1. Diagnóstico de cáncer de mamas

1.1. Ejemplos

La red elegida está compuesta por la capa de entrada con 10 neuronas (una por cada feature), una capa oculta con 12 neuronas y la capa de salida con una sola neurona.

1.1.1. Pruebas con distintos learning rates

Parámetros elegidos fijos:

- beta = 5
- mini_batch_size = 1
- epochs = 1000
- epsilon = 0.05
- reg_param = 0.0

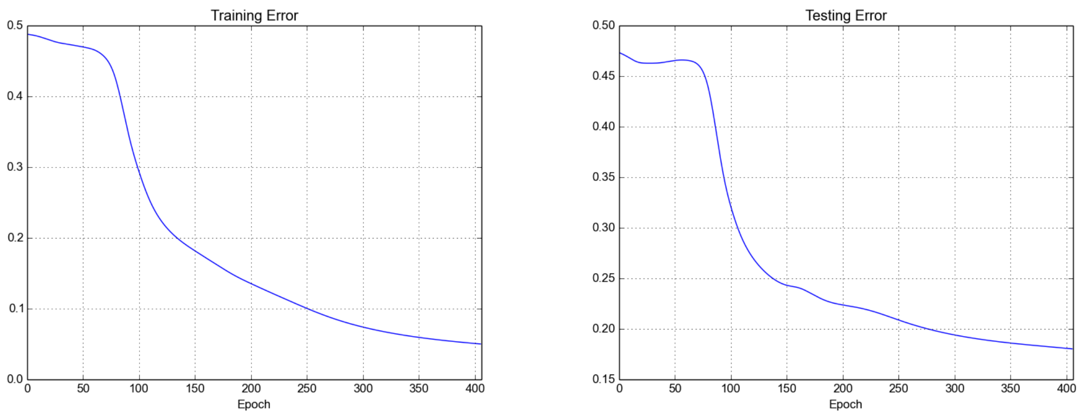

Figura 1: **learning rate: 0.0075**

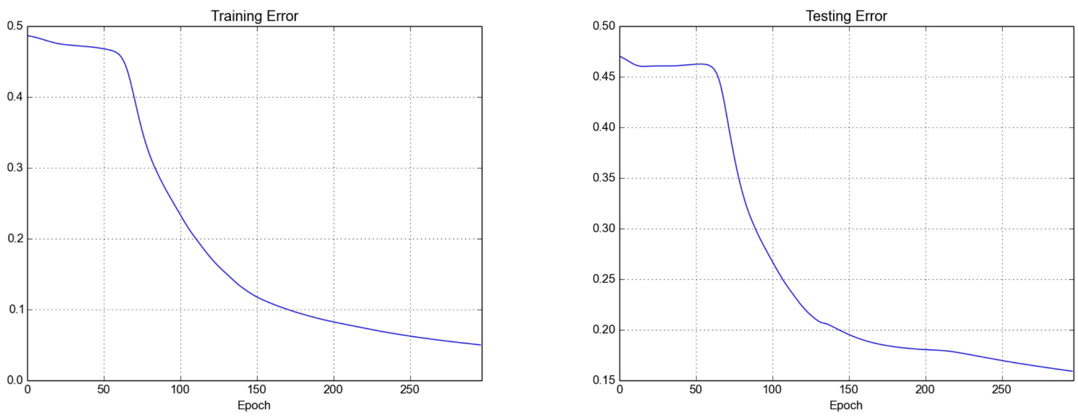

Figura 2: **learning rate: 0.01**

En estos gráficos se puede ver que el error fue menor a epsilon $= 0{,}05$. En el primero de ellos el coeficiente de aprendizaje utilizado es 0.0075 y en el segundo 0.01. Se observa que coeficiente de aprendizaje más pequeño el error sube. Por otro lado, se puede ver que cuanto más chico paso más épocas fueron necesarias para minimizar el error. Por lo que elegimos 0.01 como mejor learning rate.

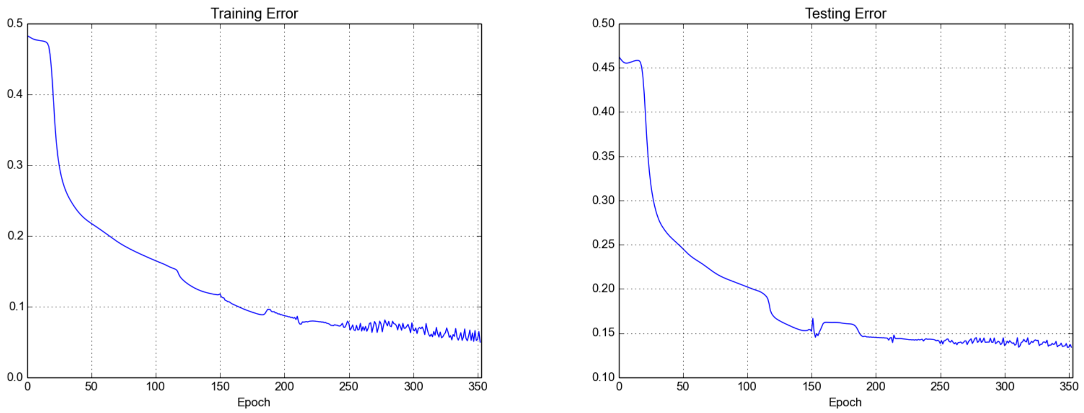

Figura 3: **learning rate: 0.025**

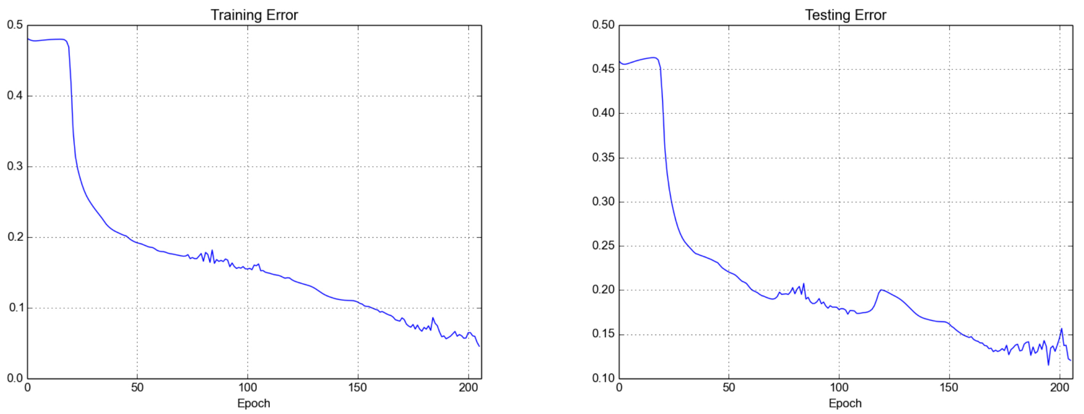

Figura 4: **learning rate: 0.05**

Estos dos gráficos muestran que al ser el coeficiente de aprendizaje es muy grande y se han encontrado mínimos locales por lo que el error empezó a oscilar. En estos casos, el error también fue menor a epsilon = 0.05 pero fueron necesarias mayor cantidad de épocas para hacerlo. Estos learning rate no son óptimos para este modelo por lo que no son los elegidos.

1.2. Modo de Uso

Para entrenar la red:

Ejemplo:

```
python trainnet1.py -m TP1/models/ej1.lmodel -o ./salida.params -t 900 -e 0.05 -l 0.005
-b 1 -x TP1/ds/tp1_ej1_training.csv
```

Explicación:

-m	Ruta al archivo de modelo (lmodel)
-o	Ruta del archivo de salida con los pesos
-t	Epochs
-e	Epsilon
-l	Learning rate
-b	Size minibatch (default 1)
-x	Archivo de samples

-m M	Ruta al archivo de modelo (lmodel)
-p O	Ruta del archivo de entrada con los pesos
-x X	Archivo de features

Para predecir:

Ejemplo:

```
python predict1.py -m TP1/models/ej1.lmodel -p TP1/salida.params -x TP1/ds/tp1_ej1_training.csv
```

Explicación:

-m	Ruta al archivo de modelo (lmodel)
-p	Ruta del archivo de entrada con los pesos
-x	Archivo de features

1.3. Código fuente

Carga de datos

Ejercicio 1:

```
import numpy as np
from cStringIO import StringIO
from preprocessor import OutlierFilter, FeatureNormalizer


class Ej1DataLoader:
    def LoadData(self, fname):


        str_data=open (fname, "r").read()

        #Reemplazo M y B por enteros
        str_data = str_data.replace("B", "0")
        str_data = str_data.replace("M", "1")

        raw_data = np.genfromtxt(StringIO(str_data), delimiter=",")
        #np.random.shuffle(raw_data)

        #Primero filtramos labels con features, luego separamos
        transformed_data=OutlierFilter().process(raw_data)

        features = transformed_data[:, 1:]
        labels = transformed_data[:, 0]
        labels = labels.reshape((labels.shape[0],1))

        #Normalizamos features
        transformed_features = FeatureNormalizer().process(features)

        return [transformed_features,labels]
```

Ejercicio 2:

```
import numpy as np

from preprocessor import OutlierFilter, FeatureNormalizer


class Ej2DataLoader:
    def LoadData(self):
        raw_data = np.genfromtxt('./ds/tp1_ej2_training.csv', delimiter=",")
        np.random.shuffle(raw_data)

        #Primero filtramos labels con features, luego separamos
        transformed_data=OutlierFilter().process(raw_data)


        #transformed_data=FeatureNormalizer().process(transformed_data)
        features = transformed_data[:, 0:8]
        transformed_features = FeatureNormalizer().process(features)


        labels = transformed_data[:, 8:]
        transformed_labels = FeatureNormalizer().process(labels)

        transformed_labels = transformed_labels.reshape((transformed_labels.shape
            [0],2))
        return [transformed_features,transformed_labels]
```

Preprocesamiento de datos

```
import numpy as np
import matplotlib.pyplot as plt
```

```
class FeatureNormalizer:
    def process(self, features):
        """
        :param features: datos a normalizar
        :return: devuelve datos normalizados por columna
        """
        feature_means = np.mean(features, axis=0)
        features_std = np.std(features, axis=0)
        features_normalized = (features - feature_means) / features_std
        return features_normalized


class OutlierFilter:
    def process(self, features):
        """
        :param features: datos
        :return: datos sin outliers
        """
        features_std = np.std(features, axis=0)
        feature_means = np.mean(features, axis=0)

        features_wo_outliers = np.array(filter(self.isOutlier(feature_means,
            features_std), features))

        # print(len(features), len(features_wo_outliers))

        # for i in range(len(features[0])):
        ##boxplot
        # fig = plt.figure(1, figsize=(9, 6))
        # ax = fig.add_subplot(111)
        # bp = ax.boxplot([features[:,i],features_wo_outliers[:,i]])
        # plt.show()

        ##histograma de la 3era feature con cortes de outliers
        # plt.hist(features[:,i],bins=np.max(features[:,i])-np.min(features[:,i]))
        # plt.axvline(feature_means[i]-2*features_std[i], color='b',
        #       linestyle='dashed', linewidth=2)
        # plt.axvline(feature_means[i]+2*features_std[i], color='b',
        #       linestyle='dashed', linewidth=2)
        # plt.show()

        # print("features_wo_outliers:", features_wo_outliers.shape)

        return features_wo_outliers

    def isOutlier(self, means, std):
        return lambda item: (np.abs(means - item) < 2 * std).all()
```

Modelo

```
import numpy as np
class LayerModel:

    # numInputUnits: Cantidad de unidades de entrada
    # hiddenLayers: Array con la cantidad de hidden layers de la red
    # numOutputLayers: Cantidad de unidades de salida
    def __init__(self, layers):
        self._layers = layers
        self._layer_sizes = [l.get_num_layers() for l in layers]
        self._activations_fns = [l.activation for l in layers]
        self._derivative_fns = [l.derivative for l in layers]

        self._biases = [np.random.randn(y, 1) / 1000 for y in self._layer_sizes[1:]]
        self._weights = [np.random.randn(y, x) / 1000
                        for x, y in zip(self._layer_sizes[:-1], self._layer_sizes[1:])
                            ]
        self._num_layers = len(self._layer_sizes)

    def getInitializedWeightMats(self):
        return self._weights

    def getInitializedBiasVectors(self):
```

```
        return self._biases

    def getZeroDeltaW(self):
        return [np.zeros(w.shape) for w in self._weights]

    def getZeroDeltaB(self):
        return [np.zeros(b.shape) for b in self._biases]

    def activation(self, layer, z):
        return self._layers[layer].activation(z)

    def derivative(self, layer, z):
        return self._layers[layer].derivative(z)

    def getNumLayers(self):
        return self._num_layers

    def getLayerSizes(self):
        return self._layer_sizes

    def getActivationFns(self):
        return self._activations_fns
    def getDerivativeFns(self):
        return self._derivative_fns
```

```
import functools

import sigmoid


class Layer:
    def get_num_layers(self):
        return self._num_layers

    def activation(self, z):
        raise NotImplementedError('subclass responsibility')

    def derivative(self, z):
        raise NotImplementedError('subclass responsibility')

class SigmoidLayer(Layer):
    def __init__(self, num_layers, beta):
        self._num_layers = num_layers
        self._activation = functools.partial( sigmoid.sigmoid_array,beta)
        self._derivative =  functools.partial( sigmoid.sigmoid_gradient_array,beta)

    def activation(self, z):
        return self._activation(z)

    def derivative(self, z):
        return self._derivative(z)

class InputLayer(Layer):
    def __init__(self, num_layers):
        self._num_layers = num_layers

    def activation(self, z):
        raise NotImplementedError('input layers do not activate')

    def derivative(self, z):
        raise NotImplementedError('input layers do not define derivative')
```

Funciones sigmoideas

```
import numpy as np


def sigmoid_array(b, x):
    return 1.0 / (1.0 + np.exp(-b * x))


def sigmoid_gradient_array(b, x):
```

```
    return b * sigmoid_array(b, x) * (1.0 - sigmoid_array(b, x))


def sigmoid_tanh_array(b, x):
    return np.tanh(b * x)


def sigmoid_tanh_gradient_array(b, x):
    return b * (1 - (np.power(sigmoid_tanh_array(b, x), 2)))
```

Algoritmos

```
import numpy as np
from layer_model import LayerModel
import sigmoid


class NetworkSolver:

    def __init__(self, layer_model, weights, biases):
        self._weights = weights
        self._biases = biases
        self._layer_model = layer_model

    def do_activation(self, sample):
        aa = [np.reshape(sample, (len(sample), 1))]
        zz = []
        # Bias
        l = 0
        for b, w in zip(self._biases, self._weights):
            l = l + 1
            z = np.dot(w, aa[-1]) + b
            a = self._layer_model.getActivationFns()[l](z)
            zz.append(z)
            aa.append(a)
        return (aa,zz)


    def do_backprop_and_return_grad(self, x, y):
        grad_w = self._layer_model.getZeroDeltaW()
        grad_b = self._layer_model.getZeroDeltaB()

        # feedforward
        activations, zs = self.do_activation(x)
        delta = (activations[-1] - np.reshape(y, (len(y), 1))) * self._layer_model.
            getDerivativeFns()[-1](zs[-1])
        grad_b[-1] = delta
        grad_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in xrange(2, self._layer_model.getNumLayers()):
            z = zs[-l]
            sp = self._layer_model.getDerivativeFns()[-l](z)
            delta = np.dot(self._weights[-l+1].transpose(), delta) * sp
            grad_b[-l] = delta
            grad_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (grad_b, grad_w)



    def correction_mini_batch(self, mini_batch, lr, n, lmbda=0.0):
        """
        :param mini_batch: set de entrenamiento
        :param lr: learning rate
        :param n: cantidad de samples
        :param lmbda: regularization parameter
        """
        grad_b = [np.zeros(b.shape) for b in self._biases]
        grad_w = [np.zeros(w.shape) for w in self._weights]
        for x, y in mini_batch:
            delta_grad_b, delta_grad_w = self.do_backprop_and_return_grad(x, y)
            grad_b = [gb+deltagb for gb, deltagb in zip(grad_b, delta_grad_b)]
            grad_w = [gw+deltagw for gw, deltagw in zip(grad_w, delta_grad_w)]
```

```
        self._weights = [(1.0 - lr*(lmbda/n)) * w - (lr/len(mini_batch)) * gw
                         for w, gw in zip(self._weights, grad_w)]
        #clasico sin regularizacion
        #    self._weights = [w - (lr / len(mini_batch)) * gw
        #                  for w, gw in zip(self._weights, grad_w)]
        self._biases = [b - (lr / len(mini_batch)) * gb
                        for b, gb in zip(self._biases, grad_b)]


    def learn_minibatch(self, mini_batches, mini_batches_testing, lr, epochs, epsilon,
         lmbda=0.0):
        """
        :param mini_batches: set de entrenamiento
        :param mini_batches_testing: set de testing
        :param lr: learning rate
        :param epochs: cantidad de epocas
        :param epsilon: cota de error
        :param lmbda: parametro de regularizacion, si no se especifica, no se
    regulariza
        imprime errores de entrenamiento y testing por cada epoca
        """
        T = epochs
        t = 0
        e = 999
        n = sum([len(mbatch) for mbatch in mini_batches])
        while e > epsilon and t < T:
            for b in mini_batches:
                self.correction_mini_batch(b, lr, n, lmbda)
            t = t + 1
            e = self.get_prediction_error(mini_batches, False)
            et = self.get_prediction_error(mini_batches_testing, False)
            print ("Training Error: ", e, "Val error:", et)

        #e = self.get_prediction_error(mini_batches, True)


    def get_prediction_error(self, mini_batches, bprint):
        e = 0
        cant = 0
        for b in mini_batches:

            for x, y in b:
                cant = cant + 1
                aa, zz = self.do_activation(x)

                e = e + np.linalg.norm(aa[-1] - y)
                if bprint:
                    print e / cant, np.linalg.norm(aa[-1] - y),aa[-1][0][0], y[0]

        return e / cant

    def predict(self, batch):
        e = 0
        for x, y in batch:
            aa, zz = self.do_activation(x)
            e = e + np.linalg.norm(aa[-1] - y)
            print "Original: ", y[0], "Predicted:",aa[-1][0]

        return e / len(batch)

    def get_hits(self, test_data):
        """
        :param test_data: set de datos de testing
        :return: Devuelve el numero de aciertos de inputs de test para los que
        los outputs que devuelve la red son correctos.
        """
        test_results = [(self.get_result(self.do_activation(x)[0][-1]), y)
                         for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def get_result(self, act):
        """
```

```
        :param act: resultado de nuestra red
        :return: el resultado es el mas cercano al resultado
        que devolvio la red entre 0 y 1
        """
        return np.argmin([abs(act-0), abs(act-1)])
```

Test

Ejercicio 1:

```
import argparse
import ej1_data_loader
from model_io import ModelIO
from params_io import ParamsIO
from layer_model import LayerModel
from feed_forward_solver import NetworkSolver
import functools


parser = argparse.ArgumentParser(description='Parametros de la red')


parser.add_argument('-m',  type=str,
                    help='Ruta al archivo de modelo (lmodel)', required=True)

parser.add_argument('-p', type=str,
                    help='Ruta del archivo de entrada con los pesos', required=True)

parser.add_argument('-x', type=str, default=1,
                    help='Archivo de features', required=True)

args = parser.parse_args()


loader = ej1_data_loader.Ej1DataLoader()
data = loader.LoadData(args.x)
features = data[0] #shape=(333,10)
labels = data[1]

test_data = zip(features, labels)

mini_batches_testing = [test_data]


mloader = ModelIO()
model =  mloader.load_model(args.m)

ploader = ParamsIO()
weights, biases = ploader.load_params(args.p)

solver = NetworkSolver(model,weights=weights,biases=biases)

E = solver.predict(mini_batches_testing[0])

print "Error cuadratico promedio: ", E

import argparse
import ej1_data_loader
from model_io import ModelIO
from params_io import ParamsIO
from layer_model import LayerModel
from feed_forward_solver import NetworkSolver
import functools


parser = argparse.ArgumentParser(description='Parametros de la red')


parser.add_argument('-m', metavar='M', type=str,
                    help='Ruta al archivo de modelo (lmodel)', required=True)

parser.add_argument('-o', metavar='O', type=str,
```

```
                    help='Ruta del archivo de salida con los pesos', required=True)

parser.add_argument('-t', metavar='T', type=int,
                    help='Epochs', required=True)

parser.add_argument('-e', metavar='E', type=float,
                    help='Epsilon', required=True)

parser.add_argument('-l', metavar='L', type=float,
                    help='Learning rate', required=True)

parser.add_argument('-b', metavar='B', type=int, default=1,
                    help='Size minibatch (default 1)', required=True)

parser.add_argument('-x', metavar='X', type=str, default=1,
                    help='Archivo de samples', required=True)

args = parser.parse_args()


loader = ej1_data_loader.Ej1DataLoader()
data = loader.LoadData(args.x)
features = data[0] #shape=(333,10)
labels = data[1]

all_data = zip(features, labels)
num_training_samples = int(len(all_data) * 0.75)
num_test_samples = len(all_data) - num_training_samples

training_data = all_data[0:num_training_samples - 1]
test_data = all_data[num_training_samples:len(all_data) -1]

mini_batch_size = args.b
n = len(training_data)

mini_batches_training = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, num_training_samples - 1, mini_batch_size)]

mini_batches_testing = [
                test_data[k:k+mini_batch_size]
                for k in xrange(0, num_test_samples - 1, mini_batch_size)]

mloader = ModelIO()
model =  mloader.load_model(args.m)

solver = NetworkSolver(model,weights=model.getInitializedWeightMats(),biases=model.
    getInitializedBiasVectors())

lr = args.l
epochs =  args.t
epsilon =  args.e
reg_param = 0.0
solver.learn_minibatch(mini_batches_training,mini_batches_testing,lr,epochs,epsilon,
    reg_param)
pio=ParamsIO()
pio.save_params( args.o , solver._weights, solver._biases)
print "Pesos guardados en ", args.o
```

Ejercicio 2:

```
import sigmoid
import ej2_data_loader
from layer_model import LayerModel
from feed_forward_solver import NetworkSolver
import functools

loader = ej2_data_loader.Ej2DataLoader()
data = loader.LoadData()
features = data[0] #shape=(333,10)
labels = data[1]

all_data = zip(features, labels)
```

```
num_training_samples = int(len(all_data) * 0.75)
num_test_samples = len(all_data) - num_training_samples

training_data = all_data[0:num_training_samples - 1]
test_data = all_data[num_training_samples:len(all_data) -1]

mini_batch_size = 1
n = len(training_data)
beta = 0.01

mini_batches_training = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, num_training_samples - 1, mini_batch_size)]

mini_batches_testing = [
                test_data[k:k+mini_batch_size]
                for k in xrange(0, num_test_samples - 1, mini_batch_size)]

model =  LayerModel([8,10,2], functools.partial( sigmoid.sigmoid_array,beta),functools
    .partial( sigmoid.sigmoid_gradient_array,beta))
solver = NetworkSolver(layer_model=model)

lr = 2.5
epochs = 1000
epsilon = 0.05
reg_param = 0.0
solver.learn_minibatch(mini_batches_training,mini_batches_testing,lr,epochs,epsilon,
    reg_param)
```