Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Profesor: Dr.-Ing. Pablo Alvarado Moya
Introducción al Reconocimiento de Patrones
II Semestre 2018

Entrega: **25 de setiembre, 2018**

Proyecto 1

1 Introducción

En este proyecto usted implementará una red neuronal desde cero, para poder comprender los principios de funcionamiento detrás de los métodos dominates en la actualidad del aprendizaje automático. Para ello se utilizará un problema *de juguete* en dos dimensiones que facilita la visualización de resultados.

Usted deberá implementar la función de error asociada a una red neuronal, utilizar métodos de optimización para encontrar los mejores parámetros de la red neuronal usando un conjunto de entrenamiento, y luego visualizar el resultado de la clasificación. Esto se programará en GNU/Octave, utilizando en algunos puntos funcionalidades del paquete `optim`. Usted deberá evaluar el desempeño de los optimizadores utilizados.

2 Objetivos

2.1 Objetivo general

Poner en acción los principios de funcionamiento de redes neuronales artificiales, con el fin de comprender su funcionamiento.

2.2 Objetivos específicos

1. Implementar una función de error asociada a un problema de clasificación.
2. Utilizar optimizadores para minimizar la función de error considerando un conjunto de entrenamiento $\{(\underline{\mathbf{x}}^{(i)}, y^{(i)}) \mid i = 1, \ldots, m\}$.
3. Visualizar un espacio de entrada de dos dimensiones superpuesto con la salida de clasificación.
4. Utilizar matrices de confusión para evaluar los resultados de clasificación.

3 Procedimiento

3.1 Función de error

Una red neuronal se puede concebir como un grafo dirigido, en donde cada arista tiene un peso asociado y representa el producto de la entrada a esa arista con su peso. Cada

neurona (nodo del grafo) realiza la suma de todas las salidas de las aristas y luego aplica una *función de activación* al resultado.

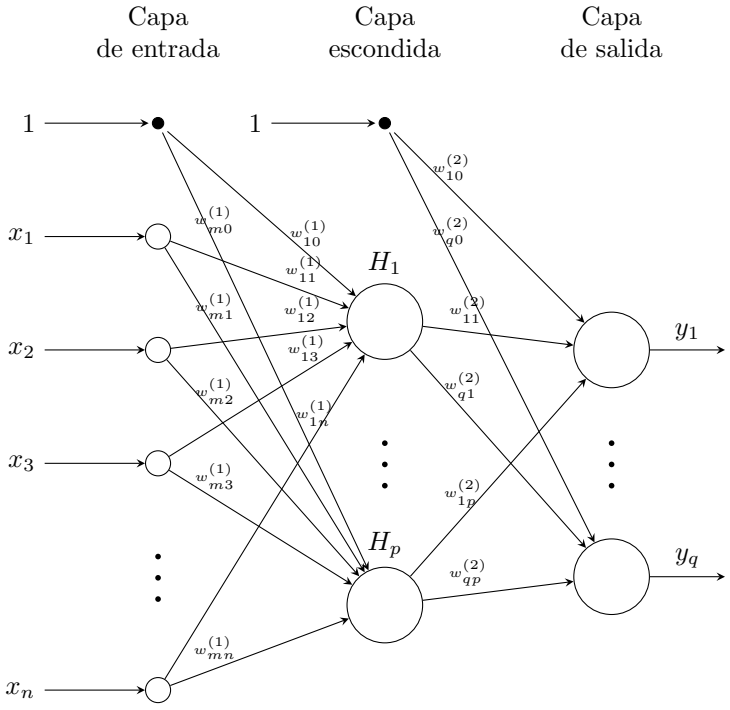

Figura 1: Red neuronal con una sola capa escondida

Por ejemplo, en la figura 1 si la neurona H_1 tiene función de activación g, entonces a su salida produce

$$y_{H_1} = g\left(\underline{\mathbf{w}}^T_{1:}\begin{bmatrix}1\\ \underline{\mathbf{x}}\end{bmatrix}\right)$$

con el vector $\underline{\mathbf{w}}^T_{1:} = \begin{bmatrix} w^{(1)}_{10} & w^{(1)}_{11} & w^{(1)}_{12} & \cdots & w^{(1)}_{1n}\end{bmatrix}$.

Como función de activación g en los planteamientos de redes neuronales clásicas usualmente se utiliza la función logística o sigmoidal

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

que tiende a cero para $x \to -\infty$ y tiende a uno para $x \to \infty$.

Si definimos que la aplicación de la función de activación a un vector equivale a aplicar la función de activación escalar a cada uno de los componentes, es decir:

$$g\left(\begin{bmatrix}x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix}\right) = \begin{bmatrix}g(x_1)\\ g(x_2)\\ \vdots\\ g(x_n)\end{bmatrix}$$

y si además agrupamos todos los pesos de la i-ésima capa en una matriz $\mathbf{W}^{(i)}$ (aquí n_i es el número de entradas a la i-ésima capa y p_i el número de neuronas de la i-ésima capa)

$$\mathbf{W}^{(i)} = \begin{bmatrix} w_{10}^{(i)} & w_{11}^{(i)} & w_{12}^{(i)} & \cdots & w_{1n_i}^{(i)} \\ w_{20}^{(i)} & w_{21}^{(i)} & w_{22}^{(i)} & \cdots & w_{2n_i}^{(i)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{p_i0}^{(i)} & w_{p_i1}^{(i)} & w_{p_i2}^{(i)} & \cdots & w_{p_in_i}^{(i)} \end{bmatrix}$$

entonces la salida $\underline{\mathbf{y}}^{(i)}$ de dicha capa ante la entrada $\underline{\mathbf{x}}^{(i)}$ es

$$\underline{\mathbf{y}}^{(i)} = g\left(\mathbf{W}^{(i)} \begin{bmatrix} 1 \\ \underline{\mathbf{x}}^{(i)} \end{bmatrix}\right) \tag{1}$$

Observe que la k-ésima fila de $\mathbf{W}^{(i)}$ contiene todos los pesos asociados a todas las aristas que *llegan* a la k-ésima neurona, mientras que la l-ésima columna de esa matriz contiene todos los pesos asociados a las aristas que *salen* de la l-ésima componente de la entrada.

De este modo, considerando que la salida de la capa i es la entrada de la capa $i+1$, la salida total ante la entrada $\underline{\mathbf{x}}$ predicha por la red neuronal en la figura 1 está dada por:

$$\underline{\hat{\mathbf{y}}}\left(\underline{\mathbf{x}}; \mathbf{W}^{(1)}, \mathbf{W}^{(2)}\right) = g\left(\mathbf{W}^{(2)} \begin{bmatrix} 1 \\ g\left(\mathbf{W}^{(1)} \begin{bmatrix} 1 \\ \underline{\mathbf{x}} \end{bmatrix}\right) \end{bmatrix}\right) \tag{2}$$

De modo similar al clasificador softmax, las neuronas de salida representan, cada una, a una clase, de modo que en los datos de entrenamiento los vectores $\underline{\mathbf{y}}^{(i)}$ son usualmente vectores unitarios canónicos, es decir, vectores cuyas componentes son iguales a cero, excepto aquella que representa la clase a la que pertenece el vector $\underline{\mathbf{x}}^{(i)}$.

Como ya es constumbre en el curso, dado un conjunto de m datos de entrenamiento $\{(\underline{\mathbf{x}}^{(j)}, \underline{\mathbf{y}}^{(j)}); j = 1 \dots m\}$ podemos definir la función de error en términos de todos los pesos de todas las capas como aquella que mide la diferencia entre lo que la red predice y lo que el conjunto de entrenamiento indica:

$$J(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}) = \frac{1}{2} \sum_{j=1}^{M} \left\| \underline{\mathbf{y}}^{(j)} - \underline{\hat{\mathbf{y}}}(\underline{\mathbf{x}}^{(j)}; \mathbf{W}^{(1)}, \mathbf{W}^{(2)}) \right\|_2^2 \tag{3}$$

Nótese el uso de la norma euclídea $\|\cdot\|_2$ para estimar la distancia entre la predicción y el valor deseado. Queremos encontrar entonces los pesos $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$ que producen el menor valor posible de J. Vimos en clase que a esto se le conoce como el problema de *mínimos cuadrados ordinarios*. Si este problema se resuelve por medio de descenso de gradiente recibe el nombre de *algoritmo de retropropagación de error* (o *backpropagation algorithm*), que es uno de los algoritmos más relevantes en la actualidad por su aplicación en el aprendizaje profundo.

La dificultad del problema reside en cómo plantear el cálculo del gradiente para realizar la optimización de J de forma eficiente.

Con el archivo `create_data.m` usted tiene a disposición una función con interfaz

```
function y=create_data(dim,numClasses,shape)

# usage [X,Y] = create_data(numSamples,numClasses=3)
#
# This function creates random examples in the two-dimensional
# space from -1 to 1 in each coordinate.  The output Y is arranged
# in numClases outputs, such that they can be used as outputs of
# artificial neurons (or units) directly.
#
# Inputs:
#   numSamples: total number of samples in the training set
#   numClasses: total number of classes in the training set
#   shape: distribution of the samples in the input space:
#          'radial' rings of clases,
#          'pie' angular pieces for each class
#          'vertical' vertical bands
#          'horizontal' horizontal bands
#
# Outputs:
#   X: Design matrix, with only the two coordinates on each row
#      (no 1 prepended).  Its size is numSamples x 2
#   Y: Corresponding class to each training sample.  Its size is
#      numSamples x numClases

# ...

endfunction;
```

para crear conjuntos de datos en diferentes distribuciones (ver figura 2). Adicionalmente

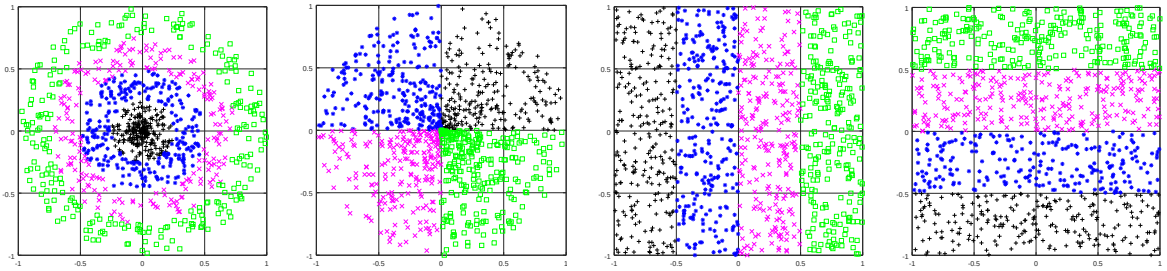

Figura 2: Distribuciones de datos de entrenamiento para primera parte del proyecto: (a) radial, (b) pie, (c) bandas verticales, (d) bandas horizontales. Los ejemplos usan 1000 datos con 4 clases.

a la matriz de datos de entrada X, la matriz Y contendrá los valores deseados en cada neurona de salida: la primera columna corresponde a las salidas de la primera neurona, la segunda columna corresponde a las salidas deseadas de la segunda neurona, y así sucesivamente. La salida de una neurona debe ser "1" si los datos de entrada corresponden a la clase que dicha neurona representa, o "0" en otro caso (observe la similitud con la salida de un clasificador softmax).

La función `plot_data.m` le permite visualizar los datos generados, tal y como los muestra la figura 2.

En este proyecto usaremos por motivos de visualización un espacio de entrada de dos dimensiones.

1. Programe una función predictora que implemente (2) con la siguiente interfaz:

```
function y=predict(W1,W2,X)

# usage predict(W1,W2,X)
#
# This function propagates the input X through the neural network to
# predict the output vector y, given the weight matrices W1 and W2 for
# a two-layered artificial neural network.
#
# W1: weights matrix between input and hidden layer
# W2: weights matrix between the hidden and the output layer
# X:  Input vector, extended at its end with a 1

# PONGA SU CODIGO AQUÍ
endfunction;
```

Usted debe implementar esta función de forma que si el usuario entrega varias entradas como filas de una matrix `X` de tamaño $m \times 2$, entonces produzca la salida `y` de $m \times$ filas$(\mathbf{W}^{(2)})$, es decir, el usuario puede indicar varias entradas, cada una en una fila de `X`, y la función debe entonces producir un vector con la salida correspondiente a cada fila de `X`.

Note que para resolver este punto usted debe implementar la función sigmoide de forma apropiada, y considerar las entradas de sesgo en cada capa (marcadas como entradas '1' en la figura 1).

2. Programe una función denominada `target` con la siguiente interfaz:

```
function y=target(W1,W2,X,Y)

# usage target(W1,W2,X,Y)
#
# This function evaluates the sum of squares error for the
# training set X,Y given the weight matrices W1 and W2 for
# a two-layered artificial neural network.
#
# W1: weights matrix between input and hidden layer
# W2: weights matrix between the hidden and the output layer
# X:  training set holding on the rows the input data, plus a final column
#     equal to 1
# Y:  labels of the training set

# PONGA SU CODIGO AQUÍ
endfunction;
```

Esta función debe evaluar (3). Observe que usted puede utilizar la función implementada en el punto anterior.

3. Programe ahora una función que calcule el gradiente de (3) con respecto a todos los pesos, evaluado sobre $\mathbf{W}^{(1)}$ y $\mathbf{W}^{(2)}$. Para ello se podrían utilizar métodos de diferenciación numérica, para derivar con respecto a cada peso (lo que es relativamente ineficiente), o preferiblemente se realiza una función programada qu eproduce directamente el vector de gradiente, lo que se obtiene con la regla de la cadena y conociendo que para la función logística se cumple $g'(x) = g(x)(1 - g(x))$:

```
function [gW1,gW2]=gradtarget(W1,W2,X,Y,batchSize=1e12)

# usage gradtarget(W1,W2,X,Y)
#
# This function evaluates the gradient of the target function on W1 and W2.
```

```
#
# W1: weights matrix between input and hidden layer
# W2: weights matrix between the hidden and the output layer
# X:  training set holding on the rows the input data, plus a final column
#     equal to 1
# Y:  labels of the training set
# batchSize: Size of the batch used to compute the gradient

# PONGA SU CODIGO AQUÍ
endfunction;
```

En Internet se encuentran bastantes fuentes de cómo realizar eficientemente esta tarea, bajo el nombre *error backpropagation*. En particular, el libro de Christopher M. Bishop *Neural Networks for Pattern Recognition* contiene una excelente descripción de este proceso, en donde ese gradiente se calcula capa por capa, desde la salida hacia la entrada (de ahí el nombre de retro-propagación).

Las dos matrices de salida de esta función tienen los mismos tamaños que las dos matrices de pesos a la entrada, y cada elemento de cada matriz contiene el valor de la derivada de la función `target` con respecto al peso correspondiente:

$$\texttt{gW1} = \begin{bmatrix} \frac{\partial J}{\partial w_{10}^{(1)}} & \frac{\partial J}{\partial w_{11}^{(1)}} & \frac{\partial J}{\partial w_{12}^{(1)}} & \cdots & \frac{\partial J}{\partial w_{1n}^{(1)}} \\ \frac{\partial J}{\partial w_{20}^{(1)}} & \frac{\partial J}{\partial w_{21}^{(1)}} & \frac{\partial J}{\partial w_{22}^{(1)}} & \cdots & \frac{\partial J}{\partial w_{2n}^{(1)}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial w_{p0}^{(1)}} & \frac{\partial J}{\partial w_{p1}^{(1)}} & \frac{\partial J}{\partial w_{p2}^{(1)}} & \cdots & \frac{\partial J}{\partial w_{pn}^{(1)}} \end{bmatrix}$$

En este proyecto usted utilizará un entrenamiento por mini-lotes, que es la combinación del decenso de gradiente por lotes y el estocástico. Así, su gradiente se calcula utilizando solo un subconjunto de elementos con un número de datos especificado por `batchSize`, que se seleccionan aleatoriamente de todo el conjunto `X` proporcionado. Si el número indicado es mayor que el total de filas de `X` entonces símplemente se usan todos los datos disponibles y el mini-batch será equivalente a lo usado en el aprendizaje por lotes. Si `batchSize`= 1 entonces el cálculo es equivalente al utilizado en aprendizaje estocástico.

3.2 Entrenamiento de la red

Usted utilizará dos estrategias para entrenar su red.

Para poder simplificar la manipulación de los datos, usted debe empaquetar todos los pesos en $\mathbf{W}^{(1)}$ y $\mathbf{W}^{(2)}$ en un solo vector $\underline{\boldsymbol{\omega}}$ en una función denominada `packweights`. También implemente una función `unpackweights` que desempaquete el vector en las dos matrices correspondientes. Para esto puede serle útil la función `reshape` de GNU/Octave.

1. Primero usted entrenará su red utilizando el descenso de gradiente usando mini-

lotes. Para ello implemente un programa iterativo que actualice los pesos:

$$\underline{\boldsymbol{\omega}} \leftarrow \underline{\boldsymbol{\omega}} - \lambda \nabla_{\underline{\boldsymbol{\omega}}} J(\underline{\boldsymbol{\omega}})$$

donde el gradiente se calcula utilizando un número limitado de elementos del conjunto de datos de entrenamiento. Usted debe decidir cómo y cuándo terminar las iteraciones.

Pruebe varios valores de λ y de `batchSize` y evalúe cuántas iteraciones se requieren para converger.

Implemente en su método una forma de capturar la evolución del error (el valor de la función `target`, también conocida en este contexto como *loss function*) durante todo el proceso en función del número de iteración. Las gráficas de caída de este error brindan información valiosa para optimizar los parámetros usados en el entrenamiento.

2. El segundo método que va a utilizar es el de gradientes conjugados. Revise cómo utilizar el método `cg_min` en el paquete `optim` de octave.

3. Cuente el número de evaluaciones de la función de error para este método también.

3.3 Visualización de los resultados

Para visualizar los resultados de la clasificación usted va a generar una imagen que representa con colores la clasificación en cada punto de dicha imagen. La imagen es cuadrada y representa el rango vertical y horizontal de los datos de entrenamiento, es decir, de -1 a 1 en ambas direcciones.

En GNU/Octave las imagenes son matrices $M \times N \times 3$ con M el número de filas, N el número de columnas y los tres canales representan las componentes rojo, verde y azul, con valores entre 0 a 1.

Para generar una representación visual de la clasificación alcanzada, revise el código brindado como ejemplo en la lección 8 para el clasificador softmax. En ese ejemplo se cuenta con una paleta de colores, donde cada color representa a una de las clases. En el ejemplo de softmax se sabe que la suma de todas las salidas del clasificador es uno, por lo que allí basta con multiplicar cada color de la paleta por la correspondiente probabilidad de cada clase, y sumar esos resultados para obtener el color asignado a la clasificación.

En el caso actual, cada clase produce una salida entre 0 y 1, pero la suma de todas las salidas no es necesariamente igual a 1, por lo que la combinación de colores debe hacerse utilizando las salidas normalizadas para simular probabilidades, es decir, cada salida de neurona se divide por la suma de todas las salidas.

Usted debe realizar el mapeo entre las coordenadas de la imagen y el espacio entrada de la red neuronal (de -1 a 1). Su imagen deberá tener un tamaño que permita visibilizar con detalle el resultado de la clasificación, como por ejemplo, 512×512. Usted debe recorrer entonces todos los píxeles de la imagen, calcular los valores de entrada correspondientes, clasificar dicha entrada y utilizar los valores a las neuronas de salida como los valores que

escalan a la paleta de colores de las clases. Con la función `imshow` muestre su imagen y sobrepóngale los puntos de entrada.

3.4 Evaluación de los resultados

1. Para evaluar el desempeño de su algoritmo, genere otro conjunto de datos (distinto del conjunto de entrenamiento) con la misma distribución de puntos que los datos de entrenamiento.

2. Investigue qué es una matriz de confusión.

3. Utilice su función de predicción para predecir cada punto en el conjunto de prueba y cree la matriz de confusión correspondiente, donde las filas deben tener las clases reales y las columnas las clases predichas.

4. Investigue qué métricas de evaluación de clasificación se pueden derivar de la matriz de confusión, en particular la sensitividad, la precisión, el valor F1, y calcúlelas para sus datos.

4 Entregables

El código fuente y el artículo científico deben ser entregados según lo estipulado en el programa del curso.

Incluya un archivo de texto README con las instrucciones para la ejecución de los programas.

En el artículo concéntrese en la evaluación del proceso de entrenamiento y resultados cuantitativos del entrenamiento con los distintos métodos de optimización implementados.