Faculty of Science and Technology

BACHELOR'S THESIS

Study program/Specialization:	Spring semester, 2021
Bachelor of Science in Computer Science	Open / Restricted access

Author(s): Vegard Hovda Vistad, Ola Andrè Flotve, Håvard Moe Jacobsen

Faculty supervisor(s): Dr. Naeem Khademi

Thesis title:

Space Archaeology: CNN-Based Transfer Learning with Remote Sensing for Detecting Ancient Structures

Credits (ECTS): 20

Keywords:	Pages: 88
Qanat Fortress Remote Sensing Transfer Learning Deep Learning Convolutional Neural Network	Stavanger 15. mai 2021

"If we knew what it was we were doing, it would not be called research, would it?"

– Albert Einstein

Abstract

The preservation and discoveries of ancient structures is an integral part in the understanding of earlier civilisations. The ever increasing speed of urbanization has lead to the loss of crucial information from our distant past. As a consequence of this, the experimentation with faster and more automated methods to detect these structures has seen an increase.

In this thesis we explored one of these methods. We have used the architectures of high performing convolutional neural networks which have been pre-trained on the ImageNet dataset. These pre-trained models have then been fine-tuned to classify qanats or fortresses on satellite and airborne imagery of areas in the Middle East. In the end, the models classify smaller segments of an image, and label them if they are predicted to contain those ancient structures. We then present the results from the models which showcase their performance.

Acknowledgements

We would like to thank our supervisor Dr. Naeem Khademi for giving us the opportunity to work on this exciting and challenging thesis. His advice and feedback during the project has pushed us to improve our work at every step, and his genuine interest in the project has provided additional motivation during the semester.

Contents

1

Introduction

1.1 Motivation

Searching for archaeological structures is a slow and time consuming task. For many years, remote sensing has been used to look for ancient sites, but this is also time consuming, albeit to a lesser degree. With the recent successes of machine learning and neural networks, and the increasing availability of satellite imagery, it is becoming easier to combine these disciplines in order to speed up the process of locating ancient sites.

1.2 Problem Definition

The number of imaging satellites in orbit is ever-increasing, in turn increasing the availability of high resolution satellite imagery. Archaeologists has been using this technology for years to detect ancient structures, but the use of convolutional neural networks (CNNs) in this process is fairly new. CNNs require a large amount of training data, but the release of more satellite imagery reduces this problem and makes the CNNs more feasible as time goes by. The increased use of CNNs in the field of archaeology could be of great help in the search for ancient structures and result in new discoveries.

In recent years there has been a competitive effort to make deep learning models for image classification. This has led to many available models that are pre-trained on large datasets

such as ImageNet [1]. The potential of using these models in the detection of ancient structures creates some interesting opportunities.

There are many different types of ancient structures with different levels of complexity. Qanats are ancient water systems that have little variation in size and appearance, and may be a good starting point for testing deep learning models. Based on results, the limitations of the models may be explored by extending the research to more complex structures, such as fortresses.

1.3 Research Questions

1. Can pre-trained models with fine tuning be used in the detection of simple ancient structures like qanats?
2. Can the approach be extended to more complex structures like fortresses?
3. What types of terrain can the models be used on? Arid, semi-arid, agricultural, urban?
4. Can the easier implementation with image-classifiers compete with more advanced object detection methods when searching for qanats and fortresses?

1.4 Use Cases

This thesis aims to help in the detection of ancient structures, primarily qanats and fortresses. As mentioned, remote-sensing done manually is time consuming. The larger the geographical area is the more time it takes. Our approach aims to make it easier to search areas in a larger scale and faster than before. The improved speed of the process can help prevent the destruction of the ancient structures by urban sprawl. This could again lead to a better understanding about how ancient humans lived.

1.5 Challenges

In order to be able to detect archaeological features from satellite images, we need to have good enough imagery. Depending on the feature, images above a certain spatial resolution are needed. When we started looking into imagery from different satellites, we noticed none of the free alternatives were close to the spatial resolution used in Bing or Google maps.

Bing and Google have APIs [2][3] that provides high resolution satellite imagery. These APIs are however intended for use on web applications, and the terms of use do not allow permanent storing of the images. This would make it impossible to create a dataset for for training CNNs.

We were offered $5000km^2$ of satellite images from Planet [4], but we were not satisfied with the spatial resolution. We also had a meeting with Skywatch [5] who where very helpful, and gave us a better understanding on which kind of data that were available. They also offered us imagery at a cost, but we did not have the budget for this.

We then discovered Mapbox [6], who offer high resolution imagery and allow them to be used for non-profit and educational purposes. Their free-tier offers 750,000 tiles every month, with a size of 512x512 pixels and a selection of different spatial resolutions.

The dataset used for training may also end up not being 100% accurate, as we are not experts at identifying features such as qanats. When manually classifying images for training we use our best judgement.

1.6 Outline

Chapter 2 - Background:
The first section goes into the technical and theoretical knowledge needed to understand the paper. Section two presents important and relevant work that has been performed in this field.

Chapter 3 - Proposed Method:
This chapter outlines the approach we have taken in the detection of ancient structures.

Chapter 4 - Implementation:
This chapter explains the whole pipeline, from the acquisition of training data to the final prediction results.

Chapter 5 - Results:
This chapter shows the results from the pipeline which was presented in chapter 4.

Chapter 6 - Discussion:
This chapter discusses the results and the limitations of our methodology.

Chapter 7 - Conclusion:
This chapter gives a conclusion for the thesis and answers the questions in section (1.3), as well as proposing what future work can be done.

2

Background

In this chapter we present the technical and theoretical background needed to understand the research and results shown in this thesis, as well as some of the related work conducted in this field.

2.1 Technical and Theoretical Background

This section describes the characteristics of qanats and fortresses, as well as explaining different types satellite and airborne imagery. Lastly we go into some theory and concepts on deep learning and performance metrics that we used to evaluate the CNN models.

2.1.1 Qanats and Fortresses

Qanats are ancient systems of wells connected with underground tunnels for transporting water from aquifers in dry climates [7]. They range over large amount of areas were the smallest run for less than 5km, but the largest can be up to 70km long [7]. The tops of qanat shafts with all the features around it measures from around 5m up to 40m in diameter from our estimations on maps (appendix A). Figure 2.1 shows an image with qanats which aims to give a better understanding of its characteristics and looks.

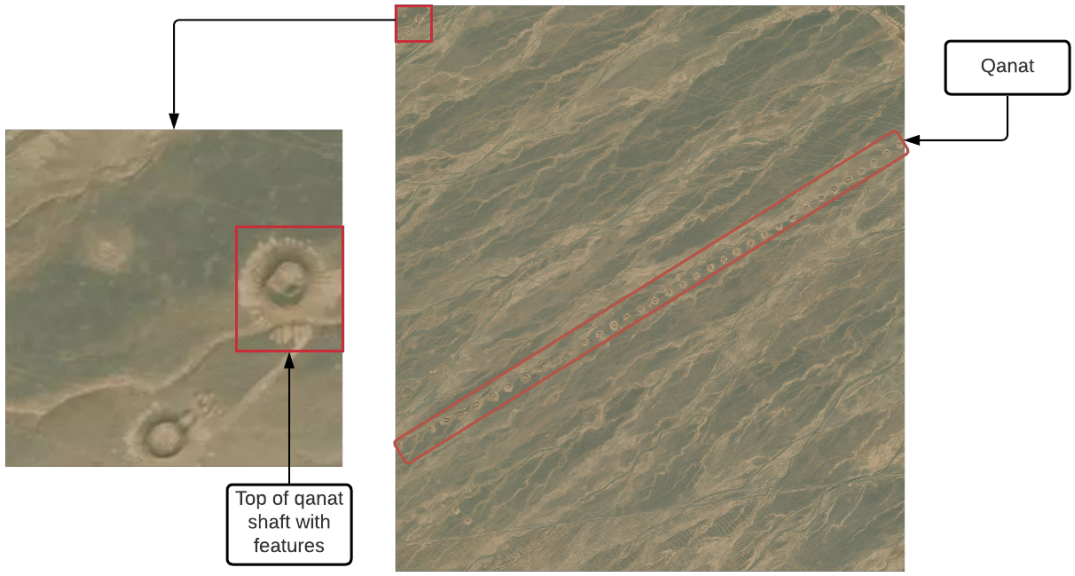


Figure 2.1: Tops of qanat shafts as seen in satellite imagery. The image to the left is a segment with a size of (256x256) pixels and the one to the right is (4608x5120) pixels.

Fortresses vary a lot in both size and shape. The smallest we have seen in Iran comes at a size of around 0.005 km^2 while the largest stretches up to 1,980 km^2. As mentioned the shape also changes from rectangular shapes to more circle like shapes. Figure 2.2 shows a satellite image of a fortress.

Figure 2.2: Fortress as seen in a satellite image marked with a red rectangle. The image size is (3584x2560) pixels.

2.1.2 Satellite and Airborne Imagery

To better understand satellite and airborne imagery it's an advantage to have an understanding of what remote-sensing is. According to the USGS remote-sensing is described as "the process of detecting and monitoring the physical characteristics of an area by measuring its reflected and emitted radiation at a distance" [8]. Airborne and satellite imagery are well known means of remote-sensing. Airborne imagery is produced by taking images from an aircraft or by airborne objects. Airborne imagery is an old practice that started in 1858 by a Frenchman by the name of Gaspard-Félix Tournachon, and were used in both the world wars [9]. Satellite imagery is taken by artificial satellites which are orbiting the earth from space. Through the ages the purpose of satellites have changed. The first applications of satellite imagery were mainly military, but this saw a change in 1972 [10]. After this, the production of earth observing satellites started. This lead to the possibility of new and interesting research to be conducted.

There are also a lot of different satellite image types which are taken by different type of sensors. Each sensor has its own characteristics that gives both benefits as well as some disadvantages. Some of the most well known sensors are the panchromatic, multispectral, LiDAR, and radar. The panchromatic sensor produces high resolution imagery in black and white color. Popular images produced by this sensor are the Corona images which offer resolutions from around 1.8-12 meters [11]. Next sensor in line is the multispectral. The special thing about this sensor is that its not just limited to the visible light, but can also offer different types of infrared as well as radar imagery. This offers the possibility to take images of the atmosphere, deep into the water, vegetation, man-made objects, soil, moisture, terrain, and more [12]. Some well known satellites that use these types of sensors are the Sentinel-2 [13] and satellites from the Landsat program [14]. The resolution available is typically larger than 10 meters. Moving on to LiDAR; this sensor uses reflected laser light, and can create 3-D representations of the earth's surface [15]. The last sensor we will look closer at is the radar. There have been created different type of these which are used for different purposes. One of the big advantages of radar is that it is cloud-penetrating and works in darkness. This makes it very useful in areas such as rain forests where there usually is a lot of cloud cover.

Satellite imagery is something that through the years have been rather exclusive, but as mentioned earlier the amount of new satellites being developed leads to more and more data being made public each year, but what types are available and at what resolution?

In terms of panchromatic imagery, images from the Corona satellites were released in the 1990's, and are available for free from USGS EarthExplorer [16]. Sentinel is another source of satellite imagery [17]. Here one can get everything from multispectral to SAR which is a type of radar that offers higher resolution than normal radar. The spatial resolution attainable varies depending on the area of interest, but its generally between high and low resolution which translates to around five meters resolution or lower. One can be lucky and find resolutions higher than five meters for free, but most of the time these are commercial. Unfortunately commercial satellite imagery often comes at a very high cost, but they can provide resolutions at up to 30cm as of the time this paper was written [18]. Again the resolution available increases all the time, so the commercial resolutions of today might be public in some years time.

When it comes to our research we used Mapbox imagery from their Raster Tiles API [19]. Mapbox gets their images from different kind of sources depending on the resolution and geographical location of the images. The high resolution images we used came from Maxar Technologies [20] which is a space technology company. These images are a combination of satellite and airborne imagery. We used a zoom level of 17 which translates to approx-

imately 0.46 meters/pixel, the images were created by visible light, and each tile had the dimensions of 512x512 pixels.

Figure 2.3: Tile from mapbox satellite that shows qanats with the zoom of 17 and the dimensions of 512x512 pixels.

2.1.3 Neural Networks

Artificial Neural Networks are inspired by the human brain [21], and are networks of nodes called artificial neurons connected together so that information can be passed between them. Each neuron sends a value to neurons connected to it, where this value is processed and passed on to the next neurons.

In Deep Neural Networks, neurons are grouped together in layers, where each neuron is connected to any number of neurons in the next layer. "Deep" refers to the number of layers. Data is first sent to an input layer, and propagates through intermediate layers, called hidden layers, before ending up at the output layer. Connections between neurons have weights associated with them. The output of each neuron is determined by an activation function, whose input is the weighted sum of the outputs of connected neurons in the previous layer. The activation function may require the input to exceed a threshold

before "activating" the neuron and passing the value to the next layer.

During training, a network learns through a process called backpropagation: The values of the output layer is compared to the true values in order to quantify the error. This is then used to adjust the weights in the previous layers, in the hopes of reducing the error of future outputs.

2.1.4 Convolutional Neural Networks

A convolutional neural network is a type of deep neural network often used for computer vision due to their ability to detect features and patterns in images [22]. This is achieved through special layers called convolution layers, which apply filters to an image using kernel convolutions. These convolutions can detect edges and more complex patterns that enable the network to detect objects within an image.

For image classification, the number of neurons in the output layer corresponds to the number of image classes the network has been trained to classify. This layer is fully connected, meaning that each neuron is connected to all neurons in the previous layer. The value of the neurons in the output layer is the probability of an input image belonging to that class.

2.1.5 Transfer Learning

Transfer learning is when you use knowledge learned from an earlier task to solve a new one. In the case of machine learning, this implies that you use a pre-trained model which is already trained on a large dataset, and further train it on a new dataset. When using an already pre-trained model, the dataset does not need to be as large as when training a model from scratch. Before predicting on data, the model will need to be fine-tuned. First a new output layer needs to be defined which matches the classes to be classified. Layers can be frozen on the input-end of the model whilst keeping some layers near the output layer unfrozen, meaning they will be trained. After a CNN is trained, the first layers tend to enhance features such as edges or circles in an image, whereas the later layers can detect more advanced patterns such as faces or items [23]. If using a pre-trained model, there are little or no positive effect on training the first layers which is already trained to detect basic features, it just increases the training time [24].

ImageNet is one of the more common large datasets which pre-trained CNNs are trained on. It contains more than 14 million images with thousands different classes and sub-classes which are classified by the use of Amazons "Mechanical Turk" crowdsourcing service [25][26]. Some of TensorFlow's models which is already trained on ImageNet are available for download, then could be used for transfer-learning [27].

2.1.6 Classification

When training a model on a dataset, each datapoint in the dataset has to be labeled with a label corresponding to one of the classes.

Binary Classification

Binary classification is when you want to classify your data into two categories. This could be anything from classification of images belonging to a specific class, to medical testing where you determine whether a patient has a disease or not. The output layer should in this case return two values: the probability of the datapoint belonging in the first, and second class.

Image Classification

Image classification could either be a binary or multi-label classification. In this thesis we are looking for a specific feature in our images, meaning we have one label for images where the feature is present, and one label for when it is not. As mentioned earlier, ImageNet has thousands of labels, where each label is mapped to if a specific feature or object is present in the image.

2.1.7 Performance Metrics

The evaluation of a model is important, and to get an idea about it's performance we need some metrics that we can analyse. Although there are a lot of different metrics many of them use the same underlying variables which needs to be defined.

True positives (TP):
These are the values correctly identified as true.

True negatives (TN):
These are the values correctly identified as false.

False positives (FP):
These are the values falsely identified as true.

False negatives (FN):
These are the values falsely identified as false.

Confusion Matrix

A confusion matrix is a way of plotting the actual values vs the predicted values. Confusion matrices can be created for both multi-class classification and binary classification.

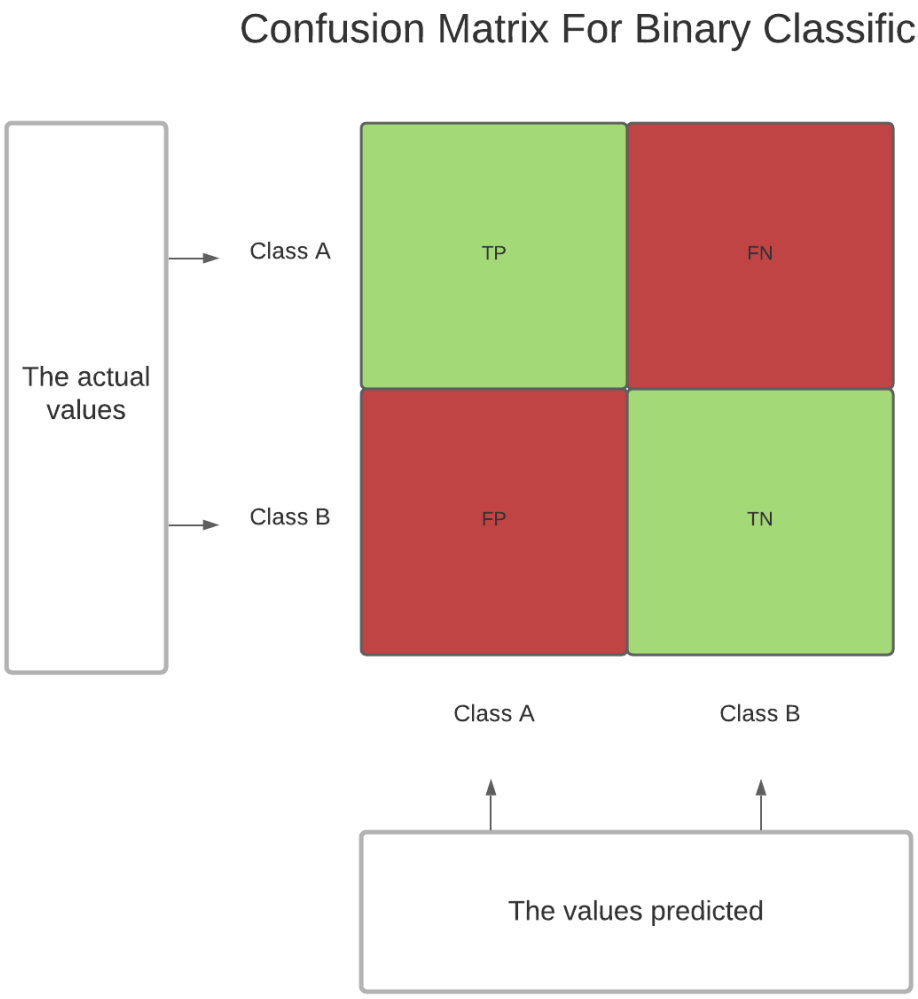


Figure 2.4: How confusion matrices works for binary classification.

Figure 2.4 shows how a confusion matrix for binary classification is set up. There are two classes and these are represented through the positive (class A) and the negative (class B) [28]. The values TP, FP, FN, and TN which are defined above and represented in the confusion matrix are central in the calculation of the metrics discussed below.

Accuracy

Accuracy is defined by the number of correctly given predictions divided by all the predictions performed. It gives us a way to see if a model was trained in the correct way. The disadvantage with this metric is that it does not perform well when you have a large class imbalance [28]. The formula for calculating the accuracy is given by the equation below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

Precision

The precision metric produces information about how often the model is correct when it predicts true positive. It is valuable when the costs of false positives are high [28]. The formula for calculating the precision is given by the equation below:

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

Recall

Recall has its base in what the values actually are. If you have two classes A and B. Recall will then answer the following: Out of all the values in class A how many of them did the model get right? The formula for calculating this is given by the equation below:

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

F1-Score

The F1-score is a combination of the precision and recall metrics. It gives a score on how many false positives and false negatives that are given by the model [28]. The score is from 0-1 and the lower FP and FN, the higher the score. The formula for calculating the F1-score is given by the equation below:

$$F1\ Score = 2 * \frac{Recall * Precision}{Recall + Precision} \tag{2.4}$$

2.1.8 CNN Architectures

ResNet

The motivation behind residual neural networks (ResNet) [29] is that increasing the depth (number of layers) of a network should improve models, but that after a certain point the increased depth leads to higher error in training and testing for normal networks that just stack layers [30].

ResNet handles this by introducing skip-connections, connections that jump over one or more layers; instead of the output of a layer just being the input of the next, some outputs are now also sent to deeper layers. The sum of this output with the "regular" input of the deeper layer is now its new input. This makes it possible to successfully train much deeper networks. Models with over 100 layers have produced good results, and even ones with over 1000 layers can give good training accuracy, although such extremely deep models may suffer from overfitting [30].

DenseNet

DenseNet [31] can be viewed as a more extreme version of ResNet, with some changes. Shortcut connections are also used, but in this case the output of each layer connects to all subsequent layers. These outputs are concatenated onto the rest of the input, instead of being summed. In other words; the input of each layer is the output of every preceding layer.

EfficientNet

Scaling CNNs can be achieved by increasing their depth, width or input resolution, but finding a good combination of these different dimensions can be a tedious process of trial and error [32]. The EfficientNet architecture [33], developed by Mingxing Tan and Quoc V. Le at Google AI, implements a compound coefficient that can be used scale these dimensions by a constant ratio with good results [32].

2.1.9 Geographic Information Systems

Geographic Information System also known as GIS is a framework used for gathering, managing, and analyzing data. Through geography GIS helps with the integration of many different types of data [34]. It makes it possible to organize data into layers, so that they can be visualized into the likes of maps and 3D-scenes.

Figure 2.5: Example from QGIS with geoJSON layers added on a Mapbox satellite map.

As one can see this makes it easier to recognise patterns , and can increase the possibility of new findings by looking at the whole picture and not just the raw data.

2.2 Related Work

There are many approaches to detecting ancient structures in satellite imagery. Aside from manual image analysis done by humans, some methods use image processing techniques to highlight specific shapes and features, such as circular and rectangular shapes that do not normally appear in nature. Machine learning techniques, for example convolutional neural networks, have also been successfully used for detecting archaeological structures. This section reviews some previous work using these these types methods.

2.2.1 Remote Sensing for Finding Archaeological Structures

Luo et al. [35] employed Circle Hough Transform (CHT) along with Canny Edge Detector (CED) and Mathematical Morphological Processing (MMP) for extracting circular qanat shaft features from two sites in northern Xinjiang, China. Very high resolution (VHR) images from Google Earth were used as data, more specifically optical RGB imagery from IKONOS and GeoEye satellites in different seasons. This method resulted in the detection of 95.9% of manually identified qanat shafts in site 1, and 79.5% in site 2. Site 2 had a larger number of shafts in total, as well a number of false positives, with no false positives in site 1. Detection using CHT alone gave a detection accuracy of 86.3% and 65.8% in sites 1 and 2 respectively. The combination of CHT with CED and MMP showed large improvement over just using CHT.

2.2.2 Types of Machine-Learning in Remote Sensing

Research done by Stott et al. [36] utilized machine learning with airborne laser scanning to address a "needle-in a-haystack" problem. One of the primary goals was to search for Viking ring fortresses throughout Denmark. Computer vision techniques were applied to a Digital Terrain Model (DTM) derived from the national Airborne Laser Scanning (ALS). Using a residual relief convolution, as well as segmentation into positive and negative elevation trends, the group tested ring detection. With the features geometric properties the results were refined. Using ensemble machine learning, the features were classified with the help of their topographic and cultural context. The outcome from this was the identification of 199 candidate features, resulting in nine correctly detected fortresses, including four of the five extant Trelleborg-type ring fortresses. Two of the most promising candidates for undetected ring fortresses were those at Trælbanken and on Borgø. Fortresses from the early Iron Age at Trælborg near Skandaberg, the Medieval fortress at Borrebanke (Lolland), and 19th Century Scone at Havmøllen (Djursland) were all detected. The research shows that with the help of machine learning it is possible to do large-scale searches for features even at a national scale.

Orengo et al. [37] conducted research on mound detection by the use of multisensor multitemporal satellite images from Sentinel 1 and 2. Data from the Sentinel satellites have a resolution of 60-10m. Their area of interest was the Cholistan Desert in Pakistan where they were looking for Indus civilization sites. They wrote a script in Javascript which collected satellite data from Google Earth Engine (GEE). The script used a Random Forest classifier which produced an overlay over the map show in GEE. They detected hundreds

of new sites, deeper in the desert than previously expected with signs of relatively frequent settlement shifts, which they guessed was a cause of climate change and hydrological network change.

Guyot et al. were looking for burial mounds in Carnac, the Bay of Quiberon, and the Gulf of Morbihan in France [38]. Their study suggests that the commonly used method with different Visualisation techniques together with Machine Learning (ML) classifications would not work well in a case where you are looking for heterogeneous structures which are different in size. Which commonly happens when archeological remains get altered over the years. They tried to combat this restriction by the use of "multi-scale analysis of topographic position with supervised machine learning". They mapped each layer of a Multi Scale Topographic Position (MSTP) image to a different scale: micro, meso, and macro. This created an image where each layer highlighted mound features from every scale. This combined with the use of a Random Forest classifier gave some good results.

2.2.3 Deep Learning

The paper by Caspari and Crespo [39] explores the use of CNNs to detect burial mounds in optical satellite data over northern Xinjiang, China, and they compare the results to a biased random guess and two support vector machines (SVMs), which are supervised learning models. Results show that the CNN model performs much better on images with burial mounds, with an F1 score of 0.91 compared to 0.20 and 0.71 for linear kernel and radial basis function (RBF) kernel SVMs respectively. On images with no burial mounds, the CNN is still better, but the scores are much closer: 0.99 (CNN) compared to 0.94 (linear kernel) and 0.97 (RBF kernel). Burial mounds have a very distinctive shape, making them an ideal target for detection with machine learning algorithms.

Soroush et al. [40] examined the use of convolutional neural networks on satellite imagery to automatically detect qanat shafts in the Kurdistan Region of Iraq. The data used was declassified Corona satellite imagery (panchromatic photographs), lower in resolution than more modern satellite imagery. The CNN model used binary classification for feature segmentation of qanat shafts. The authors found that the model performed better in patches with a high density of qanat shafts. Five patches of high density gave an F1 score, or harmonic mean of precision and recall, of 0.705, while six patches of low density gave an F1 score of 0.413. In conclusion, the authors argue that deep learning tools should be embraced with the use of remote sensing in archaeology.

The research conducted by Verschoof-van der Vaart and Lambers [41] presents a new type

of CNN called Regions-based CNN (R-CNN). This is an automated technique that has the possibility to detect multiple classes of archeological objects in LiDAR data. The targets were barrows, Celtic fields and charcoal kilns. From the results of the experiments, the group discovered that the Faster R-CNN was not able to detect charcoal kilns. However results were obtained for the barrows and celtic fields. For the recall values, barrows got a score of 0.62-0.81 (on average 0.73) and Celtic fields got 0.19–0.97 (on average 0.60). The precision values were 0.36-0.90 (avg 0.64) for barrows and 0.26-0.71 (avg 0.46) for Celtic fields. Finally the F1-scores for barrows and Celtic fields were respectively 0.49 and 0.79 (on average 0.67) and 0.29 and 0.68 (on average 0.43). From the F1-scores one can conclude that the model has better performance on barrows than celtic fields. From comparison with other techniques for detecting barrows, the one from this paper performed very well. Therefore results from this study by Verschoof-van der Vaart and Lambers with the new CNN type called R-CNN shows promising steps in the technique of detecting multiple classes of archaeological objects in LiDAR data.

3

Proposed Method

We propose a relatively simple approach to object detection that uses an image classification CNN on small segments of a larger image. Training and test data will be gathered from Mapbox's Raster Tiles API [19].

3.1 Acquisition of Training-Data

Mapbox's Raster Tiles API has a free tier that provides up to 750,000 raster tiles per month for non-profit and educational use [19]. Bounding box coordinates for areas with areas of interest (AOI).

will be used to fetch satellite image tiles with a usable spatial resolution. The tiles will then be split into smaller segments, which can then be manually classified into two classes; images containing or not containing the relevant structure. This will be done until a suitable amount of training data has been collected and classified.

3.2 Transfer Learning with ImageNet Weights

We will use transfer learning with three different CNNs that are pre-trained on the ImageNet dataset, and compare their performance at classifying ancient structures. The Keras library [42] provides several CNNs architectures with weights pre-trained on ImageNet.

3.3 Detection of Archaeological Structures

The images to be analyzed will be split into smaller segments using the same process as for training-data acquisition. These smaller images can then be classified by the trained model. A bounding box is drawn around the segment if it is classified as containing a relevant structure. The result of this is a form of "low-resolution" object detection.

The advantage of this approach is that simpler image-classification CNNs can be used, instead of more complicated object detection networks. The downside is that structures that are small and close together (such as qanats) will not be marked individually, but as a group contained within each box.

4

Implementation

This chapter describes the acquisition of training and testing data, our methodology for fine-tuning pre-trained CNN models and evaluating them. All code is implemented in Python 3.7.10 [43].

4.1 Acquisition of Training Data

This section describes the process of fetching and segmenting images for training and testing.

4.1.1 Fetch from Mapbox

The acquisition of the data takes place through the ImageFetcher class. This class takes an argument called class_type that specifies what type of structure the images are intended for, such as qanat or fortress.

The method responsible for fetching the images is called fetch_images, and it receives five arguments. The first is the Mapbox API key which makes it possible to fetch the images from Mapbox Raster Tiles API [19]. The second and third arguments is the top left (TL) and bottom right (BR) coordinates for the corners of the AOI. The fourth is the zoom

level, which determines the spatial resolution of the fetched tiles. The last arguments are location name and segment size. Location name is a name for the AOI, such as the country. Segment size will be discussed in the next subsection.

The first step is the calculation of the x and y range of map tiles. This is done through the mercantile library [44] which is created for working with tiles. After this there will be initiated a nested loop were all the tiles inside the AOI will be fetched. There will also be created a tileset-folder which includes a composite image of the fetched tiles. Each tileset is given an integer id, which will be incremented each time a new set is created. Finally, a metadata file is created which contains information about the TL and BR coordinates, zoom level, class type, and segment size.

Code 4.1: Calculation of x and y range of tiles, and Mapbox tile requests inside the nested loop. (imfetch.py).

```
    def fetch_images(self, mapbox_key, tl, br, z, location_name, ...
        seg_size):
        x_tile_range, y_tile_range, n = self.calculate_x_y_range(tl, br,...
            z)
        self.num_x_tiles = x_tile_range[1] - x_tile_range[0] + 1
        self.num_y_tiles = y_tile_range[1] - y_tile_range[0] + 1
```

...

```
                for j, y in enumerate(range(y_tile_range[0], ...
                    y_tile_range[1] + 1)):
                    r = requests.get('https://api.mapbox.com/v4/mapbox....
                        satellite/' +
                                     str(z) + '/' + str(x) + '/' + str(y)...
                                         + '@2x.png?access_token=' + str(...
                                        mapbox_key), stream=True)
                    if r.status_code == 200:
                        create_folder(path, folder_index)
                        create_folder(os.path.join(path, str(...
                            folder_index)), 'tiles')
                        create_folder(os.path.join(path, str(...
                            folder_index)), 'stats')

                        with open(str(os.path.join(path, str(...
                            folder_index), 'tiles', str(i) + '.' + str(j)...
                             + '.png')), 'wb') as f:
                            r.raw.decode_content = True
                            shutil.copyfileobj(r.raw, f)
```

After fetching, the tiles are combined into a composite image with the combine_tiles() method. This composite can then be used for testing trained models, and makes manual classification easier. The tiles are deleted afterwards to save space, as they are no longer needed.

A total of 26 tilesets of different sizes were fetched for training and testing on qanats. 17 were used for training, and the remaining 9 for testing afterwards. 46 tilesets were fetched for training and testing on fortresses. 37 were used for training, and 9 for testing. A list of coordinates for TL and BR of all tilesets can be found in A.2.

4.1.2 Segmenting and Classifying Images

Pre-trained CNN models are trained on input images of a certain size; ResNet-152 [45] expects images of size 224x224, while DenseNet-201 [46] and EfficientNet-B3 [47] expect 256x256. The fetched tiles were 512x512 pixels, so we decided to split the composite images into segments of 256x256 pixels, and rescale them to 224x224 pixels when using them with the ResNet model.

Before splitting the composite, the segments have to be manually classified. To achieve this, we display the composite using Matplotlib [48] and overlay a grid, with grid cells corresponding to the segment size. This happens through the ImageViewer class.

The constructor for the ImageViewer class takes class_type and tileset_id as arguments, and loads the segment size stored by the ImageFetcher class. It also looks for a json file, containing the classification for each segment, to load into a Python dictionary. If no such file exists, it creates a new dictionary with a key corresponding to each segment, and default value as 0 (0 meaning the segment does not contain the relevant structure). The segments are assigned an integer ID that increments from left to right, row by row, starting at the top row. These IDs are combined with the tileset ID to form the dictionary key.

Code 4.2: The ImageViewer Class

```
class ImageViewer:
    def __init__(self, class_type, tileset_id):
        assert valid_type(class_type)
        self.images_path = join(dirname(dirname(abspath(__file__))), '...
            images', class_type, str(tileset_id))
        self.class_type = class_type
        self.tileset_id = tileset_id
        self.composite_img_path = join(self.images_path, "composite.png"...
            )
        self.metadata_path = join(self.images_path, 'metadata.json')
        self.classification_path = join(self.images_path, '...
            classification.json')
        self.classification = None
        self.load_data()
```

Using an event listener, the user can click on a cell to label it, changing the corresponding dictionary value to 1 and adding a fill color to the cell. Clicking again reverses this. When exiting, the dictionary is saved to a JSON file.

Images used for training are then added to a dataset folder using ImageViewer's add_to_dataset() method. This method splits the composite image using the split_image() method of the ImageSplitter class. This method takes segment size as an argument, and splits the composite image into segments accordingly. The segments are put into a list, indexed in the same order as the segment IDs were assigned in the ImageViewer class. The method then returns the list. The add_to_dataset() method then loops through this list, comparing each segment with the classification dictionary, and sorts them into two folders according to their classifications.

Code 4.3: The split_image method splits an image into segments, and returns the segments as a list along with a list of their IDs.

```
    def split_image(self, tileset_id, image_path=None, size=256, ...
        target_size=None):
        tileset_id = str(tileset_id)
        if not self.safe_to_split(tileset_id):
            return

        segments = []
        seg_names = []

        path = join(self.image_dir, tileset_id)
```

...

```
            if image_path == None:
                image_path = join(path, 'composite.png')
            img = cv.imread(image_path)
            i = 0
            for x in range(img.shape[0]//size):
                for y in range(img.shape[1]//size):
                    segment = img[x*size:x*size+size, y*size:y*size+size]
                    filename = f'{tileset_id}.{i}.png'
                    segment_path = os.path.join(self.image_dir, tileset_id, ...
                        'segments', filename)
                    segment = cv.cvtColor(segment, cv.COLOR_BGR2RGB)
                    seg_size = [segment.shape[0], segment.shape[1]]
                    if target_size != None and seg_size != target_size:
                        segment = cv.resize(segment, target_size)
                    seg_name = f'{tileset_id}.{i}'
                    segments.append(segment)
                    seg_names.append(seg_name)
                    i += 1
            return segments, seg_names
```

4.2 Training the Models

The training notebook consists of six sections. These sections represent different steps in the path to getting a trained model.

4.2.1 Kaggle as a GPU platform

When training a deep learning model, it is an advantage to have a good GPU, because a GPU can process multiple computations simultaneously. Before starting the training process we had to decide which GPU's we were going to use. There were three options: The first option was to train locally on our personal computers, the second option was to borrow GPU's from the University of Stavanger, and the third option was to use Kaggle's GPU's [49]. We came to the conclusion that our personal computers did not have enough power to perform efficient training. The other two options were a little harder to decide on. The GPU's of the university were probably the most powerful option, but with Kaggle's user friendliness as well as it satisfying our power needs we decided to go with Kaggle.

Kaggle is a machine learning and data science community [49]. It is owned by google and offers a Jupyter notebook [50] environment which requires no special setup. Here one can create models, upload data, and access data from the community. They also offer up to 36 hours a week of free GPU access, and frequently host competitions.

4.2.2 Initial Parameters

This section describes the most important parameters and variables which will be used throughout the notebook.

The first variable will select the pre-trained model to be used. When the model is selected, its required parameters will automatically be loaded. Next in line will be to choose the classes and what the ratio should be. The ratio is the percentage of the data set that will be used for training, validation, and testing.

When training a model, a common problem is the lack of training data. A simple way to alleviate this problem is through augmentation. The amount of augmentation desired can be specified in this section. Here the typical options like rotating the images or flipping them is available in addition to other types of augmentation.

Neural networks consists of different layers and when using a pre-trained model it can be beneficial to freeze some of these layers. The freezing of a layer will stop the weights in this layer from changing and it can also decrees the time needed for training the model. The number of frozen layers can be set with the frozen_layers parameter.

The next parameters to be set are the hyperparameters. These are used to control the training, and the ones available in this section are the batch size, the learning rate, and the number of epochs.

Code 4.4: Initial parameters from train.ipynb

```
pre_tr_mdl = 'densenet201'
class_names = ['no_qanat', 'qanat']
dataset_path = custom_path
ratio = (.8, .1, .1)

rotation_range=360
channel_shift_range=0.15
horizontal_flip=True
vertical_flip=True

frozen_layers = 0
batch_size = 10
lr = 0.0001
epochs = 20
save = True
```

4.2.3 Initialisation of functions

The next section is used to initialise the functions. One of the most important functions is the one used for saving the model and the results. This makes it possible to use the trained model later for predictions, and to compare the results with different trained models. Another essential function combined with a dictionary makes it possible to easily change between the CNN architecture that is desired, and there are currently eight architectures available.

Code 4.5: Model saver function from train.ipynb

```
def model_saver(model, info_dict, pre_tr_model, dest_path, dataset_size,...
     history, cm, class_names):


    prec = info_dict['precision']
    rec = info_dict['recall']
    temp_folder_name = './p{:.4f}r{:.4f}model'.format(prec,rec)

    if not os.path.exists(temp_folder_name):
        os.makedirs(temp_folder_name)

    model.save(temp_folder_name)

    save_metadata(temp_folder_name,pre_tr_model,dataset_size, info_dict)

    plot_history(history, save = True, temp_folder = temp_folder_name)


    plot_confusion_matrix(cm=cm, classes=class_names, title="Confusion ...
        Matrix",save = True, temp_folder = temp_folder_name)

    zipf = ZipFile('trained_model.zip', 'w', ZIP_DEFLATED)

    for root, dirs, files in os.walk(temp_folder_name):
        for file in files:
            zipf.write(os.path.join(root, file))

    zipf.close()

    path = os.path.join(dest_path, str(pre_tr_model))
    if not os.path.exists(path):
        os.makedirs(path)

    ID = get_index(os.path.join(dest_path, str(pre_tr_model)))

    shutil.move('./trained_model.zip',os.path.join(os.path.join(...
        dest_path, str(pre_tr_model)),'{}p{}r{}model.zip'.format(ID,prec,...
        rec)))

    dir_path = temp_folder_name
    try:
        shutil.rmtree(dir_path)
    except OSError as e:
        print("Error: %s : %s" % (dir_path, e.strerror))
```

4.2.4 Model selection and pre-processing

At the start of this section the model to be used gets selected. This will result in the selection of the required and the best suited parameters for the model. It includes the pre-processing function and the target size. The augmentation, the batch size, and the number of epochs will also be added here, and all of this is done with the help of the ImageDataGenerator [51].

Code 4.6: Code showing the data generators an the inputs used. Taken from the train.ipynb

```
train_batches = ImageDataGenerator(
    brightness_range = brightness_range,
    rotation_range=rotation_range,
    channel_shift_range=channel_shift_range,
    horizontal_flip=horizontal_flip,
    vertical_flip=vertical_flip,
    preprocessing_function=ppf) \
    .flow_from_directory(directory=train_path, target_size=target_size, ...
        classes=class_names, batch_size=batch_size)

val_batches = ImageDataGenerator(
    brightness_range = brightness_range,
    rotation_range=rotation_range,
    channel_shift_range=channel_shift_range,
    horizontal_flip=horizontal_flip,
    vertical_flip=vertical_flip,
    preprocessing_function=ppf) \
    .flow_from_directory(directory=val_path, target_size=target_size, ...
        classes=class_names, batch_size=batch_size)

test_batches = ImageDataGenerator(preprocessing_function=ppf) \
    .flow_from_directory(directory=test_path, target_size=target_size, ...
        classes=class_names, batch_size=batch_size, shuffle=False)
```

4.2.5 Build and compile

The fourth section is were the model gets built and compiled.

Keras offers the opportunity to download some well known CNN architectures with weights from widely used data sets such as ImageNet. The parameter from the first section were one selects the desired model, will here be used to download the architecture, and the

weights offered by Keras. In our paper we decided to use DenseNet-201, ResNet-152, and EfficientNet-B3 with weights from the ImageNet data set.

Code 4.7: The code used for downloading a model. Taken from the train.ipynb

```
model_val_info =  load_model_dict[pre_tr_mdl]

base_model = mod_info['model_func'](
                    include_top=model_val_info['include_top'],
                    weights=model_val_info['weights']
                    )
print(str(mod_info['model_func'])+' has been loaded!')
```

The next part is the compiling of the model. Here the learning rate, the optimizer, the loss, and the metrics will be inserted.

Code 4.8: Code for compiling the model. Taken from the train.ipynb

```
model.compile(optimizer=Adam(learning_rate=lr), loss='...
    categorical_crossentropy', metrics=['accuracy'])
print('Done')
```

4.2.6 Training

This is were the training of the model takes place. By specifying the number of epochs its decided how many times the model should go through the training data. Its also where the training and validation data is inserted into the model.

Code 4.9: The code used for the training of the model. Taken from the train.ipynb

```
history = model.fit(x=train_batches,
          steps_per_epoch=len(train_batches),
          validation_data=val_batches,
          epochs=epochs,
          verbose=1
)
```

4.2.7 Test and Evaluation

The last section is were the model is tested and evaluated. From training, a graph with accuracy and validation accuracy is created. The testing is done with keras model.predict() [52] and the results are visualized with a confusion matrix [53]. The test performance is also evaluated with metrics called precision, recall, and F1 score.

Code 4.10: The code used for the training of the model. Taken from the train.ipynb

```
predictions = model.predict(x=test_batches, verbose=0)
np.round(predictions)

plot_history(history)

cm = confusion_matrix(y_true=test_batches.classes, y_pred=np.argmax(...
    predictions, axis=-1), labels=[1, 0])

plot_confusion_matrix(cm=cm, classes=class_names, title="Confusion ...
    Matrix")

evaluation = get_model_eval(model, cm, frozen_layers, epochs)
```

4.3 Predictions

At the start of the prediction script we have defined two dictionaries, one for selection of pre-processing function (PPF) to use on the data before doing predictions on it, the other one is for which image dimensions the model's input-layer expects. Which PPF is meant for which model is found at TensorFlow's webpage, for instance ResNet's PPF is fetched via "tensorflow.keras.applications.resnet.preprocess_input" [54].

Code 4.11: Dictionaries used in predict.py

```
PPF = {
    "densenet": tf.keras.applications.densenet.preprocess_input,
    "densenet121": tf.keras.applications.densenet.preprocess_input,
    ...
    "resnet": tf.keras.applications.resnet.preprocess_input,
    "resnet50": tf.keras.applications.resnet50.preprocess_input,
    ...
}
INPUT_SIZE = {
    "densenet": (256, 256),
    "densenet121": (256, 256),
    ...
    "resnet":  (224, 224),
    "resnet50":  (224, 224),
    ...
}
```

4.3.1 Input Arguments

Code 4.12: Input arguments to predict() from predict.py.

```
1 def predict(folder_id, class_type, model_path=None, target_class=1, ppf=...
      None, threshold=0.5):
```

There are two necessary input arguments for the function to run: folder_id and class_type. Of the two main parameters, class_type is the type of image we are going to predict on (i.e. qanat or fortress) and folder_id is the id of the tileset. Check out our github for the full folder structure (A.1). Of the non-necessary input arguments, there are the file path to which model to use, target class and PPF. By calling the function without a file path to the model, you are prompted with a tkinter filedialog window where you choose the path to the model you want to use [55]. If you call the function without a PPF as input argument, a PPF is chosen by using the model name fetched from the path as a key for the PPF dictionary. The target_class is where you choose which class to look for, where we chose 1 as an ID for a the positive label "qanat" and 0 as an ID for "no qanat". The threshold variable is 0.5 by default and is used as a value of how sure the model has to be in order for the model to classify the segment as the target class.

4.3.2 The Prediction Process

After the initialization of variables, a model is loaded by calling tensorflow.keras.models.load_model(path) [56] with the desired path. There is created an instance of the ImageSplitter class from which we call the "split_image()" function that returns a list of segmented images as well as a list with the image names. We iterate through those lists, use the the PPF on the segments and use the "model.predict(segments)" to get an array of results. Each prediction in the results-array consist of two values: the predicted probability for class-0 & class-1. The threshold together with target-class number, tells us which of the classes should be the final decision. The function returns a dictionary where the segment ID is the key which has the predicted class as value, as well as the name of the model used for the predicting.

4.4 GUI

A graphical user interface was made using Tkinter [55] to automate the pipeline for fetching and segmenting images, as well as classifying and predicting, thus making it easier to work with larger amounts of data. All code discussed in this chapter, except for the training notebook, is tied together through this GUI. Prediction results can be reviewed and saved, along with a geojson file [57] with coordinates for predicted segments.

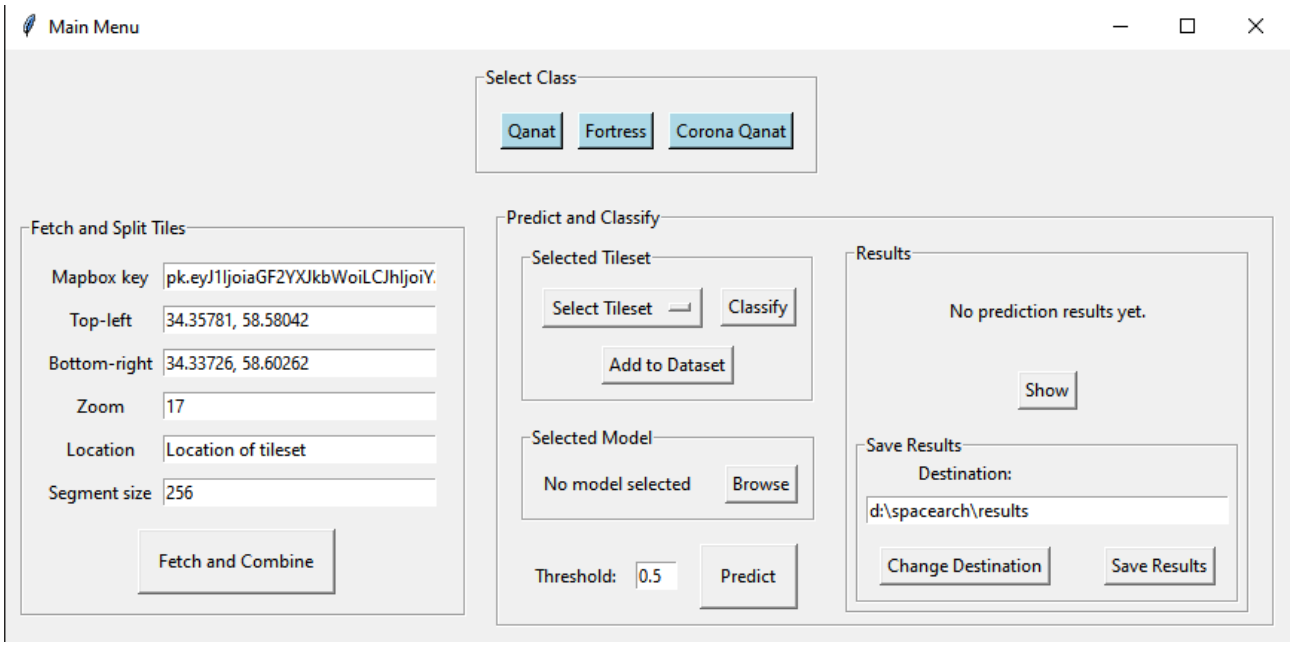


Figure 4.1: GUI
GUI from main.py

4.5 Finished Pipeline

Figure 4.2 illustrates the pipeline from the start were one fetches the images to the end with the final results.

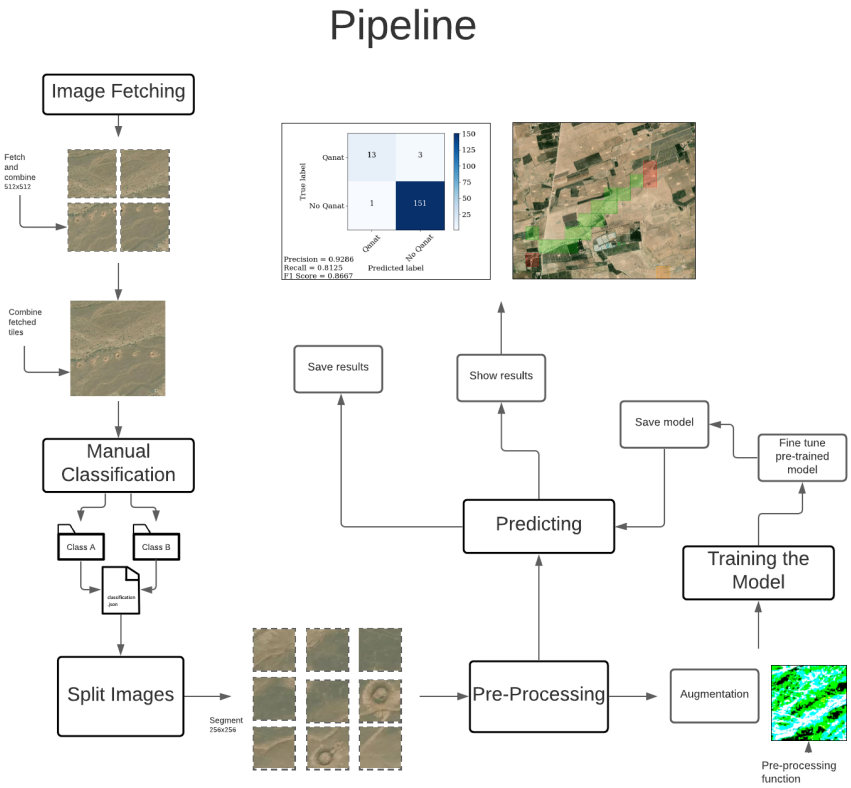


Figure 4.2: The workflow of the pipeline.

4.6 Evaluate Models Against Previous Work

To evaluate our solution against previous work, we will compare the three models with the results of the paper by Soroush et al. [40] (Locations of training patches along with labeling information is available at http://www.mdpi.com/2072-4292/12/3/500/s1). We performed 5-fold cross validation on 11 patches from CORONA Imagery in Iraq to replicate the methodology from [40], and compared the results.

5

Results

In this section the results from the training and the predicting of the models will be presented. Training results for qanats and fortresses are first presented in tables in their respective sections, comparing the different models. A graph for each model shows accuracy and validation accuracy per epoch. Confusion matrices for the testing phase of the training is also presented, along with precision, recall and F1-score.

Prediction results are shown for a selection of tilesets, along with confusion matrices, precision, recall and F1-score. Locations for all tilesets used for testing and prediction can be found in tables in the appendix A.2, and all prediction results can be found in our GitHub repository (A.1).

5.1 Training on Qanat Images

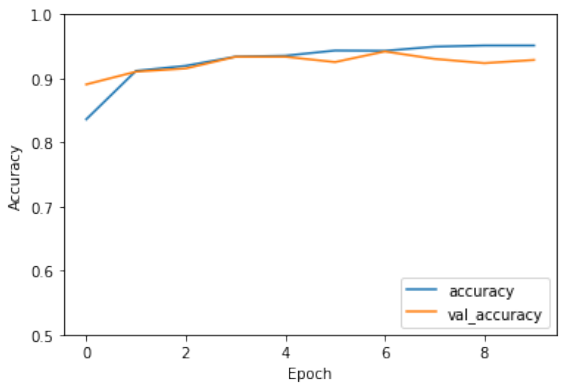
Table 5.1 shows the highest scores obtained for qanats from the training of the different models as well as the learning rate used.

Model	Learning rate	Accuracy	Precision	Recall	F1 score
ResNet-152	1e-5	0.9706	0.9013	0.8671	0.8839
DenseNet-201	1e-4	0.9645	0.8735	0.9177	0.8951
EfficientNet-B3	1e-5	0.9720	0.9051	0.9022	0.8994

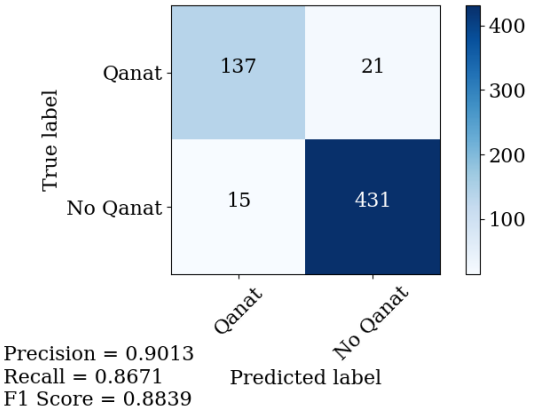
Table 5.1: Learning rate, training accuracy and testing statistics for the models (qanats).

5.1.1 Resnet-152 Training Results for Qanats

The accuracy compared with the validation accuracy can be seen in 5.1a. The accuracy is closely followed by the validation accuracy through the various epochs.



(a) Accuracy vs validation accuracy

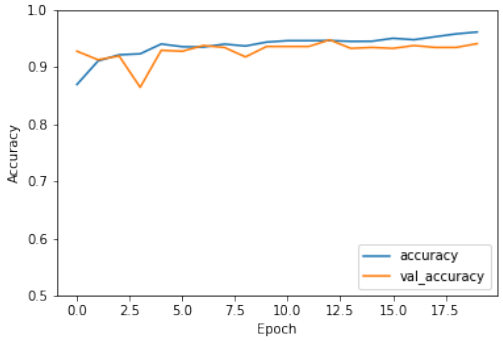


(b) Confusion matrix results

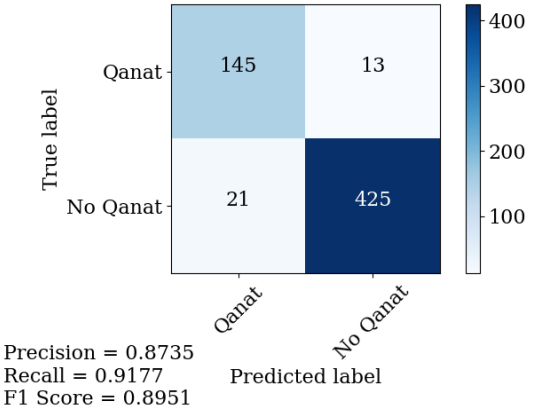
Figure 5.1: ResNet-152 accuracy and confusion matrix results for qanats

5.1.2 DenseNet-201 Training Results for Qanats

The accuracy compared with the validation accuracy can be seen in 5.2a. Except for a brief moment the accuracy is closely followed by the validation accuracy through the various epochs.



(a) Accuracy vs validation accuracy

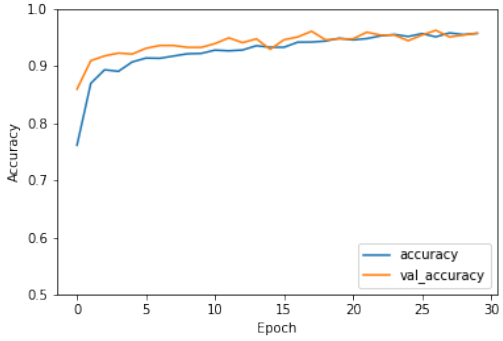


(b) Confusion matrix results

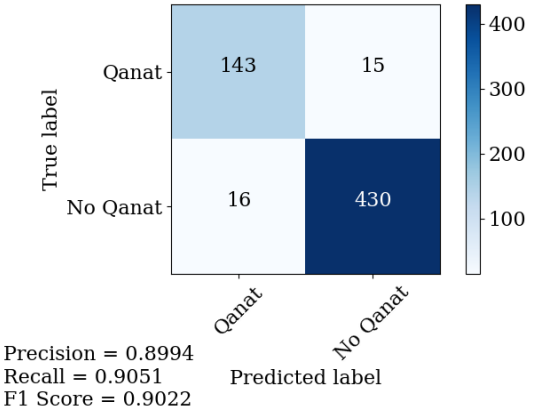
Figure 5.2: DenseNet-201 accuracy and confusion matrix results for qanats

5.1.3 EfficientNet-B3 Training Results for Qanats

We experimented with different number of epochs for EfficientNet-B3. A learning rate of $1e$-5 together with 30 epochs produced the superior results as shown in figure 5.3.



(a) Accuracy vs validation accuracy



(b) Confusion matrix results

Figure 5.3: EfficientNet-B3 accuracy and confusion matrix results for qanats

5.2 Training on Fortress Images

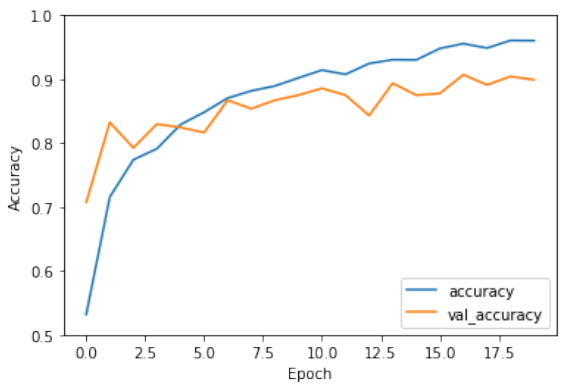
Table 5.2 shows the highest scores obtained for fortresses from the training of the different models as well as the learning rate used.

Model	Learning rate	Accuracy	Precision	Recall	F1 score
ResNet-152	1e-5	0.9801	0.5714	0.6792	0.6207
DenseNet-201	1e-4	0.9632	0.6383	0.5660	0.6000
EfficientNet-B3	5e-5	0.9698	0.6727	0.6981	0.6852

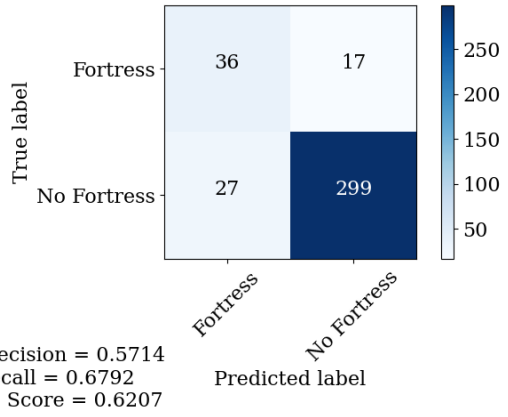
Table 5.2: Learning rate, training accuracy and testing statistics for the models (fortresses).

5.2.1 ResNet-152 Training for Fortresses

Figure 5.4 shows the training results for ResNet-152 with a graph for accuracy and with a confusion matrix.



(a) Accuracy vs validation accuracy

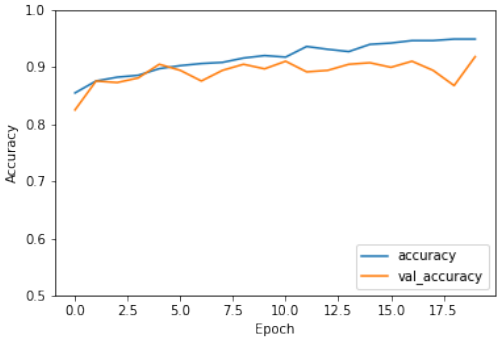


(b) Confusion matrix results

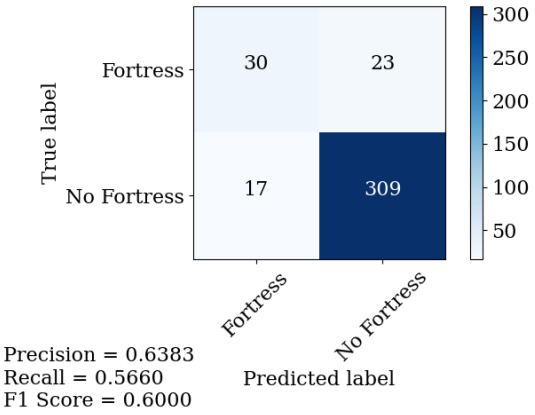
Figure 5.4: ResNet-152 accuracy and confusion matrix results for fortresses

5.2.2 DenseNet-201 Training for Fortresses

Figure 5.5a shows the accuracy from the fortress training for the DenseNet-201 model. Accuracy increases gradually while validation accuracy seems a little more unstable and even overlaps with accuracy at times.



(a) Accuracy vs validation accuracy

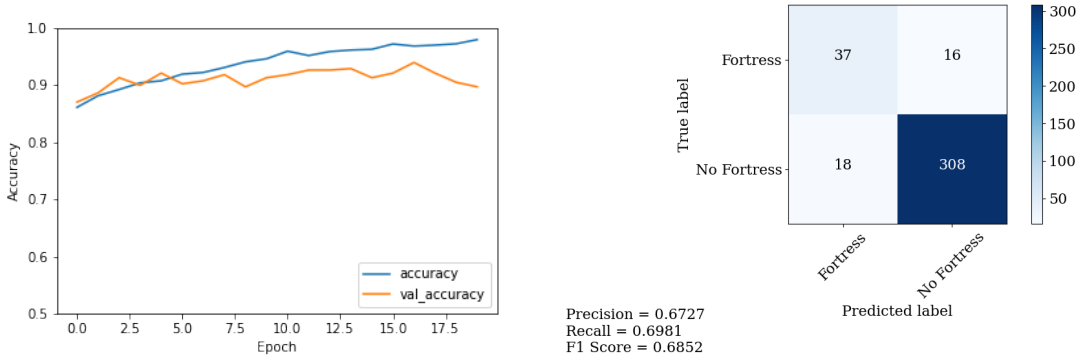


(b) Confusion matrix results

Figure 5.5: DenseNet-201 accuracy and confusion matrix results for fortresses

5.2.3 EfficientNet-B3 Training for Fortresses

Figure 5.6 shows the accuracy and validation accuracy as well as the confusion matrix.



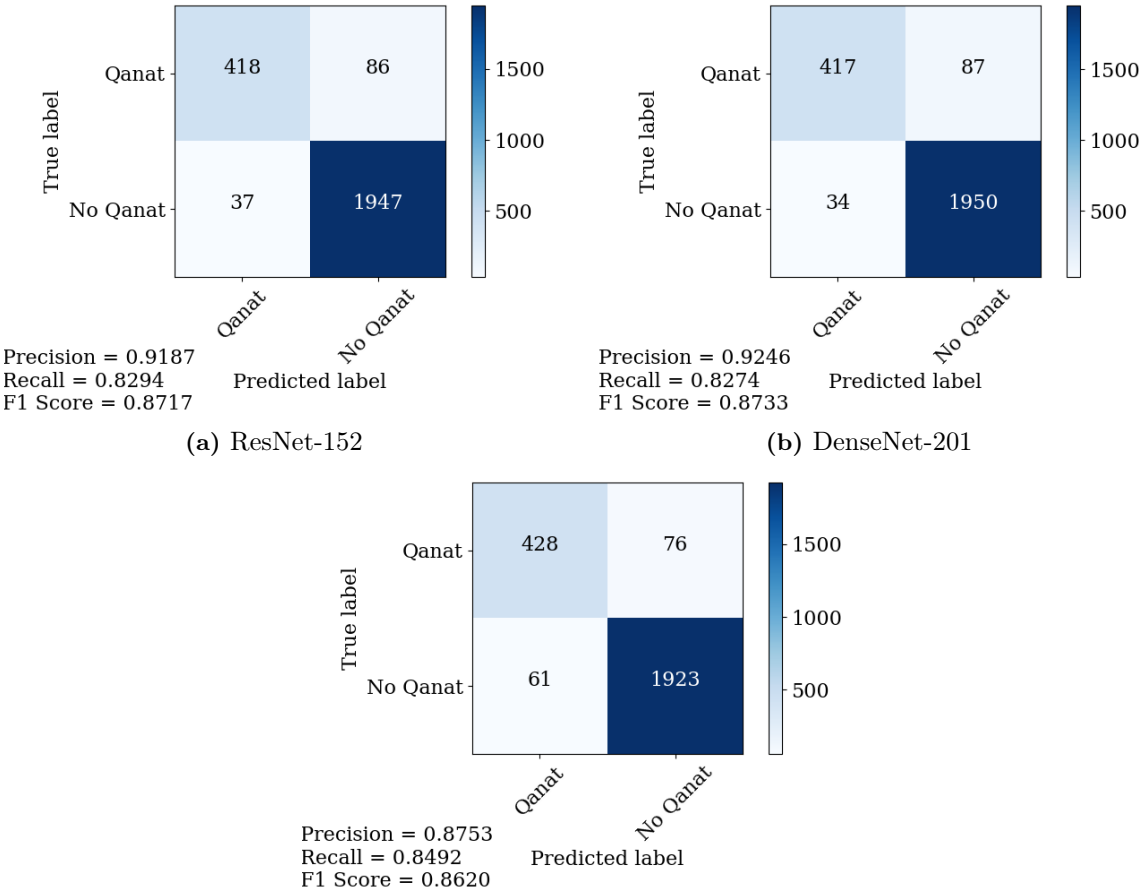
(a) Accuracy vs validation accuracy

(b) Confusion matrix results

Figure 5.6: EfficientNet-B3 accuracy and confusion matrix results for fortresses

5.3 Predicting on Qanat Areas

To get a better understanding of the overall performance from the models we combined the confusion matrices for all the areas predicted on into one. The results from this can be seen in figure 5.7.



(a) ResNet-152

(b) DenseNet-201

(c) EfficientNet-B3

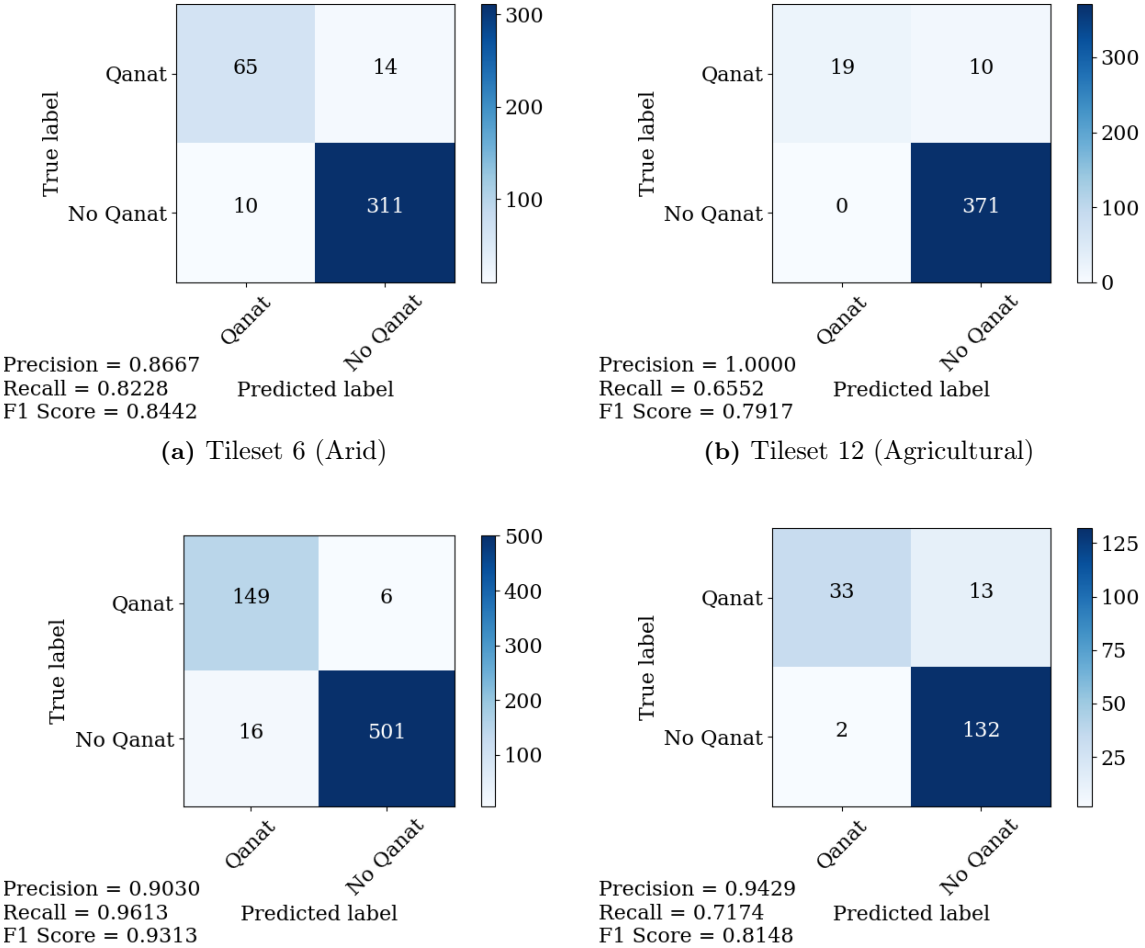
Figure 5.7: All the qanat predictions performed combined into one confusion matrix for each model.

In the following sections we present prediction results from a selection of the testing tilesets for each model. The tilesets selected have ID 6, 12, 24 and 25. Coordinates for all qanat

testing tilesets can be found in appendix table A.2.

5.3.1 Resnet-152 Qanat Results

The qanat results represented with confusion matrices can be seen in figure 5.8 and through satellite images in figure 5.9.



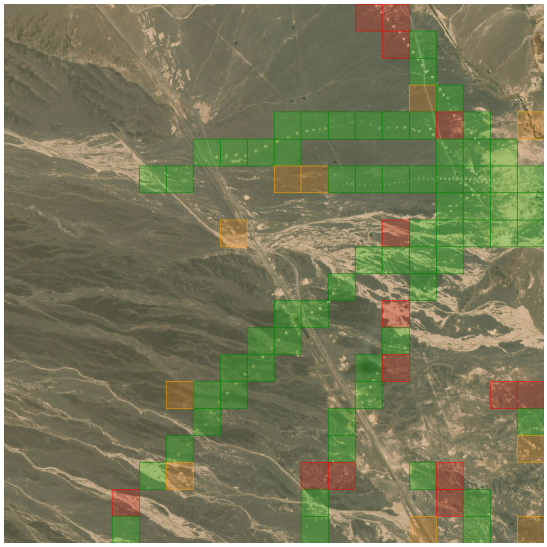
(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)

(c) Tileset 24 (Semi-arid/Urban)

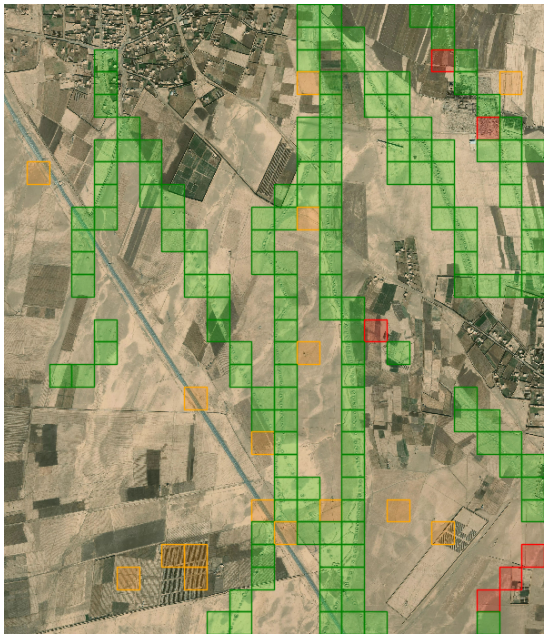
(d) Tileset 25 (Urban)

Figure 5.8: Confusion matrices for different predicted tilesets 6, 12, 24 and 25 with ResNet-152.

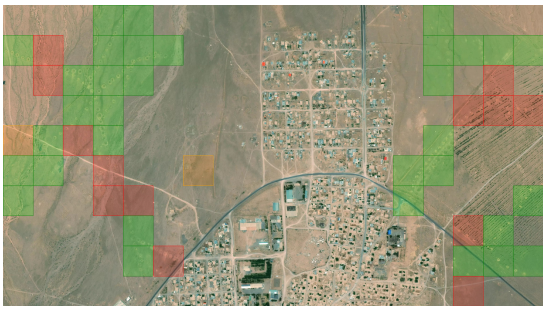


(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)



(c) Tileset 24 (Semi-arid/Urban)

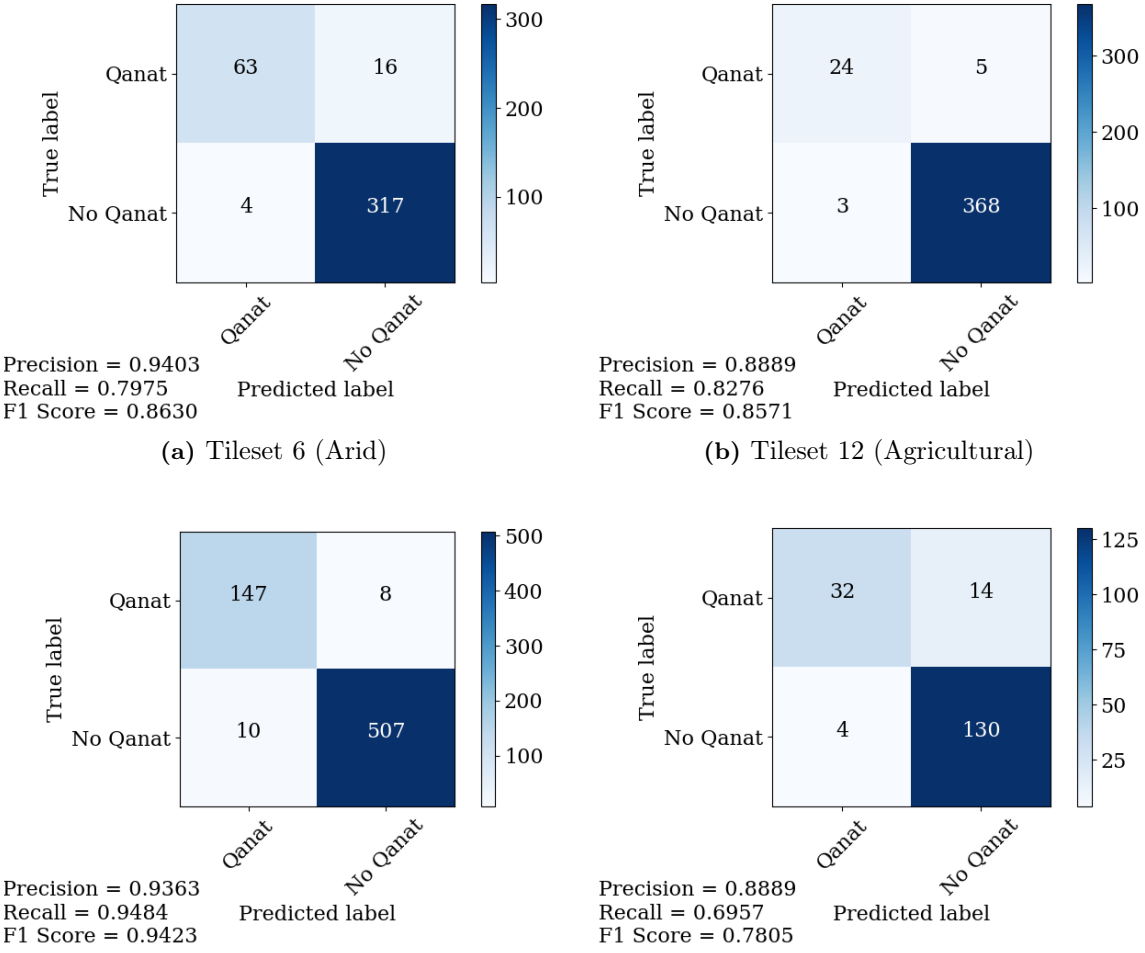


(d) Tileset 25 (Urban)

Figure 5.9: Prediction images for qanat tilesets 6, 12, 24 and 25 with ResNet-152. Segment colors: True positive (green), false positive (orange) and false negative (red).

5.3.2 DenseNet-201 Qanat Results

The qanat results with confusion matrix and metrics from DenseNet-201 can be seen in figure 5.10 and with satellite images in figure 5.11



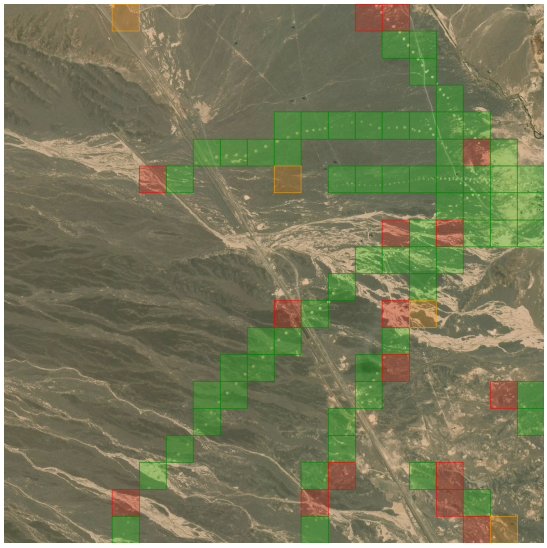
(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)

(c) Tileset 24 (Semi-arid/Urban)

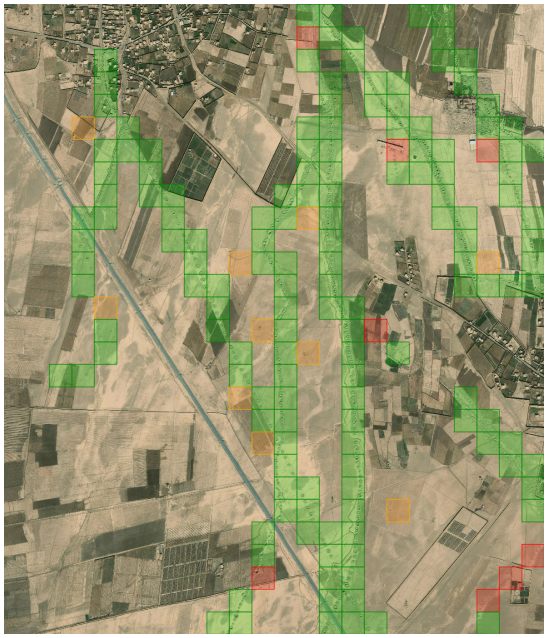
(d) Tileset 25 (Urban)

Figure 5.10: Confusion matrices for different predicted tilesets 6, 12, 24 and 25 with DenseNet-201.

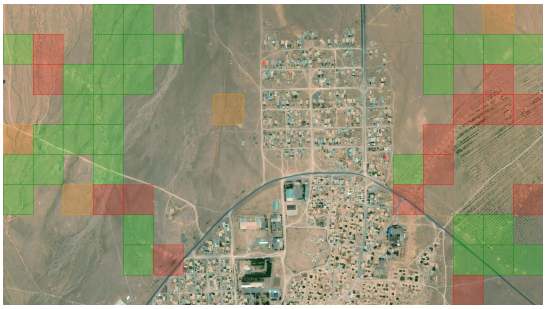


(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)



(c) Tileset 24 (Semi-arid/Urban)

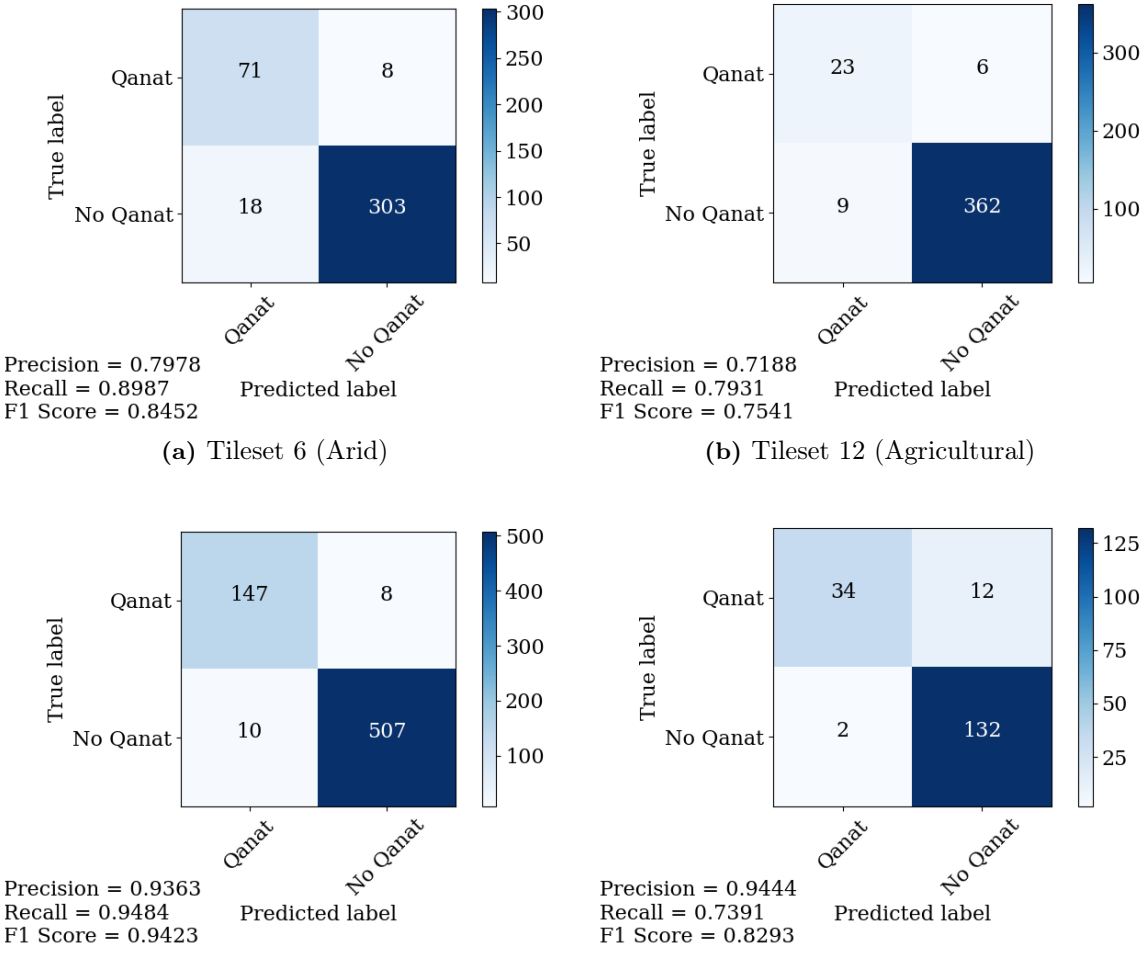


(d) Tileset 25 (Urban)

Figure 5.11: Prediction images for qanat tilesets 6, 12, 24 and 25 with DenseNet-201. Segment colors: True positive (green), false positive (orange) and false negative (red).

5.3.3 EfficientNet-B3 Qanat Results

The confusion matrices for the predictions on tileset 6, 12, 24, 25 are shown in figure 5.12 and the prediction images in figure 5.13.



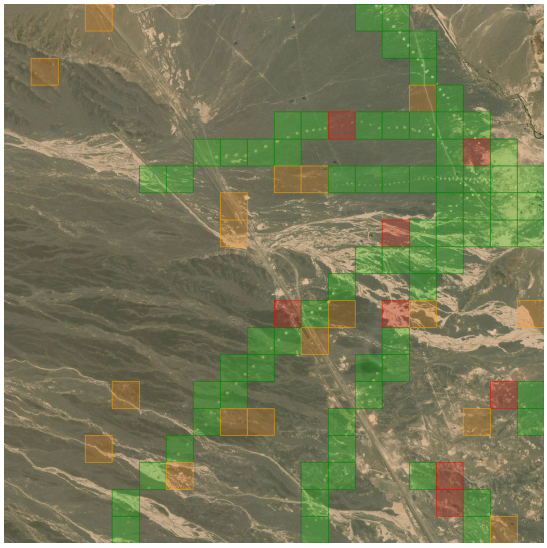
(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)

(c) Tileset 24 (Semi-arid/Urban)

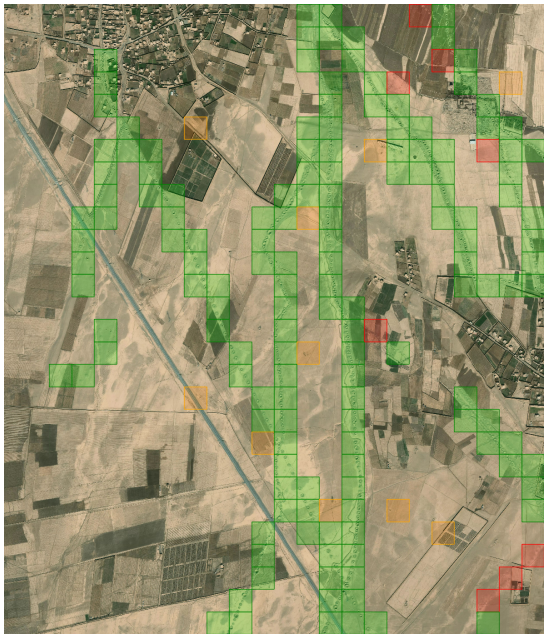
(d) Tileset 25 (Urban)

Figure 5.12: Confusion matrices for different predicted tilesets 6, 12, 24 and 25 with EfficientNet-B3.

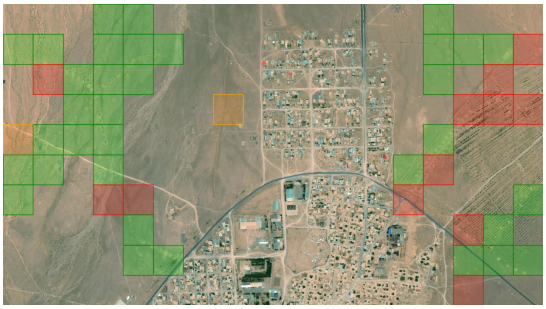


(a) Tileset 6 (Arid)

(b) Tileset 12 (Agricultural)



(c) Tileset 24 (Semi-arid/Urban)

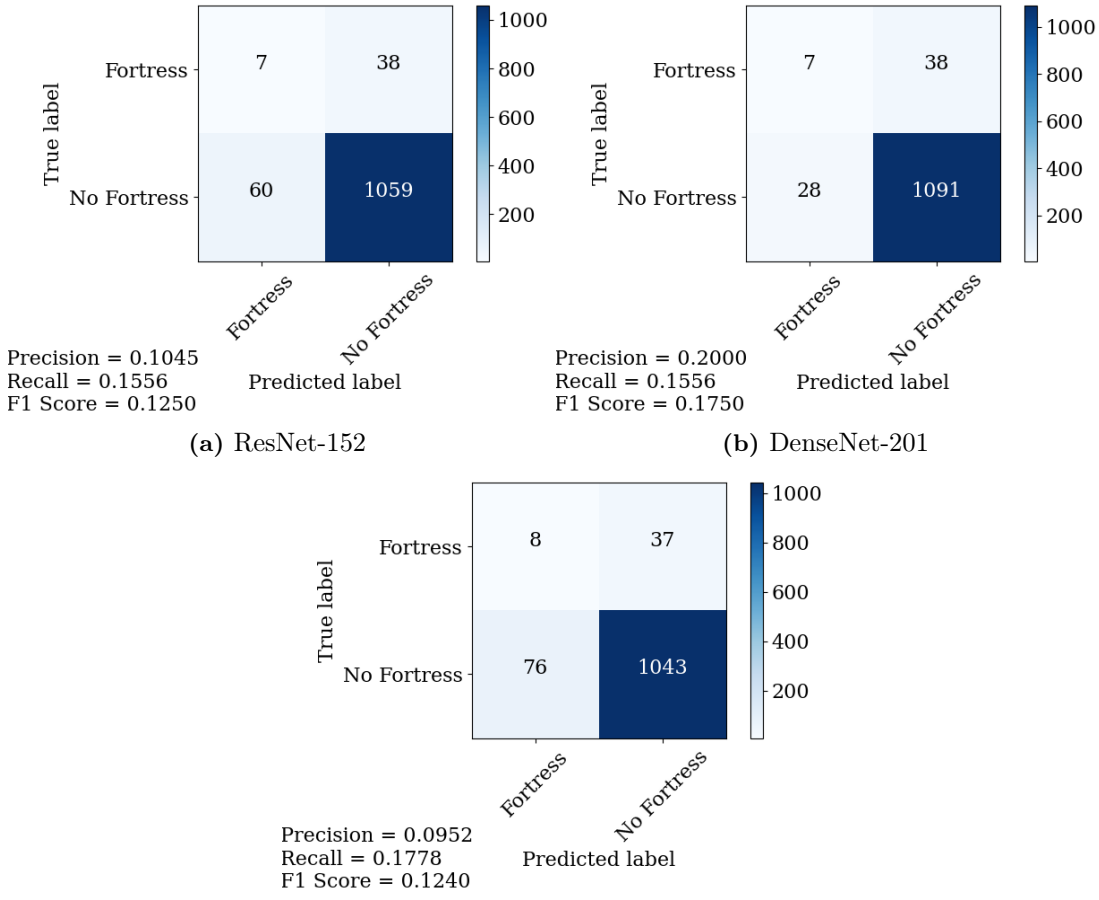


(d) Tileset 25 (Urban)

Figure 5.13: Prediction images for qanat tilesets 6, 12, 24 and 25 with EfficientNet-B3. Segment colors: True positive (green), false positive (orange) and false negative (red).

5.4 Fortress Prediction Results

Figure 5.14 displays all the different predictions for each model combined into one confusion matrix.



(a) ResNet-152

(b) DenseNet-201

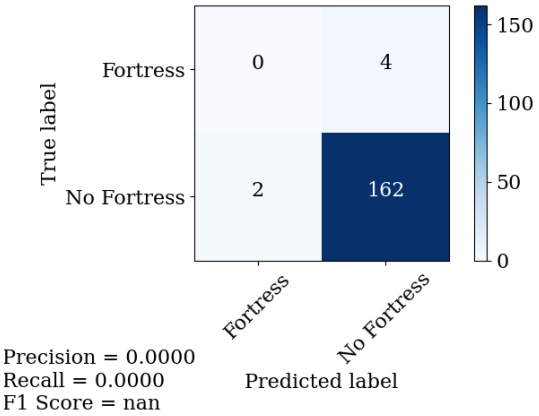
(c) EfficientNet-B3

Figure 5.14: All the fortress predictions performed combined into one confusion matrix for each model.

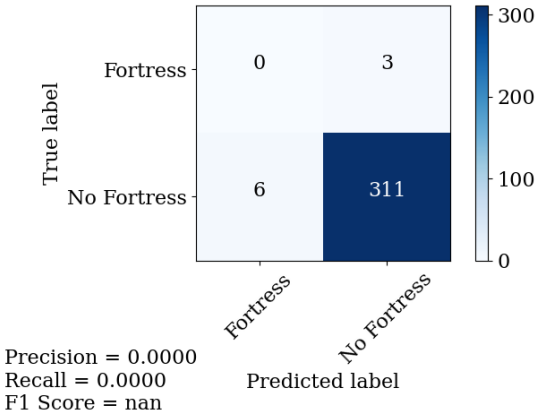
In the following sections we present prediction results from a selection of the fortress testing tilesets for each model. The tilesets selected have ID 43 and 45. Coordinates for all fortress testing tilesets can be found in appendix table A.4.

5.4.1 ResNet-152 Fortress Results

ResNet-152's results with confusion matrices and satellite images can be seen in figure 5.15.

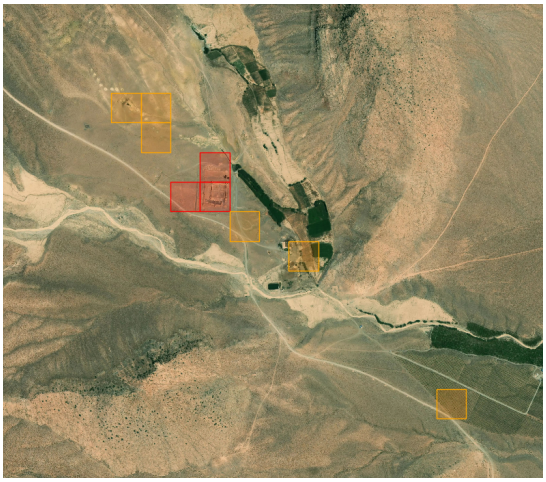


(a) Tileset 43 (Urban)



(b) Tileset 45 (Arid)

(c) Tileset 43 (Urban)

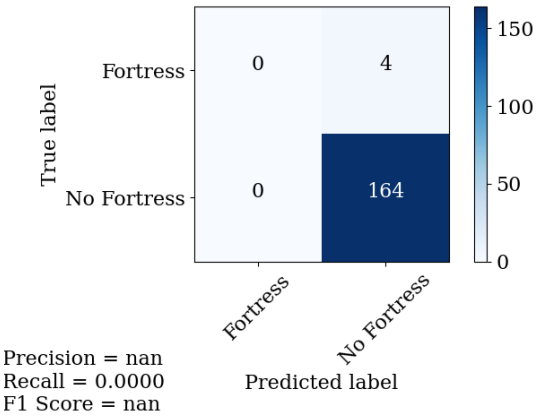


(d) Tileset 45 (Arid)

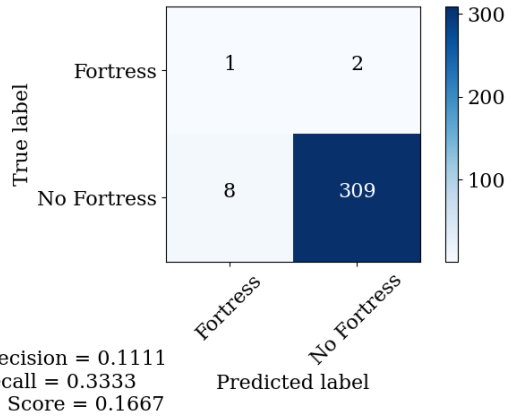
Figure 5.15: Confusion matrices and prediction images for fortress tilesets 43 and 45 with ResNet-152. Segment colors: True positive (green), false positive (orange) and false negative (red).

5.4.2 DenseNet-201 Fortress Results

The DenseNet-201 results for fortresses with confusion matrix, metrics, and images of the areas predicted on can be found in figure 5.16.



(a) Tileset 43 (Urban)



(b) Tileset 45 (Arid)

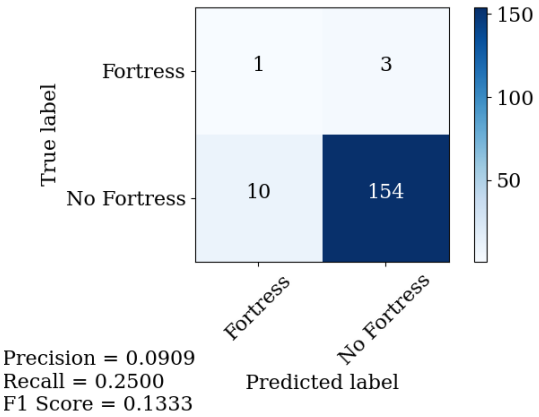
(c) Tileset 43 (Urban)

(d) Tileset 45 (Arid)

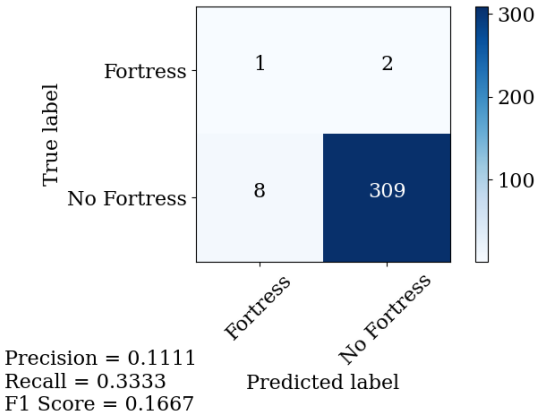
Figure 5.16: Confusion matrices and prediction images for fortress tilesets 43 and 45 with DenseNet-201. Segment colors: True positive (green), false positive (orange) and false negative (red).

5.4.3 EfficientNet-B3 Fortress Results

EfficientNet-B3's results from the predicting on fortresses can be seen in figure 5.17 and includes confusion matrices as well as satellite images.

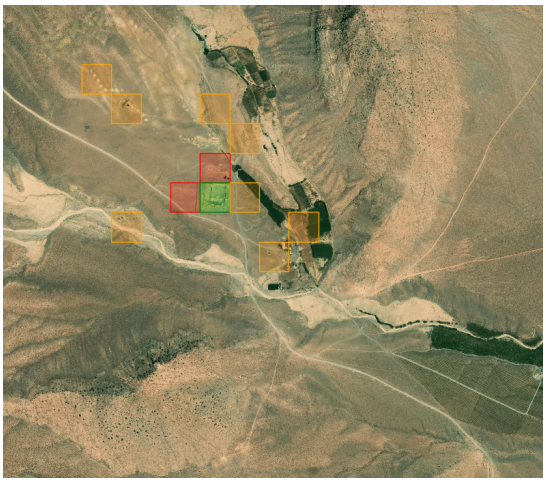


(a) Tileset 43 (Urban)



(b) Tileset 45 (Arid)

(c) Tileset 43 (Urban)



(d) Tileset 45 (Arid)

Figure 5.17: Confusion matrices and prediction images for fortress tilesets 43 and 45 with EfficientNet-B3. Segment colors: True positive (green), false positive (orange) and false negative (red).

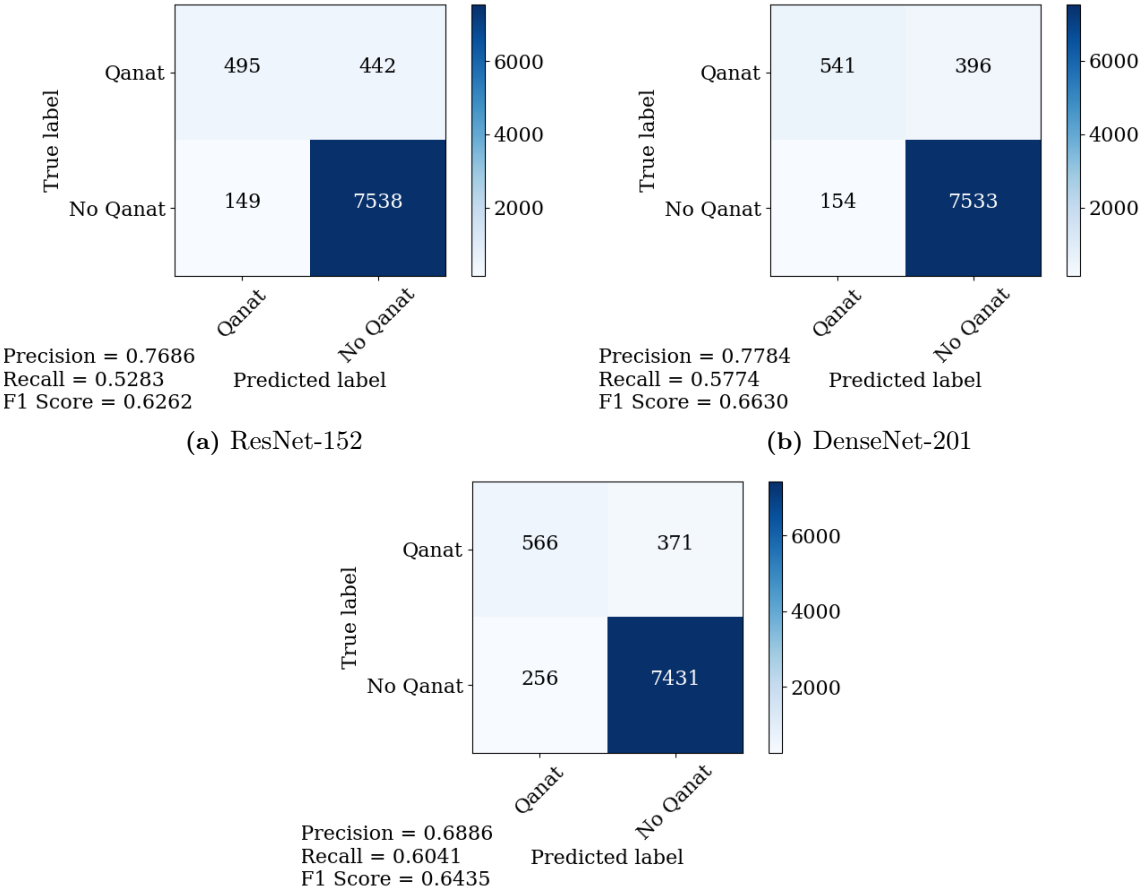
5.5 Corona Imagery Results

The numbers in table 5.3 compares the F1 scores from the Corona imagery paper by Soroush et al. [40] with the F1-scores from our three models after 5-fold cross-validation on the 11 patches.

Patch	Soroush et al.	ResNet-152	DenseNet-201	EfficientNet-B3
1	0.574	0.333	0.323	0.378
2	0.352	0.286	0.143	0.071
3	0.569	0.511	0.520	0.454
4	0.175	-	0.400	0.077
5	0.462	0.370	0.333	0.365
6	0.665	0.672	0.728	0.736
7	0.309	0.105	0.125	0.250
8	0.513	0.345	0.359	0.308
9	0.462	0.308	0.533	0.364
10	0.738	0.695	0.739	0.697
11	0.805	0.730	0.767	0.777

Table 5.3: Comparison of F1-scores on different patches between our models and results from [40].

The paper by Soroush et al. [40] gives an overall precision and recall of 0.62 and 0.74 respectively. In order to get an overall evaluation of our models, we combined the confusion matrices for the different testing tilesets as seen in figure 5.18.



(a) ResNet-152

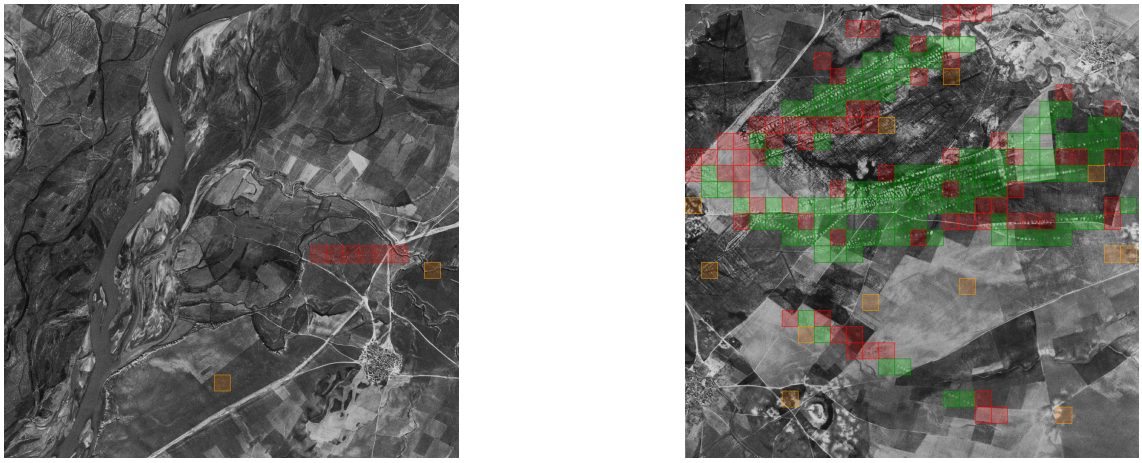
(b) DenseNet-201

(c) EfficientNet-B3

Figure 5.18: Confusion matrices for all patches after 5-fold cross validation with the different models, with precision, recall and F1 score.

5.5.1 Resnet-152 Prediction Samples from Corona Imagery

Figure 5.19 shows patches 4 and 11 from Corona imagery with prediction results from ResNet-152.

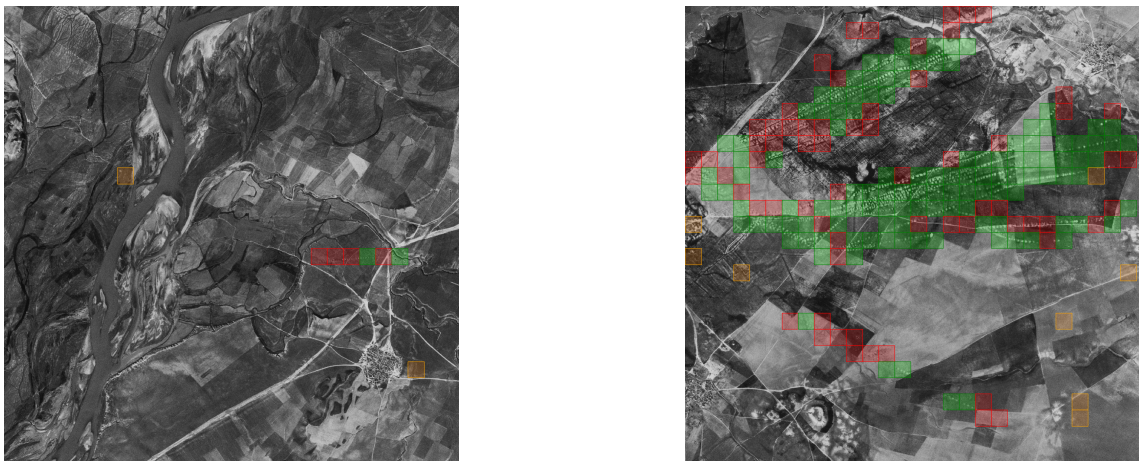


(a) Patch 4 **(b)** Patch 11

Figure 5.19: ResNet-152 predictions on patches 4 and 11.

5.5.2 DenseNet-201 Prediction Samples from Corona Imagery

Figure 5.20 shows patches 4 and 11 from Corona imagery with prediction results from DenseNet-201.

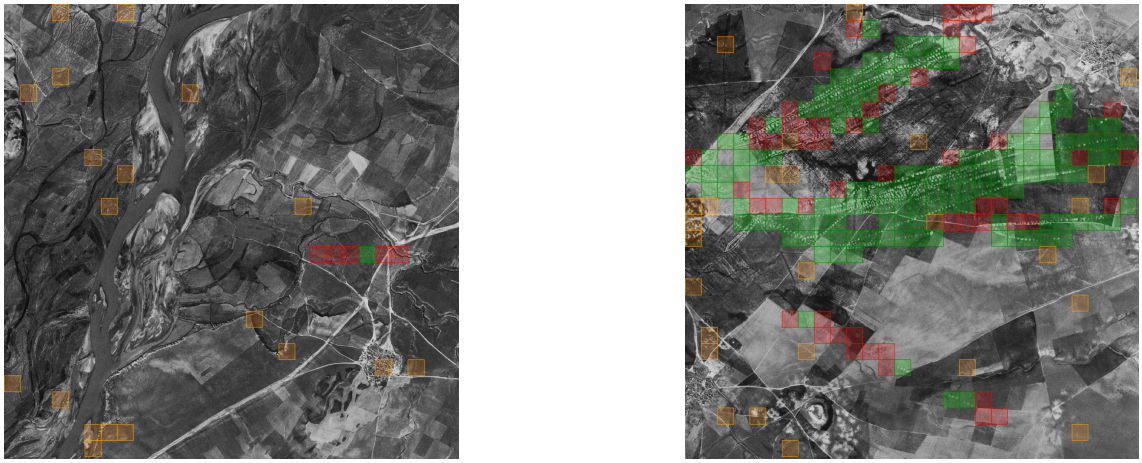


(a) Patch 4 **(b)** Patch 11

Figure 5.20: DenseNet-201 predictions on patches 4 and 11.

5.5.3 EfficientNet-B3 Prediction Samples from Corona Imagery

Figure 5.21b shows patches 4 and 11 from Corona imagery with prediction results from EfficientNet-B3.

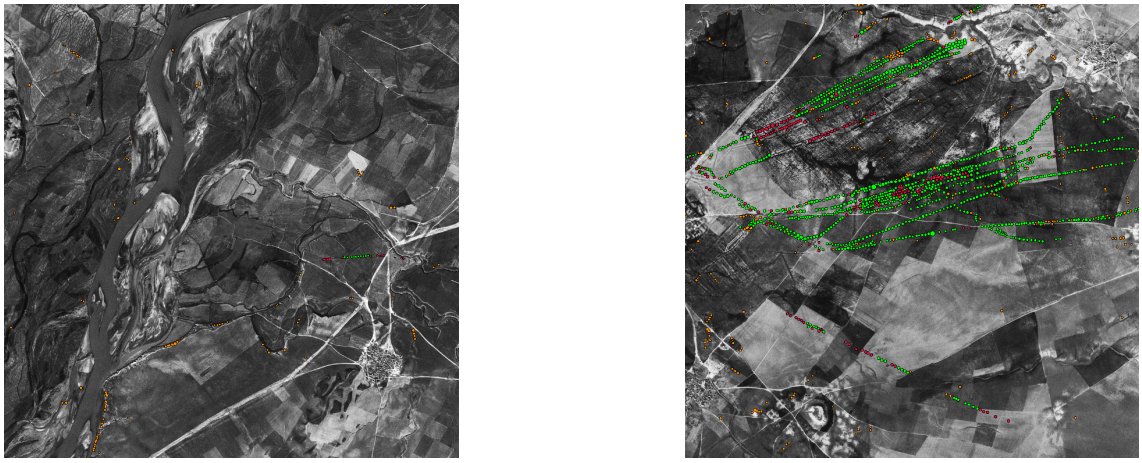


(a) Patch 4 **(b)** Patch 11

Figure 5.21: EfficientNet-B3 predictions on patches 4 and 11.

5.5.4 Prediction Samples from Paper by Soroush et al.

Figure 5.22b shows patches 4 and 11 from Corona imagery with prediction results from Soroush et al. [40].



(a) Patch 4 **(b)** Patch 11

Figure 5.22: Predictions from paper by Soroush et al. on patches 4 and 11.

6

Discussion

6.1 Results from Qanat Imagery

The results from qanat predictions were very similar across the different models. The combined results of predictions for all tilesets gave F1-scores of 0.8718, 0.8733 and 0.8620 for Resnet-152, DenseNet-201 and EfficientNet-B3 respectively. Looking at individual tilesets, there were somewhat larger differences between models; for tileset 6, the F1-scores were 0.8442, 0.8630 and 0.8452 for ResNet-152, DenseNet-201 and EfficientNet-B3 respectively. The largest differences seem to be between tilesets for a given model. For example, EfficientNet-B3 got an F1-score of 0.7541 for tileset 12, and 0.9423 for tileset 24. Tileset 12 is an agricultural area, and has a low density of qanats, while tileset 24 shows less agriculture, more desert and a high density of qanats. This seems to be a trend with other tilesets as well; areas with high qanat density produce better results. The same trend was found in [40]. It may also be due to the type of terrain in the area, but qanat-density seems to have more of an effect.

6.2 Results from Corona Imagery

Before comparing our results on the CORONA imagery with the results from [40], it must be made clear that there is a major difference in the detection approaches. Our method segments an image into smaller squares, and classifies these one by one. These squares

typically cover many qanats, resulting in a lower "detection resolution". Soroush et al. detected mostly individual qanats, so we cannot make a one-to-one comparison between these to approaches. Regardless, it may provide some indication of the comparative performance of our approach, even if it is not directly comparable.

Overall, our approach produced slightly worse results than Soroush et al. [40]. Table 5.3 shows that all of our models received a worse F1-score on almost all patches. All of our models received a better F1-score on patch 6, and DenseNet-201 scored higher on patch 9 and slightly higher on patch 10 as well. The scores seem to follow the same trend, where patches with high qanat density produce better scores than low-density patches.

In terms of combined statistics for all patches, Soroush et al. [40] achieved a precision and recall of 0.62 and 0.72 respectively. Our models have a higher precision but lower recall. The best performing model according to table 5.18, DenseNet-201, produced an overall precision of 0.7784 and recall of 0.5774.

6.3 Limitations in Methodology and Data

Segmenting images and classifying the segments worked well for detecting qanats. They are relatively small and do not vary much in size, and therefore fit easily within relatively small segments. Fortresses on the other hand vary greatly in size and form, which presents challenges to this approach. On some tilesets, e.g. 43 (figure 5.16), the models did not manage to identify a single segment belonging to the fortress. The results were consistently poor across all testing tilesets (table A.4). The limitation of our methodology is clear in that it does not work for more complex structures with varying size. Each segment can often only cover a small part of the structure, and excludes valuable context from the surrounding area. Increasing the segment size may mitigate this to a degree, but segments only covering parts of fortresses would still be a problem, since the segment locations are fixed.

Another likely limiting factor with regards to fortress detection was the dataset itself. In terms of size, it was probably to small. Statistics from the test phase of the model training (table 5.2) produced better results than predicting on the testing tilesets (figure 5.14), a sign of overfitting. The training set for qanats contained 1572 images of qanats, while the fortress training set only contained 517. Adding to this the fact that the fortress images are much more varied in appearance, this number is likely too low if the models are to generalize to new data. Furthermore, remains of fortresses are often partially or completely

buried, making them hard to detect from optical imagery. Stott et al. [36] used LiDAR data to find circular viking fortresses. Using LiDAR provides a great advantage when looking for differences in elevation, because small differences often do not appear as clearly in optical imagery. Future work in fortress detection may therefore benefit from using other types of imagery instead of optical.

Additionally, the process of manual classification/labeling of segments may have introduced problems. It was often the case that qanats were partially inside a segment, which made us unsure about whether to label the segment or not. This was even more often the case when labeling segments with fortresses. Additionally, the authors have no experience with identifying archaeological structures, which may have lead to cases of mislabeling.

None of the authors had any prior experience with machine learning when starting this thesis. Consequently, we had to learn about deep learning from scratch, in both the theoretical and practical aspects. The result has been a solution approach that is not ideal for structures with inconsistent size and complexity. Given more time, we would probably use a different type of deep learning architecture based on what we have learned, for example R-CNN or YOLO.

7

Conclusion

7.1 Answers to Research Questions

In section 1.3 we presented some research questions which this thesis aimed to give an answer to. The answers to these questions are stated below:

1. **Can pre-trained models with fine tuning be used in the detection of simple ancient structures like qanats?**
   Yes, they seem quite effective at detecting qanats. Results for all tilesets in 5.3 shows an F1-score of 0.8733 for DenseNet-201, which was the highest of the three models.
2. **Can the approach be extended to more complex structures like fortresses?**
   No, when it comes to more complex structures which vary a lot in size and form like with fortresses, the models do no perform well. Results in 5.4 shows an F1-score of 0.1750 for DenseNet-201, again the best performing model.
3. **What types of terrain can the models be used on? Arid, semi-arid, agricultural, urban?**
   Based on results in 5.3, the type of terrain predicted on may have some effect, but this may also be due to the density of qanats in the image. The detection of fortresses did not obtain good results regardless of the terrain, as seen in 5.4.
4. **Can the easier implementation with image-classifiers compete with more**

advanced object detection methods when searching for qanats and fortresses? When working with structures of similar size and shape, the image-classifiers can work as well, and may be an easier alternative to implement compared to the object detectors. However, when tasked with structures that vary a lot in size and appearance, this method falls short.

7.2 Future Work

Further work with detecting ancient structures will likely benefit from using object detection architectures, for example R-CNN. For completely or partially buried structures like fortresses, other types of imagery such as LiDAR may produce better results. Another possibility would to be use imagery with a higher spatial resolution to see how it would affect the results of our models. It would also be interesting to test the performance of the models on other types of structures, such as mounds.

Bibliography

[1] ImageNet, "Imagenet." https://www.image-net.org/index.php, May 2021. Accessed: 2021-05-13.

[2] Microsoft, "Bing maps api." http://mapsforenterprise.binginternal.com/en-us/maps/choose-your-bing-maps-api, May 2021. Accessed: 2021-05-13.

[3] Google, "Google maps platform." https://developers.google.com/maps, May 2021. Accessed: 2021-05-13.

[4] Planet, "Planet." https://www.planet.com/, May 2021. Accessed: 2021-05-13.

[5] Skywatch, "Commercial satellite imagery made accessible." https://www.skywatch.com/, May 2021. Accessed: 2021-05-13.

[6] "Mapbox." https://www.mapbox.com/. Accessed: 2021-05-11.

[7] Wikipedia, "Qanat." https://en.wikipedia.org/wiki/Qanat, May 2021. Accessed: 2021-05-11.

[8] USGS, "What is remote sensing and what is it used for?." https://www.usgs.gov/faqs/what-remote-sensing-and-what-it-used?qt-news_science_products=0#qt-news_science_products. Accessed: 2021-03-31.

[9] Wikipedia, "Aerial photography." https://en.wikipedia.org/wiki/Aerial_photography, April 2021. Accessed: 2021-04-08.

[10] U. of Minnesota, "Introduction to satellite imagery." https://www.pgc.umn.edu/guides/commercial-imagery/intro-satellite-imagery/, January 2017. Accessed: 2021-03-29.

[11] USGS, "Usgs eros archive - declassified data - declassified satellite imagery - 1." https://www.usgs.gov/centers/eros/science/usgs-eros-archive-declassified-data-declassified-satellite-imagery-1?qt-science_center_objects=0#qt-science_center_objects. Accessed: 2021-04-07.

[12] Wikipedia, "Multispectral image." https://en.wikipedia.org/wiki/Multispectral_image#Spectral_bands, March 2021. Accessed: 2021-04-08.

[13] T. E. S. Agency, "Sentinel-2." https://sentinel.esa.int/web/sentinel/missions/sentinel-2, May 2021. Accessed: 2021-05-13.

[14] NASA, "Landsat science." https://landsat.gsfc.nasa.gov/, May 2021. Accessed: 2021-05-13.

[15] Wikipedia, "Lidar." https://en.wikipedia.org/wiki/Lidar, April 2021. Accessed: 2021-04-08.

[16] USGS, "Earthexplorer." https://earthexplorer.usgs.gov/, May 2021. Accessed: 2021-05-13.

[17] Sentinel, "Access to sentinel data via download." https://sentinels.copernicus.eu/web/sentinel/sentinel-data-access, May 2021. Accessed: 2021-05-13.

[18] E. O. System, "Satellite data: What spatial resolution is enough?." https://eos.com/blog/satellite-data-what-spatial-resolution-is-enough-for-you/, April 2019. Accessed: 2021-04-09.

[19] "Mapbox Raster Tiles API." https://docs.mapbox.com/api/maps/raster-tiles/. Accessed: 2021-05-11.

[20] "Maxar Technologies." https://www.maxar.com/about. Accessed: 2021-05-11.

[21] IBM Cloud Education, "Neural networks?." https://www.ibm.com/cloud/learn/neural-networks, Aug 2020.

[22] IBM Cloud Education, "Convolutional neural networks?." https://www.ibm.com/cloud/learn/convolutional-neural-networks, Oct 2020.

[23] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8689 LNCS, no. PART 1, pp. 818–833, 2014.

[24] harshsayshi, "Train your deep model faster and sharper — two novel techniques by." https://hackernoon.com/training-your-deep-model-faster-and-sharper-e85076c3b047. Accessed: 2021-04-13.

[25] Devopedia, "Imagenet." https://devopedia.org/imagenet. Accessed: 2021-03-28.

[26] Amazon, "Amazon mechnical turk." https://www.mturk.com/, May 2021. Accessed: 2021-05-15.

[27] https://www.tensorflow.org/api_docs/python/tf/keras/applications. Accessed 2021-04-13.

[28] Pathmind, "Evaluation metrics for machine learning - accuracy, precision, recall, and f1 defined." https://wiki.pathmind.com/accuracy-precision-recall-f1. Accessed: 2021-03-30.

[29] C. University, "Deep residual learning for image recognition." https://arxiv.org/abs/1512.03385, December 2015. Accessed: 2021-05-13.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[31] C. University, "Densely connected convolutional networks." https://arxiv.org/abs/1608.06993, January 2018. Accessed: 2021-05-13.

[32] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2020.

[33] C. University, "Efficientnet: Rethinking model scaling for convolutional neural networks." https://arxiv.org/abs/1905.11946, September 2020. Accessed: 2021-05-13.

[34] esri, "What is gis?." https://www.esri.com/en-us/what-is-gis/overview#liSwitcher. Accessed: 2021-04-13.

[35] L. Luo, X. Wang, H. Guo, C. Liu, J. Liu, L. Li, X. Du, and G. Qian, "Automated Extraction of the Archaeological Tops of Qanat Shafts from VHR Imagery in Google Earth," *Remote Sensing*, vol. 6, pp. 11956–11976, dec 2014.

[36] D. Stott, S. M. Kristiansen, and S. M. Sindbæk, "Searching for viking age fortresses with automatic landscape classification and feature detection," *Remote Sensing*, vol. 11, no. 16, pp. 1–20, 2019.

[37] H. A. Orengo, F. C. Conesa, A. Garcia-Molsosa, A. Lobo, A. S. Green, M. Madella, and C. A. Petrie, "Automated detection of archaeological mounds using machine-learning classification of multisensor and multitemporal satellite data," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 117, no. 31, pp. 18240–18250, 2020.

[38] A. Guyot, L. Hubert-Moy, and T. Lorho, "Detecting Neolithic burial mounds from LiDAR-derived elevation data using a multi-scale approach and machine learning techniques," *Remote Sensing*, vol. 10, no. 2, 2018.

[39] G. Caspari and P. Crespo, "Convolutional neural networks for archaeological site detection – Finding "princely" tombs," *Journal of Archaeological Science*, vol. 110, p. 104998, oct 2019.

[40] M. Soroush, A. Mehrtash, E. Khazraee, and J. A. Ur, "Deep Learning in Archaeological Remote Sensing: Automated Qanat Detection in the Kurdistan Region of Iraq," *Remote Sensing*, vol. 12, p. 500, feb 2020.

[41] W. B. Verschoof-van der Vaart and K. Lambers, "Learning to Look at LiDAR: The Use of R-CNN in the Automated Detection of Archaeological Objects in LiDAR Data from the Netherlands," *Journal of Computer Applications in Archaeology*, vol. 2, no. 1, pp. 31–40, 2019.

[42] Keras, "keras." https://keras.io/, May 2021. Accessed: 2021-05-13.

[43] Python, "Python 3.7.10 documentation." https://docs.python.org/3.7/, May 2021. Accessed: 2021-05-13.

[44] Mapbox, "Mercantile." https://mercantile.readthedocs.io/en/latest/, 2021. Accessed: 2021-05-13.

[45] Keras, "Resnet and resnetv2." https://keras.io/api/applications/resnet/, May 2021. Accessed: 2021-05-13.

[46] Keras, "Densenet." https://keras.io/api/applications/densenet/, May 2021. Accessed: 2021-05-13.

[47] Keras, "Efficientnet b0 to b7." https://keras.io/api/applications/efficientnet/, May 2021. Accessed: 2021-05-13.

[48] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[49] "Kaggle." https://www.kaggle.com/. Accessed: 2021-05-11.

[50] Jupyter, "Jupyter." https://jupyter.org/, May 2021. Accessed: 2021-05-13.

[51] TensorFlow, "tf.keras.preprocessing.image.imagedatagenerator." https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator, March 2021. Accessed: 2021-05-13.

[52] TensorFlow, "tf.keras.model." https://www.tensorflow.org/api_docs/python/tf/keras/Model, May 2021. Accessed: 2021-05-13.

[53] scikit-learn developers, "Confusion matrix." https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html, 2020. Accessed: 2021-05-13.

[54] TensorFlow, "$tf.keras.applications.resnet.preprocess_input$." https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet/preprocess_input, January 2021. Accessed: 2021-05-13.

[55] G. Van Rossum, "The python library reference, release 3.7.10." https://docs.python.org/3.7/library/tk.html, 2020.

[56] Keras, "Models." https://keras.io/api/models/, May 2021. Accessed: 2021-05-13.

[57] e. a. H. Butler, "The geojson format," RFC 7946, RFC Editor, 8 2016.

List of Figures

List of Tables

Appendix A

Supplementary Information

A.1 Github repository and Dataset

Our code and datasets are available at our GitHub repository:
https://github.com/naeemk/SpaceArch/

A.2 Areas/tilesets used in training and testing

A KML file with the areas used in our dataset is available in our GitHub repository at
https://github.com/naeemk/SpaceArch/blob/7d4dc1986060cecf61f2c07ab80f937cb88fa547/kml_map/Areas.kml

The tables in this section list all areas/tilesets with coordinates for their top-left and bottom-right corners. The zoom level was set to 17 for all the tilesets.

A.2 Areas/tilesets used in training and testing

Tileset ID	Top Left Corner	Bottom Right Corner	Location
0	34.35781, 58.58042	34.33726, 58.60262	Iran
1	29.03114, 58.39036	29.01138, 58.41381	Iran
2	34.42147, 58.92756	34.4011, 58.94644	Iran
3	33.6569, 56.92319	33.64689, 56.93319	Iran
4	34.29336, 58.65562	34.26783, 58.71141	Iran
5	29.19981, 58.20017	29.15981, 58.24017	Iran
8	34.38103, 56.90979	34.34475, 56.96402	Iran
10	34.31308, 58.6836	34.29256, 58.71122	Iran
11	32.86582, 52.96904	32.84661, 52.99303	Iran
13	34.35564, 69.07141	34.34888, 69.08566	Afghanistan
14	31.65323, 54.50365	31.63009, 54.52545	Iran
15	31.62707, 54.50541	31.60788, 54.5335	Iran
16	33.7119, 69.34363	33.69801, 69.36953	Afghanistan
17	31.46697, 65.58468	31.45513, 65.62004	Afghanistan
18	30.10015, 56.96726	30.07976, 56.99677	Iran
19	33.47974, 69.13723	33.46417, 69.17199	Afghanistan
20	31.6565, 65.94922	31.64046, 65.97986	Afghanistan

Table A.1: Qanat tilesets used for training; tileset ID in Github repository, top left and bottom right coordinates and location.

Tileset ID	Top Left Corner	Bottom Right Corner	Location
6	29.33843, 57.84465	29.31651, 57.86969	Iran
7	33.72784, 69.40005	33.7179, 69.41684	Afghanistan
9	34.33153, 56.93918	34.31913, 56.95695	Iran
12	32.60412, 62.53076	32.58278, 62.55434	Afghanistan
21	31.52774, 54.43121	31.51644, 54.44768	Iran
22	31.45582, 65.76396	31.44465, 65.77874	Afghanistan
23	34.1472, 61.90542	34.13351, 61.92017	Afghanistan
24	34.26247, 61.97588	34.23385, 62.00492	Afganistan
25	33.64335, 60.26191	33.6335, 60.28269	Afghanistan

Table A.2: Qanat tilesets used for testing; tileset ID in Github repository, top left and bottom right coordinates and location.

A.2 Areas/tilesets used in training and testing

Tileset ID	Top Left Corner	Bottom Right Corner	Location
1	37.27764, 55.30294	37.26812, 55.31935	Iran
2	37.13968, 54.32456	37.13566, 54.32958	Iran
3	37.11102, 54.41807	37.0935, 54.43682	Iran
4	37.12023, 54.55129	37.11591, 54.55972	Iran
5	37.25078, 54.82079	37.24791, 54.82607	Iran
6	37.25901, 54.89346	37.25384, 54.90103	Iran
7	37.25694, 54.93345	37.2538, 54.93807	Iran
8	37.19043, 55.2269	37.16664, 55.26009	Iran
9	37.30916, 55.22566	37.30518, 55.23261	Iran
10	37.37964, 55.30822	37.3763, 55.31189	Iran
11	37.401, 55.33125	37.39739, 55.33663	Iran
12	37.42858, 55.37024	37.42577, 55.37547	Iran
13	37.4547, 55.41609	37.44955, 55.42448	Iran
14	37.12076, 57.10494	37.11642, 57.11017	Iran
15	37.04634, 57.46187	37.0397, 57.47483	Iran
16	37.09453, 56.91042	37.09297, 56.91334	Iran
17	28.90294, 50.82682	28.89759, 50.83362	Iran
18	35.35316, 51.66778	35.32981, 51.69104	Iran
19	39.61198, 47.75271	39.60349, 47.76397	Iran
20	39.45908, 48.41028	39.44563, 48.42585	Iran
21	39.45402, 48.4399	39.441, 48.45462	Iran
22	39.43276, 48.45931	39.42368, 48.47291	Iran
23	37.01806, 46.18619	37.00569, 46.20143	Iran
24	37.1244, 57.10579	37.12235, 57.1095	Iran
25	37.0516, 57.09168	37.04911, 57.09454	Iran
26	37.07301, 57.2583	37.07113, 57.26058	Iran
27	37.10491, 57.32414	37.10345, 57.32659	Iran
28	37.27656, 58.22959	37.27158, 58.23562	Iran
29	37.31778, 58.25096	37.30971, 58.26015	Iran
30	30.79919, 61.42296	30.7941, 61.42979	Iran
31	34.84308, 51.1437	34.84074, 51.14729	Iran
32	30.82854, 61.62571	30.82277, 61.63374	Iran
33	38.00165, 47.66022	37.98923, 47.67456	Iran
34	38.9064, 45.83698	38.89866, 45.84715	Iran
35	29.93814, 52.89342	29.93341, 52.89908	Iran
36	24.75318, 67.51852	24.74968, 67.52525	Pakistan
37	24.28927, 69.14827	24.27084, 69.17329	Pakistan

Table A.3: Fortress tilesets used for training; tileset ID in Github repository, top left and bottom right coordinates and location.

A.2 Areas/tilesets used in training and testing

Tileset ID	Top Left Corner	Bottom Right Corner	Location
38	39.66649, 44.11654	39.66066, 44.12459	Iran
39	39.70648, 44.0829	39.70156, 44.09023	Iran
40	37.06631, 49.23474	37.06256, 49.24423	Iran
41	35.05913, 51.42165	35.0474, 51.4374	Iran
42	34.0456, 51.05356	34.03204, 51.05931	Iran
43	33.97765, 51.43379	33.96704, 51.45171	Iran
44	35.70386, 53.49694	35.69168, 53.52179	Iran
45	35.88638, 53.50734	35.8707, 53.53318	Iran
46	37.37859, 47.15646	37.37262, 47.16588	Iran

Table A.4: Fortress tilesets used for testing; tileset ID in Github repository, top left and bottom right coordinates and location.