2	val_loss	5	15	43.16%	20.66
3	val_accuracy	5	28	43.39%	36.92

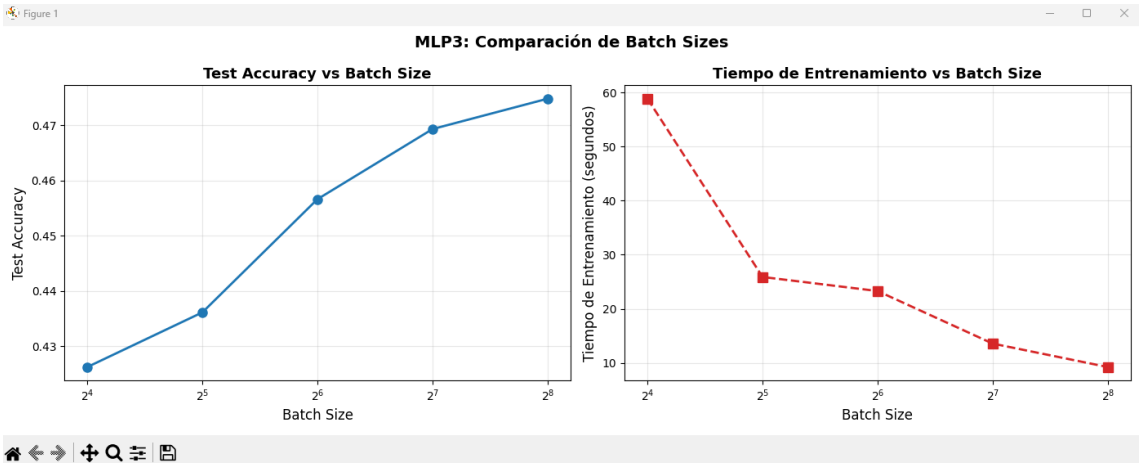
En conclusión, usare la configuración 3 ya que es aproximadamente el valor óptimo y el que mejor resultados me proporciona sin sufrir un alto sobre entrenamiento.

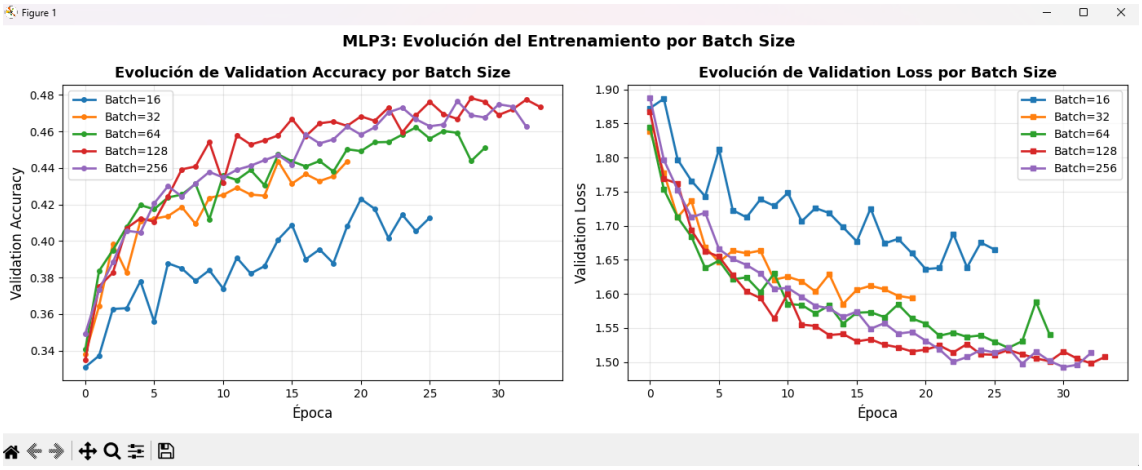


MLP3: En este apartado se me pide experimentar con el parámetro **batch_size** el cual define el número de muestras de entrenamiento que nuestro modelo procesa antes de actualizar sus parámetros internos como los pesos y sesgo, básicamente agrupa en conjuntos de X tamaños, donde X es el valor de este parámetro, los datos de entrenamiento y va corrigiendo de lote en lote.

Para ellos he definido 16,32,64,128 y 256 como valores de batch_size y el número de épocas que he usado como máximo ha sido 50, ya que los batch más grandes no realizan tantas actualizaciones como los pequeños y pueden necesitar más épocas.

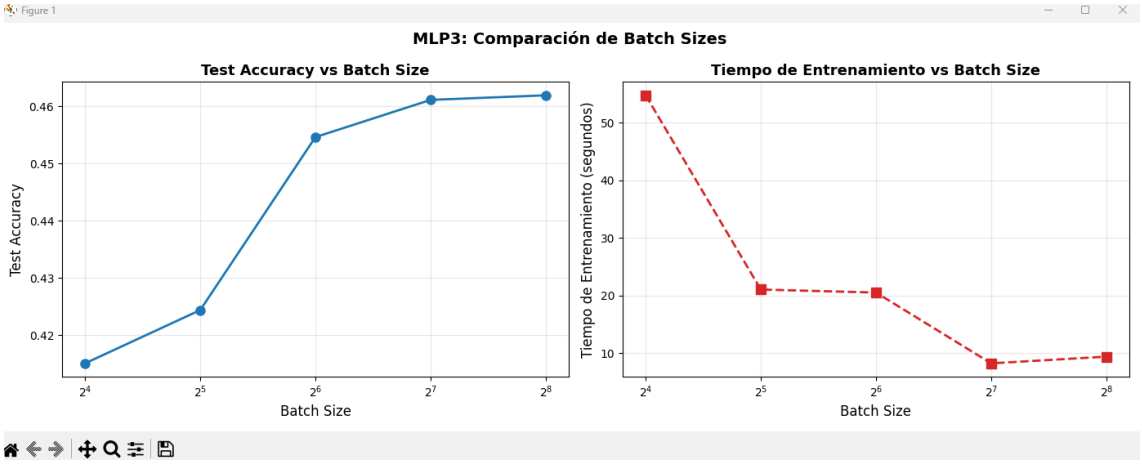
Primera ejecución:

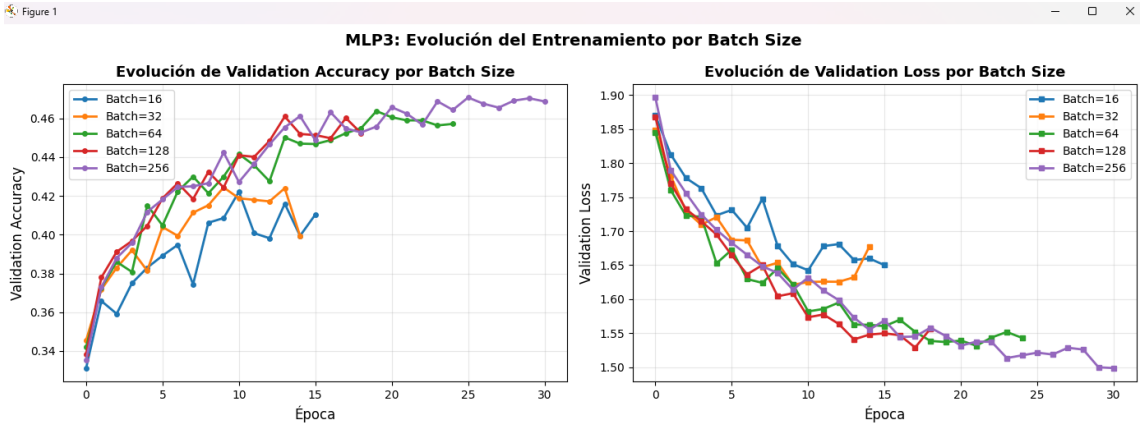




```
[MLP3] Análisis de resultados:
------------------------------------------------------------
Batch Size   Test Acc     Tiempo (s)   Épocas     Updates/Época
------------------------------------------------------------
16           0.4262       58.85        26         2812
32           0.4361       25.89        20         1406
64           0.4566       23.29        30         703
128          0.4693       13.57        34         351
256          0.4748       9.24         33         175

[MLP3] Resumen de mejores resultados:
  - Mejor accuracy: batch_size=256 (accuracy=0.4748)
  - Más rápido: batch_size=256 (tiempo=9.24s)
  - Mejor eficiencia (acc/tiempo): batch_size=256 (eficiencia=0.051386)

[MLP3] Generando gráficas comparativas...

Tarea MLP3 completada.
[MLP3] Recomendación: Batch size óptimo = 256 (eficiencia: 0.051386)
[MLP3] Mejor accuracy: batch_size=256 (accuracy=0.4748)

Presiona Enter para volver al menú...
```

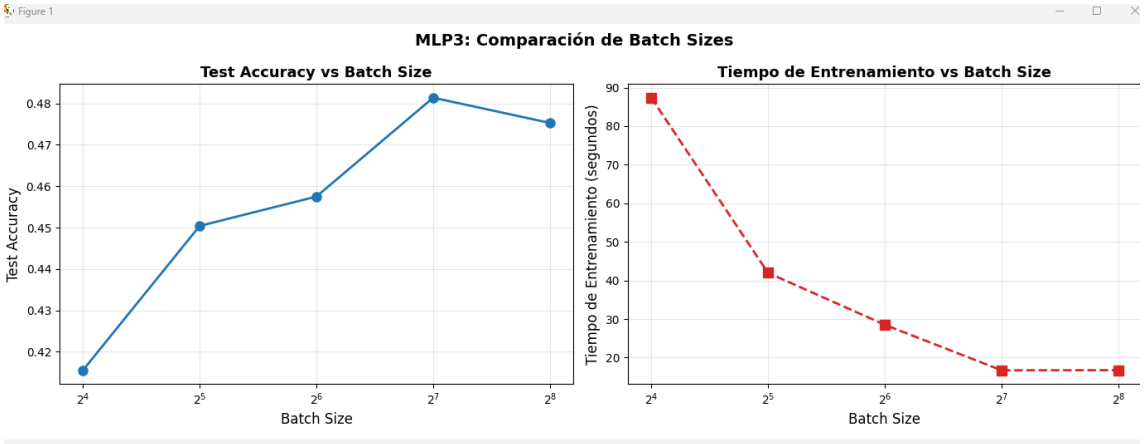
Segunda ejecución:

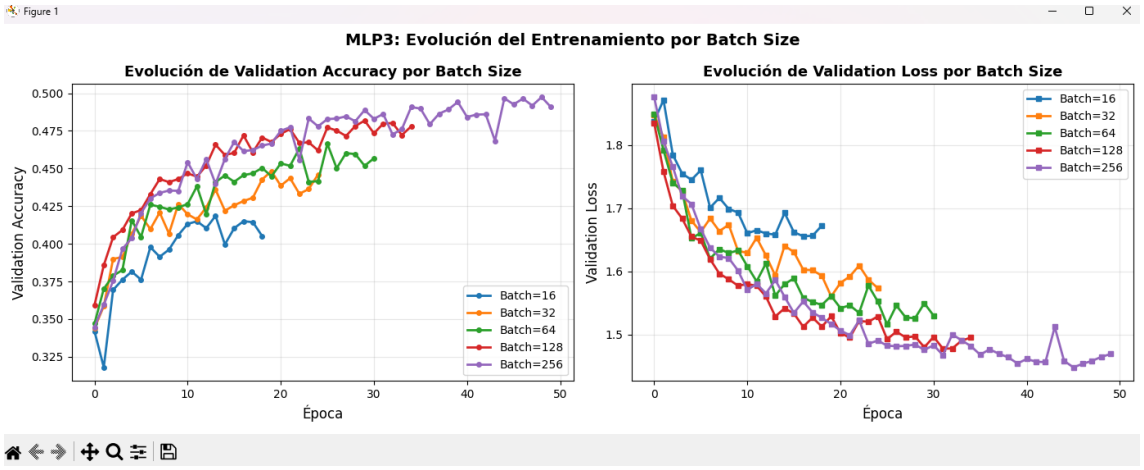


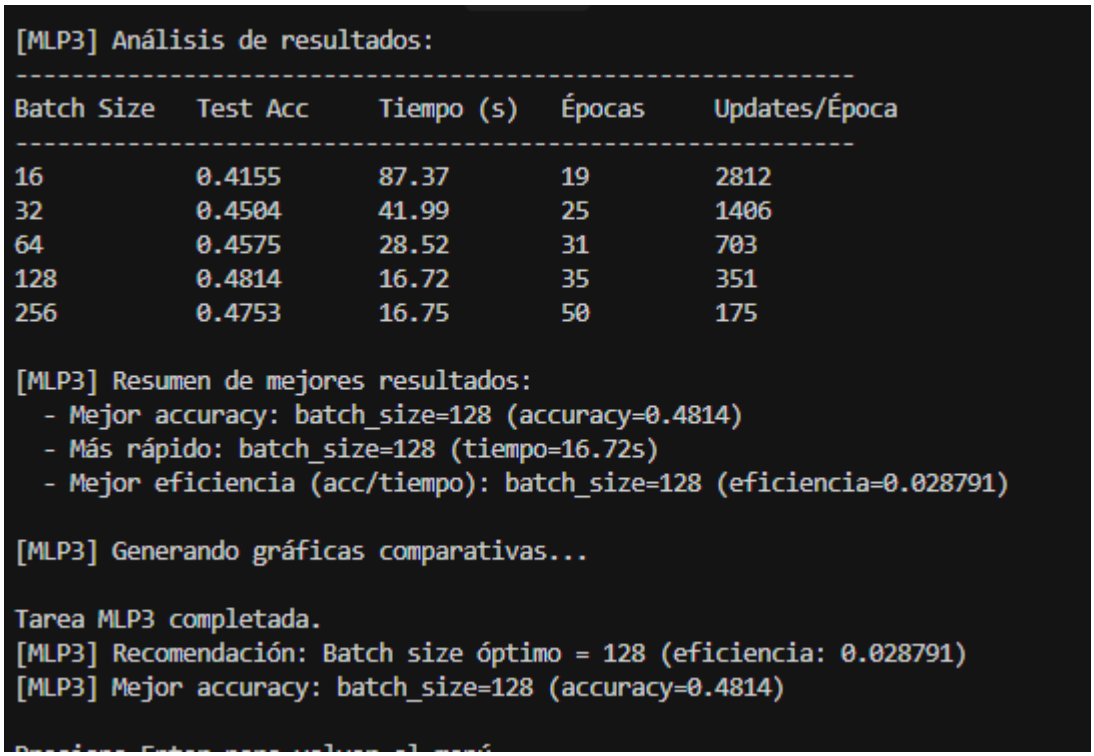


```
[MLP3] Análisis de resultados:
------------------------------------------------------------
Batch Size   Test Acc     Tiempo (s)   Épocas     Updates/Época
------------------------------------------------------------
16           0.4151       54.73        16         2812
32           0.4244       21.03        15         1406
64           0.4546       20.51        25         703
128          0.4611       8.23         19         351
256          0.4619       9.39         31         175

[MLP3] Resumen de mejores resultados:
  - Mejor accuracy: batch_size=256 (accuracy=0.4619)
  - Más rápido: batch_size=128 (tiempo=8.23s)
  - Mejor eficiencia (acc/tiempo): batch_size=128 (eficiencia=0.056060)

[MLP3] Generando gráficas comparativas...

Tarea MLP3 completada.
[MLP3] Recomendación: Batch size óptimo = 128 (eficiencia: 0.056060)
[MLP3] Mejor accuracy: batch_size=256 (accuracy=0.4619)

Presiona Enter para volver al menú...
```

Tercera ejecución:







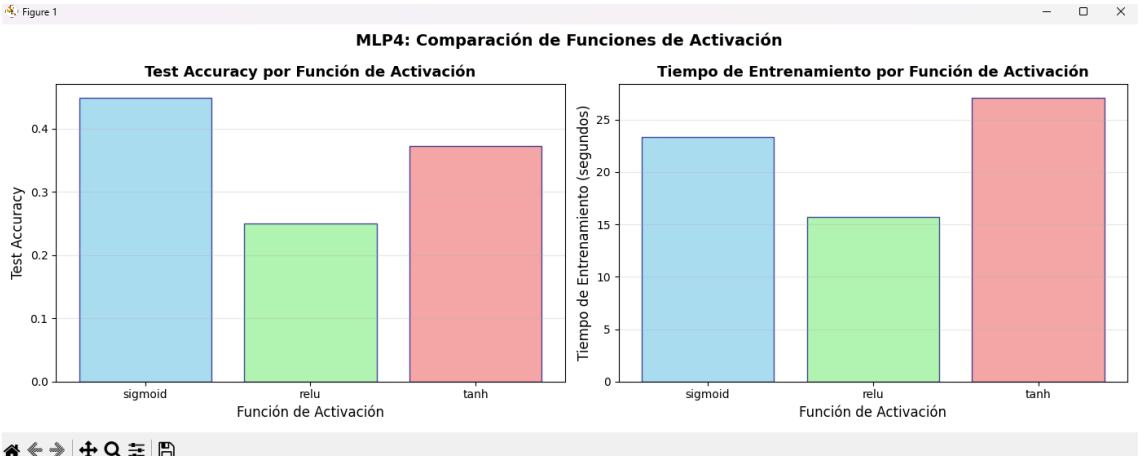
```
[MLP3] Análisis de resultados:
------------------------------------------------------------
Batch Size   Test Acc     Tiempo (s)   Épocas     Updates/Época
------------------------------------------------------------
16           0.4155       87.37        19         2812
32           0.4504       41.99        25         1406
64           0.4575       28.52        31         703
128          0.4814       16.72        35         351
256          0.4753       16.75        50         175

[MLP3] Resumen de mejores resultados:
  - Mejor accuracy: batch_size=128 (accuracy=0.4814)
  - Más rápido: batch_size=128 (tiempo=16.72s)
  - Mejor eficiencia (acc/tiempo): batch_size=128 (eficiencia=0.028791)

[MLP3] Generando gráficas comparativas...

Tarea MLP3 completada.
[MLP3] Recomendación: Batch size óptimo = 128 (eficiencia: 0.028791)
[MLP3] Mejor accuracy: batch_size=128 (accuracy=0.4814)
```

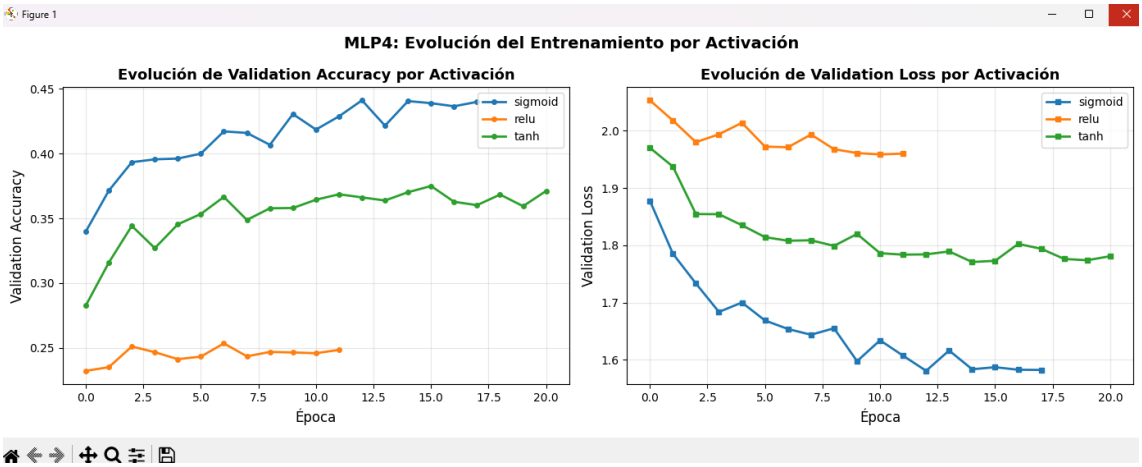
He realizado 3 ejecuciones distintas para analizar los datos. Se puede apreciar como los tamaños pequeños son los más ineficientes, suele deberse a que no aprovechan la paralelización de la GPU y CPU, ya que hay una gran diferencia de acierto con el resto. Una cosa interesante que ha sucedido durante estas ejecuciones es que en la primera de ellas el tamaño 256 ha sido más eficiente que el resto con una tasa considerable, mientras que en la segunda podemos considerar mejor a 128 ya que no hay mucha diferencia en acierto con 256, se podría considerar un empate, en cambio, en la tercera ejecución el tamaño 128 me ha dado mejores resultados que 256, y este se debe a la condición de aleatoriedad que tiene la inicialización del entrenamiento.

En conclusión, la selección de 128 como el valor para batch_size es la más conveniente para las tareas, ya que es más o menos igual de óptima que 256, pero realiza el doble de actualizaciones que esta última.

MLP4: Para este apartado se pide que experimente con el impacto que tienen los distintos tipos de activación que podemos aplicar en el aprendizaje, para ello he utilizado las funciones de activación **sigmoid, relu** y **tanh**.

Sigmoid comprime los valores entre un rango de 0 y 1; relu tiene un rango entre 0 e infinito, donde la función lineal es para los valores positivos y el 0 representa los negativos, el único inconveniente que tiene es que puede sufrir el problema de las neuronas muertas; tanh comprende un rango de -1 a 1 y esta centrada en cero lo cual facilita el entrenamiento.





Se puede apreciar en las gráficas obtenidas tras la ejecución que relu es la peor de todas cuando en un principio no debería ser así. Esto sucede porque sufre el problema de neuronas muertas mencionado antes, es decir, estas neuronas estaban en la zona negativa lo que provoca que sus salidas sean anuladas y colapsando así el aprendizaje.

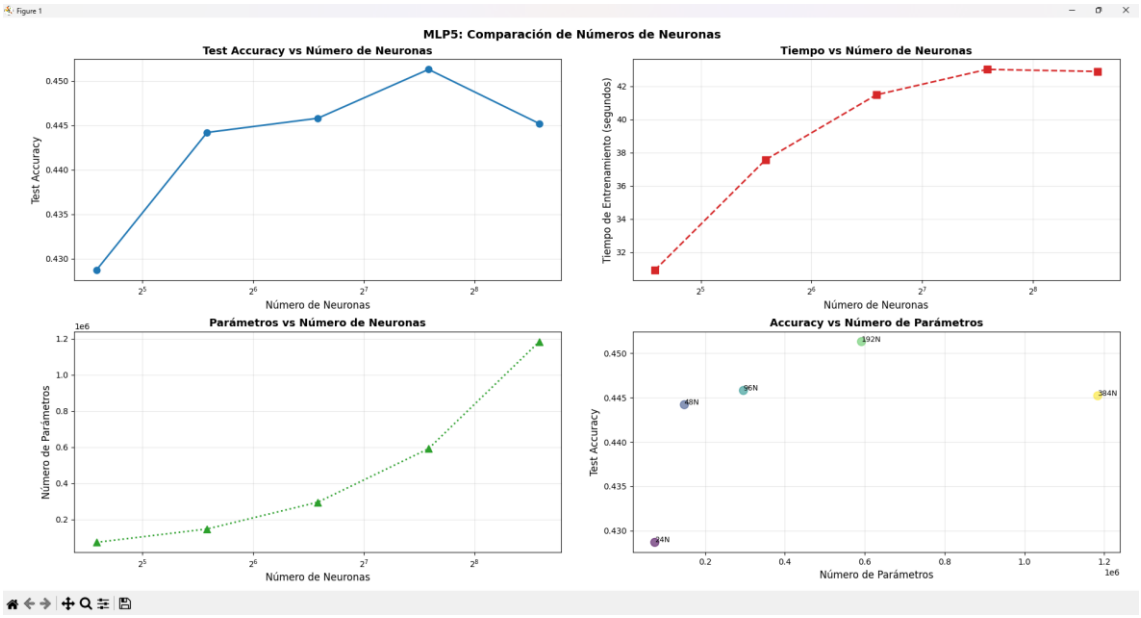
Como la red que estoy usando no es muy profunda se puede concluir que sigmoid es la más optima ya que no se ve muy afectado por el problema del desvanecimiento de gradiente, ya que solo uso una capa, y me proporciona la mayor tasa de acierto en menos tiempo que tanh.

En conclusión, relu a pesar de ser la más popular, recomendada y la que supuestamente mejores resultados debía de proporcionar ha sido la que peor tasa de acierto, tanh es eficiente y muestra buenas tasas y en un buen tiempo, pero sigmoid me proporciona aun mejor tasa y en menos tiempo, por lo tanto, selecciono esta como la óptima.

MLP5: En este apartado se me pide que experimente con la anchura de la rede neuronal, es decir, con el número de neuronas que componen mi red, las cuales permiten que mi red sea capaz de procesar las entradas y generar una salida.

Podemos encontrar varias situaciones a la hora de experimentar con el número de neuronas, los dos más comunes son **Underfitting**, que suele producirse cuando el número de neuronas es muy bajo, y **Overfitting**, que se produce cuando tenemos una cantidad excesiva de neuronas,

Para este caso he usado los valores 24, 48, 96, 192, 384 con la función de activación sigmoid y un batch_size de 128.




```
------------------------------------------------------------------------
Neuronas     Test Acc     Parámetros     Tiempo (s)   Épocas
------------------------------------------------------------------------
24           0.4287       74002          30.91        28
48           0.4442       147994         37.56        29
96           0.4458       295978         41.47        24
192          0.4513       591946         43.01        24
384          0.4452       1183882        42.89        16

[MLP5] Resumen de mejores resultados:
  - Mejor accuracy: 192 neuronas (accuracy=0.4513, params=591946)
  - Mejor eficiencia (acc/1K params): 24 neuronas (eficiencia=0.005793)

[MLP5] Generando gráficas comparativas...

Tarea MLP5 completada.
[MLP5] Mejor número de neuronas: 192 (accuracy=0.4513)
[MLP5] Mejor eficiencia: 24 neuronas

Presiona Enter para volver al menú...
```

Tras la ejecución se puede observar que hay una mejora lineal entre 24 y 96 neuronas donde con 24 neuronas se aprecio underfitting con un acierto del 42.9%.

El pico de rendimiento lo obtengo cuando se aplican 192 neuronas donde mi modelo alcanza la máxima tasa de acierto, un 45.1%, en cambio se puede apreciar como con 384 neuronas mi modelo sufre y la tasa de acierto cae a pesar de que el número de parámetros se eleva bastante.

En conclusión, usare 192 neuronas como valor óptimo ya que me ofrece la mayor tasa de acierto y su coste computacional respecto a 96 neuronas me merece la pena por la mejora de acierto.

MLP6:

MLP7:

Bibliografia:

MLP1:

- https://keras.io/api/layers/activations/

- https://keras.io/guides/sequential_model/

MLP2:

- https://keras.io/api/callbacks/early_stopping/

- https://matplotlib.org/stable/plot_types/basic/plot.html

MLP3:

- https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network

- https://www.tensorflow.org/guide/data_performance?hl=es-419#vectorizing_mapping

MLP4:

MLP5:

MLP6:

MLP7: