Redes Neuronales

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Nº1

Conformación del grupo

Integrante	LU/DNI	Correo electrónico
Grenier, Michelle	418/10	mishu.grenier@gmail.com
Gómez Vidal, Leandro	957/11	lgvidal@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción al problema

La consigna de este trabajo es utilizar redes neuronales artificiales entrenadas mediante retro-propagación de errores vistas en la materia para modelar dos tipos distintos de problemas existentes en el mundo real.

El primer problema consiste en clasificar vectores de datos obtenidos a partir de muestras de tumores (textura, tamaño, etc.) de acuerdo a si se corresponden o no con tumores malignos, relacionados con el cáncer de mamas. El segundo, en ser capaz de predecir con cierta confianza la cantidad de energía necesaria para calefacción y refrigeración respectivamente, en base a ciertos atributos de entrada de edificios, que representan valores de distintos aspectos a tener en cuenta, tales como superficie total u orientación.

Se espera entonces que utilizando lo que vimos en la materia podamos desarrollar redes que resuelvan correctamente ambos problemas, enfrentándonos con el modelado de redes neuronales y la experimentación y análisis necesarios para validar el modelo en el proceso.

2. Implementación de la solución y decisiones tomadas

Decidimos implementar un único esquema de perceptrón multicapa (*MLP* por sus siglas en inglés, *Multi Layered Perceptron*) con una única capa oculta para ambos problemas, ya que a través de los distintos parámetros de creación y entrenamiento sabemos que el modelo es capaz de adaptarse a ambos problemas, clasificando o prediciendo valores esperados. No hicimos cambios mayores sobre los pseudocódigos provistos en la práctica. Utilizamos únicamente lenguaje de programación *Python* para desarrollar el MLP, realizar la experimentación y construir el programa entregado adjunto. Usamos los paquetes *NumPy* y *PyPlot* para procesar y graficar la experimentación respectivamente.

En lo relativo al entrenamiento, decidimos implementar la técnica de *momentum* para las correcciones de los pesos luego de cada iteración, por su facilidad de implementar y sus beneficios en cuanto a la agilidad de convergencia y la superación de máximos locales cuando es aplicada correctamente. Además, optamos por aplicar la técnica de *Holdout* vista en la cursada, dividiendo el conjunto de entrada en una parte de training y otra de testing en forma aleatoria luego de cada iteración, en base a un parámetro configurable. Evitamos de esta manera el *overfitting* dentro de los propios conjuntos de entrenamiento, situación contra la que nos enfrentamos inmediatamente al hacer una única división de los datos de entrada.

En cuanto a la validación del modelo, partimos los datos disponibles aleatoriamente en partes destinadas al training y validación. La red no veía los datos de validación hasta después de entrenada utilizando *Holdout* con los datos de training, midiendo qué tan general era la solución encontrada para este segmento. Obtuvimos resultados certeros para el primer ejercicio, sospechamos que para el segundo un análisis más minucioso de los datos de entrada podría habernos generado un esquema de training y validación a través del cual hubieramos obtenido mejores resultados.

Planteamos la experimentación como una búsqueda exhaustiva a través de los posibles parámetros de entrenamiento, creando *scripts* que creaban redes y evaluaban su desempeño de forma automática. Los pesos y gráficos resultantes de las últimas ejecuciones se incluyen en la carpeta *out* de la presente entrega. En este informe mostraremos algunos casos interesantes y demostrativos. El *script* en cuestión se encuentra comentado en *tp.py*, ya que no es relevante para para la funcionalidad requerida del programa.

2.1. Normalización y consecuencias

Probamos pasar todos los datos de entrada al intervalo $[0, 1]$ simplemente restando el mínimo y dividiendo por el máximo. Resultó rápidamente evidente que esto no nos iba a servir. Pasamos entonces a lo sugerido en la práctica, restando el promedio y dividiendo por el desvío estándar, midiendo de alguna manera la distancia al promedio relativa a cada atributo o característica en el vector de entrada. Al utilizar esta técnica, se resalta la importancia de qué tan representativos son los datos de entrenamiento con respecto al dominio que se espera abarcar, ya que una distribución muy distinta de los atributos o características anula enteramente la utilidad de la red, que usa el promedio y el desvío de su entrenamiento para normalizar cualquier entrada que se le alimente.

En el primer ejercicio encontramos que no era necesario normalizar la salida, puesto que el objetivo era únicamente clasificar. La red entonces luego de ser entrenada promedia los resultados obtenidos para los tumores benignos y los de los malignos por separado, marcando en la recta real la división como el punto medio entre ambos promedios. Para el segundo ejercicio, se utilizan los datos de la distribución de los resultados energéticos provistos en el entrenamientos para trasladar los resultados de la red a predicciones de energía, ya que se supone que no se tienen las respuestas *correctas* para los datos que queremos predecir.

3. Algunos resultados obtenidos de la experimentación

En la carpeta de la entrega se encuentran los archivos *redEj1.npz* y *redEj2.npz*, que serían la solución a los problemas planteados por el enunciado. Notamos que el archivo de guardado se corresponde a varios arreglos indizados en *NumPy*, y en caso de querer extraer los pesos, basta con el siguiente código en Python:

```
import numpy as np
pesosInputHidden = np.load('redEj.npz')['pesos1']
pesosHiddenOutput = np.load('redEj.npz')['pesos2']
```

Detallamos el entrenamiento y testing de las soluciones encontradas en las secciones correspondientes a cada ejercicio a continuación. Asimismo, incluímos algunos ejemplos patológicos o de interés en cada caso. Cabe notar que para toda la experimentación presentada utilizamos la función hiperbólica, ya que no notamos diferencias en la precisión obtenida al comparar con la sigmoidea, que en algunos casos al evaluar distintas formas de normalizar los datos de entrada causó *overflows* al elevar e a exponentes muy grandes.

3.1. Ejercicio 1

Encontramos nuestro mejor resultado para este ejercicio utilizando 15 unidades en la capa oculta, un *holdout ratio* de 0.3, un *learning rate* de 0.05 y un *momentum* de 0.1. Viendo cómo convergían un par de intentos previos al ganador, determinamos que para este ejercicio las redes no solían necesitar más de 150 épocas para alcanzar su máximo desempeño. Toda la experimentación se realizó con 200 épocas como límite.

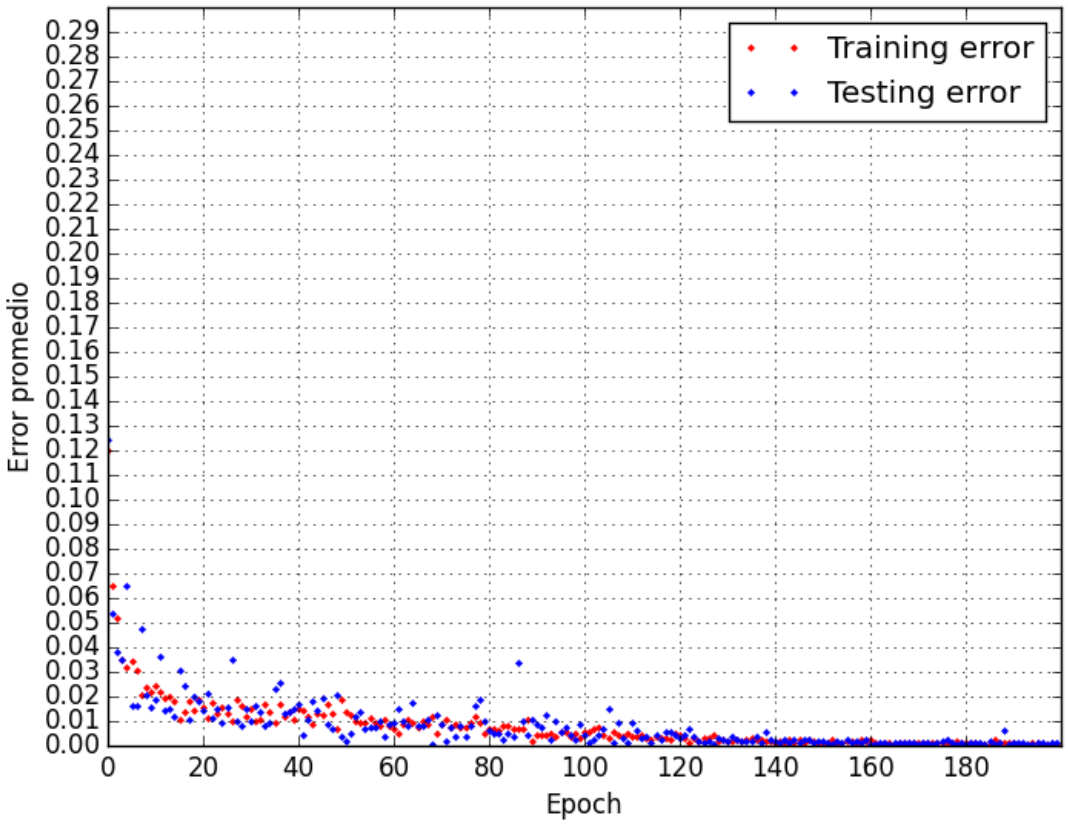

Figura 1: Errores de entrenamiento y testing promedio de la mejor red encontrada para el Ej1.

Observamos que la red se ajustó perfectamente a los datos que tenemos disponibles, pudiendo clasificar correctamente todos los datos de entrada. Queda ver qué tan general resultó esta solución.

4. Conclusiones

Lo principal que extraemos de este trabajo es el poder latente de un modelo engañosamente simple como es el del perceptrón multicapa. Si bien es relativamente sencillo de implementar, e incluso con una cantidad acotada de parámetros únicamente con fines académicos, pudimos ver que su desempeño es eficiente en casos reales. Para el primer problema llegamos a obtener una clasificación perfecta sobre los datos que teníamos, aunque resta ver qué tan general terminó siendo la red creada. Observamos que es muy importante qué tan representativos del dominio real son los datos que se destinan al entrenamiento, ya que es lo único tenido en cuenta por las distintas técnicas para configurar la red.

Más relativo a la implementación y creación de estas redes, encontramos que incluso en casos pequeños y con fin académico como los estudiados en este trabajo, la variabilidad de los resultados que se pueden obtener cambiando la forma de entrenar a la red y los propios parámetros de cada método de entrenamiento es enorme. Queda claro que la experimentación juega un papel fundamental en la aplicación de este tipo de modelos a distintos problemas y sus respectivos dominios. Buscando en internet incluso, más allá de las consultas en la práctica, encontramos que aparentemente no existe ninguna *fórmula mágica* para generar automáticamente la arquitectura de red y parámetros de entrenamiento óptimos para un problema determinado. Esto refuerza de alguna manera el paralelismo con las redes neuronales naturales, mantiendo presente la falta de un esquema general que englobe a todas las posibilidades, en el que no existen mutaciones.

Por último, existen numerosas implementaciones en múltiples lenguajes de distintos tipos de redes neuronales, formando parte de bibliotecas con un desarrollo enorme detrás y usadas a nivel profesional (ej. *OpenCV*). Nos resulta particularmente interesante la aplicación de este tipo de modelos al análisis y procesamiento de imágenes, con incontables maneras de aplicarlo a máquinas con las que interactuamos cotidianamente en el mundo real. Resulta esperable que exista la capacidad de desarrollar modelos predictivos robustos y confiables en múltiples áreas de interés general (clima y fenómenos naturales, tendencias económicas y sociales, medicina, etc.) mucho más allá de la introducción vista en esta materia.

5. Uso del programa adjunto

Se adjunta el código fuente del perceptrón multicapa implementado en Python, así como un programa ya armado para cumplir los requerimientos del enunciado (*tp.py*). No usa herramientas fuera de las recomendadas por la cátedra, habiendo sido desarrollado y probado en *Python 2.7.9*, utilizando los paquetes *NumPy* y *PyPlot* para procesar y graficar los resultados respectivamente.

Se utiliza a través de línea de comandos, cambiando los parámetros de acuerdo a si se quiere entrenar una red nueva o testear una red ya existente.

5.1. Entrenando una nueva red

```
python tp.py ejercicio -t dataCsv outNetwork param
```

donde:

- `ejercicio` Debe ser `ej1` o `ej2` para trabajar con los datos del primer o segundo ejercicios respectivamente.
- `dataCsv` Debe ser el path correspondiente al archivo que contiene los datos de entrenamiento. Se respeta el formato *csv* descripto en el enunciado. El correcto funcionamiento del programa no se garantiza para archivos con un formato distinto al de los archivos provisto por la cátedra.
- `outNetwork` Debe ser el path correspondiente al archivo en el que se quieren guardar los datos de la red entrenada. En caso de poseer una terminación distinta a *.npz*, se extenderá el nombre provisto agregando la misma. El formato de guardado es el usado por el paquete *NumPy* para guardar arreglos en forma comprimida.
- `param` Corresponde a lista de parámetros requeridos para realizar el entrenamiento, detallados en el orden correspondiente en el que deben notarse a continuación:
  - `unidadesCapaOculta` Es la cantidad de unidades presentes en la única capa oculta que tiene la red.
  - `errorAceptable` Corresponde al máximo error considerado aceptable para detener el entrenamiento. Corresponde a la suma del error de entrenamiento y el de testeo a través del método de entrenamiento descripto anteriormente.
  - `maxEpoch` Determina el máximo números de épocas de entrenamiento antes de detenerse, de acuerdo al método de entrenamiento descripto anteriormente.
  - `holdoutRate` Es la proporción de la entrada destinada a testing, debe estar en el intervalo `(0.0, 1.0)`.
  - `learningRate` Corresponde al parámetro homónimo utilizado en el entrenamiento de la red.
  - `momentumRate` Corresponde al parámetro homónimo utilizado en el entrenamiento de la red.

Cabe notar que la ejecución de este comando produce algunas imágenes en el directorio local. Estas imágenes son gráficos correspondientes al error de entrenamiento y testeo. En el ejercicio 1, es una única imágen llamada *errorTrainingTesting.png*, que muestra la evolución de ambos errores a lo largo de las épocas de entrenamiento. En el segundo ejercicio, se agregan al gráfico anterior *errorRefrig.png* y *errorCalefacc.png*, mostrando el error concreto obtenido por la red resultante para todos los datos de entrada al realizar la predicción correspondiente.

5.1.1. Ejemplo de uso

```
python tp.py ej1 -t train.csv red 15 0.005 200 0.3 0.05 0.05
```

Siendo *train.csv* alguna selección de las muestras presentes en el archivo provisto por la cátedra, *tp1_training_dataset_1.csv*. Crea una red con capa oculta de 15 unidades, un error aceptable de 0.005, máximo de 200 épocas, 0.3 para Holdout, learning rate de 0.05 y lo mismo para momentum. Crear los archivos *red.npz* con los datos de la red y *errorTrainingTesting.png* con el error promedio obtenido sobre el conjunto de testing determinado por Holdout en cada época.

5.2. Testeando una red existente

```
python tp.py ejercicio -l inNetwork inDataset
```

donde:

- `ejercicio` Debe ser `ej1` o `ej2` para trabajar con los datos del primer o segundo ejercicios respectivamente.
- `inNetwork` Debe ser el path correspondiente al archivo que contiene los datos de la red, guardado a través de la funcionalidad anteriormente descripta de este mismo programa.
- `inDataset` Debe ser el path correspondiente al archivo del que se quieren cargar los datos de testeo. Se respeta el formato *csv* descripto en el enunciado. El correcto funcionamiento del programa no se garantiza para archivos con un formato distinto al de los archivos provisto por la cátedra

5.2.1. Ejemplo de uso

```
python tp.py ej1 -l red.npz tp1_training_dataset_1.csv
```

Siendo *red.npz* la red creada en el ejemplo de uso anterior, y *tp1_training_dataset_1.csv* el archivo homónimo provisto por la cátedra. El proceso de testing no genera archivos adicionales, únicamente imprimiendo información por pantalla.

5.3. Generalidades

En cualquiera de sus modos de uso, el programa imprime progresivamente por pantalla información relativa al estado actual del procesamiento. Se recomienda fuertemente redirigir *stdout* a un archivo de texto si se está usando una consola con límite de caracteres en ventana, como por ejemplo *cmd* en sistemas Windows.

Entrenando, muestra por pantalla el error de training y testing obtenido en cada época. Testeando, muestra por pantalla los resultados obtenidos para cada muestra y cuál era el resultado esperado, indicando el error en cada caso. Al final del testeo incluye el error promedio correspondiente en cada caso.

```
import numpy as np

def fHiper(x):
    return np.tanh(x)

def fDerHiper(x):
    return 1.0-x**2

def fSigmoidea(x):
    return 1/(1+np.exp(-x))

def fDerSigmoidea(x):
    return fSigmoidea(x)*(1-fSigmoidea(x))
```

```
# -*- coding: utf-8 -*
import numpy as np
import math as m
import sys


# Perceptrón multicapa.
class PM:

    def __init__(self, unidadesPorCapa, funcActivacion, derFuncActivacion):

        # PerceptronMulticapa([2,3,1])
        # Tiene 3 capas, 2->3->1.
        # Contamos la input como una.
        # La última sería la output.

        self.pesos = []
        for i in range(1, len(unidadesPorCapa) - 1):
            self.pesos.append((2*np.random.random((unidadesPorCapa[i - 1] + 1,
            unidadesPorCapa[i]
                                + 1))-1)*0.5)
            self.pesos.append((2*np.random.random((unidadesPorCapa[i] + 1, unidadesPorCapa[i +
                                1]))-1)*0.5)
        self.deltaPesos = [] # Para momentum.
        self.funcActivacion = funcActivacion
        self.derFuncActivacion = derFuncActivacion

    def activar(self, Xh):

        #print "EntradaActivacion"
        #print Xh
        Xh = np.atleast_2d(Xh)
        X = np.ones((Xh.shape[0], Xh.shape[1]+1))
        X[:, 0:-1] = Xh #Bias

        # Propagar usando función de activación
        resUltimaCapa = X
        for k in range(0, len(self.pesos)):
            resUltimaCapa = self.funcActivacion(np.dot(resUltimaCapa,self.pesos[k]))
        # Devuelve la capa de salida
        return resUltimaCapa


    def entrenar(self, input, target, lrate=0.1, momentum=0.1):

        self.deltaPesos = []

        input = np.atleast_2d(input)
        X = np.ones((input.shape[0], input.shape[1]+1))
        X[:, 0:-1] = input #Bias
        deltas = []


        # Propago guardando todo lo obtenido. En propagaciones[-1] está la salida.

        propagaciones = [X]
        for k in range(len(self.pesos)):
```

```
            propagaciones.append(self.funcActivacion(np.dot(propagaciones[k], self.pesos[k])))

        # Error
        error = target - propagaciones[-1]
        delta = error*self.derFuncActivacion(propagaciones[-1])
        deltas.append(delta)

        # Delta de cada capa, desde la última a la segunda.
        for i in range(len(propagaciones)-2,0,-1):
            delta = np.dot(deltas[-1], self.pesos[i].T)*self.derFuncActivacion(propagaciones[
            i])
            deltas.append(delta)
        deltas.reverse()

        # Actualizar pesos
        for i in range(len(self.pesos)):
            capa = np.atleast_2d(propagaciones[i])
            delta = np.atleast_2d(deltas[i])
            dw = np.dot(capa.T,delta)
            self.deltaPesos.append(dw)
            self.pesos[i] += lrate*dw + momentum*self.deltaPesos[i]

        # Devuelve error
        return (error**2).sum()
```

```
import factivacion as fact
import numpy as np
import random as ran

def training(network, trainInput, trainTarget, ratioAprendizaje, ratioMomentum):

    print "> TRAINING"
    indexes = [x for x in range(0, len(trainInput))]
    np.random.shuffle(indexes)

    errorTraining = 0
    for n in indexes:
        errorTraining += network.entrenar( trainInput[n], trainTarget[n], ratioAprendizaje,
        ratioMomentum )
    errorTraining /= len(trainInput)
    return errorTraining

def testing(network, testInput, testTarget):

    print "> TESTING"
    errorTesting = 0
    salidas = []
    for n in range(0, len(testInput)):
        salida = network.activar( testInput[n] )
        salidas.append(salida)
        error = testTarget[n] - salida
        errorTesting += ((error)**2).sum()
    errorTesting /= len(testInput)
    return errorTesting, salidas

def holdout(network, input, target, errorAceptable, maxEpoch, tamParticionTest=0.15,
ratioAprendizaje=0.2, ratioMomentum=0.1):



    errorTraining = 1
    errorTesting = 1
    erroresTrainingPorEpoch = []
    erroresTestingPorEpoch = []
    epoch = 0


    while ((errorTraining > errorAceptable) or (errorTesting > errorAceptable)) and epoch <
    maxEpoch:

        # Partir la entrada.
        totalSets = target.shape[0]
        testIndexes = []
        trainIndexes = []
        while not testIndexes or not trainIndexes:
            testIndexes = [x for x in range(0, totalSets) if np.random.random_sample() <
            tamParticionTest]
            trainIndexes = [x for x in range(0, totalSets) if x not in set(testIndexes)]

        #print "Train> " + str(sorted(trainIndexes))
        #print "Test> " + str(sorted(testIndexes))
```

```
        # Training.
        trainInput = [input[x] for x in trainIndexes]
        trainTarget = [target[x] for x in trainIndexes]
        testInput = [input[x] for x in testIndexes]
        testTarget = [target[x] for x in testIndexes]



        print "> EPOCH " + str(epoch)

        errorTraining = training(network, trainInput, trainTarget, ratioAprendizaje,
        ratioMomentum)
        errorTesting, salidasTesting = testing(network, testInput, testTarget)
        erroresTrainingPorEpoch.append(errorTraining)
        erroresTestingPorEpoch.append(errorTesting)
        print ">> ErrorTraining   " + str(errorTraining)
        print ">> ErrorTesting    " + str(errorTesting)
        epoch = epoch + 1


    return erroresTrainingPorEpoch, erroresTestingPorEpoch, epoch
```

```
# -*- coding: utf-8 -*
from factivacion import fHiper, fDerHiper, fSigmoidea, fDerSigmoidea
from pmulticapa import PM
import numpy as np
from training import holdout, testing
import sys
from pylab import Line2D, plot, axis, show, pcolor, colorbar, bone, savefig
import pylab as plt

def ej1(archivoDataset=None, archivoGuardado=None, archivoCarga=None, archivoTest= None,
unidadesPorCapa=[30, 20, 1], errorTrainingAceptable=0.1, maxEpoch=1000, tamParticionTest=0.2,
 ratioAprendizaje=0.15, ratioMomentum=0.1):

    prefijo = '' #Para uso habitual, el otro es para batch.
    #prefijo = 'out/ej1/hidden' + str(unidadesPorCapa[1]) + 'err' +
    str(errorTrainingAceptable) + 'maxEpoch' + str(maxEpoch) + 'holdout' +
    str(tamParticionTest) + 'lrate' + str(ratioAprendizaje) + 'rmom' + str(ratioMomentum)

    print "EJERCICIO 1"

    if archivoDataset and archivoGuardado:

        print "Unidades por capa " + str(unidadesPorCapa)
        print "Error aceptable " + str(errorTrainingAceptable)
        print "MaxEpoch " + str(maxEpoch)
        print "HoldoutRatio " + str(tamParticionTest)
        print "Ratio aprendizaje " + str(ratioAprendizaje)
        print "Ratio momentum " + str(ratioMomentum)

        print "> Cargando dataset..."

        # Transformar M y B en 0 y 1.
        data = None
        with open (archivoDataset, "r") as f:
            data = f.read().replace('M', '0').replace('B', '1')
        with open (".aux", "w") as f:
            f.write(data)

        dataset =  np.loadtxt(".aux", delimiter=",", skiprows=0)

        caracteristicas = dataset[:,1:]
        medCaracteristicas = []
        desCaracteristicas = []
        # Normalizo
        for j in range(caracteristicas.shape[1]):
            medCaracteristicas.append(np.mean(caracteristicas[:,j]))
            desCaracteristicas.append(np.std(caracteristicas[:,j]))
            caracteristicas[:,j] = (caracteristicas[:,j] - medCaracteristicas[j] ) /
            desCaracteristicas[j]
        medCaracteristicas = np.array(medCaracteristicas)
        desCaracteristicas = np.array(desCaracteristicas)
        benignidad = dataset[:,0]
        ej1Samples = np.zeros(benignidad.size, dtype=[('input',  float, 30), ('output', float
        , 1)])

        for k in range(len(ej1Samples["input"])):
            ej1Samples["input"][k] = caracteristicas[k,:]
```

```
            ej1Samples["output"][k] = benignidad[k]


        print "> Entrenando red..."
        network = PM(unidadesPorCapa, fHiper, fDerHiper)
        erroresTraining, erroresTesting, epoch = holdout(network, ej1Samples["input"].copy(),
         ej1Samples["output"].copy(), errorTrainingAceptable, maxEpoch, tamParticionTest,
        ratioAprendizaje, ratioMomentum)

        print '>> epochAlcanzada: ' + str(epoch)
        print '>> errorTraining: ' + str(erroresTraining[-1])
        print '>> errorTesting: ' + str(erroresTesting[-1])


        bone()
        epochs = range(len(erroresTraining))
        plot(epochs, erroresTraining, '.', color='r', ms=4.5,label='Training error')
        plot(epochs, erroresTesting, '.', color='b', ms=4.5, label='Testing error')

        plt.xlabel('Epoch')
        plt.ylabel('Error promedio')
        plt.legend()
        #plt.ylim(0.0, 0.30)
        #plt.yticks( np.arange(0.0, 0.30, 0.01) )
        #plt.xticks( np.arange(0, maxEpoch, 20) )
        plt.grid()
        savefig(prefijo + 'errorTrainingTesting.png')
        plt.close()

        # Tengo que de alguna manera decidir cómo separar para clasificar.
        # Idea> Tomo todo el set que use para el holdout, lo testeo, y como sé las
        respuestas, calculo el mean de cada clase, partiendo a la mitad entre ambos.
        # Siendo B y M las clases, la recta real quedaría algo como>    0      M     M M
        MM M    M  B  M   M     B    BB   B B 1, debería partir en un lugar razonable.
        print "> Determinando divisor para clasificar, en base a los datos de entrenamiento"
        erroresTesting, salidasTesting = testing(network, caracteristicas, benignidad)

        benignos = [salidasTesting[k] for k in range(len(salidasTesting)) if benignidad[k] ==
         1.0]
        malignos = [salidasTesting[k] for k in range(len(salidasTesting)) if benignidad[k] !=
         1.0]
        divisionBenignidad = np.array([(np.mean(benignos) + np.mean(malignos)) / 2])
        np.savez(prefijo + archivoGuardado, pesos1=network.pesos[0], pesos2=network.pesos[1],
         medCaracteristicas=medCaracteristicas, desCaracteristicas=desCaracteristicas,
        divisionBenignidad=divisionBenignidad)

        return np.mean(erroresTraining), np.mean(erroresTesting)

    elif archivoCarga and archivoTest:

        print "> Creando red con los pesos provistos..."

        # Necesario para transformar M y B en 0 y 1.
        data = None
        with open (archivoTest, "r") as f:
            data = f.read().replace('M', '0').replace('B', '1')
        with open (".aux", "w") as f:
```

```
            f.write(data)

        loadedData = np.load(archivoCarga)
        medCaracteristicas = loadedData['medCaracteristicas']
        desCaracteristicas = loadedData['desCaracteristicas']
        pesos = [loadedData['pesos1'], loadedData['pesos2']]
        divisionBenignidad = loadedData['divisionBenignidad'][0]

        dataset =  np.loadtxt(".aux", delimiter=",", skiprows=0)
        caracteristicas = dataset[:,1:]
        # Normalizo
        for j in range(caracteristicas.shape[1]):
            caracteristicas[:,j] = (caracteristicas[:,j] - medCaracteristicas[j] ) /
            desCaracteristicas[j]
        benignidad = dataset[:,0]


        unidadesPorCapa = []
        unidadesPorCapa.append(30)
        for i in range(1, len(pesos)):
            unidadesPorCapa.append(len(pesos[i]))
        unidadesPorCapa.append(1)

        network = PM(unidadesPorCapa, fHiper, fDerHiper)
        network.pesos = pesos


        erroresTesting, salidasTesting = testing(network, caracteristicas, benignidad)
        clasificacionTesting = np.array([0.0 if x < divisionBenignidad else 1.0 for x in
        salidasTesting])

        pifiesTesting = np.count_nonzero(np.array(benignidad) - clasificacionTesting)
        for k in range(len(clasificacionTesting)):
            print ">>> Muestra " + str(k)
            print ">>> Obtenido " + str(salidasTesting[k])
            print ">>> Clasificado " + str("BENIGNO" if clasificacionTesting[k] == 1.0 else
            "MALIGNO") + " y era " + str(("BENIGNO" if benignidad[k] == 1.0 else "MALIGNO"))
            + " (" + str(("OK" if clasificacionTesting[k]  == benignidad[k] else "ERROR")) +
            ")"

        print ">> MeanErrorTesting    " + str(erroresTesting)
        print ">> Mal clasificados   " + str(pifiesTesting) + " de " + str(len(benignidad))

    else:
        # Esto no debiera pasar nunca.
        print 'ERROR'
        raise Exception("Mal codeado el pasaje de param.")

def ej2(archivoDataset=None, archivoGuardado=None, archivoCarga=None, archivoTest= None,
unidadesPorCapa=[8, 4, 2], errorTrainingAceptable=0.1, maxEpoch=1000, tamParticionTest=0.2,
ratioAprendizaje=0.15, ratioMomentum=0.1):

    prefijo = '' #Para uso habitual, el otro es para batch.
    #prefijo = 'out/ej2/hidden' + str(unidadesPorCapa[1]) + 'err' +
    str(errorTrainingAceptable) + 'maxEpoch' + str(maxEpoch) + 'holdout' +
    str(tamParticionTest) + 'lrate' + str(ratioAprendizaje) + 'rmom' + str(ratioMomentum)
```

```
print "EJERCICIO 2"

if archivoDataset and archivoGuardado:

    print "Unidades por capa " + str(unidadesPorCapa)
    print "Error aceptable " + str(errorTrainingAceptable)
    print "MaxEpoch " + str(maxEpoch)
    print "HoldoutRatio " + str(tamParticionTest)
    print "Ratio aprendizaje " + str(ratioAprendizaje)
    print "Ratio momentum " + str(ratioMomentum)


    print "> Cargando dataset..."

    dataset =  np.loadtxt(archivoDataset, delimiter=",", skiprows=0)
    atributos = dataset[:,:8]
    respuestas = dataset[:,8:]
    ej2Samples = np.zeros(respuestas.shape[0], dtype=[('input',  float, 8), ('output',
    float, 2)])


    atributos = dataset[:,:8]
    medAtributos = []
    desAtributos = []
    for j in range(atributos.shape[1]):
        #minAtributos = atributos[:,j].min()
        #maxAtributos = atributos[:,j].max()
        #atributos[:,j] = (atributos[:,j] - minAtributos) / maxAtributos
        medAtributos.append(np.mean(atributos[:,j]))
        desAtributos.append(np.std(atributos[:,j]))
        atributos[:,j] = (atributos[:,j] - medAtributos[j] ) / desAtributos[j]
    respuestas = dataset[:,8:]
    respuestasOrig = dataset[:,8:].copy()

    #minRespuestas = [0, 0]
    #maxRespuestas = [0, 0] #Los guardo para denormalizar después
    medRespuestas = []
    desRespuestas = []
    for j in range(respuestas.shape[1]):
        medRespuestas.append(np.mean(respuestas[:,j]))
        desRespuestas.append(np.std(respuestas[:,j]))
        respuestas[:,j] = (respuestas[:,j] - medRespuestas[j] ) / desRespuestas[j]
        #minRespuestas[j] = respuestas[:,j].min()
        #maxRespuestas[j] = respuestas[:,j].max()
        #respuestas[:,j] = (respuestas[:,j] - minRespuestas[j])/maxRespuestas[j]
    for k in range(len(ej2Samples["input"])):
        ej2Samples["input"][k] = atributos[k,:]
        ej2Samples["output"][k] = respuestas[k]

    print "> Entrenando red..."
    network = PM(unidadesPorCapa, fHiper, fDerHiper)

    erroresTraining, erroresTesting, epoch = holdout(network, ej2Samples["input"].copy(),
     ej2Samples["output"].copy(), errorTrainingAceptable, maxEpoch, tamParticionTest,
    ratioAprendizaje, ratioMomentum)

    print '>> epochAlcanzada: ' + str(epoch)
```

```
print '>> errorTraining: ' + str(erroresTraining[-1])
print '>> errorTesting: ' + str(erroresTesting[-1])

print '>> Evaluando contra el dataset de entrada entero'
erroresPrimero = []
erroresSegundo = []
for k in range(atributos.shape[0]):
    res = network.activar(atributos[k])
    normalizado = res #np.multiply(res, maxRespuestas) + minRespuestas
    for j in range(normalizado.shape[1]):
        normalizado[:,j] = (normalizado[:,j] * desRespuestas[j]) + medRespuestas[j]

    print(k, (normalizado,' Esperado ', respuestasOrig[k]))
    erroresPrimero.append(np.abs(respuestasOrig[k][0] - normalizado[0][0]))
    erroresSegundo.append(np.abs(respuestasOrig[k][1] - normalizado[0][1]))

print '>> Error promedio calefac.'
print np.mean(erroresPrimero)
print '>> Error promedio refrig.'
print np.mean(erroresSegundo)
print '>> Min/Max error calefac.'
print(np.min(erroresPrimero), np.max(erroresPrimero))
print '>> Min/Max error refrig.'
print(np.min(erroresSegundo), np.max(erroresSegundo))

print "> Guardando pesos y datos de normalizacion en " + str(archivoGuardado)

np.savez(prefijo + archivoGuardado, pesos1=network.pesos[0], pesos2=network.pesos[1],
 medAtributos=medAtributos, desAtributos=desAtributos, medRespuestas=medRespuestas,
desRespuestas=desRespuestas)

bone()
epochs = range(len(erroresTraining))
plot(epochs, erroresTraining, '.', color='r', ms=4.5,label='Training error')
plot(epochs, erroresTesting, '.', color='b', ms=4.5, label='Testing error')

plt.xlabel('Epoch')
plt.ylabel('Error promedio')
plt.legend()
#plt.ylim(0.0, 0.5)
#plt.yticks( np.arange(0.0, 0.5, 0.05) )
#plt.xticks( np.arange(0, maxEpoch, 50) )
plt.grid()
savefig(prefijo + 'erroresTrainingTesting.png')
plt.close()

bone()
muestras = range(len(respuestas))
promedios = [np.mean(erroresPrimero) for x in erroresPrimero]
plot(muestras, erroresPrimero, 'o', color='r', ms=4.5,label='Error')
plt.xlabel('Nro. Muestra')
plt.ylabel('Error calefac.')
#plt.ylim(0.0, 20.0)
#plt.yticks( np.arange(0.0, 20.0, 2.0) )
#plt.xticks( np.arange(0, len(muestras), 50) )
plot(muestras, promedios, color='k',label='Error promedio', linewidth=2.0, linestyle=
'--')
```

```
        plt.legend()
        plt.grid()
        ylimIzq = plt.ylim()

        plt.twinx()
        plot(muestras, promedios, color='k',label='Error promedio', linewidth=2.0, linestyle=
        '--')
        plt.yticks([np.mean(erroresPrimero)]) # Sólo promedio.
        plt.ylim(ylimIzq)
        savefig(prefijo + 'errorCalefacc.png')
        plt.close()

        promedios = [np.mean(erroresSegundo) for x in erroresSegundo]
        bone()
        plot(muestras, erroresSegundo, 'o', color='c', ms=4.5,label='Error')
        plt.xlabel('Nro. Muestra')
        plt.ylabel('Error refrig.')
        plot(muestras, promedios, color='k',label='Error promedio', linewidth=2.0, linestyle=
        '--')
        plt.legend()
        #plt.ylim(0.0, 20.0)
        #plt.yticks( np.arange(0.0, 20.0, 2.0) )
        #plt.xticks( np.arange(0, len(muestras), 50) )
        plt.grid()
        ylimIzq = plt.ylim()

        plt.twinx()
        plot(muestras, promedios, color='k',label='Error promedio', linewidth=2.0, linestyle=
        '--')
        plt.yticks([np.mean(erroresSegundo)]) # Sólo promedio.
        plt.ylim(ylimIzq)

        savefig(prefijo + 'errorRefrig.png')
        plt.close()

        return np.mean(erroresPrimero), np.mean(erroresSegundo), np.mean(erroresTraining), np
        .mean(erroresTesting)

    elif archivoCarga and archivoTest:

        print "> Creando red con los pesos provistos..."

        dataset =  np.loadtxt(archivoTest, delimiter=",", skiprows=0)
        atributos = dataset[:,:8]
        respuestas = dataset[:,8:]
        respuestasOrig = dataset[:,8:].copy()
        ej2Samples = np.zeros(respuestas.shape[0], dtype=[('input',  float, 8), ('output',
        float, 2)])
        loadedData = np.load(archivoCarga)
        medAtributos = loadedData['medAtributos']
        desAtributos = loadedData['desAtributos']
        medRespuestas = loadedData['medRespuestas']
        desRespuestas = loadedData['desRespuestas']
        pesos = [loadedData['pesos1'], loadedData['pesos2']]

        for j in range(atributos.shape[1]):
            atributos[:,j] = (atributos[:,j] - medAtributos[j] ) / desAtributos[j]
```

```
        for j in range(respuestas.shape[1]):
            respuestas[:,j] = (respuestas[:,j] - medRespuestas[j] ) / desRespuestas[j]

        for k in range(len(ej2Samples["input"])):
            ej2Samples["input"][k] = atributos[k,:]
            ej2Samples["output"][k] = respuestas[k]

        print "> Entrenando red..."



        unidadesPorCapa = []
        unidadesPorCapa.append(8)
        for i in range(1, len(pesos)):
            unidadesPorCapa.append(len(pesos[i]))
        unidadesPorCapa.append(2)


        network = PM(unidadesPorCapa, fHiper, fDerHiper)
        network.pesos = pesos
        erroresTesting, salidasTesting = testing(network, atributos, respuestas)
        erroresCal = []
        erroresRef = []

        for k in range(len(salidasTesting)):
            salidasTesting[k][0][0] *= desRespuestas[0]
            salidasTesting[k][0][0] += medRespuestas[0]
            salidasTesting[k][0][1] *= desRespuestas[1]
            salidasTesting[k][0][1] += medRespuestas[1]

            erroresCal.append(np.linalg.norm(salidasTesting[k][0][0] - respuestasOrig[k][0]))
            erroresRef.append(np.linalg.norm(salidasTesting[k][0][1] - respuestasOrig[k][1]))
            print ">>> Muestra " + str(k)
            print ">>> Calef.  Obtenida " + str(salidasTesting[k][0][0]) + "\t/ " + str(
            respuestasOrig[k][0]) + "\tEsperada (" + str(salidasTesting[k][0][0]-
            respuestasOrig[k][0]) + ")"
            print ">>> Refrig. Obtenida " + str(salidasTesting[k][0][1]) + "\t/ " + str(
            respuestasOrig[k][1]) + "\tEsperada (" + str(salidasTesting[k][0][1]-
            respuestasOrig[k][1]) + ")"
        print ">> MeanErrorCalefacc.   " + str(np.mean(erroresCal))
        print ">> MeanErrorRefrig.   " + str(np.mean(erroresRef))

    else:
        # Esto no debiera pasar nunca.
        raise Exception("Mal codeado el pasaje de param.")
"""
# Código utilizado para generar los imágenes y testeos batch. Ignorar.

ej = 2
archivoDataset = 'tp1_training_dataset_2.csv'
archivoGuardado = 'pesos'
unidadesPorCapa = [8, 20, 2] if ej==2 else [30,20,1]
errorTrainingAceptable = 0.0
maxEpoch = 500

tamParticionTest = 0.3
```

```
posiblesHidden = [22, 23, 24, 25]
posiblesAprendizajes = [0.0125, 0.025, 0.05, 0.10, 0.5]
posiblesMomentum = [0.0125, 0.025, 0.05, 0.10, 0.25, 0.5]

meanTrainingHidden = [([], []) for x in posiblesHidden]
meanTestingHidden = [([], []) for x in posiblesHidden]
meanTrainingAprendizaje = [([], []) for x in posiblesAprendizajes]
meanTestingAprendizaje = [([], []) for x in posiblesAprendizajes]
meanTrainingMomentum = [([], []) for x in posiblesMomentum]
meanTestingMomentum = [([], []) for x in posiblesMomentum]

mejorErrorObtenido = 99999.0 # Suma de ambos errores.
mejorErrorObtenidoHidden = 0.0
mejorErrorObtenidoAprendizaje = 0.0
mejorErrorObtenidoMomentum = 0.0

mejorErrorTrainingObtenido = 99999.0 # Suma de ambos errores.
mejorErrorTrainingObtenidoHidden = 0.0
mejorErrorTrainingObtenidoAprendizaje = 0.0
mejorErrorTrainingObtenidoMomentum = 0.0

mejorErrorTestingObtenido = 99999.0 # Suma de ambos errores.
mejorErrorTestingObtenidoHidden = 0.0
mejorErrorTestingObtenidoAprendizaje = 0.0
mejorErrorTestingObtenidoMomentum = 0.0

for hidden in range(len(posiblesHidden)):
    unidadesPorCapa[1] = posiblesHidden[hidden]
    for ratioAprendizaje in range(len(posiblesAprendizajes)):
        for ratioMomentum in range(len(posiblesMomentum)):

            if ej == 2:
                erroresCalefa, erroresRefri, meanErrorTraining, meanErrorTesting =
                ej2(archivoDataset, archivoGuardado, None, None, unidadesPorCapa,
                errorTrainingAceptable, maxEpoch, tamParticionTest,
                posiblesAprendizajes[ratioAprendizaje], posiblesMomentum[ratioMomentum])
            else:
                meanErrorTraining, meanErrorTesting = ej1(archivoDataset, archivoGuardado,
                None, None, unidadesPorCapa, errorTrainingAceptable, maxEpoch,
                tamParticionTest, posiblesAprendizajes[ratioAprendizaje],
                posiblesMomentum[ratioMomentum])

            meanTrainingHidden[hidden][0].append(meanErrorTraining)
            meanTestingHidden[hidden][0].append(meanErrorTesting)
            meanTrainingAprendizaje[ratioAprendizaje][0].append(meanErrorTraining)
            meanTestingAprendizaje[ratioAprendizaje][0].append(meanErrorTesting)
            meanTrainingMomentum[ratioMomentum][0].append(meanErrorTraining)
            meanTestingMomentum[ratioMomentum][0].append(meanErrorTesting)
            param = (posiblesHidden[hidden], posiblesAprendizajes[ratioAprendizaje],
            posiblesMomentum[ratioMomentum])
            meanTrainingHidden[hidden][1].append(param)
            meanTestingHidden[hidden][1].append(param)
            meanTrainingAprendizaje[ratioAprendizaje][1].append(param)
            meanTestingAprendizaje[ratioAprendizaje][1].append(param)
            meanTrainingMomentum[ratioMomentum][1].append(param)
            meanTestingMomentum[ratioMomentum][1].append(param)
```

```
            errorObtenido = meanErrorTraining + meanErrorTesting
            if errorObtenido < mejorErrorObtenido:
                mejorErrorObtenido = errorObtenido
                mejorErrorObtenidoHidden = posiblesHidden[hidden]
                mejorErrorObtenidoAprendizaje = posiblesAprendizajes[ratioAprendizaje]
                mejorErrorObtenidoMomentum = posiblesMomentum[ratioMomentum]

            if meanErrorTraining < mejorErrorTrainingObtenido:
                mejorErrorTrainingObtenido = meanErrorTraining
                mejorErrorTrainingObtenidoHidden = posiblesHidden[hidden]
                mejorErrorTrainingObtenidoAprendizaje = posiblesAprendizajes[ratioAprendizaje]
                mejorErrorTrainingObtenidoMomentum = posiblesMomentum[ratioMomentum]

            if meanErrorTesting < mejorErrorTestingObtenido:
                mejorErrorTestingObtenido = meanErrorTesting
                mejorErrorTestingObtenidoHidden = posiblesHidden[hidden]
                mejorErrorTestingObtenidoAprendizaje = posiblesAprendizajes[ratioAprendizaje]
                mejorErrorTestingObtenidoMomentum = posiblesMomentum[ratioMomentum]

# Y tiro unos grafiquines copados, mejores, peores y mean. resultados para cada param.
print 'MEJOR SUMA OBTENIDO'
print 'Suma errores ' + str(mejorErrorObtenido) + ' hidden ' + str(mejorErrorObtenidoHidden)
+ ' aprendizaje ' + str(mejorErrorObtenidoAprendizaje) + ' momentum ' +
str(mejorErrorObtenidoMomentum)
print 'MEJOR TRAINING OBTENIDO'
print 'ErrorTraining  ' + str(mejorErrorTrainingObtenido) + ' hidden ' +
str(mejorErrorTrainingObtenidoHidden) + ' aprendizaje ' +
str(mejorErrorTrainingObtenidoAprendizaje) + ' momentum ' +
str(mejorErrorTrainingObtenidoMomentum)
print 'MEJOR TESTING OBTENIDO'
print 'ErrorTesting  ' + str(mejorErrorTestingObtenido) + ' hidden ' +
str(mejorErrorTestingObtenidoHidden) + ' aprendizaje ' +
str(mejorErrorTestingObtenidoAprendizaje) + ' momentum ' +
str(mejorErrorTestingObtenidoMomentum)
print 'HIDDEN'
meanTrainingHidden.sort(key=lambda x: x[1][0])
meanTestingHidden.sort(key=lambda x: x[1][0])
for k in range(len(meanTrainingHidden)):
    for j in range(len(meanTrainingHidden[k][0])):
        print 'hidden ' + str(meanTrainingHidden[k][1][j][0]) + 'lrate ' +
        str(meanTrainingHidden[k][1][j][1]) + ' momentum ' +
        str(meanTrainingHidden[k][1][j][2]) + ' training ' +
        str(meanTrainingHidden[k][0][j]) + ' testing ' + str(meanTestingHidden[k][0][j]) + '
        minTraining ' + str(np.min(meanTrainingHidden[k][0])) + ' maxTraining ' +
        str(np.max(meanTrainingHidden[k][0])) + ' avgTraining ' +
        str(np.mean(meanTrainingHidden[k][0])) + ' minTesting ' +
        str(np.min(meanTestingHidden[k][0])) + ' maxTesting ' +
        str(np.max(meanTestingHidden[k][0])) + ' avgTesting ' +
        str(np.mean(meanTestingHidden[k][0]))

print 'APRENDIZAJE'
meanTrainingAprendizaje.sort(key=lambda x: x[1][0])
meanTestingAprendizaje.sort(key=lambda x: x[1][0])
for k in range(len(meanTrainingAprendizaje)):
    for j in range(len(meanTrainingAprendizaje[k][0])):
        print 'hidden ' + str(meanTrainingAprendizaje[k][1][j][0]) + ' lrate ' +
        str(meanTrainingAprendizaje[k][1][j][1]) + ' momentum ' +
```

```
        str(meanTrainingAprendizaje[k][1][j][2]) + ' training ' +
        str(meanTrainingAprendizaje[k][0][j]) + ' testing ' +
        str(meanTestingAprendizaje[k][0][j]) + ' minTraining ' +
        str(np.min(meanTrainingAprendizaje[k][0])) + ' maxTraining ' +
        str(np.max(meanTrainingAprendizaje[k][0])) + ' avgTraining ' +
        str(np.mean(meanTrainingAprendizaje[k][0])) + ' minTesting ' +
        str(np.min(meanTestingAprendizaje[k][0])) + ' maxTesting ' +
        str(np.max(meanTestingAprendizaje[k][0])) + ' avgTesting ' +
        str(np.mean(meanTestingAprendizaje[k][0]))


print 'MOMENTUM'
meanTrainingMomentum.sort(key=lambda x: x[1][0])
meanTestingMomentum.sort(key=lambda x: x[1][0])
for k in range(len(meanTrainingMomentum)):
    for j in range(len(meanTrainingMomentum[k][0])):
        print 'hidden ' + str(meanTrainingMomentum[k][1][j][0]) + ' lrate ' +
        str(meanTrainingMomentum[k][1][j][1]) + ' momentum ' +
        str(meanTrainingMomentum[k][1][j][2]) + ' training ' +
        str(meanTrainingMomentum[k][0][j]) + ' testing ' + str(meanTestingMomentum[k][0][j])
        + ' minTraining ' + str(np.min(meanTrainingMomentum[k][0])) + ' maxTraining ' +
        str(np.max(meanTrainingMomentum[k][0])) + ' avgTraining ' +
        str(np.mean(meanTrainingMomentum[k][0])) + ' minTesting ' +
        str(np.min(meanTestingMomentum[k][0])) + ' maxTesting ' +
        str(np.max(meanTestingMomentum[k][0])) + ' avgTesting ' +
        str(np.mean(meanTestingMomentum[k][0]))

sys.exit()
"""

# Main.

args = sys.argv
usage = "\nPara entrenar desde un dataset y guardar la red:\n\
python tp.py (ej1|ej2) -t nomArchivoData nomArchivoRed parametros\n\
Con los siguientes parametros en orden:\n\
unidadesCapaOculta errorAceptable maxEpoch holdoutRate learningRate momentum\n\n\
Para cargar una red ya guardada y testearla contra un dataset:\n\
python tp.py (ej1|ej2) -l nomArchivoRed nomArchivoData\n"

if len(args) < 5:
    print usage
    sys.exit()
else:
    cmdEj1 = args[1] == "ej1"
    cmdEj2 = args[1] == "ej2"
    cmdTrain = args[2] == "-t"
    cmdLoad = args[2] == "-l"

if cmdEj1 and cmdTrain:

    if len(args) != 11:
        print "\nIncorrecta cantidad de argumentos para entrenar."
        print usage
        sys.exit()

    archivoDataset = args[3]
```

```
    archivoRed = args[4]
    unidadesCapaOculta = int(args[5])
    errorAceptable = float(args[6])
    maxEpoch = int(args[7])
    holdoutRate = float(args[8])
    learningRate = float(args[9])
    momentum = float(args[10])

    ej1(archivoDataset, archivoRed, None, None, [30, unidadesCapaOculta, 1], errorAceptable,
    maxEpoch, holdoutRate, learningRate, momentum)

elif cmdEj1 and cmdLoad:

    if len(args) != 5:
        print "\nIncorrecta cantidad de argumentos para entrenar."
        print usage
        sys.exit()

    archivoRed = args[3]
    archivoDataset = args[4]

    ej1(None, None, archivoRed, archivoDataset)

elif cmdEj2 and cmdTrain:

    if len(args) != 11:
        print "\nIncorrecta cantidad de argumentos para entrenar."
        print usage
        sys.exit()

    archivoDataset = args[3]
    archivoRed = args[4]
    unidadesCapaOculta = int(args[5])
    errorAceptable = float(args[6])
    maxEpoch = int(args[7])
    holdoutRate = float(args[8])
    learningRate = float(args[9])
    momentum = float(args[10])

    ej2(archivoDataset, archivoRed, None, None, [8, unidadesCapaOculta, 2], errorAceptable,
    maxEpoch, holdoutRate, learningRate, momentum)

elif cmdEj2 and cmdLoad:

    if len(args) != 5:
        print "\nIncorrecta cantidad de argumentos para entrenar."
        print usage
        sys.exit()

    archivoRed = args[3]
    archivoDataset = args[4]

    ej2(None, None, archivoRed, archivoDataset)

else:
    print usage
```