Dokumentacja projektu
Języki skryptowe

Daniel Jambor, Grupa 2C

4 grudnia 2019

Część I

Opis programu

Program klasyfikuje wysłane przez użytkownika zdjęcie małpy i wyświetla jaki gatunek użytkownik pokazał programowi. Program potrafi odróżnić 10 różnych gatunków, wskazując na prawidłową odpowiedź z 60% skutecznością.

Instrukcja obsługi

Program należy uruchomić poprzez plik *MCNN.bat*, który ukazuje schludne menu:

- Make a database - pierwsza rzecz, którą użytkownik powininen wybrać, opcja ta przygotowywuje dane pod trening naszego programu. Aby wybrać tę opcję, należy nacisnąć '1' i potwierdzić Enterem.
- Train the convolutional neural network - opcja ta pozwala programowi 'nauczyć się' rozpoznawać poszczególne gatunki małp. Ta opcja nie będzie dostępna, dopóki użytkownik nie wykona opcji pierwszej. Aby ją wybrać, należy wpisać '2' i potwierdzić Enterem.
- Load your own image and see the results - opcja, która pozwala załadować własny obraz i sprawdzić odpowiedź programu. Po wybraniu opcji przez naciśnięcie '3' i potwierdzenie Enterem, jesteśmy proszeni o wpisanie nazwy naszego zdjęcia wraz z rozszerzeniem. Plik ten musi być wcześniej wrzucony do folderu o nazwie 'userdata' i być plikiem o rozszerzeniu .jpg.
- About project - wyświetla podstawowe informacje o projekcie, należy nacisnąć '4' i potwierdzić Enterem.
- Exit - kończy działanie programu, należy nacisnąć '5' i potwierdzić Enterem
- Backup - robi kopie zapasową wszystkich kluczowych plików programu, należy nacisnąć '6' i potwierdzić Enterem.

Podczas pracy z programem można napotkać różne komunikaty błędów, które jasno i przejrzyście informują użytkownika o napotkanym problemie:

- Module Error - program nie został jeszcze wytrenowany
- File Error - plik podany przez użytkownika nie istnieje
- Menu Error - wybór użytkownika w menu głównym nie jest w prawidłowej formie.

Część II

Część techniczna

Program jest napisany w języku python 3.7 i w batchu. Sam python wykorzystuje kilka dodatkowych bibliotek potrzebnych do uruchomienia oraz edytowania całego programu:

- *numpy*
- *tqdm*
- *torch*
- *cv2*
- *sys*
- *webbrowser*

Program dzieli się na kilka plików źródłowych:

- *MCNN.bat*
- *dataCreator.exe*
- *Net.py*
- *main.py*
- *getResult.py*

Oraz na foldery:

- *backup* - znajduje się w nim kopia wszystkich kluczowych plików źródłowych.
- *bin* - główny folder programu, znajdują się w nim między innymi pliki *trainingData.npy* oraz *model.pth*. Można w nim znaleźć plik *menu.txt*, które zawiera menu główne programu oraz *info.txt*, które zawiera trochę informacji o projekcie. Sam folder zawiera również foldery:
  - *data* - zawiera folder *description*, który przechuwje pliki tekstowe zawierające krótkie opisy poszczególnych gatunków małp, oraz folder *monkeys*, który zawiera zdjęcia poszczególnych gatunków małp.
  - *dataCreator* - to jest folder z wyeksportowanym skryptem *dataCreator.py* do formatu *.exe*
  - *userdata* - folder przechowujący zdjęcia użytkownika, które chce on przekazać wytrenowanej sieci neuronowej.

Program również wymaga do pracy pobranej już bazy danych małp, tak, aby każdy gatunek był w innym folderze nazywanym kolejno $n0, n1....n9$.

Opis działania

0.0.1 MCNN.bat

Skrypt batchowski który jest menu głównym całego programu. Zawiera on poszczególne opcje wymienione w Częsci Pierwszej. Skrypt składa się z kilku instrukcji warunkowych oraz z pętli. Zawiera dwie zmienne: *choice*, która odpowiada za wybór użytkownika w menu głównym, oraz *fileChoice*, która odpowiada za wybór zdjęcia przez użytkownika. Skrypt zawiera również kontrolę błędów przed niechcianymi akcjami, takimi jak wpisanie nieprawidłowego wyboru lub wybranie przez użytkownika zdjęcia, które nie istnieje w folderze *userdata*. Kontrola błędów jest zrealizowana za pomocą instrukcji warunkowych oraz pętli *GOTO*, która przenosi użytkownika do kodu odpowiedzialnego za wypisywanie komunikatów o błędach. Skrypt włącza inne skrypy za pomocą komendy *CALL*.

0.0.2 dataCreator.exe

Skrypt napisany w Python odpowiedzialny za generowanie danych. Korzysta z bibliotek:

- *os* - biblioteka potrzebna do uzyskania listy wszystkich plików w folderze.
- *cv*2 - biblioteka wykorzystywana do manipulowania plikami *.png*.
- *numpy* - biblioteka potrzebna do utworzenia pliku końcowego.

Skrypt zawiera klasę *Monkeys*, która jest odpowiedzialna za całe działanie skryptu. Program ten ma z góry narzuconą ścieżkę do wszystkich folderów z gatunkami małp. Następnie przypisuje każdemu gatunkowi odpowiednią cyfrę od 0 do 9.
Klasa zawiera jedną metodę - *makingData*(), która jest lwią częścią całego skryptu. Tak naprawdę skrypt jest jedną wielką pętlą, która przechodzi przez wszystkie labele i pliki w folderze odpowiadającym danemu gatunkowi. Ustawia ścieżkę do pliku, wczytuje go do zmiennej w trybie czarno-białym, zmienia wymiary zdjęcia i dodaje do listy razem z macierzą jednostkową, która będzie odpowiadała prawidłowym odpowiedziom na pytanie jaki gatunek małpy jest na zdjęciu. Po zakończeniu pracy pętli, dane są losowo mieszane oraz zapisywane jako plik *trainingData.npy*. Poza klasą jest tworzona zmienna *monkeys* klasy *Monkeys* i wywoływana funkcja *makingData*()

0.0.3 Net.py

Skrypt napisany w Python. Jest to klasa konwolucyjnej sieci neuronowej. Zawiera ona biblioteke *torch*, która jest biblioteką przeznaczoną do uczenia maszynowego. Na początek tworzymy klasę $Net(nn.Module)$ która dziedziczy po $nn.Module$. Jest to bowiem klasa bazowa dla wszystkich sieci neuronowych. Następnie tworzymy konstruktor klasy, a w nim wywołujemy go dopisując przedrostek $super()$.. Jest on potrzebny do wywołania naszego konstruktora, bo pochodzi on z $nn.Module$. Następnie towrzymy 3 warstwy konwolucyjne *conv* i nadajemy im wartość $nn.Conv2d()$. Jest to funkcja, która wykonuje splot w dwóch wymiarach na danych wejściowych złożonych z kilku płaszczyzn. W najprostszym przypadku, dane wyjściowe warstwy:

$$(N, C_{out}, H_{out}, W_{out})$$

Przy danych wejściowych warstwy:

$$(N, C_{in}, H, W)$$

Mogą zostać przedstawione w taki sposób:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

Gdzie: $\star$ to korelacja wzajemna; N to wielkość kroku; C to liczba kanałów; H to wysokość danych wyjściowych podana w pikselach, a W to szerokość.

Każda kolejna warstwa będzie posiadać inne argumenty: do pierwszej warstwy wysyłamy jedno zdjęcie (1), oczekujemy od niej szesnastu danych wyjściowych (16) przy wymiarach kernela 5x5 (5). Z tego wszystkiego wychodzi nam $nn.Conv2d(1, 16, 5)$. Każda kolejna warstwa jako pierwszy argument będzie przyjmnować drugi argument poprzedniej warstwy, a drugi argument będzie mnożony razy dwa. Wymiary kernela zostają bez zmian.
Następnie potrzebujemy wygenerować fałszywe dane, aby sprawdzić jaki jest kształt danych wejściowych, które będą wykorzystywane w pierwszej warstwie typu fc (z ang. fully connected). Zamieniamy więc zmienną x na tensor o wymiarach $(-1, 1, 64, 64)$ wypełniony losowymi liczbami. Następnie ustawiamy zmienną *toLinear* na wartość *None* i wywołujemy metodę $convs(x)$ o której będzie później.
Następnie tworzymy trzy warstwy fc za pomocą $nn.Linear(in, out)$. Funkcja pozwala nam zastosować przekształcenie liniowe danych:

$$y = xA^T + b$$

W pierwszej warstwie w pole *in* wrzucamy naszą zmienną *toLinear*, w pole *out* wrzucamy wartość 1024. Ostatnia warstwa typu fc będzie posiadała wartość $out = 10$, ponieważ tyle gatunków małp mamy w bazie danych. Sieć może zwracać 10 możliwych odpowiedzi: od 0 do 9
Następnie mamy metodę $convs(x)$, która za argument przyjmuje wcześniej stworzone dane. W metodzie przepuszczamy nasz tensor x przez wszystkie warstwy konwolucyjne. Jednak wcześniej wywołujemy funkcję $F.maxpool2d$, która zwraca maksymalną wartość z danych wejściowych złożonych z kilku płaszczyzn. W najprostszym przypadku, dane wyjściowe warstwy:

$$(N, C, H_{out}, W_{out})$$

Przy danych wejściowych warstwy:

$$(N, C, H, W)$$

I wymiarach kernela:

$$(kH, kW)$$

Mogą zostać przedstawione w taki sposób:

$$out(N_i, C_j, h, w) = \\ max_{m=0,...,kH-1} min_{n=0,...,kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Gdzie: N to wielkość kroku; C to liczba kanałów; H to wysokość danych wyjściowych podana w pikselach, a W to szerokość; kH to wysokość kernela podana w pikselach a kW to szerokość.

W tym przypadku jako argument podajemy również $F.relu()$ - jest to funkcja aktywacyjna:

$$ReLU(x) = max(0, x)$$

Gdy nasz x przejdzie przez wszystkie warstwy, zmiennej *toLinear* przypisywana jest wartość iloczynu wszystkich trzech wymiarów zmiennej x, a sama zmienna jest zwracana.
Ostatnią metodą w naszej klasie jest $forward(x)$, która odpowiada za przepuszczenie danych przez warstwy typu fc. Na sam początak jest wywoływana metoda $convs(x)$, która przepuszcza zmienną przez wartswy konwolucyjne i zapisuje jej stan. Następnie zmieniamy kształt tensora x na $(-1, toLinear)$. Wartość -1 oznacza, że ten atrybut nie jest znany, i może być dowolny. *toLinear* natomiast to iloczyn wszystkich wymiarów tensora x. Następnie przepusczamy nasz x przez dwie pierwsze warstwy typu fc (łącznie z funkcją aktywacyjną). W trzeciej warstwie nie używamy funkcji aktywacyjnej, ponieważ po przejściu przez trzecią warstwę powinniśmy uzyskać nasz wynik. Następnie zwracana jest $F.softmax(x, dim = 1)$. $F.softmax()$ stosuje z ang. Softmax function, czyli:

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

dim oznacza wymiar w którym będziemy liczyć - w tym przypadku jest to 1.

0.0.4 main.py

Skrypt napisany w Python odpowiedzialny za wczytanie danych oraz wytrenowanie sieci neuronowej. Skrypt zawiera biblioteki: *numpy*, *tqdm*, *torch* oraz importuje plik *Net.py*. Biblioteka *tqdm* odpowiada za wyświetlanie pasku pokazującego postęp w trenowaniu. Skrypt na początku ustala, czy nasza karta graficzna jest wspierana przez *CUDA* - jeśli jest, to ustawia *device* na *cuda* : 0, jeśli nie - ustawia *device* na *cpu*. Jeśli posiadamy wsparcie *CUDA*, to trenowanie sieci neuronowej odbywa się przy wykorzystaniu pamięci karty graficznej, co pozwala przyspieszyć cały proces. Należy pamiętać, aby przy naszej sieci oraz przy danych dopisywać $.to(device)$, inaczej skrypt wyrzuci błąd.
Zostaje utowrzona zmienna *net* klasy $Net()$ oraz zostaje wczytany plik z danymi *trainingData.npy* do zmiennej *trainingData*.
Następnie zostaje utworzona zmienna *optimizer* i przypisujemy jej $optim.Adam(net.parameters(), lr = 0.001)$.
optim.Adam impementuje tzw. algorytm Adama, który odpowiada za optymalizacje stochastyczną. Algorytm ten wygląda następująco:

Źródło: ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Data: α - wielkość kroku, nasze lr; zazwyczaj $\alpha = 0.001$
$\beta_1, \beta_2 \in [0, 1)$ - wykładnicze współczynniki zaniku momentalnego przybliżenia; zazwyczaj $\beta_1 = 0.9, \beta_2 = 0.999$
$f(\theta)$ - funkcja kosztu z argumentem θ
θ_0 - początkowy parametr modelu
$m_0 = 0$ - początkowy estymat pierwszego (średnia) momentu gradientu
$v_0 = 0$ - początkowy estymat drugiego (wariacja) momentu gradientu
$t = 0$ - czas (z ang. timestep)
$\varepsilon = 10^{-8}$ - błąd;
while θ_t *nie jest zbieżna do 0* **do**
  $t = t + 1$ (ikrementacja czasu);
  $g_t = \nabla_\theta f_t(\theta_{t-1})$ (obliczanie gradientu w chwili t);
  $m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$ (aktualizacja estymatu pierwszego);
  $v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2$ (aktualizacja estymatu drugiego; g_t^2 - podniesienie do kwadratu każdego elementu macierzy);
  $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$ (poprawka na odchylenie w stronę zera);
  $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$ (poprawka na odchylenie w stronę zera);
  $\theta_t = \theta_{t-1} - \alpha * \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$ (aktualizacja parametrów);
end
return θ_t *(końcowe parametry);*

Algorithm 1: Algorytm Adama

Jako argument do funkcji podajemy *net.parameters*() - zwraca parametry modelu, które można jeszcze czegoś nauczyć, a *lr* to z ang. The Learning Rate - wartość ta jest odpowiedzialna za prawidłowe tempo uczenia się. Jeżeli *lr* będzie za mały, to sieć będzie potrzebować za dużo czasu, aby zniwelować straty do zera, jeśli natomiast wartość będzie za duża, sieć może 'utknąć' w pewnym momencie i już niczego się nie nauczyć.
Następnie definiujemy $lossFunction = nn.MSELoss()$. Zapisujemy do zmiennej *lossFunction* funkcję, która mierzy błąd średniokwadratowy między każdą odpowiedzią sieci, a prawidłową odpowiedzią. Niezredukowane straty (z ang. loss) są opisane wzorem:

$$l(x, y) = L = l_1, ..., l_N, l_n = (x_n - y_n)^2$$

Gdzie N jest wartością reprezentującą wielkość kroku (z ang. batch size) algorytmu. W tym przypadku do wartość końcowa to

$$l(x, y) = mean(L)$$

Czyli średnia z liczonego L. x i y reprezentują odpowiedzi sieci oraz odpowiedzi prawidłowe, które są w postaci tensoru. Tensory te mogą mieć różne kształty, lecz muszą mieć taką samą liczbę elementów n.
Następnie przypisujemy zmiennej *input* wszystkie zdjęcia jako tensor o kształcie $(-1, 64, 64)$ zapisane w liście *trainingData*, i dzielimy *input* przez 255 - tak, aby tensory posiadały wartości między 0 a 1. Następnie przypisujemy zmiennej *label* wszystkie odpowiedzi jako tensory.

Następnie definiujemy funkcję *train*() która będzie trenować naszą sieć. Na samym początku ustawiamy $BATCHSIZE$ na 100. $BATCHSIZE$, jak wcześniej było wspomniane, jest wielkością kroku, czyli liczbą danych, które zostaną wykorzystane w jednej iteracji. Czyli np. jeśli mamy 1300 zdjęć oraz $BATCHSIZE = 100$, program wykona 13 iteracji. Potem definiujemy zmienną $EPOCHS$ i ustawiamy ją na 89. Zmienna ta określa ile generacji, ile razy sieć będzie się uczyć. Wartość 89 jest najbardziej optymalna dla tej sieci i skutkuje skutecznością ok.60%
Więc 89 razy sieć będzie powtarzać daną sekwencję: przypisujemy zmiennej *batchInput* zmienną listę *input*, która została podzielona na wielkość $BATCHSIZE$. Nadajemy jej również kształt $(-1, 1, 64, 64)$. Tą czynność poza nadaniem kształtu robimy z listą *label*, przypisując ją do *batchLabel*. Następnie ustawiamy gradient na zero za pomocą *net.zerograd*(), i zdobywamy odpowiedź od naszej sieci neuronowej poprzez $output = net(batchInput)$. Następnie do zmiennej *loss* przypisujemy wykalkulowaną wartość starty $lossFunction(output, batchLabel)$, czyli porównujemy odpowiedź sieci a prawdziwą odpowiedź. Poprzez *loss.backward*() liczymy gradient dla wszystkich tensorów, a *optimizer.step*() aktualizuje i używa ich (tensorów) gradientu do zaktualizowania ich parametrów. Innymi słowy, sieć uczy się odpowiadać prawidłowo. Następnie jest wywoływana funkcja *train*, a po skończonej nauce sieć jest zapisywana jako *model.pth*.

0.0.5 getResult.py

Skrypt napisany w Python odpowiedzialny za przetwarzanie zdjęcia użytkownika, uzyskiwanie odpowiedzi oraz generowanie pliku *html*.Skrypt zawiera biblioteki: *cv*2, *numpy*, *torch*, *sys*, *webbrowser*. Z niewymienionych wcześniej bibliotek, zawiera ona *webbrowser*, która odpowiada za otworzenie okna przeglądarki z wygenerowanym plikiem *html*. Skrypt tworzy zmienną przechowującą nazwę pliku użytkownika, która jest podawana jako argument przy starcie skryptu. Potem ustala ścieżkę do wytrenowanego już modelu oraz tworzy zmienną *net* klasy *Net*(). Następnie ładuje wytrenowaną sieć i ustawią ją na tryb użytkowy. Ustawiana jest ścieżka do pliku, dodając do ścieżki do folderu *userdata* nazwę zdjęcia użytkownika.
Następuję powtórzenie algorytmu ze skryptu *dataCreator.py* tzn. następuję zapisanie zdjęcia do zmiennej *img* w trybie czarno-białym, zmienienie kształtu na $64px$ x $64px$ oraz wczytanie do listy. Następuje przekonwertowanie naszego zdjęcia na tensor o kształcie $(-1, 64, 64)$ i podzielenie go przez 255.0, tak, aby wartości w nim były między 0 a 1. Następnie wysyłamy uzyskany tensor o kształcie $(-1, 1, 64, 64)$ do sieci neuronowej i zapisujemy wynik w zmiennej *netOut*. Następnie do zmiennej *predicted*, która jest naszą odpowiedzią, zapisujemy indeks największej wartość w tensorze *netOut* i rzutujemy ją na *int*, tak, aby uzyskać liczbę naturalną między 0 a 9. Potem zależnie od wartości *predicted*, przypisujemy zmiennej *monkeyName* odpowiednią nazwę gatunku oraz zmiennej *monkeyFileName* odpowiednią nazwę pliku zawierającego krótki opis gatunku.
Otwieramy plik *result.html* w trybie do zapisu oraz plik *monkeyFileName*. Przypisujemy zmiennej *description* zawartość pliku *monkeyFileName* i zamykamy go. Następnie do zmiennej *content* przypisujemy kod *html* naszej strony, która wyświetla zdjęcie użytkownika, nazwę gatunku oraz jego krótki opis. Potem zapisujemy kod do pliku *result.html*, zamykamy go oraz otwieramy w nowym oknie przeglądarki plik *result.html*.

Implementacja

Pseudokod odpowiedzialny za uzyskiwanie odpowiedzi programu:

- $filename$ - nazwa pliku .jpg podawanego jako argument.
- $MPATH$ - ścieżka do wytrenowanej sieci neuronowej.

Data: fileName, MPATH;
Result: Liczba całkowita od 0 do 9 reprezentująca dany gatunek małp;
stworzenie zmiennej net klasy $Net()$;
Ustawienie net w tryb $eval()$;
Ustawienie zmiennej $IPATH$ jako ścieżkę do zdjęcia;
Wczytanie zdjęcia do zmiennej img w trybie czarno-białym;
Zmienienie wymiarów zdjęcia na 64px x $64px$;
Dodanie zdjęcia jako tablice pixeli do zmiennej o tej samej nazwie;
Zamienienie zdjęcia na tensor o wymierach $(-1, 1, 64, 64)$;
Podzielenie zdjęcia przez liczbę pikseli: 255;
Uzyskanie odpowiedzi od sieci neuronowej;
Przekonwertowanie odpowiedzi na typ całkowity;
if *Odpowiedź jest równa m gdzie m jest liczbą całkowitą od* 0 *do* 9 **then**
  Ustawienie zmiennej monkeyName na $nazwa(nm)$ i ustawienie pliku z opisem gatunku na $nm.txt$;
end

Algorithm 2: Uzyskiwanie odpowiedzi od sieci neuronowej

Pseudokod odpowiedzialny za generowanie danych:

- $n0, ..., n9$ - ścieżki do folderów z poszczególnymi zdjęciami małp

Data: $n0, ...n9$;
Result: *trainingData.npy*;
Utworzenie zmiennej $LABELS$ przypisującej każdemu gatunkowi cyfrę od 0 do 9;
Utworzenie pustej listy *data*, która będzie przechowywać dane;
for *każdej zmiennej label w* $LABELS$ **do**
    Wyświetl komunikat informujący o aktualnym statusie przetwarzania danych;
    for *każdego pliku w danym folderze* **do**
        if *wystąpi błąd* **then**
            Wyświetl błąd;
        else
            Ustaw ścieżke do zdjęcia i zapisz w zmiennej *path*;
            Wczytaj zdjęcie w trybie czarno-białym do zmiennej *img*;
            Zmień rozmiar zdjęcia na 64px x 64px i zapisz do zmiennej *img*;
            Dodaj *img* do *data* razem z macierzą jednostkową o rozmiarach 10x10;
        end
    end
end
Posortuj listę *data* w sposób losowy;
Zapisz listę *data* jako plik *trainingData.npy*;
Wyświetl komunikat o zakończonym procesie generacji danych;

Algorithm 3: Przygotowanie danych do trenowania sieci neuronowej

Pełen kod programu

Wklejamy pełen kod z podziałem na pliki, np.:

Listing 1: MCNN.bat

```
@ECHO OFF


:MENU
CLS

TYPE "bin\menu.txt" && (
REM
) || (
PAUSE
EXIT
)

SET /P choice="Choice>>"

IF %choice% EQU 1 (
GOTO ONE
)
IF %choice% EQU 2 (
GOTO TWO
)
IF %choice% EQU 3 (
GOTO THREE
)
IF %choice% EQU 4 (
GOTO FOUR
)
IF %choice% EQU 5 (
GOTO FIVE
)
IF %choice% EQU 6 (
GOTO SIX
)


GOTO MENUERROR


:ONE
IF EXIST bin\trainingData.npy (
ECHO Data has already been created!
PAUSE
GOTO MENU
)
ECHO Creating data...
CALL bin\dataCreator\dist\dataCreator.exe


:IFFILEEXIST
IF EXIST bin\trainingData.npy (
```

```
PAUSE
GOTO MENU
    )
    GOTO IFFILEEXIST


    :TWO
    IF EXIST bin\model.pth (
        ECHO The CNN has been already trained!
        PAUSE
        GOTO MENU
    )
    IF NOT EXIST bin\trainingData.npy (
        ECHO There's no dataset!
        PAUSE
        GOTO MENU
    }

    ECHO Training the cnn...
    CALL C:\ProgramData\Anaconda3\Scripts\activate.bat
    python main.py
    ECHO Training completed!
    PAUSE


    GOTO MENU
    :THREE
    IF NOT EXIST bin\model.pth (
GOTO NOTTRAINEDERROR
    )

    SET /P fileChoice="Enter the name of the file >>"

    IF EXIST bin\userdata\%fileChoice% (
CALL C:\ProgramData\Anaconda3\Scripts\activate.bat
python getResult.py %1 %fileChoice%
PAUSE
GOTO MENU
    )

    GOTO FILEERROR


    GOTO MENU
    :FOUR
    CLS
    TYPE "bin\info.txt" && (
    PAUSE
    GOTO MENU
    ) || (
    PAUSE
    GOTO MENU
    )

    :FIVE
```

```
EXIT

:SIX
MKDIR backup
XCOPY getResult.py backup
XCOPY main.py backup
XCOPY Net.py backup
XCOPY bin\dataCreator\dataCreator.py backup
GOTO MENU

:MENUERROR
ECHO Menu Error: Incorrect choice!
PAUSE
GOTO MENU

:FILEERROR
ECHO File Error: File doesn't exist!
PAUSE
GOTO MENU

:NOTTRAINEDERROR
ECHO Module Error: The CNN isn't trained!
PAUSE
GOTO MENU

ECHO ON
```

Listing 2: dataCreator.exe

```
#Made by Daniel Jambor, 12.03.2019
#
#Sources: pythonprogramming.net by harrison@pythonprogramming.net
#Pytorch.org
#stackoverflow.com

import os
import cv2
import numpy as np

class Monkeys():

    #setting the file path of each monkey
    n0 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n0'
    n1 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n1'
    n2 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n2'
    n3 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n3'
    n4 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n4'
    n5 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n5'
    n6 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n6'
    n7 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n7'
    n8 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n8'
    n9 = 'C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
____/Project/bin/data/monkeys/n9'

    #setting labels of monkeys
    LABELS = {n0: 0, n1: 1, n2: 2, n3: 3, n4: 4, n5: 5, n6: 6, n7: 7, n8: 8, n9: 9}

    #our data list
    data = []

    #print the message
    print("Creating_the_dataset...")

    def makingData(self):
        #for every label in label list
        for label in self.LABELS:
            #print the message
            print(f"Making_data_from_{self.LABELS[label]_+_1}_label...")
            #for every file (image) in folder
            for f in os.listdir(label):
                try:
                    #setting path of image
                    path = os.path.join(label, f)
```

```
                    #lodaing the image in grayscale mode
                    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
                    #resizing the image
                    img = cv2.resize(img, (64, 64))
                    #adding to data list our image in array version and adding
                    #the array with ones on the diagonal and zeros elsewhere
                    #(size of the numbers of monkeys)
                    self.data.append([np.array(img), np.eye(10)[self.LABELS[label]]])
                except Exception as e:
                    #if error, print
                    print(e)

        #shuffle the data list randomly
        np.random.shuffle(self.data)
        #save our database as trainingData.npy
        np.save("C:/Users/danie/Documents/Studies/Term_III/Scripting_Languages
________/Project/bin/trainingData.npy", self.data)
        #print the message
        print("\nCreating_dataset_completed!\n")

monkeys = Monkeys()
monkeys.makingData()
```

Listing 3: Net.py

```
#Made by Daniel Jambor, 12.03.2019
#
#Sources: pythonprogramming.net by harrison@pythonprogramming.net
#Pytorch.org
#stackoverflow.com

import torch
import torch.nn as nn
import torch.nn.functional as F

#creating class inhereits from nn.Module - the base class for all neural network modules
class Net(nn.Module):

    #init function
    def __init__(self):

        #super() - inhereits from nn.Module and run Net's init method
        super().__init__()

        #creating convolutional layers; nn.Conv2d - applies a
        #2D convolution over an input signal composed of several input planes
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)

        #creating some fake data to check the shape of the flattened output
        x = torch.randn(64, 64).view(-1, 1, 64, 64)
        self.toLinear = None
        self.convs(x)

        #creating fully connected layers; nn.Linear(size of each input,
        #size of each output) - applies a linear transformation to the incoming data
        self.fc1 = nn.Linear(in_features=self.toLinear, out_features=1024) #flattering
        self.fc2 = nn.Linear(in_features=1024, out_features=512)
        self.fc3 = nn.Linear(in_features=512, out_features=10)

    #function which passes data through the conv layers
    def convs(self, x):

        #passing through activation function: F.max_pool2d - applies a 2D max pooling
        #over an input signal composed of several input planes (here: 2x2),
        #then runs activation function
        x = F.max_pool2d(F.relu(self.conv1(x)), kernel_size=2)
        x = F.max_pool2d(F.relu(self.conv2(x)), kernel_size=2)
        x = F.max_pool2d(F.relu(self.conv3(x)), kernel_size=2)

        #checking shape to flatting
        if self.toLinear is None:
            self.toLinear = x[0].shape[0]*x[0].shape[1]*x[0].shape[2]

        return x

    #fuction which passes data through the fc layers
```

```
    def forward(self, x):

        x = self.convs(x)

        #view - returns a new tensor with the same data but of different shape;
        #-1 means any value
        x = x.view(-1, self.toLinear)

        #passing through activation function F.relu
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        #last layer is output - dont run activation function
        x = self.fc3(x)

        #return softmax of x in dimension 1
        return F.softmax(x, dim=1)
```

Listing 4: main.py

```
#Made by Daniel Jambor, 12.03.2019
#
#Sources: pythonprogramming.net by harrison@pythonprogramming.net
#Pytorch.org
#stackoverflow.com

import numpy as np
from tqdm import tqdm

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from Net import *



#if cuda is available, set it on and print the message, if not,
#set on cpu and print the message
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("\nRunning on the GPU\n")
else:
    device = torch.device("cpu")
    print("\nRunning on the CPU\n")


#creating our net
net = Net().to(device)


#loading our database
trainingData = np.load("./bin/trainingData.npy", allow_pickle=True)


#creating the optimizer
optimizer = optim.Adam(net.parameters(), lr=0.001)
#creating the loss function
lossFunction = nn.MSELoss()


#loading our images to input and converting to the Tensor of (anyValue, imgSize) shape
input = torch.Tensor([i[0] for i in trainingData]).view(-1, 64, 64)
#divide by the number of pixels
input = input/255.0
#loading our matrix as Tensor - this is our correct answers
label = torch.Tensor([i[1] for i in trainingData])


#training function
def train(net):

    #setting the number of training examples utilized in one iteration
```

```
    BATCH_SIZE = 100
    #generation of our net
    EPOCHS = 89


    #in every generation
    for epoch in tqdm(range(EPOCHS)):

        #iterate over the length of trainset, taking steps of the size of batch size
        for i in range(0, len(input), BATCH_SIZE):

            #setting our batchInput and reshaping it
            batchInput = input[i:i+BATCH_SIZE].view(-1, 1, 64, 64).to(device)
            #setting our batchLabel
            batchLabel = label[i:i+BATCH_SIZE].to(device)


            #setting the gradients to zero
            net.zero_grad()
            #getting our output
            output = net(batchInput)
            #compare output with batchLabel and setting the loss
            loss = lossFunction(output, batchLabel)
            #computes math stuff for every parameter which requires grad
            loss.backward()
            #performing a parameter update based on the current gradient
            optimizer.step()


#acrivating train function
train(net)

#saving trained net as model.pth
torch.save(net.state_dict(), './bin/model.pth')
```

Listing 5: getResult.py

```
#Made by Daniel Jambor, 12.03.2019
#
#Sources: pythonprogramming.net by harrison@pythonprogramming.net
#Pytorch.org
#stackoverflow.com

import cv2
import numpy as np
import torch
import sys
import webbrowser

from Net import *

#set file name as argument from batch script
filename = sys.argv[1]

#set path to the trained model
MPATH = './bin/model.pth'

#creating net variable
net = Net()

#loading trained model
net.load_state_dict(torch.load(MPATH))
#setting eval mode
net.eval()

#setting path to image
IPATH = './bin/userdata/' + filename

#reading image in grayscale mode
img = cv2.imread(IPATH, cv2.IMREAD_GRAYSCALE)
#resizing image
img = cv2.resize(img, (64, 64))
#adding image to np array
img = np.array(img)

#transfer image to tensor and reshaping it
img = torch.Tensor(img).view(-1, 64, 64)
#divide by the number of pixels
img = img/255.0

#get the unconverted output
netOut = net(img.view(-1, 1, 64, 64))

#get predicted answer
predicted = int(torch.argmax(netOut))

#checking the answer and saving it; setting the name of description file
if predicted == 0:
    monkeyName = "Mantled Howler (n0)"
    monkeyFileName = "n0.txt"
```

```
elif predicted == 1:
    monkeyName = "Patas Monkey (n1)"
    monkeyFileName = "n1.txt"
elif predicted == 2:
    monkeyName = "Bald Uakari (n2)"
    monkeyFileName = "n2.txt"
elif predicted == 3:
    monkeyName = "Japanese Macaque (n3)"
    monkeyFileName = "n3.txt"
elif predicted == 4:
    monkeyName = "Pygmy Marmoset (n4)"
    monkeyFileName = "n4.txt"
elif predicted == 5:
    monkeyName = "White Headed Capuchin (n5)"
    monkeyFileName = "n5.txt"
elif predicted == 6:
    monkeyName = "Silvery Marmoset (n6)"
    monkeyFileName = "n6.txt"
elif predicted == 7:
    monkeyName = "Common Squirrel Monkey (n7)"
    monkeyFileName = "n7.txt"
elif predicted == 8:
    monkeyName = "Black Headed Night Monkey (n8)"
    monkeyFileName = "n8.txt"
elif predicted == 9:
    monkeyName = "Nilgiri Langur (n9)"
    monkeyFileName = "n9.txt"

#open the html file
f = open('result.html','w')

#open the description file
monf = open('./bin/data/description/' + monkeyFileName, 'r')
#save description into variable
description = monf.read()
#close description file
monf.close()

#save the content of html file
content = f"""<html>
    <head>
        <body>
            <center>
                    <img src="bin/userdata/{filename}" alt="Monkey"
                    align="middle" border="5" style="width:600px;height:600px;">
                    <p> <b> <font size="6"> This is {monkeyName}! </font> </b> </p>
                    <p> <i> <font size="4"> {description} </font> </i> </p>
            </center>
    </head>
</html>"""

#save html file
f.write(content)
#close html file
f.close()
```

```
#open saved html as result
webbrowser.open_new_tab('result.html')
```