Universitat Oberta de Catalunya

uoc.edu

Análisis de rendimiento del entrenamiento de una red neuronal distribuida para la clasificación de secuencias de ADN

Borja Fernández Morán
Máster Universitario de Ingeniería Informática
Área de computación de altas prestaciones

Sergio Iserte Agut
Josep Jorba Esteve

Enero de 2022


Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Análisis de rendimiento del entrenamiento de una red neuronal para la clasificación de secuencias de ADN*
Nombre del autor:	*Borja Fernández Morán*
Nombre del consultor/a:	Sergio Iserte Agut
Nombre del PRA:	*Josep Jorba Esteve*
Fecha de entrega (mm/aaaa):	01/2022
Titulación:	*Máster universitario de ingeniería informática*
Área del Trabajo Final:	*Computación de altas prestaciones*
Idioma del trabajo:	*Castellano*
Palabras clave	*Distributed Computing; Deep Learning; DNA Classification*

Resumen del Trabajo (máximo 250 palabras):

Las redes neuronales para clasificación de secuencias de virus pueden ser útiles para las autoridades sanitarias a la hora de actuar a la mayor brevedad ante brotes de virus o nuevas variantes de los mismos.

Sin embargo, la gran cantidad de datos de entrada necesarios, así como la complejidad de las propias secuencias genéticas (pueden estar formadas por decenas de miles de aminoácidos cada una), hacen que entrenar una red neuronal de este tipo sea un problema computacionalmente costoso.

En este contexto, el objetivo del presente trabajo consiste en analizar la viabilidad de una red neuronal convolucional para clasificación de secuencias de ADN cuyo entrenamiento se divide en varios nodos computacionales.

Para realizar este análisis se ha construido una red neuronal convolucional que clasifica secuencias genéticas de varios virus diferentes. Sobre esta base, se pretende comparar una versión cuyo entrenamiento se divide en varios nodos de computación respecto a la versión tradicional entrenada en un único nodo, con el objetivo de comprobar si se produce una mejora en el tiempo total de entrenamiento sin perjuicio en el resto de métricas de rendimiento de la red.

Abstract (in English, 250 words or less):

Neural networks for DNA classification can be used by health authorities to quickly identify virus outbreaks or new virus lineages.

However, the complexity of genetic sequences (each one can be made up of tens of thousands of amino acids) along with the large amount of input data required by neural networks, make the training phase of this kind of neural networks to be a computationally expensive problem.

In this context, the goal of this project is to analyze the feasibility of a convolutional neural network that classifies genetic sequences and whose training phase is divided among several computational nodes.

In order to perform this analysis, a convolutional neural network has been built with the purpose of comparing the performance of the multi-node training phase against the traditional one performed in one computational node, to know whether the total training time is improved without impacting the rest of the network performance metrics.

Índice

Lista de figuras

Lista de tablas

Lista de ecuaciones

1. Introducción

1.1 Contexto y justificación del Trabajo

Los virus suponen una de las mayores amenazas a las que se enfrenta la humanidad. En el mundo existen decenas de virus que potencialmente pueden saltar de los animales a los humanos, además de los ya conocidos como el Covid-19 o el ébola.

Por tanto, identificar brotes a través de la secuenciación y el análisis del ADN (o ARN) de los virus o de nuevas variantes de estos es imprescindible para proteger la salud pública global. Sin embargo, el análisis y clasificación de secuencias genéticas es un problema computacionalmente costoso.

En este contexto, el proyecto consiste en el análisis del rendimiento de una red neuronal convolucional para la clasificación de secuencias de ADN, cuyo entrenamiento se realizará de forma distribuida, es decir, se dividirá en varios nodos de computación que trabajarán simultánea y coordinadamente.

Se tomará como referencia el rendimiento de la misma red neuronal entrenada de la forma tradicional en un único nodo, para analizar las diferencias en tiempos de entrenamiento, precisión, consumos de CPU, etcétera.

1.2 Objetivos del Trabajo

El objetivo principal del proyecto es conocer la viabilidad de una red neuronal distribuida para la clasificación de secuencias genéticas de virus, es decir, si se produce una mejora en rendimiento respecto a una red neuronal equivalente entrenada en un único nodo sin que ello provoque una pérdida de precisión de la misma.

De forma más específica, los objetivos serían:

- Creación de un conjunto de datos etiquetados de secuencias genéticas de al menos cuatro virus diferentes que sean ampliamente estudiados, con varios miles de instancias de cada tipo. La comunidad científica mantiene bancos de datos etiquetados de secuencias genéticas de virus y otros patógenos para su identificación, distribución y estudio. Muchos de estos bancos de datos son públicamente accesibles, por lo que son una perfecta fuente de datos para este trabajo.

- Construcción de una red neuronal convolucional para clasificación de secuencias genéticas, con una precisión y un *recall* de al menos el 80%.

- Transformación de dicha red neuronal para que pueda ser ejecutada de forma distribuida en al menos dos nodos de computación, manteniendo la misma precisión y *recall* del 80% y reduciendo el tiempo de entrenamiento al menos un 30%.

- Publicación del proyecto (o al menos del código fuente) en algún tipo de repositorio público de código abierto.

1.3 Enfoque y método seguido

A grandes rasgos, existen dos estrategias a la hora de enfocar este trabajo:

- Construir un producto nuevo: Consiste en desarrollar un producto que no existe, incluyendo las investigaciones necesarias para el desarrollo del mismo.

- Adaptar un producto existente: Consiste en desarrollar un nuevo producto a partir de uno o varios ya existentes para cubrir una nueva necesidad. Se utiliza literatura o proyectos ya existentes como punto de partida para su desarrollo.

El trabajo que se describe en esta memoria puede dividirse en dos grandes fases:

- Construcción de la red neuronal: Para esta fase, se ha utilizado como referencia el artículo científico *Analysis of DNA Classification Using CNN and Hybrid Models*, de los autores Hemalatha Gunasekaran et al. Este artículo analiza diferentes redes neuronales según su rendimiento a la hora de clasificar secuencias genéticas [1]. Aunque el artículo no ofrece ningún producto, es una buena guía para poder construir un producto propio, es decir, una red neuronal convolucional apropiada para el objetivo de este trabajo.

- Desarrollo necesario para que la fase de entrenamiento se ejecute de forma distribuida: Para esta fase se ha utilizado el *framework Horovod*, que es una serie de librerías y herramientas de código abierto desarrolladas por la empresa tecnológica *Uber* que permiten adaptar una red neuronal existente para que el entrenamiento de la misma se ejecute de forma distribuida [2].

Por tanto, el presente trabajo se ha desarrollado a partir de un artículo científico y de un producto existente, con el objetivo de cubrir una necesidad nueva.

1.4 Planificación del Trabajo

A continuación, se muestra una tabla con el desglose de las tareas necesarias para el cumplimiento de los objetivos, así como los recursos necesarios para cada una. La Figura 1 muestra un diagrama de Gantt con la planificación temporal de dichas tareas.

TAREA	DESCRIPCIÓN	RECURSOS	FECHA FIN
T1	Investigación relacionada con el formato y almacenamiento del ADN para investigación biomédica	• *National Center of Biotechnology Information*	20/09/2021
T2	Investigación sobre el estado del arte de los clasificadores de secuencias de ADN	• *Analysis of DNA Sequence Classification Using CNN and Hybrid Models*	25/09/2021
T3	Investigación sobre la implementación de redes neuronales y computación distribuida: Infraestructura, lenguajes de programación, librerías necesarias, etc.	• *Python* • *Tensorflow* • *Keras* • *MPI* • *Horovod*	30/09/2021
T4	Búsqueda de repositorios de secuencias de ADN y creación del conjunto de datos para el entrenamiento de la red neuronal (número razonable de secuencias de al menos cuatro virus)	• *National Center of Biotechnology Information* • *ViruSurf*	08/10/2021
T5	Definición del alcance, viabilidad y objetivos del proyecto		11/10/2021
T6	Definición inicial de la red neuronal convolucional para la clasificación de secuencias: Cómo debe ser la entrada de los datos, qué capas debe tener la red, etc.	• *Analysis of DNA Sequence Classification Using CNN and Hybrid Models*	18/10/2021
T7	Preprocesamiento de los datos (las secuencias genéticas) según lo definido en la tarea anterior		30/10/2021
T8	Implementación de la red neuronal convolucional con entrenamiento tradicional secuencial y ejecución en entorno local		15/11/2021
T9	Ejecución del sistema en el *cluster Tirant*	• *Cluster Tirant* de la Universidad de Valencia	18/11/2021

TAREA	DESCRIPCIÓN	RECURSOS	FECHA FIN
T10	Análisis de resultados: Tiempo de entrenamiento y test, precisión de la red neuronal, etc.		25/11/2021
T11	Implementación de la red neuronal distribuida: Misma arquitectura y datos de entrada, pero el entrenamiento de la red debe estar distribuido entre varios nodos de computación		10/12/2021
T12	Ejecución del sistema en el *cluster Tirant*: Se deberán realizar varias iteraciones cambiando la configuración del sistema distribuido, como por ejemplo el número de nodos implicados, para buscar el resultado óptimo	• *Cluster Tirant* de la Universidad de Valencia	20/12/2021
T13	Análisis de resultados del sistema distribuido y comparación con el sistema tradicional secuencial		25/12/2021
T14	Elaboración de conclusiones sobre la viabilidad de las redes neuronales distribuidas en el ámbito de la biomedicina		31/12/2021
T15	Elaboración de la memoria del proyecto		11/01/2022

Tabla 1.- Lista de tareas planificadas

Figura 1.- Diagrama de Gantt con la planificación de tareas

1.5 Breve sumario de productos obtenidos

Principalmente se obtendrán dos productos de software, que serán dos redes neuronales convolucionales, una de ellas entrenada en un único nodo de computación y la otra adaptada para ser entrenada en más de un nodo. Estos productos de software estarán disponibles en alguna plataforma de código abierto.

Además de esto, también se generará una memoria del proyecto, con toda la información necesaria para poder entrenar y ejecutar ambas redes neuronales, los resultados, análisis y conclusiones.

1.6 Breve descripción de los otros capítulos de la memoria

En los próximos capítulos de la presente memoria se detalla:

- La metodología empleada para la creación del conjunto de datos de secuencias genéticas de virus y su preprocesamiento para que puedan ser utilizados como entrada de datos de las redes neuronales

- La metodología de desarrollo de los dos productos de software señalados anteriormente, las dos redes neuronales convolucionales, utilizando como guía el artículo *Analysis of DNA Classification Using CNN and Hybrid Models*.

- Las ejecuciones de los productos de software con diferentes configuraciones de hardware, tanto en entorno local como en el *cluster Tirant* de la Universidad de Valencia.

- Un análisis comparativo de las métricas obtenidas en dichas ejecuciones.

- Las conclusiones obtenidas, es decir, utilizando el análisis anterior, concretar si es viable o no entrenar de forma distribuida una red neuronal convolucional para la clasificación de secuencias genéticas. En este apartado también se analiza el grado de cumplimiento de los objetivos marcados.

2. Metodología

2.1 Formación del conjunto de datos

El conjunto de datos que se utilizará como entrenamiento de la red neuronal está formado por cinco clases de virus: Covid-19, dengue, ébola, SARS y MERS.

Las secuencias genéticas de estos virus se han descargado de dos repositorios online, el *National Center of Biotechnology Information* (NCBI) y *ViruSurf* [3]. Por cada clase, se puede descargar un fichero en formato FASTA con todas las secuencias que se hayan solicitado.

FASTA es un formato estandarizado de almacenamiento de secuencias genéticas ampliamente utilizado en el ámbito de la biomedicina. Existen por tanto multitud de herramientas que permiten tratar y visualizar estos datos [4].

>AY465926.1			
agagacactcatagagcctgtgtttgtagattgcggacatacttgtcagctatcttatta			
ccatcagttgaaagaagtgcatttacattggctgtaacagcttgacaaatgttaaagaca			
ctattagcataagcagttgtagcatcaccggatgatgttccacctggtttaacatatagt			
gagccgccacacatgaccatctcacttaatacttgcgcacactcgttagctaacctgtag			
aaacggtgtgataagttacagcaagtgttatgtttgcgagcaagaacaagaagagag			

Figura 2.- Secuencia genética en formato *FASTA*

La Figura 2 muestra el formato de una secuencia genética. La primera fila se corresponde con el identificador único de la secuencia. Desde la segunda fila en adelante se muestra la secuencia genética en sí misma.

Las cadenas de ADN están formadas por combinaciones de cuatro aminoácidos:

- Adenina, representada por la letra ‘a’.
- Citosina, representada por la letra ‘c’.
- Guanina, representada por la ‘g’.
- Timina, representada por una ‘t’. En las cadenas de ARN, la timina es sustituida por el uracilo (letra ‘u’). Todas las secuencias de las clases obtenidas son de ADN, es decir, todas contienen timinas y no uracilos.

Las combinaciones entre los aminoácidos no se producen al azar, sino que existen una serie de reglas químicas que gobiernan estas uniones. Además, cada tipo de virus muestra patrones diferentes de uniones entre los aminoácidos. Por tanto, una de las labores principales de la red neuronal será identificar estos patrones dentro de cada secuencia para poder asignarla a una clase u otra.

Las secuencias genéticas tienen un tamaño variable dependiendo de la capacidad que se haya tenido a la hora de hacer la secuenciación, pero pueden estar formadas por hasta decenas de miles de aminoácidos, lo que convierte el entrenamiento de la red neuronal en un problema computacionalmente costoso.

En la siguiente figura se puede observar la distribución de las clases en el conjunto de datos:

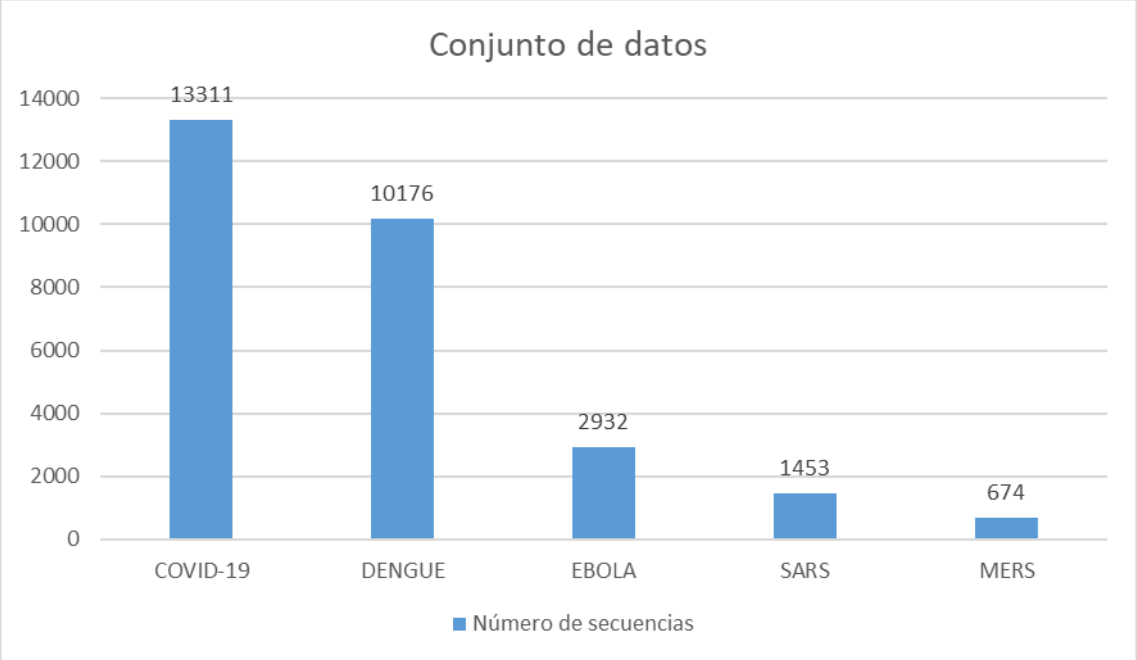

Figura 3.- Distribución de las clases en el conjunto de datos

Para la lectura de los datos se ha utilizado la librería *Biopython*, que permite leer y procesar archivos en formato FASTA de forma simplificada sobre el lenguaje *Python* [5].

```
from Bio import SeqIO
# Load each dataset separately
# for each one, create an array with the class labels
# COVID -> 0, DENGUE -> 1, EBOLA -> 2, MERS -> 3, SARS -> 4

classesDict = {0: "COVID", 1: "DENGUE", 2: "EBOLA", 3: "MERS", 4: "SARS"}
attributes = np.array([])
classes = np.array([])

for key in classesDict:
    fasta_sequences = SeqIO.parse(open("data/sequences"+classesDict[key]+".fasta"), 'fasta')
    attrsSize = attributes.shape[0]
    attributes = np.concatenate ((attributes, np.array([str(fasta.seq) for fasta in fasta_sequences])))
    classSize = attributes.shape[0] - attrsSize
    currentClass = np.empty(classSize)
    currentClass.fill(key)
    classes = np.concatenate ((classes, currentClass))
    print("class ", classesDict[key], " size: ", currentClass.shape[0])
```

Figura 4.- Lectura y creación del conjunto de datos en *Python*

Como se puede observar en la Figura 4, se utiliza esta librería para leer los ficheros. Posteriormente se genera un *array* con todas las secuencias de las diferentes clases y otro *array* con la clase a la que pertenece cada secuencia. Para generar estas estructuras de datos se ha utilizado la librería *Numpy* de *Python*, que permite crear y tratar *arrays* de forma simplificada.

2.2 Preprocesamiento de los datos

Como se muestra en la Figura 3, el conjunto de datos está claramente desbalanceado, ya que mientras la clase mayoritaria tiene unas trece mil instancias, la minoritaria apenas cuenta con seiscientas. Un conjunto de datos de este tipo puede dar lugar a una red neuronal poco precisa a la hora de clasificar elementos de las clases minoritarias, y que tienda a clasificar dichos elementos en las mayoritarias.

El artículo utilizado como guía para el desarrollo de la red neuronal recomienda la técnica *Synthetic Minority Oversampling Technique* o SMOTE para solucionar este problema. A grandes rasgos, esta técnica permite crear instancias artificiales de las clases minoritarias a partir de las instancias reales [6].

Se ha utilizado la librería *Imbalanced-learn* de *Python* para aplicar esta técnica:

```
from imblearn.over_sampling import SMOTEN
attributes = attributes.reshape(-1,1)
oversampling_classes = {2 : 10000, 3 : 10000, 4 : 10000}
sampler = SMOTEN(sampling_strategy=oversampling_classes, random_state=0)
X_res, y_res = sampler.fit_resample(attributes, classes)
```

Figura 5.- Aplicación de la técnica de *oversampling* en *Python*

La función SMOTEN permite crear instancias artificiales para un conjunto de datos nominal, como es este caso [7]. De esta forma, las clases ébola, SARS y MERS tendrán diez mil instancias cada una.

Las redes neuronales convolucionales tienen un buen rendimiento en el procesamiento y clasificación de textos, pero necesitan que el formato de entrada sea numérico. Por tanto, se necesita codificar el conjunto de datos de forma que la entrada de la red neuronal sea una matriz numérica, donde cada fila sea una secuencia genética.

La técnica empleada para ello en el artículo de referencia consiste en transformar cada secuencia genética en un *array* de números, donde a cada tipo de aminoácido se le asigna un número único (adenina → 1, citosina → 2, guanina → 3, timina → 4). La Figura 6 contiene un ejemplo de esta codificación.

```
agagacactcatagagcc → [1, 3, 1, 3, 1, 2, 1, 2, 4, 2, 1, 4, 1, 3, 1, 3, 2, 2]
```

Figura 6.- Ejemplo de transformación de secuencia en lista de números

Cabe destacar que en las secuencias genéticas también pueden aparecer letras que no se corresponden con los cuatro aminoácidos mostrados anteriormente.

IUPAC nucleotide code	Base
A	Adenine
C	Cytosine
G	Guanine
T (or U)	Thymine (or Uracil)
R	A or G
Y	C or T
S	G or C
W	A or T
K	G or T
M	A or C
B	C or G or T
D	A or G or T
H	A or C or T
V	A or C or G
N	any base
. or -	gap

Figura 7.- Códigos *IUPAC* [8]

Se ha decidido que, para la codificación numérica de las secuencias, cualquier letra que no se corresponda con uno de los cuatro aminoácidos será transformada en un cero, ya que no aportan valor a la secuencia.

De esta forma, la codificación numérica del conjunto de datos se haría conforme a la Figura 8.

```
maxSequence = 1000
X_resBinarized = np.empty((0, maxSequence), dtype=np.uint8)
for sequence in X_res:
    sequenceStr = sequence[0]
    sequenceBinarizedStr = ""
    index = 0
    for char in sequenceStr:
        if index < maxSequence:
            if char == 'a':
                sequenceBinarizedStr += "1"
            elif char == 'c':
                sequenceBinarizedStr += "2"
            elif char == 'g':
                sequenceBinarizedStr += "3"
            elif char == 't':
                sequenceBinarizedStr += "4"
            else:
                sequenceBinarizedStr += "0"
        index += 1

    if len(sequenceStr) < maxSequence:
        sequenceLength = len(sequenceStr)
        while sequenceLength < maxSequence:
            sequenceBinarizedStr += "0"
            sequenceLength += 1

    sequenceBinarized = np.array(list(sequenceBinarizedStr), dtype=np.uint8)
    X_resBinarized = np.append(X_resBinarized, [sequenceBinarized], axis=0)
```

Figura 8.- Codificación numérica de las secuencias genéticas en *Python*

Finalmente, el 70% del conjunto de datos se dedicará al entrenamiento de la red neuronal y el 30% para validación y test. Para hacer esta división, se ha utilizado la librería *Scikit-learn* de *Python*, a la que se le puede indicar que reorganice las instancias de forma que la distribución de clases sea la misma (en la medida de lo posible) en ambos conjuntos [9].

```
x_train, x_test, y_train, y_test = train_test_split(X_resBinarized, y_res, test_size=0.3, random_state=4, stratify=y_res)
```

Figura 9.- División del conjunto de datos en conjuntos de entrenamiento y test

Esta reorganización es muy importante porque, tal como se ha formado el conjunto de datos, las instancias de cada clase estan juntas en el conjunto de datos, por lo que el conjunto de test tendrían instancias de clases que no estarían presentes en el conjunto de entrenamiento, con lo cuál no serían correctamente clasificadas.

2.3 Desarrollo de la red neuronal convolucional

La arquitectura de la red neuronal convolucional o CNN está basada también en el artículo de referencia. De esta forma, las capas de la red serían las siguientes:

- *Embedding layer* de dimensión 8 y tamaño de vocabulario 5: Esta capa se encarga de transformar la matriz de entrada en vectores en los que se expresa las relaciones entre las palabras (en este caso los aminoácidos), es decir, subsecuencias de aminoácidos iguales a lo largo de todas las instancias [10]. En este sentido, a mayor dimensión, la capa buscará subsecuencias de tamaño más grande. El tamaño del vocabulario está compuesto por los cuatro aminoácidos más el cero.

- Dos capas convolucionales de una dimensión, con activación *ReLu*, *kernel* de 2x2 y 128 y 64 filtros respectivamente, para la extracción de características [11].

- Dos *Max Pooling layers* de 2x2 para reducir el tamaño de la matriz de características a la mitad después de cada capa convolucional [12].

- Una capa de tipo *Flatten* para reducir la dimensionalidad de la matriz a un *array* de una dimensión, preparando así los datos para las capas de tipo dense [13].

- Dos capas de tipo *Dense* con activación *ReLu* y dimensionalidad 128 y 64 respectivamente.

- Finalmente, una capa de tipo *Dense* de dimensionalidad 5 y activación *Softmax*, cuya salida se corresponde con la clase que la red neuronal ha asignado a cada instancia.

El modelo está compilado utilizando *RMSprop* con optimizador y *categorical crossentropy* como función de pérdida, indicados para el entrenamiento de redes multiclase [14]. El entrenamiento inicialmente se hará en 3 *epochs* con un tamaño de *batch* de 32. Estos parámetros se ajustarán en función de los resultados iniciales.

Esta arquitectura, desarrollada utilizando la librería *Keras*, puede observarse en las figuras 10 y 11.

```
(x_train, y_train), (x_test, y_test) = load_and_preprocessing.execute()
batch_size = 32
num_classes = 5
epochs = 3

# vocab size -> 0,1,2,3,4 = N,A,C,T,G = 5
vocab_size = 5
embedding_dim = 8
maxlen= x_train[0].size

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

y_test1 = np.argmax(y_test,axis=1)
y_train1 = np.argmax(y_train,axis=1)

model = keras.models.Sequential()
model.add(keras.layers.Embedding(input_dim=vocab_size,
                    output_dim=embedding_dim,
                    input_length=maxlen))
model.add(keras.layers.Conv1D(128, 2, activation='relu',
                input_shape=x_train.shape[1:]))
model.add(keras.layers.MaxPooling1D(pool_size=2))
model.add(keras.layers.Conv1D(64, 2, activation='relu',
                input_shape=x_train.shape[1:]))
model.add(keras.layers.MaxPooling1D(pool_size=2))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(num_classes, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
            optimizer='rmsprop',
            metrics=['accuracy'])

model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        callbacks=callbacks,
        validation_data=(x_test, y_test),
        shuffle=True)
```

Figura 10.- Implementación de la red neuronal en *Python*

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 1000, 8)           40
_________________________________________________________________
conv1d (Conv1D)              (None, 999, 128)          2176
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 499, 128)          0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 498, 64)           16448
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 249, 64)           0
_________________________________________________________________
flatten (Flatten)            (None, 15936)             0
_________________________________________________________________
dense (Dense)                (None, 128)               2039936
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 325
=================================================================
Total params: 2,067,181
Trainable params: 2,067,181
Non-trainable params: 0
_________________________________________________________________
```

Figura 11.- Visualización de las capas de la red neuronal durante la ejecución

2.4 Adaptación de la red neuronal para el entrenamiento distribuido con *Horovod*

Horovod es un *framework* de código abierto para el entrenamiento distribuido de redes neuronales, compatible con las librerías *Tensorflow* y *Keras*, entre otras. Este *framework* dispone de dos alternativas para coordinar el trabajo entre nodos:

- *Gloo*: Es un protocolo de comunicación de código abierto desarrollado por *Facebook*. Está incluido en la instalación de *Horovod*.

- MPI: *Message Passing Interface* o Interfaz de Paso de Mensajes es un protocolo de comunicación para computación paralela, diseñado para ser ejecutado en nodos de computación que no comparten memoria. Como los nodos no comparten memoria, no tienen visibilidad sobre la información almacenada en los otros nodos del sistema, como el valor de las variables, etc. MPI define un sistema de paso de mensajes y de sincronización entre nodos a través del protocolo de red TCP, por lo que los nodos del sistema deben ser accesibles entre sí a través de una red *Ethernet* o similar [15].

MPI muestra un mejor rendimiento respecto a *Gloo* para el entrenamiento en CPU, por tanto, MPI es la alternativa elegida. Por tanto, se necesita una instalación de MPI en los sistemas en los que se vaya a ejecutar *Horovod*, ya que internamente, implementará operaciones de MPI, como *allgather* y *allreduce*, para repartir el entrenamiento de la red entre varios nodos computacionales que se comunican entre sí [16].

Horovod está diseñado para ser ejecutado en CPUs o GPUs de nodos que no comparten memoria y se encuentran accesibles entre sí por red. Algunos de los conceptos básicos de este *framework*, que a su vez están basados en conceptos de MPI, son los siguientes [17]:

- *Size*: Es el número de procesos que ejecutarán el programa.
- *Rank*: Es el identificador único de cada proceso, desde 0 al número de procesos menos uno.
- *Allreduce*: Es una operación de MPI que consiste en realizar una operación básica (como suma, resta, la media, etc.) sobre los datos de entrada de todos los procesos involucrados. Posteriormente, se distribuye el resultado a todos los procesos.

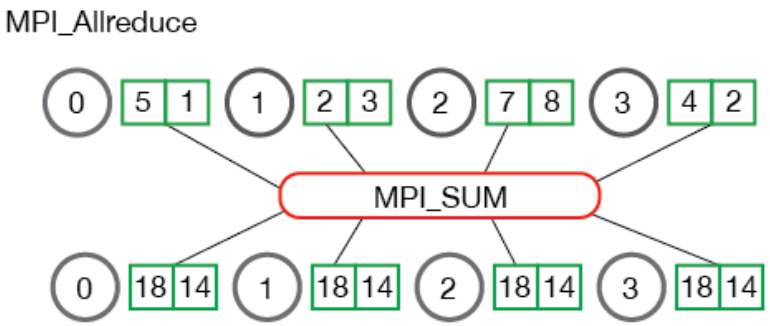

Figura 12.- Esquema de la operación *allreduce*

En este ejemplo, la función *MPI_Allreduce* recibe un *array* por cada proceso, hace la suma agregada en cada una de las dos posiciones del *array*, y envía a todos los procesos el *array* resultante.

De acuerdo con la documentación de *Horovod*, esta operación se utiliza para calcular la media entre los tensores.

- *Allgather*: Es una operación MPI que consiste en recoger datos de cada proceso y repartirlos entre todos los procesos.

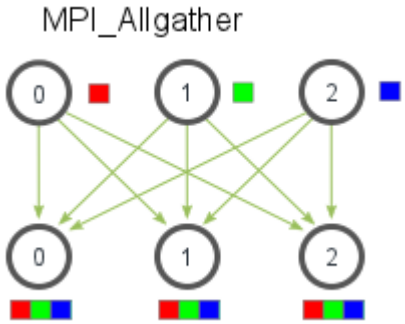

Figura 13.- Esquema de la operación *allgather*

Horovod utiliza esta operación para recoger valores de los tensores dispersos o *sparse tensors*, que son aquellos que contienen una gran cantidad de ceros.

- *Broadcast*: Esta operación MPI consiste en distribuir la información desde un proceso (normalmente el *root* o proceso cero) a todos los demás.

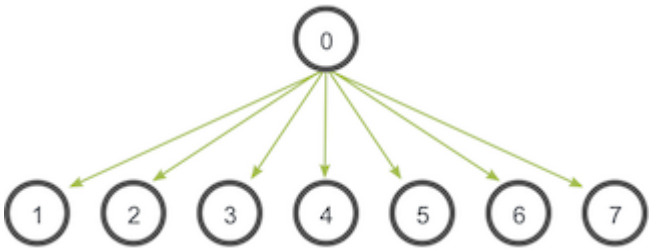

Figura 14.- Esquema de la operación *broadcast*

Horovod utiliza esta operación para distribuir los pesos iniciales de la red desde el proceso *root* a todos los demás.

La Figura 15 muestra la adaptación de la red neuronal para que el entrenamiento se ejecute de forma distribuida. Para ello, primeramente hay que inicializar *Horovod* llamando a la función *init()*. A partir de este momento, el código se ejecutará en paralelo en cada nodo. Después, se debe compilar el modelo utilizando el optimizador distribuido, que se encarga de ejecutar las operaciones MPI mostradas anteriormente sobre el optimizador original.

También se deben compartir las variables a todos los nodos utilizando un *callback,* para que los pesos iniciales de la red y demás variables sean consistentes en todos los nodos. Por último, es recomendable ajustar el *learning rate* dependiendo del número de nodos.

```
import horovod.keras as hvd
# Initialize Horovod
hvd.init()

(x_train, y_train), (x_test, y_test) = load_and_preprocessing.execute()
…
# Horovod: adjust learning rate based on number of GPUs
opt = keras.optimizers.RMSprop(learning_rate=0.0001*hvd.size())

# Horovod: add horovod distributed optimizer
opt = hvd.DistributedOptimizer(opt)
…
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'],
              experimental_run_tf_function=False)

callbacks = [
    TimingCallback(),
    # Horovod: broadcast initial variable states from rank 0 to all other processes
    # This is necessary to ensure consistent initialization of all workers when
    # training is started with random weights or restored from a checkpoint
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),
]

# Horovod: save checkpoints only on worker 0 to prevent other workers from
corrupting them
if hvd.rank() == 0:
    callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))

model.fit(x_train, y_train,
          batch_size=batch_size,
          callbacks=callbacks,
          epochs=epochs,
          verbose=1 if hvd.rank() == 0 else 0,
          validation_data=(x_test, y_test),
          shuffle=True)
```

Figura 15.- Adaptación de la red neuronal con la librería de *Horovod*

3. Ejecución y resultados

3.1 Ejecución de las redes neuronales

La ejecución de la red neuronal se ha hecho en el supercomputador *Tirant* de la Universidad de Valencia. Este supercomputador se compone de 336 nodos conectados por *Ethernet* de 1Gbit/s e *Infiniband* de 40 Gbit/s. Cada nodo dispone de 16 *cores* y 32 GB de RAM. No dispone de procesadores gráficos (GPUs) [18].

El supercomputador implementa el sistema de colas SLURM. Este sistema permite encolar trabajos y asignarles los recursos de hardware necesarios. SLURM los ejecutará según el orden de prioridad definido por los administradores [19].

Así pues, cuando un usuario se conecta a *Tirant*, lo hace en realidad a un servidor de *login*. Desde este servidor, el usuario puede administrar o desarrollar su trabajo. Cuando el usuario quiera ejecutarlo, debe encolarlo en el sistema de colas. SLURM le dará una prioridad basada en múltiples factores. Cuando sea su turno, reservará los servidores de ejecución necesarios (o parte de ellos) y ejecutará el trabajo.

De esta forma, el *script* que permite ejecutar la red neuronal desarrollada para un nodo es el siguiente:

```
#!/bin/bash
#SBATCH --job-name="virusCNN"
#SBATCH --workdir=.
#SBATCH --output=virusCNN_%j.out
#SBATCH --error=virusCNN_%j.err
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --exclusive
#SBATCH --mem-per-cpu=16000MB
#SBATCH --time=06:00:00

python main.py
```

Figura 16.- *Script* de ejecución de la red neuronal en un nodo

El *script* de la Figura 16 define lo siguiente:

- El nombre del trabajo (visible en la cola).
- El directorio en el que se encuentra el programa (en este caso, el directorio actual).
- Los ficheros donde se guardarán la salida estándar y la de error del programa.
- El número de *tasks* (*--ntasks*) hace referencia al número de procesos que creará el sistema operativo para ejecutar el programa. En este caso, un único proceso. Dentro de este proceso, *Tensorflow* generará el número de hilos que crea conveniente, y estos hilos se ejecutarán a lo largo de todos los *cores* del nodo (a no ser que se indique lo contrario), pues estos *cores* comparten el mismo contexto (memoria, etc.).
- El número de nodos de computación que se reservarán para la ejecución del programa. En este caso también uno.
- La directiva *–exclusive* significa que ningún otro programa estará ejecutándose en el nodo reservado al mismo tiempo que este trabajo.
- La memoria RAM reservada para la ejecución del trabajo es de 16 GB. En caso de que el programa se ejecutase en más de un nodo, se reservarían 16 GB por nodo.
- SLURM cancelará la ejecución del trabajo si ésta dura más de seis horas.
- La última línea es el comando de ejecución del programa.

Por tanto, este programa se ejecutará en un proceso y en un nodo donde todos sus *cores* se reservarán de forma exclusiva para albergar los hilos que la red neuronal utilice para su entrenamiento.

Para encolar el trabajo basta con ejecutar este comando:

```
sbatch submit.cmd
```

Figura 17.- Comando para añadir el trabajo a la cola

Para la ejecución de la red neuronal distribuida se debe especificar el número de procesos y nodos, así como cuántos procesos ejecutará cada nodo. La Figura 18 contiene el *script* para dos procesos:

```
#!/bin/bash
#SBATCH --job-name="CNN2x1"
#SBATCH --workdir=.
#SBATCH --output=CNN2x1_%j.out
#SBATCH --error=CNN2x1_%j.err
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=1
#SBATCH --exclusive
#SBATCH --mem-per-cpu=16000MB
#SBATCH --time=6:00:00

horovodrun -np 2 -H $(scontrol show hostnames $SLURM_JOB_NODELIST | awk NR==1'{print $1}'):1,$(scontrol show hostnames $SLURM_JOB_NODELIST | awk NR==2'{print $1}'):1 --timeline-filename timeline2x1.json --autotune --autotune-log-file autotune2x1.csv python mainHorovod.py
```

Figura 18.- *Script* de ejecución de la red neuronal en dos nodos

Este *script* define lo siguiente:

- En este caso, el número de procesos que se crearán será dos.
- La directiva *–ntasks-per-node* hace referencia al número de procesos que se ejecutarán en cada nodo. Como el valor de la directiva es uno, indirectamente se están reservando dos nodos
- Como se comentó anteriormente, la RAM reservada es de 16 GB por nodo.

Por tanto, SLURM reservará todos los *cores* de dos nodos en exclusiva y ejecutará un proceso en cada uno. *Tensorflow* creará cuantos hilos crea necesarios por cada proceso para aprovechar al máximo los *cores* reservados.

Mención especial requiere el comando de ejecución de la red neuronal:

- El comando *horovodrun* es un *wrapper* de *mpirun*, es decir, internamente se está ejecutando MPI con una serie de parámetros que son transparentes al usuario.

- *horovodrun* necesita recibir el número de procesos que se generarán en ejecución, en este caso dos (*-np 2*).

- Además, es necesario especificar en qué nodos se ejecutarán los procesos y cuántos procesos por nodo. Esto se hace con el parámetro *–H nodo1:1,nodo2:1*.

- Para poder hacer ésto, es necesario recuperar el *hostname* de los nodos reservados por SLURM. Esto se hace con el comando *scontrol show hostnames $SLURM_JOB_NODELIST*.
- Este comando devuelve el nombre completo de los nodos (*$SLURM_JOB_NODELIST* contiene los nombres de forma abreviada), cada uno en una fila.

- Por tanto, utilizando el comando *awk* se puede procesar la salida del comando anterior, sabiendo que cada fila contiene el nombre completo de cada nodo. Por ejemplo, *NR==1* devolverá el nodo de la fila uno.

- El parámetro *–timeline-filename* permite generar un fichero con las métricas de la ejecución de *Horovod*. Este fichero se analizará con detalle más adelante.

- El parámetro *–autotune* indica que *Horovod* ajustará sus parámetros internos en los primeros pasos del entrenamiento para buscar la combinación que maximice la eficiencia del entrenamiento.

3.2 Resultados iniciales de la red neuronal ejecutada en un nodo

Las métricas más relevantes a la hora de analizar el rendimiento de una red neuronal son las siguientes [20]:

- Precisión: Respecto a una clase, hace referencia al porcentaje de casos realmente positivos respecto al total de casos que se han etiquetado como positivos (los clasificados correctamente más los que en realidad pertenecen a otra clase). Esta métrica da visibilidad a los miembros de otra clase que fueron asignados a ésta.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Ecuación 1.- Precisión de una clase

- *Recall*: Respecto a una clase, hace referencia al porcentaje de casos clasificados como positivos respecto al total de casos realmente positivos (los clasificados correctamente más aquellos que debieron asignarse a esta clase pero se asignaron a otra). Esta métrica da visibilidad a los miembros de la clase que se escapan y se asignan a otra.

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Ecuación 2.- *Recall* de una clase

- *Accuracy*: Respecto al modelo, es el número de predicciones correctas respecto al número total de predicciones.

Las estadísticas del entrenamiento de la red neuronal tradicional no distribuida son las siguientes:

Classification report				
	Precision	**Recall**	**F1-score**	**support**
Covid	1.00	1.00	1.00	3994
Dengue	1.00	1.00	1.00	3053
Ebola	0.99	0.99	0.99	3000
Mers	0.99	1.00	0.99	3000
Sars	0.99	0.99	0.99	3000
Accuracy			0.99	16047
Macro avg	0.99	0.99	0.99	16047
Weighted avg	0.99	0.99	0.99	16047

Tabla 2.- Métricas de rendimiento de la red neuronal

Como se puede observar en la Tabla 2, las métricas descritas en el apartado anterior presentan resultados muy buenos, cercanos al 100%.

La matriz de confusión mostrada en la Tabla 3 permite visualizar la distribución de las predicciones que ha hecho la red respecto a la clase real a la que pertenece cada instancia.

Confusion matrix					
Covid	3986	0	2	6	0
Dengue	0	3041	11	1	0
Ebola	0	4	2971	4	21
Mers	1	2	7	2988	2
Sars	2	6	10	21	2961
Real/predicted	**Covid**	**Dengue**	**Ebola**	**Mers**	**Sars**

Tabla 3.- Matriz de confusión de la red neuronal

Como se puede observar, la mayoría de las instancias se han clasificado correctamente. Por ejemplo, de los 3.994 miembros de la clase Covid-19, 3986 miembros fueron clasificados correctamente, 2 se clasificaron como de la clase ébola, y 6 como de la clase MERS.

El entrenamiento de la red se ha hecho en tres *epochs*, aunque, como se puede observar en la siguiente figura, el error de precisión se mantiene muy bajo y estable ya desde el primero.

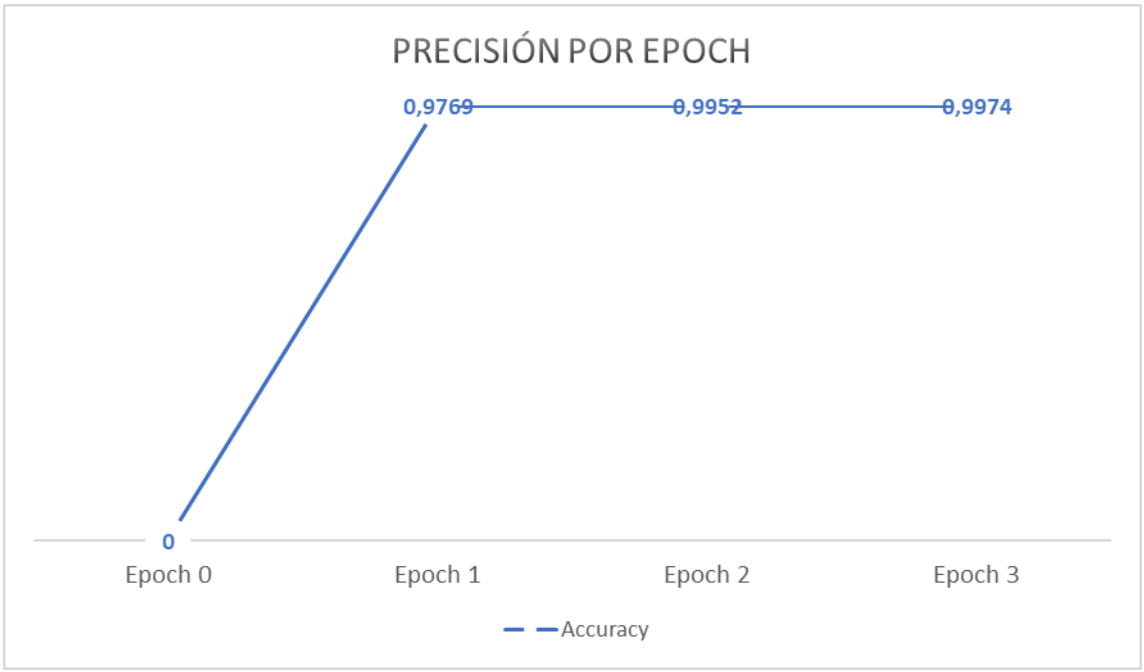

Figura 19.- Precisión de la red neuronal en cada *epoch*

También se ha medido el tiempo de entrenamiento por *epoch*, que en esta caso se encuentra un poco por debajo de los 30 segundos, tal como muestra la Figura 20.

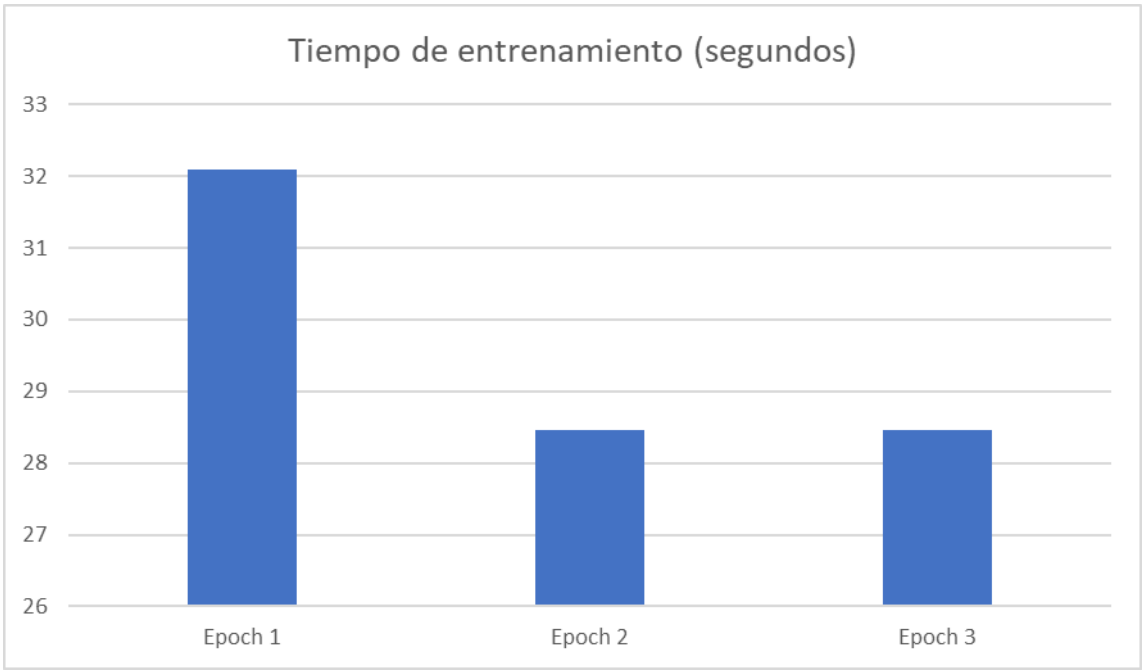

Figura 20.- Tiempo de entrenamiento en cada *epoch*

En esta figura se puede observar que el tiempo de entrenamiento es superior en el primer *epoch* y después se estabiliza. Cada *epoch* es una iteración en la que los datos pasan completamente a través del modelo y se produce una salida, por tanto, puede ser que el primer *epoch* sea más largo debido a la inicialización del modelo, de los pesos, etc. Sin embargo, sería necesario hacer un *profiling* de la red neuronal para poder conocer las causas con exactitud.

3.3 Resultados iniciales de la red neuronal con entrenamiento distribuido

A juzgar por las métricas de la clasificación, la red neuronal con entrenamiento distribuido utilizando la librería *Horovod* mantiene un rendimiento muy similar.

Classification report				
	Precision	**Recall**	**F1-score**	**support**
Covid	1.00	1.00	1.00	3994
Dengue	1.00	1.00	1.00	3053
Ebola	0.96	1.00	0.98	3000
Mers	1.00	0.99	0.99	3000
Sars	0.99	0.97	0.98	3000
Accuracy			0.99	16047
Macro avg	0.99	0.99	0.99	16047
Weighted avg	0.99	0.99	0.99	16047

Tabla 4.- Métricas de rendimiento de la red neuronal en dos nodos

La matriz de confusión y la precisión por *epoch* tampoco muestran diferencias significativas.

Confusion matrix					
Covid	3989	0	2	2	1
Dengue	0	3038	12	0	3
Ebola	0	1	2997	1	1
Mers	2	2	9	2997	10
Sars	4	4	89	6	2897
Real/predicted	**Covid**	**Dengue**	**Ebola**	**Mers**	**Sars**

Tabla 5.- Matriz de confusión de la red neuronal en dos nodos

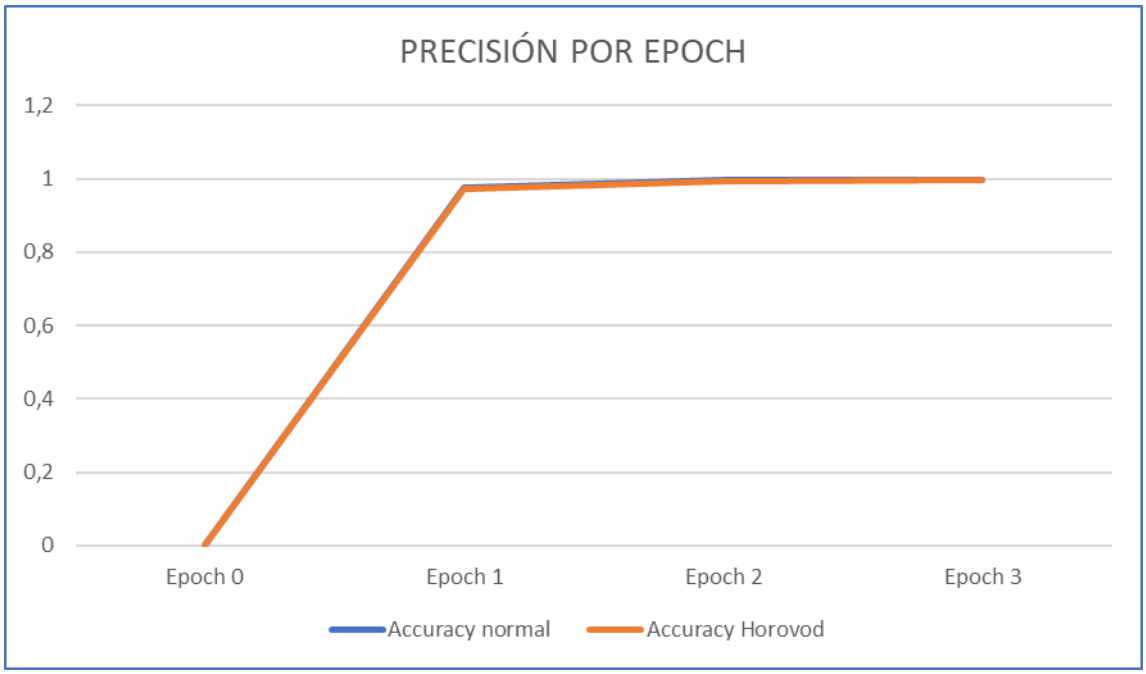

Figura 21.- Comparación de la precisión por *epoch* entre uno y dos nodos

Sin embargo, el tiempo de entrenamiento es mayor, a pesar de que teóricamente el trabajo se reparte entre dos nodos comunicados mediante MPI.

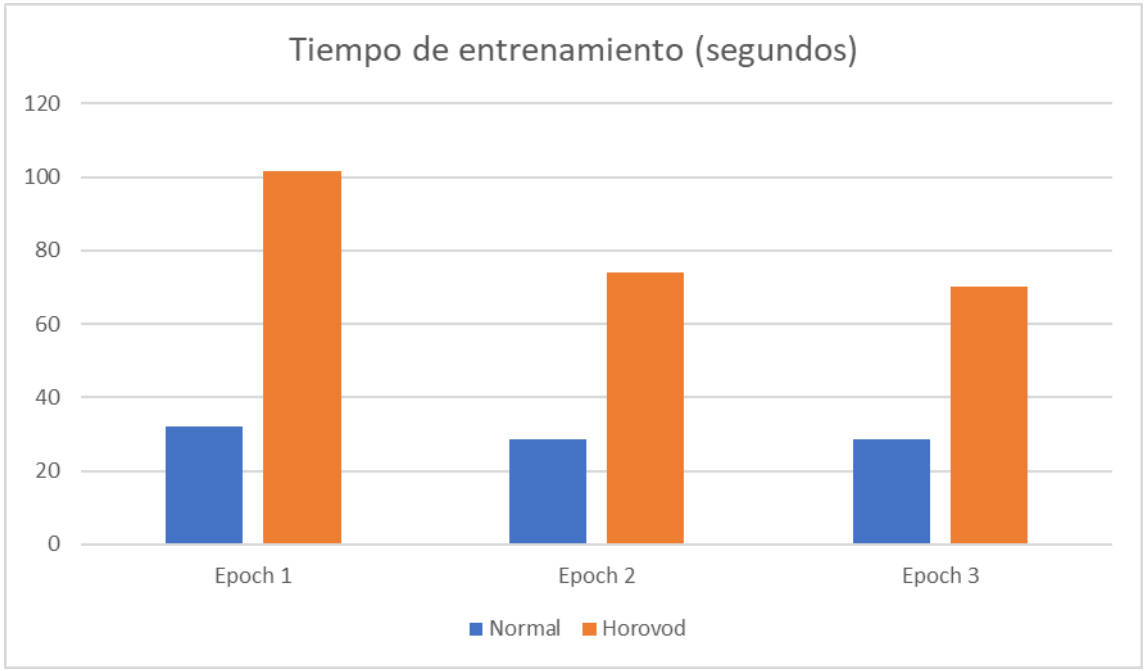

Figura 22.- Comparación del tiempo de entrenamiento entre uno y dos nodos

La diferencia en el tiempo de entrenamiento entre el primer *epoch* y los demás es más notable en la versión de la red implementada con *Horovod*. Esto puede ser debido al parámetro *–autotune* comentado anteriormente, que hace que *Horovod* invierta tiempo en maximizar la eficiencia de sus parámetros internos durante el primer *epoch*.

3.4 Redimensionamiento artificial del problema

Los resultados iniciales muestran un empeoramiento en el tiempo de entrenamiento en la versión distribuida en dos nodos respecto a la versión entrenada en un único nodo.

Teniendo en cuenta que *Horovod* implementa MPI, de forma que los nodos deben sincronizarse y enviarse mensajes mediante TCP, puede que esta situación se deba a que el tamaño del problema sea demasiado pequeño y la mejora en el procesamiento no sea visible por el tiempo que se invierte en la sincronización entre los dos nodos.

Para comprobarlo, se ha hecho un redimensionamiento artificial del problema para hacerlo más grande, tanto en el número total de instancias como en el tamaño de cada secuencia genética. Este redimensionamiento consiste en:

- El número de instancias de cada clase se aumenta a 13000 utilizando la técnica SMOTE vista anteriormente.

- El tamaño de cada secuencia se aumenta de 1000 aminoácidos a 6000.

- El número de *epochs* se aumenta a 6 para poder observar más claramente en qué momento se estabiliza el tiempo de entrenamiento.

Este ajuste se ha hecho teniendo en cuenta las limitaciones de hardware de *Tirant*, especialmente los 32 GB de RAM de los que dispone cada nodo. Se han hecho pruebas con un tamaño de matriz más grande (mayor número de secuencias y/o secuencias más largas), y los trabajos terminan cancelándose porque no disponen de memoria RAM suficiente.

Se ha preparado un banco de pruebas utilizando hasta tres nodos y cuatro procesos por nodo (por ejemplo, 1x2 será una configuración de un nodo, dos procesos por nodo). Los resultados en cuanto a la precisión de los modelos son similares para todas las configuraciones.

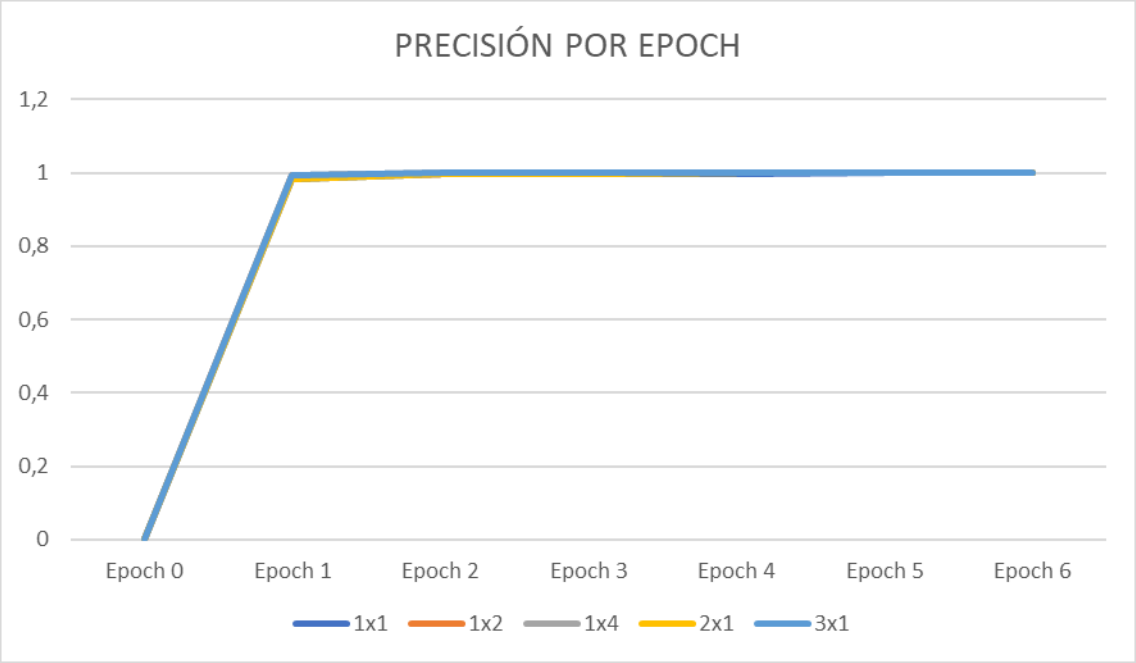

Figura 23.- Comparación de la precisión por *epoch* con mútliples configuraciones

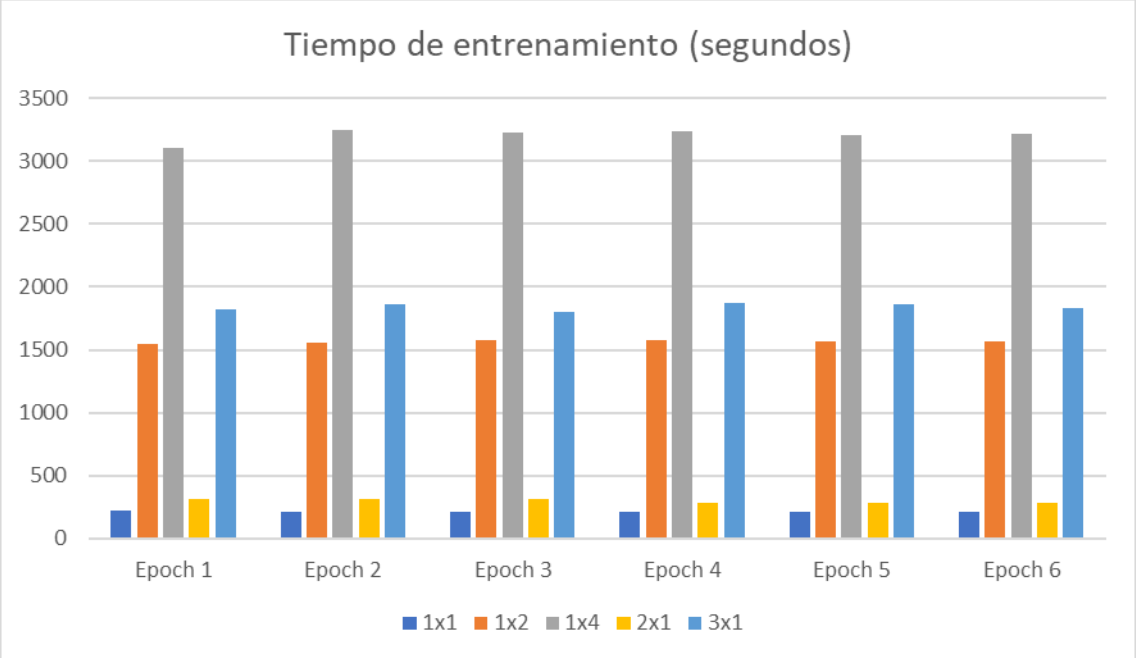

Figura 24.- Comparación del tiempo de entrenamiento con múltiples configuraciones

Sin embargo, como se puede observar en la Figura 24, la versión 1x1 (un nodo, un proceso por nodo) sigue siendo la de menor tiempo de entrenamiento.

El supercomputador *Tirant* dispone de dos conexiones de red, *Ethernet* de 1 Gbit/s e *Infiniband* de 40 Gbit/s. En los *scripts* mencionados hasta ahora, no se realiza ningún tipo de configuración para especificar qué red deben utilizar los nodos para comunicarse. MPI ofrece una serie de parámetros que permiten hacer esta configuración, aunque no están disponibles directamente en *wrapper horovodrun*.

Así pues, para poder especificar la red a través de la cual se comunicarán los nodos, es necesario utilizar el comando *mpirun* que se ejecuta por debajo de *horovodrun*. La Figura 25 muestra esta configuración para la red *Ethernet*, cuyo puerto de red recibe el nombre de *eth0*. Para realizar esta misma configuración sobre la red *infiniband*, basta con cambiar *eth0* por *ib0*. El resto de parámetros del comando *mpirun* están definidos en la documentación de *Horovod* como el comando que realmente se ejecuta cuando se ejecuta *horovodrun*.

Tras realizar pruebas con estas dos configuraciones, se ha observado que el tiempo de entrenamiento por *epoch* no mejora respecto a las configuraciones expuestas anteriormente, de lo que se puede extraer que la red de comunicación de los nodos no es un factor determinante a la hora de explicar los resultados.

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
default         capote.red.uv   0.0.0.0         UG        0 0          0 eth1
10.1.0.0        *               255.255.0.0     U         0 0          0 ib0
10.2.0.0        *               255.255.0.0     U         0 0          0 eth0
147.156.0.0     *               255.255.252.0   U         0 0          0 eth1


mpirun -np 2 -H $(scontrol show hostnames $SLURM_JOB_NODELIST | awk NR==1'{print $1}'):1,$(scontrol show hostnames $SLURM_JOB_NODELIST | awk NR==2'{print $1}'):1 --bind-to none --map-by slot -x NCCL_DEBUG=INFO -x LD_LIBRARY_PATH -x PATH --mca pml ob1 --mca btl ^openib --mca oob_tcp_if_include eth0 --mca btl_tcp_if_include eth0 python mainHorovod.py
```

Figura 25.- Especificación de la red de conexión en la ejecución

3.5 Resultados finales y discusión

Horovod dispone de una herramienta para analizar el rendimiento de su propia ejecución, llamada *Horovod Timeline*. Para generar el fichero con el rendimiento, basta con añadir el parámetro *–timeline-filename XXX.json* al comando *horovodrun*.

Este fichero puede abrirse con la herramienta *chrome://tracing* de *Google Chrome*. A continuación, se muestra el *timeline* para la configuración 2x1 (dos nodos, un proceso por nodo).

En la Figura 26, puede observarse a simple vista como, a lo largo de los seis *epochs*, el optimizador distribuido va ejecutando ciertas operaciones sobre los vectores tensoriales. Cada fila se corresponde con un *worker* o *thread*, que tiene un identificador único o *pid*. En este caso, la ejecución ha generado hasta 36 *threads*. No se ha limitado de ninguna forma la cantidad de *threads* que se puede crear *Horovod* y quizá esto esté provocando tiempos de espera al ejecutarse varios hilos en el mismo procesador. Quizá sea esta una de las razones por las que la versión distribuida es más lenta

Las operaciones entre varios *workers* se realizan en dos fases, una de negociación y otra de procesamiento. A continuación, se muestran las fases en detalle para la operación *allreduce*, pero el funcionamiento es equivalente para el *allgather* y *broadcast* [19]:

- Negociación: Cada *worker* reporta que está listo cuando aparece un *tick* durante su operación *NEGOTIATE_ALLREDUCE*. Una vez que todos los *workers* involucrados están listos esta operación finaliza.
- Procesamiento: En esta fase se ejecutan las operaciones *MPI*. Se puede dividir en varias subfases, entre las que destacan:
    - *INIT_FUSION_BUFFER*: Operación de inicialización del *buffer*. No es necesaria en todas las iteraciones.
    - *MEMCPY_IN_FUSION_BUFFER*: Operación de copia de datos al *buffer* de *tensor fusion*.
    - *MPI_ALLREDUCE*: Operación *allreduce* de *MPI* en CPU.
    - *MEMCPY_OUT_FUSION_BUFFER*: Copia de datos desde el *buffer* de *tensor fusion*.

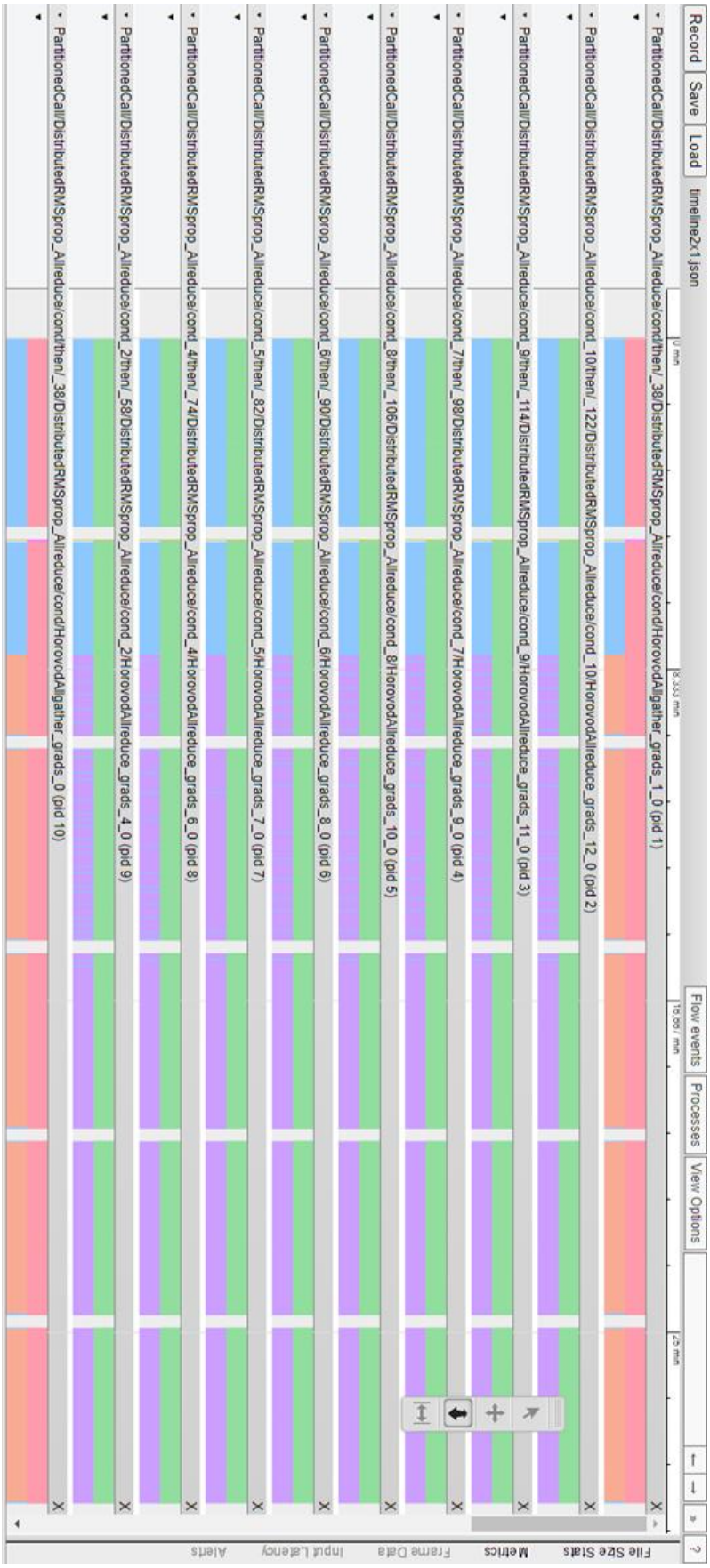

Figura 26.- Trazas de la ejecución 2x1: dos nodos, un proceso por nodo

La Figura 27 es bastante representativa de lo que ocurre a lo largo de todo el *timeline*, donde diferentes grupos de *threads* realizan diferentes tipos de operaciones MPI. En la figura se muestran ejemplos de una operación *allreduce*, *broadcast* y *allgather*. En todas estas operaciones el tiempo de negociación entre los *threads* y las operaciones de *buffer* (ya sea inicialización o entrada/salida) ocupan más tiempo que la parte de procesamiento, que es la operación MPI. Esta puede ser una de las razones por las que la versión distribuida de la red neuronal sea más lenta.

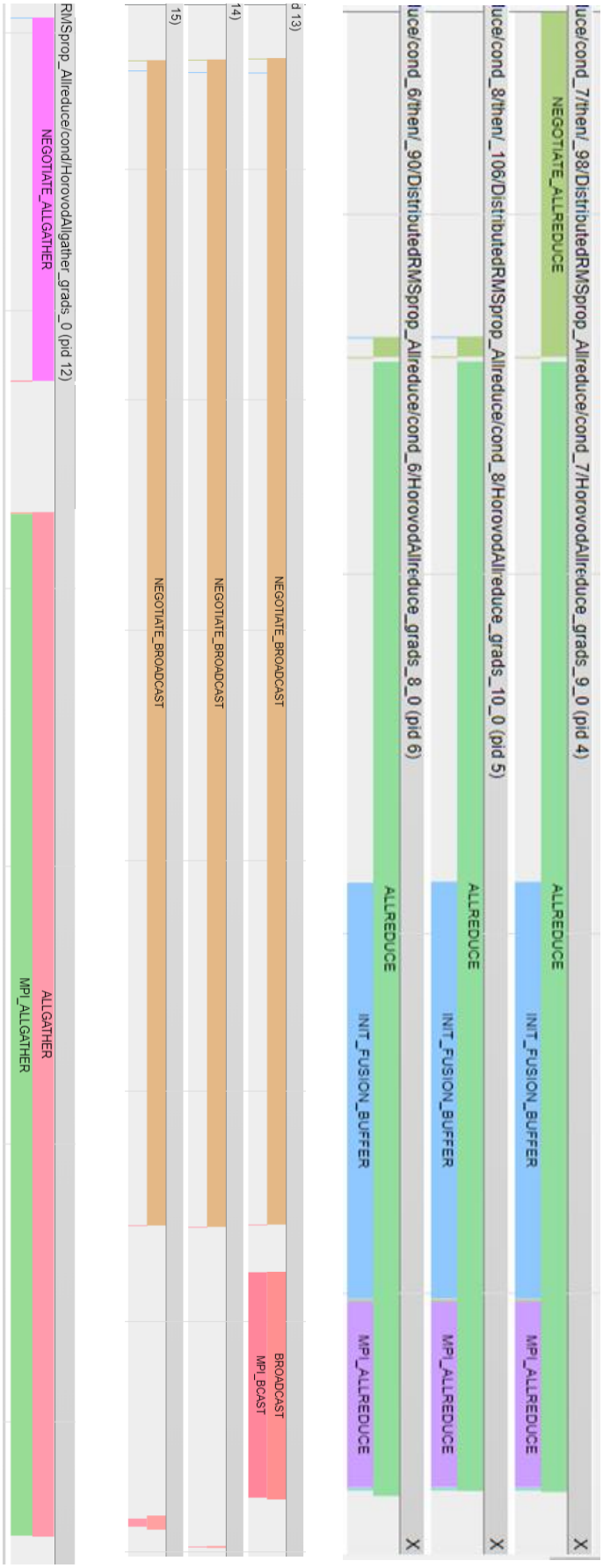

Figura 27.- Trazas de operaciones *allreduce*, *broadcast* y *allgather*

4. Conclusiones

La red neuronal construida a partir del artículo *Analysis of DNA Sequence Classification Using CNN and Hybrid Models* muestra un rendimiento superior a los objetivos marcados. Mientras que el objetivo era conseguir al menos un 80% en precisión y *recall*, estas dos métricas están muy cercanas al 100% en todas las clases.

Respecto a la creación del conjunto de datos, ha sido bastante sencillo encontrar un número más que suficiente de secuencias genéticas del Covid-19. Sin embargo, ha sido más difícil encontrar secuencias del resto de virus, dando lugar a un conjunto de datos desbalanceado. En cualquier caso, este problema ha podido solventarse aplicando una técnica de generación de instancias nuevas a partir de las existentes.

La red neuronal desarrollada para el entrenamiento distribuido mantiene unas métricas muy similares a su homóloga entrenada en un único nodo, por lo que este objetivo también se ha cumplido.

Sin embargo, no se ha conseguido el objetivo de disminuir el tiempo de entrenamiento en un 30% respecto a la red entrenada en un nodo. Es más, el tiempo de entrenamiento ha aumentado. Algunas de las razones de este resultado pueden ser:

- El supercomputador *Tirant* no dispone de GPUs, por lo que no puede explotarse toda la capacidad de *Horovod*.

- *Horovod* no ha sido configurado correctamente (número de hilos, ajuste de parámetros en la ejecución, etc.), o quizás la red neuronal desarrollada no es la más apropiada para este *framework*.

- El tamaño del problema no es lo suficientemente grande para que la reducción en el procesamiento compense el tiempo de negociación y sincronización añadido.

En cuanto a la planificación, las fases de pruebas de la versión distribuida con diferentes configuraciones en *Tirant* se han demorado más de lo esperado. A medida que se ha ido aumentando el tamaño del problema o el número de nodos implicados los trabajos han pasado más tiempo en la cola esperando a ser ejecutados, con un tiempo de espera de varios días o incluso semanas.

Una vez que se comprobó que con el tamaño inicial del problema el resultado no era el esperado, se ha necesitado realizar muchas pruebas hasta dar con un tamaño que maximizase los recursos de *Tirant*, especialmente los 32 GB de RAM por nodo. Esto ha dado lugar a

muchas iteraciones con diferentes configuraciones y con el hasta dar con el redimensionamiento expuesto en este trabajo.
Una vez constatado que el problema del tiempo de entrenamiento seguía sin solucionarse, se lanzó una batería de pruebas con diferentes configuraciones de *Horovod* (con los correspondientes tiempos de espera), que tampoco dieron resultado.

Esta desviación respecto a la planificación inicial ha provocado que no haya podido realizarse un análisis en profundidad de las causas de la no reducción del tiempo de entrenamiento.

Así pues, en este punto se puede concluir que, con el trabajo realizado hasta el momento, no es viable utilizad *Horovod* en CPU para disminuir el tiempo de entrenamiento de una red neuronal para clasificación de secuencias de ADN.

De cara al futuro, sería necesario hacer un análisis en profundidad para conocer las causas de estos resultados, como un *profiling* de ambas redes neuronales para medir con más precisión los tiempos de procesamiento y sincronización, consumos de CPU, número de threads, etc. La herramienta *Horovod Timeline* no permite hacer un análisis lo suficientemente concreto, y como no puede aplicarse a la versión de la red para un nodo no se pueden hacer comparaciones. Sin este análisis, las razones expuestas anteriormente son simples conjeturas.

En el futuro, en base a dicho *profiling*, se podrían hacer las modificaciones necesarias a los productos de software para intentar cumplir el objetivo de la disminución del 30% en el tiempo de entrenamiento.

El código fuente de este trabajo se encuentra en un repositorio en *GitHub* accesible desde este enlace: github.com/borjafm14/virusCNN.

Este trabajo ha sido posible gracias al clúster *Tirant III* del Servei d'Informática de la Universitat de València.

5. Glosario

Framework: Entorno de desarrollo de software que puede incluir librerías, herramientas y lenguajes de programación, con un propósito específico.

Array: Lista de elementos ordenados en fila. A más bajo nivel, una zona de almacenamiento contiguo que contiene elementos del mismo tipo.

Script: Fichero que contiene una lista de comandos ejecutables por el intérprete de comandos del sistema operativo u otro programa.

CPU: Unidad Central de Procesamiento, por sus siglas en inglés. Es el procesador dentro de un ordenador, que recibe instrucciones y realiza operaciones aritméticas.

GPU: Unidad de Procesamiento Gráfico, por sus siglas en inglés. Procesador destinado a gráficos y operaciones de coma flotante.

Core: Agrupación de procesadores o CPUs. Los ordenadores modernos suelen tener varios *cores* con varias CPUs cada uno.

RAM: Memoria de acceso aleatorio, por sus siglas en inglés. Chip de memoria de menor tamaño que el disco duro y de acceso más rápido. Se utiliza para almacenar los datos temporales de la ejecución de los programas, entre otras cosas.

Wrapper: En el contexto de este trabajo, es un comando que internamente ejecuta otro más complejo. De esta forma, la complejidad del comando interno es transparente al usuario, lo que le facilita el trabajo.

Epoch: En el contexto de las redes neuronales, se trata de una iteración, es decir, un ciclo en el que los datos de entrada pasan a través de todas las capas de la red neuronal y se produce una salida o resultado.

Thread: Hilo del sistema operativo. El concepto es similar a un proceso con la diferencia de que los hilos comparten el mismo contexto (memoria, variables, pila de ejecución, etc.).

CNN: *Convolutional Neural Network*, por sus siglas en inglés. Se trata de las redes neuronales convolucionales o de varias capas.

TCP: Transmission Control Protocol, por sus siglas en inglés. Protocolo de red para el intercambio de paquetes ampliamente utilizado.

Profiling: Análisis del rendimiento de un programa de software.

Login: Interfaz, proceso, servidor, etc; de acceso a un sistema.

6. Bibliografía

[1] Hemalatha Gunasekaran, K. Ramalakshmi, A. Rex Macedo Arokiaraj, S. Deepa Kanmani, Chandran Venkatesan, C. Suresh Gnana Dhas, "Analysis of DNA Sequence Classification Using CNN and Hybrid Models", Computational and Mathematical Methods in Medicine, vol. 2021, Article ID 1835056, 12 pages, 2021. https://doi.org/10.1155/2021/1835056

[2] Horovod, distributed deep learning training framework. horovod.ai

[3] Canakoglu A, Pinoli P, Bernasconi A, Alfonsi T, Melidis DP, Ceri S. ViruSurf: an integrated database to investigate viral sequences. Nucleic Acids Res. 2021 Jan 8;49(D1):D817-D824. doi: 10.1093/nar/gkaa846. PMID: 33045721; PMCID: PMC7778888.

[4] Sergio Hernández López. Genómica Computacional. Formato FASTA. sites.google.com/site/genomicaciencias/laboratorio/formato-fasta

[5] Biopython. biopython.org

[6] SMOTE for imbalance classification in python. machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/

[7] Imbalanced learn library. github.com/scikit-learn-contrib/imbalanced-learn

[8] IUPAC codes. bioinformatics.org/sms/iupac.html

[9] Scikit-learn. Machine learning in python. scikit-learn.org/stable/

[10] Keras Embedding layer. keras.io/api/layers/core_layers/embedding/

[11] Conv1D layer. keras.io/api/layers/convolution_layers/convolution1d/

[12] Keras API reference. MaxPooling1D layer. keras.io/api/layers/pooling_layers/max_pooling1d/

[13] Flatten layer. keras.io/api/layers/reshaping_layers/flatten/

[14] Keras API reference. Losses. keras.io/api/losses/

[15] Open MPI. Open Source High Performance Computing. open-mpi.org/s

[16] Horovod – GitHub. github.com/horovod/horovod

[17] Horovod distributed training framework. Concepts. horovod.readthedocs.io/en/stable/concepts_include.html

[18] Universitat de Valencia. Tirant v3 User's Guide. uv.es/tirant/Manuales/Tirant3_user_manual.pdf

[19] SLURM Workload Manager. Documentation. slurm.schedmd.com/documentation.html

[20] Precision, Recall, F1, Accuracy en clasificación. Jose Martinez Heras. iartificial.net/precision-recall-f1-accuracy-en-clasificacion/