Optimizaación de red

December 3, 2023

[1]:
```
# %load mnist_loader.py

#### Libraries
# Standard library
import pickle
##Esta libreria nos permite serializar objetos es decir convertir un objeto en␣
 ↪base a
#un lenguaje específico y deserializar que es el proceso análogo de string a␣
 ↪objeto)
import gzip ##para comprimir y descomprimir archivo.zip como el mnist.pkl.gz

# Third-party libraries
import numpy as np## nos permite realizar los calculos con arrays

def load_data(): ##vamos a definir una función

    f = gzip.open('mnist.pkl.gz', 'rb') #abre el archivo.zip, el modo rb,␣
 ↪predeterminado como lectura de datos binarios)
    training_data, validation_data, test_data = pickle.load(f,␣
 ↪encoding="latin1")#definimos las variables a las que con
    #.load vamos a deserealizar y a dodificar en ascii global
    f.close()# cierra el mnist.pk
    return (training_data, validation_data, test_data)
#regresa las variables en datos de entrenamiento, datos de validación, y de␣
 ↪prueba

def load_data_wrapper():## pre procesamiento de los datos

    tr_d, va_d, te_d = load_data()##a estas variables les aplicamos la función
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]] ##reformamos␣
 ↪una matriz de 784 x1 para la lista tr_d
    #comenzando por el elemento 0
    training_results = [vectorized_result(y) for y in tr_d[1]]##vamos a definir␣
 ↪antes la función
    training_data = zip(training_inputs, training_results)##como un ziper de␣
 ↪una chamrra retorna un nuevo iterable
```

```
    #cuyos elementos son tuplas que contienen un elemento de training data con␣
 ↪otro de training results
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])#proceso análogo al zip␣
 ↪para training data
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)#regresa los nuervos␣
 ↪arrayas que formamos en la función.

def vectorized_result(j):##esta función toma un entero
    e = np.zeros((10, 1))#matriz de ceros de 10x1
    e[j] = 1.0
    return e
```

[22]:
```
# %load network.py



#### Libraries
# Standard library
import random # está libreria nos permite obtener datos aleatorios para␣
 ↪alimentar


# Third-party libraries
import numpy as np

class Network(object):##definimos la clase que será la neurona
    def __init__(self, sizes):##es la función con los parametros no de neuronas␣
 ↪por capas y pesos


        self.num_layers = len(sizes)#tamaño de capas
        self.sizes = sizes
        self.biases =[np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)#esta funcion crea los pesos␣
 ↪aleatoriamente
                        for x, y in zip(sizes[:-1], sizes[1:])]

def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)##producto punto y le sumamos la bias
        return a

def SGD(self, training_data, epochs, mini_batch_size,momentum,eta,##obtiene los␣
 ↪pesos para el backpropagation y evita que se atore
```

```
            test_data=None):
    training_data = list(training_data)##lista de datos para entrenar
    n = len(training_data)#nos da el tamaño de training data
    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)
        for j in range(epochs):
            random.shuffle(training_data)##reorganizamos aleatoriamente los␣
↪datos
            mini_batches = [
                training_data[k:k+mini_batch_size]#reparte los datos entre el␣
↪númeroi de minibatches
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)##el.update le actualiza␣
↪al minibatch
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.
↪evaluate(test_data),n_test))#este ciclo nos permite visualizar
                #datos o epocas
            else:
                print("Epoch {} complete".format(j))

    def update_mini_batch(self, mini_batch, eta):#la función que actualiza el␣
↪minibatch

        nabla_b = [np.zeros(b.shape) for b in self.biases]#llena de ceros
        #las listas que definimos se van llenando de la suma del grad por␣
↪minibatch
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)#funcion de coste
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]#A los nuevos␣
↪datos que obtuvimos del
        #grad del minibatch  se les resta el valor del aprendizaje
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):

        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
```

```
        activation = x
        activations = [x] # lista de todas las activaciones de capa en capa
        zs = [] # definimos una lista a la que le vamos a meter los pesos por␣
 ↪capa
        for b, w in zip(self.biases, self.weights):#ciclo para  ir juntando␣
 ↪pesos y bias por capa
            z = np.dot(w, activation)+b
            zs.append(z)#añade
            activation = sigmoid(z)#aplica la función sigmoide a mi producto␣
 ↪punto antes definido
            activations.append(activation)#añade a activations cada valor␣
 ↪previamente calculado
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])#calcula el error de atrás hacía adelante es␣
 ↪decir
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in range(2, self.num_layers):#errores para el gradiente
            z = zs[-l]
            sp = sigmoid_prime(z)#derivada
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):

        test_results = [(np.argmax(self.feedforward(x)), y)#función para␣
 ↪argumento mayor de la función feedforward para calcular
                        #la salida de la red
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)#regersa la suma

    def cost_derivative(self, output_activations, y):##funcion de costo en␣
 ↪relacion con las salidas

        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))##función sigmoide

def sigmoid_prime(z):##derivada de la función sigmoide
    return sigmoid(z)*(1-sigmoid(z))
```

[ ]:
```
import numpy as np
class AdamOptim():
    def __init__(self, sizes):##es la función con los parametros no de neuronas␣
 ↪por capas y pesos
        self.num_layers = len(sizes)#tamaño de capas
        self.sizes = sizes
        self.biases =[np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)#esta funcion crea los pesos␣
 ↪aleatoriamente
                        for x, y in zip(sizes[:-1], sizes[1:])]

def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)##producto punto y le sumamos la bias
        return a

    def update(self, t, w, b, dw, db):
        ## dw, db are from current minibatch
        ## momentum beta 1
        # *** weights *** #
        self.m_dw = self.beta1*self.m_dw + (1-self.beta1)*dw
        # *** biases *** #
        self.m_db = self.beta1*self.m_db + (1-self.beta1)*db

        ## rms beta 2
        # *** weights *** #
        self.v_dw = self.beta2*self.v_dw + (1-self.beta2)*(dw**2)
        # *** biases *** #
        self.v_db = self.beta2*self.v_db + (1-self.beta2)*(db)

        ## bias correction
        m_dw_corr = self.m_dw/(1-self.beta1**t)
        m_db_corr = self.m_db/(1-self.beta1**t)
        v_dw_corr = self.v_dw/(1-self.beta2**t)
        v_db_corr = self.v_db/(1-self.beta2**t)

        ## update weights and biases
        w = w - self.eta*(m_dw_corr/(np.sqrt(v_dw_corr)+self.epsilon))
        b = b - self.eta*(m_db_corr/(np.sqrt(v_db_corr)+self.epsilon))
        return w, b
```

[18]:
```
def SGDmom(self, training_data, epochs, mini_batch_size, eta, momentum,
           test_data=None):
        training_data = list(training_data)
```

[23]:
```
import mnist_loader ## importar código/bloque de los datos
import Network #importar código de la red
```

```
import network_SGDmom
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
training_data = list(training_data)
test_data=list(test_data)


net = Network.Network([784, 30, 10]) ##parametros para entrenar
net.SGDmom(training_data, 10, 2, 3.0, .82, test_data=test_data)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[23], line 2
      1 import mnist_loader ## importar código/bloque de los datos
----> 2 import Network #importar código de la red
      3 import network_SGDmom
      4 training_data, validation_data, test_data = mnist_loader.
 ↪load_data_wrapper()

ModuleNotFoundError: No module named 'Network'
```

[13]:
```
def SGD_momentum(self, training_data, epochs, mini_batch_size, eta, momentum):
```

```
  Cell In[13], line 1
    def SGD_momentum(self, training_data, epochs, mini_batch_size, eta, momentu )

 ↪^
SyntaxError: invalid syntax
```

[28]:
```
def softmax(x):## probabilidad de pertenencia a n clase
    e_x = np.exp(x - np.max(x, axis=1))#
    return e_x / e_x.sum(axis=1)

def crossentropy(y_true, y_pred):#predicción del modelo y como est dada
    m = y_true.shape[0]
    logx = -np.log(y_pred[range(m), np.argmax(y_true, axis=1)])
    loss = np.sum(logx) / m ##que tan grande es el error

    return loss#

# Calcula la pérdida de entropía cruzada con Softmax
loss = categorical_crossentropy(y_true, y_pred)
print(f"Pérdida de entropía cruzada: {loss}")
```

[ ]:
```
#combinando momentum con rns

class Adam:
    def adamio(self, learning_rate=0.0001, beta1=0.9, beta2=0.999,␣
 ↪epsilon=1e-7):#función adamio variables para sgd y rmsprop
        self.learning_rate = learning_rate##self porque es clase
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.t = 0
        self.m = None
        self.v = None

    def minimizar(self, gradientes, parametros):
        if self.m is None:
            self.m = np.zeros_like(parametros)
            self.v = np.zeros_like(parametros)

        self.t += 1 #sumar
        self.m = self.beta1 * self.m + (1 - self.beta1) * gradients#por␣
 ↪definición del algoritmo
        self.v = self.beta2 * self.v + (1 - self.beta2) * (gradients ** 2)#vars␣
 ↪momentum

        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)#vars rms prop

        update = self.learning_rate * m_hat / (np.sqrt(v_hat) + self.epsilon)
        parameters -= update


    for i in range(num_iterations):
        # Calcula el gradiente
        gradient = 2 * initial_parameters
```