Inteligencia Artificial
Grado en Ingeniería Informática en Sistemas de Información
ENSEÑANZAS PRÁCTICAS Y DE DESARROLLO

EPD 4: Machine Learning – Redes Neuronales

Objetivos

- Implementación en Python de un algoritmo de Redes Neuronales para la construcción de un modelo de clasificación.

Bibliografía Básica

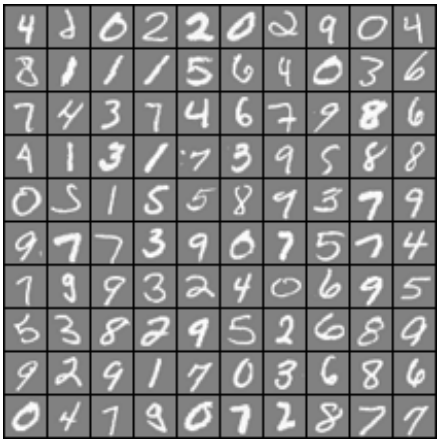
Machine Learning. Tom Mitchell. MacGraw-Hill, 1997

Ejercicios

Implementar redes neuronales para reconocimiento de dígitos escritos a mano, del 0 hasta el 9.

Dispones de 5000 ejemplos de dígitos escritos a mano en `ex4data1.mat`. La extensión. mat indica que contiene datos salvados en formato matriz Octave/Matlab nativo en vez de en formato texto. Después de cargar los datos tendrás en memoria las matrices con las dimensiones y los valores correctos.

Cada ejemplo de entrenamiento es una matriz de píxeles de 20x20 que constituye un dígito en una escala de grises. Un píxel se representa por un número decimal que indica la intensidad de gris en una posición determinada. Por tanto, la dimensión de `X` será de 5000x400, donde cada fila es un ejemplo de entrenamiento de una imagen con un dígito escrito a mano.



El vector `y` contiene las etiquetas del conjunto de entrenamiento, de manera que los dígitos del 1 al 9 están etiquetados con su propio dígito, mientras que el 0 se etiqueta con el 10. En la figura de la derecha puedes ver una muestra de los datos.

Utiliza el script `main.py` para ir incorporando el código y las llamadas a las funciones que se piden en los siguientes ejercicios. Ya están escritas las instrucciones para cargar los datos de entrenamiento y también para recuperar los parámetros `theta` de una red ya entrenada (`ex4weights.mat`), comprobar las dimensiones de las distintas matrices. Se usarán 25 neuronas en la capa oculta.

EJ1. Implementa la función de coste para una red neuronal (`nnCostFunctionSinReg.py`). En este ejercicio sólo se pide el coste según:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

es decir, sin regularizar. La llamada a esta función utilizando los parámetros suministrados `Theta1` y `Theta2` debe devolver un coste de `0.28763`.

EJ2. Implementa la función `nnGradFunctionSinReg.py` para que devuelva el gradiente sin regularización. En el script `main.py` se realiza una llamada a la función `checkNNGradients` para comprobar si la implementación ha sido correcta. Esta función compara los gradientes calculados usando back-propagation y usando una aproximación numérica. Las diferencias deberían ser inferiores a 1e-9.

EJ3. Como se ha visto en EB, en el entrenamiento de una red neuronal es importante inicializar aleatoriamente los parámetros `theta`. Implementa la función `randInitializeWeights(hidden_layer_size, num_labels)` con un `épsilon` de `0.12`.

EJ4. En este punto, ya tienes implementado todo lo necesario para entrenar la red neuronal. Para obtener un buen conjunto de parámetros, utiliza la optimización de la librería scipy, vista en otras ocasiones. Estos optimizadores avanzados son capaces de entrenar a nuestras funciones de costo eficientemente, siempre y cuando les proporcionemos los cálculos del gradiente.

EJ5. Después de haber entrenado la red neuronal, utilízala para predecir las etiquetas. Implementa la función `predict` para predecir las etiquetas del conjunto de entrenamiento, de manera que devuelva un vector que contenga valores entre 1 y el número de etiquetas posibles.

Se aconseja el uso de la función `argmax` de la librería numpy para devolver el índice del elemento máximo.

Muestra la exactitud obtenida calculando el porcentaje de ejemplos clasificados correctamente. Si la implementación es correcta, debería indicar una exactitud de 94,9%, aunque podría variar sobre un 1% debido a la inicialización aleatoria.

Problemas

P1. Implementa la función de coste con regularización. Utilizando los parámetros `Theta1` y `Theta2` cargados inicialmente, y con `lambda` igual a `1`, debe devolver un coste de `0.383770`.

P2. Implementa una función que divida en dos partes los conjuntos de datos `X` e `y` de `ex4data1.mat`, una parte que sea el conjunto de entrenamiento y otro de test, contiendo el primero el 70% de los ejemplos elegidos aleatoriamente, mientras que el segundo contendrá el resto. Se aconseja que los subconjuntos sean estratificados, es decir, que se haga la partición 70-30 por cada etiqueta de la clase del conjunto. Al finalizar deberías tener `Xtrain`, `Xtest`, `ytrain` e `ytest`.

A continuación, comprobar los resultados entrenando la red neuronal con los nuevos conjuntos de entrenamiento y haciendo la predicción sobre los conjuntos de test. Al igual que en el EJ5, predecir de nuevo con los mismos conjuntos de entrenamiento y comparar con los resultados obtenidos

P3. Implementa una clasificación multiclase “One-vs-all” con clasificadores de regresión logística regularizada, un clasificador por cada clase. Completa el código en oneVsAll.m para entrenar un clasificador por cada clase, de manera que devuelva una matriz donde cada fila corresponda con los parámetros theta para una clase. Utilizar el conjunto de entrenamiento obtenido en el paso anterior y realiza la predicción con el conjunto de test. Finalmente, compara los resultados obtenidos con los resultados del problema anterior.

P4. Implementa una clasificación multiclase “One-vs-rest” (también llamada “One-vs-all”) con el método de regresión logística de la librería sklearn donde se implementa un clasificador por cada clase. Para ello debe indicar al constructor el parámetro multi_class='ovr'. Realiza la predicción con el conjunto de test y compara los resultados obtenidos con los resultados del problema anterior.