UNIVERSIDAD DE GRANADA

Inteligencia Computacional

Redes neuronales: Reconocimiento óptico de caracteres MNIST

Práctica 1

Víctor Vázquez Rodríguez
victorvazrod@correo.ugr.es

Máster universitario en Ingeniería Informática

Curso 2019/20

Índice

1. Introducción

El objetivo de esta práctica es obtener un modelo de red neuronal capaz de clasificar las imágenes de la base de datos de dígitos manuscritos de MNIST. Se trata de imágenes de 28x28 píxeles que representan números del 0 al 9 escritos a mano.

Para ello, he implementado y experimentado con varios modelos hasta obtener el mejor resultado. Todas las implementaciones se han realizado con el lenguaje Go y haciendo uso del paquete *mat* de la librería Gonum para realizar el álgebra lineal con matrices necesario.

La base de datos de MNIST nos proporciona 60000 ejemplos para entrenamiento y 10000 para la prueba de nuestros modelos. A lo largo de este documento, se van a exponer las distintas implementaciones que se han realizado y los resultados obtenidos al clasificar estos dos conjuntos para cada una de ellas.

2. Perceptrón

En primer lugar, se implementó el modelo más simple visto en clase: el perceptrón. Esta red consta de 10 neuronas en la capa de salida (una por clase) y 784 neuronas en la capa de entrada (una por cada píxel de la imagen de entrada). Las neuronas de la capa de salida poseen un peso (w) para cada entrada (x) y un sesgo (b).

$$z_i = b_i + \sum_{j=1}^{n} w_i^j x_i^j \tag{1}$$

La entrada neta de cada neurona (z) se calcula usando la ecuación 1. A partir de esta entrada neta, cada neurona de salida del perceptrón producirá un 1 o un 0, dependiendo de si la entrada pertenece a la clase que representa dicha neurona o no, respectivamente.

$$y_i = \begin{cases} 1 & \text{si } z_i \geq 0 \\ 0 & \text{en otro caso} \end{cases} \tag{2}$$

La ecuación 2 es la función de activación que producirá estas salidas para cada neurona. Como se puede apreciar, se trata de una función lineal, de forma que nuestro perceptrón es, en definitiva, un clasificador lineal.

Al principio, inicializamos los pesos con valores aleatorios, usando los ejemplos del conjunto de entrenamiento para ir ajustando los pesos y obtener mejores resultados. La manera de entrenar el perceptrón es la siguiente:

- Si la salida de una neurona es 1 y debería ser 0, se restan a sus pesos los valores de la entrada.
- Si la salida de una neurona es 0 y debería ser 1, se suman a sus pesos los valores de la entrada.

El algoritmo del perceptrón nos garantiza encontrar un conjunto de pesos que clasifique las entradas correctamente, en caso de que dicho conjunto exista. Sin embargo, las clases de este problema no son linealmente separables, por lo que los resultados obtenidos al clasificar los ejemplos con este modelo no son muy buenos, como se puede ver en la siguiente tabla:

$Épocas$	$Tiempo$	$Error_{train}$	$Error_{test}$
10	23,94s	18,13 %	18,73 %

3. Multicapa

La manera de mejorar nuestros resultados consiste en añadir más capas a nuestra red, las cuales serán capaces de extraer características de las imágenes de entrada. Al añadir nuevas capas, ya no podemos seguir utilizando el algoritmo de aprendizaje del perceptrón. Este algoritmo se basaba en comparar la salida deseada con la obtenida y variar los pesos en consecuencia, cosa que no se puede hacer con el nuevo modelo al no saber cual es la salida que deben tener las neuronas de las capas ocultas. En su lugar, lo que hacemos es establecer una función de coste o pérdida que proporcione un valor del error de la salida final respecto a la deseada y, a continuación, calcular como varía dicho error respecto a cada uno de los pesos de la red. Con ello lo que buscamos es descender por el gradiente de este error hacia los valores de los pesos que lo minimizan.

$$E = \frac{1}{2}\sum_{n}(t^n - y^n)^2 \tag{3}$$

$$\Delta w_{ij} = -\eta\frac{\delta E}{\delta w_{ij}} \tag{4}$$

En este primer modelo multicapa, mido esta pérdida usando la función del error cuadrático medio para un conjunto de n ejemplos de entrada, tal y como se puede ver en la ecuación 3. En mi caso, al realizar entrenamiento *online*, esta n es 1. La ecuación 4 muestra la manera de calcular el incremento que se debe aplicar a cada peso para disminuir el error obtenido.

Como podemos apreciar, se aplica una constante η conocida como la tasa de aprendizaje. Esta constante se encarga de controlar los "saltos"que da nuestro modelo por el gradiente del error. Conforme entreno el modelo, reduzco esta tasa para evitar que las actualizaciones de los pesos sean demasiado grandes y nos pasemos del mínimo objetivo. La ecuación 5 es la que utilizo para calcular la tasa de aprendizaje para una época t en función de la tasa inicial η_0.

$$\eta_t = \frac{\eta_0}{1 + \frac{t}{2}} \tag{5}$$

Si las neuronas de estas nuevas capas ocultas usan también una función de activación lineal como la de la ecuación 2, seguiríamos teniendo un clasificador lineal como ocurría con el perceptrón. Es por ello que debemos introducir una función de activación no lineal, que en mi caso será la función logística (ecuación 6).

$$y_i = \frac{1}{1 + e^{-z_i}} \tag{6}$$

Construyo entonces una red con una capa oculta de 256 neuronas y una capa de salida de 10 neuronas, todas ellas usando la función logística como función de activación. Cambian las salidas, que ya no van a ser 0 o 1, si no que son valores entre 0 y 1, por lo que interpretamos que la clase predicha por el modelo es la salida más alta. Obtengo los siguientes resultados:

$\acute{E}pocas$	$Tiempo$	$Error_{train}$	$Error_{test}$
10	1631,12s	1,57 %	2,89 %

Para intentar mejorar estos resultados, cambio la capa de salida por una capa *softmax*, que es más apropiada para los problemas de clasificación en los que hay más de dos clases. Las neuronas de la capa de salida van a utilizar entonces la función de activación *softmax* (ecuación 7), la cual hace que la suma de todas las salidas sea igual a 1. La salida de cada neurona va a ser entonces una probabilidad de que la entrada pertenezca a la clase que representa la neurona.

$$y_i = \frac{e^{z_i - c}}{\sum_{j=1}^{n} e^{z_j - c}} \tag{7}$$

Al usar exponenciales, los valores pueden dispararse y he llegado a experimentar problemas de *overflow*. Es por ello que uso la mayor de las entradas netas de la capa de salida como constante c para controlar el tamaño de estas exponenciales.

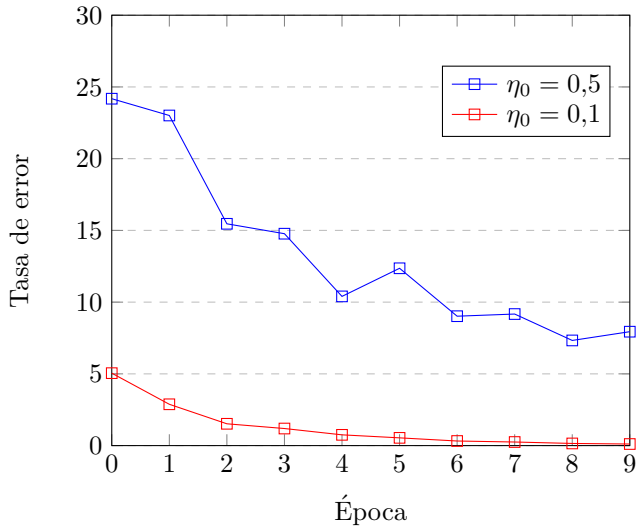
Como uso esta nueva función, debo cambiar también la función de coste del modelo por la de la entropía cruzada (ecuación 8).

$$C = -\sum_{j} t_j \log y_j \tag{8}$$

Usando una capa oculta de 256 unidades logísticas y una capa de salida *softmax* de 10 clases, obtengo los siguientes resultados:

η_0	$\acute{E}pocas$	$Tiempo$	$Error_{train}$	$Error_{test}$
0,5	10	1666,71s	7,94 %	8,92 %
0,1	10	1647,90s	0,11 %	2,51 %
0,1	30	4966,82s	0,0083 %	2,48 %

En el primer experimento obtuve resultados peores que los obtenidos con el modelo anterior. Esto se debía a que estaba usando una tasa de aprendizaje inicial demasiado elevada. Al reducirla, alcancé en el mismo número de épocas el 2,51 % de error sobre el conjunto de prueba. En la siguiente gráfica se puede apreciar cómo va descendiendo la tasa de error sobre el conjunto de entrenamiento al avanzar las épocas para estas dos tasas de aprendizaje iniciales.



Aumentando el número de épocas, conseguimos disminuir el error sobre el conjunto de entrenamiento casi al 0 %, pero el error sobre el conjunto de prueba apenas se ve afectado.