TP2 Redes Neuronales

Suárez, Germán y Suárez Barón, Juan Carlos

Julio 2019

Introducción

En este trabajo se desarrollaron dos modelos distintos de redes neuronales artificiales sobre el mismo conjunto de datos. Los modelos se basaron en aprendizaje hebbiano no supervisado (implementando los códigos de los métodos de Oja y Sanger) y redes autoorganizadas (mapa auto-organizado) utilizados en una tarea de clasificación. Los métodos de Oja y Sanger se usaron para reducir la dimensión de un set de datos de 900x851 que contenía documentos de 9 categorias. La reducción de dimensiones se hizo al proyectar cada observación de los datos sobre sus componentes principales. Para mapa auto-organizado se usó primero todo el set de datos y luego el set de datos reducido, una vez proyectando todos los datos sobre las primeras tres componentes principales, y después proyectando todos los datos sobre las primeras nueve. El mapa autorganizado permitió agrupar los documentos en grupos según la categoría a la que pertenecían, activando neuronas de salidas cercanas para patrones de entrada cercanos.

1. Modos de uso del programa

1.1. Aprendizaje Hebbiano

Para calcular las componentes principales utilizamos los algoritmos Oja y $Sanger$, los cuales implementamos en dos *scripts*: *oja.py* y *sanger.py*, ambos llamados dentro de $TP2.py$. El script *oja.py* está basado en la clase llamada Oja, la cual contiene las funciones $forward$, $updateWeight$ y $train$. El script *sanger.py* está basado en la clase $Sanger$, la cual tiene una estructura idéntica a Oja, salvo por la diferencia en el modo de actualizar los pesos.

Dentro de $TP2.py$ se llama a la clases Oja y $Sanger$ a través de las líneas: $red3 = Sanger(inputs, 3)$ y $red = Oja(inputs, 3)$, donde $inputs$ corresponde al parámetro $DimTrain$ que va a recibir el tamaño de las neuronas de entrada. Además, tanto Oja como $Sanger$ incluyen un parámetro correspondiente a la tasa de aprendizaje eta, el cual preseteamos en 0.001. Ambas clases actualizan los pesos mediante la función $updateWeight$ la cual es llamada desde la función $train$, ésta última recorre por épocas a todos los datos de entrada. Antes de aplicar Oja y $Sanger$, separamos los datos en un 80 % para entrenamiento, y un 20 % para validación. Una vez entrenada la red, proyectamos los datos de entrenamiento y los de validación sobre las primeras 3 componentes principales y los graficamos en un mismo $scatter - plot$ en 3D, diferenciando las categorías por colores, y utilizando círculos para los datos de entrenamiento y triángulos para los de validación.

1.2. Mapas Auto-Organizados

Aquí armamos un mapa autoorganizado en un script llamado *som.py*, el cual se basa en el algoritmo $Kohonen$. Dentro del script tenemos cuatro funciones: h, la cual calcula la distancia entre dos neuronas de salida dadas y, junto con un parámetro llamado $sigma$, calcula y devuelve la función de vecindad del SOM aplicado a dichas neuronas; $winner$, que se encarga de encontrar la neurona de salida que tiene asociada los pesos más cercanos al patrón de entrada elegido (mediante distancia euclídea), $updateWeigth$, que modifica los pesos utilizando las dos anteriores funciones y un parámetro eta que corresponde a la tasa de aprendizaje; y por último la función $train$ con la que se entrena a la red. Esta última función empieza asignándole a los pesos asociados a cada neurona de salida un patrón de entrada elegido al azar. Para la etapa de ordenamiento comenzamos con un $eta = 0{,}1$ y un $sigma = 3$, los cuales iremos reduciendo en cada

iteración, es decir, para cada vez que tomemos de manera aleatoria un patrón de entrada para entrenar a la red. La etapa de ordenamiento concluye cuando $eta < 0{,}01$. A partir de ese momento comienza la etapa de convergencia, la cual funciona de la misma manera que la de ordenamiento, con la diferencia que antes de iterar, le damos nuevos valores a *eta* y a *sigma*, 0.01 y 0.1, respectivamente. La convergencia finalizará cuando alguno de estos parámetros sean menores que $1e{-}9$. Cuando tenemos entrenada a nuestra red, desde el script *TP2.py* le aplicamos la función *winner* a los datos de entrenamiento y de validación, contando las veces que se activó cada neurona de salida, y cuál fue la categoría del patrón de entrada que la activó. Finalmente, cada grilla del mapa tendrá el color de la categoría que más veces activó a la neurona correspondiente. De esta manera, graficamos 3 pares de mapas: Los primeros dos corresponden a los mapas que obtuvimos de aplicar la red a los datos de entrenamiento y de validación para todos las variables de entrada (sin proyectar). El siguiente par corresponde al mapa obtenido de aplicar la red a los datos de entrenamiento y validación proyectados sobre las primeras tres componentes principales. El último par igual que los anteriores, pero con los datos proyectados sobre las primeras nueve componentes principales.

2. Pruebas y resultados de los algoritmos

2.1. Aprendizaje Hebbiano

La figura 1 muestra la representación de las primeras 3 componentes principales obtenidas con la regla de Sanger (no graficamos utilizando la regla de Oja porque obtuvimos resultados muy similares).

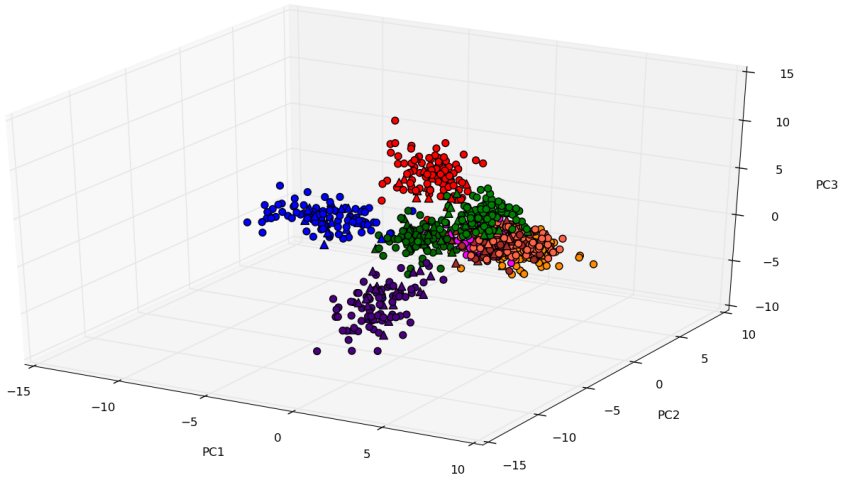

Figura 1: Representación gráfica de los datos proyectados sobre las componentes principales

La figura 2 representa las proyecciones de las primeras 3 componentes principales obtenidas de la figura 1, vistas en perspectiva.

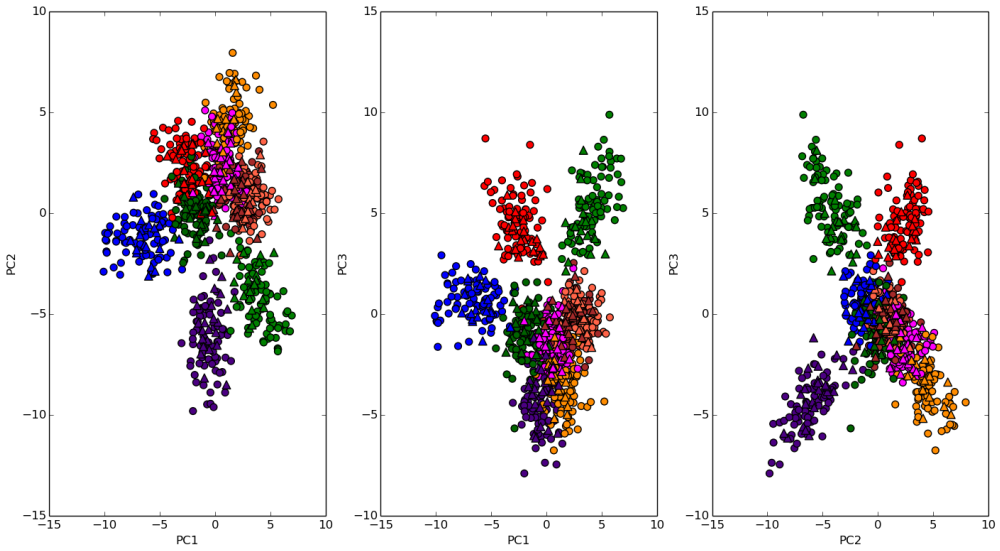

Figura 2: Plano 2D de los datos proyectados sobre las tres primeras componentes principales obtenidas con la regla de Sanger, en perspectiva

2.2. Mapas Auto-organizados

El primer par de mapas lo obtuvimos a la aplicar la red entrenada a todos los datos sin proyectar, el primero corresponde a los datos de entrenamiento y el segundo a los de validación. Obtuvimos un error aproximado entre ambos mapas de 0.167.

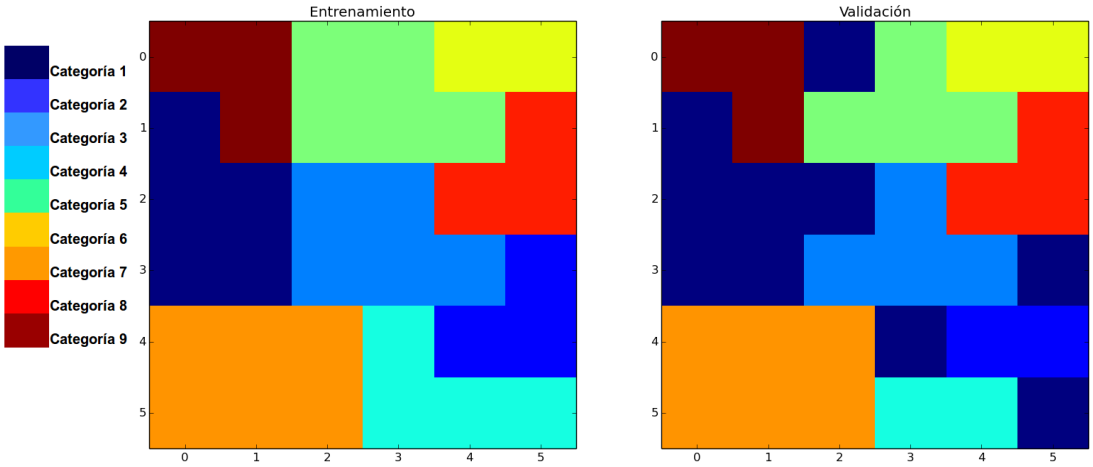

Figura 3: Datos de entrenamiento y validación después del ordenamiento

El segundo par lo obtuvimos a la aplicar la red entrenada a todos los datos proyectados sobre las primeras tres componentes principales. Obtuvimos un error aproximado entre ambos mapas de 0.333.

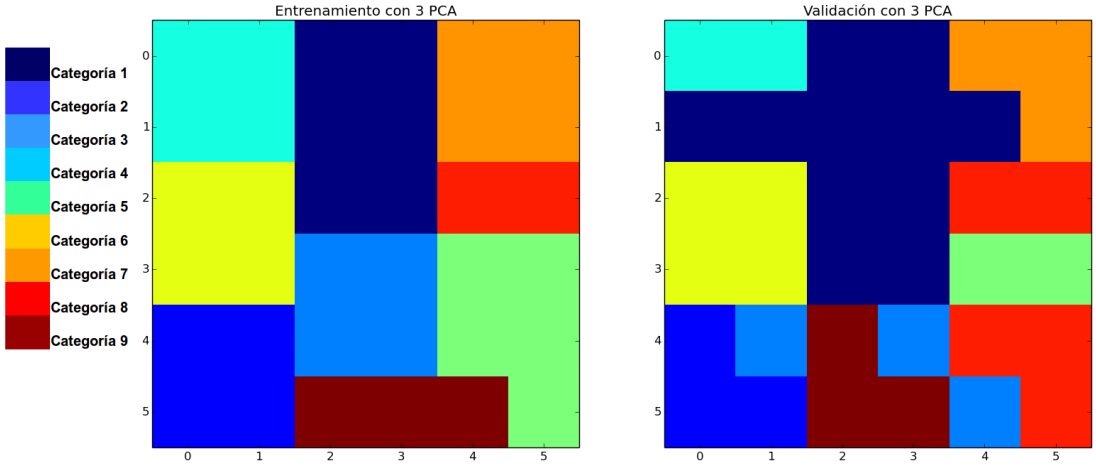

Figura 4: Entrenamiento y validación con 3 componentes principales

El tercer par lo obtuvimos a la aplicar la red entrenada a todos los datos proyectados sobre las primeras nueve componentes principales. Obtuvimos un error aproximado entre ambos mapas de 0.167.

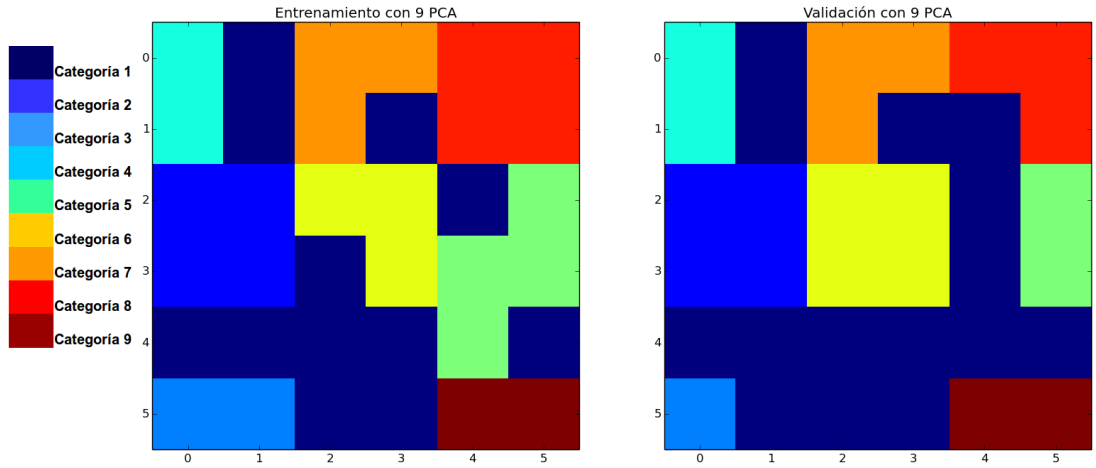

Figura 5: Entrenamiento y validación con 9 componentes principales

3. Conclusiones

Proyectar los datos sobre sus componentes principales disminuye el tiempo de ejecución del SOM debido a las dimensiones reducidas, se debe tener en cuenta que esto implica una etapa adicional en el entrenamiento, aumentando la complejidad del algoritmo. La calidad de los resultados se vió visiblemente afectada, inferimos que al estar los datos menos dispersos esto hace que la diferencia entradas - pesos se

haga más chica en cada patrón, como consecuencia, los patrones no quedan bien clasificados. Al igual que con el aprendizaje supervisado, estas redes son sensibles a los parámetros iniciales y a la arquitectura que se use. Obtener los parámetros y arquitecturas ideales conlleva tiempo y representó un reto en el desarrollo de este trabajo práctico.