Міністерство освіти і науки України
Львівський національний університет ім. Івана Франка
Факультет електроніки та комп’ютерних технологій

Звіт
Про виконання лабораторної роботи № 1
з курсу “Інтеграція інформаційних систем”
Розгортання LM моделей

Виконав:
Студент групи ФЕІ-45
Місюра Володимир
Перевірив:
Асистент Прокопів Р.В.

Львів 2025

Мета лабораторної роботи – ознайомлення з процесом створення та інференсу моделей машинного навчання для обробки аудіо даних.

1. Файл model.py - модель нейромережі для розпізнавання голосових команд

У цьому файлі я реалізував модель SpeechCommandCNN, яка отримує мел-спектрограму аудіо й визначає, яку команду вимовили: yes, no, up або down. Модель складається з кількох згорткових шарів і простого класифікатора, і використовується в усіх частинах проєкту - під час навчання, тестування та інференсу.

```python
import torch                    # Імпорт бібліотеки PyTorch для роботи з тензорами
import torch.nn as nn           # Імпорт модулю для створення нейромережевих шарів
import torch.nn.functional as F  # Імпорт функцій активації та інших функцій


# === 1. Архітектура моделі ===
class SpeechCommandCNN(nn.Module):          # Оголошення класу моделі  10 usages
    def __init__(self, num_classes=4):      # Конструктор класу, параметр — кількість класів
        super(SpeechCommandCNN, self).__init__()  # Виклик конструктора батьківського класу

        # Вхідна форма спектрограми ≈ [1, 64, ~100] (1 канал, 64 мел-фільтри)
        self.conv1 = nn.Conv2d( in_channels: 1,  out_channels: 16, kernel_size=3, stride=1, padding=1)  # Перший згортковий шар
        self.bn1 = nn.BatchNorm2d(16)          # Батч-нормалізація після першої згортки
        self.pool1 = nn.MaxPool2d( kernel_size: 2,  stride: 2)       # Пулінг для зменшення розміру фіч-карти

        self.conv2 = nn.Conv2d( in_channels: 16,  out_channels: 32, kernel_size=3, stride=1, padding=1)  # Друга згортка
        self.bn2 = nn.BatchNorm2d(32)          # Нормалізація після другої згортки
        self.pool2 = nn.MaxPool2d( kernel_size: 2,  stride: 2)       # Ще один MaxPool для зменшення розміру

        self.conv3 = nn.Conv2d( in_channels: 32,  out_channels: 64, kernel_size=3, stride=1, padding=1)  # Третя згортка
        self.bn3 = nn.BatchNorm2d(64)          # Нормалізація після третьої згортки
        self.pool3 = nn.AdaptiveAvgPool2d((1, 1))  # Адаптивний пулінг до розміру 1×1

        # Вихід CNN перетворюємо у вектор
        self.fc = nn.Linear( in_features: 64, num_classes)  # Фінальний повнозв'язний шар для класифікації
```

```python
    def forward(self, x):                    # Метод forward — шлях проходження даних через модель
        # Очікуємо вхід [batch, 1, mel_bins, time]
        x = F.relu(self.bn1(self.conv1(x)))   # Перша згортка + нормалізація + ReLU
        x = self.pool1(x)                     # Пулінг

        x = F.relu(self.bn2(self.conv2(x)))   # Друга згортка + нормалізація + ReLU
        x = self.pool2(x)                     # Пулінг

        x = F.relu(self.bn3(self.conv3(x)))   # Третя згортка + нормалізація + ReLU
        x = self.pool3(x)                     # Адаптивний пулінг

        x = torch.flatten(x, 1)               # Перетворення у вектор (batch, 64)
        x = self.fc(x)                        # Класифікація через лінійний шар
        return x                              # Повертаємо логіти (оцінки для кожного класу)


# === 2. Тест (запуск напряму) ===
if __name__ == "__main__":                  # Виконується тільки при прямому запуску файлу
    model = SpeechCommandCNN(num_classes=4)   # Створення моделі
    sample = torch.randn(8, 1, 64, 100)       # Вхідний випадковий тензор (8 прикладів)
    out = model(sample)                       # Пропускаємо дані через модель
    print("Форма виходу:", out.shape)         # Виводимо форму результату
```

Результат роботи коду:

```
C:\Users\rakom\OneDrive\Desktop\7SEM\IIS\1\.venv
Форма виходу: torch.Size([8, 4])

Process finished with exit code 0
```

Після запуску файлу model.py створюється тестовий випадковий батч розміром 8 мел-спектрограм, який подається на вхід моделі. У результаті модель повертає вихідний тензор розмірності [8, 4], де 8 — кількість прикладів, а 4 — кількість класів команд. Це означає, що модель коректно збирається та успішно пропускає дані вперед.

2. Файл data_loader.py - завантаження й підготовка аудіоданих

Тут я зробив завантаження датасету Google Speech Commands і відібрав тільки ті файли, які належать до потрібних команд. У цьому файлі також виконується перетворення аудіо в мел-спектрограми та поділ даних на тренувальну й тестову частини, щоб їх можна було легко використовувати під час навчання моделі.

```python
import torch                          # Робота з тензорами та нейромережами
import torchaudio                     # Бібліотека для роботи з аудіо
from torch.utils.data import random_split, DataLoader  # Інструменти для поділу та пакетної загрузки даних
from torchaudio.transforms import MelSpectrogram, AmplitudeToDB  # Перетворення аудіо у спектрограму


# === 1. Клас обгортки для датасету з трансформацією ===
class SpeechCommandsDataset(torch.utils.data.Dataset):   # Клас, що дозволяє застосувати трансформацію до кожного елемента
    def __init__(self, dataset, transform=None):         # Приймає оригінальний датасет і трансформацію
        self.dataset = dataset                           # Зберігаємо базовий датасет
        self.transform = transform                       # Зберігаємо трансформацію (може бути None)

    def __len__(self):                                   # Повертає кількість елементів у датасеті
        return len(self.dataset)

    def __getitem__(self, idx):                          # Отримання одного елементу за індексом
        waveform, sample_rate, label, *_ = self.dataset[idx]  # Завантажуємо аудіо, частоту дискретизації та мітку

        if self.transform:                               # Якщо задана трансформація
            waveform = self.transform(waveform)          # Перетворюємо waveform у мел-спектрограму

        # === Вирівнюємо довжину спектрограми ===
        max_len = 100  # довжина по осі часу (можна змінити)

        if waveform.size(-1) < max_len:                  # Якщо спектрограма коротша за потрібну
            pad = max_len - waveform.size(-1)            # Рахуємо скільки нулів треба додати
            waveform = torch.nn.functional.pad(waveform, pad: (0, pad))  # Додаємо нульовий паддінг справа

        elif waveform.size(-1) > max_len:                # Якщо спектрограма довша
            waveform = waveform[:, :, :max_len] if waveform.ndim == 3 else waveform[:, :max_len]  # Обрізаємо

        return waveform, label                           # Повертаємо готову спектрограму і текстову мітку
```

```python
# === 2. Функція для завантаження та підготовки датасету ===
def load_data(batch_size=32):                                # Основна функція повертає два DataLoader-и  5 usages
    import os                                                # Імпорт os (використовується тільки всередині)
    os.makedirs( name: "./data", exist_ok=True)                    # Створюємо папку "data", якщо її немає

    # === Отримуємо базовий датасет ===
    dataset = torchaudio.datasets.SPEECHCOMMANDS(           # Завантаження датасету Google Speech Commands
        root="./data",                                       # Куди завантажувати
        download=True                                        # Завантажити, якщо нема
    )

    # === Обмежимось кількома класами ===
    selected_labels = ["yes", "no", "up", "down"]           # Використовуємо тільки 4 команди

    # 🔹 Фільтрація без повного зчитування аудіо
    all_items = []                                           # Список шляхів до потрібних файлів
    for wav_path in dataset._walker:                         # Проходимо по всіх шляхах датасету
        label = os.path.basename(os.path.dirname(wav_path))  # Назва папки — це мітка
        if label in selected_labels:                         # Якщо мітка нам підходить
            all_items.append(wav_path)                       # Додаємо шлях у список

    # 🔹 Створюємо новий датасет лише з потрібних файлів
    dataset._walker = all_items                              # Перезаписуємо внутрішній список датасету

    # === Трансформація аудіо → мел-спектрограма ===
    transform = torch.nn.Sequential(                         # Створюємо послідовність перетворень
        MelSpectrogram(sample_rate=16000, n_mels=64),        # Перетворюємо аудіо у мел-спектрограму
        AmplitudeToDB()                                      # Переводимо у децибели (лог-шкала)
    )
```

```python
    # === Обгортка з трансформацією ===
    processed_dataset = SpeechCommandsDataset(dataset, transform)  # Датасет з трансформацією

    # === Поділ на train/test ===
    train_size = int(0.8 * len(processed_dataset))          # 80% у тренування
    test_size = len(processed_dataset) - train_size          # 20% у тест
    train_dataset, test_dataset = random_split(processed_dataset, lengths: [train_size, test_size])  # Розділяємо

    # === DataLoader-и ===
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)   # Тренувальний loader
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)    # Тестовий loader

    return train_loader, test_loader                         # Повертаємо два готові DataLoader-и


# === 3. Перевірка (тільки якщо запускати напряму) ===
if __name__ == "__main__":                                   # Виконується тільки при прямому запуску файлу
    train_loader, test_loader = load_data()                  # Завантажуємо дані
    print("Кількість батчів для тренування:", len(train_loader))  # Виводимо кількість батчів train
    print("Кількість батчів для тесту:", len(test_loader))        # Виводимо кількість батчів test
```

Результат роботи коду:

```
C:\Users\rakom\OneDrive\Desktop\7SEM\IIS\1
Кількість батчів для тренування: 391
Кількість батчів для тесту: 98

Process finished with exit code 0
```

На скріншоті видно, що після запуску файла data_loader.py програма успішно завантажила датасет Google Speech Commands, відібрала лише команди **yes**, **no**, **up**, **down**, а також поділила дані на тренувальну та тестову частини.

У консолі виводиться:

- **Кількість батчів для тренування: 391** - це означає, що вся навчальна вибірка була розбита на 391 пакет (batch), які модель буде обробляти під час тренування.
- **Кількість батчів для тесту: 98** - це кількість пакетів у тестовій вибірці, які використовуються для перевірки якості моделі.

3. Файл train.py - навчання моделі й збереження результату

У цьому файлі я налаштував повний процес тренування: завантаження даних, створення моделі, запуск кількох епох, підрахунок точності та збереження ваг у файл saved_model/model.pth. Після навчання цей файл використовується для інференсу та тестування.

```python
import os                          # Робота з файловою системою (створення папки для моделі)
import torch                       # Основна бібліотека PyTorch
import torch.nn as nn              # Нейромережеві шари та функції втрат
import torch.optim as optim        # Оптимізатори (Adam)
from tqdm import tqdm              # Прогрес-бар для наочного відображення тренування

from data_loader import load_data  # Функція для завантаження навчальних і тестових даних
from model import SpeechCommandCNN  # Імпорт класу моделі


# === 1. Основна функція тренування ===
def train_model(epochs=5, batch_size=32, learning_rate=0.001):  1 usage
    # Пристрій (GPU якщо доступний)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Вибір CPU або GPU
    print(f"Використовується пристрій: {device}")                          # Вивід, який пристрій використовується

    # Завантаження даних
    train_loader, test_loader = load_data(batch_size=batch_size)           # Завантаження датасетів

    # Ініціалізація моделі
    model = SpeechCommandCNN(num_classes=4).to(device)                     # Створення моделі і перенесення на GPU/CPU

    # Функція втрат і оптимізатор
    criterion = nn.CrossEntropyLoss()                                      # Вибір функції втрат для класифікації
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)           # Оптимізатор Adam
```

```python
# === Тренування ===
for epoch in range(epochs):                                          # Цикл по епохах
    model.train()                                                    # Переводимо модель у режим тренування
    running_loss = 0.0                                               # Накопичувач втрат за епоху
    correct, total = 0, 0                                            # Правильні та загальні передбачення

    progress_bar = tqdm(train_loader, desc=f"Епоха {epoch+1}/{epochs}") # Прогрес-бар

    for inputs, labels in progress_bar:                              # Проходимо батчі
        # Вхідні дані у форматі [batch, 1, 64, time]
        inputs = inputs.to(device)                                   # Переносимо тензор на GPU/CPU
        labels = torch.tensor([["yes", "no", "up", "down"].index(l)  # Перетворюємо текстову мітку у індекс
                               for l in labels]).to(device)

        # Обнулення градієнтів
        optimizer.zero_grad()                                        # Скидаємо старі градієнти

        # Прямий прохід
        outputs = model(inputs)                                      # Пропускаємо батч через модель

        # Обчислення втрат
        loss = criterion(outputs, labels)                            # Обчислюємо loss
        loss.backward()                                              # Зворотне поширення помилки
        optimizer.step()                                             # Оновлення ваг моделі

        running_loss += loss.item()                                  # Додаємо поточні втрати

        # Обчислення точності
        _, predicted = torch.max(outputs.data, 1)                    # Обираємо клас з найбільшою ймовірністю
        total += labels.size(0)                                      # Загальна кількість прикладів
        correct += (predicted == labels).sum().item()                # Скільки модель вгадала правильно

        progress_bar.set_postfix(loss=loss.item(),                   # Оновлюємо прогрес-бар
                                 acc=f"{100 * correct / total:.2f}%")

    print(f"Епоха [{epoch+1}/{epochs}] - Втрата: {running_loss/len(train_loader):.4f} "
          f"| Точність: {100*correct/total:.2f}%")                  # Підсумок епохи
```

```python
    # === 2. Збереження моделі ===
    os.makedirs( name: "saved_model", exist_ok=True)                    # Створюємо папку, якщо її немає
    torch.save(model.state_dict(), f: "saved_model/model.pth")          # Зберігаємо ваги моделі
    print("\n✅ Модель збережена у saved_model/model.pth")              # Підтвердження

    # === 3. Оцінка після тренування ===
    evaluate_model(model, test_loader, device)                          # Запуск оцінки на тесті


# === 4. Функція для оцінки моделі ===
def evaluate_model(model, test_loader, device):  1 usage
    model.eval()                                                        # Режим оцінки моделі
    correct, total = 0, 0                                               # Підрахунок точних відповідей

    with torch.no_grad():                                               # Відключаємо обчислення градієнтів
        for inputs, labels in test_loader:                              # Проходимо тестові батчі
            inputs = inputs.to(device)
            labels = torch.tensor([["yes", "no", "up", "down"].index(l)
                                   for l in labels]).to(device)

            outputs = model(inputs)                                     # Передбачення моделі
            _, predicted = torch.max(outputs.data, 1)                   # Вибір класу
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    acc = 100 * correct / total                                         # Обчислення точності
    print(f"\n🎯 Точність на тестових даних: {acc:.2f}%")                # Вивід точності
    return acc                                                          # Повертаємо точність


# === 5. Точка входу ===
if __name__ == "__main__":                                              # Запуск файлу напряму
    train_model(epochs=3, batch_size=32, learning_rate=0.001)           # Старт процесу тренування
```

Результат роботи коду:

```
C:\Users\rakom\OneDrive\Desktop\7SEM\IIS\1\.venv\Scripts\python.exe C:\Users\rakom\One
Використовується пристрій: cpu
Епоха 1/3:   0%|          | 0/391 [00:00<?, ?it/s]C:\Users\rakom\OneDrive\Desktop\7SEM
  warnings.warn(
Епоха 1/3: 100%|██████████| 391/391 [05:05<00:00,  1.28it/s, acc=61.35%, loss=0.753]
Епоха 2/3:   0%|          | 0/391 [00:00<?, ?it/s]Епоха [1/3] - Втрата: 1.0271 | Точні
Епоха 2/3: 100%|██████████| 391/391 [00:39<00:00,  9.86it/s, acc=76.20%, loss=0.953]
Епоха [2/3] - Втрата: 0.7078 | Точність: 76.20%
Епоха 3/3: 100%|██████████| 391/391 [00:40<00:00,  9.71it/s, acc=81.89%, loss=0.639]
Епоха [3/3] - Втрата: 0.5465 | Точність: 81.89%

✅ Модель збережена у saved_model/model.pth

🎯 Точність на тестових даних: 79.71%

Process finished with exit code 0
```

На скріншоті показано процес навчання моделі протягом трьох епох.
Під час кожної епохи у прогрес-барі відображаються поточні значення loss та accuracy, що дозволяє стежити за якістю навчання в реальному часі.
Наприкінці кожної епохи виводиться підсумкова втрата (loss) і точність моделі на тренувальних даних.

Після завершення навчання модель успішно збережена у файл saved_model/model.pth.
Далі запускається оцінка на тестовій вибірці — у цьому прикладі модель досягла точності 79.71%.

4. Файл evaluate.py - оцінка точності й швидкості роботи моделі

Тут я реалізував перевірку моделі на тестових даних, щоб побачити, наскільки точно вона розпізнає команди. Також у цьому файлі вимірюється середня затримка одного прогнозу (latency), щоб оцінити, наскільки швидко модель працює на практиці.

```python
import time                             # Для вимірювання часу інференсу
import torch                            # PyTorch для роботи з моделлю і тензорами
import numpy as np                      # NumPy для роботи з масивами та обчисленням середнього
from data_loader import load_data       # Імпорт функції для завантаження test_loader
from model import SpeechCommandCNN      # Імпорт архітектури моделі


# === 1. Функція для обчислення точності ===
def calculate_accuracy(model, test_loader, device):  1 usage
    model.eval()                        # Переводимо модель у режим оцінки
    correct, total = 0, 0               # Лічильники правильних та всіх відповідей

    with torch.no_grad():               # Вимикаємо обчислення градієнтів
        for inputs, labels in test_loader:            # Проходимо всі батчі тестових даних
            inputs = inputs.to(device)                # Переносимо тензори на CPU/GPU
            labels = torch.tensor([["yes", "no", "up", "down"].index(l)
                                   for l in labels]).to(device)  # Перетворення текстових міток у індекси

            outputs = model(inputs)                   # Отримуємо прогноз моделі
            _, predicted = torch.max(outputs.data, 1)  # Обираємо клас з найбільшим значенням

            total += labels.size(0)                   # Додаємо кількість прикладів у батчі
            correct += (predicted == labels).sum().item()  # Рахуємо правильні передбачення

    accuracy = 100 * correct / total                  # Обчислюємо точність у %
    return accuracy
```

```python
# === 2. Функція для вимірювання затримки (latency) ===
def measure_latency(model, test_loader, device, num_batches=10):  1 usage
    model.eval()                                        # Режим оцінки
    latencies = []                                      # Масив для збереження часу кожного батчу

    with torch.no_grad():
        for i, (inputs, _) in enumerate(test_loader):  # Перебір батчів
            if i >= num_batches:                        # Обмежуємо кількість батчів
                break

            inputs = inputs.to(device)
            start = time.time()                         # Початок вимірювання
            _ = model(inputs)                           # Прогін моделі
            end = time.time()                           # Кінець вимірювання

            latency = (end - start) / len(inputs) * 1000  # Затримка для 1 прикладу в мілісекундах
            latencies.append(latency)                   # Зберігаємо результат

    avg_latency = np.mean(latencies)                    # Середнє значення затримки
    return avg_latency
```

```python
# === 3. Основна функція оцінки ===
def evaluate_model():  1 usage
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Вибираємо CPU або GPU
    print(f"Використовується пристрій: {device}")

    # Завантаження даних
    _, test_loader = load_data(batch_size=32)          # Беремо лише тестовий набір

    # Завантаження моделі
    model = SpeechCommandCNN(num_classes=4).to(device) # Створюємо модель
    model.load_state_dict(torch.load( f: "saved_model/model.pth", map_location=device))  # Завантажуємо ваги
    print("✅ Модель завантажена з saved_model/model.pth")

    # Оцінка точності
    accuracy = calculate_accuracy(model, test_loader, device)
    print(f"🎯 Точність моделі: {accuracy:.2f}%")

    # Вимірювання latency
    avg_latency = measure_latency(model, test_loader, device)
    print(f"⚡ Середня затримка (latency): {avg_latency:.2f} мс / приклад")


# === 4. Точка входу ===
if __name__ == "__main__":
    evaluate_model()                    # Запуск оцінки моделі
```

Результат роботи коду:

```
✅ Модель завантажена з saved_model/model.pth
🎯 Точність моделі: 80.32%
⚡ Середня затримка (latency): 0.55 мс / приклад
```

Програма розраховує точність класифікації на тестових даних - у моєму випадку вона становить **80.32%**, що означає, що модель правильно класифікує приблизно 8 із 10 команд.
Також вимірюється середня затримка одного прогнозу (**0.55 мс на приклад**), що демонструє дуже високу швидкість інференсу.

5. Файл inference.py - розпізнавання команд із файлу або мікрофона

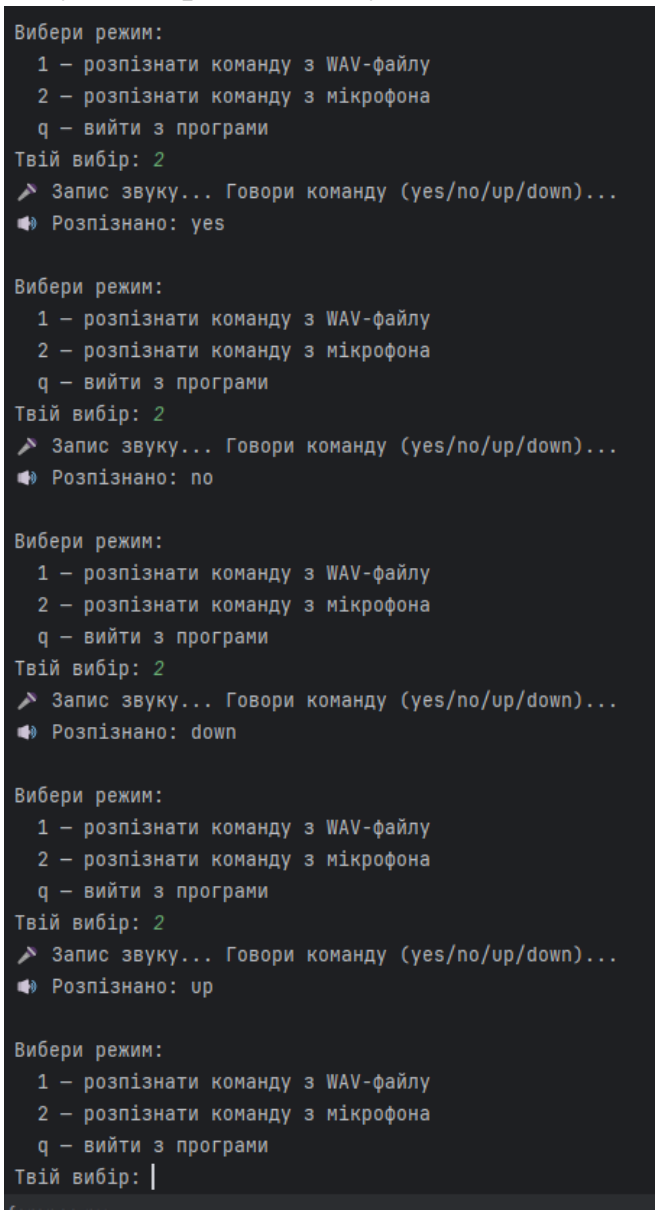
У цьому файлі я додав можливість запускати модель на реальних звуках: або з WAV-файлу, або напряму з мікрофона. Програма перетворює звук у спектрограму й повертає мережевий прогноз - так можна перевірити, як модель працює вживу.

```python
import torch                                # PyTorch для роботи з моделлю та тензорами
import torchaudio                           # Завантаження аудіофайлів
import sounddevice as sd                    # Запис аудіо з мікрофона
import numpy as np                          # Використовується у разі обробки масивів (можна прибрати, якщо не треба
from model import SpeechCommandCNN          # Імпорт архітектури моделі
from torchaudio.transforms import MelSpectrogram, AmplitudeToDB  # Перетворення аудіо у мел-спектрограми


# === 1. Класи команд ===
LABELS = ["yes", "no", "up", "down"]        # Список можливих команд, що класифікує модель


# === 2. Трансформація аудіо у мел-спектрограму ===
def preprocess_audio(waveform, sample_rate=16000):  2 usages
    transform = torch.nn.Sequential(        # Послідовність перетворень
        MelSpectrogram(sample_rate=sample_rate, n_mels=64),  # Мел-спектрограма
        AmplitudeToDB()                     # Перехід до децибелів
    )
    spec = transform(waveform)              # Застосовуємо трансформацію до сигналу
    return spec                             # Повертаємо спектрограму


# === 3. Завантаження моделі ===
def load_model(device):  1 usage
    model = SpeechCommandCNN(num_classes=len(LABELS)).to(device)  # Створюємо модель і переносимо на пристрій
    model.load_state_dict(torch.load( f: "saved_model/model.pth", map_location=device))  # Завантажуємо ваги
    model.eval()                            # Перемикаємо модель у режим інференсу
    return model                            # Повертаємо готову модель
```

```python
# === 4. Інференс із WAV-файлу ===
def predict_from_file(filepath, model, device):  1 usage
    waveform, sample_rate = torchaudio.load(filepath)   # Завантажуємо аудіо з файлу
    if waveform.shape[0] > 1:                           # Якщо кілька каналів (стерео)
        waveform = waveform.mean(dim=0, keepdim=True)   # Перетворюємо у моно

    spec = preprocess_audio(waveform, sample_rate)      # Перетворюємо аудіо в мел-спектрограму
    spec = spec.unsqueeze(0).to(device)                 # Додаємо batch-вимір: [1, 1, 64, time]

    with torch.no_grad():                               # Вимикаємо градієнти для інференсу
        outputs = model(spec)                           # Отримуємо логіти від моделі
        _, predicted = torch.max(outputs, 1)            # Обираємо клас з найбільшим значенням
        label = LABELS[predicted.item()]                # Отримуємо текстову мітку за індексом
    return label                                        # Повертаємо розпізнану команду


# === 5. Інференс із мікрофона ===
def predict_from_mic(model, device, duration=1.0, sample_rate=16000):  1 usage
    print("🎤 Запис звуку... Говори команду (yes/no/up/down)...")
    recording = sd.rec(                                 # Записуємо звук з мікрофона
        int(duration * sample_rate),                    # Кількість семплів = тривалість * частота
        samplerate=sample_rate,
        channels=1,
        dtype='float32'
    )
    sd.wait()                                           # Чекаємо завершення запису
    waveform = torch.tensor(recording.T)                # Перетворюємо запис у тензор і транспонуємо
    spec = preprocess_audio(waveform, sample_rate)      # Перетворюємо у мел-спектрограму
    spec = spec.unsqueeze(0).to(device)                 # Додаємо batch-вимір

    with torch.no_grad():
        outputs = model(spec)                           # Пропускаємо через модель
        _, predicted = torch.max(outputs, 1)            # Обираємо найбільш ймовірний клас
        label = LABELS[predicted.item()]                # Отримуємо текстову мітку
```

```python
# === 6. Точка входу ===
if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Вибираємо CPU або GPU
    model = load_model(device)                                             # Завантажуємо модель
    print("✅ Модель завантажена.")

    # Цикл, щоб можна було багаторазово тестувати модель
    while True:
        print("\nВибери режим:")
        print("  1 – розпізнати команду з WAV-файлу")
        print("  2 – розпізнати команду з мікрофона")
        print("  q – вийти з програми")
        choice = input("Твій вибір: ")

        if choice == "1":
            path = input("Введи шлях до WAV-файлу: ")   # Запитуємо шлях до файлу
            result = predict_from_file(path, model, device)  # Отримуємо прогноз
            print(f"🔊 Розпізнано команду: {result}")   # Виводимо результат

        elif choice == "2":
            predict_from_mic(model, device)             # Запис і розпізнавання з мікрофона

        elif choice.lower() == "q":                     # Якщо користувач хоче вийти
            print("👋 Вихід з програми.")
            break                                       # Виходимо з циклу

        else:
            print("❌ Невірний вибір. Спробуй ще раз.") # Повідомлення про неправильне значення
```

Результат роботи коду:



На скріншоті показано багаторазове тестування програми inference.py у режимі розпізнавання команд через мікрофон. Кожного разу модель записувала короткий фрагмент голосу та повертала відповідну команду - yes, no, down або up.

6. Файл app.py - веб-сервіс для розпізнавання команд через Flask

Тут я підняв простий REST API на Flask, який приймає аудіофайл через HTTP-запит і повертає відповідь моделі у форматі JSON. Завдяки цьому

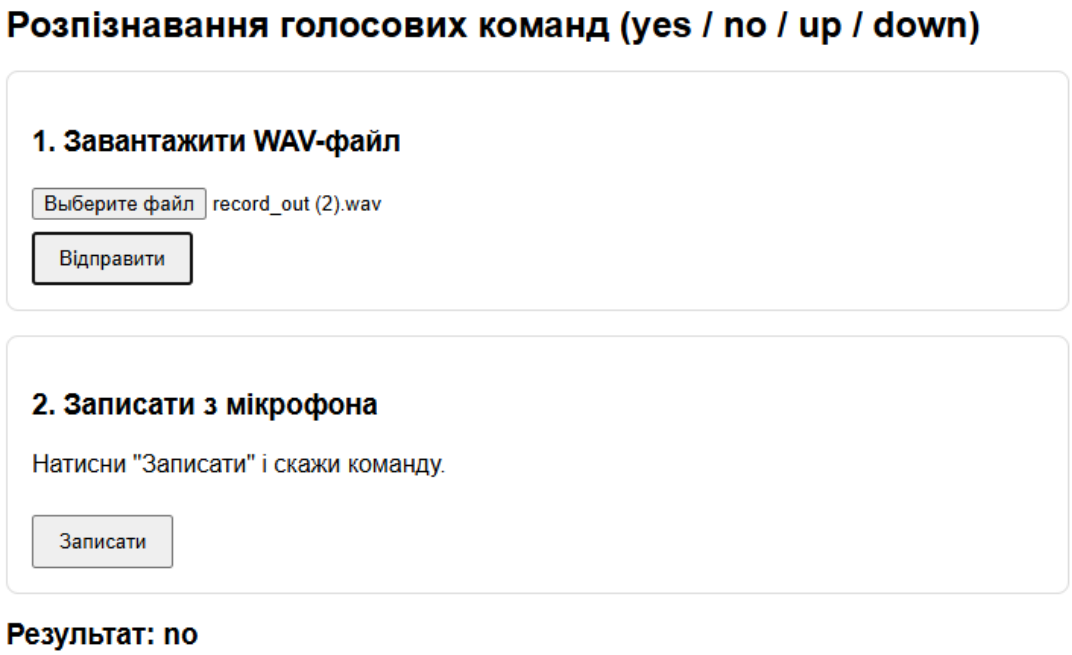
модель можна використовувати як веб-сервіс і підключати до інших застосунків або систем.

```python
import os
import torch
import torchaudio
from flask import Flask, request, jsonify, render_template_string
from model import SpeechCommandCNN
from torchaudio.transforms import MelSpectrogram, AmplitudeToDB

# --- Константи ---
LABELS = ["yes", "no", "up", "down"]
UPLOAD_FOLDER = "uploads"
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

# --- Flask-додаток ---
app = Flask(__name__)
app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER

# --- Пристрій і модель ---
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def load_model(device):  1 usage
    """
    Завантаження моделі з файлу saved_model/model.pth
    """
    model = SpeechCommandCNN(num_classes=len(LABELS)).to(device)
    model.load_state_dict(torch.load( f: "saved_model/model.pth", map_location=device))
    model.eval()
    return model


model = load_model(device)
print(f"✅ Модель завантажена ({device})")
```

```python
# --- Преобробка аудіо ---
def preprocess_audio(waveform):  1 usage
    """
    Перетворення сирого сигналу (16 kHz, 1 канал) у мел-спектрограму.
    На виході тензор форми [1, 64, time].
    """
    transform = torch.nn.Sequential(
        MelSpectrogram(sample_rate=16000, n_mels=64),
        AmplitudeToDB()
    )
    spec = transform(waveform)
    return spec
```

```python
# --- Роут для прогнозу ---
/predict
@app.route( rule: "/predict", methods=["POST"])
def predict():
    if "file" not in request.files:
        return jsonify({"error": "Файл не знайдено"}), 400

    file = request.files["file"]

    if file.filename == "":
        return jsonify({"error": "Ім'я файлу порожнє"}), 400

    filepath = os.path.join(app.config["UPLOAD_FOLDER"], file.filename)
    file.save(filepath)

    try:
        # Завантажуємо аудіофайл
        waveform, sample_rate = torchaudio.load(filepath)

        # Якщо стерео — робимо моно
        if waveform.shape[0] > 1:
            waveform = waveform.mean(dim=0, keepdim=True)

        #  Ресемплінг до 16 kHz, якщо треба
        if sample_rate != 16000:
            resampler = torchaudio.transforms.Resample(orig_freq=sample_rate, new_freq=16000)
            waveform = resampler(waveform)
            sample_rate = 16000

        # Перетворення у спектрограму (тепер завжди 16 kHz)
        spec = preprocess_audio(waveform)     # [1, 64, time]
        spec = spec.unsqueeze(0).to(device)   # [1, 1, 64, time]
```

Результат роботи коду це простий інтерфейс з таким же функціоналом



Висновок: У ході цієї лабораторної роботи я пройшов повний цикл створення та інтеграції моделі машинного навчання для розпізнавання голосових команд. Я побудував згорткову нейромережу, підготував датасет аудіозаписів, виконав навчання моделі та оцінив її точність і швидкість роботи. Після цього я реалізував інференс як у вигляді консольного застосунку, так і у форматі веб-сервісу на Flask, що дозволяє тестувати модель на WAV-файлах і записах із мікрофона в реальному часі. Окрема увага була приділена попередній обробці аудіосигналів та правильному перетворенню їх у мел-спектрограми, що є критично важливим етапом для коректного розпізнавання. У результаті вдалося отримати стабільно працюючу модель, інтегровану у повноцінний веб-інтерфейс, яка демонструє достатньо високу точність та швидкість роботи. Загалом виконана робота дозволила на практиці закріпити знання зі створення ML-моделей, роботи з аудіоданими та побудови API-сервісів для їх використання.