Tarea1: Desarrollar y entrenar Feed Forward Neural Networks

Ignacio Haeussler

22 de octubre de 2018

1. Dataset

El dataset utilizado fue https://archive.ics.uci.edu/ml/datasets/Balance+Scale. Este consiste en datos generados artificialmente que representan distintas configuraciones en las que una balanza imaginaria puede disponerse. Las variables de entrada son 4, las que representan el peso de los dos objetos a comparar, uno a la izquierda y uno a la derecha, y las distancias de estos al centro de la balanza. Es decir, en vez de comparar el peso de los objetos, la balanza compararía los torques generados por estos. Cada una de estas cuatro variables toma valores enteros entre 1 y 5 (incluyendo ambos). Las 3 clases consisten por lo tanto del resultado de la comparación: la balanza se inclina a la izquierda, la derecha o a ninguna de las dos (queda balanceada debido a una igualdad de torques aplicados). El dataset consiste de 625 ejemplos y un 20 % es extraído aleatoriamente en todas las pruebas con fines de testeo (ver grado de generalización y overfitting).

El preprocesamiento aplicado consistió en transformar todos los ejemplos de entrada desde arreglos de 4 variables a unos de 10, en las que se tendría el primer peso dentro de las primeras 5 componentes y la segunda dentro de las segundas 5 componentes, con sus posiciones dadas por la distancia al centro reportada para cada una. Por lo tanto, las redes recibirían 10 features y tendrían 3 salidas.

2. Sistema operativo, lenguaje de programación y librerías utilizadas

El sistema operativo utilizado fue ubuntu 18.04, el lenguaje fue C++14 y para compilar fue creado una archivo Makefile, el cual una vez estando en el directorio raíz del proyecto, puede ser ejecutado utilizando el comando make. Adicionalmente, éste asume un ambiente linux para crear carpetas asociadas a la creación de los ejecutables.

Por otro lado, la única librería utilizada fue OpenCV 3.4 para realizar gráficos en C++. Para su instalación en Ubuntu 18.04 las instrucciones en el siguiente link fueron seguidas: https://www.pyimagesearch.com/2018/05/28/ubuntu-18-04-how-to-install-opencv/, las que además permiten en ubuntu 18.04 compilar código C++14.

3. Correr implementación

Una vez estando en la raíz del código y habiendo compilado satisfactoriamente, el siguiente comando ejecuta el programa de la tarea, el cual consiste en el despliegue de gráficos que entregan los resultados asociados a las distintas preguntas de la tarea:

```
./build/Network/task1/task1
```

Por otro lado, la implementación de la red neuronal y de sus tests unitarios están en los archivos src/Neuron.h, src/Neuron.cpp y src/Network/unitTests.cpp. El ejecutable asociado a los test unitarios estaría en build/Network/unitTests/unitTests.

4. Arquitectura utilizada

Las redes utilizadas poseen 10 entradas en su capa inicial y 3 salidas en su capa final, las capas escondidas poseen neuronas sigmoides y las finales softmax, un batch size de 10, 100 épocas, y un learning rate de 0.25 y una capa escondida con 3 neuronas (a menos que se indique lo contrario para estos dos últimos parámetros). El algoritmo de entrenamiento fue stocastic gradient descent y la función de pérdida cross entropy.

5. Variando número de capas

La siguiente imagen muestra los resultados de entrenar la red con distintas cantidades de capas:

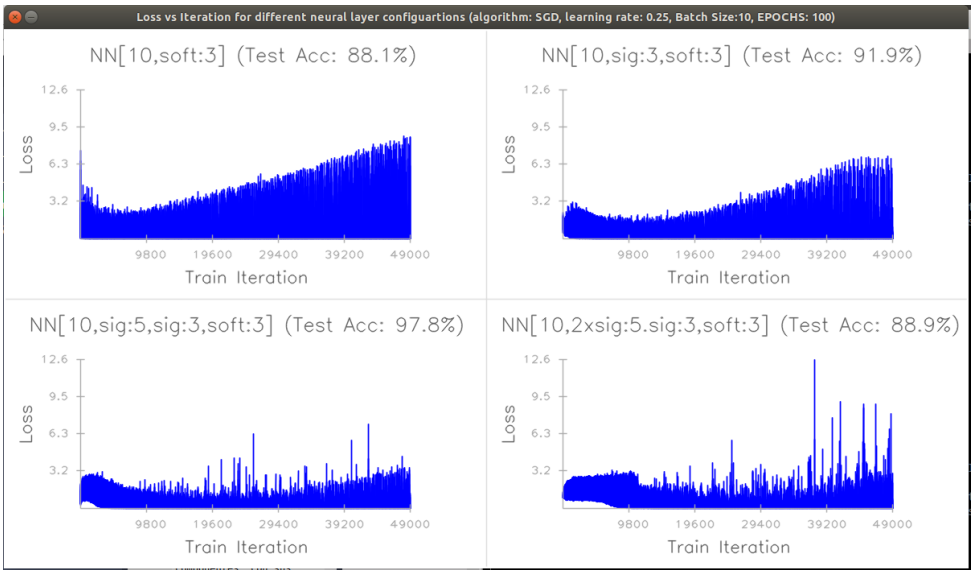

Figura 1: Loss vs Iteración para diferentes configuraciones de red

Se puede observar que un aumento inicial del número de capas conlleva un aumento de la accuracy en testing y una disminución en el error reportado. Sin embargo, parece ser que posteriores adiciones disminuyen los beneficios (posiblemente debido al vanishing gradient de las neuronas sigmoidales).

	Entrenamiento	Época	Ejemplo
NN[10,soft:3]	$2.1 \cdot 10^5$	$2.1 \cdot 10^3$	4.2
NN[10,sig:3,soft:3]	$3.4 \cdot 10^5$	$3.4 \cdot 10^3$	6.8
NN[10,sig:5,sig:3,soft:3]	$5.9 \cdot 10^5$	$5.9 \cdot 10^3$	11.8
NN[10,2xsig:5,sig:3,soft:3]	$8.1 \cdot 10^5$	$8.1 \cdot 10^3$	16.2

Tabla 1: Tiempos de entrenamiento para cada configuración en nanosegundos

Por otro lado, en la Tabla 1 se pueden observar los tiempos que tarda cada configuración en ser entrenada, en procesar todos los datos de ejemplo una vez (forward y backward) y lo que tardan en procesar un único ejemplo en promedio. Los cálculos son simples de calcular debido a que las épocas son 100 y los ejemplos de entrenamiento 500. Se puede apreciar que los entrenamientos tardan entre un 1/5 y 4/5 de un segundo, que el procesamiento de todos los ejemplos de entrenamiento está en el orden de los milisegundos, y que se destaca la cantidad de tiempo requerida para procesar un único ejemplo por estar en el orden de los nanosegundos.

6. Variando el learning rate

En la siguiente figura se pueden apreciar los resultados de variar el learning rate:

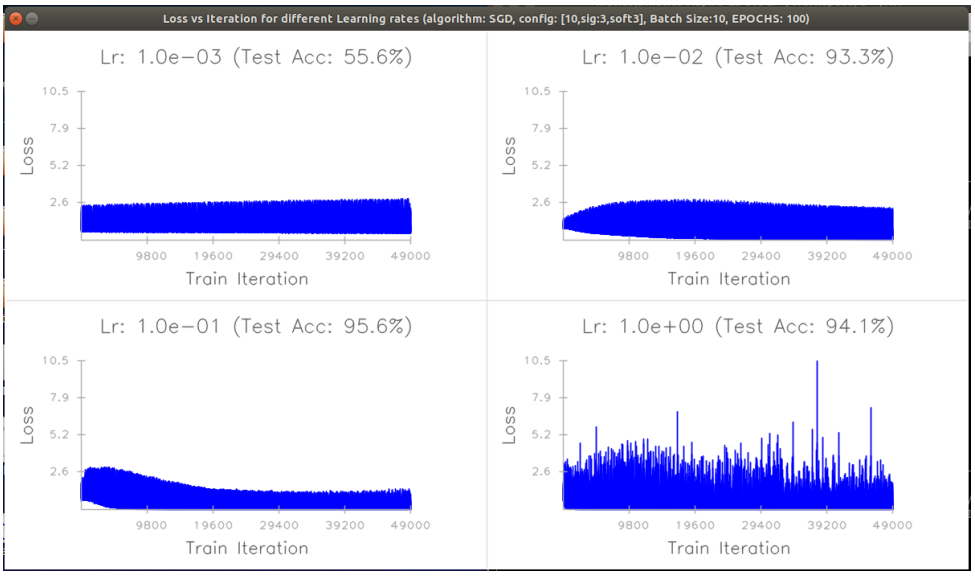

Figura 2: Loss vs Iteración para diferentes configuraciones de red

Se puede apreciar que un learning rate cercano 0.1 parece ser óptimo, mientras que los inferiores resultan en un bajo aprendizaje y superiores en una mayor inestabilidad.

7. Efecto de barajar

Finalmente, en la Figura 3 se pueden apreciar las diferencias de barajar o no los ejemplos:

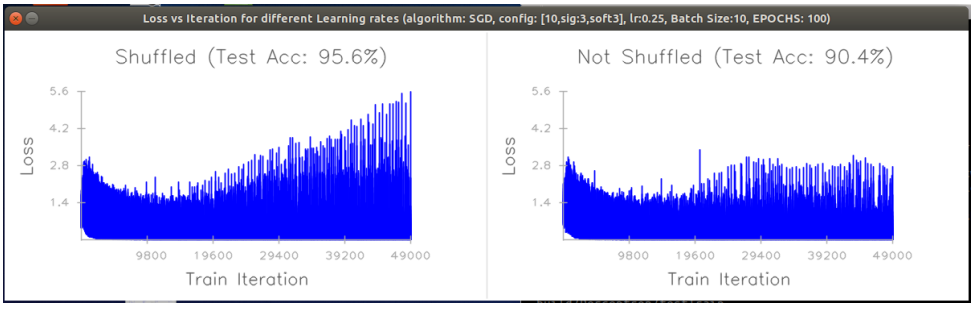

Figura 3: Loss vs Iteración para datos barajados y no barajados

Se aprecia una clara superioridad de la opción de barajar en la accuracy de testing de la opción barajada.