Reconocimiento de dígitos en imágenes con redes neuronales

Mauricio Neira[1], Juan Diego Chaves[2], Juan Camilo Pinilla[3] y Camilo Anzola[4]

***Abstract*— En este trabajo, se clasifican dígitos escritos en imagenes de 28x28 usando redes neuronales. Se implementan 3 métodos para entrenar las redes: algoritmos evolutivos, metrolpolis-hastings (simulated annealing) y stochastic gradient descent (SGD). Los tres algoritmos entrenan la red neuronal pero de los 3, el más eficiente es SGD que logra un porcentaje de acierto en la clasificación de 96%. Se cree que la convergencia lenta de los otros dos algoritmos se debe al gran tamaño del espacio de los parámetros de la red.**

I. INTRODUCTION

El trabajo desarrollado se basa en el libro "Neural Networks and Deep Learning" de Michael Nielsen. La motivación de esta entrega es crear una función que clasifique imágenes de números escritos tales como se muestran a continuación:[1]

Fig. 1. Digitos escritos, imágenes de 28x28.

Dada cualquiera de estas imágenes, se quiere conocer el dígito que representan. Por ejemplo, la siguiente imágen de 28x28 representa un 0:[1]

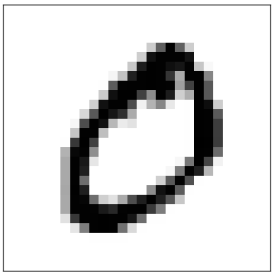

Fig. 2. Digitos escritos, imágenes de 28x28.

[1]201424001 m.neira10@uniandes.edu.co
[2]201533528 jd.chaves@uniandes.edu.co
[3]201533888 jc.pinilla@uniandes.edu.co
[4]201529838 ca.anzola@uniandes.edu.co

Cada pixel corresponde a algún color del rango de 0-255 donde 0 es negro y 255 blanco. Así,cada valor en el rango $[1, 264]$ representa una tonalidad de gris. La idea es tomar el valor de todos los píxeles de la imágen y operar sobre ellos para saber a qué dígito corresponde la imágen. Esto se discutirá a profundidad en la siguiente sección.

II. MODELO MATEMÁTICO

A. Neurona

Para entender cómo se modela una red neuronal, comencemos por modelar una neurona:[1]

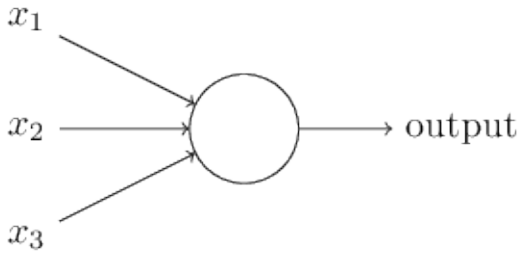

Fig. 3. Neurona de una red neuronal.

Cada neurona se puede pensar como una función que toma n valores $(x_1, x_2, ..., x_n)$ y los mapea a $\mathbb{R}$:

$$f(x_1, x_2, ..., x_n) = b + w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n \quad (1)$$

Esto no es más que una combinación lineal con un término de sesgo b agregado. Ahora bien, es útil tratar a las salidas de las neuronas en el rango $[0, 1]$. Por esto, se pasa el resultado de la funcion 1 por la función sigmoide $S : \mathbb{R} \mapsto [0, 1]$:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Así pues, el valor de salida de una neurona es:

$$\Phi(x_1, x_2, ..., x_n) = S(f(x_1, x_2, ..., x_n)) \quad (2)$$

B. Red neuronal

Una red neuronal no es más que un conjunto de nueronas conectadas entre sí[1]:

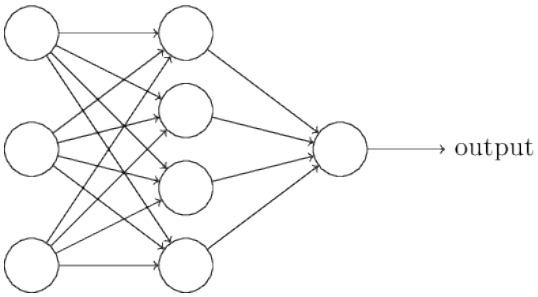

Fig. 4. Red neuronal simple.

Los nodos de la izquierda son las entradas de la red. Los nodos de la siguiente capa toman a los nodos de la izquierda como entradas y calculan su propio valor usando la ecuación 2. A su vez, el nodo de salida (output) toma los nodos de la capa de la mitad como entradas y calcula su valor con la misma ecuación.

C. Red neuronal implementada

Ahora bien, hemos hablado de redes con una sola neurona como salida. Para nuestro trabajo, necesitamos un total de 10 neuronas de salida; una para cada dígito posible. Esta extensión es muy natural pues los nodos adicionales sólo tendrían que usar la función 2 sobre la capa directamente anterior. La red implementada y optimizada en este trabajo es la siguiente:

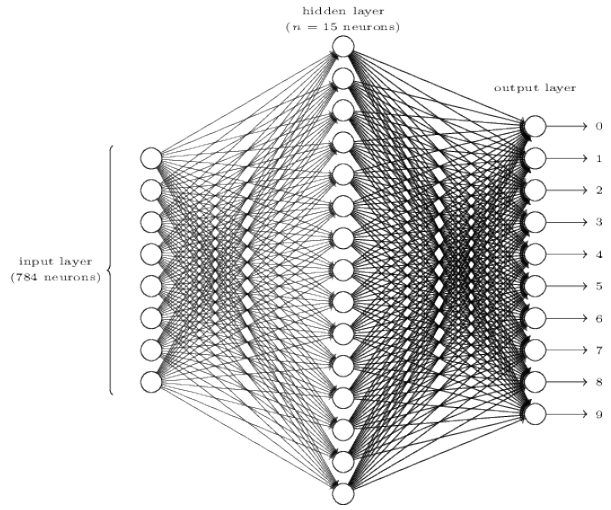

Fig. 5. Red neuronal imlementada para reconocer dígitos escritos.

Cada imágen tiene un total de $28 \times 28 = 784$ pixeles. Así pues, cada pixel representa una entrada de nuestra red. Como capa intermedia, se tiene un total de 15 neuronas y la capa de salida son 10 neuronas, 1 por cada dígito posible. Ahora bien, para obtener el resultado se va a tomar el valor máximo de la capa de salida. Por ejemplo, si el resultado de la red fue el vector:

$$\Phi(x) = \begin{bmatrix} 0.12 \\ 0.13 \\ 0.22 \\ 0.11 \\ 0.82 \\ 0.12 \\ 0.23 \\ 0.22 \\ 0.4 \\ 0.11 \end{bmatrix} \tag{3}$$

Se tomaría 4 como el dígito clasificado a partir de la imágen.

Teniendo el modelo matemático construido, ya se puede hablar del aprendizaje de la red. El objetivo de la red a implementar va a ser minimizar la cantidad de imágenes clasificadas erróneamente. Dada una imágen x del conjunto de imágenes de entrenamiento X, el error de esa clasificación está dado por $|\Phi(x) - a|^2$ donde $a \in \mathbb{R}^{10}$ vale 0 en todas las entradas salvo en la entrada que representa el valor del dígito de la imágen donde vale 1. Así pues iterando sobre todas las imágenes, se tiene que la función de error total es:

$$C(w, b) = \frac{1}{2n} \sum_{x \in X} |\Phi(x) - a|^2 \tag{4}$$

El objetivo de todos los siguientes algoritmos será minimizar ésta función.

III. Algoritmos

A. Algoritmo evolutivo

En main.py se ejecuta lo siguiente:

```
#se cargan los datos
training_data, validation_data, test_data =
    mnist_loader.load_data_wrapper()

#se inicializa la poblacion
networks = ga.genPopulation(10)

#se calcula el fitness inicial
ga.calcFitnessAll(networks, test_data)

#se realizan 100 generaciones
for i in range(100):
    iterate(networks)
```

Iterate(networks) se encarga de avanzar las generaciones por 1. Por el momento, dadas las restricciones computacionales, se decidió avanzar 100 generaciones.

Miremos en detalle iterate(networks):

```
def iterate(networks):

    chosen = []
```

```
# escogemos a los 10 mejores
for i in range(10):
    chosen.append(i)

temps = []
for i in range(len(chosen)/2):

    # se crean hijos segun los mejores
    #individuos de la poblacion

    son1, son2 = ga.crossover(
        networks[random.choice(chosen)],
        networks[random.choice(chosen)])
    #y se mutan
    for i in range(5):
        ga.mutate(son1)
        ga.mutate(son2)

    temps.append(son1)
    temps.append(son2)

#se eliminan todos los individuos
#salvo los 10 mejores
while(len(networks) > 10):
    networks.pop(-1)

#se agregan los hijos
for i in temps:
    networks.append(i)

#se calcula el fitness de
#todos y se ordena la poblacion
#por fitness descendientemente
ga.calcFitnessAll(networks, test_data)

#imprimimos a consola
prints = []
for net in networks:
    prints.append(net.fitness)
print(prints)

#exportamos a un archivo al mejor
#de esta generacion,
#el promedio de la generacion y
#el peor de la generacion
prints = np.array(prints)
with open("metaResults.txt", "a") as myfile:
    myfile.write(
        str(np.average(prints)) + " " +
        str(np.max(prints)) + " " +
        str(np.min(prints)) + "\n")
```

Las funciones crossover y mutate se encargan de mezclar el código genético y mutarlo. Al final se decide exportar los resultados de la poblacion para graficarlos después. Éste algorimto y sus resultados se presentaron en la entrega anterior.

B. Simulated annealing

La idea general del algoritmo es inicializar una red neuronal con pesos y biases aleatorios. De ahí, se salta de forma aleatoria a otro punto en el espacio y se evalua su optimalidad relativa al punto donde se encontraba previamente. Entre mejor sea este nuevo punto, mayor es la probabilidad de que se mueva allí. La temperatura aquí es un parámetro auxiliar para evitar que el algoritmo se quede atorado en un mínimo local.

El corazón del algoritmo se encuentra a continuación:

```
# se crea una red neuronal principal,
# una auxiliar para evaluar el nuevo punto
# y una red para guardar la mejor
#red hasta el momento
net = network.Network(sizes)
net2 = network.Network(sizes)
best = network.Network(sizes)

# se igualan todas las redes al comienzo
net2.cambiarPesos(net)
best.cambiarPesos(net)

arch = open("progreso.dat","a+")

# se inicializa la temperatura
T = 0.10
while(T>0):
   #se muta la red
   mutar(net.weights, net.biases)
   # se calcula la probabilidad de saltar
   talpha = alpha(net2,net,T)
   # se determina si se salta
   if(talpha>np.random.rand()):
       print("entra")
       # se cambian las redes relevantes si se salta
       net2.cambiarPesos(net)
       if(best.evaluateNormed(test_data) <
          net.evaluateNormed(test_data)):
           best.cambiarPesos(net2)

   # se registran los cambios
   arch.write(str(T) + " " +
        str(best.evaluateNormed(test_data)) + "\n")

   T= T-0.0001
   print "Net2: " +
         str(net2.evaluateNormed(test_data))+
         " Net: " +
         str(net.evaluateNormed(test_data)) +
         " Best: " +
         str(best.evaluateNormed(test_data)) +
         " T: " + str(T) + " alpha: " + str(talpha)

arch.close()
```

C. Stochastic gradient descent

Fundamentalmente, este algoritmo calcula el gradiente de la función de error y se mueve en la dirección de menor crecimiento. Es decir, se mueve en la dirección que minimiza la función de error. El problema principal con este algoritmo es que puede quedarse atorado en mínimos locales. Sin embargo, es considerablemente más rápido que los otros

algoritmos. El corazón de el algoritmo es el siguiente [1]

```
def procesarLote(net, lote, eta):
    #gradientes de pesos y biases
    gradB = []
    gradW = []
    #los llenamos con ceros
    for i in range(len(net.biases)):
        gradB.append(np.zeros(net.biases[i].shape))
        gradW.append(np.zeros(net.weights[i].shape))
    #se itera sobre cada imagen del lote
    for im, dig in lote:
        dgradB, dgradW = retropropag(net, im, dig)
        gradB = [nb+dnb for nb,
                 dnb in zip(gradB, dgradB)]
        gradW = [nw+dnw for nw,
                 dnw in zip(gradW, dgradW)]
    #se actualizan los pesos y los biases
    net.weights = [w-(eta/len(lote))*dw
                   for w, dw in zip(net.weights,
                                    gradW)]
    net.biases = [b-(eta/len(lote))*db
                  for b, db in zip(net.biases,
                                   gradB)]
```

```
#creamos una red
net = network.Network(sizes)
#escribimos resultados a un archivo
arch = open("./dataSGD/data.dat","a+")
print str(0) + " " +
      str(net.evaluateNormed(test_data))
arch.write(str(0) + " " +
           str(net.evaluateNormed(test_data)) +
           "\n")

#iteramos iteraciones veces sobre
#los datos de entrenamiento
for i in range(1, iteraciones):
    #reorganizamos los datos para un
    #aprendizaje mas eficiente
    shuffle(training_data)
    lotes = []
    #particionamos la información en lotes
    for j in range(0, len(training_data), tam):
        lotes.append(training_data[j:j+tam])
    #procesamos cada lote
    for lote in lotes:
        procesarLote(net, lote, 3.0)
    #se evalua la eficiencia de la iteracioin
    print str(i) + " " +
          str(net.evaluateNormed(test_data))
    arch.write(str(i) + " " +
               str(net.evaluateNormed(test_data)) +
               "\n")
arch.close()
```

[1]Este algoritmo fué desarrollado de la mano de el libro [1], basandose en gran parte del código fuente provisto por el autor en: https://github.com/mnielsen/neural-networks-and-deep-learning.

IV. Resultados

Los resultados de los tres algoritmos se encuentran a continuación. En el eje y se encuentra el porcentaje de acierto de la mejor red y en el eje x se encuentran el número de iteraciones.

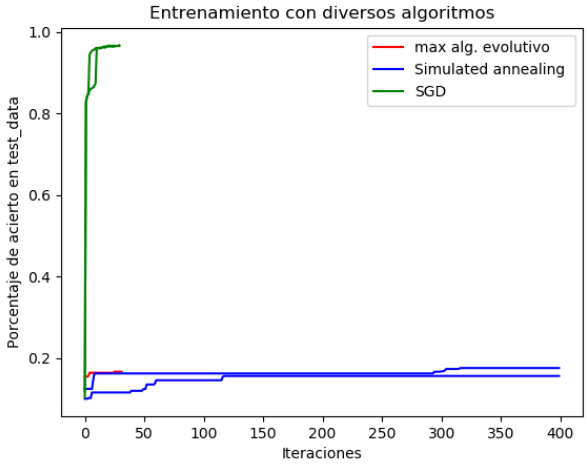

Fig. 6. Resultados de los 3 algoritmos.

Hay 2 items importantes que resaltar con base en los anteriores resultados:

- Todos los algoritmos mejoran el desempeño de la red neuronal, es decir, todas hacen que la red neuronal aprenda.
- SVG aprende a una rata considerablemente mayor a sus contrapartes estocásticas.

Si el lector se detiene a detallar la figura IV, notará que todas las redes empiezan con un desempeño de aproximadamente 0.1. Esto tiene sentido pues hay un total de 10 posibles resultados así que a priori, la red va a adivinar correctamente 10% de las veces.

Ahora bien, el algoritmo evolutivo y simulated annealing logran un porcentaje de acierto de aproximadamente 18%, ciertamente mayor a el valor inicial pero considerablemente menor a el valor de SVG (96%). La falta de eficiencia puede ser causada por el tamaño del espacio de búsqueda pues en total, contando pesos y sesgos, el algoritmo tiene que optimizar un total de $|w| + |b| = (15 \times 784 + 10 \times 15) + (15 + 10) = 11935$ parámetros. Es decir, tiene que encontrar el máximo global en $\mathbb{R}_{+}^{11935}$. Es posible que la magnitud de este espacio junto con la naturaleza estocástica hagan ineficientes a éstos dos algoritmos.

En contraste, SVG toma el gradiente local de la función y camina en la dirección de menor crecimiento. Es muy posible que este algorimto quede atascado en un mínimo

local (a diferencia de los dos algoritmos anteriores) pero definitivamente aumenta su eficiencia bajo este costo.

V. Conclusiones

Se clasifican dígitos escritos en imagenes de 28x28 usando redes neuronales implementando 3 métodos para entrenar las redes: algoritmos evolutivos, metrolpolis-hastings (simulated annealing) y stochastic gradient descent (SGD). Los tres algoritmos lograron entrenar la red neuronal pero de los 3, el más eficiente es SGD que logra un porcentaje de acierto en la clasificación de 96%. Se cree que la convergencia lenta de los otros dos algoritmos se debe al gran tamaño del espacio de los parámetros de la red.

Reconocimientos

Todo el trabajo anterior no hubiera sido posible sin la ayuda del libro de Michael Nielsen. Su libro, “Neural Networks and Deep Learning”[1] es áltamente recomendado para cualquiera que quiera entender en profundidad el funcionamiento de redes neuronales.

References

[1] M. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com/