Proyecto 5

BPN de una neurona

Descripción

Hacer una función que logre hacer clasificación de dos clases. La función implementa una red neuronal feedforward de una sola neurona, con entrenamiento Backpropagation, mejor conocida como BPN.

Datos

- La prueba para la clasificación la puede hacer con el archivo de exámenes de admisión o con las funciones lógicas AND y OR.
- La prueba para la predicción la puede hacer con el archivo de casas con múltiples features. Pruebe sin normalización y con normalización.

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

Modelo de Representación

El objetivo es implementar una RN que clasifique en dos clases (si la activación es sigmoidal) o prediga el valor de la salida (si la activación es lineal). La red se entrenará utilizando Backpropagation y la arquitectura será la mostrada en la figura 1.

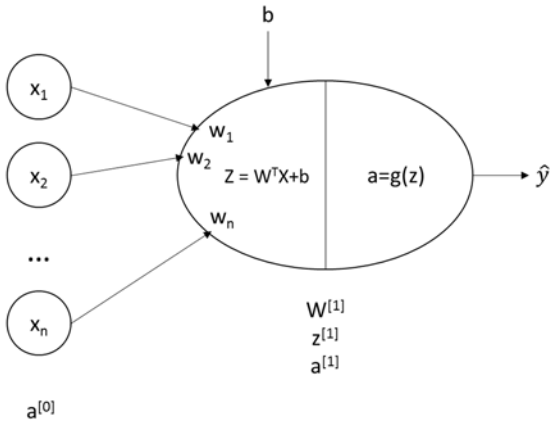
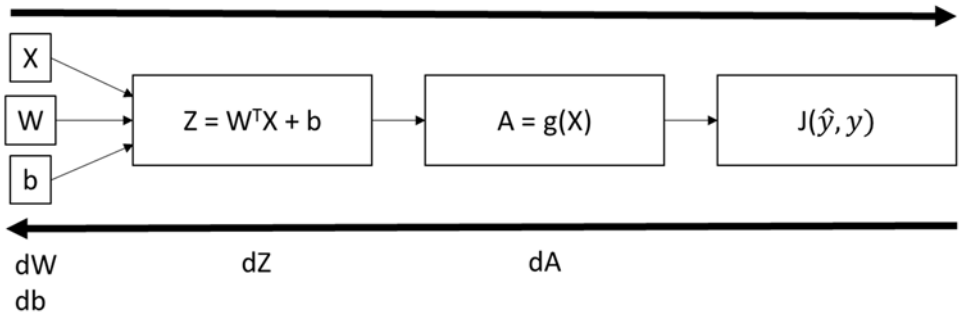


Figura 1. Arquitectura de la RN de una neurona.

La RN cuenta con 1 layer de entrada de tantas neuronas como features tengan los ejemplos, y un layer de salida de una sola neurona (como el perceptrón o el adaline).

Grafo de cálculo

El grafo de cálculo para la red neuronal dada es:



Los cálculos se hacen en dos fases:

Fase 1: Propagación hacia adelante, donde se calculan Z, A y J
Fase 2: Propagación hacia atrás, donde se calculan dA usando J, dZ usando A, dW usando dZ y db usando dZ.

Funciones Solicitadas

Aunque se solicitan 5 funciones, la función más importante es llamada **bpnUnaFuncionCosto**, al cual calcula el costo y el gradiente aproximado de una RN.

A continuación se explican las 5 funciones solicitadas:

1. **bpnUnaNeuronaSigmoidal (nn_params, input_layer_size, X, y, alpha, activacion)**.

   nn_params: parámetros de la RN (pesos).
   input_layer_size: número de unidades de la capa de entrada.
   activación: es un string que puede ser "sigmoidal" o "lineal"

   Recuerde que la función de costo sin regularización es:

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

   Donde $h_\theta\left(x^{(i)}\right)$ es la función sigmoidal. Cuando la activación es lineal la función de costo es el promedio de los errores al cuadrado (con el 2 para que se vaya en la derivada).

   Esta función modifica el vector **nn_params** usando gradiente descendente utilizando el conjunto de ejemplos de entrenamiento (**X**,**y**). Para calcular los gradientes utiliza backpropagation.

2. **sigmoidGradiente(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. **linealGradiante(z)**. El gradiente para la función lineal es 1:

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

Donde $g(z) = z$

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

5. **prediceRNYaEntrenada(X, nn_params, activacion).** Esta función simplemente va a crear una red neuronal muy específica que ya fue previamente entrenada. Recibe el vector de parámetros (pesos), **nn_params** (pesos) y un vector **X** que contiene los ejemplos que se desean clasificar, en este caso, **n** features, además recibe la función de activación utilizada. La función regresa un valor de **y** que contiene la predicción de las clases para todos los ejemplos en el vector **X**.

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 adelante (*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.

Pseudocódigo para BPN de una neurona

Para una pasa del archivo ejemplos el pseudocódigo, cuando la activación es sigmoidal, es:

```
Z = W^T X + b = np.dot(W.T, X) + b
A = σ(Z)
dZ = A − Y
dW = (1/m)*X*dZ^T
db = (1/m)*np.sum(dZ)
W = W - αdW
b = b - αdb
```

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 $\delta_j^{(1)}$ (porque la capa de salida es la capa 1).

Se deben implementar los pasos del pseudocódigo en un ciclo de **n** iteraciones o hasta convergencia, por ejemplo, para **n** ciclos:

```
for iter in range(n):
```

Si la activación es lineal, se deben hacer los cálculos de las derivadas requeridos y el pseudocódigo cambiaría un poco.

Entrega y Revisión

- Salve su archivo con el nombre **Proyecto5.py**. Tiene que ser con ese nombre porque es el que se va a usar para la revisión automática.
- Suba sus archivos dos archivos, el del proyecto (.py) y el de la documentación (.docx o .pdf) **por separado**, NO los ponga en ZIP.
- Para la revisión se harán pruebas sobre las funciones solicitadas. Para esto, tanto el archivo que contiene las funciones (el suyo) como el de prueba (el mío), se colocarán en el mismo folder. El programa de prueba al inicio hará un:

  ```
  from file import function
  ```

  y luego se utilizará en el script de prueba llamándola como como:

  ```
  function(<parámetros>)
  ```