Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Iniciación al Entorno de Deep Learning Torch

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Javier Martínez Marhuenda

Tutor: Federico Silla Jiménez

Director Experimental: Javier Prades Gasulla

Curso 2017-2018

Agradecimientos

A mi familia, por no dejar nunca de ser mi principal apoyo y mi motivación para tratar de superarme día a día.

A todos aquellos que me han acompañado a lo largo mi vida como estudiante y han hecho de ella una melancólica experiencia.

A todas esas personas que se esfuerzan diariamente en crear una sociedad mejor, tanto personal como culturalmente.

A mis tutores Fede y Javi, que me han librado de muchos dolores de cabeza sin queja alguna.

Resumen

El Deep Learning es una tecnología puntera actual que va a tener gran presencia en el futuro. Sin embargo, dominarlo es una tarea compleja y laboriosa que no todo el mundo está dispuesto a emprender. En este trabajo se ofrece una guía de iniciación al Deep Learning para poder hacer uso de él mediante distintas aplicaciones con la ayuda de Torch. Conoceremos los conocimientos base del Deep Learning y la Inteligencia Artificial, aprenderemos a entrenar redes neuronales obteniendo los mejores resultados de la forma más eficiente y finalmente, realizaremos un ejemplo práctico sobre una interesante aplicación que hace uso del Deep Learning: Neural Style.

Así pues, este estudio va dirigido para aquellos y aquellas sin conocimiento avanzado sobre Deep Learning y desean iniciarse y experimentar utilidades de esta tecnología mediante Torch. Además, también sirve para quienes desean aprender sobre el redes neuronales de reconocimiento de imagen con Torch y el uso de múltiples GPUs.

Palabras clave: Deep Learning, Torch, Inteligencia Artificial, Redes Neuronales, Neural Style, Iniciación, GPU, Reconocimiento de imagen, Entrenamiento

Resum

El Deep Learning és una tecnologia puntera actual que tindrà gran presència en el futur. Tanmateix, si el vols dominar és una tasca complexa i laboriosa que no tothom està disposat a emprendre. En aquest treball, s'ofereix una guia d'iniciació al Deep Learning per poder fer ús d'ell mitjançant diferents apliacions amb l'ajuda de Torch. Coneixerem els coneixements base del Deep Learning i la Intel·ligència Artificial, aprendrem a entrenar xarxes neuronals obtenint els millors resultats de la manera més eficient i finalment, farem un exemple pràctic sobre una interessant apliació que fa ús del Deep Learning: Neural Style.

Així doncs, aquest estudi va dirigit per a tots aquells i aquelles sense un avançat coneximent sobre Deep Learning i volen iniciar-se i experimentar utilitats d'aquesta tecnologia amb Torch. A més, serveix per a qui vol aprendre sobre les xarxes neuronals de reconeixement dimatge amb Torch i l'ús de múltiples GPUs.

Paraules clau: Deep Learning, Torch, Inteligència Artificial, Redes Neuronals, Neural Style, Iniciació, GPU, Reconoxeiment de images, Entrenament

Abstract

Deep Learning is a modern technology that will have a great presence in the future. However, mastering it is a complex and laborious task that not everyone is willing to undertake. In this paper, we will offer an introductory guide to Deep Learning for using it through different applications with the help of Torch. We will know basic knowledge of Deep Learning and Artificial Intelligence, we will learn how to train neural networks obtaining the best results in the most efficient way and finally, we will carry out a practical example about an interesting application that uses Deep Learning: Neural Style.

Therefore, this study is aimed at those without advanced knowledge about Deep Learning who wants to start and experience utilities of this technology through Torch. In addition, it is also helpful for those who wish to learn about neural networks for image recognition with Torch and the use of multiple GPUs.

Key words: Deep Learning, Torch, Artificial Intelligence, Neural Networks, Neural Style, Initiation, GPU, Image recognition, Training

Índice general

Índice general VII
Índice de figuras IX

1 Introducción 1
1.1 Contexto . . . 1
1.2 Objetivos . . . 1
1.3 Motivación . . . 2
1.4 Equipo y tecnologías usados . . . 2
1.5 Estructura . . . 3
2 Estado del arte 5
2.1 Inteligencia Artificial . . . 5
2.2 Deep Learning . . . 6
2.3 Torch . . . 7
3 Torch 11
3.1 Instalando Torch . . . 11
3.2 Primera prueba con Torch . . . 12
3.3 Clasificador de Imágenes . . . 14
3.3.1 Introducción al reconocimiento de imágenes . . . 14
3.3.2 Primer entrenamiento con el Clasificador . . . 15
3.3.3 Reduciendo el tiempo de entrenamiento . . . 20
3.3.4 Comparando tipos de redes . . . 22
3.3.5 Obteniendo la máxima precisión posible . . . 24
3.4 Entrenando con dos GPUs . . . 25
4 Ejemplo práctico: Neural Style 29
4.1 Explicación del algoritmo . . . 29
4.2 Primeras pruebas . . . 33
4.3 Variando el estilo para una misma imagen . . . 36
4.4 Variando el contenido para un mismo estilo . . . 37
4.5 Usando múltiples GPUs . . . 39
4.5.1 Creando una imagen de gran tamaño . . . 42
5 Conclusiones 45
5.1 Visión General . . . 45
5.2 Relación del trabajo desarrollado con los estudios cursados . . . 46
6 Futuros trabajos 47
Bibliografía 49

Índice de figuras

2.1 Ramas de la inteligencia artificial 6
2.2 Función hipótesis de distintas redes neuronales en diferentes etapas de entrenamiento 8

3.1 Script *exports.sh* 11
3.2 Primera conversación con el modelo obtenido 13
3.3 Reconocimiento de objetos avanzado de una red neuronal 15
3.4 Parámetros para main.lua 17
3.5 Resultados del primer entrenamiento 19
3.6 Resultados del test del primer entrenamiento 19
3.7 Archivo main.lua original 20
3.8 Modificación de MainTrain.lua 21
3.9 Gráfica comparación de los tres tipos de redes 23
3.10 Entrenamiento Alexnetowtbn 10.000 i.p.e 25
3.11 Entrenamiento Alexnetowtbn 10.000 i.p.e 2 GPUs 26
3.12 Comparación entrenamiento 1 GPU vs 2 GPUs 27

4.1 Clasificación de estilo y contenido de la imagen por capas. 30
4.2 Distintas imágenes creadas mediante el algoritmo. 31
4.3 Foto de la Ciutat de les Arts i les Ciències, Valencia. 33
4.4 *Starry Night*. Famoso cuadro pintado por Vincent Van Gogh en el 1889. 34
4.5 Resultados primera prueba con las capas seleccionadas por defecto. 35
4.6 Resultado obtenido con modelo Rough Faces 36
4.7 Distintos ejemplos variando el estilo de la reconstrucción 37
4.8 Cuadro del artista contemporáneo Leonid Afremov. 38
4.9 Distintos ejemplos obtenidos variando el contenido. 38
4.10 Fotografía de la ciudad de Valencia 39
4.11 *Guernica*, cuadro pintado por Pablo Picasso en 1937. 40
4.12 Imagen resultado con 2 GPUs. 41
4.13 Imagen reestructurada con colores originales del contenido. 41
4.14 Fotografía de Dublín, Irlanda. 42
4.15 *The Papal Palace*, cuadro pintado por Paul Signac en 1900. 42
4.16 Imagen reestructurada a gran tamaño. 43

CAPÍTULO 1
Introducción

1.1 Contexto

Llegas a tu casa, y ves en las noticias que hay una computadora insuperable al ajedrez, coches conduciéndose solos y ordenadores que pueden “ver”, los cuales en un futuro serán capaces de reconocer enfermedades de una manera inalcanzable para el ser humano. ¿Cómo harán todo eso? Te preguntas si no estás familiarizado con el tema. “Quiero ser capaz de hacer eso”, dirás si te ha interesado lo suficiente. Todo esto es posible gracias a una tecnología puntera en auge: el Deep learning. Luego explicaremos más detenidamente qué es el Deep Learning, pero en resumen: es dar capacidad a las máquinas de aprender como si de un cerebro humano se tratase.

Programar y desarrollar todo eso es un proceso complicado, pero nosotros vamos a tratarlo desde un punto de vista más sencillo, así que no vamos a necesitar avanzados conocimientos de programación ni técnicos para usar esta maravilla tecnológica. Hoy en día hay muchas herramientas para trabajar con todo el ámbito del Machine Learning (campo general del Deep Learning), pero nosotros nos vamos a centrar en Torch y le daremos uso especialmente para reconocimiento de imágenes.

Este libro es un amplio resumen sobre los conocimientos esenciales del Deep Learning [1]

1.2 Objetivos

A lo largo de este proyecto, deseamos cumplir varios objetivos, los cuales trataremos como premisas para finalizar el trabajo habiendo cumplido todos ellos:

- Realizar una introducción al Deep learning sin tratar temas técnicos sobre programación.
- Que una persona con nociones básicas sobre Deep learning sea capaz de ejecutar diferentes aplicaciones complejas con Torch.

• Crear una guía sobre cómo instalar Torch, las diferentes librerías necesarias para trabajar y cómo hacer uso de él.

• Investigar cómo entrenar una red neuronal de reconocimiento de imagen de la forma más óptima obteniendo los mejores resultados posibles.

• Comprobar las mejoras que supone entrenar redes neuronales con múltiples GPUs.

• Llevar a cabo el uso de un software Torch que hace uso de la tecnología de Deep Learning.

Como vemos, el objetivo general es hacer uso de Torch para conocer y hacer uso del Deep Learning.

1.3 Motivación

La principal motivación para realizar este proyecto ha sido mi propia situación, la cual pienso que se da en muchas otras personas: adentrarse en el mundo del Deep learning sin tener una previa formación sólida. Es un campo que requiere complejos conocimientos informáticos para poder desarrollar aplicaciones que hagan uso de él, además de la necesidad de conseguir las grandes cantidades de datos con las que se desee trabajar. Por ello, creo que es conveniente mostrar un ejemplo de cómo hacer uso de aplicaciones relacionadas con el Deep learning sin estar muy familiarizado con él. Desde mi punto de vista, el Deep Learning es un arte que está revolucionando el mundo de la informática y pienso que va a estar muy presente en el futuro, ya que es una tecnología con un potencial increíble aún por descubrir.

Además, me gustó la idea de investigar sobre una herramienta interesante y útil como es Torch; tanto para hacer uso de aplicaciones ya programadas como para programarlas. A esto hay que añadirle el disponer de un nodo de la UPV con potentes características y 2 GPUs, las cuales no puedo permitirme económicamente.

1.4 Equipo y tecnologías usados

Para realizar las pruebas de la forma más rápida posible, la UPV nos ha permitido el acceso a un cluster privado, el cual dispone de 35 nodos y nosotros hemos dispuesto de permiso para usar dos de ellos: nodo1 y nodo2. Ambos nodos tienen las mismas características, la diferencia es que el nodo2 dispone de dos GPUs, mientras que el nodo1 tiene solo una. Las especificaciones generales son las siguientes:

- CPU: 23 procesadores Intel Xeon e5-2620 (2.1 Ghz).
- GPU: Gpu: Tesla K20m de 4GB (dos modelos iguales en el caso del nodo 2).

- Sistema Operativo Linux
- RAM: 32 GB

Para conectarnos a los nodos, es necesario estar conectado a la red de la UPV. Esto puede realizarse estando conectado directamente a ella o a través de una VPN. Una VPN (Virtual Private Network), es una tecnología de red que permite la conexión a una red privada a mediante una red pública (Internet). De esta manera, podemos conectarnos a la red de la UPV (red privada) siempre que tengamos acceso a internet. Esta conexión la realizaremos mediante el comando ssh en una terminal que disponga de éste y trabajaremos en todo momento dentro de ésta terminal mediante comandos.

Para conectarnos al cluster, utilizamos un comando ssh (Secure Shell). El comando que usamos para acceder al cluster es el siguiente:

ssh -p 3322 jamarmar@cluster.upv.es.

Ya en el cluster, vemos la lista de todos los nodos y podemos conectarnos al que deseemos mediante *ssh <nombre del nodo>.* Para no tener que escribir toda la dirección del cluster, hemos hecho uso de un script básico.

Todos los modulos que usamos pueden ser encontrados en GitHub, que es una blablabla donde las personas o equipos comparten el código de su aplicación y ésta puede ser descargsds. En el nodo no disponemos de permisos de administrador y no podemos instalar ningún programa nuevo. Así que para poder hacer uso de distintos programas o comandos que necesitan de instalación previa (Git, Wget, etc.) tenemos que pedírselo expresamente a los administradores del cluster (en nuestro caso a nuestros tutores).

1.5 Estructura

En este primer capítulo nos hemos puesto en contexto con el ámbito del trabajo, los objetivos que vamos a tratar de cumplir, la motivación que nos ha llevado a hacer este trabajo y las herramientas que vamos a usar.

En el segundo capítulo hablaremos sobre el Deep Learning y todo el campo de la Inteligencia Artificial en general. Además, explicaremos qué es Torch y diferentes herramientas simirales.

El tercer capítulo servirá para familiarizarnos a trabajar con Torch: desde su instalación hasta el entrenamiento de una red neuronal con múltiple GPUs. Primero ejecutaremos una apliación básica y luego nos centraremos en el entrenamiento de redes para reconocimiento de imagen.

El cuarto capítulo consistirá plenamente en hacer uso de una aplicación real de Deep Learning,*Neural Art*. Con ella, podremos hacer uso de un software en Torch para obtener resultados visuales de aplicar este tipo de redes neuronales entrenadas.

El capítulo 5 serán las conclusiones obtenidas con la finalización del proyecto, así como mi valoración personal.

El sexto capítulo tratará sobre las herramientas de la carrera de las que hemos necesitado hacer uso y cómo he aumentado mi aprendizaje en ellas.

El capítulo 7 será una reflexión sobre las tareas que pueden llevarse a cabo a partir de este proyecto.

Por último, dispondremos de la bibliografía con las fuentes a la que hemos accedido.

CAPÍTULO 2
Estado del arte

2.1 Inteligencia Artificial

La inteligencia artificial es la rama de la informática sobre la que tratará este proyecto y consiste, principalmente, en tratar de dotar a las computadoras de cierta forma de inteligencia.

Hace millones de años, los Australopithecus golpeaban los huesos animales con piedras para obtener el tuétano, los nutrientes más energéticos. La constante evolución del homínido y su desarrollo del sistema cognitivo, dió lugar a seres capaces de usar el entorno a su favor y crear “herramientas”. Había comenzado nuestra desemejanza del resto de animales: el uso de la inteligencia. En este artículo de Carlos A. Marmelada podemos leer sobre las diferentes teorías de cómo comenzamos a ser seres inteligentes [2]

Pero ¿qué es la inteligencia?. Existen diversos tipos de inteligencia, pero nosotros vamos a tratar el término como algo general. Su origen etimológico es intus (“entre”) y legere (“escoger”). Así que podríamos decir que la inteligencia es la capacidad de saber escoger correctamente; la capacidad de resolver problemas. Cuando te enfrentas a un problema y llevas a cabo una solución para el mismo, obtienes un resultado. Si el resultado ha sido bueno, la próxima vez que tengas el problema, realizarás la misma solución. Si no, intentarás otra cosa. Lógico.
Este es el procedimiento básico para obtener conocimientos y gracias la transmisión de estos con los pasos de los años, hemos aumentado nuestra calidad de vida notablemente.

Por ejemplo, en 1847, cerca de un tercio de las mujeres que daban a luz en el hospital, morían. Cuando el médico húngaro Ignác Semmelweis comenzó a obligar al personal de su hospital a lavarse las manos con jabón, se redujo la mortalidad de éste a menos del 10 %. Cuando se demostró su eficacia, fue llevado a cabo en todo el mundo, salvando millones de vidas. Podemos decir que fue un conocimiento adquirido.

Desde siempre, hemos hecho uso de esta inteligencia y el constante aprendizaje para adaptarnos mejor a nuestro entorno. Esto también lo hemos aplicado a nuestras herramientas. En el último siglo, los nuevos avances tecnológicos

podían incluso cambiar el curso de una guerra, como lo hizo Alan Turing y su Máquina de Turing en la Segunda Guerra Mundial descifrando los mensajes nazis, como explica Carmen Torran en este artículo [3]. En 1950, Turing, escribió el artículo “Máquinas de Computación e Inteligencia” [4] donde trata la posibilidad de que una máquina sea capaz de imitar la conducta de la mente humana. Una máquina inteligente.

Esa fue la primera vez que surgió el término “Inteligencia artificial” y a partir de entonces, se ha ido avanzando en el campo hasta el punto de crear máquinas con una capacidad de “aprendizaje” y procesamiento que ningún ser humano puede llevar a cabo. En este artículo [5] disponemos de un resumen de la historia contemporánea de la inteligencia artifical Dentro del campo de la inteligencia artificial, está el Machine Learning, dentro de éste, se encuentra el Deep learning, como vemos en la siguiente imagen 2.1:

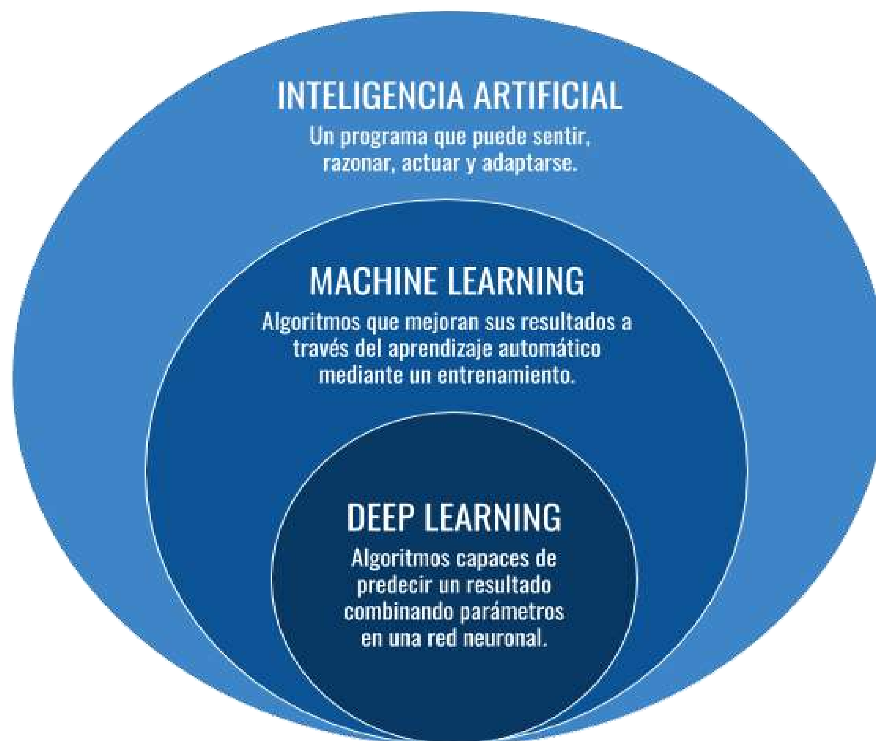


Figura 2.1: Ramas de la inteligencia artificial

2.2 Deep Learning

El deep learning es un aspecto de la inteligencia artificial que se centra en emular la manera de aprendizaje de la mente humana. Para tomar una idea básica, vamos a imaginar un niño pequeño, al cual le enseñamos la palabra coche y le mostramos fotos de coches y otras fotos cualquiera. En cada foto, el niño intenta adivinar si es un coche o no y le decimos si ha acertado. De esta manera, el niño va obteniendo información sobre distintos coches, “aprende” qué es un

coche y es capaz de reconocerlo en un futuro.
Ahora imaginemos el mismo procedimiento, pero con un ordenador y un millón de imágenes etiquetadas como "coche" o "no coche". El computador obtiene información a través de los píxeles de la imagen, construye un modelo con patrones de estos pixeles para determinar si es un coche o no, y obtiene un resultado, que puede ser un número entre 0 y 1, siendo 0 "no es coche" y 1 "sí es coche". Cuando termina un grupo de imágenes, comprueba qué porcentaje ha acertado y reconfigura su modelo basándose en los patrones que mejor han funcionado. Así pues, como norma general, cuantos más datos procesa, el porcentaje de acierto es mayor. Esta arquitectura de aprendizaje se llama Red neuronal, y es la más frecuente en Deep learning.

Una red neuronal es un modelo con distintos nodos (o neuronas) divididos en capas. Cada nodo está conectado con otros nodos mediante uniones, las cuales tienen asociado un peso. La combinación de los valores de las neuronas de la capa con el peso de cada conexión, determinará el valor de las neuronas en la siguiente capa. En las redes neuronales, siempre hay una capa de entrada y una de salida y entre ambas pueden existir capas ocultas, que son las "entrenadas" para obtener la salida buscada combinando correctamente los parámetros de entrada.

Como hemos ido comentando, por norma general, una mayor cantidad de datos significa mayor aprendizaje y mejores resultados, ya que si no le proporcionamos la información suficiente, el modelo no es capaz de generar una regla general precisa. Sin embargo, si le insertamos demasiados datos de entrenamiento, nuestra red se ajustará demasiado a éstos y no será capaz de clasificar nuevas muestras en el futuro. Este caso se llama sobreentrenamiento. Un ejemplo sencillo de sobreentrenamiento sería una red que ha obtenido 1.000.000 de imágenes de coches, pero todos amarillos. Nuestro modelo puede haber "aprendido" erróneamente que los coches han de ser amarillos y luego no sería capaz de clasificar un coche azul como tal. Por eso es tan importante una gran variedad de datos de entrenamiento y redes capaces de obtener la información más importante de ellos. Si deseamos conocer información más técnica sobre estos entrenamiento, se puede visitar este amplio libro digial [6].

En la figura 2.2, podemos observar la función hipótesis de varias redes neuronales y su correspondiente clasificación de dos distintas clases. Esta función es, por así decirlo, el nivel de aprendizaje de la red y la que determinará la predicción de distintos valores según su valor de entrada. Así que nuestro objetivo será entrenar esta función.

2.3 Torch

Torch es un entorno de trabajo (*framework*) de computación científica basado en el lenguaje de programación LUA, con gran soporte para algoritmos de machine learning usando principalmente GPUs. Torch tiene una de las comunidades de código abierto más grande (está apoyado por Facebook), lo que hace más fácil encontrar implementaciones útiles y gratuitas que lo soportan. Actualmente, solo puede ser instalado en Ubuntu 12+ y macOS, nosotros vamos a utilizar Ubuntu.

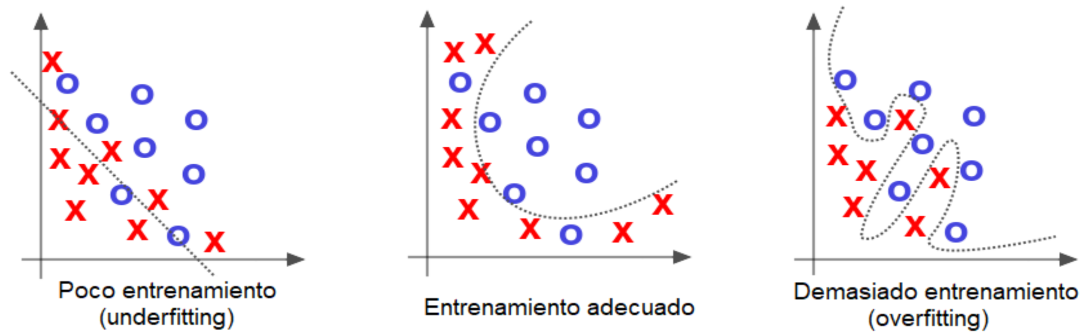


Figura 2.2: Función hipótesis de distintas redes neuronales en diferentes etapas de entrenamiento

En todo momento trabajaremos con la terminal (tanto Linux como Windows), con la cual nos conectaremos al nodo de la UPV.

Como hemos comentado, Torch es uno de los entornos relacionados con Machine Learning más usados, esto se debe a la flexibilidad y velocidad de creación de forma sencilla de las redes neuronales. Además de un gran soporte para paralelizar aplicaciones con múltiples GPUs, algo con lo que nosotros vamos a tener la posibilidad de experimentar.

A parte de Torch, existen otras alternativas como pueden ser:

Tensorflow:

Es una librería de código abierto para computación numérica. Es el más usado actualmente, ya que fue de los primeros softwares libre de computación numérica y además, desarrollado por Google.

Ventajas:

- Gran cantidad de softwares desarrollados con Tensorflow.
- Soporta Python y C++.
- Buena visualización de grafos computacionales.
- Grscalabilidad en distintos harwares.

Desventajas

- Librería de bajo nivel, lo que hace complicado desarrollar nuevos softwares.
- Sólo soporta GPUs de Nvidia.
- Es más lento que sus competidores.
- Las aplicaciones recientes

Caffe:

Es un framework desarrollado por la Universidad de Berkeley, dedicado especialmente a la visión artificial. Está desarrollado con C++ y arquitectura CUDA y tiene una colección abierta sobre modelos de deep learning.

Ventajas:

· Capacidad para aplicar redes neuronales profundas sin necesidad de escribir código.

· Rápido y gran robustez.

· Tiene interfaz tanto en Python como en Matlab.

Desventajas

· Dispone de pocos formatos de entrada y sólo uno de salida (HDF5).

· No tiene total soporte para entrenamientos con múltiples GPUs.

· La interfaz de Python es complicada de usar.

· No está propuesta para aplicaciones sobre textos o sonidos.

Theano

Ventajas:

· Gran facilidad para anyadir nuevas características en modelos preentrenados.

· Muy popular en la comunidad de investigación.

· Gran flexibilidad de uso gracias a su implementación en Python.

Desventajas:

· Gran dificultad de aprendizaje inicial.

· No soporta multi-GPU ni escalados horizontales.

· Usado principalmente en investigación, pero no en los demás campos y se está quedando atrás.

Hay otros frameworks a tener en cuenta que son: Keras (librería de python fácil de usar que corre sobre Theano o Tensorflow), Pytorch (paquete de Python que hace uso de Torch) o Lasagne (librería que funciona encima de Theano).

Como vemos, los entornos más usados actualmente tienen distintas características y no se puede elegir uno mejor o peor, todo dependerá de la utilidad que les quieras dar. Nosotros hemos elegido Torch, ya que es uno de los más

potentes en cuanto a velocidad de entrenamiento y dispone de una gran cantidad de repositorios interesantes que podemos usar. A pesar del inconveniente de que los demás entornos se centran en Python y Torch lo hace en LUA (algo que se “soluciona” con Pytorch), nosotros no vamos a desarrollar ningún código y no necesitaremos aprender LUA. Además, es uno de los frameworks que mejor soporte para multi-GPUs tiene.

CAPÍTULO 3
Torch

Todos los softwares que usaremos pueden ser accedidos a través de su repositorio en Git.

3.1 Instalando Torch

Comenzamos a instalar torch siguiendo la guía de la página oficial [7]:

1.- Clonamos el repositorio Git donde se encuentra el instalador de Torch:

```
clone https://github.com/torch/distro.git  /torch -recursive
```

Con este comando, vamos a crear un nuevo directorio llamado “torch” en nuestra carpeta raíz (o home), el cual contendrá los archivos del repositorio clonado.

2.- Indicamos la ubicación de las distintas librerías que necesitamos. En nuestro caso, será exportando las rutas a las distinas librerías (hemos omitido el nombre real por cuestión de privacidad).
Como este paso tenemos que hacerlo cada vez que iniciamos sesión en el nodo, vamos a usar un script “exports.sh” (ver figura 3.8 con los comandos export a realizar, el cual ejecutaremos cada vez que vayamos a trabajar:

```
1
2 export CMAKE_LIBRARY_PATH=:LIBS/PROTOBUF/3.1.0/lib
3 export CMAKE_INCLUDE_PATH=:LIBS/PROTOBUF/3.1.0/include
4 export PATH=$PATH:LIBS/PROTOBUF/3.1.0/bin:LIBS/CUDA/8.0/bin
5 export LD_LIBRARY_PATH=LIBS/OPENBLAS/0.2.19/lib:LIBS/CUDA/8.0/lib64:
    LIBS/CUDNN/8.0-v5.1/lib64:LIBS/PROTOBUF/3.1.0/lib
6 export CUDA_TOOLKIT_ROOT_DIR=:LIBS/CUDA/8.0/lib64
```

Figura 3.1: Script *exports.sh*

Para ejecutar este script, sólo tenemos que realizar el siguiente comando:

```
source exports.sh
```

3.- Ejecutamos el script de instalación: Dentro del repositorio creado, encontramos el script “install.sh”, el cual se encargará de instalar el compilador de Lua, LuaJIT y LuaRocks, que nos permite crear e instalar módulos de Lua en nuestro equipo. Además, se encarga de instalar paquetes secundarios y añadir torch a nuestra variable PATH. Primero, navegamos dentro del repositorio clonado, donde está el script.

```
cd  /torch
```

Ejecutamos el script de instalación.

```
./install.sh
```

4.- Comprobamos la instalación: Para verificar que Torch se ha instalado en nuestro equipo, probamos a ejecutarlo mediante el comando *th*.
Entonces se mostrará en la Terminal la cabecera de Torch y podremos ejecutar comandos de Lua para inicializarnos con ellos si así lo deseamos. Para terminar la sesión de Torch, sólo hay que presionar las teclas Ctrl+C dos veces.

3.2 Primera prueba con Torch

Para familiarizarnos con las pruebas que vamos a llevar a cabo, primero probaremos un módulo sencillo de Torch. El módulo en cuestión es un Chatbot, que consiste en una Inteligencia Artificial capaz de “mantener una conversación”. Para lograr esto, obtendremos la red neuronal del aprendizaje y los datos de entrenamiento. Estos datos consisten en un corpus que contiene una gran cantidad de diálogos extraídos de películas.
Como norma general, en cada módulo que usemos podremos encontrar los pasos a seguir para usarlo en su página de Git.

Primero, clonamos el repositorio e instalamos las dependencias que necesitamos siguiendo como se indica en el propio repositorio:

```
git clone https://github.com/macournoyer/neuralconvo
```

Además, tendremos que descargar los datos de entrenamiento, que consisten en un corpus que contiene 220.579 frases de 10.292 conversaciones de 617 películas. Este corpus lo podemos descargar de:
http://www.cs.cornell.edu/ cristian//Cornell_Movie-Dialogs_Corpus.html

Una vez descargado, navegamos dentro del repositorio clonado y procederemos a realizar un entrenamiento básico de nuestro primer modelo:

```
th train.lua -cuda -dataset 50000 -hiddenSize 1000
```

Los parámetros de este comando son los siguientes:

th: comando para ejecutar Torch. train.lua: fichero que contiene el algoritmo que utilizará nuestra red para entrenar el modelo. No entraremos en detalles técnicos, como hemos comentado anteriormente. En caso de tener interés, se puede acceder a él a través del repositorio de GitHub o abriéndolo desde nuestro repositorio con un editor de texto.

–*cuda*: Este parámetro establecerá el método con el que se entrenará nuestra red. En nuestro caso hemos elegido CUDA, que es una arquitectura de computación en paralelo de Nvidia que reparte los procesos entre la CPU y la GPU, logrando una mayor velocidad de cómputo. En caso de no querer usar la GPU, debemos omitir este parámetro.

–*dataset*: con esta opción podemos controlar el tamaño del conjunto de datos que usaremos.

–*hiddenSize*: es el número de neuronas que usaremos en la capa oculta. Así pues, dejamos la red entrenando hasta que finalice. Dependiendo de la potencia de procesamiento de nuestra máquina tardará más o menos. En nuestro caso ha tardado 35 horas. Al finalizar, vemos que ha generado un archivo llamado “model.t7” en la carpeta “data”. Este es el modelo que hemos obtenido después del entrenamiento. Para comenzar a chatear, ejecutamos el siguiente comando en la carpeta raíz del repositorio (neuralconvo): th eval.lua
La aplicación sólo acepta el idioma inglés, así que hay que hacer uso de éste.

Probamos a mantener la primera conversación y este es el resultado 3.2:

```
[jamarmar@r      neuralconvo]$ th eval.lua
Loading vocabulary from data/vocab.t7 ...
-- Loading model

Type a sentence and hit enter to submit.
CTRL+C then enter to quit.

you> hi
neuralconvo> Hi, how'd your day go?
you> fine, yours?
neuralconvo> You're not going to kill us?
you> if you're intelligent, you will live
neuralconvo> I did?
you> you have to show me more
neuralconvo> I don't think so.
you>
```

Figura 3.2: Primera conversación con el modelo obtenido

Traducción:

Tú: *Hola*

Modelo: *Hola, ¿cómo ha ido el día?*

Tú: *Genial, ¿y el tuyo?*

Modelo: *¿No vas a matarnos?*

Tú: *Si eres inteligente, vivirás.*

Modelo: *¿Lo fui?*

Tú: *Tienes que demostrarme más.*

Modelo: *No lo creo.*

Como vemos, la red ha podido mantener una conversación “básica”, pero con contestaciones algo avanzadas y sin perder el hilo. Después de un rato chateando con él podemos comprobar que tiene complicaciones con frases más elaboradas y a veces responde cosas sin sentido, pero da la sensación de tener cierta “inteligencia” con respuestas como la que hemos visto arriba. No vamos a centrarnos más en esta aplicación, pues sólo era una pequeña demostración del poder del Deep learning. A continuación, procedemos a entrenar una red para que sea capaz de reconocer todo tipo de imágenes.

3.3 Clasificador de Imágenes

3.3.1. Introducción al reconocimiento de imágenes

Una de las principales virtudes de cualquier animal es el sentido de la vista. Nuestra mayor parte del tiempo estamos haciendo uso de él para reconocer el entorno y es nuestro cerebro el que se encarga de obtener la información. Esta información puede ser la forma, el color o la distancia de los objetos que nos rodean. Por ello, uno de los mayores retos en la actualidad es dar a las máquinas la capacidad de “ver” y ser capaces de conseguir los máximos datos posibles de una imagen. Para cualquier programa con inteligencia artificial que trabaje en un entorno real, es necesario que pueda reaccionar a los cambios que se producen en éste. Por ejemplo, para los coches autónomos, la capacidad de reconocer una persona cruzando la calle o un semáforo en rojo es algo fundamental. Y es necesario que la precisión de este reconocimiento sea muy alta.

En este caso, vamos a entrenar una red neuronal para que sea capaz de clasificar los objetos de una imagen, lo cual puede servir en el futuro para crear un programa que pueda clasificar una imagen como se muestra en la figura 3.3 Obtener esta precisión con cualquier imagen es algo inalcanzable hoy en día, pero con el avance de la tecnología y las técnicas que usamos, en un futuro será algo real. Nosotros vamos a centrarnos en entrenar redes ya creadas y tratar de obtener la máxima precisión posible.

Nosotros vamo a hacer uso de un repositorio [8] que nos servirá como guía para inicializarnos en el entrenamiento de redes neuronales.

El repositorio trata sobre entrenar con Torch un clasificador de imágenes con objetos haciendo uso de los datos de Imagenet. Imagenet es una base de datos de imágenes creada específicamente para softwares de reconocimiento de imágenes. En ella, podemos encontrar 14 millones de imágenes donde se han descrito a mano los objetos que contiene. Así pues, estas anotaciones sirven para que los softwares entrenados puedan comprobar por sí mismos si han acertado o no

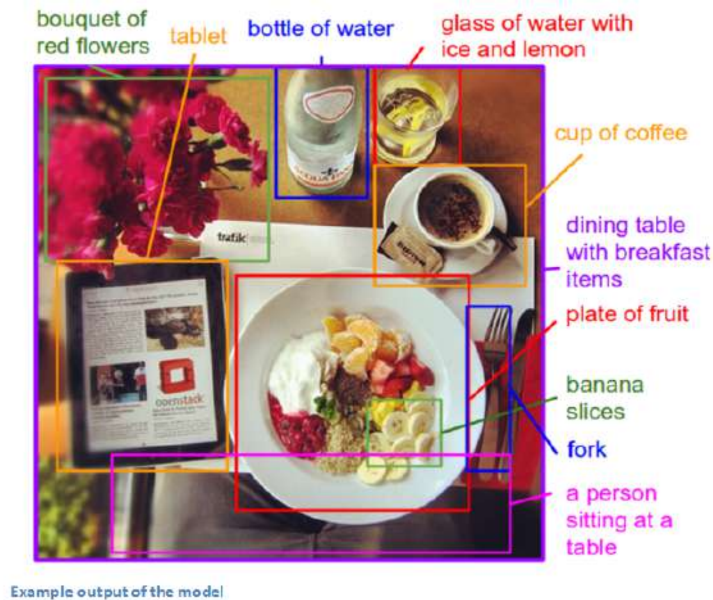


Figura 3.3: Reconocimiento de objetos avanzado de una red neuronal

su clasificación. Además, el repositorio también se centra en entrenar usando varias GPUs, es por esto por lo que hemos decidido focalizarnos en él.

3.3.2. Primer entrenamiento con el Clasificador

Para comenzar a familiarizarnos con el clasificador, vamos a realizar un primer entrenamiento básico y explicaremos los diferentes parámetros que vamos a usar para ello.
Para comenzar a trabajar, primero clonaremos el proyecto de Git en nuestra máquina:

```
git clone https://github.com/soumith/imagenet-multiGPU.torch.git
```

Luego, descargaremos nuestro conjunto de datos desde la página ofcial de Imagenet, concretamente en el enlace:
http://image-net.org/download-images
Al descomprimir, tendremos una carpeta imagenet' con dos directorios dentro, 'train' y 'val'. El primero es el conjunto de datos de entrenamiento, los cuales se usarán para el aprendizaje. El segundo directorio es el conjunto de datos de test, o validación. Metemos todo esto en una nueva carpeta llamada "data" y la moveremos dentro de del repositorio creado al clonar el proyecto.

El principal funcionamiento de esta aplicación está en el archivo "main.lua", que es el que se encarga de ejecutar todos los procesos requeridos. En resumen, este script realiza un bucle, donde cada iteración se llama época. En cada época se lanza un método train, que se encarga de ejecutar un número de iteraciones, donde realiza una predicción en cada iteración y modifica su modelo según acierte o no. Al terminar estas iteraciones, crea un modelo y estado óptimo nuevos moldeados según los resultados del entrenamiento. Estos se guardan

como dos archivos distintos en el directorio que le indiquemos. Después de esto, realiza un método test, donde se pone a prueba el modelo con distintas imágenes. En este caso, no se realiza aprendizaje ni se modifica el modelo. Al finalizar, comienza la siguiente época y sigue los mismos pasos. Cuando se han ejecutado todas las épocas, tenemos como resultado una carpeta con la red y parámetros que hemos usado, dentro de ésta la fecha que se inició el entrenamiento y en ella, los archivos generados. Estos archivos son:

- Cada uno de los modelos de cada época “model_[época].t7”. El modelo se representa como una matriz, donde cada elemento hace referencia al peso de la conexión entre los nodos de la red neuronal, como explicamos anteriormente.
- El estado óptimo de cada época “optimState_[época].t7”: se le introduce como parámetro los pesos del modelo y se guarda el error de cada peso concreto y su gradiente respecto al peso. Esto sirve para mejorar el aprendizaje en la siguiente época, centrándose en moldear los pesos que menos están acertando.
- Archivo train.log, que es un documento de texto con dos columnas y una fila por cada época. En la columna de la izquierda se encuentra avg loss, que es la media del error de cada peso de la red. En la columna derecha está % top1 accuracy, que es el mejor porcentaje de imágenes acertadas que ha obtenido el modelo en una iteración.
- Archivo test.log, es igual que el archivo train, sólo que esta vez son los resultados del caso de test y las columnas están intercambiadas (la de la izquierda es el porcentaje de acierto y la de la derecha es la pérdida).
- Archivo classes.t7. Es el archivo donde se guardan las clases que se usarán en el entrenamiento.

Este script tiene una gran variedad de parámetros que le podemos pasar y necesitamos conocer. Para obtener la lista de ellos, ejecutamos la ayuda mediante:

```
th main.lua -help
```

Y se nos muestra el listado por pantalla, como vemos en la figura 3.4:
Como vemos, la primera columna es el nombre del parámetro, seguido por una pequeña explicación en inglés, donde el valor entre corchetes (p.e. [55]) significa el valor por defecto, en caso de que omitamos el parámetro. Nosotros a continuación, explicaremos los que más vamos a usar:

- *cache* : subdirectorio donde guardar los archivos que generen las pruebas. Lo dejaremos por defecto.
- *data*: directorio que contiene los datos de entrenamiento y testo. Será donde hemos colocado la carpeta imagenet.
- *GPU*: índice de la GPU que deseamos usar. Usaremos el 1 por defecto cuando sólo nos haga falta una.

```
[jamarmar@i    : imagenet-multiGPU.torch]$ th main.lua --help
ω

Torch-7 Imagenet Training script

Options:
  -cache          subdirectory in which to save/log experiments [./imagenet/ heckpoint/]
  -data           Home of ImageNet dataset [./imagenet/imagenet_raw_images/2 6]
  -manualSeed     Manually set RNG seed [2]
  -GPU            Default preferred GPU [1]
  -nGPU           Number of GPUs to use by default [1]
  -backend        Options: cudnn | nn [cudnn]
  -cudnnAutotune  Enable the cudnn auto tune feature Options: 1 | 0 [0]
  -nDonkeys       number of donkeys to initialize (data loading threads) [2]
  -imageSize      Smallest side of the resized image [256]
  -cropSize       Height and Width of image crop to be used as input layer [224]
  -nClasses       number of classes in the dataset [1000]
  -nEpochs        Number of total epochs to run [55]
  -epochSize      Number of batches per epoch [10000]
  -epochNumber    Manual epoch number (useful on restarts) [1]
  -batchSize      mini-batch size (1 = pure stochastic) [128]
  -LR             learning rate; if set, overrides default LR/WD recipe [0]
  -momentum       momentum [0.9]
  -weightDecay    weight decay [0.0005]
  -netType        Options: alexnet | overfeat | alexnetowtbn | vgg | googlenet [alexnetowtbn]
  -retrain        provide path to model to retrain with [none]
  -optimState     provide path to an optimState to reload from [none]
  -wInit          Options kaiming | xavier  [none]

[jamarmar@r     imagenet-multiGPU.torch]$
```

Figura 3.4: Parámetros para main.lua

- *nGPU*: es el número de GPUs que usaremos. Nosotros utilizaremos una o dos según la prueba que realicemos.
- *backend*: es la librería de aceleradora de GPU. Nosotros usaremos cudnn, ya que es la que menos memoria requiere para ejecutar las pruebas.
- *nClasses*: es el número de clases que obtendremos del conjunto de datos. Vamos a dejar por defecto 1000, que es el número máximo de clases disponibles en Imagenet.
- *nEpochs*: es el número de épocas totales a realiar. Cada época realizará un número de iteraciones que también podemos cambiar. Este parámetro será uno de los principales que usaremos para realizar distintas pruebas.
- *epochSize*: este es el parámetro que determina el número de iteraciones que realizará cada época.
- *netType*: es el tipo de red neuronal que vamos a usar. En nuestra máquina solo funciona alexnet, overfeat y alexnetowtbn, ya que vgg y goglenet usan demasiados recursos y nos da un error de memoria sobrepasada.

En algunos casos tendremos que reiniciar un entrenamiento desde una época concreta. Para ello, usaremos los siguientes parámetros:

- *epochNumber*: indica el número de época desde el que comenzará el entrenamiento.
- *retrain*: ruta al modelo desde el que se continuará el entrenamiento.
- *optimState*: ruta al estado óptimo del que queremos partir en el nuevo entrenamiento.

Así pues, vamos a proceder a realizar el primer entrenamiento sencillo de una red neuronal:

```
th main.lua -data  /imagenet-multiGPU.torch/data/imagenet/ -nGPU 1
-backend cudnn -netType alexnet -nEpochs 50 -batchSize 10000
```

Usamos una sola GPU y alexnet como tipo de red. Será un entrenamiento de 50 épocas con 10000 iteraciones en cada una de ella. Lo ejecutamos y en la terminal vemos cómo va avanzando el proceso. Un entrenamiento sencillo como éste puede durar varios días. Por ello, para entrenar redes en Deep Learning conviene tener un servidor donde poder dejar ejecutando las pruebas en caso de realizar entrenamientos de gran magnitud. La otra solución es dejar nuestra computadora encendida hasta que se finalicen las pruebas.

Este es el principal problema para trabajar con Deep Learning, ya que se requieren muchos recursos para realizar pruebas de forma eficiente. Nosotros tenemos la suerte de disponer del cluster remoto de la UPV para conseguir resultados más rápidamente. Además, haremos uso de tmux, que es una herramienta de la terminal de linux para poder dejar las pruebas ejecutándose remotamente en el nodo. Para que se muestre el tiempo que tarda en finalizar el proceso, podemos añadir la instrucción time al inicio de nuestro comando:

```
time th main.lua -data  /imagenet-multiGPU.torch/data/imagenet/ -nGPU
1 -backend cudnn -netType alexnet -nEpochs 50 -batchSize 10000
```

Después de 91 horas y media ha finalizado el entrenamiento. Si vamos a la carpeta que se ha creado (dentro de checkpoint, con los parámetros como nombre), observamos una gran cantidad de archivos. Como hemos comentado anteriormente, hay 50 distintos modelos, 50 distintos estados óptimos, un archivo train.log y otro test.log. En caso de no haber usado el comando time o no haber guardado el tiempo, podemos comprobar cuánto ha tardado calculando la diferencia entre el tiempo de modificación del archivo classes.t7 (primer archivo creado) y el archivo train.log (el cual se edita al finalizar). Lo que nos interesa de todos estos archivos son train y test, ya que contienen los resultados obtenidos en cada iteración. Abrimos el archivo train.log y plasmamos los resultados como puede verse en la figura 3.5:
Como una captura de las 50 épocas era una imagen muy larga, hemos adjuntado las primeras 25 épocas a la izquierda y las 25 siguientes al lado derecho para crear una imagen más ancha y cómoda a la vista. En la figura vemos las dos columnas que comentamos anteriormente, donde a la izquierda está la media del error y a la derecha la precisión. Vamos a analizar los resultados obtenidos:

La primera iteración ha obtenido 6.4080e+00 de avg loss y 1.2014e+00 de accuracy. Hay que tener en cuenta que están dados en notación científica. Estos resultados son muy pobres, ya que alcanza poco más del 1 % de precisión y hay muchas pérdidas. Si continuamos observando las filas siguientes, vemos que estos números van mejorando rápidamente según avanza a la siguiente época. La mejora es mucho mayor al principio, ya que en los modelos iniciales hay mucho margen de mejora y una vez ha establecido un aprendizaje básico, le es más costoso mejorar. En la última iteración hemos obtenido un error de 1.77 y un por-

avg loss	% top1 accuracy		
6.4080e+00	1.2014e+00	3.0148e+00	3.5103e+01
5.0761e+00	8.3255e+00	3.0099e+00	3.5225e+01
4.4964e+00	1.4259e+01	2.9949e+00	3.5439e+01
4.2263e+00	1.7542e+01	2.9915e+00	3.5537e+01
4.0707e+00	1.9608e+01	2.5125e+00	4.4095e+01
3.9718e+00	2.0875e+01	2.3754e+00	4.6513e+01
3.8959e+00	2.2005e+01	2.3155e+00	4.7492e+01
3.8505e+00	2.2660e+01	2.2621e+00	4.8466e+01
3.8092e+00	2.3216e+01	2.2214e+00	4.9251e+01
3.7824e+00	2.3588e+01	2.1832e+00	4.9849e+01
3.7527e+00	2.4017e+01	2.1456e+00	5.0558e+01
3.7270e+00	2.4435e+01	2.1183e+00	5.0995e+01
3.7122e+00	2.4699e+01	2.0997e+00	5.1363e+01
3.6967e+00	2.4876e+01	2.0662e+00	5.1936e+01
3.6848e+00	2.5003e+01	2.0454e+00	5.2296e+01
3.6748e+00	2.5212e+01	2.0221e+00	5.2748e+01
3.6598e+00	2.5374e+01	2.0069e+00	5.2884e+01
3.6474e+00	2.5630e+01	1.9846e+00	5.3381e+01
3.1905e+00	3.2478e+01	1.8842e+00	5.5462e+01
3.1047e+00	3.3684e+01	1.8561e+00	5.5990e+01
3.0779e+00	3.4128e+01	1.8318e+00	5.6427e+01
3.0639e+00	3.4327e+01	1.8139e+00	5.6858e+01
3.0415e+00	3.4701e+01	1.8031e+00	5.6922e+01
3.0416e+00	3.4736e+01	1.7895e+00	5.7209e+01
3.0209e+00	3.5092e+01	1.7700e+00	5.7540e+01

Figura 3.5: Resultados del primer entrenamiento

centaje de acierto del 57.54 %. Esto significa que en la última época, nuestra red ha sido capaz de acertar ese porcentaje de objetos de los casos de entrenamiento. Antes de analizar más profundamente estos resultados, vamos a comprobar el archivo test.log, ya que es aquí donde se ve la eficacia de la red en un caso real de test. Al igual que anteriormente, insertamos los resultado como se muestra en la figura 3.6 Cabe recordar que en este archivo, las columnas de error y

% top1 accuracy	avg loss		
4.4200e+00	5.5624e+00	3.6068e+01	2.9827e+00
1.2650e+01	4.6100e+00	3.6116e+01	2.9628e+00
1.7394e+01	4.2400e+00	3.6804e+01	2.9253e+00
2.0638e+01	3.9839e+00	4.5552e+01	2.4401e+00
2.2332e+01	3.8434e+00	4.6690e+01	2.3982e+00
2.2030e+01	3.8656e+00	4.7488e+01	2.3476e+00
2.3476e+01	3.7943e+00	4.7884e+01	2.3182e+00
2.5334e+01	3.6490e+00	4.8402e+01	2.2965e+00
2.5338e+01	3.6533e+00	4.8570e+01	2.2937e+00
2.6026e+01	3.6217e+00	4.8700e+01	2.2933e+00
2.6938e+01	3.5321e+00	4.9072e+01	2.2703e+00
2.6728e+01	3.5666e+00	4.9228e+01	2.2626e+00
2.6288e+01	3.5892e+00	4.9378e+01	2.2471e+00
2.6420e+01	3.5917e+00	4.9800e+01	2.2370e+00
2.7406e+01	3.5392e+00	4.9690e+01	2.2361e+00
2.8350e+01	3.4756e+00	4.9920e+01	2.2267e+00
2.7138e+01	3.5410e+00	4.9790e+01	2.2377e+00
2.7130e+01	3.5482e+00	5.1676e+01	2.1516e+00
3.4886e+01	3.0518e+00	5.1878e+01	2.1389e+00
3.5478e+01	3.0100e+00	5.1962e+01	2.1443e+00
3.5274e+01	3.0191e+00	5.1836e+01	2.1455e+00
3.5976e+01	2.9719e+00	5.1954e+01	2.1393e+00
3.5824e+01	2.9624e+00	5.2196e+01	2.1275e+00
3.5894e+01	2.9829e+00	5.2222e+01	2.1400e+00
3.6642e+01	2.9365e+00	5.2106e+01	2.1357e+00

Figura 3.6: Resultados del test del primer entrenamiento

precisión están intercambiadas. Como podemos observar, son unos resultados parecidos al que obtuvimos en el entrenamiento. La precisión ha ido subiendo según avanzaban las épocas y el error medio ha ido bajando. En la última iteración ha conseguido una precisión del 52.11 %, algo por debajo del 52.22 % de la

penúltima iteración. Esto nos hace pensar que la red ha empezado a converger en este punto, pero no podemos asegurarlo sin realizar más épocas.

De esta manera, se ha conseguido un acierto del 57.54 % en entrenamiento y un 52.22 % en test. La diferencia se debe a que el modelo, después de muchas iteraciones entrenando, está “familiarizado” con estos datos y es capaz de predecirlos mejor. En cambio en el test, son datos totalmente nuevos. Así pues, podemos decir que nuestra red es capaz de reconocer el objeto de una imagen entre 1000 clases distintas un 52.22 % de las veces. Aún lejos del ojo humano, que alcanza una precisión del 98 %. Sin embargo, este es un gran logro, ya que en sólo 4 días y con unos recursos limitados hemos conseguido ese alto porcentaje.

Como hemos visto, nuestra red ha necesitado casi 4 días para lograr un modelo aceptable, realizando una prueba con dimensiones normales muy parecidas a las que vienen por defecto. Con esto ya nos hacemos una idea del inmenso tiempo que vamos a necesitar para ejecutar las pruebas que deseamos. Por ello, vamos a intentar disminuir lo máximo posible el tiempo de ejecución.

3.3.3. Reduciendo el tiempo de entrenamiento

Hemos comprobado que una gran parte del tiempo del entrenamiento está dedicado al test que se realiza al finalizar cada época. Si analizamos el código del archivo que ejecuta el test (test.lua), observamos que éste no modifica el modelo obtenido en el entrenamiento, sino que sólo realiza validaciones y devuelve los resultados obtenidos. Así pues, vamos a crear un nuevo archivo, mainTrain.lua, el cual es idéntico a main.lua, pero éste no realizará la etapa de test. Para conseguir esto, tenemos que modificar el archivo main.lua. Nosotros lo vamos a hacer con vim, que es un editor de texto en la terminal de linux. Primero creamos la copia del archivo main.lua:

```
cp main.lua mainTrain.lua
```

Abrimos el archivo con vim:

```
vi mainTrain.lua
```

Bajamos hasta las últimas líneas del código, donde se encuentran los procesos de train y test:

```
1   ...
2   epoch = opt.epochNumber
3
4   for i=1,opt.nEpochs do
5       train()
6       test()
7       epoch = epoch + 1
8   end
```

Figura 3.7: Archivo main.lua original

Pulsamos la letra "i" para comenzar el modo edición y poder insertar nuevos elementos en el archivo. Para que el entrenamiento eluda la fase de test, comentamos con dos guiones la línea que la realiza:

```
...
epoch = opt.epochNumber

for i=1,opt.nEpochs do
    train()
    --test()
    epoch = epoch + 1
end
```

Figura 3.8: Modificación de MainTrain.lua

Para guardar, tenemos que presionar la tecla Esc y escribir :wq para guardar los cambios y cerrar el editor. De esta manera, nuestra red pasará al entrenamiento de la siguiente época inmediatamente después de finalizar el entrenamiento de la anterior.

Para comprobar la eficacia de esta modifiación, vamos a realizar dos pruebas con los mismos parámetros que la primera, pero realizando 1000 iteraciones por época (-batchSize 1000), en vez de 10000, para que sea más breve. La primera prueba será con test, ejecutando el archivo main.lua y la segunda será usando el nuevo mainTrain.lua.

Una vez han finalizado, entramos a la carpeta correspondiente dentro de /checkpoint y observamos que hay dos carpetas. Estos directorios son las dos pruebas, cada una con su fecha de creación. Entramos a la primera creada, calculamos el tiempo que ha tardado y observamos que han sido 10 horas y 52 minutos (10.87 horas). El mejor porcentaje de acierto en train ha sido de 28.042 % (con 3.48 de error medio) en la última iteración. En test, ha alcanzado una precisión de 29.706, sorprendentemente algo más alto que el del train, pero esto se debe a que la red aún está lejos de converger y le falta mucho por aprender.

Ahora pasamos a comprobar los resultados que ha ofrecido la misma prueba sin realizar test: Observamos que ha tardado 8 horas y 33 minutos (8.55 horas), 2.27 horas menos que la prueba con test. En cuanto al porcentaje en train ha sido de 27.789, algo menor que antes, y con un error medio de 3.48, exactamente el mismo. Comprobamos que el archivo test.log está vacío, así que para comprobar testear este modelo, vamos a crear un nuevo archivo mainTest.lua. Este fichero será idéntico a mainTrain.lua, sólo que en vez de eliminar la fase donde se hace test, comentaremos la línea donde se ejecuta el train y así dejar sólo el test. Estando en el directorio de los resultados de la última prueba, vamos a ejecutar este nuevo archivo con unos parámetros un tanto especiales:

```
th /imagenet-multiGPU.torch/mainTest.lua -data
/imagenet-multiGPU.torch/data/imagenet/ -cache . -nGPU 1 -backend
cudnn -netType alexnet -epochSize 1000 -nEpochs 1 -optimState
./optimState_50.t7 -retrain ./model_50.t7 -epochNumber 50
```

Este comando ejecuta directamente una sola época del archivo mainTest.lua, con los mismos parámetros introducidos en la fase de train, pero añadimo -cache . , para guardar los resultados en la carpeta actual y -optimState y -retrain para tomar el modelo y estado óptimo correspondiente. Esto hace que se ejecute una única etapa de testeo al último modelo obtenido, que normalmente será el mejor (podemos elegir el que queramos). Al terminar (sólo ha tardado 3.56 minutos, 0.06 horas), vemos que se ha creado una nueva carpeta, con otro directorio dento de éste, ya que ha sido una ejecución nueva. Dentro de este último, observamos un archivo test.log, lo abrimos y observamos una única línea, donde la precisión conseguida ha sido de 29.82 % con un error medio de 33.55.

Para una visión más clara de los resultados obtenidos, hemos insertado los resultados obtenidos en la tabla de la tabla 3.1: Como podemos comprobar, en

Prueba	Precisión Train	Precisión Test	Tiempo
main	28.042	29.706	10.87 h
mainTrain + mainTest	27.789	29.820	8.55 h

Tabla 3.1: Tabla comparación de entrenamiento con test y sin test.

la segunda prueba hemos obtenido incluso mejor precisión en la etapa de test que la primera prueba ahorrando un 25,22Así pues, a partir de ahora vamos a ejecutar la prueba de esta manera, realizando primero el entrenamiento y luego el test, pues vamos a economizar una gran cantidad de tiempo. Además, vamos a usar la precisión de test para comparar diferentes ejecuciones, ya que es la prueba más parecida a los casos reales.

3.3.4. Comparando tipos de redes

Otro factor a tener en cuenta es el tipo de red que vamos a usar, ya que no todas tienen la misma arquitectura y ofrecen resultados distintos. No vamos a realizar todas las pruebas con cada uno de ellas, pues requeriría hacer todo tres veces, así que vamos a realizar el mismo entrenamiento con los mismos parámetros para cada una de las redes. De esta manera, podemos compararlas y averiguar cuál nos presenta las mejores prestaciones en nuestra aplicación para hacer uso de ella en nuestras pruebas venideras.

Para llevar a cabo esta comparación, vamos a realizar un entrenamiento estándar, con los mismos parámetros que el primer caso que llevamos a cabo, pero variando el netType en cada caso. Como comentamos en el apartado de los parámetros, en nuestra máquina sólo funcionan los modelos Alexnet, Overfeat y Alexnetowtbn, así que vamos a ejecutar el siguiente comando para comparar los resultados:

```
for i in alexnet, overfeat, alexnetowtbn; do th main.lua -data
/imagenet-multiGPU.torch/data/imagenet/ -nGPU 1 -backend cudnn
-netType $i -nEpochs 50 -epochSize 10000; done
```

Como podemos ver, hemos creado un bucle for para que cada iteración sea una prueba con un modelo distinto y así no hay necesidad de ejecutar cada una de las pruebas.

Después de un largo tiempo, obtenemos los resultados de train de cada tipo de red. Vamos a comprobar los resultados de test de todas las redes en las épocas 1, 10, 20, 30 y 50 para observar el avance de los resultados a lo largo del tiempo. Para esto, ejecutaremos el siguiente comando en cada uno de los directorios generados:

```
for i in 1,10,20,30,50; do th  /imagenet-multiGPU.torch/mainTest.lua
-data  /imagenet-multiGPU.torch/data/imagenet/ -cache . -nGPU 1
-backend cudnn -netType <red que queremos comprobar> -epochSize 10000
-nEpochs 1 -optimState ./optimState_$i.t7 -retrain ./model_$i.t7
-epochNumber $i; done
```

Esto creará 5 carpetas nuevas en cada directorio, donde cada una de ellas corresponderá al test realizado.

Introducimos estos resultados clasificados en una tabla con tres columnas: iteraciones, precisión de test y tiempo de ejecución para cada una de las redes. Luego, creamos la gráfica que podemos ver en la figura 3.9

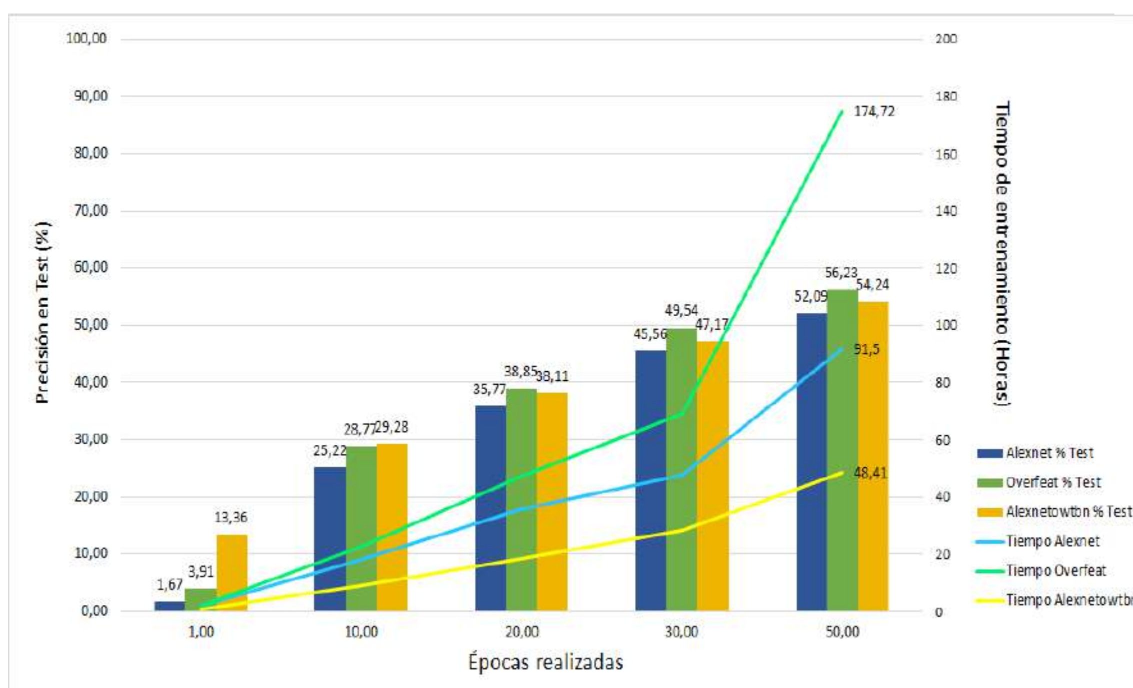


Figura 3.9: Gráfica comparación de los tres tipos de redes

Las columnas representan el porcentaje de acierto máximo obtenido en la etapa de test (eje izquierdo), siendo la más alta la mejor. Las líneas significan el tiempo necesitado (eje derecho), donde una mayor altura implica más tiempo requerido. Como podemos comprobar, Overfeat ha sido la red que mayor precisión ha obte-

nido en las pruebas, sin embargo, su tiempo de ejecución siempre ha sido mayor, sobre todo a partir de las 30 épocas, donde se ha disparado. Esto es debido a que el margen de ajuste no es tan grande y las redes necesitan más tiempo para calcular las predicciones. Para llegar a la época 50 con 56.226 % de precisión ha tardado más de 150 horas. En cambio, Alexnetowtbn ha obtenido 54.236 % en menos de 50 horas. Es decir, ha obtenido casi la misma precisión en 1/3 del tiempo requerido por Overfeat. Alexnet ha sido la que menos precisión ha obtenido y además su tiempo de ejecución ha sido cercano a 100 horas, lo que hace que la descartemos rápidamente.
Como hemos visto, los tres tipos de redes consiguen una precisión similar en épocas avanzadas, pero el tiempo de ejecución es muy variado. Así pues, alexnetowtbn es la red que más eficacia ofrece en nuestra aplicación y es la que vamos a usar a partir de ahora en las pruebas.

3.3.5. Obteniendo la máxima precisión posible

Ahora que hemos reducido notablemente el tiempo de entrenamiento que necesitan nuestras pruebas, nuestro objetivo será alcanzar la máxima precisión posible. Para ello, vamos a realizar diferentes entrenamientos, variando distintos parámetros y comparando los resultados.

Lo primero que hemos hecho ha sido variar las iteraciones por época (i.p.e) y hemos comprobado que 10.000 i.p.e es la mejor solución, ya que:

- 100 i.p.e: no realizaba ningún aprendizaje debido al poco entrenamiento entre época y época. Después de 2 horas y 500 épocas, la precisión fue de 1,09 %
- 1000 i.p.e: lograba una precisión de test del 34,818 % en la época 50 después de sólo 4 horas. Sin embargo, convergía también rápidamente, ya que en la época 150, la precisión era del 38,506 % y en la época 200, después de 17.21 horas, la precisión de test era del 38,816 %.
- 100.000 i.p.e: a pesar de obtener en la primera época una precisión del 23.224 % después de 8.63 horas, su nivel máximo de aprendizaje se ha estancado en una precisión del 31.882 % con un tiempo de ejecución de 62.03 horas.

p.e: no realizaba ningún aprendizaje debido al poco entrenamiento entre época y época. Después de 2 horas y 500 épocas, la precisión fue de 1,09 %la época 150, la precisión era del 38,506 % y en la época 200, después de 17ciones por época (i.p.e) antes de que converga. Como ya tenemos el modelo de la época 50 que hemos obtenido en la comparación de redes, sólo hay que continuar entrenándolo. Vamos a su correspondiente directorio y proseguimos el entrenamiento desde la época 50 con el siguiente comando:

```
th  /imagenet-multiGPU.torch/mainTrain.lua -data
/imagenet-multiGPU.torch/data/imagenet/ -cache . -nGPU 1 -backend
cudnn -netType alexnetowtbn -epochSize 10000 -nEpochs 30 -optimState
./optimState_50.t7 -retrain ./model_50.t7 -epochNumber 51
```

Esta ejecución tiene los mismos parámetros que el entrenamiento inicial, sólo que vamos a realizar 30 épocas más desde la época 51 con el modelo y estado óptimo de la época 50.
Una vez ha finalizado, ejecutamos el test para las épocas 60,70 y 80 , recopilamos los resultados y creamos la gráfica que vemos en la figura 3.10:

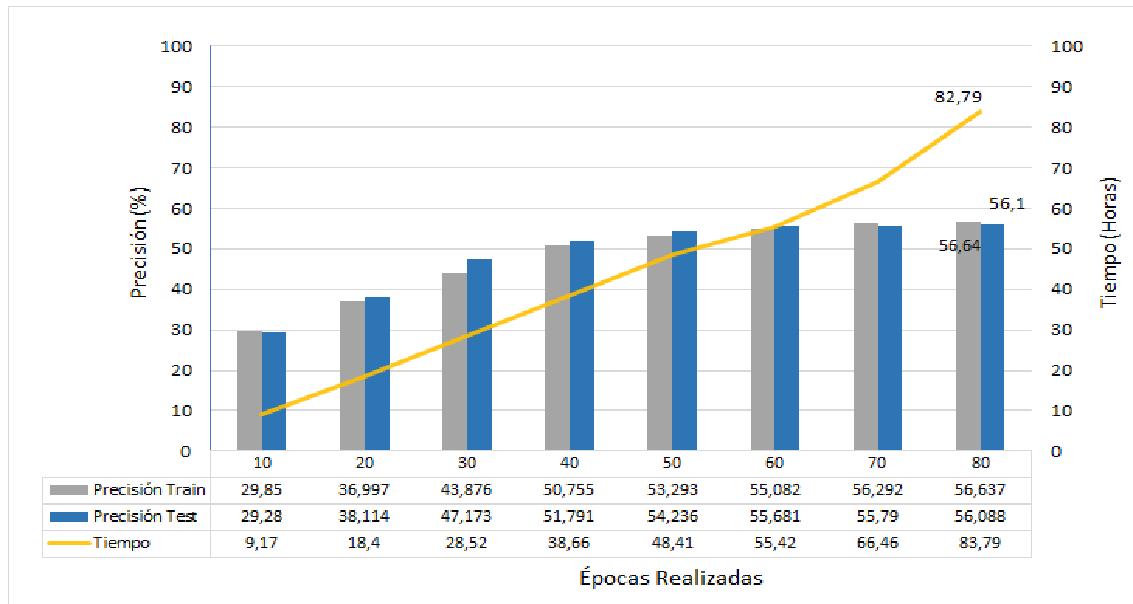


Figura 3.10: Entrenamiento Alexnetowtbn 10.000 i.p.e

Como observamos, el nivel máximo de precisión en test ha sido del 56.088 % en la última época, habiendo tardado 82.79 horas. Sin embargo, si observamos la época 50, el porcentaje de precisión es del 54.236 %, sólo un 1.852 % de diferencia y el tiempo en este punto era de 48,41 horas, que es un 42.53 % menor al tiempo en la época 80. Así pues, podemos asumir que la red ha comenzado a converger a partir de la época 50 aproximadamente y en relación precisión-tiempo, no sale rentable entrenar más de esta época.

3.4 Entrenando con dos GPUs

Uno de nuestros objetivos es conseguir el mejor entrenamiento en el menor tiempo posible. Por ello, vamos a comprobar cómo afecta a los resultados realizar los entrenamientos con 2 GPUs. Nosotros hemos podido disponer de las 2 GPUs del Nodo 2 que nos ha ofrecido la universidad, pero disponer de 2 GPUs no está al alcance ecónomico de cualquier usuario que desee realizar experimentos de Deep Learning por afición. Sin embargo, existen páginas web que permiten “alquilar” un nodo remoto donde se dispone de varias GPUs de

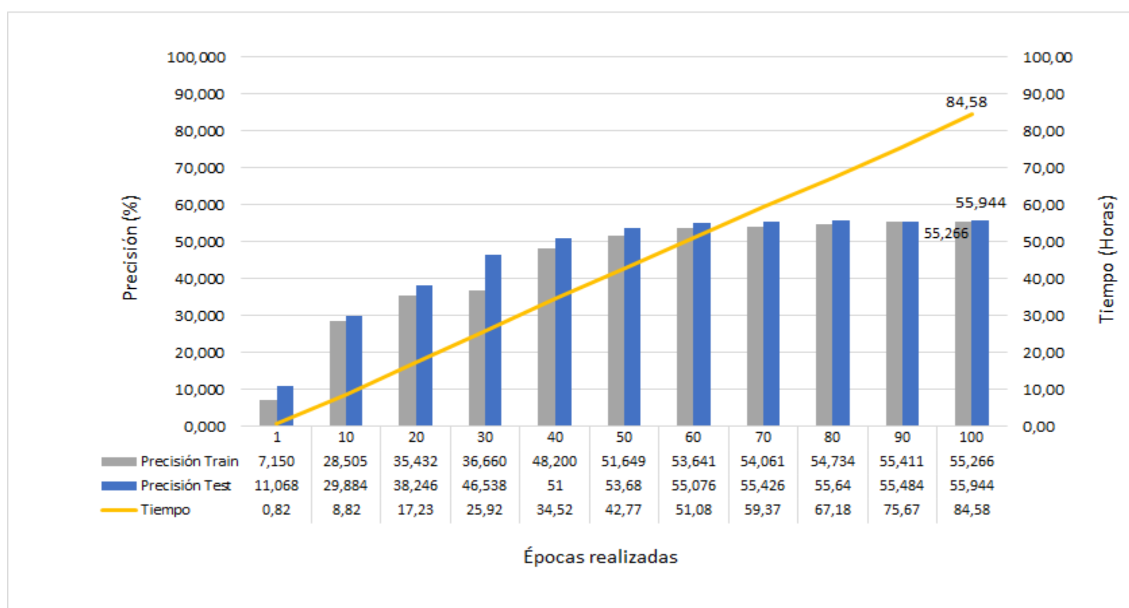
gran calibre para realizar pruebas y el gasto que debes realizar está sujeto al tiempo que necesites el nodo. Aquí tenemos varias páginas que permiten esto:
https://www.leadergpu.com/
https://www.foxrenderfarm.com/

La primera prueba que vamos a llevar a cabo es con 10,000 i.p.e, ya que ha sido la que mejores resultados ha ofrecido. Como disponemos de 2 GPUs, vamos a realizar un entrenamiento de 100 épocas para poder tener una amplia visión de cómo ha avanzado la prueba y comprobar si obtenemos grandes diferencias en cuanto a la precisión y la convergencia obtenidas con una única GPU. Así pues, realizaremos la siguiente ejecución:

```
th  /imagenet-multiGPU.torch/mainTrain.lua -data
/imagenet-multiGPU.torch/data/imagenet/ -nGPU 2 -backend cudnn
-netType alexnetowtbn -epochSize 10000 -nEpochs 100
```

Es importante tener en cuenta el parámetro -nGPU 2 para hacer uso de ambas GPUs.

Cuando ha acabado la ejecución, realizamos el test para las épocas que son múltiplo de 10, guardamos los resultados en una tabla y generamos la gráfica de la figura 3.11. Todo esto siguiendo el mismo procedimiento que hemos hecho hasta ahora.



	1	10	20	30	40	50	60	70	80	90	100
Precisión Train	7,150	28,505	35,432	36,660	48,200	51,649	53,641	54,061	54,734	55,411	55,266
Precisión Test	11,068	29,884	38,246	46,538	51	53,68	55,076	55,426	55,64	55,484	55,944
Tiempo	0,82	8,82	17,23	25,92	34,52	42,77	51,08	59,37	67,18	75,67	84,58

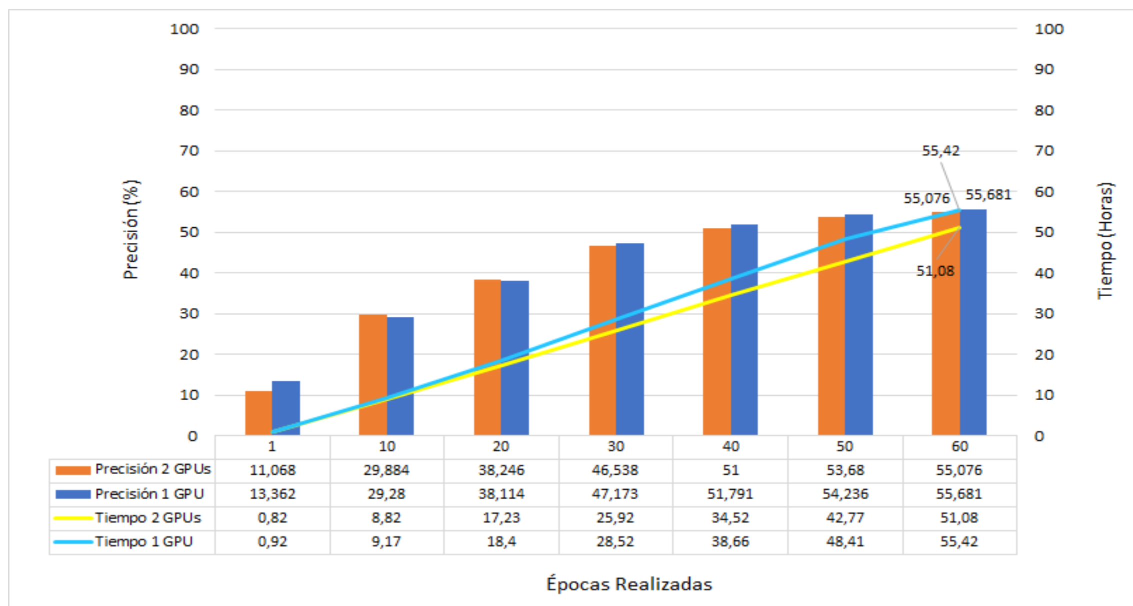
Figura 3.11: Entrenamiento Alexnetowtbn 10.000 i.p.e 2 GPUs

Con el primer vistazo podemos observar una clara convergencia de las columnas de precisión a partir de las épocas 50 y 60. También vemos una progresión lineal del tiempo, al contrario que con una GPU, donde se disparaba cuando empezaba a converger.

Analizando los datos, la mayor precisión obtenida ha sido en la fase de test de la época 100 con un 55.944 % después de 84.58 horas. En train no se ha su-

perado el 55.411 % en la época 90. Esto indica que el margen de entrenamiento es mínimo. En cuanto al punto de inflexión en el que empieza a converger, observamos una precisión de test del 55.076 % en la época 60 después de 51.08 horas. Vemos que a partir de la época 60 se ha conseguido aumentar un 0.868 % la precisión habiendo invertido 33 horas y media más, lo que deja claro la convergencia comentada.

Ahora vamos a comparar estos resultados con los obtenidos realizando el entrenamiento con una GPU. Para ello, cogemos la precisión de test y el tiempo de ejecución de ambas pruebas hasta la época 60, que es donde ambas han alcanzado el punto de inflexión y generamos una nueva gráfica, como vemos en la figura 3.12.



	1	10	20	30	40	50	60
Precisión 2 GPUs	11,068	29,884	38,246	46,538	51	53,68	55,076
Precisión 1 GPU	13,362	29,28	38,114	47,173	51,791	54,236	55,681
Tiempo 2 GPUs	0,82	8,82	17,23	25,92	34,52	42,77	51,08
Tiempo 1 GPU	0,92	9,17	18,4	28,52	38,66	48,41	55,42

Figura 3.12: Comparación entrenamiento 1 GPU vs 2 GPUs

Como podemos ver, la diferencia entre la precisión de cada una es muy similar en todas las épocas, siendo un poco superior la conseguida por el entrenamiento con 1 GPU en las épocas finales. En la época 60, hemos obtenido un 55.681 % con una GPU frente al 55.076 % con dos GPUs, una diferencia mínima. Esta poca desemejanza es lógica, ya que el hecho de usar una GPU más no implica lograr una mejor predicción ni un moldeado más correcto del modelo.

En cuanto al tiempo dedicado, observamos que se van distanciando cada vez más según avanza el entrenamiento. Con una GPU se ha completado en 55.42 horas, mientras que con dos GPUs ha sido en 51.08 horas, un 7,83 % más rápido. Esta no es una diferencia notable en comparación con las expectativas que genera haber duplicado el número de GPUs. Esto puede deberse a que el código no está completamente optimizado para el uso de múltiples GPUs. Sin embargo, si tomamos una visión de un entrenamiento mucho mayor, supone un ahorro importante de tiempo.

CAPÍTULO 4
Ejemplo práctico: Neural Style

4.1 Explicación del algoritmo

Hasta ahora, solo hemos entrenado redes para reconocer el objeto de una imagen. En este reconocimiento se llevan a cabo identificaciones de patrones comunes entre los objetos de las imágenes. De esta manera, una red entrenada para reconocer estos patrones será capaz de identificarlos en cualquier imagen, aunque ellos no formen un objeto concreto. Con este ejemplo queremos realizar una interesante demostración visual de la utilidad de este aprendizaje y otra muestra de la potencia del Deep Learning.

Este repositorio[9] es una implementación en Torch realizada por Justin Johnson del Artículo *A Neural Algorithm of Artistic Style*[10] de Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge.

Este artículo trata sobre el desarrollo de un algoritmo neuronal capaz de reproducir la información que procesa el cerebro cuando ve una imagen artística. El principal factor es cómo la vista humana puede distinguir entre el contenido y el estilo de una imagen. Para ello, se crea una red neuronal como las hemos usado que extraiga los patrones de las imágenes en capas, cada una de estas capas posee una información diferente de la imagen.

El artículo pretende demostrar que si se seleccionan las capas concretas, pueden reconstruir el contenido de una imagen. Para esto, entrenan una red neuronal profunda que se añade capas del contenido gradualmente en una imagen nueva y comprueba si verifica si está 'acertando' si la forma creada coincide con el contenido general de imagen original. Para obtener el estilo de una imagen, añaden un filtro de patrones texturas en cada capa y realizan el mismo procedimiento, creando una red neuronal que cree un nuevo estilo y compruebe si coincide con el original.

En la figura 4.1 vemos las capas del estilo que se han obtenido de una imagen y las capas contenido de otra distina:

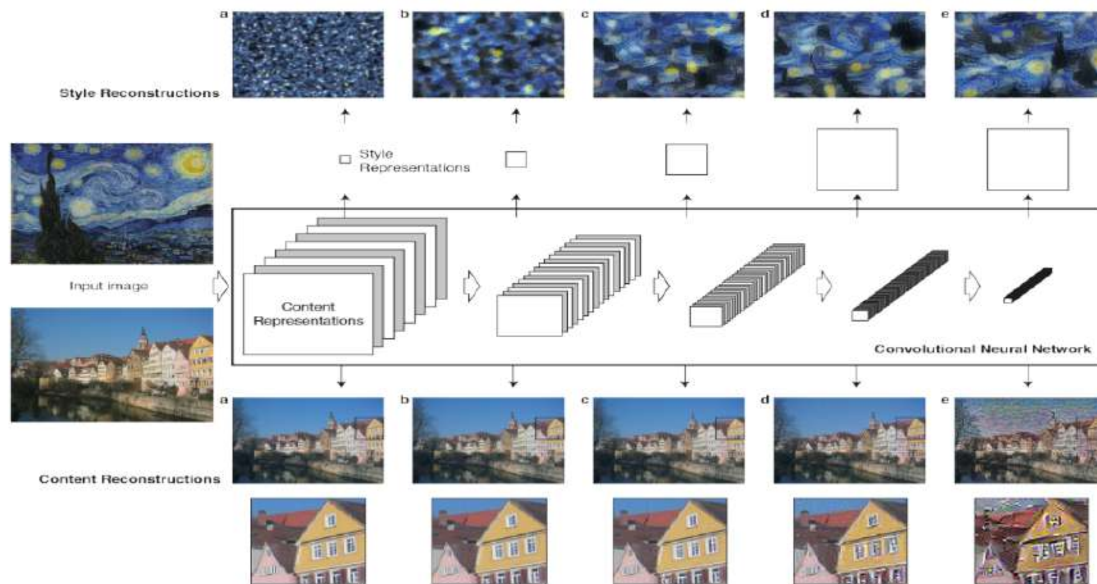


Figura 4.1: Clasificación de estilo y contenido de la imagen por capas.

En la parte de arriba de la figura, vemos el cuadro La Noche Estrellada de Van Gogh, de la cual se han obtenido diferentes capas por patrones y se han clasificado como estilo. Abajo vemos el mismo caso, pero con una imagen de la ciudad alemana de Tubingen, la cual vemos cómo la red neuronal ha obtenido la información de la casa por sus capas. En el centro, vemos las diferentes capas que se han obtenido en el modelo de la red neuronal.

Una vez disponen de las capas, el algoritmo reconstruye una imagen mezclando los patrones obtenidos del contenido con los patrones obtenidos del estilo.

Como vemos en la figura 4.2, el ejemplo A ha sido el contenido de la nueva imagen creada. El estilo de ésta ha sido cada una de las pequeñas imágenes en la esquina inferior izquierda. Como se puede observar en la figura, cada uno de los ejemplos resultantes son como “obras de arte” pintadas mezclando el contenido que transmite la imagen A con el estilo que transmite cada ejemplo.

Podemos intuir varios patrones que se siguen en la reconstrucción de la imagen, como son:

- Las formas y los objetos de la imagen contenido se han mantenido en la nueva imagen fielmente en la mayoría de casos, excepto en el ejemplo F. Por ejemplo, el cielo se ha recompuesto por los cielos de las imágenes estilo que también tenían.

- Se han reconstruido las formas de la imagen contenido con los patrones que coinciden en el estilo. Por ejemplo, la torre más alta del contenido con la torre de la Noche estrella en el ejemplo.

- Los colores de la imagen contenido se han sustituido muchas veces por los colores más similares de la imagen estilo.

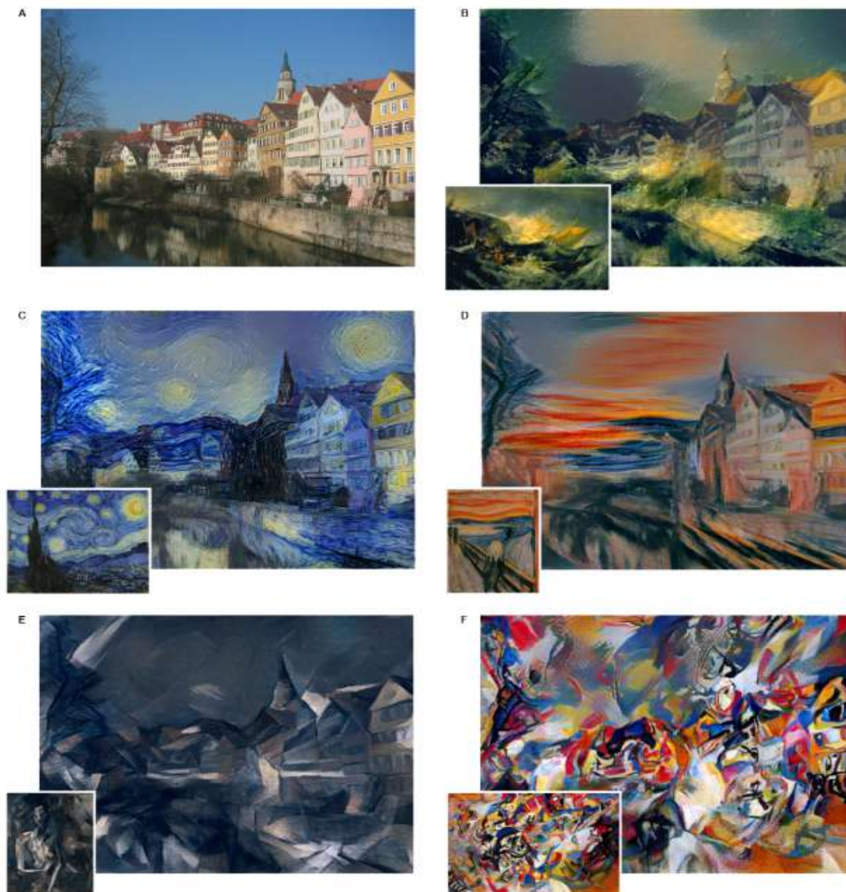


Figura 4.2: Distintas imágenes creadas mediante el algoritmo.

- Se ha tratado de mantener objetos del estilo en caso de que esa posición del contenido no sea importante, como vemos con la luna de Noche Estrellada en el ejemplo C.

Todos estos patrones han significado las distintas capas, las cuales se han mantenido o eliminado.

Lo que realiza este proyecto es tratar de reproducir en Torch ese algoritmo, el cual los creadores implementaron en Caffe. Para ello, vamos a hacer uso de las redes neuronales profundas ya entrenados para obtener nuestras imágenes con la mejor calidad en el menor tiempo posible. Estas redes obtendrán varios parámetros:

- -style_image <image.jpg>: imagen estilo, la cual determinará el estilo de la imagen generada.
- -content_image <image.jpg>: imagen contenido, será la imagen principal de la nueva imagen.
- -image_size: tamaño máximo en píxeles de la nueva imagen generada. Por defecto son 512.
- -gpu: índice de la o las GPUs que queremos usar. En caso de querer usar CPU, insertar -1 en este parámetro.

- -content_weight: el porcentaje de peso que tendrá el contenido en la nueva imagen. Una cantidad mayor indica una pérdida menor de información.
- -style_weight: el porcentaje de peso que tendrá el estilo en la nueva imagen. Esto quiere decir que los patrones y colores de la imagen estilo predominarán en la imagen creada.
- -num_iterations:número de iteraciones máximos que realizará la red neuronal.
- -output_image:nombre y ruta en la que queremos guardar la nueva imagen.
- -save_iter: guarda la imagen resultado cada este número de iteraciones.
- -content_layers: nombre de las capas generadas que se usarán para el contenido de la imagen.
- -style_layers: nombre de las capas generadas que se usarán para el estilo de la imagen.
- -style_scale: escala a la que se extraerán los elementos del estilo, cuanto mayor sea, más se aumentará el tamaño de los patrones del estilo en la imagen resultado. Por defecto es 1.
- -original_colors: opción para sólo mantener los colores de la imagen contenido. Puedes activar esta opción cambiando su valor de 0 a 1.
- -proto_file: ruta al archivo .txt que contendrá la información de las capas de cada modelo.
- -model_file: ruta al modelo Caffe descargado.
- -backend: tipo de backend que vamos a usar. Nosotros seguiremos con cudnn porque es el que mejor funciona con GPUs Nvidia,

Como vemos son muchos parámetros y usaremos muchos de ellos, por esta razón el comando de ejecución será algo extenso. Nosotros vamos a tratar de conseguir reconstruir imágenes usando el estilo de una y el contenido de otra distinta. Además, vamos a realizar una etapa de test, ya que vamos a probar el modelo en escenarios nuevos con diferentes estilos. Esto lo haremos introduciendo varios parámetros al modelo loadcaffe implementado para Torch. Estos parámetros pueden ser la imagen contenido, la imagen estilo y demás variaciones que luego comentaremos.

Primero, realizaremos varias pruebas con la misma imagen contenido para comparar resultados de estilo y luego variariemos el contenido para un mismo estilo. Además, probaremos también los resultados que ofrece realizar las pruebas usando las dos GPUs, ya que al crear una imagen grande, con muchos píxeles, significa más información que procesar y esto requiere un consumo de tarjeta gráfica alto. Incluso veremos cómo no podremos ejecutar ciertas pruebas con una sola GPU debido a un error out of memory, que significa que nuestra GPU no puede procesar tanta información.

Así pues, vamos a seguir la guía del repositorio[9] para comenzar. Clonamos el proyecto en nuestra máquina como hemos hecho anteriormente: Git clone https://github.com/jcjohnson/neural-style.git
Instalamos Protobuf, que son las librerías para poder compilar archivos Caffe

```
sudo apt-get install libprotobuf-dev protobuf-compiler
```

Luego instalamos loadCaffe, que es una aplicación de Torch capaz de cargar las redes neuronales de Caffe.

```
luarocks install loadcaffe
```

Después de instalar las dependencias anteriores, vamos al directorio clonado y dentro de la carpeta models, ejecutamos el siguiente script:

```
sh models/download_models.sh
```

Este script descargará el primer modelo que usaremos, que es el modelo VGG-19 original. En adelante descargaremos más, pero de momento vamos a comenzar con el mismo que la guía.

4.2 Primeras pruebas

Antes de empezar con las pruebas, vamos a organizar nuestras imágenes. Crearemos un directorio inputs, y dentro suya, styles y contents. Así tendremos clasificadas las imágenes que usaremos entre estilo y contenido. Además, crearemos otra carpeta outputs para las imágenes generadas.

La primera imagen contenido que vamos a usar es la siguiente (figura 4.3):

Figura 4.3: Foto de la Ciutat de les Arts i les Ciències, Valencia.

Hemos elegido esta imagen porque dispone de solo dos elementos importantes que son los dos edificios del centro y la derecha. También nos interesa observar cómo reconstruye los edificios de la izquierda y el cielo y cómo remoldea el agua con los reflejos. En general, es una imagen que valdría como contenido para una obra de arte.

El primer estilo que vamos a usar es el de la Noche Estrellada de Van Gogh (figura 4.4), ya que es un cuadro muy característico y es más sencillo percibir los elementos artísticos característicos.

Figura 4.4: *Starry Night*. Famoso cuadro pintado por Vincent Van Gogh en el 1889.

La primera prueba que vamos a ejecutar es la que está en el repositorio, ya que contiene las capas que se usarán en el contenido y las que serán del estilo.

```
th neural_style.lua -style_image inputs/starry_night.jpg -content_image
inputs/ciutatNoche.jpg -output_image ciutatStarryN.png -model_file
models/nin_imagenet_conv.caffemodel -proto_file models/train_val.prototxt
-gpu 0 -backend cudnn -num_iterations 1000 -seed 123 -content_layers
relu0,relu3,relu7,relu12 -style_layers relu0,relu3,relu7,relu12
-content_weight 10 -style_weight 1000 -image_size 640 -optimizer adam
```

Al ejecutar este comando en la carpeta raíz del repositorio, nos ha dado un error con el backend, así que lo hemos cambiado a cuda. Luego, ha salido otro error, ya que no se encontraba el modelo indicado. Hemos visto que en el directorio models hay 2 modelos .loadcaffe y dos archivos .prototxt, que son los archivos que contienen la información de las capas que extrae cada modelo anterior. Los del comando no coincide con ninguno de estos dos, así que seleccionamos el VGG 19. Seguidamente, hemos obtenido un error extraño se size mismatch (tamaño no coincide). Así que probamos a dejar las capas que vienen son seleccionadas por defecto y probamos:

```
th neural_style.lua -style_image inputs/starry_night.jpg -content_image
inputs/ciutatNoche.jpg -output_image ciutatStarryN.png -model_file
models/VGG_ILSVRC_19_layers.caffemodel -proto_file
models/VGG_ILSVRC_19_layers_deploy.prototxt -gpu 0 -backend cudnn
-num_iterations 1000 -seed 123 -content_weight 10 -style_weight 1000
```

```
-image_size 512 -optimizer adam
```

Una vez ha terminado, después de nosecuantos minutos, vamos a la carpeta output y vemos que se han creado 10 imágenes, cuyo nombre es prueba1_X, donde X es la época que se ha creado esa imagen. Es decir, cada 100 épocas la red ha creado una imagen reultante según estaba entrenado el modelo en ese momento. En la figura 4.5 podemos observar las distintas imágenes creadas:

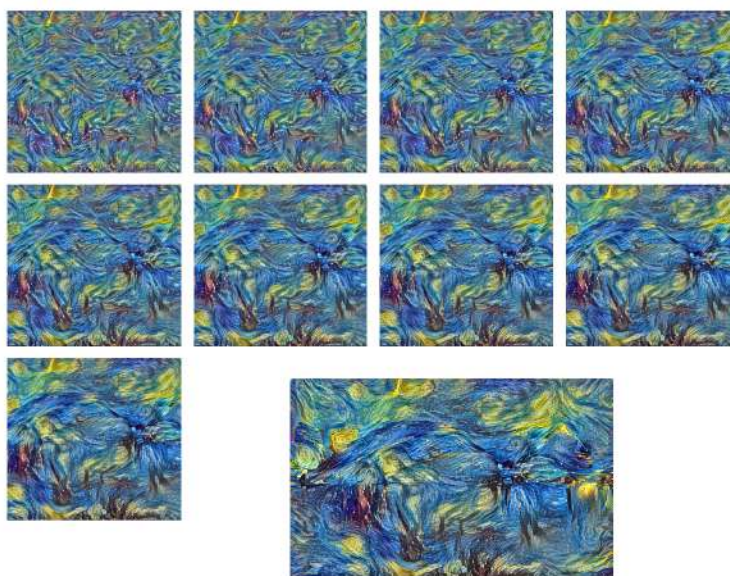


Figura 4.5: Resultados primera prueba con las capas seleccionadas por defecto.

Hemos colocado en la esquina superior izquierda el resultado obtenido en la época 100 y hacia la derecha es la siguiente con 100 épocas más. La última imagen es más grande, ya que es cuando la red ha alcanzado su punto más alto de aprendizaje, generalmente, y la imagen creada es “mejor”.

Como vemos, el resultado no ha sido bueno, ya que no se distingue el contenido. Se observa la forma de las figuras un poco, pero no es lo que buscamos.

Buscamos información por el foro del repositorio y encontramos una página donde se pueden descargar modelos externos: https://github.com/jcjohnson/neural-style/wiki/Using-Other-Neural-Models
Nosotros descargamos los siguientes, los cuales vemos más interesantes:

- VGG16_SOD_finetune, que es una red neuronal profunda que se centra en el reconocimiento de objetos. Fue desarrollada en 2016.
- SOD Finetune ’Low Noise’. Es una red que ha sido entrenada minuciosamente con grandes cantidad de datos y luego además con pequeñas.
- Rough_Faces. Esta red neuronal ha sido entrenada específicamente para detectar caras masculinas y femeninas. Así podemos probar con rostros. Nosotros descargamos un modelo que se ha entrenado después de 14.000 iteraciones.

Realizamos la siguiente prueba con Rough Faces:

```
th neural_style.lua -style_image inputs/starryNight.jpg -content_image
inputs/ciutatNoche1.png -output_image outputs/ciutatStarryN.png;
-model_file models/VGG16_SOD_finetune_rough_faces_iter_14000.caffemodel
-proto_file models/vgg16_train_val.prototxt -gpu 0 -backend cudnn
-num_iterations 1000 -save_iter 0 -seed 123 -content_weight 10
-style_weight 1000 -style_scale 1 -image_size 860 -content_layers
relu1_1,relu2_1,relu3_1,relu4_1,relu5_1 -style_layers
relu1_1,relu2_1,relu3_1,relu4_1,relu5_1
```

Y obtenemos la siguiente imagen reconstruida (4.6):

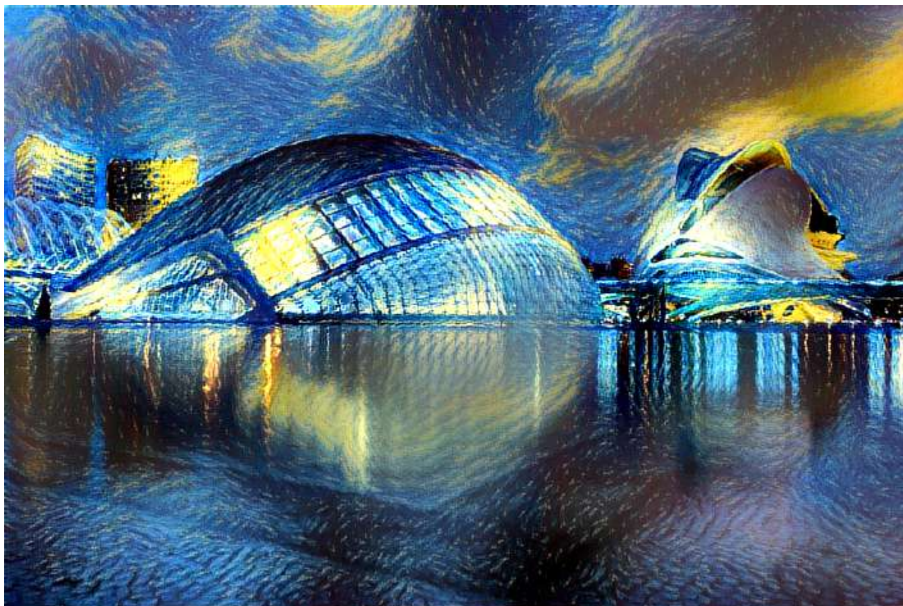


Figura 4.6: Resultado obtenido con modelo Rough Faces

En este caso, el resultado sí que obtiene una calidad visible a primera vista, ya que se distingue perfectamente el contenido y está "pintado" con el estilo del cuadro de Van Gogh. Aunque hay detalles que se podrían mejorar, es un resultado aceptable para esta primera prueba. A partir de ahora usaremos el modelo Rough Faces para nuestras pruebas, así ahorramos tiempo de ejecución.

4.3 Variando el estilo para una misma imagen

Para comprobar cómo varía el estilo en el mismo contenido, ejecutamos esta misma prueba, pero sólo cambiando el parámetro de la imagen estilo- La figura 4.7 son algunos de los resultados obtenidos:

Figura 4.7: Distintos ejemplos variando el estilo de la reconstrucción

En estos ejemplos podemos ver claramente cómo ha variado el estilo de cada imagen, pero siempre manteniendo fielmente el contenido de la original. El uso de los colores está muy logrado, ya que a pesar de ser una foto nocturna, en el primer ejemplo, por ejemplo, al coger la paleta de colores de un cuadro con colores claros y vivos, parece una foto diurna. Es decir, el estilo de la nueva foto ha suplantado totalmente el original.

4.4 Variando el contenido para un mismo estilo

Ahora vamos a probar el mismo estilo en diferentes imágenes contenido. El estilo que hemos elegido ha sido la figura 4.8

Figura 4.8: Cuadro del artista contemporáneo Leonid Afremov.

Esta imagen posee un estilo característico debido a sus tonos azules y naranja con pinceladas largas y gruesas. Para comprobar cómo se mantiene este estilo, vamos a seleccionar varias imágenes con distinto contenido: un monumento, un cuadro artístico, un retrato y un animal.

Realizamos el mismo comando que anteriormente, cambiando las imágenes de entrada correspondientes y obtenemos los resultados que se muestran en la figura 4.9:

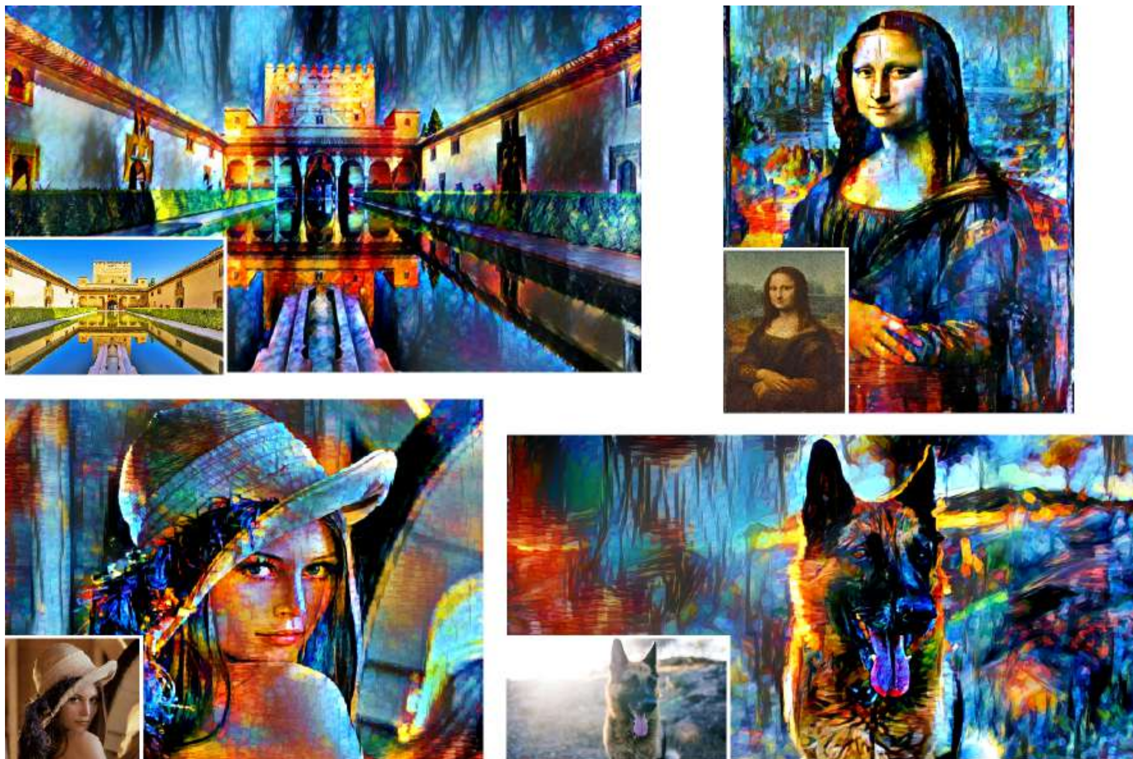


Figura 4.9: Distintos ejemplos obtenidos variando el contenido.

Podemos comprobar que todos los ejemplos han mantenido su contenido y además, se han reconstruido usando los colores y las trazas que comentábamos anteriormente, logrando que parezcan haber sido pintados siguiendo los patrones del estilo.

4.5 Usando múltiples GPUs

Como hemos comentado anteriormente, el algoritmo se centra en procesar las capas estilo y contenido de la imagen y genera una nueva imagen reconstruida a partir de ellas. Con dos GPUs, podemos dividir esta tarea entre ambas tarjetas gráficas. Es decir, trabajar paralelamente. Con esto podemos conseguir renderizar imágenes más rápidamente y además, reconstruir imágenes con un gran tamaño sin obtener el error de memoria sobrepasada.

Lo primero que vamos a comprobar es la diferencia realizando una prueba con una GPU y exactamente la misma prueba, pero haciendo uso de ambas GPUs. Para llevar a cabo una reconstrucción con múltiples GPUs, debemos cambiar los siguientes parámetros de nuestra ejecución:

- -gpu 1,2. Aquí indicamos los índices de las GPUs que vamos a usar.
- -multigpu_strategy 4. Con este nuevo parámetro estamos indicando que la GPU 1 se encargará de computar las capas 1, 2 y 3 y la GPU 2 computará las capas 4 en adelante. En caso de disponer de más GPUs, habría que añadir tantas asignaciones como GPUs.

La imagen contenido (figura 4.10):

Figura 4.10: Fotografía de la ciudad de Valencia

La imagen estilo (figura 4.11):

Figura 4.11: *Guernica*, cuadro pintado por Pablo Picasso en 1937.

Hemos elegido esta imagen contenido, ya que tiene un tamaño grande para poder distinguir mejor los detalles obtenidos en cada proceso y dispone de una gran variedad de colores. La razón de la imagen estilo es debido a su complejidad de formas. Además, así podremos observar también cómo queda el resultado manteniendo los colores del contenido con el parámetro -original_colors.

Así pues, realizamos ambas pruebas poniendo el comando time al inicio de la ejecución para guardar el tiempo que tarda cada una.

Una vez tenemos los resultados, comprobamos que se han obtenido imágenes totalmente idénticas (excepto por algún detalle casi imperceptible). Esto es debido a que el algoritmo y el modelo es el mismo, así que obtiene los mismos resultados para cada patrón. La diferencia es que cada capa está siendo computada por una GPU. En cuanto al tiempo de ejecución, la prueba con una sola GPU ha tardado 8 minutos y 32 segundos, y la prueba con las dos GPUs, ha tardado 7 minutos y 46 segundos. La diferencia de tiempo ha sido bastante baja, lo que significa que el algoritmo no está optimizado completamente para ejecutarse más rápidamente con dos GPUs.

En la figura 4.12 podemos comprobar la imagen resultante:

Figura 4.12: Imagen resultado con 2 GPUs.

Ahora, volvemos a realizar cualquiera de las pruebas anteriores añadiendo el parámetro -original_colors 1 para mantener los colores del contenido y obtenemos la reconstrucción de la figura 4.13.

Figura 4.13: Imagen reestructurada con colores originales del contenido.

Como podemos apreciar, la forma de esta imagen es prácticamente igual que la anterior. Sin embargo, los colores son una combinación entre los del contenido y el estilo, ya que se ha mantenido la gama de colores original, pero con unos tonos más claros al haberse mezclado con el blanco.

4.5.1. Creando una imagen de gran tamaño

Hasta ahora, las imágenes que hemos creado tenían una resolución máxima de 860 píxeles, ya fuese de largo o ancho. Este tamaño es un tamaño aceptable para obtener imágenes con este algoritmo. Ahora vamos a tratar de conseguir una imagen de 1200 píxeles, lo que implica obtener una reestructuración más elaborada.

La imagen contenido será la siguiente (figura 4.14):

Figura 4.14: Fotografía de Dublín, Irlanda.

Con la siguiente imagen estilo (figura 4.15):

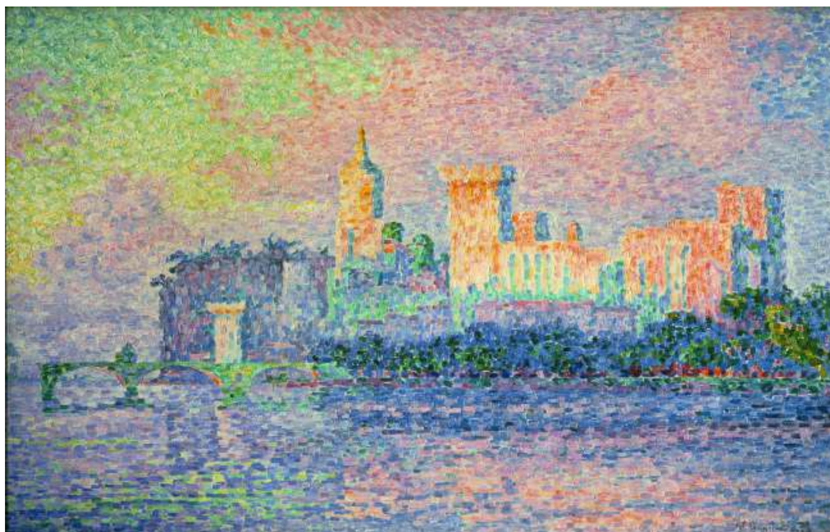


Figura 4.15: *The Papal Palace*, cuadro pintado por Paul Signac en 1900.

Hemos elegido estas imágenes, ya que tienen gran resolución (1696x992 la fotografía de Dublín y 2250x1789 la pintura) y así la nueva imagen que creemos no tendrá que perder calidad redimensionando al alza las imágenes iniciales.

Sin embargo, si intentamos ejecutar cualquier prueba realizada anteriormente, pero intentamos conseguir una imagen de de 1200 píxeles (image_size 1200), CUDA nos da un error out of memory. Lo que significa que la GPU no puede procesar tanta información.

Ahora realizamos la misma prueba, pero añadiendo los parámetros para hacer uso de dos GPUs, tal como hemos explicado anteriormente. Después de 15 minutos, obtenemos el resultado de la figura 4.16.

Figura 4.16: Imagen reestructurada a gran tamaño.

Esta nueva imagen cumple la premisa principal que es mantener el contenido aplicando el estilo. Además, se ha conseguido con una resolución final de 1200x700, algo que sólo podemos conseguir usando varias GPUs. En esta imagen podemos apreciar cómo ha recreado la imagen con los puntos coloreados característicos del estilo, lo que implica un nivel de reconocimiento de patrones inmenso.

CAPÍTULO 5
Conclusiones

5.1 Visión General

Con este trabajo, hemos conseguido hacer uso práctico de la tecnología Deep Learning desde prácticamente cero. Primeramente, hemos conocido qué es el Deep Learning y las técnicas básicas que se usan para conseguir Inteligencia Artificial. Luego, hemos instalado la herramienta Torch y hemos tomado contacto con ella y las redes neuronales. Con estos conocimientos hemos sido capaces de entrenar distintas redes y modelos para reconocimiento de imágenes. Todo esto siguiendo unas pautas para lograr la mayor eficiencia posible. Por último, hemos comprobado la utilidad estas redes y hemos aplicado nuestros conocimientos en Torch para hacer uso de un interesante proyecto de Deep Learning. Además, hemos tenido la oportunidad de inspeccionar cómo influye realizar los experimentos con múltiples GPUs y los resultados que ofrece.

Desde mi persona, a nivel laboral, he conseguido aprender sobre muchos campos variados. Por ejemplo, he conocido más profundamente qué es el Deep Learning y los principios fundamentales que sigue esta tecnología, he aprendido a gestionar distintos entrenamientos a redes neuronales, me he instruido en la herramienta Torch y he conseguido dominar los comandos de Linux. A todo esto, se añade el aprendizaje de realizar una memoria científica con Latex, crear las distintas gráficas y el uso de un servidor remoto. Muchas de estas son cosas que se han impartido en la universidad, pero yo he necesitado profundizar en ellas y gracias al trabajo dedicado a ello he podido alcanzar un buen nivel.

A nivel personal he aprendido a organizar las diferentes tareas de un trabajo de gran tamaño, tener las ideas claras antes de realizar nada y poder compaginar mi trabajo laboral con este proyecto sabiendo que es el cometido más importante de la carrera.

Los principales problemas han surgido con el hecho de trabajar remotamente en el nodo. Al principio me resultaba muy incómodo trabajar sin interfaz gráfica, todo desde terminal y además, en una máquina que no me pertenece. Muchas veces he necesitado la ayuda de mis tutores para poder instalar paquetes necesarios o para acceder a librerías de las que desconocía su directorio. A todo esto hay que sumar lo muchos errores que he obtenido a lo largo del proyecto.

Así pues, espero que este trabajo pueda ser de utilidad para cualquier persona interesada en el Deep Learning, al igual que lo ha sido para mí.

5.2 Relación del trabajo desarrollado con los estudios cursados

Este proyecto está basado en 2 principales cosas: Deep Learning y Torch. El Deep Learning es algo que he impartido en la universidad, pero sin entrar en detalle ni llevando a cabo ningún ejercicio práctico. Gracias a la rama de computación, tenía conocimientos sobre Inteligencia artificial y Machine Learning que me han servido para entender los procesos que se estaban llevando a cabo y cómo obtener los mejores resultados. Torch ha sido una herramienta totalmente nueva, sin embargo, a lo largo de la carrera he tenido que aprender muchas otras aplicaciones desde 0, lo cual me ha servido para poder familiarizarme a él fácilmente.

Este proyecto no ha requerido de ningún desarrollo software o programación de código, que es en lo que se ha basado principalmente la carrera. En este trabajo sólo he programado los scripts personales que he usado para poder trabajar con más comodidad con el servidor remoto. Esto se ha debido a que la idea fundamental del proyecto era crear una guía de iniciación al Deep Learning y Torch. Para trabajar detalladamente y programar con estas herramientas, es necesario un profundo conocimiento sobre redes neuronales y además, manejar el lenguaje de programación Lua, el cual no se ha impartido en mis estudios.

En cuanto a Linux, es un sistema operativo que he usado frecuentemente en el grado, así que me he sentido cómodo desde el principio y he conseguido aumentar en gran medida mi conocimiento sobre él.

En resumen, al finalizar este proyecto mi sensación es de haber aumentado significativamente mi entendimiento sobre el tema y las tecnologías que se usan ese ámbito, ya que comencé con las nociones básicas sobre Deep Learning y he necesitado dominar y trabajar con diversas herramientas para finalizar con un amplio campo de nuevos conocimientos.

CAPÍTULO 6
Futuros trabajos

Con este proyecto, hemos descubierto un amplio campo de posibilidades que pueden llevarse a cabo con Deep Learning. Sin embargo, no nos ha sido posible profundizar en muchos de ellos y realizar distintas tareas que pueden ser interesantes como:

- Crear una aplicación para comprobar la precisión de las redes de reconocimiento de imagen creadas donde le pudiésemos introducir una imagen a nuestra elección y se muestre la predicción por pantalla.
- Poder probar cada uno de los parámetros disponibles a la hora de entrenar las redes y ver si afectan al rendimiento.
- Experimentar las distintas posibilidades que ofrece Neural Style, como mezclar varios estilos en una misma imagen.
- Poder desarrollar nuestro propio modelo de Neural Style a partir de los ya creados para, por ejemplo, aplicar un desenfoque en el resultado para crear una imagen final más homogénea.

Todo esto son labores que pueden ser desarrolladas en un futuro, tanto por mí como por cualquier persona que esté interesada en el tema.

Bibliografía

[1] Wei Di Jianing Wei, Anurag Bhardwaj. *Deep Learning Essentials*. Packt Publishing, January 2018.

[2] Carlos A. Marmelada. *Sobre el origen de la inteligencia humana*. http://www.unav.edu/web/ciencia-razon-y-fe/sobre-el-origen-de-la-inteligencia-humana, 2013.

[3] Carme Torras. *Turing: el nacimiento del hombre (1912), la máquina (1936) y el test (1950)*. http://blogs.elpais.com/turing/2012/07/turing-el-nacimiento-del-hombre-1912-la-maquina-1936-y-el-test-1950.html, July 2012.

[4] Alan M. Turing. *Computing Machinery and Intelligence*, 1950.

[5] Alberto Iglesias Fraga. *La historia de la inteligencia artificial: desde los orígenes hasta hoy*. http://www.ticbeat.com/innovacion/la-historia-de-la-inteligencia-artificial-desde-los-origenes-hasta-hoy/, 2016.

[6] Josh Patterson Adam Gibson. *Deep Learning*. https://www.safaribooksonline.com/library/view/deep-learning/9781491924570/ch01.html.

[7] Guía de instalación de torch. http://torch.ch/docs/getting-started.html.

[8] Soumith Chintala. Proyecto de torch para entrenar un clasificador de objetos con múltiples gpus. https://github.com/soumith/imagenet-multigpu.torch.

[9] Justin Johnson. Proyecto que implementa el algoritmo de neural style en torch. https://github.com/jcjohnson/neural-style.

[10] Matthias Bethge Leon A. Gatys, Alexander S. Ecker. *A Neural Algorithm of Artistic Style*. https://arxiv.org/abs/1508.06576, September 2015.