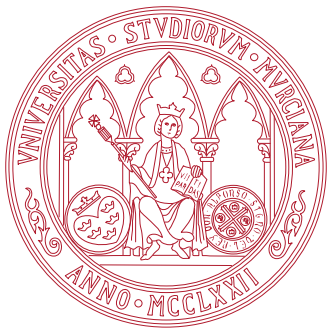

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

Prácticas de

Arquitecturas Multimedia y de Propósito Específico

4º de Grado en Ingeniería Informática

Boletín de prácticas 1 – Construcción en PyTorch de una DNN para reconocimiento de dígitos

Curso 2022/2023

Índice

1. Introducción

El ser capaz de reconocer un dígito capturado en una imagen es una tarea trivial para el ser humano. Todos nosotros somos capaces de interpretar correctamente un "5" cuando lo vemos escrito, incluso con distintas grafías.

Sin embargo, a pesar de lo natural que nos pueda parecer, delegar esta tarea a un sistema de cómputo ya no es tan sencillo. Los computadores codifican las imágenes como un array de valores denominados píxeles y el averiguar el dígito que codifican esos píxeles requiere de técnicas complejas de inteligencia artificial que ayuden al reconocimiento de diversos patrones en ese mar de píxeles.

En esta práctica construiremos, analizaremos y evaluaremos un sistema de reconocimiento de dígitos basado en deep learning. Para ello, se utilizará PyTorch, un framework de DNNs desarrollado por Facebook que ayudará a simplificar el desarrollo.

1.1. Objetivos

Al terminar el boletín el alumno debe ser capaz de:

- Conocer los conceptos básicos sobre las redes neuronales, sus capacidades, limitaciones y funcionamiento.
- Entender las particularidades de los procesos de entrenamiento e inferencia de una red neuronal.
- Ser capaz de codificar un modelo de red neuronal sencillo en PyTorch, entrenarlo y usarlo posteriormente en procedimientos de inferencia.
- Evaluar y analizar tiempos de ejecución en Python.

1.2. Herramientas utilizadas

Las herramientas que usaremos en este boletín son:

- `Anaconda` : distribución de los lenguajes de programación Python y R (usaremos el primero) para, entre otras cosas, simplificar la gestión e implementación de paquetes de machine learning. Además, es capaz de establecer diferentes entornos para poder utilizar distintas versiones de Python y de paquetes. Se puede utilizar también la versión reducida, denominada Miniconda.
- Intérprete de `Python` v.3.7.3
- `PyTorch` versión AMPE : versión personalizada de PyTorch para la asignatura que usaremos para en esta prácticas, además de en las prácticas 2 y 4. Está basada en PyTorch 1.7.0.

En las páginas de manual de cada una de estas herramientas encontrarás información detallada sobre cómo usarlas.

2. Redes Neuronales Profundas

Aunque la idea de red neuronal emergió en los años 50, no ha sido sino hasta recientemente cuando se ha vuelto fundamental. Gracias a la gran cantidad de datos que se generan hoy en día en Internet o en despliegues de sensores procedentes del IoT, y a las grandes capacidades de cómputo que proporciona el hardware actual, es posible entrenar redes neuronales profundas y complejas con mínimo error de precisión, obteniendo resultados nunca antes vistos en diversos dominios de aplicación de la Inteligencia Artificial.

De esta forma, las Redes Neuronales Profundas (del inglés *Deep Neural Networks* o DNNs) constituyen hoy en día la base de una gran cantidad de aplicaciones. El reconocimiento de imágenes, la detección de enfermedades, los condicción autónoma, la traducción de texto, el reconocimiento del lenguaje natural, etc, son solo algunos ejemplos de aplicaciones que tienen como base esta tecnología.

En esta sección, veremos los conceptos básicos sobre el funcionamiento de una red neuronal profunda.

2.1. Neurona artificial

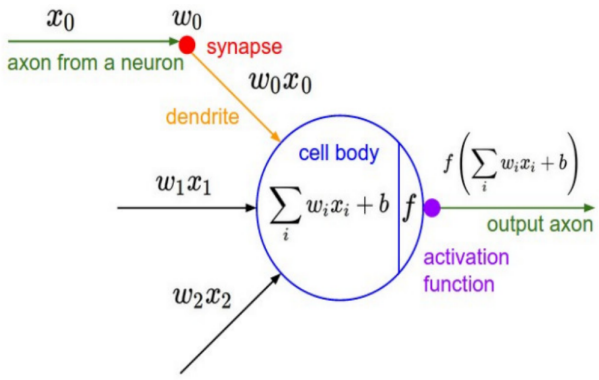

Figura 1: Ejemplo de neurona artificial.

Una Red Neuronal (*Neural Network* o NN) se construye a partir de varias de neuronas artificiales interconectadas entre sí. Como vemos en la Figura 1, de forma similar al modelo biológico, la neurona artificial representa la unidad básica de funcionamiento de una NN. En concreto, cada neurona artificial realiza un cómputo bastante simple, que consiste en multiplicar cada valor de entrada de otra neurona por un peso que la conexión correspondiente tiene asociado, y acumular el resultado de todas las multiplicaciones (son, por tanto, simples operaciones de multiplicación y acumulación o MAC–*Multiply-ACcumulate*). A la suma acumulada, se le aplica una función no lineal de activación, como por ejemplo sigmoide o hiperbólica, y su resultado se envía a las neuronas con las que estuviese conectada.

El aprendizaje más común de una red neuronal consiste en ajustar los pesos de las conexiones con el objetivo de que el valor obtenido sea el adecuado para cada caso concreto. Así, mediante el uso de una gran cantidad de ejemplos de entrada, es posible hacer aprender a una red neuronal, en un proceso denominado **entrenamiento**, a través del cual se ajustan los valores de los pesos de la red de acuerdo a la tarea para la que va a ser utilizada. Posteriormente, la red ya entrenada se emplea para realizar predicciones a partir de una entrada de datos no vista anteriormente, en un procedimiento llamado **inferencia**.

2.2. Estructura básica de una red neuronal

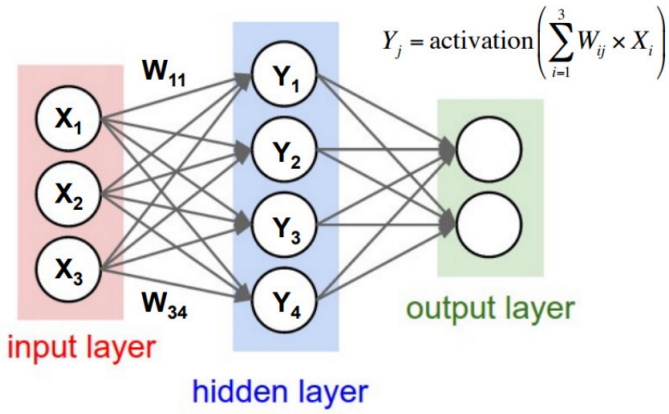

Figura 2: Ejemplo de red neuronal con varias capas.

La Figura 2 muestra el funcionamiento básico de una NN. Como vemos, las neuronas se organizan en capas y las conexiones entre las mismas se hacen desde una capa a la siguiente. El término red neuronal profunda (*Deep Neural Network* o DNN) se suele utilizar para referirse a una NN con más de 5 capas. Concretamente, existen 3 tipos diferentes de capas:

- **Capa de entrada**: Es la capa a través de la que se reciben los datos con los que entrenar o inferir, como por ejemplo los píxeles de una imagen. Las conexiones de salida de esta capa van hacia la siguiente capa de la red.
- **Capas ocultas**: Las capas ocultas son las capas intermedias de una NN. Su número puede variar drásticamente de una red a otra, ya que depende del diseño de la propia red. Cada capa oculta recibe como datos de entrada las salidas de la anterior capa (otra capa intermedia o la de entrada), y envía los resultados a la siguiente capa de la red (otra capa intermedia o la capa de salida).
- **Capa de salida**: Es la última capa de la red neuronal. Recibe los datos de la anterior capa (ya sea la de entrada u otra capa intermedia) y envía los resultados finales, que pueden ser por ejemplo, un valor de probabilidad, o cualquier otro valor escogido por el diseñador de la red.

2.3. Entrenamiento e inferencia

Para que una red neuronal obtenga el resultado deseado a la hora de realizar una cierta tarea, es necesario utilizar la red en dos fases bien diferenciadas, descritas a continuación:

- **Entrenamiento**: Como hemos visto en la sección 2.1, el entrenamiento de una NN consiste en ajustar los pesos de las conexiones sinápticas de la red, de forma que el resultado final de la inferencia de toda la red sea el esperado. Este tipo de entrenamiento puede hacerse de varias formas, pero la técnica más utilizada actualmente es la denominada **aprendizaje supervisado o etiquetado**, en la cual cada ejemplo dentro del conjunto de ejemplos de entrada tiene asignada una etiqueta con el resultado correcto. La idea es realizar la inferencia con cada ejemplo y comparar el resultado real obtenido por la red con el correcto. Si los resultados difieren, se modifican los pesos de la red en consecuencia, de manera que cuando se vuelva a realizar la inferencia, el resultado obtenido por la red sea el correcto. El proceso de modificación de pesos más utilizado se conoce como **backpropagation** que consiste en modificar cada peso proporcionalmente a la variación que tiene el resultado final al modificar el valor de dicho peso.
- **Inferencia**: Una vez que la red ha sido entrenada, está lista para entrar en producción, es decir para ser utilizada para el propósito para el que se diseñó. Así, para una nueva entrada no vista durante el entrenamiento, la red neuronal va procesándola, de forma que, capa a capa, las neuronas van computando su resultado y enviándolo hacia la siguiente capa hasta obtener el resultado final. A este proceso se le denomina inferencia.

3. Frameworks para el desarrollo de DNNs: PyTorch

Uno de los grandes factores que ha dado lugar al tremendo crecimiento en el uso de las técnicas de deep learning en los últimos años, ha sido el desarrollo de diferentes frameworks de alto nivel para facilitar la especificación de modelos de DNNs. Hay varios ejemplos de estos frameworks, como por ejemplo, Caffe, TensorFlow o PyTorch.

El empleo de estos frameworks proporciona varios beneficios fundamentales:

- **Facilidad de desarrollo**: Proporcionan mecanismos de alto nivel que permiten encapsular las operaciones llevadas a cabo durante el entrenamiento y la inferencia de una DNN. De esta forma, el usuario solo tiene que seleccionar las operaciones a realizar, sin tener que programarlas, proporcionándole una interfaz sencilla y versátil.
- **Abstracción**: Esta encapsulación permite abstraer al programador de la interfaz hardware. De esta forma, el programador se olvida de tratar de explotar el hardware al máximo, consumiendo, por ejemplo, su tiempo en paralelizar el código según la arquitectura concreta.
- **Portabilidad y reuso**: Gracias también a la encapsulación de operaciones, el usuario podrá ejecutar la DNN en cualquier entorno donde el framework se encuentre disponible, sin tener que preocuparse sobre detalles de bajo nivel.

Actualmente existen multitud de frameworks para el desarrollo de DNNs, cada uno con un método particular de codificar las operaciones llevadas a cabo durante la ejecución de DNNs.

Uno de los frameworks más populares hoy en día, y en el que nos centraremos durante el desarrollo de las prácticas de la asignatura es **PyTorch**, un software de código abierto basado en Python desarrollado por Facebook y liberado bajo la licencia modificada de BSD.

Durante las próximas secciones de esta práctica aprenderemos a utilizar PyTorch para desplegar una NN sencilla, que será capaz de reconocer dígitos en una imagen de entrada.

4. Instalación del entorno de desarrollo

En esta sección veremos cómo instalar una versión personalizada de PyTorch, además de todo el entorno necesario para la correcta realización de las prácticas de esta asignatura.

4.1. Requerimientos

Puesto que vamos a instalar PyTorch desde su propio código se va a requerir una versión previa de Python (usaremos la 3.7.3) y un compilador de C++14 o superior (en este caso, cualquier versión reciente de GCC nos valdrá).

Precisamente Anaconda nos va a permitir crear un entorno (*environment*) con la versión que deseemos de Python. Vamos a usar la versión 2022.05, la más reciente en el momento de la escritura de este documento. Podemos descargarla a través del siguiente enlace:https://repo.anaconda.com/archive/Anaconda3-2022.05-Linux-x86_64.sh. Seguidamente ejecutaremos lo siguiente:

```
$ sh Anaconda3-2022.05-Linux-x86_64.sh
```

Nos pedirá que aceptemos los términos de la licencia y, al final, nos dirá si queremos que cada vez que abramos un terminal se inicie conda. Le diremos que «no», opción por defecto, ya que iniciaremos conda de forma manual (lo único que hace el instalador si le decimos que «sí» es modificar el fichero `.bashrc` para incluir la orden de puesta en marcha).

Una vez instalada Anaconda, creamos un nuevo entorno e instalamos en él todas las dependencias previas a la instalación de PyTorch. Podemos hacer esto con las siguientes órdenes:

```
$ eval "$(~/Anaconda3/bin/conda shell.bash hook)" # Ponemos en marcha Conda
$ conda create -n ampe python=3.7.3  # Creamos un nuevo entorno con la versión 3.7.3 de Python
$ conda install numpy ninja pyyaml mkl mkl-include setuptools
$ conda install cmake cffi typing_extensions future six requests dataclasses
$ conda deactivate # Salimos del entorno
```

4.2. Instalación de PyTorch

La descarga e instalación de PyTorch puede ser realizada de forma rápida y sencilla con la orden *conda install pytorch*. Sin embargo, esto descargaría e instalaría la versión más reciente disponible en los repositorios oficiales. Para las prácticas, sin embargo, vamos a instalar una versión adaptada del framework, por lo que deberemos de hacerlo directamente desde el código fuente localizado en un repositorio github que hemos preparado, accesible a través del siguiente enlace: https://github.com/meacacio/ampe_environment.git.

A continuación se detallan las órdenes a ejecutar:

```
$ git clone https://github.com/meacacio/ampe_environment.git
$ cd ampe_environment
$ cd pytorch-frontend
$ source ~/Anaconda3/bin/activate ampe # Entramos al entorno ampe creado anteriormente
$ export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(dirname $(which conda))/../"}
$ python setup.py install
```

Para comprobar que la instalación se ha realizado con éxito, bastará con ver si podemos importar la librería *torch* en Python. Para ello, podemos realizar el siguiente procedimiento (una vez está activo el entorno `ampe`);

```
$ python
(ampe) $ python
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> exit()
```

Si el módulo torch puede ser importado con éxito podemos proseguir con la realización de la práctica.

El siguiente paso sería instalar `torchvision`, pero, de nuevo, tendremos que hacerlo desde el propio código fuente, concretamente a partir de la versión incluida en el entorno de la práctica. Para ello, nos iremos al directorio `vision`, dentro de `ampe_environment`, y ejecutaremos:

```
$ cd ../vision
$ python setup.py install
```

Finalmente, instalaremos el paquete `matplotlib` mediante la ejecución de:

```
$ conda install matplotlib
```

4.3. Algunos problemas conocidos

El proceso de instalación anterior de nuestra versión modificada de PyTorch ha sido validado en una distribución `Ubuntu 18.04`. En otras versiones de Ubuntu, o de Linux, sin embargo, podrían requerirse ciertos ajustes para su compilación. Por ejemplo, durante la instalación en `Fedora 34` se observaron problemas de compilación con gcc 11 que se resolvieron añadiendo la línea `#include <limits>` en el fichero `ampe_environment/pytorch-frontend/third_party/benchmark/src/benchmark_register.h` y la línea `#include <cstddef>` en el fichero `ampe_environment/pytorch-frontend/c10/util/hash.h`

En otras distribuciones se han observado también problemas con alguna librería de CUDA y otra funcionalidad que no usaremos. Si este fuese el caso, puedes probar a hacer la instalación de la versión de PyTorch de la asignatura, de la siguiente forma:

```
$ USE_CUDA=0 USE_OPENMP=0 USE_CUDNN=0 BUILD_TEST=0 python setup.py install
```

5. Construcción de un reconocedor de dígitos basado en DNNs paso a paso.

Tras la correcta instalación y puesta en marcha del framework para DNNs PyTorch utilizando los pasos descritos en la sección 4, ahora vamos a ver cómo es posible utilizar dicho framework para el entrenamiento y despliegue de una DNN capaz de realizar la tarea de reconocimiento de dígitos en una imagen.

5.1. Paso 1: Preparación del conjunto de datos

Tal y como se ha visto anteriormente, para enseñar a una DNN a realizar una cierta tarea es necesario entrenarla utilizando ejemplos ya etiquetados. Para ello, se utiliza lo que se denomina un **conjunto de datos** o dataset. Este conjunto de datos se suele dividir en 2 partes: un **subconjunto de entrenamiento** utilizado para el entrenamiento de la red y otro **subconjunto de validación** para comprobar su precisión en un momento determinado.

Por esta razón, el paso previo a cualquier entrenamiento de una DNN es la definición y preparación del conjunto de datos (**dataset**) a utilizar en base a la tarea que se quiera realizar. En concreto, para el reconocimiento de dígitos existe un conjunto de datos bien conocido denominado **MNIST**, el cual está formado por una colección de 70.000 imágenes de tamaño 28×28 con dígitos escritos a mano repartidos en 60.000 y 10.000 para los subconjuntos de entrenamiento y validación, respectivamente. Durante esta práctica entrenaremos una DNN utilizando este conjunto de datos para posteriormente utilizarla con ejemplos reales.

Aunque el conjunto de datos está directamente disponible a través de su página web oficial (http://yann.lecun.com/exdb/mnist/), PyTorch ya proporciona una librería sencilla para la descarga y utilización de MNIST utilizando únicamente unas pocas líneas de código.

En primer lugar, vamos a importar todos los paquetes necesarios para la realización de esta práctica:

```
>>> import numpy as np
>>> import torch
>>> import torchvision
>>> import matplotlib.pyplot as plt
>>> from time import time
>>> from torchvision import datasets, transforms
>>> from torch import nn, optim
```

Antes de descargar el conjunto de datos, tenemos que definir las transformaciones que vamos a hacer sobre todas las imágenes de este conjunto de datos. Esto es lo que se conoce como el preprocesamiento de la entrada y es necesario para que la DNN sepa exactamente en qué formato, tamaño y con qué propiedades va a recibir los datos de entrada. En PyTorch podemos hacer esto usando el paquete *torchvision.transforms*:

```
>>> transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,)),])
```

Como vemos, el método *Compose* recibe una lista de transformaciones a aglutinar. En este caso, utilizaremos dos:

1. *transforms.ToTensor()*: Convierte las imágenes en un formato entendible por el sistema. En concreto, transforma cada uno de los píxeles escalando cada valor al rango entre 0 y 1 para finalmente, guardar todos estos píxeles en un objeto tipo *Tensor*, reconocido y utilizado por PyTorch.

2. *transforms.Normalize()*: Normaliza el tensor con la media y la desviación típica indicadas como parámetros (0,5).

Una vez que tenemos un objeto definido que almacena todas las transformaciones que vamos a realizar sobre cada una de las imágenes de entrada, vamos a descargar el conjunto de datos MNIST, barajarlo, y transformar todas sus imágenes utilizando dicho objeto. Para hacer esto, vamos a utilizar el paquete *torch.utils.data.DataLoader*, el cual combina distintos conjunto de datos y proporciona iteradores para recorrerlos de forma sencilla.

```
>>> trainset = datasets.MNIST('PATH_TO_STORE_TRAINSET', download=True, train=True,
    transform=transform)
>>> valset = datasets.MNIST('PATH_TO_STORE_TESTSET', download=True, train=False,
    transform=transform)
>>> trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
>>> valloader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True)
```

El código anterior crea 2 objetos **trainset** y **valset** que aglutinan los subconjuntos de datos de entrenamiento y validación de MNIST. Además, creamos dos iteradores asociados a dichos objetos **trainloader** y **valloader**, los cuales nos permitirán leer los datos de forma sencilla. Es importante apreciar que el **batch_size** utilizado en ambos iteradores es 64. Esto es el número de imágenes que serán leídas por cada iteración que se invoque a los objetos.

Para comprobar que el conjunto de datos puede ser leído correctamente, vamos a realizar un pequeño análisis sobre el mismo. Vamos a comprobar la forma (dimensiones) de los tensores donde se almacenan las imágenes y sus etiquetas asociadas:

```
>>> dataiter = iter(trainloader)
>>> images, labels = dataiter.next()

>>> print(images.shape)
>>> print(labels.shape)
```

Como comprobarás, la forma de las imágenes debería de ser *torch.Size([64,1,28,28])*, lo cual indica que hay 64 imágenes en cada *batch*, cada una de ellas con un canal (son imágenes en blanco y negro) de

tamaño 28×28. Del mismo modo, las etiquetas se almacenan en un Tensor con forma *torch.Size([64])*, donde cada posición del tensor corresponderá con el número asociado a cada imagen.

Además, podemos visualizar cualquier imagen del conjunto de entrenamiento, como por ejemplo la primera[1]:

```
>>> plt.imshow(images[0].numpy().squeeze(), cmap='plasma');
>>> plt.show()
```

Utilizando este mismo mecanismo puedes investigar y visualizar todas las imágenes que desees del conjunto de datos.

5.2. Paso 2: Construcción de la DNN para el reconocimiento de dígitos

La definición de una DNN es una tarea compleja que llevan a cabo expertos en inteligencia artificial. En esta práctica, utilizaremos la DNN mostrada en la Figura 3, la cual ya se ha comprobado que funciona con alta precisión para la tarea de reconocimiento de dígitos.

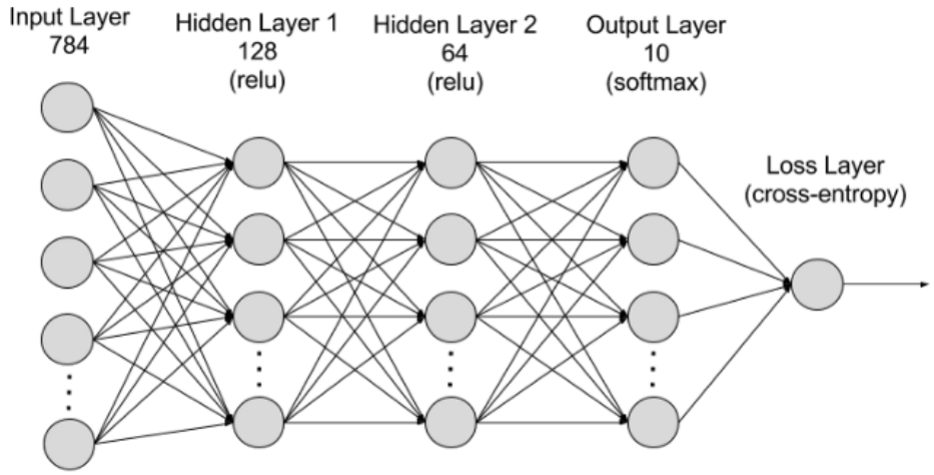

Figura 3: DNN que utilizaremos para el reconocimiento de dígitos en esta práctica.

Como podemos apreciar, esta DNN está formada por 3 capas completamente conectadas (Fully-Connected o Linear). La capa de entrada está formada por 784 neuronas de entrada, lo cual corresponde exactamente con el tamaño de las imágenes a utilizar (28×28). Esto obviamente no es casualidad. Cada neurona se corresponde con cada uno de los píxeles de la imagen, y estos son combinados durante esta primera capa para formar las 128 neuronas de salida, que constituyen la entrada a la siguiente capa. Esta, realiza la misma operación, combinando estas 128 entradas en 64 neuronas de salida. Del mismo modo, la tercera y última capa completamente conectada combina estas 64 neuronas de entrada en 10 neuronas de salida, las cuales son ciertamente la salida de la red, ya que cada una de estas neuronas corresponde con la probabilidad de que un un cierto dígito sea el de la imagen de entrada. Finalmente, la última capa de salida de la red es una función **LogSoftMax** que simplemente coge estos 10 valores de probabilidad y los normaliza para que se parezcan realmente a una distribución de probabilidad respecto a los 10 posibles valores de salida. El valor de probabilidad más alto será por tanto, el valor predicho por nuestra red.

Un aspecto fundamental a tener en cuenta, es que las primeras capas de la DNN van acompañadas de la función de activación **ReLU (Rectified Linear Unit)**. Esta función sirve para añadir no linearidad a la red y es esencial para que se obtenga un buen resultado.

Definir esta DNN utilizando PyTorch es muy sencillo, ya que todas las operaciones necesarias se encuentran en el paquete *torch.nn*. Para ello, bastará con crear una clase que herede de *torch.Module* y seleccionar las capas a ejecutar en su constructor, para posteriormente utilizarlas en la redefinición de su método *forward*. Podemos ver un ejemplo a continuación:

[1]Para poder apreciar más fácilmente la intensidad de color de cada píxel, se ha utilizado la escala de colores plasma que proporciona la librería `matplotlib`.

```
>>> import torch.nn as nn
>>> import torch.nn.functional as F

>>> class My_DNN(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.fc1 = nn.Linear(784, 128)
...         self.fc2 = nn.Linear(128, 64)
...         self.fc3 = nn.Linear(64,10)
...     def forward(self, x):
...         x = self.fc1(x)
...         x = F.relu(x)
...         x = self.fc2(x)
...         x = F.relu(x)
...         x = self.fc3(x)
...         x = F.log_softmax(x, dim=1)
...         return x
```

Como vemos, las operaciones *nn.Linear* corresponden con las capas fully-connected descritas anteriormente y reciben 2 parámetros de entrada: el número de neuronas de entrada y el número de neuronas de salida. La operación *F.relu* corresponde con la función de activación ReLU que sirve para añadir no linearidad a la red. Finalmente, la operación *F.log_softmax* obtiene el vector de probabilidades final. Una vez el método *forward* de la clase es invocado, las operaciones se ejecutan en el orden correspondiente utilizando como entrada el tensor resultante de la operación anterior. Finalmente, el resultado final es devuelto por el método y será el resultado inferido por nuestra red.

Ejecutar nuestra DNN es tan sencillo como declarar un objeto de tipo nuestra DNN, seleccionar una imagen de entrada y pasársela como parámetro a la DNN. Lo podemos ver a continuación:

```
>>> model = My_DNN()
>>> print(model)
>>> images, labels = next(iter(valloader))
>>>
>>> img = images[0].view(1, 784)
>>> with torch.no_grad():
...     logps = model(img)
>>>
>>> ps = torch.exp(logps)
>>> probab = list(ps.numpy()[0])
>>> print("Predicted Digit =", probab.index(max(probab)))
```

De forma gráfica, podemos ver el dígito de la imagen y la distribución de probabilidades:

```
>>> ps = ps.data.numpy().squeeze()
>>> fig, (ax1, ax2) = plt.subplots(figsize=(6, 9), ncols=2)
>>> plt.tight_layout()
>>>
>>> ax1.imshow(img.resize_(1, 28, 28).numpy().squeeze(), cmap='plasma')
>>> ax1.axis('off')
>>>
>>> ax2.barh(np.arange(len(ps)), ps)
>>> ax2.set_aspect(0.1)
>>> ax2.set_yticks(np.arange(len(ps)))
>>> ax2.set_yticklabels(np.arange(len(ps)))
>>> ax2.set_title('Class Probability')
>>> ax2.set_xlim(0, 1.1)
>>>
>>> plt.show()
```

Tal y como cabría esperar, la NN no reporta un resultado pues aún no ha sido instruida en la labor de reconocer dígitos. Recuerda que esto lo lograremos a través del procedimiento de entrenamiento. En los pasos posteriores vamos a proceder a entrenar la NN para *enseñarla* a que pueda detectar los dígitos de forma correcta.

5.3. Paso 3: Definición de la función de pérdida

Una DNN aprende iterando múltiples veces sobre ejemplos de datos. Cada uno de los ejemplos es inferido por la DNN y su resultado es comparado con la etiqueta asociada al ejemplo (aprendizaje supervisado). Después, el error producido por la red es mitigado modificando los pesos de la DNN de forma proporcional a dicho error. Necesitamos por tanto, una función que nos devuelva la diferencia producida entre el resultado inferido por la NN y el valor real de la etiqueta. Esta función, se denomina **función de pérdida** o *loss* y varía según la tarea a realizar y el formato en el que se encuentren los datos de salida.

Generalmente, la función de pérdida más utilizada en los problemas de clasificación con C clases (en este caso 10 clases) es la función **negative log-likelihood loss**. En esta práctica utilizaremos esta función de pérdida que se encuentra ya implementada en PyTorch y lista para ser usada. A continuación, podemos ver un ejemplo:

```
>>> criterion = nn.NLLLoss()
>>> images, labels = next(iter(trainloader))
>>> images = images.view(images.shape[0], -1)
>>>
>>> logps = model(images) #log probabilities
>>> loss = criterion(logps, labels) #calculate the NLL loss
```

El ajuste de pesos de la red es realizado en función de esta diferencia producida, utilizando el método **backward()** de la función de pérdida:

```
>>> print('Before backward pass: \n', model.fc1.weight.grad)
>>> loss.backward()
>>> print('After backward pass: \n', model.fc1.weight.grad)
```

Como vemos, antes de ejecutar el método **backward()** los pesos de la DNN están inicializados a **none**. Tras la llamada al método, estos pesos son actualizados en función del valor de ese error producido. Para ello, se utiliza generalmente un algoritmo conocido como **backpropagation**.

5.4. Paso 4: Entrenamiento de la DNN

Una vez que ya tenemos nuestro subconjunto de entrenamiento preparado y hemos definido nuestra DNN y nuestra función de pérdida, el proceso de entrenamiento consiste en iterar sobre todos nuestros datos, calcular el error producido por la DNN para cada uno de ellos, y actualizar los pesos mediante el método **backward()** mencionado anteriormente.

Para ello, haremos uso del paquete *torch.optim*, el cual es un módulo proporcionado por PyTorch para realizar todo el proceso de forma automática. Así, por cada **época** (cada pasada que el procedimiento de entrenamiento realiza sobre el conjunto de entrenamiento completo), veremos un decremento gradual del error producido por la red. Todo este proceso se muestra a continuación:

```
>>> optimizer = optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
>>> time0 = time()
>>> epochs = 15
>>> for e in range(epochs):
...     running_loss = 0
...     for images, labels in trainloader:
...         # Flatten MNIST images into a 784 long vector
...         images = images.view(images.shape[0], -1)
...
...         # Training pass
...         optimizer.zero_grad()
...
...         output = model(images)
...         loss = criterion(output, labels)
...
...         #This is where the model learns by backpropagating
...         loss.backward()
...
...         #And optimizes its weights here
...         optimizer.step()
```

```
...
...         running_loss += loss.item()
...     else:
...         print("Epoch {} - Training loss: {}".format(e, running_loss/len(trainloader)))
...
>>> print("\nTraining Time (in minutes) =",(time()-time0)/60)
```

5.5. Paso 5: Comprobación y evaluación de la precisión de la red

Una vez que la red está entrenada, y por lo tanto, los pesos están ajustados, el último paso es evaluar que la DNN es capaz de realizar su tarea de forma precisa. Para ello, en primer lugar, vamos a inferir una imagen de ejemplo y vamos a comprobar si el resultado es correcto:

```
>>> images, labels = next(iter(valloader))
>>> img = images[0].view(1, 784)
>>> with torch.no_grad():
...     logps = model(img)
...
>>> ps = torch.exp(logps)
>>> probab = list(ps.numpy()[0])
>>> print("Predicted Digit =", probab.index(max(probab)))
>>> print(labels[0].item())
```

Como vemos, para la primera imagen de nuestro conjunto de datos la DNN ha predicho el dígito de forma correcta. Aun así, ¿funcionará para todos los casos? ¿cuál es la precisión de la red?

Para comprobar esto tendremos que utilizar el conjunto de datos de validación. Vamos a inferir todos los ejemplos de este subconjunto de datos (diferente al subconjunto de entrenamiento utilizado para entrenar la DNN), y vamos a ver qué porcentaje de ejemplos son acertados por la DNN. Esto es lo que se denomina como la precisión de la red y podemos calcularla mediante el siguiente código:

```
>>> correct_count, all_count = 0, 0
>>> for images,labels in valloader:
...  for i in range(len(labels)):
...     img = images[i].view(1, 784)
...     with torch.no_grad():
...         logps = model(img)
...
...
...     ps = torch.exp(logps)
...     probab = list(ps.numpy()[0])
...     pred_label = probab.index(max(probab))
...     true_label = labels.numpy()[i]
...     if (true_label == pred_label):
...       correct_count += 1
...     all_count += 1
...
>>> print("Number Of Images Tested =", all_count)
>>> print("\nModel Accuracy =", (correct_count/all_count))
```

5.6. Guardando los pesos entrenados

Una vez que hemos comprobado que nuestra DNN entrenada ofrece altos niveles de precisión la idea es guardar esos pesos, con el objetivo de utilizarlos en un futuro sin tener que volver a entrenar de nuevo la DNN. En PyTorch, podemos hacer esto de forma muy sencilla:

```
>>> WEIGHTS_PATH='./my_weights.pt'
>>> torch.save(model.state_dict(), WEIGHTS_PATH)
```

Tras esto, los pesos se quedarán guardados en el fichero **./my_weights.pt** y podrán ser cargados cuando se requieran mediante *torch.load* de la misma forma.

```
>>> model2 = My_DNN()
>>> model2.load_state_dict(torch.load(WEIGHTS_PATH))
>>> print(model2)
```

6. Ejercicios propuestos

Los cinco ejercicios que se proponen a continuación, tienen como objetivo entrenar, validar y usar en procedimientos de inferencia la red neuronal de reconocimiento de dígitos que se ha visto en esta práctica. De esta forma, la mayor parte del código de los ejemplos anteriores será la que haya que utilizar para resolverlos. Empecemos por crearnos el fichero `AMPE_dnn.py` donde almacenaremos el modelo de red neuronal y que importaremos posteriormente en cada uno de los ejercicios siguientes.

1. Implementa un script Python llamado `entrenamiento.py` que reciba como único parámetro el número de épocas de entrenamiento. El script llevará a cabo el procedimiento de entrenamiento de la red neuronal de reconocimiento de dígitos (que tienes almacenada en el fichero `AMPE_dnn.py`) con el número de épocas indicado, y escribirá los valores de los pesos obtenidos en el fichero `pesos-<n.epocas>.pt`, donde `<n.epocas>` será el valor del número de épocas pasado como argumento. A continuación, realiza el entrenamiento para 1, 10, 20 y 80 épocas (lo que dará lugar a los ficheros de pesos `pesos-1.pt`, `pesos-10.pt`, `pesos-20.pt` y `pesos-80.pt` (esta última prueba requerirá más de 10 minutos, aunque el tiempo exacto dependerá de la capacidad de cómputo de tu equipo). Realiza una tabla en la que se registren los tiempos de entrenamiento para cada número de épocas.

2. Implementa un script Python llamado `validacion.py` que reciba como único argumento un fichero de pesos, y utilizándolo, realice la validación de la red neuronal de reconocimiento de dígitos. En concreto, el script mostrará por pantalla la precisión del modelo (*accuracy*) usando el conjunto de pesos que contiene el fichero pasado como argumento. Realiza la validación usando los 4 ficheros de pesos generados en el ejercicio anterior y construye una tabla en la que se muestre, para cada uno de ellos, la precisión obtenida. ¿Qué efecto observas? ¿Puedes explicarlo[2]?

3. Implementa un script Python llamado `inferencia.py` que reciba dos argumentos: el primero, el nombre de un fichero con una imagen de tamaño 28×28 en escala de grises que contiene un dígito (puedes usar *Image.open(IMAGEN_INPUT_PATH)* de la librería *PIL* para cargar la imagen); y el segundo, el nombre de un fichero de pesos con que configurar la NN. El script realizará la inferencia con el modelo de NN desarrollado e informará acerca de qué dígito se trata. Obtén, ahora, 10 imágenes elegidas al azar del subconjunto de validación del dataset MNIST, guárdalas en 10 ficheros diferentes y, a continuación, pásaselas al script que has construido en este apartado para realizar la inferencia. En todos los casos usa el fichero de pesos para el que se obtuvo mejor precisión. ¿Cuántas de las 10 imágenes se infieren correctamente?

4. Vamos a realizar ahora la inferencia de dígitos escritos por ti. Utilizando papel y lápiz, escribe tres dígitos y échale una foto a cada uno. Cambia el tamaño de cada foto a 28×28 y pásala a escala de grises usando, por ejemplo, la herramienta *imagemagic* de Linux mediante la orden *convert -resize 28X28! -colorspace gray tuimagensrc imagendst* y cárgala mediante el método `Image.open()` de la librería PIL[3]. Utilizando el script del apartado anterior y el fichero de pesos con mejor precisión, ¿cuántos dígitos se reconocen correctamente? Puedes probar también a dibujar los dígitos usando un programa de edición gráfica (como GIMP).

5. Vamos a tratar mejorar la precisión de las predicciones realizando un preprocesamiento de las imágenes, eliminado el posible *ruido* que estas pudiesen tener y ajustándolas al formato que tienen las imágenes en MNIST. En concreto, prueba a convertir los colores claros a partir de un determinado valor umbral (que tendrás que determinar) a blanco y haz un poco más oscuros el resto (esto puedes hacerlo usando el método `Image.convert()` de PIL). Además, dado que en las imágenes de MNIST se usa el valor «0» para el blanco y el «255» para el negro, que es justo lo contrario que se hace en las imágenes que estamos tomando, hay que invertir los colores usando el método `ImageOps.invert()` de PIL.

[2] Pista: Busca información acerca de un fenómeno llamado *overfitting* que puede suceder en el entrenamiento de DNNs.

[3] Este mismo formato es entendible por el objeto `transform` de PyTorch para realizar el preprocesamiento final de las imágenes antes de ser pasadas a la NN.

Bibliografía

- **PyTorch docs**,
  https://PyTorch.org/docs/stable/index.html
- **Why You Need Python Environments and How to Manage Them with Conda**,
  https://www.freecodecamp.org/news/why-you-need-python-environments-and-how-to-manage-them-with-conda-85f155f4353c/
- **Handwritten Digit Recognition using PyTorch - Intro to Neural Networks**,
  https://towardsdatascience.com/handwritten-digit-mnist-PyTorch-977b5338e627