Proyecto 3

Redes Neuronales BPN

Descripción

Hacer una función que logre hacer clasificación multiclase para reconocer dígitos escritos a mano. La función implementa una red neuronal feedforward con entrenamiento Backpropagation, mejor conocida como BPN. El reconocimiento de dígitos es muy utilizado por ejemplo para clasificar las cartas de acuerdo a su Código Postal o cantidades de cheques en los bancos.

Datos

- El archivo **dígitos.txt**, contiene 5000 ejemplos de dígitos escritos a mano (es el mismo que para la tarea de OneVsAll).
- Cada ejemplo de entrenamiento es una imagen en escala de grises en el rango [-1,1] (-1 es negro, +1 es blanco), de 20 X 20 pixeles, y está etiquetado con la clase a la que pertenece que es un número del {1..10} (el 10 es para el 0).
- Cada imagen se desenrolló en una fila de 400 datos, por lo que cada renglón contiene un ejemplo y la clase a la que pertenece.
- El archivo lo deben separar en dos matrices:
  1. El vector **X** de 5000 X 400, conteniendo TODAS las entradas. Se refiere a los pixeles de las imágenes. Se le debe agregar la columna de 1's para procesarla.
  2. El vector **y** de 5000 X 1, conteniendo TODAS las salidas. Se refieren al dígito que contiene la imagen.

Fuente: A. Ng. Machine Learning. Curso de Coursera-Stanford (2011).

Modelo de Representación

El objetivo es implementar una RN que reconozca dígitos escritos a mano. La red se entrenará utilizando Backpropagation y la arquitectura será la mostrada en la figura 1.

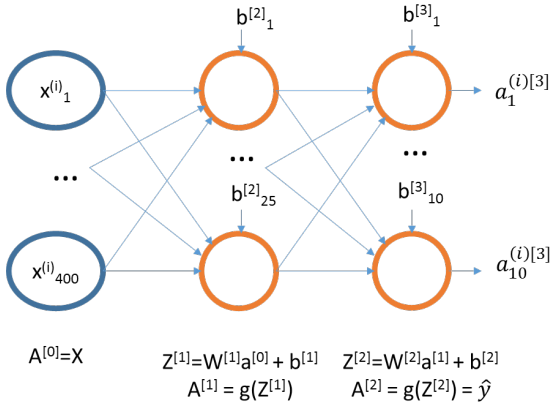

Figura 1. Arquitectura de la RN utilizada para el reconocimiento de dígitos.

La RN cuenta con 3 layers, uno de entrada, un oculto y uno de salida. El número de unidades en el layer de entrada es 400 (sin incluir el bias unit) debido a que recibe imágenes de 20 X 20 pixeles. El layer oculto cuenta con 25 unidades (sin contar la bias) y el de salida 10 unidades, una para cada clase (dígitos).

Funciones Solicitadas

Aunque se solicitan 4 funciones, la función más importante es llamada **entrenaRN**.

A continuación se explican las 3 funciones solicitadas:

1. **entrenaRN(input_layer_size, hidden_layer_size, num_labels, X, y)**.

   En este caso, como es una RN con pocos *layer*, los parámetros se pueden manejar en forma individual como W1, b1, W2 y b2, lo mismo para las derivadas que hagan falta.

   NOTA:
   Cuando hay varias capas, los parámetros se pueden manejar por medio de un diccionario llamado params, el cual está indexado por un string "Wn" o "bn", donde n es el número de layer. Los mismo se puede hacer para las derivadas con "dZn", dWn" y "dbn".

   La descripción de los parámetros recibidos es la siguiente:

   input_layer_size: número de unidades de la capa de entrada.
   hidden_layer_size: número de parámetros de la capa oculta.
   num_labels: número de etiquetas en los ejemplos, en este caso es 10, una para cada dígito.

   Recuerde que la función de costo sin regularización es:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(g(z_k^{(i)[K]})\right) - \left(1-y_k^{(i)}\right)\log\left(1-g(z_k^{(i)[K]})\right)\right]$$

   Para este ejercicio se utilizará activación sigmoidal para calcular la $g\left(z^{[3]}\right)$ y K = 10, es el total de posible etiquetas. Recuerde que $a_k^{[3]}$, es la activación de la salida de la k-ésima unidad. También recuerde que la salida ***y*** actualmente es una etiqueta pero que la que realmente vamos a usar para entrenar la red es un vector de 0's que sólo tiene un 1 en el lugar que corresponde a la etiqueta del ejemplo actual. Por ejemplo, si el ejemplo i tiene como etiqueta un 5, y el cero lo ponemos en el lugar 10, su vector y para entrenamiento será:

$$y = [0,0,0,0,1,0,0,0,0,0]^T$$

NOTA:
Se darán 20 puntos extras sobre este proyecto al que entregue el código trabajando para bases de datos de cualquier dimensión y cualquier número de etiquetas.

Después del entrenamiento el costo promedio debe ser de alrededor de 0.287629.

2. **sigmoidalGradiente(z)**. El gradiente para la función sigmoidal se calcula de la siguiente manera:

$$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z))$$

Donde

$$g(z) = \frac{1}{1 + e^{-z}}$$

Para probar la función, con valores grandes, tanto positivos como negativos, el valor del gradiente de la sigmoidal debe ser cercano a 0. Para z=0, el valor debe ser 0.25.

3. **randInicializacionPesos(L_in, L_out)**. Inicializa aleatoriamente los pesos de una capa que tienen L_in entradas (unidades de la capa anterior, sin contar el bias) y L_out salidas (unidades de la capa actual). La inicialización aleatoria se hace para evitar la simetría. Una buena estrategia es generar valores aleatorios en un rango de $[-\varepsilon_{init}, \varepsilon_{init}]$. Utilice una ε=0.12. Este rango garantiza que los parámetros se mantienen pequeños y hacen el aprendizaje más eficiente.

4. **prediceRNYaEntrenada(X,W1,b1,W2,b2).** Esta función simplemente va a crear una red neuronal muy específica que ya fue previamente entrenada. Recibe 4 matrices, **W1** de 25 X 400 (porque son 400 neuronas en la capa de entrada), **b1** de 400 X 1, **W2** de 10 X 25 (porque son 25 neuronas en la capa intermedia) y **b2** de 25 X 1, que contienen los pesos para las capas intermedia y de salida más el *bias* (**b**), respectivamente.
Recibe un vector **X** que contiene los ejemplos que se desean clasificar, en este caso, cada ejemplo contiene 400 features (1 por pixel de la imagen). Recuerde que el vector X.
La función regresa un vector **y** que contiene la predicción de las clases para todos los ejemplos en el vector **X**. Estas clases se colocan con la salida de la neurona k de la capa de salida, llamada $(a(z^{[K]}))_k$, que tenga el valor más grande.
Si prueba esta función con el archivo de entrenamiento, y las W's y b's encontradas en el entrenamiento de la RN, debe darle correctos el 97.5% de los ejemplos.

Backpropagation intuitivo

La idea intuitiva detrás del algoritmo de backpropagation es como sigue. Dado un ejemplo de entrenamiento $(x^{(t)}, y^{(t)})$, primero hacemos una propagación hacia adelente (*feedforward*) para calcular todas las activaciones de la red incluyendo el valor de salida de la hipótesis $g(z)$. Luego, para

cada nodo j en la capa l, calcular un término de error $\delta_j^{(l)}$ que mide cuánto ese nodo es responsable de algún error en la salida.
Para un nodo de salida se puede calcular directamente el error mediante la diferencia entre la activación de la red y la salida deseada. Este cálculo se usa para calcular $\delta_j^{(3)}$ (porque la capa de salida es la capa 3).
Para las unidades de la capa oculta se debe calcular $\delta_j^{(l)}$ basado en un promedio ponderado de los errores de los nodos en el layer $(l+1)$.

Verificación del Gradiente (opcional)

Suponer que se tiene una función $f_i(\theta)$ que supuestamente calcula $\frac{d}{d\theta_i}J(\theta)$. Sería bueno verificar que f_i está calculando correctamente los valores de la derivada. Sea:

$$\theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \quad \text{y} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

es decir $\theta^{(i+)}$ es lo mismo que θ salvo el i-ésimo elemento que ha sido incrementado por ϵ y $\theta^{(i+)}$ igual pero decrementado. Ahora se puede verificar numéricamente si la función es correcta, verificando para cada i que se cumpla:

$$f_i(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon}$$

El grado en que los dos valores se deben aproximar depende de los detalles de J. Pero asumiendo $\epsilon = 10^{-4}$ normalmente se encontrará que son iguales al menos hasta 4 dígitos significativos (y es muy común que sean muchos más).
De esta forma se puede comprobar que la función gradiente es correcta. Una función **checkNNGradients.m** que implemente esta verificación es opcional pero vale la pena programarla. Esto se probaría antes de la regularización. Si la implementación de backpropagation es correcta se encontrarán valores de diferencia menores a 1e-9.

En producción: reconociendo dígitos

Una vez ya entrenada la red, es decir, ya obtenidos los valores adecuados para los pesos, la red se puede poner en producción de la misma forma que se hizo en la tarea anterior, esto es, llamando directamente a la función **prediceRNYaEntrenada(X,W1,b1,W2,b2)**. La forma de usarla se explica en la definición de la función que se hizo anteriormente.