Red de Federated Learning: de modelo cliente / servidor a P2P

Tomás García Pérez

Máster en Ingeniería Informática

Félix Freitag

Junio 2021

FICHA DEL TRABAJO FINAL

Título del trabajo:	Red de Federated Learning: de modelo cliente / servidor a P2P
Nombre del autor:	Tomás García Pérez
Nombre del consultor:	Félix Freitag
Fecha de entrega (mm/aaaa):	06/2021
Área del Trabajo Final:	Sistemas Distribuidos
Titulación:	*Máster en Ingeniería Informática*

Resumen del Trabajo (máximo 250 palabras):

Como resultado de un Trabajo Fin de Máster anterior, se obtuvo un software que aplicaba el concepto de Federated Learning, esto es, la habilidad de hacer Machine Learning sin la necesidad de compartir ni centralizar los datos en una sola máquina.

Este software seguía el diseño original propuesto por los ingenieros de Google, en el que hay un servidor central y varios nodos cliente que realizan el Machine Learning.

Este proyecto tiene como objetivo principal realizar la versión descentralizada P2P de ese software, en el que todos los nodos no tienen ningún rol definido.

Abstract (in English, 250 words or less):

As a result of a previous Master's Thesis, a software was obtained that applied the concept of Federated Learning, that is, the ability to do Machine Learning without the need to share or centralize the data in a single machine.

This software followed the original design proposed by Google engineers, in which there is a central server and several client nodes that perform Machine Learning.

The main objective of this project is to realize the decentralized P2P version of this software, in which all nodes do not have any defined role.

Palabras clave (entre 4 y 8):

Federated Learning, Machine Learning, Privacy, Distributed computation, P2P

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

Machine Learning es un campo del cual, no sólo hay un inmenso desarrollo científico, sino que también levanta pasiones entre los ciudadanos de a pie, a los cuales se les inunda desde los medios de información con artículos alrededor del Machine Learning e Inteligencia Artificial, en el que se promete un futuro maravilloso gracias a estas tecnologías.

Machine Learning es la posibilidad de hacer que el ordenador adquiera un aprendizaje sobre un conjunto de datos, es decir, desarrolle un modelo automáticamente y poder con ello, hacer predicciones, clasificaciones, etc. que con programación específica no sería posible.

Federated Learning sería la habilidad de poder entrenar un modelo de Machine Learning de manera distribuida, es decir, que los datos de entrenamiento del modelo no están centralizados, con lo cual es posible hacer Machine Learning garantizando la privacidad de los usuarios que han generado esos datos, ya que estos datos no saldrán del dispositivo en el cual se generaron o actualmente residen.

La primera aparición de Federated Learning en la literatura científica viene de la mano de científicos de la empresa Google[1] en 2017 y desde entonces es un campo que está en continua investigación, pues, entre otros, se propone como una de las mayores aplicaciones que se podrán realizar dentro de la siguiente generación de redes de datos móviles de sexta generación, la 6G[2].

Respecto al desarrollo que se ha realizado, no se empezó con las manos vacías. El punto de partida fue la red de Federated Learning, el cual sigue un modelo cliente / servidor, fruto del Trabajo Final de Máster (TFM) en Ingeniería Informática de Eduardo Yáñez Parareda[3] y el cual estuvo al frente, Félix Freitag, también consultor del presente TFM.

Nótese que ya de por sí Federated Learning tiene apariencia de red distribuida, por lo que, el presente TFM busca aproximarse a la versión Peer-to-Peer (P2P) de dicha red y evaluar el comportamiento del Machine Learning en dicha situación.

1.2 Objetivos del Trabajo

Se espera extender la red Federated Learning actual de forma que lo haga más útil y aprovechable para posteriores proyectos y/o trabajos de investigación, con los siguientes objetivos generales y específicos.

1.2.1 Generales

- Adquirir conocimientos sobre Federated Learning y sobre el producto generado en un anterior Trabajo Final de Máster, la red de Federated Learning.

- Aplicar los conocimientos adquiridos en la realización del Máster, en especial en las áreas de Sistemas Distribuidos, Ingeniería del Software y Machine Learning.

1.2.2 Específicos

- Convertir el producto generado en un anterior Trabajo Final de Máster, la red de Federated Learning, en su versión P2P, al menos lo mínimo necesario para poder verificar resultados que se puedan obtener en esa situación.

- Verificar sus resultados y la utilidad o no de la versión P2P respecto al Machine Learning que éste puede generar.

1.3 Enfoque y método seguido

Dada la amplitud de este campo y el desconocimiento del proyectando sobre él, se ha decidido por una primera fase en el que se realizará un aprendizaje previo sobre Federated Learning y se estudiará el estado actual del código de la red de Federated Learning disponible.

Inicialmente se dejó en carácter abierto sobre qué se querría desarrollar en el presente TFM, precisamente porque hay varios caminos por delante. De manera natural, surgió el poder comprobar qué pasaría si la red se convirtiera en modo P2P.

La segunda parte consiste en realizar la versión P2P y comprobar sus resultados, para acabar de terminar redactando la memoria y realizar el vídeo correspondiente.

Lo cierto es que estamos ante un TFM con un componente de investigación en el que no hay un horizonte fijado y un camino a seguir, por lo que hace falta flexibilidad completa, siguiendo un sistema iterativo de realizar y presentar lo realizado a fin de que haya la suficiente dirección, con el cual conseguir un resultado tangible final.

1.4 Planificación del Trabajo

1.4.1 Recursos necesarios

Respecto a los recursos necesarios para la consecución de este proyecto, bastará con disponer de un ordenador con la suficiente memoria RAM y procesador para realizar tareas de Machine Learning. El sistema operativo, podría ser cualquiera compatible con el lenguaje de programación Python y las librerías utilizadas, por lo que cualquiera de los sistemas operativos generalistas, Windows, Linux o macOS es válido.

Por supuesto, respecto a Linux y cualquier otro tipo de ordenador que no sea uno de escritorio de consumo, como un Single Board Computer, por ejemplo Raspberry Pi, va a depender bastante de lo completo que sea el entorno Linux que estos dispositivos dispongan y de la compatibilidad de las librerías con la ISA de éstos.

El proyectando usa un ordenador portátil con un procesador Intel I7 7700HQ, el cual tiene 4 núcleos con 2 hilos por núcleo, y 16 GB de memoria RAM, usando sistema operativo Archlinux. También están disponibles para pruebas unos SBCs Alix APUs de 4GB, los cuales tienen 2 núcleos y 2 hilos por núcleo.

Como entorno de desarrollo, se usará PyCharm Community[4] de JetBrains.

1.4.2 Tareas

Las tareas y su planificación vinieron relacionadas con las entregas de la PEC establecidas al principio de la asignatura del TFM, siendo las siguientes fechas:

- PEC2: Lunes, 12 de Abril de 2021.
- PEC3: Domingo, 17 de Mayo de 2021.
- Entrega final (PEC4): Viernes, 4 de Junio de 2021.

Nótese que se empieza por la PEC2, porque la PEC1 es la definición inicial del TFM, lo que podría haberse denominado Fase 0 siguiendo el esquema propuesto.

La descripción de las tareas es la siguiente:

- 1. Fase 1:
    - Tareas:
        - 1.1 Aprendizaje sobre Federated Learning y Machine Learning: se estudiará qué es más a fondo, cómo funciona, qué parámetros de configuración entran, etc y su relación con el Machine Learning, a fin de tener una base teórica con el que poder tratar con la red Federated Learning con mayor confianza.
        - 1.2 Estudio sobre de red de Federated Learning:
            - 1.2.1 Preparar el entorno para arrancar el código base: tarea para conocer el entorno que necesita e instalar las dependencias en el sistema.

        - 1.2.2 Arrancar la red y probar con los ejemplos de su documentación: probar el tutorial disponible, ver sus resultados, etc.

        - 1.2.3 Estudiar código y librerías utilizadas: como hay que hacerle una extensión, hay que conocer cómo funciona por dentro la red.

    - 1.3 Definir posibles extensiones para hacer un desarrollo: dado los conocimientos adquiridos, se trata de poner sobre la mesa las diferentes posibilidades que puedan codificarse en el tiempo disponible.

    - 1.4 Elección de extensión a desarrollar: tarea de toma de decisión de la extensión a desarrollar.

    - 1.5 Redacción de capítulo de introducción de la memoria: tarea para redactar el entregable de la PEC2.

  - Hito: 1.4 PEC2.

  - Entregable: capítulo de introducción de la memoria del TFM.

- 2. Fase 2:

  - Tareas:

    - 2.1 Programación: tarea para codificación de la extensión.

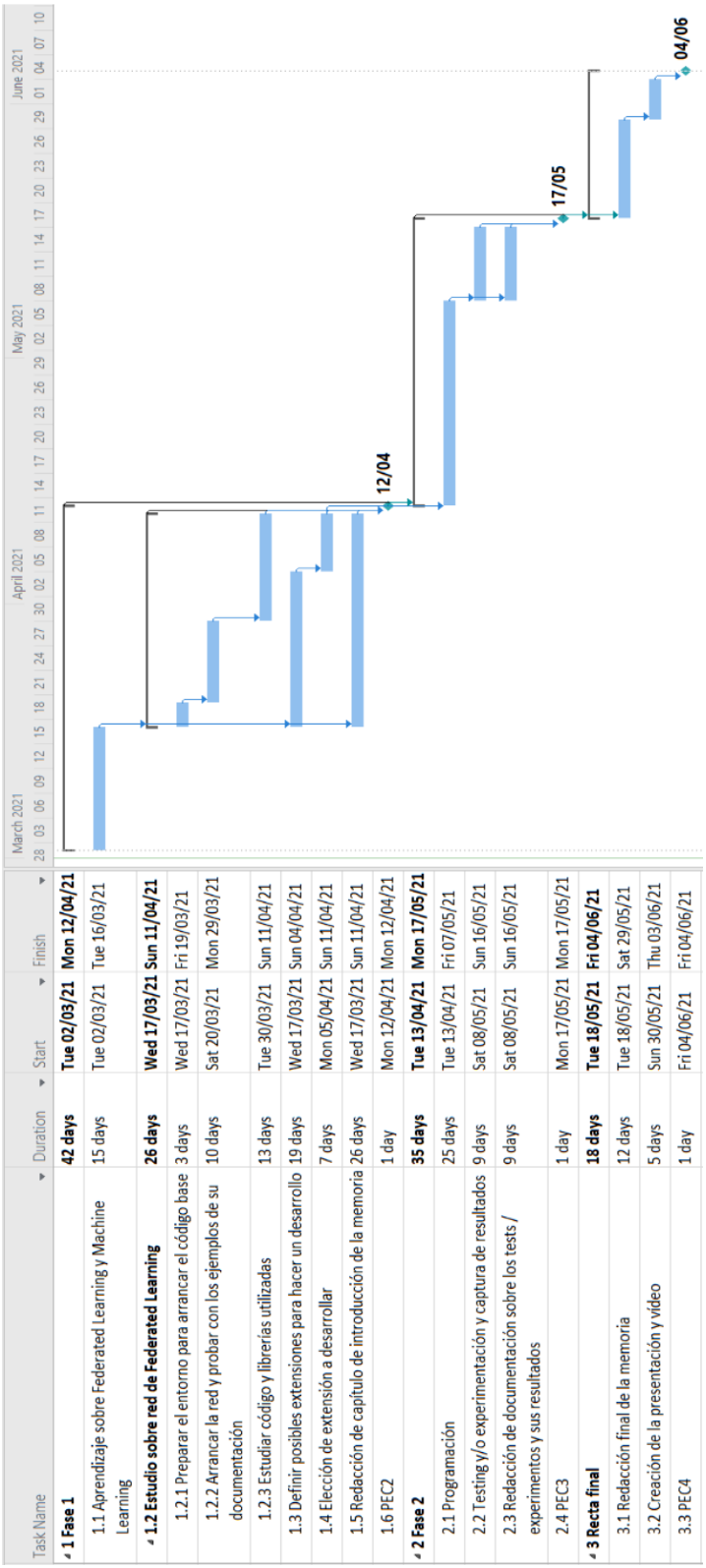
    - 2.2 Testing y/o experimentación y captura de resultados: dependiendo de lo que se codifique, finalmente habrá más testing que experimentación o viceversa. Lo comprobado, se capturará para realizar la documentación.

    - 2.3 Redacción de documentación sobre los tests / experimentos y sus resultados: tarea para redactar el entregable de la PEC3.

- Hito: 2.4 PEC3.
- Entregables: producto realizado y documentación de los test / experimentos realizados con sus resultados.

- 3. Recta final:
  - Tareas:
    - 3.1 Redacción final de la memoria: tarea para, usando los entregables anteriores, terminar de redactar la memoria final del TFM.
    - 3.2 Creación de la presentación y vídeo: creación de la presentación y el vídeo que servirán para la entrega final
  - Hito: 3.3 PEC4.
  - Entregables: memoria con presentación y vídeo.

En la siguiente figura, se puede observar el diagrama de Gantt con la planificación temporal y las tareas.



Task Name	Duration	Start	Finish
1 Fase 1	**42 days**	**Tue 02/03/21**	**Mon 12/04/21**
1.1 Aprendizaje sobre Federated Learning y Machine Learning	15 days	Tue 02/03/21	Tue 16/03/21
1.2 Estudio sobre red de Federated Learning	**26 days**	**Wed 17/03/21**	**Sun 11/04/21**
1.2.1 Preparar el entorno para arrancar el código base	3 days	Wed 17/03/21	Fri 19/03/21
1.2.2 Arrancar la red y probar con los ejemplos de su documentación	10 days	Sat 20/03/21	Mon 29/03/21
1.2.3 Estudiar código y librerías utilizadas	13 days	Tue 30/03/21	Sun 11/04/21
1.3 Definir posibles extensiones para hacer un desarrollo	19 days	Wed 17/03/21	Sun 04/04/21
1.4 Elección de extensión a desarrollar	7 days	Mon 05/04/21	Sun 11/04/21
1.5 Redacción de capítulo de introducción de la memoria	26 days	Wed 17/03/21	Sun 11/04/21
1.6 PEC2	1 day	Mon 12/04/21	Mon 12/04/21
2 Fase 2	**35 days**	**Tue 13/04/21**	**Mon 17/05/21**
2.1 Programación	25 days	Tue 13/04/21	Fri 07/05/21
2.2 Testing y/o experimentación y captura de resultados	9 days	Sat 08/05/21	Sun 16/05/21
2.3 Redacción de documentación sobre los tests / experimentos y sus resultados	9 days	Sat 08/05/21	Sun 16/05/21
2.4 PEC3	1 day	Mon 17/05/21	Mon 17/05/21
3 Recta final	**18 days**	**Tue 18/05/21**	**Fri 04/06/21**
3.1 Redacción final de la memoria	12 days	Tue 18/05/21	Sat 29/05/21
3.2 Creación de la presentación y vídeo	5 days	Sun 30/05/21	Thu 03/06/21
3.3 PEC4	1 day	Fri 04/06/21	Fri 04/06/21

Figura 1: Diagrama de Gantt con la planificación del Trabajo Final de Máster

1.5 Breve sumario de productos obtenidos

Se ha obtenido un software en el que cada ejecución de dicho software instanciaría un nodo sin rol definido en una red de Federated Learning P2P.

Todos los nodos se comunican entre todos y son capaces de entrenar modelos de Machine Learning de manera distribuida.

1.6 Breve descripción de los otros capítulos de la memoria

1. Introducción: el presente capítulo que comenta los objetivos y la planificación.

2. Federated Learning: breve capítulo comentando qué es el Federated Learning, lo importante a tener en cuenta y un breve análisis del código fuente de partida.

3. De modelo cliente / servidor a P2P: decisiones técnicas, explicación de todo lo que sucede dentro del software realizado y evaluación de resultados.

4. Trabajos relacionados: comparación del trabajo realizado contra los puntos arquitecturales esenciales de otros trabajos obtenidos de la literatura científica.

5. Conclusiones: resumen y valoración del trabajo realizado, junto a las lecciones aprendidas y líneas de trabajo futuro.

6. Glosario: selección de palabras usadas durante el texto con explicación rápida.

7. Bibliografía: referencias y citaciones a los artículos y páginas web consultados.

8. Anexos: información extra, como por ejemplo, los datos capturados para obtener la media y desviación estándar de la precisión obtenida por parte de los entrenadores en las pruebas.

2. Federated Learning

Se denomina Aprendizaje Federado, porque se conforma una “federación” poco definida de nodos que participan en un entorno determinado, aportando sus datos y sus capacidades de cómputo locales para realizar la tarea de aprendizaje, de la cual se beneficiarían o no todos los participantes.

Como se indicó brevemente en la introducción, el modelo y los datos no están en el mismo nodo de cómputo: en vez de enviar los datos al nodo donde está el modelo para qué este haga los cálculos, el estado global del modelo se manda a cada uno de los nodos donde están los datos y se realiza una computación de Machine Learning en cada uno de ellos, utilizando el conjunto local de datos que contiene cada uno, para terminar enviando sus resultados a un nodo central, el cual los mezclará para conformar un nuevo modelo.

Este envío del último modelo y recepción de resultados se hace en varias rondas. En cada ronda, los científicos e ingenieros de Google[1] han propuesto que se elijan C nodos aleatoriamente de entre el conjunto K de nodos disponibles al que enviarles los parámetros más recientes del modelo. Esto es así, porque en sus experimentos han comprobado que el valor que aportaría que todos los nodos computen es relativamente poco, por lo que no usar todos los nodos es más eficiente.

Existen una serie de problemas a tener en cuenta en esta situación:

- La presencia de datos no independientes ni distribuidos idénticamente (non-IID): es de esperar que algunos o muchos nodos locales tengan datos de un solo tipo, por lo que el entrenamiento del modelo en estos nodos estará sesgado.

- El desequilibrio: algunos nodos pueden haber generado más datos de ciertos tipos en particular que otros.

- Distribución masiva: puede haber más nodos participando en la red que la cantidad de datos promedio que hay por nodo.

- Comunicaciones limitadas: no es lo mismo la comunicación de red dentro de un centro de datos que en una red de comunicaciones móviles.

Respecto al Machine Learning en sí a realizar en los nodos locales, el algoritmo considerado es el del descenso del gradiente estocástico, que es una optimización respecto al algoritmo clásico del descenso del gradiente.

El descenso del gradiente trata con el conjunto completo de datos en una sola iteración para obtener el mínimo global de una función, que dado que estamos con Machine Learning, sería la función de pérdida del modelo, la que mide la distancia entre las predicciones debido a los parámetros del modelo y los datos reales en sí y nos indica gracias a ello, cuán bueno es el modelo para hacer futuras predicciones.

La versión estocástica es una estimación / optimización respecto a la versión no estocástica que implica realizar varias iteraciones con un lote de un dato o varios, pero no todo el conjunto, seleccionados aleatoriamente, por lo que harán falta varias iteraciones para ir acercándose a ese mínimo global. Y como algunos lectores ya sabrán, esto no siempre ocurre así y se puede acabar encontrando un mínimo local, dependiendo de la función.

2.1 Parámetros en Federated Learning

Como vemos, este algoritmo se presta a ser utilizado en entorno de Federated Learning, dado su carácter iterativo, en el que cada nodo obtendría un conjunto aleatorio de sus datos locales y hace un cálculo para “descender” hacia el mínimo global, por lo que los parámetros que harán falta ser recibido en los nodos serán:

- Los parámetros del modelo en sí, w.

- El ratio de aprendizaje, η: a menor ratio de aprendizaje, pasos más pequeños hacia el mínimo global se irán dando y más iteraciones / rondas harán falta para llegar a él. Por el contrario, un ratio de

aprendizaje excesivamente grande, podría hacer que nunca llegue al mínimo local, dado los saltos que daría (ver Figura 2).

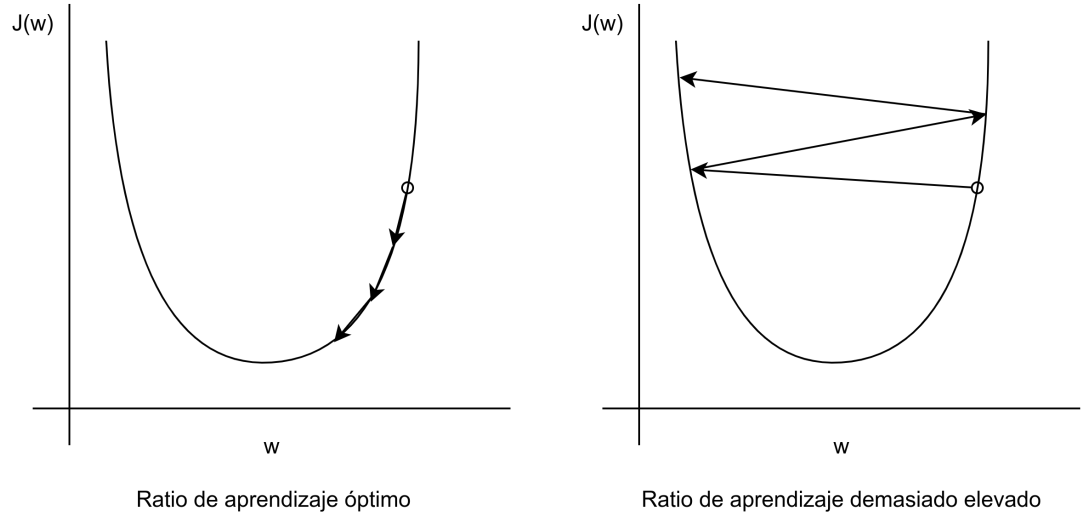


Figura 2: Efecto del ratio de aprendizaje en el camino hacia el mínimo global óptimo de la función de pérdidas

- El número de iteraciones a realizar localmente, E: esto sirve para aumentar la computación en los nodos locales antes de que éstos envíen el resultado de su computación local al nodo central, minimizando la comunicación entre nodos y evitando hacer más rondas de envío / recepción de parámetros / resultados.

- La cantidad de muestras del conjunto de datos local a tener en cuenta en cada iteración, B.

El resultado de la computación es un nuevo conjunto de parámetros del modelo, w', en el que localmente están más cerca del mínimo local dada la función de pérdida. Estos parámetros se envían al nodo central y éste realiza el promediado entre los nuevos parámetros del modelo obtenidos de cada nodo, el cual recibe el nombre de Federated Averaging (FedAvg).

Teniendo en cuenta estos parámetros, los compromisos en Federated Learning vienen por el lado de minimizar la comunicación entre los nodos y el nodo central y la cantidad de computación que cada nodo debe realizar, para poder alcanzar una determinada precisión del modelo.

A la hora de realizar comprobaciones y verificaciones, hay que tener en cuenta todos los detalles aquí nombrados para generar experimentos que verifiquen que los parámetros escogidos y el algoritmo de Federated Learning, compensan los problemas propios de este tipo de redes arriba nombrados sin desviarse en demasía.

2.2 Casos de uso de Federated Learning

Respecto a aplicaciones prácticas de Federated Learning, cualquier necesidad de hacer Machine Learning sin la posibilidad de transmitir datos originales a un tercero se verá beneficiada por éste.

Podemos imaginar casos de investigación de Machine Learning con datos de salud, que son considerados de máxima protección y es raro que se cedan libremente a terceros para su uso en investigación, como hacer predicciones de enfermedades mediante análisis de radiografías o Machine Learning de texto predictivo, generado por cada usuario en sus dispositivos móviles sin que los textos originales escritos por sus usuarios salgan de sus dispositivos móviles.

2.3 Breve análisis de la red de Federated Learning

La red de Federated Learning, realizado por Eduardo Yáñez Parareda en su Trabajo Final de Máster[3] consiste en un modelo cliente / servidor, en el que tanto el servidor como el cliente proveen una capa REST API con intercambio de mensajes en JSON para la comunicación entre ellos, mediante unas pocas operaciones para hacer la transacción de Federated Learning.

No se pretende aquí exponer todo lo realizado en dicha red, ya que para eso el Trabajo Final de Máster original es más que suficiente, para verlo en profundidad y entender cómo se ha llegado hasta ahí. Sólo se expondrán unas breves pinceladas, a fin de tenerlas en cuenta a la hora de transformarlo en una versión P2P.

Tanto el servidor como el cliente comparten los mismos conjunto de datos, en vez de que, los nodos clientes “generasen” de alguna manera sus datos

propios. Nótese que estamos ante un software académico para investigar y comprobar posibilidades, al igual que la versión P2P que se ha realizado.

Estos conjuntos de datos son los del MNIST, una base de datos etiquetada de dígitos manuscritos y Chest Ray X Pneumonia, que son radiografías de pecho ya etiquetados indicando si el paciente tiene neumonía o no.

El cliente tiene una REST API con los siguientes comandos (Punto de entrada: client/app.py):

- POST a /training: desde donde el servidor, le indica el conjunto de datos a utilizar (MNIST o Chest Ray X Pneumonia)), los parámetros del modelo actual (w), el ratio de aprendizaje (η), el número de iteraciones a realizar (E) y el tamaño del conjunto local de datos (B), tal y como vimos en la introducción sobre Federated Learning. Menos los parámetros del modelo, el resto de parámetros están fijos y dentro del propio código (server/server.py → función start_training) por lo que no son configurables en el servidor.

El servidor tiene una interfaz web, por lo que en la REST API hay comandos para la interfaz web y comandos para el cliente. Los comandos que ofrece el servidor son (Punto de entrada: server/__init__.py):

- POST a /client: para registrar un cliente en el servidor.
- DELETE a /client: para desregistrar un cliente en el servidor.
- PUT a /model_params: para que un cliente envíe los resultados de su computación local al servidor: el formato de estos resultados dependen del dataset elegido, pero lo mínimo a enviar son los nuevos parámetros del modelo, w'. En cada actualización, se hace la media de los resultados recibidos, actualizándose así el modelo global del servidor para el conjunto de datos indicado.

- POST a /training: usado por la interfaz web para empezar el entrenamiento, haciendo un POST a /training a los clientes registrados para que comiencen el entrenamiento.

Los clientes tienen varios estados: ocioso y ocupado, para indicar si están a la espera o no. Si se les trata de pedir que entrenen mientras ya están entrenando un modelo, se ignora la petición.

Los clientes hacen Machine Learning usando distintas librerías, según el conjunto de datos seleccionados: PyTorch para MNIST y Keras de Tensorflow para Chest Ray X Pneumonia.

Para terminar, no se toma una selección aleatoria de clientes C de entre el total de clientes registrados K para participar en la siguiente ronda de entrenamiento, el cual se ejecuta mediante un botón desde la interfaz de usuario (método POST de /training del servidor como se indicó arriba).

Sino que se manda a entrenar a todos los clientes registrados siempre, es decir, no contempla la optimización tomada por los científicos de Google[1] nombrada al comienzo del presente capítulo (elección de C nodos aleatorios).

3. De modelo cliente / servidor a P2P

En el presente capítulo se mostrará el desarrollo realizado, cambios, correcciones, mejoras y pruebas realizadas.

El objetivo primario es realizar la versión P2P de la red de Federated Learning, partiendo de un código que actúa en modo cliente / servidor.

Es decir, en vez de que un servidor central haga las operaciones finales y sea sólo ese nodo quién contenga el modelo promediado, todos los nodos harán todas las operaciones del Federated Learning incluyendo Federated Averaging y dispondrán de su modelo promediado localmente.

3.1 Detalles y decisiones tomadas para el desarrollo

Antes y durante el desarrollo, se han tomado una serie de decisiones que afectan al diseño final, las cuales discutiremos aquí.

3.1.1 El paso a modelo a P2P

Al comienzo, a la hora de afrontar el desarrollo, se pensó en lo que ocurre en otros sistemas de P2P. El primer paso sería el del “bootstrapping”, es decir, cómo un nodo entra por primera vez en una red P2P.

El código original basa su capa de comunicación en una interfaz REST API con intercambio de mensajes en formato JSON. Con el tiempo disponible, no se va a innovar cambiando esa capa por otra más avanzada, por ejemplo, basada en otros sistemas de red superpuesta P2P ya conocidos, como BitTorrent.

Teniendo en cuenta todo esto, se decide hacer que los nodos sean capaces de conectarse a una lista de nodos inicial suministrada manualmente, y además, en este “bootstrap” inicial, los nodos que participan deben intercambiarse los nodos que conocen cada uno.

Con este intercambio, los nuevos nodos se anunciarán al resto de nodos de la red P2P, los cuales a su vez comparten qué nodos conocen, que el nodo que se anuncia desconoce.

Cuando ya no haya más nodos a los que anunciarse, se acaba este proceso y se llega a un punto en el que todos los nodos se conocen entre todos.

Otro detalle, a tener en cuenta y es que sería muy normal, que los nodos no serán de características homogéneas, es decir, algunos pueden tener más potencia de procesador, más núcleos, más hilos por núcleo o más memoria RAM.

En este sentido, se puede esperar que unos nodos tarden más que otros en hacer su entrenamiento debido a esto, por lo que es importante que el protocolo tenga en cuenta este detalle. Queda para trabajo futuro el desarrollar este aspecto más a fondo respecto al entrenamiento en sí.

A su vez, cada nodo tendrá su modelo local, es decir, cada nodo recibirá del resto de nodos, el modelo al final de la ronda de entrenamiento que obtuvieron y en algún momento se hará el Federated Averaging, en cada uno de ellos, de forma local, en cada nodo.

Es por ello, que hay que contemplar la posibilidad de que pueda ocurrir que los nodos en un modelo P2P, puedan caerse y volver más tarde. La regla que se sigue es que cada nodo hará Federated Averaging siempre y cuando se hayan recibido los modelos de todos los nodos conocidos por parte del nodo. Cuando no se reciben todos, un temporizador fuerza el Federated Averaging con los modelos que hayan llegado.

También se forzará el Federated Averaging inmediatamente si llega una petición de hacer la siguiente ronda de entrenamiento.

3.1.2 Salvaguarda usando las precisiones de los modelos

Se ha propuesto una pequeña innovación. Al finalizar el Federated Averaging, se calcula la precisión del modelo y se hacen intercambios de estos modelos con el resto de nodos. Cada nodo decidirá si usar el modelo que les

ha llegado o no, obteniendo la precisión contra el conjunto de datos local y comparándola con la de su modelo local.

Hay que tener en cuenta lo siguiente: todos los nodos se intercambian sus modelos y todos ellos hacen Federated Averaging, por lo que no se esperan divergencias en cada nodo, siempre y cuando todos los nodos hagan la misma manera de hacer el entrenamiento y además, siempre obtengan los modelos de todos los nodos para hacer el Federated Averaging.

Es decir, esa innovación habría que verla como una salvaguarda para casos extremos, como que ocurriera una partición de red y una serie de nodos aislados tomaran otro rumbo respecto al resto.

3.1.3 La importancia de la obtención de los conjuntos de datos

Debido a lo nombrado anteriormente, se hizo importante valorar cómo se estaban obteniendo los conjuntos de datos para los distintos cálculos. En un principio, se tenía conjunto de datos para el entrenamiento y conjunto de datos de validación. Las librerías utilizan el dataset de validación justamente para afinar los parámetros del modelo durante el entrenamiento.

Es por ello, que no es recomendable usarlos también para evaluar el modelo en sí después de acabar el entrenamiento[5]. Se hizo necesario pues, obtener un conjunto de datos de test para obtener las precisiones, que serán utilizadas para determinar si hacer intercambio de modelos o no.

En el código original, los conjuntos de datos se obtienen de manera aleatoria.

Se ha decidido no alterar esta manera de obtener los datos que alimentan los entrenadores. Los datos de test se obtendrán extraiéndolos del conjunto de datos de entrenamiento. Podemos hablar entonces, que ningún nodo en este modelo P2P tiene memoria respecto a los datos y que pueden aparecer y desaparecer muestras utilizadas en nodos anteriores (caso non-IID).

Esto no es impedimento alguno para hacer un entrenamiento: es como si en cada ronda, los nodos fueran distintos, pero en una situación real, hay que

tener en cuenta que los nodos mantendrían su conjunto de datos local siempre. No desechan su conocimiento acumulado.

3.1.4 Mejorando la arquitectura del software

Respecto al desarrollo en sí, partimos de dos aplicaciones separadas realmente, aplicación cliente y aplicación servidor en que además no había una jerarquía ni arquitectura visibles en el sistema de archivos: todo eran archivos .py en la misma carpeta.

Se decidió hacer una refactorización intensiva, pues el autor del presente TFM sigue la regla básica propia del movimiento “Scout”: “Deje siempre el campamento base más limpio de cómo se lo encontró”. De esta manera, en la mezcla de las dos aplicaciones, la aplicación cliente y la aplicación servidor, se hará evidente la aplicación de una arquitectura “normalizada”, tipo Modelo-Vista-Controlador.

Dicha arquitectura ya existía como tal, pero no era visible. Además, si el objetivo es tener un “framework”, hay que resolver un detalle detectado, también nombrado en su TFM por parte del autor del código original, y es que había que retocar partes del código interna a la hora de añadir un nuevo entrenador al sistema.

3.1.5 Concurrencia y paralelismo

También se tuvo en cuenta, que hay que tener cuidado con la concurrencia y el paralelismo. En un principio, sólo se utilizaba la librería asyncio[6], que son corutinas para Python.

Las corutinas son rutinas cooperativas, es decir, dentro de las propias rutinas se “cede” voluntariamente (con la palabra reservada: “await”) la ocupación del hilo de ejecución cuando se necesita esperar por entrada / salida. Por lo tanto, con un solo hilo de ejecución, todo aquel código que no use CPU, pero que tenga que esperar por, por ejemplo, un paquete de red, permitiría a otras corutinas trabajar concurrentemente. Para que esto funcione, se utiliza lo que se denomina un “bucle de eventos”, el cual irán ejecutando

todas las corutinas que están en espera de una a una cada vez que se cede el puesto.

En cambio, todo código que use mucha CPU no puede ni debe usar esas corutinas, pues ocuparían el hilo manejado por el bucle de eventos, impidiéndole actuar correctamente. Ambos casos están presentes en el código: el entrenamiento usa mucha CPU, sin embargo, el servidor web puede perfectamente manejarse con corutinas debido a que es básicamente entrada / salida de red.

Concretamente, se cambió de servidor web Flask a Quart/Hypercorn[7][8], que usa corutinas completamente. Y para todo código que utiliza CPU intensivamente, se lanzó ese código en un pool de hilos con 1 solo worker, pues no tiene sentido lanzar más de una operación costosa contra la CPU a la vez por la parte de la app en sí.

Las librerías de Machine Learning son multi-núcleo también, por lo que harán presión sobre la CPU. En Python, no se encontró maneras de designar prioridades respecto a la multi-tarea, por lo que queda como línea de desarrollo futuro el experimentar para resolver mejor estos aspectos.

3.1.6 Imagen de Docker con permisos non-root

Docker[9] faciliita mucho la vida del desarrollador al no tener que instalar en su sistema todo lo que requiere una aplicación para funcionar. Permite crear una imagen con la aplicación y el entorno de sistema operativo que necesita.

El Dockerfile es el archivo de configuración que permite montar la aplicación como una imagen para Docker. Dicha imagen se usa como base para ser ejecutado en uno o varios contenedores, tantos como se desee: como máquinas virtuales, pero más ligeros, casi como si fueran procesos normales dentro del sistema operativo, pero aislados del resto.

Se revisó también qué ocurría dentro del Dockerfile ya disponible.

Originalmente el archivo creaba una imagen que requería permisos de administrador (“root”) internamente, sin embargo, se preparó para que use

permisos de usuario base (“non-root”). Esto es una buena práctica de seguridad y además, permitiría a dicha imagen poder ejecutarse en sistemas como OpenShift de Red Hat[10].

3.2 Arquitectura

La arquitectura propuesta consiste en una serie de nodos que se comunican entre ellos, por lo que estamos ante una “malla completa” de nodos (ver Figura 3). Cada nodo tendría arrancada una sola copia del software.

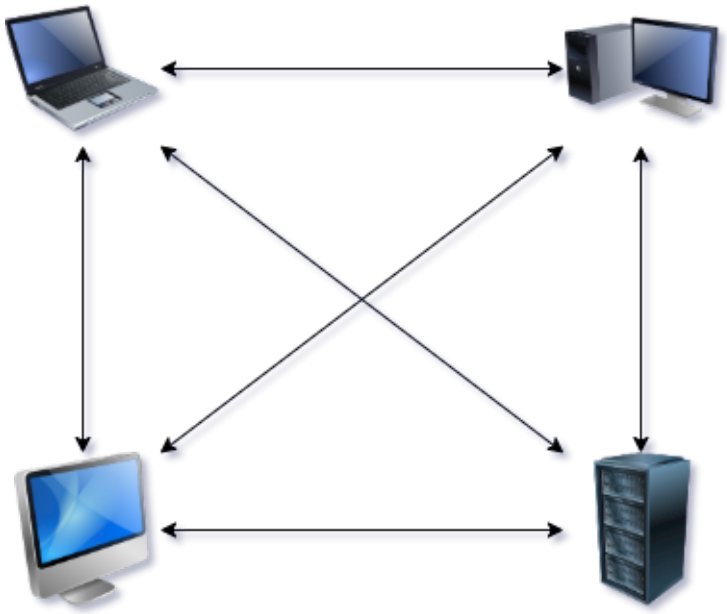


Figura 3: Malla completa de nodos

Como ya se nombró al principio, los nodos pueden ser cualquier ordenador mientras sea compatible con Python 3.8 y las librerías de Machine Learning (pyTorch, Tensorflow, Keras, FastAI) usados en los entrenadores.

No existe ningún rol particular, pero se mantuvo del diseño original, que la iniciación de una ronda de entrenamiento se realice manualmente por parte de un usuario.

3.2.1 Un primer vistazo al código

He aquí una breve exposición de cómo el código está organizado. En la raíz tenemos:

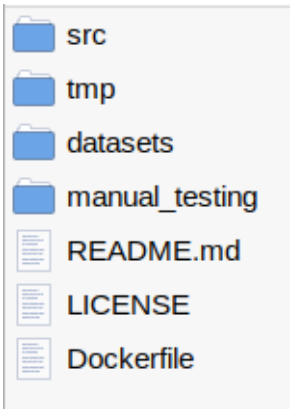


Figura 4: La raíz del código del proyecto

- El directorio “src”: contiene el código del proyecto en sí, hecho en Python 3.8.

- El directorio “tmp”: es usado por el proyecto para guardar temporalmente extractos de los conjuntos de datos de los entrenadores. En concreto y por ahora, sólo del entrenador para “Chest X Ray Pneumonia”.

- El directorio “datasets”: es usado para meter manualmente los conjuntos de datos para el entrenamiento. Por ahora, sólo usado por el entrenador de “Chest X Ray Pneumonia”. El resto de entrenadores descargan por sí solos el conjunto de datos que necesitan.

- El directorio “manual testing”: contiene algunos scripts para Linux que fueron útiles para hacer pruebas de forma fácil y rápida. Tiene un script para arrancar un nodo local (“run_local_node.sh”) y un script para arrancar tantos terminales como se le pida arrancando un nodo local en cada terminal (“open_node.sh”). Estos scripts están ajustados al sistema de desarrollo usado y son difíciles de generalizar, pero se compartieron pues igual pueden ser útiles para futuros desarrolladores.

Aparte tenemos el Dockerfile, que como ya se indicó sirve para crear la imagen de Docker, un README con el manual de usuario indicando cómo arrancar el código y la licencia del código.

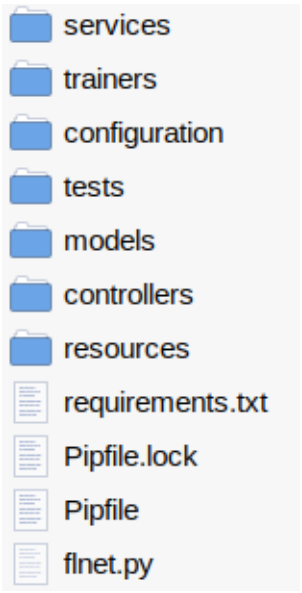


Figura 5: El código dentro del directorio "src"

Ya dentro del código, directorio “src”, se puede observar la arquitectura “Modelo-Vista-Controlador”:

- El directorio “resources”: contiene las vistas, la capa de presentación / interfaz gráfica en forma de HTML y el resto de recursos estáticos asociados.

- El directorio “controllers”: contiene dos controladores, uno con las operaciones de la capa REST API para el manejo del protocolo P2P y otro con las operaciones de la capa REST API para su uso por parte de las vistas, la interfaz gráfica.

- El directorio “models”: contiene los modelos, los cuales son los que contienen el estado de la aplicación.

- El directorio “services”: contiene los servicios, usados por los controladores para extraer información, realizar acciones, etc. contra los

modelos, es decir, son los que contienen la mayor parte de la lógica de negocio.

Aparte tenemos:

- El directorio “tests”: contiene los tests unitarios. Éstos comprueban los entrenadores y algunas operaciones de uno de los servicios, el servicio de nodo.

- El directorio “configuration”: aquí están todos los parámetros de configuración, el código inicial que configura la app al comienzo y constantes utilizadas a lo largo del proyecto.

- El directorio “trainers”: contiene los entrenadores de Machine Learning de la aplicación y es en el módulo principal __init__.py dentro de ese directorio, donde los registraríamos para que el resto de la aplicación sepa que existen.

- Los archivos “requirements.txt”, “Pipfile.lock” y “Pipfile”: todos ellos tienen una cierta relación y es que se utilizan para el control de versiones de las dependencias de la aplicación, las cuales se pueden instalar tanto con Pip[11] como con Pipenv[12], herramientas muy útiles para entornos Python. En el archivo README anteriormente nombrado está cómo utilizarlos.

- El archivo “flnet.py” es el punto de entrada de la aplicación, es decir, es el archivo principal que permite arrancar el nodo.

En la Figura 6, se puede ver el diagrama de clases. Se puede observar que no solo hay clases, ya que la aplicación no es totalmente orientada a objetos: hay partes que utilizan programación imperativa.

La figura sólo ofrece una visión muy por encima de lo que está ocurriendo, por lo que vamos a desglosar las partes esenciales del código en siguientes apartados.

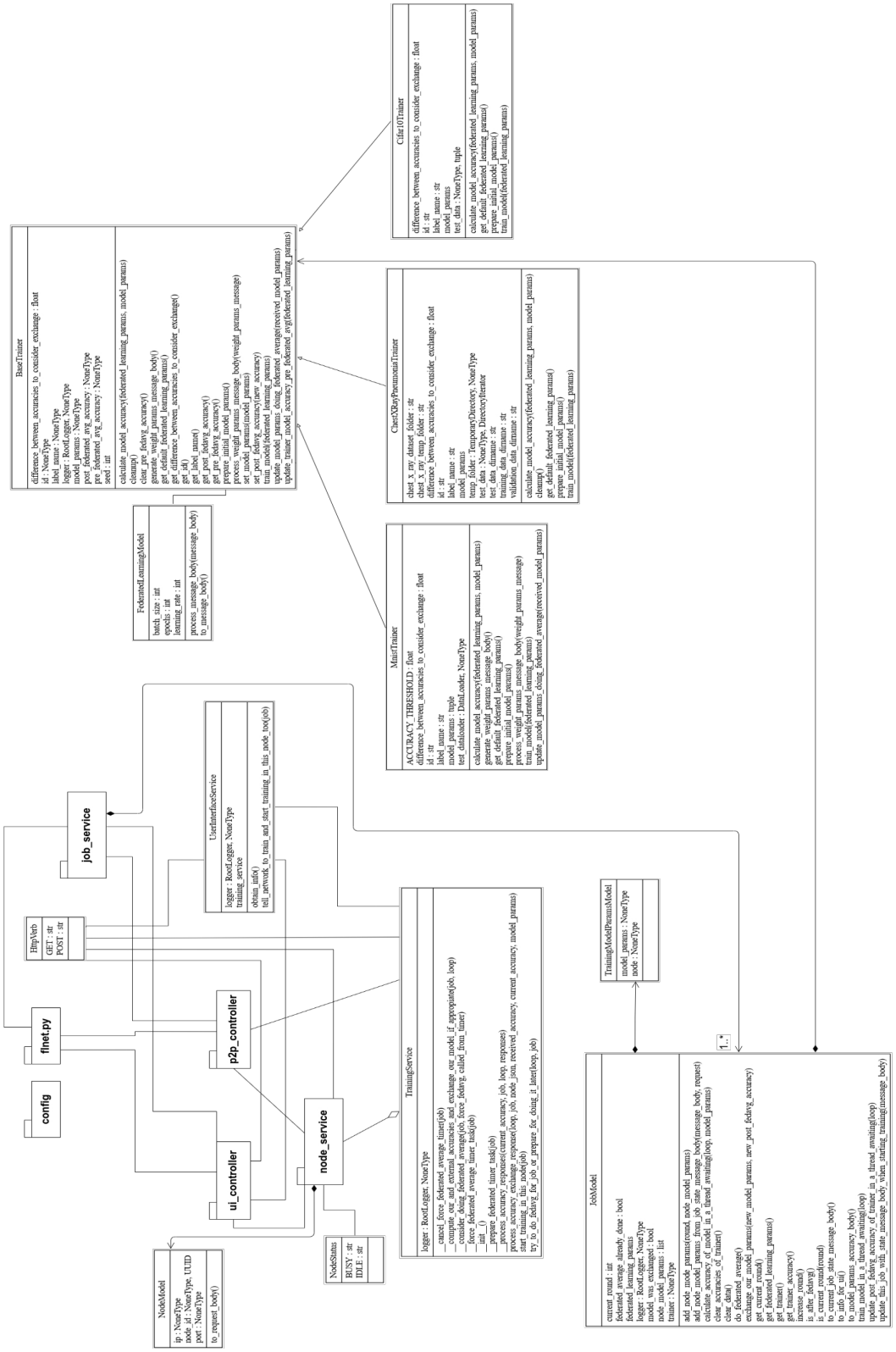


Figura 6: Diagrama de clases de la aplicación

3.2.2 El modelo de nodo

Contiene la información de un nodo, ya sea la información del nodo propio como la del resto de nodos.

Esta información es el identificador de nodo, la dirección IP y el puerto.

La dirección IP y el puerto sirven para identificar unívocamente un nodo, siendo el identificador sólo una etiqueta identificativa.

Hay un parámetro de arranque de la aplicación para ponerle nombre al nodo. Si no se indica, se generará un UUID.

Para facilitar la comunicación entre nodos, se decidió obtener la dirección IP directamente de las peticiones que llegan. Es decir, los nodos cuando mandan su información sólo mandan su identificador de nodo y el puerto. Cuando el nodo recibe la petición, en su copia local de la información de nodo mantenida en memoria, se rellenará la dirección IP.

3.2.3 El modelo de Job

Este modelo es el que tiene el mayor peso de la aplicación. El modelo de “job” (ver Figura 7) se encarga de mantener el estado respecto a un entrenador de Machine Learning específico y provee operaciones genéricas respecto a éstos.

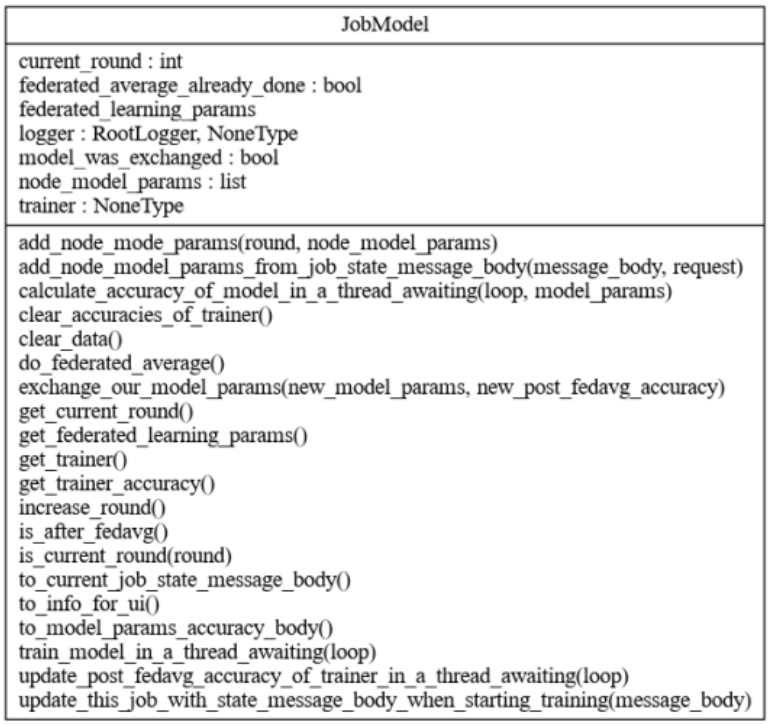


Figura 7: El modelo de Job

Un job está asociado con uno y sólo un entrenador de Machine Learning disponible en la aplicación. En el proyecto hay 3 entrenadores implementados, por lo que se instanciarán 3 jobs al comienzo de la aplicación, uno para cada uno de esos 3 entrenadores. Este instanciamiento ocurre en el servicio de Job (services/job_service.py) en su método “prepare_jobs_at_startup”.

El job a la hora de instanciarse, no solo instancia el entrenador asociado sino que obtiene los parámetros de Federated Learning por defecto del entrenador. No se pudo hacer por falta de tiempo, pero ésto abriría la puerta a que estos párametros sean configurables fácilmente desde la interfaz de usuario.

De los datos que contiene el job, dos de ellos son los más importantes:

- La ronda actual en la que se encuentra el entrenador (“current_round”), siendo la ronda “-1”, la ronda actual, es decir, “-1” indicaría que todavía no se ha empezado a usar ese entrenador.

- La lista de modelos de pesos recibidos por parte de los otros nodos en la ronda actual (“node_model_params”).

El modelo de Job contiene varias operaciones pero podemos destacar las siguientes:

- “add_node_model_params_from_job_state_message_body”: cuando un nodo manda su nuevos pesos a un nodo, el nodo acaba llamando a este método para procesar el mensaje que llegó de la REST API, extraer los pesos dentro de él e incluirlos a su lista de pesos recibidos dentro del job.

  Se hace un control de seguridad: si los pesos se mandaron para una ronda inferior a la actual, se descartan; si es para una ronda superior, se descartan los datos actuales del nodo, ya que significa que la red está más avanzada que el nodo actual.

- “update_this_job_with_state_message_body_when_starting_training”: este método se arranca siempre antes de empezar una nueva ronda de entrenamiento. Sirve para actualizar el job en memoria, con los datos del job que se recibieron del nodo que inició la ronda.

  Aquí hay una decisión de diseño importante: si y solo si es la primera ronda, se utiliza el modelo promediado del nodo que inició la ronda de entrenamiento. Esto forma parte de la inicialización común para todos los nodos.

  El Federated Learning tradicional postula que se debe utilizar siempre el modelo promediado del nodo servidor central en todas las rondas.

  En la práctica, tengamos en cuenta que si todos los nodos reciben los modelos de los demás y hacen el promediado cada uno, deberían todos llegar al mismo modelo, pues todos tienen la misma información (recordemos que es una “malla completa”), por lo que no hay necesidad de siempre inicializar cada ronda con un modelo común.

  En cualquier caso, para futuros desarrollos y pruebas, con eliminar el control de ronda en este método, se consigue que siempre se utilice el modelo remoto antes de hacer la siguiente ronda de entrenamiento.

- “do_federated_average”: mientras haya al menos un modelo de pesos recibidos de otro nodo, lo que hará es entregarle al entrenador los pesos de todos los nodos recibidos para que haga el Federated Averaging.

  La razón por la que el entrenador hace esta operación y no el propio job, es porque los modelos de pesos pueden tener alguna forma particular, como es el caso del entrenador MNIST, el cual no es un vector de vectores solamente como los otros, si no que está dividido en dos partes, modelo y sesgo.

- “train_model_in_a_thread_awaiting”: método que inicia una ronda de entrenamiento pidiendo al entrenador entrenar usando el pool de hilos de la app y cuando éste acaba, también usando el pool de hilos, pide al entrenador que obtenga la precisión del modelo contra el dataset de test.

- “calculate_accuracy_of_model_in_a_thread_awaiting”: dado un modelo de pesos como argumento del método, pedir al entrenador que compruebe su precisión contra el dataset de datos, y haciéndolo dentro del pool de hilos.

- “update_post_fedavg_accuracy_of_trainer_in_a_thread_awaiting”: usando el método anterior, calcular la precisión del modelo que contiene el entrenador del job y guardarla en la variable “post_fedavg_accuracy” del entrenador.

Se debe destacar que los jobs están desacoplados totalmente de las implementaciones concretas de los entrenadores a través del uso de polimorfismo.

En el directorio “trainers”, tenemos una clase llamada BaseTrainer, que contiene un “contrato”, que deben cumplir todas las implementaciones de los entrenadores, para que los jobs sepan cómo usarlos de manera genérica.

3.2.4 Servicios más importantes

De los 4 servicios que hay, dos de ellos, el servicio de nodo y el servicio de entrenamiento, son el eje principal de la aplicación, de los cuales destacaremos algunos detalles.

3.2.4.1 El servicio de nodo

Este servicio guarda la lista de nodos de la red P2P que conoce el nodo y el estado del nodo, ocioso u ocupado.

Y entre otros, tiene dos responsabilidades importantes:

- Sirve para realizar el "bootstrapping" inicial, es decir, el anunciarse a los nodos que se le indique a la aplicación por línea de comandos (función "bootstrap_to").

- Contiene una función denominada "broadcast_message", que sirve para mandar un mensaje de manera genérica por la capa REST API a todos los nodos conocidos desde diversas partes de la aplicación, es decir, el resto de la aplicación está casi totalmente abstraído de cómo comunicarse con los nodos. "Casi", porque indicar el verbo HTTP de la operación, la ruta y el JSON del cuerpo del mensaje, argumentos de esta función, son un detalle de implementación de la capa REST API.

Este es uno de los lugares en los que se usa programación imperativa, en vez de programación orientada a objetos.

Se ha hecho así para aprovechar una característica de Python, por el cual se puede disponer de variables globales y operaciones contra esas variables, como si de el patrón "Singleton" dentro de una clase se tratase.

Hacer esto mismo con programación orientada a objetos en Python es algo más complicado[13] y ésta es la solución más sencilla de todas.

3.2.4.2 El servicio de entrenamiento

El servicio de entrenamiento (ver Figura 8) es el que contiene las operaciones que permiten coordinar tanto el entrenamiento cómo la obtención

de precisiones, además de tomar la decisión respecto a intercambiar modelos o no en base a las precisiones de los modelos remotos.

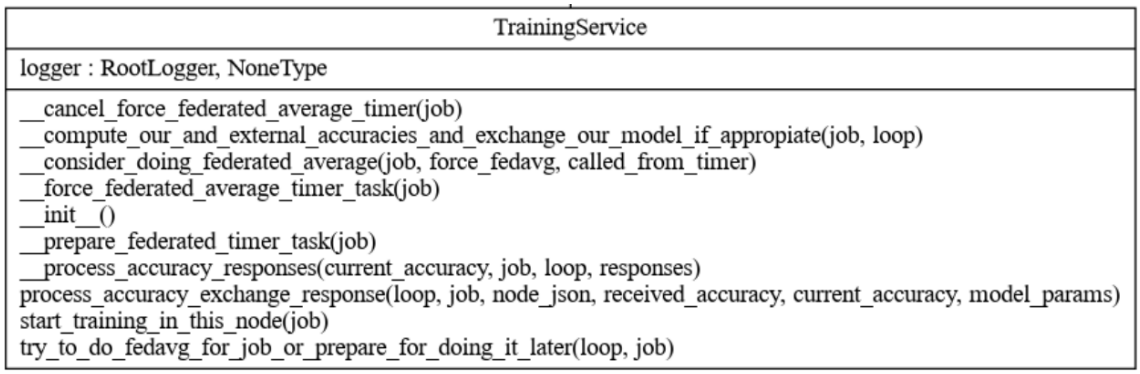


Figura 8: El servicio de entrenamiento

Sus dos métodos principales son:

- “start_training_in_this_node”: método que provoca una nueva ronda de entrenamiento en el nodo. En la Figura 9 se muestra el algoritmo que se sigue:
    - Si el nodo no está ocioso, se cancela.

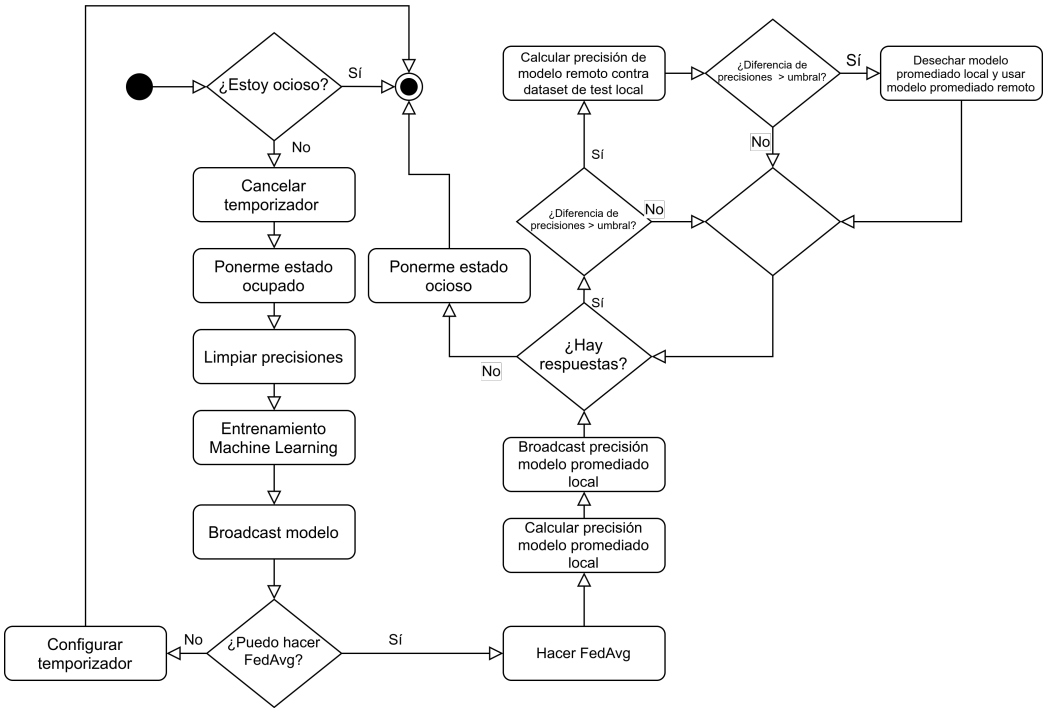


Figura 9: Diagrama de flujo mostrando el algoritmo completo de una ronda de entrenamiento

- Si el nodo está ocioso, se comienza el algoritmo cancelando el temporizador para provocar Federated Averaging si lo hubiera (método __cancel_force_federated_average_timer).

- Se cambia el estado del nodo a ocupado y se limpian las precisiones obtenidas en la ronda anterior.

- Se le pide al job asociado a la petición de nueva ronda actual a entrenar en un hilo, quedándose el servicio a la espera.

- Al finalizar el entrenamiento, se hace un broadcast al resto de nodos del modelo entrenado para que lo guarden en su lista de nodos para la ronda .

- Se mira si se cumplen las condiciones para hacer Federated Averaging en el nodo para obtener el modelo promediado (método “try_to_do_fedavg_for_job_or_prepare_for_doing_it_later”).

- La condición para que ocurra el Federated Averaging de manera natural es que se haya recibido tantos modelos de pesos como nodos se conozcan.

- Si no se puede, se prepara el temporizador para que éste ocurra de manera forzosa dentro de un tiempo.

  Este tiempo está indicado por las constantes WAIT_TIME_IN_SECONDS_TO_FORCE_FEDAVG, al que se le suma un tiempo aleatorio entre 0 y el valor de la constante WAIT_TIME_IN_SECONDS_TO_FORCE_FEDAVG_MARGIN, ambos disponibles en el módulo de configuración.

  Esta aleatoriedad hará que haga menos probable que los nodos coincidan haciendo su Federated Averaging, de forma que algunos nodos tengan más oportunidad de obtener modelos de otros, ya que en el momento que el Federated Averaging ocurra, ya no se aceptarán más modelos en el nodo.

- Si el Federated Averaging ocurre, ya sea porque se cumplían las condiciones o de manera forzosa desde el temporizador, se mandará a calcular la precisión del modelo promediado contra el data set de datos y ocurrirá el algoritmo de intercambio de modelos (método "__compute_our_and_external_accuracies_and_exchange_our_model_if_appropiate"):

  - Se hace un broadcast del modelo promediado y su precisión local al resto de nodos.

  - Cada nodo que haya acabado de hacer Federated Averaging, responderá con su modelo promediado y su precisión local.

  - Entonces, ambos lados, obtendrán la diferencia entre la precisión que le ha llegado remotamente y la precisión local que tenían (métodos "__process_accuracy_responses" y "process_accuracy_exchange_response").

    Si supera el umbral definido en la variable "difference_between_accuracies_to_consider_exchange" del entrenador del job asociado a esta operación, se procede a calcular la precisión del modelo remoto contra el conjunto de datos local.

    Una vez más, se vuelve a calcular la diferencia entre la precisión local que se tenía y la precisión del modelo remote que se obtiene contra el conjunto de datos local. Si consigue superar el umbral, entonces se intercambia el modelo. Si no, se ignora el modelo promediado remoto.

    El umbral definido lo que evita es estar haciendo cálculos de precisión innecesariamente cuando no se espera una mejora sustancial respecto a lo que ya se tiene localmente.

- "process_accuracy_exchange_response": es el encargado de hacer el algoritmo de intercambio de modelos en sí mismo.

Este método, que nombramos anteriormente, se usa tanto internamente en el servicio de entrenamiento, para procesar los modelos y precisiones obtenidos en el broadcast de precisiones, como por parte del controlador que contiene la interfaz de comunicación P2P, cuando llega un modelo remoto con su precisión.

3.2.5 Los entrenadores

Del punto de partida, se heredó un entrenador para el dataset de MNIST[14] (ver Figura 10) y otro para el Chest X Ray Pneumonia[15] (ver Figura 11).

Figura 10: Muestras del dataset de MNIST. La aplicación entrena un modelo que sólo distingue entre los tres y los sietes.

Con el fin de probar toda la abstracción realizada en el código para hacer que éste trabaje de manera agnóstica respecto a las implementaciones de los entrenadores, se propuso añadir otro.

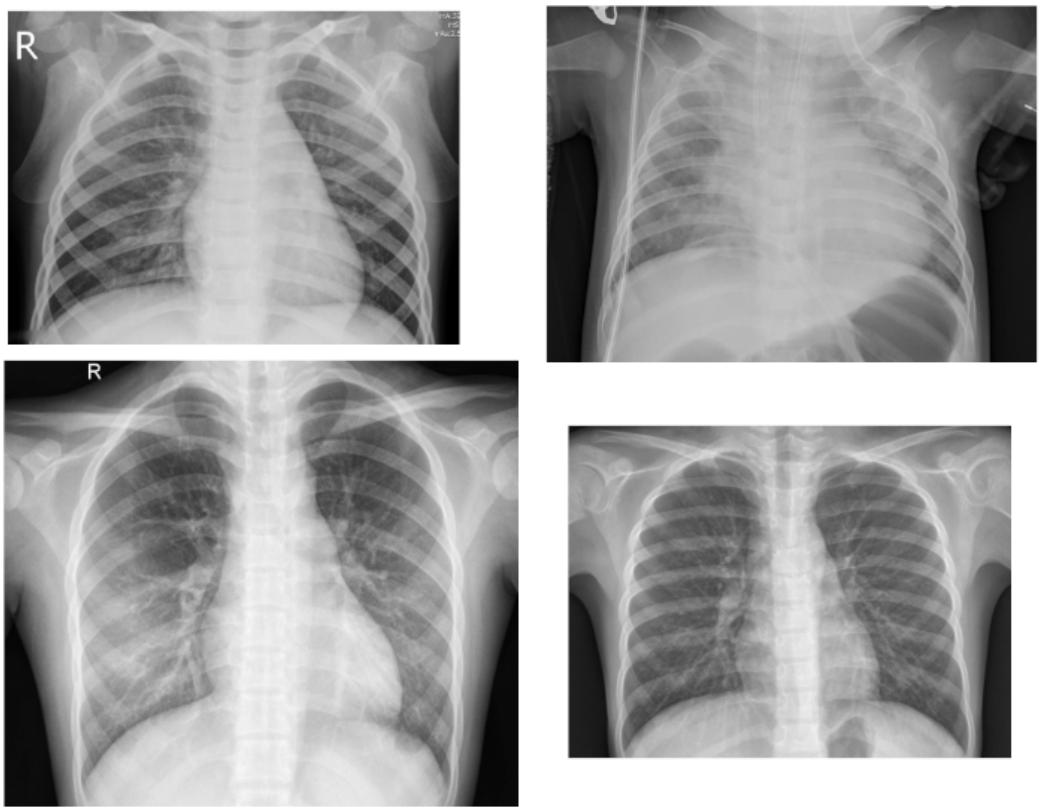


Figura 11: Muestras del dataset de Chest X Ray Pneumonia

Se eligió el dataset CIFAR-10[16], usando como base un notebook de Kaggle[17]. Se puede ver en la Figura 12 un ejemplo de las imágenes y etiquetas que contiene.

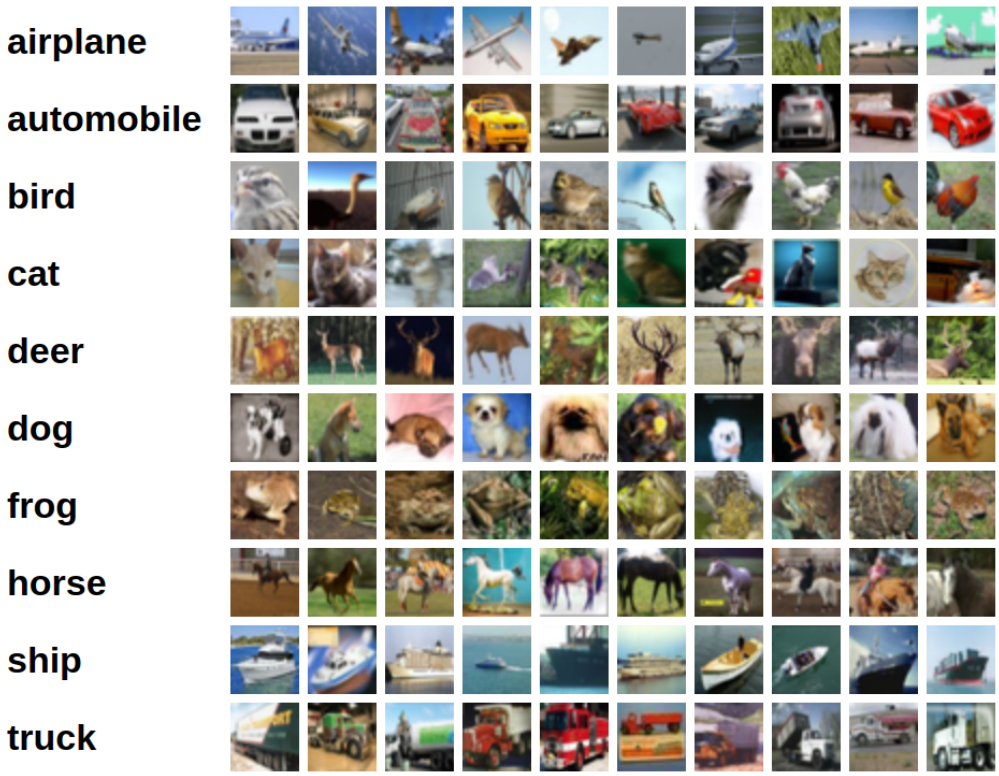


Figura 12: Muestras del dataset CIFAR-10 con su etiquetado

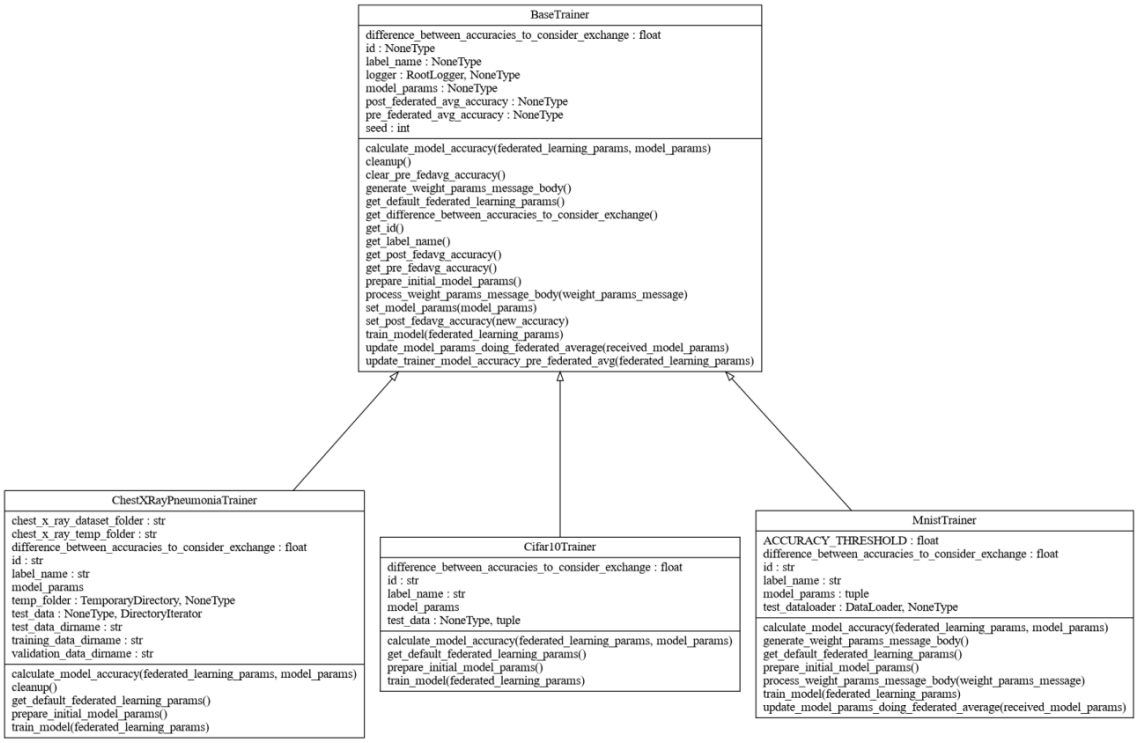


Figura 13: Los tres entrenadores con su herencia de la clase BaseTrainer

Como ya hemos comentado, los jobs no conocen nada concreto respecto a los entrenadores. Todos los entrenadores tienen que heredar de la clase “BaseTrainer” y sobrecargar los métodos necesarios para cumplir con su función.

Hay una serie de métodos que son obligatorios sobrecargarlos. Se pueden identificar porque en el BaseTrainer lanzan una excepción. Y otros, que opcionalmente se pueden sobrecargar si la implementación concreta del entrenador lo necesita.

Es el caso del entrenador MNIST, cuyo modelo de pesos, está dividido en 2 partes: modelo y sesgo, por lo que, por ejemplo, el método “update_model_params_doing_federated_average”, que sí está implementado en el BaseTrainer, hubo que sobrecargarlo para poder soportar ese caso de uso concreto.

Como se puede observar en la clase “BaseTrainer”, los entrenadores no sólo hacen el entrenamiento (método “train_model”):

- Se encargan de almacenar las precisiones antes y después del Federated Averaging.

- Tienen una configuración de Federated Learning a aplicar por defecto y tienen la configuración del umbral definido para el intercambio de modelos.

- Se responsabilizan de hacer:

    - El cálculo de precisiones para el modelo de pesos que se le indique (método “calculate_model_accuracy”).

    - El Federated Averaging con los modelos que les pase el Job (método “update_model_params_doing_federated_average”).

    - Generar y procesar los mensajes con el modelo a usar en la capa de comunicación (métodos “generate_weight_params_message_body” y “process_weight_params_message_body”).

Los entrenadores son autocontenidos, en el sentido que son ellos quienes tienen que manejar su conjunto de datos internamente. MNIST y CIFAR-10 lo descargan de Internet al principio, pero Chest X Ray depende de que el usuario coloque el dataset manualmente en el directorio “datasets” antes de empezar a usar la aplicación.

Para finalizar, aparte de implementar el entrenador heredando de la clase “BaseTrainer”, hay que modificar el archivo “__init__.py” del directorio “trainers”, para registrarlo ahí. Simplemente hay que instanciar la nueva implementación del entrenador y añadirlo al array “registered_trainers”.

En Python, es posible generar un sistema de plugins, es decir, carga dinámica de módulos, por lo que hay maneras de evitar hacer este último registro manual en desarrollos futuros.

3.3 El protocolo de comunicación

Hay dos situaciones en las que ocurren intercambios de mensajes por parte del protocolo: el “bootstrap” inicial y el entrenamiento en sí.

Para el “bootstrap” inicial, tenemos sólo una interfaz:

- POST a /announce: cuando un nodo quiere anunciarse a otro, utiliza esta interfaz. No sólo envía su información de nodo, sino los nodos que éste conoce. El nodo destino que recibe esta petición, mirará qué otros nodos el nodo que envió la petición no conoce y los devolverá en la respuesta.

  Con los nodos desconocidos recibidos, el nodo se anunciará a ellos, volviéndose a repetir este algoritmo continuamente, hasta que no haya más nodos a los que anunciarse.

  - Ejemplo de cuerpo de mensaje en esta operación:

    {'iAm': {'id': 'NODE_3', 'ip': None, 'port': '4002'}, 'myKnownPeers': [{'id': 'NODE_1', 'ip': '127.0.0.1', 'port': '4000'}]}

  - Ejemplo de respuesta con estado HTTP CREATED:

    {'announceYourselfToPeers': [{'id': 'NODE_2', 'ip': '127.0.0.1', 'port': '4001'}], 'iAm': {'id': 'NODE_4', 'ip': None, 'port': '4003'}}

En la Figura 14, podemos ver una posible situación de anuncio de nodos.

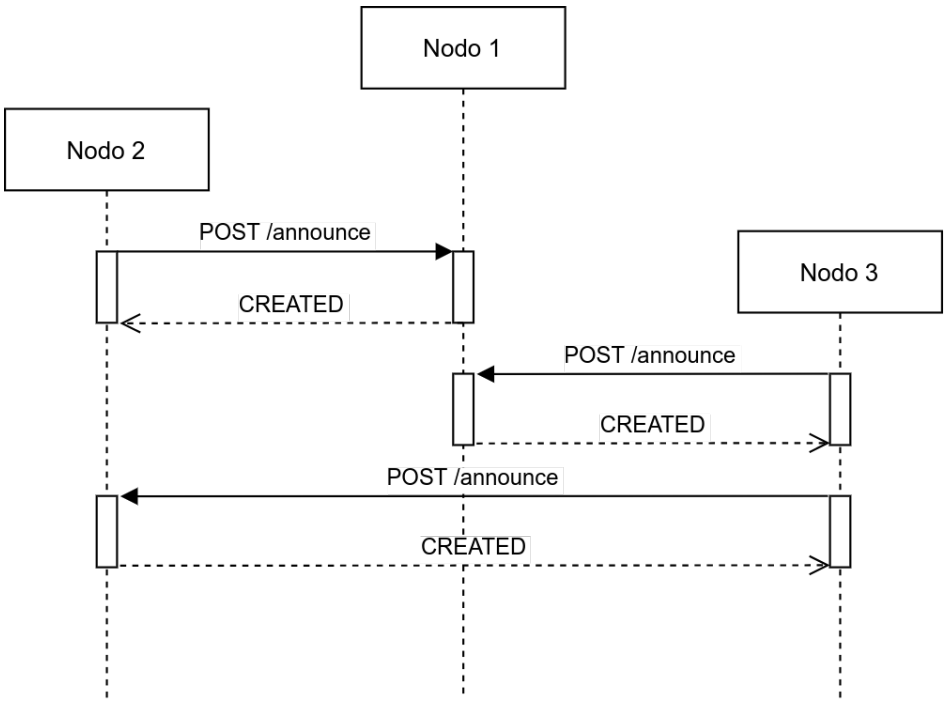


Figura 14: Diagrama de secuencia mostrando el proceso de "bootstrap" entre nodos

El nodo 1 se lanza primero y como es el primero de la red, no ocurre nada al respecto. Luego se instancia el nodo 2 y se le indica el nodo 1 como servidor de “bootstrap”.

El nodo 1 ahora mismo no conoce ningún otro nodo por lo que la respuesta hacia nodo 2 no indicará ningún otro nodo al que conectarse.

El nodo 3 aparece, también usa nodo 1 como servidor de “bootstrap” y nodo 1 le indica a nodo 3 que no conoce a nodo 2, rellenando el campo “announceYourselfToPeers” en la respuesta.

El nodo 3 inmediatamente irá a anunciarse al nodo 2 indicando que conoce al nodo 1 en “myKnownPeers”. El nodo 2 responde a nodo 3 indicando que no le hace falta anunciarse a ningún otro nodo, ya que conoce a todos los que él conoce.

El nodo 3 podía haber usado el nodo 2 como servidor de “bootstrap” y el resultado hubiera sido el mismo. Debido a cómo funciona el algoritmo, todos los nodos acabarían conociéndose.

Respecto a las interfaces de la capa REST API para el entrenamiento tenemos:

- POST a /training: comienza el entrenamiento en el nodo. Este método es para ser usado por parte de la interfaz gráfica.
    - Ejemplo de cuerpo de mensaje en esta operación:

      {'iAm': {'id': 'NODE_1', 'ip': None, 'port': '4000'}, 'trainerId': 'MNIST', 'modelParams': {'weights': [], 'bias': []}, 'round': 0, 'federatedLearningParams': {'e': 2, 'u': 1e-05, 'b': 64}}

    - Respuesta: NOT FOUND si no encuentra el job, OK si se encuentra.
- POST a /training/model_params: cuando un nodo termina de entrenar, hará un broadcast al resto de nodos en a esta interfaz para enviarle su modelo.

  Cada vez que llega un modelo remoto de otro nodo, también se comprueba si es posible hacer Federated Averaging.
    - Ejemplo de cuerpo de mensaje en esta operación:

      {'iAm': {'id': 'NODE_1', 'ip': None, 'port': '4000'}, 'trainerId': 'MNIST', 'modelParams': {'weights': [], 'bias': []}, 'round': 0, 'federatedLearningParams': {'e': 2, 'u': 1e-05, 'b': 64}}

    - Respuesta: NOT FOUND si no encuentra el job, OK si se encuentra.
- POST a /training/model_params/accuracy: después de hacer el Federated Averaging y calcular la precisión de su modelo promediado local contra su conjunto de datos de test, el nodo hará un broadcast a todos los nodos en esta interfaz para enviarle su modelo promediado y la precisión que el nodo fue capaz de obtener con él.
    - Ejemplo de cuerpo de mensaje en esta operación:

{'iAm': {'id': 'NODE_1', 'ip': None, 'port': '4000'}, 'trainerId': 'MNIST', 'modelParams': {'weights': [], 'bias': []}, ‘accuracy’: 0.500, 'round': 0, 'federatedLearningParams': {'e': 2, 'u': 1e-05, 'b': 64}}

- Respuesta: NOT FOUND, si no encuentra el job, BAD REQUEST si el nodo no está ocioso, no está en la ronda correcta o no ha terminado de hacer Federated Averaging. OK si todo correcto con una respuesta parecida a esta:

  {'iAm': {'id': 'NODE_2', 'ip': None, 'port': '4001'}, 'modelParams': {'weights': [], 'bias': []}, 'accuracy': 0.700}

En la Figura 15 podemos ver un posible ejemplo de interacción entre dos nodos, después de que un usuario inicie desde la interfaz gráfica una ronda de entrenamiento (interfaz POST a /training/trigger).

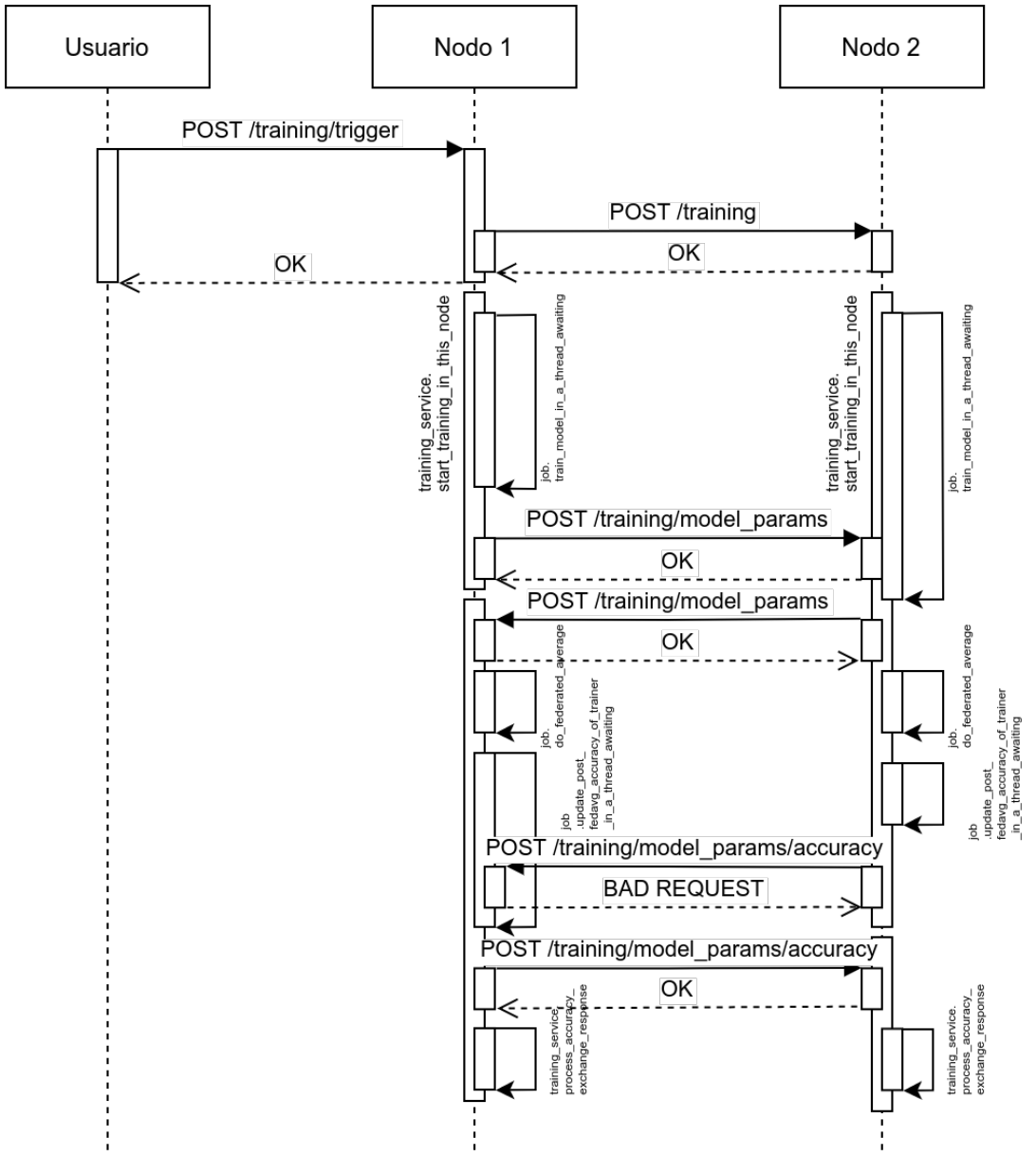


Figura 15: Diagrama de secuencia mostrando interacción entre nodos cuando ocurre el entrenamiento

El nodo 1 hace un broadcast avisando al resto de nodos (POST a /training), de que hay que empezar el entrenamiento. Ambos lo hacen, y cuando acaban, envían al otro nodo su nuevo modelo de pesos. Cuando pueden, hacen Federated Averaging, calculan su precisión y se lo envían al otro nodo.

Se puede observar que uno de esos envíos, falla con un BAD REQUEST debido a que el nodo todavía seguía ocupado calculando su precisión.

Pero cuando éste último acaba, envía su precisión al otro y comienza la comparación de precisiones para hacer intercambio de modelos, ya comentado

cuando se explicó el algoritmo central del servicio del entrenamiento, el que se encuentra en su método “start_training_in_this_node”.

3.4 La interfaz de usuario

Se han realizado una serie de pequeños cambios en la interfaz de usuario, para que sea más útil de cara a evaluar lo que pasa en la red.

La interfaz de usuario se accede por un navegador web, usando la dirección IP y el puerto abierto del nodo con protocolo HTTP, por ejemplo, http://127.0.0.1:4000.

Figura 16: La interfaz de usuario

Básicamente, se ha añadido la información relevante de cada job disponible en cada nodo. También del nodo del cual se entró en su interfaz gráfica se puede ver su información: es la información encima del botón verde de “Do a training round”.

Aparte, como se puede observar, para cada nodo aparece su identificador.

Para poder obtener esta información, cada nodo expone la siguiente operación en su capa REST API:

- GET /info: obtiene la información de todos los jobs del nodo.

- Respuesta: OK con un cuerpo de mensaje similar a este:

  {'iAm': {'id': 'NODE_4', 'ip': None, 'port': '4003'}, 'jobsInfo': [{'current_round': 0, 'fed_learn_done': 'Yes', 'federated_learning_params': 'Learning Rate: 1e-05, Epochs: 2, Batch size: 64', 'model_was_exchanged': 'No', 'post_fedavg_accuracy': 0.4545, 'pre_fedavg_accuracy': 0.4545, 'trainer_label': 'MNIST Dataset'}, {'current_round': 0, 'fed_learn_done': 'Yes', 'federated_learning_params': 'Learning Rate: 0.0001, Epochs: 1, Batch size: 2', 'model_was_exchanged': 'No', 'post_fedavg_accuracy': 0.7749999761581421, 'pre_fedavg_accuracy': 0.6499999761581421, 'trainer_label': 'Chest X-Ray Pneumonia Dataset'}, {'current_round': 0, 'fed_learn_done': 'Yes', 'federated_learning_params': 'Learning Rate: 0.0001, Epochs: 1, Batch size: 2', 'model_was_exchanged': 'No', 'post_fedavg_accuracy': 0.09759999811649323, 'pre_fedavg_accuracy': 0.094200000166893, 'trainer_label': 'CIFAR-10 Dataset'}], 'lastNodeStatus': 'IDLE'}

Por ello, cada vez que se pide la interfaz gráfica HTML a un nodo, éste llama a las interfaces “/info” de todos los nodos para obtener la información.

Veamos lo que nos muestra cada línea de la información. He aquí un ejemplo textual de una de esas líneas:
Trainer: MNIST Dataset**; Current Round:** -1**; Fed. Learning Params:** Learning Rate: 1e-05, Epochs: 2, Batch size: 64**; Fed. Avg. done?** No**; Pre FedAvg. Accuracy:** None**; Post FedAvg Accuracy:** None**; Was model exchanged?** No

- “Trainer” indica el el entrenador que corresponde con esa línea, en este caso MNIST.

- “Current Round” indica el identificador de ronda actual. Recordemos que “-1” significa que aún no ha habido trabajo con ese entrenador.

- “Fed. Learning Params” indica los hiperparámetros del Federated Learning.

- “Fed. Avg. Done?” indica si el Federated Averaging ha ocurrido o no.

- “Pre FedAvg. Accuracy“ indica la precisión obtenida del modelo contra el dataset local de test antes de que ocurra el Federated Averaging

- “Post FedAvg Accuracy” indica la precisión después de haber hecho el Federated Averaging, es decir, el modelo promediado, contra el dataset local de test.

- “Was model exchanged?” indica si el modelo fue intercambiado o no posteriormente por el algoritmo de intercambio de modelos ya explicado anteriormente.

Tendremos una línea como esa por cada job que se instancie en el nodo. Como hay 3 entrenadores, hay 3 jobs y por lo tanto habrá 3 líneas mostrándonos toda la información.

3.5 Evaluación

Se han realizado diversas pruebas y verificaciones manuales que permiten indicarnos que la red hace lo que se buscaba:

- Todas las pruebas realizadas han sido en el equipo del desarrollador como se nombró al principio del presente TFM.

  No se han realizado de manera automática, es decir, siempre se ha usado la interfaz gráfica para arrancar una sesión de entrenamiento.

  Para que sean 100% automáticas habría que hacer que la aplicación avise a otra de que ha acabado definitivamente o capturar que la aplicación está ociosa con un cierto margen de tiempo para empezar la siguiente ronda de entrenamiento.

- Aparte, se hicieron pruebas de funcionamiento externas con 6 SBCs Alix APUs de 4GB, los cuales tienen 2 núcleos y 2 hilos por núcleo. En dichas pruebas, se constató que la red P2P se desplegó adecuadamente y cada nodo hizo su parte.

- Dado que los nodos o están en estado ocioso u ocupado, no están preparados para entrenar dos modelos distintos a la vez. Todas las pruebas son siempre mandando a entrenar un dataset y esperando resultados. Se deja para trabajos futuros esta posibilidad.

- Se realizaron tests unitarios en el proyecto. En concreto, doce, realizados en parte, por miedo a haber cometido algún fallo a la hora de hacer el código más genérico, en parte porque viene siempre bien tenerlos y es una buena práctica.

  El archivo README en el código explica cómo arrancarlos y es deseable arrancarlos antes de intentar usar la aplicación a fin de encontrar posibles problemas antes de comenzar a operar con la red.

  Hubiera sido deseable hacer muchos más, sobre todo a nivel de comprobar las operaciones hechas en el protocolo de comunicación P2P, sin embargo, el tiempo disponible no permitió hacerlos.

  Los tests unitarios cubren estos aspectos:

  - Operaciones de inserción y borrado de nodos en el servicio de nodos.

  - Entrenamiento, obtención de precisión y realización del Federated Averaging en los 3 entrenadores, más las operaciones de serialización / deserialización de los modelos de pesos en formato JSON, para su paso por la red.

- Con MNIST se denota que la precisión antes y después del Federated Averaging, es igual o parecida. Es probable que la forma implementada

de obtener los conjunto de datos de entrenamiento, validación y test en cada ronda esté afectando en demasía.

Lo que sí es normal es que la precisión con MNIST es menor que en el proyecto original, pues se rebajó el ratio de aprendizaje de 1.0 a varias órdenes de magnitud menor, para que éste tuviera efecto en el aprendizaje.

A su vez, hay que tener en cuenta que este entrenador no está codificado con una red neuronal convolucional (CNN) como en el artículo original de Federated Learning, sino de manera lineal por lo que su Machine Learning puede verse menos beneficiado por el Federated Learning. Habría que profundizar en este aspecto probando con una versión CNN.

Sin embargo, el entrenador del Chest X Ray y CIFAR-10, sí que se notan que están beneficiados por el Machine Learning distribuido, mostrando mejoras de precisión en los nodos después del Federated Averaging.

- Se ha tomado muestras a fin de mostrar con los parámetros actuales entregados en el código, qué precisiones medias con su desviación estándar se pueden obtener después de 10 rondas (mirar Anexo 1, página 69).

  Con MNIST y Chest X Ray se usaron 10 nodos, pero con CIFAR-10, se usaron 8 nodos, debido al uso excesivo de RAM que impide que los nodos sobrevivan en el ordenador usado para el desarrollo:

  - MNIST: 0.560995 ± 0.135774846
  - Chest X Ray Pneumonia: 0.91875 ± 0.033253111
  - CIFAR-10: 0.23499375 ± 0.014362059

Se denota que la variabilidad con MNIST es alta, con una precisión bastante mejorable. Con Chest X Ray, podemos estar hablando que está cercano al “over-fitting”, dado sus buenas precisiones y que CIFAR-10 es extremadamente lento.

Con este último, se continuó haciendo rondas y se constata que la precisión sigue aumentando.

- Si en CIFAR-10 quisiéramos más precisión en menos rondas hay varias posibilidades.

  Por ejemplo, aumentando varias órdenes de magnitud el ratio de aprendizaje, es suficiente para conseguir precisiones entorno a 0.33 en la tercera ronda.

  Por supuesto, está la otra posibilidad de hacer que los nodos hagan más iteraciones por ronda, es decir, aumentar el parámetro E del Federated Learning, lo cual haría todo mucho más lento. Debido al uso elevado de recursos de computación, habría que valorar cómo obtener más precisión con el mínimo gasto.

  Eso sí, siempre hay que tener en cuenta que ratios de aprendizaje elevados pueden tener un efecto minimizador de la precisión obtenida, es decir, el cálculo de la función de pérdidas podría no llegar al mínimo local óptimo, pues saltaría a un lado u otro del mínimo.

- Respecto a los intercambios de modelo, se ha probado poniendo el umbral de diferencia de precisiones a 0 tanto en MNIST y Chest X Ray, por ser los más rápidos de entrenar, para provocar posibles intercambios. Se constata que no se producen, siendo esto lo esperado.

  Se puede observar en los propios logs de la aplicación, en el momento que se comprueba el modelo de un nodo externo contra su conjunto de datos de test local, que la precisión obtenida es exacta hasta el último decimal respecto a la precisión del modelo local del nodo.

Esto es consecuencia de que los nodos trabajan exactamente igual y se inicializan de la misma manera.

Tengamos en cuenta que se hace un Federated Averaging, que es obtener la media de los pesos de los modelos y todos los modelos se intercambian entre todos los nodos, entonces todos los nodos poseen la misma información y hacen los cálculos de la misma manera, produciendo los mismos resultados.

La única diferencia es que cada nodo tiene un conjunto de datos distinto, y por ello, las precisiones que se obtienen en cada nodo son similares, pero no exactamente iguales.

Durante el desarrollo sí se ha visto intercambios, aunque era una ocurrencia rara también, pero entonces no estaba todo tan afinado como lo está ahora al final del desarrollo.

Probablemente, estos intercambios ocurrirán con nodos que se hayan descolgado en algún momento y vuelvan a la red más tarde sin re-anunciarse a la red, pues éstos habrán perdido rondas y probablemente ya no estén en "sintonía" con el resto.

Habría que hacer más pruebas, y más profundas, para finalmente determinar si tiene sentido mantener esta característica.

3.6 Resultados obtenidos

- Se dispone de un software que despliega una red de Federated Learning P2P a malla completa con una capa de comunicación basada en REST API y mensajes tipo JSON.

- Dicha red, hace que todos los nodos calculen el Federated Averaging individualmente con los resultados de todos los nodos que obtengan.

- Esta red permitirá hacer experimentos de configuración / optimización / aplicaciones prácticas de redes Federated Learning P2P en todo aquel

dispositivo computacional de cualquier tipo, siempre y cuando soporte Python 3.8 y las librerías utilizadas o como mínimo, soporte para arrancar contenedores de Docker con la imagen que se genera a partir del archivo Dockerfile.

- Sigue un diseño arquitectónico definido y ampliamente utilizado en Ingeniería del Software, con suficiente modularidad y responsabilidades definidas.

- Es fácil añadir entrenadores nuevos de Machine Learning, sin necesidad de estar haciendo retoques internos al código.

- La red tiene una cierta resistencia a caídas de los nodos, es decir, la red puede seguir funcionando pese a ello, pudiendo los nodos unirse más tarde a la red sin que ello suponga algún problema.

- Se hace Machine Learning en los entrenadores acorde a buenas prácticas de la industria, aunque claramente mejorable.

- Sobre la usabilidad:

    - La interfaz gráfica muestra mucha información sobre el estado interno de los nodos.

    - La usabilidad es mejorable ya que:

        - Las rondas de entrenamiento hay que realizarlas manualmente desde la interfaz gráfica.

        - La interfaz gráfica no se auto-actualiza, por lo que el usuario tiene que refrescar manualmente si quiere ver información reciente.

- Sobre el rendimiento:

    - Las librerías de Machine Learning son multi-núcleo y la aplicación usa corutinas asíncronas de entrada / salida e hilos acordemente.

- Sin embargo, se ha detectado que hace falta más desarrollo en este aspecto, pues el consumo de RAM y de CPU puede ser factores limitantes, como se verá en las lecciones aprendidas, apartado 5.1.

- Sobre otras limitaciones:

  - El hecho de que se haya realizado una malla completa de nodos y las comunicaciones sean síncronas, hace que su escalabilidad se vaya a resentir, es decir, probablemente no funcione bien con cantidades masivas de nodos.

4. Trabajos relacionados

El trabajo realizado puede ser contrastado con otros trabajos más complejos tomados de la literatura científica como los aquí nombrados:

- En ”BACombo — Bandwidth-Aware Decentralized Federated Learning”[18], los autores proponen lo siguiente:

  - En vez de extraer el modelo completo de un nodo, el modelo se divide en partes y cada nodo, extrae una parte del modelo de otros nodos.

    El tráfico seguirá siendo el mismo, como si se hubiera obtenido el modelo completo de un solo nodo, pero así se distribuye entre varios enlaces a los nodos, por lo que tiene en cuenta que en una situación real, los nodos no necesariamente van a estar en el mismo lugar y las restricciones de ancho de banda de cada uno de los enlaces de red que interconectan los nodos, se hacen más notables.

  - En cada iteración, para extraer los segmentos de los modelos y hacer Federated Averaging, se realiza o una selección estocástica de nodos, es decir, se hace una exploración con tal de conocer nuevos nodos o se utilizan los nodos ya conocidos con mejores anchos de banda. Utilizan el tiempo del round-trip (ida-vuelta), manteniendo un

historial de anchos de banda anteriores por cada nodo para realizar esta selección.

La idea aquí es que se asume que los anchos de banda de los pares cambian con el tiempo, por lo que usar siempre los mismos nodos implica que con el tiempo habrá un detrimento de velocidad en la red debido a los anchos de banda cambiantes.

Nuestra implementación no contiene trabajo alguno en el sentido de reducir el ancho de banda utilizado, al contrario que con este sistema, que lo tiene muy en cuenta.

Además, hicieron que los nodos se traen del resto de nodos la información, cuando nuestra aplicación “empuja” los modelos, las precisiones, etc., es decir, cada nodo los envía al resto de nodos de manera síncrona.

- En “BAFFLE : Blockchain Based Aggregator Free Federated Learning”[19] , los autores proponen lo siguiente:
  - La idea principal es utilizar “blockchain” como coordinador central de Federated Learning, para hacer descentralización mediante el uso de contratos inteligentes en una red privada Ethereum.

    Se tienen en cuenta además, las restricciones particulares introducidas por la propia cadena de bloques, como el tamaño de los bloques, para reducir el coste computacional.

El contraste es evidente. Nuestra implementación es totalmente autocontenida y no necesita de un software de terceros, al contrario que con este sistema, que requiere del software de Ethereum.

Por otra parte, esa solución será más escalable que la aquí realizada en cuanto los nodos interactúan directamente con la cadena de bloques, mientras que en nuestra implementación todos los nodos se hablan entre ellos.

- En “BrainTorrent: A Peer-to-Peer Environment for Decentralized Federated Learning”[20], los autores proponen lo siguiente:
    - Plantean un algoritmo que funciona tal que así:
        - Los modelos están versionados.
        - Los nodos trabajan localmente por unas cuantas iteraciones.
        - Aleatoriamente un nodo decide hacer Federated Learning pidiendo la versión de sus modelos al resto de nodos.
        - Todos aquellos que tienen actualizaciones, este nodo los recoge, el cual los mezcla junto a su copia local y los envía a aquellos nodos que no tienen actualizaciones.

    Se puede decir que nuestro control de ronda es muy parecido al versionado de modelos aquí implementado, que es más sofisticado.

    La diferencia estriba que en nuestro sistema todos los nodos recibirán todos los modelos del resto y todos harán Federated Averaging individualmente; en cambio, en BrainTorrent, un nodo aleatorio harmoniza su modelo local contra las actualizaciones del resto de nodos y en cada iteración, un solo nodo realiza esa operación.

    Se ahorrará mucho ancho de banda respecto al sistema aquí implementado y será escalable, pues no todos interactúan siempre.

    El sistema desarrollado en este TFM sólo tiene la ventaja de que todos obtendrán el modelo promediado más actualizado lo antes posible, al contrario que en BrainTorrent.

- En “IPLS : A Framework for Decentralized Federated Learning”[21], los autores proponen lo siguiente:

- Para descentralizar, se utiliza IPFS[22], el “InterPlanetary File System”, un protocolo P2P de almacenamiento de archivos distribuido concebido entre otros para crear una Web resiliente.

- Aparte del modelo, la petición de entrenamiento incluye la función de pérdidas y el algoritmo de optimización.

- Debido al uso de IPFS, tuvieron que pensar cómo controlar la replicación del modelo para no abusar y asegurar unos mínimos de distribución, usando modelo publicador / subscriptor. Así los “agentes” (como llaman a los nodos), según las restricciones, podrán replicar particiones del modelo o no según la situación.

- La idea es conseguir que el modelo quede lo suficientemente distribuido para que los agentes puedan realizar el entrenamiento. A su vez, se tiene cuidado en quién recibe las particiones del nuevo modelo calculado después del entrenamiento

Respecto a contrastar este sistema con el nuestro, estamos ante una situación parecida a la del anteriormente nombrado BAFFLE.

La forma de hacer la distribución viene condicionada por las restricciones y ventajas del medio de transmisión, IPFS en este caso, blockchain en el caso de BAFFLE.

Al igual que con BAFFLE, este sistema es más escalable que el aquí implementado.

5. Conclusiones

Se ha estudiado qué es el Federated Learning, cómo es su implementación y el código inicial del cual partimos para hacer este TFM.

Se ha realizado un nuevo código basado en el anterior que hace su operativa en modo descentralizado P2P, se ha tenido en cuenta una serie de detalles durante su transformación, se han realizado arreglos y mejoras varios

y se ha caracterizado, no solo sus resultados, sino sus necesidades de CPU y RAM.

Se ha comentado durante toda la memoria posibles mejoras y líneas de trabajo futuro que concretaremos en el siguiente apartado, además de lecciones aprendidas obtenidas durante el camino, que se verán en el apartado 5.1.

También se ha contrastado el sistema desarrollado en este TFM contra los de otros trabajos relacionados encontrados en la literatura científica.

Podemos concluir que se han logrado todos los objetivos iniciales propuestos y se dispone de una buena base de la que partir para futuros desarrollos e investigaciones.

Respecto a planificación, se ha respetado mayormente: se consiguió adelantar una semana en su primera fase, pero la fase de implementación requirió de algo más de tiempo del esperado, debido a los defectos del abuso de CPU detectados en el último momento que requirieron verificar qué estaba pasando y cómo mejorarlo.

Por parte del autor del presente TFM, se valora muy positivamente la experiencia. Ha habido mucho aprendizaje, a varios niveles y en varias áreas como Machine Learning, además de tratar bastante con el lenguaje de programación Python.

Se hubiera deseado tener más tiempo para valorar muchas más posibilidades, pero viendo hasta donde hemos llegado, podemos estar satisfechos con lo obtenido.

5.1 Lecciones aprendidas

A lo largo del desarrollo hubo una serie de sucesos, que ayudaron a aprender aún más sobre la implementación realizada:

- Como ya se indicó, se ha hecho una distinción entre 3 datasets: training, validación y test. Durante la codificación para el entrenador del MNIST se cometió un fallo interesante que se corrigió. Se metió un sesgo en el dataset de training, el cual contuvo más muestras de datos etiquetados como “3” que de “7” y el de test más muestras de datos etiquetados como “7” más que de “3”.

  El resultado fue que la precisión era baja y además no subía. Una vez detectado el problema, se hizo revisión de cómo se hacía la obtención de los 3 datasets en todos los trainers, por si acaso.

- Del punto de partida, el entrenador del Chest X Ray daba precisiones entorno a 0.5 y no subía. Se identificó por qué pasaba esto.

  Se empezó leyendo el artículo original del Federated Learning de nuevo [1], llamando mucho la atención que indicaba que es necesario que hubiera una inicialización común para todos los nodos, esto es, el modelo inicial del cual se parte para el entrenamiento debe ser el mismo y de usar operaciones aleatorias, todos los nodos deben usar la misma semilla.

  Es decir, todos los nodos deben trabajar de la misma manera o si no, no habrá convergencia y/o se pueden quedar entorno a un mínimo local no óptimo.

  En un principio, no se usaba el mismo modelo inicial en todos los nodos. Después de corregirlo, todos los nodos alcanzan precisiones altas como ya se nombró anteriormente.

  También se hizo que la semilla para el generador de números aleatorios fuera igual para todos, aunque solo el entrenador del CIFAR-10 tiene capas en su modelo de red neuronal (capas “Dropout”) que utilizan la operación “random” en su interior.

- El orden de importación (sentencias “import” o “from module import” en Python) de las librerías de Machine Learning es muy importante: en el orden erróneo provocaríamos un fallo de segmentación en la aplicación.

  Se entiende que los desarrolladores de las diferentes librerías no comprueban su “convivencia” entre ellas en una misma aplicación, por lo que se considera normal.

  Es por ello, que lo ideal es apostar por una sola librería y hacer todos los entrenadores con ella, a fin de no provocar malos comportamientos por parte de la aplicación.

- En un principio, haciendo pruebas con 10 nodos se denotó que el sistema es incapaz de trabajar en condiciones.

  Se tuvo que aumentar a 30 segundos el timeout de las peticiones debido a esto, ya que algunos morían en el intento.

  Entrenar modelos de Machine Learning es muy demandante de recursos de CPU por lo que se entiende que es una situación normal, ya que si se levantan varios procesos, superando el número de núcleos del sistema (4 CPUs con 2 hilos por CPU, recordemos, en el sistema del autor del TFM) y cada uno de ellos crea un hilo de entrenamiento, el sistema operativo no necesariamente va a dar mucho tiempo de CPU a los hilos con bucle de eventos, los cuales necesitan el suficiente tiempo para procesar las peticiones que llegan de los otros nodos en el mismo ordenador.

  En una situación en el que un ordenador de varios núcleos es un sólo nodo, se espera que el timeout de las peticiones dependa solamente de las condiciones de red más que por el abuso de la CPU.

  Sin embargo, se denota que Tensorflow es multi-núcleo y puede seguir habiendo un abuso de CPU, por el cual el sistema operativo no de el tiempo suficiente al hilo principal.

De hecho, se ha visto que es probable “romper” el servidor HTTP que lleva cada nodo y dejarlo en un estado irrecuperable cuando se entrenan muchos nodos de CIFAR-10 o Chest X Ray en el mismo ordenador.

Se hicieron algunos ajustes en el algoritmo para disminuir el problema, aunque no se descarta que siga ocurriendo según la cantidad de nodos que se levanten dentro de una misma máquina o si se abusa de la CPU demasiado.

También entra en juego el consumo de memoria RAM, siendo tan excesivo que es posible provocar fácilmente que se arranque el “OOM-Killer” en Linux, la subrutina del núcleo del sistema operativo que elimina procesos con tal de rebajar la presión por parte de los procesos sobre la memoria RAM, que puede haber en un determinado momento.

Dicho “OOM-Killer” “asesina” nodos durante el entrenamiento hasta que el consumo de RAM sea aceptable.

Es por ello, que no se recomienda crear muchos nodos en una misma máquina.

En el caso del entorno de desarrollo, se puede mantener 10 nodos de MNIST y Chest X Ray, pero no de CIFAR-10, el cual es más consumidor de memoria RAM.

5.2 Líneas de trabajo futuro

A lo largo de la memoria, se han indicado qué posibles mejoras podrían hacerse, sin embargo, expondremos aquí todo lo que consideremos más importante y/o interesante de perseguir en futuros desarrollos:

- Hacer realidad la posibilidad de que la red entrene de manera autónoma, mediante un objetivo.

  Para llegar a este punto, es recomendable valorar el protocolo en sí siguiendo las pautas de comunicación asíncrona arriba nombradas, pues

ahora hay que calcular cuándo hay que darle a la siguiente ronda, detectando que no hay actividad en el sistema.

- Cambiar a comunicación asíncrona. En el diseño actual, los nodos “empujan” su información al resto de nodos constantemente, es decir, se realiza comunicación síncrona. Se ha comprobado que hace más difícil la tarea del desarrollador a la hora de confeccionar un algoritmo de comunicación.

  Al igual que en los trabajos relacionados, un modelo de comunicación en el que los nodos extraigan la información del resto de nodos a conveniencia sería conveniente de explorar.

  Una comunicación asíncrona mediante un sistema de mensajería publicador / subscriptor como el nombrado en el artículo sobre IPLS de los trabajos relacionados sería deseable, aunque deje así el código de ser autocontenido por posiblemente necesitar de otro software de terceros para resolverlo.

- Al ser una malla completa, hay demasiada comunicación en esta red. Comunicación que no se ha valorado no solo cómo reducirla a nivel de protocolo, es decir, cantidad de mensajes, sino en el peso de los mensajes que se envían en sí.

  Hay que tener en cuenta que se puede explotar lo indicado por Google en su artículo original (ver introducción del capítulo 2). Esta es una vía a explorar en futuros desarrollos.

- Los dos puntos anteriores, ayudarían a aumentar la escalabilidad del sistema, que es otro apartado a desarrollar ampliamente en el futuro.

- Revisar el uso del multi-núcleo y el multi-hilo en la aplicación. Es importante controlar ese aspecto bien pues es fuente de problemas.

  Mientras se hacía el código, se constató que al contrario que otros lenguajes de programación, Python no permite un control fino de los

hilos / procesos, para por ejemplo, darle prioridades y que el sistema operativo sepa qué se considera importante a la hora de repartir tiempo de CPU.

Se podría resolver haciendo que el Machine Learning ocurra en un proceso y usar herramientas del sistema operativo para hacer que estos procesos tengan menor prioridad que el servidor web, que es lo que interesa.

- Revisar el uso de la RAM. Parte de la causa del excesivo consumo es debido a tener que mantener en memoria el dataset de test durante bastante tiempo para poder hacer los cálculos de precisión posteriores al Federated Averaging. Habría que valorar si es posible ejecutar alguna estrategia que permita bajar ese consumo.

- Respecto al dataset de test en sí, su confección actual no es la forma más correcta[5], pues debería ser un humano el que cuidadosamente seleccione las muestras para el dataset de test.

  Aún así, hoy día se usarían técnicas de validación cruzada para hacer la evaluación de los modelos. Habría que evaluar esas mejores formas de mejorar el Machine Learning aplicado.

- Revisar el algoritmo de intercambio de modelos, para lo cual habría que hacer pruebas particionando la red y descubrir cuán efectivo es.

- Tener en cuenta que las características de los nodos no son homogéneas (CPU, RAM, etc.), por lo que cada nodo podría tener parámetros de Federated Learning personalizados (ratio de aprendizaje, iteraciones y cantidad de muestras por iteración) para explotar mejor sus características.

- La comunicación entre nodos no es segura. El modelo generado por los nodos puede ser un resultado que se quiera mantener privado, libre de ser “robado” por un tercero en su tránsito por la red, por lo que hay que explorar cómo aumentar la seguridad en este aspecto.

- Se detectó un detalle interesante de la operación “numpy.mean”, la operación de cálculo de media de un vector utilizada.

  La propia documentación[23] de esa operación nos indica que si no se utiliza un tipo especial, “float64”, habría pérdidas de precisión. Cabría preguntarse cómo ésto afecta no solo al cálculo de dicha media, sino al propio Machine Learning.

- Optimizar la creación de los modelos de Machine Learning. El desarrollo actual instancia una configuración del modelo nuevo en cada ronda de entrenamiento. Si Federated Learning impone que todos los nodos deben trabajar igual, el hecho de recrear el modelo suministrándole la misma semilla aleatoria en cada iteración, hace que la secuencia de números sea igual y partiendo siempre del mismo número en cada iteración.

  Probablemente lo deseable es configurar el modelo de Machine Learning una sola vez y reutilizarlo en subsecuentes iteraciones, al menos las capas del modelo que utilizan el valor de semilla aleatoria fijada en la configuración.

- Mejorar la interfaz gráfica para que ésta se actualice sola. También, que se puedan cambiar los parámetros de los entrenadores y además sea más agradable su visualización.

- Hacer persistencia de los modelos. Los modelos se pierden cuando los nodos mueren, así que hay que valorar cómo hacer su persistencia a disco

  Hay que considerar también que los modelos deben usarse en otros sistemas. El objetivo de hacer modelos es que éstos sean útiles en algún momento para hacer predicciones. Predicciones que no necesariamente tiene que hacer el sistema aquí desarrollado.

- Hacer que el sistema trabaje de forma continua y autónoma, cambiando y mejorando a medida que se le introduzcan más datos en el dataset.

  Ejemplo: si se hace un proyecto usando este código para una red de hospitales para discriminar enfermedades a raíz de radiografías, querrán poder usar los resultados del Machine Learning para hacer predicciones y además, querrán que funcione continuamente, pues las enfermedades evolucionan y la “firma” de la enfermedad en las radiografías puede evolucionar a otras formas, que deben ser apropiadamente categorizadas.

6. Glosario

Algoritmo del Descenso del Gradiente – algoritmo de optimización que ajusta los parámetros de un modelo de forma que acabe encontrando el mínimo global óptimo de una determinada función utilizando todo el conjunto de datos.

Algoritmo del Descenso del Gradiente Estocástico – versión más optimizada del Algoritmo del Descenso del Gradiente, en el que en vez de usar todo el conjunto de datos, se usa un lote aleatorio por iteraciones.

Blockchain – cadena de bloques utilizada para guardar información de manera que no se puedan modificar los registros ya almacenados posteriormente y pueda ser consultada por cualquier actor en una comunicación. A su vez, puede contener contratos inteligentes, que son programas que imponen un determinado flujo de trabajo o protocolo.

Bootstrap – fase en un protocolo de red superpuesta P2P en el que se realiza la inserción de un nodo en la red de forma que quede accesible para el resto de nodos.

Broadcast – cuando un nodo hace una transmisión simultánea a multitud de nodos.

Convolutional Neural Network (CNN) – red neuronal convolucional. Modelo para ser usado en el entrenamiento de Machine Learning, que contiene una serie de capas interconectadas en el que algunas de esas capas utilizan la operación de convolución.

Dataset – conjunto de datos que sirve para alimentar un algoritmo de Machine Learning, con tal de que éste sepa entrenar el modelo o evaluarlo.

Fallo de segmentación – ruptura de la aplicación debido a realizar alguna operación considerada ilegal por parte del sistema operativo contra la memoria, como acceder a un rango de memoria no asignado.

Federated Averaging (FedAvg) – es el paso final en el algoritmo de Federated Learning, consistente en promediar todos los modelos obtenidos de forma que se obtenga un modelo mejorado, resultado de la unión de todos los esfuerzos de Machine Learning realizados individualmente en cada nodo de la red.

Federated Learning – término originado en los laboratorios de Google en el que se busca realizar Machine Learning de manera distribuida evitando compartir los datos, sólo compartiendo los modelos. El objetivo es garantizar la privacidad de los datos a la hora de obtener modelos que permitan hacer predicciones.

Instruction Set Architecture (ISA) – define los elementos a muy bajo nivel como tipos de datos, registros, direccionamiento de memoria, etc. que implementan las distintas arquitecturas de computación.

JavaScript Object Notation (JSON) – define una forma particular para describir información, que se ha hecho muy popular como forma de intercambio de mensajes en servicios web.

Log – registro continuado del funcionamiento de la aplicación. Puede verse en la consola o guardarse en un archivo para verlo posteriormente.

Machine Learning – rama de la Inteligencia Artificial, cuyo objetivo es conseguir que los ordenadores aprendan, mediante la creación de modelos basados en el análisis de datos que permitan hacer predicciones.

Mensajería publicador / subscriptor – modelo de intercambio de mensajes en red de manera asíncrona que permite desacoplar las aplicaciones que lo utilizan.

OOM-Killer - subrutina del núcleo del sistema operativo que elimina procesos con tal de relajar la presión sobre la memoria RAM que puede haber en un determinado momento.

PEC – abreviatura de “pruebas de evaluación continua”, utilizadas en una organización educativa para evaluar el rendimiento y el trabajo constante del alumnado.

Peer-to-Peer (P2P) – se dice que un software es P2P cuando éste genera una red de ordenadores superpuesta capaz de comunicarse entre otros ordenadores sin necesidad de disponer de un servidor central.

Polimorfismo – en programación orientada a objetos es la posibilidad de disponer de la misma interfaz de acceso para unas determinadas operaciones por parte de objetos inherentemente distintos. Cada lenguaje de programación aplica este concepto de distintas maneras.

Pool de hilos – elemento de software que mantiene uno o varios hilos de concurrencia abiertos en memoria (los “workers”) para evitar perder tiempo en su creación y destrucción continua.

Refactorización – técnica de la Ingeniería del Software para reestructurar un código fuente sin alterar cómo éste funciona de cara al exterior.

Representational State Transfer Application Programming Interface (REST API) – nos indica qué elementos se pueden usar y cómo en una interfaz de programación para servicios web con protocolo HTTP.

Single Board Computing (SBC) – concentra todo lo mínimo necesario para hacer computación en una pequeña placa totalmente integrada.

Singleton – patrón de diseño en Ingeniería del Software aplicado en programación orientada a objetos, con el que se consigue que sólo haya una sola instancia en memoria de un determinado objeto.

Trainer – software que realiza el Machine Learning para un dataset concreto.

Universally Unique Identifier (UUID) – permite generar un identificador de manera independiente en sistemas distintos sin coordinador centralizado que no producirá duplicados cada vez que éstos se generen.

7. Bibliografía

[1] McMahan B., Moore E., Ramage D., Hampson S., Agüera y Arcas B. (2017). Communication-Efficient Learning of Deep Networks from Decentralized Data. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, in PMLR*, *54* (n/a), *1273-1282*. http://proceedings.mlr.press/v54/mcmahan17a.html

[2] Liu Y., Yuan X., Xiong Z, Kang J., Wang X., Niyato D. (2020). Federated learning for 6G communications: Challenges, methods, and future directions. *China Communications*, *17* (9), *105-118*. https://doi.org/10.23919/JCC.2020.09.009

[3] Yáñez Parareda E. (2021). *Federated Learning Network: Training distributed Machine Learning models with the Federated Learning paradigm*. [Trabajo final de máster]. Universitat Oberta de Catalunya. http://hdl.handle.net/10609/126546

[4] JetBrains. (2021). *PyCharm*. https://www.jetbrains.com/pycharm/

[5] Shah T. (2017, 6 de Diciembre). About Train, Validation and Test Sets in Machine Learning [entrada de blog]. *towards data science*. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

[6] Python. (2021). *asyncio — Asynchronous I/O*¶. https://docs.python.org/3/library/asyncio.html

[7] Quart. (2021). *Quart*. https://pgjones.gitlab.io/quart/

[8] Hypercorn. (2021). *Hypercorn*. https://pgjones.gitlab.io/hypercorn/

[9] Docker. (). *Docker*. https://www.docker.com/

[10] Red Hat OpenShift. (2021). *OpenShift Container Platform-Specific Guidelines*. https://docs.openshift.com/container-platform/3.3/creating_images/guidelines.html#openshift-container-platform-specific-guidelines

[11] The Python Packaging Authority. (2020). *pip*. https://pip.pypa.io/

[12] The Python Packaging Authority. (2020). *Pipenv: Python Dev Workflow for Humans*. https://pipenv.pypa.io/

[13] Stack Overflow. (2011). *Creating a singleton in Python*. https://stackoverflow.com/q/6760685

[14] DeepAI. (2019). *MNIST Dataset*. https://deepai.org/dataset/mnist

[15] Mendeley Data. (2018). *Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification*. https://data.mendeley.com/datasets/rscbjbr9sj/2

[16] Alex Krizhevsky. (2009). *The CIFAR-10 dataset*. https://www.cs.toronto.edu/~kriz/cifar.html

[17] kaggle. (2020). *Object Recognition: Compare MLP and CNN CIFAR*. https://www.kaggle.com/guesejustin/object-recognition-mlp-cnn-efficientnet-on-cifar

[18] Jiang J., Hu L., Hu C., Liu J., Wang Z. (2020). BACombo — Bandwidth-Aware Decentralized Federated Learning. *Electronics*, *9* (3), *440*. https://doi.org/10.3390/electronics9030440

[19] Ramanan P., Nakayama K. (2020). BAFFLE : Blockchain Based Aggregator Free Federated Learning. https://arxiv.org/abs/1909.07452

[20] Guha R., Siddiqui S., Pölsterl S, Navab N., Wachinger C. (2019). BrainTorrent: A Peer-to-Peer Environment for Decentralized Federated Learning. https://arxiv.org/abs/1905.06731

[21] Pappas C., Chatzopoulus D., Lalis S., Vavalis M. (2021). IPLS : A Framework for Decentralized Federated Learning. https://arxiv.org/abs/2101.01901

[22] Protocol Labs. (2021). *IPFS powers the Distributed Web*. https://ipfs.io/

[23] NumPy. (2021). *numpy.mean*.
https://numpy.org/doc/stable/reference/generated/numpy.mean.html

[24] Python. (2021). *logging — Logging facility for Python*.
https://docs.python.org/3/library/logging.html

8. Anexos

8.1 Anexo 1: Datos recopilados para obtener la media y desviación estándar de la precisión de los tres entrenadores

Los datos aquí mostrados se han obtenido de la manera siguiente:

- Se han arrancado X nodos.
- Se hacen 10 rondas de entrenamiento en cada entrenador, es decir hasta la ronda número 9 que muestra la interfaz gráfica, pues las rondas comienzan en el 0.
- Se recopilan los datos del campo “Post FedAvg” de la interfaz gráfica de cada nodo.
- Se eliminan los nodos y se vuelve a empezar este algoritmo. Así, 4 veces.
- Con los datos obtenidos, se hacen la media y desviación estándar global del entrenador.

Con MNIST y Chest X Ray se usaron 10 nodos. Con CIFAR-10, se usaron 8.

Para MNIST:

Nodo/Repetición	**1**	**2**	**3**	**4**
1	0,4706	0,4	0,65	0,75
2	0,4783	0,6087	0,55	0,7143
3	0,5385	0,6667	0,3333	0,8824
4	0,6667	0,5714	0,5	0,5909
5	0,56	0,5263	0,5833	0,8
6	0,6316	0,52	0,2917	0,6111
7	0,6471	0,4167	0,5652	0,7143
8	0,4615	0,52	0,4286	0,6818
9	0,5833	0,4167	0,4091	0,8462
10	0,4737	0,4	0,3889	0,5909
	Media		**D. Estándar**	
	0,560995		0,135774846	

Para Chest X Ray:

Nodo/Repetición	1	2	3	4
1	0,899999976	0,9375	0,912500024	0,975000024
2	0,899999976	0,987500012	0,887499988	0,925000012
3	0,925000012	0,912500024	0,925000012	0,962499976
4	0,949999988	0,975000024	0,899999976	0,887499988
5	0,912500024	0,949999988	0,9375	0,875
6	0,875	0,987500012	0,899999976	0,912500024
7	0,887499988	0,949999988	0,850000024	0,949999988
8	0,949999988	0,949999988	0,912500024	0,925000012
9	0,887499988	0,887499988	0,887499988	0,899999976
10	0,912500024	0,912500024	0,899999976	0,875

Media	D. Estándar
0,91875	0,033253111

Para CIFAR-10:

Nodo/Repetición	1	2	3	4
1	0,239600003	0,224000007	0,251399994	0,221799999
2	0,224399999	0,218199998	0,254000008	0,244399995
3	0,233400002	0,219600007	0,255600005	0,235599995
4	0,222000003	0,220400006	0,252600014	0,231399998
5	0,223399997	0,219400004	0,252999991	0,240600005
6	0,234799996	0,229399994	0,265199989	0,234599993
7	0,223800004	0,213	0,259200007	0,236399993
8	0,227599993	0,223000005	0,257600009	0,230399996

Media	D. Estándar
0,23499375	0,014372059

8.2 Anexo 2: Variables de entorno a utilizar en el arranque de la aplicación

Aunque en el archivo README encontrado en el código suministrado, considerado el manual del usuario, se explica cómo arrancar la aplicación, aquí se explican todas las posibilidades de las variables de entorno utilizables en el arranque.

Todos los nodos de la red necesitan usar la siguiente variable de entorno:

- FLNET_NODE_PORT: indica el puerto HTTP que tiene que abrir el nodo

El primer nodo de la red se instancia solamente con FLNET_NODE_PORT, pero cuando se lanza el segundo nodo de la red en adelante, hay que utilizar la siguiente variable de entorno también:

- FLNET_BOOTSTRAP_PAIRS: variable de entorno en el que se le indica a qué nodos debe conectarse al comienzo para anunciarse. Puede ser uno o varios separados por comas. Ejemplo de uso:

  - FLNET_BOOTSTRAP_PAIRS='127.0.0.1:5000': significa que el nodo que se está arrancando se anunciará al nodo existente en la dirección IP 127.0.0.1 en su puerto 5000.

  - FLNET_BOOTSTRAP_PAIRS='127.0.0.1:5001,192.168.1.3:5003': significa que el nodo que se está arrancando se anunciará a los nodos existentes en las direcciones IPs 127.0.0.1 y 192.168.1.3, en sus puertos 5001 y 5003 respectivamente.

Las variables de entorno esenciales son las arriba nombradas, pero existen otras variables de uso opcional:

- FLNET_DATASETS_PATH: sirve para indicarle a la aplicación una ruta alternativa donde encontrar los datasets que tienen que ser manualmente descargados.

  Esta variable de entorno la utiliza la imagen de Docker generada, pudiéndose ver cómo se utiliza en el archivo Dockerfile.

- FLNET_NODE_ID: sirve para indicar manualmente el identificador del nodo. Si no se utiliza, un UUID se generará como identificador del nodo.

  Esta variable de entorno para hacer pruebas puede ayudar bastante. Por ejemplo, para hacer el testing, se hicieron unos scripts que instancian nodos en terminales individuales, colocando el nombre del nodo en el título del terminal, de forma que se pueda identificar fácilmente qué terminal contiene un determinado nodo.

- FLNET_NODE_IP: permite indicar una dirección IP para ser usada en la fase inicial de bootstrap de los nodos.

  Debido a algunas configuraciones de red, especialmente con las redes creadas en Docker, puede ocurrir que la dirección IP obtenida de las peticiones cuando éstas se reciben en los nodos, no sea la correcta para poder establecer la comunicación entre ellos.

  Esta situación se solventa gracias a esta variable de entorno, que fuerza el envío de una dirección IP específica configurada por el usuario de la aplicación.

8.3 Anexo 3: Párametros de configuración

En el directorio “configuration”, en su archivo config.py dentro del código podemos encontrar las siguientes variables:

- LOGGING_LEVEL: la aplicación se ha preparado para utilizar el subsistema de “logging” de Python, y esta variable indica el nivel de “logging”[24] que queremos ver por la consola.

- TMP_PATH: indica la ruta hacia el directorio de archivos temporales, utilizado por el entrenador de “Chest X Ray” para colocar ahí un dataset temporal.

- DATASETS_PATH: indica la ruta hacia donde se tienen guardados los datasets para los entrenadores que hay que descargarse manualmente.

- REQUEST_TIMEOUT_IN_SECONDS: indica el timeout cuando se hacen peticiones a otros nodos. Si se supera este tiempo, la petición se marcará como fallida y un mensaje de “log” aparecerá en la consola.

- BOOTSTRAP_WAIT_IN_SECONDS: el servidor web Quart / Hypercorn no tiene la posibilidad de avisar al programador cuando ha terminado de inicializarse.

Esta variable de entorno lo que hace es configurar un tiempo de espera para empezar el proceso de “bootstrap” de nodos

- PRUNE_UNRESPONSIVE_NODES: existe código para eliminar nodos de la lista de nodos que han fallado a la hora de intentar conectar con ellos.

  Se pensó hacer pruebas y considerar posibilidades, pero no se pudo hacer por lo que por defecto esta opción está desactivada.

- WAIT_TIME_IN_SECONDS_TO_FORCE_FEDAVG: si después de entrenar, no se han recibido los modelos del resto de nodos, un temporizador fuerza el Federated Averaging después del tiempo aquí configurado más un valor aleatorio entre 0 y el valor indicado en WAIT_TIME_IN_SECONDS_TO_FORCE_FEDAVG_MARGIN.

- WAIT_TIME_IN_SECONDS_TO_FORCE_FEDAVG_MARGIN: margen máximo para activar el temporizador que fuerza el Federated Averaging con los modelos disponibles.

- RANDOM_SEED: Federated Learning impone que todos los nodos deben trabajar igual para conseguir la convergencia en los cálculos, por lo que este parámetro configura la semilla única que se utilizará en aquellas operaciones que generen números aleatorios de los entrenadores de Machine Learning.