\subsection\*{Deployment}

%\label{subsec:designdep}

A tool must be developed on the \gls{devboard}. This program should preload the testing data, load the trained model, and pass

tensors for inference. In early development (for debugging), 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.

\subsection\*{Evaluation}

After successful implementation and verification of the training and deployment pipelines,

multiple versions of \glspl{cnn} can be designed and trained using the developed tools.

Training metrics must be documented. Following successful deployment on the \gls{devboard},

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 \secshortref{subsec:evalmetrics}.

The results and key insights will be documented for future applications.

\section{Evaluation Metrics of Convolutional Neural Networks}

\label{subsec:evalmetrics}

After perfoming a binary cloud segmentation task, a \gls{cnn} can output a probability prediction mask,

which typically contains predicted continuous values of [0,1]. The decision has to be made, after what \code{threshold} the values have to be

binarized to one (cloud is present on pixel). The values smaller than this threshold will be turned to zero (cloud is not present on pixel).

After that, the predicted cloud segmentattion mask can be compared with \gls{gt} mask. There can emerge four possible combination of overlapped pixels:

\begin{itemize}

    \item \textbf{**True Positive (TP):**} Pixel is cloud in prediction and in \gls{gt}.

    \item \textbf{**True Negative (TN):**} Pixel is not cloud in prediction and in \gls{gt}.

    \item \textbf{**False Positive (FP):**} Pixel is cloud in prediction but not in \gls{gt}.

    \item \textbf{**False Negative (FN):**} Pixel is not cloud in prediction but cloud in \gls{gt}.

\end{itemize}

Using these combinations, today's widely accepted metrics for evaluating \gls{cnn} model performance on cloud segmentation task are made.

Following metrics will be utilized in this work.

\[

\begin{array}{llcl}

\text{Accuracy}  &= \dfrac{TP + TN}{TP + TN + FP + FN}

& \quad \text{Precision} &= \dfrac{TP}{TP + FP} \\[1.2em]

\text{Recall}    &= \dfrac{TP}{TP + FN}

& \quad \text{Jaccard Index}       &= \dfrac{TP}{TP + FP + FN} \\[1.2em]

\text{Dice Coefficient}      &= \dfrac{2TP}{2TP + FP + FN} & &

\end{array}

\]

Accuracy represents the fraction of correctly predicted pixels in relation to total amount of pixels.

Precision is a metric, that represents how conservative is a model. High precision reflects that the model is predicting a pixel as cloud only when its very sure,

but this recults in potentionally missed clouds. High Recall, on the other hand, represents a model trying to catch all clouds, resulting in fewer misses,

but potentially predicting not cloud as a cloud. Jaccard Index \cite{jaccardindex} is also known in the context of computer vision also under the name: Intersection over Union.

Together with Dice Coefficient \cite{dicecoefficient} these metrics are relating recall and precision and are measures of the similarity between two sets.

It has furthermore to be noted, that the best threshold, i.e. after applying which the predicted binary mask will correspond to the \gls{gt} mask the most,

is counterintuitively not 0.5, rather it can be any value in the predicted values' range. In order to find that valuse, Precision-Recall-Curve evaluation

has to be performed \cite{prc}.

One of the most influential architectures for image segmenation is the \code{U-Net}.

Originally introduced by Ronneberger et al.~\cite{ronneberger2015u} for biomedical image segmentation,

\code{U-Net} has become a standard in many domains, including remote sensing and cloud detection.

This network architecture proved that it can become remarkably good at segmentation tasks, even given the less amount of training data.

A \code{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 results in output imgage, that has the same spatial dimensions height and width, but could have a various number of channels, depending on the perceived task.

The \code{U-Net} architecture shown in the diagram 1 was used by Mohajerani et al.~\cite{mohajerani2019cloudnet}  for their cloud detection algorithm.

\glspl{cnn} are a class of neural networks that apply small, learnable filters, known as convolutional kernels,

across an image to extract spatial features. A \gls{cnn} typically consists of multiple convolutional layers stacked sequentionally.

Each layer applies a set of filters that capture different visual patterns, such as edges or textures.

As the network goes deeper, the spatial resolution of the image decreases, while the depth (i.e., 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.

Filter parameters can be handcrafted to extract specific features of the image. This technique is used in image processing.

In the field of deep learning these parameters, however, become learnable. Given the sufficient depth' network of filters,

gradient descend techniques are applied on the function, that measures how widely the results from the network correspond to the known \gls{gt} result.

In such a way the network can be optimized in extracting necessary information from images.

Automated extraction of cloud masks may be challenging when analyzing only visible light bands: \gls{rgb}.

Through the use of additional sensors on satellite several other wavelengths can be captured, such as \glsxtrlong{nir},

Short-wave Infrared, Thermal Infrared. Each of these channels can benifit cloud segmentation in its own way.

Given the 38-Cloud dataset, which contains \gls{rgb} and \gls{nir} channel images from Landsat 8 mission,

this work will focus on them.

\gls{nir} especially helps to distinguish clouds over the liquid water regions, as it is being absorbed by them strongly. It furthermore

helps separate clouds from green vegetation regions, as they are bright in \gls{nir} channel.

But OOV-CUBE satellite hosts only a \gls{rgb} camera onboard. Cloud segmenation, utilitizing only these channells

is surely possible, but could provide less accurate segmentation masks. So the necessity to provide and evaluate

the model, that utilizes only visible channels will be considered in this work.

Later on, more and more sophisticated systems and algorithms are being developed

The first successful weather satellite TIROS-1 was launched on April 1st, 1960. During its mission,

it delivered thousands of photographs containing cloud-cover views of the Earth. These pictures have proven to be extremely

helpful in predicting large-scale weather changes and significantly improved our weather forecasting capabilities.

After that, the necessity to distinguish cloud from non-cloud regions has arisen, because of the necessity to

remove clouds from pisture for one task or stydy the cloud cover in the other, therefore requiring to distinguish it from the ground structures.

In weather foresasting time series, cloud layer jourey can be tracked and predicted, if analyzed by the human secialist.

But the automation of this is, where cloud segmenation comes into play. At the early stages of satellite weather observations,

a lot was relied on manual and impirical thresholding, coming over the years to non-machine learning image processing, to utilizing

deep learning models, such as \glspl{cnn} to achive this task.

\section{Conversion}

\label{sec:conversion}

The file \code{convert.py} contains all necessary functions and utilities for model conversion and \gls{edgetpu} compiling.

These functions are called from \code{main.py} right after the end of the training process.

As mentioned in \secshortref{subsec:hardware} the \gls{edgetpu} supports tensors with at most three dimensions.

This necessitates using batch size one, since input tensor shape already is \ensuremath{image height\times image width\times number of channels}.

During training, however, greater batch sizes are used for effectiveness.

Therefore immediatly after completing the training the function \code{asBatchOne} transfers the trained weights to the exact same model architecture,

but with batch size set to one at input, during all inside computations and at the output.

Therefore \code{tf.lite.TFLiteConverter} is used to perform \gls{ptq},

as well as convert a model to \code{.tflite} format \footnote{\url{https://www.tensorflow.org/api\_docs/python/tf/lite/TFLiteConverter}}.

Representative dataset is utilized and generated using \code{representativeDatasetGen} function.

Training \gls{rgb} and \gls{nir} patches are used, while their number can be varied using \code{numCalBatches}, defined in \secshortref{sec:training}.

The more calibration data is used, the higher is the probability, that it will represent the value range of inference data as close as possible.

At a next step, all supported \gls{tfop} are quantized. Model inputs and outputs are quantized to \gls{int8} as well.

It is important to mention, that in general various datatypes can be used for quantization,

in the scope of this work and regarding the compatibility with \gls{edgetpu} (as outlined in \secshortref{subsec:hardware}) \gls{int8} or \gls{uint8} are used.

Quantized model is then saved and \gls{edgetpu} compiler is called.

Because the training process could either be done on a personal use machine running on \code{Windows},

as well as on cloud GPU instance from Thunder Compute running on \code{Linux},

the cross platform automation of the pipline is ensured by using helper functions and platform identification before invoking an \gls{edgetpu} Compiler.

Due to the ability of \gls{edgetpu} Compiler to run only on Debian-based Linux systems, the Windows Subsystem for Linux on a \code{Windows} machine was used.

As a final result the \code{quant\_edgetpu.tflite} file is saved, representing a quantized \gls{edgetpu} compatible model, ready for inference.

\todo{TODO put netron.app with before and after compile. Describe it in text. Mention netron! Look out for other todos in text above!}

\subsection{Conversion}

After the model is trained it has to be converted and compiled in order to prepare it for \gls{edgetpu} inference.

Modular design has to be implemented here too, simplifying the debugging and ensuring correct conversion and compilation at the early development stages.

Later on, the embedding into training pipeline, as mentioned in \secshortref{subsec:designtrain}, has to be ideally implemented for convenience.

The utulization of the \gls{edgetpu} compiler, provided by Google \footnote{\url{https://coral.ai/docs/edgetpu/compiler/}} has to be automated,

in order to receive at the end of the training and conversion process the ready-for-deployment on \gls{devboard} model.

In order to convert the model after training to the Edge TPU format and to perform PTQ with calibration, several helper functions are defined in convert.py.

As mentioned in [], the Edge TPU only supports models with a batch size of one. Howerver, for effective training, it is necessary to use and experiment with different batch sizes.

Therefore, as the first step in the conversion process, the asBatchOne function transfers the trained weights to a new model with the exact same architecture but with an input batch size of one.

Next, using TensorFlow's TFLiteConverter, the model is converted to the .tflite format, which is specifically desinged for compact deployment on mobile or edge devices.

During this conversion, calibration <TODO explain in concept> is performed using a representative dataset, typically consisting of numerous patches from the training dataset.

Furthermore, the inputs and outputs of the converted model are set to int8 for quantized inference.

As the final step, the model is compiled for Edge TPU inference using the provided edgetpu-compiler (link).

After successful compilation, all ops (explain what it is) in the model are ideally converted into a single Edge TPU custon op ready for deployment.

If the compiler cannot map a certain op to the Edge TPU, it will be mapped to the CPU instead, which can significantly increase inference time.

The goal of this thesis is to meet all model requirements for Edge TPU [chapter with requirements] and ensure that all ops are mapped to it.

The aspired final model conversion result is illustrated below.

After implementing utilities for dataset building and defining the model architecture the training pipeline has to be implemented.

Firstly, it has to start with a crude straightforward training pipeline for initial simple model architectures.

The resulting model files are then passed over next for conversion and deployment.

It has, however, to be further improved, ideally to a single executable script, which covers data loading and passing for training,

steering various necessary training configurtations, and, ideally, using conversion concept to prepare the model for immediate deployment.

One-click training pipeline is aspired.

Training configuration, history, intermediate trained weights have to ideally be stored for reproducibility and backup.

Additionally, computational resources have to taken into consideration. Avoiding long training times by involving powerful machines has to be considered.

\subsubsection{Technical Setup}

Training of pioneer proof-of-concept models was performed on a personal use computer with the following hardware configuration:

CPU Intel Core i5-13400F and GPU NVIDIA GeForce RTX 4060.

In order to use GPU acceleration for \gls{tf} training process, additional special \gls{api} called \gls{cuda} from NVIDIA has to be installed and set up.

Furthermore a specific library \gls{cudnn} has to be used by \gls{cuda}. The following versions were used: \gls{cuda} v11.2, \gls{cudnn} v8.1.1.

These versions are not the newest ones provided by NVIDIA by the time this work is implemented.

The background to this choice of specific older versions lies in compatibility of certain \glspl{tfop} with runtime library \code{libedgtpu}.

This will be discussed more detailed in \secshortref{sec:conversion} and \secshortref{sec:deployment}.

Pioneer models, which are containing approximately 300,000 parameters were trained without \gls{qat} for tens of epochs.

On the described setup, using the full training dataset, the time needed to train for 1 epoch lied under acceptable 15 seconds.

With introduction of more complex models cantaining 2,000,000 and 30,000,000 parameters, as well as at the same time using \gls{qat},

the training time increased drastically. With times needed for 1 epoch of full training dataset to be approximately 3 and 12 minutes respectively.

The training process necessitated, however, at least 100 epochs to reach adequate model performance.

In order to sustain the acceptable training time more powerfull hardware resources were needed.

Thunder Compute is an american start-up GPU cloud platform for machine learning research and data science \cite{thundercompute}.

Utilizing its instance with NVIDIA A100XL GPU the training of larger models was reduced significantly,

dropping to approximately 20 seconds per epoch on 2,000,000 parameter network and 40 seconds per epoch on 30,000,000 parameter network.

A 10x increase in GPU memory to 80GB on A100XL compared to 8GB on GeForce RTX 4060 also allowed training with higher number of batches,

which as a result increased training effectiveness and model robustness.

A significant effort, however, had to be put into setting up the instance in a necessary compatible configuration.

Firstly, the more expensive production mode had to be used, since prototyping mode has not allowed of \gls{cuda} and \gls{cudnn} downgrade \cite{thundercomputeProtProd}.

\gls{cuda} and \gls{cudnn} versions had to be downgraded from existing newest to that point of time ones on the platform to th compatible ones with an older \gls{tf} version.

This was particularly difficuilt due to the necessity of purging the newest \gls{cuda} and \gls{cudnn} files without deleting hardware GPU drivers,

needed and used by installed older \gls{cuda} and \gls{cudnn} versions. As a result, \gls{cuda} v... and \gls{cudnn} v... were installed on the instance and used for further training processes.

Furthermore, development team of Thunder Compute was contacted with a suggestion of improvement to choose \gls{cuda} and \gls{cudnn} at the state of instance creation.

It is already planned as improvement, to pass \gls{cuda} and \gls{cudnn} version as an environment variable during instance startup.

In the \code{main.py} file the full end to end training and conversion pipeline is implemented.

This pipeline loads the training and validation dataset, performs training on a chosen model architecture,

prepares the resdulting trained model for \gls{edgetpu} conversion, quantizes the model by applying the \gls{ptq}, and, as the last step,

compiles it in ready to use on \gls{edgetpu} format.

The following crucial functions are utilized:

\begin{itemize}

    \item Dataset building function \code{builDS} as well as helper utilites if applicable from \code{load.py} file.

    \item Model architecture functions and utilites from \code{model.py}

    \item Quantized conversion functions from \code{convert.py}

\end{itemize}

In order to compile the model to a compatible \code{edgetpu} format, the compiler program provided by Google is used directly after \gls{ptq} \cite{edgetpuCompiler}.

The following essential steps in quantized model development, \gls{ptq} and \gls{qat},

are supported by these utilities and will be introduced here, with their practical application demonstrated in \secshortref{chapter:implementation}.

\subsubsection{Quantization Aware Training}

The \gls{ptq} can be applied on every normally pre-trained model in order to convert its weights from \gls{float32} to \gls{int8}.

Due to precisions loss after clip-effects,

as outlined in \secshortref{subsec:quantization} the model accuracy and precision can be significantly affected and reduced.

To prevent this, the quantization effects are introduced already during the model training.

This is achieved through inserting fake quantization nodes in model layers.

The quantization and immediate dequantization is then preformed on model's weights, biases, activations, inputs and outputs.

\subsubsection{Post-Training Quantization}

After training is complete, the model must be quantized to prepare it for edge deployment.

This means that its weights, biases, and activations must be converted to a quantized representation.

As outlined in \secshortref{subsec:hardware}, the \gls{edgetpu} supports only static full-\gls{int8} quantization.

While various data types can be used to store model parameters during training and after conversion,

in the context of this thesis, \gls{float32} is used for training,

and \gls{int8} is used for quantized model parameters intended for \gls{edgetpu} inference.

It is important to note that after training, the minimum and maximum values of weights and biases can be directly determined,

as these parameters are static and deterministic, and can be used for quantization as described in \secshortref{subsec:quantization}.

By contrast, model activations and outputs depend on the input data, which varies during inference.

Therefore, their quantization requires additional considerations.

Under these preliminaries,

the three general quantization configurations below will be explained for a comprehensive understanding of the quantization process.

Each subsequent method includes the features of the previous one and introduces further improvements:

\begin{itemize}

\item \textbf{**Weight-Only Quantization**}: In this approach, all model weights and biases are quantized,

but the model's inputs, outputs, and activations remain in their original \gls{float32} format.

This already reduces model size and can speed up inference.

However, quantize-dequantize operations are needed within each layer to handle the interface between \gls{float32} inputs/outputs/activations

and \gls{int8} weights/biases.

\item \textbf{**Dynamic Range Quantization**}: This method extends weight-only quantization by quantizing computations involving both

weights and activations on runtime. During inference, input data is used to estimate the value ranges of activations,

which enables more operations to be performed in integer arithmetic, further accelerating inference compared to weight-only quantization.

However, input and output tensors, as well as activations, are still stored and computed in \gls{float32};

only the intermediate computations are quantized.

\item \textbf{**Full-Integer Quantization**}: This method, which is required for compatibility with the \gls{edgetpu},

involves quantizing all \gls{float32} tensors, including inputs, outputs, weights, biases, and activations, to \gls{int8}.

Since the value ranges of activations and outputs depend on the input data, a calibration dataset is required.

Ideally, this dataset should closely represent the expected distribution of inference data. During calibration,

this dataset is passed through the model,

and the observed minima and maxima of the input, output, and activation tensors are used to derive the quantization scales and zero-points.

As a result, the model is fully and statically quantized.

\end{itemize}

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 \gls{tf} utilities.

The following essential steps in quantized model development, \gls{ptq} and \gls{qat},

are supported by these utilities and will be introduced here, with their practical application demonstrated in \secshortref{chapter:implementation}.

\subsubsection{Post Training Quantization}

After the training is complete, the model has to be quantized in order to prepare it for edge deployment.

The means, that its weights, biases and activations have to be quantized.

As outlined in \secshortref{subsec:hardware}, \gls{edgetpu} supports only static full-\gls{int8} quantization.

Various data types can be used for storing the model's parameters both during training and and after conversion.

In the scope of this thesis, however, \gls{float32} is used as data type to store parameters during for model training

and \gls{int8} is used for storing quantized model parameters ready for inference on \gls{edgetpu}.

Furthermore it has to be mentioned, that after training the minimum and maximum values of weights and biases can be gathered,

as they are deterministic and static, and used for quantization process, described in \secshortref{subsec:quantization}.

The model activations and outputs, however, are depending on inputs, which are different every time, especially during inference.

Under these preliminaries, following three general quantization configurations will be briefly explained for full understanding of quantization processes.

Each next method includes the previous one and adds new features:

\begin{itemize}

  \item \textbf{**Weight-Only Quantization**}: All model weights and biases are quantized, but model's inputs, outputs and activations,

  however remain in their initial format \gls{float32}. This method already reduces the model size and speeds up the inference process.

  However, additional quantization-dequantization operations inside of layers have to be performed in order to work with \gls{float32}

  inputs/outputs/activations and \gls{int8} quantized weights/biases.

  \item \textbf{**Dynamic Range Quantization**}: Includes all features of Weight-Only Quantization.

  The key difference is, however, that it additionally \gls{int8} quantizes computations involving both weights and activations during runtime.

  Since actual inputs during inference are providing estimates for the range of values for the activations.

  It boosts the speed of inference further, compared to Weight-Only Quantization.

  The activations themselves as well as input and output tensors are, however, still stored and computed in \gls{float32}.

  \item \textbf{**Full-Integer Quantization**}: This method, ultimately the exclusive one supported by \gls{edgetpu},

  requires quantization of all \gls{float32} tensors, including inputs, output, weights, biases, and activations. This necessitates,

  however a certain estimation of input, output, and activations value range. In order to overcome this, the calibration dataset has to be used.

  This dataset contains either distributed estimated values, which value ranges are ideally corresponding to the estimated distribution of inference

  dataset values. But ideally it has to contain a certain amount of real dataset samples for best results.

  Tensors from this calibration dataset are then forward-passed through the network:

  extrema of input, output, as well as activation values are observed and corresponding scales and zero-points are derived.

  Hence the model is fully statically quantized.

\end{itemize}

2.1.3 Dataset (Refined)

Each image consists of four spectral channels: \gls{rgb} and \gls{nir}, along with a manually annotated \gls{gt} mask that labels cloud regions at the pixel level.

The four spectral channels are encoded using 16-bit unsigned integers per pixel, whereas the \gls{gt} masks are represented with 8-bit unsigned integers. A single raw image at full resolution of approximately 8000x8000 pixels would result in a file size of around 1GB. Moreover, processing such images would require model input tensors of shape \code{batch\_size×8000×8000×4} in single-precision floating-point after normalization. Since efficient training typically necessitates a batch size greater than one, this setup would impose substantial computational demands and significantly slow down both training and inference --- making it particularly unsuitable for deployment on embedded systems. To adress this, the dataset is provided in a pre-cropped format, with each image divided into 384x384 pixel patches. These patches are saved as \code{.TIF} files within their respective channel-specific directories, such as \code{train\_red}, \code{train\_green}, and so forth.

*Each image consists of four spectral channels: \gls{rgb} and \gls{nir}, along with a manually annotated \gls{gt} mask that labels cloud regions at the pixel level.*

*The four spectral channels are encoded using 16-bit unsigned integers per pixel, whereas the \gls{gt} masks are represented with 8-bit unsigned integers. A single raw image at full resolution of approximately 8000x8000 pixels would result in a file size of around 1GB. Moreover, later on, the model input tensor would need the minimum input size of batch\_sizex8000x8000x4 of single-precision floating-points after normalization. With a batch\_size needing to be over 1 for efficient training. This would result in a very heavy and slow model, making it unsutable particularly in the context of embedded deployment. To facilitate more efficient training and testing processes, as well as keep model input tensor size compact, each image is cropped into smaller patches of 384x384 pixels. These patches are saved as \code{.TIF} files in their respective channel-specific directories, such as \code{train\_red}, \code{train\_green}, and so forth.*

*Each pixel of information in each of the four channels is represented by an 16 bit unsigned integer. Ground truth mask pixels are represented by an 8 bit unsigned integer. If representing one image in its full size of approx. 8000x8000 pixels the image file would have a weight of around 244MB. In order to avoid this for accelerated training and testing process, each image is cropped into 384x384 pixel patches. These patches are then stored as .TIF images in their respective channel folders such as train\_red, test\_green etc.*

*This thesis utilizes a dataset consisting of 38 annotated satellite images from the Land-*

*sat 8 mission, commonly referred to as the 38-Cloud dataset [5]. It has been*

*introduced and adapted in the following scientific publications [6], [10]. 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 represented as follows:*

*<folder structure>*

2.1.4 Quantization

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. <cite>

This technique is now widely employed in machine learning, where it is applied to the model weights and activations <cite>. Quantization significantly reduces model size 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 asymetric approach is of primary importance.

To perform quantization, the boundaries of both the original (floating-point) and target (quantized) value sets must be defined. Once these are known, the corresponding scale and zero-point parameters are computed according to the following formulas:

<formulas>

With these parameters determined, input values can be transformed into quantized form and subsequently dequantized using the following formulas.

<formulas>

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 -7.840 to 5.360. To map this range onto a discrete set defined by the integer interval [-128, 127], the scale and zero-point are calculated using formulas (1) and (2):

<provide values>

After this step, every value from the original dataset can be represented by its corresponding quantized counterpart. The following equations illustrate the quantization and subsequent dequantization of five examples using 3 and 4: x=0, x=5,36, x=2,45, x=2,5 and x=2,55.

The last three examples implicitly demonstrate the loss of precision introduced by quantization. In this particular case, any input value between 2,95 and 3,00 will be represented, after conversion, by one of these two discrete values.

It is important to emphasize that the scale parameter etirely defines the quantizer’s precision, under the assumption that the bit-width (i.e., 255 discrete representable values) is fixed, as well as the floating-point range from -7.840 to 5.360 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. However, the detailed explanation of these techniques falls outside the scope of this thesis.

*Quantization is a mathematical method of mapping a large set of (continuous) values to a smaller countable set of values. It was first practically used in pulse code modulation in signal processing in the year 1957, whereas the scientific documentation of the method is first found in 1982 paper based on a draft manuscript from the year mentioned above. <cite>*

*This technique is now widely used also in machine learning, where it is applied on model weights and activations. It reduces the model size significantly trading it for a certain amount of precision loss. There are two quantization methods: general asymmetric zero-point and its special case – symmetric absolute maximum (absmax) quantization. In this work the general case is of an importance.*

*In order to quantize a value the boundaries of both sets have to be known. Then the scale and zero-point are calculated using the following formulas. <formulas>*

*After calculating these two parameters the input values can now be quantized and dequantized using the next two following formulas <formulas>*

*It is important to mention, that the two conversion parameters scale and zero-point have to be stored in order to perform calculations. There always exists a pair of these parameters for each range of values that has to be quantized. The following hands-on example demonstrates the use of zero-point quantization:*

*Lets assume the set of continuous values ranging from -7.840 to 5.360. In order to map this value range onto discrete set from -128 to 127 included, firstly the scale and zero-point are calculated using the formulas 1 and 2: <provide values>*

*After this every value from initial dataset can be represented by its corresponding quantized value. The following equations are representing results of quantization of x=0, x=5.36 and x=2.5 with their corresponding dequantized values:*

*The last three samples are implicitly showing the loss of precision. In this particular case every input value that lies between 2,95 or 3 will be represented after conversion by either of these two numbers.*

*An important note is that scale entirely defines the quantizer’s precision, based on the assumptions in this particular example that bit-width (255 discrete values can be represented) and float-range (-7.840 to 5.360 without outliers) are fixed.*

*In practice more complex quantization methods could be applied, such as, for instance, outlier handling, per-axis/per-tensor, as well as symmetric/asymmetric quantization. The explanation of these will, however, not be covered by this thesis.*

2.2 Concept

2.2.1 Data Preparation

To efficiently utilize TensorFlows dataset loading and preprocessing capabilities, a versatile data handling method is 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 consistensy and flexibility. The function outputs tf.data.Dataset (link) objects that are ready for immediate use in training pipelines.

2.2.2 Model Architecture

Separate from the dataset pipeline, the model architecture is developed independently. Given the lack of existing practical examples and strict deployment constraints, the design process starts with a minimal architecture and gradually increases in complexity based on deployment success and evaluation feedback. A trial-and-error methodology is applied, with iterative adjustments made in response to embedded systems limitations. These limitations necessitate such architectural decisions, as the use of quantization aware training (QAT). For this purpose, model layers are annotaded appropriately to support later quantization and efficient deployment on hardware with limited resources.

*2.2.1 Data preparation*

*The versatile method in order to effectively take advantage of TensorFlow dataset loading and procissing tools has to be implemented. One of the core ideas is to combine training, validation and testing substes under one function with numerous parameters as inputs. This function its has to handle all dataset preparation and internal data transformations as required by user through given parameters. The function has to output ready to use for training TensorFlow Dataset [link] objects.*

*2.2.2 Model architecture*

*Separately from dataset vreation a model architecting should take place. Given implementation restrictions and the absence of functional practical examples the architecture has to start from the simpliest one and become more complex in the scenario of successful deployment. Try out and evaluate the results method has to be used and adjustments needed to be implemented on the go with respect to prerequisites and constraits of embedded systems. Such as constructing a model with quantized annotaded layers for quantization aware training (QAT) which will be mentioned in the immediate next subchapter.*

1.3 Document structure

This thesis is structured as follows: in Chapter 2, preliminary knowledge on essential topics is introduced, along with the concept developed to achieve the goals of this thesis. Chapter 3 provides a detailed description of the implementation of these conceptual ideas. In Chapter 4, a quantitative evaluation of the obtained results is presented. Chapter 5 concludes the thesis by summarizing the key findings and providing an outlook on future work. A secondary goal of this thesis is to thoroughly document the entire process – from initial concept to a fully functional solution – in a clear and practical manner, enabling efficient replication and learning from our experience.

*This thesis is structured as follows: in the second chapter we indroduce preliminary knowledge on the necessary topics, as well as the concept, which was made by us, to achieve the goals of this thesis. In the 3d chapter the implementation of concept ideas gets detailed description. The 5th chapter proceeds with quantitative evaluation of gathered results. In the 6th chapter conslusions are discussed as well as outlook ideas are generated. A persieved secondary goal of this thesis is to describe the full process of going from ideas to a fully functional product in such a practical way, that whoever gets to replicate it could do it in a fast and efficient way leardned from our experience.*

Implementation

1. Dataset loading pipeline

As mentioned in section 2.3, the dataset is split into 18 training and 20 test images. Each full scene is stored as a 16-bit unsigned integer .TIF file with four channels: RGB and NIR (TODO maybe add spectral range in nm). These images have a resolution of around 8000x8000 pixels and a size of approximately 130 MB. To reduce memory load and optimize training, validation and testing performance, the dataset is preprocessed by cropping each scene into smaller patches of 384x384 pixels.

Unlike typical RGB images used in computer vision, the channels are not combined. Instead, each channel is stored in its respective directory for both training and testing subsets. For visualization purposes, the full scene images are rendered using a standard false-color composite in which the NIR band is mapped to red, red to green, and green to blue. This mapping coresponds to the commonly used Color Infrared (CIR) visualization in remote sensing. CIR imagery enhances features such as vegetation and clouds, which exhibit distinct reflectance patterns across the visible and NIR spectrum.

However, these false-color visualizations were not used as input for the CNN during training, validation, or inference. They served solely as visual references for qualitative insprection by the human observer.

TODO check patches and scenes to verify what bands were replaced

TODO provide 1 scene false color image and rgb nir gt patches as example images

The following folder structure of the dataset was used:

TODO provide folder structure

Additionally .csv files containing lists of patch filenames and corresponding scene IDs were provided and leveraged for efficient dataset construction using TensorFlow data pipeline utilites.

One notable detail is that, due to the cropping and padding of border patches (in order to achieve standard patch size 384x384) and the tilted geometry of Landsat 8 imagery, some resulting patches contain no meaningful information – appearing completelly black across all four channels. These empty patches were excluded from training and validation to avoid introducing noise or misleading the model.

*As mentioned in section 2.3 the dataset is split into 18 training and 20 test images. Each entire scene is a .TIF 16 bit uint 4 channel (RGB and NIR <maybe add spectral range in nm?>) image with size around 8000x8000 pixels. Weighing around 130MB each it requeres preprocessing in order to lighten and maximize the effectiveness of training, validation and testing. The dataset is already prepared by cropping each scene into 384x384 pixels patches. Each one out of 4 channels is represented separately in training as well as in testing subsets. ГUnlike other computer vision images, these channels are not combined together. Instead, they are in their correspondig directories. For additional visualization purposes the entire scenes are represented using standard false-color composite, where the NIR band was mapped to red, red to green, green to blue. This is consistent with traditional false-color visualizations in remote sensing, commonly reffered to as Color Infrared (CIR). Such composites are particularly useful for distinguishing vegetation and clouds, which reflects differently across the visible and NIR spectrum. However, the natural false color entire scenes were not used by CNN at any time during training, validation and testing. They served more as visual confirmation for human eye results checking. <TODO check patches and scenes to verify what bands were replaced>.*

*<Provide 1 entire scene false color image and r,g,b,nir,gt patches as example images>*

*Following folder structure of the dataset was exploited <provide folder sturture>*

*.csv files containing names of training and testing patches and sceneids were provided and used for efficient dataset building with TensorFlow methods.*

*An important detail to mention is, that after cropping and padding border patches in order to stadartise 384x384 pixel patches as well as due to tilted images in original Landsat 8 scenes a certain amount of patches contains no information and is therefor pitch black in all 4 channels. These patches havent participated in training and validation.*

* 1. Train and validation subsets

To build a training and validation pipeline, the function BuildDS was implemented along with several supporting helper functions, all included in the load.py file. Later on, the function was extended to optionally load the test dataset as well – this will be discussed separately in !!!Test subset!!!

For efficient data handling and memory management, built-in TensorFlow utilities (referred to as tf in the following text and code) were used. Specifically, tf.Data.TextLineDataset() was used to read the .csv file line by line – each line representing the filename of a patch – which served as the foundation for the dataset pipeline.

The complete training subset originally contains 8400 patches. However, as noted earlier in !!!chapter!!!, only 5155 of them actually contain valid data and therefore used for training and validation. The dataset is shuffled and then split into training and validation subsets. The ration between the two can be configured as needed.

Using the .map methon, each text line is first expanded into five full paths: the corresponding red, green, blue, NIR and ground truth patch filenames. Each of these file paths is then replaced by ist actual image content, loaded as tensor. This transformation is implemented in the helper function loadDS, which itself calls another utility, loadTIF. At this stage, each dataset element is a tuple of TensorFlow tensors representing the input image and its corresponding ground truth.

The loadDS together with loadTIF function are loading each .TIF image and performing the necessary conversions:

* RGB and NIR patches, originally stored as 16-bit unsigned integers (range 0 - 65535), are cast to float32 and normalized to the range 0-1
* Ground truth masks, which originally contain values of either 0 or 255 (as uint8), are binarizes to values of 0 and 1, and also cast to float32.

An additional feature of the loadDS function allows optional resizing (downsampling) of the input images, if a target image size is provided. Importantly, the image loading and transormation pipeline ensures no information loss up to the point of resizing, where a reduction in resolution is intentional and controlled.

As final preparation steps, the dataset is: shuffled, batched, prefetched, and set to repeat indefinitely (cycled).

*In order to build training a validation pipeline the function buildDS was implemented together with the correspondent helper functions altogether provided in load.py file. Later on the optional loading af test dataset was builded in in the same function which will be further discussed in [Test subset]*

*For an effective data flow and for the sake of memory the TensorFlow (in further code snippets and mentionings reffered to as tf) built in utilites were take advantage of. With the help of a package function tf.data.TextLineDataset() the lines of .csv file – that are representing patches filenames – were taken as basis for future dataset pipeline. The full training subset size provided by default comprises to 8400 patches in total. However, as mentioned earlier in <>, only 5155 of them actually contain useful information for training and validation. The respective yet textline dataset is shuffled and splitted furthermore into test and validation parts. The ratio test/val can be varied. At this point each object of the dataset is a TensorFlow instance, utilizing the .map method (or feature) each textline is exanded first into 5 full paths to the corresponding red, green, blue, nir and ground truth patches. And then each replaced with tensor arrays representing loaded .TIF images. Additionaly in order to implement this the helper function loadDS together with loadTIF helper function were implemented. Feel free to investigate the functionality of them in the provided code. Important note:*

*While loading the RGB and NIR patches get ultimately casted from 0 to 65535 16 bit unsigned integer into 0 to 1 range float32. Whereat ground thruth mask originally containing only 0 or 255 unsignet 8 bit integer values gets casted into binary 0 or 1 format but represented in float32 too. Furthermore loadDS function posesses the possibility to resize (actually downsize) the image to load if needed and target image size provided. It has to be mentioned that it is guaranteed up until resizing (where its inevitable) there is no information loss while loading and transforming images. As final steps in preparation the dataset is shuffled, batched, prefetched and infinitely cycled.*

* 1. Test subset

The buildDS funciton additionally provides functionality to load and construct the test dataset used for model evaluation. There are two available modes for buildung the test pipeline:

1. Using all 9201 patches cropped from the 20 test scenes, or
2. Selecting patches from a single scene – either by specifying its sceneID or allowing the function to randomly select one – based on pre-generated .csv files that map patch names to sceneIDs.

After loading the filenames, the pipeline follows a similar flow to the training and validation datasets. However, ground truth masks are not available for individual test patches. Instead, each test dataset element is represented as a 384x384x4 (or predefined image size) float32 TensorFlow tensor containing the RGB and NIR channels.

Unlike training and validation datasets, the test dataset is not shuffled. This is necessary to preserve the spatial order of patches for later reconstruction. The dataset is batched and prefetched for efficient processing.

Since per-patch ground truth annotations are not provided, evaluation requires stitching the patches back into their original scene layout. This is done using the stitchPatches function, which reconstructs the entire scene by aligning the patches along their original positions. The function supports stitching either a single selected scene or all scenes in the test set.

*The function buildDS provides furthermore the functionlatiy to load and build testing subset for model evaluation. There are 2 possibilities to create the tensorflow tensor pipeline: either by considering all provided 20 test scenes cropped into 9201 patches or there is also an implemented feature which allows to either choose concrete scene using its ID or let the function choose a random sceneID and load it from premade .csv files containing patches corresponding to each sceneID. After choosing the mode of building test subset there are steps followed that are similar to building train and validation subsets. However, there is no ground truth mask present for each patch anymore. So each test dataset element is now represented as a 4 channel Tensorflow tensor. Dataset is not shuffled, as it is needed for further evaluation which will be explained in chapter !!!chapter!!!. Dataset is batched and prefetched.*

*Due to non existing/absence of ground truth masks for each individual patch but the presence of entire scene ground thruth masks in order to continue with evaluation process the patches need to be stitched together. The patches are stitched exactly at the border. The function stitchPatches has the ability to prceed either one given scene or the entire test dataset.*

2. Model architecture implementation

To facilitate flexible experimentation with different model architectures, the core building blocks and utility functions are organized in model.py. The CNN models are constructed using TensorFlow’s built-in APIs, allowing for modular and reusable design. To simplify architecture changes and streamline the process of constructing and loading different models, common patterns – such as convolutional blocks – are implemented as helper functions.

During development, it was necessary to test multiple model architectures and sizes. Therefore, the implementation supports fast and compatible switching between different models.

Additionally, where standard TensorFlow loss functions are insufficient, custom loss functions such as Soft Jaccard Loss are implemented withing this module. This also includes custom metrics, for example, the Dice Coefficient, which are particularly relevant for evaluating segmentation tasks.

By centralizing model definition and auxiliary functionality in model.py, the workflow supports rapid iteration and clear separation between architecture design and the rest of the implementation pipeline.

*In order to conveniently construct, load, change, and try out different model architectures several functions were implemented in model.py. The CNN model itself is constructed using provided TensorFlow utilites, helpers like whole convolutional blocks are implemented. If the model needs to utilize custom loss functions, whichh arent a part of standard TensorFlow loss functions, e.g. Soft Jaccard Loss, these are implemented here too. Along with custom metrics such as Dice Coefficient.*

*Multiple model architectures and sizes have to be tested out during working process, fast compatible switching from one to another wis supported.*

3. Converting the model

In order to convert the model after training to the Edge TPU format and to perform PTQ with calibration, several helper functions are defined in convert.py. As mentioned in [], the Edge TPU only supports models with a batch size of one. Howerver, for effective training, it is necessary to use and experiment with different batch sizes. Therefore, as the first step in the conversion process, the asBatchOne function transfers the trained weights to a new model with the exact same architecture but with an input batch size of one.

Next, using TensorFlow’s TFLiteConverter, the model is converted to the .tflite format, which is specifically desinged for compact deployment on mobile or edge devices. During this conversion, calibration <TODO explain in concept> is performed using a representative dataset, typically consisting of numerous patches from the training dataset. Furthermore, the inputs and outputs of the converted model are set to int8 for quantized inference.

As the final step, the model is compiled for Edge TPU inference using the provided edgetpu-compiler (link). After successful compilation, all ops (explain what it is) in the model are ideally converted into a single Edge TPU custon op ready for deployment. If the compiler cannot map a certain op to the Edge TPU, it will be mapped to the CPU instead, which can significantly increase inference time. The goal of this thesis is to meet all model requirements for Edge TPU [chapter with requirements] and ensure that all ops are mapped to it. The aspired final model conversion result is illustrated below.

TODO put netron.app with before and after compile. Describe it in text. Mention netron!

*In order to convert the model after training to edgeTPU format and to perform PTQ with calibration, several helpers were defined in convert.py. As mentioned in [] the edgeTPU supports only singlular batches, for the effective training, however, its necessary to be able to set and try out different batch sizes. Therefore as first step of conversion, in the asBatchOne function, the trained weights are transferred to a new model with the exact same architecture, but an input batch size of 1. Then, with help of TensorFlow TFLiteConverter the model is first converted in .tflite format, sophisticated for compact mobile or edge model deployment. During conversion, calibration <TODO explain it in concept> is performed with the help of representative dataset containing numerous patches of e.g. training dataset. Furthermore, inputs and outputs of converted model are set for int8 for inference. At final step the model is converted to edgeTPU inference ready format utilizing the provided edgetpu-compiler (link). Ater successful compilation all ops (exaplain what it is) in the model are converted into a single edgeTPU custom op ready for deployment. There is a deviation, if compiler can not map a certain op to edgeTPU it will be mapped to CPU, which can significantly increase inference time. The dedication of this thesis is to meet all model requirements for edgeTPU [chapter] to map all ops to it. The aspired final model conversion result is pictured below.*

*TODO put netron.app with before and after compile. Describe it in text.*

Background

1. State-of-the-art CNNs for Cloud Image Segmentation

CNNs are a class of neural networks that apply small, learnable filters – known as convolutional kernels – across an image to extract spatial features. A CNN typically consists of multiple convolutional layers stacked sequentionally. Each layer applies a set of filters that capture different visual patterns, such as edges or textures. As the network goes deeper, the spatial resolution of the image decreases, while the depth (i.e., 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 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 segmenation is the U-Net. Originally introduced by Ronneberger et al [\*\*\*] for biomedical image segmentation, U-Net has become a standard in many domains, including remote sensing and cloud detection. 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. The U-Net architecture shown in the diagram 1 was used by Mohajerani et al. [\*\*\*] for their cloud detection algorithm.

A key innovation in U-Net is the use of skip connections, which dierectly 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.

1. Google Coral Dev Board Mini and Edge TPU

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

The Edge TPU delivers up to 4 trillion operations per second (TOPS) 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.

To deploy a model on Edge TPU, it must be:

* Converted into the TensorFlow Lite (.tflite) format,
* And fully quantized to 8-bit integers.

These requirements directly affect model architecture, training strategy, and tooling. For instance, certain operations unsopported by Edge TPU must be avoided, and quantization aware training or post-training quantization must be considered early in the development process.

The following image from Coral documentation summarizes the model conversion and deployment workflow.

1. 38-Cloud Landsat 8 dataset

Landsat 8 is an Earth observation satellite launched on February 11, 2013, providing high-resolution multispectral imagery, including visible, near infrared (NIR), and thermal infrared bands. For cloud segmentation tasks, the red, green, blue (RGB) and NIR channels are particularly valueable due to their ability to capture both visual and athmospheric information.

This thesis utilizes a dataset consisting of 38 annotated satellite images from the Landsat 8 mission, commonly referred to as the 38-Cloud dataset. [\*\*\*]. The dataset has been introduced and adapted in the following scientific publications [\*\*\*]. Each image contains four spectral channels (RGB and NIR), along with a manually annotated dround truth mask that labels cloud pixels at the pixel level.

The dataset is divided into a training set containing 18 scenes and a test set with the remaining 20 scenes.

*Landsat 8 is a satellite mission launched in 2013 that provides high-resolution multispectral imagery, which includes visible, near-infrared, and thermal bands. [\*\*\*] For cloud segmentation, the red, green, blue (RGB) and near-infrared channels are paricularly useful.*

*In this thesis, a dataset consisting of 38 cloud-annotated images from the Landsat 8 mission is used. The dataset was presented and adjusted in the following scientific papers. Each image includes 4 channels – RGB and NIR – as well as the corresponding manually extracted pixel-level ground truth. The dataset is split into training part containing 18 sceneids and testing part containing respectively 20 sceneids.*

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 use certified space-grade components. However, design, production, and certification of such components are costly and time-consuming.

The AITHER project aims to investigate whether COTS (Commercial Off-The-Shelf) components can be effectively used in mano- 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, 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 Coral Dev Board Mini – a single-board computer developed by Google with an embedded Edge TPU designed for fast, low-power machine learning inference in constrained environments.

Alongside radiation shielding and thermal managements strategies, a technology demonstrator is being developed to assess the tolerance of these components to space radiation (e.g. protons, gamma rays). 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.

Thesis goal

AITHER project launched a 10kg nanosatellite OOV-CUBE on 9th of July 2024. Besides other hosted payloads not relevant for this thesis it is carrying Coral Dev Board Mini developed by Google – a single-board computer with an embedded Edge TPU module primarily made to provide fast machine learning inferencing in a small form factor. Radiation shielding and thermal diffusion were already taken into consideration by design of the satellite. The software for AI inference has now to be developed and uploaded to the board in orbit. As a software a CNN for cloud segmentation was chosen. Even though efficient CNN architectures are already existing and were studied in scientific papers, the embedded implementation of a complex deep learning algorithm has its own caveats. Due to restricted resources provided by an embedded system such as limitations in flash and RAM-memory, CPU and TPU capacity and speed, there are certain decisions to be made and the tricks applied in order to implement the CNN model in Coral Dev Board mini flawlessly. This is the main goal of this thesis

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 in the orbit. This thesis focuses on the implementation of a convolutional neural network (CNN) for cloud segmentation from satellite imagery. Although lightweight 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 tensor processing unit (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 Coral Dev Board Mini. 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.

**Chapter 1: Introduction**

Electronic components used in satellites and spacecraft are exposed to intense radiation, high-energy particles, and extreme temperature fluctuations. To ensure reliability in such harsh conditions, traditional space missions rely on certified space-grade hardware. However, this approach significantly increases development time and cost.

The AITHER project, launched by TU Berlin, aims to explore the feasibility of using commercial off-the-shelf (COTS) components in small satellites, particularly nanosatellites in the 10–100 kg range. As part of this initiative, the 10 kg nanosatellite OOV-CUBE was launched into orbit on July 9th, 2024. Among its hosted payloads is the Coral Dev Board Mini, a single-board computer equipped with an Edge TPU module designed for fast and efficient AI inference in embedded applications.

While the satellite’s hardware was designed with radiation shielding and thermal management in mind, the software for onboard inference still needs to be developed and uploaded. This thesis focuses on the implementation of a convolutional neural network (CNN) for cloud segmentation from satellite imagery. Although lightweight CNN architectures already exist and have been described in recent literature, porting such a model to an embedded platform introduces unique challenges. These include limitations in memory, computation, and model format compatibility.

The goal of this thesis is to design, train, and deploy a CNN model for cloud segmentation on the Coral Dev Board Mini. 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.

The scope of the thesis is limited to software-side implementation and validation. No real-time satellite communication is covered, and radiation tolerance is considered only in context, not experimentally. The input dataset is restricted to the publicly available Landsat 8 satellite imagery.

To achieve this goal, the thesis first reviews the relevant CNN architectures for cloud detection, selects an appropriate model, and trains it using available satellite data. The trained model is then converted and quantized for deployment on the Edge TPU. Finally, inference performance is evaluated directly on the Coral Dev Board Mini.

The structure of the thesis is as follows:  
Chapter 2 provides background on cloud segmentation and embedded deep learning.  
Chapter 3 outlines the implementation steps, from model selection to deployment.  
Chapter 4 presents the evaluation results.  
Chapter 5 concludes the work and discusses possible future directions.