Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Profesor: Dr.-Ing. Pablo Alvarado Moya
Introducción al Reconocimiento de Patrones
I Semestre 2020

Entrega: **3 de abril, 2020**
Versión: 10 de marzo,2020

Proyecto 1: Redes neuronales artificiales y el algoritmo de retropropagación

1 Introducción

En este proyecto usted implementará una red neuronal desde cero, para poder comprender los principios de funcionamiento detrás de los métodos dominates del aprendizaje automático en este instante. Para ello se utilizará un problema *de juguete* en dos dimensiones que facilita la visualización de resultados. Si los métodos son realizados correctamente, en realidad la red podrá aplicarse a cualquier tipo de problemas.

Usted deberá implementar la función de error asociada a una red neuronal por medio de un grafo computacional. Este grafo facilita el cálculo del gradiente por retropropagación, que será utilizado por los métodos de optimización para encontrar los mejores parámetros de la red neuronal usando un conjunto de entrenamiento. Luego podrá visualizar el resultado de la clasificación de datos en un área densa, lo que permitirá observar qué tan bien generaliza su red.

Esto se programará en GNU/Octave. Usted deberá evaluar el desempeño de la red, variando varios de los hiperparámetros que tendrá a disposición.

2 Objetivos

2.1 Objetivo general

Implementar redes neuronales artificiales por medio de grafos computacionales, con el fin de comprender su funcionamiento en detalle.

2.2 Objetivos específicos

1. Implementar un grafo computacional para representar la función de error asociada a un problema de clasificación.
2. Utilizar optimizadores para minimizar una 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 Conceptos y definiciones

Una red neuronal se modela como un grafo dirigido, en donde cada arista multiplica el peso que tiene asociado con el valor a su entrada. Cada *neurona* (nodo del grafo) realiza la suma de todas las salidas de las aristas y luego aplica una *función de activación* al resultado.

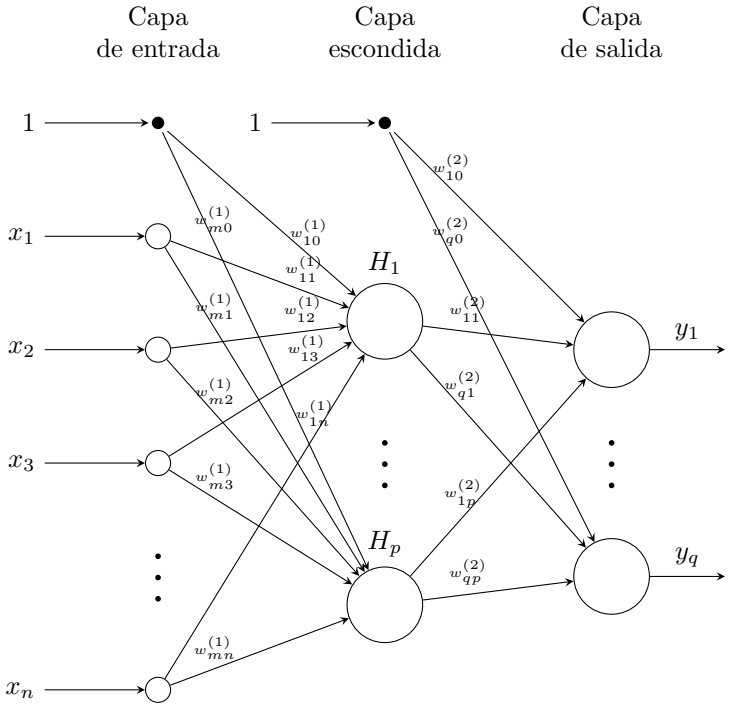


Figura 1: Red neuronal con una sola capa escondida

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 $x \to \infty$. En redes neuronales modernas se utiliza con frecuencia la función de activación ReLU (*rectified linear unit*)

$$g(x) = \max(0, x)$$

pero hay otras opciones disponibles.

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:

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

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

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

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:

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

lo que se puede generalizar fácilmente para κ capas.

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 codificados *one-hot* (uno encendido), 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:

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

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 $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(\kappa)}$ 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, en donde utilizamos grafos computacionales para el cálculo del gradiente, entonces tenemos el *algoritmo de retropropagación de error* (o *error backpropagation algorithm*), que es uno de los algoritmos más relevantes en la actualidad por su aplicación en el aprendizaje profundo.

En este proyecto se deberá utilizar el descenso estocástico de gradiente, lo que implica que se calculará el gradiente para un subconjunto de los datos disponible (mini-lote).

La implementación deberá implementar una capa totalmente conectada con sesgo, capas de activación sigmoidal, ReLU, SoftMax y alguno de los otros tipos vistos en clase. Además, deberá derivarse teóricamente e implementarse la capa para el cálculo del error de un mini-lote.

4 Datos

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

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,shape)
#
# This function creates random examples in the two-dimensional
# space from -1 to 1 in each coordinate.  The output Y is arranged
# in numClasses 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
#          'curved' curved pies
#          'spirals' interlaced spirals
#
# 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 numClasses

# ...

endfunction;
```

para crear conjuntos de datos en diferentes distribuciones (ver figura 2). La matriz de diseño `X` sigue la convención de que cada dato se almacene en una fila de `X`.

La matriz `Y` contendrá en su i-ésima fila la clase a la pertenece el dato en la i-ésima

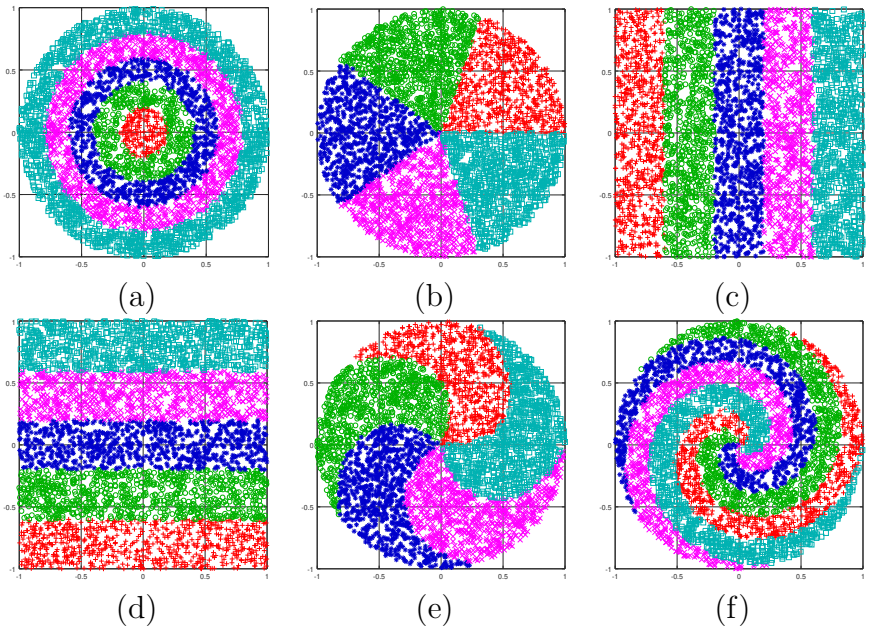


Figura 2: Distribuciones de datos de entrenamiento para primera parte del proyecto: (a) radial, (b) pastel, (c) bandas verticales, (d) bandas horizontales, (e) pastel curvos, (f) espirales. Los ejemplos usan 4000 datos con 5 clases.

fila de `X`. La clase está codificada en *one-hot*, es decir, como vector, de modo que solo aquella componente correspondiente a la clase es 1 y el resto 0. En otras palabras, la primera columna de `Y` corresponde a las salidas de la primera neurona que representa a la clase 1, la segunda columna corresponde a las salidas deseadas de la segunda neurona que representa a la clase 2, 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.

Los conjuntos de datos vertical, horizontal y pastel por su separabilidad lineal son relativamente simples y los puede usar para fines de depuración de su código con redes de una capa (horizontal y vertical) o de dos capas (pastel). Cuando estos casos le funcionen, pueden probarse los casos radial, pastel curvo y espirales, que tienen una dificultad mucho mayor y requieren por lo general más de dos capas.

5 Consideraciones para el código

Utilice orientación a objetos. La complejidad del proyecto se mantiene fácilmente bajo control si cada capa se encapsula en su propia clase.

Además, se recomienda fuertemente encapsular el modelo mismo de la red neuronal, de modo que las capas sean propiedades de la clase, y los métodos de entrenamiento y predicción sean parte de esa clase. Esto simplifica enormemente la manipulación de la

red neuronal.

Por ejemplo, en un archivo `model.m` usted puede definir algo como:

```
classdef model < handle
## Modelo
##
## Esta clase encapsula una red neuronal, con métodos para
## almacenar, cargar, entrenar y predecir.

properties
  ## Constantes:
  numInputs=2;
  hiddenNeurons1=8;
  ## ...

  ## Training parameters
  nEpochs=2000;
  miniBatch=128;

  alpha=0.01;      ## Learning rate
  ## ...

  ## Capas:
  l1a=fullyconnectedbiased();
  l1b=relu();

  l2a=fullyconnectedbiased();
  l2b=sigmoide();

  l3a=fullyconnectedbiased();
  l3b=sigmoide();

  l4=olsloss();

  ## Pesos
  W1=[];
  W2=[];
  W3=[];
endproperties

methods
  function s=model()
    s.init()
  endfunction

  function init(s)
    ## Inicialice el modelo con valores al azar
    ## ...

  endfunction

  function losslog=train(s,X,y,valSetX=[],valSetY=[])
    ## Entrene el modelo
    ## X: matriz de diseño (datos de entrenamiento en filas)
    ## y: matriz de salida, cada fila codificada one-hot
    ## valSetX: set de validación (opcional) (entradas en filas)
    ## valSetY: set de validación (opcional) (salidas en filas)
    ## losslog: protocolo con pérdida por época, para set de
    ##          entrenamiento y opcionalmente el set de validación

    ## ...

  endfunction


  function y=test(s,X)
  ## Predicción con modelo preentrenado
```

```
    ## ...
  endfunction

  function save(s,file)
    ## Trucos para salvar las matrices

    W1 = s.W1;
    W2 = s.W2;
    W3 = s.W3;

    save(file,"W1","W2","W3");
  endfunction

  function o=load(s,file)
    ## Cargue las matrices salvadas anteriormente y asegure consistencia
    ## de parámetros

    load(file);

    s.W1=W1;
    s.W2=W2;
    s.W3=W3;

    s.numInputs=columns(s.W1)-1;
    s.hiddenNeurons1=rows(s.W1);
    s.hiddenNeurons2=rows(s.W2);
    s.numClasses=rows(s.W3);

  endfunction

endmethods
endclassdef
```

donde se ha asumido que usted tiene sus capas `relu`, `sigmoide` y `fullyconnectedbiased` ya definidas, de forma similar al ejemplo visto en clase.

Con un modelo de red definido anteriormente, en otro archivo `train.m` usted podría usar lo anterior:

```
numClasses=5;
[X,Y]=create_data(500,numClasses,'spirals');   ## Training

ann=model();

ann.numClasses = numClasses;
ann.nEpochs    = 2500;
ann.alpha      = 0.01;  ## Learning rate

file="weights.dat";

if (exist(file,"file")==2)
  ann.load(file); ## Si archivo existe, carge los parámetros aprendidos
else
  ann.init();     ## Si no, use pesos al azar
endif

loss=ann.train(X,Y,vX,vY);  ## Entrene

ann.save(file);             ## Salve la red entrenada
```

Obviamente usted debe definir sus capas apropiadamente para que algo como lo anterior pueda ser usado.

De forma similar puede definir otro archivo para la predicción que genere las figuras

solicitadas.

6 Procedimiento

6.1 Grafos computacionales

1. Diseñe e implemente una capa totalmente conectada con sesgo como nodo computacional, que recibe una matriz de diseño `X` con el mini-lote de datos, y los pesos `W`. La capa debe generalizar el concepto resumido en (1) para ser aplicada a varias muestras a la vez (el número de filas de `X`). La capa además debe permitir realizar la retropropagación, recibiendo un gradiente de J respecto a la salida de la capa, y produciendo los gradientes correspondientes de J respecto a `X` y `W`.

   Puede usar como punto de partida el ejemplo visto en clase para una neurona.

2. Diseñe e implemente capas de activación, cada una por separado, que reciben la salida de una capa totalmente conectada y producen las correspondientes activaciones. Debe considerar que las capas totalmente conectadas entregarán resultados para un mini-lote completo. Estas capas deberán además tener la funcionalidad para retropropagar el error, como corresponda. Las funciones de activación a implementar son el sigmoide, ReLU, SoftMax y por lo menos una función adicional de las vistas en clase.

3. Diseñe e implemente dos capas para el cálculo del error J (o pérdida), que reciba la salida de la última capa de la red neuronal y los valores verdaderos (*ground truth*), para el mini-lote completo. En la propagación hacia adelante calcula entonces el error, y en la retropropagación el gradiente correspondiente.

   Una capa debe usar error cuadrático (MSE) y otra entropía cruzada. Note que estas deben usarse con las capas de activación que correspondan.

6.2 Entrenamiento de la red

1. Con las capas desarrolladas anteriormente implemente un programa para el entrenamiento de la red con descenso estocástico de gradiente. Utilice los datos que genera la función entregada.

2. Implemente además las técnicas de descenso de gradiente con momento y ADAM, para que las evalúe.

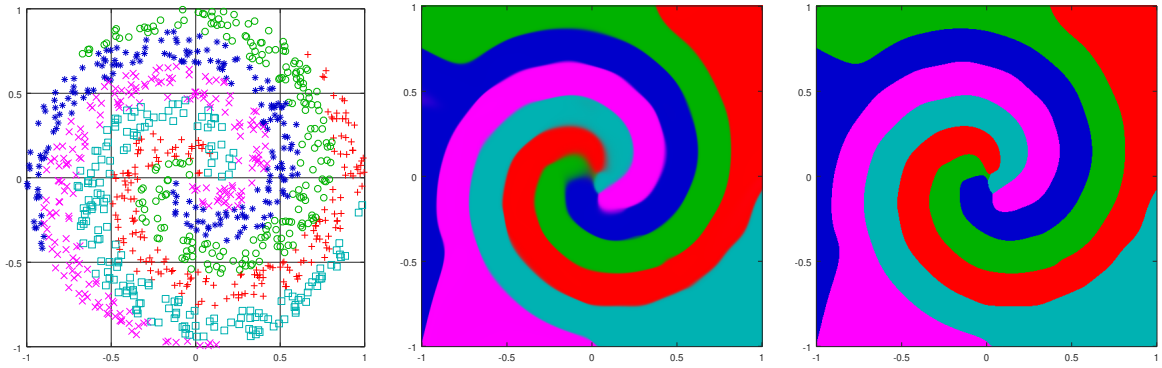
3. Para evaluar el desempeño de su algoritmo, genere otro conjunto de datos de prueba (distinto del conjunto de entrenamiento) con la misma distribución de puntos que los datos de entrenamiento.

4. La función de entrenamiento debe permitir almacenar los errores (si el usuario así lo desea) para posteriormente realizar el gráfico de error en función del número de

época (una época es la presentación de todos los datos de entrenamiento disponibles), tanto con el conjunto de pruebas como con el conjunto de entrenamiento.
.

6.3 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. La figura 3 muestra un ejemplo del resultado deseado para los datos de entrenamiento espirales mostrados a la izquierda.



(a) (b) (c)

Figura 3: Ejemplo de visualización del entrenamiento para los datos mostrados en (a). La salida ponderada se muestra en (b) y la clase ganadora en (c). El ejemplo utiliza los datos espirales para 5 clases, para una red de 3 capas totalmente conectadas con funciones de activación sigmoidales.

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

Para generar una representación visual de la clasificación alcanzada, revise el código brindado como ejemplo 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

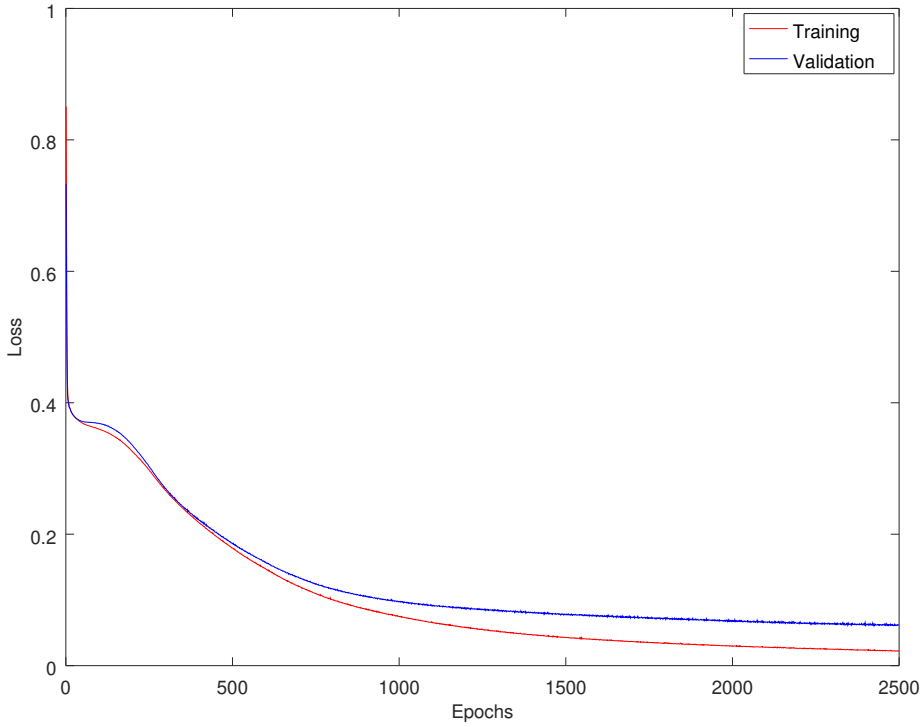


Figura 4: Ejemplo de curvas de pérdida en función de las épocas para el conjunto de entrenamiento y otro de validación.

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 `image` muestre su imagen y sobrepóngale los puntos de entrada. La alternativa `imshow` no permite sobreponer los puntos de entrenamiento.

6.4 Evaluación de la red

1. Para depurar la red muestre los datos de error en función de las épocas para los conjuntos de validación y de entrenamiento, tomando como base lo solicitado en el punto 6.2.4 (ver figura 4).

   Debe asegurarse de que las pérdidas mostradas de entrenamiento y validación sean comparables, puesto que en la definición usual mientras más datos se usen, mayor será la pérdida. Esto quiere decir que posíblemente usted requiera normalizar la definición de pérdida.

2. Investigue qué es una matriz de confusión.

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

4. Implemente una función para evaluación de la red por medio de matrices de confusión, que calcule precisión y exhaustividad alcanzados con el conjunto de datos de validación. Las filas deben tener las clases reales y las columnas las clases predichas.

5. Evalúe los resultados obtenidos sobre precisión y exhaustividad para distintas funciones de activación, funciones de pérdida, número de capas, tamaños de mini-lotes, tasas de aprendizaje y valores de momento.

6. Modifique el entrenamiento para que además de las curvas de error con el set de entrenamiento, se puedan producir curvas de error con un set aparte de prueba. Compare el comportamiento de ambas curvas para los casos del punto anterior que considere relevantes.

7 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.