Dokumentacja Projektu
Języki Skryptowe
Klasyfikator ręcznego pisma

Chłąd Paweł

2019
Grudzień

1 Opis działania

1.1 Wstęp

Główny program składa się z serwera HTTP, którego zadaniem jest obsługa użytkownika. Drugą częścią jest strona internetowa która stanowi interfejs pomiędzy serwerem a użytkownikiem. Trzecią częścią jest program uczący, za jego pomocą została wytrenowana sieć neuralna, która potrafi rozpoznawać napisane przez użytkownika liczby. Sprawność sieci wynosi 95% ale ze względu na brak normalizacji danych wejściowych, procent ten będzie niższy dla prawdziwych (nieznormalizowanych) danych.

1.2 Instrukcja Obsługi

Program uruchamiamy przez Launch.bat albo server.exe. Launch.bat przedstawi nam dodatkowe opcje:

- Start server - uruchomi server.exe
- Backup - utworzy kopię sieci neuralnej i zebranych zdjęć
- Exit - wyjdzie z pliku wsadowego

2 Strona techniczna

2.1 Struktura

Serwer HTTP oraz program uczący zostały zrealizowane w języku Python (ver. 3.7.5). Strona internetowa w języku markupowym HTML oraz skrypty w języku JavaScript.

Wszelkie pliki źródłowe znajdują się pod adresem: https://github.com/Madoxen/HandwritingMLService
Lista użytych bibliotek:

- numpy
- PIL (Python Imaging Library)

Program ma następującą strukturę

- bin - folder zawierający pliki wykonywalne
- src - folder zawierający repozytorium git
- data - folder zawierający dane do nauki
- backup - folder zawierający kopię zapasową danych do nauki i repozytorium git

2.2 Opis kodu

2.2.1 server.py

Moduł python zawierający klasę Server. Klasa ta odpowiada za uruchamianie i wstrzymywanie faktycznego serwera HTTP (http.server) oraz jego handler'a (http_handler).

Klasa Server zawiera następujące właściwości:

- **addr** - tuple zawierający adres i port serwera
- **server** - instancja serwera http
- **server_thread** - instancja wątku serwera
- **restarting** - flaga oznaczająca restart serwera

Klasa Server zawiera następujące metody:

- **__init__(addr)** - konstruktor klasy Server; Addr - krotka (tuple) złożona z adresu IP oraz portu, na którym ma nasłuchiwać serwer.
- **run()** - metoda która uruchamia wątek serwera i oczekuje wprowadzenia potencjalnej komendy od użytkownika na wątku głównym
- **server_loop()** - metoda która jest używana podczas konstrukcji wątku serwera

2.2.2 http_handler.py

Moduł python zawierający klasę webServerHandler, której bazą jest klasa http.server.BaseHTTPRequestHandler. webServerHandler odpowiada za obsługę zapytań GET i POST.

Klasa webServerHandler zawiera następujące właściwości:

- **root** - ścieżka do folderu zawierającego serwowaną stronę internetową

Klasa webServerHandler zawiera następujące metody:

- **do_GET()** - metoda wywoływana podczas zapytania GET - serwuje stronę i jej skrypty znajdujące się w root
- **do_POST()** - metoda wywoływana podczas zapytania POST - obsługuje wymianę danych dot. wpisanego numeru. Użytkownik klikając przycisk "wyślij" wysyła dane o stworzonym przez siebie obrazie, w pliku JSON. Następnie plik ten jest czytany przez serwer, który następnie dokonuje klasyfikacji przy użyciu już wytrenowanej sieci neuralnej.

2.2.3 nnetwork.py

Moduł python zawierajacy klasę Network. Klasa ta zawiera w sobie dane o sieci neuralnej oraz algorytm uczenia.

Sieć neuralna składa się z czterech warstw:

- 784 neurony wejściowe (obraz 28x28 pix, wartości od 0 - 1 gdzie 1 to zarysowany obszar)
- Dwie warstwy ukryte po 30 neuronów
- 10 neuronów wyjściowych (oznaczające pewność klasyfikacji dla cyfr od 0 do 9)

Do nauczenia sieci został wykorzystany algorytm *gradientu prostego (gradient descend)* połączony z algorytmem *propagacji wstecznej (backpropagation)*.

Gradient prosty Metoda gradientu prostego opiera się na prostej obserwacji, nasza sieć neuralna to tak naprawdę pewna skomplikowana funkcja. Problem polega na tym, że nie możemy zminimalizować sieci bezpośrednio (co by to w ogóle znaczyło?), natomiast możemy zminimalizować pewną inną powiązaną funkcję. Taką funkcję nazywa się funkcją kosztu (eng. cost/loss function). Funkcją kosztu może być dowolna funkcja, która będzie wskazywała na pewien "odchył" od prawidłowego stanu.
W przypadku programu funkcją kosztu jest funkcja kosztu kwadratowego:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \tag{1}$$

Gdzie:

- w - wagi
- b - biasy
- $y(x)$ - prawidłowe wartości na neuronach wyjściowych (wektor 10 wymiarowy)
- a - rzeczywiste wartości powstałe na neuronach wyjściowych (wektor 10 wymiarowy)

Musimy więc znaleźć minimum funkcji kosztu, tradycyjna metoda niestety tutaj nie zadziała, gdyż funkcja ta w przypadku tego programu będzie miała tysiące zmiennych, dlatego też użyjemy metody przybliżonej. Wyliczając gradient takiej funkcji możemy łatwo określić *kierunek* w którym musimy się poruszać, aby dotrzeć do jej minimum.

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \cdots \frac{\partial C}{\partial v_n} \right)^T. \tag{2}$$

Pozwala nam to na określenie wektora zmian

$$\Delta v = -\eta \nabla C, \tag{3}$$

gdzie η to parametr uczenia (tzn. jak daleko przemieścimy się w następnym kroku). To wszystko przekłada się na generalną "zasadę" poruszania się w kierunku minimum.

$$v \to v' = v - \eta \nabla C. \tag{4}$$

Więc w naszym przypadku przekładamy to na odpowiednie wagi i biasy

$$w_k \quad \to \quad w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \tag{5}$$

$$b_l \quad \to \quad b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \tag{6}$$

Klasa Network zawiera następujące właściwości:

- **num_layers** - liczba warstw neuronów
- **sizes** - tuple zawierający liczbę neuronów w każdej warstwie
- **biases** - lista, zawierająca tablice (numpy) biasów odpowiednich warstw (pomijając warstwę wejściową)
- **weights** - lista, zawierająca tablice (numpy) wag odpowiednich warstw, np. dla warstwy (10) i warstwy (2) tablica ta będzie miała wymiar 10x2 gdyż do każdego neuronu musimy powiązać każdy neuron z następnej warstwy

Klasa Network zawiera następujące metody:

- **__init__(sizes=None, path=None)** - konstruktor klasy Network; dołączenie *sizes* spowoduje utworzenie tablic o zadanych rozmiarach, z losowymi elementami (losowymi w rozkładzie gaussa pomiędzy 0 a 1); dołączenie argumentu path spowoduje próbę odczytania sieci z zadanego pliku. (Formatem sieci jest prosty plik JSON)
- **feedforward(a)** - oblicza wektor wyjściowy na podstawie zadanego wektora wejściowego
- **SGD(training_data, epochs, mini_batch_size, eta, test_data=None)** - wykonuje algorytm prostego gradientu; training_data - lista tupli, w których przechowywane są parami wektory wejściowe i poprawne wektory wyjściowe; epochs - ilość powtórzeń; mini_batch_size - rozmiar małych paczek z danymi wejściowymi (ilość w tuplach na paczkę), ustawienie tego parametru na 1 spowoduje wykorzystanie zwykłego algorytmu gradientu prostego, ustawienie parametru na wielkość większą od jeden spowoduje użycie przybliżonego algorytmu gradientu prostego; eta - szybkość uczenia; test_data - tak samo jak training_data, jeśli zostanie podane spowoduje to, że w każdym epochu/powtórzeniu sieć zostanie przetestowana, a postęp nauki wyświetlany co powtórzenie.
- **backprop(x,y)** - funkcja obliczająca $\frac{\partial C}{\partial w_k}$ oraz $\frac{\partial C}{\partial b_l}$ stosując algorytm propagacji wstecznej; x - wektor wejściowy; y - spodziewany wektor wyjścia.
- **evaluate(test_data)** - funkcja sprawdzająca liczbę poprawnych ewaluacji sieci neuralnej.
- **cost_derivative(output_activations, y)** - oblicza wartość pochodnej dla funkcji kosztu.

2.2.4 ml_service.py

Moduł python zawierający klasę MLService. Klasa ta jest odpowiedzialna za przygotowywanie informacji wejściowych.

Klasa MLService zawiera w sobie jedną właściwość statyczną:

- **net** - instancja klasy Network

Klasa MLService zawiera w sobie następujące metody:

- **prepareImage(img_data)** - przygotowuje obraz w gotowy do użycia wektor 784 wymiarowy; img_data - dane obrazu w formie JSON
- **evaluate(img_data)** - zwraca wektor wyjściowy; img_data - wektor 784 wymiarowy z danymi obrazu.

```
from http.server import HTTPServer
import http_handler
from threading import Thread

class Server():
    #addr - tuple of IP and port
    def __init__(self, addr):
        self.addr = addr
        self.server = HTTPServer(addr, http_handler.webServerHandler)
        self.server_thread = Thread(target = self.server_loop)
        self.restarting = False

    #run httpserver on it's own thread and listen for commands on the main thread
    def run(self):
        self.server_thread.start()
        print("Server started on" + str(self.addr[0]) + ":" + str(self.addr[1]))
        while True:
            command = input("comm:")
            if command == "shutdown":
                self.server.shutdown()
                print("Server shutting down...")
                self.server_thread.join() #wait for thread to safely terminate
                break
            elif command == "restart":
                self.restarting = True
                self.server.shutdown()
                print("Server restarting ... ")


    #operates server running and it's interrupts
    def server_loop(self):
        print("Server instance running")
        while True:
            self.server.serve_forever()
            if self.restarting == False:
                break
            else:
                self.restarting = False #After restart, set the restart flag to False again


s = Server(("127.0.0.1", 8888))
s.run()
```

```
from PIL import Image
from PIL import ImageEnhance
from PIL import ImageOps
from nnetwork import Network
import numpy as np

class MLService():
    # initialize  network with 28*28 input neurons (image size)20 hidden neurons and 10 output neurons
     net = Network(path="network.json")

     #prepare incoming image to fit it in 28x28 pixs
     def prepareImage(self,img_data):
          image = Image.frombytes('L', (img_data['width'], img_data['height']) , bytes(img_data['array']) )
          image = image.resize((28,28), resample=Image.BICUBIC)
          enhancer = ImageEnhance.Contrast(image)
          image = enhancer.enhance(5.0)
          image = enhancer.enhance(-5.0)
          image = enhancer.enhance(5.0)
          return list(image.getdata())


     def evaluate( self , img_data):
          raw_data = self.prepareImage(img_data)
          raw_data = np.array(raw_data)
          image = np.zeros((784,1))
          image = raw_data.flatten('F')
          image = np.reshape(image,(784,1))
          image = image / 255.0
          print(image.shape)
          activations = self.net.feedforward(image)
          return [int(np.argmax(activations)), float( activations [np.argmax(activations)])]
```

```
# %load network.py
#from http://neuralnetworksanddeeplearning.com by Michael Nielsen – http://michaelnielsen.org/

#### Libraries
# Standard library
import random
import json

# Third-party libraries
import numpy as np

class Network(object):

    def __init__(self, sizes=None, path=None):
        if sizes:
            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:])]
            print("loaded random")
            return

        if path:
            with open(path, "r") as f:
                data = json.loads(f.read())
                self.num_layers = data["num_layers"]
                self.sizes = data["sizes"]
                self.biases = [np.array(b) for b in data["biases"]]
                self.weights = [np.array(w) for w in data["weights"]]
                print("loaded from file")

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):

        training_data = list(training_data)
        n = len(training_data)

        if test_data:
            test_data = list(test_data)
            n_test = len(test_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("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test));
        else:
            print("Epoch {} complete".format(j))

def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    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)
        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)]
    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] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        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)
                    for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
    return (output_activations-y)

# dumps network to a file
def dump(self):
    # dump format:
    with open("network.json", "w+") as f:
        f.write(json.dumps({"sizes": self.sizes, "num_layers": self.num_layers, "biases": [
                b.tolist() for b in self.biases], "weights": [w.tolist() for w in self.weights]}))
```

```
#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

```
#Basic HTTP server
#reacts to GET and POST requests only
#launches on localhost:9999, you can change params if you want
from http.server import BaseHTTPRequestHandler, HTTPServer
import threading
import re
import os.path
import json
from ml_service import MLService


#Handling strategy for this WebServer
class webServerHandler(BaseHTTPRequestHandler):

    # initializes this web server
    #site_path - path to index.html file
    def __init__(self, request, client_address, server):
        self.site = "<html><body><h1>Error loading page</h1></body></html>"
        self.root = "../site/"
        with open("../site/index.html") as f:
            self.site = f.read()
        super().__init__(request, client_address, server)

    #sets default headers
    def _headers(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()

    #method that will be called when user requests something from a server with a GET request
    #sends requested file to a user
    def do_GET(self):
        #prepare file path
        true_path = self.root
        if self.path == "/":
            true_path += "index.html" #if site root was requested, serve index.html
            self._headers()
        else:
            true_path += self.path[1:] #because self.path begins with '/' we silce it out

        #check if file even exists and if it is valid
        if os.path.exists(true_path) == False or os.path.isfile(true_path) == False:
            self.send_response(404) #TODO: Investigate why 404 is sent 8 times in a row
            return


        #check what type of file is being requested and send appropiate meta-data
        if re.match("\.js$", self.path):
            self.send_response(200)
            self.send_header("Content-type", "text/javascript")
            self.end_headers()
        elif re.match("\.html$", self.path):
            self._headers()
        #at the end write requested file to the send buffer
        with open(true_path) as f:
            self.wfile.write(bytes(f.read(), 'utf-8'))
```

```
#method that will be called when user wants to write some data to a server with a POST request
#gets image data from a user and redirects it to a ML service
#then sends JSON data about what ML service thinks that user has written
def do_POST(self):
    self.send_response(200);
    length = int(self.headers["Content-Length"])
    data = self.rfile.read(length)
    json_data = json.loads(data.decode("utf-8"))
    #pass json_data to ML service
    ml = MLService()
    self.wfile.write(bytes(json.dumps(ml.evaluate(json_data)),'utf-8'))
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My test page</title>
    <script type="text/javascript" src="scripts/point.js"></script>
    <script type="text/javascript" src="scripts/drawer.js"></script>
    <script type="text/javascript" src="scripts/main.js"></script>
  </head>
  <body>
     <p>Write the number into box below, click "send" to send data for recognition</p>
     <canvas id="drawing_area" width="280" height="280" style="border:1px solid #000000;"></canvas>
     <button id="clear_button" type="button">Clear</button>
     <button id="send_button" type="button">Send</button>
     <p id="results_text"></p>
  </body>
</html>
```

```
class DrawingModule
{
    constructor()
    {
        this.canvas = document.getElementById("drawing_area");
        this.context = this.canvas.getContext("2d");
        this.rect = this.canvas.getBoundingClientRect();
        this.scale = new Point(this.canvas.width/this.rect.width, this.canvas.height / this.rect.height);
        this.last_point = new Point();
    }


    draw(point)
    {
        this.context.beginPath();
        this.context.strokeStyle = 'black';
        this.context.lineWidth = 15;
        this.context.moveTo(this.last_point.x, this.last_point.y);
        this.context.lineTo(point.x, point.y);
        this.context.stroke();
        this.context.closePath();
        this.last_point = point;
    }


    clear()
    {
        this.context.clearRect(0,0, this.canvas.width, this.canvas.height);
    }
}
```

```
window.onload = function () {
    var module = new DrawingModule();
    var clear_button = document.getElementById("clear_button");
    var drawing = false;
    var result_text = document.getElementById("results_text");

    function draw(env) {
        if (drawing === true) {

            var curr_point = new Point(env.clientX, env.clientY);
            curr_point.x -= module.rect.left;
            curr_point.y -= module.rect.top;
            module.draw(curr_point);

        }

    }

    function stopDrawing() {
        drawing = false;
        module.last_point = new Point();
    }

    function startDrawing() {
        drawing = true;
    }


    //sends image to a server
    function sendDrawing() {
        var img_data = module.context.getImageData(0, 0, module.canvas.width, module.canvas.height)
        var grayscale_data = prepareDrawing(img_data)
        var xhr = new XMLHttpRequest();
        xhr.open("POST", "/");
        xhr.onreadystatechange = function () { // Call a function when the state changes.
            if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
                // Request finished.
                console.log("POST finished");
                var resp = JSON.parse(xhr.response)
                result_text.innerHTML = "Your written number is : " + resp[0] + "| Confidence: " +
                    resp[1];
            }
        }

        xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
        xhr.send(JSON.stringify({ "width": module.canvas.width, "height": module.canvas.height, "array":
            grayscale_data }));
    }


    //Converts image to grayscale, reducing amount of necessary data
    function prepareDrawing(img_data) {
        var result = new Array();
        for (i = 3; i < img_data.data.length; i += 4) {
            //Convert to greyscale (only alpha counts)
```

```
            result.push(img_data.data[i]);
        }
        return result;
    }


    //Add canvas event handlers
    module.canvas.onmousemove = draw;
    module.canvas.onmouseup = stopDrawing;
    module.canvas.onmouseleave = stopDrawing;
    module.canvas.onmousedown = startDrawing;
    clear_button.onclick = function () { module.clear() };
    send_button.onclick = sendDrawing;
}
```

```
class Point
{
    constructor(x, y)
    {
        this.x = x;
        this.y = y;
    }

    //creates point at 0,0
    costructor()
    {
        this.x = 0;
        this.y = 0;
    }

}
```