# Paralelización del entrenamiento y validación de redes neuronales mediante CUDA

En esta práctica, se utilizará la plataforma CUDA para paralelizar el entrenamiento y la validación de redes neuronales. Para ello, es necesario comprender qué son las redes neuronales y cómo se entrenan y validan. Para este propósito, la primera sección proporciona una explicación de las redes neuronales. Luego, en la sección 2, se explicará la estructura de un proyecto realizado en lenguaje C para entrenar y validar redes neuronales[1]. Finalmente, en la sección 3, se presentará el servidor a utilizar. En dicho servidor está instalado el software SLURM para gestionar el uso de GPU, que gestionará las ejecuciones de todos los usuarios.

## 1. Redes neuronales

Una red neuronal o **Neural Network** es un paradigma de programación inspirado en el cerebro biológico [4]. Básicamente, lo que hacen es copiar el funcionamiento de las neuronas y de sus conexiones, con el mismo objetivo que otras técnicas de inteligencia artificial, programar soluciones a problemas que son difíciles de solucionar mediante algoritmos convencionales.

Las redes neuronales fueron propuestas por primera vez por Warren McCullogh y Walter Pitts de la Universidad de Chicago (EEUU) en 1944 [2], y desde entonces su importancia y popularidad en el campo de la informática ha fluctuado significativamente: tras pasar a un segundo plano en 1969, en la década de los 80 volvieron a convertirse en una importante línea de investigación, hasta, de nuevo, perder fuerza en los primeros años del milenio (2000-2010). La situación se ha revertido por completo en los últimos años, gracias al aumento generalizado de la capacidad de cómputo y al abaratamiento del *hardware*, es decir, gracias a las **GP-GPU** y a **CUDA**.

En la siguiente sección, hay una breve explicación de qué es una neurona que es la base de las redes neuronales, de cómo se organizan las redes y de cómo se realiza el proceso de entrenamiento y validación.

### 1.1. Neurona artifizialak

**Las neuronas artificiales** son los componentes básicos de las redes neuronales. En una neurona con $n$ entradas, a cada una de estas entradas se le asignará un **peso** $w_i$ , y la salida de la neurona dependerá de la suma ponderada de los pesos de entrada. A este valor se le asigna un ***sesgo*** para que el comportamiento de la neurona se pueda ajustar independientemente de la entrada. Para calcular la salida, la suma total $z$ se aplica a la **función de activación** $f$ (Ecuación 1). Dependiendo del diseño, se puede elegir entre varias funciones de activación lineales y no lineales. Las entradas y salidas de las neuronas pueden ser números enteros o reales, según cuál sea el problema a resolver.

$$y = f(z) = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

Este modelo, cuando solo hay una neurona, se llama *perceptron*, y en la Figura 1 se puede ver una representación gráfica de él.

#### 1.1.1. Funciones de activación

La función de activación más básica es una función binaria escalonada: la salida tomará el valor 1 si $z$ es mayor que un valor predefinido de *umbral*; 0 en caso contrario. El problema que surge con este tipo de función es obvio: es difícil medir el efecto de un cambio de peso (o *sesgo*) en la salida; más aún cuando desea ajustar una red completa en lugar de una sola neurona. Con grandes cambios, la salida puede permanecer igual, cuando $z$ es muy positivo o muy negativo; o invertir completamente con un pequeño cambio, de 0 a 1, cuando $z$ está cerca del valor 0.

[1] https://gitlab.com/ehp-par/nn

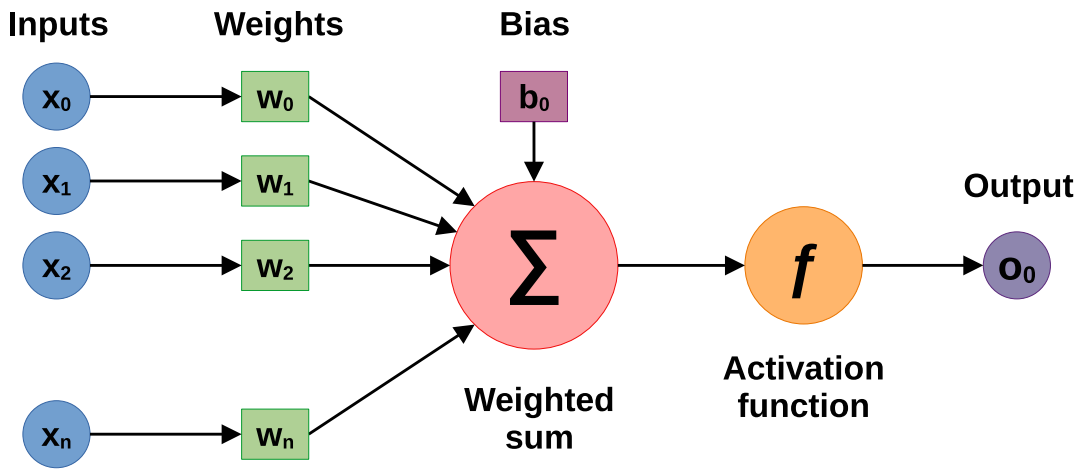


Figura 1: Representación gráfica de un *Perceptron*.

Este problema es grave cuando se entrenan redes. Como se explicará en la sección de entrenamiento, en el aprendizaje supervisado se realizan pequeños cambios en los pesos y *bias* para reducir la diferencia entre el resultado del entrenamiento de la red y el esperado, y para su mejora es necesario controlar y medir el efecto de estos cambios.

Por ello, se suelen utilizar funciones de activación más avanzadas, que tienen un cambio más suave en función de las entradas. Las definiciones de las más utilizadas, están recopiladas en el conjunto de ecuaciones 2: *Sigmoid* (2a), *Hyperbolic Tangent* (2b), *Unidad lineal rectificada* (2c), *LeakyReLU* (2d) y *Softplus* (2e).

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2a}$$

$$tanh(x) = \frac{2}{1+e^{-2x}} - 1 \tag{2b}$$

$$ReLU(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2c}$$

$$LReLU(x) = \begin{cases} -\alpha x, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2d}$$

$$soft(x) = \log(1+e^{x}) \tag{2e}$$

## 1.2. Estructura de las redes

Como sucede en biología, una sola neurona no puede ser tomada como un elemento "inteligente"; por lo tanto, para poder resolver problemas, se forman estructuras de red encadenando las salidas de las neuronas con las entradas de otras. En estas estructuras, las neuronas se organizan en ***layers*** o capas, normalmente asignando una función específica a cada capa.

Como resultado de muchos años de investigación, han surgido varias familias de redes para brindar soluciones a problemas específicos, cada una con sus características específicas; como **redes neuronales convolucionales** utilizadas para trabajar con imágenes.

Sin embargo, en todas las propuestas se suelen distinguir tres tipos de capas, según su ubicación en la red (ver Imagen 2):

- **Capa de entrada** o *Input Layer*: realiza la función de interfaz de entrada a la red. Recibe la información (datos) del problema a resolver y, cuando es necesario, la preprocesa para las siguientes capas.

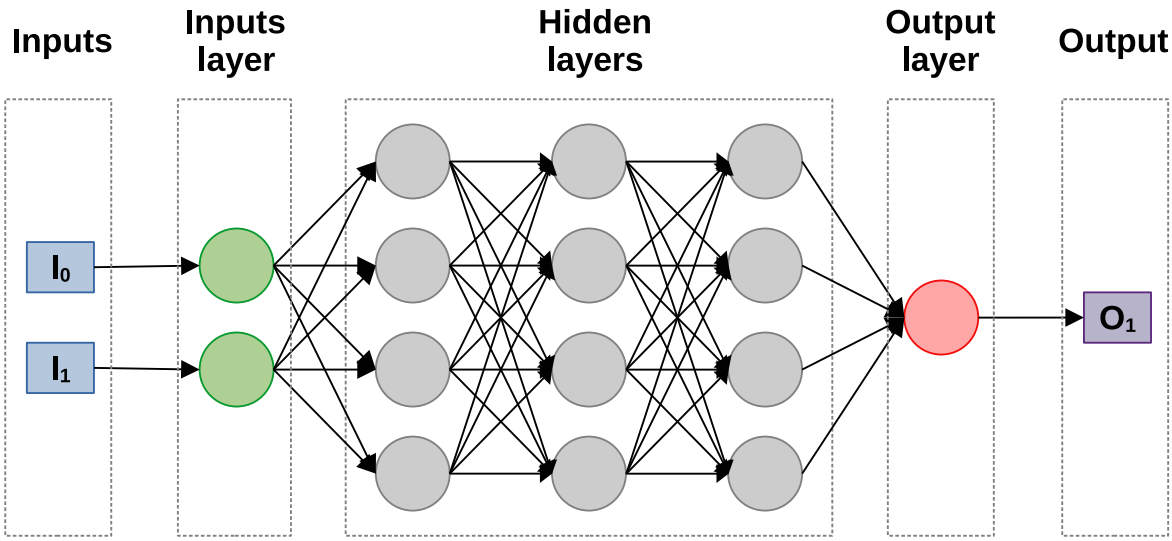


Figura 2: Estructura de una red donde se ven los tres tipos de capas.

- **Capa de salida** u *Output Layer*: la red transforma los resultados de los cálculos en información comprensible; por lo que es la interfaz de salida. También en este caso, el formato de la información cambia de un problema a otro.
- **Capa oculta** o *Hidden Layer*: estas son las demás capas entre las capas de entrada y salida. Su trabajo es extraer información significativa de los datos recopilados por la red.

Las redes, en cambio, pueden ser ***Feed Forward*** y *Recurrent* según la dirección de los enlaces entre las capas: en el primer caso, el flujo de información es unidireccional, y la entrada de una capa nunca depende de su salida; en el segundo, sin embargo, utilizando una capa intermedia, la salida de una capa se puede vincular nuevamente a su entrada. Lo explicado en este análisis teórico se aplica a las de tipo *Feed Forward* y de este tipo serán las que utilizaremos en esta práctica.

## 1.3. Entrenamiento de redes neuronales

Cuanto más similar sea la salida de una red neuronal a la esperada o al objetivo, mejor será la solución del problema proporcionada por la red. **El proceso de ajustar los pesos y los *bias*** se llama entrenamiento o ***training***. Dependiendo del tipo de problema, el aprendizaje puede ser supervisado (*supervised*), no supervisado (*unsupervised*) o una combinación de los dos (*semi-supervided*) [1]:

- *Aprendizaje* ***supervisado***: las salidas deseadas de los datos de entrada (normalmente llamados *ground truth*) se conocen, gracias a un trabajo de etiquetado o *labeling*. Durante el entrenamiento, el objetivo es hacer que la salida de la red se desvíe lo menos posible de estos resultados esperados, ajustando los parámetros de la red durante el proceso. Este método se utiliza con mayor frecuencia en clasificación y detección, predicciones y pronósticos (clasificación de imágenes, texto o voz, pronósticos meteorológicos...).
- *Aprendizaje* ***no supervisado***: la red intenta extraer patrones ocultos de un gran conjunto de datos. Para ello, explota las similitudes entre los datos, sin necesidad de un etiquetado manual previo. Son muy útiles para (*clustering*, p. ej. para agrupar clientes en una tienda), asociación (*asociation*, para recomendar canciones similares) o para reducir el número de dimensiones de los datos (para preprocesamiento de datos).
- *Aprendizaje* ***semi-supervisado***: la red utiliza datos etiquetados y no etiquetados juntos. Se puede utilizar para evitar el tener que etiquetar completamente las bases de datos, para extraer ciertas características de los datos, para refinar la precisión, etc., y la forma en que se implementa varía de un caso de uso a otro.

Una vez finalizado el entrenamiento se deben guardar los pesos y sesgos (*bias*)obtenidos. A este conjunto de números se le denomina modelo y se utilizará en la fase de validación para medir su calidad. Además, cuando el modelo tenga la calidad requerida, se utilizará para realizar inferencia, es decir, se utilizará para clasificar nuevos casos.

## 1.4. Función de coste

Para medir la diferencia entre la salida de la red y el valor real se utiliza la llamada función de coste o *cost-function*[2]. La función se aplica a la diferencia (calculada directamente) entre la salida de la red y la esperada; por lo tanto, el coste depende implícitamente de los pesos y del *bias*.

Para medir la precisión de la clasificación, MSE (*Mean Square Error*), MAE (*Mean Absolute Error*) y *Logarithmic Loss* son las funciones de coste más utilizadas. Por ejemplo, para cada dato de entrada $i$, donde $y(i)$ es el vector de salida de la red y $y^*(i)$ es la salida etiquetada de la entrada (la clase correspondiente es 1, el resto son 0 ), la matriz de pesos $W$ y $B$ es el vector de *bias*, la ecuación 3 muestra el valor MSE para $n$ entradas.

$$C(W,B) = \frac{1}{2n}\sum_{x=1}^{n} ||y^*(x) - y(x)||^2 \tag{3}$$

## 1.5. Algoritmo Backpropagation

La función de coste que se calcula a partir de la salida de la red depende de todos los pesos y *bias*. Debido a ello, no es factible buscar analíticamente la combinación óptima (de pesos y *bias* en una red con miles de parámetros. En su lugar, se deben utilizar técnicas que exploren solo un subconjunto de los posibles valores que pueden tomar los parámetros.

Traducido literalmente, el nombre del algoritmo, ***error backpropagation***, significa propagar el error hacia atrás. Fue propuesto en la década de 1970 y desde 1986 es el algoritmo más utilizado para entrenar redes neuronales. En el libro [4] aparecen las cuatro ecuaciones que se muestran en el conjunto de ecuaciones 4 como la base del algoritmo *backpropagation*[3].

El efecto de la suma de cada neurona $j$ en la capa $l$ sobre la función de coste de la red, $\delta_j^l$, se mide por la derivada parcial de la función $C$ con respecto a $z_j^l$ (Ecuación 4a). Las derivadas parciales de la última capa forman el vector $\nabla_a C$. Por otro lado, el error de entrada de una capa ($\delta^l$) dependerá del error de la siguiente, la derivada de su activación y de los pesos entre las dos capas (Ecuación 4b). Esta segunda ecuación permite que el error se "mueva" de atrás a adelante por la red. Finalmente, es posible analizar cómo varía el costo según cualquier *bias* o peso: por definición, el efecto de un cambio de *bias* sobre el coste total es igual al error de la neurona correspondiente (Ecuación 4c); el efecto del peso conectando $k$ neuronas de la capa $l-1$ y $j$ neuronas de la capa $l$ dependerá del error de la neurona $j$ de la capa $l$ y de la activación de la neurona $k$ de la capa anterior (Ecuación 4d).

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{4a}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{4b}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{4c}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{4d}$$

En el algoritmo 1 es posible ver cómo se aplican las ecuaciones para ajustar los parámetros de una red. Los datos se introducen en la capa entrada y se recorre la red hasta obtener las salidas; con estas salidas se

[2]En la literatura también se le llama función de pérdida y función objetivo; *loss-function* y *objetive-function*.

[3]El capítulo 2 del libro está dedicado a explicar el algoritmo. Para una comprensión más profunda de la base matemática, consultar: *Capítulo 2: Cómo funciona el algoritmo de retropropagación.*

**Algorithm 1:** Algoritmo *Backpropagation*

**Input:** $X = \{x_1, ..., x_n\}$, Conjunto de datos para el entrenamiento
**Input:** $\sigma(z)$, Función de activacion de las neuronas
**Input:** $\sigma'(z)$, Primera derivada de la funcion de activación
**Input:** $f(x)$, Funcion de activación de las neuronas de entrada
**Input:** $W = (w^1 ... \ w^L)$, Matriz de pesos
**Input:** $B = (b^1 ... \ b^L)$, Matriz de *bias*
**Input:** $\eta$, Coeficiente de aprendizaje
**Input:** $L$, Número de capas
**Output:** $W$ y $B$ actualizados

**for** $i \leftarrow 1$ ***to*** $n$ **do**
  $x \leftarrow x_i$
  $a^{1,i} \leftarrow f(x)$ // Activación de la prmera capa a partir de los datos de entrada
  // Feedforward
  **for** $l \leftarrow 2$ ***to*** $L$ **do**
    $z^{i,l} \leftarrow w^l a^{i,l-1} + b^l$
    $a^{i,l} \leftarrow \sigma(z^{i,l})$
  **end**
  $\delta^{i,L} \leftarrow \nabla_a C_i \odot \sigma'(z^{i,L})$ // Calcular el error de la última capa
  // Backpropagation
  **for** $l \leftarrow L - 1$ ***to*** $2$ **do**
    $\delta^{i,l} \leftarrow ((w^{l+1})^\top \delta^{x,l+1}) \odot \sigma'(z^{i,l})$
  **end**
**end**
// Stochastic Gradient Descent
**for** $l \leftarrow L$ ***to*** $2$ **do**
  $w^l \leftarrow w^l - \frac{\eta}{n} \sum_{i=1}^{n} \delta^{i,l} (a^{i,l-1})^\top$
  $b^l \leftarrow b^l - \frac{\eta}{n} \sum_{i=1}^{n} \delta^{x,l}$
**end**

calcula el error de la última capa; y el error se propaga hacia atrás de forma ponderada. En cada capa se puede usar este error para ajustar los parámetros, *Gradient Descent* [4]

Cuando el volumen de datos es grande, en lugar de considerar la base de datos completa, los datos generalmente se le entregan al algoritmo en lotes llamados ***mini batch***, y los parámetros se actualizan en función de los datos en ese batch[5]. En cada iteración, se selecciona un conjunto de datos de la base de datos, y cuando se han utilizado todas las instancias, se dice que ha pasado una ***época***.

## 1.6. Ajuste de hiperparámetros

Los parámetros de una red son el *bias* de sus neuronas y los pesos de las conexiones, especialmente adaptados a los datos disponibles, después del entrenamiento. Los hiperparámetros, en cambio, son aquellos que se sitúan antes del entrenamiento, que definen las características de la red y el proceso de entrenamiento: la topología o arquitectura de la red (número de neuronas y capas), el coeficiente de aprendizaje para actualizar los parámetros, el tamaño del batch, el número de *epochs*, etc.

[4] En cada iteración se usa el gradiente de la función de coste (o una aproximación de la misma) para actualizar los parámetros con el objetivo de acercarse al local mínimo usando algún criterio de actualización como puede ser [3].

[5] Cuando se utilizan gradientes estimados usando un subconjunto o muestra, se le llama *Descenso de gradiente estocástico* [3].

Hoy en día, se han propuesta múltiples topologías de red o familias de topologías, y en las aplicaciones prácticas se elige la que mejor se adapta. La disposición de capas y neuronas, funciones de activación, etc. están definidas en la propia arquitectura.

Los hiperparámetros relacionados con el entrenamiento necesitan ser controlados para buscar un trade-off entre precisión y velocidad ("¿Cuánto tiempo hay para entrenar?"), el tamaño de cada *batch*, el número de ***epochs*** y el ***learning-rate***. Este último también tendrá un efecto en evitar los óptimos locales. Los hiperparámetros apropiados se eligen a través de métodos heurísticos: basándose en la experiencia previa o la literatura; con la técnica *grid search* que analiza conjuntos discretos de combinaciones; con búsquedas heurísticas más avanzad, etc.

Estos hiperparámetros son importantes para evitar los problemas que aparecen en el entrenamiento de muchos tipos de algoritmos de *Machine Learning* (incluyendo redes neuronales):

- ***Overfitting*:** los parámetros se han sobreajustado a los datos de entrenamiento y el modelo ha perdido generalidad. En lugar de extraer las características del problema a resolver, ha aprendido solo los ejemplos proporcionados, y aunque da muy buenos resultados con esos datos, no funcionará tan bien con casos nuevos.
- ***Underfitting*:** la red se ha entrenado menos de lo necesario y aún no ha podido aprender a extraer información significativa de los datos.

## 1.7. Métricas para la validación

Ikasitako ereduaren kalitatea ebaluatzeko erabiltzen diren teknika eta metrika ugarien artetik, atal honetan azaldutakoak dira praktikan erabiliko direnak.

Entre las muchas técnicas y métricas utilizadas para evaluar la calidad de los modelos aprendidos, las descritas a continuación son las que se utilizarán en esta práctica.

Teniendo en cuenta las siguientes definiciones:

1. Tenemos dos clases: la clase 0 será la negativa (Negative) y la clase 1 será la positiva (Positive).
2. Una entrada debe clasificarse en la clase 0, pero el clasificador la clasifica en la clase 1. A esto se le llama *False Negative* ($FN$).
3. Una entrada debe clasificarse en la clase 0 y el clasificador la clasifica en la clase 0. A esto se le llama *True Negative* ($TN$).
4. Una entrada debe clasificarse como clase 1, pero el clasificador la clasifica como clase 0. A esto se le llama *False Positive* ($FP$).
5. Una entrada debe estar clasificada en la clase 1 y es clasificada en la clase 1. A esto se le llama *True Positive* ($TP$).

A partir de estas medidas, se pueden calcular las métricas ***Precisión*** o $P$ (Ecuación 5) y ***Recall*** o $R$ (Ecuación 6). La primera mide el porcentaje de aciertos a partir de las clasificaciones realizadas por la red; el segundo, cuántas de las clasificaciones que se deben hacer se han hecho. El equilibrio entre los dos valores se tiene en cuenta por la métrica $F_1$ (Ecuación 7) de la familia de métricas ***F-score***.

$$P = \frac{TP}{TP + FP} \tag{5}$$

$$R = \frac{TP}{TP + FN} \tag{6}$$

$$F_1 = 2 \times \frac{TP}{TP + \frac{1}{2}(FP + FN)} = 2 \times \frac{P \times R}{P + R} \tag{7}$$

# 2. Entrenamiento de redes neuronales en CPU

En el siguiente repositorio de Gitlab[6] disponéis de un proyecto desarrollado en C lenguaje para entrenar redes neuronales. En esta sección se describirá la estructura de dicho proyecto y las funcionalidades ofrecidas por cada uno de los módulos.

## 2.1. Estructura del proyecto

El directorio *src* contiene 7 archivos ".c". A continuación se describe la funcionalidad principal de cada uno de ellos:

- **main**: Contiene la función principal y se encarga de la lectura de parámetros y datos. Además, según los parámetros pasados por línea de comandos, realiza el entrenamiento exportando el modelo aprendido o realiza la evaluación importando un modelo previamente aprendido.
- **ds**: Este fichero contiene las funciones necesarias para la lectura de los datos (dataset). El dataset debe estar en un fichero de tipo cvs, las características en las primeras columnas y en la última la clase a la que corresponde a este elemento.
- **nn**: Este módulo implementa las funciones de creación, exportación e importación de redes neuronales. Además, tiene las funciones necesarias para realizar el proceso de entrenamiento y validación en la CPU.
- **nn_aux**: Funciones auxiliares para la construcción de una red neuronal. Este módulo implementa, entre otras, las funciones de coste y las funciones de activación.
- **train**: Funciones utilizadas para realizar el proceso de aprendizaje. Este módulo contiene 3 funciones: *feedforward*, *backpropagation* y *Stochastic Gradient Descent* (update).
- **test**: Funciones de validación y métricas para medir la calidad del modelo.
- **utils**: Funciones auxiliares.
- **matrix**: Varias funciones para realizar operaciones con matrices y vectores.

## 2.2. Parámetros del proyecto

A continuación se muestran los parámetros que pueden utilizarse para configurar una ejecución. Existen dos ejemplos en el fichero README.MD.

- −−verbose: Información útil que sale por la salida estandar durante el proceso de aprendizaje o validación.
- −−seed: Semilla que se utiliza en el proceso de aprendizaje.
- −−train: En la ejecución se aprenderá un modelo.
- −−test: En la ejecución se evaluará un modelo.
- −−layers: Representa la estructura de la red neuronal que se utilizará en el aprendizaje.
- −−epochs: Número de *epochs* que se utilizarán para aprender el modelo.
- −−batch_number: Tamaño de los *batches* que se utilizarán para aprender el modelo.
- −−learning_rate: Parámetro que se utiliza en el proceso de aprendizaje.
- −−dataset: Dataset que se utilizará en el proceso de aprendizaje o de validación.
- −−model: Nombre del fichero en el que se guardará el modelo aprendido. Si se hace validación, representa el fichero que se cargará y que contiene el modelo.

[6] https://gitlab.com/ehp-par/nn

## 3. Servidor para medir el rendimiento

Una vez que se implemente el código CUDA, será necesario medir el rendimiento. Para hacer esto, utilizaremos un servidor[7] que dispone de 2 tarjetas gráficas Nvidia RTX A4000. La dirección de ese servidor es: *U107949.ehu.es*.

Ese servidor usa el gestor de colas SLURM para organizar las ejecuciones de los trabajos de los usuarios. Para usar este software, puede encontrar el script que usaremos en el repositorio (*slurm.sbatch*). Lo único que hay que configurar en ese script es, por un lado, la opción #SBATCH –gpus=X, para usar 1 o 2 GPU, y, en la última línea, el comando que se desea ejecutar.

Una vez configurado el script, se debe usar el siguiente comando para enviar la ejecución a SLURM: **sbatch slurm.sbatch**. Una vez enviado, el comando *squeue* mostrará el estado del trabajo enviado.

## 4. Objetivo de la práctica y entrega

El objetivo de esta práctica es paralelizar el proceso de aprendizaje y validación de una red neuronal utilizando CUDA. Para ello, las funciones en los módulos train.c y test.c deben ser "traducidas" a CUDA. Además, necesitaremos nuevas funciones train(...) y test(...) en el módulo nn.c para implementar las transferencias necesarias en CUDA y en los módulos train.c y test.c implementar los kerneles necesarios.

Una vez desarrollado el código, se debe redactar un breve informe explicando la estrategia seguida para la paralelización. Además, utilizando el dataset CC (ubicado en el directorio datasets), se debe indicar la aceleración obtenida en comparación con la versión que utiliza la CPU. Debes publicar todo el código en un repositorio git (indicar la dirección en el informe). Se debe subir un pdf a Egela para el 22 de enero.

## Referencias

[1] Julianna Delua. Supervised vs unsupervised learning. `https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning`. 2021/03/12.

[2] Larry Hardesty. Explained neural networks and deep learning. *MIT News*.

[3] Microsoft Research Leon Bottou. Stochastic gradient descent tricks. `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/tricks-2012.pdf`.

[4] Michael Nielsen. Neural Networks and Deep Learning. `http://neuralnetworksanddeeplearning.com`.

[7] Asignamos las cuentas en clase, si necesitas una escribe al profesor.