mundialis GmbH & Co. KG

Sitz u. Registergericht: Bonn
Amtsgericht Bonn HRA 8528

Steuer-Nr.: 205/5823/1574
Ust.-ID-Nr.: DE300200756

Komplementärin:
mundialis Verwaltungsgesellschaft mbH

vertreten durch:
M. Eichhorn

Dokumentation

vom 29.08.2025

Entwicklung einer Methodik für Training und Anwendung eines Neuronalen Netzes zur Einzelbaumerkennung

2.0.0

erarbeitet für:

Regionalverband Ruhr
Referat Geoinformation und Raumbeobachtung
Team Geodaten, Stadtplanwerk, Luftbilder

Kronprinzenstraße 35
45128 Essen

E-Mail: geodaten@rvr.ruhr

Dieses Projekt wird von der Bezirksregierung Münster aus Mitteln des Ministeriums für Umwelt, Naturschutz und Verkehr des Landes NRW gefördert.

erarbeitet von:

mundialis GmbH & Co. KG
Kölnstraße 99
53111 Bonn

Telefon: +49 228 - 387 580 80
Telefax: +49 228 - 962 899 57

E-Mail: info@mundialis.de
Internet: www.mundialis.de

Ansprechpersonen:
Dr. E. Panzenböck panzenboeck@mundialis.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Änderungen

Tabelle 1: Änderungshistorie des Dokumentes

Datum	Autor:in	Beschreibung
16.01.2025	mundialis	Version v1.0.0
29.08.2025	mundialis	Version v2.0.0: Erweiterung um Training, Evaluierung und Anwendung eines KI-Modells sowie Nutzung für Einzelbaum-Postprozessierung

Diese Dokumentation erläutert die notwendigen Arbeitsschritte zur Entwicklung einer Methodik für Training und Anwendung eines Neuronalen Netzes zur Einzelbaumerkennung.

Mit Stand 1.0.0 der Dokumentation umfasst diese die im Teilprojekt „Entwicklung eines GRASS GIS Addons zur kollaborativen Erstellung von Trainingsdaten für neuronale Netze“ enthaltenen Leistungen. Konkret wurden zwei Addons entwickelt, die die Erstellung von Trainingsdaten ermöglichen und diese für die Verwendung in einem Neuronalen Netz aufbereiten. Diese Prozedur wird ausführlich in Kapitel 2.1 erläutert.

Mit Stand 2.0.0 der Dokumentation wurden weitere Addons zum Trainieren, Testen und Anwenden eines Neuronalen Netzes der Methodik hinzugefügt. Erläuterungen zur Verwendung dieser finden sich in den Kapiteln 2.2 bis 2.4. Weiterhin wurde ein Addon zum Zusammensetzen der klassifizierten Kacheln hinzugefügt, sodass die Einzelbaum-Postprozessierung mit dem angepassten bekannten Addon aus dem RVR-Interface vorgenommen werden kann. Dazu finden sich Beschreibungen in Kapitel 3.

Um Querverweise zu vermeiden und eine eigenständige Dokumentation zu ermöglichen, sind einige Abschnitte aus der Dokumentation_Baumstandorte übernommen. Konkret sind dies große Teile von Abschnitt 1 und 3 sowie der komplette Abschnitt 4. Sofern diese Inhalte als bekannt vorausgesetzt werden können, sind hier die Änderungen hervorgehoben:

Im Dockerfile werden das neue Addon *m.neural_network* sowie notwendige Python Pakete für Neuronale Netze installiert. Das Docker Image muss daher neu gebaut werden (siehe Abschnitt 1.3.2)

- Das Modul *m.import.rvr* ist um die Option ***type***=*“neural network“* erweitert worden, um nur die für das Neuronale Netz benötigten Datensätze in GRASS zu importieren. Dies ist jedoch nur nötig, sofern die Datensätze nicht bereits zuvor mit der Option ***type***=*“trees analysis“* importiert wurden (siehe Abschnitt 1.4.1).
- Die Prozedur zur Erstellung von Trainingsdaten und das Nutzen eines Neuronalen Netzes für die Einzelbaumerkennung sind der Hauptteil dieses Dokuments und in Abschnitt 2 beschrieben.
- Kapitel 3 umfasst wie oben beschrieben die Postprozessierung vom klassifizierten Ergebnis. Dabei werden die Addons aus dem RVR-Interface verwendet, wobei sich der Aufruf des *r.trees.postprocess* vom klassischem Random Forest Ansatz unterscheidet.

1 Vorbereitung

1.1 Testumgebung

Die Software zur vorliegenden Dokumentation wurde auf einem Linux System mit 16 GB RAM, 16 CPUs und dem GRASS 8.5 Ubuntu GUI Docker Image getestet. Die für KI-Anwendung empfohlene Nutzung einer GPU wurde mit einer NVIDIA GeForce RTX getestet.

1.2 Skripte und GRASS-Addons herunterladen

Für den Import und die Postprozessierung werden die Addons aus dem rvr_interface GitHub Repository genutzt und können hier heruntergeladen werden: https://github.com/mundialis/rvr_interface.

Die notwendigen Skripte und GRASS-Addons für den Neuronalen Netz Ansatz können aus dem folgenden GitHub Repository heruntergeladen werden: https://github.com/mundialis/m.neural_network.

Durch Klick auf den grünen „Code“-Button oben rechts kann das Repository entweder für die Nutzung mit der Git-Software geklont werden:

```
# Im Zielverzeichnis ausführen:
$ git clone https://github.com/mundialis/m.neural_network.git
$ git clone https://github.com/mundialis/rvr_interface.git

# bzw. wenn ein SSH-Key auf GitHub hinterlegt wurde:
$ git clone git@github.com:mundialis/m.neural_network.git
$ git clone git@github.com:mundialis/rvr_interface.git
```

oder als ZIP-Datei heruntergeladen und auf dem Rechner entpackt werden. Das Klonen des Repositorys hat den Vorteil, dass es einfach aktualisiert werden kann, während beim Weg über die ZIP-Datei immer das ganze Repository neu herunterzuladen ist.

Ab der Version 7.3.0 ist die Methodik zur Erstellung von Trainingsdaten enthalten.

Ab der Version 7.4.0 ist die Methodik zum Trainieren, Testen und Anwenden eines Neuronalen Netzes, sowie dessen Nutzung für die Postprozessierung enthalten.

1.3 Benötigte Software

Das Vorbereiten der Daten und der manuell zu labelnden Kacheln für das Erstellen der Trainingsdaten erfolgt in **GRASS GIS**. Das Labeln der vorbereiteten Kacheln ist in **QGIS** vorzunehmen.

GRASS GIS, oder kurz GRASS (*Geographic Resources Analysis Support System*) ist ein kostenloses Open-Source Geographisches Informationssystem (GIS). Mit seinem großen Funktionsumfang wird es für die Verwaltung und Analyse von Geodaten, die Bildverarbeitung, die Erstellung von Grafiken und Karten, die räumliche Modellierung und Visualisierung verwendet. GRASS wird der-

zeit in akademischen und kommerziellen Einrichtungen auf der ganzen Welt sowie von vielen Regierungsbehörden und Umweltberatungsunternehmen eingesetzt.

Um für KI-Anwendungen, vor allem das Trainieren eines KI-Modells, GPU Ressourcen benutzen zu können, sind ein paar Vorbereitungen auf dem Host-System notwendig. Auf dem Host-System muss der NVIDIA-GPU Treiber installiert werden, siehe https://docs.nvidia.com/datacenter/tesla/driver-installation-guide/index.html. Auf dem Host-System muss zudem das NVIDIA container toolkit installiert werden, siehe https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html.

Der Aufruf des Docker Images erfordert die zusätzlichen Parameter ***--runtime=nvidia --gpus all*** damit im Docker auch auf die Host-GPU zugegriffen werden kann. Um zu testen, ob die GPU auch im Docker Image erkannt wird, kann

```
sudo docker run --rm --runtime=nvidia --gpus all ubuntu nvidia-smi
```

ausgeführt werden. Dies sollte Informationen über die GPU und den NVIDIA Treiber ausgeben, siehe als Beispiel https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/sample-workload.html.

1.3.1 GRASS GIS-Basics

Zur Nutzung von GRASS GIS sind einige grundlegende Kenntnisse über die Software nötig.

GRASS GIS unterscheidet sich von anderen GIS-Anwendungen wie ESRI ArcGIS oder QGIS, in denen die Nutzenden sofort verschiedene Daten aus verschiedenen Datenquellen in verschiedenen Projektionen laden und mit der Arbeit an einem Projekt beginnen können. Beim Starten von GRASS GIS müssen die Nutzenden zunächst das Arbeitsprojekt definieren, in dem die GRASS-Sitzung arbeiten soll.

Beim Start von GRASS GIS sind drei Voraussetzungen erforderlich (über den Startbildschirm oder die Kommandozeile einrichtbar):

- **Datenbankverzeichnis:** Ein Verzeichnis auf der lokalen oder Netzwerkplatte, das alle Daten enthält, auf die GRASS GIS zugreift. Dies ist üblicherweise ein Verzeichnis namens „*grassdata*“, das sich im individuellen Home-Verzeichnis befindet.
- ***Project*** *(früher Location)***:** Spielt die Rolle eines "Projekts". Alle Geodaten, die innerhalb eines *Project* gespeichert sind, müssen das gleiche räumliche Koordinatensystem haben (GRASS GIS unterstützt aus verschiedenen Gründen keine *on-the-fly*-Projektion, kann aber beim Import reprojizieren).
- ***Mapset*****:** Enthält aufgabenbezogene Daten innerhalb eines Projekts. Dies unterstützt die Organisation der Daten in logischen Gruppen oder ermöglicht die parallele Arbeit mehrerer Nutzenden am selben Projekt.

Ein weiterer Unterschied zu anderen GIS-Anwendungen ist das Konzept der ***Computational Region*****.** Diese legt für Berechnungen und Analysen von Rasterdaten den räumlichen Bereich fest, für den die Berechnung durchgeführt wird. Das heißt, mit der *Region* wird die Ausdehnung und Auflösung des Ergebnisrasters bestimmt. Die aktive *Region* kann mit dem GRASS-Befehl *g.region* festgelegt werden.

Zum weiteren Verständnis der GRASS GIS-Basics kann zum Beispiel dieses Intro oder diese GRASS Manual-Seite konsultiert werden.

Weitere hilfreiche Links zum Umgang mit GRASS sind:

- GRASS GIS Homepage
- GRASS GIS 8.3 Reference Manual
- GRASS User Wiki Installation Guide
- GRASS GIS Wiki
- GRASS GIS Tutorial

1.3.2 Einrichten und Starten von GRASS in Docker unter Linux und Windows

Im ersten Schritt muss zunächst GRASS GIS eingerichtet werden. Alle Daten, die in GRASS GIS verwendet werden, liegen in einer Datenbank (*grassdb*) in GRASS-spezifischen Formaten. Diese Datenbank kann als normaler Ordner neu erstellt werden. Im gewünschten Verzeichnis, in dem die GRASS-Datenbank angelegt werden soll (Achtung: diese kann im Laufe der Analyse sehr groß werden!), muss ein neuer Ordner erstellt werden (üblicherweise als *„grassdata“* im Home-Verzeichnis):

```
$ mkdir grassdata
```

Die Erstellung der Datenbank ist nur einmalig zu Beginn notwendig. Besteht bereits eine *grassdb*, kann auch diese verwendet werden.

Zur Nutzung der Baummodule wird GRASS GIS in Docker verwendet. Hierfür muss Docker auf dem System installiert sein. Genutzt wird das offizielle und öffentlich zur Verfügung stehende Docker_Image GRASS 8.5 Ubuntu. Es enthält alle für die Baummodule notwendigen Abhängigkeiten wie Python3, GDAL und PDAL. Dafür muss zunächst ein lokales Docker Image gebaut werden. Dies geschieht im Wurzelverzeichnis des RVR-Interfaces, in dem auch ein Dockerfile liegt. Dieser Schritt muss nur einmal ausgeführt werden:

```
# Navigieren zum Wurzelverzeichnis des RVR-Interfaces:
$ cd /pfad/zum/interface

# Aufbauen des lokalen Docker Images
# Der Punkt am Ende bedeutet, dass Docker nach dem Dockerfile im aktuellen
# Verzeichnis suchen soll:
$ docker build -t rvr_interface:latest .
```

Statt *latest* kann auch das Datum zum Zeitpunkt der Erstellung (*231004* o. ä.) angegeben werden.

Auf Basis dieses Images kann ein Docker-Container gestartet werden. Die Verzeichnisse, hier die GRASS-Datenbank und die Input-Daten, werden dabei mit -v in den Container gemountet. Zusätzlich werden die flags gesetzt, damit Docker auch auf die Host-GPU zugreifen kann:

```
# Starten des Docker-Containers
$ xhost local:*
$ docker run -it --privileged --rm --ipc host \
         -v /pfad/zu/grassdata:/grassdb \
         -v /pfad/zu/Eingangsdaten:/mnt/data \
         -v "/tmp/.X11-unix:/tmp/.X11-unix:rw" \
         --runtime=nvidia --gpus all \
         --env DISPLAY=$DISPLAY \
         --device="/dev/dri/card0:/dev/dri/card0" \
         rvr_interface:latest bash

# Oder alternativ
$ xhost local:*
$ docker run -it --privileged --rm --ipc host \
         -v /pfad/zu/grassdata:/grassdb \
         -v /pfad/zu/Eingangsdaten:/mnt/data \
         -v "/tmp/.X11-unix:/tmp/.X11-unix:rw" \
         --runtime=nvidia --gpus all \
         --env DISPLAY=$DISPLAY \
         --device="/dev/dri/card0:/dev/dri/card0" \
         rvr_interface:231004 bash
```

Für **Windows** muss vorher sichergestellt sein, dass erst Docker Desktop und VcXsrv Windows X Server installiert und eingerichtet sind. *VcXsrv Windows X Server* kann mit den folgenden Schritten installiert und eingerichtet werden:

1. Download und installieren von VcXsrv Windows X Server
2. Starten von Xlaunch und dann konfigurieren (siehe auch hier):
   - Im "Extra Settings"-Fenster Haken setzen bei "Disable access control"
   - Im "Finish Configuration"-Fenster auf "Save configuration" klicken und die Konfiguration z.B. auf dem Schreibtisch/ Desktop speichern

Dann kann mit den folgenden Befehlen der Docker gestartet werden:

```
# eigene IP Adresse herausfinden (verwenden des Wertes bei „IPAddress“ z.B.
10.211.55.10 und nicht 127.0.0.1)
$ Get-NetIPAddress
# oder
$ ipconfig

# Setzen der DISPLAY-Variablen (dabei <YOUR-IP> setzen)
$ set-variable -name DISPLAY -value <YOUR-IP>:0.0

# Starten des Dockers mit den mounts der Daten
$ docker run -it --privileged --rm --ipc host -v C:/Users/path/to/
grassdata:/grassdb -v C:/Users/path/to/rvr_daten:/mnt/data --runtime=nvidia
--gpus all --env DISPLAY=$DISPLAY --device="/dev/dri/card0:/dev/dri/card0"
rvr_interface:latest bash
```

Es ist nicht notwendig, das Docker Image als root (sudo docker run …) zu starten. Für weitere Rechte können die Nutzenden der Gruppe „docker“ hinzugefügt werden:

```
# Nutzende der Gruppe „docker“ hinzufügen
$ sudo usermod -a -G docker <username>

# überprüfen mit
$ id
```

Solange GRASS GIS innerhalb des Docker-Containers genutzt wird, werden die Pfade im Docker anstelle der lokalen Pfade verwendet. Nach dem obigen Einmounten sind das die folgenden Stammpfade:

- Pfad zur grassdb: */grassdb/…*
- Pfad zu den Input-Daten: */mnt/data/…*

Diese müssen also beim Durchgehen der Schritte aus der Dokumentation ggf. angepasst werden.

Zunächst wird ein neues *Project* mit dem EPSG-Code 25832 (Projektion: ETRS89 / UTM zone 32N) angelegt. Alle darin importierten Datensätze verfügen dann über dieselbe Projektion:

```
# Allgemeines Schema zum Erstellen eines neuen GRASS GIS Projects
$ grass -c epsg:epsg-code /grassdb/neuer_project_name

# Erstellen eines neuen GRASS GIS Projects mit dem Namen „project_25832“
$ grass -c epsg:25832 /grassdb/project_25832
```

Im selben Terminal wird nun GRASS GIS gestartet, hier erfolgen die weiteren Schritte. Die GUI öffnet sich normalerweise automatisch. Die weiteren Schritte können entweder im Terminal oder in der GUI erfolgen. Auch in einer geöffneten GRASS-Sitzung können weiterhin alle Terminal-Befehle ausgeführt werden (z. B. Verzeichnisse wechseln, *bash*-Skripte starten, etc.). Falls die GRASS-GUI sich nicht automatisch öffnet oder geschlossen wurde, kann sie mit folgendem Befehl aufgerufen werden:

```
$ g.gui
```

Innerhalb der GRASS-*Projects* gibt es *Mapsets*, die die Sammlung von thematisch zusammenhängenden Datensätzen ermöglichen. Bei der Erstellung eines neuen *Projects* wird automatisch das PERMANENT-*Mapset* erstellt. Es ist jedoch sinnvoll, ein selbst benanntes **neues** *Mapset* zu erstellen:

```
# Neues Mapset namens “Gelsenkirchen_2020” erstellen und dort hinein wechseln
$ g.mapset -c Gelsenkirchen_2020

# TIPP: Für alle GRASS-Befehle können Sie <Befehl> --help ausführen, um
# mehr über die Benutzung des Befehls zu erfahren, z.B.:
$ g.mapset --help
```

Alternativ kann auch direkt beim Starten von GRASS GIS ein neues *Mapset* erstellt werden:

```
# Allgemeines Schema zum Erstellen eines neuen Mapsets in einer bestehenden
# Location
$ grass -c /grassdb/location_name/neuer_mapset_name

# Erstellen eines neuen Mapsets „Gelsenkirchen_2020“ in dem Project
„location_25832“
$ grass -c /grassdb/location_25832/Gelsenkirchen_2020
```

Nun befinden sich die Nutzenden in GRASS GIS innerhalb des (neuen) *Projects* (im Beispiel: “location_25832”) und des neu angelegten *Mapsets* (im Beispiel: „Gelsenkirchen_2020“). Von hier aus können die im weiteren Verlauf in GRASS GIS stattfindenden Schritte ausgeführt werden.

Der folgende Befehl öffnet GRASS GIS zu einem anderen Zeitpunkt in einem bestimmten *Project* und *Mapset* erneut:

```
# Allgemeines Schema zum Öffnen von GRASS in bestimmten Mapsets
$ grass /pfad/zu/grassdata/yourlocation/yourmapset

# Wiederaufnahme einer GRASS-Sitzung in dem Project “location_25832” und dem
# Mapset “Gelsenkirchen_2020”
$ grass /grassdb/location_25832/Gelsenkirchen_2020
```

Zum Umbenennen oder Löschen eines bestehenden *Mapsets* kann direkt der Ordner des *Mapsets* in der *grassdb* umbenannt bzw. gelöscht werden, z. B.:

```
# Allgemeines Schema zum Umbenennen/Löschen von GRASS Mapsets
$ mv /pfad/zu/grassdata/yourlocation/yourmapset
/pfad/zu/grassdata/yourlocation/new_name
$ rm -rf /pfad/zu/grassdata/yourlocation/yourmapset
```

Alle GRASS GIS entwickelten Addons, die für die Einzelbaumerkennung mittels Neuronalem Netz benötigt werden, haben den Key „neural network“ und können mit diesem in der GRASS GIS GUI gesucht werden.

1.3.3 Ändern der Eigentumsverhältnisse der erstellten GRASS-Daten

Der GRASS GIS Docker-Container läuft mit Root-Rechten. Daher werden die im Docker erstellten Daten als root erstellt, weshalb das Zugreifen auf diese Daten nicht wie gewohnt von allen Nutzenden möglich ist. Um z. B. eine im Docker erstellte GRASS GIS *Project/Mapset* von außerhalb des Dockers öffnen zu können, müssen vorher ggf. die Nutzenden und die Gruppe außerhalb des Dockers angepasst werden. Dies kann z. B. über die folgenden Befehle erfolgen:

```
# Abfrage der Rechte auf die gesamte GRASS-Datenbank
$ ls -lah /pfad/zu/grassdata
drwxr-xr-x 14 root root   4,0K Jan 31 13:43 project_erstellt_im_docker
drwxrwxr-x 5 USER GRUPPE 4,0K Jan 24 15:36 project_erstellt_außerhalb_docker

# Anpassen der Nutzenden und der Gruppe (beide können aus der Antwort zu
# dem obigen ‚ls‘ an dem anderen Project entnommen werden und
# dementsprechend gesetzt werden)
$ sudo chown USER:GRUPPE -R /pfad/zu/grassdata/project_erstellt_im_docker
```

1.4 Benötigte Datensätze

Für das Erstellen der Trainingsdaten werden die folgenden Datensätze benötigt:

- Gebiet:
  - Es wird ein Gebiet im Vektordatenformat (z. B. Shapefile, Geopackage) benötigt, für das die Prozessierung ausgeführt werden soll.
  - Alle anderen Datensätze müssen dieses Gebiet vollständig abdecken.
- TOPs:
  - Die benötigten True Orthophotos (TOPs) im .tif Format mit vier Farbkanälen (RGBI) müssen innerhalb eines gemeinsamen Verzeichnisses vorliegen (in diesem Verzeichnis dürfen keine weiteren Dateien enthalten sein).
- Punktwolken:
  - Die benötigten Punktwolken im .laz Format müssen in einem gemeinsamen Ordner vorliegen (in diesem Verzeichnis dürfen keine weiteren Dateien enthalten sein).
- ggf. Digitales Geländemodell (DGM; engl. digital terrain model (DTM)):
  - Falls nicht das NRW-DGM1 zur nDOM-Erstellung genutzt werden soll, kann auch ein individuell definiertes DGM mit einer idealen Auflösung von 0,5 - 1,0 m verwendet werden. Dieses muss in einem GDAL-konformen Format vorliegen (z. B. GeoTIFF, ASCII XYZ).

1.4.1 Import und Aufbereitung der Datensätze in GRASS GIS

Für den Import und die Aufbereitung der Eingangsdaten wurde das GRASS-Addon *m.import.rvr* entwickelt. Dieses fasst mehrere Arbeitsschritte zum Datenimport und zur Aufbereitung zusammen, wobei auch die Möglichkeit besteht, Gebäudedaten von OpenNRW bei Bedarf automatisch herunterzuladen. Das Import-Tool kann ebenso für das Gebäude-Tool (s. Dokumentation „Gebäude- und Dachbegrünungsdetektion“) genutzt werden. Nachfolgend findet sich eine Übersicht der Aufgaben des Import-Addons:

- Importieren der benötigten Datensätze

    - Gebiet, TOPs, 2,5D-Punktwolken, DGM

- Aufbereitung der TOPs
    - Importieren der einzelnen TOP-.tif-Dateien, die in dem angegebenen Gebiet liegen, und Resampling dieser auf 0,2 m Auflösung
    - Zusammenfassen der einzelnen TOP-Tiles zu einem virtuellen Raster (VRT). Jeder Kanal liegt als eigene Rasterkarte (*top_red_02*, *top_green_02*, *top_blue_02* und *top_nir_02*) im *Mapset*. Diese VRT-Karten setzen sich wiederum aus den einzelnen TOP-Tiles zusammen, z. B *top_red_02* aus verschiedenen anderen *top_*_02*.

- Aufbereitung der 2,5D-Punktwolken
    - Importieren der 2,5D-Punktwolken in GRASS GIS und Verarbeitung zu einem Digitalen Oberflächenmodell (DOM; engl. *digital surface model* (DSM)) im Rasterformat mit 0,5 m Auflösung.
    - Importieren der .laz-Tiles als DOM-Raster und anschließend Zusammensetzung zu einem Gesamtraster. Dies geschieht mit Hilfe des Import-Addons, das intern das Addon *r.in.pdal.worker* aufruft, um die einzelnen .laz-Dateien auch parallel importieren zu können.

- Berechnung des normalisierten Digitalen Oberflächenmodells (nDOM; engl. *normalised digital surface model* (nDSM))
    - Zur Berechnung des nDOM (der Differenz aus DOM und DGM) wird das Addon *r.import.ndsm_nrw* im Import-Addon aufgerufen. Dieses interpoliert eventuelle NoData-Lücken im importierten DOM, die durch zu geringe Dichte der Punktwolken auftreten können, berechnet aus vorhandenem DOM und einem DGM das nDOM und resampled es auf die aktuelle *Region*.
    - Zur Berechnung des nDOM wird entweder ein individuell definiertes DGM oder das NRW-DGM verwendet. Zur Nutzung eines eigenen DGMs kann der Parameter *dsm_dir* im Import-Addon angegeben werden. Wird kein DGM definiert, bezieht das Addon automatisch die benötigten Tiles des NRW-DGM.
    - Das DGM für das Verbandsgebiet wird sowohl vom RVR als auch vom Land NRW gekachelt im XYZ-Format zur Verfügung gestellt. Dabei ist zu beachten, dass sich die XY-Koordinaten hier nicht wie allgemein üblich auf die Pixelmitte beziehen, sondern auf die linke untere Ecke eines Pixels. Wenn dies nicht beachtet wird, kann es zu einem Versatz um einen halben Pixel kommen, außerdem passen die Grenzen dann nicht mehr zu den Grenzen der TOP-Kacheln.
    - Bei der Nutzung eines individuell definierten DGMs wird dessen Versatz wie folgt gehandhabt:
        - Bei der Nutzung von XYZ-Tiles wird der Versatz vom Import-Addon korrigiert (Auflösung des DGMs muss angegeben werden).

- Bei der Nutzung eines GeoTiffs wird von einer vorher erfolgten Korrektur ausgegangen und das GeoTiff „*as is*“ eingeladen.

Hinweis:
Im Rahmen des Teilprojekts „Trainingsdatenerstellung“ wurde *m.import.rvr* erweitert, sodass es mit der Option ***type=“neural network“*** nur die für die NN-Analyse benötigten Daten importiert. Sofern die benötigten Daten bereits zuvor im Rahmen der regel- bzw. ML-basierten Baumerkennung mit ***type=“trees analysis“*** importiert wurden und in GRASS vorliegen, kann *m.import.rvr* übersprungen werden.

Im Folgenden werden die Parameter des Import-Addons genauer aufgeschlüsselt, die für die Trainingsdatenerstellung zur Einzelbaumerkennung relevant sind:

Eingangsparameter:

- ***type***: Analysetyp, für den die benötigten Daten importiert werden sollen: „neural network“.
- ***area***: Pfad zur Vektordatei, die das zu berechnende Gebiet enthält.
- ***dsm_dir***: Pfad zum Ordner, in dem die DOM-LAZ liegen.
- ***dsm_tindex*:** Pfad zu einem / für einen Tileindex für die DOM LAZ-Dateien, die im *dsm_dir* liegen (optional). Der Tileindex muss ein Attribut *location* haben, in dem der absolute Pfad zu den .laz-Daten (innerhalb des Dockers) steht. Dieser wird entweder verwendet, sofern die Datei existiert und er nicht bei jedem Lauf des Addons erstellt werden muss, oder gespeichert, wenn die Datei nicht existiert, damit er für zukünftige Läufe wiederverwendet werden kann. Das erste Erstellen des Indexes *dsm_tindex* kann lange dauern, weshalb es sinnvoll ist, vor allem diesen Index wiederzuverwenden. Ändert sich hingegen die Auflösung der Eingangsdaten (z. B. zwischen zwei Bildflugjahren von 10 cm auf 7,5 cm), muss dieser neu berechnet werden (optional).
- ***dtm_dir***: Pfad zu einem Ordner mit mehreren XYZ-Dateien des DGMs (Import und Shift, um die Angabe der unteren linken Ecke statt des Pixelzentrums zu korrigieren, hierfür wird die dtm_resolution benötigt) (optional; *Leerlassen, um automatisch das DGM von Open.NRW runterzuladen*)
- ***dtm_file***: Pfad zur DGM-Datei im Format GeoTiff (Import und Resampling ohne Korrektur) oder XYZ (Import und Shift, um die Angabe der unteren linken Ecke statt des Pixelzentrums zu korrigieren, hierfür wird die *dtm_resolution* benötigt) (optional; *Leerlassen, um automatisch das DGM von Open.NRW runterzuladen*).
- ***dtm_resolution***: Original-Auflösung des DGM, damit bei Verwendung einer DGM-XYZ-Datei oder mehreren DGM-XYZ-Dateien in einem Ordner eine Korrektur des Versatzes durchgeführt werden kann (optional; notwendig, wenn *dtm_dir* oder *dtm_file* als XYZ-Datei gegeben)

- ***dtm_tindex***: Pfad zum / für einen Tileindex für die DTM-XYZ-Dateien, die im *dtm_file* Ordner liegen. Der Tileindex muss ein Attribut *location* haben, in dem der absolute Pfad zu den XYZ-Daten (innerhalb des Dockers) steht. Dieser wird entweder verwendet, sofern die Datei existiert, damit er nicht bei jedem Lauf des Addons erstellt werden muss, oder gespeichert, wenn die Datei nicht existiert, damit er für zukünftige Läufe wiederverwendet werden kann. Ändert sich hingegen die Auflösung der Eingangsdaten (z. B. zwischen zwei Bildflugjahren von 10 cm auf 7,5 cm), muss dieser neu berechnet werden (optional).
- ***top_dir***: Pfad zum Ordner, in dem die TOP-GeoTiffs liegen.
- ***top_tindex***: Pfad zum / für einen Tileindex für die TOPs, die im *top_dir* liegen. Der Tileindex muss ein Attribut *location* haben, in dem der absolute Pfad zu den TOPs (innerhalb des Dockers) steht. Dieser wird entweder verwendet, sofern die Datei existiert und er nicht bei jedem Lauf des Addons erstellt werden muss, oder gespeichert, wenn die Datei nicht existiert, damit er für zukünftige Läufe wiederverwendet werden kann. Ändert sich hingegen die Auflösung der Eingangsdaten (z. B. zwischen zwei Bildflugjahren von 10 cm auf 7,5 cm), muss dieser neu berechnet werden (optional).
- ***-c***: Flag zum Prüfen, ob alle für den Analysetyp erforderlichen Input-Daten importiert werden können, ohne dabei den eigentlichen Import zu starten.

Ein Teil der Berechnung im Import-Modul kann parallel erfolgen. Für die Konfiguration der Parallelisierung können folgende Parameter gesetzt werden:

- ***memory***: Arbeitsspeicher in MB, der dem Addon zur Verfügung stehen soll (optional, Defaultwert ist 300 MB)
- ***nprocs:*** Anzahl der Kerne für die Parallelprozessierung (optional, Defaultwert ist -2 (Anzahl der verfügbaren Kerne minus 1), für serielle Prozessierung auf 1 setzen)

```
# m.import.rvr ausführen:
$ DATAFOLDER=/mnt/data
$ m.import.rvr memory=300 type="neural network" \
   area=${DATAFOLDER}/Gebiet/gelsenkirchen.gpkg\
   top_dir=${DATAFOLDER}/2020_Sommer/TOP/ \
   dsm_dir=${DATAFOLDER}/2020_Sommer/Punktwolke_2_5D_RGBI/ \
   dtm_file=${DATAFOLDER}/2020_Sommer/DGM/dgm1_gelsenkirchen_2020_correcte-
d.tif
```

Hinweis:
Dieser Schritt kann aufgrund der Größe der Eingangsdateien und gemounteten Datenverzeichnisse viele Stunden in Anspruch nehmen.
Zur Vermeidung von Fehlern empfehlen wir für den Arbeitsspeicher den Defaultwert von 300 MB. Für *memory* erfolgt dann keine Angabe. Darüber hinaus ist es sinnvoll, die maximale Anzahl der verfügbaren Kerne minus 1 anzugeben (Default für *nprocs*), um die Rechenzeit gering zu halten.

Das Import-Addon importiert dann die folgenden Daten für die Einzelbaumerkennung mittels eines Neuronalen Netzes:

- ***study_area:*** eine Vektorkarte des Gebietes
- ***top_red_02, top_green_02, top_blue_02, top_nir_02:*** die Rasterkarten der einzelnen Kanäle des TOP-Mosaiks in einer Auflösung von 0.2m
- ***dsm_02:*** die Rasterkarte des importierten DOMs in einer Auflösung von 0.2m
- ***ndsm:*** die Rasterkarte des berechneten nDOMs in einer Auflösung von 0.2m

2 Training und Anwendung eines Neuronalen Netzes

Das Multi-Addon *m.neural_network* beinhaltet die Addons *m.neural_network.preparedata_part1*, mit wiederum den Workern *m.neural_network.preparedata_part1.worker_nullcells* und *m.neural_network.preparedata_part1.worker_export*, sowie *m.neural_network.preparedata_part2* mit dem Worker *m.neural_network.preparedata_part2.worker_label.* Diese sind für die Aufbereitung der Daten (Kapitel 2.1.1 - 2.1.3). Für das Modelltraining, dessen Evaluierung und das Anwenden eines Modell gibt es die Addons *m.neural_network.train, m.neural_network.test* und *m.neural_network.apply* (Kapitel 2.2 - 2.4). Für das Zusammensetzen der einzelnen Kacheln nach der Anwendung zu einem Gesamtgebiet wird *m.neural_network.postprocess_patch* verwendet. Siehe dazu Kapitel 2.5. Zur anschließenden Postprozessierung findet sich mehr in Kapitel 3.

Die Installation des Multi-Addons installiert automatisch die oben genannten Module sowie ihre zugehörigen Worker-Module (im Dockerfile bereits enthalten). Die Worker-Module werden indirekt durch die Hauptmodule aufgerufen und dienen der parallelen Prozessierung über ein Gitter. Durch das sogenannte Tiling erfolgt ein Teil der Berechnung über mehrere Kacheln.

2.1 Trainingsdatenerstellung

Der erste Schritt für Training und Anwendung eines Neuronalen Netzes zur Einzelbaumerkennung besteht in der Erstellung von Trainingsdaten. In diesem Abschnitt werden die dazu benötigten Einzelschritte beschrieben.

2.1.1 Datenvorbereitung für die Erstellung der Trainingsdaten

Mit dem Multi-Addon *m.neural_network* wird der Prozess des Erstellens von Trainingsdaten über mehrere Schritte ermöglicht. Die importierten TOPs sowie das nDOM werden zur Weiterverarbeitung gekachelt und aus GRASS GIS exportiert. Weiterhin werden entweder schon vorher vorhandene Baumkarten (aus dem Einzelbaum-Tool der mundialis) oder eine automatisch segmentierte Kachel exportiert. Diese Baumkarte oder segmentierte Kachel ist die Grundlage für den Labellayer der Trainingsdaten. Die Bäume sind in Abhängigkeit der TOPs und des nDOM als solche zu labeln und zu ergänzen. Dies passiert in QGIS.

Das Addon *m.neural_network.preparedata_part1* erstellt eine regelmäßige Kachelung der Eingangsdaten und exportiert sie anschließend kachelweise. Die Kachelung ist notwendig für das

Training und die Anwendung des Neuronalen Netzes, da dieses Bilddaten und Labels jeweils in kleinen Ausschnitten (z.B. 512 x 512 Pixel) verarbeitet. Da am Ende beim Zusammensetzen der Ergebnisse die Ränder der Kacheln abgeschnitten werden, um Randeffekte zu eliminieren, kann es sinnvoll sein, das Anwendungsgebiet größer als das angezielte Untersuchungsgebiet zu wählen. Konkret erzeugt das Addon die Datensätze

- kachelweise TOPs mit vier Kanälen (RGBI) als TIF (*image_tile_xx_xx.tif*)
- kachelweise nDOM als TIF (einmal mit der Höhe in Metern (*ndsm_tile_xx_xx.tif*) und einmal auf 0 bis 30 m abgeschnitten und dann auf 1 bis 255 skaliert (*ndsm_1_255_tile_xx_xx.tif*), damit das Neuronale Netz die Daten als Input verwenden kann)
- kachelweise Label-Vektordatensatz (*label_tile_xx_xx.gpkg*). Dieser muss im Rahmen der Trainingsdatenerstellung anschließend händisch angepasst werden. Das Addon erleichtert diesen Schritt jedoch dadurch, dass der Datensatz bereits vorgeschlagene Vektorobjekte enthält, die entweder aus einem existierenden Baum-Datensatz stammen, oder die automatisch durch eine Segmentierung auf Basis der Bilddaten erzeugt werden.
- Tile-Index (*tindex.gpkg* mit QGIS-Styledatei *tindex.qml*)

als vorbereitende Schritte für das manuelle Labeln der Trainingsdaten und speichert sie zur weiteren Verarbeitung. Ein Anteil der Kacheln wird innerhalb des Ergebnisordners in den Ordner **train** geschrieben. Sie sind für das Training des Neuronalen Netzes vorgesehen und müssen gelabelt werden. Die restlichen Kacheln werden zur Verwendung im späteren Prozessschritt des Anwendens des Neuronalen Netzes im Ordner **apply** aufbewahrt. Die Aufteilung in **train**/**apply** Kacheln erfolgt zufällig, der Anteil der **train** Kacheln kann jedoch vom Nutzer vorgegeben werden (s.u.). Sollen alle Daten für das Training genutzt werden, kann dies über die *-t*-Flag gesetzt werden. Es wird dann kein ***apply*** Ordner erstellt. Wenn im späteren Prozessschritt alle Daten für die Anwendung genutzt werden sollen, kann dies über die *-a*-Flag gesetzt werden. In diesem Fall werden keine Kacheln zum Labeln in einen ***train*** Ordner abgelegt. Neben den **train** und **apply** Ordnern liegt ein Tileindex (*tindex.gpkg*) mit Informationen über die vorhandenen Kacheln und ihre Ordnerzuordnung (**train**/**apply**). Die Ordnerstruktur nach dem Durchlauf von *m.neural_network.preparedata_part1* sieht wie folgt aus:

```
Ergebnisordner
├── apply
│   ├── tile_00_00
│   │   ├── image_tile_00_00.tif
│   │   ├── ndsm_1_255_tile_00_00.tif
│   │   └── ndsm_tile_00_00.tif
│   ├── tile_00_02
│   │   ├──...
├── tindex.gpkg
├── tindex.qml
└── train
    ├── tile_00_01
    │   ├── image_tile_00_01.tif
    │   ├── label_tile_00_01.gpkg
    │   ├── label_tile_00_01.qml
    │   ├── ndsm_1_255_tile_00_01.tif
    │   └── ndsm_tile_00_01.tif
    ├── tile_00_05
    │   ├── ...
```

Im Folgenden werden die Parameter des Addons *m.neural_network.preparedata_part1* aufgeführt.

Eingangsparameter:

- ***image_bands***: Die Namen der Bildraster, z.B. die einzelnen RGBI Raster der TOPs. Diese müssen in der gewünschten Auflösung (meist 0.2m) vorliegen, ebenso das ndsm. Dies ist jedoch normalerweise bereits durch *m.import.rvr* sichergestellt.
- ***ndsm***: Name des nDOM Rasters. Default:ndsm
- ***reference*** *(optional)*: Name der Referenzvektorkarte. Sofern bereits ein Einzelbaum-Vektordatensatz vorliegt (z.B. aus dem regelbasierten Ansatz) kann dieser hier übergeben werden. In diesem Fall erhält der Label-Vektordatensatz *label_tile_xx_xx.gpkg* die entsprechenden Objekte. Wenn die ***reference*** Option nicht genutzt wird, erhält der Label-Vektordatensatz *label_tile_xx_xx.gpkg* Objekte aus einer Segmentierung auf Basis der Bilddaten. Diese Segmentierung kann über die Parameter **segmentation_minsize** und **segmentation_threshold** gesteuert werden.
- ***tile_size***: Die Größe der zu kreierenden Kacheln, auf denen gelabelt wird, in der Einheit Zellen und der resultierenden Form *tile_size x tile_size*. Default: 512

- ***tile_overlap***: Die Überschneidung der Kacheln in Einheit Zellen. Default: 128
- ***segmentation_minsize***: Die minimale Anzahl an Zellen in einem Segment. Default: 80
- ***segmentation_threshold***: Unterschiedlichkeitsgrenzwert zwischen 0 und 1 für die Segmente. Ein Grenzwert von 0 bedeutet, dass nur identische Segmente verschmolzen werden, während 1 eine Verschmelzung aller Segmente zur Folge hat. Default: 0.3
- ***train_percentage***: Der Anteil an allen Kacheln, die zum Trainieren verwendet werden sollen und im Ordner **train** abgelegt werden. Default: 30
- ***output_dir***: Der Ergebnisordner, unter dem die Daten abgelegt werden. Da im folgenden Schritt in kollaborativer Arbeit gelabelt wird, sollte die ***output_dir*** auch außerhalb des Docker Containers erreichbar sein, also in einem eingemounteten Verzeichnis liegen. Hier muss außerdem genügend Speicherplatz vorliegen, da die gesamten Bilddaten kachelweise als .tif-Dateien exportiert werden.
- ***nprocs***: Anzahl der Kerne für die Parallelprozessierung. Default: 1 führt zu serieller Prozessierung.
- ***suffix*** *(optional)*: Ermöglicht die Vergabe eines einheitlichen Suffixes an alle Kacheln und exportierte Datensätze. Dies kann z.B. das Befliegungsjahr sein, was es ermöglicht, die Ergebnisse nach mehrmaligem Aufruf von *m.neural_network.preparedata_part1* mit unterschiedlicher Datengrundlage zusammenzuführen, ohne dass es zu Redundanzen in den Dateinamen kommt (siehe auch den Hinweis am Ende von Abschnitt 2.1.2).
- ***-t:*** Flag wenn alle Daten fürs Training genutzt und aufbereitet werden sollen
- ***-a:*** Flag wenn alle Daten für die Anwendung eines Modells genutzt und aufbereitet werden sollen

```
# Beispiel für die Anwendung von m.neural_network.preparedata_part1 um TOPs
# und nDOM als .Tifs und Labellayer als Geopackage kachelweise zu exportieren

$ m.neural_network.preparedata_part1 \
    image_bands=top_red_02,top_green_02,top_blue_02,top_nir_02 \
    ndsm=ndsm tile_size=512 output_dir=/mnt/data/labeled_data_2024 \
    nprocs=6 suffix=2024
```

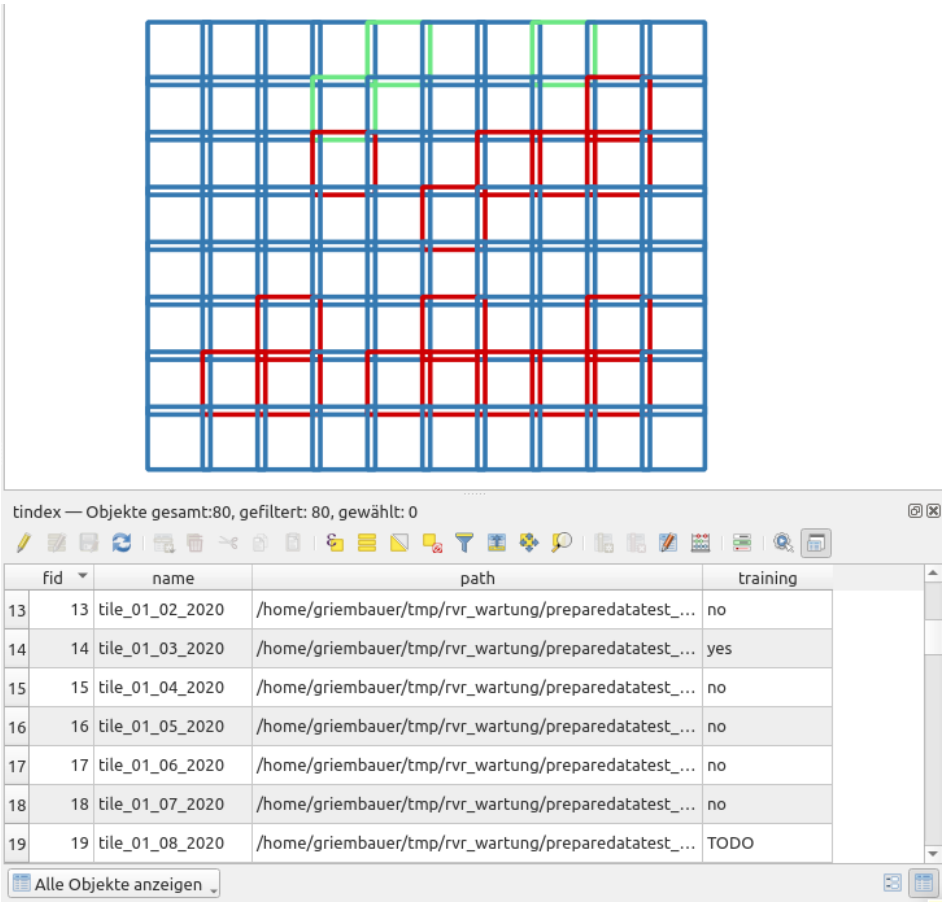
```
# Beispiel für die Anwendung von m.neural_network.preparedata_part1 um TOPs
# und nDOM als .Tifs kachelweise zu exportieren
# Aufbereitung für nur Anwendung eines Modells (kein Training)

$ m.neural_network.preparedata_part1 \
   image_bands=top_red_02,top_green_02,top_blue_02,top_nir_02 \
   ndsm=ndsm tile_size=512 output_dir=/mnt/data/labeled_data_2024 \
   nprocs=6 suffix=2024 -a
```

2.1.2 Trainingsdaten erstellen in QGIS

Im Anschluss an den Durchlauf von *m.neural_network.preparedata_part1* sind die Daten so strukturiert, dass das kachelweise Zuweisen von Klassen („Labeln“) erfolgen kann und somit die eigentlichen Trainingsdaten erstellt werden können. Hierfür kann QGIS verwendet werden. QGIS ist nicht im Docker-Image enthalten, sodass dieser Schritt auf dem jeweiligen Host-System passieren sollte. Hierzu ist ggf. eine Änderung der Eigentumsverhältnisse der von *m.neural_network.preparedata_part1* erstellen Dateien nötig, siehe auch Abschnitt 1.3.3.

Da es sich beim Labeln um einen arbeits- und zeitintensiven Schritt handelt, der am Besten kollaborativ ausgeführt wird, kann der Tile-Index (*tindex.gpkg*) genutzt werden, um den Bearbeitungsstand festzuhalten (siehe Abbildung 1). In der Spalte *training* ist angegeben, ob es sich bei der jeweiligen Kachel um eine Trainingskachel handelt, d.h. ob sie zu labeln ist (*„TODO“, rot*) oder nicht (*„no“, blau*). Ist die Kachel fertig gelabelt, kann der Wert auf „*yes*“ geändert werden, sodass die Kachel grün dargestellt wird. Die Tile-Index Datei wird von den folgenden Addons nicht mehr benötigt, sondern dient nur dem Festhalten des Bearbeitungsstands, sodass sie beliebig verändert oder erweitert (z.B. Name der bearbeitenden Person) werden kann.



	fid	name	path	training
13	13	tile_01_02_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	no
14	14	tile_01_03_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	yes
15	15	tile_01_04_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	no
16	16	tile_01_05_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	no
17	17	tile_01_06_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	no
18	18	tile_01_07_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	no
19	19	tile_01_08_2020	/home/griembauer/tmp/rvr_wartung/preparedatatest_...	TODO

Abbildung 1: Beispiel eines Tile-Index in QGIS

Für das Labeln der einzelnen Kacheln sind die folgenden Schritte auszuführen:

1. Einladen von TOP (*image_tile_xx_xx.tif*), nDOM (*ndsm_tile_xx_xx.tif*) und Labellayer (*label_tile_xx_xx.gpkg*) in QGIS. Eine Beispielansicht in ist Abbildung 2 zu sehen. Der Labellayer wurde in diesem Fall von *m.neural_network.preparedata_part1* durch eine Segmentierung der Bilddaten erzeugt. Alle Features dieses Layers haben jedoch noch den Wert 0 in der Spalte *class_number* („TODO“) und sind also anzupassen.

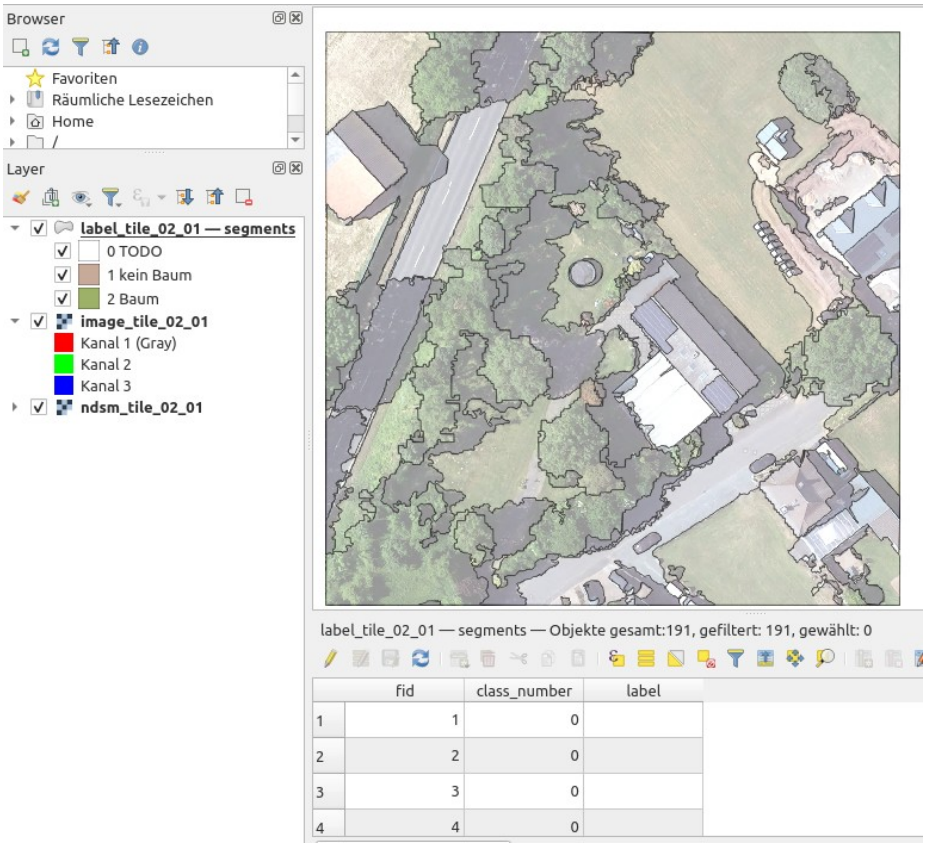


Abbildung 2: Beispielhafte QGIS-Ansicht für eine Kachel mit TOP (image_tile_02_01) und Label-Vektordatei (label_tile_02_01) vor dem manuellen Labeln.

2. Die Vektorobjekte in der Label-Vektordatei (*label_tile_xx_xx.gpkg)* müssen anschließend manuell anhand der Bilddaten (und ggf. nDOM zur Überprüfung der Höhe) angepasst werden. Hierzu muss der Bearbeitungsmodus aktiviert werden. Sofern die bereits vorhandenen Objektgrenzen gut mit den sichtbaren Bäumen übereinstimmen, muss lediglich der Wert in der Spalte c*lass_number* auf ***2*** (Baum) geändert werden. Andernfalls müssen die Polygone mit den QGIS-Vektorisierungstools angepasst, aufgeteilt, bzw. hinzugefügt werden. Weiterhin sind die folgenden Regeln zu beachten:

- Es müssen keine Einzelbäume als Einzelobjekte aufgeteilt werden, d.h. zusammenhängende Baumgruppen können als ein gemeinsames Objekt den Wert ***2*** erhalten. Die Aufteilung in Einzelbäume erfolgt erst im Postprocessing.
- Objekte/Flächen, die keine Bäume sind, können entweder den Wert ***1*** erhalten, oder auch komplett gelöscht werden. Der fertige Vektorlayer darf also auch Lücken innerhalb der Kachel enthalten, die dann als nicht-Baum (***1***) interpretiert werden.
- Wenn die gesamte Kachel keinen Baum erhält, müssen entweder alle Objekte den Wert ***1*** erhalten oder alle Vektor-Features gelöscht werden (die Vektordatei selbst muss in diesem Fall jedoch beibehalten werden, sie enthält dann nur keine Features mehr).
- Die Spalte *label* dient nur der Protokollierung und kann mit beliebigem Text gefüllt werden.
- Wichtig ist, dass alle sichtbaren Bäume in der Kachel den Wert **2** erhalten, da alle anderen Flächen innerhalb der Kachel automatisch zum Training der Klasse „nicht-Baum“ (***1***) verwendet werden.
- Insgesamt dürfen im fertig gelabelten Vektordatensatz nur die Werte ***1*** und ***2*** in der Spalte *class_number* vorkommen.
- Die Veränderungen müssen gespeichert werden. Unter Umständen kann ein Entfernen des gespeicherten Layers aus der Layerliste in QGIS notwendig sein, z.B. wenn man die Daten danach auf einen Server hochladen möchte.
- Der fertige Label-Vektordatensatz muss im ursprünglichen Kachel-Ordner mit der ursprünglichen Benennung liegen bleiben.

Hinweis: Eine hohe Anzahl an unterschiedlichen Trainingsdaten wirkt sich positiv auf die Performance des Modells aus. Um verschiedene Bildcharakteristika in den Trainingsdaten abzubilden, ist es möglich, mehrere Bildflüge bzw. mehrere Jahre miteinzubeziehen. Wichtig ist, dass alle zusammengeführten Daten in der gleichen räumlichen Auflösung vorliegen, was jedoch bei der Verwendung von *m.import.rvr* gewährleistet ist. Das Addon *m.neural_network.preparedata_part1* ist dazu einzeln, jeweils mit einem eindeutigen ***Suffix***, auf jede Befliegung anzuwenden und es wird unabhängig voneinander gelabelt. Danach müssen die Inhalte der „train“ und „apply“ Ordner manuell zusammengeführt werden, um den folgenden Schritt *m.neural_network.preparedata_part2* auf alle Daten gemeinsam anzuwenden.

2.1.3 Aufbereitung der Labels und Bilddaten für das Training

Dieser Schritt ist erst auszuführen, wenn das Labeln komplett abgeschlossen ist. Das Neuronale Netz benötigt die Label-Dateien in Rasterform, sowie alle Bilddaten in einer bestimmten Struktur. Dies wird vom Addon *m.neural_network.preparedata_part2* durchgeführt. Dieses erstellt einen **komplett neuen** Ordner, in dem die rasterisierten Label-Dateien im .tif-Format, sowie die benötig-

ten Bilddaten abgelegt werden. Um Speicherplatz zu sparen, werden die Bilddaten nicht erneut als .tif, sondern als .vrt Dateien abgelegt, wobei diese die im vorigen Schritt durch *m.neural_network.preparedata_part1* erstellten Bilddaten im .tif-Format referenzieren. Daher ist es wichtig, dass die Ergebnisse von *m.neural_network.preparedata_part1* im Anschluss an den Durchlauf von *m.neural_network.preparedata_part2* nicht verschoben oder gelöscht werden, da sonst die von *m.neural_network.preparedata_part2* erstellten .vrt Dateien ungültig werden. Da in den .vrt-Dateien jedoch relative Pfade abgelegt sind, ist es möglich, die Ergebnisse von *m.neural_network.preparedata_part1* und *m.neural_network.preparedata_part2* gemeinsam zu verschieben. Zudem behalten die .vrts aufgrund der relativen Pfade sowohl innerhalb als auch außerhalb des Docker-Containers ihre Gültigkeit.

m.neural_network.preparedata_part2 überprüft zunächst, ob die Label-Vektordatensätze nur gültige Werte (***1*** oder ***2***) enthalten und bricht andernfalls mit einem Fehler und einen Hinweis auf die entsprechende Datei ab. Sofern eine Kachel laut Label-Vektordatensatz keine Bäume enthält, was durchaus korrekt sein kann, wird eine Warnung ausgegeben, und unter der Annahme, dass die gesamte Kachel in die Klasse „nicht-Baum“ (***1***) fällt, weitergerechnet. Weiterhin werden alle Trainingskacheln nochmals in Trainings-, Validierungs- und ggf. Testkacheln aufgeteilt. Die Validierungsdaten werden später während des Trainings genutzt, um die Genauigkeit des jeweiligen Trainingsdurchlaufs zu bewerten. Sie sind daher kein unabhängiger Testdatensatz, der für die Bewertung der Gesamtklassifikation genutzt werden kann. Die Aufteilung in Trainings-, Validierungs- und Testdaten erfolgt zufällig, jedoch kann der Anteil der Validierungskacheln über den Parameter ***val_percentage*** gesteuert werden (s.u.). Ein unabhängiger Testdatensatz kann erzeugt werden, indem der Parameter ***test_percentage*** auf einen Wert größer 0 gesetzt wird. Nach einem Trainingsdurchlauf mit den Trainings- und Validierungsdaten kann dieser Testdatensatz zur Evaluierung des Modells verwendet werden.

m.neural_network.preparedata_part2 ist aus dem Docker Container heraus auszuführen, sodass dieser ggf. wieder hochgefahren (siehe Abschnitt 1.3.2) und eine GRASS-Sitzung in einem beliebigen Mapset mit der Zielprojektion (EPSG:25832) gestartet werden muss.

Im Ergebnisordner von *m.neural_network.preparedata_part2* sind wiederum ein ***train*** und/oder ein ***apply*** Ordner enthalten. Diese enthalten jeweils die folgenden Unterordner, die auf die Schnittstelle zum Neuronalen Netz optimiert sind:

- *singleband_vrts*: enthält kachelweise .vrts der TOP-Einzelbänder und referenziert die von *m.neural_network.preparedata_part1* erstellten TOP-tifs. Diese Einzelbänder werden als Hilfsdatensatz benötigt, um Multiband-vrts zu erzeugen (s.u.).

Nur im ***train*** Ordner:

- *train_images*: enthält kachelweise Multiband-vrts für alle Trainingskacheln. Die .vrts referenzieren dabei die *singleband_vrts* für die TOP-Kanäle und das umskalierte nDOM-tif, das von *m.neural_network.preparedata_part1* erstellt wurde, als fünften Kanal.

- *val_images*: enthält kachelweise Multiband-vrts für alle Validierungskacheln. Die .vrts referenzieren dabei die *singleband_vrts* für die TOP-Kanäle und das umskalierte nDOM-tif, das von *m.neural_network.preparedata_part1* erstellt wurde, als fünften Kanal.
- *test_images*: enthält kachelweise Multiband-vrts für alle Testkacheln. Die .vrts referenzieren dabei die *singleband_vrts* für die TOP-Kanäle und das umskalierte nDOM-tif, das von *m.neural_network.preparedata_part1* erstellt wurde, als fünften Kanal. Dieser Testdatensatz sollte möglichst unabhängig von den Trainings- und Validierungsdaten sein, um die Übertragbarkeit des Modells beurteilen zu können. Falls insgesamt nur wenig Trainingsdaten zur Verfügung stehen, sollten keine Kacheln als Testkacheln reserviert werden, sondern möglichst viele Kacheln für das eigentliche Training verwendet werden.
- *train_masks*: enthält die rasterisierten Label-Vektordatensätze der Trainingskacheln im .tif Format.
- *val_masks*: enthält die rasterisierten Label-Vektordatensätze der Validierungskacheln im .tif Format.
- *test_masks*: enthält die rasterisierten Label-Vektordatensätze der Testkacheln im .tif Format.

Nur im ***apply*** Ordner:

- *apply_images*: Hier sind die Kacheln enthalten, auf die später das trainierte Neuronale Netz angewendet werden soll.

Die Schnittstelle zum Neuronalen Netz erfordert diese Ordnerstruktur und -inhalte.

Die Eingangsparameter sind nachfolgend beschrieben.

Eingangsparameter:

- ***input_traindir***: Pfad zum von *m.neural_network.preparedata_part1* erstellten ***train*** Ordner. In den darin enthaltenen kachelweisen Ordnern müssen u.a. auch die fertig gelabelten Vektordatensätze liegen. Parameter muss nicht gesetzt werden, wenn alle Daten für die Anwendung eines Modells aufbereitet wurden.
- ***input_applydir***: Pfad zum von *m.neural_network.preparedata_part1* erstellten ***apply*** Ordner. Parameter muss nicht gesetzt werden, wenn alle Daten für das Training eines Modells aufbereitet wurden.
- ***val_percentage***: Anteil an Trainingskacheln, die zur Validierung während des Trainings verwendet werden. Default: 20
- ***test_percentage***: Anteil an Trainingskacheln, die zur Evaluierung des trainierten Modells verwendet werden. Default: 0
- **class_column**: Spaltenname im Label-Vektordatensatz, der die Klassennummer enthält. Default: *class_number*

- **class_values**: Erlaubte Werte für die Klassennummer/n. Default: 2. Theoretisch können hier mehrere Klassennummern angegeben werden, damit das Addon auch für andere Aufgaben als eine binäre Baum/Nicht-Baum Klassifikation geeignet ist. Für den konkreten Anwendungsfall genügt jedoch die einzelne Klassennummer (2).
- **no_class_value**: Wert der „Rest"-Klasse, in diesem Fall „nicht-Baum". Default: 1
- **output**: Ergebnisordner, in dem o.g. Ordnerstruktur erstellt wird. Dieser Ordner darf noch nicht existieren. Sofern während der Laufzeit ein Fehler auftritt, wird der unter ***output*** angegebene Ordner entfernt, sodass neu gerechnet werden kann.
- ***nprocs***: Anzahl der Kerne für die Parallelprozessierung. Default: 1 führt zu serieller Prozessierung.

```
# Beispiel für die Anwendung von m.neural_network.preparedata_part2
# wenn alle Daten für die Anwendung eines Modells aufbereitet wurden

$ m.neural_network.preparedata_part2 \
   input_applydir=/mnt/data/labeled_data_2024/apply nprocs=6 \
   output=/mnt/data/labeled_data_2024_restructured
```

```
# Beispiel für die Anwendung von m.neural_network.preparedata_part2
# wenn die Daten sowohl für Training als auch Anwendung eines Modells
# aufbereitet wurden

$ m.neural_network.preparedata_part2 \
   input_traindir=/mnt/data/labeled_data_2024/train \
   input_applydir=/mnt/data/labeled_data_2024/apply \
   val_percentage=20 test_percentage=20 \
   class_column=class_number class_values=2 no_class_value=1 \
   nprocs=6 \
   output=/mnt/data/labeled_data_2024_restructured
```

2.2 Trainieren eines KI-Modells

Ein KI-Modell kann mit *m.neural_network.train* definiert und trainiert werden. Um ein Modell neu zu definieren und initial zu trainieren, werden die Angaben zu ***model_arch***, ***encoder_name***, und ***encoder_weights*** benutzt, wobei Defaultwerte gesetzt sind. Details zu weiteren möglichen Architekturen und Encodern finden sich in der Dokumentation von smp: https://smp.readthedocs.io/.

Das Addon kann auch verwendet werden, um ein bereits trainiertes und lokal gespeichertes Modell weiter zu trainieren (fine-tuning). Dazu muss der Ordner des bestehenden Modells inklusive der entsprechenden *model.safetensors* und *config.json* Dateien mit der Option ***input_model_path*** angegeben werden. In diesem Fall werden die Gewichte nicht über ***encoder_weights*** definiert, sondern die Werte aus der *model.safetensors* genutzt.

Während des Trainings wird nach jedem Durchlauf (jeder Epoche) unter anderem der Validierungs-Loss berechnet. Der Validierungs-Loss sollte während des Trainings einen niedriger wer-

denden Trend aufweisen, wobei kleinere Schwankungen der Norm entsprechen. Wenn ein neuer niedrigster Validierungs-Loss erreicht wurde, wird das aktuelle Modell unter ***output_model_path*** abgespeichert. Das zuletzt abgespeicherte Modell ist also nicht zwingend das der letzten Epoche, dafür aber das mit dem niedrigstem Validierungs-Loss. Konkret liegen im ***output_model_path*** zwei Dateien: eine *config.json,* mit den Einstellungen der Modell-Architektur, und eine *model.safetensors*, welches die trainierten Modellgewichte enthält. Diese beiden Dateien bilden zusammen das Model.

Zusätzlich zum Modell werden während des Trainings verschiedene gängige Metriken ausgeschrieben, die bei der Bewertung eines Modells unterstützen können. Diese werden in ***output_train_metrics_path*** in *metrics.csv* und zusätzlich als Abbildung pro Metrik (als Verlauf über die Epochen) gespeichert. Die berechneten Metriken sind der bereits zuvor erwähnte Loss, die Genauigkeit, die Intersection over Union (IoU), und der F1-Wert, einmal für den Trainingsdatensatz und einmal für den Validierungsdatensatz. Dabei bildet jede Metrik verschiedene Aspekte ab. Die Genauigkeit misst das Verhältnis der korrekten Vorhersagen in Bezug auf die Gesamtzahl der Vorhersagen. Bei unausgeglichenen Klassenverhältnissen kann die dominierende Klasse die Metrik stark beeinflussen (siehe auch hier) und ist daher nicht immer aussagekräftig. Die IoU misst die Überlappung zwischen der Vorhersage und der „Ground Truth“ und gibt damit an wie genau die Position und Größe einer Vorhersage korrekt erfasst wurde (siehe auch hier). Der F1-Wert ist der Mittelwert zwei weiterer Metriken (Präzision und Recall) und ist u.a. bei unausgeglichenen Klassenverhältnissen nützlich (siehe auch hier).

Allgemein gilt, dass der Loss über das Training abnehmen, und die Genauigkeit, die IoU und der F1-Wert über das Training zunehmen soll. Dies gilt sowohl für den Verlauf der Trainingsdaten, als auch der Validierungsdaten. Wenn der Verlauf für Trainings- und Validierungsdaten zu stark voneinander abweicht, kann dies Rückschlüsse über z.B. Overfitting geben. Eine Hilfestellung wie die Kurven für ein generalisiertes Modell aussehen können, und welche weiteren Rückschlüsse der Verlauf der Metriken geben kann ist beispielsweise hier gegeben.

Die Eingangsparameter des Addons sind nachfolgend beschrieben.

Eingangsparameter:

- ***data_dir***: Pfad zum von *m.neural_network.preparedata_part2* erstellten ***train*** Ordner mit den Unterordnern ***train_images***, ***train_masks***, ***val_images*** und ***val_masks***
- ***img_size***: Größe der Trainingskacheln in Pixeln, Default: 512
- ***in_channels***: Anzahl der Input Kanäle, Default: 5, für Bilder mit RGBI und einem nDOM als 5. Kanal
- ***out_classes***: Anzahl zu erkennender Klassen, voreingestellt ist 2 für Baum – Nichtbaum Klassifikation
- ***model_arch***: Name der KI-Architektur, voreingestellt ist „Unet“.
- ***encoder_name***: Name des Encoders, voreingestellt ist „resnet34“.

- ***encoder_weights***: Gewichte für den Encoder, voreingestellt ist „imagenet“.
- ***epochs***: Anzahl Epochen (Trainingsdurchläufe), voreingestellt ist 50
- ***batch_size***: Batch Größe beim Training, es werden immer ***batch_size*** Kacheln auf einmal geladen, voreingestellt ist 8. Je nach verfügbarer RAM Größe kann dies angepasst werden
- ***input_model_path***: Pfad zu einem trainierten und lokal gespeicherten Modell (Finetuning); optional, sonst wird ein neues Modell mit den ***encoder_weights*** trainiert
- ***output_model_path***: Ordner, in dem das trainierte Modell gespeichert werden soll
- ***output_train_metrics_path***: Ordner, in dem die Metriken des Trainings als *metric.csv,* sowie als *.png* pro Metrik gespeichert werden sollen. Hinweis: die csv-Datei wird überschrieben, wenn schon eine in ***output_train_metrics_path*** existiert.

```
# Beispiel für das Training eines neuen Modells mit m.neural_network.train
# mit Defaultwerten

$ m.neural_network.train \
   data_dir=/mnt/data/labeled_data_2024_restructured/train \
   output_model_path=/mnt/data/labeled_data_2024_restructured_model \
   output_train_metrics_path=/mnt/data/labeled_data_2024_restructured_metrics
```

```
# Beispiel für das Weitertrainieren (fine-tuning) eines Modells
# mit m.neural_network.train mit Defaultwerten

$ m.neural_network.train \
   data_dir=/mnt/data/labeled_data_2024_restructured/train \
   input_model_path=/mnt/data/labeled_data_2024_restructured_model \
   output_model_path=/mnt/data/labeled_data_2024_restructured_finetune \
   output_train_metrics_path=/mnt/data/labeled_data_2024_restructured_metrics
```

2.3 Evaluieren eines KI-Modells

Ein KI-Modell kann mit *m.neural_network.test* evaluiert werden, dazu muss es bereits trainiert und lokal gespeichert sein. Zur Evaluierung wird ein möglichst unabhängiger Testdatensatz verwendet, um Konfusionsmatrizen zu erstellen und verschiedene Metriken wie Genauigkeit, IoU und F1-Wert zu berechnen.

Das Addon hat folgende **Eingangsparameter**:

- ***data_dir***: Pfad zum von *m.neural_network.preparedata_part2* erstellten ***train*** Ordner mit den Unterordnern ***test_images*, *test_masks***
- ***input_model_path***: Pfad zu einem trainierten und lokal gespeicherten Modell
- ***num_classes***: Anzahl zu erkennender Klassen, Default: 2 für Baum – Nichtbaum Klassifikation

- ***class_names***: Namen der zu erkennenden Klassen, Default: tree, no tree für Baum – Nichtbaum Klassifikation
- ***output_path***: Ordner, in dem die ausgegebenen Statistiken abgelegt werden sollen. Konkret sind dies die Konfusionsmatrix im *.png* und *.csv* Format, sowie die mit „True Class“ normalisierte Konfusionsmatrix (ebenfalls als *.png* und *.csv*). Außerdem wird eine txt-Datei *metrics_per_class* erstellt, in der die Gesamtgenauigkeit, die IoU pro Klasse und gemittelt, sowie der F1-Wert pro Klasse und gemittelt, enthalten sind.

```
# Beispiel für die Anwendung von m.neural_network.test

$ m.neural_network.test \
   data_dir=/mnt/data/labeled_data_2024_restructured/train \
   input_model_path=/mnt/data/labeled_data_2024_restructured_model \
   output_path=/mnt/data/labeled_data_2024_restructured_model/test_output
```

2.4 Anwenden eines KI-Modells

Wenn sich für ein Modell entschieden wurde, kann dieses mit *m.neural_network.apply* angewendet werden. Sollen hierbei Daten als Input verwendet werden, die bisher nicht für die Trainingsdatenaufbereitung genutzt wurden, ist dies problemlos möglich. Diese müssen wie die Trainingsdaten mit *m.neural_network.preparedata_part1* und *m.neural_network.preparedata_part2* aufbereitet werden (siehe Kapitel 2.1.1 und 2.1.3, dabei entsprechend die *-a*-Flag und nur den ***input_apply-dir*** Parameter nutzen).

Das Addon zum Anwenden des Modelles hat folgende **Eingangsparameter**:

- ***data_dir***: Pfad zum von *m.neural_network.preparedata_part2* erstellten ***apply*** Ordner mit dem Unterordner ***apply_images***
- ***input_model_path***: Pfad zu einem mit trainierten und lokal gespeicherten Modell
- ***num_classes***: Anzahl zu erkennender Klassen, Default: 2 für Baum – Nichtbaum Klassifikation
- ***output_path***: Ordner, in dem die klassifizierten Einzelkacheln abgelegt werden sollen

```
# Beispiel für die Anwendung von m.neural_network.apply

$ m.neural_network.apply \
   data_dir=/mnt/data/labeled_data_2024_restructured/apply \
   input_model_path=/mnt/data/labeled_data_2024_restructured_model \
   output=/mnt/data/labeled_data_2024_restructured/apply_output
```

2.5 Zusammensetzen der Klassifikations-Ergebnisse

Da das KI-Modell auf viele kleine Kacheln angewendet wurde und dementsprechend genau so viele neue Kacheln produziert wurden, müssen diese Ergebnis-Kacheln zusammengesetzt werden. Dies erfolgt mit dem Addon *m.neural_network.postprocessing_patch.* Bei diesem Schritt werden die Daten außerdem in GRASS GIS eingeladen, um für die folgende Einzelbaum-Postprozessierung im richtigen Format vorzuliegen.

Das Addon hat folgende **Eingangsparameter**:

- ***tiles_path***: Pfad zum von *m.neural_network.apply* erstellten Ordner mit den klassifizierten Einzelkacheln
- ***edge_cut:*** Rand der Einzelkacheln in Einheit Zellen, der vor dem Zusammensetzen abgeschnitten werden soll. Dies ist hilfreich um Randeffekte zu eliminieren. Default: 64

  Hinweis: Dies entspricht in der Regel der Hälfte der Überlappung aus der initialen Kachelung (***tile_overlap*** von *m.neural_network.preparedata_part1)*
- ***output:*** Name des zusammengefügten Raster Karte. Default: classification_patch
- ***tiles_filelist:*** Optionaler Input, wenn nicht alle Dateien aus dem ***tiles_path*** Ordner genutzt werden sollen. In diesem Falle kann eine txt-Datei mit diesem Parameter übergeben werden, welche die Dateinamen der Kacheln, die prozessiert werden sollen, enthält. Dabei muss ein Dateiname pro Zeile gegeben werden (Nur Dateiname, nicht der komplette Pfad)
- ***-b:*** optionale Flag damit äußere Bereiche von Randkacheln nicht abgeschnitten werden. Hilfreich wenn Ausdehnung des Untersuchungsgebietes nicht kleiner werden soll. Hinweis: dabei können Randeffekte bestehen bleiben

```
# Zuerst die Region für die Ausdehnung und Auflösung setzen!
$ g.region rast=ndsm

# Beispiel für die Anwendung von m.neural_network.postprocessing_patch
$ m.neural_network.postprocessing_patch \
   tiles_path=/mnt/data/labeled_data_2024_restructured/apply_output \
   output=classification_patch
```

3 Einzelbaum-Postprozessierung

In diesem Kapitel werden die benötigten Schritte beschrieben um aus dem *m.neural_network.postprocessing_patch* Raster Output die Vektor Einzelbäume zu generieren. Grundsätzlich werden hierfür dieselben Addons wie für die Postprozessierung nach dem Random Forest Ansatz benutzt (RVR-Interface). Hierbei werden keine Schwellenwerte zum weiteren Filtern gesetzt.

3.1 Geomorphologische Extraktion von Baumgipfel

Die geomorphologische Extraktion von Baumgipfeln erfolgt mit dem Addon *r.trees.peaks*. Dabei wird jedem Baumgipfel eine eigene ID vergeben. Die Ergebnisse dieses Addons sind eine Rasterkarte mit der am nächsten gelegenen Baumgipfel-ID, eine Rasterkarte mit geomorphologischen Gipfeln und Bergrücken und eine Rasterkarte mit dem Gefälle des nDOMs.

Die Nähe der Baumgipfel-ID und entsprechende Zuweisung der Pixel erfolgt mit einer sogenannten „Watershed Segmentation“, die das Gefälle des nDOMs als Kostenfaktor benutzt. Diese Objekte beinhalten nicht nur Bäume, sondern auch andere höhere Objekte.

Das Addon hat folgende **Eingangsparameter**:

- ***area***: Name des Vektors für das aktuelle Gebiet (wie mit *m.import.rvr* importiert, Default: study_area)
- ***ndsm***: Name des nDOM-Rasterdatensatzes (Default: ndsm)
- ***forms_res***: Auflösung zur Berechnung geomorphologischer Formen (Default: 0,8 m)

Ausgangsparameter:

- ***nearest***: Output Rasterkarte mit der ID des nächstgelegenen Baumgipfels (Default: nearest_tree)
- ***peaks***: Output Rasterkarte mit geomorphologischen Gipfeln und Bergrücken (Default: tree_peaks)
- ***slope***: Output Rasterkarte mit dem Gefälle des nDOMs (Default: ndsm_slope)

Für die Konfiguration der Parallelisierung können folgende Parameter gesetzt werden:

- ***tile_size***: Kantenlänge der Gitterkacheln für das Tiling in m (Default: 2000 m)
- ***memory***: Arbeitsspeicher in MB, der dem Addon zur Verfügung stehen soll (optional, Default: 300 MB)
- **nprocs**: Anzahl der Kerne für Parallelprozessierung (optional, Default: -2 (Anzahl der verfügbaren Kerne minus 1), für serielle Prozessierung auf 1 setzen)

Beispiel für den Aufruf von *r.trees.peaks*:

```
# r.trees.peaks für die Postprozessierung vom Output von
# m.neural_network.postprocessing_patch:
$ r.trees.peaks ndsm=ndsm nearest=nearest_tree peaks=tree_peaks memory=1000
```

3.2 Zusammenfassen der klassifizierten Baumpixel zu Einzelbäumen

Im Anschluss werden die vom neuronalen Netz klassifizierten Baumpixel mit dem Addon *r.trees.postprocess* zu Einzelbäumen zusammengefasst. Als Input werden neben einer Baumklassifikation (*output* von *m.neural_network.postprocessing_patch)* die Ergebnisse der geomorphologischen

Analyse mit *r.trees.peaks* verwendet. Im Folgenden werden die **Parameter** dieses Addons, die für die Postprozessierung des NN Ansatzes benötigt werden, genauer aufgeschlüsselt. Die benötigten Parameter für die Postprozessierung mit dem Random Forest Ansatz sind weiterhin verfügbar, aber da sie hier keine Anwendung finden, werden sie nicht weiter aufgeführt:

Eingangsparameter:

- ***tree_pixels****:* Name des Rasters mit der Baumklassifikation (Default: tree_pixels)

  Hinweis: der Output-Default von *m.neural_network.postprocessing_patch* ist *classification_patch*
- ***nearest***: Rasterkarte mit der ID des nächstgelegenen Baumgipfels (Default: nearest_tree)
- ***peaks***: Rasterkarte mit geomorphologischen Gipfeln und Bergrücken (Default: tree_peaks)
- ***area_threshold***: Mindestgröße für Einzelbäume in m² (Default: 5 m²)
- ***-n***: Flag, die gesetzt werden **muss**, wenn der NN Ansatz gewählt wurde. Hierbei wird die Filterung mit den Schwellenwerten ausgeschaltet, sowie für das NN notwendige Klassen-Anpassungen vorgenommen.

Beispiel für den Aufruf von *r.trees.postprocess*:

```
# r.trees.postprocess für die Postprozessierung vom NN Ansatz:
$ r.trees.postprocess tree_pixels=classification_patch \
   nearest=nearest_tree peaks=tree_peaks area_threshold=5 -n
```

Die Ableitung der Baumparameter kann wie gehabt erfolgen. Siehe dazu auch die Dokumentation der Baumstandorte hier.

4 Umgang mit GRASS Warnungen und Fehlern

Die GRASS-Module geben prinzipiell eher häufig Warnungen aus, um eine eventuelle Fehlersuche zu erleichtern. Diese dienen der Vollständigkeit und können daher normalerweise ignoriert werden. Sollte jedoch ein Modul nicht das gewünschte Ergebnis produzieren, lohnt es sich, die GRASS-Ausgabe auf eventuelle Warnungen hin zu überprüfen. Eine gute Anlaufstelle für mehr Informationen sind außerdem die Handbuch-Seiten der jeweiligen Module. Die Modul-Übersichten, geordnet nach Gruppen (*vector*, *raster*, *general*, etc.), können z. B. über diese Manual-Seite erreicht werden.

4.1 Häufige Fehler und Warnungen

Im Folgenden werden häufige Fehler und Warnungen beim Prozessieren von Daten mit GRASS GIS sowie der Umgang mit ihnen beschrieben. Darüber hinaus lohnt es sich bei Fehlern, insbesondere beim Im- und Export, auch in den Handbuch-Seiten der Module nach weiteren Optionen zu schauen, z. B. das Aktivieren von Flags oder Besonderheiten in den *NOTES* oder *EXAMPLES*.

4.1.1 Warnungen zu Packages

Beim Starten von GRASS GIS kann die folgende oder eine ähnliche Warnung auftreten:

```
$ DeprecationWarning: The distutils package is deprecated and slated for re-
moval in Python 3.12. Use setuptools or check PEP 632 for potential alterna-
tives from distutils.dir_util import copy_tree
```

Warnungen dieser Art können ignoriert werden. Sie beziehen sich auf Systemkomponenten (z. B. Python Packages), die zukünftig Änderungen erhalten werden. Dies hat keinen Einfluss auf die aktuelle Funktionstüchtigkeit von GRASS.

Speziell obige Warnung sollte bei der Nutzung von GRASS 8 (im Docker) nicht mehr auftreten, da die Kompatibilität bereits gewährleistet wurde. Generell laufen aktuelle GRASS-Versionen mit aktuellen Python-Versionen.

4.1.2 Warnungen beim Installieren von GRASS-Addons

Bei der Installation von GRASS-Addons können solche oder ähnliche Warnungen auftreten:

```
$ g.extension m.analyse.trees url=/src/grass-gis-addons/m.analyse.trees -s
...
WARNING: Unable to create '/usr/local/grass83/docs/html/grassdocs.css':
         '/usr/local/grass83/docs/html/grassdocs.css' and
         '/usr/local/grass83/docs/html/grassdocs.css' are the same file. Is
         the GRASS GIS documentation package installed? Installation
         continues, but documentation may not look right.
Compiling...
Installing...
Updating extensions metadata file...
Updating extension modules metadata file...
WARNING: No metadata available for module 'm.analyse.trees': Unable to
         fetch interface description for command '<m.analyse.trees>'.

         Details: <[Errno 2] No such file or directory:
         'm.analyse.trees'>
Installation of <m.analyse.trees> successfully finished
```

Die Warnungen zur GRASS GIS-Dokumentation während der Installation können ignoriert werden. Hier geht es um Metadaten und Dokumentation, die Funktion der Addons ist dadurch nicht eingeschränkt. Ob die Installation eines Addons erfolgreich verlaufen ist, kann an der letzten Zeile abgelesen werden.

4.1.3 Warnungen beim Nutzen der GUI

Beim Nutzen der Graphischen Oberfläche (GUI) treten mitunter folgende oder ähnliche Fehler auf:

```
$ (wxgui.py:6500): Gtk-CRITICAL * *: 13:46:27.691: gtk_box_gadget_distribute:
assertion 'size >= 0' failed in GtkScrollbar
$ (wxgui.py:6500): Gtk-CRITICAL * *: 14:38:27.866: gtk_box_gadget_distribute:
assertion 'size >= 0' failed in GtkCheckButton
$ (wxgui.py:6851): Gtk-CRITICAL * *: 11:49:20.118: gtk_box_gadget_distribute:
assertion 'size >= 0' failed in GtkSpinButton
```

Warnungen dieser Art können ignoriert werden. Die GUI funktioniert dennoch normal.

4.1.4 Warnungen zu inkorrekten Grenzen, *collapsed areas*, Flächenzentroiden, etc.

```
$ WARNUNG: Anzahl inkorrekter Grenzen: 8
WARNUNG: Vect_get_point_in_poly_isl(): collapsed area
WARNUNG: Kann den Flächenzentroiden nicht berechnen.
WARNUNG: Vect_get_point_in_poly_isl(): collapsed area
WARNUNG: Kann keinen Zentroid für die Fläche 101268 berechnen.
WARNUNG: Vect_get_point_in_poly_isl(): collapsed area
WARNUNG: Kann keinen Zentroid für die Fläche 201928 berechnen.
WARNUNG: Flächen mit Größe 0.0 ignoriert.
WARNUNG: Anzahl inkorrekter Grenzen: 7
WARNUNG: Die Datenbankverbindung für den Layer<1> ist nicht definiert
```

Treten Warnungen dieser Art beim Import von Daten auf, sollten sie **nicht** ignoriert werden. Bei der anschließenden Analyse hingegen können die Warnungen ignoriert werden, da die genutzten GRASS-Module, wie. z.B. *v.patch,* in der Regel auch eine topologische Säuberung beinhalten. Das heißt die Warnungen werden der Vollständigkeit halber ausgegeben und eventuelle Probleme dann direkt behoben. Eine Datenbankverbindung ist für GRASS-Vektorlayer nicht zwingend erforderlich, deswegen kann diese Warnung meist ignoriert werden. Für den Fall, dass Vektor-Attribute fehlen, könnte hier jedoch ein Grund dafür liegen.

4.1.5 Warnung beim Export: features without category were skipped

Die Warnung kann ignoriert werden. Es ist empfehlenswert, die *-c*-Flag beim Vektor-Export mit *v.out.ogr* nicht zu setzen. Bei Nutzung der *-c*-Flag wird das Ergebnis nicht das Erwartete sein, da in der *Simple Feature Definition* damit Polygone erstellt werden, die es eigentlich gar nicht gibt. Die C*ategory* als GRASS-Feature ID ist nicht zu verwechseln mit den Attributen von Features (Polygonen). „*Features without category*“ bezieht sich auf Löcher in Polygonen. Diese haben weder eine eigene ID noch Attribute.

```
$ v.out.ogr in=result_trees out=/pfad/zum/output.gpkg
...
WARNUNG: 14 features without category were skipped. Features without
         category are written only when -c flag is given.
```

4.1.6 Fehler beim Export der Farbtabelle von Rasterdaten (z. B. DOM, nDOM) als GeoTiff

```
$ r.out.gdal input=ndsm_map output=/pfad/zum/output.tif format=GTiff
Using GDAL data type <Float32>
Die Eingabe-Rasterkarte enthält Zellen mit dem NULL-Wert (no-data). Der
Wert -nan wird verwendet, um NoData-Werte in der Eingabekarte zu
kennzeichnen. Sie können NoData-Wert mit dem Parameter nodata bestimmen.
ERROR 6: /pfad/zu/tiff/nDOM.tif, band 1: SetColorTable() only supported for
Byte or UInt16 bands in TIFF format.
```

Beim Export als GTiff ist es für Raster, die nicht als Byte oder UInt16 Datentyp vorliegen, empfehlenswert, die Farbregeln nicht mit zu exportieren. Dafür kann die *-c*-Flag gesetzt werden. Für maximale Kompatibilität mit anderen GIS-Programmen ist auch die *-m*-Flag empfehlenswert. Die Farbregeln können alternativ mit der *-t*-Flag in eine separate Raster-Attributtabelle geschrieben werden. Informationen zu den Flags finden sich auf der Handbuchseite von r.out.gdal.

Der Export kann z.B. mit folgendem Befehl wiederholt werden:

```
$ r.out.gdal -cmt input=ndsm_map output=/pfad/zum/output.tif format=GTiff --o
```

5 Referenzen

Hier befindet sich eine Übersicht der wichtigsten Referenzen in dieser Dokumentation. Kontaktieren Sie uns gerne bei Fragen oder Problemen.

Kontakt mundialis

Ansprechpersonen:

V. Brunn	brunn@mundialis.de
G. Riembauer	riembauer@mundialis.de
A. Weinmann	weinmann@mundialis.de
L. Krisztian	krisztian@mundialis.de
M. Eichhorn	eichhorn@mundialis.de

GitHub Repository mit Skripten und GRASS-Addons

https://github.com/mundialis/rvr_interface

GRASS GIS

GRASS Website

- GRASS GIS Homepage
- GRASS GIS Download

GRASS GIS Manual

- GRASS GIS Manual
- GRASS Basics im Manual

GRASS GIS Wiki

- GRASS GIS Wiki
- GRASS User Wiki Installation Guide

Tutorials

- GRASS GIS Workshop
- Analysing environmental data with GRASS GIS