# Aprenentatge computacional

## Práctica 2

**Profesores:**
Pau Rodríguez
Jordi Gonzàlez

**Alumnos:**
Alejandro Garcia Carballo
1423957
Juan Plúa Gutiérrez
1358255

# Índice

# 1. Introducción

Este informe detalla  todo el proceso de desarrollo que se ha seguido para realizar la segunda práctica. Esta segunda práctica se presentan varios problemas que resolveremos y comprenderemos aplicando métodos de redes neuronales artificiales a problemas de clasificación multiclase.

En primer lugar, se aplicarán las diferentes tipologías de redes neuronales donde jugaremos con todos sus parámetros asociados, y, probaremos distintos modelos de la arquitectura de la red neuronal para determinar, a partir de los resultados obtenidos, qué influencia tiene cada uno de ellos.

En segundo lugar, una vez comprobados las diferentes configuraciones de redes neuronales, se evaluará y comparará entre las diferentes configuraciones estudiando los efectos de overfitting y underfitting según el valor que se le haya dado a los parámetros, se calculará la medida óptima de la red dependiendo del número de datos que tenemos disponibles, se estudiará la influència de diferentes métodos de inicialización de la red, y, también, la forma de agilitar la velocidad de la red reduciendo el tiempo de convergencia de esta a partir de una selección adecuada de los parámetros.

Por último, se intentará hacer aprender una única red neuronal con 10 neuronas de salida, donde cada neurona corresponderá a cada clase. Es decir, tendrá que aprender todas muestras de todas las clases a la vez, y la clasificación en cada clase corresponderá a aquella neurona de salida que se haya activado con más fuerza.

# 2. Estudio preliminar

En este apartado haremos un estudio del conjunto de datos que se nos ha proporcionado y cual es su significado. En esta práctica usaremos una base de datos MNIST, la cual es una gran base de datos de dígitos escritos a mano y que son especialmente utilizados para el entrenamiento de sistema de procesamiento de imágenes, así también, como en nuestro caso, para el aprendizaje automático.

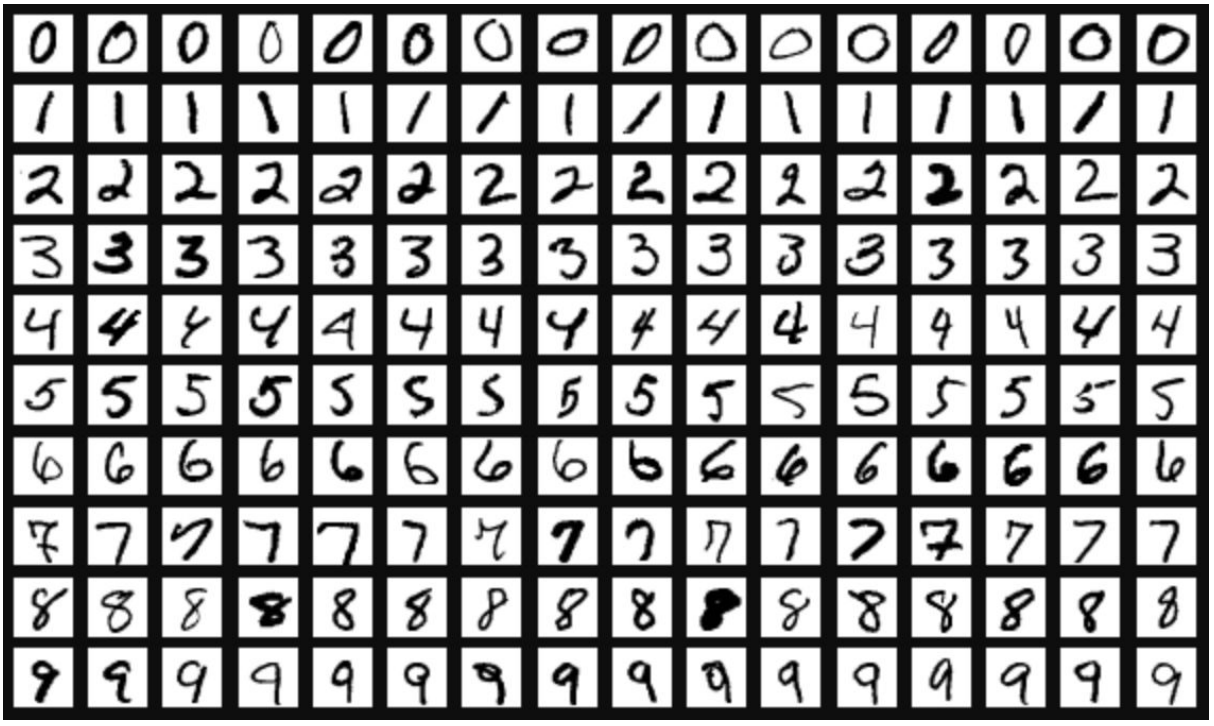


*Figura 1: muestra de imágenes de la base de datos MNIST*

Esta base de datos está formada por 60.000 imágenes de entrenamiento y 10.000 imágenes de prueba. El objetivo de esta base de datos es conseguir una tasa de error de reconocimiento lo más baja posible utilizando técnicas de redes neuronales.

# 3. Análisis e interpretación de los datos

En este apartado iremos explicando todo el desarrollo que se ha seguido en el análisis y el tratamiento de datos que se ha ido siguiendo. Responderemos a las preguntas propuestas con ejemplos con sus respectivas gráficas y resultados obtenidos.

### ¿Es el *accuracy* es una buena medida para este apartado?

El accuracy no es una buena medida ya que comparamos una clase positiva con el resto que son negativas (nomás un 10% son positivas), el número de clases negativas será mucho más grande que el de las clases positivas, por lo que si hacemos un recuento de las *True Negatives* obtendremos un resultado que se podría aceptar como correcto incluso no habiendo acertado ningún número, y, por lo tanto, no teniendo *True Positives*.

### ¿Podríais calcular qué accuracy se obtendría si el modelo tuviera output constante a 0?

Para comprobarlo, hemos ejecutado la red neuronal con un output fijado a 0 cambiando la siguiente línea:

```
def forward(self, x):
    # Here define architecture behavior
    x = torch.sigmoid(self.layer1(x))
    x = torch.sigmoid(self.layer2(x))
    #return torch.sigmoid(self.output(x)) # Binary output
    return torch.zeros((x.shape[0],1), requires_grad=True)
```

*Figura 2: Código fijar valores a 0*

Una vez fijados los output a zero, comprobamos los resultados de la gráfica y obtenemos un accuracy de 0.8977 de validación muy cercano al valor de entrenamiento que es 0.8978.

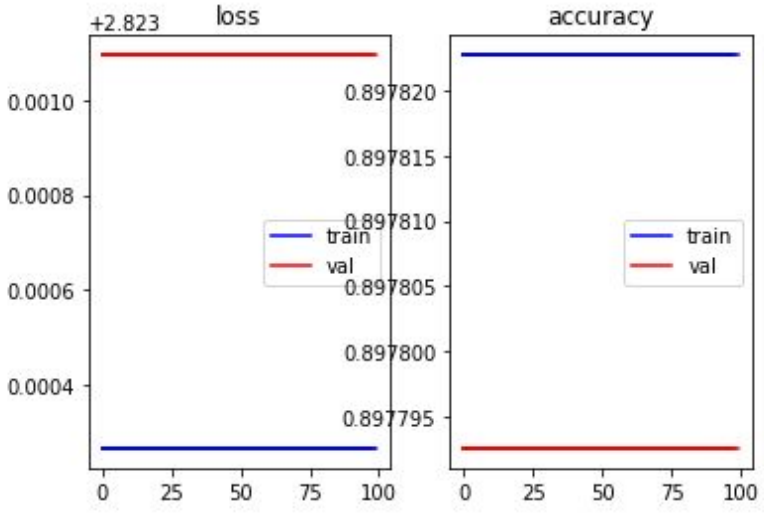



*Figura 3: Ejecución con valores output fijados a 0*

Por ejemplo:

**Valores reales:** 0 0 0 1 0 0 0 0 1 0 0
**Valores predecidos:** 0 0 0 0 0 0 0 0 0 0 0

True negative = 9
False negative = 2
Accuracy = 9/11 = 0.82

Para este caso, creemos que los True Positive son más importantes y por lo tanto una medida como la precisión podría ser más útil para este caso.

### ¿Y si fuera aleatorio?

Para fijar el output a aleatorio hemos cambiado la siguiente línea de código en el forward:

```
def forward(self, x):
    # Here define architecture behavior
    x = torch.sigmoid(self.layer1(x))
    x = torch.sigmoid(self.layer2(x))
    #return torch.sigmoid(self.output(x)) # Binary output
    return (0 - 1) * torch.rand((x.shape[0],1), requires_grad=True) + 1
```

*Figura 4: Código fijar valores aleatorios*

Si fijamos los output a un valor aleatorio entre 0 y 1 lo que obtendremos será una modelo con un accuracy cercano a 0.5, ya que acertarà aproximadamente la mitad de las veces y fallará la otra mitad.

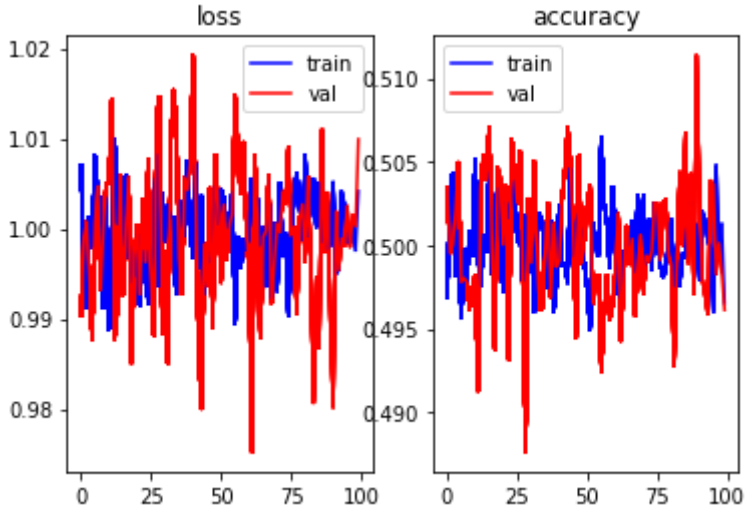



*Figura 5: Ejecución con valores output fijados aleatoriamente*

Para poder comprobar la influencia de los diferentes parámetros sobre la red neuronal, hemos hecho un conjunto de pruebas variando los hiperparámetros y viendo cuales son los resultados que obtenidos:

| Hyperparameter | | | | | Loss | |
|---|---|---|---|---|---|---|
| **N. Prueba** | **Momentum** | **Learning rate** | **batch size** | **epochs** | **training** | **validation** |
| 1 | 0.5 | 0.07 | 100 | 100 | 0.1 | 0.65 |
| 2 | 0.1 | 0.07 | 100 | 100 | 0.12 | 0.21 |
| 3 | 0.1 | 0.01 | 100 | 100 | 0.09 | 0.42 |
| 4 | 0.1 | 0.05 | 100 | 100 | 0.1 | 0.45 |
| 5 | 0.1 | 0.1 | 1000 | 100 | 0.19 | 0.16 |
| 6 | 0.1 | 0.1 | 1000 | 10 | 0.30 | 0.34 |
| 7 | 0.1 | 0.000001 | 1000 | 10 | 0.84 | 0.84 |
| 8 | 0.1 | 0.07 | 1000 | 100 | 0.11 | 0.13 |
| 9 | 0 | 0.1 | 1000 | 100 | 0.1 | 0.12 |

*Figura 6: Tabla pruebas con diferentes parámetros*

Como podemos observar en la *Figura 6*, teniendo en cuenta el *Loss*, la prueba número 9 es la que nos ha dado un mejor resultado, obteniendo un *Loss* de 0.1 en entrenamiento y un 0.12 en validación y obteniendo un accuracy de 0.97 en train y de 0.96 en validación.

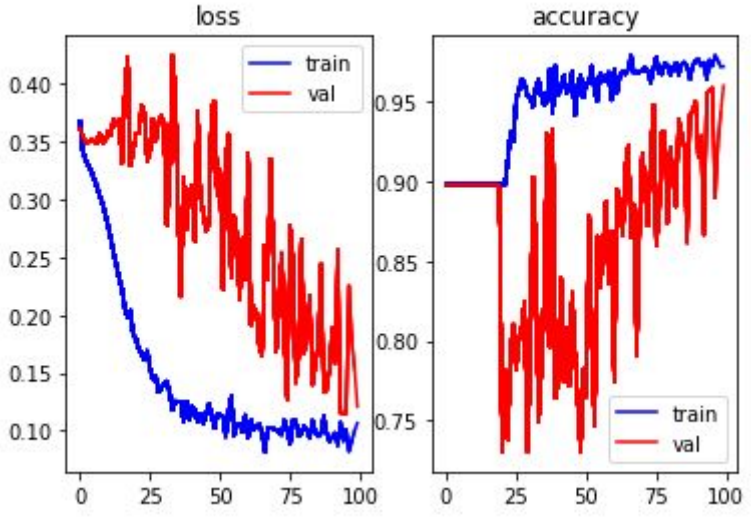



*Figura 7: Gráfica resultados prueba 10*

### ¿Qué influencia tiene el *learning rate* sobre el resultado final?

El learning rate influye en muchos aspectos. El *learning rate* es un hiperparámetro que controla la velocidad con la que recorremos la pendiente en el descenso de gradiente. Cuanto más bajo sea el valor, más lentamente recorreremos la pendiente.

Si el coeficiente de aprendizaje es mucho más pequeño, el descenso de gradiente puede ser muy lento, y si el *learning rate* es muy grande, el descenso de gradiente puede sobrepasar el mínimo y no converger. Por lo tanto, se tiene que ir probando un *learning rate* de más bajo a más alto (o a la inversa) e ir comprobando cuál es el que nos da un mejor resultado.

### ¿Qué arquitectura se comporta mejor?

A continuación, en la figura 8, mostramos los resultados obtenidos al variar la arquitectura, en concreto, la primera mitad, el número de sus capas, y la segunda, el número de neuronas.

Todas las pruebas realizadas en la tabla anterior se han ejecutado con un learning rate de 0,05 para comprobar si alguna arquitectura diferente añadiendo capas o neuronas nos da un mejor resultado. La primera prueba ha utilizado la arquitectura que venía por defecto en el código de muestra, con una capa de input, una de output y una capa oculta. Al hacer variaciones, hemos probado añadiendo capas sin modificar el número de neuronas, y el resultado no ha mejorado. También hemos probado posteriormente añadiendo capas y neuronas y tampoco hemos conseguido que mejore el resultado con de la arquitectura inicial. Más adelante haremos pruebas donde modificaremos también la función de activación y comprobaremos los efectos que tiene sobre la red neuronal.

| Architectures | | | | | | |
|---|---|---|---|---|---|---|
| # layers | Layer | # neurons | Accuracy | | Loss | |
| | | | Training | Validation | Training | Validation |
| 3 | Layer 1 | $28^2$ | 0.9600375 | 0.7934194 | 0.1126689 | 0.3824305 |
| | Layer 2 | 4 | | | | |
| | Layer 3 | 4 | | | | |
| 4 | Layer 1 | $28^2$ | 0.9734764 | 0.7059558 | 0.0862636 | 0.5263381 |
| | Layer 2 | 4 | | | | |
| | Layer 3 | 4 | | | | |
| | Layer 4 | 4 | | | | |
| 5 | Layer 1 | $28^2$ | 0.9732055 | 0.4119950 | 0.0846971 | 1.0026668 |
| | Layer 2 | 4 | | | | |
| | Layer 3 | 4 | | | | |
| | Layer 4 | 4 | | | | |
| | Layer 5 | 4 | | | | |
| 2 | Layer 1 | $28^2$ | 0.9289717 | 0.1022074 | 0.1979476 | 1.7746183 |
| | Layer 2 | 4 | | | | |
| 3 | Layer 1 | $28^2$ | 0.9809355 | 0.7556851 | 0.0762669 | 0.7112880 |
| | Layer 2 | 10 | | | | |
| | Layer 3 | 10 | | | | |
| 3 | Layer 1 | $28^2$ | 0.9885821 | 0.5063915 | 0.0377745 | 1.8137933 |
| | Layer 2 | 32 | | | | |
| | Layer 3 | 64 | | | | |
| 4 | Layer 1 | $28^2$ | 0.9928325 | 0.4370678 | 0.0244635 | 1.8945795 |
| | Layer 2 | 50 | | | | |
| | Layer 3 | 50 | | | | |
| | Layer 4 | 50 | | | | |

*Figura 8: Tabla pruebas con diferentes arquitecturas*

## ¿Tiene sentido normalizar cada muestra?

Lo primero que hemos hecho ha sido comprobar cómo era la estructura de las imágenes:

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0  36  43  43  22   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  29  94
  190 242 253 252 212  28   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  22 187 252
  252 252 253 252 252 239 100   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 141 253 252 252
  252 252 253 252 252 252 252   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   9 176 246 253 182  51
```

*Figura 8: Imagen sin normalizar*

Como comprobamos, las imágenes contienen valores RGB, donde el valor máximo es el 255. Para realizar la normalización, hemos dividido todos los valores de las estructuras de las imágenes entre 255:

```
#Normalitzem les dades, pasem de RGB a un valor entre 0 i 1
train_images = train_images.astype('float32')
val_images = val_images.astype('float32')
val_images /= 255
train_images /= 255
```

*Figura 9: Normalización de los datos*

Y volvemos a comprobar que la estructura ahora está normalizada:

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.14117648 0.16862746 0.16862746 0.08627451 0.
 0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.11372549 0.36862746
 0.74509805 0.9490196  0.99215686 0.9882353  0.83137256 0.10980392
 0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.08627451 0.73333335 0.9882353
 0.9882353  0.9882353  0.99215686 0.9882353  0.9882353  0.9372549
 0.39215687 0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.5529412  0.99215686 0.9882353  0.9882353
```

*Figura 10: Imagen normalizada*

Todos los valores están en un rango entre 0 y 1. Ahora queremos hacer una ejecución con los datos normalizados para comprobar si los resultados mejoran al tener los datos normalizados con los mismos hiperparámetros que una ejecución sin normalizar:

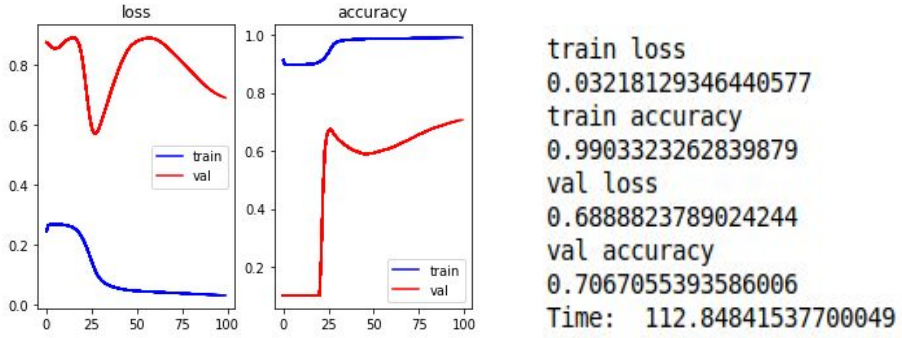



*Figura 11: Resultado con datos normalizados*

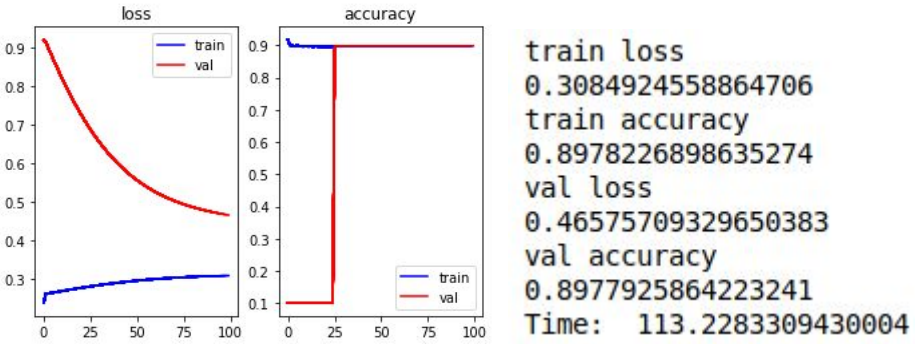



*Figura 12: Resultado con datos sin normalizar*

Como observamos en los resultados obtenidos, el hecho de normalizar no nos está haciendo conseguir un mejor resultado, más bien nos ha empeorado el resultado al utilizar las imágenes con los valores normalizados.

### ¿Se consigue siempre el mismo resultado para una misma arquitectura? ¿Por qué?

No siempre se consigue el mismo resultado cuando el peso que reciben las neuronas es aleatorio, podría decirse que el resultado obtenido al realizar varias pruebas es similar pero no el mismo.
Pero, si nosotros inicializamos peso de las neuronas cada vez que hagamos una prueba, en este caso, sí obtendremos el mismo resultado. Esta es una forma de comprobar si el cambio que hemos añadido a nuestra arquitectura ha resultado beneficiosa respecto al resultado.

Para inicializar los pesos, hemos creado la siguiente función:

```
#init weights
def init_weights(m):
    if type(m) == torch.nn.Linear:
        m.weight.data.fill_(0.01)
```

*Figura 13: Resultado con datos sin normalizar*

Inicializamos los pesos a 0.01. Para aplicar la inicialización, podemos hacerlo de la siguiente manera:

```
#Binary output
#Instantiate network
model = NeuralNet()
model.apply(init_weights)
model.state_dict()
```

*Figura 14: Inicialización de pesos*

Con la última línea de código, lo que hacemos es mostrar el estado de la red:

```
OrderedDict([('layer1.weight',
              tensor([[0.0100, 0.0100, 0.0100,  ..., 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100,  ..., 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100,  ..., 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100,  ..., 0.0100, 0.0100, 0.0100]])),
             ('layer1.bias', tensor([ 0.0339, -0.0337,  0.0317, -0.0268])),
             ('layer2.weight', tensor([[0.0100, 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100, 0.0100],
                      [0.0100, 0.0100, 0.0100, 0.0100]])),
             ('layer2.bias', tensor([-0.2732,  0.3924,  0.2143,  0.4799])),
             ('output.weight', tensor([[0.0100, 0.0100, 0.0100, 0.0100]])),
             ('output.bias', tensor([-0.4550]))])
```

*Figura 15: Resultado con datos sin normalizar*

Como podemos observar, la inicialización se realiza correctamente. A partir de aquí, podemos hacer pruebas con distintas arquitecturas para ver cómo influye la arquitectura y comprobar si alguna de ellas tiene mejores resultados.

### ¿Qué relación hay entre el número de pesos y el overfitting/underfitting?

Hay varios elementos que pueden influir en que haya overfitting o underfitting en un modelo de red neuronal. El hecho de incrementar el número de capas o el número de neuronas puede ocasionar que haya overfitting en nuestro modelo. Al aumentar el número de capas o de neuronas, también se incrementa el número de pesos.

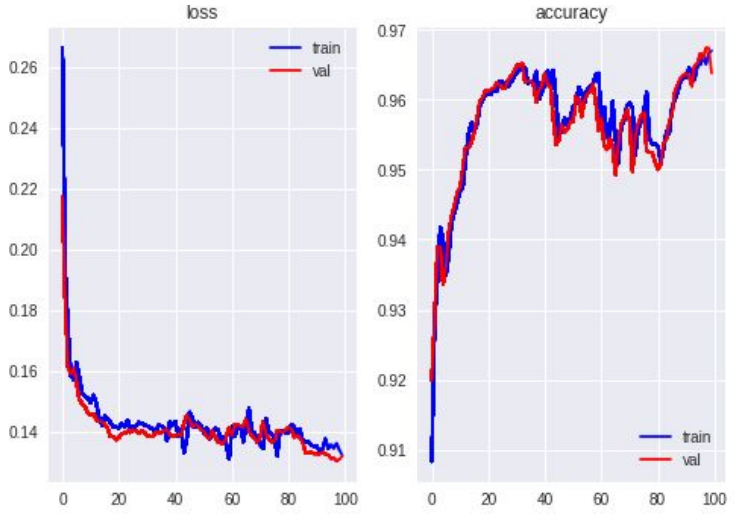



*Figura 16: Gráfica con overfitting*

### ¿Hay alguna diferencia entre la medida óptima/teórica de la red y la que da mejor resultado?

Para comprobar cuál es la arquitectura que nos da un mejor resultado, iremos haciendo prueba fijando el valor de la inicialización de los pesos y el valor del learning rate. El *learning rate* lo fijaremos a 0.1 mientras que la inicialización de los pesos la inicializaremos a 0.01. Una vez fijados estos parámetros, añadiremos más capas y más neuronas para comprobar cuál es la que nos da un mejor rendimiento y probaremos también alguna función de activación distinta a la sigmoid.

| Architectures | | | | | | | |
|---|---|---|---|---|---|---|---|
| # layers | Layer | # neurons | Activation function | Accuracy | | Loss | |
| | | | | Training | Validation | Training | Validation |
| 3 | Layer 1 | $28^2$ | Sigmoid | 0.897822 | 0.8977925 | 0.335947 | 0.3401864 |
| | Layer 2 | 4 | | | | | |
| | Layer 3 | 4 | | | | | |
| 4 | Layer 1 | $28^2$ | Sigmoid | 0.897822 | 0.8977925 | 0.335953 | 0.3406793 |
| | Layer 2 | 4 | | | | | |
| | Layer 3 | 4 | | | | | |
| | Layer 4 | 4 | | | | | |
| 4 | Layer 1 | $28^2$ | Sigmoid | 0.897822 | 0.8977925 | 0.336695 | 0.3491937 |
| | Layer 2 | 10 | | | | | |
| | Layer 3 | 10 | | | | | |
| | Layer 4 | 10 | | | | | |
| 3 | Layer 1 | $28^2$ | Tanh | 0.98291 | 0.871303 | 0.05447 | 0.276374 |
| | Layer 2 | 4 | Tanh | | | | |
| | Layer 3 | 4 | Sigmoid | | | | |
| 4 | Layer 1 | $28^2$ | Tanh | 0.972726 | 0.891461 | 0.085433 | 0.246182 |
| | Layer 2 | 50 | Tanh | | | | |
| | Layer 3 | 50 | Tanh | | | | |
| | Layer 4 | 50 | Sigmoid | | | | |
| 5 | Layer 1 | $28^2$ | relu | 0.98181 | 0.864639 | 0.06610 | 0.28013 |
| | Layer 2 | 50 | relu | | | | |
| | Layer 3 | 50 | relu | | | | |
| | Layer 4 | 50 | Sigmoid | | | | |
| 3 | Layer 1 | $28^2$ | relu | 0.97837 | 0.87788 | 0.07526 | 0.25703 |
| | Layer 2 | 4 | relu | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | Layer 3 | 4 | Sigmoid | | | | |

*Figura 17: Tabla de pruebas con inicialización fija*

Aunque la mayoría de arquitecturas nos están dando unos resultados muy similares, no hemos mejorado el accuracy utilizando más capas y más neuronas a la hora de probar arquitecturas distintas, con distintas funciones de activación. Hemos probado de utilizar las funciones de activación de tanh y de relu a parte de la función sigmoid que es la que venía en el código inicialmente.

### ¿Para cual inicialización de pesos del aprendizaje mejora o es más rápido en converger?

Para comprobar que inicialización es más rápida en converger o mejora el aprendizaje lo que haremos será comprobar distintas inicializaciones utilizando los mismos hiperparametros en todas las ejecuciones y ver como varían los resultados.

| Función | Tiempo | Accuracy | Loss |
|---|---|---|---|
| Xavier uniform | 41.4 | 0.94 | 0.15 |
| Valores fijados | 52.5 | 0.95 | 0.14 |
| Kaiming uniform | 45.8 | 0.94 | 0.15 |
| Kaiming normal | 45.1 | 0.95 | 0.14 |
| Orthogonal | 46.9 | 0.94 | 0.16 |
| Fan in and fan out | 46.9 | 0.94 | 0.15 |
| Random | 47.3 | 0.94 | 0.18 |

*Figura 18: Gráfica del tiempo con diferentes inicializaciones*

Como podemos comprobar, de todas las inicializaciones, la que está dando mejor rendimiento en tiempo es la de xavier uniform con un tiempo de 41.4, aunque en accuracy hay algunas alternativas que nos dan un resultado muy ligeramente superior, como puede ser kaiming normal.

En el anexo se muestran el conjunto de pruebas que se han realizado con sus respectivas gráficas y las funciones utilizadas.

# 4. Dificultades

Durante el proceso de realización de este proyecto no hemos tenido dificultades, más bien hemos tenido dudas a la hora de saber escoger una metodología o saber qué pautas seguir para llevar a cabo nuestra práctica.

Estas dudas se han ido despejando a la hora de ver las presentaciones de nuestros compañeros en las sesiones informativas, y, también, gracias al foro en el cual han salido varias cuestiones que nosotros también nos planteamos en su momento.

Al igual que en las otras prácticas, los resultados obtenidos nos hacían replantear nuestro trabajo, pero, tal y como se pedía en esta, hemos ido más allá de los resultados y hemos buscado una relación y sentido tenían estos.

# 5. Conclusiones

En esta práctica hemos logrado hacer completar en gran parte todos los apartados donde hemos obtenido resultados muy variados al aplicar diferentes parámetros a las variables y diferentes arquitecturas.

Aplicando diferentes valores en las variables en nuestro mejor resultado hemos conseguido un *Loss* de 0.1 en entrenamiento y un 0.12 en validación, obteniendo un accuracy de 0.97 en entrenamiento y de 0.96 en validación. En cuanto a la modificación de la arquitectura, al añadir más y menos capas, más y menos neuronas, no hemos podido conseguir mejorar los resultados que ya teníamos por defecto en el código de muestra.

Por lo que respecta al entorno de trabajo, la experiencia de programar el código en los programas de lenguaje python en Jupyter Notebook ha resultado satisfactorio ya que es una manera de trabajar ordenada, cómoda y sencilla. La información recogida en la página web de Pytorch ha resultado de gran ayuda ya muestra con ejemplos cómo aplicar funciones para mejorar nuestro código.

Por último, consideramos que lo aprendido a lo largo de esta segunda práctica ha sido muy interesante y positivo con lo que se puede llegar a lograr aplicando técnicas de redes neuronales.

# 6. Anexo

En este apartado mostraremos cómo hemos instalado las librerias necesarias para realizar el proyecto y como hemos hecho una contribución a codalab, aunque con no muy buena puntuación.

Primero de todo hemos instalado la librería pytorch. Hemos trabajado en un sistema operativo Linux, Ubuntu. La instalación la hemos llevado a cabo a través de conda con el siguiente comando:

```
conda install pytorch-cpu torchvision-cpu -c pytorch
```

Una vez instalado torch, ejecutamos el script run_tests.sh, el cual pasa por parámetro los valores de los hiperparámetros que queremos utilizar en la ejecución y ejecuta el fichero practica_3.py proporcionado en el material de la práctica.

```
-----------------------------------------------------------------------
| End of epoch:   6 | Time: 0.06s | Val loss: 0.167 | Val acc: 0.945|
-----------------------------------------------------------------------
| End of epoch:   7 | Time: 0.71s | Train loss: 0.170 | Train acc: 0.927|
-----------------------------------------------------------------------
| End of epoch:   7 | Time: 0.09s | Val loss: 0.190 | Val acc: 0.915|
-----------------------------------------------------------------------
| End of epoch:   8 | Time: 0.56s | Train loss: 0.160 | Train acc: 0.937|
-----------------------------------------------------------------------
| End of epoch:   8 | Time: 0.04s | Val loss: 0.153 | Val acc: 0.935|
-----------------------------------------------------------------------
| End of epoch:   9 | Time: 0.56s | Train loss: 0.161 | Train acc: 0.938|
-----------------------------------------------------------------------
| End of epoch:   9 | Time: 0.04s | Val loss: 0.170 | Val acc: 0.932|
Accuracy: 0.1042898792
```

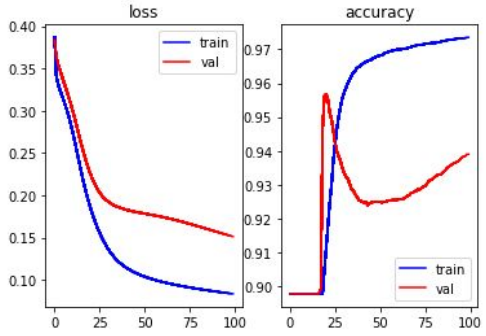
A continuación, con la cuenta de codalab ya creada, vamos al apartado de competitions y subimos el archivo .pkl generado al ejecutar Practica_3.py, que se encuentra en la carpeta result/final/val_preds.pkl.

| # | Username | Score |
|---|---|---|
| 1 | Grup104 | 0.108100 |
| 2 | Grup608 | 0.100700 |
| 3 | apc2018_Grup02 | 0.097500 |

A continuación se mostrarán las diferentes funciones y gráficas obtenidas con los distintos tipos de inicialización.

Primeramente probaremos con la inicialización de xavier uniform. Para ello, utilizaremos la función de torch y lo aplicaremos a la red de la siguiente manera:

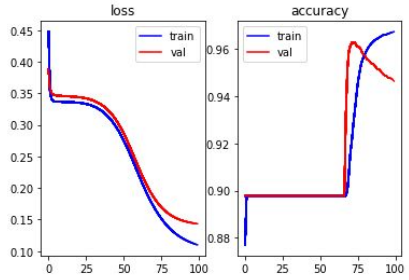
```
train loss
0.08369061665952746
train accuracy
0.9735180747994583
val loss
0.15131380115757878
val accuracy
0.9390254060807997
Time:  41.40203987700079
```




```
model = NeuralNet()
#model.apply(init_weights)
torch.nn.init.xavier_uniform_(model.layer1.weight)
torch.nn.init.xavier_uniform_(model.layer2.weight)
torch.nn.init.xavier_uniform_(model.output.weight)
```

Ahora probaremos fijando los pesos a un mismo valor para todas las capas. Para ello, usaremos la siguiente función:

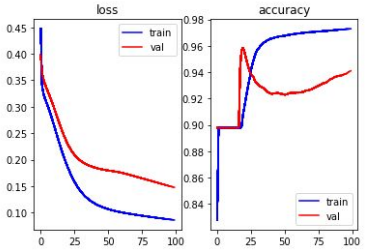
```
train loss
0.1098154271232025
train accuracy
0.9673716012084592
val loss
0.14332745431215255
val accuracy
0.946438983756768
Time:  52.45787909299543
```




```
#init weights
def init_weights(m):
    if type(m) == torch.nn.Linear:
        m.weight.data.fill_(0.01)
```

Para el caso de kaiming uniform:

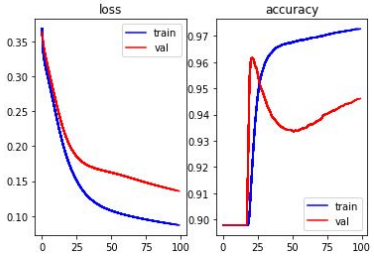
```
train loss
0.08627113864851912
train accuracy
0.9728930096885092
val loss
0.14779798991230317
val accuracy
0.9410245730945439
Time:  45.84458324600564
```




```
model = NeuralNet()
#model.apply(init_weights)
torch.nn.init.kaiming_uniform_(model.layer1.weight)
torch.nn.init.kaiming_uniform_(model.layer2.weight)
torch.nn.init.kaiming_uniform_(model.output.weight)
```

Utilizando la inicialización de kaiming normal:

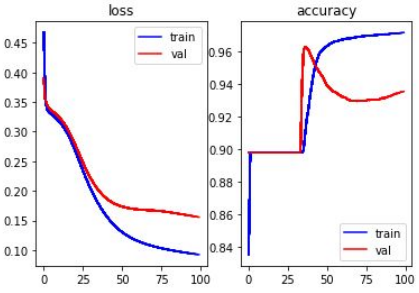
```
train loss
0.0869343728700278
train accuracy
0.9727054901552245
val loss
0.13563579109895732
val accuracy
0.9461057892544773
Time:  45.12582462100545
```




```
model = NeuralNet()
#model.apply(init_weights)
torch.nn.init.kaiming_normal_(model.layer1.weight)
torch.nn.init.kaiming_normal_(model.layer2.weight)
torch.nn.init.kaiming_normal_(model.output.weight)
```

Usando la inicialización orthogonal:

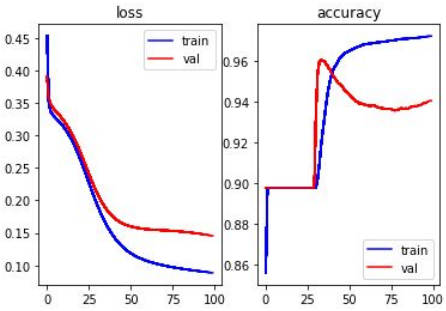
```
train loss
0.09265956155088795
train accuracy
0.971497030940723
val loss
0.1557589540787907
val accuracy
0.9353602665556018
Time:  46.89332027500495
```




```
model = NeuralNet()
#model.apply(init_weights)
torch.nn.init.orthogonal_(model.layer1.weight)
torch.nn.init.orthogonal_(model.layer2.weight)
torch.nn.init.orthogonal_(model.output.weight)
```

Fan in Fan out:

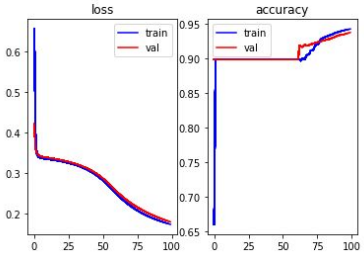
```
train loss
0.08854604295199459
train accuracy
0.9721220960516721
val loss
0.1453688324168045
val accuracy
0.9404414827155352
Time:  46.925324876996456
```




```
model = NeuralNet()
#model.apply(init_weights)
torch.nn.init._calculate_fan_in_and_fan_out(model.layer1.weight)
torch.nn.init._calculate_fan_in_and_fan_out(model.layer2.weight)
torch.nn.init._calculate_fan_in_and_fan_out(model.output.weight)
```

Inicialización random:

```
train loss
0.17275513348132077
train accuracy
0.9420356287113241
val loss
0.17896523325704425
val accuracy
0.9371095376926281
Time:  47.28642894500081
```




```
model = NeuralNet()
model.apply(weights_init_random)
```

# 7. Bibliografía


- Material de clase: https://caronte.uab.cat/
- Página de Pytorch: https://pytorch.org/
- Towards Data Science. Image classification in 10 minutes with MNIST Dataset: https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d
- MNIST dataset introduction: https://corochann.com/mnist-dataset-introduction-1138.html