Universidad de los Andes
Colombia | Facultad de Ingeniería

Universidad de los Andes
Maestría en Inteligencia Analítica para la Toma de Decisiones.
Modelos Avanzados para el Análisis De Datos 2.
Prof. Martín Andrade Restrepo.

Taller 2. Construcción Explícita de Redes Neuronales

Para los siguientes tres ejercicios:

Deberá realizarlos sin la implementación de código en Python.

Observe el video de las demostraciones de las primeras dos ecuaciones fundamentales del Backpropagation. Para observar el video, haga clic aquí.

Ejercicio I

(10 pts) Verifique de dónde vienen las ecuaciones fundamentales 3 y 4. Intente demostrarlo.

Pista:

Escriba el input ponderado en términos de los pesos y los sesgos y encuentre las derivadas buscadas utilizando dichas ecuaciones.

Ejercicio II

(10 pts) Suponga que tiene una red con 4 capas (una capa de input, dos capas ocultas y una capa de salida) y con una sola neurona en cada capa. Trate de construir el gradiente con las derivadas de la función de costo con respecto a los pesos y los sesgos "manualmente" y verifique que muchos de los términos que aparecen en las ecuaciones coinciden.

Para el siguiente ejercicio:

Deberá realizarlos con la implementación de código en Python.

Ejercicio III

(80 pts) Ejercicios a realizar con python. Evite usar Pytorch, Tensorflow o Keras.

Objetivos generales:

- Programaremos (completaremos algunas funciones de) una red neuronal genérica de forma explícita usando lo visto en clase.

- Realizaremos el ejercicio en grupos de 2-3 personas.

- Entrenaremos (y evaluaremos) una versión de dicha red para identificar los dígitos manuscritos cuyos datos están en la base de datos de MNIST.

- Usaremos programación orientada por objetos para darle la mayor versatilidad posible a nuestro programa (podremos utilizar este código para otras redes con distintos objetivos).

- Cada grupo evaluará distintas versiones de dicha red para encontrar aquella que ofrezca mejores resultados.

Universidad de los Andes Colombia | Facultad de Ingeniería

Instrucciones:

1. Descargue el archivo de ***Bloque Neón*** con el esqueleto del código de la clase "network". Lea cuidadosamente las instrucciones en el archivo y revise las funciones. Describa.

2. Complete dicho código con las funciones que estén vacías (aquellas que tengan el instructivo "*TODO*").

3. Cree un código adicional que cargue los datos de la base de datos *MNIST* (puede usar el código sugerido del Taller 1 asegurándose que los datos estén en el formato correcto). Solo acá puede hacer uso de Tensorflow o Keras.

4. Cree un código "`main`" que cargue los datos (utilizando la función del numeral anterior), y que cree, entrene y evalúe la red. Para esto debera crear una instancia (un objeto) de la clase network. Este objeto será su red. Luego puede llamar las funciones de dicho objeto para entrenarla y evaluarla.

   A su código main añadale las líneas: "`if __name__ == "__main__":`" (esto se acostumbra para identificar el código principal de un programa).

5. Si tiene tiempo, cree (puede buscar en Internet) una función que grafique los dígitos a partir de los datos (opcional).

6. Utilice su red para identificar datos nuevos sin incluir la etiqueta (puede extraerlos aleatoriamente de las bases de entrenamiento y prueba disponibles). Verifique usando su función de visualización que la red haga el trabajo correcto.

7. Cree distintas versiones de su red con distintas arquitecturas y describa sus resultados más notables (en términos de desempeño).

8. Evite utilizar librerías especializadas en ML (keras, pytorch, tensorflow, etc.). Buscamos una comprensión profunda de los métodos, no únicamente el saber utilizarlos.

9. Trate de no buscar ni en internet ni en el libro de Nielsen la solución (obviamente la encontrarán, es un ejercicio clásico).

```
"""
network.py
~~~~~~~~~~

A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation.  Note that I have focused on making the code
simple, easily readable, and easily modifiable.  It is not optimized,
and omits many desirable features.

LEER!!!:
Ejercicio basado en on Neural Networks and Deep Learning. M. Nielsen. (2018)
Por favor no mirar las soluciones en la fuente! Trate de hacerlas usted mismo.

```

```
Revise primero la estructura de los vectores y matrices generados.
Para cada funcion a realizar lea muy cuidadosamente la forma y el formato en el
    que se esperan los parametros.

Puede usar cualquier libreria de python que desee

"""

#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a): # TODO
        """Return the output of the network if ``a`` is input."""


    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print (f"Epoch {j}: {self.evaluate(test_data)} / {n_test}")
            else:
                print ("Epoch {j} complete")

```

```
76      def update_mini_batch(self, mini_batch, eta):   # TODO
77          """Update the network's weights and biases by applying
78          gradient descent using backpropagation to a single mini batch.
79          The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
80          is the learning rate."""
81
82
83      def backprop(self, x, y):
84          """Return a tuple ``(nabla_b, nabla_w)`` representing the
85          gradient for the cost function C_x.  ``nabla_b`` and
86          ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
87          to ``self.biases`` and ``self.weights``.
88
89          Observe bien que ``nabla_b`` y ``nabla_w`` se generan para cada elemento
                del minibatch. Esta funcion debe llamarse una vez por cada elemento
                del
90          minilote.
91
92          """
93          nabla_b = [np.zeros(b.shape) for b in self.biases]
94          nabla_w = [np.zeros(w.shape) for w in self.weights]
95          # feedforward
96          activation = x
97          activations = [x] # list to store all the activations, layer by layer
98          zs = [] # list to store all the z vectors, layer by layer
99          for b, w in zip(self.biases, self.weights):
100             z = np.dot(w, activation)+b
101             zs.append(z)
102             activation = sigmoid(z)
103             activations.append(activation)
104         # backward pass
105         delta = self.cost_derivative(activations[-1], y) * \
106             sigmoid_prime(zs[-1])
107         nabla_b[-1] = delta
108         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
109         # Note that the variable l in the loop below is used a little
110         # differently to the notation in Chapter 2 of the book.  Here,
111         # l = 1 means the last layer of neurons, l = 2 is the
112         # second-last layer, and so on.  It's a renumbering of the
113         # scheme in the book, used here to take advantage of the fact
114         # that Python can use negative indices in lists.
115         for l in range(2, self.num_layers):
116             z = zs[-l]
117             sp = sigmoid_prime(z)
118             delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
119             nabla_b[-l] = delta
120             nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
121         return (nabla_b, nabla_w)
122
123     def evaluate(self, test_data):   # TODO
124         """Return the number of test inputs for which the neural
125         network outputs the correct result. Note that the neural
126         network's output is assumed to be the index of whichever
127         neuron in the final layer has the highest activation."""
128
129
130     def cost_derivative(self, output_activations, y):
131         """Return the vector of partial derivatives \partial C_x /
132         \partial a for the output activations."""
133         return (output_activations-y)
134
135 #### Miscellaneous functions
```

```
136 def sigmoid(z):  # TODO
137     """The sigmoid function."""
138
139
140 def sigmoid_prime(z):  # TODO
141     """Derivative of the sigmoid function."""
```