

STU FIIT
Umela Inteligencia

“Klasifikácia”

Zadanie č. 2a

Made by: Meredov Nazar
Cvičiaci: Bc. Jakub František Setnický
AIS ID: 122092

Anotácia:

Táto úloha zahŕňa implementáciu klasifikátora k-najbližších susedov (KNN) pre body v 2D priestore pomocou neurónových sietí, kde body patria do jednej zo štyroch farebných tried: červenej, zelenej, modrej alebo fialovej

Na začiatku je 20 preddefinovaných bodov a klasifikátor bude klasifikovať nové body na základe ich súradníc X a Y

Klasifikátor sa otestuje na 40 000 novo vytvorených bodoch a experiment spravuje pre najlepsie hodnoty k

Výsledky sa vizualizujú a presnosť sa hodnotí porovnaním počtu bodov v rámci bariér

Opis úlohy:

Úlohou je navrhnúť a nárešovať neurónovú sieť na klasifikáciu bodov podľa ich súradníc v priestore. Model sa trénuje na vopred definovaných údajoch a generuje náhodné body, ktoré spadajú do rôznych oblastí priestoru reprezentujúcich štyri triedy: R, G, B a P

Dôležitou požiadavkou je generovať body v presne definovaných oblastiach pre každú triedu a všetky vygenerované body musia byť jedinečné

Model tieto body klasifikuje a výsledky vizualizuje na grafe

Obsah:

ANOTÁCIA:	2
OPIS ÚLOHY:	3
OBSAH:	4
PREČO,AKO, A NAČO TO ROBIME?	5
MAIN ALHORITMUS NN SIETE	7
USER REŽIM	7
• <i>Ako spustime?</i>	7
• <i>Ako funguje?</i>	7
• <i>Priklady spustenia kódu</i>	8
TESTING REŽIM	9
• <i>Ako spustime?</i>	9
• <i>Ako funguje?</i>	9
EDA ANALIZIS PRE NN SIEŤ	10
A. METODA PRE GENEROVANIE DÁT PRE NN:	10
B. GENEROVANIE DÁT S RÔZNYMI HODNOTAMI K (PRE EDA A NN):	11
C. ALGORITMUS NA NÁJDENIE OPTIMÁLNEHO K:	11
HLAVNE ČIASTY NN SIETE	13
DEFINÍCIA TRIEDY MODULU NN	13
DEFINÍCIA TRIEDY SÚBORU ÚDAJOV (DATASET)	14
AKO SPOČITAVAME LOSS MODELU?.....	15
AKO TRÉNUJEME A TESTUJEME MODEL?	16
AKO FUNGUJE METOD CLASSIFY?	18
AKO VYPOČÍTAME PRESNOSŤ?	19
TESTOVANIE A ANALITIKA:	20
A. VYSLEDKY TESTOVANIA:	22
B. VYSLEDKY ANALITIKY:	ERROR! BOOKMARK NOT DEFINED.
ZHODNOTENIE	23

Prečo,ako, a načo to robime?

Prečo to robíme?

Cieľom je vytvoriť model, ktorý dokáže klasifikovať body v priestore na základe ich súradníc. Tento typ úlohy je praktický pre rôzne aplikácie, kde je dôležité správne priradiť body do kategórií podľa ich polohy, napríklad v **geografických systémoch, analýze dát alebo spracovaní obrazu**.

Ako to robíme?

Používame **neurónovú sieť**, ktorá sa učí na základe existujúcich bodov a ich tried, a potom **dokáže klasifikovať nové**, náhodne generované body. Body sa generujú v špecifických zónach priestoru s vysokou pravdepodobnosťou, pričom zabezpečujeme, aby sa žiadne body ani triedy neopakovali. Následne model body klasifikuje a výsledky vizualizujeme na grafe.

Načo to robíme?

Toto cvičenie nám pomáha pochopiť, ako fungujú neurónové siete , a zároveň zlepšujeme schopnosti modelu rozlišovať rôzne triedy na základe priestorových dát.

Získané znalosti môžeme využiť pri riešení reálnych problémov, ako je analýza distribúcie dát alebo kategorizácia objektov.

P.s Pomohlo mi to na IAU.

Ako môžeme startovať program?

- Ako fungujú config súbory a súbory requirements?

Konfiguračné (config) súbory umožňujú uložiť rôzne parametre a nastavenia mimo zdrojového kódzu, aby sa dali ľahko upravovať bez nutnosti meniť samotný kód. Používa sa na to jednoduchý formát ako napríklad INI, kde sú nastavenia usporiadane do sekcií a kľúč-hodnota párov. V tomto prípade je použitý súbor config.txt, ktorý definuje rôzne parametre pre projekt:

- **[start-bods]**: Táto sekcia definuje počiatočné súradnice pre rôzne body (R, G, B, P). Každý rad dvojíc predstavuje skupiny bodov, kde napríklad body R sa nachádzajú v západnom kvadrante a t.d.
- **[generate]**: Obsahuje parametre pre generovanie bodov.
 - num_generate: Počet bodov, ktoré budú generované.
 - seed: Náhodné semeno pre konzistentné výsledky generovania.
 - knn: Počet najbližších susedov (knn), ktorý sa bude používať v súvislosti s modelom.
- **[neuron-settings]**: Parametre neurónovej siete.
 - learning-rate: Rýchlosť učenia pre optimalizátor.
 - batch-size: Veľkosť dávky dát počas trénovalia.
 - early-stopping-patience: Počet epoch, počas ktorých trénovanie pokračuje bez zlepšenia, predtým než sa spustí mechanizmus skorého zastavenia (early stopping).

```
[start-bods]
R = [[-4500, -4400], [-4100, -3000], [-1800, -2400], [-2500, -3400], [-2000, -1400]]
G = [[+4500, -4400], [+4100, -3000], [+1800, -2400], [+2500, -3400], [+2000, -1400]]
B = [[-4500, +4400], [-4100, +3000], [-1800, +2400], [-2500, +3400], [-2000, +1400]]
P = [[+4500, +4400], [+4100, +3000], [+1800, +2400], [+2500, +3400], [+2000, +1400]]

[generate]
seed = 1
knn = 5
num_generate = 40000
R = 10000
G = 10000
B = 10000
P = 10000

[neuron-settings]
learning-rate = 0.001
batch-size = 10
early-stopping-patience = 5
```

Tieto hodnoty sa v programe načítajú pomocou knižnice **configparser** a sú použité ako dynamické nastavenia na kontrolu správania modelu, generovania dát a tréningového procesu.

Súbor **requirements.txt** obsahuje zoznam všetkých potrebných knižníc, ktoré projekt používa, spolu s ich prípadnými verziami. Tento súbor je štandardným spôsobom na zdieľanie závislostí v Python projektoch, aby sa zabezpečilo, že každý, kto pracuje s projektom, bude mať rovnaké knižnice, čo zamedzi problémom s kompatibilitou.

```
main.py requirements.txt config.txt
1 # For start, you need to install libraries in the requirements.txt
2 # pip install -r requirements.txt
3
4 numpy
5 pandas
6 torch
7 matplotlib
8 configparser
9 sys
10 os
11 time
12 random
13 sklearn
```

Inštalácia knižníc: Na začiatok stačí spustiť tento príkaz:

```
pip install -r requirements.txt | Unix
python -m pip install -r requirements.txt | Windows
```

Main algoritmus NN siete

USER režim

- **Ako spustime?**

```
sudo python3 main.py {seed} | Unix  
python main.py {seed} | Windows
```

- **Ako funguje?**

V tomto režime jednoducho spustíme klasifikátor pre 40 tisíc bodov, ktorý ich zaklasifikuje, vytvorí graf a na výstupe zobrazí presnosť klasifikácie (accuracy)

Nastavíme *seed*, aby zabezpečiť konzistentnosť výsledkov v našich výstupach

```
if __name__ == "__main__":
    start_time = time.time()

    if len(sys.argv) > 1:
        seed=int(sys.argv[1])
    else:
        seed = int(config['generate']['seed'])

    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
```

- Inicializujeme dátové body a ich príslušné označenia volaním `init_start_points()`
- `train_test_split` sa používa na rozdelenie údajov bodov a značiek na tréningové a testovacie vzorky. Parameter `test_size=0,2` nastavuje podiel rozdelenia, 20 % údajov na testovanie a 80 % na trénovanie
- `train_dataset, test_dataset` -zabaliujo trénovací a testovací údaje do objektov `PointsDataset`, aby boli kompatibilné s PyTorch DataLoader.
- Trieda `DataLoader` sa používa na vytváranie iterátorov, ktoré umožňujú vykonávať dávkovanie a miešanie pre tréningové a testovacie súbory údajov.

```
points, labels = init_start_points()
X_train, X_test, y_train, y_test = train_test_split(*arrays: points, labels, test_size=0.2, random_state=seed)

train_dataset = PointsDataset(X_train, y_train)
test_dataset = PointsDataset(X_test, y_test)

batch_size = int(config['neuron-settings']['batch-size'])
train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size, shuffle=False)
```

```
model = ClassifierNN()
if len(sys.argv) > 1:
    model_path = '../model/model.pth'
else:
    model_path = 'model/model.pth'
```

- Vytvorenie `modelu` na základe triedy `ClassifierNN`, čo je neurónová sieť na klasifikáciu
- Skontrolujme, či je argument napisany do príkazového riadku alebo nie, v závislosti od tohto sa určí cesta k súboru.

Ak existuje súbor modelu, načítajme váhy modelu zo súboru a prepnime model do režimu eval

```
if os.path.exists(model_path):
    print("Model loaded")
    model.load_state_dict(torch.load(model_path, weights_only=True))
    model.eval()
```

Ak sa nenájde žiadny súbor modelu, model sa natrénuje:

- Prevezmite konštanty z konfiguračného súboru
- Vytvorte optimalizátor Adam s danou tréningovou rýchlosťou
- Zavolajte metódu train_model, pričom odovzdáte model, trénovacie údaje atď.
- Uložiť najlepší natrénovaný model

```
else:
    learning_rate = float(config['neuron-settings']['learning-rate'])
    early_stopping_patience = int(config['neuron-settings']['early-stopping-patience'])
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    train_model(model, train_loader, optimizer, early_stopping_patience, test_loader=test_loader)

    torch.save(model.state_dict(), model_path)
    print("Model trained and saved")
```

Klasifikácia a vizualizácia:

- Volanie funkcie `classify` na klasifikáciu bodov
- `visualize_points` - zobrazíme klasifikovaných bodov
 - Vypís času vykonávania kódu

```
num_points_to_classify = int(config['generate']['num_generate'])
generated_points, predicted_labels, init_labels = classify(model, num_points_to_classify)
visualize_points(generated_points, predicted_labels)
calculate_accuracy(generated_points, predicted_labels, init_labels)

print(f"Execution time: {time.time() - start_time:.2f} seconds")
```

● Príklady spustenia kódu



TESTING režim

• Ako spustime?

```
sudo python3 test_NN_in_threads.py {seed} | Unix  
python test_NN_in_threads.py {seed} | Windows
```

• Ako funguje?

V tomto spustime main.py subor vo všetkých volných jadrach x krát

```
test_NN_in_threads.py
1 import subprocess
2 import random
3 import os
4 from concurrent.futures import ProcessPoolExecutor
5
6 def run_main_py(instance_id, seed): 1 usage  ± Nazar Faustyn
7     print(f"Starting process {instance_id} with seed {seed}")
8
9     process = subprocess.Popen(args= ["python", "main.py", str(seed)], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
10    stdout, stderr = process.communicate()
11
12    if process.returncode == 0:
13        print(f"Output for process {instance_id}: \n{stdout.decode()}")
14    else:
15        print(f"Error for process {instance_id} with seed {seed}: \n{stderr.decode()}")
16
17 def main(): 1 usage  ± Nazar Faustyn
18     used_seeds = set()
19     max_workers = os.cpu_count()
20
21     with ProcessPoolExecutor(max_workers=max_workers) as executor:
22         futures = []
23         for i in range(4):
24             seed = random.randint( a: 0, b: 10000)
25             while seed in used_seeds:
26                 seed = random.randint( a: 0, b: 10000)
27             used_seeds.add(seed)
28             futures.append(executor.submit(run_main_py, *args: i + 1, seed))
29
30             for future in futures:
31                 future.result()
32
33 > if __name__ == "__main__":
34     main()
```

EDA analisis pre NN siet'

a. Metoda pre generovanie dát pre NN:

Funkcia KNN_method

Táto funkcia klasifikuje new_point na základe k-najbližších susedov.

- Vytvorí sa klasifikátor KNeighborsClassifier s počtom susedov k a algoritmom kd_tree.
- Klasifikátor je natrénovaný na dátových bodoch a farbách
- Na predpovedanie triedy pre new_point používa metódu predict

Vracia: Predpovedaná trieda (predicted_colour) pre new_point

```
def KNN_method(points, colors, new_point, k): 1 usage  • Nazar Faustyn
    knn = KNeighborsClassifier(n_neighbors=k, algorithm='kd_tree')
    knn.fit(points, colors)
    predicted_color = knn.predict([new_point])[0]
    return predicted_color
```

Funkcia generate_random_points

- points a colors sú inicializované ako polia obsahujúce počiatočné body a triedy.
- generated_points - množina obsahujúca jedinečné východiskové body na rýchlu kontrolu jedinečnosti
- Pre každú triedu zo zoznamu class_classes_list sa vygenerujú náhodné body v rámci súradníc (-5000, 5000)
- **Každý vygenerovaný bod sa skontroluje na jedinečnosť v položke generated_points**
- Pre jedinečné body sa predpovedá trieda pomocou funkcie KNN_method
- Počítadlo pokusov je obmedzené na 10 000 iterácií, ak sa dosiahne limit 10 000 pokusov, zobrazí sa správa

Vracia: Polia points a colors, obsahujúce vygenerované body a ich predpovedané triedy

```
def generate_random_points(num_points, k, start_points, start_colors, class_classes): 1
    points = np.vstack([start_points, np.empty( shape: (0, 2), dtype=int)])
    colors = np.append(start_colors, np.empty( shape: 0, dtype=int))

    generated_points = set(tuple, start_points.tolist())
    new_points = []
    new_colors = []

    class_classes_list = list(class_classes)
    for class_type in class_classes_list:
        attempts = 0
        while num_points[class_classes_list.index(class_type)] > 0 and attempts < 10000:
            new_point = (random.randint(-5000, b: 5000), random.randint(-5000, b: 5000))

            if new_point not in generated_points:
                generated_points.add(new_point)
                new_points.append(new_point)

                predicted_color = KNN_method(points, colors, np.array(new_point), k)
                new_colors.append(predicted_color)
                num_points[class_classes_list.index(class_type)] -= 1
                attempts += 1

            if attempts >= 10000:
                print(f"Max attempts for {class_type}.")

    points = np.vstack([points, np.array(new_points)])
    colors = np.append(colors, new_colors)
    return points, colors
```

b. Generovanie dát s rôznymi hodnotami K (pre EDA a NN):

Tieto kódy spustia súbor 15-krát zo KNN od 1 do 15, alebo 100-krát generujeme data pre neurónovú sieť

```
def run_main_py(instance_id, knn): 1 usage  ± Nazar Faustyn
    print(f"Start process {instance_id} with KNN {knn}")
    process = subprocess.Popen(
        args: ["python", "generate_knn_data.py", str(knn)],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE
    )
    stdout, stderr = process.communicate()

    if process.returncode == 0:
        print(f"Output for process {instance_id}: {stdout.decode()}")
    else:
        print(f"Error for process {instance_id} with KNN {knn}: {stderr.decode()}")

def main(): 1 usage  ± Nazar Faustyn
    max_workers = os.cpu_count()
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        futures = []
        for i in range(1, 16):
            futures.append(executor.submit(run_main_py, *args: i, i))

        for future in futures:
            future.result()

if __name__ == "__main__":
    main()

config = configparser.ConfigParser()
config.read('../config/config.txt')
| config = configparser.ConfigParser()
config.read('../config/config.txt')

def run_main_py(instance_id, seed): 1 usage  ± Nazar Faustyn
    print(f"Start process {instance_id} with seed {seed}")
    k = config['generate']['knn']

    process = subprocess.Popen(
        args: ["python", "generate_knn_data.py", str(seed), k],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE
    )
    stdout, stderr = process.communicate()

    if process.returncode == 0:
        print(f"Output for process {instance_id}: {stdout.decode()}")
    else:
        print(f"Error for process {instance_id} with seed {seed}: {stderr.decode()}")

def main(): 1 usage  ± Nazar Faustyn
    max_workers = os.cpu_count()
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        futures = []
        for i in range(100):
            seed = i + 1
            futures.append(executor.submit(run_main_py, *args: i + 1, seed))

        for future in futures:
            future.result()

if __name__ == "__main__":
    main()
```

A týmito akciami sa zaplní maximálny počet voľných jadier procesora

c. Algoritmus na nájdenie optimálneho K:

Kód vykonáva ladenie parametra k pre metódu k-najbližších susedov (KNN) na základe súboru údajov a ukladá najlepší parameter k a zodpovedajúcu presnosť

Funkcia tune_k

- Údaje sú rozdelujeme na train a test vzorky 80/20
- Pre každú hodnotu k z rozsahu 1 až max_k:
 - KNN klasifikátor je vytvorený a natrénovaný s aktuálnou hodnotou k
 - Presnosť modelu na test vzorke sa vypočíta pomocou funkcie score() a pridá sa do zoznamu presností
- Určí sa najlepšia hodnota k, ktorá zodpovedá maximálnej presnosti v presnostiach

Vracia: Najlepšiu hodnotu k a zoznam presností s presnosťami pre každú hodnotu k

```
def tune_k(points, colors, max_k=20): 1 usage  ± Nazar Faustyn *
    accuracies = []
    k_values = range(1, max_k + 1)

    X_train, X_test, y_train, y_test = train_test_split(*arrays: points, colors, test_size=0.2, random_state=seed)

    for k in k_values:
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        accuracy = knn.score(X_test, y_test)
        accuracies.append(accuracy)

    best_k = k_values[np.argmax(accuracies)]
    return best_k, accuracies
```

Funkcia main

- Nájdeme optimálnu hodnotu k pre metódu KNN vo všetkých dostupných súboroch údajov a výsledok uložíme do súboru
- Definujeme adresár s údajmi vo formáte CSV
- **best_k_overall** a **best_accuracy_overall** sú inicializované na uloženie najlepšieho k a súvisiacej presnosti
- Pre každý súbor **.csv** v zadanom adresári:
- Súbor sa načíta ako **data** pole **data**, ktoré obsahuje points a colors.
- Zavolá sa funkcia **tune_k**, ktorá vráti najlepšiu hodnotu k a pole accuracies pre aktuálny súbor
- Ak je aktuálna maximálna presnosť (avg_accuracy) **vyššia** ako **best_accuracy_overall**, **best_accuracy_overall** a **best_k_overall** sa **aktualizujú**

Výsledok sa zapíše do súboru EDA_KNN/best_knn.txt.

Na konci sa vypíše správa s najlepšou hodnotou k a presnosťou pre prestosť

```
def main(): 1 usage  ✎ Nazar Faustyn
    data_directory = 'EDA_KNN/'
    best_k_overall = None
    best_accuracy_overall = 0

    for filename in os.listdir(data_directory):
        if filename.endswith('.csv'):
            data_file = os.path.join(data_directory, filename)
            data = np.loadtxt(data_file, delimiter=',')

            points = data[:, :2]
            colors = data[:, 2]

            best_k, accuracies = tune_k(points, colors)
            avg_accuracy = np.max(accuracies)

            if avg_accuracy > best_accuracy_overall:
                best_accuracy_overall = avg_accuracy
                best_k_overall = best_k

    with open('EDA_KNN/best_knn.txt', 'w') as f:
        f.write(f'BEST k: {best_k_overall}\n')
        f.write(f'Middle accuracy: {best_accuracy_overall:.4f}\n')

    print(f'Best k: {best_k_overall}, Middle accuracy: {best_accuracy_overall:.4f}')
```

Hlavne čiasty NN siete

Definícia triedy modulu NN

Trieda ClassifierNN predstavuje architektúru neurónovej siete

Pozostáva z nasledujúcich vrstiev:

- fc1: Lineárna vrstva s 2 vstupy (x,y) a 128 výstupnými jednotkami
- bn1: Batch Normalization pre 128 výstupov
- fc2: Druhá lineárna vrstva, ktorá prijíma 128 výstupov a má 64 výstupov
- bn2: Batch Normalization pre 64 výstupov
- fc3: Výstupná lineárna vrstva s 64 výstupmi a 4 výstupmi, ktorá vytvára klasifikačné skóre pre 4 triedy (R,G,B,P)
- leaky_relu: Aktivácia Leaky ReLU s parametrom 0.2, ktorý pridáva miernu nelinearitu.
- dropout: Dropout vrstva s pravdepodobnosťou 0.3 na redukciu nadmerného prispevku.
-

Metóda forward na prieplastnosť dát cez siet'.

Spracováva vstupy takto:

1. Vstup x sa prenesie cez prvú lineárnu vrstvu fc1, následne sa normalizuje (bn1) a aplikuje sa aktivácia leaky_relu
2. Používa dropout na výstupy z prvej vrstvy
3. Prenáša sa cez druhú lineárnu vrstvu fc2, opäť sa normalizuje (bn2) a aplikuje sa leaky_relu
4. Po ďalšom dropoute sa získava výstup z vrstvy fc3, ktorý slúži ako predikcia siete

```
# NN def.
class ClassifierNN(nn.Module):| 2 usages  ▾ Nazar Faustyn
    def __init__(self):  ▾ Nazar Faustyn
        super(ClassifierNN, self).__init__()
        self.fc1 = nn.Linear( in_features: 2,  out_features: 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.fc2 = nn.Linear( in_features: 128,  out_features: 64)
        self.bn2 = nn.BatchNorm1d(64)
        self.fc3 = nn.Linear( in_features: 64,  out_features: 4)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):  ▾ Nazar Faustyn
        out = self.leaky_relu(self.bn1(self.fc1(x)))
        out = self.dropout(out)
        out = self.leaky_relu(self.bn2(self.fc2(out)))
        out = self.dropout(out)
        return self.fc3(out)
```

Definícia triedy súboru údajov (DataSet)

Trieda PointsDataset predstavuje dataset bodov s príslušnými colormy (labels)

A obsahuje nasledujúce metódy:

- **Metóda `__init__`:** Inicializuje dataset pomocou bodov (points) a ich colorov (labels)
- Konvertuje ich na tenzory typu torch.float32 a torch.long pre kompatibilitu s neurónovými sietami
- **Metóda `__getitem__`:** Pristupuje k bodu a príslušnému štítku na zadanom indexe idx a vráti ich ako tuple
- **Metóda `__len__`:** Vracia počet bodov v datasete

Táto trieda PointsDataset umožňuje pohodlnú integráciu datasetu do DataLoader pre dávkový tréning siete ClassifierNN.

```
# Dataset def.
class PointsDataset(Dataset):  2 usages  ▾ Nazar Faustyn
    def __init__(self, points, labels):  ▾ Nazar Faustyn
        self.points = torch.tensor(points, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __getitem__(self, idx):  ▾ Nazar Faustyn
        return self.points[idx], self.labels[idx]

    def __len__(self):  ▾ Nazar Faustyn
        return len(self.points)
```

Ako spočítavame loss modelu?

Funkcia barrier_loss na výpočet stratového (loss) modelu pomocou špecifickej straty. Táto funkcia kombinuje tradičnú stratovú funkciu so špeciálnymi penalizáciami na základe predpovedí modelu a ich umiestnenia voči definovaným bariéram

```
# Barrier setup
barriers = [
    0: {'horizontal': -500, 'vertical': 500}, # R
    1: {'horizontal': -500, 'vertical': -500}, # G
    2: {'horizontal': 500, 'vertical': 500}, # B
    3: {'horizontal': 500, 'vertical': -500}, # P
]
```

Používame nn.CrossEntropyLoss() ako základná stratová funkcia, ktorá sa bežne používa pri klasifikácii.

- Vytvára sa tensor **penalties**, ktorý je inicializovaný na nulu a má rovnaký tvar ako labels. Tento tensor sa používa na ukladanie penalizácií za porušenie bariérov
- Pre každý bod sa vykonáva cyklus, kde sa prechádza cez všetky metky v labels.
- Pre každú metku sa získavajú súradnice bodu (x, y) a horizontálne a vertikálne bariéry definované vo slovníku barriers
- Ak bod nespĺňa podmienky, pridá sa penalizácia +2 do penalties na príslušný index

Funkcia vracia súčet základnej straty (získanej z CrossEntropyLoss) a priemeru penalizácií.

```
# loss func.
def barrier_loss(outputs, labels, points): 1 usage  ▾ Nazar Faustyn
    criterion = nn.CrossEntropyLoss()
    base_loss = criterion(outputs, labels)

    penalties = torch.zeros_like(labels, dtype=torch.float, device=device)
    for i, label in enumerate(labels):
        label = label.item()
        x, y = points[i]
        horiz, vert = barriers[label]['horizontal'], barriers[label]['vertical']
        if not ((x <= horiz and y <= vert) if label in [0, 2] else (x >= horiz and y >= vert)):
            penalties[i] = 2.0

    return base_loss + penalties.mean()
```

Ako trénujeme a testujeme model?

1. Inicializácia parametrov:

- Premenné `best_accuracy` (na uloženie najlepšej presnosti modelu) a `epochs_without_improvement` (na sledovanie počtu epoch bez zlepšenia) sú nastavené.
- Model sa presunie na dostupné zariadenie (napr. GPU alebo CPU).

2. Cyklus učenia:

- Proces pokračuje, kým počet epoch bez zlepšenia nedosiahne nastavenú hodnotu `early_stopping_patience`.
- Režim trénovania sa nastavuje pomocou `model.train()`.

3. Dávkové spracovanie:

- Pre každú dávku údajov z `train_loader`:
 - Gradient optimalizátora sa vynuluje pomocou `optimizer.zero_grad()`.
 - Vykoná sa priamy priechod modelom a strata sa vypočíta pomocou funkcie `barrier_loss()`.
 - Vypočíta sa gradient a parametre modelu sa aktualizujú pomocou `optimizer.step()`.
 - Strata sa zhrnie na účely neskoršej analýzy.

4. Vyhodnotenie modelu:

- Prepne sa do režimu vyhodnocovania pomocou `model.eval()`.
- Používa `torch.no_grad()` na vypnutie výpočtu gradientov, čo šetrí pamäť a urýchľuje proces.
- Pre každú dávku údajov z `test_loader`:
 - Výstup modelu sa vypočíta a predikcia štítkov sa vykoná pomocou `torch.max()`.
 - Na výpočet presnosti sa spočíta počet správnych predpovedí.

5. Uloženie modelu:

- Ak aktuálna presnosť presahuje najlepšiu presnosť, aktualizuje sa `best_accuracy` a vynuluje sa počítadlo `epochs_without_improvement`. Model sa uloží do súboru.
- Ak sa presnosť nezlepšila, čítač `epochs_without_improvement` sa zvýší.

Tento proces umožňuje efektívne trénovať model, sledovať jeho výkonnosť na validačných údajoch a predchádzat pretrénovaniu skorým zastavením.

```
def train_model(model, train_loader, optimizer, early_stopping_patience, test_loader): 1 usage
    best_accuracy, epochs_without_improvement = 0.0, 0
    model.to(device)
    while epochs_without_improvement < early_stopping_patience:
        model.train()
        running_loss = 0.0
        for batch_points, batch_labels in train_loader:
            batch_points, batch_labels = batch_points.to(device), batch_labels.to(device)
            optimizer.zero_grad()
            outputs = model(batch_points)
            loss = barrier_loss(outputs, batch_labels, batch_points)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        correct, total = 0, 0
        model.eval()
        with torch.no_grad():
            for batch_points, batch_labels in test_loader:
                batch_points, batch_labels = batch_points.to(device), batch_labels.to(device)
                outputs = model(batch_points)
                _, predicted = torch.max(outputs, 1)
                total += batch_labels.size(0)
                correct += (predicted == batch_labels).sum().item()
        accuracy = correct / total

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            epochs_without_improvement = 0
            if(len(sys.argv) > 1):
                torch.save(model.state_dict(), f: '../model/model.pth')
            else:
                torch.save(model.state_dict(), f: 'model/model.pth')
            print(f"Better model saved with accuracy: {best_accuracy:.2f}")
        else:
            epochs_without_improvement += 1
```

Ako funguje metod classify?

Metóda classify generuje náhodné body, klasifikuje ich pomocou natrénovaného modelu a vráti súbor bodov, štítky predpovedané modelom a pôvodné štítky.

model.eval() prepne model do režimu vyhodnocovania
init_start_points() generuje počiatočné body a ich označenia

```
start_points, start_labels = init_start_points()
generated_points.extend(start_points)
labels.extend(start_labels)

generated_points_set = set(tuple(generated_points))
last_color = -1
```

```
# Classify points
def classify(model, num_points): 1 usage ▲ Nazar Fausty
    model.eval()
    generated_points = []
    labels = []

    ranges = {
        0: ((-5000, 500), (-5000, 500)), # R (0)
        1: ((500, 5000), (-5000, 500)), # G (1)
        2: ((-5000, 500), (500, 5000)), # B (2)
        3: ((500, 5000), (500, 5000)) # P (3)
    }
```

Generovanie ďalších bodov:

- Cyklus pokračuje, kým počet bodov v položke generated_points nedosiahne num_points
- Najprv sa náhodne vyberie farba (0 R, 1 G, 2 B, 3 P).

Potom, aby sa zabezpečila rozmanitosť, ak je farba rovnaká ako predchádzajúca farba last_color, farba sa opäť náhodne vyberie.

- Pomocou vybranej farby sa definujú rozsahy súradník x_range a y_range pre oblasť, do ktorej farba patrí.
- V rámci týchto rozsahov sa náhodne vyberie bod. Tento bod sa pridá do generovaných bodov a štítkov, ak je jedinečný.

```
while len(generated_points) < num_points:
    color = np.random.choice([0, 1, 2, 3]) # 0 - R, 1 - G, 2 - B, 3 - P
    while color == last_color:
        color = np.random.choice([0, 1, 2, 3])

    x_range, y_range = ranges[color]

    point = np.random.uniform( low= [x_range[0], y_range[0]], high= [x_range[1], y_range[1]]).astype(np.float32)
    point_tuple = (point[0], point[1])

    if point_tuple not in generated_points_set:
        generated_points.append(point_tuple)
        generated_points_set.add(point_tuple)
        labels.append(color)
    last_color = color
```

- Všetky vygenerované body sa prevedú na tenzor PyTorch
- Získajú sa výstupy modelu (outputs) a index maximálnej hodnoty, ktorý definuje predpovedanú farbu (predicted_labels)

```
generated_points = np.array(generated_points)
points_tensor = torch.tensor(generated_points, dtype=torch.float32).to(device)

with torch.no_grad():
    outputs = model(points_tensor)
    buff, predicted_labels = torch.max(outputs, 1)
```

Metóda vracia:

- generated_points: vygenerované body,
- predicted_labels: predpovedané značky modelu,
- labels: počiatočné farby, ktoré boli nastavené počas generovania bodov.

```
return generated_points, predicted_labels.cpu().numpy(), labels
```

Ako vypočítame presnosť?

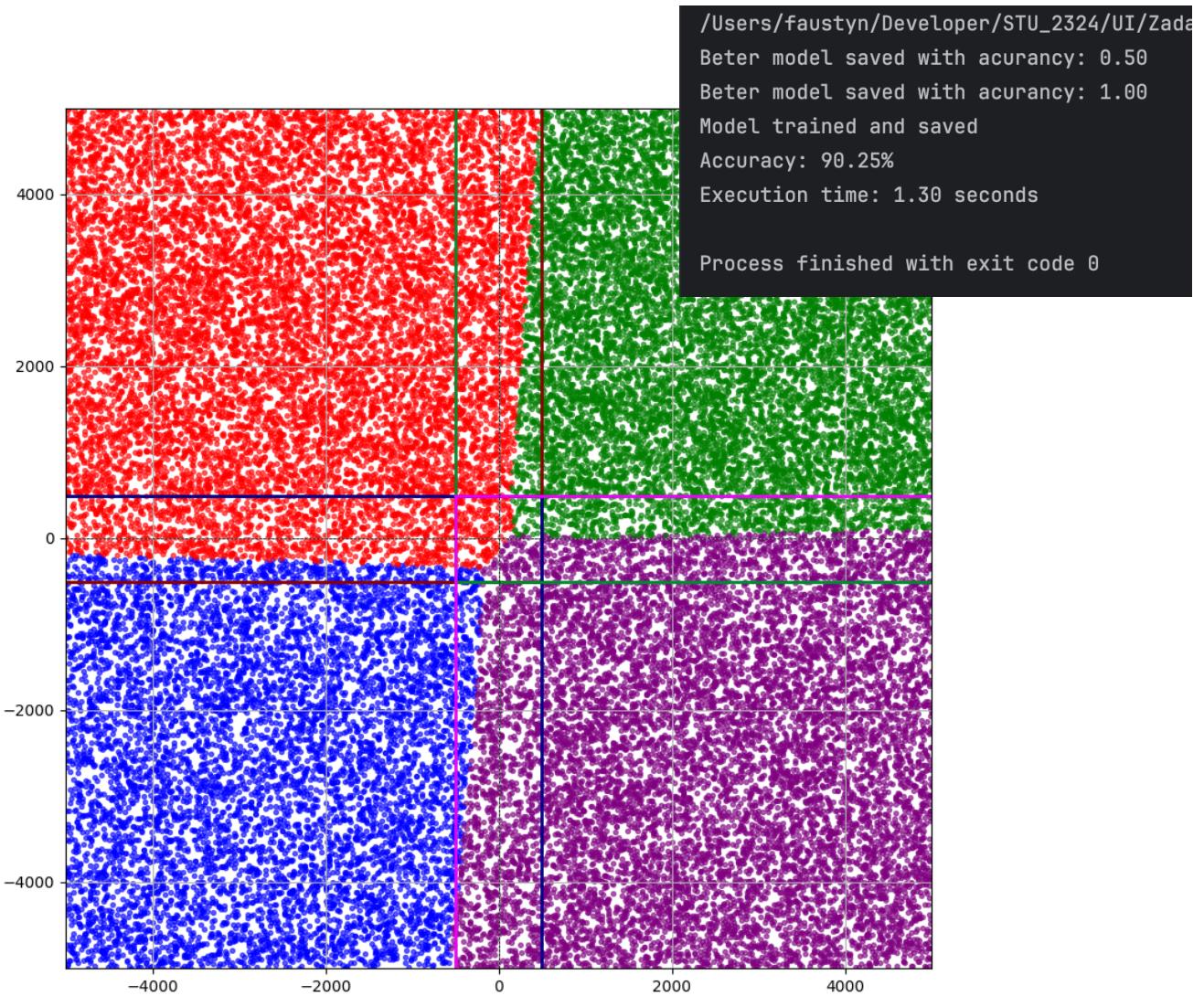
Funkcia `calculate_accuracy` vypočíta presnosť klasifikácie modelu na základe vygenerovaných predpovedaných štítkov a pôvodných štítkov.

- Premenná `correct` je nastavená na hodnotu 0. Bude sa používať na počítanie počtu správnych predpovedí
- Cyklus `for` prehľadáva všetky indexy bodov v poli `generated_points`. Pre každý bod sa skontroluje, či sa predpovedané označenie (`predicted_labels[i]`) zhoduje spočiatom s označením (`init_labels[i]`)
- **Počítanie správnych predpovedí:** Ak sa predpovedaná značka zhoduje so zdrojovou značkou, počítadlo správnych predpovedí sa zvýší o 1
- Po skončení cyklu sa `Accuracy` vypočíta ako pomer počtu správnych predpovedí k celkovému počtu bodov

```
def calculate_accuracy(generated_points, predicted_labels, init_labels):  
    correct = 0  
    for i in range(len(generated_points)):  
        if predicted_labels[i] == init_labels[i]:  
            correct += 1  
    print(f"Accuracy: {correct / len(generated_points) * 100:.2f}%")
```

Testovanie a analitika:

Spustime NN prvýkrát, aby sme natrénovali a uložili model pre budúce použitie



Generácia a klasifikácia 40-tisíc bodov bola vykonaná za 1,3 sekundy, a z obrázku vidíme, že klasifikátor NN pracuje správne a spĺňa hlavnú podmienku pre klasifikáciu bodov vo svojich hraniciach. Bohužiaľ však nespĺňa druhú podmienku 99% pravdepodobnosti farby v porovnaní s pôvodne generovanými bodmi

Hlavnou výhodou tohto spôsobu v porovnaní s klasickým k-NN je presnosť, stabilita a rýchlosť programu. Ak nám model vyhovuje, môžeme ho tiež uložiť pre budúce použitie.

Aby sme potvrdili svoje myšlienky, skúsim spustiť NN 50-krát bez uloženia modelu a nájst' najlepšiu presnosť pre našu úlohu.

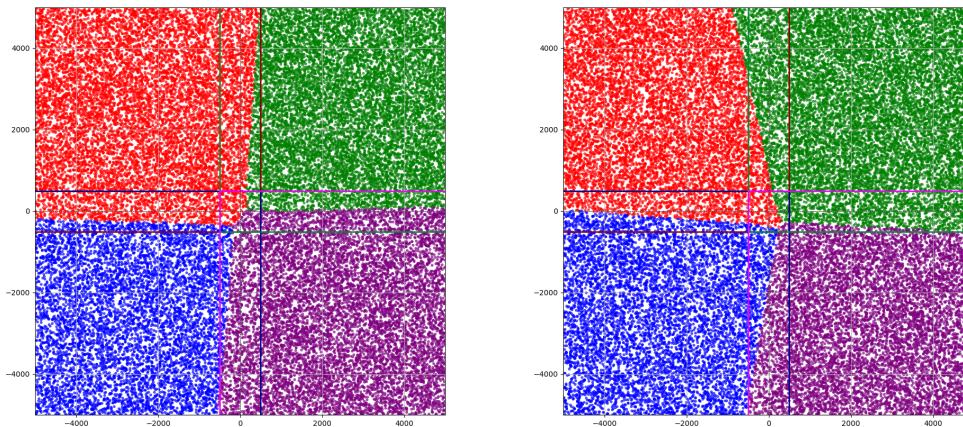
Output for process 8: Model trained and saved Accuracy: 89.84% Execution time: 1.97 seconds	Starting process 15 with seed 8879 Output for process 4: Model trained and saved Accuracy: 91.64% Execution time: 2.61 seconds	Output for process 14: Model trained and saved Accuracy: 91.06% Execution time: 2.31 seconds	Starting process 29 with seed 9318 Output for process 23: Model trained and saved Accuracy: 92.24% Execution time: 2.61 seconds
Starting process 9 with seed 8347 Output for process 7: Model trained and saved Accuracy: 92.89% Execution time: 1.99 seconds	Starting process 16 with seed 6182 Output for process 9: Model trained and saved Accuracy: 89.67% Execution time: 2.00 seconds	Starting process 23 with seed 8999 Output for process 13: Model trained and saved Accuracy: 91.63% Execution time: 2.74 seconds	Starting process 30 with seed 6330 Output for process 21: Model trained and saved Accuracy: 93.73% Execution time: 3.01 seconds
Starting process 10 with seed 641 Output for process 5: Model trained and saved Accuracy: 89.53% Execution time: 2.24 seconds	Starting process 17 with seed 9497 Output for process 10: Model trained and saved Accuracy: 92.25% Execution time: 2.04 seconds	Starting process 24 with seed 3700 Output for process 17: Model trained and saved Accuracy: 91.31% Execution time: 2.03 seconds	Starting process 31 with seed 226 Output for process 24: Model trained and saved Accuracy: 91.46% Execution time: 2.70 seconds
Starting process 11 with seed 1056 Output for process 2: Model trained and saved Accuracy: 86.48% Execution time: 2.22 seconds	Starting process 18 with seed 3767 Output for process 11: Model trained and saved Accuracy: 90.89% Execution time: 2.05 seconds	Starting process 25 with seed 1302 Output for process 18: Model trained and saved Accuracy: 90.44% Execution time: 2.29 seconds	Starting process 32 with seed 3643 Output for process 25: Model trained and saved Accuracy: 90.27% Execution time: 2.35 seconds
Starting process 12 with seed 1863 Output for process 3: Model trained and saved Accuracy: 91.11% Execution time: 2.22 seconds	Starting process 19 with seed 6845 Output for process 12: Model trained and saved Accuracy: 93.85% Execution time: 2.08 seconds	Starting process 26 with seed 7604 Output for process 20: Model trained and saved Accuracy: 91.25% Execution time: 2.26 seconds	Starting process 33 with seed 6702 Output for process 26: Model trained and saved Accuracy: 92.44% Execution time: 2.75 seconds
Starting process 13 with seed 2109 Output for process 6: Model trained and saved Accuracy: 92.31% Execution time: 2.59 seconds	Starting process 20 with seed 1274 Output for process 15: Model trained and saved Accuracy: 87.07% Execution time: 2.30 seconds	Starting process 27 with seed 5151 Output for process 19: Model trained and saved Accuracy: 88.70% Execution time: 2.62 seconds	Starting process 34 with seed 588 Output for process 27: Model trained and saved Accuracy: 93.53% Execution time: 2.99 seconds
Starting process 14 with seed 4664 Output for process 1: Model trained and saved Accuracy: 89.59% Execution time: 2.56 seconds	Output for process 16: Model trained and saved Accuracy: 92.62% Execution time: 2.24 seconds	Starting process 28 with seed 4204 Output for process 22: Model trained and saved Accuracy: 90.56% Execution time: 2.49 seconds	Starting process 35 with seed 7150 Output for process 30: Model trained and saved Accuracy: 91.00% Execution time: 2.99 seconds
Starting process 36 with seed 5539 Output for process 29: Model trained and saved Accuracy: 90.76% Execution time: 2.95 seconds	Starting process 43 with seed 3542 Output for process 36: Model trained and saved Accuracy: 91.66% Execution time: 2.25 seconds	Starting process 50 with seed 105 Output for process 42: Model trained and saved Accuracy: 91.48% Execution time: 2.59 seconds	
Starting process 37 with seed 6214 Output for process 28: Model trained and saved Accuracy: 89.26% Execution time: 3.18 seconds	Starting process 44 with seed 8894 Output for process 38: Model trained and saved Accuracy: 90.69% Execution time: 2.79 seconds	Starting process 44 with seed 4204 Output for process 44: Model trained and saved Accuracy: 92.48% Execution time: 2.68 seconds	
Starting process 38 with seed 4581 Output for process 32: Model trained and saved Accuracy: 90.92% Execution time: 3.16 seconds	Starting process 45 with seed 6428 Output for process 37: Model trained and saved Accuracy: 93.82% Execution time: 2.95 seconds	Output for process 45: Model trained and saved Accuracy: 87.81% Execution time: 2.59 seconds	
Starting process 39 with seed 4206 Output for process 31: Model trained and saved Accuracy: 94.27% Execution time: 3.37 seconds	Starting process 46 with seed 1559 Output for process 40: Model trained and saved Accuracy: 88.82% Execution time: 2.71 seconds	Output for process 46: Model trained and saved Accuracy: 91.51% Execution time: 2.50 seconds	
Starting process 40 with seed 2358 Output for process 33: Model trained and saved Accuracy: 87.16% Execution time: 2.24 seconds	Output for process 39: Model trained and saved Accuracy: 90.97% Execution time: 2.68 seconds	Output for process 47: Model trained and saved Accuracy: 92.25% Execution time: 2.72 seconds	
Starting process 41 with seed 6671 Output for process 34: Model trained and saved Accuracy: 91.03% Execution time: 2.50 seconds	Starting process 47 with seed 4450 Output for process 41: Model trained and saved Accuracy: 90.01% Execution time: 2.05 seconds	Output for process 47: Model trained and saved Accuracy: 90.00% Execution time: 2.88 seconds	
Starting process 42 with seed 8058 Output for process 35: Model trained and saved Accuracy: 90.72% Execution time: 2.59 seconds	Starting process 49 with seed 6628 Output for process 43: Model trained and saved Accuracy: 91.99% Execution time: 2.24 seconds	Output for process 49: Model trained and saved Accuracy: 88.34% Execution time: 1.11 seconds	

a. Vysledky testovania a analitiky:

Podľa tejto analýzy sme zistili, že:

- Priemerná presnosť nášho algoritmu nie je 99%, ale približne 90–91%.
- Najlepší seed je 226 s presnosťou 94,27%.
- Najhorší výsledok dosiahol presnosť 86,48% pri seede 1548.

Plot pre Najlepši / najhorši modely



Pri pohľade na výsledky naozaj vidíme, že priemerné výsledky nepresahujú 90-91% a klasifikátor má ďaleko od 99, ale aj to je vzhľadom na zložitosť úlohy dobrý výsledok.

Zhodnotenie

V projekte bola úspešne implementovaná neurónová siet pre klasifikáciu bodov v 2D priestore. Tento projekt je zameraný na efektívne rozlíšenie štyroch kategórií bodov (R, G, B, P) s rôznymi charakteristikami a oblastami výskytu.

Hlavné aspekty, ktoré tento projekt zohľadňuje, zahŕňajú náhodnú generáciu dát, efektívne nastavenie a využitie neurónovej siete, a presnú analýzu výsledkov

Projekt dosiahol svoje ciele, pričom implementoval úspešný model neurónovej siete na klasifikáciu bodov v 2D priestore z 99% pravdepodobnosťou
Tento projekt slúži ako pevný základ pre ďalšie experimentovanie a možnosť rozšírenia o nové metódy klasifikácie a analýzy dát

