National College of Ireland

TrailSurfNET: Trail Surface Classification Using Convolutional Neural Networks and OpenStreetMap Annotations - Configuration Manual

MSc (Research) Practicum
MSc Top-up Artificial Intelligence

Mark Finlay
Student ID: x10209221

School of Computing
National College of Ireland

Supervisor: Faithful Chiagoziem ONWUEGBUCHE

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Mark Finlay
Student ID:	x10209221
Programme:	MSc Top-up Artificial Intelligence
Year:	2025
Module:	MSc (Research) Practicum
Supervisor:	Faithful Chiagoziem ONWUEGBUCHE
Submission Due Date:	11/08/2025
Project Title:	TrailSurfNET: Trail Surface Classification Using Convolutional Neural Networks and OpenStreetMap Annotations - Configuration Manual
Word Count:	
Page Count:	26

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	15th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

TrailSurfNET: Trail Surface Classification Using Convolutional Neural Networks and OpenStreetMap Annotations - Configuration Manual

Mark Finlay
x10209221

1 Introduction

This document provides detailed instructions for replicating the experimental setup of the TrailSurfNET project. The primary objective of this research is to classify hiking trail surfaces using Sentinel-2 satellite imagery and OpenStreetMap (OSM) data. This manual outlines the necessary hardware, software, dependencies, and procedural steps to reproduce the data pipeline and model training experiments described in the thesis.

2 Hardware Requirements

The experiments were conducted on a machine with the following specifications. While these are not strict minimums, performance may vary on different hardware.

- **CPU:** Apple Silicon that supports MPS or a GPU supporting Cuda. The training scripts will automatically detect and use "mps" for Apple Silicon, "cuda" for NVIDIA GPUs, or fall back to "cpu".
- **Memory (RAM):** A minimum of 16 GB is recommended.
- **Storage:** A minimum of 50 GB of free space is required for the PostgreSQL database, OSM data files, and generated datasets.

3 Software and Library Dependencies

The project relies on a Python environment with several key libraries. The recommended virtual environment to use for this project is `venv`.

3.1 Core Software Dependencies

- **Python environment manager:** `venv`
- **Python:** Version 3.11 or later should be installed and available in the path.
- **Python Libraries** The required Python libraries are listed in `requirements.txt`.

- **PostgreSQL:** Version 17 or later, with the PostGIS extension for spatial data support, Hstore extension for storing sets of key/value data types (required by osm2pgsql import). Ensure that the database is running (`brew services start postgresql on a mac`)
- **osm2pgsql:** A command-line tool for importing OSM data into a PostgreSQL/PostGIS database. `https://osm2pgsql.org/doc/install.html`. This is a dependency of the `setup_db.py` script.

4 Setup

Follow these steps to set up the project environment:

4.1 Environment

1. Clone the repository:

```
git clone git@github.com:rudedoc/TrailSurfNET.git
cd TrailSurfNET
```

2. Create and activate a Python virtual environment:

```
python3 -m venv myenv
source myenv/bin/activate
```

3. Install the required Python packages:

```
pip install -r requirements.txt
```

```
markfinlay@Marks-MacBook-Pro-2 Documents % cd TrailSurfNET
markfinlay@Marks-MacBook-Pro-2 TrailSurfNET % python3 -m venv myenv
markfinlay@Marks-MacBook-Pro-2 TrailSurfNET % source myenv/bin/activate
(myenv) markfinlay@Marks-MacBook-Pro-2 TrailSurfNET % pip install -r requirements.txt
Collecting pandas
  Using cached pandas-2.3.1-cp39-cp39-macosx_11_0_arm64.whl (10.8 MB)
Collecting sqlalchemy
  Using cached sqlalchemy-2.0.42-cp39-cp39-macosx_11_0_arm64.whl (2.1 MB)
Collecting psycopg2-binary
  Using cached psycopg2_binary-2.9.10-cp39-cp39-macosx_14_0_arm64.whl
Collecting geoalchemy2
  Using cached GeoAlchemy2-0.17.1-py3-none-any.whl (77 kB)
Collecting numpy
  Using cached numpy-2.0.2-cp39-cp39-macosx_14_0_arm64.whl (5.3 MB)
```

Figure 1: Environment Setup

4.2 setup_db.py

Once your environment is setup run: `python setup_db.py`. This script can up to 45 minutes to run depending on your system. This script will:

1. Download the necessary OSM data files for Ireland and the UK from Geofabrik.
2. Drop and rebuild a database with the name `osm`.
3. Enable the `postgis` and `hstore` extensions.
4. Download `ireland-and-northern-ireland-latest.osm.pbf` (500MB) and `united-kingdom-latest.osm.pbf` (2GB) OSM files from `https://download.geofabrik.de/europe/`
5. Use `osm2pgsql` to import the downloaded OSM data into the database.

```
(myenv) markfinlay@Marks-MacBook-Pro-2 TrailSurfNET % python setup_db.py
Dropping database osm (if it exists)...
Creating fresh database osm...
Enabling PostGIS and hstore extensions...
Database osm has been reset and extensions enabled.
ireland-and-northern-ireland-latest.osm.pbf already exists, skipping download.
united-kingdom-latest.osm.pbf already exists, skipping download.
Importing OSM data using osm2pgsql...
2025-08-06 14:32:14  osm2pgsql version 2.0.1
2025-08-06 14:32:14  Database version: 17.4 (Homebrew)
2025-08-06 14:32:14  PostGIS version: 3.5
2025-08-06 14:32:14  WARNING: The pgsql (default) output is deprecated. For details see https://osm2pgsql.org/doc/faq.html
2025-08-06 14:32:14  Initializing properties table '"public"."osm2pgsql_properties"'.
2025-08-06 14:32:14  Storing properties to table '"public"."osm2pgsql_properties"'.
2025-08-06 14:32:14  Setting up table 'planet_osm_point'
2025-08-06 14:32:14  Setting up table 'planet_osm_line'
2025-08-06 14:32:14  Setting up table 'planet_osm_polygon'
2025-08-06 14:32:14  Setting up table 'planet_osm_roads'
Processing: Node(262835k 355.2k/s) Way(38560k 112.09k/s) Relation(399600 755.4/s)
2025-08-06 14:59:07  Writing 0 entries to table 'planet_osm_users'...
2025-08-06 14:59:08  Reading input files done in 1614s (26m 54s).
2025-08-06 14:59:08    Processed 262835403 nodes in 740s (12m 20s) - 355k/s
2025-08-06 14:59:08    Processed 38560539 ways in 344s (5m 44s) - 112k/s
2025-08-06 14:59:08    Processed 399742 relations in 530s (8m 50s) - 754/s
2025-08-06 14:59:08  Clustering table 'planet_osm_line' by geometry...
2025-08-06 14:59:08  Clustering table 'planet_osm_roads' by geometry...
2025-08-06 14:59:08  Clustering table 'planet_osm_polygon' by geometry...
2025-08-06 14:59:08  Clustering table 'planet_osm_point' by geometry...
2025-08-06 15:02:26  Creating geometry index on table 'planet_osm_roads'...
2025-08-06 15:02:46  Creating osm_id index on table 'planet_osm_roads'...
2025-08-06 15:02:46  Creating hstore indexes on table 'planet_osm_roads'...
2025-08-06 15:02:58  Creating geometry index on table 'planet_osm_point'...
2025-08-06 15:03:15  Analyzing table 'planet_osm_roads'...
2025-08-06 15:03:21  Done postprocessing on table 'planet_osm_nodes' in 0s
2025-08-06 15:03:21  Building index on table 'planet_osm_ways'
2025-08-06 15:03:37  Creating osm_id index on table 'planet_osm_point'...
2025-08-06 15:04:01  Creating hstore indexes on table 'planet_osm_point'...
2025-08-06 15:05:16  Analyzing table 'planet_osm_point'...
2025-08-06 15:05:20  Building index on table 'planet_osm_rels'
2025-08-06 15:08:00  Creating geometry index on table 'planet_osm_line'...
2025-08-06 15:08:49  Creating osm_id index on table 'planet_osm_line'...
2025-08-06 15:09:03  Creating hstore indexes on table 'planet_osm_line'...
2025-08-06 15:09:18  Creating geometry index on table 'planet_osm_polygon'...
2025-08-06 15:10:27  Creating osm_id index on table 'planet_osm_polygon'...
2025-08-06 15:10:50  Creating hstore indexes on table 'planet_osm_polygon'...
2025-08-06 15:11:14  Analyzing table 'planet_osm_line'...
2025-08-06 15:13:08  Analyzing table 'planet_osm_polygon'...
2025-08-06 15:13:18  Done postprocessing on table 'planet_osm_ways' in 597s (9m 57s)
2025-08-06 15:13:18  Done postprocessing on table 'planet_osm_rels' in 12s
2025-08-06 15:13:18  All postprocessing on table 'planet_osm_point' done in 371s (6m 11s).
2025-08-06 15:13:18  All postprocessing on table 'planet_osm_line' done in 731s (12m 11s).
2025-08-06 15:13:18  All postprocessing on table 'planet_osm_polygon' done in 844s (14m 4s).
2025-08-06 15:13:18  All postprocessing on table 'planet_osm_roads' done in 252s (4m 12s).
2025-08-06 15:13:18  Storing properties to table '"public"."osm2pgsql_properties"'.
2025-08-06 15:13:18  osm2pgsql took 2464s (41m 4s) overall.
OSM data imported successfully.
(myenv) markfinlay@Marks-MacBook-Pro-2 TrailSurfNET %
```

Figure 2: `setup_db.py` script progress

Caveats: The script has a hard-coded value for the `DB_USER` set to "markfinlay".

4.3 Dataset Restoration from Pre-compiled Dump

For convenience and to ensure reproducibility, a complete SQL dump of the compiled training and validation dataset has been made available. This allows an examiner to bypass the data collection and preprocessing steps outlined in the 'setup_db.py' script and restore the database to the exact state used for the experiments.

4.3.1 Download the Dataset

The SQL dump file can be downloaded from the following Google Drive link:

Download Compiled Dataset

Please download the file and place it in a convenient directory on your local machine. The file is in a PostgreSQL custom-format archive.

4.3.2 Restoring the Database using pg_restore

The 'pg_restore' utility is a standard PostgreSQL client application used to restore a database from an archive created by 'pg_dump'.

Prerequisites:

- You must have PostgreSQL installed on your system.
- You must have already created a target database. If you have not, you can create one using the command: `createdb -U your_username osm`

Steps to Restore:

1. Open a terminal or command prompt. 2. Navigate to the directory where you downloaded the SQL dump file. 3. Execute the following command, replacing the placeholders with your own details:

```
pg_restore --dbname=osm --username=your_username --verbose your_dump_file
```

Command Breakdown:

- `--dbname=osm`: Specifies the name of the database to restore into. This should be an existing, empty database.
- `--username=your_username`: Specifies the PostgreSQL user that has permissions to write to the 'osm' database.
- `--verbose`: This is an optional flag that instructs 'pg_restore' to output detailed progress and status messages. It is highly recommended to see the restoration process.
- `your_dump_file.dump`: The path to the downloaded SQL dump file.

Upon successful completion, the 'osm' database will be populated with all the necessary tables and data required to run the experiment notebooks.

5 Data Preparation

Data Preparation is conducted in 2 separate notebooks (`OSM_Data.ipynb` and `Sentinel_2_Data.ipynb`). They are designed to be run from top to bottom with no interaction required from the user.

5.1 OSM Data Processing (`OSM_Data.ipynb`)

After the `setup_db.py` script has finished the `jupyter notebook` command can be run to start the JupyterLab UI in the browser. In the UI open the `OSM_Data.ipynb` notebook. This notebook connects to the `osm` database and performs the following steps:

1. Select relevant trail geometries from OSM Database (paths, footways, tracks).
2. Harmonize surface tags into 5 classes: asphalt, paved, gravel, mud/dirt, and grass.
3. Segments the trail geometries into 160-meter sections.
4. Creates a balanced dataset by down-sampling the majority classes.
5. Store final balanced segments database table named `final_balanced_trail_segments`.

5.1.1 Select relevant trail geometries from OSM Database

A sub selection of the original `planet_osm_line` data was made to narrow down the data to hiking trail data. The main clause being `WHERE (highway IN ('path', 'footway', 'track') OR route = 'hiking')`. This was also coupled with a other additional clauses to reduce the amount of null fields and also to ensure any selected lines were longer than 160m.

```
# SQL query to create the new table with the filtered data
create_new_table_query = f"""
DROP TABLE IF EXISTS {new_filtered_table_name}; -- Optional: Drop if it exists, for idempotency

CREATE TABLE {new_filtered_table_name} AS
SELECT
    osm_id,
    name,
    surface,
    highway,
    route,
    width,
    tags,
    way,
    ST_Length(way) AS length_m
FROM
    planet_osm_line
WHERE
    (highway IN ('path', 'footway', 'track') OR route = 'hiking')
    AND way IS NOT NULL
    AND NOT ST_IsEmpty(way)
    AND ST_Length(way) > 160
    AND surface IS NOT NULL;
"""
```

Figure 3: Initial sub selection of `planet_osm_line` data - OSM Data.ipynb cell 4

An occurrence count for each unique value in the surface column reveals that we have 280+ unique values. The next step is to map these 280+ various surface tags into one of 5 possible tags: `asphalt, paved, gravel, mud/dirt, grass` via a SQL CASE statement (see figure 4).

Once the `surface_group` column mapping has been complete we inspect the count for each unique value type in the column:

In order to balance this data it was decided that the all classed would be down sampled to 30,000 and stored in a new table `balanced_planet_osm_line`.

After the initial filtering the the data contained in `filtered_planet_osm_line` is visualized on a using `geopandas explore` function in cell 8.

The next crucial step is to generate the 160m x 160m trail segments calculating the center of each segment store this data in a new table called `trail_segments` see `OSM`

```python
from sqlalchemy import create_engine, text

with engine.begin() as conn:
    # a) Add the column if it doesn't already exist
    print("Adding 'surface_group' column to 'filtered_planet_osm_line' ...")
    conn.execute(text("ALTER TABLE filtered_planet_osm_line ADD COLUMN IF NOT EXISTS surface_group text;"))

    # b) Update the column using a CASE expression
    conn.execute(text("""
        UPDATE filtered_planet_osm_line
        SET surface_group = CASE
            WHEN lower(surface) IN (
                'asphalt', 'asphaltsidewalk', 'asphalt;paved',
                'tarmac', 'bituminous', 'macadam'
            )
            THEN 'asphalt'
            WHEN lower(surface) IN (
                 'concrete', 'concrete asphalt', 'concrete:plates',
                 'concrete:lanes', 'concrete_slabs', 'concrete_with_bitumen', 'cement',
                'unpaved/paved', 'paved', 'paving_stones', 'paving_stones:30',
                'brick', 'bricks', 'paving_stones', 'chipseal',
                'cobblestone', 'cobblestone:flattened'
            )
            THEN 'paved'

            WHEN lower(surface) IN (
                 'gravel', 'fine_gravel', 'gravel,_dirt', 'gravel_and_rock', 'aggregate',
                 'sand/gravel/stones', 'ground;gravel', 'compacted',  'sand;pebbles'
               )
            THEN 'gravel'

            WHEN lower(surface) IN (
                 'dirt', 'dust', 'earth', 'earth,_grass', 'dirt/sand',
                 'dirt; gravel; dirt', 'unpaved',
                 'ground', 'clay', 'soil'
               )
            THEN 'mud/dirt'

            WHEN lower(surface) IN (
                 'grass', 'grass/trees', 'grass_paver', 'green'
               )
            THEN 'grass'

            ELSE NULL
        END
        WHERE surface IS NOT NULL;
    """))
```

Figure 4: `surface` column to `surface_group` mapping

:

	surface_group	count
0	mud/dirt	104211
1	asphalt	62385
2	gravel	47213
3	grass	46967
4	paved	35612
5	None	5934

Figure 5: Count of instances per surface_group before balancing

Figure 6: `filtered_planet_osm_line` visualized on a map with `geopandas explore`

`Data.ipny cell 18`. Figure 7 shows the code that processes `balanced_planet_osm_line` data into smaller, square-shaped segments identified by a center coordinate. It is designed to ensure that no segments overlap.

The code prepares a new, empty database table called `trail_segments`, which is de-

signed to store the output. This table will hold information for each new segment, including a unique ID, the original trail's ID, its surface type, and the geographic coordinates of its center point. It loads the trail data (represented as lines) from another database table into a GeoDataFrame, a specialized data structure for managing geographic information.

A loop traverses each trail line to place 160x160 meter square tiles separated by `STRIDE`. For each trail, it starts at the beginning and proposes a square centered on the path. It then checks if this proposed square would overlap with any square it has already saved. If there's an overlap, the script nudges its position forward by a small amount (10 meters) and tries again. If there is no overlap, the square is accepted, its details are saved to the `trail_segments` database table, and the script makes a large jump forward along the trail to find the next candidate spot. This process repeats until the entire length of every trail has been covered by these non-overlapping square tiles.

```
# ── parameters ──────────────────────────────────────────────────────
TILE_SIDE = 160.0           # m  – edge of the square
HALF      = TILE_SIDE / 2.0 # 80 m
STRIDE    = TILE_SIDE * sqrt(2)   # 226.274 m  (optimistic jump)
STEP_MIN  = 10.0            # m  – nudge if candidate overlaps

# ── load balanced ways (already EPSG:3035) ──────────────────────────
balanced_sql = """
SELECT osm_id, surface_group, way
FROM   balanced_planet_osm_line;
"""
balanced_gdf = (
    gpd.read_postgis(balanced_sql, engine, geom_col="way")
       .to_crs(3035)
)
print(f"Read {len(balanced_gdf)} OSM ways")

insert_sql = text("""
INSERT INTO trail_segments
      (filename,  osm_id,  surface_group,  center_geom, image_data)
VALUES (:filename, :osm_id, :surface_group,
        ST_SetSRID(ST_MakePoint(:x, :y), 3035),
        NULL)
""")

# ── tiling loop ─────────────────────────────────────────────────────
rt_idx = index.Index()       # spatial index of accepted squares
tile_id_global = 0           # for filename enumeration

with engine.begin() as conn:
    for _, row in balanced_gdf.iterrows():
        line   = row["way"]
        length = line.length
        pos    = 0.0

        while pos < length:
            # candidate centre & square
            centre = line.interpolate(pos)
            sq     = box(
                centre.x - HALF, centre.y - HALF,
                centre.x + HALF, centre.y + HALF
            )

            # if overlaps any previous square → nudge forward STEP_MIN
            if list(rt_idx.intersection(sq.bounds)):
                pos += STEP_MIN
                continue

            # accept tile
            rt_idx.insert(tile_id_global, sq.bounds)
            tile_id_global += 1

            params = {
                "filename":      f"{row.osm_id}_seg_{tile_id_global}",
                "osm_id":        int(row.osm_id),
                "surface_group": row.surface_group,
                "x":             centre.x,
                "y":             centre.y,
            }
            conn.execute(insert_sql, params)

            # optimistic jump ahead by STRIDE
            pos += STRIDE

print(f"Inserted {tile_id_global} non-overlapping 160 m tiles into trail_segments.")
```

Figure 7: Calculation and persistence of `trail_segments`

The last cells in (`OSM_Data.ipynb`) deal with balancing the `trail_segments` by number of segments per `surface_group` and persists this balanced dataset in another table called `final_balanced_trail_segments`. this is done my down-sampling all classes down to `target_segment_count` which is the minimum of segment counts per group. Finally the segments are visualized.

5.2 Sentinel-2 Imagery Acquisition (`Sentinel_2_Data.ipynb`)

This notebook encapsulates the work of retrieving the Sentinel 2 satellite imagery from the Sentinel-hub API. Due to rate limits and the volume of data being transferred this process can take + 10 hours .

5.2.1 Sentinel Hub Credentials

The notebook requires API credentials to access the Sentinel Hub API. You can create a free account and API credentials at: `https://www.sentinel-hub.com/`. Once you have created an account and signed in you can navigate to your setting page (`https://apps.sentinel-hub.com/dashboard/#/account/settings` to create OAuth Credentials (see figure 8.

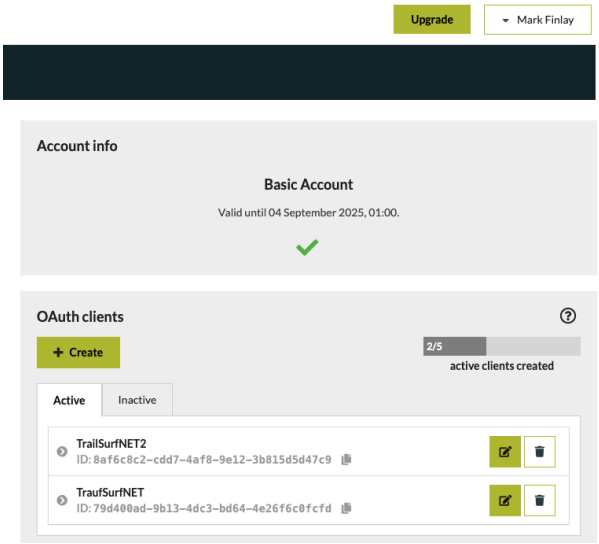

Figure 8: Sentinel Hub OAuth clients page

Create a `.env` file in the project's root directory with your credentials:

```
SH_CLIENT_ID="your_client_id"
SH_CLIENT_SECRET="your_client_secret"
```

5.2.2 Image Retrieval

The notebook iterates through each segment in the `final_balanced_trail_segments` table, calculates a 160 × 160 meter bounding box around the segment's centroid, and downloads the corresponding 9-band Sentinel-2 L2A imagery.

evalscript The evalscript shown in figure 9 is a set of instructions for the Sentinel Hub defining the data to be returns. It requests 8 different spectral bands from the Sentinel-2 satellite. These include normal colors (Red, Green, Blue) as well as non-visible bands like Near-Infrared (NIR) and Short-Wave Infrared (SWIR), which are useful for analyzing vegetation and moisture. It also requests a dataMask, which tells us if a pixel contains real data or is empty. It specifies the final image should combine these into a single file with 9 bands of data.

```
# — evalscript: 8 spectral + dataMask = 9 bands total ————————
evalscript = """
//VERSION=3
function setup() {
  return {
    input:["B02","B03","B04","B05","B06","B08","B11","B12","dataMask"],
    output:{bands:9, sampleType:"FLOAT32", id:"spec"}
  };
}
function evaluatePixel(s){
  const f = 8.0;
  return [           // 8 spectral
    s.B04*f, s.B03*f, s.B02*f,   // RGB
    s.B05,  s.B06,               // red-edge
    s.B08,                       // NIR
    s.B11, s.B12,                // SWIR
    s.dataMask                   // + mask = 9 bands
  ];
}
"""
```

Figure 9: Sentinel Hub Evalscript

5.2.3 Image Storage

The downloaded image data is stored as a byte array in the `image_data` column of the `final_balanced_trail_segments` table.

5.2.4 Inspecting Retrieved Images

Finally after all the images have been retrieved 30 images are visualized. From a single 9-band data array, it generates four distinct images, each revealing different information.

6 Model Training and Evaluation

The project evaluates several CNN architectures. Each model is trained and evaluated using a dedicated Jupyter Notebook.

- `VggCNN.ipynb`
- `ResNET18_Scratch.ipynb`
- `ResNET34_Scratch.ipynb`
- `ResNET50_Scratch.ipynb`
- `ResNET18_BigEarthNet.ipynb`
- `ResNET50_BigEarthNet.ipynb`

6.1 Model Configurations

The following section outlines the configuration and hyperparameters used for each of the five deep learning models in this research. All models were trained for a maximum of 60 epochs, with an early stopping mechanism configured with a patience of 12 epochs to prevent overfitting. The Adam optimizer was used consistently across all experiments. A batch size of 32 was used for training and validation.

All models developed in this research share a consistent set of core hyperparameters to ensure a fair comparison of their architectures. Each model was trained for a maximum of 60 epochs, utilizing an early stopping mechanism with a patience of 12 epochs to prevent overfitting. The Adam optimizer was employed with a learning rate of 0.001 and a weight

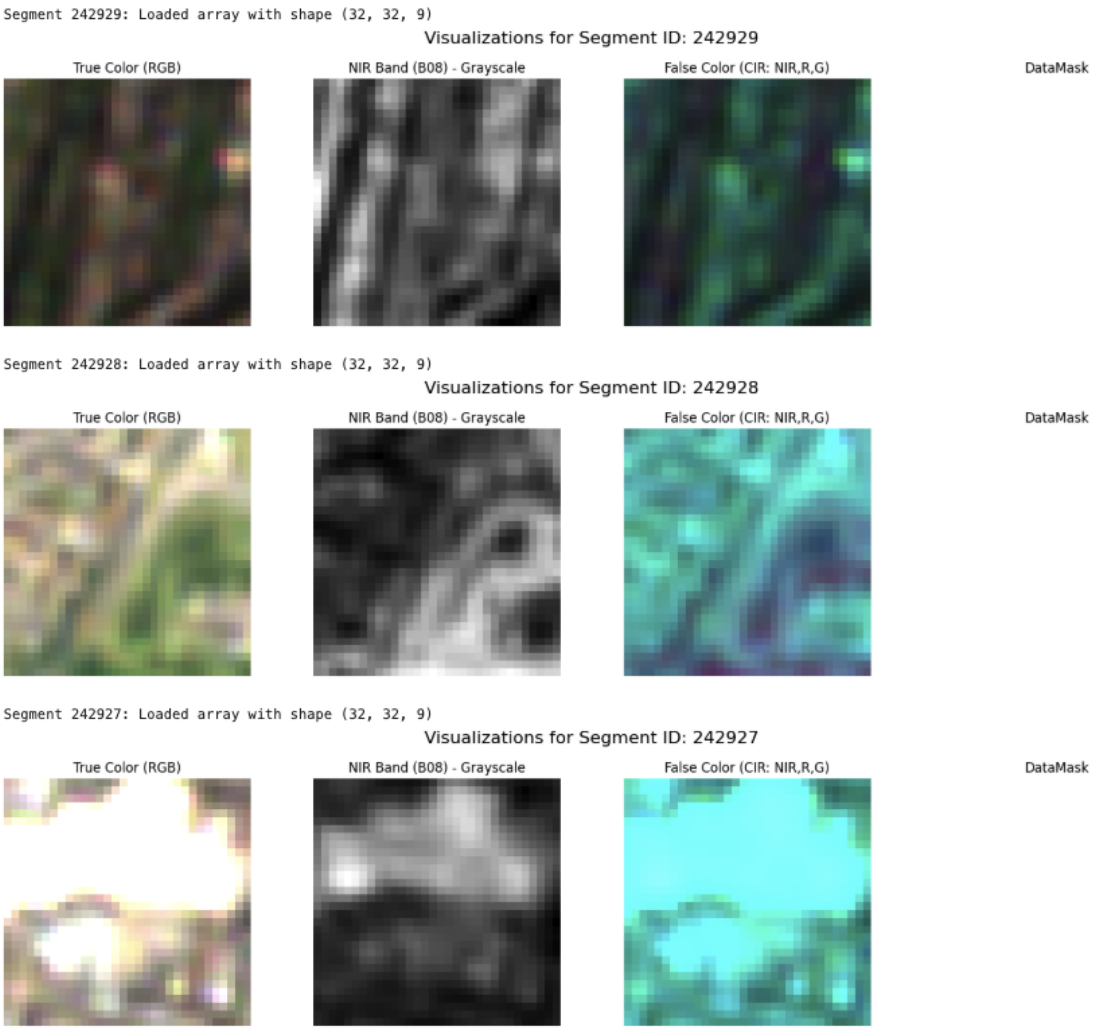

Figure 10: Inspecting Satellite Images

decay of 1e-4. A batch size of 32 and the Cross-Entropy loss function were used for all training and validation routines.

- **Models from Scratch:** The VGG, ResNet18, ResNet34, and ResNet50 models were all built from scratch. Their initial convolutional layer was specifically designed to accept a 9-channel input tensor, corresponding to the available spectral bands.
- **Pre-trained ResNet50:** The pre-trained ResNet50 model was sourced from the Hugging Face model hub, specifically the `BIFOLD-BigEarthNetv2-0/resnet50-all-v0.2.0` checkpoint. The original model was pre-trained on the BigEarthNet dataset with a 12-channel input. To align with this project's 9-channel dataset, the weights of the first convolutional layer were adapted by averaging across specific channels to reduce the input dimension from 12 to 9.

Each notebook follows a similar structure:

1. **Data Loading:** A custom `TrailSegmentDataset` class loads the image data and labels from the PostgreSQL database.

2. **Model Definition:** The respective CNN architecture is defined. For models using BigEarthNet pre-trained weights, the notebook will automatically download the weights from the Hugging Face Hub.

3. **Training Loop:** The model is trained for a specified number of epochs with early stopping based on validation accuracy. The best-performing model weights are saved to a `.pth` file.

4. **Evaluation:** The trained model is evaluated on the validation set, and a confusion matrix is generated to visualize performance across the different surface classes.

To run an experiment, simply execute the cells in the desired notebook. The training process will begin, and the results, including the saved model and performance metrics, will be stored in the project directory.

6.2 Example Walkthrough: Training the Pre-trained ResNet18 Model

This section provides a step-by-step guide to running the `ResNET18_BigEarthNet.ipynb` notebook. This notebook trains a ResNet18 model that has been pre-trained on the BigEarthNet dataset, adapting it for the specific task of trail surface classification.

6.2.1 Cell 1: Initial Setup and Imports

The first cell of the notebook imports all necessary libraries. This includes standard libraries like `torch` for deep learning, `SQLAlchemy` for database interaction, `scikit-learn` for data splitting and metrics, and `numpy` for numerical operations. Crucially, it also imports `hf_hub_download` and `load_file` from the Hugging Face Hub and safetensors libraries, which are used to download and load the pre-trained model weights. This cell also defines several key constants for the training loop:

- `epoch_count`: The maximum number of training epochs (60).
- `best_val_accuracy`: A variable to track the best validation accuracy achieved, initialized to 0.0.
- `patience_early_stop`: The number of epochs to wait for an improvement in validation accuracy before stopping the training (12).
- `patience_counter`: A counter to track epochs with no improvement.
- `model_save_path`: The filename for saving the best model weights.

6.2.2 Cell 2: Custom ResNet-18 Model Definition

This cell defines the `create_resnet18_for_9_channels_bigearthnet_hf` function, which customizes a ResNet-18 architecture for this project. The function performs four key steps:

1. It instantiates a standard `resnet18` model from `torchvision` without any pre-trained weights.

```python
# --- Imports ---
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torchvision.models as models
from sqlalchemy import create_engine, text
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import numpy as np
import os
import io
import json
from collections import defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
from huggingface_hub import hf_hub_download
from safetensors.torch import load_file

# --- Training Loop Consts ---
epoch_count = 60
best_val_accuracy = 0.0
patience_early_stop = 12
patience_counter = 0
model_save_path = "best_resnet50_BigEarthNET_model.pth"
```

Figure 11: Notebook Imports and Training Loop Constants

2. It downloads and loads the weights of a ResNet-18 model pre-trained on the BigEarthNet dataset, which uses all 12 Sentinel-2 bands. The weights are loaded in a non-strict mode to accommodate layer modifications.

3. It replaces the model's initial convolutional layer (`conv1`). The original layer expects a 3-channel (RGB) input. The new layer is configured to accept a 9-channel input to match our Sentinel-2 data. The weights for the new channels are initialized by averaging the weights of the original three channels and repeating them.

4. It replaces the final fully connected layer (`model.fc`), which acts as the classifier. The original classifier is replaced with a new linear layer that has an output size equal to the number of classes in our trail surface dataset (5).

6.2.3 Cell 3: Database Helper Function and TrailSegmentDataset class

This cell contains the data loading logic. The `get_initial_records_and_map` function connects to the PostgreSQL database, retrieves the IDs and surface group labels for all segments that have associated image data, and creates a numerical mapping for the text-based labels (e.g., 'asphalt': 0). It then uses `train_test_split` to stratify and divide these records into training and validation sets.

The `TrailSegmentDataset` class is a custom PyTorch `Dataset`. Its `__getitem__` method is designed to fetch a single image's byte data from the database by its ID, load it as a NumPy array, and apply any specified transformations. This lazy-loading approach is memory-efficient as it only loads images into memory when they are needed for a batch.

```python
def create_resnet18_for_9_channels_bigearthnet_hf(num_classes=5):
    """
    Loads ResNet-18 pretrained on BigEarthNet v2.0 (all Sentinel bands),
    adapts for 9-channel input and `num_classes`.
    """
    # 1) Instantiate without any weights
    model = resnet18(weights=None)

    # 2) Download & load the BigEarthNet v2.0 "all bands" weights (safetensors format)
    repo_id = "BIFOLD-BigEarthNetv2-0/resnet18-all-v0.2.0"
    safetensors_path = hf_hub_download(repo_id=repo_id, filename="model.safetensors")
    state_dict = load_file(safetensors_path)
    model.load_state_dict(state_dict, strict=False)

    # 3) Replace first conv layer to accept 9 channels instead of 3
    orig = model.conv1
    new_conv1 = nn.Conv2d(
        in_channels=9,
        out_channels=orig.out_channels,
        kernel_size=orig.kernel_size,
        stride=orig.stride,
        padding=orig.padding,
        bias=(orig.bias is not None)
    )
    with torch.no_grad():
        averaged_weights = orig.weight.mean(dim=1, keepdim=True)
        new_conv1.weight.copy_(averaged_weights.repeat(1, 9, 1, 1))
        if orig.bias is not None:
            new_conv1.bias.copy_(orig.bias)
    model.conv1 = new_conv1

    # 4) Replace the classifier for `num_classes`
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)

    return model
```

Figure 12: Cell 2 - Custom ResNet-18 Model Definition

```python
# --- Helper function and Dataset Class ---
def get_initial_records_and_map(db_url_str):
    engine = create_engine(db_url_str)
    all_records_from_db, label_map, train_records_list, val_records_list = [], {}, [], []
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT id, surface_group FROM final_balanced_trail_segments WHERE image_data IS NOT NULL AND surface_
            all_records_from_db = result.fetchall()
        if not all_records_from_db: return [], [], {}
        unique_labels = sorted(list(set(r[1] for r in all_records_from_db)))
        label_map = {label: idx for idx, label in enumerate(unique_labels)}
        all_records_with_labels = [(r[0], label_map[r[1]]) for r in all_records_from_db]
        train_records_list, val_records_list = train_test_split(
            all_records_with_labels, test_size=0.2, random_state=42,
            stratify=[r[1] for r in all_records_with_labels] if len(all_records_with_labels) > 1 else None
        )
        print(f"Pre-fetched data: {len(train_records_list)} train, {len(val_records_list)} val records. Label map size: {len(label_map)}")
    except Exception as e: print(f"Error during initial data fetch: {e}")
    finally:
        if engine: engine.dispose()
    return train_records_list, val_records_list, label_map

class TrailSegmentDataset(Dataset):
    def __init__(self, db_url, records_list, label_map, transform=None, for_stats_calc=False):
        self.db_url, self.engine, self.records, self.label_map, self.transform, self.for_stats_calc = \
            db_url, None, records_list, label_map, transform, for_stats_calc
    def _get_engine(self):
        if self.engine is None: self.engine = create_engine(self.db_url)
        return self.engine
    def __len__(self): return len(self.records)
    def __getitem__(self, idx):
        seg_id, label = self.records[idx]
        try:
            with self._get_engine().connect() as conn:
                res = conn.execute(text("SELECT image_data FROM final_balanced_trail_segments WHERE id = :id"), {"id": seg_id})
                image_bytes = res.scalar()
            if image_bytes is None: return (None, label) if self.for_stats_calc else (_ for _ in ()).throw(ValueError(f"Img None for id {seg_
            image_array = np.load(io.BytesIO(image_bytes)).astype(np.float32)
            return (self.transform(image_array) if self.transform else image_array), label
        except Exception as e:
            print(f"Error in __getitem__ for seg_id {seg_id}: {e}")
            return (None, label) if self.for_stats_calc else (_ for _ in ()).throw(e)
```

Figure 13: Database Helper Function and TrailSegmentDataset class

6.2.4 Cell 4: Data Preparation and Normalization

This cell orchestrates the entire data preparation pipeline. It first calls `get_initial_records_and_map` to get the training and validation record lists. A key step performed here is the calculation of the channel-wise mean and standard deviation across the entire training dataset.

```python
# --- Main Data Preparation ---
db_url = os.getenv('DATABASE_URL', "postgresql://markfinlay:@localhost:5432/osm")

train_img_records, val_img_records, global_label_map = get_initial_records_and_map(db_url)

# --- Normalization Calculation ---
print("Calculating mean and std for normalization...")
temp_train_dataset = TrailSegmentDataset(db_url, train_img_records, global_label_map, for_stats_calc=True)
all_imgs = [img for img, _ in temp_train_dataset if img is not None]
all_imgs_np = np.stack(all_imgs, axis=0)
channel_means = np.mean(all_imgs_np, axis=(0, 1, 2))
channel_stds = np.std(all_imgs_np, axis=(0, 1, 2))
channel_stds = np.maximum(channel_stds, 1e-6)
print(f"Means: {channel_means.tolist()}\nStds: {channel_stds.tolist()}")

# --- Transforms and DataLoaders ---
train_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=channel_means.tolist(), std=channel_stds.tolist()),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
])
val_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=channel_means.tolist(), std=channel_stds.tolist()),
])

train_dataset = TrailSegmentDataset(db_url, train_img_records, global_label_map, transform=train_transform)
val_dataset = TrailSegmentDataset(db_url, val_img_records, global_label_map, transform=val_transform)

num_workers = 0
pin_memory = False
if torch.cuda.is_available(): pin_memory = True

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=num_workers, pin_memory=pin_memory)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=num_workers, pin_memory=pin_memory) if val_dataset else None
```

```
Pre-fetched data: 98096 train, 24524 val records. Label map size: 5
Calculating mean and std for normalization...
Means: [0.2704830765724182, 0.2651905417442322, 0.24170875549316406, 0.04852844402194023, 0.08895786851644516, 0.10506097972393036, 0.067018
5536146164, 0.043866924941539764, 0.1670200675725937]
Stds: [0.657516598701477, 0.6269733309745789, 0.6625603437423706, 0.08884993195533752, 0.1327875852584839, 0.16703945398330688, 0.0921852365
1361465, 0.061533838510513306, 0.40868088603019714]
```

Figure 14: Data Preparation and Normalization

This is done by creating a temporary dataset instance and iterating through all images. These statistics are essential for normalizing the input data, which is a standard practice that helps stabilize and speed up model training. Finally, it defines the data augmentation and transformation pipelines (`train_transform` and `val_transform`) using these calculated statistics and creates the PyTorch `DataLoaders` that will feed batches of data to the model during training.

6.2.5 Cell 5: Model Training Loop

This is the core execution cell for training the model. It begins by setting up the training device (MPS, CUDA, or CPU), instantiating the model using the function from Cell 2, and defining the Adam optimizer. A weighted `CrossEntropyLoss` is used as the criterion to counteract class imbalance in the dataset. A `ReduceLROnPlateau` scheduler is also initialized to automatically reduce the learning rate if the validation loss plateaus.

The main training loop then iterates for the number of epochs defined in Cell 1. In each epoch, it performs a training phase (forward pass, loss calculation, backpropagation, and optimizer step) followed by a validation phase. After validation, it checks if the validation accuracy has improved. If it has, the model's state dictionary is saved. If there is no improvement for a number of epochs equal to `patience_early_stop`, the training is halted.

```
# --- Setup for Training ---
device = torch.device("mps" if torch.backends.mps.is_available() else "cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

num_classes = len(global_label_map)
model = create_resnet18_for_9_channels_bigearthnet_hf(num_classes).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)

# Class weights
from collections import Counter
train_labels = [label for _, label in train_img_records]
counts = Counter(train_labels)
total = sum(counts.values())
weights = torch.tensor([total / (num_classes * counts.get(i, 1)) for i in range(num_classes)], dtype=torch.float).to(device)
criterion = nn.CrossEntropyLoss(weight=weights)

# Learning Rate Scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=5)

print("--- Starting resnet18_bigearth Training ---")
for epoch in range(epoch_count):
    model.train()
    running_loss, train_correct, train_total = 0.0, 0, 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
        train_correct += (outputs.argmax(dim=1) == labels).sum().item()
        train_total += labels.size(0)

    epoch_loss = running_loss / train_total
    print(f"Epoch [{epoch+1}/{epoch_count}] - Train Loss: {epoch_loss:.4f}, Train Acc: {(100*train_correct/train_total):.2f}%")

    if val_loader:
        model.eval()
        val_loss, val_correct, val_total = 0.0, 0, 0
        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                val_loss += criterion(outputs, labels).item() * images.size(0)
                val_correct += (outputs.argmax(dim=1) == labels).sum().item()
                val_total += labels.size(0)

        epoch_val_loss = val_loss / val_total
        epoch_val_accuracy = 100 * val_correct / val_total
        print(f"→ Validation Loss: {epoch_val_loss:.4f}, Validation Acc: {epoch_val_accuracy:.2f}%")

        # Scheduler step
        scheduler.step(epoch_val_loss)

        if epoch_val_accuracy > best_val_accuracy:
            best_val_accuracy = epoch_val_accuracy
            patience_counter = 0
            torch.save(model.state_dict(), model_save_path)
            print(f"✓ Saved new best model to {model_save_path} with accuracy: {best_val_accuracy:.2f}%")
        else:
            patience_counter += 1

        if patience_counter >= patience_early_stop:
            print(f"Early stopping triggered after {patience_early_stop} epochs of no improvement.")
            break
print("--- resnet18_bigearth Training complete ---")
```

Figure 15: Model Training Loop

6.2.6 Cell 6: Saving Training History

This cell captures the training and validation loss and accuracy metrics logged during the training loop in Cell 5. The metrics for each epoch are hardcoded into a dictionary and then serialized to a JSON file named `resnet18_bigearth_training_history.json`. This allows for persistent storage of the results, enabling later analysis and visualization without needing to re-run the entire training process.

6.2.7 Cell 7: Plotting Training and Validation Metrics

Using the data saved in the previous step, this cell generates and displays plots of the model's performance over the epochs. It creates two subplots using `matplotlib` and `seaborn`: one for training and validation loss, and the other for training and validation accuracy. This visualization is critical for diagnosing the training process, identifying potential issues like overfitting (where training accuracy is high but validation accuracy is low), and observing the point at which the model's performance converges. The final plot is saved as a PNG file.

```
# --- Plotting Training History ---
plt.style.use('seaborn-v0_8-whitegrid')
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

# Plot 1: Model Loss 🚿
ax1.plot(history['train_loss'], label='Train Loss', color='royalblue', marker='o', linestyle='--')
ax1.plot(history['val_loss'], label='Validation Loss', color='orangered', marker='o', linestyle='-')
ax1.set_title('Model Loss Over Epochs', fontsize=16)
ax1.set_xlabel('Epoch', fontsize=12)
ax1.set_ylabel('Loss', fontsize=12)
ax1.legend()
ax1.grid(True)

# Plot 2: Model Accuracy 🎯
ax2.plot(history['train_acc'], label='Train Accuracy', color='royalblue', marker='o', linestyle='--')
ax2.plot(history['val_acc'], label='Validation Accuracy', color='orangered', marker='o', linestyle='-')
ax2.set_title('Model Accuracy Over Epochs', fontsize=16)
ax2.set_xlabel('Epoch', fontsize=12)
ax2.set_ylabel('Accuracy (%)', fontsize=12)
ax2.legend()
ax2.grid(True)

plt.suptitle('resnet18_bigearth Training and Validation Metrics', fontsize=20)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# Save and show the figure
output_filename = "resnet18_bigearth_training_history.png"
plt.savefig(output_filename, dpi=300)
print(f"Confusion matrix plot saved to '{output_filename}'")
```

Figure 16: Plotting Training and Validation Metrics

6.2.8 Cell 8: Generating the Confusion Matrix

This cell evaluates the best-performing model (saved during the training loop) on the entire validation set. It loads the saved model weights, sets the model to evaluation mode, and iterates through the validation loader to gather all true labels and model predictions. Using these, it computes a confusion matrix with `scikit-learn`. The matrix is then visualized as a heatmap using `seaborn`, showing the counts and percentages of predictions for each true label. This provides a detailed view of the model's performance, highlighting which classes are often confused with one another.

```python
# 1. Initialize and load the best model
# Ensure the model architecture is defined and pass the number of classes
model = create_resnet18_for_9_channels_bigearthnet_hf(num_classes=len(global_label_map))
# Load the state dictionary from your saved model file
model.load_state_dict(torch.load(model_save_path, map_location=device))
# Move the model to the appropriate device (GPU/CPU)
model.to(device)
# Set the model to evaluation mode
model.eval()

# 2. Get predictions on the validation set
all_labels = []
all_preds = []

with torch.no_grad():
    for images, labels in val_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        preds = torch.argmax(outputs, dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# 4. COMPUTE AND PLOT CONFUSION MATRIX
print("Generating confusion matrix...")
cm = confusion_matrix(all_labels, all_preds)
class_names = [item[0] for item in sorted(global_label_map.items(), key=lambda item: item[1])]

plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(12, 10))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues', ax=ax, xticklabels=class_names, yticklabels=class_names)

# Add annotations (counts and percentages for each true label)
for i in range(len(class_names)):
    for j in range(len(class_names)):
        count = cm[i, j]
        row_sum = np.sum(cm[i])
        percentage = f"{(count / row_sum) * 100:0.1f}%" if row_sum > 0 else "0.0%"
        text = f"{count}\n{percentage}"
        # Use a contrasting color for text if the background is dark
        text_color = "white" if cm[i, j] > (cm.max() / 2) else "black"
        ax.text(j + 0.5, i + 0.5, text, ha='center', va='center', color=text_color, fontsize=12)

ax.set_xlabel('Predicted Label', fontsize=14, labelpad=10)
ax.set_ylabel('True Label', fontsize=14, labelpad=10)
ax.set_title('Confusion Matrix for resnet18_bigearth', fontsize=16, pad=20)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()

# Save and show the figure
output_filename = "resnet18_bigearth_confusion_matrix.png"
plt.savefig(output_filename, dpi=300)
print(f"Confusion matrix plot saved to '{output_filename}'")
plt.show()
```

Figure 17: Generating the Confusion Matrix

6.2.9 Cell 9: Generating the Classification Report

To provide a more quantitative evaluation, this cell generates a detailed classification report using `scikit-learn`'s `classification_report` function. The report includes key metrics for each surface class:

- **Precision**: The ratio of correctly predicted positive observations to the total predicted positive observations.
- **Recall**: The ratio of correctly predicted positive observations to all observations in the actual class.

- **F1-Score**: The weighted average of Precision and Recall.

This report offers a comprehensive summary of the model's predictive accuracy for each specific trail surface type.

6.2.10 Cell 10: Visualizing Sample Predictions

The final cell provides a qualitative assessment of the model's performance. It takes a random batch of images from the validation set and displays them alongside their true and predicted labels. A helper function, `denormalize`, is used to reverse the normalization transformation so the images can be displayed with their correct colors. The title for each image is colored green if the prediction was correct and red if it was incorrect, offering an intuitive, visual confirmation of the model's capabilities on individual examples. The resulting grid of images is saved as a PNG file.

AI Usage Disclaimer

This paper was developed with the support of OpenAI's ChatGPT and Google's Gemini. I used these language models extensively throughout the writing process, including rewording sections for clarity and flow, assisting in ideation, organizing the structure of the paper, and interpreting and understanding complex concepts from the literature reviewed. The insights provided by ChatGPT and Gemini were instrumental in enhancing the quality of this paper, although final analysis, interpretations, and conclusions are my own.