**Ejercicio 1 (Notebook: 1.setup\_p37.ipynb)**

En este notebook se realizan una serie de operaciones que muestran al usuario el HW con el que se encuentra uno cuando accede a la plataforma de Google Colab. También enseña como desde collab se pueden hacer llamadas al SO mediante Bash, utilizando distintos comandos como los siguientes:

Imagen que contiene Patrón de fondo

Descripción generada automáticamente

Esta información mostrará tanto los directorios disponibles en el entorno como las especificaciones de la “máquina” que nos asigna Collab cuando nos conectamos a la plataforma:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente con confianza baja

Para el entrenamiento acelerado de algoritmos de aprendizaje profundo es útil tener GPUs. Estos aceleradores hardware también están disponibles en el entorno de ejecución y pueden ser verificados mediante el siguiente comando:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

También se realizan una serie de instalaciones de librerías que serán necesarias para poder entrenar a nuestros algoritmos de aprendizaje profundo. Estás librerías serían PyTorch y Pillow:

Texto

Descripción generada automáticamente

Por otro lado, también se interactúa con la librería de google, para el manejo de directorios de Drive:

Texto

Descripción generada automáticamente

Por último, se realizan una serie de ejecuciones a sentencias en las que se interactúa con el sistema operativo. Estas son, por ejemplo:

Movimiento de directorios:

Imagen que contiene Texto

Descripción generada automáticamente

Generación de scripts “on the run”:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Ejecución de scripts:

Interfaz de usuario gráfica

Descripción generada automáticamente

**Ejercicio 2 (Notebook: 2.dataset\_loading\_p37.ipynb)**

En este dataset se cargan los datos que provienen del dataset CIFAR10. Este dataset tiene las siguientes características:

* 60,000 imágenes a color RGB (con resolución 32x32)
* 10 clases (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)
* Datos están divididos entre entrenamiento/train (50K) y test/validation (10K).

Se convierte el dataset a un formato PyTorch. Este formato tiene dos parámetros clave:

1. **batch\_size**: tamaño del grupo de datos que se generan para el entrenamiento y la validación
2. **shuffle**: FLAG que indica si el agrupamiento es aleatorio o secuencial.

Posteriormente, se definen unas funciones que permiten la visualización de datos:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

También se presenta una funcionalidad para el tratamiento de datasets no genéricos. En este caso descargaremos el el dataset [scene15](http://www-vpu.eps.uam.es/~jcs/pub/Scene15.zip).

Se volverán a hacer las mismas operaciones, carga en memoria de los datos, transformación al formato PyTorch y la muestra de las imagenes.

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

Por último, se añade una parte bonus, donde se explica la transformación de los datos, explicando la librería transform de PyTorch. Las trasnformaciones entre otras serán:

* Resize:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* Normalización:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* Crop(s):

Texto, Escala de tiempo

Descripción generada automáticamente con confianza media

Posteriormente, se explica como se pueden aplicar distintas transformaciones sucesivas a los datos. En este detalle, explica buenas prácticas a la hora de generar datos (imágenes) como hacer que todas tengan las mismas dimenesiones (cosas que se pueden alcanzar con un resize o con un crop) y normalizarlas para un entrenamiento óptimo.

**Ejercicio 3 (Notebook: 3.** **network\_p37.ipynb)**

En este ejercicio se estudiará la topología de una red neuronal convolucional (CNN), de la mano de PyTorch como framework para poder crearla, entrenarla y obtener resultados de ella. Se explican los pasos y la definición de una CNN:

* Definir la estructura de la red, que tendrá algunos parámetros entrenables (i.e. weights\*).
* Definir la secuencia de procesado de los datos para obtener una salida (i.e. \*forward pass\*).
* Calcular la función de pérdidas que indique la precisión de la salida con respecto a nuestras anotaciones (i.e. \*loss function\*).
* Calcular los gradientes de retropropagación ejecutando en modo inverso la red (i.e. \*backward propagation\*).
* Actualización de los parámetros de la red acorde a los gradientes anteriores.

Posteriormente, en este notebook se creará una red neuronal convolucional. Como tal, lo primero que se deberá de saber es que se ha definido una clase “Net” la cual actúa como “wrapper” de la funcionalidad que alberga PyTorch para poder crear redes neuronales. Este wrapper estará basado en el paquete **torch.nn**.

El paquete **torch.nn** tiene una serie de métodos para definir las capas de la red neuronal, los cuales son explicados en el notebook.

En primer lugar, tenemos la clase que define las capas convolucionales 2D. Esto se hace mediante la función Conv2d, que tiene como argumentos principales:

* **in\_channels**: valor para la tercera dimension del tensor de entrada (e.g. 3 para imágenes RGB, 1 para imágenes en Gris).
* **out\_channels**: número de mapas de salida (i.e. número de convoluciones o \*kernels\* que aplicamos sobre los datos de entrada).
* **kernel\_size**: tamaño del \*kernel\* aplicado (tamaño x tamaño)
* **stride:** desplazamiento de la aplicación del operador de convolución
* **padding**: tipo de padding aplicado

Posteriormente explica otros tipos de redes, como las Linear, las cuales cabe destacar su propiedad fully connected, que hace que todas las neuronas estén conectadas. Esto puede afectar a la convolución y dar pie a redes que produzcan overfitting.

También se tienen etapas, como la explicada en el notebook, MaxPooling2D, que hace una reducción en la dimensionalidad de los datos, y que tiene los siguientes argumentos de interés:

* **kernel\_size**: tamaño del \*kernel\* aplicado (tamaño x tamaño).
* **stride**: desplazamiento de la aplicación del operador de convolución.
* **padding**: tipo de padding aplicado.

Posteriormente se realiza una definición de la red mediante la clase “Net”, el wrapper que se comentó al principio de esta sección.

Una vez definida la red, se explica la primera parte del funcionamiento de la red. En este caso, la idea es sencilla, se definen una serie de valores aleatorios con dimensión igual a la entrada de la red. Estos valores se definirán como los valores de entrada de la red. La red producirá una salida, la cual podremos ver como un vector de valores, el cual, dependiendo del problema, se habrá de interpretar.

Para que la red aprenda, tal y como pasa con las especies de la naturaleza, se debe “recompensar” o “castigar” las acciones que esta desempeñe. Si las acciones son beneficiosas (una correcta clasificación de una imagen) se ha de notificar a la red que va por el camino correcto. Para eso se tienen funciones (de pérdida) que le dicen al algoritmo como de bien (o de mal) va encaminado.

Texto

Descripción generada automáticamente

Una vez que hemos “premiado” o “castigado” a la red, los pesos de la misma (parámetros que la red “aprende”) deberán de ser actualizados. Para eso se usa el algoritmo de backpropagation:

Texto

Descripción generada automáticamente

El cual computará el descenso por gradiente, dando a cada neurona un valor al que ajustarse.

Una vez calculado el valor de ajuste, se tendrá en cuenta la proporción de aprendizaje de la red. Este valor dice como de rápido bajamos por el descenso de gradiente, es decir, la velocidad con la que tratamos de acercarnos al mínimo local mas cercano que obtenemos de la función de activación:

Texto

Descripción generada automáticamente

Y con eso, se ajusta, para una época, los pesos de una red neuronal.

**Ejercicio 4**

Tal y como se indica en la transparencia 21 de la práctica, los resultados que se tienen ejecutando el cuaderno **4.training\_baseline\_p37\_cpu.ipynb** tienen una precisión de entrenamiento del 100%, pero un 35% de validación/test. Esto nos hace pensar que la red no generaliza correctamente cuando datos nuevos entran en la entrada de la red.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Para ello se proponen unas sugerencias a modo de mejora:

1. Estudio del efecto del parámetro batchsize en 50 épocas.
2. Estudio del efecto del parámetro Resizing\_factor.
3. Explore el uso de normalización de imágenes como última operación de una transformación compuesta.
4. Explore y compare distintas estrategias de optimización.
5. Explore y compare una aplicación progresiva de un factor learning rate mediante el elemento lr\_scheduler.
6. Aplique técnicas de Data Augmentation para mejorar la calidad de los datos de entrenamiento.
7. Aplique fine-tuning al problema para adaptar una red ‘relativamente’ compleja al problema de esta práctica.
8. Modificar la arquitectura original de mylenet.py añadiendo un número mayor de capas convoluciones con un mayor campo de visión