Trabajo Práctico 2

Redes Neuronales

1er Cuatrimestre 2018

UBA

Alumnos:

- Eden Torres, LU: 017/17
- Gino Marceca, doctorado.

Índice

Breve introducción al problema

El problema consiste en clasificar en forma no supervisada un conjunto de datos correspondientes a 900 documentos de texto donde cada uno pertenece a una dada categoría (1 a 9) y es representado mediante un vector de 850 atributos en el que se indica en cada uno la frecuencia de repetición de una dada palabra. Los 900 documentos fueron separados en 850 y 50 correspondientes a los sets de entrenamiento y validacion respectivamente.
Dada la alta dimensionalidad del problema, se plantea utilizar el método PCA para reducir la dimensión y poder visualizar los datos en un plot 3-D. Los datos comprimidos serán luego utilizados para el entrenamiento y validación mediante un algoritmo de mapeo autoorganizado que permite clasificar cada documento en una grilla 2-D.
Se utilizaron las reglas de aprendizaje de Sanger / Oja para descomponer los documentos en su 3 primeras componentes principales / 3 vectores equivariantes, respectivamente.
Para el caso del algoritmo de mapeo autoorganizado, se comparó el resultado de la clasificación usando los datos crudos (sin PCA) contra los datos comprimidos (PCA en 2, 3 y 9 dimensiones) y para distintas configuraciones de la grilla 2-D.

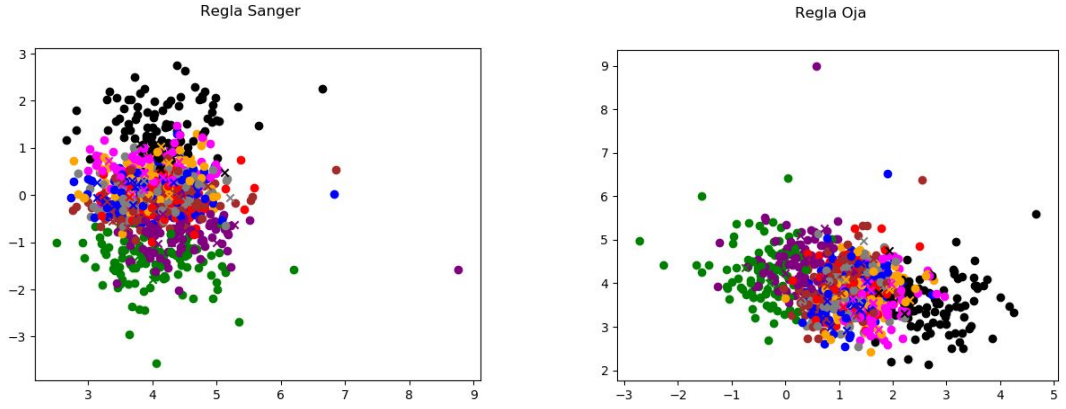
Solución

Ejercicio 1

En este ejercicio se pide comprimir la alta dimensionalidad de los features (850) en sus 3 primeras componentes principales. Para esto se utilizó un modelo de red neuronal basado en un aprendizaje Hebbiano en el que consiste en 2 capas, una de entrada de 850 nodos y una de salida de 3 nodos. Para la actualización de los pesos se utilizaron la reglas de Oja y Sanger.
En lo que respecta al aprendizaje, un learning rate de 0.01 y (85000 / 5000) iteraciones para los sets de (entrenamiento / validacion) respectivamente fueron suficientes para lograr la descomposición en las primeras componentes principales.

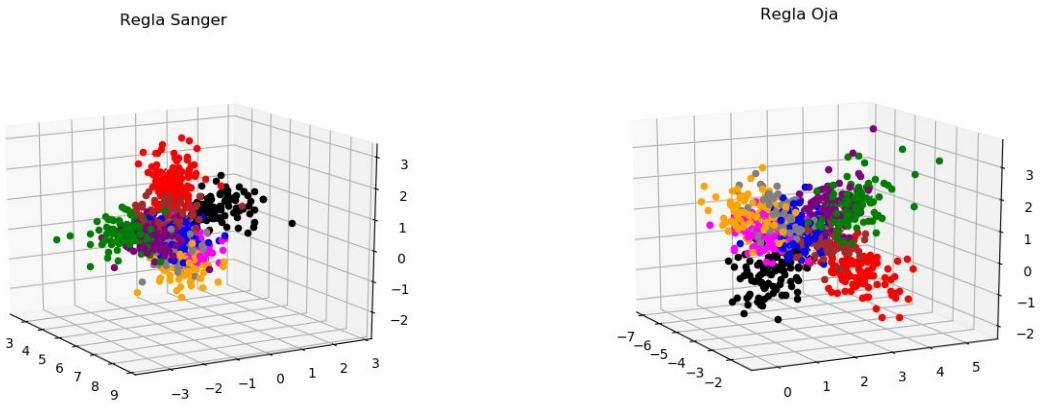
A continuación se observan los resultados correspondientes a usar PCA en 2 y 3 dimensiones usando la regla de Sanger y Oja, izquierda y derecha respectivamente. Cada color corresponde a la etiqueta de cada documento (1 a 9). Los círculos y cruces corresponden a los sets de entrenamiento y validación respectivamente.
Como se puede observar, los datos obedecen una distribución en forma de clusters siendo mas facil/dificil de distinguir dependiendo de la categoría y de la dimensionalidad del problema.

PCA en 2 Dimensiones



PCA en 2-D no es suficiente para lograr una separación de los clusters. Una posible solución para lograr una separación óptima en 2-D es hacer un mapeo de características utilizando un algoritmo de mapeo autoorganizado, como se verá en el ejercicio 2.

PCA en 3 Dimensiones



PCA en 3-D funciona mejor, sin embargo aún quedan regiones de overlap entre distintas categorías, siendo no posible su distinción.
Curiosamente se observa una mejor separación de categorías usando la regla de Oja que la de Sanger. Una posible hipótesis al respecto es que si bien Oja no obtiene las 3 primeras componentes principales, los vectores que obtiene generan el mismo subespacio que las 3 componentes principales y además tienen igual varianza al hacer la proyección. Por lo tanto, en este ejemplo en particular, tener equivarianza parece ser mejor para distinguir los clusters que descomponer en las componentes principales.

Ejercicio 2

En el ejercicio 2 se nos pide construir un modelo de mapeo de características auto-organizado que clasifique automáticamente los documentos en un arreglo de dos dimensiones.
Lo que primero hicimos fue definir algunas variables de entrada. Decidimos que plano de salida sea de tamaño 25 * 25, que el radio inicial sea el tamaño del mayor eje dividido 2, que el learning rate inicial sea 0.1. Luego para t1 y t2 tomamos los valores recomendados por la clase práctica.

Antes de arrancar con el training Inicializamos los pesos con un valor random. Para continuar, agarramos un input random y calculamos la neurona más cercana. Al obtenerlo, actualizamos el peso de esa neurona y de las vecinas. Repetimos esto 3000, achicando en cada paso el área de vecindad y el learning rate.

En los siguientes gráficos vemos luego de 3000 iteraciones como quedan clasificados los datos. Están pintados por categorías. Las cruces representan los datos de validación y los círculos de training. Los círculos están pintados con transparencias así se aprecia la mayor concentración de los datos en algunas zonas.

Gráfico con input de 3 dimensiones

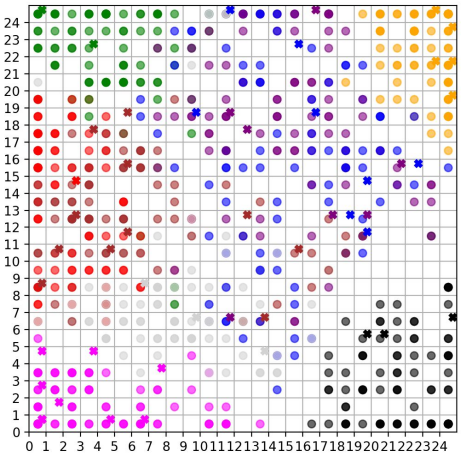


Gráfico con input de 9 dimensiones

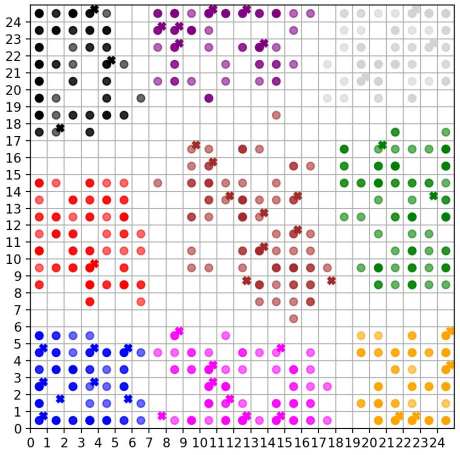
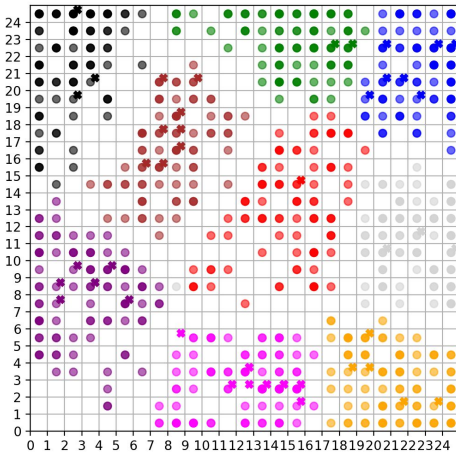


Gráfico con input de 850 dimensiones



En el gráfico donde el input está en 3 dimensiones, se ve que hay salidas que tienen más de una categoría. Luego para los gráficos de 9 y 850 dimensiones se ve que esto no ocurre y puede clasificar correctamente las categorías.

Observamos que subir el número de iteraciones a 4000 no genera demasiados cambios en los gráficos.

Al achicar el learning rate, la red tarda mucho más en aprender de manera parecida a la actual. Y si agradamos este valor, la red no aprende correctamente.
De manera similar se comporta el radio inicial al modificarlo. Si lo agrandamos mucho la red no aprende de manera correcta y si se achica, los datos quedan amontonados en una zona pequeña del plano.

Por último, dado los datos que tenemos llegamos a la conclusión que el tamaño ideal del plano de entrada es entre 20 y 25. Si es más grande queda espacio de más y la separación de las categorías no adecuada. Y si es mas chico, no se llega a apreciar la separación de categorías. Podemos observar esto mismo en los siguientes gráficos:

Gráfico con input de 850 dimensiones

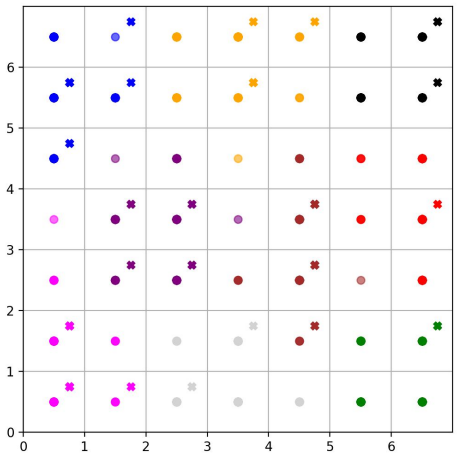
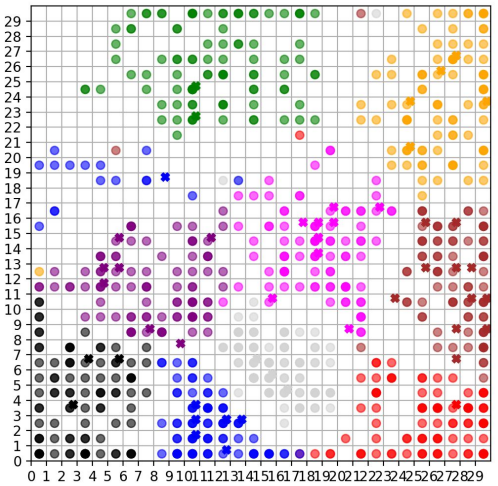


Gráfico con input de 850 dimensiones



El código fuente impreso del programa

Ejercicio 1

```
import numpy as np
from mpl_toolkits import mplot3d

import matplotlib
matplotlib.use('TkAgg') #TEMP
import matplotlib.pyplot as plt


class NeuralNetwork:

    def __init__(self, data, validatingDataSet, oja=True):
        # Seteamos una seed
        np.random.seed(1)

        self.trainingDataSet = data[:, 1:data.shape[1]]
        self.validatingDataSet = validatingDataSet[:, 1:data.shape[1]]

        # Obtenemos las categorias de la data
        self.trainingDataLabels = data[:, 0:1]
        self.validatingDataLabels = validatingDataSet[:, 0:1]

        # Definimos el numero de iteraciones que queremos correr
        self.numeroDeIteraciones = 100

        # Seteamos las dimesiones de los datos
        self.dimensionSalida = 3
        self.dimensionEntrada = self.trainingDataSet.shape[1]

        # Inicializo los pesos en valores random
        self.weights = np.array((self.dimensionSalida, 0), dtype=float)
        self.randoms = np.random.rand(self.dimensionSalida, self.dimensionEntrada);
        self.weights = (2 * self.randoms - 1) * 0.25

        self.learningValue = 0.01

        # Definimos si queremos correr la regla de oja o sanger
        self.oja = oja

    def train(self):
        count = 0
        while (count < self.numeroDeIteraciones) :
            for index in range(0, self.trainingDataSet.shape[0]) :
                x = self.trainingDataSet[index].reshape(self.dimensionEntrada, 1)

                # Calculamos las salidas S en forma vectorial S
                salidas = np.dot(self.weights, x)
                assert(salidas.shape == (self.dimensionSalida, 1))

                if (self.oja): # Regla de Oja en forma vectorial
                    self.weights = self.weights + self.learningValue * np.dot(salidas, (x -
np.dot(self.weights.T, salidas)).T)
```

```
                else: # Regla de Sanger
                    for i in range(0, self.dimensionSalida):
                        for j in range(0, self.dimensionEntrada):
                            suma = 0
                            for k in range(i + 1):
                                suma += salidas[k, 0] * self.weights[k, j]
                            self.weights[i,j] =  self.weights[i,j] + self.learningValue *
salidas[i, 0] * (x[j, 0] - suma)

            count += 1

        # Creamos los outpus vacios
        trainingOutputs = np.empty((self.trainingDataSet.shape[0], 3), dtype=float)
        validationgOutputs = np.empty((self.validatingDataSet.shape[0], 3), dtype=float)

        # Calculamos las salidas
        for i in range(self.trainingDataSet.shape[0]):
            trainingOutputs[i] = np.dot(self.trainingDataSet[i], np.transpose(self.weights))
        for i in range(self.validatingDataSet.shape[0]):
            validationgOutputs[i] = np.dot(self.validatingDataSet[i], np.transpose(self.weights))

        # Creamos el grafico
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        fig.suptitle('Regla Oja' if self.oja else 'Regla Sanger')

        # Definimos los colores
        colors = ['gray', 'black', 'red', 'orange', 'green', 'blue', 'magenta', 'purple',
'brown']

        # Graficamos las salidas
        for i in range(trainingOutputs.shape[0]):
            ax.scatter3D(trainingOutputs[i][0], trainingOutputs[i][1], trainingOutputs[i][2],
c=colors[int(self.trainingDataLabels[i][0])-1],  marker='o');
        for i in range(validationgOutputs.shape[0]):
           ax.scatter3D(validationgOutputs[i][0], validationgOutputs[i][1],
validationgOutputs[i][2], c=colors[int(self.validatingDataLabels[i][0])-1],  marker='x');

        plt.show()

if __name__ == '__main__':
    filename = 'tp2_training_dataset.csv'
    raw_data = open(filename, 'rt')
    data = np.loadtxt(raw_data, delimiter=",", usecols=range(851))

    # Dividimos la data en un set de training y un set de validacion
    trainingDataSet = data[0:850]
    validatingDataSet = data[850:900]

    nn = NeuralNetwork(trainingDataSet, validatingDataSet, True)
    nn.train()
```

Ejercicio 2

```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import math as math
from sklearn.decomposition import PCA

class SOM:
    def __init__(self, data, validatingDataSet, trainingLabels, validatingLabels):
        # Seteamos una seed
        np.random.seed(1)

        self.data = data
        self.validatingDataSet = validatingDataSet
        self.trainingDataLabels = trainingLabels
        self.validatingDataLabels = validatingLabels

        # Definimos el numero de iteraciones que queremos correr
        self.numeroDeIteraciones = 3000

        # Seteamos las dimesiones de los datos
        self.dimensionSalida = 2
        self.dimensionEntrada = self.data.shape[1]

        # Seteamos el tamaño del plano
        self.sizeX = 25
        self.sizeY = 25

        # Seteamos los pesos de manera random
        self.weights = np.array((self.sizeX , self.sizeY, self.dimensionEntrada), dtype=float)
        self.randoms = np.random.rand(self.sizeX, self.sizeY, self.dimensionEntrada);
        self.weights = self.randoms

        # Seteamos los parametros iniciales para el enfriamento
        self.radioInicial = max(self.sizeX, self.sizeY) / 2
        self.t1 = 1000 / math.log10(self.radioInicial)
        self.t2 = 1000
        self.initLearningRate = 0.1

    def train(self):
        count = 0
        while (count < self.numeroDeIteraciones) :
            # Elegimos un dato random
            randomEntrie = np.random.randint(low=0, high=self.data.shape[0])
            inputX = self.data[randomEntrie].reshape(self.dimensionEntrada,1)

            # Buscamos el indice que mas se parece a x
            indexMinimo = self.encontrarMinimo(inputX)

            # Obtenemos el radio y el learning rate para esta iteracion
            radio = self.obtenerRadio(count)
            learningRate = self.obtenerLearningRate(count)

            for x in range(self.weights.shape[0]):
                for y in range(self.weights.shape[1]):
```

```
                    # Para cada peso calculamos la distancia entre el peso y el indice minimo
                    weight = self.weights[x][y].reshape(inputX.shape[0], 1)
                    distancia = np.sum((np.array([x, y]) - indexMinimo) ** 2)
                    # Si esta en la zona de vecinidad actualizamos los pesos
                    if distancia <= radio ** 2:
                        vecinidad = self.obtenerVecinidad(distancia, radio)
                        self.weights[x][y] = (weight + (learningRate * vecinidad * (inputX -
weight))).reshape(inputX.shape[0])
            count += 1

        # Creamos los outpus vacios
        trainingOutputs = np.empty((self.data.shape[0], 2), dtype=float)
        validationgOutputs = np.empty((validatingDataSet.shape[0], 2), dtype=float)

        # Definimos los colores. Los de entramiento tienen un alpha asi se nota la densidad de
inputs que caen en un cuadrante
        colors = ['#D3D3D395', '#00000095', '#FF000095', '#FFA50095', '#00800095', '#0000FF95',
'#FF00FF95', '#80008095', '#A52A2A95']
        validatingColors = ['#D3D3D3', '#000000', '#FF0000', '#FFA500', '#008000', '#0000FF',
'#FF00FF', '#800080', '#A52A2A']

        # Creamos el grafico
        fig, ax = plt.subplots(figsize=(6, 6))
        fig.suptitle('Gráfico con input de ' + str(self.dimensionEntrada) + ' dimensiones')

        # Calculamos los outputs y los graficamos
        for i in range(trainingOutputs.shape[0]):
            trainingOutputs[i] = self.encontrarMinimo(self.data[i].reshape(self.dimensionEntrada,
1))
            plt.plot(trainingOutputs[i][0]+.5, trainingOutputs[i][1]+.5,  marker='o',
color=colors[int(self.trainingDataLabels[i][0])-1])

        for i in range(validatingDataSet.shape[0]):
            validationgOutputs[i] =
self.encontrarMinimo(self.validatingDataSet[i].reshape(self.dimensionEntrada, 1))
            plt.plot(validationgOutputs[i][0]+.75, validationgOutputs[i][1]+.75 ,  marker='X',
color=validatingColors[int(self.validatingDataLabels[i][0])-1])

        # Terminamos de configurar el grafico
        ax.set_xlim([0, self.sizeX])
        ax.set_ylim([0, self.sizeY])
        plt.xticks(np.arange(self.sizeX))
        plt.yticks(np.arange(self.sizeY))
        ax.grid(which='both')
        plt.show()

    # Esta funcion encuentra el peso mas cercano al input dado. Devuelve los indices de ese peso
    def encontrarMinimo(self, inputX):
        indiceMinimo = np.array([0, 0])
        # Calculamos la primera distancia
        primeraDistancia = np.sum((self.weights[0][0].reshape(self.dimensionEntrada, 1) - inputX)
** 2)
        distanciaMinima = primeraDistancia

        for x in range(self.weights.shape[0]):
            for y in range(self.weights.shape[1]):
                # Para todo peso vemos si la distancia es menor a la que tengo guardada
```

```
                weight = self.weights[x][y].reshape(self.dimensionEntrada, 1)
                distancia = np.sum((weight - inputX) ** 2)
                if distancia < distanciaMinima:
                    distanciaMinima = distancia
                    indiceMinimo = np.array([x, y])
        return indiceMinimo

    # Obtiene el tamaño del radio en la iteracion i
    def obtenerRadio(self, i):
        return self.radioInicial * np.exp(-i / self.t1)

    # Obtiene el valor del learing rate en la iteracion i
    def obtenerLearningRate(self, i):
        learningRate = self.initLearningRate * np.exp(-i / self.t2)
        return learningRate

    # Obtiene la funcion de vecinidad
    def obtenerVecinidad(self, distancia, varianza):
        return np.exp(- distancia / (2 * (varianza ** 2)))

if __name__ == '__main__':
    filename = 'tp2_training_dataset.csv'
    raw_data = open(filename, 'rt')
    data = np.loadtxt(raw_data, delimiter=",", usecols=range(851))

    originalData = True
    threeDimesionData = False
    nineDimesionData = False

    # Obtenemos las categorias de la data
    trainingLabels = data[0:850, 0:1]
    validatingLabels = data[850:900, 0:1]

    if originalData == True:
        trainingDataSet = data[0:850, 1:851]
        validatingDataSet = data[850:900, 1:851]

    elif threeDimesionData == True:
        # Transformamos la data en 3 dimeniones
        pca = PCA(n_components = 3)
        principalComponents = pca.fit_transform(data[:, 1:851])

        trainingDataSet = principalComponents[0:850]
validatingDataSet = principalComponents[850:900]

    elif nineDimesionData == True:
        # Transformamos la data en 9 dimeniones
        pca = PCA(n_components = 9)
        principalComponents = pca.fit_transform(data[:, 1:851])

        trainingDataSet = principalComponents[0:850]
        validatingDataSet = principalComponents[850:900]

    som = SOM(trainingDataSet, validatingDataSet, trainingLabels, validatingLabels)
    som.train()
```

Conclusiones

De cada ejercicios pudimos concluir lo siguiente:

Ejercicio 1

- El learning rate debe ser chico. Usamos un learning rate de 0.01
- PCA en 2-D no es suficiente para lograr una distinción entre categorías.
- PCA en 3-D funciona mejor pero aún quedan regiones de overlap imposible de distinguir.
- La regla de oja logra una mejor separación que la de Sanger.
- La red aprendía más rápido recorriendo todos los inputs en cada iteración, en vez de agarrar un input random y repetir el procedimiento la misma cantidad de iteraciones por la cantidad de inputs

Ejercicio 2

- Al reducir la cantidad de dimensiones de los inputs de entrada, le cuesta más a la red aprender
- El learning rate no tienen que ser tan chico como el ejercicio 1
- El tamaño del plano debería tener las dimensiones de la longitud de las dos primeras componentes principales de los datos de entrada
- El radio inicial del área de vecindad tiene que estar relacionado al tamaño del plano
- En cada paso se tiene achicar el learning rate y el área de vecindad. Así la actualización de los pesos se va volviendo más exacta con el tiempo