Práctica 2: Perceptrón multicapa para problemas de clasificación

Ricardo Espantaleón Pérez
31033081D
i92esper@uco.es

Introducción a los Modelos Computacionales

Ingeniería Informática Ciencias de la computación

Escuela Politécnica Superior de Córdoba

Universidad de Córdoba

Cuarto curso. Primer cuatrimestre
Curso 2022/2023

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Descripción de las adaptaciones a los modelos

Para esta práctica se requiere llevar modelos enfocados a clasificación partiendo del esqueleto de la práctica anterior. Para este tipo de problemas, la opción más conveniente a utilizar es la función **softmax como salida del modelo para la clasificación**.

El caso más sencillo para explicar las diferencias respecto a la práctica anterior, sería con el ejemplo de XOR. En el caso de la práctica anterior se tenía una única salida como función sigmoide, donde se esperaba que devolviese 0 o 1 en función de las entradas, para ello el objetivo era intentar optimizar el modelo para que la función sigmoide en esos casos devolviese valores cercanos a 0 y 1. Para este caso se tienen **dos nodos de salida, uno correspondiente a cada posible salida del modelo, 1 o 0 en este caso, donde la salida de cada neurona es la probabilidad de pertenencia a dicha clase haciendo uso de la función softmax** 1.1.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad para\ i = 1, 2, \ldots, K \tag{1.1}$$

Se puede deducir que para **clasificar** N **clases, solamente se requieren** $N - 1$ **nodos en capa de salida, ya que la última clase se puede calcular como la diferencia de probabilidades una vez aplicada softmax**.

A su vez, al realizar clasificación es conveniente optar por la función de error **entropía cruzada**, ya que *MSE* es más conveniente para problemas de regresión, y para el caso de la función *softmax* al aplicar *backpropagation* y optimizar las derivadas asociadas a la función de error y capa de salida, tiene más cabida en problemas de clasificación, ofreciendo mejores resultados. También como métrica para evaluar el modelo final se usará el ***CCR***, el porcentaje de patrones clasificados correctamente.

También se ha implementado el **tipo de aprendizaje *batch* u *offline***, donde el cambio más significativo se encuentra en el acumulado de los *deltas* asociados tras aplicar backpropagation, donde dicha acumulación no se realizará por cada patrón y se aplicará al peso, **en este caso se acumulará tras examinar todos los patrones (*batch*)** y se aplicará finalmente al peso $w_{i,j}^{l}$ correspondiente.

Capítulo 2

Descripción en pseudocódigo de las modificaciones aplicadas

Los cambios más significativos enfocado al aprendizaje de modelos llevado a cabo serían los siguientes:

1. Modo *online* u *offline* 1.
2. *ForwardPropagate*: para contemplar *softmax* o *sigmoide* como nodos de salida 2.
3. *BackPropagate*: donde se debe determinar, en base a la función de error, la derivada correspondiente asociada para llevar a cabo la optimización del modelo mediante descenso del gradiente, además de determinar en base a la última función de activación de la capa de salida la derivada a llevar a cabo 4.
4. *WeightAdjustmentOffline*: ajuste de pesos en base al número de patrones para el modo *offline* 5.
5. *Softmax*: función a parte requerida dado que se necesita ver el valor de los nodos vecinos de la misma capa para el cálculo del sumatorio de las exponenciales 3.
6. Normalización: en este caso **no se deberán normalizar las salidas para problemas de clasificación**.

Algorithm 1 Modo batch u online mode para entrenamiento

```
1: procedure TRAIN(trainDataset, errorFunction, onlineMode)
2:     if onlineMode then
3:         for all patrón p de trainDataset do
4:             performEpochOnline(p.inputs, p.outputs, errorFunction)  ▷ La limpieza de los deltas y el ajuste se realiza dentro de la propia función cada vez que se invoca, igual que en la práctica anterior
5:     else
6:         cleanDeltaWeights()
7:         for all patrón p de trainDataset do
8:             performEpochOffline(p.inputs, p.outputs, errorFunction) ▷ Dentro de esta función no se realiza ni la limpieza ni el ajuste
9:         weightAdjustmentOffline()
```

Algorithm 2 Propagación hacia delante

procedure FORWARDPROPAGATE
  for all capa l desde la capa 1 **do**
    for all neurona j de la capa l **do**
      $net \leftarrow 0$
      for all neurona k de la capa $l-1$ **do**
        if $n_j.w \neq NULL$ **then** ▷ Consideración en caso de usar la función softmax, donde se prescinde de la última neurona
          $net \leftarrow net + n_k.out * n_j.w_k$
        if $n_j.w \neq NULL$ **then** ▷ Sesgo
          $net \leftarrow net + n_j.w_{layers^{l-1}.nOfNeurons}$
        if $l = LAST\_LAYER$ **then** ▷ Si estamos en la última capa
          if $outputFunction = 0$ **then** ▷ Se aplica la función *sigmoide* en la última capa

$$n_j.out = sigmoid(net)$$

          else ▷ Se aplica la función *softmax* a la última capa
            $softmax(l)$ ▷ Se realiza una función a parte dada la vencidad requerida en el calculo de la función
        else ▷ Al resto de neuronas de la red se le aplica *sigmoide*

$$n_j.out = sigmoid(net)$$

Algorithm 3 Función softmax

procedure SOFTMAX(capa l)
  $net(l.nOfNeurons) \leftarrow 0$ *for each one* ▷ Vector de *nets* asociado a cada neurona de la última capa
  $sumOfExp \leftarrow 0$
  for all neurona j de la capa l **do** ▷ Siempre se aplicará a la última capa
    if $n_j = l.nOfNeurons$ **then** ▷ última neurona
      $net_j \leftarrow 0$ ▷ Se obvia dado que no se requiere
    for all neurona k de la capa $l-1$ **do**
      $net_j \leftarrow net_j + n_j.w_k * n_k.out$
    $net_j \leftarrow net_j + n_j.w_{layers^{l-1}.nOfNeurons}$ ▷ Sesgo
    $sumOfExp \leftarrow sumOfExp + exp(net_j)$
  for all neurona j de la capa l **do**
    $n_j.out = \frac{exp(net_j)}{sumOfExp}$

Algorithm 5 Ajuste de pesos para el caso offline

procedure WEIGHTADJUSTMENTOFFLINE
  for all capa i desde la 1 **do**
    for all neurona j de la capa i **do**
      for all neurona k de la capa $i-1$ **do**
        if $n_j.w_k \neq NULL$ **then**
          $n_j.w_k \leftarrow n_j.w_k + ((\eta * n_j.deltaW_k)/nOfTrainingPatterns) + ((\mu * \eta * n_j.lastDeltaW_k)/nOfTrainingPatterns)$
      if $n_j.w_k \neq NULL$ **then** ▷ Sesgo
        $n_j.w^{nOfNeurons-1}_{layers_{i-1}} \leftarrow n_j.w^{nOfNeurons-1}_{layers_{i-1}} + ((\eta * n_j.deltaW^{nOfNeurons-1}_{layers_{i-1}})/nOfTrainingPatterns) + ((\mu * \eta * n_j.lastDeltaW^{nOfNeurons-1}_{layers_{i-1}})/nOfTrainingPatterns)$

Algorithm 4 Backpropagation

```
1: procedure BackpropagateError(target, errorFunction)
2:     for all capa i desde la l − 1 hasta la 0 do
3:         for all neurona j de la capa i do
4:             if i = nOfLayers − 1 then                          ▷ Última capa
5:                 if outputFunction = 0 then          ▷ Función sigmoide como salida
6:                     if errorFuncion = 0 then                              ▷ MSE
7:                         out ← n_j.out
8:                         error ← target_j − out
9:                         n_j.delta = 2 ∗ error ∗ out ∗ (1 − out)
10:                    else                                    ▷ Entropía cruzada
11:                        out ← n_j.out
12:                        error ← target_j/out
13:                        n_j.delta = error ∗ out ∗ (1 − out)
14:                else                                        ▷ Función softmax
15:                    if errorFuncion = 0 then                          ▷ MSE
16:                        out ← n_j.out
17:                        if n_j.deltaW ≠ NULL then
18:                            n_j.delta ← 0
19:                            for all neurona k de la capa i do
20:                                bool ← (k == j)
21:                                n_j.delta ← n_j.delta+((target_k−n_k.out)∗out∗(bool−
    n_k.out))
22:                    else                                    ▷ Entropía cruzada
23:                        out ← n_j.out
24:                        if n_j.deltaW ≠ NULL then
25:                            n_j.delta ← 0
26:                            for all neurona k de la capa i do
27:                                bool ← (k == j)
28:                                n_j.delta ← n_j.delta+((target_k/n_k.out)∗out∗(bool−
    n_k.out))
        ▷ Para el resto de capas se calcula delta, igual que en la práctica anterior
```

Capítulo 3

Experimentos y análisis de los modelos

Para la selección de topologías se han probado y testado las arquitecturas propuestas en la memoria, seleccionándose en base al valor del *CRR* en los dataset de *tests*. La selección se ha hecho acorde a cada dataset, mediante la media del error (considerándose también σ) en las 5 semillas establecidas inicialmente en la práctica.

Los hiper-parámetros se han fijado por defecto, siendo estos los siguientes:

- η : 0,7
- μ : 1
- *Learning Rate Scheduler* : *None*
- *Iteraciones* : 1000. 500 para *noMNIST*.
- *Normalización de los valores de entrada* : *False* (*True solamente en ILDP*)

Una vez sean determinadas las mejores topologías, se realizará una optimización de hiper-parámetros mediante un *script* realizado en *python* (B), para intentar mejorar lo máximo posible el valor de *CCR*.

No se realizará la prueba de entropía cruzada con función *sigmoide* en la salida, ya que esta combinación funcionará mal. Esto se debe a que la función

sigmoide no contempla la vecindad de sus neuronas en la última capa de salida, no creando una probabilidad de pertenencia normalizada como sería en el caso de *softmax*, no siendo por tanto óptimo para la entropía cruzada, ya que la función *sigmoide* **determina la pertenencia si supera cierto umbral**, no como el caso de *softmax* que, al ser probabilidades normalizadas, se puede hacer uso del *argmax*.

3.1 Dataset XOR

3.1.1. Descripción del dataset

El dataset XOR es un problema clásico de perceptrones multicapa, puesto que este problema no es posible que sea resuelto mediante el perceptrón simple, ya que este no es linealmente separable. Este mismo problema se realizó en la práctica anterior, enfocado a regresión. La diferencia fundamental respecto a la práctica anterior se halla en que el dataset posee 2 salidas, respecto a la única del problema anterior en regresión 3.1.

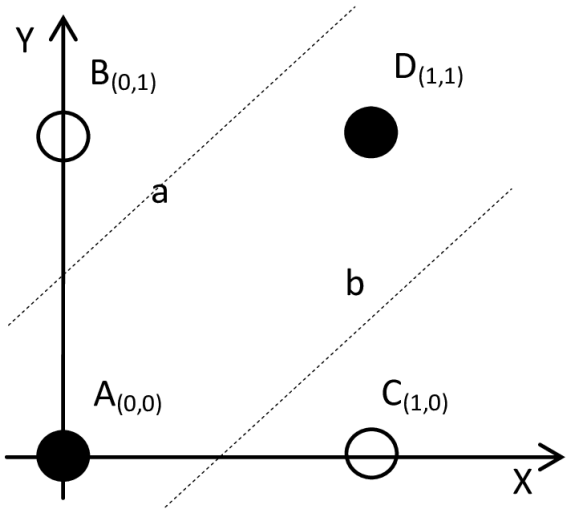


Figura 3.1: Distribución espacial del dataset *XOR*

1	*1*	-1	1	0
2	*-1*	-1	0	1
3	*-1*	1	1	0
4	*1*	1	0	1

Tabla 3.1: Extracto del fichero *.dat* del dataset *XOR*

3.1.2. Mejor topología encontrada

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 100 : 100 : k}	***True***	***softmax***	***Crossentropy***	**1 +- 0**

Tabla 3.2: Resultados del experimento 1 en el dataset de *XOR*

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 100 : 100 : k}	***True***	***sigmoide***	***MSE***	**1 +- 0**
{n : 100 : 100 : k}	*False*	*sigmoide*	*MSE*	0.55 +- 0.01
{n : 100 : 100 : k}	*True*	*softmax*	*MSE*	1 +- 0
{n : 100 : 100 : k}	*False*	*softmax*	*MSE*	1 +- 0
{n : 100 : 100 : k}	*True*	*softmax*	*Crossentropy*	1 +- 0
{n : 100 : 100 : k}	*False*	*softmax*	*Crossentropy*	1 +- 0

Tabla 3.3: Resultados del experimento 2 en el dataset de *XOR*

Se puede observar que el mejor modelo propuesto es aquel con **2 capas ocultas y 100 neuronas por capa oculta** 3.2.

Tras el experimento 3.3, se puede determinar que cualquiera de los modelos funciona de manera óptima en *CCR* para el dataset de *XOR*.

Se va a considerar seleccionar el modelo con **función de activación *sigmoide* y función de error *MSE* y modo *offline***.

3.1.3. Optimización de hiper-parámetros

Dado que el modelo resultante ofrece resultados muy buenos sin necesidad de optimizar, se obviará este apartado para problemas que requieran de mayor optimización. Los valores iniciales se adjuntan en la tabla 3.4.

Mejor eta	0.7
Mejor mu	1
Mejor Learning Rate Scheduler	*None*
Mejor Factor de Normalización	*False*
Arquitectura final	$\{2 : 100 : 100 : 2\}$

Tabla 3.4: Hiper-parámetros finales del dataset *XOR*

3.1.4. Modelo final

El modelo resultante tras las 1000 iteraciones ha obtenido un resultado de **1 +- 0 en *CCR*** y **0.0022 +- 1.6518e-06 en *MSE*** en *tests*. En este caso no ha podido ser posible realizar *crossvalidation* con un porcentaje del dataset, ya que no se podría generalizar nunca dado que faltarían patrones fundamentales para recrear *XOR*, por lo que la gráfica de convergencia se ha realizado mediante el dataset de *tests*, que es el mismo que el de entrenamiento, por ello se encuentran solapadas en las gráficas.

Para el caso del *CCR* se ha optado por mostrar las 100 primeras iteraciones para apreciar de manera más clara la gráfica de convergencia del *CCR* 3.2, donde se aprecia que a partir de la época 70 no se requiere seguir el entrenamiento ya que el resultado en *tests* es máximo. Para el caso de la convergencia mediante *MSE* 3.3, la convergencia del error es esperable, un descenso muy rápido en las primeras épocas donde a partir de la época 70 el error se mantiene con valores muy bajos. Para todas las semillas el modelo acaba convergiendo de manera óptima al máximo CCR en clasificación 3.4, lo cual es un resultado esperable ya que se trata del problema *XOR*.

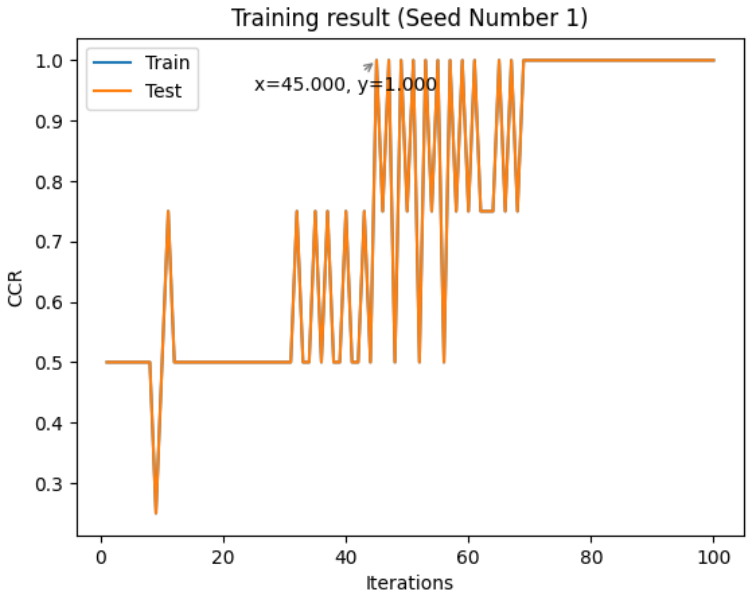


Figura 3.2: Resultado del entrenamiento en *CCR* del modelo optimizado en el dataset *XOR* con las 100 primeras iteraciones

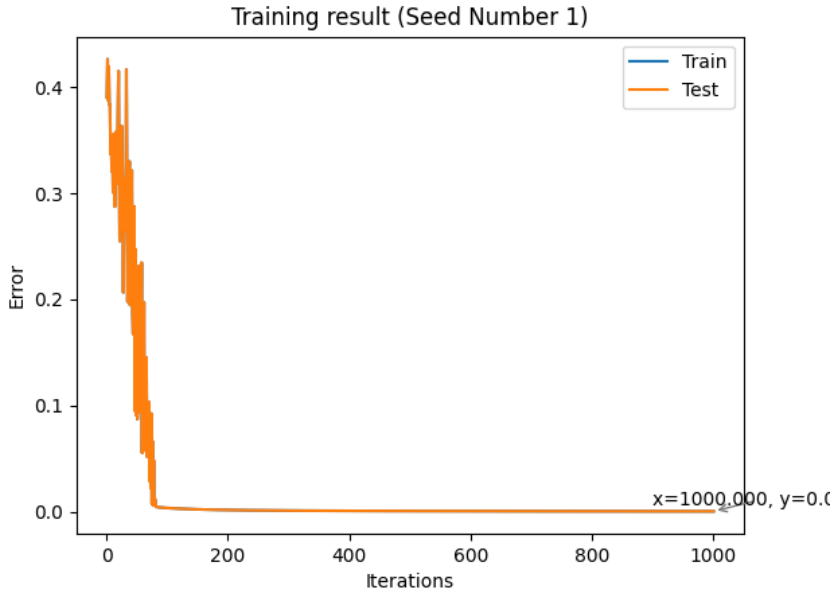


Figura 3.3: Resultado del entrenamiento en *MSE* del modelo optimizado en el dataset *XOR* con todas las iteraciones

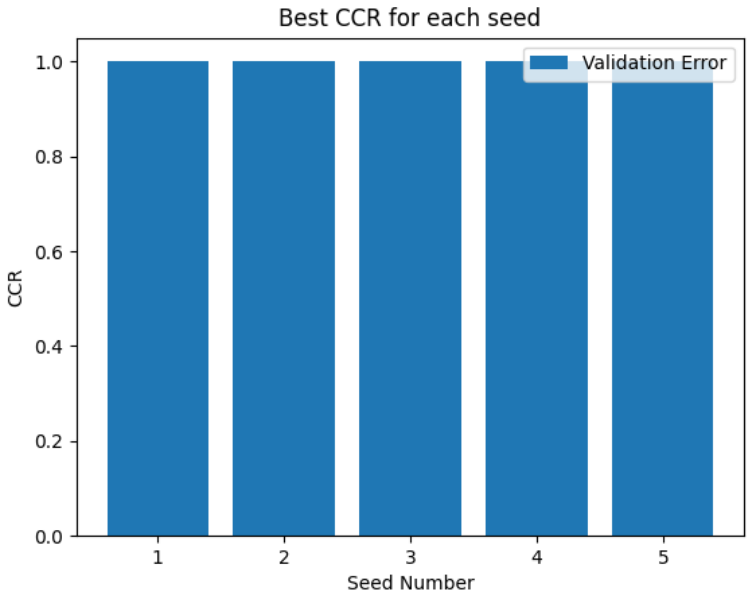


Figura 3.4: Resultados finales de *XOR* en *CCR* en las 5 semillas iniciales

Se puede apreciar en la gráfica 3.2 los valores tan fluctuantes presentes, y esto se debe a que el *CCR* solamente podrá tomar valores comprendidos entre 0, 0.25, 0.5 y 1, ya que solamente hay presente 4 patrones de entrenamiento. Por ello esta gráfica de convergencia tan característica.

Se puede apreciar en las gráfica, una indicación visual del mejor valor obtenido tanto en *CCR* como en *MSE* 3.2 3.3.

3.2 Dataset ILDP

3.2.1. Descripción del dataset

El dataset en cuestión trata sobre la clasificación de pacientes hepáticos o no. Hay presentes 441 registros de los cuales corresponden a hombres, mientras que 142 corresponden a mujeres. Se poseen un total de 405 patrones de entrenamiento y 174 de tests 3.5. Hay un total de 10 variables de entrada que incluyen:

1. Age: Edad del paciente.
2. TB: Bilirrubina total.
3. DB: Bilirrubina directa.
4. AAP: Fosfotasa Alcalina.
5. Sgpt: Alamina Aminotransferasa.
6. Sgot: Aspartato Aminotransferasa.
7. TP: Prótidos totales.
8. ALB: Albúmina.
9. AG Ratio: Ratio de albúmina y globulina.
10. Gender: género.

1	65.0	1.1	...	0	1
2	20.0	1.1	...	1	0
3	45.0	2.9	...	0	1
4	54.0	0.8	...	0	1
5	47.0	3.5	...	1	0

Tabla 3.5: Extracto del fichero *.dat* del dataset *ildp*

Para este modelo se requiere **normalizar los datos de entrada** para poder llevar a cabo el correcto entrenamiento. Para llevar a cabo el ajuste y comprobación del modelo, se ha realizado ***crossvalidation* con un 20 % del dataset de entrenamiento**.

3.2.2. Mejor topología encontrada

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 4 : k}	*True*	*softmax*	*Crossentropy*	0.624138 +- 0.027
{n : 8 : k}	*True*	*softmax*	*Crossentropy*	0.701149 +- 7.92e-05
{n : 16 : k}	*True*	*softmax*	*Crossentropy*	0.703448 +- 4.75e-05
{n : 64 : k}	*True*	*softmax*	*Crossentropy*	0.651724 +- 0.0121
{n : 4 : 4 : k}	***True***	***softmax***	***Crossentropy***	**0.706897 +- 0**
{n : 8 : 8 : k}	*True*	*softmax*	*Crossentropy*	0.706897 +- 0
{n : 16 : 16 : k}	*True*	*softmax*	*Crossentropy*	0.706897 +- 0
{n : 64 : 64 : k}	*True*	*softmax*	*Crossentropy*	0.637931 +- 0.016

Tabla 3.6: Resultados del experimento 1 en el dataset de *ildp*

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 4 : 4 : k}	*True*	*sigmoide*	*MSE*	0.706897 +- 0
{n : 4 : 4 : k}	***False***	***sigmoide***	***MSE***	**0.706897 +- 0**
{n : 4 : 4 : k}	*True*	*softmax*	*MSE*	0.706897 +- 0
{n : 4 : 4 : k}	*False*	*softmax*	*MSE*	0.705747 +- 5.28e-06
{n : 4 : 4 : k}	*True*	*softmax*	*Crossentropy*	0.706 +- 0
{n : 4 : 4 : k}	*False*	*softmax*	*Crossentropy*	0.622989 +- 0.027

Tabla 3.7: Resultados del experimento 2 en el dataset de *ild*

Se puede observar que de nuevo se tienen varios modelos que funcionan practicamente igual, por lo tanto se va a optar por el modelo de **2 capas ocultas y 4 neuronas con función de activación *softmax* y entropía cruzada como función de error**.

Para la optimización de los hiper-parámetros se tomará como punto de partida estas topologías acordes para cada problema, para intentar mejorar el resultado obtenido y, el propuesto por la práctica inicialmente.

3.2.3. Optimización de hiper-parámetros

Para llevar a cabo la optimización de hiper-parámetros se han establecido los siguientes valores:

- η: *linspace(0.1,1,10)*, el cual establece 10 valores comprendidos en el intervalo [0-1] donde todos los valores son equidistantes entre si.
- μ: *linspace(0.1,1,10)*, el cual establece 10 valores comprendidos en el intervalo [0-1] donde todos los valores son equidistantes entre si.
- *Learning rate Scheduler*: *None*, *exponential*, *step*, *cosine* y *linear*, son transformaciones que se aplican al learning rate en cada iteración, para que este no se mantengan constante en el tiempo [1], [2].
  - ***Exponential***: $\eta_i = \eta_i * 0{,}99$
  - ***Step***: $Si\ countTrain \mod 100\ es\ 0 \rightarrow \eta_i = \eta_i * 0{,}5$
  - ***Cosine***: $\eta_i = \eta_0 * 0{,}5 * (1 + \cos(\pi * \frac{countTrain}{maxiter}))$
  - ***Linear***: $\eta_i = \eta_0 * (1 - \frac{countTrain}{maxiter})$
- *Normalizing values*: $False$ y $True$, consiste en normalizar o no los datos de entrada.
- *Iterations*: 100, en este caso el problema no supone un gran coste computacional y es posible realizar la optimización ejecutando 1000 iteraciones del algoritmo por combinación.

Tras la optimización se han obtenido los siguientes hiper-parámetros 3.12.

Mejor eta	0.1
Mejor mu	0.1
Mejor Learning Rate Scheduler	*None*
Mejor Factor de Normalización	*True*
Arquitectura final	$\{10 : 4 : 4 : 2\}$

Tabla 3.8: Hiper-parámetros finales del dataset *ildp*

3.2.4. Modelo final

El modelo resultante ha obtenido un total de **0.706897 +- 0 en *CCR* y 0.315427 +- 5.37218e-05 de error en entropía cruzada** en *tests*. Se puede apreciar que no existe mejora aparente, ya que este parte desde un punto inicial bastante favorable, y no consige converger con la arquitectura propuesta de ninguna manera. El error decrece de manera sutil pero el *CCR* permanece invariante 3.5 3.6.

Esto se puede deber principalmente a las características del dataset, el cual tiene ciertas divisiones que dificultan bastante la clasificación, además de ser bastante reducido en cuanto a muestras de entrenamiento para la complejidad que se requiere del mismo. Pese a la optimización realizada ha sido muy complejo ajustar los hiper-parámetros.

Para las 5 semillas el mejor valor alcanzado en *CCR* sigue permaneciendo similar entre todas las propuestas 3.7.

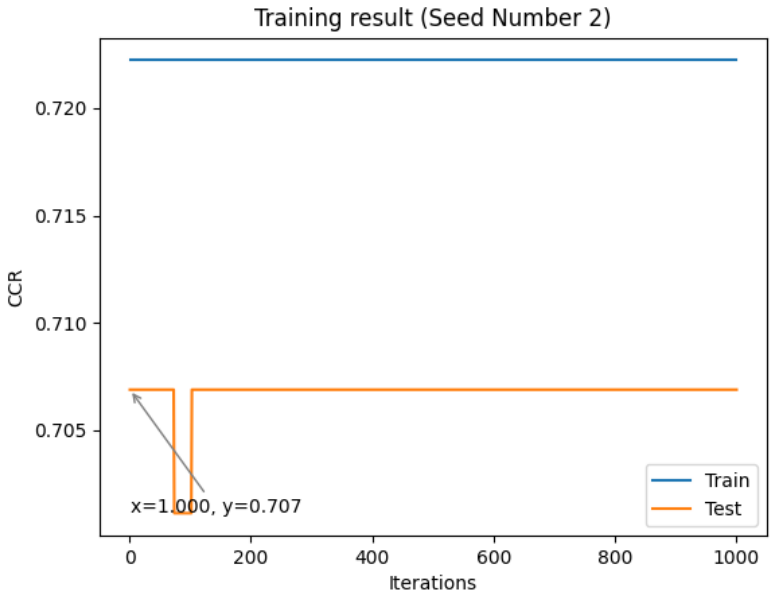


Figura 3.5: Resultado del entrenamiento en *CCR* del modelo optimizado en el dataset *ildp*

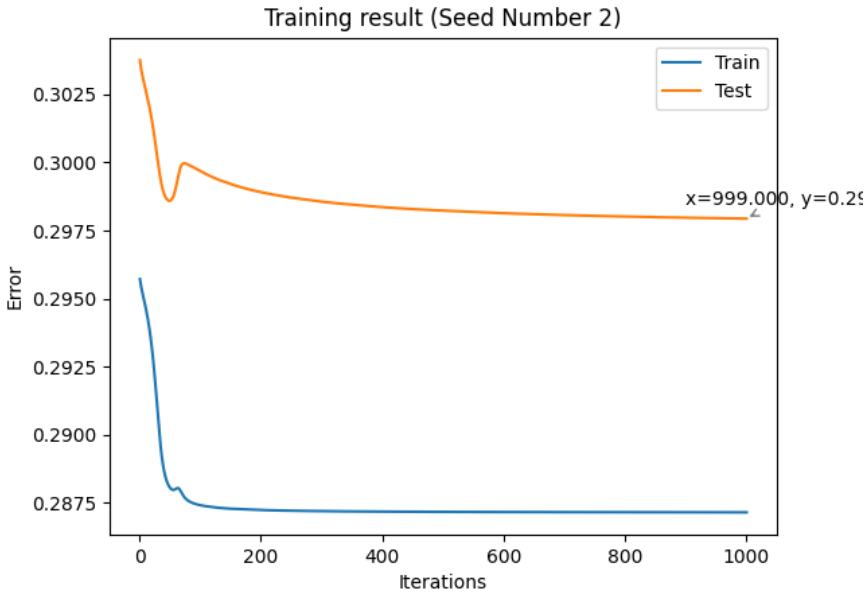


Figura 3.6: Resultado del entrenamiento en entropía cruzada del modelo optimizado en el dataset *ildp*

Se puede apreciar en las gráficas propuestas, el mejor valor encontrado tanto para *CCR* como para la entropía cruzada 3.5 3.6.

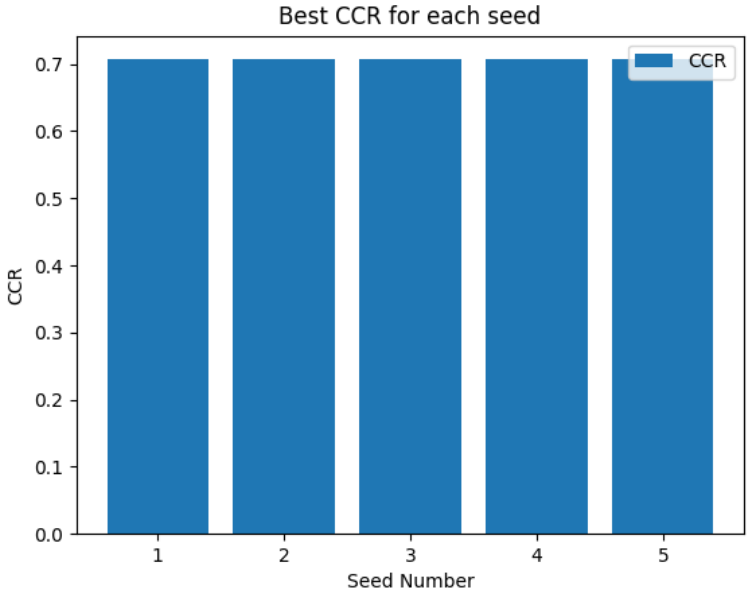


Figura 3.7: Resultados finales de *ildp* en *CCR* en las 5 semillas iniciales

3.3 Dataset noMNIST

3.3.1. Descripción del dataset

El dataset de *noMNIST* es un dataset enfocado a la clasificación de letras, siendo esta una tarea clásica de **visión por computador**. Para la práctica propuesta, se propone una versión reducida de 900 patrones de entrenamiento y 300 patrones de test. Dado que el código está adaptado a recibir entradas mediante ficheros planos *.dat*, se están pasando como variables de entrada todos los valores de cada píxel de los 28×28 totales presente que hay, siendo un **dataset bastante grande**. Se pretenden clasificar **6 letras (*a, b, c, d y f*)**, poseyendo así un total de 6 clases 3.9 3.8.

1	-1	-1	...	1	...
2	-0.96	-0.027	...	0	...
3	-0.09	-1	...	1	...
4	-0.08	0	...	0	...
5	0.55	-1	...	0	...

Tabla 3.9: Extracto del fichero *.dat* del dataset *noMnist*

Figura 3.8: Extracto de los patrones del dataset de *NoMnist*

Todos los valores de entrada del modelo se encuentra ya normalizados. Para llevar a cabo el ajuste y comprobación del modelo, se ha realizado ***crossvalidation* con un 20 % del dataset de entrenamiento**.

3.3.2. Mejor topología encontrada

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 4 : k}	*True*	*softmax*	*Crossentropy*	0.91 +- 1.66e-6
{n : 8 : k}	*True*	*softmax*	*Crossentropy*	0.942 +- 2.46-e3
{n : 16 : k}	*True*	*softmax*	*Crossentropy*	0.944 +- 1.53e-4
{n : 64 : k}	*True*	*softmax*	*Crossentropy*	0.94 +- 8.44e-05
{n : 4 : 4 : k}	*True*	*softmax*	*Crossentropy*	0.887 +- 2e-3
{n : 8 : 8 : k}	*True*	*softmax*	*Crossentropy*	0.918 +- 2.6e-3
{n : 16 : 16 : k}	*True*	*softmax*	*Crossentropy*	0.945 +- 1.36e-3
{n : 64 : 64 : k}	***True***	***softmax***	***Crossentropy***	**0.951 +- 7.38e-05**

Tabla 3.10: Resultados del experimento 1 en el dataset de *NoMnist*

Arquitectura	***offline***	**F. activación**	**F. Error**	**CCR en Tests**
{n : 64 : 64 : k}	*True*	*sigmoide*	*MSE*	0.947 +- 4.178e-05
{n : 64 : 64 : k}	*False*	*sigmoide*	*MSE*	0.903 +- 0.002
{n : 64 : 64 : k}	*True*	*softmax*	*MSE*	0.942 +- 1.95e-05
{n : 64 : 64 : k}	*False*	*softmax*	*MSE*	0.93 +- 8.8e-4
{n : 64 : 64 : k}	***True***	***softmax***	***Crossentropy***	**0.952 +- 7.37e-05**
{n : 64 : 64 : k}	*False*	*softmax*	*Crossentropy*	0.857 +- 0.001

Tabla 3.11: Resultados del experimento 2 en el dataset de *NoMnist*

Se puede observar que el mejor modelo para el experimento 3.10, es aquel con **2 capas ocultas y 64 neuronas por capa oculta**. Finalmente tras realizar el experimento 3.11 se obtiene un modelo mejor con **modo *offline*, función de activación *softmax* y entropía cruzada como función de error**.

Para la optimización de los hiper-parámetros se tomará como punto de partida estas topologías acordes para cada problema, para intentar mejorar el resultado obtenido y, el propuesto por la práctica inicialmente.

3.3.3. Optimización de hiper-parámetros

Para llevar a cabo la optimización de hiper-parámetros se han establecido los siguientes valores:

- η: *linspace(0.1,1,10)*, el cual establece 10 valores comprendidos en el intervalo [0-1] donde todos los valores son equidistantes entre si.
- μ: *linspace(0.1,1,10)*, el cual establece 10 valores comprendidos en el intervalo [0-1] donde todos los valores son equidistantes entre si.
- *Learning rate Scheduler*: *None*, *exponential*, *step*, *cosine* y *linear*, son transformaciones que se aplican al learning rate en cada iteración, para que este no se mantengan constante en el tiempo [1], [2].
  - ***Exponential***: $\eta_i = \eta_i * 0{,}99$
  - ***Step***: $Si\ countTrain\ \ \text{mód}\ 100\ es\ 0 \rightarrow \eta_i = \eta_i * 0{,}5$
  - ***Cosine***: $\eta_i = \eta_0 * 0{,}5 * (1 + \cos(\pi * \frac{countTrain}{maxiter}))$
  - ***Linear***: $\eta_i = \eta_0 * (1 - \frac{countTrain}{maxiter})$
- *Normalizing values*: *False* y *True*, consiste en normalizar o no los datos de entrada.
- *Iterations*: 10, en este caso el problema supone un gran coste computacional, por lo que se ha reducido el número de iteraciones considerablemente.

Tras la optimización se han obtenido los siguientes hiper-parámetros 3.12.

Mejor eta	0.1
Mejor mu	0.3
Mejor Learning Rate Scheduler	*Exponential*
Mejor Factor de Normalización	*False*
Arquitectura final	$\{784 : 64 : 64 : 6\}$

Tabla 3.12: Hiper-parámetros finales del dataset *NoMNIST*

3.3.4. Modelo final

Con la optimización realizada, se ha obtenido un valor de **0.957 +- 7.38e-05 en *CCR* y 0.115022 +- 0.0001 como error de entropía cruzada** en *tests*. Para la cómoda visualización de las gráficas se han reducido a las primeras 100 épocas de entrenamiento para observar los mejores valores obtenidos en *CCR* y entropía cruzada 3.10 3.11. La semilla que ha obtenido mejor valor en *CCR* ha sido la primera, siendo la cual la que se ha mostrado en las gráficas de convergencia resultantes.

Se pueden observar fuertes oscilaciones en las primeras épocas respecto al *CCR* 3.10, pero a partir de la época 20 el modelo empieza a converger de manera óptima, llegando a alcanzar su mejor valor en la **época 77**. Este se puede deber principalmente a que al ser una red con 784 neuronas de entrada, en las primeras épocas el propio aprendizaje del modelo es realmente complejo dado el gran ajuste de pesos que se debe realizar, ya que este tipo de tareas suelen dejarse como se ha mencionado previamente, a modelos de *deep learning*. Por tanto, es normal esta tendencia y la posterior mejora exponencial tras la ejecución de las primeras 20 épocas, ya que se poseen bastantes pesos que ajustar de la capa de entrada a la primera capa oculta 3.9, y dado el valor de η tan reducido, se ve lógica esta mejora a partir de un mínimo de épocas.

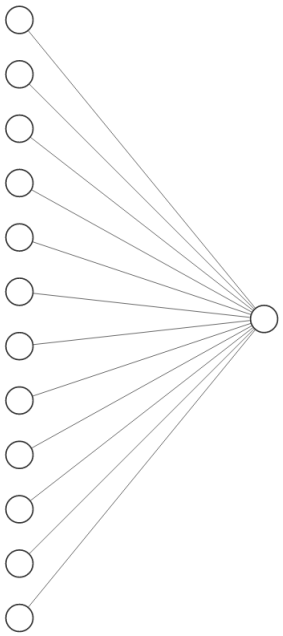


Figura 3.9: Grafo comparativo de 784 neuronas de entrada a 64 ocultas (entre 64), con 12 neuronas de entrada 1 oculta

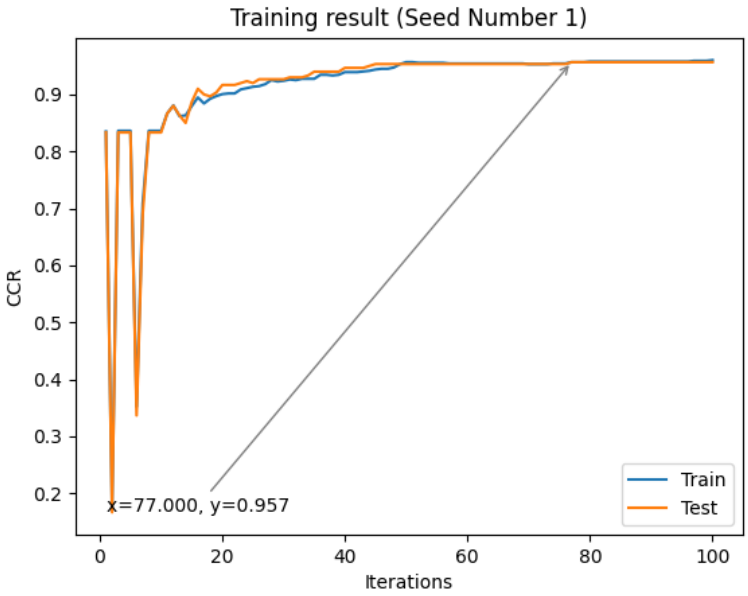


Figura 3.10: Resultado del entrenamiento en *CCR* del modelo optimizado en el dataset *NoMNIST*

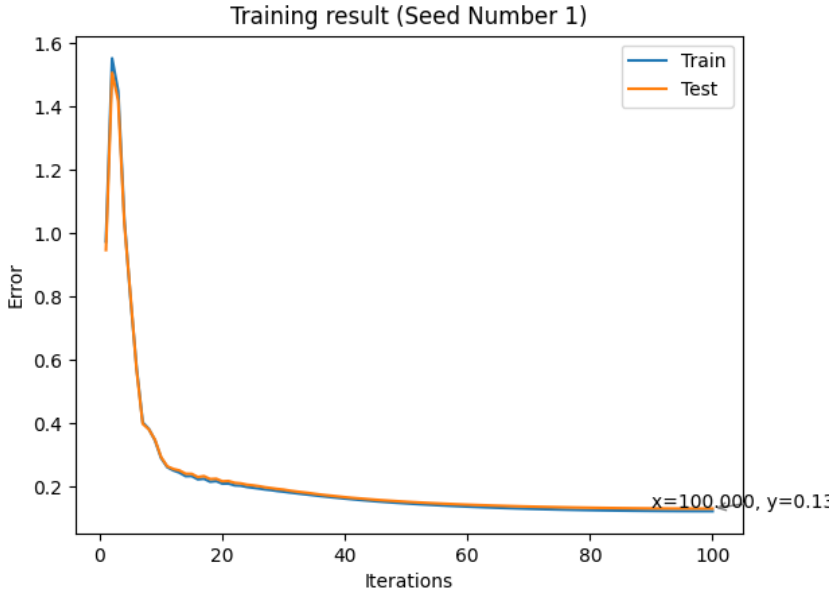


Figura 3.11: Resultado del entrenamiento en entropía cruzada del modelo optimizado en el dataset *NoMNIST*

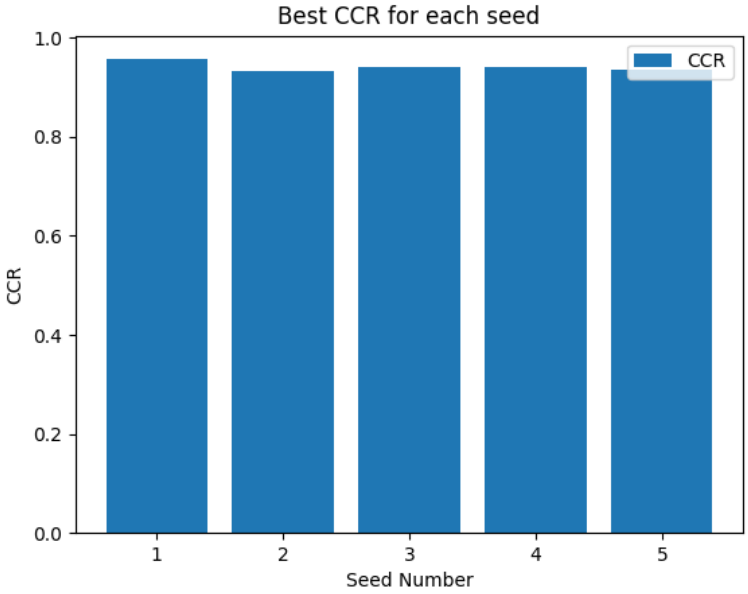


Figura 3.12: Resultados finales de *NoMNIST* en *CCR* en las 5 semillas iniciales

3.3.5. Predicciones finales

Se ha generado la matriz de confusión final de las predicciones realizadas en los *tests* finales 3.13.

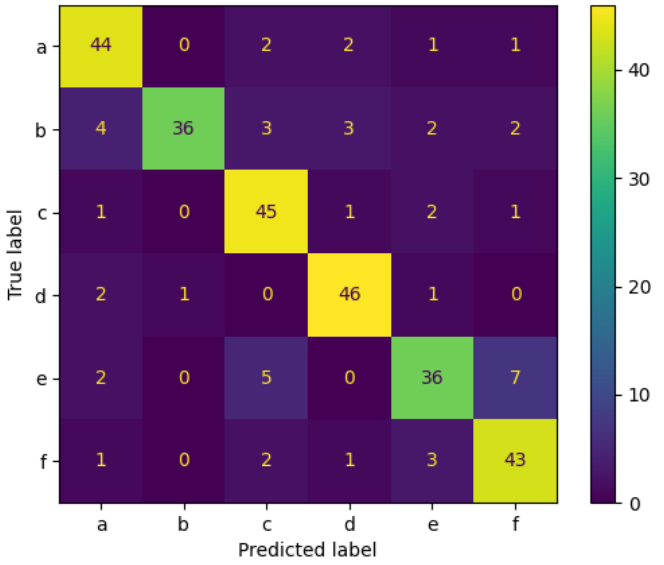
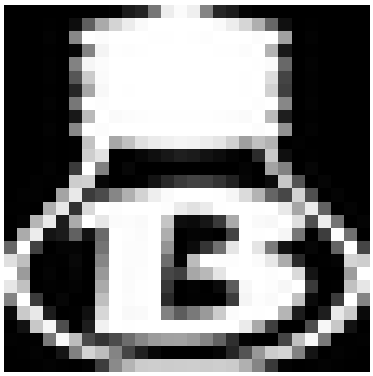


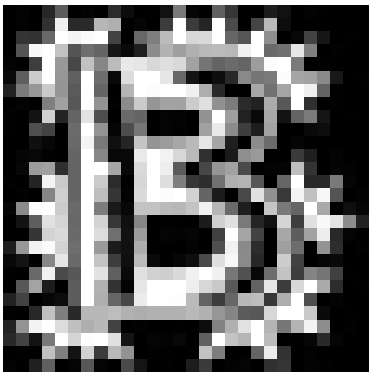
Figura 3.13: Matriz de confusión en *NoMNIST* en el dataset de *tests*

Se puede observar que el modelo clasifica correctamente la mayoría de patrones, sin embargo tiene ciertas dificultades en algunos que son un poco confusos.

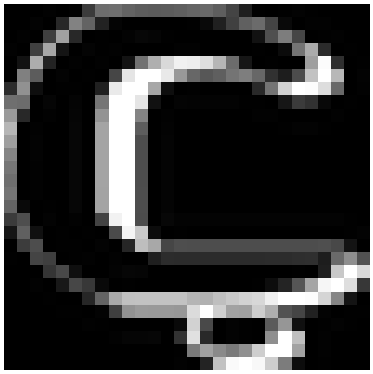
Se puede observar en las figuras 3.14 como las imágenes en las que se equivoca el modelo son ciertamente confusas, no asimilándose a los patrones de entrenamiento dados. Además, hay que tener en cuenta que este modelo se basa en aprendizaje **píxel por píxel, no siendo posible que adquiera aprendizaje mediante la vecindad de píxeles y pudiendo así adquirir capacidad de localización espacial, como es el caso de las redes convolucionales, provocando así estos fallos en este tipo de patrones más "confusos"**.



(a) Predicción de una E respecto a una B



(b) Predicción de una E respecto a una B



(c) Predicción de una E respecto a una C

(d) Predicción de una B respecto a una F

Figura 3.14: Predicciones resultantes fallidas sobre el dataset de *tests*

3.4 Arquitecturas y resultados finales

Para simplificar la lectura se elaborará una tabla final indicando las topologías finales y los resultados obtenidos para cada dataset en cuestión 3.13

Dataset	**Arquitectura**	**Media obtenida en tests**	**Media a superar**
XOR	$\{2:100:100:2\}$	1 +- 0	0.5
ILDP	$\{10:4:4:2\}$	0.706897 +- 0	0.6897
NoMNIST	$\{784:64:64:6\}$	0.957 +- 7.38e-05	0.8267

Tabla 3.13: Topologías finales para cada dataset

Las ganancias totales en porcentaje, en cuanto a mejoras de *CCR* respecto a lo propuesto, serían las siguientes:

- *XOR*: $200\,\%$
- *ILDP*: $102,49\,\%$
- *NoMNIST*: $115,76\,\%$

Bibliografía

[1] Suki Lau. *Learning rate schedules and adaptive learning rate methods for deep learning*. Ago. de 2017. URL: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1 (vid. págs. 16, 22).

[2] Katherine (Yi) Li. *How to choose a Learning Rate Scheduler for Neural Networks*. Jul. de 2022. URL: https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler (vid. págs. 16, 22).

Apéndice A
Código adicional implementado al esqueleto

A.1 Nuevos argumentos de entrada vía *CLI*

Los argumentos agregados son los siguientes:

- *S <Scheduler type>* (*Scheduler argument*): Se especificar el tipo de learning rate scheduler a aplicar al programa. Se han considerado los siguientes:
  - ***Exponential***: $\eta_i = \eta_i * 0{,}99$
  - ***Step***: $Si\ countTrain\ \ \text{mód}\ 100\ es\ 0 \rightarrow \eta_i = \eta_i * 0{,}5$
  - ***Cosine***: $\eta_i = \eta_0 * 0{,}5 * (1 + \cos(\pi * \frac{countTrain}{maxiter}))$
  - ***Linear***: $\eta_i = \eta_0 * (1 - \frac{countTrain}{maxiter})$
  - ***None***: Valor por defecto, el learning rate se mantiene constante.
- *v* (*Verbose mode*): Booleano para indicar que queremos guardar en un fichero *.txt* los datos obtenidos del experimento, guardando el *MSE* medio obtenido en el entrenamiento y tests, además del número totales de iteraciones, error en cada iteración del entrenamiento, y el error actual tras cada iteración en tests (gracias a esto se han podido llevar a cabo las gráficas de convergencia de crossvalidation).
- *V <Splitting value>* (*Validation split*): flotante al cual se le indica el porcentaje del dataset de entrenamiento del cual queremos hacer el split (por defecto toma el valor de 0,2) para llevar a cabo el *crossvalidation*.
- *c <print current CCR training and validation>*: Booleano para indicar si queremos mostrar el *CCR* actual en cada iteración en el dataset de *training* y *validation*, necesario para llevar a cabo las gráficas de convergencia del modelo.

A.2 Implementación de la selección de los *schedulers*

```
if (learningRateScheduler && strcmp(learningRateSchedulerTpe,
↪  "None") != 0) {
    if (strcmp(learningRateSchedulerTpe, "exponential") == 0)
        eta = eta * 0.99;

    else if (strcmp(learningRateSchedulerTpe, "step") == 0) {
        if (countTrain % 100 == 0)
            eta = eta * 0.5;
    } else if (strcmp(learningRateSchedulerTpe, "cosine") == 0)
        eta = oldEta * 0.5 * (1 + cos(M_PI * countTrain /
↪  maxiter));

    else if (strcmp(learningRateSchedulerTpe, "linear") == 0)
        eta = oldEta * (float)(1 - (float)countTrain /
↪  (float)maxiter);

    else
        throw std::invalid_argument("The learning rate scheduler
↪  type is not valid");
}

std::cout << "Current learning rate --> " << eta << std::endl;
```

A.3 Implementación del modo *verbose* para el *cross-validation*

```
if (verbose) {
    trainErrorVector.push_back(trainError);
    testErrorVector.push_back(test(validationDataset));
}
```

A.4 Implementación del *splitting* del dataset de entrenamiento para el *crossvalidation*

```
Dataset *util::splitDataset(Dataset *trainDataset, float
↪  validationSplit, int seed = 0) {
    srand(seed);

    int validationSize = (int) (trainDataset->nOfPatterns *
↪  validationSplit);
    Dataset *validationDataset = (Dataset *) calloc(1,
↪  sizeof(Dataset));

    validationDataset->nOfInputs = trainDataset->nOfInputs;
    validationDataset->nOfOutputs = trainDataset->nOfOutputs;
    validationDataset->nOfPatterns = validationSize;

    validationDataset->inputs = (double **) calloc(validationSize,
↪  sizeof(double *));
    validationDataset->outputs = (double **)
↪  calloc(validationSize, sizeof(double *));

    for (int i = 0; i < validationSize; i++) {
        validationDataset->inputs[i] = (double *)
↪  calloc(trainDataset->nOfInputs, sizeof(double));
        validationDataset->outputs[i] = (double *)
↪  calloc(trainDataset->nOfOutputs, sizeof(double));
    }
```

```
// We shuffle the dataset given a seed
    for (int i = 0; i < validationSize; i++) {

        // The seed to use is given in la1.cpp
        int index = randomInt(0, trainDataset->nOfPatterns - 1);

        for (int j = 0; j < trainDataset->nOfInputs; j++) {
            validationDataset->inputs[i][j] =
↪   trainDataset->inputs[index][j];
        }

        for (int j = 0; j < trainDataset->nOfOutputs; j++) {
            validationDataset->outputs[i][j] =
↪   trainDataset->outputs[index][j];
        }

        free(trainDataset->inputs[index]); // Deleting the pattern
↪   i-th
        free(trainDataset->outputs[index]); // Deleting the
↪   pattern i-th
        trainDataset->nOfPatterns -= 1;

        for (int k = index; k < trainDataset->nOfPatterns; k++) {
            trainDataset->inputs[k] = trainDataset->inputs[k + 1];
            trainDataset->outputs[k] = trainDataset->outputs[k +
↪   1];
        }

    }

    return validationDataset;
}
```

A.5 Implementación del modo *verbose*

```
// Create a ofstream file
ofstream myfile;
std::string filename = std::string(verboseStr) + "-i:" +
↪  std::to_string(iterations) + "-l:" +
   std::to_string(hiddenLayers) + "-h:" +
   std::to_string(hiddenLayersNeurons) + "-e:" +
↪  std::to_string(eta) +
   "-m:" + std::to_string(mu) + ".txt";
   std::string pathToSave = "./experiments/" + filename;

   myfile.open(pathToSave);
   myfile << "FINAL REPORT" << endl;
   myfile << "************" << endl;

   myfile << "Train error (Mean +- SD): " << averageTrainError <<
↪  " +- " << stdTrainError << endl;
   myfile << "Test error (Mean +- SD):  " << averageTestError <<
↪  " +- " << stdTestError << endl;

   myfile << "CONVERGENCE DATA" << endl;
   myfile << "****************" << endl;

   myfile << "\n\nIterations: \n";
   for (int i = 1; i <= iterations; i++) {
       myfile << i << ",\n";
   }

   myfile << "\n\nTrain errors: \n";
   for (int i = 0; i < iterations; i++) {
       myfile << trainErrorsVector[i] << ",\n";
   }

   if (Vflag) {
       myfile << "\n\nTest errors: \n";
       for (int i = 0; i < iterations; i++) {
           myfile << testErrorsVector[i] << ",\n";
       }
   }
```

Apéndice B
Código adicional para la ejecución de los experimentos

B.1 Script de Python para la selección de la mejor arquitectura e hiper-parámetros

```
train_dataset_path = '../../datasetsPr1IMC/dat/train_xor.dat'
test_dataset_path = '../../datasetsPr1IMC/dat/test_xor.dat'
experiment_name = 'prueba_optimized_dataset'

learning_rates = np.linspace(0.1, 1, 10)
momentums = np.linspace(0.1, 1, 10)
lr_schedulers = np.array(["None", "exponential", "step", "cosine",
↪  "linear"])
normalizing_values = np.array([False, True])

hidden_layers = np.array([2])
neurons_per_layer = np.array([100])
iterations = 10

best_test_error = sys.float_info.max

total = len(learning_rates) * len(momentums) * len(lr_schedulers)
↪  * len(hidden_layers) * len(neurons_per_layer) *
↪  len(normalizing_values)
bar_length = 30

i = int(1)
```

```
for learning_rate in learning_rates:
    for momentum in momentums:
        for lr_scheduler in lr_schedulers:
            for hidden_layer in hidden_layers:
                for neuron in neurons_per_layer:
                    for normalizing_value in normalizing_values:

                        if normalizing_value:
                            os.system(
                                f"./la1 -t {train_dataset_path} -T
                                ↪ {test_dataset_path} -l
                                ↪ {hidden_layer} -h {neuron} -i
                                ↪ {iterations} -e
                                ↪ {learning_rate} -m {momentum}
                                ↪ -S {lr_scheduler} -s | grep
                                ↪ \"Test error\" | grep -oP
                                ↪ \'(?<=:).*\' | grep -Eo \".*
                                ↪ \" | tr -d \'-\' | tr -d \'+\'
                                ↪ > .trash.out")
                        else:
                            os.system(
                                f"./la1 -t {train_dataset_path} -T
                                ↪ {test_dataset_path} -l
                                ↪ {hidden_layer} -h {neuron} -i
                                ↪ {iterations} -e
                                ↪ {learning_rate} -m {momentum}
                                ↪ -S {lr_scheduler} | grep
                                ↪ \"Test error\" | grep -oP
                                ↪ \'(?<=:).*\' | grep -Eo \".*
                                ↪ \" | tr -d \'-\' | tr -d \'+\'
                                ↪ > .trash.out")
```

```
with open('.trash.out', 'r') as f:
    current_test_error = float(f.read())

    if current_test_error < best_test_error:
        print(f"\t>>> New best test error: {current_test_error} vs
        ↪  {best_test_error}")
        best_test_error = current_test_error

        best_learning_rate = learning_rate
        best_momentum = momentum
        best_lr_scheduler = lr_scheduler
        best_hidden_layer = hidden_layer
        best_neuron = neuron
        best_normalizing_value = normalizing_value

    else:
        print(f"\t>>> Current test error: {current_test_error} vs
        ↪  {best_test_error}")

        percent = 100.0 * i / total
        sys.stdout.write('\r')
        sys.stdout.write("Completed: [{:{}}] {:>3}%\n".format('='
        ↪  * int(percent / (100.0 / bar_length)),



        sys.stdout.flush()

    i += 1
```

```
if best_normalizing_value:
    os.system(
        f"./la1 -t {train_dataset_path} -T {test_dataset_path} -l
        ↪ {best_hidden_layer} -h {best_neuron} -i {iterations}
        ↪ -e {best_learning_rate} -m {best_momentum} -S
        ↪ {best_lr_scheduler} -v {experiment_name} -s")

else:
    os.system(
        f"./la1 -t {train_dataset_path} -T {test_dataset_path} -l
        ↪ {best_hidden_layer} -h {best_neuron} -i {iterations}
        ↪ -e {best_learning_rate} -m {best_momentum} -S
        ↪ {best_lr_scheduler} -v {experiment_name}")

with open(f'{experiment_name}_best_params.out', 'w') as f:
    f.write(f"Best learning rate: {best_learning_rate}\n")
    f.write(f"Best momentum: {best_momentum}\n")
    f.write(f"Best lr scheduler: {best_lr_scheduler}\n")
    f.write(f"Best hidden layer: {best_hidden_layer}\n")
    f.write(f"Best neuron: {best_neuron}\n")
    f.write(f"Best normalizing value: {best_normalizing_value}\n")

os.system(f"rm .trash.out")
```

B.1.1. Ejemplo de salida tras la ejecución del Script

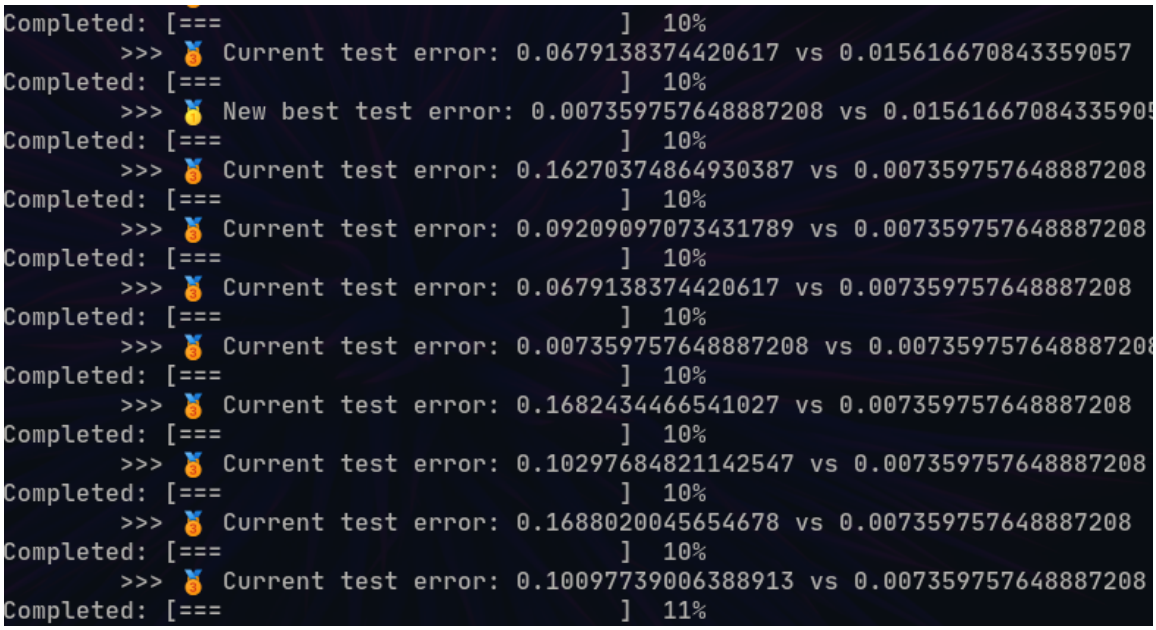


Figura B.1: Ejemplo de salida tras ejecutar el script *execute_optimization*

B.2 Script para la creación de gráficas del *CCR*

```
path = sys.argv[1]
with open(path, 'r') as f:
    # read every line
    lines = f.readlines()
    seeds_count = 1
    seeds = []
    training_error = []
    validation_error = []
    iterations = [1]
    lines = [line.strip() for line in lines]
    lines = [line for line in lines if line]
```

```
for index, line in enumerate(lines):

    # Find the line that contains the iterations
    if line.startswith('SEED'):
        index += 2

        while index < len(lines):
            if lines[index].startswith('NETWORK'):
                # The i-th seed is finished
                iterations.pop()

                # Plot validation and training
                plt.clf()
                annot_max(np.array(iterations), np.array(ccr),
                ↪  ax=None)
                plt.plot(iterations, training_error,
                ↪  label='Train')
                plt.plot(iterations, validation_error,
                ↪  label='Test')
                plt.title(f'Training result (Seed Number
                ↪  {seeds_count})')
                plt.xlabel('Iterations')
                plt.ylabel('CCR')
                plt.legend()
                plt.show()
```

```
# Adding to seed the best validation error with the corresponding
↪  iteration
    seeds.append((seeds_count, iterations[np.argmax(ccr)],
                  ccr[np.argmax(ccr)]))

    training_error.clear()
    validation_error.clear()
    iterations.clear()
    iterations.append(1)
    seeds_count += 1

    break

new_line = re.split("Training CCR: (.*) Validation CCR: (.*)",
↪  lines[index])
```

```
if len(new_line) == 4:
    new_line[1] = new_line[1].replace('|', '')

    training_error.append(float(new_line[1]))
    validation_error.append(float(new_line[2]))
    iterations.append(iterations[-1] + 1)

print(lines[index])

index += 1
```

```
# Plot the best validation error for each seed in her
↪  corresponding iteration
plt.clf()
plt.bar([seed[0] for seed in seeds], [seed[2] for seed in seeds],
↪  label='CCR')
plt.title('Best CCR for each seed')
plt.xlabel('Seed Number')
plt.ylabel('CCR')
plt.legend()
plt.show()
```

B.3 Script para la creación de la matriz de confusión

```
with open('output.txt', 'r') as f:
    lines = f.readlines()
    lines = [line.strip() for line in lines]
    lines = [line for line in lines if line]
    classes = ['a', 'b', 'c', 'd', 'e', 'f']
    y_pred = []
    y_actual = []
    index_image = 0
```

```
for line in lines:
    # Split the line for every whitespace
```

```
    line = re.split(r'\s+', line)
    # Remove empty strings
    line = [x for x in line if x]

    # Remove all elements that are '--'
    line = [x for x in line if x != '--']

    # Convert all values into numerical type
    line = [float(x) for x in line]

    # Create a new array with all values that are integers
    expected_classes = [x for x in line if x.is_integer()]
```

```
# Create a new array with all values that are floats
predicted_classes = [x for x in line if not x.is_integer()]

# Get the index of the highest value in the predicted_classes
↪  array
predicted_class = predicted_classes.index(max(predicted_classes))

# Get the index of the highest value in the expected_classes
↪  array
expected_class = expected_classes.index(max(expected_classes))

print(
    f"Image {index_image} is predicted as
    ↪  {classes[predicted_class]} and is actually
    ↪  {classes[expected_class]}")
index_image += 1

y_pred.append(classes[predicted_class])
y_actual.append(classes[expected_class])
```

```
# Create a confusion matrix
confusion_matrix = metrics.confusion_matrix(y_actual, y_pred)
cm_display =
↪  metrics.ConfusionMatrixDisplay(confusion_matrix=confusion_matrix,
↪  display_labels=classes)
cm_display.plot()
plt.show()
```