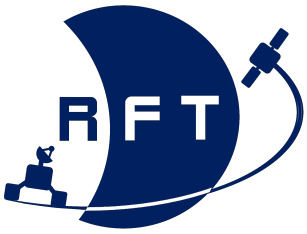


Technische Universität Berlin
Fakultät V für Maschinenbau und Verkehrssysteme

Bachelor's Thesis

Implementation of an AI algorithm for cloud detection on an embedded system Tensor Processing Unit

Author:
Andrei Zharski

August 26, 2025

Supervisor:
Prof. Dr.-Ing. Enrico Stoll
Lehrstuhl für Raumfahrttechnik
Technische Universität Berlin

Second supervisor:
M.Sc. Alexander Balke

Selbständigkeitserklärung gemäß § 60 Abs. 8 AllgStuPO

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt. Hinweis zum Einsatz von KI-Werkzeugen: Das KI-Tool ChatGPT wurde ausschließlich zur grammatikalischen Überprüfung und Verbesserung des Textes verwendet. Die inhaltliche Ausarbeitung stammt von mir. Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 15. Februar 2023. https://www.static.tu.berlin/fileadmin/www/10002457/K3-AMBl/Amtsblatt_2023/Amtliches_Mitteilungsblatt_Nr._16_vom_30.05.2023.pdf habe ich zur Kenntnis genommen. Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

26.08.2025, Berlin

Datum, Ort

Unterschrift der/des Studierenden

Agreement on utilization rights

According to § 60 of the AllgStuPO, students of the TU Berlin must complete a final thesis during their studies. The totality of the achievements associated with a final thesis are based not only on the written documentation and the commitment of the student, but also on the supervisory effort of the department, including the preparation of the assignment, the specific requirements, guidance of the students, help and advice in organizational, technical and scientific terms. Therefore, the student is the sole author of the written work, but not the sole author of the entirety of the services associated with a thesis.

By supervising the student, the Technische Universität Berlin, represented by the Chair of Astronautics, becomes co-author of the entirety of the achievements and the associated results according to § 8 section 1 of the German Copyright Act (Urheberrechtsgesetz).

For non-profit use in teaching and research, the Chair may use the results of the present work. As co-author, the chair is entitled to do so according to § 8 section 2 of the Copyright Act. If the work is written independently in such a way that there is no co-authorship, this right of use is granted by the student according to § 31 section 2 of the Copyright Act. This right of use is unrestricted and includes content of any kind (e.g. documentation, presentations, photos, videos, procedures, drafts, drawings, software including source code and the like) with naming of the author.

Due to the reference to ongoing research projects, publication without explicit permission of the department will not be permitted to the student. This is to be checked and approved by the department in each individual case.

Any commercial use of either site will only occur with the consent of all authors of the present work, with appropriate participation in the earnings.

26.08.2025

Date, Place

Signature of the Student

Zusammenfassung

Die Technische Universität Berlin hat am 01.01.2022 das Projekt AITHER ins Leben gerufen. Im Rahmen dieses Projekts wurde der *OOV-CUBE* Satellit in die Erdumlaufbahn gebracht. An Bord befindet sich die KI-Datenverarbeitungseinheit, auf der rechenintensive und zuverlässige Aufgaben durchgeführt werden sollen. Ziel des Projekts ist die Untersuchung der Strahlungstoleranz verschiedener Onboard-Architekturen unter Weltraumbedingungen. Zu diesem Zweck soll ein KI-Modell entwickelt werden, das Rechenoperationen direkt an Bord ausführt. Die Aufgabe zur Wolkenerkennung auf den aufgenommenen Satellitenbildern eignet sich besonders gut dafür. Für die Berechnungen ist der KI-Beschleuniger, in diesem Fall eine *Tensor Processing Unit (TPU)* auf einem *Google Coral Dev Board Mini (Dev Board)*, zuständig. Die TPU wird die Tensoroperationen des Algorithmus zur Wolkenerkennung direkt im Orbit ausführen.

In dieser Bachelorarbeit wird die Erkennung der Wolken auf den Satellitenbildern mithilfe von *Convolutional Neural Network (CNN)* implementiert. Als Ausgangspunkt werden die *state-of-the-art* Architekturen der bestehenden CNNs zur Wolkenerkennung untersucht, wie sie beispielsweise in den wissenschaftlichen Artikeln [MS19] und [S M18] beschrieben sind. Das CNN wird speziell für die Ausführung auf einer eingebetteten Plattform *Dev Board* angepasst. Die fertige Lösung soll eine miniaturisierte Onboard-Datenverarbeitungseinheit sein, die auf den aufgenommenen Bildern direkt auf dem Satelliten im Orbit die Wolken erkennen kann.

Es wird ein Datensatz aus der Landsat 8 Mission verwendet. Er umfasst 38 Wolkenbilder, die aus 4 Kanälen bestehen: *Red, Green, Blue (RGB)* und *Near Infrared (NIR)*. Um das Modell zu trainieren und zu validieren, wird dieser in Trainings- und Testdaten aufgeteilt. Die Funktionalität des Modells soll überprüft werden und die Performanz auf dem eingebetteten System wird evaluiert.

In weiteren Schritten sollte die Software auf den sich bereits im Orbit befindenden Satelliten *OOV-CUBE* hochgeladen werden. Mit folgender wissenschaftlicher Mission wird nicht nur die Robustheit der handelsüblichen elektronischen Komponenten gegenüber kosmischen Strahlung untersucht, sondern auch die Möglichkeit geprüft, den Downlink vom Satelliten zur Erde zu entlasten, um die Effizienz von Erdbeobachtungs- und Kommunikationsmissionen zu erhöhen.

Abstract

The Technical University of Berlin launched the AITHER project on January 1st, 2022. As part of this project the OOV-CUBE satellite was placed into Earth orbit. An onboard AI data processing unit is designed to perform computationally intensive and reliable tasks. The goal of the project is to study the radiation tolerance of different onboard architectures under space conditions. For this purpose, an AI model is being developed to perform calculations directly onboard. Cloud detection in satellite images is particularly well suited for this task. These computations are carried out by a dedicated AI hardware accelerator — a TPU embedded on a Dev Board. The TPU will execute the tensor operations of the cloud detection algorithm directly in orbit.

In this bachelor's thesis, the detection of clouds in satellite images is implemented using CNN. The thesis begins by reviewing state-of-the-art CNN architectures for cloud detection, as described in scientific articles [MS19] and [S M18]. The CNN is specifically adapted to run on the Dev Board. The final solution will be a miniaturized data processing unit capable of detecting clouds directly onboard the satellite.

A dataset from the Landsat 8 mission is used. It contains 38 cloud images, each consisting of four channels: RGB, and NIR. The dataset is divided into training and test sets for model training and validation. The model's functionality will be verified, and its performance on the embedded system will be evaluated.

In the next steps, the software is intended to be uploaded to the OOV-CUBE satellite already in orbit. The scientific mission aims not only to study the robustness of commercial electronic components against cosmic radiation but also to evaluate the potential for relieving the satellite-to-Earth downlink. This would improve the efficiency of Earth observation and communication missions.

Contents

List of Acronyms

API Application Programming Interface 27, 29, 39, 44–47

BCE Binary Crossentropy 50, 51

BN Batch Normalization 27, 32, 33, 36, 37, 50

CNN Convolutional Neural Network 4, 5, 10–13, 16, 23, 27, 54

COTS Commercial Off-The-Shelf 9, 10, 54

CPU Central Processing Unit 10, 29, 42, 43, 48

CUDA Compute Unified Device Architecture 29

cuDNN CUDA Deep Neural Network Library 29

Dev Board Google Coral Dev Board Mini 4, 5, 9, 10, 20, 23, 24, 27, 39, 40, 44–48, 50, 51, 53, 54

Edge TPU Google Edge TPU Accelerator 9, 10, 20–24, 28, 30, 37–45, 48, 51, 54

float32 Single-precision floating-point format 14, 19, 20, 25, 30, 31, 34–36, 46, 48

GPU Graphics Processing Unit 29, 38

GT Ground Truth 12–16, 22, 24, 25, 44, 45, 48, 49

int8 8-bit signed integer 21, 30, 31, 48

LSB Least Significant Bit 35, 37

NIR Near Infrared 4, 5, 10, 11, 14, 16, 24, 25, 30, 50, 52–54

PTQ Post Training Quantization 19, 21, 28, 30, 31, 37, 40

QAT Quantization Aware Training 19, 21, 22, 27–32, 40, 41, 50

RAM Random-Access Memory 38, 46–48, 51

RGB Red, Green, Blue 4, 5, 10, 11, 14, 16, 24, 25, 30, 50, 53, 54

TF TensorFlow 19, 22, 24–29, 34–36, 40–44, 46–48

TF Lite TensorFlow Lite 20, 30, 40, 41, 43–48

TF op TensorFlow Operation 27, 29, 30, 34, 36–44, 50

TFMOT TensorFlow Model Optimization 40–44

TPU Tensor Processing Unit 4, 5, 10, 48

uint16 16-bit unsigned integer 14

uint8 8-bit unsigned integer 14, 21, 25

1 Introduction

1.1 Motivation

Electronic components in satellites and spacecraft are exposed to intense radiation, high-energy particles, and extreme temperature fluctuations. To ensure reliability under these harsh conditions, manufacturers traditionally rely on certified space-grade components. However, design, production, and certification of such components are costly and time-consuming.

The AITHER project aims to investigate whether Commercial Off-The-Shelf (COTS) components can be effectively used in nano- and microsatellites (10-100 kg). Specifically, the project explores the feasibility of using COTS-based onboard architecture to perform reliable and computationally demanding tasks, such as matrix operations in deep-learning based cloud segmentation, directly in space.

As part of this initiative, the 10kg nanosatellite OOV-CUBE was launched into orbit on July 9th 2024. Among its hosted payloads is the Dev Board — a single-board computer developed by Google with an embedded Google Edge TPU Accelerator (Edge TPU) designed for fast, low-power machine learning inference in constrained environments.

Alongside radiation shielding and thermal management strategies, a technology demonstrator is being developed to assess the tolerance of these components to space radiation. A central objective of the project is to determine not only how broadly COTS components can be applied in space systems, but also whether complex image processing tasks can be carried out onboard. This would reduce the need for data downlink and significantly improve the efficiency of Earth observation and communication missions.

1.2 Objective

While the satellite's structure was designed with radiation shielding and thermal management in mind, the software for onboard inference still needs to be developed and uplinked into the orbit. This thesis focuses on the implementation of CNN for cloud segmentation from satellite imagery. Although efficient CNN architectures already exist and have been described in recent scientific literature, porting such a model to an embedded platform introduces unique challenges. These include limitations in memory, computation power of both Central Processing Unit (CPU) and TPU as well as model format compatibility.

The goal of this thesis is to design, train and deploy a CNN model for cloud segmentation on the Dev Board. The core task is to adapt the model for efficient inference on the Edge TPU, overcoming hardware constraints while maintaining acceptable segmentation performance. This work serves as a demonstration of the potential to carry out deep learning inference onboard a satellite using COTS hardware.

1.3 Scope and Delimitations

This work focuses on overcoming the implementation challenges of a deep learning algorithm on an embedded system. The `Cloud-Net` [MS19] architecture for binary cloud segmentation in satellite images will be modified for deployment on the Google Edge TPU Accelerator located on the Google Coral Dev Board Mini. The pipeline foundation will be laid to enable the transfer of other AI algorithms to the Edge TPU format in the future. Predictions of a CNN, adapted for embedded inference on the Edge TPU, will be evaluated.

Model training will be limited to the 38-Cloud dataset referenced in [MS19]. The RGB and NIR channels of the input satellite imagery may be used. Extensive hyperparameter tuning to achieve extraordinary performance in cloud segmentation will not be undertaken. Software engineering requirements specific to satellite development are not considered.

2 Preliminaries

2.1 Cloud Segmentation in Satellite Imagery

The first successful weather satellite, TIROS-1, was launched on April 1, 1960. During its mission it returned thousands of photographs showing cloud-cover views of the Earth, which proved extremely helpful for predicting large-scale weather changes and significantly improving forecasting capabilities.

From this point, the need to distinguish cloud from non-cloud regions became clear, either to remove clouds for surface-focused analyses or to study the cloud field itself, both of which require separating clouds from the rest of the image. In time-series for weather forecasting, the motion of cloud layers can be tracked and predicted by expert analysts. Automating this process is one precise example where cloud segmentation is essential.

Early segmentation workflows relied on manual inspection and empirical thresholding. Over the years, these evolved into image processing methods, then classical machine learning, and today into deep learning models, most notably CNNs, to achieve reliable, scalable cloud segmentation.

Automated extraction of cloud masks can be challenging when analyzing only the visible light bands: RGB. With additional sensors, satellites capture other wavelengths such as Near Infrared, Shortwave Infrared, and Thermal Infrared. Each of these channels can benefit cloud segmentation in its own way. Given the 38-Cloud dataset, which contains RGB and NIR channel images from the Landsat 8 mission, this work will focus on them. NIR is especially helpful for distinguishing clouds over liquid water, where it is strongly absorbed. It also helps separate clouds from green vegetation, which is bright in the NIR band.

However, the OOV-CUBE satellite hosts only an RGB camera onboard. Cloud segmentation utilizing only these channels is possible, but may yield less accurate masks. Therefore, the necessity to provide and evaluate a model that uses only the visible channels will be considered in this work.

2.2 Convolutional Neural Networks for Cloud Segmentation

CNNs are a class of neural networks that apply small, learnable filters (convolutional kernels) across an image to extract spatial features. A CNN typically consists of multiple convolutional layers stacked sequentially. Each layer applies a set of filters that capture different visual patterns, such as edges, textures, or simple shapes. As the network goes deeper, the spatial resolution of the image decreases, while the depth (number of channels) increases. This is a result of applying multiple filters and optionally using pooling layers, which downsample feature maps to reduce computational complexity and introduce spatial invariance.

In classical image processing, filter parameters can be hand-crafted to extract specific features. In deep learning, however, these parameters are learned from data. Given a sufficiently deep network of filters, gradient descent techniques are applied to a loss function that measures the discrepancy between the network's output and the known Ground Truth (GT). In this way, the network is optimized to extract the necessary information from images.

In image segmentation tasks such as cloud detection, preserving spatial resolution is critical. Therefore, architectures often include an upsampling mechanism to reconstruct high-resolution output from compressed feature representations. This is achieved through transposed convolution (also known as deconvolution). While standard convolution reduces spatial resolution by aggregating local pixel values, transposed convolution performs the reverse: it distributes each value in the smaller feature map across a larger output, effectively increasing spatial dimensions and reversing the compression.

One of the most influential architectures for image segmentation is the `U-Net`. Originally introduced by Ronneberger et al. [RFB15] for biomedical image segmentation, `U-Net` has become a standard in many domains, including remote sensing and cloud detection. This architecture has proven highly effective for segmentation tasks, even with limited training data. A `U-Net` consists of an encoder-decoder structure. The encoder compresses the input image spatially while increasing its feature dimensionality. The decoder then reconstructs the spatial dimensions, progressively reducing the number of channels. This yields an output image with the same spatial dimensions (height and width) as the input, but could have a various number of channels, depending on the perceived task. The `U-Net` architecture shown in Figure 1 was used by Mohajerani et al. [MS19] for their cloud detection algorithm.

A key innovation in `U-Net` is the use of residual (skip) connections [He+15], which directly link feature maps from the encoder to corresponding layers in the decoder with the same spatial size. These connections preserve fine-grained spatial details and significantly enhance segmentation quality. Moreover, they mitigate the vanishing gradient problem, facilitating the training of deeper networks and improving convergence.

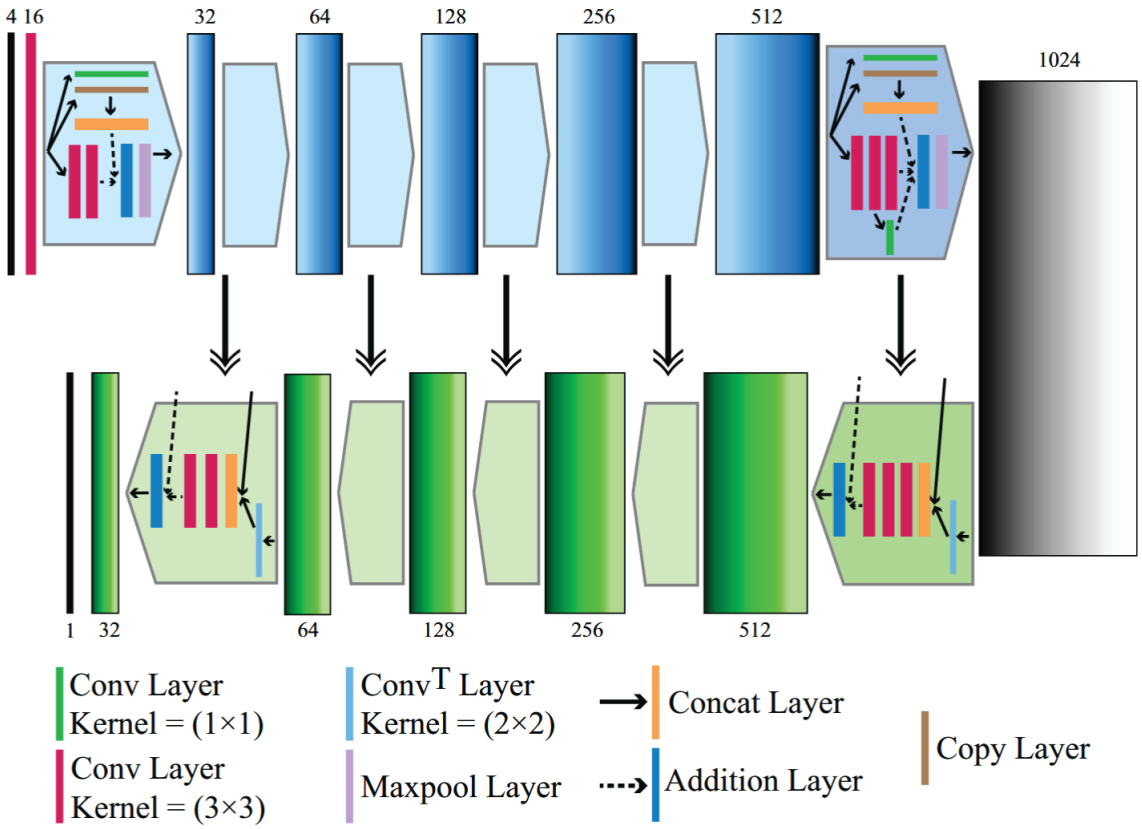


Figure 1: U-Net architecture adapted for cloud detection by Mohajerani et al. [MS19]

2.3 Evaluation Metrics for Cloud Segmentation

After performing a binary cloud segmentation task, a CNN outputs a probability mask with continuous values in $[0, 1]$. A threshold is then selected to binarize the probability mask: values at or above the threshold are set to one (cloud present), and values below the threshold are set to zero (no cloud present).

The predicted cloud mask is then compared with the GT mask. For each overlapping pixel, four outcomes are possible:

- **True Positive (TP):** Pixel is cloud in the prediction and in the GT.
- **True Negative (TN):** Pixel is not cloud in the prediction and in the GT.
- **False Positive (FP):** Pixel is cloud in the prediction but not in the GT.
- **False Negative (FN):** Pixel is not cloud in the prediction but cloud in the GT.

Using these outcomes, widely accepted metrics for evaluating CNN performance on the cloud segmentation task are defined. The following metrics will be used in this work:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN} \qquad \text{Jaccard Index (IoU)} = \frac{TP}{TP + FP + FN}$$

$$\text{Dice Coefficient} = \frac{2TP}{2TP + FP + FN}$$

Accuracy represents the fraction of correctly predicted pixels relative to the total number of pixels. Precision reflects how selective the model is: high precision means the model labels a pixel as cloud only when confident, which may increase false negatives (missed clouds). High recall, on the other hand, indicates the model tries to catch as many clouds as possible, which may increase false positives. The Jaccard Index (Intersection over Union) [Jac01] and the Dice Coefficient [Dic45] both relate precision and recall and measure the overlap between the predicted and GT sets.

Note that the optimal threshold, the value that yields the closest match between the predicted binary mask and the GT, is not necessarily 0.5. It can lie anywhere in $[0, 1]$. A common approach is to evaluate the precision-recall curve and select the threshold according to a chosen criterion (for example, maximum F1-Score) [DG06; Bla79].

2.4 38-Cloud dataset

Landsat 8[1] is an Earth observation satellite launched on February 11, 2013, providing high-resolution multispectral imagery, including RGB, NIR, and Thermal Infrared bands. This thesis utilizes a dataset consisting of 38 annotated satellite images from the Landsat 8 mission, commonly referred to as the 38-Cloud dataset[2]. It has been introduced and adapted in the following scientific publications [S M18; MS19]. 38 images are divided into a training set containing 18 scenes and a test set with the remaining 20 scenes. The folder structure of the dataset is illustrated in Figure 2.

Each image consists of four spectral channels: RGB and NIR, along with a manually annotated GT mask that labels cloud regions at the pixel level. The four spectral channels are encoded using 16-bit unsigned integers (uint16s) per pixel, whereas the GT masks are represented with 8-bit unsigned integers (uint8s). A single raw image at full resolution of approximately 8000×8000 pixels would result in a file size of around 1GB. Moreover, processing such images would require model input tensors of shape $batchsize \times 8000 \times 8000 \times 4$ in Single-precision floating-point format (float32) after normalization. Since efficient training typically necessitates a *batchsize* greater than one, this setup would impose substantial computational demands due to increased model size and would significantly slow down both training and inference — making it particularly unsuitable for deployment on embedded systems. To address this, the dataset is provided in a pre-cropped format, with each image and its corresponding GT mask divided into 384×384 pixel patches. These patches are saved as `.TIF` files within their

[1] https://landsat.gsfc.nasa.gov/satellites/landsat-8/
[2] https://github.com/SorourMo/38-Cloud-A-Cloud-Segmentation-Dataset

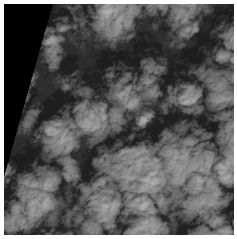
respective channel-specific directories, such as `train_red`, `train_green`, and so forth. The GT patches for training are stored in the `train_gt` directory.

It is important to note, however, that pre-cropped GT patches are not available for the test subset. Instead, complete scene GT masks are provided in the `Entire_scene_gts` directories for both the train and test subsets.

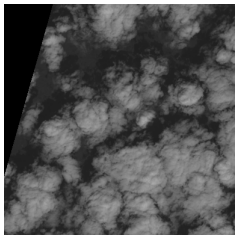
```
38-Cloud Dataset
├── 38-Cloud_training
│   ├── train_red
│   ├── train_green
│   ├── train_blue
│   ├── train_nir
│   ├── train_gt
│   ├── Natural_False_Color
│   ├── Entire_scene_gts
│   ├── training_patches_38-Cloud.csv
│   └── training_sceneids_38-Cloud.csv
├── 38-Cloud_test
│   ├── test_red
│   ├── test_green
│   ├── test_blue
│   ├── test_nir
│   ├── Natural_False_Color
│   ├── Entire_scene_gts
│   ├── test_patches_38-Cloud.csv
│   └── test_sceneids_38-Cloud.csv
└── training_patches_38-cloud_nonempty.csv
```

Figure 2: Folder structure of the 38-Cloud dataset.

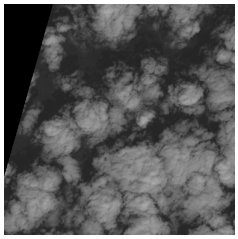
The examples of training patches from all four spectral channels and the corresponding GT mask at a fixed location are shown in Figure 3 as normalized greyscale images.



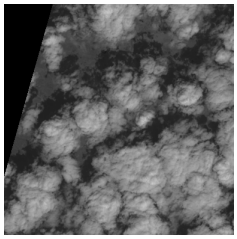
(a) Red



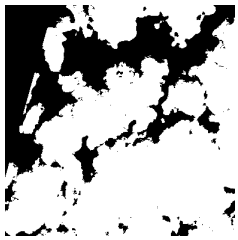
(b) Green



(c) Blue



(d) NIR



(e) GT

Figure 3: Four spectral input patches and the corresponding GT mask.

Each RGB, NIR, and GT patch follows a unified filename convention:

```
<band>_patch_<index>_<row>_by_<col>_<metadata>.TIF
```

The corresponding components are:

band ..... Indicates the spectral band or GT mask.
index ..... Ordinal number of the patch within the scene. Patches are extracted sequentially from left to right, top to bottom.
row & col .. Patch's position in the scene, specified by its row and column indices.
metadata ... Encodes the satellite and sensor ID, preprocessing precision, scene path/row, acquisition and processing dates, and collection tier metadata.

An example filename:

```
blue_patch_103_5_by_11_LC08_L1TP_063013_20160920_20170221_01_T1.TIF
```

This naming scheme ensures consistency and supports automated patch retrieval across channels. The filenames, excluding the `band` prefix, are listed in the `training_patches_38-Cloud.csv` and `test_patches_38-Cloud.csv` files, respectively. The corresponding scene-level filenames are stored in `training_sceneids_38-Cloud.csv` and `test_sceneids_38-Cloud.csv`.

One notable detail is that, due to the cropping and padding of border patches (to standardise the patch size to 384×384 pixels) and the tilted geometry of Landsat 8 imagery, some resulting patches contain no meaningful information — appearing completely black with all-zero pixel values across all four channels. These zero-information patches were excluded from the training and validation sets to avoid overrepresenting non-informative inputs, which could degrade the model's ability to learn meaningful patterns. The remaining useful patch names are stored in `training_patches_38-cloud_nonempty.csv` file.

Together, these `.csv` files serve as the basis for dataset construction in section 3.1. Furthermore, the attributes such as `index`, `row & col` will be directly utilized during the stitching process described in subsection 3.1.2.

For visualization purposes, the full scene images are rendered using a false-color composite. However, these visualizations were not used as input for the CNN during training, validation, or inference. They served solely as visual references for qualitative inspection by the human observer. The corresponding images are located in the `Natural_False_Color` directories. An example scene is shown below in Figure 4:

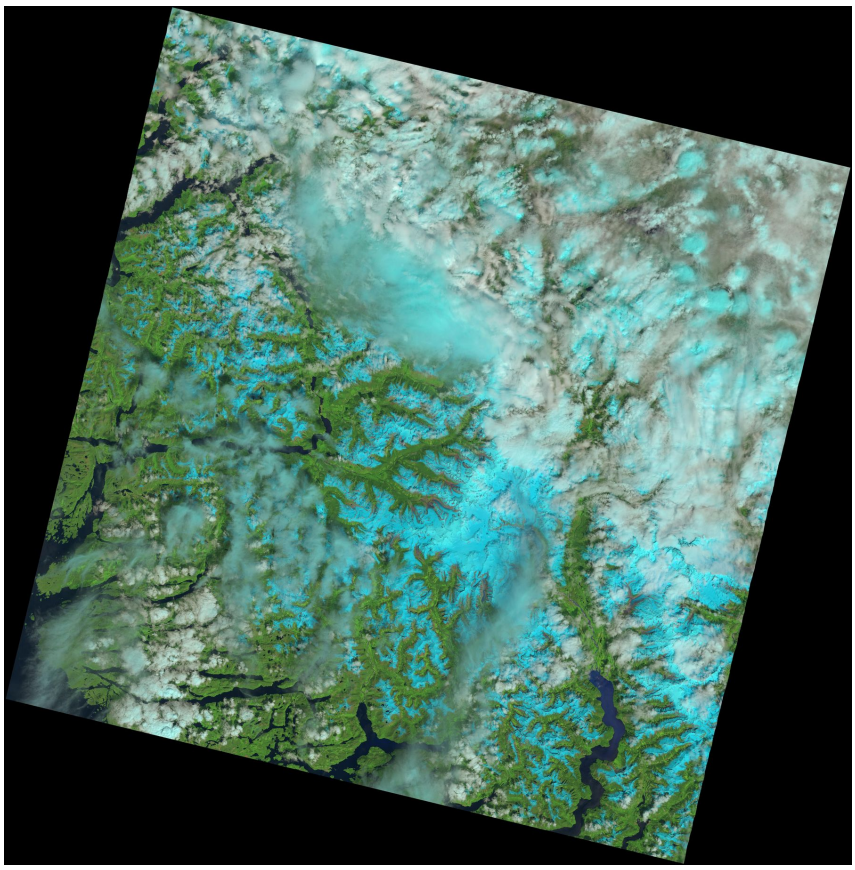


Figure 4: False-color composite of an entire scene.

2.5 Quantization Concept

Quantization is a mathematical method that maps a large set of (typically continuous) values to a smaller, discrete and countable set. Its first practical application can be traced back to 1957, in the context of pulse-code modulation within the field of signal processing. The earliest formal scientific documentation of the method appears in a publication from 1982, which is based on a draft manuscript originally authored in 1957 [Llo82].

This technique is now widely employed in machine learning [Wu+16; Hub+16; Nag+21]. Quantization significantly reduces model size [HMD16] by compressing numerical precision, typically at the cost of a controlled reduction in accuracy. Two primary quantization methods are commonly used: general asymmetric zero-point quantization, and its special case — symmetric absolute maximum (absmax) quantization. In the context of this work, the general asymmetric approach is of primary importance.

To perform quantization, the boundaries x_{min} and x_{max} of the original (floating-point) range, and q_{min} and q_{max} of the target (quantized) value set must be defined. Once these are known, the corresponding scale and zero-point parameters are computed according to the following equations:

$$\text{scale} = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}} \tag{2.1}$$

$$\text{zero-point} = \text{round}\left(q_{\min} - \frac{x_{\min}}{\text{scale}}\right) \tag{2.2}$$

With these parameters determined, input value x can be transformed into quantized form q_x and subsequently dequantized to an approximate value x_q using the following formulas:

$$q_x = \text{round}\left(\frac{x}{\text{scale}}\right) + \text{zero-point} \tag{2.3}$$

$$x_q \approx (q_x - \text{zero-point}) \cdot \text{scale} \tag{2.4}$$

It is essential to note that both the scale and zero-point must be preserved in order to carry out the quantization and dequantization processes. For every value range subject to quantization, a unique pair of these parameters exists.

The following example demonstrates the use of zero-point quantization. Consider a set of continuous values ranging from -9.75 to 3.00. To map this range onto a discrete set defined by the integer interval [-128, 127], the scale and zero-point are calculated using Equation 2.1 and Equation 2.2:

$$\text{scale} = \frac{3.00 - (-9.75)}{127 - (-128)} = 0.05 \qquad \text{zero-point} = \text{round}\left(-128 - \frac{-9.75}{\text{scale}}\right) = 67$$

After this step, every value from the original dataset can be represented by its corresponding quantized counterpart. The following calculations illustrate the quantization and subsequent dequantization of four examples using Equation 2.3 and Equation 2.4: $x = 0$, $x = 3.00$, $x = 2.98$, $x = 2.95$:

$$\begin{aligned}
q_0 &= \text{round}\left(\tfrac{0}{0.05}\right) + 67 = 67 & x_{67} &\approx (67 - 67) \cdot 0.05 = 0 \\
q_{3.00} &= \text{round}\left(\tfrac{3.00}{0.05}\right) + 67 = 127 & x_{127} &\approx (127 - 67) \cdot 0.05 = 3.00 \\
q_{2.98} &= \text{round}\left(\tfrac{2.98}{0.05}\right) + 67 = 127 & x_{127} &\approx (127 - 67) \cdot 0.05 = 3.00 \\
q_{2.95} &= \text{round}\left(\tfrac{2.95}{0.05}\right) + 67 = 126 & x_{126} &\approx (126 - 67) \cdot 0.05 = 2.95
\end{aligned}$$

The last three examples illustrate the precision loss introduced by quantization. In this case, any input between 2.925 and 3.00 (depending on the rounding convention) is mapped, after conversion, to one of two discrete values: 126 or 127.

It is important to emphasize that the scale parameter etirely defines the quantizer's precision, under the assumption that the bit-width (256 discrete representable values in this case) is fixed, as well as the floating-point range from -9.75 to 3.00 being evenly distributed across the entire interval and free of outliers.

In practice, more advanced quantization techniques may be employed, such as outlier-aware clipping, per-axis and per-tensor quantization, symmetric and asymmetric schemes, or methods like dynamic range adjustment and mixed-precision quantization. While the detailed explanation of these methods falls outside the scope of this thesis, it is important to note that such techniques are already available within various TensorFlow (TF) [Aba+16] utilities, which will be utilized in this work.

To quantize a deep learning model, its weights, biases, and, if required, its inputs, activations, and outputs must be quantized. This is performed through Post Training Quantization (PTQ). Due to the previously described loss of precision introduced by quantization, model effectiveness metrics may degrade. To mitigate this effect, the model architecture can be adapted to perform Quantization Aware Training (QAT) prior to PTQ. QAT does require additional computational operations, resulting in increased training time. It should be clarified that model training can be performed using various numerical data types for model parameters; in this work float32 is used.

2.6 Quantization-Aware Training

The model architecture is modified by inserting quantization-dequantization operations at the connections between layers, and by applying them to weights, biases, and activations. In the forward pass, dequantization is applied immediately after quantization for each QAT-annotated float32 value. In this way, all values remain in float32 format for training, but quantization inaccuracies are introduced, exposing the model to quantization effects during training and allowing it to adapt its parameters accordingly.

The rounding function essential to quantization is, however, not differentiable. For this reason, the Straight-Through Estimator [Hub+16] is applied during the backward pass, setting the gradient of quantization operations to 1 within the quantized range and to 0 outside of it. This allows gradients to flow through the quantization-dequantization operations.

At the end of QAT, the model parameters are adapted to the effects of quantization and will therefore exhibit better performance after PTQ compared to a model without QAT annotation.

2.7 Post-Training Quantization

The actual conversion of model parameters is performed during PTQ. Regardless of whether QAT was performed, PTQ can always be applied to quantize a model. There are three common PTQ methods, with each subsequent method quantizing more parameters until covering all float32 values in the model:

- **Weight-Only Quantization**: Model weights and biases are statically determined after training; hence, their minimum and maximum values can be directly observed and used to calculate the scales and zero-points required for quantization. Model inputs, activations, and outputs, however, remain in float32 format. This method already reduces model size and speeds up inference. However, additional computations are needed at runtime to interface between quantized weights/biases and float32 inputs, activations, and outputs.
- **Dynamic Range Quantization**: This method extends weight-only quantization by quantizing the most computationally intensive operations (such as matrix multiplications and convolutions) to use integer arithmetic. For each such operation, the input activations are temporarily converted to a quantized format for computation, and the results are subsequently converted back to float32. The model inputs, activations, and outputs remain in float32 format. No explicit range estimation is needed prior to conversion, as value ranges are determined dynamically at inference time or via heuristics. This approach offers further speed improvements with minimal impact on model accuracy.
- **Full-Integer Quantization**: In this method, all float32 values in the neural network, including inputs, activations, and outputs, are quantized. To accurately determine the value ranges of inputs, activations, and outputs, a representative (calibration) dataset is forward-passed through the model during conversion. The observed extrema are then used to set the quantization parameters for every tensor in the network. A key assumption is that the real inference data will have a similar value range. This trade-off maximally reduces model size and enables highly efficient inference using integer-only arithmetic.

2.8 Google Coral Dev Board Mini

Running machine or deep learning inference on embedded systems is referred to as edge inference. The Google Coral Dev Board Mini is a compact single-board computer designed for such edge AI applications. It features quad-core MediaTek 8167s System-on-a-Chip on the Armv8-A architecture, along with a dedicated Edge TPU — a hardware accelerator optimized for executing TensorFlow Lite (TF Lite) models using 8-bit integer operations.

The Edge TPU delivers up to 4 trillion operations per second of performance while consuming only around 2 watts of power, making it ideal for use in resource-constrained environments such as satellites, where energy efficiency and reliability are crucial.

The host system for the Edge TPU, the Dev Board, runs Mendel GNU/Linux 5 (Eagle). Interface to the accelerator is provided by the `libedgetpu`[3] runtime. To delegate tensor operations to the Edge TPU via `libedgetpu`, the model's operators must be compiled into the `edgetpu-custom-op` format. This compilation is performed by the Edge TPU Compiler[4] applied to the model.

[3] https://github.com/google-coral/libedgetpu
[4] https://coral.ai/docs/edgetpu/compiler/

For Edge TPU compatibility, the model must meet the following requirements[5]:

- Tensor parameters are quantized in 8-bit signed integer (int8) or uint8
- Tensor sizes are constant at compile-time
- Model parameters (such as bias tensors) are constant at compile-time
- Tensors are either 1-, 2-, or 3-dimensional. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- The model must use only operations supported by the Edge TPU. The operations used in this work are listed in Table 2.1 below. Table 2.1 represents a subset of all operations[6] supported by the Edge TPU.

Operation name	Runtime version	Known limitations
Add	All	—
Concatenation	All	No fused activation function. If any input is a compile-time constant tensor, there must be only 2 inputs, and this constant tensor must be all zeros (effectively, a zero-padding op).
Conv2d	All	Must use the same dilation in x and y dimensions.
MaxPool2d	All	No fused activation function.
Mul	All	—
ReLU	All	—
TransposeConv	$\geq$13	—

Table 2.1: Edge TPU supported operations used in this work.

These requirements directly affect model architecture, training strategy, and tooling. For instance, certain operations unsupported by Edge TPU must be avoided, PTQ and QAT must be considered early in the development process.

The following image summarizes the model conversion and deployment workflow[7]:

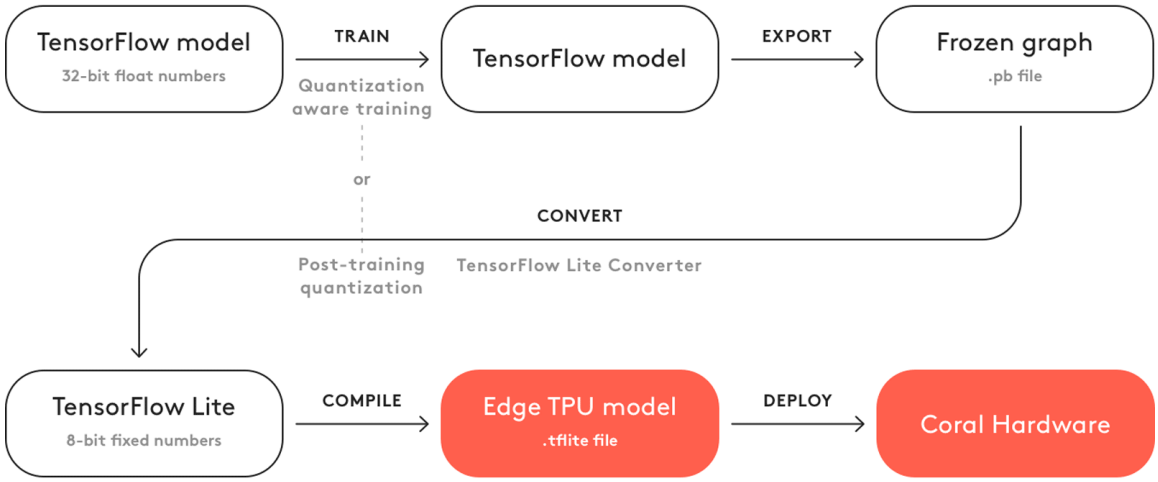


Figure 5: The basic workflow to create a model for the Edge TPU

[5] https://coral.ai/docs/edgetpu/models-intro#model-requirements
[6] https://coral.ai/docs/edgetpu/models-intro#supported-operations
[7] https://coral.ai/docs/edgetpu/models-intro#compatibility-overview

2.9 Methodology

Data

To efficiently utilize TF's dataset loading and preprocessing capabilities, a versatile data handling method has to be implemented. A core idea behind this approach is to unify the preparation of training, validation and test subsets within a single configurable function. This function must manage all aspects of dataset construction and transformation according to user-defined parameters, thereby ensuring consistency and flexibility. The function should output `tf.data.Dataset`[8] objects that are ready for immediate use in training pipelines.

Ideally, the testing pipeline should also be incorporated within this flexible, unified function. The absence of pre-cropped test subset GT patches introduces the necessity for additional preparation steps.

Model

The model architecture is tailored for deployment on embedded systems, where QAT must be taken into account. Due to the lack of existing practical examples for deploying this specific model architecture on Edge TPU, the design process begins with a minimal configuration, using the smallest feasible number of layers and filters, and gradually increases in complexity based on deployment outcomes and evaluation feedback. An iterative, trial-and-error methodology is employed, allowing for adjustments in response to the resource constraints encountered on embedded hardware.

The objective is to implement multiple configurable functions, each responsible for constructing a distinct model architecture. These functions include all necessary helper routines and utilities required to assemble the desired architecture, allowing convenient switching between models during training.

Training

After implementing utilities for dataset construction and defining the model architecture, the next step is to develop the training pipeline. Initially, a simple and direct training pipeline should be created to validate basic model architectures, as a minimal setup both reduces potential error sources and enables rapid proof-of-concept. During early development, model conversion is performed separately after training.

However, the pipeline should be further refined, ideally as a single executable script, that encompasses data loading, training, configuration management, and model conversion, streamlining the path to deployment. The goal is to enable a convenient, one-click training and conversion workflow.

[8] https://www.tensorflow.org/api_docs/python/tf/data/Dataset

To ensure reproducibility and facilitate backups, it is important to store all training configurations, logs, and intermediate model weights for each run.

Finally, computational resources must be considered. To avoid prohibitively long training times, the use of more powerful machines or cloud-based resources should be considered.

Conversion

After training, the model must be converted and compiled for Edge TPU inference. A modular approach is important here as well, simplifying debugging and ensuring correct conversion and compilation during early development. Later on, the integration of the conversion process directly into the training pipeline is ideal for convenience and efficiency.

Automated use of the Edge TPU Compiler is required so that the entire training and conversion pipeline outputs a model ready for deployment on the Dev Board.

Deployment

A tool must be developed on the Dev Board. This program should preload the testing data, load the trained model, and pass tensors for inference. In early development, for debugging purposes, the testing data can be represented by a single patch. The goal, however, is scene-wise prediction. Ideally, all necessary steps, from preparing the input data as a series of patches to stitching the predictions, are implemented in a unified pipeline. The use of C++ inference should be considered for greater control and potential performance improvements. Inference time must be documented as an important performance metric.

Evaluation

After successful implementation and verification of the training and deployment pipelines, multiple versions of CNNs can be designed and trained using the developed tools. Training metrics must be documented. Following successful deployment on the Dev Board, the models will be evaluated on the complete testing dataset. To minimize the time needed for full-dataset inference, the evaluation is planned to be implemented and executed on a personal workstation. Models are then evaluated using the metrics outlined in section 2.3. The results and key insights will be documented for future applications.

3 Implementation

A detailed description of the software development process used to achieve the thesis objective is provided in this chapter. Along the way, three major implementation milestones were reached. Despite several hurdles, success was achieved in (i) training the model on powerful cloud machines, (ii) mapping all its components, fully quantized, to the Edge TPU, and (iii) performing C++ inference on the Dev Board. The technical obstacles and their resolution are presented in three "Technical Challenges" chapters inside Training, Conversion, and Deployment.

Software developed in the scope of this work is made available for free and unrestricted use on the author's GitHub profile:

- `https://github.com/An3Zh/BachelorThesis`

Details on relevant file locations, with references to this work's chapters, as well as further information on the repository structure, are provided in `README.md`.

3.1 Data

3.1.1 Training & Validation

In order to construct the training and validation pipeline, the function `BuildDS` was implemented, along with several supporting helper functions, all located in the `load.py` file. The function was later extended to optionally include the test dataset, which will be discussed in subsection 3.1.2.

Efficient data handling and memory management are achieved through the use of built-in TF utilities. In particular, the `tf.Data.TextLineDataset`[1] function is employed to sequentially read the `.csv` files, which contain the patch filenames as outlined in section 2.4. This provides the foundation for the dataset pipeline.

The complete training set originally contains 8400 patches. However, as explained earlier in section 2.4, some patches are entirely zero-valued. Only 5155 of them contain valid data and are thus retained for subsequent use. The dataset is shuffled and split into training and validation subsets, with the ratio configurable as needed.

Using the `.map` method, each text line is first expanded into five full paths to the corresponding RGB, NIR and GT mask patches. Each filepath is then replaced by its image

[1] `https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset`

content, loaded as `tf.Tensor`[2]. This transformation is handled by the helper function `loadDS`, which itself calls utility function `loadTIF`. At this stage, each dataset element is a tuple of TF tensors representing the four-channel input image and its corresponding GT mask.

The `loadDS` and `loadTIF` functions perform the following operations:

- RGB and NIR patches are cast to float32 and normalized to the range [0,1].
- GT masks, originally in uint8, are binarized to values of 0 and 1, and also cast to float32.

An additional feature of the `loadDS` function allows for optional resizing of the input images if a target size is provided. The image loading and transformation pipeline is designed to avoid information loss until the controlled resizing step, where a reduction in resolution is intentional. In the case of resizing, GT masks are resized using nearest-neighbor interpolation. This method copies the value of the closest original pixel for each target pixel, thus preserving hard edges in the cloud segmentation mask and avoiding the introduction of intermediate values. Conversely, the RGB and NIR inputs are resized with bilinear interpolation, which computes each new pixel value as a weighted average of the nearest 2×2 neighborhood. This results in smooth pixel transitions while maintaining important image details. Nearest-neighbor interpolation is among the earliest digital image resizing techniques, dating back to the origins of digital image processing in the 1960s, whereas bilinear interpolation gained prominence in early computer graphics literature and is now standard in deep learning pipelines [GW17; Ruk23].

As a final preparation step, the dataset is: shuffled, batched, set to repeat indefinitely, and prefetched to optimize data retrieval during training. Because the datasets are repeated indefinitely, it is essential to define the number of steps required to complete one full pass through the training and validation subsets. These step counts are used to ensure the correct number of iterations per epoch during training, and they are computed as follows: `trainSteps(valSteps) = trainSubsetSize(valSubsetSize) / batchSize`.

3.1.2 Testing

An additional capability of the `buildDS` is the construction of the test dataset. As outlined in section 2.4, only cropped RGB and NIR test patches, together with complete scene GT masks, are available for testing. This arrangement necessitates two specific preparation steps prior to building the test pipeline:

[2] https://www.tensorflow.org/api_docs/python/tf/Tensor

1. **Metadata collection:** To uniquely identify each scene, the `sceneID` is introduced, derived from the Landsat 8 metadata (path/row), and is unique within this test dataset. The total number of patches, as well as the number of rows and columns for each scene, is collected for every `sceneID`. The patch filenames, as outlined in section 2.4, are used for this process. Additional `.csv` files were manually created — each containing the ordered patch filenames corresponding to every full scene. Filenames in these `.csv` files are organized in the order resulting from cropping (left to right, top to bottom). Furthermore, the `fullTestDS.csv` file was generated, containing ordered patch filenames for all 20 full scenes. These `.csv` files, not provided in the original dataset, were developed in the scope of this thesis to support the test pipeline and are stored in the `additionalCSVs` folder within the testing subset. Based on this manually collected metadata, the `getSceneGridSizes` function was implemented in `load.py` file, returning a dictionary that maps each of the 20 `sceneIDs` to the respective number of rows and columns in each scene.

2. **Patch stitching:** The `stitchPatches` function was implemented in `load.py`. This function uses the `.csv` files and the `getSceneGridSizes` function to reconstruct entire scenes from the model's output patches for evaluation. The function can operate in two modes: it can either stitch together all scenes at once, saving all 20 full scenes, or stitch only a single specified scene. The single-scene mode is intended for debugging and verification, avoiding time-consuming processing of the full test subset.

Following these preparatory steps, the `buildDS` function was extended with additional functionality to construct the test subset. Using the optional boolean parameter `includeTestDS`, which indicates whether to include the test dataset, and the parameter `singleSceneID`, which allows the selection of a specific scene, the function now supports multiple modes for testing:

- Utilizing all 9201 patches from the 20 test scenes, or
- Selecting patches from a single scene, either by specifying its `singleSceneID` or allowing the function to randomly select one.

The dataset is then constructed using the same TF utilities employed for the training and validation subsets. It is important to emphasize that the test set is not shuffled, as preserving the initial patch order is essential for the subsequent stitching process.

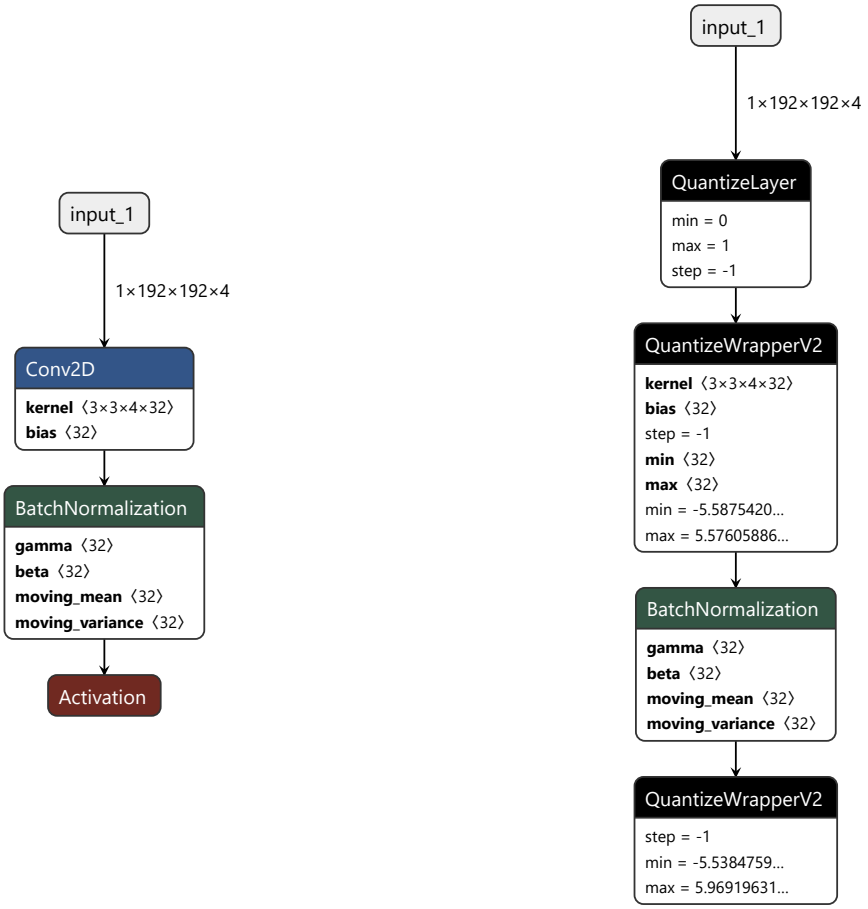
Consequently, the `buildDS` function returns the training and validation subsets, along with their respective step counts based on the `batchSize`. Optionally, it also returns test subset, which may contain either the entire test dataset or a single scene. After inference, the model's output patches can be passed to the `stitchPatches` function, which reconstructs and saves the final scene images for further evaluation.

3.2 Model

The core building blocks and utility functions for architecting the model are organized in `model.py`. CNN layers are constructed using TF's built-in Application Programming Interfaces (APIs), allowing for a modular and reusable design.

In early development, the `simple` model architecture was implemented. It served as a proof of concept and was used for initial debugging of the training and conversion pipelines, as well as to obtain the first inference results on the Dev Board. The model was not annotated for QAT.

Below, in Figure 6 are shown the examples of layer diagrams for two models: the left is not annotated for QAT, the right is annotated. These visualizations were produced with Netron[3]. Following the concept outlined in section 2.6, the `QuantizeWrapperV2` and `QuantizeLayer` TensorFlow Operations (TF ops) are inserted into the architecture, wrapping the base layers with quantize-dequantize operations.



(a) model part not annotated for QAT (b) model part annotated for QAT

Figure 6: Model before and after QAT annotation

It is important to emphasize that Batch Normalization (BN) is applied to the output of the `Conv2D` TF op and precedes the corresponding `Activation` function [IS15].

[3] https://netron.app/

Subsequently, more complex `U-Net` architectures were implemented, with greater depth and skip connections. Early versions of these models were not QAT annotated, instead, PTQ was applied at conversion time.

Where standard TF loss functions are insufficient, custom losses such as `softJaccardLoss` and `diceLoss` are implemented in this module. Custom metrics, particularly those used for segmentation evaluation and referenced in section 2.3, are also provided within the `model.py` file.

3.3 Training

The complete model training and conversion pipeline is implemented in the `main.py` file.

The process begins with configuration settings, which include:

- Batch size for training
- Image size for both training and inference
- Number of training epochs
- Selected model architecture
- Ratio between validation and training subsets
- Number of batches used for the calibration dataset

Additional settings can be incorporated as needed.

The pipeline starts by loading the dataset using functions from `load.py`. The selected model architecture is then compiled utilizing utilities from `model.py`.

Each training run is stored in a dedicated folder, named with a timestamp corresponding to the start of the run. This folder contains all configuration files, training-related artifacts, and final results, including the compiled Edge TPU model.

Prior to training, the following configurations and callbacks are set up:

- **Model Checkpoints**: Model checkpoints are saved during training whenever the validation loss improves, preserving the best-performing model.
- **Early Stopping**: If the validation loss stops improving, training continues for a predefined number of additional epochs before termination.
- **Learning Rate Reduction**: The learning rate is reduced if the validation loss does not improve for a set number of epochs, helping to fine-tune convergence.

Once all configurations are saved, model training is initiated with the specified callbacks active.

Upon completion of training, the final model weights are saved and immediately used for model conversion.

Technical Challenges

Training of initial proof-of-concept models was performed on a personal workstation equipped with an Intel Core i5-13400F CPU and an NVIDIA GeForce RTX 4060 Graphics Processing Unit (GPU). To enable GPU acceleration for the TF training process, the NVIDIA Compute Unified Device Architecture (CUDA) API and the corresponding CUDA Deep Neural Network Library (cuDNN) library must be installed and configured. The following versions were used: CUDA v11.2 and cuDNN v8.1.1. These versions are not the latest available from NVIDIA at the time of this work, but were selected for compatibility with certain TF ops and the `libedgetpu` runtime library. Further details on these compatibility considerations are discussed in section 3.4 and section 3.5.

Initial models containing approximately 300,000 parameters were trained without QAT for tens of epochs. On the described setup and using the full training dataset, the time required for a single epoch was less than 15 seconds, which was deemed acceptable.

With the introduction of more complex models containing 2,000,000 and 30,000,000 (experimental, proven to be inefficient at edge inference) parameters, and when employing QAT, training times increased significantly. For these larger models, one epoch on the full training dataset required approximately 3 minutes and 12 minutes, respectively. Since at least 100 epochs were necessary to achieve adequate model performance, more powerful hardware resources became essential.

Thunder Compute[4], an American startup with GPU cloud resources for machine learning and data science, provided solutions for large-scale experiments. By using an instance with an NVIDIA A100XL GPU, training times for the 2,000,000 parameter network were reduced to approximately 20 seconds per epoch, and for the 30,000,000 parameter network to 40 seconds per epoch. A 10× increase in GPU memory, to 80GB on the A100XL compared to 8GB on the RTX 4060, also enabled larger batch sizes, resulting in more effective training and improved model robustness.

However, significant effort was required to configure the cloud instance for compatibility. The more expensive production mode[5] had to be used, as the prototyping mode[6] did not allow downgrading CUDA and cuDNN versions. The CUDA and cuDNN libraries needed to be downgraded from their most recent platform versions to those compatible with an older TF version. This was challenging, as it required purging the latest CUDA and cuDNN files without removing the GPU drivers required by the older versions. Ultimately, CUDA v11.2 and cuDNN v8.1.0 were installed on the instance for subsequent training. In addition, the Thunder Compute development team was contacted with a suggestion to allow selection of CUDA and cuDNN versions at instance creation. As a result, an improvement is planned that will allow passing the desired CUDA and cuDNN version as an environment variable during instance startup.

[4] https://www.thundercompute.com/
[5] https://www.thundercompute.com/docs/production-mode
[6] https://www.thundercompute.com/docs/prototyping-mode

3.4 Conversion

The file `convert.py` contains all necessary functions and utilities for model conversion and Edge TPU compilation. These functions are called from `main.py` immediately after training is complete.

3.4.1 Weight Transfer

As outlined in section 2.8, the Edge TPU supports input tensors with at most three dimensions. This necessitates the use of batch size one, since the input tensor shape is already $image\ height \times image\ width \times number\ of\ channels$. During training, larger batch sizes are typically used for efficiency. Therefore, after training, the function `asBatchOne` is used to transfer the trained weights to a model with identical architecture, but with batch size set to one for the input, all intermediate computations, and the output.

3.4.2 Post-Training Quantization

The `tf.lite.TFLiteConverter`[7] utility is used to perform PTQ and convert the model into the `.tflite` format. A representative dataset, consisting of training RGB and NIR patches, is generated using the `representativeDatasetGen` function. The number of calibration batches used can be adjusted via `numCalBatches`, as defined in section 3.3. The greater the amount of calibration data, the more likely the observed value range will closely match the inference data, improving quantization accuracy.

As a next step, all supported TF ops as well as model inputs and outputs are quantized. While various data types can be used for quantization in general case, this work utilizes int8, as required for Edge TPU compatibility in section 2.8.

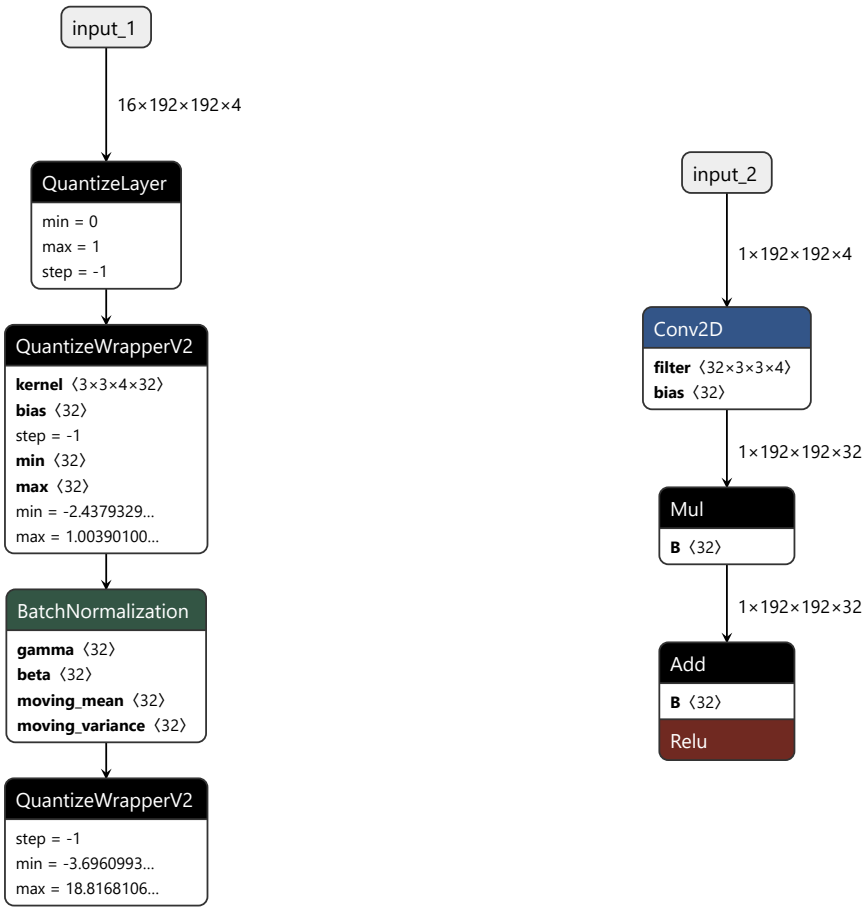
The following subchapter provides a deeper investigation into quantization of model parameters by analyzing a QAT-annotated float32 model and its already quantized version. The Netron tool is used here as well for visualizing the model's parts. This chapter is aimed at readers who wish to deepen their understanding of parameter quantization and gain insight into the general techniques behind full-integer operation execution on edge devices such as the Edge TPU.

It should be noted that both QAT annotation and PTQ are already wrapped into convenient, ready-to-use TF Lite commands. Thus, this section is not intended to teach the implementation of quantized models from scratch. Instead, it aims to offer a deeper understanding that can be beneficial for debugging, designing custom (potentially more powerful) models, or simply appreciating the engineering trade-offs between speed, memory, and accuracy on constrained devices.

[7] https://www.tensorflow.org/api_docs/python/tf/lite/TFLiteConverter

3.4.3 Practical Example of Model Quantization

Below, a QAT annotated part of the model is shown in the left panel of Figure 7. On the right, the same part is shown after PTQ, yielding a fully int8 quantized model.



(a) model part annotated for QAT (b) model part after full int8 quantization

Figure 7: Model before and after quantization

Netron also provides detailed information about graph nodes (model layers) and their connections (tensor passes). For example, layer weights and biases can be directly inspected, and values can be examined. In the tables below, quantization-relevant information is displayed for further analysis.

For both the QAT annotated model and the resulting quantized model, the relevant information for four layers is summarized. All floating-point numbers are shown in full float32 precision. Representative calculations are performed in full precision and displayed with extended decimals. The goal is to illustrate, using a practical example, the error effects of quantization and the limits of float32 precision.

QAT Annotated Model (left in Figure 7)

Layer Type: QuantizeLayer **Layer Name: Quantize Input**

Input value range:
$\min = 0$ $\max = 1$

Layer Type: QuantizeWrapperV2 **Layer Name: Conv2D No. 1**

Kernel shape:
$[3, 3, 4, 32]$

Kernel No. 1 value range:
$\min = -0.544256329536438$ $\max = 0.544256329536438$

First element of Kernel No. 1 value:
0.032761260867118835

Bias shape:
$[32]$

Bias No. 1 value:
0.1983194351196289

Activation value range:
$\min = -2.4379329681396484$ $\max = 1.0039010047912598$

Layer Type: Batch Normalization **Layer Name: BN No. 1**

Shape of γ, β, μ_{moving}, σ^2_{moving} parameters:
$[32]$

Parameter γ_1 value:
1.110548973083496

Parameter β_1 value:
-0.11593257635831833

Parameter $\mu_{\text{moving},1}$ value:
0.17263321578502655

Parameter $\sigma^2_{\text{moving},1}$ value:
0.00221889466047287

Layer Type: QuantizeWrapperV2 **Layer Name: Activation No. 1**

Output value range:
$\min = -3.696099329375535e{-}11$ $\max = 18.816810607910156$

Model After Quantization (right in Figure 7)

Layer Type: Conv2D **Layer Name: Conv2D No. 1**

Input affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.003921568859368563 \times (q_{\mathrm{x}} + 128)$$

Kernel No. 1 symmetric dequantization formula:

$$x_{\mathrm{q}} \approx 0.004285482689738274 \times q_{\mathrm{x}}$$

First element of Kernel No. 1 quantized value:

8

Bias No. 1 symmetric dequantization formula:

$$x_{\mathrm{q}} \approx 0.000016805815903353505 \times q_{\mathrm{x}}$$

Bias No. 1 quantized value:

11801

Layer Type: Mul **Layer Name: BN No. 1 Scale**

Input affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.013497387990355492 \times (q_{\mathrm{x}} - 53)$$

Fused BN multiplier M_c affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.10698594152927399 \times (q_{\mathrm{x}} + 128)$$

Multiplier M_1 quantized value:

55

Layer Type: Add **Layer Name: BN No. 1 Shift**

Input affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.17354172468185425 \times (q_{\mathrm{x}} - 26)$$

Fused BN additive term A_c affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.0339626707136631 \times (q_{\mathrm{x}} + 10)$$

Additive term A_1 quantized value:

-113

Layer Type: ReLU **Layer Name: Activation No. 1**

Output affine dequantization formula:

$$x_{\mathrm{q}} \approx 0.07379141449928284 \times (q_{\mathrm{x}} + 128)$$

Before presenting the calculations, the Quantization Specification Table[8] is reviewed. An excerpt relevant to the TF op `Conv2D` is shown below:

```
CONV_2D
  Input 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8
    range      : [-127, 127]
    granularity: per-axis (dim = 0)
    restriction: zero_point = 0
  Input 2 (Bias):
    data_type  : int32
    range      : [int32_min, int32_max]
    granularity: per-axis
    restriction: (scale, zero_point) = (input0_scale*input1_scale[...], 0)
  Output 0:
    data_type  : int8
    range      : [-128, 127]
    granularity: per-tensor
```

Using this information, quantization calculations can be retraced for a better understanding of model transformation. The `input_1` values, or the model's input tensors, are observed first from the `QuantizeLayer`. As expected, these range from 0 to 1 due to normalized input patches.

The input scale and zero-point can be calculated using equations 2.1 and 2.2 from section 2.5:

$$\text{Input scale} = \frac{1-0}{127-(-128)} = 0.003921568627450980$$

$$\text{Input zero-point} = \text{round}\left(-128 - \frac{0}{\text{Input scale}}\right) = -128$$

The calculated values align with the "Input affine dequantization formula" for the `Conv2D` layer in the quantized model. The precision difference at the 10th decimal place is expected due to float32 limits. This confirms the use of affine (asymmetric) quantization for `Input 0`, consistent with the table's range [-128,127] and the absence of a zero-point restriction.

Moving to the `Conv2D` layer weights, the table specifies per-axis granularity with symmetric quantization (zero-point=0). Inspection of the 32 kernel value ranges (only Kernel No. 1 shown here) confirms symmetry around zero. This suggests that TF enforces clipping of weight extremes to fit symmetric ranges.

[8] https://ai.google.dev/edge/litert/models/quantization_spec

Netron reveals that Kernel No. 1 has the largest max value among the 32 kernels (not represented here), yet it still does not match the actual tensor extrema ($min = -0.4122757017612457, max = 0.5442604422569275$), indicating that TF may apply additional optimization, likely outlier-aware clipping or mean adjustment, since real values are not perfectly zero-mean ($mean = -0.005126346834471305$). A detailed investigation of this effect lies outside the scope of this work.

Using the clipped extrema, the symmetric quantization scale is:

$$\begin{aligned}\text{Kernel No. 1 scale} &= \frac{0.544256329536438 - (-0.544256329536438)}{127 - (-127)} \\ &= 0.004285482909735732\end{aligned}$$

This matches the Kernel No. 1 symmetric dequantization formula from the quantized model (within float32 precision limits). Applying equation 2.4, the quantized value can be approximately dequantized back to the first element of Kernel No. 1:

$$\text{First element of Kernel No. 1} \approx 0.004285482689738274 \cdot 8 = 0.034283861517906192$$

The resulting precision loss is small and can be quantified by absolute, relative, and Least Significant Bit (LSB) errors:

- $e_{\text{abs}} = 0.034283861517906192 - 0.032761260867118835 = 0.001522600650787357$
- $e_{\text{rel}} = \frac{0.001522600650787357}{0.032761260867118835} \times 100\% \approx 4.647\%$
- $e_{\text{lsb}} = \frac{0.001522600650787357}{0.004285482689738274} \approx 0.355,\ \Delta = 0.004285482689738274$

Here, the LSB error < 0.5 is a good indicator: it suggests that scale boundaries are not exceeded and quantization was applied without mismatches, clipping errors, or encoding/decoding inconsistencies.

Looking at the bias restrictions in the Quantization Specification Table, an interesting point emerges: the bias has no dedicated quantization scale derived from its own extrema. Hence, no min/max values are logged by `QuantizeWrapperV2` for the `Conv2D` layer. Instead, its scale is computed as the product of the layer's input scale and the per-channel weight scale, yielding one bias scale per output channel (matching the weight scales per axis). This engineering choice accelerates inference at the cost of a slight precision loss.

Multiplying input values by weights (each with its own scale) produces a new effective scale. To add the bias to this product, the bias must be quantized using the same scale (with zero-point=0). One could quantize the bias with its own scale and then rescale the product accordingly, but this introduces extra operations that would significantly slow inference. Instead, fast fixed-point arithmetic is employed, using a precomputed quantized multiplier and a bit-shift, to reconcile the mixed scales that arise when multiplying quantized values.

$$\begin{aligned}\text{Bias No. 1 scale} &= 0.003921568859368563 \cdot 0.004285482689738274 \\ &= 0.000016805815460634406445 \\ \text{Bias No. 1 value} &\approx 0.00001680581590335350 5 \cdot 11801 \\ &= 0.198325433475474713\end{aligned}$$

Analogous calculations can be performed for the Bias No. 1 to determine its scale and recover its approximate float32 value.

As can be seen in the Netron analysis, the BN TF op is decomposed into two primitive TF ops after quantization: `Mul` and `Add`. This follows from the affine form of batch normalization (and its folding during quantization), which can be expressed as:

$$y = \gamma \cdot \frac{x - \mu_{\text{moving}}}{\sqrt{\sigma^2_{\text{moving}} + \epsilon}} + \beta \tag{3.1}$$

$$M_c = \frac{\gamma_c}{\sqrt{\sigma^2_{\text{moving},c} + \epsilon}}, \quad A_c = \beta_c - \frac{\gamma_c \cdot \mu_{\text{moving},c}}{\sqrt{\sigma^2_{\text{moving},c} + \epsilon}} = \beta_c - M_c \cdot \mu_{\text{moving},c} \tag{3.2}$$

- x: input value to be normalized.
- y: output value after batch normalization.
- $\mu_{\textbf{moving},c}$: moving mean for channel c, computed during training as an exponential moving average of per-batch means; used in inference because batch statistics from a single sample are often unreliable.
- $\sigma^2_{\textbf{moving},c}$: moving variance for channel c, computed during training as an exponential moving average of per-batch variances; also used in inference for the same reason as the moving mean.
- ϵ: small constant to avoid division by zero (default[9] TF value is set to 0.001).
- γ_c: learnable scale parameter for channel c.
- β_c: learnable shift parameter for channel c.
- M_c: per-channel multiplier used in quantized inference.
- A_c: per-channel additive term used in quantized inference.

It should be noted that the affine dequantization of the `Mul` input is computed from the extrema of the outputs of `Conv2D` No. 1, analogous to the previous calculations of scales and zero-points.

In quantized form, μ_{moving}, σ^2_{moving}, γ, and β are fused into per-channel multiplier M_c and additive term A_c. Reverse-engineering from the quantized values recovers the original float32 parameters with minimal error:

[9] https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization

$$M_1 = 0.10698594152927399 \cdot (55 + 128) \approx 19.5784273$$
$$\sqrt{\sigma^2_{\text{moving},1} + \epsilon} = \sqrt{0.00221889466047287 + 0.001} \approx 0.0567353$$
$$\hat{\gamma}_1 = M_1 \cdot \sqrt{\sigma^2_{\text{moving},1} + \epsilon} \approx 19.5784273 \times 0.0567353 \approx 1.1107880$$
$$\hat{\gamma}_1 = 1.1107880 \approx \gamma_1 = 1.110548973083496$$

$$A_1 = 0.0339626707136631 \cdot (-113 + 10) \approx -3.4981550$$
$$\beta_1 = A_1 + M_1 \cdot \mu_{\text{moving},1} \approx -3.4981550 + 19.5784273 \times 0.1726332 \approx -0.1182682$$
$$\beta_1 = -0.1182682 \approx \beta_{1,\text{float}} = -0.11593257635831833$$

Finally, the "Output affine dequantization formula" is derived from the extrema of the (ReLU) Activation No. 1 layer:

$$\begin{aligned} \text{Output scale} &= \frac{18.816810607910156 - (-3.696099329375535\text{e}{-11})}{127 - (-128)} \\ &= 0.073791414148812224 \\ \text{Output zero-point} &= \text{round}\left(-128 - \frac{-3.696099329375535\text{e}{-11}}{\text{Output scale}}\right) \\ &= -128 \end{aligned}$$

All subsequent quantization steps for TF ops in the model follow the same general procedure, adhering to predefined constraints and reusing common underlying methods.

From this deeper investigation, the following key quantization principles emerge:

- After PTQ, the `QuantizeInput` layer defines the input scale and input zero-point based on observed extrema. Same calculation principles apply further.
- Weights are typically quantized per-axis (per-channel) with symmetric ranges.
- Bias scales are derived from input and weight scales, enabling fixed-point computation without extra rescaling.
- Certain operations, such as BN, are decomposed into simpler arithmetic operations for inference efficiency.
- The precision loss is small, and LSB error below 0.5 indicates well-matched scales.

After quantized model is saved, the Edge TPU Compiler is invoked.

3.4.4 Edge TPU Compiling

Since training may be performed either on a local Windows machine or on a Linux-based cloud GPU instance from Thunder Compute, cross-platform automation of the pipeline is ensured by using helper functions and platform detection before invoking the Edge TPU compiler. Because the Edge TPU compiler can only run on Debian-based Linux systems, the Windows Subsystem for Linux (WSL) is used when working on a Windows machine.

The Edge TPU Compiler parses the `.tflite` model, verifies that the restrictions are satisfied and that it is compatible with the Edge TPU runtime, and then attempts to fuse all supported TF ops into a single `edgetpu-custom-op`. An example of the compiler's console output during compilation is shown below.

```
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 898 ms.

Input model: dev/results/run_20250719_170647/quant.tflite
Input size: 1.90MiB
Output model: dev/results/run_20250719_170647/quant_edgetpu.tflite
Output size: 2.07MiB
On-chip memory used for caching model parameters: 1.90MiB
On-chip memory remaining for caching model parameters: 3.82MiB
Off-chip memory used for streaming uncached model parameters: 46.00KiB
Number of Edge TPU subgraphs: 1
Total number of operations: 53
Operation log: dev/results/run_20250719_170647/quant_edgetpu.log
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

Listing 3.1: Edge TPU compiler console output. Successful compilation

The Edge TPU has ~ 8 MiB of static Random-Access Memory (RAM), allowing model parameters to be cached. It is further specified[10] that multiple models can be co-compiled and stored in the Edge TPU's on-chip memory, enabling fast switching and inference when different models are required on a single Edge TPU. Conversely, a model can be partitioned for pipelining across multiple Edge TPUs, allowing parallel inference or acceleration of larger models. In the scope of this thesis, however, only a single Edge TPU is used. Therefore, the on-chip static RAM should ideally not be exceeded to preserve inference speed. Falling back to external memory can degrade performance due to parameter reloads.

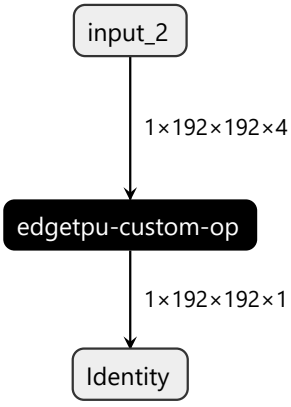
From the Listing 3.1 log example, useful information about memory usage can be inferred. The model executable occupies approximately $8 - 1.90 - 3.82 \approx 2.28$ MiB of the Edge TPU on-chip cache. Its parameters occupy 1.90 MiB, leaving 3.82 MiB free, thus, the entire model is stored in the Edge TPU's static RAM. In chapter 4, memory

[10] https://coral.ai/docs/edgetpu/compiler/#parameter-data-caching

usage is also considered when comparing different models. For this particular model, however, 46.00 KiB of external memory was still utilized despite the remaining free on-chip memory. The reason is not documented officially and likely requires deeper investigation of the compilation process, which lies outside the scope of this thesis.

After successful compilation, a `.log` file is saved alongside the Edge TPU model. This log reports, for each TF op, whether mapping to the Edge TPU succeeded (and, if not, why), as illustrated by the `.log` excerpt shown right in Figure 8.

One way to confirm that the entire model compiles for the Edge TPU is to visualize the compiled model in Netron. An example demonstrating that every TF op was successfully mapped to the Edge TPU is shown left in Figure 8. In contrast to the pre-compilation graphs (for example, see Figure 7), no additional structural information is visible: all supported TF ops have been fused into a single `edgetpu-custom-op` that executes exclusively on the Edge TPU. As shown, the input and output tensor shapes are readable, and the associated quantization scales and zero points can be inspected, as in subsection 3.4.3; no further details are exposed.




```
Edge TPU Compiler version 16.0.384591198
Input: dev/results/run_20250719_170647/quant.tflite
Output: dev/results/run_20250719_170647/quant_edgetpu.tflite

Operator                       Count      Status

CONCATENATION                  3          Mapped to Edge TPU
LOGISTIC                       1          Mapped to Edge TPU
ADD                            14         Mapped to Edge TPU
MAX_POOL_2D                    3          Mapped to Edge TPU
MUL                            14         Mapped to Edge TPU
CONV_2D                        15         Mapped to Edge TPU
TRANSPOSE_CONV                 3          Mapped to Edge TPU
```

Figure 8: Edge TPU-mapped model (left) and compiler log excerpt (right).

The final artifact is a `.tflite` model file ready for Edge TPU inference.

Technical Challenges

The Dev Board is running on an end of life version of Mendel Linux, which is the only one latest officially supported software version for Dev Board. With the OS itself comes already pre-installed runtime library `libedgetpu` of version v16.0, exposing a low-level C++ API for direct interaction with Edge TPU. Consequently, this library version is the latest officially available in Coral repository for this Mendel Linux release. Building the latest version from source entails a more complex Bazel-based process that risks breaking dependencies and incurs a nontrivial debugging burden; this lies outside the scope of this thesis.

The newest as of today Edge TPU Compiler version v16.0 is used, requiring minimum[11] runtime version v14.0, which is fulfilled by `libedgetpu` v16.0.

[11] https://coral.ai/docs/edgetpu/compiler/#compiler-and-runtime-versions

At early development stages, it was unclear which TF version was supported by the provided runtime library and compiler, as this information was not available in the official documentation. Consequently, the newest TF version, v2.19.0 (at the time this work was conducted), was used heuristically for initial attempts at dataset pipeline construction, model architecture, training, and conversion. No QAT annotation was performed, relying solely on PTQ prior to compiling the model for trial inference on the Edge TPU. At compile time, the following error from the Edge TPU Compiler was encountered:

```
Edge TPU Compiler version 16.0.384591198
ERROR: Didn't find op for builtin opcode 'TRANSPOSE_CONV' version '4'.
An older version of this builtin might be supported.
Are you using an old TFLite binary with a newer model?
```

To avoid this error, the first approach was to omit the `TRANSPOSE_CONV` TF op while keeping the newest TF, replacing deconvolution with the simpler combination `UpSampling2D + Conv2D`. This may reduce accuracy but could still yield acceptable results for edge-device inference. Alternatively, the TF version was downgraded to v2.5.0, corresponding to the TF Lite library on the Dev Board, to maximize version compatibility with the target hardware. After the downgrade, the first successful compilation and inference were achieved. Results are presented in section 3.6.

With the introduction of QAT annotated model layers, the TensorFlow Model Optimization (TFMOT) package must be installed, with a version compatible with TF v2.5.0. This required installing TFMOT v0.6.0. During the TF Lite conversion step (while performing PTQ), the following error kept occurring:

```
error: 'tfl.max_pool_2d' op quantization parameters violate the same
scale constraint: !quant.uniform<i8:f32, 840332.86281612806:-128>
vs. !quant.uniform<i8:f32, 0.027852464160498452:-127>
```

This error refers to the TF Lite restriction (see subsection 3.4.3) that the inputs and outputs of `MAX_POOL_2D` must share the same scale and `zero_point`, as specified in the Quantization Specification Table[12]. As part of debugging, it was verified that the input and output minimum and maximum ranges were essentially identical. Only the mean distributions differed, which is normal for batches with different compositions:

```
---- INPUT TO MAXPOOL ----
Shape: (1, 384, 384, 16)
min:   0.0
max:   3.411252021789551
mean:  1.003772497177124
```

```
---- OUTPUT OF MAXPOOL ----
Shape: (1, 192, 192, 16)
min:   0.0
max:   3.411252021789551
mean:  1.0165772438049316
```

Furthermore, a sanity check of the value ranges across different input batches was performed, confirming expected values from 0 to $\sim$ 5, without extreme outliers, thus, there is no justification for very large scale 840 332.8 as indicated by the error message. Activation ranges of the preceding `Conv2D` layer were inspected in Netron and showed no anomalies. These findings suggest an internal issue with this specific TFMOT ver-

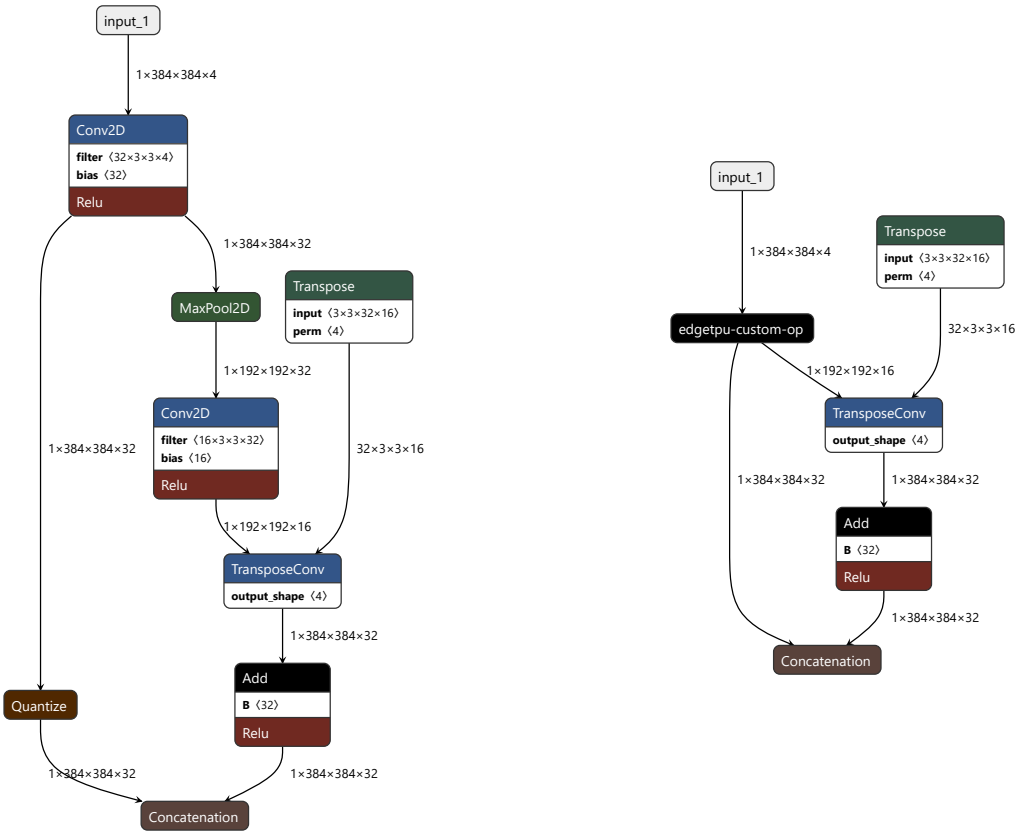
[12] https://ai.google.dev/edge/litert/models/quantization_spec

sion, as the observed input/output statistics contradict the scale values implied by the error message.

During debugging, it was further identified that a similar issue[13] had been reported for a later TFMOT version. Combined with the observations above, this suggested trying a newer TFMOT release in which the bug might be fixed. This raised concerns about compatibility with the older TF v2.5.0 required for TF Lite (TF op v3) compatibility mentioned earlier. It was decided to proceed, however, given the absence of documentation explicitly stating incompatibility of newer TFMOT with older TF. Multiple version combinations of these modules were evaluated.

Selecting a combination of newer TFMOT and TF resolved the issue. This, in turn, supports the conclusion that the error originated in the TF/TFMOT implementation and was corrected in later releases.

Successful conversion to the `.tflite` format was subsequently achieved. During conversion, the newer TF/TFMOT toolchain automatically inserted a `Transpose` TF op into the model. This operator was not used during model design, indicating an internal optimization related to QAT handling. A plausible explanation is the need to convert between common tensor layouts — `[batch, H, W, C]` (`NHWC`) and `[batch, C, H, W]` (`NCHW`) — at some point in the pipeline. Limitations in one component's ability to process a given layout can trigger such a transformation, implemented as a `Transpose` placed before `TransposeConv`. Below in Figure 9, the same model architecture is shown before (left) and after (right) the Edge TPU compilation process.



(a) Before compilation

(b) After compilation

Figure 9: Inserted `Transpose` TF op prevents fusion into a single `edgetpu-custom-op`.

[13] https://github.com/tensorflow/model-optimization/issues/1053

Both the crucial console output and the `.log` file are provided in Listing 3.2, indicating the failed attempt to fuse multiple TF ops into the intended `edgetpu-custom-op`.

```
Model successfully compiled but not all operations are supported
by the Edge TPU. A percentage of the model will instead run on the CPU,
which is slower. If possible, consider updating your model to use only
operations supported by the Edge TPU.
For details, visit g.co/coral/model-reqs.
Number of operations that will run on Edge TPU: 4
Number of operations that will run on CPU: 7
See the operation log file for individual operation details.

Logfile:
Edge TPU Compiler version 16.0.384591198
Input: quant.tflite
Output: quant_edgetpu.tflite

Operator                 Count       Status

CONV_2D                  2           More than one subgraph is not supported
CONV_2D                  2           Mapped to Edge TPU
TRANSPOSE_CONV           1           Filter, bias, or other param is not
                                     constant at compile-time
LOGISTIC                 1           More than one subgraph is not supported
ADD                      1           More than one subgraph is not supported
MAX_POOL_2D              1           Mapped to Edge TPU
CONCATENATION            1           More than one subgraph is not supported
TRANSPOSE                1           Tensor has unsupported rank
                                     (up to 3 innermost dimensions mapped)
QUANTIZE                 1           Mapped to Edge TPU
```

Listing 3.2: Failed mapping to Edge TPU

This behaviour confirms that multiple layers of the model do not meet the requirements defined in section 2.8, and it is consistent with the compilation procedure described in the official documentation[14]. Specifically, the compiler stops at the first operation not supported by the Edge TPU and maps all subsequent operations to the CPU, even if some of those later operations would otherwise be supported on the Edge TPU. This behaviour is illustrated below in Figure 10.

Delegation of tensor operations to the CPU will almost certainly increase inference time drastically, which is inappropriate here, as the goal is to map the entire model to the Edge TPU.

Because insertion of the `Transpose` TF op occurs entirely internally and automatically, there is no practical way to influence it or to enforce layer parameters to be constant at compile time and of shape < 3, as required by the Edge TPU Compiler.

Two options were considered: (i) avoid `TransposeConv` altogether, or (ii) continue searching for compatible TF and TFMOT versions. Pursuing the second option places one between a rock and a hard place: the TF version should be kept as low as possible

[14] https://coral.ai/docs/edgetpu/models-intro/#compiling

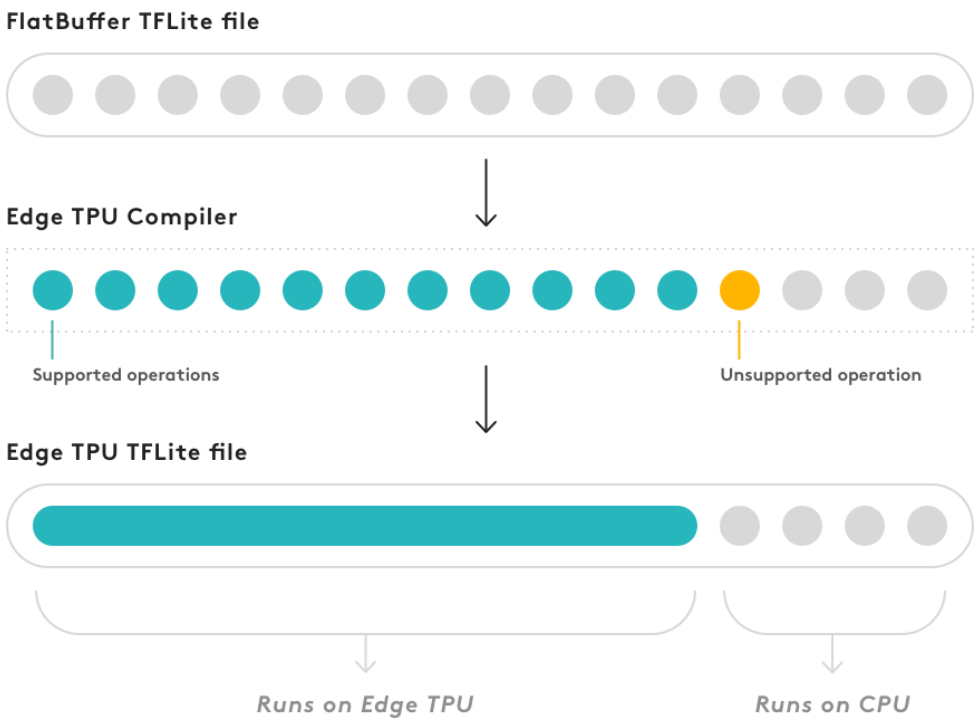


Figure 10: The compiler creates a single custom op for all Edge TPU compatible ops, until it encounters an unsupported op; the rest stays the same and runs on the CPU

to maintain compatibility with the device TF Lite runtime and to avoid the previously observed `TRANSPOSE_CONV` version 4 compiler error, yet sufficiently high to include the bug fix for the scale incompatibility encountered earlier. Accordingly, the following approach was adopted: identify the highest TF release that still uses `TransposeConv` version 3. This approach assumes that any newer TF ops introduced in later TF versions neither cause compilation issues nor conflict with the older TF Lite v2.5.0 on the inference device. In effect, if the Edge TPU Compiler successfully fuses all TF ops into a single `edgetpu-custom-op`, the TF Lite runtime v2.5.0 only needs to parse and load the model before delegation to the Edge TPU. Some compatibility uncertainty therefore remains, especially for newer (or custom) layer types, but for the purposes of this work, this assumption is considered sufficient.

In the absence of official documentation at the required level of detail for individual TF ops, the compatible `TransposeConv` version was identified by inspecting the `register.cc` file in the TF repository under `tensorflow/tensorflow/lite/kernels` (noting that its exact location may vary slightly by version). The newest TF version found to register `TransposeConv` v3 was v2.10.0. The corresponding `register.cc` lines are:

```
AddBuiltin(BuiltinOperator_TRANSPOSE_CONV, Register_TRANSPOSE_CONV(),
           /* min_version = */ 1,
           /* max_version = */ 3);
```

It was found that the latest TFMOT v0.8.0 is backward compatible with TF v2.10.0, despite being tested against TF v2.14.1, as noted on the official releases page[15].

[15] https://github.com/tensorflow/model-optimization/releases

The observed version incompatibilities presumably stem from introducing a newer `libedgetpu` runtime while retaining an older TF Lite on the Dev Board, without an official upgrade path to align versions. As demonstrated here, this can be misleading when selecting inter-compatible packages, especially when newer releases become available.

Ultimately, the combination of TF v2.10.0 and TFMOT v0.8.0 resolved the earlier errors and enabled successful compilation, fully mapping all TF ops to the Edge TPU, as shown in Figure 8. This setup was used thereafter throughout the pipeline.

3.5 Deployment

After training, conversion, and compilation, the `.tflite` model is deployed on the Edge TPU to run inference.

At early stages of development, deployment was most conveniently performed using Python utilities. Provided by Google, these libraries encapsulate the necessary boilerplate into a small set of functions to load the model onto the Edge TPU, pass the input tensor, and retrieve the output tensor. A simplified Python example is presented below to illustrate the ease of operating the Edge TPU with the provided helpers.

```python
from pycoral.utils.edgetpu import make_interpreter
from pycoral.adapters import common

# Load model
interpreter = make_interpreter(model_path)
interpreter.allocate_tensors()

# Prepare input
common.set_input(interpreter, input_tensor)

# Inference
interpreter.invoke()

# Output
output = common.output_tensor(interpreter, 0)
```

Using the provided Python modules, the first inference results were obtained and are shown alongside the corresponding GT in Figure 11.

These initial single-patch inference results marked a milestone in this work and laid the foundation for the subsequent inference pipeline.

To simplify operation of the Edge TPU hardware, two Python libraries are provided[16]: the PyCoral API and the TF Lite API, with PyCoral wrapping repetitive TF Lite inference steps into simpler helpers.

[16] https://coral.ai/docs/edgetpu/tflite-python/

(a) Predicted cloud mask (b) Corresponding GT

Figure 11: First Edge TPU inference results

After successful implementation of single-patch inference in Python, expansion to a full-scene inference pipeline in C++ was pursued. The main rationale was increased control over the inference process and potential acceleration. Python helper functions can conveniently read `.TIF` images, transform them, and pass tensors to the Edge TPU in a single script. However, this convenience may introduce overhead that is undesirable on an embedded system. A more typical embedded approach was therefore adopted: transform input data on another device, perform inference on the edge device, and post-process outputs elsewhere. An additional factor was a specific error encountered at the end of each Python inference. Greater control over the inference sequence via C++ was expected to clarify and eliminate this issue (see section 3.5).

The Dev Board's Mendel Linux includes, by default, most libraries needed to operate the Edge TPU module: the PyCoral and TF Lite Python APIs, as well as the `libedgetpu` runtime. The essential TF Lite C++ API is not included, however. Moreover, the TF Lite Python package cannot be reused from C++, as its binary targets Python and does not provide C++ headers. Figure 12 summarizes the flow of running the model on the Edge TPU on the Dev Board, highlighting the differences between Python and C++ based inference.

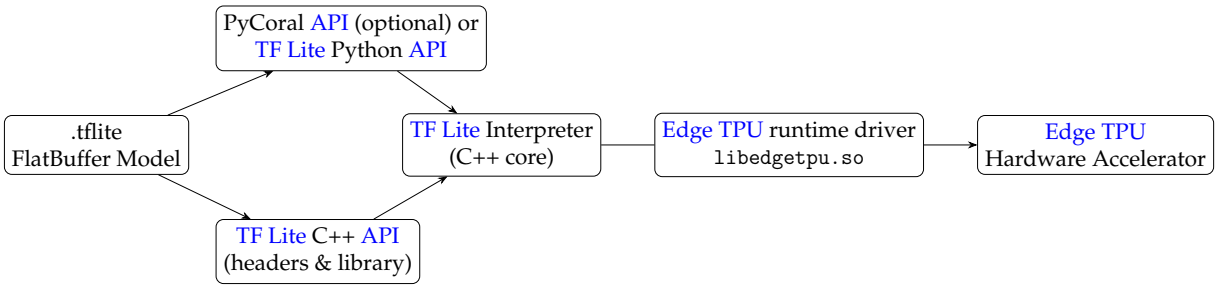


Figure 12: Running the model on Edge TPU with Python and C++

Since no prebuilt TF Lite C++ API libraries were available for the Dev Board, it had to be built from source. The process is described in detail in section 3.5 and marks the third and final technical implementation challenge in this work.

After successfully incorporating the TF Lite C++ API on the Dev Board, it was used to implement the first C++ inference in `inference.cpp`. Following this, a folder-reading utility was implemented in C++ to process multiple patches sequentially.

These patches are stored as `.bin` files with known shape and data type (float32). After inference, the program writes `.bin` outputs containing the predicted cloud masks in float32. Quantization parameters for the input and output are retrieved from the model at runtime and used to quantize the input tensor and dequantize the output tensor. Later, the stitching process was also implemented in C++. By running `stitch.cpp`, all `.bin` patches in the input folder are stitched into a single `.bin` scene without gaps. This `.bin` scene can then be converted to a `.png` for visual inspection. This conversion can optionally be performed on the Dev Board using the Python script `convert.py`.

Note that the entire inference, stitching, and conversion pipeline could have been consolidated into a single C++ executable. It was intentionally kept separate for two reasons: (i) to allow greater flexibility and lower overhead for a future performance-critical satellite implementation, and (ii) because the persistent error described in section 3.5 prevents running any code immediately after the inference step.

As final steps concluding the inference implementation on the Dev Board, the following improvements were made. An SD card containing prepared `.bin` patches for each scene is mounted on the Dev Board, allowing the entire dataset (several GB) to be hosted, otherwise infeasible on the Dev Board's internal storage. The stitching and conversion processes were unified under a single executable: after C++ `.bin` stitching, a Python function is invoked to perform the conversion. During both inference and stitch-conversion processes, the user is prompted to select one of the 20 test scenes for inference and/or conversion. Furthermore, the Python `.png` conversion was improved by avoiding loading the entire output array into the Dev Board's RAM. Instead, chunked processing is used, enabling conversion of larger full resolution scenes.

Technical Challenges

The final significant implementation milestone in this work was building the TF Lite C++ API library for the Dev Board from source. As shown in Figure 12, this library is crucial for implementing pure C++ inference.

Guidance for this task is sparse in the official documentation[17]. Taking those notes into account, a compatible TF Lite version for the `libedgetpu` runtime preinstalled on the Dev Board had to be determined. The installed `libedgetpu` was identified as v16.0. However, the releases page[18] does not use a consistent numeric versioning convention, so the modification date of the `libedgetpu.so.1` symlink was used to approximate the release. On the Dev Board used here this date was 09.07.2021, which predates the official "Grouper" `libedgetpu` release[19] (26.07.2021) by two weeks. From that release's `workspace.bzl`, the exact TF commit corresponding to TF 2.5.0 (05.07.2021) was identified.

The first build attempt was performed on a Windows machine via cross-compilation for `aarch64` Linux. Although compilation succeeded, the first inference test failed due to low-level incompatibilities with the Dev Board's atomic instructions. It was therefore concluded that the library must be built either with Bazel using a full Dev Board

[17] https://coral.ai/docs/edgetpu/tflite-cpp/#build-your-project-with-libedgetpu
[18] https://github.com/google-coral/libedgetpu/releases
[19] https://github.com/google-coral/libedgetpu/tree/release-grouper

configuration or directly on the device. The latter was chosen. An SD card was used to host the large TF source tree (specific v2.5.0 commit) and to provide swap space, to avoid running out of RAM during compilation. The build was performed using the then-latest portable `CMake`.

After successfully building the static library `libtensorflow-lite.a`, several additional steps were required. The `edgetpu.h` header[20] and a FlatBuffers library (v1.12.0) compatible with the build were obtained, and all required TF Lite static libraries were included. The necessary include paths and libraries are recorded in the repository and can also be reviewed in the `MAKEFILE`.

Once compiler and linker issues were resolved and all requirements satisfied, successful model loading and tensor allocation using the C++ API marked the final technical milestone of this work.

Open Issue

In the scope of this work, and with regard to the specific Dev Board with its installed Mendel Linux OS and modules, an unusual behaviour after inference was observed. From early development through the final C++ inference improvements, one of two errors persistently appeared only after the inference had completed: `Segmentation fault` or `Bus error`. The phenomenon was first encountered during the simple proof-of-concept Python inference, at program termination after results had been successfully saved. Although such errors are typically disastrous, their occurrence at shutdown had no effect on obtaining the required inference outputs. As they clearly originated in C++ backend code, debugging from Python was not feasible; nonetheless, pipeline development proceeded unaffected.

The move to a pure C++ inference was motivated in part by the desire to eliminate these errors. However, even after the first successful C++ results, the errors continued to alternate upon exiting `main()` in `inference.cpp`. During debugging, the source was narrowed down to the `libedgetpu` library — presumably during its cleanup routine after inference. Exact lines could not be traced, as the provided `libedgetpu` was built without debugging symbols. Diagnosing this would require rebuilding the library with debug information, a process likely to entail dependency issues and other challenges typical of custom low-level builds. This appears to be a worthwhile line of investigation but lies beyond the scope of this thesis. It is presumed that the issue was fixed in later `libedgetpu` releases[21], but without an official upgrade path on the device, a custom rebuild would have been inevitable.

Work was therefore continued, as the errors had no impact on performance or results. Multiple `Interpreter` invocations can be executed to process folders of files. However, no code can be run after inference completes, because the program crashes at exit. Consequently, the stitching and conversion stages were implemented as separate programs.

[20] https://github.com/google-coral/libedgetpu/blob/master/tflite/public/edgetpu.h
[21] https://github.com/tensorflow/tensorflow/issues/62371

3.6 Evaluation Pipeline

With the implementation of the deployment pipeline on the Dev Board, cloud prediction masks can be obtained. Visual inspection by a human observer is used to qualitatively confirm successful implementation and attainment of the thesis objective. Furthermore, a quantitative performance metric was implemented in `inference.cpp` on the Dev Board to determine the inference time per processed patch; by summing these times, the model's per-scene inference time is obtained.

The flexibility of the training and deployment pipelines allows convenient, modular improvement and feature expansion. They can be tailored to the needs and requirements of model development intended for immediate satellite deployment. For this, additional quantitative evaluation of model performance is necessary. The results are ultimately used in chapter 4, where the performance of several Edge TPU optimized models is assessed.

The evaluation pipeline is implemented as an additional stage following the training pipeline and is run on the training machine, not on the Dev Board. The evaluation pipeline consists of two parts: inference and evaluation.

Inference

The necessary functions to perform inference and save results are implemented in the files `inference.py` and `inferenceQ.py`. `inferenceQ.py` uses the TF Lite `Interpreter` with the fully int8 quantized `.tflite` model, whereas `inference.py` relies on TF utilities and loads the `.h5` saved-model format. It was observed that running the quantized model on a CPU results in significantly slower inference than the float32 `.h5` model, which slows down the evaluation. Moreover, executing the quantized model on a general-purpose CPU Intel Core i5-13400F yielded inference times even slower than deployment on the dedicated Edge TPU. This inherently emphasizes the sophistication of the TPU in performing these operations, compared to a general-purpose CPU that is otherwise more powerful for general tasks. Consequently, quantized inference was omitted during evaluation, and `inference.py` is used for the remainder.

First, the `buildDS` function is used to construct the testing dataset. As mentioned in subsection 3.1.2, prebuilt `.csv` files are used to assemble either a single-scene dataset (specified by ID or chosen at random) or the full test set. In later versions of `inference.py`, full-dataset inference is handled internally on a scene-wise basis, reducing RAM usage and preventing crashes. Accordingly, `fullTestDS.csv` is no longer used. Next, the path to the desired `run` folder (containing the `.h5` and `.tflite` files) is specified. In `inference.py` the `.h5` model is loaded, providing any required custom TF objects (e.g., `softJaccardLoss`, `diceCoefficient`).

If model predictions are produced at reduced resolution (e.g., 192×192 patches), they are upsampled to the full 384×384 resolution to match full-scene GTs during evaluation. Upsampling is performed with `bilinear` interpolation (see also section 3.1), which is appropriate for continuous probability maps.

Finally, the inference results are saved per scene as `.npy` files for use in the evaluation

process. Optionally, masks can also be saved as `.png` files for visual inspection.

Evaluation

After obtaining inference results for the complete test set (20 scenes), they are quantified using the evaluation metrics defined in section 2.3, as implemented in `evaluate.py` file. By specifying the `run` folder, that contains the `evaluation/inference` subfolder, the algorithm locates and processes each scene's `.npy` file produced by the inference step.

As mentioned in section 2.4, the test input scenes were cropped and padded to fit the 384×384 patch format. This introduces zero padding at the scene borders. To restore the original scene size of the GT masks, the border padding must be removed. This is performed by center-cropping the prediction masks via the helper function `unpadToMatch`, following the procedure used in the original dataset papers [MS19; S M18] (cf. their `unzeropad` routine used for evaluation[22]).

Next, the evaluation pipeline offers two ways to choose the threshold (see section 2.3) for binarizing the predicted masks: (i) apply the best threshold estimated on the validation set, or (ii) determine the best threshold per test scene using its GT mask. The effects of these choices are discussed in the next chapter 4. This functionality is controlled via the optional `fixedThreshold` argument to `evaluateAll` function.

As a final step, the prediction masks are binarized and metrics are computed using `computeMetrics` function. Results are saved to `metricsValThr.csv` or `metricsTestThr.csv`, depending on the thresholding variant. Optionally, per-scene `.npy` results can be saved alongside `.png` masks for visual inspection.

This process concludes the *Implementation* chapter and finalizes the software development conducted within the scope of this work.

[22] https://github.com/SorourMo/38-Cloud-A-Cloud-Segmentation-Dataset/blob/master/evaluation/evaluation.m

4 Evaluation

A flexible pipeline enables convenient training, conversion, and deployment of diverse model configurations for the Dev Board. It supports fast switching between model architecture, training, and deployment settings — for example, changing the number of input channels, the spatial resolution of the input tensor, or the composition of the training/validation/testing subsets. Tailoring these configurations to user needs requires only minor code changes.

During development, numerous model architectures were designed and trained, most serving as prototypes for debugging. For instance, the `simple` model produced the first inference results (see section 3.5). After the implementation phase, five models were selected for evaluation. These were trained for an appropriate number of epochs, their inference time measured on the Dev Board, and their metrics computed via the evaluation pipeline. One of these models was the first one to yield qualitatively high results: a simplified, edge adapted `Cloud-Net` architecture [MS19], referred to here as `earlyCloudEdgeQ`. The remaining four models share an improved, identical architecture `improvedCloudEdgeQ`, and differ only in (i) input channels (RGB+NIR vs. RGB) and (ii) input patch size (192×192 vs. 384×384).

Note that all five models follow the classical `U-Net` structure (see section 2.2), comprising an encoder, bottleneck, and decoder, with skip connections between corresponding encoder and decoder blocks.

Detailed descriptions of both architectures follow:

- **earlyCloudEdgeQ**: The model hosts three encoder blocks, a bottleneck block, and three decoder blocks. Each encoder block consists of two `Conv2D` layers and one `MaxPooling2D` layer, connected sequentially. The bottleneck consists of two sequential `Conv2D` layers. Each decoder block contains a `Conv2DTranspose` layer, a `Concatenate` layer, and two `Conv2D` layers, connected sequentially. The output layer is a `Conv2D` layer. The base number of filters is 32 and is doubled in each encoder block and halved in each decoder block, resulting in 256 filters at the bottleneck. BN is applied after each `Conv2D` layer and before its activation; it is not applied in the output layer. All suitable TF ops are QAT-annotated. Binary Crossentropy (BCE) is used as the loss function and `Adam` [KB14] as the optimizer.
- **improvedCloudEdgeQ**: This model follows the previous architecture with the following refinements. Between the input and the first encoder block, a `Conv2D` layer with 16 filters is inserted to improve feature representation at the input stage, enabling the encoder to capture more consistent low-level patterns before deeper processing. The model comprises four encoder blocks, a bottleneck, and four decoder blocks. The bottleneck design is adapted from `Cloud-Net`, where a residual connection with a 1×1 projection ensures feature-dimension alignment and efficient gradient flow. Dropout regularization is added to reduce overfit-

ting and improve robustness at the network's deepest layer. To meet the Edge TPU's limited on-chip static RAM capacity, the decoder blocks were simplified by removing one `Conv2D` layer from each block, except for the final decoder block before the output layer, which retains two `Conv2D` layers to preserve finer spatial detail. Additionally, using BCE combined with $0.5 \cdot \text{DiceLoss}$ as the loss function balances pixel-wise accuracy with overlap-based region matching, leading to better segmentation performance than plain BCE [Raj21; Mon+23].

`improvedCloudEdgeQ` can be summarized as an adjusted `Cloud-Net` model tailored to the constraints of embedded Edge TPU development. Necessary simplifications were made. Both `earlyCloudEdgeQ` and `improvedCloudEdgeQ` can be examined in more detail in `model.py` and compared with the original `Cloud-Net` model architecture.[1]

Table 4.1 details the differences between the models, reporting the number of input channels (CH), input-patch size (In size), number of trainable parameters in millions (Params (M)), number of training epochs (Epochs, suffixed with `eS` if `EarlyStopping` was triggered), estimated arithmetic operations in billions (Ops (B)), on-chip/off-chip memory used to cache model parameters (On-chip (MB) / Off-chip (KB)), and the average per-batch inference time on the Dev Board (Inf. Time (ms)).

Model	CH	In size	Params (M)	Epochs	Ops (B)	On-chip (MB)	Off-chip (KB)	Inf. Time (ms)
early	4	192	1.9	100	10.4	1.9	46	506
improved 1	3	192	6.0	54 eS	11.6	5.7	342	514
improved 2	3	384	6.0	46 eS	46.3	6.0	349	2435
improved 3	4	192	6.0	51 eS	11.6	5.7	342	515
improved 4	4	384	6.0	45 eS	46.4	6.0	349	2440

Table 4.1: Model properties and deployment metrics.

Models were evaluated on a test set containing 20 scenes. The evaluation metrics defined in section 2.3, IoU, Dice coefficient, Precision, Recall, and Accuracy, were computed per scene and then averaged. This averaging does not bias the results because all scenes are approximately the same size.

For all models except `early`, the binarization threshold was calibrated immediately after training by selecting the `bestF1` on the validation set, yielding an inference-ready model with its own threshold for mask binarization. For `early`, no threshold was computed because the feature had not yet been implemented. Reproducible dataset shuffling was also not available at that time, precluding a comparable PRC evaluation on the same validation set.

In addition to the validation threshold, a per-scene threshold was computed for each of the 20 test scenes and then averaged. This procedure is even more optimistic than using a single global test threshold, since it adapts to each scene and thus overestimates true generalization performance. Nevertheless, reporting these results is useful: they illustrate an upper bound under ideal threshold tuning and underscore the strong influence of threshold choice on segmentation quality.

[1] https://github.com/SorourMo/Cloud-Net-A-semantic-segmentation-CNN-for-cloud-detection/blob/master/Cloud-Net/cloud_net_model.py

The results are shown in Figure 13 below. The second and third panels are derived from the first by splitting results by validation threshold versus test threshold for clearer visual comparison. `improvedCloudEdgeQ` models are labeled by configuration (number of input channels and input patch size), and `earlyCloudEdgeQ` is additionally marked as `early`.

Figure 13: Evaluation Results

By analyzing these results, the following key insights emerge:

- A general upward performance trend is evident with increasing input patch resolution and with the addition of the NIR channel. This is expected, as the model benefits from additional spectral information.
- `improvedCloudEdgeQ` generally outperforms `earlyCloudEdgeQ`, consistent with its architectural refinements.
- At 192×192 input size, adding NIR did not improve recall and in fact led to a slight reduction (when using the validation-derived threshold). A likely cause is that the extra spectral channel increases input dimensionality without providing sufficient spatial context at the lower resolution: the model becomes more conservative at cloud boundaries and thin structures, reducing false positives but slightly increasing missed cloud pixels (lower recall).

- Accuracy is consistently very high, but it should not be treated as a reliable metric for cloud segmentation. If non cloud pixels vastly outnumber cloud pixels, accuracy is dominated by true negatives, so even a model that misses many clouds can score highly: hence IoU, Dice, Precision, and Recall are preferred.

Considering the evaluation metrics together with inference time on the Dev Board, the `improvedCloudEdgeQ` model with RGB+NIR inputs and 192×192 resolution is recommended as the most suitable choice in this work, striking a balance between edge-device speed and segmentation quality. However, if NIR channel is unavailable, the preferred model is `improvedCloudEdgeQ` at 384×384, trading a significantly longer inference time for stronger performance.

5 Conclusions & Outlook

The objective of this thesis was to design, train, and deploy a CNN model for cloud segmentation on an embedded system Google Coral Dev Board Mini. To achieve this objective, configurable training and deployment pipelines were implemented. During implementation, three major technical issues were identified and successively addressed; they are documented as milestones in chapter 3. The complete pipelines enabled convenient training, deployment, and performance evaluation of five CNN networks designed in this work. Particular attention was paid to tailoring these models for efficient inference on the Edge TPU. Difficulties and caveats in conversion and compilation were overcome, and the successful delegation of all tensor operations to the Edge TPU was confirmed and documented.

The `improvedCloudEdgeQ` architecture achieved performance comparable to the original `Cloud-Net` [MS19]. With RGB inputs and 192×192 patches, it reached a Jaccard Index of 75.85%, versus 78.50% for `Cloud-Net` on the same 20 scene test set. This corresponds to a modest 2.65 percentage points (3.38% relative) decrease in performance in exchange for fast embedded inference that does not require NIR input and processes one Landsat 8 scene in 216 s on the Dev Board.

Several improvements are suggested, based on insights gathered during development. An extension of the 38-Cloud dataset, as introduced in [MS21], could be used for more effective training. Architecture improvements proposed in [MS21] can be adapted and ported to the Dev Board using the pipelines developed here.

When loading the model onto the Edge TPU, off-chip memory usage should be minimized to preserve inference speed. The error outlined in section 3.5 should be investigated and eliminated to allow post-processing immediately after inference. The C++ inference implementation and the associated development experience enable further refinements, ultimately supporting adaptation of the deployment algorithms for onboard satellite use under space-grade requirements.

Overall, this work lays a foundation for exploring the feasibility of COTS components in satellites. Beyond simple tensor-processing payloads for radiation tests, the resulting pipeline and models constitute a versatile toolchain for developing sophisticated cloud segmentation algorithms on Edge TPU based embedded systems.

Bibliography

[Aba+16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC]. URL: https://arxiv.org/abs/1603.04467.

[Bla79] David C. Blair. "Information Retrieval, 2nd ed. C.J. Van Rijsbergen. London: Butterworths; 1979: 208 pp. Price: $32.50". In: *Journal of the American Society for Information Science* 30.6 (1979), pp. 374–375. DOI: https://doi.org/10.1002/asi.4630300621. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/asi.4630300621. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.4630300621.

[DG06] Jesse Davis and Mark Goadrich. "The Relationship Between Precision-Recall and ROC Curves". In: vol. 06. June 2006. DOI: 10.1145/1143844.1143874.

[Dic45] Lee R. Dice. "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3 (1945), pp. 297–302. DOI: https://doi.org/10.2307/1932409. eprint: https://esajournals.onlinelibrary.wiley.com/doi/pdf/10.2307/1932409. URL: https://esajournals.onlinelibrary.wiley.com/doi/abs/10.2307/1932409.

[GW17] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. English. 4th ed. E-ISBN: 978-1-292-22307-0. Global Edition: Pearson, 2017, p. 1024. ISBN: 978-1-292-22304-9.

[He+15] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[HMD16] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV]. URL: https://arxiv.org/abs/1510.00149.

[Hub+16] Itay Hubara et al. *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*. 2016. arXiv: 1609.07061 [cs.NE]. URL: https://arxiv.org/abs/1609.07061.

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[Jac01] Paul Jaccard. "Etude de la distribution florale dans une portion des Alpes et du Jura". In: *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (Jan. 1901), pp. 547–579. DOI: 10.5169/seals-266450.

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106.

[Llo82] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[Mon+23] M. Montazerolghaem et al. "U-Net Architecture for Prostate Segmentation: The Impact of Loss Function on System Performance". In: *Bioengineering (Basel)* 10.4 (2023), p. 412. DOI: 10.3390/bioengineering10040412.

[MS19] Sorour Mohajerani and Parvaneh Saeedi. *Cloud-Net: An end-to-end Cloud Detection Algorithm for Landsat 8 Imagery*. 2019. arXiv: 1901.10077 [cs.CV]. URL: https://arxiv.org/abs/1901.10077.

[MS21] Sorour Mohajerani and Parvaneh Saeedi. "Cloud and Cloud Shadow Segmentation for Remote Sensing Imagery Via Filtered Jaccard Loss Function and Parametric Augmentation". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 14 (2021), pp. 4254–4266. ISSN: 2151-1535. DOI: 10.1109/jstars.2021.3070786. URL: http://dx.doi.org/10.1109/JSTARS.2021.3070786.

[Nag+21] Markus Nagel et al. *A White Paper on Neural Network Quantization*. 2021. arXiv: 2106.08295 [cs.LG]. URL: https://arxiv.org/abs/2106.08295.

[Raj21] Vishal Rajput. *Robustness of different loss functions and their impact on networks learning capability*. 2021. arXiv: 2110.08322 [cs.LG]. URL: https://arxiv.org/abs/2110.08322.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[Ruk23] Olivier Rukundo. "Effects of Image Size on Deep Learning". In: *Electronics* 12.4 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12040985. URL: https://www.mdpi.com/2079-9292/12/4/985.

[S M18] P. Saeedi S. Mohajerani T. A. Krammer. "A Cloud Detection Algorithm for Remote Sensing Images Using Fully Convolutional Neural Networks". In: *IEEE International Workshop on Multimedia Signal Processing* (2018).

[Wu+16] Jiaxiang Wu et al. *Quantized Convolutional Neural Networks for Mobile Devices*. 2016. arXiv: 1512.06473 [cs.CV]. URL: https://arxiv.org/abs/1512.06473.