



REDES NEURONALES

· Trabajo para la asignatura de Inteligencia Artificial ·
· Ingeniería del Software, Curso 2015-2016 ·

Realizado por los alumnos:

Juan Jesús Pérez Luna

Gonzalo Delgado Chaves

Índice

Índice general

| | |
|--|-----------|
| 1 Introducción | 3 |
| 1.1 Consideraciones iniciales | 3 |
| 1.2 Metodología y herramientas de trabajo | 4 |
| 2 Soluciones y mejoras desarrolladas..... | 5 |
| 2.1 Diseño directo píxel a píxel | 5 |
| 2.1.1 Descripción general | 5 |
| 2.1.2 Configuración y ejecución del algoritmo | 5 |
| 2.1.3 Parámetros pseudo-configurables | 11 |
| 2.1.4 Detalles de implementación | 12 |
| 2.2 Diseño con “compresión” virtual de imagen | 13 |
| 2.2.1 Descripción general | 13 |
| 2.2.2 Configuración y ejecución del algoritmo | 14 |
| 2.2.3 Parámetros pseudo-configurables | 14 |
| 2.2.4 Detalles de implementación | 15 |
| 2.3 Diseño con “momentum” del gradiente | 15 |
| 2.3.1 Descripción general | 15 |
| 2.3.2 Configuración y ejecución del algoritmo | 16 |
| 2.3.3 Parámetros pseudo-configurables | 17 |
| 2.2.4 Detalles de implementación | 17 |
| 3 Benchmarking de los algoritmos | 18 |
| 3.1 Consideraciones iniciales y metodología | 18 |
| 3.1.1 Introducción..... | 18 |
| 3.1.2 Entorno de pruebas | 18 |
| 3.1.3 Metodología | 18 |
| 3.2 Resultados del proceso de benchmarking | 18 |
| 3.2.1 Comparativa entre algoritmos | 18 |
| 3.2.2 Conclusiones | 20 |
| 3.2.3 Curiosidades | 20 |
| 4 Posibles mejoras planteadas (no implementadas)..... | 21 |
| 4.1 Otro tipo de “compresión”, convoluciones, etc..... | 21 |
| 4.2 Preprocesado de imágenes con “visión por computador” | 21 |
| 4.3 Programación concurrente | 21 |
| 5 Material de consulta y enlaces de interés..... | 22 |
| 5.1 Introducción | 22 |
| 5.2 Material y referencias..... | 22 |
| 5.2.1 Material de teoría sobre redes neuronales..... | 22 |
| 5.2.2 Material sobre programación en python | 22 |
| 5.2.3 Otros materiales de consulta | 22 |

Índice de figuras

| | |
|-----------------|----|
| Figura 1 | 3 |
| Figura 2 | 6 |
| Figura 3 | 6 |
| Figura 4 | 7 |
| Figura 5 | 7 |
| Figura 6 | 8 |
| Figura 7 | 9 |
| Figura 8 | 9 |
| Figura 9 | 10 |
| Figura 10 | 10 |
| Figura 11 | 11 |
| Figura 12 | 11 |
| Figura 13 | 12 |
| Figura 14 | 14 |
| Figura 15 | 15 |
| Figura 16 | 15 |
| Figura 17 | 16 |
| Figura 18 | 19 |

1. Introducción

1.1 Consideraciones iniciales:

Una vez estudiado el enunciado del problema propuesto sobre redes neuronales, y basándonos en el material visto en clase (más información en el apéndice de recursos utilizados) planteamos a priori un conjunto de diversas soluciones para el problema, de las cuales a su vez, ofreceremos los resultados de distintas parametrizaciones para cada uno de ellos.

Antes de ofrecer estas soluciones, consideramos importante señalar, que hemos tomado algunas consideraciones iniciales para especificar más la información ofrecida por el enunciado, y son las que enunciamos a continuación:

- El conjunto de entrenamiento sólo estará formado por un subconjunto de 7 letras minúsculas del abecedario básico español, sin incluir números ni símbolos. Éstas han sido elegidas por nuestro tutor del trabajo, y son las siguientes:

d f h j l n p

- Las imágenes del conjunto de entrenamiento, así como las del conjunto de prueba posterior, estarán binarizadas de origen, es decir, aunque el formato PGM admita 255 distintos niveles de gris, trabajaremos con originales que solo posean píxeles con valor 0 o 255. No obstante, en todas nuestras implementaciones, si la imagen no estuviese binarizada, todo aquello diferente del valor '255' lo convertirá a '0' automáticamente.

- Estas imágenes anteriormente mencionadas, han sido creadas utilizando la herramienta *GIMP 2* (Ver apartado de recursos utilizados), de forma manuscrita utilizando una tableta digitalizadora, siendo exportadas posteriormente a formato '.pgm', hasta un total de 70 para el conjunto de entrenamiento, y otras 70 para el conjunto prueba. A continuación un ejemplo de un conjunto de 7 letras arbitrario de todos esos conjuntos:



Figura 1

- Todas las imágenes, tal como se plantea en el enunciado miden 30x30 píxeles (900 píxeles en total).

- Los conjuntos de entrenamiento y prueba, están separados en directorios diferentes en cada uno de los algoritmos aportados para el trabajo.

- Dado que el tamaño de las imágenes y su proporción de imagen es fija para el ejercicio planteado, así como lo es el subconjunto de letras a identificar, el tamaño de las capas de entrada y salida para la red neuronal, no está contemplado para ser configurable (a priori) salvo escribiendo en el fichero .py correspondiente, como se verá más adelante.
- La introducción de datos podrá ser manual mediante una interfaz tipo consola, o puede ser cargada mediante un fichero de configuración.
- Debido al gran tamaño de la red, hemos limitado la entrada de datos, referidas al total o por capas, para valores de los bias, pesos de las conexiones, etc. por comodidad del usuario a la hora de configurar y ejecutar el algoritmo (Más adelante se ofrecen los detalles).
- La información de salida se muestra por pantalla mediante interfaz de texto tipo consola, siendo opcional el volcado del estado final de la red (bias, pesos, etc.) en un fichero .dat que se creará o sobrescribirá según el estado inicial del mismo.
- Todos los algoritmos realizados para este trabajo han sido creados desde cero en su totalidad, no utilizando herramientas externas como pudiera ser la vista durante la práctica correspondiente de la asignatura.
- Se adjunta un fichero ejecutable '.bat' para autoejecutar el algoritmo en sistemas *Microsoft Windows*. (Puede consultarse su contenido desde 'editar' o abriéndose con cualquier lector de texto plano).

1.2 Metodología y herramientas de trabajo:

Para la realización del trabajo, ambos integrantes nos hemos basado en la teoría y ejercicios de clase sobre redes neuronales de perceptrones multicapa con retropropagación, así como consulta online de diferentes recursos de programación en python, así como artículos, referencias y documentos sobre el mismo tema, que vienen recogidas en el final de este documento.

Para la elaboración del trabajo se han utilizado las siguientes herramientas:

Imágenes: GIMP 2 (PGM's) , Adobe Photoshop CC 2015 (Imágenes documentación).

Documentación: Google Docs, Microsoft Word 2013, Microsoft Excel 2013.

Programación: Microsoft Visual Studio Code, Idle, Notepad++, Wordpad, Bloc de notas.

Otros: VMware Workstation 12 (Para probar en sistemas linux), Dropbox.

2. Soluciones y mejoras desarrolladas.

2.1 Diseño directo píxel a píxel.

2.1.1 Descripción general:

En este apartado presentaremos paso a paso, el algoritmo “base” que hemos desarrollado. Este algoritmo toma como entrada a la red neuronal un perceptrón por cada píxel de la imagen correspondiente, y dado que para este problema son imágenes de 30x30 píxeles, eso suma un total de 900 perceptrones en la capa de entrada. La capa de salida es conformada por 7 perceptrones, representando las 7 letras diferentes a identificar como se vio anteriormente, cuya salida se traducirá en el resultado correspondiente.

Tras configurar el algoritmo de forma manual o mediante fichero, este iterará sobre el conjunto de entrenamiento, adaptando la red según los parámetros configurados, durante un número de iteraciones marcado como máximo, o hasta que la cota de error se alcance (ambos parámetros configurables por el usuario).

El error de cada iteración que se compara con la cota dada, se calcula como la media de los errores cuadráticos de las salidas por cada elemento del conjunto, es decir, elemento a elemento del conjunto de entrenamiento, por cada salida, se resta la salida esperada menos la obtenida (en todas las salidas) y a estas diferencias se le aplica el error cuadrático, posteriormente haciéndose la media aritmética de cada ejemplo del conjunto (errores cuadráticos), lo que conforma una iteración del algoritmo.

Tras el entrenamiento de la red, se pasará por la misma el conjunto de imágenes de prueba, y finalmente, obtendremos la tasa de aciertos, rendimiento y tiempo de ejecución del algoritmo para la configuración dada, pudiendo almacenar la configuración de pesos de la red en un fichero externo de forma opcional.

2.1.2 Configuración y ejecución del algoritmo:

En cada directorio de algoritmo en el proyecto, encontramos lo siguiente:

- Los directorios “entrenamiento” y “prueba” contienen las imágenes “.pgm”.
- **trabajo_ia.py** es el fichero principal del trabajo (por donde empieza la ejecución).
- **ejecutar(windows).bat** ejecuta en python automáticamente “trabajo_ia.py” para sistemas Windows. (Para comodidad del usuario)
- **Unix/Linux:** Utilizar la instrucción “**python3**”, y no “**python**” en sistemas linux/unix, para que la versión de python sea la adecuada a la hora de ejecutar el fichero. (Y así evitar errores).
- **red.conf** es el fichero de configuración, desde el que opcionalmente se puede configurar la red.
- El resto de ficheros completan la programación del algoritmo, divididos en diferentes módulos. (Aunque en todos los algoritmos compartan nombre, no son intercambiables en su mayoría)

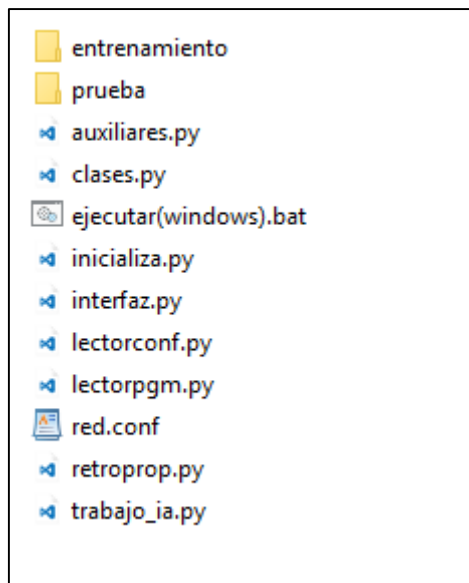


Figura 2

Para el ejemplo que seguiremos en esta documentación, se efectúa sobre un sistema *Windows*, pero el funcionamiento y configuración no varía según el sistema operativo que se esté utilizando, simplemente lo tomamos como referencia para las capturas que aparecerán más adelante.

Tras iniciar la ejecución mediante el método correspondiente, obtendremos lo siguiente:

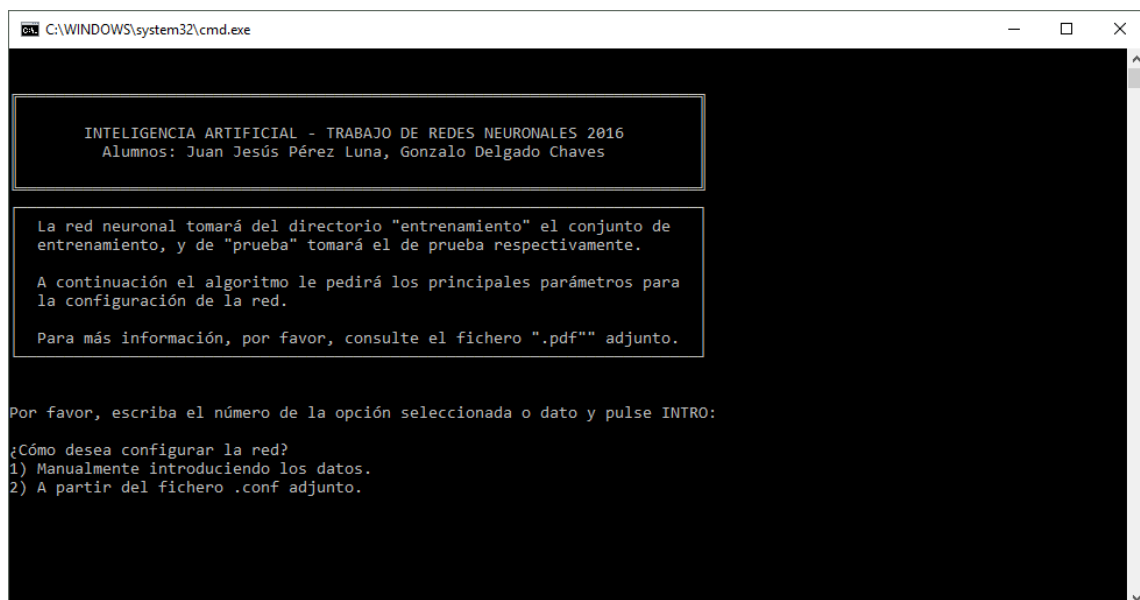


Figura 3

Tal como pide la interfaz, podremos elegir las opciones deseadas escribiendo por teclado la opción y pulsando “ENTER”, respondiendo el algoritmo en consecuencia si la entrada proporcionada no es aceptable como puede verse a continuación. (Eso incluye utilizar tipos de número que no se ajusten a lo que se pide, por ejemplo intentar meter 3.5 en el número de capas cuando debe ser un valor entero).

```
C:\WINDOWS\system32\cmd.exe

INTELIGENCIA ARTIFICIAL - TRABAJO DE REDES NEURONALES 2016
Alumnos: Juan Jesús Pérez Luna, Gonzalo Delgado Chaves

La red neuronal tomará del directorio "entrenamiento" el conjunto de
entrenamiento, y de "prueba" tomará el de prueba respectivamente.

A continuación el algoritmo le pedirá los principales parámetros para
la configuración de la red.

Para más información, por favor, consulte el fichero ".pdf" adjunto.

Por favor, escriba el número de la opción seleccionada o dato y pulse INTRO:

¿Cómo desea configurar la red?
1) Manualmente introduciendo los datos.
2) A partir del fichero .conf adjunto.

ejemplo introduciendo texto 203'9ur 30'2uj3 9r2
Por favor, introduzca una respuesta válida:
```

Figura 4

Como vemos, nos vuelve a pedir el dato tras introducir una opción inválida. En este primer caso configuraremos la red manualmente con la primera opción.

```
C:\WINDOWS\system32\cmd.exe

La red neuronal tomará del directorio "entrenamiento" el conjunto de
entrenamiento, y de "prueba" tomará el de prueba respectivamente.

A continuación el algoritmo le pedirá los principales parámetros para
la configuración de la red.

Para más información, por favor, consulte el fichero ".pdf" adjunto.

Por favor, escriba el número de la opción seleccionada o dato y pulse INTRO:

¿Cómo desea configurar la red?
1) Manualmente introduciendo los datos.
2) A partir del fichero .conf adjunto.

ejemplo introduciendo texto 203'9ur 30'2uj3 9r2
Por favor, introduzca una respuesta válida:
1
Opción elegida: 1

Por favor, elija el número de capas ocultas de la red (Puede ser 0): 2
Su respuesta: 2

¿Cómo desea configurar el número de neuronas por cada capa oculta?
1) Manualmente introduciendo los valores por cada capa.
2) Todas las capas ocultas con el mismo número de neuronas
```

Figura 5

El algoritmo nos pedirá los diferentes parámetros de configuración que restan del mismo modo. En el caso que observamos en la *Figura 5*, el usuario ha elegido 2 capas ocultas para la red (El algoritmo soporta 0 para una configuración sin capas intermedias). También nos preguntará como en el presente caso, si para configurar algún parámetro múltiple queremos aplicarlo por igual a toda la red o capa por capa (mínima granularidad que permitimos para configurar los parámetros iniciales).



```
C:\WINDOWS\system32\cmd.exe

Por favor, proporcione el número máximo de iteraciones a realizar: 10000
Su respuesta: 10000

Red configurada con éxito

Procesando...

Iteración 1 completada.
Iteración 2 completada.
Iteración 3 completada.
Iteración 4 completada.
Iteración 5 completada.
Iteración 6 completada.
Iteración 7 completada.
Iteración 8 completada.
Iteración 9 completada.
Iteración 10 completada.
Iteración 11 completada.
Iteración 12 completada.
Iteración 13 completada.
Iteración 14 completada.
Iteración 15 completada.
Iteración 16 completada.
Iteración 17 completada.
```

Figura 6

Una vez configurados todos los parámetros de la red, recibiremos una confirmación de ello por parte del algoritmo y comenzará el proceso de entrenamiento, mostrándonos la iteración actual por la que va ejecutando, tal como se muestra en la figura 6 (De ese modo podemos hacernos una idea intuitiva del tiempo que tardará el algoritmo en completar el procesado).

Una vez completado el proceso, podremos visualizar los aciertos que ha conseguido nuestra red tras ser entrenada y pasar el conjunto de prueba por ella, visualizando además el porcentaje de rendimiento respecto de ese conjunto de prueba, así como el tiempo total en segundos, dedicado al entrenamiento y prueba de la red, como se puede ver en la figura 7, (Para este ejemplo hemos hecho una configuración rápida con menos elementos de prueba para poder visualizarlo en una única captura). En el caso de obtener un asterisco en las soluciones obtenidas (“*”) se considera “no concluyente”, es decir, que más de una salida ha obtenido el mismo valor máximo (no se puede decidir correctamente) y se considera igual que un caso negativo.


```
C:\WINDOWS\system32\cmd.exe
Iteración 98 completada.
Iteración 99 completada.
Iteración 100 completada.

Red entrenada tras 100 iteraciones. Evaluando conjunto de prueba.

Conjunto de prueba ejecutado sobre la red entrenada.

Prueba N°# · Obtenido · Esperado · Criterio
1          f          d        -NO-
2          f          f        -OK-
3          h          h        -OK-
4          h          j        -NO-
5          f          l        -NO-
6          h          n        -NO-
7          h          p        -NO-

Resultado Final: 2/7      Rendimiento: 28%

Tiempo de ejecución: 2.240209450000205 segundos.

¿Desea volcar el valor de los pesos en el fichero ".dat" adjunto?
1) Sí.
2) No.
```

Figura 7

Finalmente, acabará la ejecución tras elegir la última opción, que en el caso de elegir guardar esa información, se generará (o sobrescribirá) un fichero de texto plano tal como aparece en las figuras 8 y 9 (fichero y contenido) Nota: Tan solo es un ejemplo, los datos de una captura y otra no tienen por qué corresponderse ya que han sido tomadas en diferentes momentos)

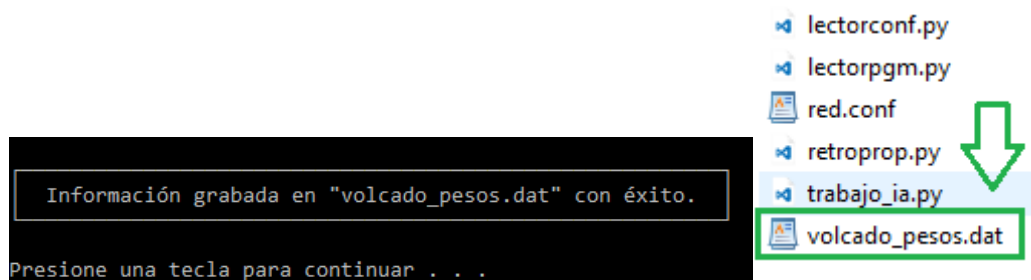
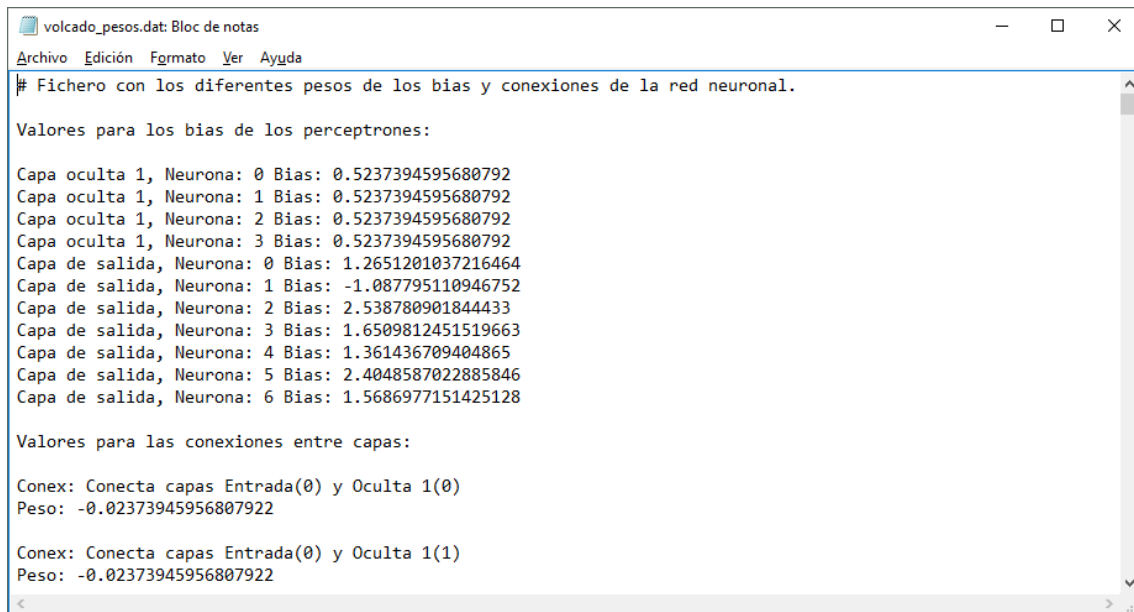


Figura 8



```
# Fichero con los diferentes pesos de los bias y conexiones de la red neuronal.

Valores para los bias de los perceptrones:

Capa oculta 1, Neurona: 0 Bias: 0.5237394595680792
Capa oculta 1, Neurona: 1 Bias: 0.5237394595680792
Capa oculta 1, Neurona: 2 Bias: 0.5237394595680792
Capa oculta 1, Neurona: 3 Bias: 0.5237394595680792
Capa de salida, Neurona: 0 Bias: 1.2651201037216464
Capa de salida, Neurona: 1 Bias: -1.087795110946752
Capa de salida, Neurona: 2 Bias: 2.538780901844433
Capa de salida, Neurona: 3 Bias: 1.6509812451519663
Capa de salida, Neurona: 4 Bias: 1.361436709404865
Capa de salida, Neurona: 5 Bias: 2.4048587022885846
Capa de salida, Neurona: 6 Bias: 1.5686977151425128

Valores para las conexiones entre capas:

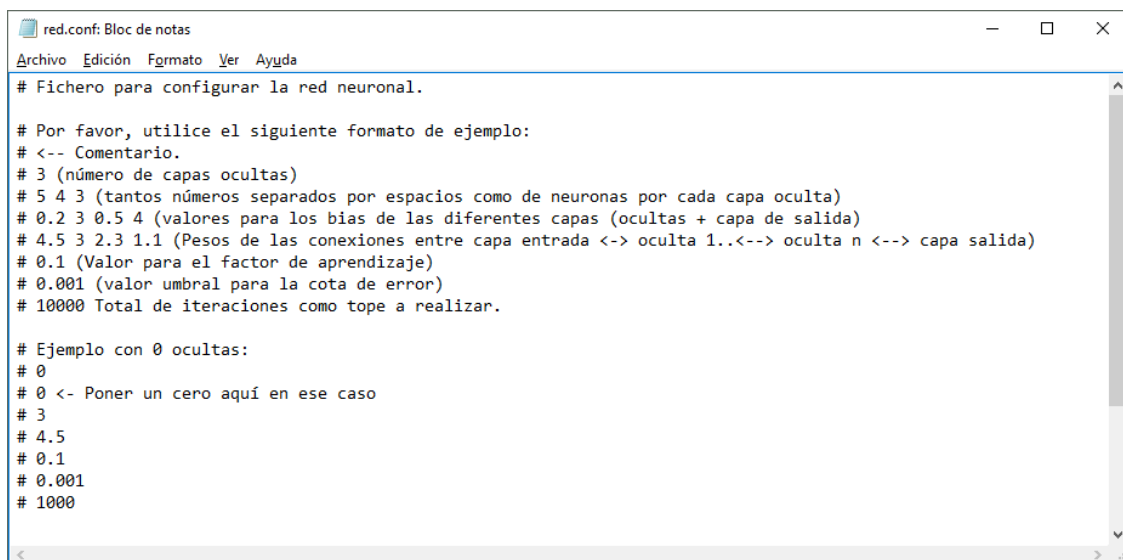
Conex: Conecta capas Entrada(0) y Oculta 1(0)
Peso: -0.02373945956807922

Conex: Conecta capas Entrada(0) y Oculta 1(1)
Peso: -0.02373945956807922
```

Figura 9

En el ejemplo de la figura 9, vemos el volcado del fichero, en el que por ejemplo no aparecen los “bias” de la capa de entrada, ya que no se le aplican (aunque compartan en código la misma estructura que el resto, están inicializados a 0).

La segunda forma de configurar el algoritmo es mediante el fichero ‘.conf’ que se adjunta, cuyo contenido puede verse parcialmente en la figura 10, de modo que al utilizar esa opción el algoritmo arranca directamente con los datos que lee de dicho fichero (es una opción más rápida a la hora de probar diferentes configuraciones.)



```
# Fichero para configurar la red neuronal.

# Por favor, utilice el siguiente formato de ejemplo:
# <-- Comentario.
# 3 (número de capas ocultas)
# 5 4 3 (tantos números separados por espacios como de neuronas por cada capa oculta)
# 0.2 3 0.5 4 (valores para los bias de las diferentes capas (ocultas + capa de salida)
# 4.5 3 2.3 1.1 (Pesos de las conexiones entre capa entrada <--> oculta 1.<--> oculta n <--> capa salida)
# 0.1 (Valor para el factor de aprendizaje)
# 0.001 (valor umbral para la cota de error)
# 10000 Total de iteraciones como tope a realizar.

# Ejemplo con 0 ocultas:
# 0
# 0 <- Poner un cero aquí en ese caso
# 3
# 4.5
# 0.1
# 0.001
# 1000
```

Figura 10

El propio documento ‘.conf’ contiene las instrucciones para rellenarlo correctamente, en caso contrario la aplicación mostrará fallo. El fichero soporta comentarios de una línea y en la misma línea, así como líneas en blanco. (Implementación propia que puede verse en el fichero “lectorconf.py”)

2.1.3 Parámetros pseudo-configurables:

Algunos datos de la configuración del algoritmo, pueden cambiarse sólo vía código, ya que no consideramos externalizarlos porque son intrínsecos al problema que se plantea, no obstante, el algoritmo soportaría su cambio, como por ejemplo, dentro de “*inicializa.py*” (ver un fragmento en figura 11), donde se lleva a cabo la inicialización de la red con los parámetros que hemos configurado previamente, podemos manualmente cambiar la capa de entrada (Aunque para funcionar los datos de entrada, como en el caso de las imágenes, deberían ser acordes).

```
def inicializa_red(conf):
    red_neuronal = [] # Información de salida

    # Parámetros de inicialización.
    """Entradas manuales -----"""
    perceptrones_entrada=900
    perceptrones_salida=7
    """-----"""

    num_capas_ocultas=conf[0]
    num_neu_capa_oculta=conf[1]
    bias_capas=conf[2]
    peso_conex=conf[3]
```

Figura 11

En la misma línea, la función de activación está como método en la clase “*Perceptron*” el cual podemos hallar en “*clases.py*” (tal como se ve en la figura 12). Para cambiarla bastaría con sobrescribir este método. (Aunque en el enunciado se especifica que sea el sigmoide).

```
8 class Perceptron(object):
9     def __init__(self, idx, bias):
10         self.idx = idx #Índice dentro de su capa, con el objetivo de acceder directamente
11         self.bias = bias
12         self.salida = 0 # Provisionalmente a cero hasta que la activación se calcule
13         self.conex_in = []
14         self.conex_out = []
15         self.delta = 0.0
16
17     def activa(self, input):
18         return 1 / (1 + math.exp(-input))
19
20     def str (self):
```



Figura 12

2.1.4 Detalles de implementación:

No vamos a aportar una descripción detallada de la implementación, ya que consideramos que para eso está el código ejecutable, además de que las fórmulas usadas para el algoritmo son las que hemos visto durante la asignatura, sin modificaciones sustanciales, no obstante, si consideramos interesante explicar en abstracto, la arquitectura base del algoritmo, para dar una mejor idea de su funcionamiento, ya que esta versión fue completamente reprogramada en busca de una mejor eficiencia.

Básicamente, tal como podemos ver en el esquema de la figura 13, utilizamos dos clases, “perceptron” y “conexion” (que representan lo que su propio nombre indica), en el primero, guardamos como parámetro su valor asignado de “bias”, la salida del perceptrón (ahí se almacenará el resultado de aplicar el sigmoide durante el proceso), un identificador “idx”, que recogerá la posición relativa en la capa que está situado (Para facilitar acceso directo en la estructura de perceptrones), un parámetro delta, que nos sirve para almacenar temporalmente este parámetro durante el cálculo de valores de la retropropagación (En teoría este parámetro no tiene pertenencia al perceptrón, pero por eficiencia en la ejecución del algoritmo nos conviene utilizarlo de esta forma). Podemos ver también que cada perceptrón contiene dos listas, “conex_in” y “conex_out”, en las que almacenaremos respectivamente las conexiones de entrada y salida de ese perceptrón en concreto. (Habrá dos referencias a cada objeto de tipo conexión, uno en la lista de salientes de un perceptrón y otra en la lista de entrantes de otro perceptrón de la “siguiente capa”).

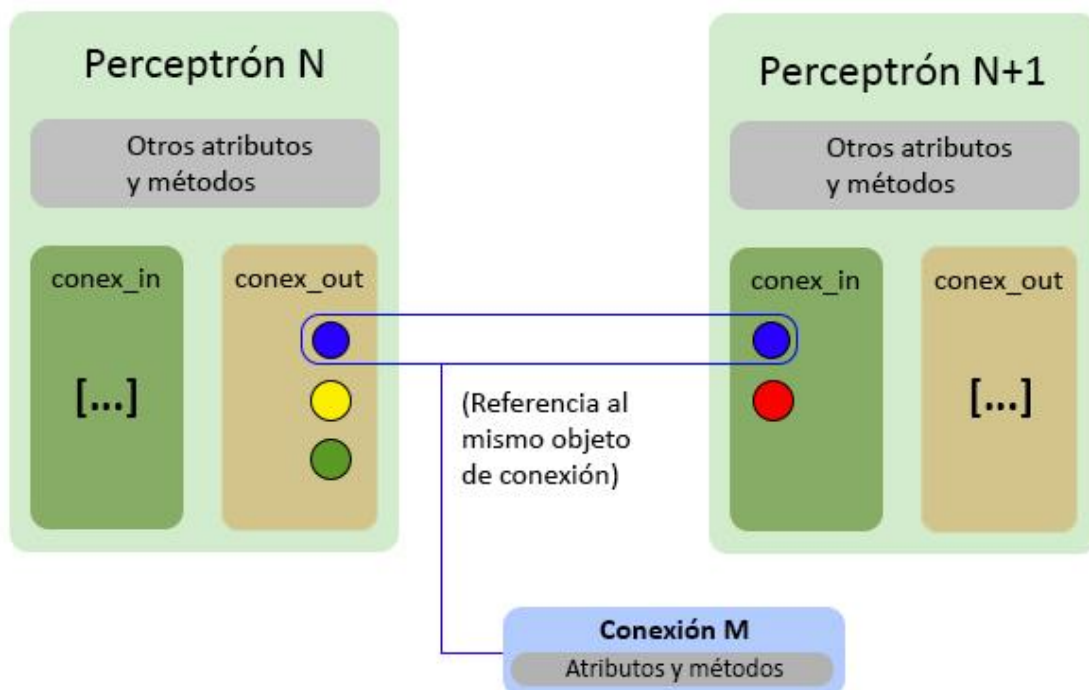


Figura 13

En cuanto a la estructura para las conexiones, contienen su peso, y dos índices, que representan el índice del perceptrón que conecta como entrada dentro de su propia capa, y de igual modo el del perceptrón de salida, de este modo, a la hora de manejar la arquitectura de la red, no necesitamos realizar ninguna búsqueda, siendo todas las operaciones accesos directos y recorridos. La versión que implementamos previamente, contemplaba las conexiones en una lista independiente, y contenía referencias a los objetos perceptrón, pero la necesidad de realizar muchísimas búsquedas, destruía por completo la eficiencia del algoritmo, por lo que tuvimos que replantear toda la arquitectura, afortunadamente con éxito.)

2.2 Diseño con “compresión” virtual de imagen.

2.2.1 Descripción general:

En este apartado, planteamos una versión alternativa basada en el algoritmo básico que hemos descrito en el apartado anterior. Si bien el funcionamiento del mismo es muy similar, la principal variación está en el módulo de entrada, ya que aun utilizando las mismas imágenes para el conjunto de entrenamiento y prueba, pre-procesamos esta información, de forma que tomamos un conjunto representativo de cada una de las nombradas imágenes, en lugar de su totalidad.

Para ello, una vez leída la información de cada imagen, tomamos submatrices NxN no solapadas (no sería el caso de una convolución), de lo que tomamos la moda sus valores. El tamaño de esta submatriz (cuadrada) es un parámetro configurable por el usuario, al que llamamos “factor de compresión”.

Hemos desarrollado esta variante por dos principales razones, la primera, tiene que ver con la eficiencia temporal, ya que disminuye cuadráticamente el tamaño de las entradas y por tanto el número de conexiones, lo que implica un gran recorte en el número de operaciones a realizar. Por ejemplo, veámoslo en un supuesto numérico:

Datos: 1 capa oculta de 15 perceptrones.

Versión sin compresión: 900 perceptrones (Entrada), 15(Oculto), 7(Salida) hace un total de 922 perceptrones, y $900 \times 15 + 15 \times 7$ conexiones, que suman un total de **13605**.

Versión con compresión (factor 3): 100 perceptrones (Entrada), 15(Oculto), 7(Salida) hace un total de 122 perceptrones, y $100 \times 15 + 15 \times 7$ conexiones, que suman un total de **1605**.

Versión con compresión (factor 5): 36 perceptrones (Entrada), 15(Oculto), 7(Salida) hace un total de 56 perceptrones, y $36 \times 15 + 15 \times 7$ conexiones, que suman un total de **645**.

Como puede observarse, la disminución del número total de conexiones a manejar es considerable, y más si tenemos en cuenta la cantidad de operaciones a realizar en cada una de las muchas iteraciones que llevará a cabo el algoritmo, lo que tiene un impacto fácilmente observable en el tiempo de ejecución.

Sin embargo, un “factor de compresión” demasiado alto, puede provocar que perdamos demasiada información útil de cada imagen original, y perdamos pues rendimiento por la semántica del problema.

La segunda razón para utilizar esta variante, se trata de que si bien mucha “compresión” puede distorsionar la información útil de entrada, un poco puede ser hasta beneficioso, ya que letras muy parecidas, pero con pequeñas diferencias de posicionamiento en el marco de la imagen, quedan mucho más cercanas siguiendo este método, por lo que incluso podría en determinados casos, mejorar el rendimiento del algoritmo, a la par de ser menos costoso en tiempo (Dependerá de la parametrización exacta que se aplique).

2.2.2 Configuración y ejecución del algoritmo:

Dado que nos basamos en el algoritmo del primer apartado, la forma de configurar y ejecutar es prácticamente la misma, con la única diferencia, de que ya sea manualmente o utilizando el fichero de configuración adjunto, se nos solicitará un parámetro adicional para el “factor de compresión” (por ejemplo ‘3’ para comprimir utilizando submatrices ‘3x3’ o 6 para utilizar submatrices ‘6x6’). (Aunque se permita en la configuración, durante la ejecución del algoritmo se exigirá que este valor sea divisor del tamaño de lado de la imagen original, o detendrá la ejecución, avisando al usuario de la forma que podemos observar en la figura 14 que aparece a continuación).



```
C:\WINDOWS\system32\cmd.exe

La red neuronal tomará del directorio "entrenamiento" el conjunto de
entrenamiento, y de "prueba" tomará el de prueba respectivamente.

A continuación el algoritmo le pedirá los principales parámetros para
la configuración de la red.

Para más información, por favor, consulte el fichero ".pdf" adjunto.

Por favor, escriba el número de la opción seleccionada o dato y pulse INTRO:

¿Cómo desea configurar la red?
1) Manualmente introduciendo los datos.
2) A partir del fichero .conf adjunto.
2
Opción elegida: 2

Red configurada con éxito

El factor de compresión no cumple las restricciones establecidas para el problema.
Presione una tecla para continuar . . .
```

Figura 14

2.2.3 Parámetros pseudo-configurables:

Tal como aparecían en el algoritmo base del primer apartado, existen los mismos parámetros configurables solo directamente desde el propio fichero “.py” correspondiente, salvo por un caso concreto a señalar, sobre el número de perceptrones de entrada, que se recalcula automáticamente según el “factor de compresión” que estemos utilizando.

En la figura 15 que aparece a continuación, puede verse un fragmento de “inicializa.py” donde se trata lo anteriormente descrito.

```

12     # Parámetros de inicialización.
13     """Entradas manuales -----"""
14     perceptrones_entrada=900
15     perceptrones_salida=7
16     lado=30 #Fijo por el problema a resolver.
17     """-----"""
18
19     """Modificación automática según factor -"""
20     bloque=conf[7] #Factor de compresión: 3=submatrices 3x3 (tiene que ser divisor de 'lado')
21     if lado%bloque==0: #Es divisor correcto
22         perceptrones_entrada=perceptrones_entrada//(bloque*bloque)
23     else:
24         print(' ')
25         print("El factor de compresión no cumple las restricciones establecidas para el problema.")
26         print(' ')
27         sys.exit(1) #Para cortar la ejecución del programa.
28     """-----"""
29
30     num_capas_ocultas=conf[0]

```

Figura 15

2.2.4 Detalles de implementación:

Salvo por pequeños cambios para adaptar la lectura de fichero y manuscrita de datos, así como la detección automática del tamaño de la entrada (subapartado anterior), el único cambio más interesante de analizar podemos encontrarlo en la función “comprimir_vector_imagen” que aparece en “auxiliares.py”, y es precisamente el método que aplica el factor y genera una imagen “virtual” comprimida de salida respecto de la original. (Puede verse un fragmento en la figura 16).

```

11 def comprimir_vector_imagen(vect,factor,tam_1):
12
13
14     solucion_matriz=[]
15     solucion_vector=[]
16     m=[]
17     # Convierto a formato matricial (no una fila sola)
18     i=0
19     while i<len(vect):
20         fila=vect[i:i+tam_1]
21         m.append(fila)
22         i+=tam_1
23
24     m_comprimida = []
25     #Índices i_principio, i_final, j_principio, j_final... proceso las submatrices.
26     i_p=0

```

Figura 16

2.3 Diseño con “momentum” del gradiente.

2.3.1 Descripción general:

En este apartado introducimos una mejora sobre la versión más reciente del algoritmo, basada en la técnica de momento sobre el gradiente. Esta técnica es independiente de la configuración de la red, y afecta a la fase de actualización de los pesos durante el proceso de aprendizaje, dando lugar a posibles mejoras en el entrenamiento de la red, que resultan en

porcentajes favorables de aceptación de conjuntos de prueba con respecto a versiones sin esta mejora.

El 'momentum', o inercia del descenso del gradiente, es una técnica según la cual se añade un pequeño porcentaje del valor aprendido de los pesos de cada iteración a la siguiente. El algoritmo de retropropagación, al trabajar con una técnica de descenso por gradiente, es susceptible a sufrir problemas de mínimos locales, en los cuales la variación del gradiente se vuelve mínima o inmóvil. Esto puede causar que el aprendizaje de la red se vea afectado en términos de tiempo, coste computacional y resultados finales.

Al actualizar los pesos de las conexiones entre perceptrones, se suma un nuevo término, que incluye dos factores: uno configurado libremente y estático para el proceso de aprendizaje, y otro calculado en función de la iteración previa. La fórmula para actualizar los pesos, queda de la siguiente forma:

$$\Delta w_{ij}(t) = \mu_i \delta_i y_j + m \Delta w_{ij}(t-1)$$

Figura 17

Donde t es la iteración actual, m el factor de momento y el sumando incluyendo a dicha m el momento asociado a la conexión w .

El momento en cada iteración se calcula a partir del valor actualizado de cada conexión en la iteración anterior. Para la primera iteración, no existiría dicho momento, de modo que la retropropagación del error se produciría de igual forma a la observada en versiones anteriores; adicionalmente, este valor se guardaría para emplear en la siguiente iteración, y así sucesivamente.

El uso de la técnica del momento en el gradiente es ampliamente soportada en la mayoría de problemas de aprendizaje mediante redes neuronales, ya que 'empuja' el gradiente para que converja más rápidamente a un aprendizaje satisfactorio, evitando pequeños 'valles' o 'mínimos locales' en el proceso.

2.3.2 Configuración y ejecución del algoritmo:

Puesto que la mejora del momento no actúa sobre ninguno de los parámetros anteriores, es posible ejecutarlo sobre cualquier configuración del algoritmo. Para nuestro ejemplo, nos hemos basado en la versión con compresión, ya que en ella se encuentra implícitamente la asignación 1:1 de pixel a entrada (Submatrices 1x1 equivalen al algoritmo original). Tanto la interfaz gráfica como el fichero de configuración permiten en este caso introducir un valor del factor momento al usuario, cumpliendo la restricción de que dicho valor se encuentre entre 0 y 1, ambos excluidos. Dado que el factor momento es único para la completitud de la fase de aprendizaje, no es necesario hacer otras consideraciones en la ejecución de la interfaz.

2.3.3 Parámetros pseudo-configurables:

Para esta versión del algoritmo, no se añade ningún parámetro que sólo pueda configurarse desde el propio código, que no comparta con los casos anteriormente ya descritos.

2.3.4 Detalles de implementación:

Para la implementación del momento, se ha modificado el método de propagación de error “actualiza_pesos” que se puede encontrar en “retroprop.py”, efectivamente dividiéndose en dos variantes: una para la primera iteración, donde el momento es inexistente, pero los resultados para cada conexión se deben guardar, y otra para las sucesivas iteraciones, donde se añade el momento de la iteración previa y se recalcula para la siguiente, llamada “actualiza_pesos_con_momentum”.

La implementación del momento es sencilla, mediante el uso de parámetros internos a las clases “Perceptron” y “Conexión”. El perceptrón, al tener su bias como propiedad individual, necesita también de una propiedad momento que asigne el sumatorio a dicho bias. Haciendo uso de estas propiedades, guardar los cálculos de retropropagación y usarlos como sumatorios (multiplicados por el factor momento, que es un parámetro de entrada a la función) es sencillo y no requiere de operaciones adicionales.

Como nota, la implementación del momento añade un número lineal de operaciones por iteración, que en el conjunto de un proceso completo de aprendizaje no suponen un incremento porcentual alto.

3. Benchmarking de los algoritmos.

3.1 Consideraciones iniciales y metodología:

3.1.1 Introducción:

En este apartado presentaremos un benchmarking de los 3 algoritmos presentados para diferentes valores de la configuración, así como unas conclusiones sobre la parametrización de los mismos. También se acompaña de una tabla comparativa y las pruebas realizadas en un fichero adjunto de tipo excel (“.xls”).

3.1.2 Entorno de pruebas:

Las pruebas han sido efectuadas en diferentes microprocesadores, pero en aquellas que hemos obtenido mejores resultados, han sido re-ejecutadas sobre el mismo procesador para poder comparar entre ellas. En este caso el hardware utilizado es una CPU Intel i7 4770K con frecuencias de stock junto a 16GB de RAM a 2400MHz, sobre un sistema Windows 10 Education 64 bits (Derivado de enterprise edition). Solo hemos utilizado Linux con Ubuntu 16.04 LTS 64 bits bajo virtualización para comprobar su correcto funcionamiento, pero no para el proceso de benchmarking.

3.1.3 Metodología:

Para las pruebas, hemos utilizado parametrizaciones parecidas a las vistas en los ejemplos de la asignatura, más las encontradas en algunos de los artículos que vienen enlazados en la sección de materiales al final de este documento. Sobre esa base, hemos detectado que parámetros suelen dar mejores resultados, y moviéndonos con valores en un entorno a ellos, hemos concebido distintos casos de prueba y tomado nota de los parámetros utilizados en el documento “.xls”, hasta conseguir un rendimiento que consideramos aceptablemente bueno.

3.2 Resultados del proceso de benchmarking:

3.2.1 Comparativa entre algoritmos:

En este documento, sólo compararemos algunas de las mejores parametrizaciones obtenidas por cada uno de los algoritmos, estando el resto recogidas en el documento “.xls” adjunto antes mencionado, para su consulta.

A continuación mostramos una tabla comparativa con las mejores versiones mencionadas:

| MEJORES CONFIGURACIONES | | | | | | | |
|---------------------------------|--------------|--------------|-------------|-------------|-------------|-------------|-------------|
| Algoritmo | Momentum | Momentum | Momentum | Momentum | Básico | Compresión | Básico |
| Capas ocultas | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| Perceptrones por capa oculta | 5 | 8 | 8 | 10 | 0 | 5 | 20 |
| Pesos de conexiones entre capas | 0 0 | 0.001 0.001 | 0.001 0.001 | 0.01 0.01 | 0 | 0.001 0.001 | 0 0 |
| Bias por capa | 0.5 0.5 | 0.5 0.5 | 0.3 0.3 | 0.5 0.5 | 0.5 | 0.5 0.5 | 0 0 |
| Factor de aprendizaje | 0.1 | 0.1 | 0.5 | 0.2 | 0.1 | 0.1 | 0.15 |
| Compresión (Entradas) | 3x3 | 3x3 | 3x3 | 2x2 | No | 6x6 | No |
| Cota de error | 0'01 | 0'0001 | 0'0001 | 0'001 | 0'000001 | 0.001 | 0'000001 |
| Iteraciones (tope) | 1000 | 1000 | 1000 | 500 | 1000 | 1500 | 1000 |
| Iteraciones usadas | 104 | 1000 | 300 | 141 | 1000 | 1500 | 1000 |
| Tiempo de ejecución (Segundos) | 3'8524 | 55'5549 | 16'4914 | 20'2676 | 284'5025 | 13'7855 | 800'8525 |
| Rendimiento (Aciertos) | 100% (70/70) | 100% (70/70) | 97% (68/70) | 95% (67/70) | 92% (65/70) | 61% (43/70) | 51% (36/70) |
| Momento | 0.9 | 0.6 | 0.2 | 0.5 | No | No | No |

Figura 18

Tal como podemos observar, los mejores datos son obtenidos por el algoritmo que implementa el “momentum”, (El cual también es a su vez una variante del algoritmo de compresión), hasta tal punto de conseguir un **100%** de aciertos en más de una ocasión, incluso en menos de **4 segundos**. (Recordamos que los conjuntos de entrenamiento y prueba tienen 70 elementos cada uno).

Aunque en la figura 18, (tomada de la segunda hoja del fichero “.xls” adjunto que contiene el proyecto) aparecen 4 resultados del algoritmo con momentum, existen más resultados superiores a los demás algoritmos utilizando esta técnica, pero hemos querido incluir algunos como “mejores de su clase” para los resultados que hemos obtenido.

En el caso del algoritmo que incorpora solo compresión, en casi todos los casos hemos obtenido rendimientos entre 40% y 60% de donde parece estancarse, a lo que atribuimos un posible mínimo local, alcanzándose en muy diferentes configuraciones.

Por su parte, el algoritmo básico (el primero que hemos definido en el documento, que no incorpora compresión ni momentum) consigue una buena tasa en un caso muy específico con ninguna capa intermedia, aunque aparezca en la tabla, “dejamos fuera” este caso por esa misma razón, ya que tratamos con algoritmos de perceptrones multicapa con retropropagación y no sería el caso. Para el resto de los algoritmos, tal como en el caso de la compresión, no pasa del aparente mínimo local que aparece sobre el 40% ~ 50%, aunque en este caso el tiempo dedicado a procesamiento es notablemente superior (más de diez minutos para 51%).

Aun así, no descartamos que haya configuraciones mejores para cada uno de los algoritmos, ya que al fin y al cabo, el método es de prueba y error, y depende de muchos factores combinados al mismo tiempo.

3.2.2 Conclusiones:

- Un factor moderado de compresión ayuda a aligerar la carga y tiempo de procesamiento, con resultados satisfactorios, normalmente los mejores entorno a 3x3 y 5x5.
- Los mejores resultados tienden a obtenerse utilizando una única capa oculta intermedia.
- Es recomendable utilizar valores iniciales para el peso de las conexiones cercanos a cero.
- Los valores iniciales de los “bias” parecen funcionar bien en torno a 0,5.
- Es recomendable utilizar valores cercanos a cero para el factor de aprendizaje.
- La cota de error, no tiene que ser extremadamente pequeña, ya que con 0,01 o 0,001 es suficiente si el algoritmo converge adecuadamente.
- En cuanto al número de iteraciones, especialmente utilizando momentum, suelen estar comprendidas entre 100 y 10000 para obtener buenos resultados en este problema concreto.
- Incorporar momentum en el algoritmo incrementa notablemente la convergencia, y eligiendo bien la configuración, ayuda a evitar mínimos locales.
- Es de notar que el factor momento y el factor de aprendizaje están relacionados. Dado que ambos determinan la velocidad de descenso del gradiente, si ambos factores son muy pequeños, el aprendizaje se ralentizará, y si son muy altos, el conjunto tenderá a exceder el mínimo deseado. Es deseable que ambos factores sean proporcionalmente inversos.

3.2.3 Curiosidades:

- Quisiéramos comentar una anomalía que ha surgido en el proceso de benchmarking, al comparar el resultado de una misma configuración en dos ordenadores diferentes, en la mayor parte obtuvimos idénticos resultados, sin embargo, en algunos, a pesar de que todos los algoritmos presentados son completamente deterministas, obtuvimos ligeras variaciones en el rendimiento (tasa de aciertos, no en el tiempo de ejecución que es obviamente derivado de la capacidad de cada procesador y carga en el momento de la ejecución), por ejemplo diferencias entre 100% y 97% (± 2) de tasa de aciertos.

La hipótesis que manejamos para ello, es que probablemente cada procesador tenga alguna variación en la precisión en operaciones de coma flotante, y como resultado de minúsculas variaciones en los redondeos.

4. Posibles mejoras planteadas (no implementadas).

4.1 Otro tipo de “compresión”, convoluciones, etc.

Una de las posibles ideas que no hemos aplicado a nuestro proyecto, sería la aplicación de otro tipo de “compresión” más allá de las submatrices, como pudiera ser comprimir toda la información de cada imagen en un único elemento, y de forma análoga para las columnas. También en este apartado consideramos la posibilidad de hacer recorrido de submatrices pero con solapamiento, es decir, aplicar filtros de convolución con un factor dado.

4.2 Preprocesado de imágenes con “visión por computador”.

Otra de las posibilidades a aplicar, es centrar esfuerzos en la detección de la imagen por parte del algoritmo, aplicando patrones de “esqueletización” con tratamientos de “erosión” y/o “dilatación” tal como se hacen en muchos algoritmos de reconocimiento de imágenes, con el objetivo de que las imágenes que representan la misma letra, sean en su captura lo más parecidas posible unas de otra, incluso si aparecen a priori “movidas” o “rotadas”. La idea por lo tanto sería un preprocesado completo que adapte la entrada a unos estándares preestablecidos, para aumentar la eficacia de la red neuronal, al existir menos “ruido” o “diferencias” entre letras que son iguales o representan la misma salida en el algoritmo. Aunque sea aplicado con asiduidad en entornos reales, hemos descartado esta opción ya que se sale de los conceptos enseñados en la asignatura y tendría un impacto considerable en el tiempo de desarrollo para el mismo.

4.3 Programación concurrente.

Por último, consideramos que de cara a plantear un algoritmo generalizado para perceptrones multicapa, dado el tiempo que consumen las configuraciones con muchos elementos e iteraciones, sería conveniente paralelizar el código, de modo que aumente muchísimo la eficiencia cuando se trabaja con procesadores multinúcleos y/o multihilo, tal como es lo usual en la actualidad. Incluso podríamos considerar dar un paso más, y adaptar ese tipo de paralelización para sistemas con soporte CUDA y similares. (Nuevamente, la propuesta se sale de los objetivos de la asignatura).

5. Material de consulta y enlaces de interés.

5.1 Introducción.

En este apartado recogemos todo el material que hemos utilizado para el aprendizaje, consulta, y codificación de la red neuronal para el trabajo.

5.2 Material y referencias.

5.2.1 Material de teoría sobre redes neuronales:

- Diapositivas y ejercicios de la asignatura Inteligencia Artificial.
- Redes Multicapa: https://en.wikipedia.org/wiki/Multilayer_perceptron
- Redes Multicapa: http://catarina.udlap.mx/u_dl_a/tales/documentos/lep/mejia_s_ia/capitulo3.pdf
- Redes Multicapa: <http://es.slideshare.net/GINPAO1/presentacion-backpropagation>
- Redes Neuronales: http://html.rincondelvago.com/redes-neuronales-artificiales_1.html
- Redes Multicapa: <http://www.icee.upc.es/JCEE2002/Aldabas.pdf>
- Sobre Momentum: <https://www.willamette.edu/~gorr/classes/cs449/momrate.html>
- Redes Neuronales: <https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721#.z2pou3csi>

5.2.2 Material sobre programación en *python*:

- Diapositivas y documentos sobre python aportados en la asignatura Inteligencia Artificial.
- Sigmoide: <http://stackoverflow.com/questions/3985619/how-to-calculate-a-logistic-sigmoid-function-in-python>
- Medición de tiempos: <https://docs.python.org/2/library/time.html#module-time>
- Comprobaciones: <https://docs.python.org/3/library/stdtypes.html#str.isdigit>
- Comprobaciones: <https://datafull.co/p/como-puedo-comprobar-si-una-cadena-es-un-numero-en-python>
- Índices en bucles: <http://stackoverflow.com/questions/522563/accessing-the-index-in-python-for-loops>
- Sobre ficheros: http://librosweb.es/libro/python/capitulo_9/sobre_el_objeto_file.html
- Sobre ficheros: http://librosweb.es/libro/python/capitulo_9/metodos_del_objeto_file.html
- Cortar ejecución de programa: <https://mail.python.org/pipermail/python-es/2010-October/028467.html>
- Existencia fichero: <http://www.lawebdelprogramador.com/foros/Python/1075833-Comprobar-existencia-de-archivo.html>

5.2.3 Otros materiales de consulta:

- Apuntes y contenido de la asignatura Procesamiento de Señales Multimedia (3º Ingeniería del Software)
- Sobre ficheros PGM: <http://netpbm.sourceforge.net/doc/pgm.html>
- Biblioteca alternativa: <http://pybrain.org/docs/quickstart/network.html>
- Error cuadrático: https://es.wikipedia.org/wiki/Error_cuadr%C3%A1tico_medio
- Aplicación interactiva: <http://playground.tensorflow.org/>