

# **Final Sistemas Operativos**

## **Tensorflow 2.1: comparación entre tamaños de batch y tipos de ejecución**

**Luciano Tangorra**

**DNI:** 40.455.501

**LU:** 248533

**E-Mail:** lucianotangorra@gmail.com

Link a mi wandb, donde figuran los datos de interes de las pruebas realizadas:  
<https://app.wandb.ai/lucholomas>

## Introducción

Este informe presenta un análisis acerca del uso de recursos en diferentes tamaños de batch y diferentes tipos de ejecución (imperativa y mediante grafos) sobre modelos de deep learning. Para la implementación de los modelos se utilizó TensorFlow 2.1, junto con su módulo keras, el cual facilita el prototipado.

En este trabajo se realizó una comparación sobre los mismos teniendo en cuenta el tiempo de ejecución, consumo de memoria del GPU, consumo de memoria RAM y pérdida (loss) con la que las redes neuronales clasifican los casos de prueba (test).

A su vez, se implementaron dos tipos de redes neuronales: CNN (Convolutional Neural Network) y RNN (Recurrent Neural Network) con el objetivo de probar las diferencias presentes en ambas cuando se aplican los cambios planteados anteriormente.

## Librería: Keras

Keras es una API de alto nivel, escrita en Python, útil para implementar redes neuronales velozmente y de este modo beneficiar el prototipado de soluciones para diversos problemas. Esta librería soporta tanto redes convolucionales como redes recurrentes (y combinaciones entre ambas), las cuales sirven para resolver gran cantidad de problemas, desde análisis de texto hasta reconocimiento de objetos en imágenes.

Keras es capaz de ejecutar su código utilizando TensorFlow, CNTK y Theano, entre otras librerías, aunque para esto se deben tener implementados los métodos utilizados por Keras. Mayormente se utiliza TensorFlow ya que CNTK y Theano no poseen soporte, por lo que no tendrán nuevas versiones.

La librería Keras no posee integración con TensorFlow 2.0, por lo que, para aprovechar el potencial que éste ofrece, se optó por utilizar el módulo keras implementado dentro de la librería Tensorflow. Por lo tanto, se podrá ver que, al importar la funcionalidad requerida, se utiliza *from tensorflow.keras import <funcionalidad a importar>*.

Hay que aclarar que la librería Keras no tiene relación con el módulo de Keras implementado en TensorFlow, es decir, no son compatibles entre sí y, además, se encuentran desarrolladas y poseen soporte de diferentes entidades.

En este informe se hará énfasis en el backend TensorFlow 2.0, el cual utilizaremos para realizar las pruebas.

## TensorFlow

TensorFlow es una librería de código abierto, creada por el equipo Google Brain, utilizada para realizar cálculos numéricos mediante tensores (muy similares a los arreglos numpy) y, más específicamente, para desarrollar modelos de machine learning. Su implementación se basa en los paradigmas de programación de flujo de datos y de programación diferenciable (auto diferenciación). Su núcleo se encuentra escrito en C++, ya que ofrece mayor performance que

Python. Esta librería es una de las principales en lo referente a diseño de modelos de machine learning en Python.

TensorFlow posee dos maneras de escribir y ejecutar el código: mediante forma imperativa, donde las operaciones son evaluadas inmediatamente, y mediante grafos, en donde se crea el mismo con las operaciones que luego serán ejecutadas.

La integración de Keras en el módulo de TensorFlow 2.0 usa una implementación imperativa por defecto y, con un pequeño cambio de código, se puede utilizar una implementación por grafos.

A continuación se resumen las dos formas de ejecución de código implementadas en TensorFlow<sup>[1, 2]</sup>:

- Ejecución mediante grafos: menos intuitiva a la hora de programar, aunque más eficiente ya que permite optimización gracias a poseer un grafo estático con las operaciones del modelo implementado. Los grafos son estructuras ampliamente conocidas y estudiadas, por lo que es posible analizarlas para realizar modificaciones sobre las mismas. Algunas de las posibles optimizaciones a aplicar: paralelismo (al conocer todo el grafo se puede identificar qué operaciones pueden ejecutarse en paralelo), optimización de los cálculos (eliminar nodos sin uso y operaciones redundantes), portabilidad (TensorFlow usa Protocol Buffers para serializar sus grafos y convertirlos en lenguajes-neutros) y ejecución distribuida. Una vez creado el grafo se podrá ejecutar el código para realizar cualquiera de las operaciones planteadas en el mismo. Al compilar un modelo implementado con Keras, éste utilizará el tipo de ejecución mediante grafos, reduciendo de este modo la complejidad temporal del programa.
- Ejecución imperativa (eagerly): es una manera más pythonica y orientada a objetos de implementar TensorFlow. Las operaciones se evalúan inmediatamente y retornan valores concretos, en lugar de construir un grafo estático que va a ser ejecutado luego. Esto facilita la comprensión del código ejecutado (por lo tanto menor tiempo de desarrollo) y ofrece una mejor forma de debuggear, ya que se puede ver al momento cuál es el valor obtenido por las operaciones. Es menos eficiente pero más amigable para el usuario e implementa grafos dinámicos, lo que es útil cuando no se sabe cuánto es el trabajo que se tiene que hacer. Por ejemplo, en NLP, se puede dar el caso de tener una sentencia con pocas palabras y otra que puede ser un párrafo completo. TensorFlow 2.0 posee este tipo de ejecución por defecto, ya que permite mayor legibilidad, un desarrollo de prototipos mucho más rápido y permite debuggear con mayor facilidad.

## Tensores

Los datos utilizados por TensorFlow suelen ser tensores, pero, ¿qué es un tensor? Un tensor representa un cálculo computacional parcialmente definido, el cual en algún momento producirá un valor (cuando se lo corra en una sesión, en caso de TensorFlow 1.x). Asimismo, se puede ver como un arreglo multidimensional (de 1 a n dimensiones), el cual puede ser una variable

(variables), una constante (constant) o un valor que próximamente será definido (placeholders). A partir de TensorFlow 2.0, los placeholders serán sustituidos por Variables, a las cuales se les permite no estar definidas con anterioridad. Se dice que un tensor es un cálculo parcialmente definido ya que no produce valor hasta ejecutarlo explícitamente, es decir, se crea un grafo computacional con los cálculos que deberán realizarse. Luego, al ejecutar una sesión, se alojan estos valores en el hardware designado, ya sea una o varias CPU, GPU o TPU, y se retorna el tensor producto de los cálculos realizados. En el modo de ejecución imperativo, los tensores continúan siendo un cálculo computacional parcialmente definido, el cual es ejecutado instantáneamente, permitiendo obtener el resultado de la operación en el momento en el que se ejecuta, sin necesidad de una sesión.

### Uso de recursos (GPU)

Por defecto TensorFlow bloquea, para el proceso a ejecutar, una gran cantidad de la memoria GPU de todas las placas GPU visibles (según `CUDA_VISIBLE_DEVICES`). Esto se lleva a cabo para obtener un uso más eficiente del GPU, evitando la fragmentación de memoria. Para evitar el bloqueo de la GPU hay dos alternativas: plantear un máximo de memoria que podrá utilizar TensorFlow (deseable si hay otros programas que deban usar GPU), o permitir el uso de una porción de la memoria y que el bloqueo de memoria máxima utilizable crezca sólo cuando el programa así lo necesita. Hay que tener en cuenta que una vez que se bloquea cierta cantidad de memoria, ésta no se liberará hasta no terminar el proceso, evitando la fragmentación de memoria lo más posible.

Para liberar memoria se utiliza el garbage collector de Python, por lo que se pueden hacer llamados a `gc.collect()` para pedir la limpieza de espacio que ya no se usa.

### Implementación mediante grafos

Sea el caso de una función del tipo  $f(a,b)=(a*b)/(a+b)$  (código 1), TensorFlow crea un grafo (figura 1) que posee nodos representando cada valor y operación. Cada uno de ellos puede poseer diversas partes:

- Variables: como las utilizadas en un programa común. Se pueden modificar en cualquier momento, pero deben ser inicializadas antes de correr el grafo en una sesión. En una red neuronal las variables son los pesos y los bias.
- Placeholders (obsoleto a partir de TF 2.0): nos permiten ingresar información al grafo sin necesidad de ser inicializadas previamente, definiendo su forma (shape) y tipo de dato. En una red neuronal las variables son los inputs y los labels.
- Constants: parámetros que no pueden ser modificados.
- Operations (ops): representan nodos en el grafo que realizan las operaciones entre tensores.

Una vez que el grafo es creado, puede ser ejecutado completamente o sólo un subgrafo a elección. Para esto se debe ingresar la información necesaria (input data), la cual “fluye” a través del grafo elegido. El grafo será optimizado antes de ser ejecutado, por lo que se

obtendrán incrementos en la velocidad de procesamiento. Una vez ejecutado, el grafo calculará automáticamente los valores de gradiente aplicando la regla de la cadena sucesivamente (auto diferenciación). De este modo el programador sólo debe escribir la fase de forward, ya que la de backprop se hace automáticamente.

Esta manera de programar no es intuitiva, y mucho menos en un lenguaje dinámico. Plantear una estructura estática se aleja de como fue escrito el resto del código (lenguaje dinámico con código estático). Además, también se pueden encontrar dificultades a la hora de probar el código ya que es muy difícil de debuggear. Por esto los ingenieros de Google diseñaron la herramienta TensorFlow Debugger, la cual es de utilidad a la hora de buscar errores en el código. Aún así trabajar con este modo de ejecución continua siendo poco intuitivo.

Siempre que se pueda se deberá utilizar este modo de ejecución, ya que permite mayor optimización en los cálculos requeridos.

TensorFlow 2.0 utiliza el modo de ejecución imperativa por defecto, por lo tanto se deberá desactivar manualmente para aprovechar la ejecución mediante grafos. Hay dos maneras: desactivarlo para todo el código, colocando una instrucción al principio del programa o, según sugieren los creadores, utilizar el decorador `@tf.function` en las funciones Python creadas. Esto último mantiene un grafo con las instrucciones dentro del método invocado y de los métodos que éste, a su vez, invoque.

```
1 import tensorflow as tf
2
3 with tf.Session() as sess:
4     # Phase 1: constructing the graph
5     a = tf.constant(15, name="a")
6     b = tf.constant(5, name="b")
7     prod = tf.multiply(a, b, name="Multiply")
8     sum = tf.add(a, b, name="Add")
9     res = tf.divide(prod, sum, name="Divide")
10
11     # Phase 2: running the session
12     out = sess.run(res)
13     print(out)
```

Código 1. Código TensorFlow 1.x para generar grafo de  $f(a,b)=(a*b)/(a+b)$ .

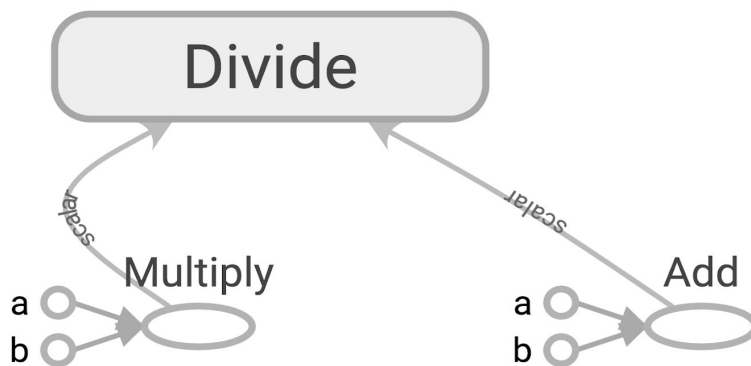


Figura 1. Representación de grafo generado por TensorFlow para la operación  $f(a,b)=(a*b)/(a+b)$ .

### Implementación de forma imperativa (eagerly)

A diferencia de la implementación por grafos, la implementación imperativa evalúa las operaciones inmediatamente, sin necesidad de construir un grafo: las operaciones retornan valores concretos en vez de construir un grafo computacional que luego será ejecutado. Esto permite escribir código de manera más “pythonica”, es decir, del mismo modo que se escribiría cualquier programa en python y, también, permite la utilización de las herramientas estándar del lenguaje para debuggear, utilizándolas directamente sobre las operaciones.

Las operaciones, constantes y variables siguen presentes en este modo de ejecución, pero en vez de crearse como referencias a los futuros valores que tomarán (nodos de grafo), se crearán y se evaluarán al ser ejecutadas, sin necesidad de una sesión. De este modo se facilita la escritura y comprensión del código.

El principal problema que se nos viene a la mente es: ¿cómo se calculan los gradientes? Al no tener creado un grafo por el cual “fluyen” los datos y por donde se calculan automáticamente los gradientes, no podemos calcularlos fácilmente. Para resolver esto, a los ingenieros de Google se les ocurrió implementar el concepto de Gradient Tape.

La “cinta” (tape) guarda todas las operaciones ejecutadas en la etapa de forward una vez ejecutadas, para luego crear un grafo (se crea en ejecución, no se compila como en el modo de ejecución por grafos). Éste será recorrido de atrás hacia adelante para aplicar diferenciación automática. Esto se realizará cada vez que se necesite (n veces en un loop, por ejemplo).

## Comparación entre los dos tipos de ejecución

|                    | Ejecución mediante grafos  | Ejecución imperativa  |
|--------------------|--|---|
| <b>Ventajas</b>    | <ul style="list-style-type: none"><li>- Más eficiente, ya que se optimiza el grafo de ejecución antes de ser ejecutado.</li></ul>                      | <ul style="list-style-type: none"><li>- Código más simple, escrito de forma más “pythonica”.</li><li>- Mayor facilidad para debuggear, ya que las operaciones se evalúan en el momento de ser ejecutadas.</li></ul> |
| <b>Desventajas</b> | <ul style="list-style-type: none"><li>- Código más difícil de escribir y de debuggear al tener que crear un grafo estático y luego correrlo.</li></ul> | <ul style="list-style-type: none"><li>- Menos eficiente ya que no se optimiza el código antes de ser ejecutado.</li></ul>   |

Aunque siempre se querrá un código más eficiente, hay que tener en cuenta el tiempo que lleva construir ese código. Utilizar el modo de ejecución mediante grafos permite obtener un código más veloz pero llevará más tiempo de desarrollo y, durante este proceso, será trabajoso hallar y resolver los errores por las dificultades a la hora de realizar debug.

Se recomienda utilizar tensorflow.keras puro para modelos simples, y, de ser necesario modificar el modelo por alguna funcionalidad no existente en esa librería, se sugiere emplear el modo de ejecución imperativa (por defecto en tensorflow 2.x).

Una de las nuevas funcionalidades aplicadas en tensorflow 2.x es la posibilidad de cambiar el modo de ejecución simplemente decorando las funciones con `@tf.function`. Esto cambiará el tipo de ejecución a grafos, optimizando el código previamente testado con la ejecución imperativa, permitiendo obtener los beneficios de ambos tipos sin necesidad de realizar grandes modificaciones en el código.

## Batch

Batch es un conjunto de datos agrupados que serán utilizados, en este caso, para entrenar una red neuronal. El tamaño de batch define la cantidad de datos que van a ser propagados por la red.

Por ejemplo, si se tienen 1050 datos de entrenamiento y un tamaño de batch (batch\_size) de 100, se tomarán los primeros 100 datos del conjunto de entrenamiento y se entrenará a la red, realizando la etapa de forward, calculando los gradientes y modificando los pesos. Luego se tomarán otros 100 datos diferentes del mismo conjunto de entrenamiento y se entrenará nuevamente a la red. Cuando queden los últimos 50 datos, se los utilizarán para entrenar a la red, ignorando el tamaño de batch y utilizando un menor número de datos.

Según el tamaño de batch elegido, la etapa de aprendizaje de parámetros se definirá como batch gradient descent, stochastic gradient descent o mini-batch gradient descent. A continuación se plantean las ventajas y desventajas de cada uno.

En el batch gradient descent (tamaño del batch = tamaño del conjunto de entrenamiento), se computa el gradiente sobre todo el dataset, por lo que la trayectoria que se toma para llegar al mínimo global está dirigida por todos los datos. Realizar este tipo de entrenamiento conlleva mucha memoria, ya que se deberá almacenar todo el batch en cuestión. Normalmente no se suele usar esto ya que no se poseen los recursos para alojar todo el dataset en memoria, aunque es recomendable.

En el stochastic gradient descent (descenso por el gradiente estocástico, tamaño de batch = 1), al contrario, se modifican los parámetros por cada instancia del conjunto de entrenamiento. Al utilizar un único dato, la muestra posee mucho ruido y puede alejarse de los óptimos. La principal desventaja es su ineficiencia (ya que no se beneficia de la vectorización) y requerirá mucho tiempo encontrar una buena solución.

Por último, en mini-batch gradient descent ( $1 < \text{tamaño de batch} < \text{tamaño del conjunto de entrenamiento}$ ) se computa el gradiente sobre un subconjunto de datos de entrenamiento, por lo que se posee ruido, evitando caer en malos óptimos locales, aunque no es suficiente para evitar la rápida convergencia de la red. Al utilizar un subconjunto de datos, se requerirá menos memoria que en batch gradient descent, ya que se almacenará sólo el batch a utilizar. La principal desventaja que se tiene es que, mientras más pequeño sea el tamaño de batch, menos preciso es el gradiente estimado. Se recomienda utilizar números potencia de 2 por cómo funciona el hardware existente y, además, utilizar el tamaño máximo que soporte la GPU, ya que esto beneficiará enormemente el uso de recursos y la velocidad del modelo.

En la figura 2 se podrá observar una comparación entre los tres tipos de batch.

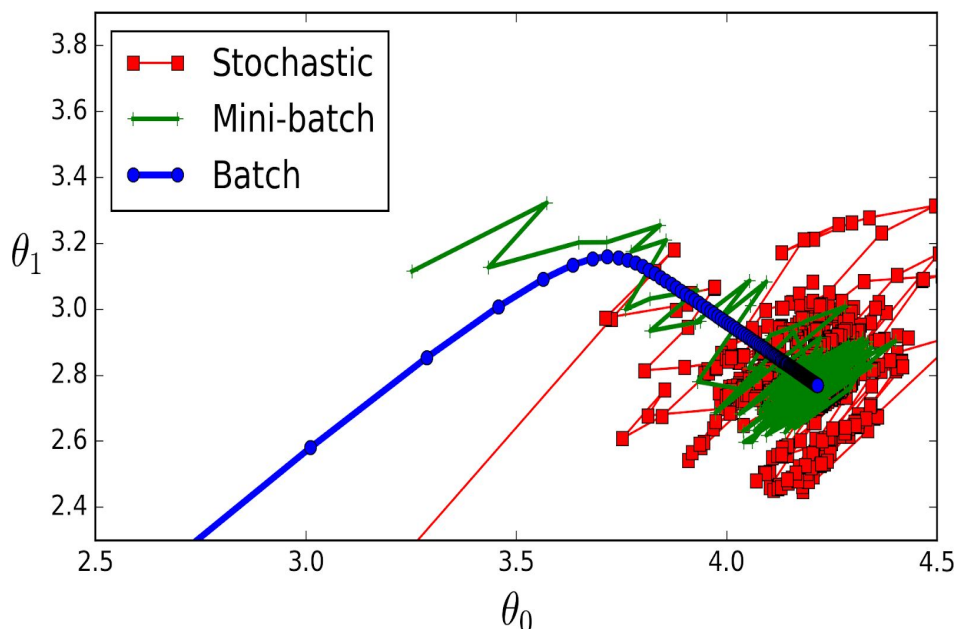


Figura 2. Comparación entre SGD, M-B GD, BGD para llegar al óptimo.



Se puede observar que si el tamaño de batch es grande nos asegura llegar al óptimo global, aunque de manera lenta, ya que calcular el gradiente sobre todo el conjunto de datos conlleva mucho tiempo. En cambio, utilizar tamaños de batch reducidos demuestra llegar a una buena solución rápidamente. Esto se puede explicar intuitivamente partiendo de que con tamaño de batch reducido se permite al modelo aprender antes de ver todo el conjunto de datos de entrenamiento. La desventaja de usar tamaños pequeños en un modelo es que no garantizan obtener el mínimo global.

Hay que tener en cuenta que lo que se busca es minimizar la función de pérdida, por lo que un batch más grande asegura descender por el gradiente verdadero en lugar de estimarlo. El problema es que favorece el overfitting y perjudica la generalización del modelo sobre casos de prueba y reales. Por ende, hay que encontrar un tamaño de batch que aproveche la velocidad de cómputo de la GPU y que permita al modelo generalizar. Este tamaño se encuentra a prueba y error. No hay una fórmula predefinida que nos pueda ayudar a definir este valor.

## Optimizadores




Los algoritmos de optimización nos ayudan minimizar (o maximizar) una función objetivo. En el caso de deep learning la función objetivo es la función de error ( $E(x)$ ). Esta función depende de los parámetros (que se pueden aprender) que tiene el modelo (pesos y biases), los cuales se utilizan para computar los valores objetivo ( $Y$ ) según los valores de entrada ( $X$ ).

Se adjunta un link a la documentación de Keras, el cual explica los optimizadores implementados.

Link: <https://keras.io/optimizers/>

Se probaron todos los optimizadores presentes en Keras y se llegó a la conclusión de que no hay cambios notables en el uso de recursos. Esto se podría deducir ya que el optimizador es simplemente un pequeño cálculo matemático a realizar, y ningún optimizador realiza cálculos costosos. Los optimizadores que más cálculos realizan podrán incrementar el tiempo total en una pequeña fracción, lo cual no justifica la elección de uno u otro. Al final, se optó por utilizar SGD por mini-batch como optimizador de los modelos.

A continuación se adjuntará un link con pruebas realizadas en una red convolucional.

Para filtrar sobre todas las posibles run existentes se podrá pulsar el siguiente ícono →    y, en la primera opción que aparece, elegir tag y filtrar por los valores deseados (ya predeterminados). Luego se podrá seleccionar cualquiera de las run y elegir la opción llamada System. Ahí se observarán los recursos utilizados por cada ejecución.

Link: <https://app.wandb.ai/lucholomas/cnn-so-tensorflow-gpu?workspace=user-lucholomas>

## Hardware utilizado

Se utilizó la GPU “Titan Xp” para este trabajo. Las especificaciones pueden encontrarse a continuación: <https://www.nvidia.com/en-us/titan/titan-xp/>. Las únicas placas utilizables en TensorFlow son las fabricadas por NVidia, ya que sólo tiene soporte para CUDA (API diseñada por NVidia para utilizar sus GPUs para procesamiento de propósito general). Se intentó realizar una comparación entre GPU vs CPU, pero la velocidad de procesamiento de CPU es mucho menor que la GPU utilizada, tardando días en completar una sola ejecución, por lo que se descartó esta idea.

## Modelos probados

Se implementaron, en Python 3.6.8 con TensorFlow 2.1, dos modelos: uno aplicando redes convolucionales y otro con redes recurrentes. Cada uno fue probado con diferentes tamaños de batch, desde 1 hasta 4096 (en potencias de 2). Estos modelos fueron planteados con el fin de poder comparar la utilización de recursos (GPU) y su comportamiento sobre la pérdida obtenida. Además de ejecutar diferentes pruebas con varios tamaños de batch, se utilizaron los dos tipos de ejecución posibles en TensorFlow.

| Red convolucional  |              |                   | Red recurrente   |              |                 |
|--------------------|--------------|-------------------|------------------|--------------|-----------------|
| Tipo               | # Parametros | Tamaño salida     | Tipo             | # Parametros | Tamaño salida   |
| InputLayer         | 0            | ,50,50,3          | InputLayer       | 0            | None, None      |
| Conv2D             | 4200         | None, 48, 48, 150 | Embedding        | 5208448      | None, None, 128 |
| BatchNormalization | 600          | None, 48, 48, 150 | LSTM             | 131584       | None, 128       |
| MaxPooling2D       | 0            | None, 24, 24, 150 | Dense            | 12900        | None, 100       |
| Conv2D             | 1125300      | None, 20, 20, 300 | Dense            | 303          | None, 3         |
| BatchNormalization | 1200         | None, 20, 20, 300 |                  |              |                 |
| MaxPooling2D       | 0            | None, 10, 10, 300 |                  |              |                 |
| Dropout            | 0            | None, 10, 10, 300 |                  |              |                 |
| Conv2D             | 540200       | None, 8, 8, 200   |                  |              |                 |
| MaxPooling2D       | 0            | None, 4, 4, 200   |                  |              |                 |
| Flatten            | 0            | None, 3200        |                  |              |                 |
| Dropout            | 0            | None, 3200        |                  |              |                 |
| Dense              | 3201         | None, 1           |                  |              |                 |
| Total params       | 1,674,701    |                   | Total params     | 5,353,235    |                 |
| Trainable params   | 1,673,801    |                   | Trainable params | 5,353,235    |                 |

## Dockerización de los modelos

Los modelos fueron desarrollados en mi computadora personal, la cual no posee GPU. Por lo tanto, fue necesario utilizar una GPU en una computadora remota para las pruebas. Para evitar problemas con librerías no instaladas o versiones antiguas, se optó por la implementación de Docker, ya que permite aislar la ejecución de nuestro software mediante containers. Cada

container posee su propio software (un container para cada modelo), librerías y archivos de configuración.

Se reducirá la cantidad de comandos a escribir en el archivo de Docker utilizando alguna imagen que posea los requerimientos deseados. TensorFlow posee varias imágenes subidas a Docker Hub, por lo que ya nos aseguramos la compatibilidad de versiones y paquetes instalados. Se encontró una que contiene tensorflow-gpu, Python3, y toda compatibilidad requerida, junto con Ubuntu 18.04, por lo que sólo hizo falta instalar las librerías ajenas a TensorFlow que utilizamos en nuestro código e incluir los archivos de entrenamiento y test.

## **Problemas encontrados**

El enunciado original del trabajo era comparar los diferentes backends implementados en la librería Keras. Se implementaron los modelos correspondientes, pero cuando se intentaron dockerizar no fue posible por problemas de compatibilidad. Por ejemplo, Theano tenía problemas con pygpu, una librería hecha para utilizar la GPU. Esta librería fue creada para Theano. Como Theano no posee actualmente más soporte, pygpu tampoco, por lo que hardware y software nuevos generaban problemas. Al ser imposible correr estos modelos en la GPU, las comparaciones no tenían sentido. La ejecución en un CPU (de mi máquina local) es un caso poco viable (y demasiado lento, tardando días con los modelos planteados) y ninguno de estos frameworks se encuentran optimizados para su uso en CPU. En deep learning se realizan muchos cálculos matriciales, por lo que utilizar hardware especializado acelera obtener resultados, por ende se acelera el prototipado de modelos que ofrezcan buenos resultados.

## **Comparación de los modelos planteados**

Se realizará una comparación sobre cada modelo utilizando diferentes tamaños de batch, en los cuales se podrán observar las diferencias de uso de recursos, tiempo de entrenamiento y valores de pérdida obtenidos.

La comparación del hardware utilizado se realizará sobre porcentaje de GPU empleado y porcentaje de memoria del GPU accedida, ya que el resto de los valores permanecen iguales (CPU, RAM, y disco utilizados).

## **Red convolucional**

### **Uso de GPU**

Las pruebas realizadas muestran que el uso de GPU se relaciona con el tamaño de batch elegido, donde tamaños más pequeños utilizan un menor porcentaje de GPU y tamaños más grandes utilizan más, hasta usarla casi completamente (94% de uso, teniendo reservado un 96% para TensorFlow). Además, se visualiza claramente la optimización sobre el uso de los

recursos que posee la ejecución por grafos, quedando muy por encima del modo ejecución imperativa.

El acceso a memoria de GPU incrementa según aumenta el uso de GPU, comenzando en un acceso de 10% para el modo de ejecución mediante grafos y, para la ejecución imperativa, tan solo un 2%. En valores más altos de uso de GPU encontramos el mismo nivel de acceso a memoria en ambos modos de ejecución.

La utilización de GPU depende, además del tamaño de batch, del tipo de ejecución empleada. En la ejecución del modelo mediante grafos se puede observar un uso del 47% con tamaño de batch 1, aprovechando en mayor medida que el modo imperativo, el cual, con el mismo tamaño de batch, utiliza tan solo un 13%.

Esta diferencia empieza a disminuir a medida que aumenta el tamaño de batch. Con un tamaño de batch 32 se utiliza un 77% del GPU en ejecución mediante grafos y un 42% en ejecución imperativa.

El máximo tamaño de batch probado fue de 1024 ya que la GPU utilizada no soporta batch de 2048 porque se queda sin memoria posible para aloarlos. Teniendo en cuenta este máximo, el uso de GPU es el mismo en ambos tipos de ejecución, aunque no así el tiempo de ejecución.

Se puede observar, en la figura 3, el uso de GPU para diferentes tamaños de batch, junto con una comparación entre ambos tipos de ejecución. El aumento del uso de los recursos se condice con lo explicado anteriormente, ya que mientras más grande es el tamaño de batch (mientras el GPU sea capaz de procesarlo) mayor es el paralelismo de procesamiento aplicado, aprovechando la vectorización. Esto tiene un límite. La transferencia de datos para el procesamiento puede resultar en un cuello de botella, donde en un tamaño de batch mayor se alcanzan tiempos mayores que en tamaños menores, donde se utiliza menos GPU. Esto se ve en la figura 4 y 5, donde a partir de tamaño 256 no se observan mejoras significativas en tiempo de ejecución.

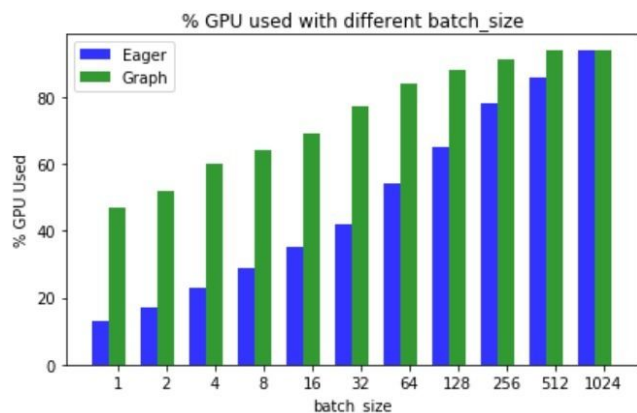


Figura 3. Comparativa en el uso de GPU entre diferentes tamaños de batch para ambos tipos de ejecución en una red convolucional.

## Tiempo de ejecución

El tiempo de ejecución fue contemplado como el tiempo total desde que inicia el entrenamiento hasta que finaliza. En cada ejecución se puede ver que procesar la primer epoch conlleva un leve incremento de tiempo con respecto a las siguientes. Esto se debe a que en las últimas versiones se postergaron etapas de construcción de grafo y de procesamiento de estructuras utilizadas durante el entrenamiento para ser computadas durante la primer epoch. De esta manera se evita realizar cálculos innecesarios.

En el modo de ejecución mediante grafos el tiempo de ejecución es máximo con el tamaño de batch 1, llegando a tardar 28 minutos. Utilizando modo de ejecución imperativa, se observa un tiempo de ejecución de dos horas y media, 5 veces más que el modo de ejecución por grafo. Como se comentó anteriormente, esto se debe al poco aprovechamiento de los recursos del sistema y, en caso de la ejecución imperativa, a las pocas o nulas optimizaciones que se pueden realizar sobre el código. En la figura 4 se visualiza la diferencia en tiempos de ejecución, contemplando la gran desigualdad en performance ofrecidas por ambos tipos de ejecución.

En la figura 5 se observan las diferencias más sutiles, como por ejemplo, si se poseen tamaños de batch más grandes, los tiempos de ejecución son menores que en tamaños de batch pequeños. Se evidencia que el uso de tamaño de batch grande (1024) puede empeorar el tiempo de entrenamiento ya que no se podrán procesar los datos al momento requerido y se tendrá un overhead al realizar mayor cantidad de transferencias. Esto se debe principalmente al hardware utilizado. Si se contase con un hardware más potente esto no debería suceder, obteniendo de este modo un tiempo menor al utilizar batch de 1024 para este modelo.

Está claro que el tiempo de ejecución imperativa es menos eficiente que el modo de ejecución mediante grafos. En tamaños de batch grandes esta diferencia se ve reducida. Si tomamos en cuenta que el modelo probado es simple y posee pocos parámetros a entrenar, se puede deducir que en modelos grandes esta diferencia será más notoria, concluyendo en la utilización del modo de ejecución mediante grafos de manera casi obligatoria.

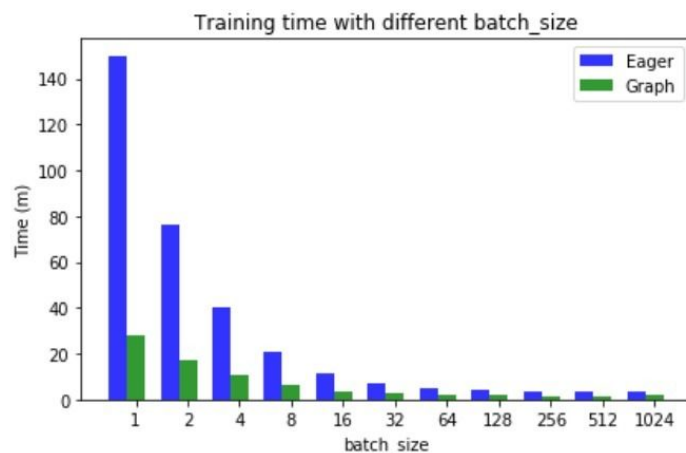


Figura 4. Comparativa en tiempo de ejecución entre diferentes tamaños de batch para ambos tipos de ejecución en una red convolucional.

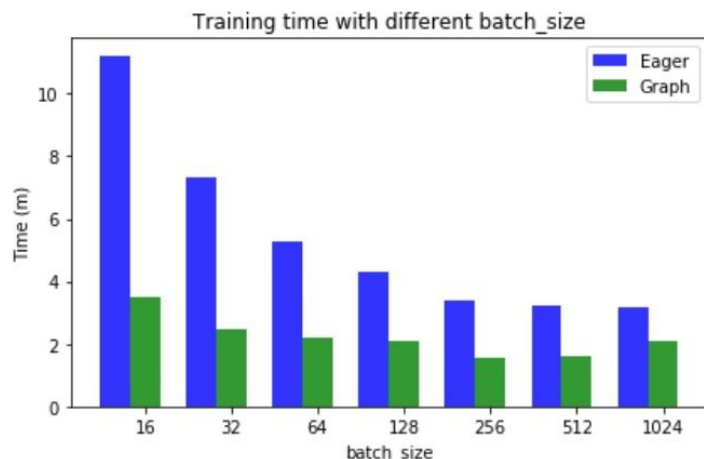


Figura 5. Comparativa en tiempo de ejecución entre diferentes tamaños de batch (acotados) para ambos tipos de ejecución en una red convolucional.

### Pérdida de los modelos (loss) y precisión (accuracy)

Se realizó una comparación entre las pérdidas (loss) de los modelos, así como también de su precisión (accuracy), para contemplar su entrenamiento según diferentes tamaños de batch. Se buscó encontrar que tan bien entrena cada modelo sabiendo cómo se comportan estas dos métricas en conjunto:

- Precisión: aciertos del modelo sobre los datos.
- Pérdida: distancia entre los valores reales del problema y los que el modelo predijo.
- Precisión baja y pérdida grande: se realizaron grandes errores en muchos datos.
- Precisión baja y pérdida pequeña: se realizaron pocos errores en muchos datos.
- Precisión alta y pérdida grande: se realizaron grandes errores en pocos datos.
- Precisión alta y pérdida pequeña: se realizaron pequeños errores en pocos datos (mejor caso).

¿Qué significa realizar grandes errores? Sea el caso de tener dos clases, A y B, si nuestro modelo predice que para una entrada su clase correspondiente es A con un 90% de seguridad y, realmente la clase correcta era la B, el modelo tuvo mucho errores sobre esa predicción. Tener un pequeño error significa que, en vez de predecir con un 90% de seguridad, prediga con un 55% de seguridad, por lo que el modelo se equivocaría igualmente, pero encontrándose más cerca del verdadero valor.

También se tendrá en cuenta cuando se produce overfitting, underfitting y cuando el entrenamiento es el correcto. Para poder distinguir esto, se deberá tener en cuenta lo siguiente:

- Overfitting: pérdida de entrenamiento << pérdida de validación.
- Underfitting: pérdida de entrenamiento >> pérdida de validación.
- Entrenamiento correcto: pérdida de entrenamiento  $\approx$  pérdida de validación.

Es indistinto qué tipo de ejecución se utiliza ya que ambos entrenan de igual modo. La diferencia es con la eficiencia que lo hacen.

Para observar todos los valores obtenidos, se podrá visitar el siguiente link: <https://app.wandb.ai/lucholomas/cnn-graph-so-tensorflow-gpu?workspace=user-lucholomas>

Para tamaños de batch pequeños (hasta 8), se podrá observar una precisión casi nula (0.5 en un clasificador binario) con una pérdida en entrenamiento muy baja y en valoración muy alta. Esto provoca que los modelos sean malos, ya que “memorizan” los datos de entrenamiento, evitando poder generalizar para los casos de prueba, por lo que para cada dato nuevo que ingrese, el modelo elige una clase al azar. Esto se debe a que los modelos dan pequeños “pasos” para aprender sobre los datos, utilizando un reducido subconjunto de los mismos. Esto trae como consecuencia la obtención de mucho ruido, complicando obtener el gradiente verdadero de la función. En las figuras 6, 7 y 8 se podrá observar la pérdida en entrenamiento, la pérdida en validación y la precisión para los datos de validación respectivamente.

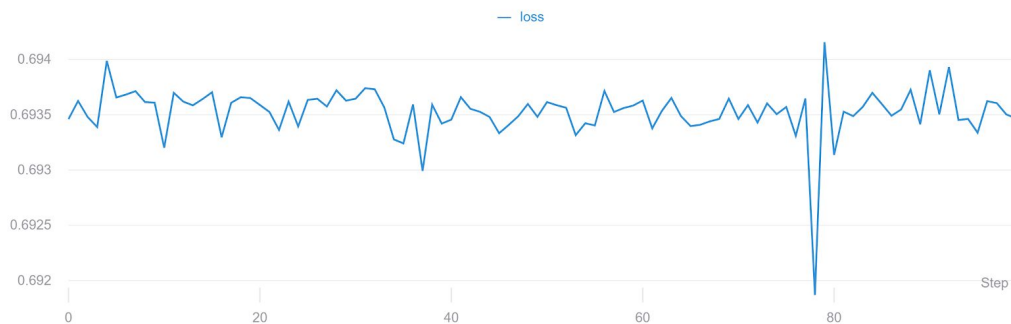


Figura 6. Pérdida obtenida en entrenamiento de la red convolucional para un tamaño de batch de 2.

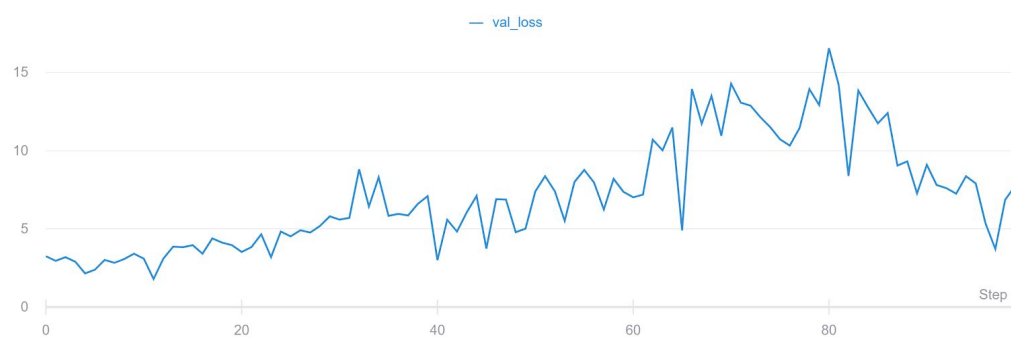


Figura 7. Pérdida obtenida en validación de la red convolucional para un tamaño de batch de 2.

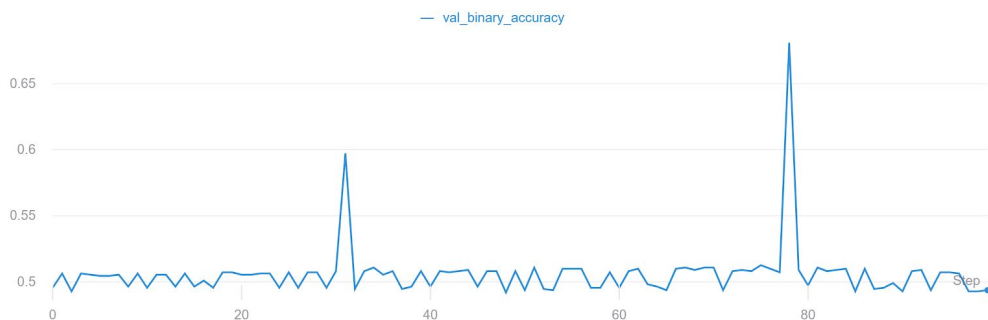


Figura 8. Precisión obtenida en validación de la red convolucional para un tamaño de batch de 2.

Para tamaño de batch 16 se podrá observar la menor diferencia conseguida en las pruebas entre la pérdida en entrenamiento y validación. Se puede asumir, al ser los dos valores parejos, que no se produjo overfitting ni underfitting. También se consiguió una precisión elevada, alcanzando un 78% al final del entrenamiento. Esto se debe a que al tener pocos datos y las imágenes ser parecidas entre sí, utilizar un batch pequeño (aunque no tanto) beneficia la velocidad de aprendizaje, ya que se toma una cantidad de datos por batch suficientemente grande como para tener un poco de ruido (evitando memorizar los datos de entrenamiento) pero no tanto como para alejarnos de un óptimo. Se observa en las figuras 9, 10 y 11 los valores de pérdida en entrenamiento, la pérdida en validación y la precisión para los datos de validación respectivamente.

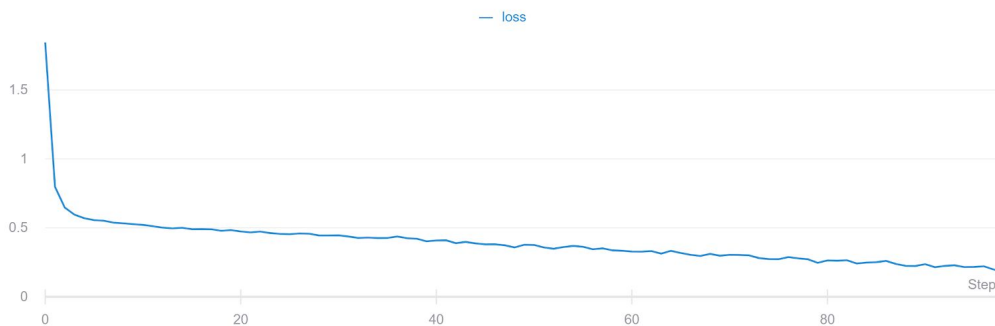


Figura 9. Pérdida obtenida en entrenamiento de la red convolucional para un tamaño de batch de 16.

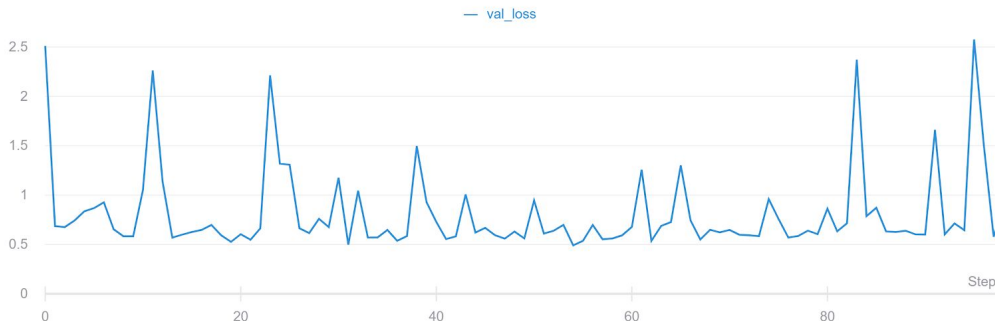


Figura 10. Pérdida obtenida en validación de la red convolucional para un tamaño de batch de 16.

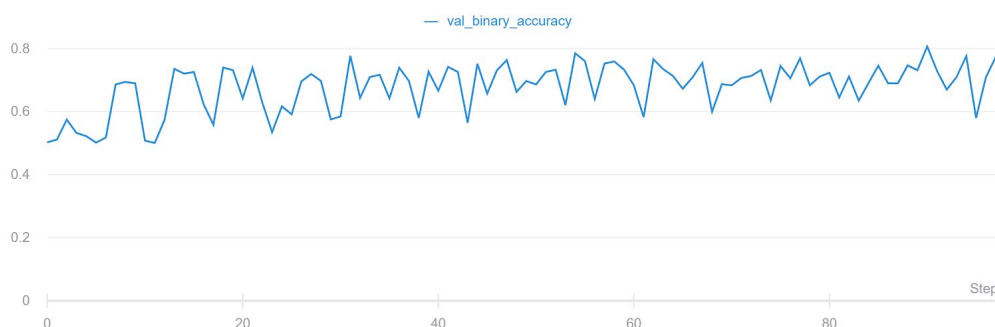


Figura 11. Precisión obtenida en validación de la red convolucional para un tamaño de batch de 16.



Para los tamaños más grandes se podrá observar un incremento del overfitting en los modelos, visualizado en las imágenes 12 y 13, las cuales muestran una clara diferencia entre las pérdidas producidas en entrenamiento y en validación. Al obtener un batch de tamaño grande, se consigue obtener valores de gradiente óptimos para los datos de entrenamiento, no así para los datos de validación, por lo que el modelo aprende fácilmente a memorizar las imágenes de entrada, en vez de generalizar para poder obtener buenas predicciones con todos los datos. Aunque la predicción obtenida sea mayormente correcta (como se observa en la figura 14) y, algunas veces supere a la obtenida con un tamaño de batch menor, hay que tener en cuenta que el modelo se equivoca no solo en la predicción final, sino en los porcentajes que le asigna a cada clase. En un escenario real, donde las imágenes pueden poseer alguna diferencia no presente en el dataset original, puede desencadenar en un modelo con una precisión menor a la demostrada con los datos de validación, es decir, un mal modelo con predicción baja.

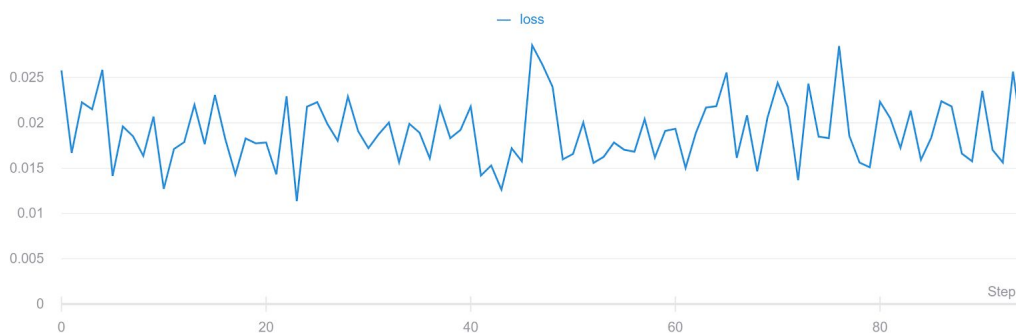


Figura 12. Pérdida obtenida en entrenamiento de la red convolucional para un tamaño de batch de 512.

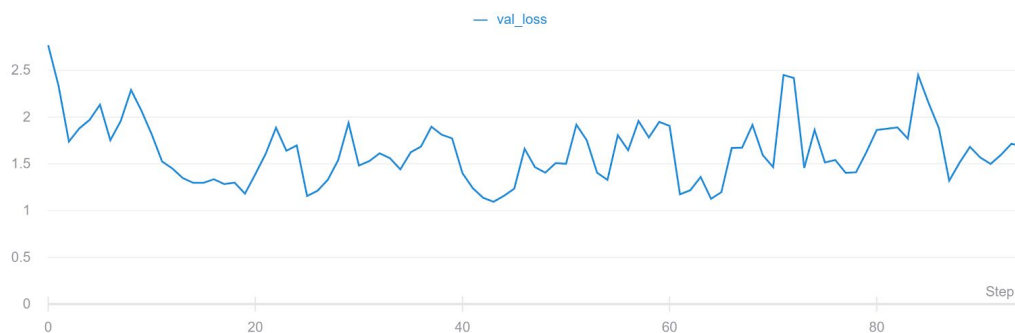


Figura 13. Pérdida obtenida en validación de la red convolucional para un tamaño de batch de 512.

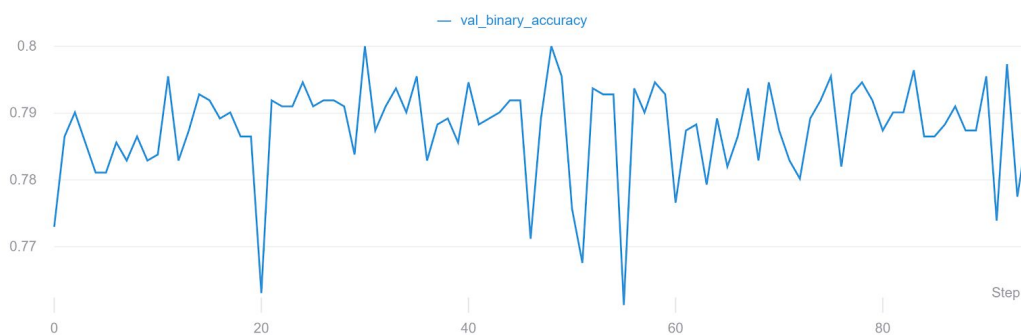


Figura 14. Precisión obtenida en validación de la red convolucional para un tamaño de batch de 512.

## Red recurrente

Las pruebas con redes recurrentes se realizarán próximamente, ya que la ejecución de tales redes con diferentes batches y tipos de ejecución llevará mucho tiempo, quizás semanas.

## Conclusiones

Puede verse que el modo de ejecución imperativa es más lento en ejecución y, aunque no fue demostrado en este trabajo, más rápido para prototipar. Gracias a la funcionalidad añadida en tensorflow 2.x, podremos beneficiarnos de ambos modos de ejecución, implementando y testeando el modelo de forma imperativa y, luego, modificar ligeramente el código para indicar que se desea utilizar la ejecución por grafo.

Además de beneficiarse del modo de ejecución, podremos ver beneficiado el modelo en precisión y reducción del valor de pérdida eligiendo correctamente el tamaño de batch. Aunque si bien no hay un valor correcto para cada dataset, siempre se recomienda utilizar tamaños potencia de 2 y, según el hardware que tengamos disponible, podremos elegir batches de tamaño 16, 32, 64 o 128. No recomendaría utilizar valores más bajos ni más altos ya que los primeros son altamente ineficientes y los segundos favorecen el overfitting del modelo. Con el tamaño correcto de batch podremos obtener el mayor beneficio de nuestra unidad de procesamiento gráfica y el suficiente ruido en cada batch, evitando de este modo el overfitting.

Nos podremos dar cuenta si el tamaño de batch que elegimos es el correcto comparando la pérdida de entrenamiento con la pérdida en la validación, como se explicó anteriormente en el apartado de Batch. Hay que tener en cuenta que puede darse el caso de que un tamaño que favorezca la performance, perjudique la precisión del modelo, aunque todo esto dependerá del modelo implementado y de los datos que poseamos.

## Bibliografía

Eager execution:

<https://www.tensorflow.org/guide/eager>

tf.Graph:

[https://www.tensorflow.org/api\\_docs/python/tf/Graph](https://www.tensorflow.org/api_docs/python/tf/Graph)

Why mini batch size is better than one single "batch" with all training data?:

<https://datascience.stackexchange.com/questions/16807/why-mini-batch-size-is-better-than-one-single-batch-with-all-training-data>

Understanding Mini-Batch Gradient Descent:

[https://www.youtube.com/watch?v=-\\_4Zi8fCZO4](https://www.youtube.com/watch?v=-_4Zi8fCZO4)

What is meant by "Batch" in machine learning?:

<https://www.quora.com/What-is-meant-by-Batch-in-machine-learning>

Effect of batch size on training dynamics:

<https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e>

What is batch size in neural network?:

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

Vectorization, Why and What?:

<https://www.quantifisolutions.com/vectorization-part-2-why-and-what>

Dockerize your Python Application:

<https://runnable.com/docker/python/dockerize-your-python-application>

What is Docker and How to Use it With Python (Tutorial):

<https://djangostars.com/blog/what-is-docker-and-how-to-use-it-with-python/>

What is the relationship between the accuracy and the loss in deep learning?:

<https://datascience.stackexchange.com/questions/42599/what-is-the-relationship-between-the-accuracy-and-the-loss-in-deep-learning>

How Does The Machine Learning Library TensorFlow Work?:

<https://afteracademy.com/blog/how-does-the-machine-learning-library-tensorflow-work>

TensorFlow 101: Understanding Tensors and Graphs to get you started in Deep Learning:

<https://www.analyticsvidhya.com/blog/2017/03/tensorflow-understanding-tensors-and-graphs/>

What is TensorFlow? The machine learning library explained:

<https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html>

How does Tensorflow `tf.train.Optimizer` compute gradients?:

<https://stats.stackexchange.com/questions/257746/how-does-tensorflow-tf-train-optimizer-compute-gradients>

How Tensorflow Calculates Gradients:

<https://www.linkedin.com/pulse/how-tensorflow-calculate-gradients-shamane-siriwardhana/>

Eagerly awaiting TensorFlow Eager Execution?:

<https://agi.io/2018/05/31/eagerly-awaiting-tensorflow-eager-execution/>

PyTorch – Tensors and Dynamic neural networks in Python:

<https://news.ycombinator.com/item?id=13428098>

Graph and Session:

<https://www.easy-tensorflow.com/tf-tutorials/basics/graph-and-session>

Understand TensorFlow by mimicking its API from scratch:

<https://medium.com/@d3lm/understand-tensorflow-by-mimicking-its-api-from-scratch-faa55787170d>

Understanding Dataflow graphs in TensorFlow:

<https://www.datasciencecentral.com/profiles/blogs/understanding-dataflow-graphs-in-tensorflow>

TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems:

<http://download.tensorflow.org/paper/whitepaper2015.pdf>

How to prevent tensorflow from allocating the totality of a GPU memory?:  
<https://stackoverflow.com/questions/34199233/how-to-prevent-tensorflow-from-allocating-the-totality-of-a-gpu-memory>

What's new in TensorFlow 2.0?:

<https://towardsdatascience.com/whats-new-in-tensorflow-2-0-ce75cdd1a4d1>

Everything you need to know about TensorFlow 2.0:

<https://towardsdatascience.com/everything-you-need-to-know-about-tensorflow-2-0-b0856960c074>

TensorFlow 1.0 vs 2.0, Part 1: Computational Graphs:

<https://medium.com/ai%C2%B3-theory-practice-business/tensorflow-1-0-vs-2-0-part-1-computational-graphs-4bb6e31c1a0f>

Migrate your TensorFlow 1 code to TensorFlow 2:

[https://www.tensorflow.org/guide/migrate#low-level\\_variables\\_operator\\_execution](https://www.tensorflow.org/guide/migrate#low-level_variables_operator_execution)

tf.keras uses Eager execution or Graph execution in tf 2.0 ?:

<https://github.com/tensorflow/tensorflow/issues/34771>

Hands-On Neural Networks with TensorFlow 2.0: Understand TensorFlow:

[https://books.google.com.ar/books?id=w5iwDwAAQBAJ&pg=PA127&lpg=PA127&dq=eager+execution+uses+graphs+on+each+loop&source=bl&ots=LBJKUNuHS1&sig=ACfU3U1INHt0GZkoNyoRCBOWmLOAeFzouw&hl=en&sa=X&ved=2ahUKEwigw\\_Sw2JnmAhUWFLkGHcMdD9UQ6AEwEXoECAoQAQ#v=onepage&q=eager%20execution%20uses%20graphs%20on%20each%20loop&f=false](https://books.google.com.ar/books?id=w5iwDwAAQBAJ&pg=PA127&lpg=PA127&dq=eager+execution+uses+graphs+on+each+loop&source=bl&ots=LBJKUNuHS1&sig=ACfU3U1INHt0GZkoNyoRCBOWmLOAeFzouw&hl=en&sa=X&ved=2ahUKEwigw_Sw2JnmAhUWFLkGHcMdD9UQ6AEwEXoECAoQAQ#v=onepage&q=eager%20execution%20uses%20graphs%20on%20each%20loop&f=false)

Code with Eager Execution, Run with Graphs: Optimizing Your Code with RevNet as an Example:

<https://medium.com/tensorflow/code-with-eager-execution-run-with-graphs-optimizing-your-code-with-revnet-as-an-example-6162333f9b08>

Memory management for tensorflow:

[https://github.com/miglopst/cs263\\_spring2018/wiki/Memory-management-for-tensorflow](https://github.com/miglopst/cs263_spring2018/wiki/Memory-management-for-tensorflow)

Derivatives With Computation Graphs:

<https://www.youtube.com/watch?v=nJyUyKN-XBQ>

Eager execution:

<https://www.tensorflow.org/guide/eager>