Inteligencia Computacional
Master en Ingeniería Informática

Práctica 1: Redes Neuronales

Luis Alberto Segura Delgado

Miércoles 9 de Diciembre de 2015

Contents

1 Introducción **3**

2 Implementación **3**

2.1 Red Neuronal Multicapa Simple 3

2.2 Mejora en BackPropagation: *BackPropagation con Refuerzo* 4

2.3 Mejora en BackPropagation: *BackPropagation Adaptativo* 4

1 Introducción

En esta práctica el objetivo es familiarizarse con técnicas de aprendizaje basadas en Redes Neuronales para resolver un problema concreto, en este caso, el reconocimiento de patrones numericos.
En clase se han visto las redes neuronales más básicas, las originales. Y a partir de estás se ha ido profundizando hasta conocer el funcionamiento y las mejoras que se han ido añadiendo para que sean capaces de resolver mejor y de forma más eficiente los problemas, resolviendo los problemas iniciales que se descubrieron en los modelos más básicos. De esta forma, ahora conocemos una técnica de aprendizaje automático más que podemos usar para aquellos problemas que se nos planteen en el futuro, siempre que aplicar redes neuronales sea una estrategia correcta para resolver el problema.

2 Implementación

El trabajo de implementación realizado ha sido principalmente desarrollar por mi mismo una red neuronal multicapa y entrenarla ajustando los diferentes parámetros (tasa de aprendizaje, neuronas de cada capa, capas ocultas..) de forma que se obtuviese la menor tasa de error en la clasificación de los ejemplos del problema propuesto.

2.1 Red Neuronal Multicapa Simple

En primer lugar se ha implementado una red neuronal multicapa sencilla. La implementación se ha realizado en un nuevo lenguaje de programación similar a C/C++ llamado *Rust*[1]. La implementación en detalle se puede ver en el código que se entrega junto a esta documentación.

La implementación es sencilla, y nos permite crear una red neuronal con el número de capas ocultas que deseemos. Simplemente tenemos que indicar el número de entradas (neuronas de entrada), el número de capas ocultas, el número de neuronas ocultas (todas las capas tendrán el mismo número de neuronas), el número de salidas de la red y la tasa de aprendizaje de la red a la hora de entrenar con BackPropagation. Automáticamente se creará una red neuronal con la topología indicada. La inicialización de los pesos es aleatoria, con valores entre -0.5 y 0.5, y el término *bias* se inicializa para todas las neuronas a 1.0.

Una vez que tenemos nuestra red neuronal creada, ésta debe poder ser ejecutada, para ellos se implementa una función *ejecutar* que simplemente realiza las operaciones de multiplicar pesos por entradas y acumularlos, y finalmente calcular la salida final de las neuronas (y) en base a la función de activación, que en este caso es la función *Sigmoide*.

$$o = \sum_{i=0}^{n} w_i x_i + b_i \tag{1}$$

$$y = \frac{1}{1 + e^{-o}} \tag{2}$$

Para cada neurona se calcula su salida aplicandole la función *Sigmoide*, haciendo propagación hacia delante, finalmente obtenemos la salida de la red para unas entrada determinadas.

Y ya solamente quedaba implementar un método para entrenar nuestra red neuronal. Este método es BackPropagation. Después de pelear mucho con las fórmulas y su implementación, finalmente parece que está implementado correctamente y funciona. El método de entrenamiento concreto es un entrenamiento *online*, es decir, los pesos de las neuronas se actualizan cada vez que le pasamos una entrada (foto de dígito) en función del error cometido al clasificar. Como sabemos, cada vez que se le pasa una entrada, se ejecuta

[1] `https://www.rust-lang.org`

la red, de forma que obtenemos la salida de la red. Se calcula el error de la clasificación y se propaga hacia atrás a todas las neuronas de la red para actualizar sus pesos en la medida en la que afectan a la salida (al error que se produce).

Una vez que tenemos esta red neuronal sencilla y su método de entrenamiento, solo queda ponerla a entrenar para empezar a obtener los primeros resultados. Tras una serie de experimentos y pruebas con diferentes configuraciones (tasa de aprendizaje, número de neuronas ocultas, etc), vimos que los errores no se reducían suficiente, solían quedar en torno al 20%, por lo que se han implementado algunas mejoras que se nos han ocurrido para tratar de reducir el error (de entrenamiento).

2.2 Mejora en BackPropagation: *BackPropagation con Refuerzo*

En primer lugar se nos ocurrió que, como nuestra red neuronal se quedaba atascada en un porcentaje de error en torno al 20% para el conjunto de entrenamiento, quizás podríamos "forzar" la red a aprender aquellos ejemplos que parecía que le costaban más. Para ello, se implementó una modificación de BackPropagation (*BackPropagation con Refuerzo*) que, en primer lugar, entrenaba una época[2]. Después de entrenar en una época solamente, se calcula la tasa de error y se guardan los ejemplos en los que se ha equivocado. Y a continuación, se entrena una época usando solamente los ejemplos en los que ha fallado. Para este nuevo entrenamiento que solamente usa los ejemplo erróneos, la tasa de error se reduce para evitar que se "olviden" los ejemplos que ya se aprendieron. El entrenamiento con los ejemplos erróneos se realiza tantas veces como se indique. De forma que, una vez que se entrena la primera vez con salidas falladas, se vuelve a calcular el error y se guardan los nuevos errores, para entrenar con los nuevos errores solamente, y en cada iteración (época de refuerzo) se reduce progresivamente la tasa de error (esto de reducir la tasa de error parece funcionar bastante bien).

Con este método de entrenamiento, nuestra red neuronal mostró mejoras, reduciendo su tasa de error en el conjunto de entrenamiento del 20% hasta un 11%. Buscando mejorar aún más la tasa de error (pues aún era alta), se implementaron nuevas ideas.

2.3 Mejora en BackPropagation: *BackPropagation Adaptativo*

[2]pasada de todos los datos de entrenamiento una vez (los dígitos 60000)