UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Adaptación de algoritmos de aprendizaje automático para su ejecución sobre GPUs

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Victor Grau Moreso

**Tutor**: Jon Ander Gómez Adrian

2014-2015

# Resumen

El presente trabajo de final de grado implementa redes neuronales en Unidades de Procesamiento Gráfico (GPU) manejadas con tecnología CUDA bajo C++ para efectuar el reconocimiento de cifras manuscritas. El sistema utilizado demuestra ser eficiente y aplicable a múltiples programas de Inteligencia Artificial.

**Palabras clave:** red neuronal, GPU, CUDA, Inteligencia Artificial.

# Resum

El present treball de final de grau implementa xarxes neuronals en Unitats de Processament Gràfic (GPU) manejades amb tecnología CUDA sota C++ per a efectuar el reconeixement de xifres manuscrites. El sistema utilitzat demostra ser eficient i aplicable a múltiples programes d'Intel·ligència Artificial.

**Paraules clau:** xarxa neuronal, GPU, CUDA, Intel·ligència Artificial.

# Abstract

This project implements neural networks in Graphical Processing Units (GPU) managed with CUDA technology under C++ in order to recognize handwritten digits. The system used proves to be efficient and applicable to multiple Artificial Intelligence programs.

**Keywords:** neural network, GPU, CUDA, Artificial Intelligence.

# Agradecimientos

A mi familia, por todo el apoyo recibido a lo largo de estos años.

A Lluís, Javier, Álvaro y Mark, por todas las buenas experiencias que hemos compartido.

A mi tutor Jon, tanto por la orientación impartida en este proyecto como por sus enseñanzas, en especial en este último curso de carrera.

# Tabla de contenidos

# 1. Introducción

En este apartado se expone el problema que ha motivado la realización de este proyecto, se define su objetivo y se listan los componentes usados en su desarrollo.

## 1.1 Motivación y análisis preliminar

La motivación para realizar el presente proyecto ha sido abrir la posibilidad de extender el uso de las redes neuronales a unas velocidades adecuadas en nuestros propios ordenadores personales.

Las Redes Neuronales son un modelo de aprendizaje estadístico usado para aproximar funciones desconocidas a priori que normalmente dependen de un gran número de entradas. Puesto que se trabaja con grandes cantidades de datos y las operaciones realizadas suelen ser en coma flotante, el coste computacional de entrenar y perfeccionar una red neuronal es muy elevado [1] y puede llevar días o incluso semanas de proceso.

La mayor parte de las operaciones realizadas en el entrenamiento de una red neuronal son independientes entre sí (en concreto las operaciones de cada neurona de una capa son independientes de las del resto de neuronas de la misma capa), por lo que si cada operación independiente se ejecutase a la vez que las demás, el tiempo de ejecución final se vería drásticamente reducido.

Para implementar redes neuronales eficientes hay que recurrir, o bien a la programación paralela, o bien a la programación distribuida. Puesto que el número de unidades de procesamiento independiente de un microprocesador no es muy elevado [2], la programación distribuida parece ser la mejor opción [3]. No obstante, esto supone un aumento significativo en la complejidad de la programación [4] y en los costes de adquisición del sistema [5].

Las unidades de procesamiento gráficas (GPU) son dispositivos de bajo coste [6] con centenares de hilos de ejecución simultánea preparados para operar con números reales y por ello se adaptan perfectamente a nuestro propósito. La programación de estos dispositivos se

ha realizado en lenguajes de programación de bajo nivel durante mucho tiempo, pero desde hace algunos años existen plataformas que permiten el uso de lenguajes de alto nivel en las GPU [7], reduciendo en gran medida la complejidad y el tiempo de desarrollo.

Resulta por tanto de interés el desarrollo de una red neuronal sobre una GPU en un lenguaje de alto nivel.

## 1.2 Objetivo

El objetivo de este trabajo es comprobar la viabilidad del uso de GPU para el procesamiento de redes neuronales. Para ello se ha hecho uso de una plataforma que permite la ejecución de código en estos dispositivos y se ha escogido un conjunto de datos amplio para poder apreciar la velocidad del sistema desarrollado.

## 1.3 Planificación del proyecto

Para llevar a cabo este proyecto, se ha debido elegir:

- Un tipo de red neuronal, la Máquina de Boltzmann Restringida.
- Los algoritmos para el aprendizaje de la red neuronal. Se ha usado el algoritmo *Contrastive Divergence* para la primera fase conocida como *Unsupervised Pretraining* (preentrenamiento), el *Backpropagation* con la red desdoblada para reconstruir la misma entrada, lo que se conoce como *Autoencoder*, para la segunda fase de preentrenamiento. Finalmente el *Backpropagation* para el ajuste final utilizando la salida que se desea para la red, esta última fase se conoce como *Supervised Finetuning*.
- Una plataforma para ejecutar código de alto nivel en la GPU, en este caso CUDA de nVidia.
- Un lenguaje de alto nivel soportado por CUDA. Hemos elegido C++ por su eficiencia y versatilidad.
- Un conjunto de datos sobre los que efectuar las pruebas. Dado su extenso uso en el campo de la clasificación automática, nos hemos decantado por la base de datos de cifras manuscritas de MNIST.

# 2. Componentes utilizados

En este apartado se detallan los distintos componentes básicos de nuestro proyecto con el objetivo de facilitar la comprensión de las fases posteriores del trabajo.

## 2.1. CUDA

CUDA es una plataforma de computación paralela y un modelo de programación que facilita el uso de una Unidad de Procesamiento Gráfico (GPU) para computación de propósito general. Se programa en uno de los lenguajes soportados (C, C++, Fortran...), usando extensiones de estos lenguajes mediante unas pocas palabras clave básicas.

Las GPU superan en el número máximo de operaciones en coma flotante por segundo (FLOP/s) a las Unidades de Procesamiento Central (CPU) en varios órdenes de magnitud. Esto se debe a que la arquitectura de las GPU maximiza el número de unidades de procesamiento, sacrificando cache de datos y control de flujo [8], partes predominantes en las CPU (véase figura 2.1). Esto no obstante reduce su uso a resolución de problemas en los que un mismo programa se ejecuta sobre muchos elementos independientes entre sí a la vez, dado que no posee un control de flujo sofisticado. Al no poseer apenas caché, para compensar la latencia de los accesos a memoria [9] las operaciones realizadas sobre los datos deben tener una alta intensidad aritmética, reduciendo así el efecto de dicha latencia.

Aunque las GPU fueron creadas con el propósito de hacer cálculos gráficos (cada píxel de la pantalla puede ser procesado en un hilo diferente), se puede aprovechar su alta potencia para computación de propósito general, puesto que la mayor parte del tiempo de uso regular de los ordenadores personales prescinde de una gran parte de la capacidad de éstos componentes.

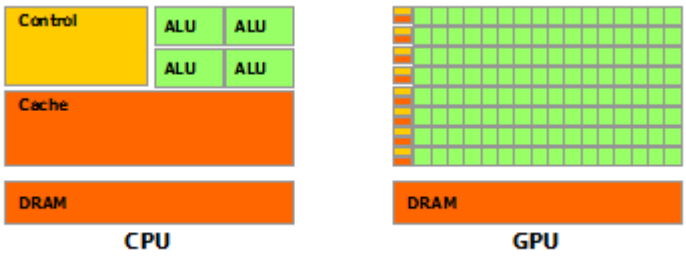


Figura 2.1: Arquitecturas generales de CPU y GPU

La ejecución de código en GPU con CUDA se realiza mediante "`kernels`", que son funciones C precedidas por la etiqueta "`__global__`", y llamadas con el operador "`<<<..., ...>>>`" (ejemplo: "`KernelA<<< numeroDeBloques, hilosPorBloque >>>(x)`").

Los parámetros especificados en el operador especial de los kernels hacen referencia a la organización de CUDA de los hilos paralelos: se genera un número `N` de bloques, cada uno formado por `M` hilos, de forma que cada bloque puede ser ejecutado por un Multiprocessor Streaming (MS) cada vez. Las GPU están formadas por una serie de MS, por lo que al dividir de esta manera los hilos a ejecutar, una GPU con más MS automáticamente ejecutará el programa en menos tiempo que una con menos MS al poder ejecutar un mayor número de bloques en paralelo.

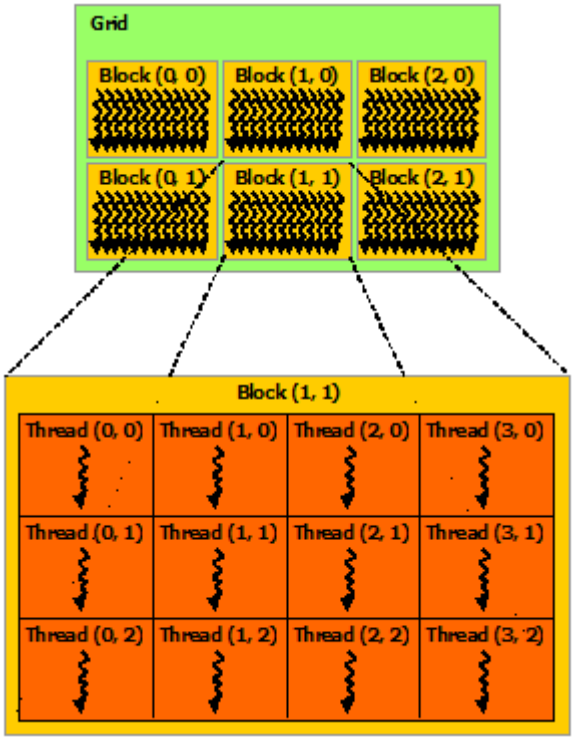


Figura 2.2: Esquema de organización de hilos por bloques

Puesto que las llamadas a kernels se encuentran en el código que ejecuta la CPU, una vez efectuadas se prosigue con el programa sin esperar ningún tipo de respuesta de la GPU. Esto permite que mientras la GPU está ocupada ejecutando la función kernel en paralelo, la CPU pueda seguir efectuando procesos, sin tener que esperar a que la GPU finalice. Si llegamos a un punto del programa en el que requerimos de los resultados de la función kernel ejecutada, CUDA se encarga de poner una barrera en dichas acciones de forma que el proceso de la CPU se detendrá hasta que la GPU haya finalizado la ejecución del anterior kernel.

## 2.2. Máquina de Boltzmann Restringida

Una máquina de Boltzmann es un tipo de red neuronal recurrente estocástica. Las máquinas de Boltzmann sin restricciones de conectividad presentan un grave problema práctico: el aprendizaje deja de producirse correctamente cuando la máquina se amplía a algo más grande que una máquina trivial. Es ahí donde entran en juego las máquinas de Boltzmann restringidas (RBM, por sus siglas en inglés), que eliminan las conexiones entre neuronas de la misma capa.

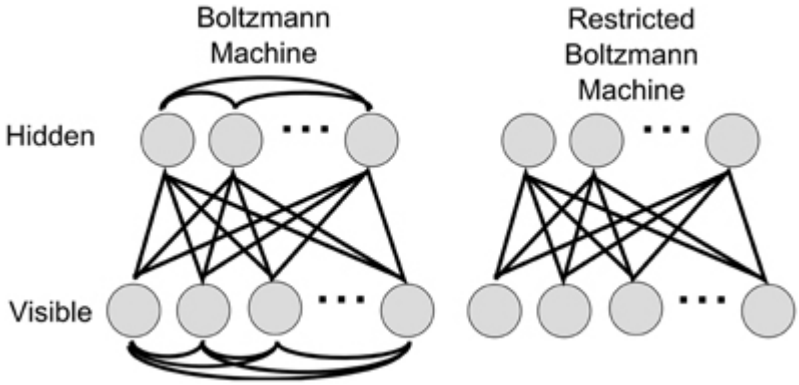


Figura 2.3: Máquna de Boltzmann y Máquina de Boltzmann Restringida

Si tras entrenar una Máquina de Boltzmann restringida tratamos las unidades ocultas como datos de entrada para una nueva RBM, podemos crear una red neuronal de múltiples capas que podemos entrenar y obtener resultados satisfactorios pese a la complejidad de la red [10].

Para el entrenamiento de una RBM es recomendado empezar con el algoritmo de ascenso por gradiente Contrastive Divergence [11]. Éste entrenamiento capa a capa no supervisado sirve como inicialización de las matrices de pesos, por lo que puede considerarse un preentrenamiento.

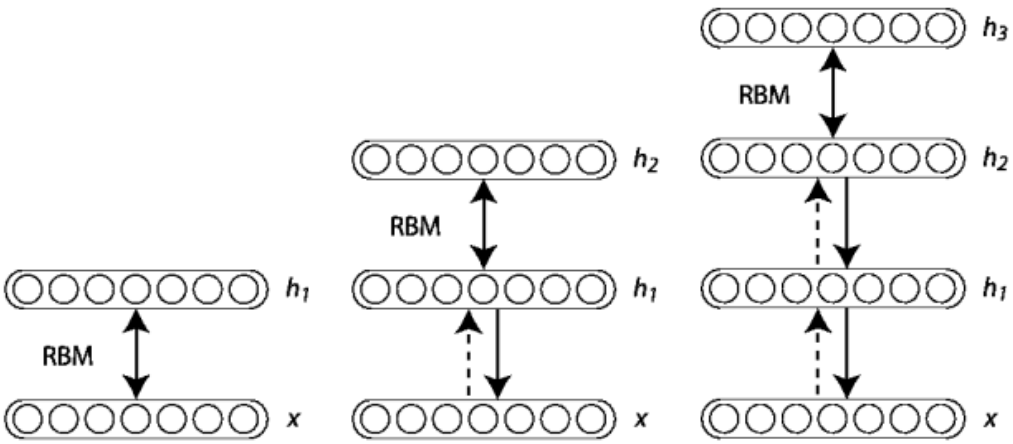


Figura 2.4: Entrenamiento capa a capa de una RBM

## 2.3. MNIST

La base de datos MNIST (Mixed National Insitute of Standards and Technology database) consiste en un conjunto de dígitos (del 0 al 9) manuscritos guardados como imágenes en niveles de gris a una resolución 28x28 pixeles con etiquetas que indican el valor numérico de dichas imágenes [12].

La base de datos consta de cuatro archivos:

- 2 para entrenamiento, uno con 60.000 imágenes y el otro con las 60.000 etiquetas asociadas.
- 2 para evaluación, uno con 10.000 imágenes y otro con sus correspondientes 10.000 etiquetas.

Esta base de datos es ampliamente utilizada en el entrenamiento de sistemas de reconocimiento de imágenes.

Figura 2.5: Muestra de la base de datos MNIST

Para efectuar la prueba de MNIST, deberán usarse las imágenes de entrenamiento y sus correspondientes etiquetas para corregir repetidamente el sistema que estemos entrenando de forma que para una imagen de entrada el sistema la clasifique correctamente en la etiqueta correspondiente. Una vez hecho el entrenamiento (con tantas repeticiones como deseemos) se procede a usar el sistema empleado para la clasificación de las imágenes de evaluación, guardando el número de errores cometidos y dividiéndolo por el número de casos.

El sistema de clasificación a utilizar no está restringido de ningún modo, pudiendo observar en la web oficial[1] que en pruebas efectuadas hasta la fecha se han usado desde simples clasificadores lineales hasta Máquinas de Vectores Soporte o Redes Neuronales.

1: http://yann.lecun.com/exdb/mnist/

## 2.4. Algoritmos

En el proyecto utilizamos tres algoritmos principales, dos para pre-entrenamiento y uno para entrenamiento de la red neuronal implementada.

### 2.4.1. Contrastive Divergence

Es un algoritmo de ascenso por gradiente usado para entrenar redes neuronales capa a capa de forma no supervisada, es decir, sin necesidad de etiquetas que identifiquen las entradas. Su uso es especialmente recomendado en las Máquinas de Boltzmann Restringidas (RBM) dada su eficacia en dicho tipo de red neuronal.

El primer paso en la ejecución de este algoritmo es propagar la capa de entrada (*forwarding*) hasta la capa en la que vamos a ejecutar el algoritmo. Una vez disponemos de datos de entrada en la capa visible de la RBM a entrenar, efectuaremos un *forwarding* más para obtener datos en la capa oculta. Llamaremos a esta capa visible $V_0$ y a la oculta $H_0$. Procedemos a efectuar la operación inversa sobre $H_0$, lo que nos deja con una capa de las mismas características que $V_0$ pero con diferentes valores, a la que llamaremos $V_1$. Al efectuar un avance sobre $V_1$, obtendremos otra capa oculta $H_1$. Este procedimiento se conoce como Gibbs Sampling [13], y puede seguir efectuándose tantas veces como se desee.

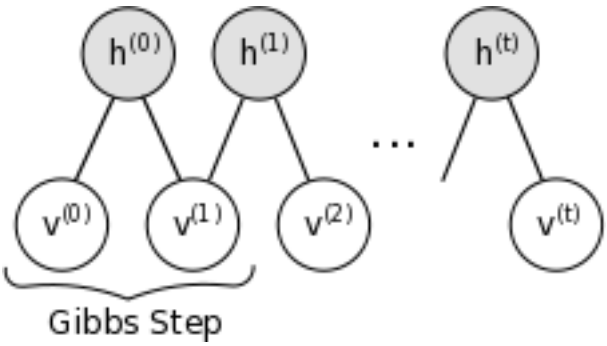


Figura 2.6: Gibbs Sampling

Una vez disponemos de V0, H0, Vx y Hx podemos obtener el incremento de la matriz de pesos entre la capa visible y la capa oculta sobre las que estamos trabajando de forma trivial:

$\Delta W = H0 * V0 - Hx * Vx$

Por último, añadimos a la matriz de pesos el producto del factor de aprendizaje por $\Delta W$.

Pueden añadirse repeticiones al algoritmo, las cuales también se efectuarán capa a capa (n repeticiones sobre la primera capa, a continuación de n repeticiones sobre la segunda capa, y así sucesivamente).

### 2.4.2. Autoencoder

Al igual que el Contrastive Divergence, este algoritmo se utiliza como fase previa al entrenamiento, y prescinde de la capa de salida en su ejecución.

El primer paso es crear una nueva red neuronal a partir de la red neuronal dada "desdoblando" la estructura de forma que si nuestra topología consistía en 3 capas V → H1 → H2, la nueva red está formada por 5 capas V → H1 → H2 → H1' → V', donde V' y H1' son dos capas de las mismas características que V y H1 respectivamente. Siguiendo el ejemplo, las matrices de pesos H2 → H1' y H1' → V' son las transpuestas de H1 → H2 y V → H1, y un cambio en la matriz transpuesta afecta a la matriz no transpuesta y viceversa.

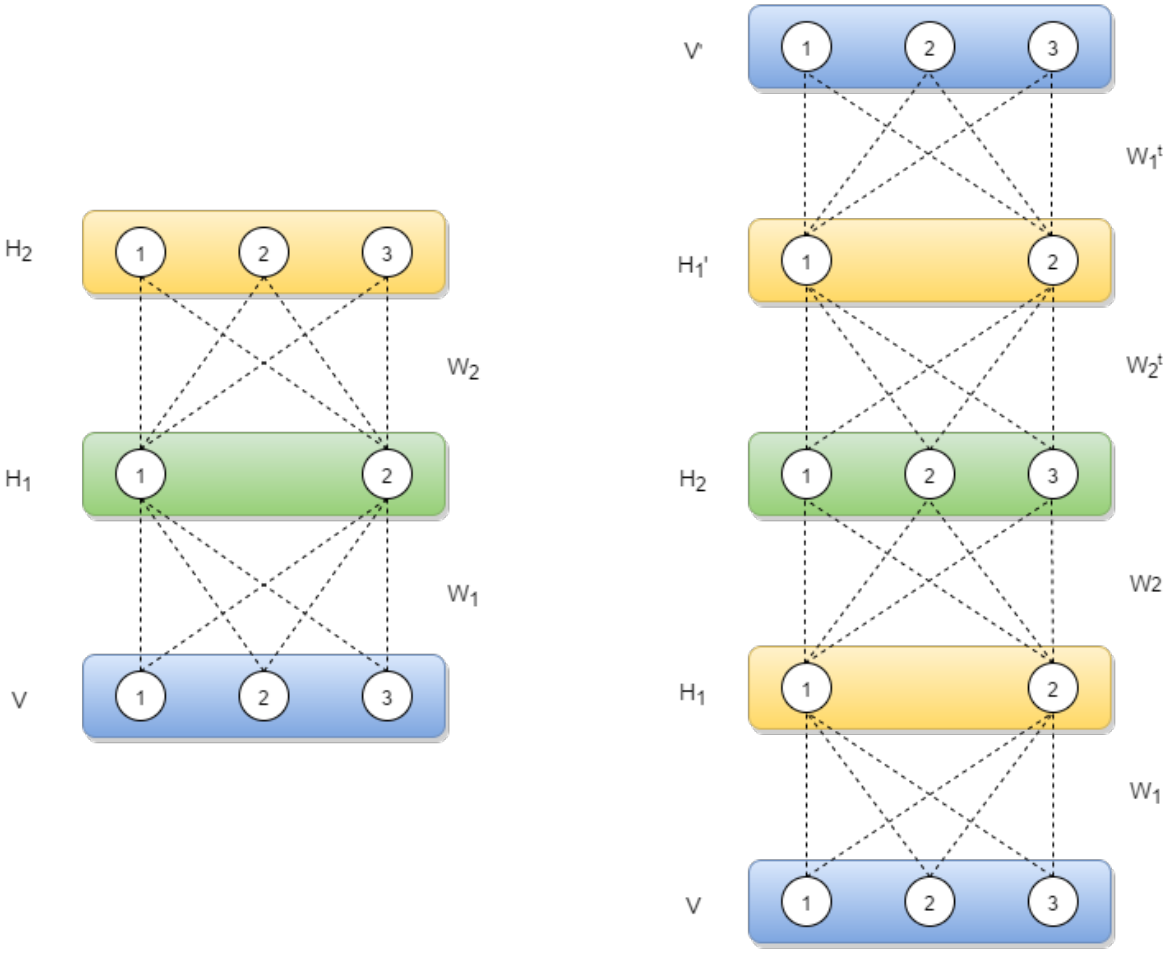


Figura 2.7: Desdoblado Autoencoder

Una vez tenemos la nueva red, tan solo hay que propagar la entrada hasta la última capa usando *forwarding* y aplicar posteriormente el algoritmo (que a continuación detallaremos) de retropropagación del error usando los valores de la entrada como valores de salida deseados.

### 2.4.3. Retropropagación del error

Este algoritmo es uno de los más usados para el entrenamiento de redes neuronales. Requiere de una salida deseada para cada entrada para así calcular el gradiente de una función de pérdida con respecto a los pesos de la red.

Comenzamos realizando una propagación de los datos de la capa de entrada hasta la capa de salida, es decir, un *forwarding* completo. Con los valores de salida obtenidos y los deseados, calculamos el error multiplicando la diferencia entre las salidas por la derivada de la función de activación de los valores propagados (sin haber aplicado sobre ellos la función de activación).

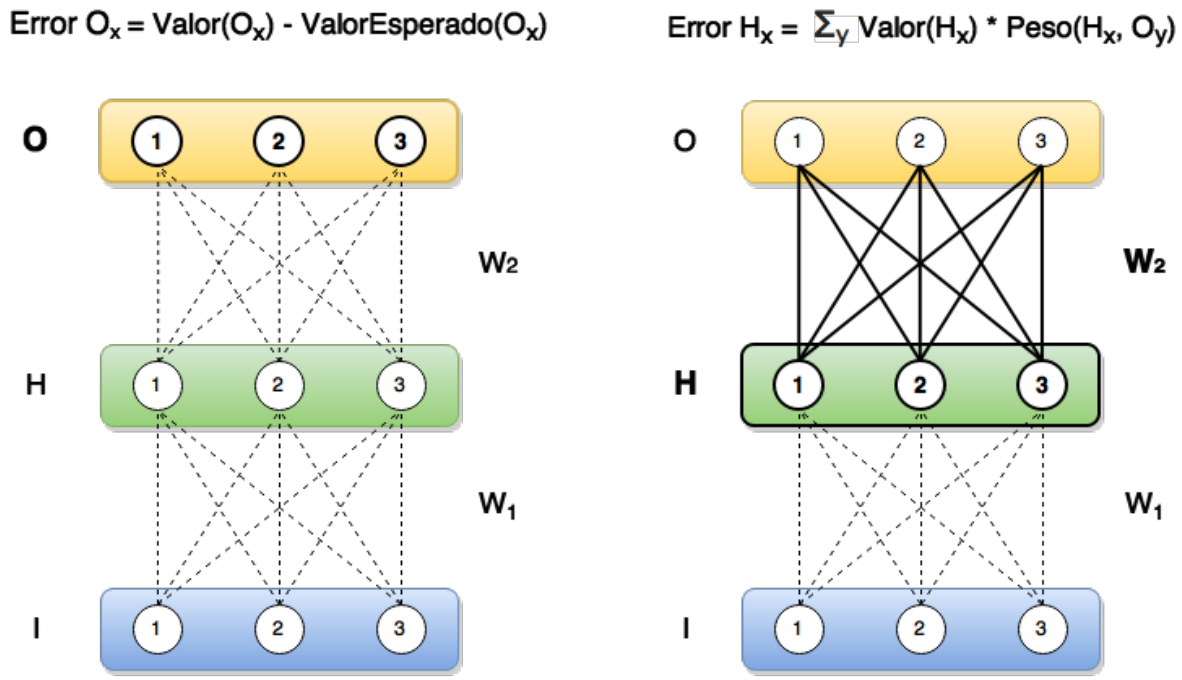


Figura 2.8: Retropropagación del error

Una vez obtenido el error de la capa de salida, éste es propagado hacia las capas anteriores multiplicando el error de cada neurona de la capa de salida por el peso de cada una de las neuronas de la capa anterior. El error de una neurona de la capa anterior es la suma de los resultados de los productos que corresponden a los pesos de dicha neurona (ver figura 2.8).

El incremento de una matriz de pesos se obtiene multiplicando el factor de aprendizaje por el producto del vector de errores de la capa oculta por el vector de valores propagados mediante *forwarding* de la capa visible.

Para actualizar los pesos, basta con restar el incremento obtenido ($\Delta$W) a la matriz de pesos.

# 3. Desarrollo del proyecto

En este apartado se explica en detalle el proyecto realizado, así como la estructura de éste y las distintas funciones implementados.

## 3.1. Asignación de memoria con CUDA

Dado que los hilos de la GPU acceden a su propia memoria global, hay que asegurarse de reservar el espacio requerido y copiar los datos que necesitemos de la memoria de la CPU a la de la GPU. Para esto CUDA facilita dos funciones: `cudaMalloc()` y `cudaMemCpy()`.

`cudaMalloc()` funciona de forma similar al `malloc()` de C/C++, requiriendo el espacio de memoria a reservar y, en el caso de `cudaMalloc()`, una variable puntero donde guardar la dirección al inicio de la memoria reservada. Esta dirección pertenece a la memoria de la GPU, por lo que no podemos interactuar con ella de forma convencional. Tan solo podremos usar `cudaMemCpy()` para copiar el contenido (o parte de él) de una dirección de memoria local (CPU) a la dirección de memoria del dispositivo (GPU).

Esto genera el problema de cómo resolver los punteros multinivel, dado que no podemos operar con una dirección de `cudaMalloc()` directamente. Para resolverlo, reservamos el espacio del puntero de primer nivel, y uno del mismo tamaño en memoria local. A continuación, se guarda en cada posición una dirección proporcionada por `cudaMalloc()`. Una vez poblados de contenido estos punteros, procedemos a copiar el contenido del puntero de primer nivel de la memoria local al puntero de primer nivel de la memoria del dispositivo.

Para hacer uso de objetos en el dispositivo, se deberán de reservar cada uno de los atributos de tipo puntero mediante el método anterior. Una vez obtenido el puntero a la dirección de la memoria del dispositivo y haberlo poblado de contenido, haremos un `cudaMemCpy()` desde dicho puntero hasta la dirección de memoria del atributo en cuestión del objeto residente en la memoria del dispositivo.

## 3.2. Estructura del proyecto

Aunque no era necesario crear ninguna estructura concreta en el proyecto, separar el código en múltiples archivos permite abstraer el código específico de CUDA de forma que programar diferentes topologías y/o entrenamientos de la red neuronal resulte sencillo. También nos permite compilar cada archivo por separado, de modo que si no se realizan cambios en un archivo, éste no es incluido en la siguiente compilación, lo que reduce las esperas entre ejecuciones con distintos parámetros.

El proyecto se divide en los siguientes archivos:

- *readMNIST.cpp/.h*: Contiene el código que lee los datos de los archivos de MNIST y devuelve un puntero al primer elemento de este conjunto de datos.
- *cudaNN.cuh*: Declara la clase NeuralNetwork.
- *cudaKernelCalls.h*: Declara las funciones que abstraen de la arquitectura CUDA (kernels).
- *cudaNN.cu*: Implementa las funciones de la clase NeuralNetwork declarados en *cudaNN.cuh*, así como los kernels que los llaman y las funciones de abstracción de *cudaKernelCalls.h*.
- *main.cpp*: Obtiene los datos de MNIST usando las funciones declaradas en *readMNIST.h* e inicializa una red neuronal y los distintos parámetros necesarios para efectuar su entrenamiento (tasa de aprendizaje, número de epochs, tamaño de batch...). Hace uso de las funciones declaradas en *cudaKernelCalls.h*.

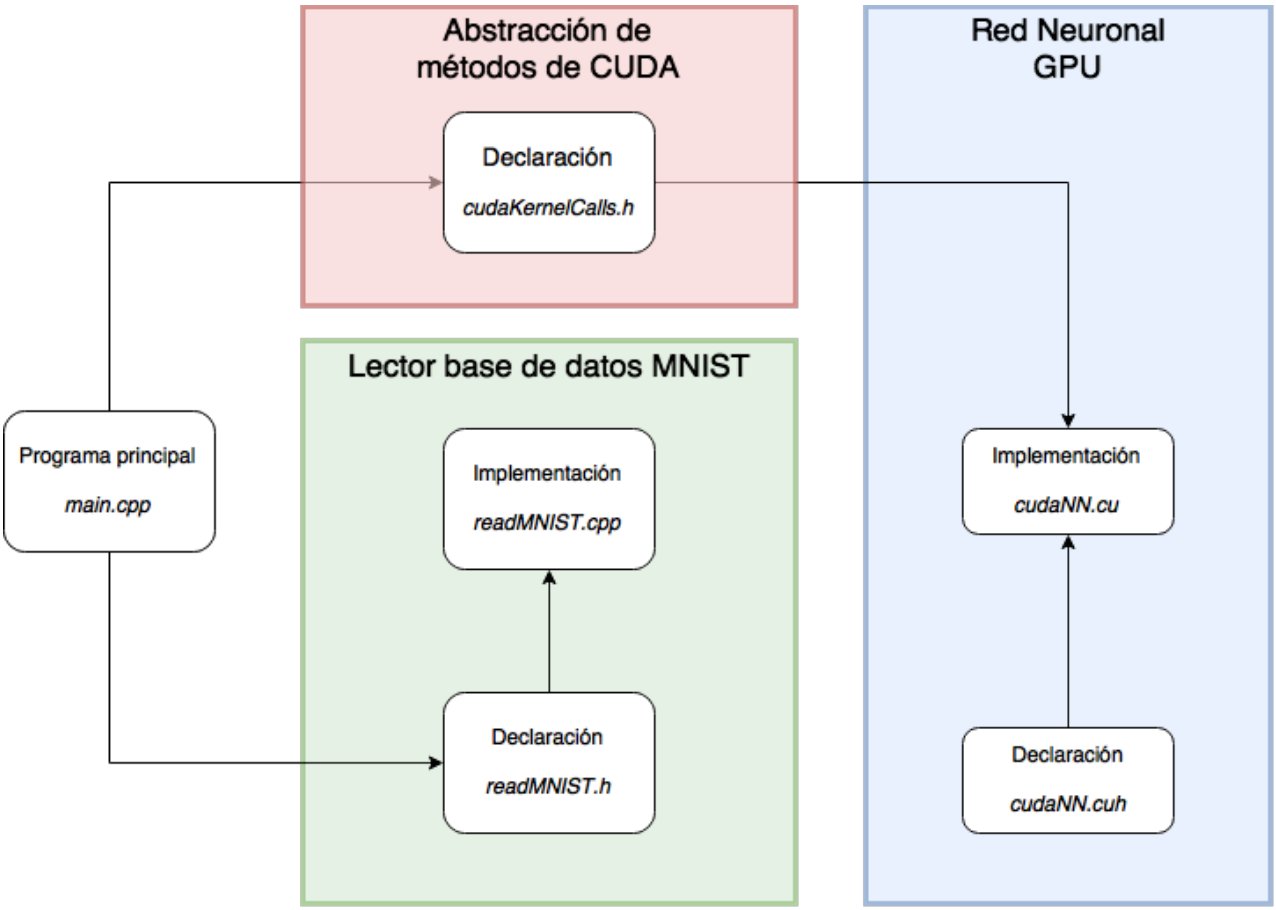


Figura 3.1: Diagrama de la estructura del proyecto

## 3.3. La clase NeuralNetwork

El primer paso en la realización de este proyecto fue la implementación de una clase `NeuralNetwork` que contuviese los atributos que definen a una red neuronal, así como las funciones que trabajaban sobre estos atributos.

Dichas funciones son ejecutadas por bloques de hilos en la GPU y son las siguientes:

- `computeLayer()`: Calcula el valor de una neurona de la capa oculta especificada sumando la multiplicación de los elementos de la capa visible por sus pesos correspondientes y aplicando la función de activación que corresponde con la capa oculta.
- `computeInverseLayer()`: Realiza el proceso inverso a `computeLayer()`, calculando el valor de una neurona de la capa visible especificada sumando la multiplicación de los elementos de la capa oculta por los pesos que corresponden (usando la matriz de pesos transpuesta). Los resultados se guardan en una capa de igual topología que la capa visible.
- `computeErrorOutput()`: Calcula el error de la capa de salida a partir del valor esperado dado. Esta función se usa tanto en la retropropagación del error del entrenamiento como en la del preentrenamiento (*Autoencoder*).
- `computeError()`: Calcula el error de la capa visible a partir de los errores de la capa oculta (o de salida) y de los pesos correspondientes.
- `computeFinetuningError()`: Realiza el proceso inverso a `computeError()` sobre las capas C' del proceso de *Autoencoder*.
- `backpropagation()`: Actualiza los pesos de una RBM mediante el algoritmo de retropropagación del error, usando los errores calculados con la función `computeError()`.
- `finetuningBackpropagation()`: Aplica el algoritmo de retropropagación del error en las capas C' del proceso de *Autoencoder*.
- `contrastiveDivergence()`: Actualiza los pesos de una RBM usando las capas visible y oculta y sus versiones prima que deben ser previamente calculadas usando las funciones `computeLayer()` y `computeInverseLayer()`.
- `activationFunction()`: Aplica la función de activación correspondiente dado el valor calculado de una neurona.

Hay aproximadamente una función kernel por cada función de la clase `NeuralNetwork`, y se limitan a controlar el rango de las llamadas a la función correspondiente y a pasar a ésta los parámetros que requiere.

```
__global__
void neural_network_computation(NeuralNetwork *nn, int level)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  int hidden_rows;
  if( level >= nn->num_layers ) hidden_rows = nn-
>networkTopology[level+1 - nn->num_layers];
  else hidden_rows = nn->networkTopology[level+1];

  if (i < hidden_rows && j < nn->batch){
    // Do not act on bias neuron
    if( i != 0 ){
      nn->computeLevel(i, j, level);
    }
  }
}
```

Figura 3.2: Kernel de `computeLevel()`

Como se puede observar en la figura 3.2, se usan las coordenadas de los hilos y bloques de hilos para obtener las coordenadas a usar en la función `computeLevel()`, que corresponden a una capa de la red neuronal.

Las llamadas a los kernels se hacen desde funciones `void` que definen el número de bloques de hilos a lanzar y ejecutan un procedimiento completo (por ejemplo, para realizar un *forwarding* completo, hay que hacer n-1 llamadas al kernel de `computeLevel()`). Esto abstrae de los parámetros de los kernels y permite trabajar con la red neuronal en el dispositivo usando solamente código C++ desde un archivo *main.cpp*, pudiendo crear múltiples archivos con diferentes configuraciones de redes neuronales fácilmente.

```
  void CKCall::forwarding(dim3 threadsPerBlock, NeuralNetwork*
d_nn, int* net_top, int num_layers, int batch, bool all){
    // Do not act on the real output layer unless specified
    int last_level = num_layers-2;
    if( all ) last_level++;

    for( int i = 0; i < last_level; i++ ){
      dim3 numBlocks( ( (net_top[i+1])+(threadsPerBlock.x-1)
)/threadsPerBlock.x, ( (batch)+(threadsPerBlock.y-1)
)/threadsPerBlock.y );

      neural_network_computation<<<numBlocks,
threadsPerBlock>>>(d_nn, i);
    }
  }
```

Figura 3.3: Función invocadora del kernel de `computeLevel()`

En la figura 3.3 vemos cómo una llamada a la función `forwarding()` ejecuta el kernel de `computeLevel()` para todas las capas, encargándose de lanzar el número de bloques de hilos correcto a cada llamada. Se hace uso de una variable boolean para omitir la capa de salida del proceso (en el pre-entrenamiento de *Autoencoder* no debe de usarse la capa de salida).

Muchos de los procesos del entrenamiento de las redes neuronales implican una multiplicación de un vector (las capas de neuronas se representan con vectores) por una matriz (los pesos entre dos capas se representan mediante una matriz). Con el objetivo de aprovechar aún más las capacidades de paralelización de CUDA, ejecutaremos las operaciones en bloque ("batch"), convirtiendo los vectores de entrada en matrices y consiguiendo el efecto de procesar más de una entrada en cada operación. Ésto, no obstante, impacta negativamente en algunas operaciones de entrenamiento, como en el algoritmo de retropropagación del error, que al calcular la desviación de muchas entradas respecto a un mismo estado de las matrices de pesos, la corrección de dichos pesos es menos precisa.

## 3.4. El archivo main.cpp

Como su propio nombre indica, este archivo contiene la función `main()` del proyecto, es decir, la función invocada por el ejecutable que se obtiene al compilar el proyecto. En él, inicializaremos la red neuronal en el dispositivo y ejecutaremos los preentrenamientos, el entrenamiento y la comprobación de resultados haciendo uso de las funciones de abstracción de tipo `void` anteriormente mencionados.

La ejecución se puede controlar usando las constantes que definirán la topología de la red neuronal y los entrenamientos. Estas constantes son:

- `INPUT_SIZE`: El número de muestras a usar en el entrenamiento. En el caso de MNIST, éste es de hasta 60000. Resulta útil para limitar el tamaño del problema y conseguir ejecuciones muy rápidas para, por ejemplo, encontrar errores en menos tiempo.
- `TEST_INPUT_SIZE`: El número de muestras a usar en la comprobación del resultado. En el caso de MNIST, éste es de hasta 10000.
- `TRAINING_EPOCHS`: Indica el número de epochs (repeticiones) de entrenamiento a ejecutar.
- `CONTRASTIVE_EPOCHS`: Número de epochs del preentrenamiento Contrastive Divergence.
- `FINETUNING_EPOCHS`: Número de epochs del preentrenamiento *Autoencoder*.
- `LEARNING_RATE`: Factor de aprendizaje del proceso de entrenamiento. Es un número entre 0 y 1 que a más alto más rápido aprende la red, pero menos probabilidad de converger tiene.
- `CONTRASTIVE_LEARNING_RATE`: Factor de aprendizaje del preentrenamiento Contrastive Divergence.
- `FINETUNING_LEARNING_RATE`: Factor de aprendizaje del preentrenamiento *Autoencoder*.
- `OS`: Tamaño de la capa de salida. Con MNIST, donde queremos clasificar números entre el 0 y el 9, éste es de 10.
- `IS`: Tamaño de la capa de entrada. MNIST usa imágenes de 28x28, por lo que usaremos 784.

- `B`: Tamaño de batch. A más grande, menos llamadas a la función que transfiere datos de entrada a la red, aunque el volumen de datos transferido siempre será el mismo por lo que sólo reduce el coste de sincronización de la llamada.

También deberemos inicializar 2 vectores que definirán la topología de la red y las funciones de activación a usar en cada capa.

```
const int INPUT_SIZE = 60000;
const int TEST_INPUT_SIZE = 10000;
const int TRAINING_EPOCHS = 50;
const int CONTRASTIVE_EPOCHS = 20;
const int FINETUNING_EPOCHS = 20;
const float LEARNING_RATE = 0.0001;
const float CONTRASTIVE_LEARNING_RATE = 0.0001;
const float FINETUNING_LEARNING_RATE = 0.00005;
const int OS = 10;
const int IS = 28*28;
const int B = 500;
int *net_top;
int *act_func_top;
unsigned char *input = MNIST::getTrainingSet();
unsigned char *expected_output = MNIST::getTrainingLabelSet();
unsigned char *test_input = MNIST::getTestSet();
unsigned char *test_expected_output = MNIST::getTestLabelSet();

net_top = (int*)malloc(4*sizeof(int));
act_func_top = (int*)malloc(4*sizeof(int));

net_top[0] = IS;
net_top[1] = 800;
net_top[2] = 800;
net_top[3] = OS;

act_func_top[0] = SIGMOID;
act_func_top[1] = TANH;
act_func_top[2] = TANH;
act_func_top[3] = TANH;
```

Figura 3.4: Ejemplo de parámetros iniciales

En la figura 3.4 podemos observar la configuración de parámetros para una red neuronal de 4 capas, topología 784-800-800-10 con un tamaño de batch de 500 y topología de funciones de activación SIGMOID-TANH-TANH-TANH. Ésta ejecución efectuará, sobre la totalidad de los datos de MNIST, 50 epochs de entrenamiento con factor de aprendizaje 0,0001, 20 epochs de Contrastive Divergence con factor de aprendizaje 0,0001 y 20 epochs de *Autoencoder* con factor de aprendizaje 0,00005.

# 4. Resultados

En este apartado se exponen los resultados obtenidos en la realización de diversas pruebas y se analiza el rendimiento y la viabilidad del uso de CUDA.

## 4.1. Descripción de resultados

Para evaluar el sistema, se han ejecutado múltiples entrenamientos con diferentes configuraciones de la red neuronal. Las siguientes pruebas han sido efectuadas sobre una nVidia GTX 770 de 2GB de memoria dedicada[1].

Para empezar, hemos elegido una topología simple (784-800-800-10) y hemos ajustado los ratios de aprendizaje del entrenamiento y de los pre-entrenamientos comenzando con un valor de 0,01 y dividiendo por 10 hasta obtener resultados satisfactorios. Dichos resultados se han obtenido con los siguientes valores:

- **Ratio de aprendizaje del entrenamiento:** 0,0001.
- **Ratio de aprendizaje de Contrastive Divergence:** 0,0001.
- **Ratio de aprendizaje de Autoencoder:** 0,00005.

A continuación se han efectuado ejecuciones con diferente número de epochs de cada pre-entrenamiento, para comprobar cómo influyen en el ratio de error resultante.

1: http://www.nvidia.es/object/geforce-gtx-770-es.html#pdpContent=2

| | | Autoencoder | |
|---|---|---|---|
| | | 0 epochs | 20 epochs |
| Contrastive | 0 epochs | 2,56% – 11,81% | 2,25% - 4,07% |
| | 20 epochs | 2,43% - 3,81% | 2,33% - 3,05% |

Tabla 4.1: Ratio de error con topología 784-800-800-10, Batch 500 y 50 epochs de entrenamiento

Como podemos observar en la tabla 4.1 los pre-entrenamientos ayudan a estabilizar los resultados obtenidos, es decir, reducen la variación en los ratios de error entre ejecuciones de una misma configuración de red neuronal, pero no reducen en gran medida los ratios de error obtenidos.

Con el objetivo de mejorar la topología de la red elegida, hemos probado 2 nuevas topologías diferentes de creciente complejidad.

| Topología | Ratio de error | Tiempo transcurrido |
|---|---|---|
| 784-800-800-10 | 2,70% | 1h 02m 45s |
| 784-800-800-800-10 | 2,70% | 1h 37m 31s |
| 784-900-1000-900-1000-10 | 2,89% | 2h 58m 01s |

Tabla 4.2: Ratio de error y tiempo transcurrido con Batch 500, 100 epochs de entrenamiento y 20 epochs de cada preentrenamiento

En la tabla 4.2 podemos ver cómo aumenta el tiempo a medida que aumenta la complejidad de la topología, y que el uso de capas ocultas de tamaño superior a 800 empeora los resultados. Observamos también cómo el ratio de error no mejora al aumentar el número de capas ocultas de tamaño 800, por lo que determinamos que 2 capas de tamaño 800 es la mejor opción dado que su tiempo de ejecución es menor.

Otro parámetro a evaluar ha sido el tamaño del batch. Dado que los datos de entrada deben ser enviados a la memoria del dispositivo en cada epoch de cada proceso mediante la función `cudaMemCpy()` podemos intuir que ésta función conllevará un tiempo de sincronización entre memorias y que por tanto a menor número de llamadas menor será el tiempo total transcurrido en el paso de datos.

| Batch | Ratio de error | Tiempo transcurrido |
|---|---|---|
| 1 | 2,47% | 1h 29m 25s |
| 10 | 2,49% | 0h 17m 30s |
| 100 | 2,45% | 0h 17m 17s |
| 500 | 2,56% | 0h 16m 57s |

Tabla 4.3: Ratio de error y tiempo transcurrido con topología 784-800-800-10, 50 epochs de entrenamiento

Como se observa en la tabla 4.3 el tamaño de batch no afecta de forma significativa al ratio de error, pero tal y como habíamos predicho sí afecta al tiempo transcurrido cuando elegimos un tamaño muy pequeño. Daremos pues preferencia a tamaños de batch de 100 o mayores.

Es importante comentar que al ejecutar el presente proyecto en un dispositivo de capacidad limitada (GPU nVidia GT 540M[2]) no se ha podido hacer uso del tamaño de batch de 500 viéndonos obligados a usar un tamaño de 100, por lo que no siempre podemos escoger el tamaño de batch más grande.

Una vez determinados todos los parámetros de la red (ratios de aprendizaje, número de epochs, topología y tamaño de batch) hemos comprobado hasta cuánto es posible reducir el ratio de error ejecutando muchas epochs de entrenamiento.

| Entrenamiento | Ratio de error | Tiempo transcurrido |
|---|---|---|
| 200 epochs | 2,15% | 1h36m39s |
| 500 epochs | 2,05% | 3h18m21s |

Tabla 4.4: Ratio de error y tiempo transcurrido con Batch 500, topología 784-800-800-10 y 20 epochs de cada pre-entrenamiento.

Tal y como ilustra la tabla 4.4, el mínimo ratio de error se ha obtenido con 500 epochs de entrenamiento y ha sido de 2,05%. El tiempo de ejecución de esta prueba ha sido de 3 horas, 18 minutos y 21 segundos.

2: http://www.geforce.com/hardware/notebook-gpus/geforce-gt-540m/specifications

## 4.2. Rendimiento del sistema

Para evaluar el rendimiento de nuestra implementación hemos calculado el número de sumas y multiplicaciones en coma flotante (FLOPS) de las funciones básicas.

Los datos de la siguiente tabla se han obtenido utilizando las fórmulas detalladas en el Anexo A.

| | GFLOPS/epoch | Segundos/epoch | GFLOPS/s |
|---|---|---|---|
| Contrastive | 1064,83 | 45,87 | 23,21 |
| Autoencoder | 990,73 | 41,70 | 23,76 |
| Entrenamiento | 461,05 | 20,3 | 22,71 |

Tabla 4.5: Rendimiento de los distintos algoritmos.

Observamos en la tabla 4.5 que el rendimiento obtenido es significativamente menor al máximo ofrecido por nuestro dispositivo.

Con el propósito de descubrir el motivo de los resultados, implementamos una versión de la red neuronal sin clases ni objetos. Aunque sí que percibimos cierta mejoría ésta era muy poco significativa, siendo así descartada esta posibilidad.

Durante el desarrollo del proyecto comprobamos que las operaciones de mayor coste en código del dispositivo son la lectura y escritura en memoria, habiendo logrado reducir el tiempo de ejecución de algunas funciones a la mitad al operar sobre una variable auxiliar y guardar en memoria solamente el resultado final.

Desarrollamos una implementación de la red neuronal para su ejecución totalmente en CPU basada en la estructura de la implementación de CUDA para medir la aceleración obtenida al hacer uso de una GPU. Las ejecuciones sobre un único núcleo de una CPU Intel i7-4770[3] resultaron ser más de 10 veces peor en cuanto a rendimiento, demostrando la viabilidad del uso de CUDA pese al problema de las operaciones de memoria.

3: http://ark.intel.com/es-es/products/75122/

# 5. Conclusiones

En este apartado se describen las conclusiones extraídas de la realización del presente proyecto y se detallan algunos problemas surgidos durante su desarrollo. También se proponen algunas mejoras de cara al futuro del proyecto.

## 5.1. Experiencia y conclusión

En el desarrollo del presente proyecto hemos obtenido experiencia en paralelización, programación general sobre GPU y clasificación automática, uno de los campos de la Inteligencia Artificial.

Hemos observado una serie de puntos a los que prestar especial atención:

- Las escrituras en memoria pueden ralentizar en gran medida los programas en GPU.
- El modelo de programación orientado a objetos, aunque factible, sí afecta negativamente al tiempo de ejecución de los programas en GPU.
- Con redes neuronales relativamente simples se pueden obtener muy buenos resultados en el reconocimiento de formas y la clasificación automática.

La conclusión obtenida es que aun teniendo que realizar muchas escrituras en memoria para el procesamiento de una red neuronal, la simplicidad de uso de plataformas como CUDA hace viable el uso de GPU para trabajar con estas estructuras.

## 5.2. Dificultades superadas

La principal complicación surgida en el desarrollo del proyecto fue la dificultad de la corrección de errores del programa. Por un lado, no es trivial obtener un registro del estado de la memoria en diferentes puntos del programa debido a la separación entre la memoria

local (CPU) y la del dispositivo (GPU). Por otro, el uso de 2 dispositivos diferentes en la ejecución del programa, la paralelización masiva del código en GPU y el hecho de que los resultados de las redes neuronales varían enormemente con pequeños cambios en sus parámetros hizo muy complicada la localización de pequeños errores en el código.

También resultó complicado averiguar la forma de construir un objeto en la memoria de la GPU. Esto se debe a que tan solo disponemos de una función para la reserva de espacio de memoria en el dispositivo en forma de puntero, y de otra para la copia de datos entre punteros a memoria local y punteros a memoria del dispositivo. Todo ello sumado a la carencia de ejemplos o explicaciones al respecto hizo que lograr el traspaso de datos completo de un objeto conllevara más tiempo del esperado.

Por último, debido a que la ejecución de código en GPU se realiza mediante llamadas desde el código local, para tener acceso a un dato calculado en un kernel desde una llamada a kernel posterior, dicho dato deberá ser guardado en la memoria del dispositivo via un puntero previamente asignado y proporcionado desde el código local, lo que obliga a recalcular ciertas variables cada vez que se efectúa una sincronización (el segundo kernel no se ejecuta hasta que todos los hilos del primer kernel finalizan la tarea).

## 5.3. Futuro

Como lo que limita la velocidad de ejecución del proyecto en mayor medida es la escritura en memoria, sería interesante introducir técnicas complejas en el entrenamiento de la red neuronal para aprovechar al máximo la capacidad de cómputo de la GPU. También se podría optimizar el código estudiando con detalle el manejo de la caché por parte de CUDA y asegurándose de que se produzcan el mínimo número de fallos de caché.

Dado que el código de CUDA se limita a definir la estructura y las operaciones realizadas sobre una red neuronal, se puede aprovechar el código que hemos desarrollado para muchas aplicaciones de Inteligencia Artificial más allá de la clasificación automática de dígitos manuscritos de MNIST.

Por último indicar que con sólo repartir los bloques de hilos en los que se dividen las tareas de CUDA del proyecto actual entre múltiples GPU podría obtenerse una mayor velocidad de procesamiento de las redes neuronales.

# Anexo. Análisis de rendimiento

En este anexo se describe el cálculo de FLOPS de nuestro proyecto.

## Cálculo de FLOPS

El cálculo de FLOPS de un código se efectúa contabilizando el número de operaciones en coma flotante ejecutadas por ese código. En el caso de las distintas funciones de nuestro proyecto los FLOPS son dependientes de los parámetros proporcionados en su llamada.

Las fórmulas de cálculo de FLOPS de las funciones son las siguientes:

- `computeLayer()`: (topologia[l]*2 + activacion[l+1]) * topologia[l+1] * Batch * numBatches.
- `computeInverseLayer()`: (topologia[l+1]*2 + activacion[l]) * topologia[l] * Batch * numBatches.
- `computeErrorOutput()`: (2 + dactivacion[l] + 1) * topologia[l] * Batch * numBatches.
- `computeError()`: (topologia[l+1]*2 + dactivacion + 1) * topologia[l] * Batch * numBatches.
- `computeFinetuningError()`: (topologia[l]*2 + dactivacion + 1) * topologia[l+1] * Batch * numBatches.
- `backpropagation()`: (3 * Batch + 1) * topologia[l] * topologia[l+1] * numBatches.
- `finetuningBackpropagation()`: (3 * Batch + 1) * topologia[l] * topologia[l+1] * numBatches.

- `contrastiveDivergence()`: (7 * Batch + 1) * topologia[l] * topologia[l+1] * numBatches.
- `activationFunction()`:
  - TANH: activacion = 3, dactivacion = 2.
  - SIGMOID: activacion = 3, dactivacion = 2.

Donde Batch es el tamaño de batch escogido, numBatches es el total de datos de entrada dividido por el tamaño de batch, topologia[x] se refiere al tamaño de la capa x, activacion[x] se refiere al número de FLOPS de la función de activación de la capa x y dactivación[x] hace referencia al número de FLOPS de la derivada de la función de activación de la capa x.

Una vez disponemos de las fórmulas de las funciones básicas, necesitamos saber qué funciones básicas se ejecutan con cada llamada a las funciones de abstracción:

- `forwarding()`:
  - Normal: Ejecuta `computeLayer()` n-2 veces empezando en la capa 0 y subiendo de 1 en 1 hasta la capa n-3.
  - Completo: Ejecuta `computeLayer()` n-1 veces empezando en la capa 0 y subiendo de 1 en 1 capa hasta la capa n-2.
  - Inverso: Ejecuta `computeInverseLayer()` n-2 veces empezando en la capa n-3 y bajando de 1 en 1 hasta la capa 0.
- `finetuningErrors()`: Ejecuta `computeErrorOutput()` una vez en la capa 0, `computeFinetuningError()` de la capa 0 a la capa n-3 y `computeError()` de la capa n-3 a la capa 1.
- `classificationErrors()`: Ejecuta `computeErrorOutput()` una vez en la capa n-1 y `computeError()` de la capa n-2 a la capa 1.
- `backpropagation()`:
  - Normal: Ejecuta `backpropagation()` n-2 veces empezando en la capa n-3 y bajando de 1 en 1 hasta la capa 0.
  - Completo: Ejecuta `backpropagation()` n-1 veces empezando en la capa n-2 y bajando de 1 en 1 hasta la capa 0.
  - Inverso: Ejecuta `finetuningBackpropagation()` n-2 veces empezando en la capa 0 y subiendo de 1 en 1 hasta la capa n-3 y `backpropagation()` n-2 veces empezando en la capa n-3 y bajando de 1 en 1 hasta la capa 0.

Por último, obtenemos las funciones de abstracción de las que se compone cada epoch de cada algoritmo de entrenamiento:

- Contrastive Divergence: Varía entre las capas 0 y n-3. Ejecuta `computeLayer()` hasta la capa de Contrastive en la que nos encontramos, seguido de un `computeInverseLayer()` y otro `computeLayer()`, ambos en la misma capa. A continuación ejecuta un `contrastiveDivergence()` también usando dicha capa.

- Autoencoder: Ejecuta, en orden, un `forwarding()` normal, un `forwarding()` inverso, un `finetuningErrors()` y un `backpropagation()` inverso.

- Entrenamiento Backpropagation: Ejecuta, en orden, un `forwarding()` completo, un classification`Errors()` y un `backpropagation()` completo.

# Glosario

**algoritmo** Es un conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad.

**aprendizaje** Es el proceso por el cual una red neuronal modifica los pesos de las conexiones entre neuronas en respuesta a una información de entrada.

**cache** Memoria de acceso rápido que almacena temporalmente los datos recientemente procesados.

**clase** Plantilla para la creación de objetos en lenguajes orientados a objetos. Se usan para representar entidades y conceptos tales como una casa o una red neuronal, y están compuestas de atributos (variables de estado) y métodos (funciones de comportamiento).

**clasificación automática** Área de la Inteligencia Artificial que busca crear programas que sean capaces de etiquetar de forma correcta unos elementos dados, como por ejemplo, clasificar emails en Spam y No spam.

**coma flotante** Forma de representar números racionales usada en los microprocesadores con la que se pueden realizar operaciones aritméticas.

**control de flujo** Es la manera en que un lenguaje de programación determina el orden en el que se ejecutarán las instrucciones en nuestros programas.

**epoch** Cada una de las ejecuciones de un algoritmo de entrenamiento (o pre-entrenamiento) sobre la totalidad del conjunto de datos de entrada.

**función** Módulo independiente de código que resuelve una tarea específica. Puede recibir valores de entrada en forma de parámetros.

**función de activación** Función matemática que se aplica sobre la entrada de una neurona, generando una salida.

**hilo** En sistemas operativos, un hilo de ejecución o subproceso es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. La creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez.

**Inteligencia Artificial** Campo de estudio sobre la creación de ordenadores y programas capaces de reproducir el comportamiento inteligente propio de los seres humanos.

**latencia** Retardo producido por una acción o secuencia de acciones.

**matriz de pesos** Matriz que representa el peso de las conexiones entre las neuronas de dos capas de una red neuronal.

**neurona** Unidad de las redes neuronales que recibe una serie de entradas y emite una salida.

**pixel** Es la menor unidad direccionable que forma parte de una imagen digital o un dispositivo de visualización.

**programación distribuida** Es un paradigma de programación en el que se divide un problema y se reparte entre múltiples ordenadores conectados a una red.

**programación paralela** Es un paradigma de programación en el que se divide un problema y se reparte entre múltiples unidades de proceso, normalmente núcleos de un mismo procesador.

**puntero** Objeto de un lenguaje de programación cuyo valor hace referencia a la posición en memoria de otro valor.

**puntero multinivel** Puntero en el que el valor de la dirección de memoria a la que apunta es otro puntero.

# Bibliografía

1: Hamidreza Rashidy Kanan, Mahdi Yousefi Azar Khanian, Reduction of Neural Network Training Time Using anAdaptive Fuzzy Approach in Real Time Applications, 2012

2: Intel España, Intel Desvela la Próxima Generación de Plataformas de Sobremesa , 2015

3: Insup Lee, Introduction to Distributed Systems, 2007

4: Anand Ranganathan, Roy H. Campbell, What is the Complexity of a Distributed System?, 2005

5: Jean-Pierre Dijcks-Oracle, Price Comparison for Big Data Appliance and Hadoop, 2014

6: NVIDIA, NVIDIA Leads Performance Per Watt Revolution With "Maxwell" Graphics Architecture, 2014

7: NVIDIA, About CUDA, 2015, https://developer.nvidia.com/about-cuda

8: NVIDIA Corporation, CUDA C PROGRAMMING GUIDE, 2015

9: Monica S. Lam, Edward E. Rothberg and Michael E. Wolf, The Cache Performance and Optimizations of Blocked Algorithms,

10: Geoffrey Hinton, A Practical Guide to Training Restricted Boltzmann Machines, 2010

11: Geoffrey Hinton, A Practical Guide to TrainingRestricted Boltzmann Machines, 2010

12: Yann LeCun, Corinna Cortes, Christopher J.C. Burges, THE MNIST DATABASE of handwritten digits, 2015, http://yann.lecun.com/exdb/mnist/

13: Theano Development Team, Restricted Boltzmann Machines (RBM), 2008, http://deeplearning.net/tutorial/rbm.html

# Figuras