Tarea 8

Redes Neuronales BPN

Objetivo

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 **ex4data1.mat**, 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 (0 a 255) de 20X20 pixeles.
- Cada imagen se desenrolló en una fila de 400 datos.
- Cada ejemplo está etiquetado del 1 al 10 (el 10 es para el 0).
- 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.
- El archivo **ex4wigths.mat**, contiene los pesos de una RN ya entrenada para reconocer dígitos, con la misma arquitectura que la solicitada en esta actividad.

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.

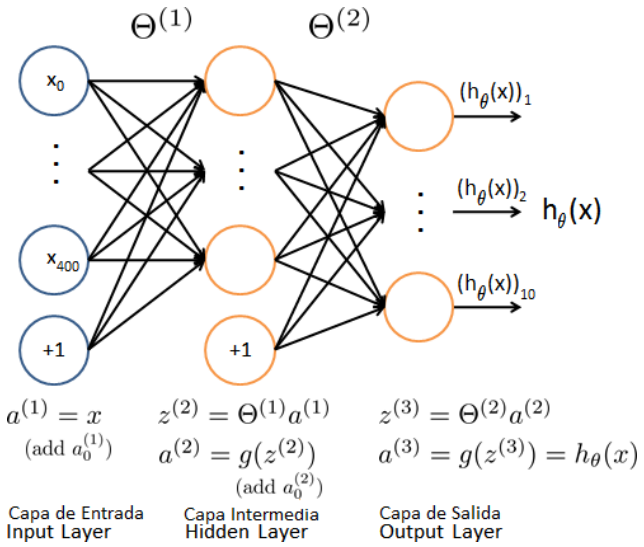

Figura 1. Arquitectura de la RN utilizada para el reconocimiento de dígitos.

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 cuento con 25 unidades (sin contar la bias) y el de salida 10 unidades, una para cada clase (dígitos).

Funciones Solicitadas

Aunque se solicitan 3 funciones, la función más importante es llamada **rnFuncionCosto**, al cual calcula el costo y el gradiente aproximado de una RN. Se recomienda implementarla en tres partes:

a. Función SIN REGULARIZACIÓN
b. Agregar REGULARIZACIÓN
c. Agregar Backpropagation

A continuación se explican las 3 funciones solicitadas.

1. **rnFuncionCosto(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, lambda)**.
   Parte 1: SIN REGULARIZACIÓN.
   Los parámetros de la RN vienen enrollados en **nn_params** y deben ser regresados a su forma de matriz en Theta1 y Theta2 que son los pesos para nuestros 2 layers (oculto y salida). El parámetro **grad** debe ser un vector enrollado de las derivadas parciales de la RN. Para desenrollar los parámetros en matrices se usa la instrucción **reshape**:

   ```
   Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
            hidden_layer_size, (input_layer_size + 1));

   Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
            num_labels, (hidden_layer_size + 1));
   ```

   para enroller el vector grad se hace:

   ```
   grad = [Theta1_grad(:) ; Theta2_grad(:)];
   ```

   La descripción de los parámetros recibidos es la siguiente:

   nn_params: parámetros enrollados de la RN.
   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:

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

Donde $h_\theta\left(x^{(i)}\right)$ se calcula como se muestra en la figura 1 y K = 10, es el total de posible etiquetas. Recuerde que $h_\theta\left(x^{(i)}\right)_k = a_k^{(3)}$, 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:

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

El código de esta función debe trabajar para bases de datos de cualquier dimensión y cualquier número de etiquetas (suponga siempre que K ≥ 3).
Una vez codificada correctamente esta función, si se prueba con los parámetros Theta1 y Theta2 del archivo de pesos proporcionado, SIN REGULARIZACIÓN el costo debe ser 0.287629

Parte 2. AGREGAR REGULARIZACIÓN

La función con regularización es:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(h_\theta\left(x^{(i)}\right)_k\right) - \left(1-y_k^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)_k\right)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}\left(\Theta_{j,k}^{(1)}\right)^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}\left(\Theta_{j,k}^{(2)}\right)^2\right]$$

Desde luego que el código debe funcionar para cualquier número de unidades en las capas (cualquier dimensión de $\Theta^{(i)}$) y cualquier número de capas (cualquier número de Θ). Aquí se pones los números (10, 25 o 400) sólo por claridad para este ejemplo. Si se corre la función CON REGULARIZACIÓN y λ = 1, el costo debe ser 0.383770.

Parte 3: AGREGAR BACKPROPAGATION

Se requiere completar la función para que regrese un valor aproximado de **grad**. Una vez implementada la obtención del gradiente se puede utilizar una de las funciones de optimización avanzada del Octave para entrenar la red (ej. **fmincg**).

Esta parte a su vez se recomienda implementarla en 2 partes. En la primera se hace Bacpropagation sin regularización y posteriormente, en la segunda parte, se le agrega la regularización. Ver explicación de Backpropagation más adelante.

2. **sigmoidGradient(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. **randInitializeWeights(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,Theta1,Theta2).** Esta función simplemente va a crear una red neuronal muy específica que ya fue previamente entrenada. Recibe 2 matrices, **Theta1** de 25 X 401 (porque son 400 neuronas en la capa de entrada más el *bias*) y **Theta2** de 10 X 26 (porque son 25 neuronas en la capa intermedia más el *bias*), que contienen los pesos para las capas intermedia y de salida, 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 debe contener una columna de 1's.
   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 $(h_\theta(x))_k$, que tenga el valor más grande.
   Si prueba esta función con el archivo de entrenamiento, y las Thetas encontradas en el entrenamiento de la RN, debe darle correctos el 97.5% de los ejemplos.

Backpropagation

La idea 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 $h_\Theta(x)$. 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 $(l+1)$.

El algoritmo backpropagation esquemático se puede ver en la figura 2.

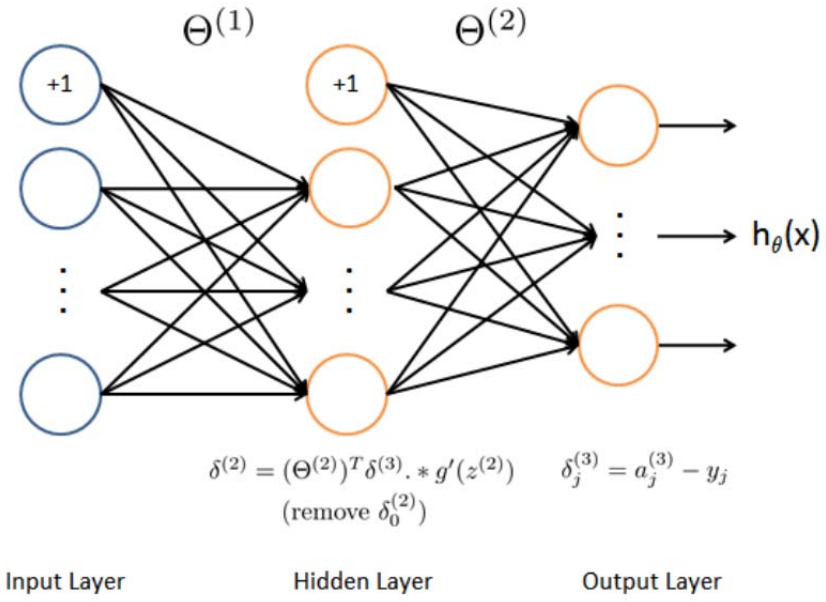

Figura 2. Algoritmo backpropagation en forma esquemática.

El algoritmo en detalle es el siguiente:

1. Colocar los valores en la capa de entrada ($a^{(1)}$) para el t-ésimo ejemplo de entrenamiento $x^{(t)}$. Realizar una proceso feedforward (como en la figura 1) para calcular las activaciones ($z^{(2)}$; $a^{(2)}$; $z^{(3)}$; $a^{(3)}$) para las capas 2 y 3. Recordar que se debe agregar un término +1 del bias en $a^{(1)}$ y $a^{(2)}$.
2. Para cada unidad de salida k en la capa 3 (capa de salida) hacer:
$$\delta_k^{(3)} = (a^{(3)} - y_k)$$
Donde $y_k \in \{1,0\}$ indica si el ejemplo actual pertenece o no a la clase k ($y_k = 1$) o si pertenece a una clase diferente ($y_k = 0$).
3. Para la capa oculta $l = 2$ hacer:
$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$
4. Acumular el gradiente de este ejemplo usando la siguiente fórmula. Notar que se debe saltar o remover $\delta_0^{(2)}$, lo cual en Octave se hace como **delta2=delta2(2:end)**.
$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$
5. Obtener el gradiente (SIN Regularización) de la función de costo de la RN dividiendo el gradiente acumulado entre $\frac{1}{m}$.
$$\frac{d}{d\Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

Se deben implementar los pasos del 1 al 4 en un ciclo que procese un ejemplo a la vez (se recomienda no vectorizar en la primera vez que se implementa el algoritmo). El paso 5 sólo divide los gradientes acumulados entre m para obtener los gradientes de la función de costo de la red neuronal.

Verificación del Gradiente

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.

Red Neuronal Regularizada

Ahora simplemente se le debe agregar regularización al gradiente. Se debe recordar que para agregar regularización a lo ya hecho en backpropagation, simplemente se le debe agregar un término extra una vez que se ha calculado el gradiente sin regularización.

Específicamente, después de que se han calculado $\Delta_{ij}^{(l)}$ usando backpropagation, la regularización se le agregar usando:

$$\frac{d}{d\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \qquad para\ j = 0$$

$$\frac{d}{d\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \quad para\ j \geq 1$$

Se debe notar que no se debe regularizar la primera columna de $\Theta^{(l)}$ el cual es usando para el término *bias*. Por eso, en los parámetros $\Theta_{ij}^{(l)}$, i es indexado iniciando en 1 y j es indexado iniciando en 0 (recordar que la indexación en Octave inicia en 1).

Ahora sí, la función **rnFuncionCosto.m** está completa y lista para usarse en el aprendizaje de los parámetros con los métodos de optimización avanzada.

Aprendizaje de los parámetros usando *fmincg*

fmincg es una function de Octave para hacer optimización parecida a fminunc que ya se había usado antes. La forma de usar esta nueva función es:

1. Establecer las opciones de optimización, las cuales sólo requiere el número máximo de iteraciones. Por ejemplo:

   options = optimset('MaxIter', 50);

   El número de iteraciones se puede modificar para controlar el aprendizaje total.
2. Establecer el valor de λ a ser usado. Por ejemplo:

   lambda = 1;
3. Se puede crear un nombre más corto para la función de costo que se desea minimizar. Por ejemplo:
   costFunction = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size, num_labels, X, y, lambda);
   para que se llame sólo como un argumento.
4. Llamar a la function:

   [nn_params, cost] = fmincg(costFunction, initial_nn_params, options);
5. Obtener **Theta1** and **Theta2** como matrices a partir de **nn_params:**

```
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));
```

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,Theta1,Theta2)**. La forma de usarla se explica en la definición de la función que se hizo anteriormente.