Středoškolská technika 2025

Setkání a prezentace prací středoškolských studentů na ČVUT

Vývoj mobilní aplikace v Pythonu

Eduard Wojnar

Gymnázium Mnichovo Hradiště
Studentská 895, Mnichovo Hradiště

Poděkování

Rád bych poděkoval panu Jelínkovi za podporu, kterou mi poskytl během práce na tomto projektu.

Abstrakt

V této ročníkové práci jsem se zaměřil na vývoj mobilní aplikace v jazyce Python, konkrétně s využitím frameworku Kivy a nástroje Buildozer. Během řešení jsem zjistil, že Python není vždy tou nejvhodnější volbou pro tvorbu mobilních aplikací, a to mimo jiné kvůli jeho omezené podpoře na některých platformách. Přesto se podařilo prokázat, že s pomocí knihoven Buildozer a Kivy lze vytvářet a úspěšně nasazovat aplikace i na mobilní zařízení.

Klíčová slova

Python
Kivy
OOP
OpenCV
Buildozer
TensorFlow
Android
MNIST

Obsah

Úvod

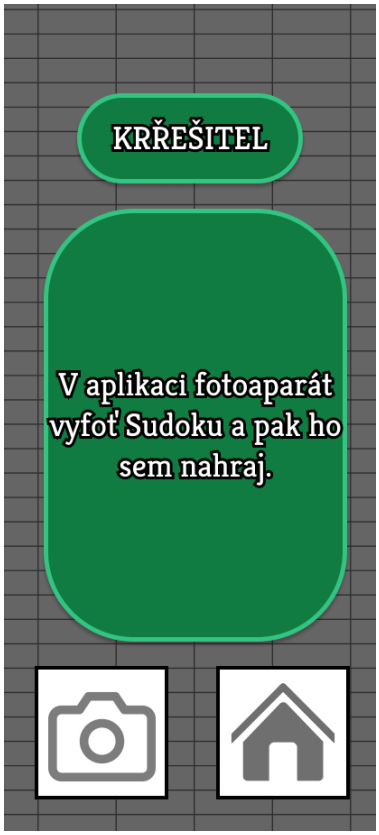
Během hodin SVT jsme se s vedoucím mé práce dostali k tématu vývoje mobilních aplikací v Pythonu. Rozhodl jsem se tuto oblast prozkoumat z praktického hlediska a vytvořit funkční aplikaci. Práci jsem rozdělil na tři části: návrh designu, samotné programování v Pythonu a finální sestavení (build). K tomu jsem využil virtuální prostředí Buildozer, u něhož jsem narazil na nedostatky v dokumentaci – a právě to se ukázalo jako hlavní výzva celého projektu. Díky zkušenostem, které jsem při vývoji získal, bych nyní dokázal vytvořit další aplikace, jež bychom mohli využít i v rámci výuky SVT.

1. Front-end

Nejprve jsem se věnoval tomu, co uživatel v aplikaci vidí a na co kliká. Tohle uživatelské rozhraní (front-end) jsem připravoval v grafickém programu Figma, který slouží ke kreslení a navrhování designů. Pro celkový vzhled mě inspirovala aplikace Microsoft Excel, protože je dostatečně jednoduchá a zároveň přehledná. Navíc jsem se rozhodl pojmenovat aplikaci po panu učiteli Lubomíru Krskovi.

Základem mého designu jsou PNG obrázky, ve kterých jsou jednotlivé prvky viditelné jako grafika. Aby uživatel mohl s aplikací pracovat, přidávám do nich v Kivy takzvaná „průhledná tlačítka“ – ve skutečnosti je to neviditelná vrstva, kterou Kivy rozezná jako stisknutelné tlačítko. Tato tlačítka jsou umístěna relativně, což znamená, že se přizpůsobují různým velikostem okna nebo obrazovky. Ať už tedy aplikaci spustím na malém mobilu nebo na větším tabletu, rozložení prvků zůstává funkční. Tímto způsobem dokážu zachovat přizpůsobitelnost grafiky, a nebude problém přidat darkmode nebo jinou barevnou kombinaci.

Obr. 1: Hlavní stránka aplikace
(Zdroj: autor)



Obr. 2: Stránka pro načtení fotky
(Zdroj: autor)

2. Python

Python patří mezi takzvané „vysokoúrovňové“ jazyky. Znamená to, že je relativně snadný na naučení a čtení kódu, oproti nízkoúrovňovým jazykům, kde se hodně pracuje s detaily hardwaru. Proto se Python často používá k rychlému vytváření různých skriptů, správě dat, nebo dokonce ke strojovému učení (například k rozpoznávání obrázků či hlasu).
Když se ale bavíme o vývoji mobilních aplikací, není Python úplně běžnou volbou. I tak se s ním dá uspět, protože k němu existují speciální knihovny a nástroje (například Kivy nebo Buildozer), které nám pomohou převést aplikaci do podoby, kterou si telefon dokáže nainstalovat. Toto řešení ovšem může být složitější a někdy i o něco pomalejší než využití klasických jazyků, jež se na mobilních platformách používají nejčastěji.
V profesionální praxi se v Pythonu píše hlavně v objektově orientovaném stylu (OOP) – to znamená, že místo psaní dlouhého seznamu příkazů si vytváříme různé „objekty“ s vlastnostmi a funkcemi, což programátorům pomáhá udržet pořádek v kódu. Já jsem se pro Python rozhodl jednak proto, že ho používáme v hodinách SVT, a taky mě zajímalo, jestli se dá v tomto jazyce udělat mobilní aplikace od A až do Z.

2.1. Knihovny

V programování se pod slovem „knihovna“ rozumí balíček předem připraveného kódu, který nám pomáhá rychle a jednoduše řešit nějaký konkrétní problém. Může to být třeba kreslení grafiky, práce s kamerou nebo rozpoznávání obrázků. Místo toho, abychom si vše museli programovat sami od začátku, použijeme už hotové řešení, které do svého projektu snadno přidáme.
Pythonové knihovny jsou obvykle dobře optimalizované (fungují efektivně) a mají kvalitní dokumentaci (návody a příklady), takže se s nimi pracuje poměrně snadno. Jedinou nevýhodou někdy bývá jejich velikost, což může výrazně zvětšit výsledný soubor s aplikací, který instalujeme na telefon nebo počítač.
Díky knihovnám se nám tedy otevírá cesta k mnoha užitečným funkcím, aniž bychom museli programovat všechno sami. Stačí je přidat do kódu a říct jim, co přesně chceme dělat. Pokud ale použijeme hodně velkých knihoven, můžeme skončit s rozměrnou aplikací, která zabírá víc místa a může se někdy i déle spouštět.

2.1.1. Pygame

Mojí prvotní myšlenkou bylo použít knihovnu Pygame, která je považována za jeden z nejjednodušších frameworků pro tvorbu aplikací v Pythonu. V podstatě využívá jednu hlavní smyčku (main loop) a vykresluje textury přímo na obrazovku, pomocí speciální funkce, aniž by vyžadovala speciální objekty. Díky tomu se s ní dobře začíná lidem, kteří ještě nemají s programováním žádné zkušenosti.

2.1.2. Kivy

Kivy přišlo na řadu jako druhá volba ve chvíli, kdy jsem zjistil, že Pygame neumí nativně pracovat s kamerou a musel bych zobrazovat dvě okna zároveň. Kivy je obecně považované za nejrozšířenější framework pro mobilní vývoj v Pythonu a má k tomu dobré důvody. Nabízí vlastní sadu objektů, například App nebo Screen, díky nimž je kód přehlednější a lépe strukturovaný. Počítá ovšem s tím, že vývojář má vyšší znalost objektově orientovaného programování, což se ale v mém případě hodilo, protože se OOP učíme v rámci SVT.

2.1.3. OpenCV

Na možnost využít OpenCV jsem narazil v jednom GitHub repozitáři, který řešil podobný úkol jako já. Tato knihovna se zaměřuje na zpracování obrazu, ať už v reálném čase, nebo z pořízených fotografií. Nabízí řadu metod pro detekci objektů a jejich následné rozpoznání. Já jsem těchto funkcí využil k nalezení pole Sudoku.

2.1.4. Tensorflow

TensorFlow je knihovna, která se používá na trénování modelů a jejich následní nasazení. V projektu jsem použil obě tyto využití. Nejprve jsem pracoval s formátem *.h5, ale kvůli omezením na platformě Android jsem musel přejít na TensorFlow Lite. Tensorflow pro Python, ale není pro nasazení na Android moc dobrá volba z důvodů, problému datových typů.

2.1.5. Ostatní knihovny

Z dalších knihoven jsem využil hlavně os a random, které jsou součástí standardní výbavy Pythonu. os mi pomáhá se správou adresářů, například při ukládání vyfocených snímků, a random zase používám pro generování sudoku pole. Kromě toho si také pomocí standardních knihoven obstarávám datum a čas, které pak slouží jako název pro ukládané soubory. To je užitečné i proto, že Android může s adresářovou strukturou zacházet poněkud odlišně, a je tedy dobré vše mít jasně pojmenované.

2.2. OOP

Objektově orientované programování je způsob, jak promýšlet a psát kód tak, aby místo jednotlivých kroků (instrukcí) vznikaly "virtuální objekty", se kterými pak program pracuje. Tyto objekty mají své vlastnosti (například barvu, jméno či velikost) a metody (tedy činnosti, které dovedou vykonávat).

Můžeme si to představit třeba na příkladu *auta* (objektu), které má určitou barvu (vlastnost) a může jet nebo zastavit (metody). Pomocí OOP tak jednoduše určíme, co všechno náš "objekt" dovede a jak se má chovat. Jednotlivé objekty mají každý stejné základy(třídu) a pak se lišit jen konkrétními detaily (například barvou nebo dalšími vlastnostmi). Třída je vlastně nějaká forma a objekty jsou už vyrobené odlitky. Díky tomu se dá kód snadno udržovat a znovu používat, protože máme jasně nadefinované, co jednotlivé části programu dělají.

3. Programování

Programování této aplikace nakonec nebylo tak složité a časově náročné, jak jsem si původně myslel. Většina práce spočívala v tvorbě uživatelského rozhraní (front-end), kde se nevyskytují žádné extrémně složité postupy. Hlavním „oříškem“ byla implementace herního Sudoku, ale i to se vyřešilo snadno pomocí takzvaného backtrackingu (česky „procházení zpětnou cestou“). Ten funguje na principu postupného zkoušení všech možných řešení, dokud se nenajde to správné. Pro Sudoku je tento způsob řešení velmi oblíbený a bohatě stačí na všechny běžné případy.

3.1. Struktura kódu

Můj program je momentálně uložený v jednom souboru. Dělám to tak hlavně kvůli Buildozeru, který si s více soubory někdy hůře poradí. Buildozer poté představím ve své vlastní kapitole.
Pokud bych chtěl, aby kód splňoval přísnější pravidla (třeba Pylint, což je nástroj na kontrolu stylu a správnosti kódu), musel bych pravděpodobně rozdělit program do více modulů (menších souborů) a dodržet další doporučení. Zatím ale vše funguje, a tak jsem zvolil jednodušší cestu v jednom souboru.

3.2. Implementace logiky a funkčnost aplikace

V téhle aplikaci jsem se nesetkal s ničím příliš složitým, co se týče různých optimalizačních či matematicky náročných postupů. Většina logiky je jednoduchá a dá se snadno pochopit i bez pokročilých znalostí. Nejnáročnější část jsem řešil v nápovědě (hintu), kde jsem musel použít další backtracking algoritmus (podobný jako u Sudoku). Ten projde různé možnosti a najde tu, která nám v danou chvíli pomůže s řešením. Ani tento postup ale není natolik komplexní, aby vyžadoval nějaké speciální znalosti.

3.3. Ukázky kódu

Jednotlivé ovládací prvky (widgety) v Kivy fungují podobně jako v jiných GUI knihovnách, takže není složité se v tom zorientovat. Každé políčko Sudoku mám vytvořené jako průhledné tlačítko, do kterého mohu vložit číslo nebo sadu malých číslic pro poznámky. Když uživatel klepne na dané políčko, aplikace hned zvýrazní celý příslušný řádek, sloupec a 3×3 blok. Tento mechanismus jsem od začátku plánoval, aby uživatelům ulehčil sledovat veškeré vazby mezi zadanými hodnotami. Pro mě to znamenalo napsat si funkci, která po kliknutí prochází mřížku a nastavuje zvýraznění vybraných polí. V praxi jde o to, že Kivy mi poskytuje jednoduché napojení na události (např. on_press), v nichž volám vlastní kód, který toto zvýraznění provede.
Další důležitou částí je rozvržení celé obrazovky. V Kivy běží hlavní třída, kterou mám pojmenovanou například MainApp (dědí z kivy.app.App). Uvnitř je ScreenManager, do kterého jsem přidal několik samostatných „obrazovek“ (Screenů). Jedna z nich slouží výhradně k zobrazení a ovládání Sudoku, další třeba k nastavení nebo k zobrazení menu. Tím, že mám všechno takto oddělené, se mi snadno spravuje kód – když chci něco změnit v hlavním menu, nemusím listovat nekonečným souborem, ve kterém by se míchala logika Sudoku s kódem menu. Prostě si otevřu příslušnou obrazovku, kde řeším jen widgety a události patřící právě k menu.
Klíčovým prvkem tohoto projektu je backtracking algoritmus, který zajišťuje řešení Sudoku a zároveň umí i vygenerovat novou úlohu. Rozhodl jsem se ho umístit do samostatné třídy SolverLogic, abych logiku oddělil od uživatelského rozhraní. V momentě, kdy Sudoku řeším, procházím prázdná políčka a zkouším do nich umístit čísla, jež splňují základní pravidla (nesmí se opakovat v řádku, sloupci a v 3×3 bloku). Pokud narazím na konflikt, algoritmus se vrátí zpátky (backtrackne) a zkusí jinou variantu. Tohle mi přijde jako rozumný, a hlavně běžně srozumitelný přístup, který se dá snadno upravovat a ladit. Kdybych třeba potřeboval v budoucnu implementovat různé úrovně obtížnosti, stačí ovlivnit generátor zadání, aniž by to nějak rozhodilo grafiku Kivy.
Díky tomu, že SolverLogic funguje zcela nezávisle, můžu v rámci Sudoku obrazovky posílat do třídy SolverLogic informace o tom, co uživatel zadal, a v reakci na to můžu třeba zkontrolovat správnost. Mně osobně se osvědčilo i to, že do SolverLogic můžu přidat další doplňkové funkce – například kontrolu, jestli vyplněné hodnoty netvoří nekonzistentní stav. V momentě, kdy by to celé bylo smíchané s kódem Kivy, bych musel mít neustále na paměti, kde se, co vykresluje, a vývoj by se snadno proměnil v chaos. Takhle si Kivy obstarává ovládání a vykreslování, zatímco SolverLogic drží pravidla a backtracking.
Další částí, kterou jsem do aplikace integroval, je rozpoznávání čísel z nahraných obrázků. Měl jsem v plánu, že by aplikace mohla usnadnit zadávání Sudoku, kdy uživatel jen vyfotí Sudoku z novin nebo nějaké knihy. Tady jsem využil kombinaci OpenCV a TensorFlow. Proces v OpenCV začíná převedením obrázku do odstínů šedé a následným použitím Otsu thresholdingu, abych získal co nejostřejší přechody mezi čísly a pozadím. Protože můj model pro rozpoznávání byl trénovaný s černým pozadím a bílými čísly, musel jsem pak obrátit barvy, aby to odpovídalo formátu, který neuronová síť očekává.
Abych správně uchopil celé Sudoku, vyhledávám největší čtyřúhelník v obrázku – typicky je to samotná mřížka. Ten pak matematicky „narovnám“ do roviny, kdyby byl obrázek focený z úhlu. Poté je nutné identifikovat a případně narýsovat či vyčistit linky mezi jednotlivými políčky. Do toho spadá i odstraňování rušivých čar na krajích, protože se někdy stává, že se kříží s vytištěným okrajem Sudoku a dělá to nepořádek. Když mám všechny buňky správně ohraničené, rozdělím je na 81 menších segmentů, každý o velikosti 28×28 pixelů. Tahle velikost odpovídá vstupnímu formátu pro můj model v TensorFlow, který vyplivne odhad, jaké číslo v buňce je. V praxi to vrací deset pravděpodobností pro číslice 0 až 9, z nichž vyberu tu nejvyšší a vepíšu ji do pole. S takto sestavenou maticí čísel už můžu opět pracovat v SolverLogic.

Musím podotknout, že získat spolehlivý výsledek při reálných fotografiích není vždy úplně triviální. Největší úskalí je, když je fotka rozmazaná nebo špatně nasvícená. Právě v takových situacích je nutné pečlivě ošetřit průběh thresholdingu nebo úpravu kontrastu, aby nedošlo k „rozmazání“ číslic. Učitelé i studenti informatiky, kteří se někdy pustili do počítačového vidění, potvrdí, že drobné změny ve světle můžou obrátit výsledky rozpoznávání vzhůru nohama. Mně se celkem osvědčilo testovat různé fotky, třeba Sudoku vyfocené telefonem večer v místnosti, aby se ukázaly reálné slabiny. Když se mi něco nepovedlo rozpoznat, mohl jsem pak upravit příslušný krok v OpenCV, ať už to znamenalo použít jiný způsob detekce hran, nebo jen změnit parametr v thresholdu.
Main.py soubor má přes 1800 řádků. Zde jsou některé ukázky. Kompletní kód je na Githubu. Odkaz je v závěru práce.

```python
# Importing the kivy framework
from kivy.app import App
from kivy.uix.button import Button
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.core.window import Window
from kivy.uix.image import Image
from kivy.core.audio import SoundLoader
from kivy.uix.screenmanager import NoTransition
from kivy.clock import Clock
from kivy.uix.label import Label
from kivy.graphics.texture import Texture
from kivy.uix.camera import Camera
from kivy.resources import resource_find
from kivy.resources import resource_add_path
from kivy.utils import platform

# Importing all the other libraries
import random
import os
import cv2
import numpy as np
from plyer import filechooser
import android
from android.storage import primary_external_storage_path
from jnius import autoclass
from android.permissions import request_permissions, Permission
from android import activity, mActivity


File = autoclass("java.io.File")
Interpreter = autoclass("org.tensorflow.lite.Interpreter")
InterpreterOptions = autoclass("org.tensorflow.lite.Interpreter$Options")
TensorBuffer = autoclass("org.tensorflow.lite.support.tensorbuffer.TensorBuffer")
ByteBuffer = autoclass("java.nio.ByteBuffer")
```

Obr. 3: Importy aplikace
(Zdroj: autor)

```
# Column class for the sudoku board
class Column(Button):
    def __init__(self, name, rel_x, rel_y, rel_width, rel_height, block, **kwargs):···

    # Functions for the column
    def _update_label_size(self, instance, size):···

    def _update_notes_position(self, *args):···

    def display_number(self, number=0):···

    def correct_font_size(self):···
```

Obr. 4:Column objekt a jeho funkce
(Zdroj: autor)

```
class SolverLogic:
    def __init__(self):···

    def _precompute_relationships(self):···

    def is_solved(self, list_of_columns):···

    def check(self, list_of_columns):···

    def validate(self, column, list_of_columns, number):···

    def create_a_riddle(self, list_of_columns, difficulty=20):···

    def solve_a_riddle(self, list_of_columns):···

    def _solve(self, cells):···

    def _is_valid(self, cells, row, col, num):···
```

Obr. 5: Objekt SolverLogic a jeho funkce
(Zdroj: autor)

```python
def extract_sudoku_digits(self, image_path):
    print(f"[DEBUG] Začínám zpracovávat obrázek: {image_path}")
    try:
        image = cv2.imread(image_path)
        if image is None:
            print(f"[ERROR] cv2.imread vrátil None pro cestu: {image_path}")
            return None

        print(f"[DEBUG] Rozměry obrázku: {image.shape}")

        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        _, th = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

        th_inv = cv2.bitwise_not(th)

        contours, _ = cv2.findContours(
            th_inv, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
        )
        biggest = None
        max_area = 0
        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area > 1000:
                approx = cv2.approxPolyDP(
                    cnt, 0.02 * cv2.arcLength(cnt, True), True
                )
                if len(approx) == 4 and area > max_area:
                    biggest = approx
                    max_area = area

        contour_debug = image.copy()
        if biggest is not None:
            cv2.drawContours(th_inv, [biggest], -1, (0, 255, 0), 2)

        if biggest is None:
            return None

        img_area = image.shape[0] * image.shape[1]
        if max_area < 0.2 * img_area:
            print(
                f"[ERROR] Největší kontura je příliš malá ({max_area}/{img_area}) - není to sudoku!"
            )
            return None

        # Kontrola poměru stran
        rect = self.order_points(biggest.reshape(4, 2))
        width = np.linalg.norm(rect[1] - rect[0])
        height = np.linalg.norm(rect[3] - rect[0])
        aspect_ratio = width / height
        if not (0.8 < aspect_ratio < 1.2):  # Sudoku je téměř čtverec
```

Obr. 6: Část funkce extract_sudoku_digits
(Zdroj: autor)

4. Build na Android

Když vyvíjíme mobilní aplikaci, musíme ji nakonec převést do formátu, který si telefon nebo tablet skutečně nainstaluje. U systému Android se tomu říká „build“ a výsledkem je soubor s příponou *.apk. Pro klasický vývoj (například v jazyce Java nebo Kotlin) se používají oficiální nástroje od Googlu, ale pokud programujeme aplikaci v Pythonu, situace je o něco složitější.
Právě proto existuje Buildozer – speciální nástroj (knihovna), který nám z terminálu (příkazového řádku) pomůže aplikaci sestavit. Funguje to tak, že mu dodáme náš Pythonový kód a nastavení (co vše aplikace používá), a Buildozer se už postará o převedení do *.apk souboru. Zároveň také stahuje a instaluje potřebné komponenty, aby na výsledném zařízení (mobilu nebo tabletu) vše fungovalo.
Z mého pohledu byla tahle část vývoje největší výzva, protože dopředu jsem nevěděl, jak Buildozer přesně funguje a co všechno bude chtít nastavit. Musel jsem ladit různé chyby, které se objevovaly během kompilace, a zkoumat dokumentaci i různá internetová fóra. Nakonec se ale ukázalo, že když se podaří Buildozer správně vyladit, je to poměrně jednoduché řešení – stačí zadat příkaz do terminálu a on už zařídí zbytek. Samozřejmě, aplikaci jde stavět i ručně, ale to je mnohem náročnější a složitější.

4.1. Buildozer

Buildozer bohužel funguje jen na Linuxu, takže pokud pracujete na počítači s Windows, potřebujete nějaké řešení, které Linux “napodobí”. Já jsem k tomu využil WSL (Windows Subsystem for Linux), což je vlastně taková “miniverze” Linuxu běžící přímo ve Windows. Při práci s WSL je ale občas potřeba být opatrný – několikrát se mi stalo, že jsem si celý systém v Ubuntu zablokoval, takže jsem to musel znovu nastavovat.
Dalším omezením je, že Buildozer nepodporuje nejnovější verze Pythonu, takže si musíme pohlídat, jakou verzi přesně instalujeme. Kromě toho Buildozer vytváří soubor buildozer.spec, kde nastavujeme všechno, co naše aplikace potřebuje: od verzí knihoven až po různá povolení. Ne vždycky ale Buildozer podporuje všechny verze knihoven, a ještě k tomu je můžeme do projektu přidávat různými způsoby (přímo z PyPI, z lokálních zdrojů nebo z otevřených repozitářů). Najít správnou kombinaci nastavení ve buildozer.spec tak někdy dá dost zabrat – to byl pro mě asi největší oříšek.
Jakmile je vše nastavené, Buildozer si stáhne potřebné balíčky, zkombinuje je s naší aplikací a vytvoří výsledný APK soubor. Ten proces ale může trvat docela dlouho – třeba u mě na 20jádrovém procesoru průměrně tři a půl hodiny, i když je pravda, že WSL může být pomalejší než opravdový Linux. Jakmile se ale Buildozer “naučí” vše, co potřebuje, dá se aplikace postupně sestavovat rychleji. A hlavní výhoda je, že po tomhle maratonu opravdu dostanete mobilní aplikaci připravenou k instalaci na telefon.

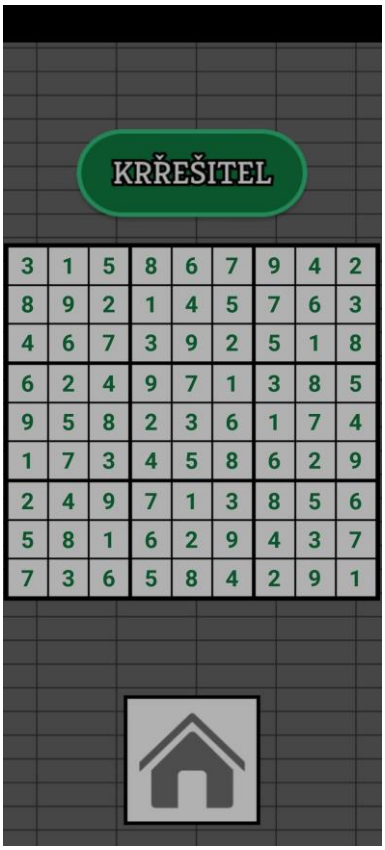
```
[app]
# Name of your application
title = Krdoku

# Package name (must be lowercase, no spaces, no special characters)
package.name = krdoku

# Your domain (reverse domain style, e.g., com.example)
package.domain = org.test

# Directory containing your application code
source.dir = .

# File extensions to include in the APK
source.include_exts = py,png,jpg,tflite,mp3,kv,atlas,txt,dm

# Python modules your app depends on
requirements = python3,kivy==2.3.1,numpy,pillow,opencv_extras,opencv,filetype,android,jnius,plyer

# Application version
version = 1.0

# Screen orientation
orientation = portrait

# Make the app fullscreen
fullscreen = 1
```

5. Debugging

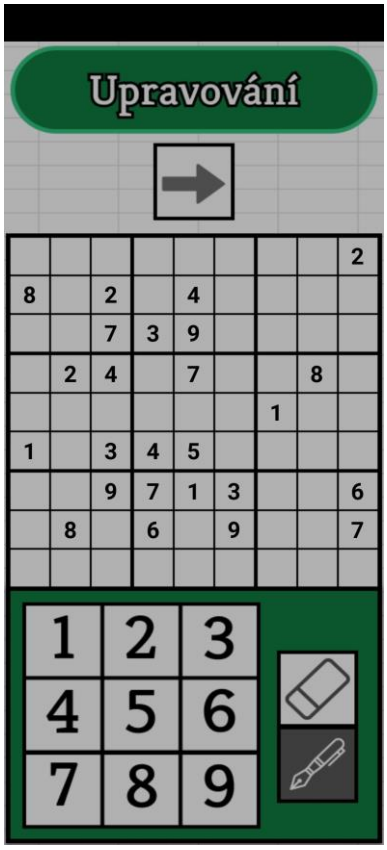
Debugging celého projektu byl náročný, protože se stále objevovaly různé chyby, které jsem musel řešit krok za krokem. Hlavně Buildozer vyžaduje speciální nastavení, které se nikde jednoduše nedočtete, takže jsem musel spoléhat na vlastní pokusy a úpravy.
Používat TensorFlow, který je napsaný v Pythonu, v aplikaci pro Android, jež běží na Javě, se ukázalo jako složité hlavně kvůli rozdílným datovým typům. Bez knihovny pjnius a funkce autoclass byla velice kritická. V závěru jsem také musel natrénovat nový model TensorFlow, tentokrát trénovaný na MNIST s 50 epochami, dosahuje 99,5% úspěšnosti, i když někdy stále plete devítku a šestku.
Backtracking algoritmus, který řeší solving, jsem nechal optimalizovat pomocí Grok Ai, což zrychlilo jeho fungování. Celkově bylo ladění hodně o tom, abych správně vybral chatbota (ChatGPT, DeepSeek, Grok), který mi vždycky pomohl najít a opravit konkrétní problém.
Všechen debugging probíhal v aplikaci Android Studio,

6. Praktická část

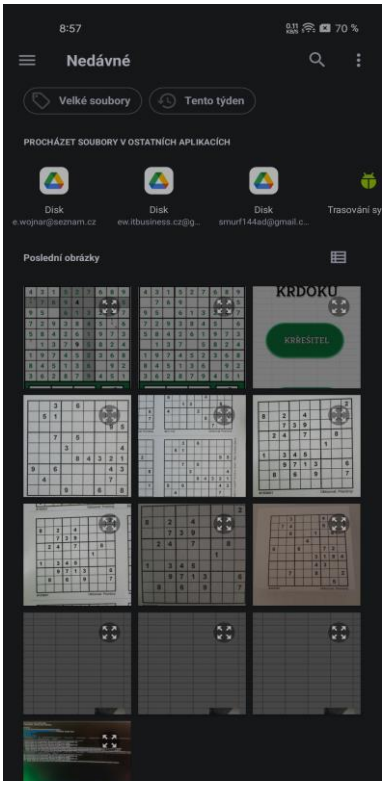
Moje praktická část je naprogramovaná aplikace Krdoku. Aplikaci je možné stáhnout z mého Github repozitáře.

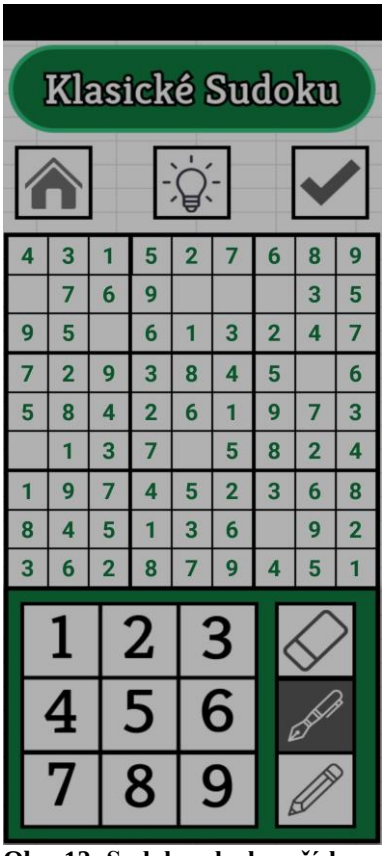


Obr. 9: Vyřešené Sudoku z fotky
(Zdroj: autor)

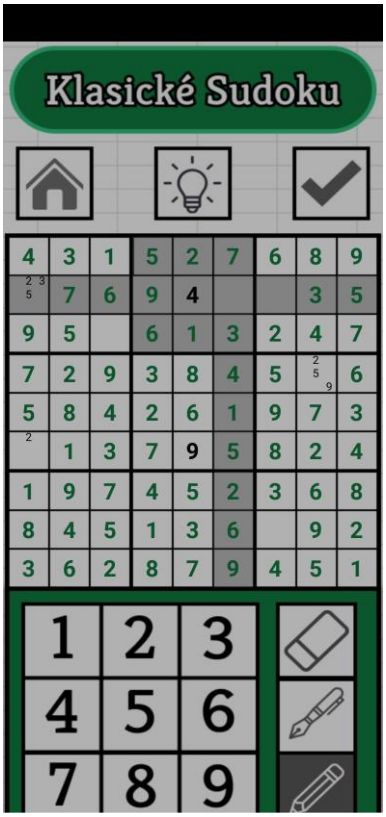


Obr. 8: Možnost upravení po načtení fotky
(Zdroj: autor)





Obr. 12: Sudoku plocha s čísly
(Zdroj: autor)



Obr. 13: Hlavní stránka aplikace
(Zdroj: autor)

Závěr

Aplikace nakonec funguje podle plánu a díky tomuto projektu jsem se naučil mnohem víc o Pythonu, ale i o tom, jak se dělají mobilní aplikace. Získané znalosti teď můžu využít při výuce SVT a třeba tak pomoci ostatním spolužákům nebo i novým zájemcům o programování. Dozvěděl jsem se, že použití kamery je velice náročné z důvodů stejného rozlišení preview a potom výsledného obrázku, proto jsem se rozhodl ve finální aplikaci pouze načítat ze souborů. Samozřejmě to není bezchybné a jsou tam problémy, když je papír prohnutý, nebo přesvícený. Ale i tak si troufám říct, že je to hezký výsledek. Dále bych doporučil nepsat ročníkovou práci v aplikaci microsoft Word, ale v OnlyOffice alternativě, funguje mnohem lépe😀.