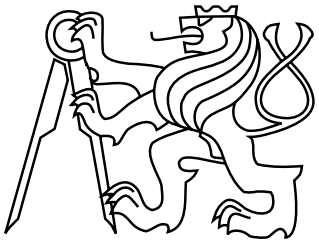
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF CIVIL ENGINEERING
SPECIALIZATION OF STUDY GEOMATICS



MASTER'S THESIS

MASK R-CNN IN GRASS GIS

MASK R-CNN V PROSTŘEDÍ GRASS GIS

Supervisor Ing. Martin Landa, Ph.D.
Department of Geomatics

2018 Praha

Bc. Ondřej PEŠEK

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební
Thákurova 7, 166 29 Praha 6

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: Pešek Jméno: Ondřej Osobní číslo: 423996

Zadávající katedra: Katedra geomatiky

Studijní program: Geodézie a kartografie

Studijní obor: Geomatika

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce: Mask R-CNN v prostředí GRASS GIS

Název diplomové práce anglicky: Mask R-CNN in GRASS GIS

Pokyny pro vypracování:
Cílem diplomové práce je návrh softwarového nástroje umožňujícího použití Mask R-CNN (Mask region-based convolutional neural networks) za účelem získání vektorových masek z rastrových dat. V praktické části se počítá s jeho implementací jako modulu do prostředí open source projektu GRASS GIS.

Seznam doporučené literatury:
Kaimeng He et al.: Mask R-CNN, 2017
Shaoqing Ren et al.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, 2015
Ross Girshick: Fast R-CNN, 2015

Jméno vedoucího diplomové práce: Ing. Martin Landa, PhD.

Datum zadání diplomové práce: 1.3.2018 Termín odevzdání diplomové práce: 20.5.2018
Údaj uveďte v souladu s datem v časovém plánu příslušného ak. roku

Podpis vedoucího práce Podpis vedoucího katedry

III. PŘEVZETÍ ZADÁNÍ

Beru na vědomí, že jsem povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je nutné uvést v diplomové práci a při citování postupovat v souladu s metodickou příručkou ČVUT „Jak psát vysokoškolské závěrečné práce“ a metodickým pokynem ČVUT „O dodržování etických principů při přípravě vysokoškolských závěrečných prací“.

Datum převzetí zadání Podpis studenta(ky)

ABSTRACT

The goal of this thesis is to develop software tools allowing the user to use Mask R-CNN (Mask region-based convolutional neural networks) in GRASS GIS. These tools allow the user to train his own Mask R-CNN model and use it to get vector masks from raster data. In the text of the thesis, the theory behind convolutional neural networks is introduced, followed by a list of their possible applications in the field of computer vision, a brief sketch of used technologies and is ended with a part dedicated to the implementation itself. The appendix contains a user manual and examples of usage.

KEYWORDS

GIS, GRASS GIS, Python, artificial neural networks, convolutional neural networks, Mask R-CNN, instance segmentation

ABSTRAKT

Cílem diplomové práce je návrh softwarových nástrojů umožňujících uživateli využití Mask R-CNN (Mask region-based convolutional neural networks) v prostředí GRASS GIS. Tyto nástroje zprostředkovávají možnost učit svůj vlastní Mask R-CNN model a aplikovat jej za účelem získání vektorových masek objektů z rastrových dat. V textu práce je nejprve nastíněn teoretický základ konvolučních neuronových sítí, následuje přehled možností jejich využití v počítačovém vidění, dále kapitoly o použitých technologiích, a uzavírá jej část věnovaná implementaci samotné. Přílohy obsahují uživatelskou příručku a ukázku výsledků dosažených za využití vytvořených modulů.

KLÍČOVÁ SLOVA

GIS, GRASS GIS, Python, umělé neuronové sítě, konvoluční neuronové sítě, Mask R-CNN, instanční segmentace

DECLARATION OF AUTHORSHIP

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged. Formulations and ideas taken from other sources are cited as such.

In Prague ....

(author sign)

ACKNOWLEDGEMENT

I would like to thank my parents for their support during my studies. Then I would like to thank Martin Landa, not only for supervising my thesis but also for the initial impulse in the direction to artificial neural networks and open source GIS generally. My thanks also belong to Margherita Di Leo for long initial discussions about the usage of neural networks in the field of GIS, to Moritz Lennert for testing and comments during the code sprint in Bonn, and to Luca Delucchi and Fondazione Edmund Mach for the willingness to exploit their time and resources to test modules.

Contents

Content of figures

Content of pseudocodes

1 Introduction

In the last years, the field of computer science is permanently shaken by one term: Artificial neural networks (ANNs). Some people perceive it almost as a magical formula. And even though ANNs are not a spell able to solve everything, they have wide applications. Their applications are in finance, data mining, language recognition, computer vision and many more.

Another term that can be heard more and more is *the information age*. The availability of computers and the growth of memory limits result in huge amounts of data.

The huge amount of data is a fuel for ANNs. One hundred years ago, an artificial intelligence was a phantasmagoria of science-fiction writers. Fifty years ago, it was an idea facing only derision. Twenty-five years ago, a design doomed to failure due to a lack of training data. Twelve years ago, a bold idea of few men. Five years ago, an earthquake in each branch of the field of computer science.

The data availability and their opening in the field of geomatics, as well as the computer performance, open doors for the usage of ANNs in geographic information systems (GIS). Results of a special type of ANNs called convolutional neural networks (CNNs) promises a lot in computer vision and therefore also in GIS, in tasks of detection and classification.

Chapter 2 will briefly introduce the theory behind CNNs. In the first part, the history and the motivation behind them will be described. The second part covers multiple types of layers used in CNNs. Few pioneering architectures will be covered.

Chapter 3 will introduce the term computer vision. Various tasks of computer vision will be named and few breakthrough architectures not mentioned in the CNN chapter will be described there. The research which concluded in the selection of Mask R-CNN as the architecture for the implementation will be depicted in this chapter.

Chapter 4 will describe some of the most important technologies used during the above-mentioned implementation.

The implementation will be the topic of chapter 5. It will summarize the motivation behind the architecture choose and code decisions, and then describe the uppermost parts of the code. The practical part of the thesis is exactly this implementation.

2 Convolutional neural networks

Although it is assumed that the reader has sufficient prior knowledge of ANNs and CNNs, this part briefly introduces convolutional neural networks, their layer types and few selected architectures.

For a better understanding of the topic, it is recommended to take a look at the holy book of deep learning, [13].

2.1 Introducing convolutional neural networks

If you try to find an introduction to CNNs on the internet, you may bump into a common statement that CNNs are neuroscience-based deep neural networks using convolution and presuming the input is an image. It is not exact.

Though images are the most common input, according to [13], CNNs presume the input has a grid-like topology; apart from the computer vision, other applications include for example natural language processing (as in [29]) or anything representable as a grid-like topology (audio waveform as 1-D grid, RGB images as multichannel 2-D, CT scan as 3-D, etc.).

A paradox inexactness is the term *convolution* as in mathematical meaning, many CNNs implement cross-correlation instead of real convolution. Cross-correlation may be seen as convolution without a kernel flipping. The reader can get more mathematical insight about the difference and harmlessness of this change from [13].

It is true that CNNs are based on a neuroscience. They are inspired by Nobel prize laureates Hubel and Wiesel's research on mammalian vision systems (firstly cats in [18] and [19], later monkeys in [20]). Hubel and Wiesel found that some neurons (sorted in columns) strongly respond to specific edge-like patterns but just a bit to other patterns.

The eye stimulus on the retina is transferred through the optic nerve and the lateral geniculate nucleus into the primary visual cortex (sometimes referred to as V1), a part of the visual cortex located in the posterior pole of the occipital lobe. The primary visual cortex is organized in a 2-D spatial map representing visual stimuli from the retina and contains two cell types, simple cells and complex cells. Simple cells purpose is to compute a linear function (although some counterarguments against

the linearity have been raised, see [6]) of the image in a spatially localized field, while complex cells operations are to some extent position and lighting invariant.

2.2 Layer types

While the common approach in image processing of classical ANNs is to vectorize the input, CNNs emulate the neuroscientific approach outlined in chapter 2.1 using feature maps (each colour channel is a feature map). CNNs profit from the fact that pixels in an image are ordered according to some structure. That allows neurons in layers to be connected just to the certain region instead of heavily arduous fully-connected architecture.

CNNs consist from many layers with different functions. In the following, individual layer types are briefly described.

2.2.1 Convolutional layers

The first layers through which the input is passed are convolutional layers. So, firstly, what is the convolution?

In the geomatics field, we very often encounter the term *kernel*. A kernel can be seen as a matrix (or a window) sliding across all the image pixels. The pixels contained in this window are a receptive field. As both the kernel and the receptive field are matrices of the same shape, in each position element-wise multiplication is computed and outputted as an output matrix element. Because after such a filtering, the output matrix contains a 2-D activation map (a map where each position values say with which probability the requested feature is on that position in the original image), the output is called a *feature map*. Kernels/filters are the subjects of training.

In case of stride 1 and without zero-padding, the feature map is naturally of shape $[original\_width - kernel\_width + 1] \times [original\_height - kernel\_height + 1]$. An example may be seen in figure 2.1.

With convolution, we reduce both computational requirements and a threat of overfitting by using local connections (representing weights) between input and output. However, these connections are local only in two dimensions, in width and height of the input; these connections have to be full along the depth of the channels - e.g. in RGB images, the last dimension of connections is always 3. Different

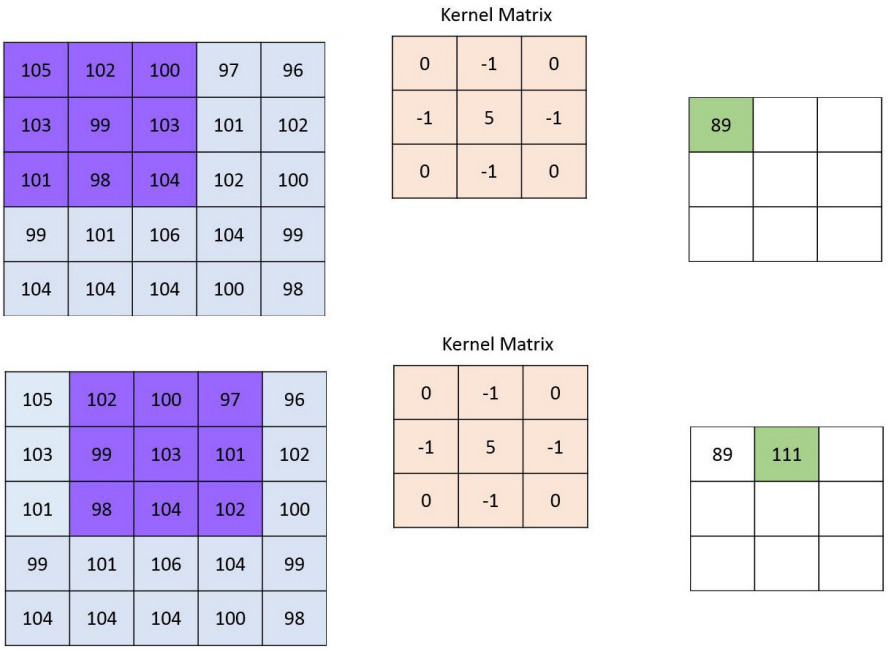


Figure 2.1: Two steps of kernel convolution, source: author

is it with the output; its third dimension (*depth*) is determined by the number of neurons referencing the same spatial location, e.g. by the number of kernels we use.

In chapter 2.1, translational invariance was mentioned. In convolutional layers, the first step to this invariance was achieved by another huge parameter reduction, by parameter sharing. The idea of parameter sharing raised from the premise that when one feature is useful in one location, it could be useful also in another one. This simple presumption which works apart from for example centred special structures allows sharing a set of parameters throughout the whole depth slice.

Using the parameters from [23] as an example, assume that the feature map has size $55 \times 55 \times 96$ and we apply it to images of size $227 \times 227 \times 3$[1] using kernels of size $11 \times 11 \times 3$. Sharing parameters within a depth slice, we can reduce the parameter amount from $55 * 55 * 96 * (11 * 11 * 3 + 1)$ to $96 * (11 * 11 * 3 + 96)$, where 1 and 96 are biases; it means from more than 105 million to less than 35 thousand. Speaking only about the first layer. I believe that this example said it all.

An inquiring reader may raise a question: In the chapter name, there is a plural. What happens in deeper layers?

[1]The paper claims that the images were of size $224 \times 224 \times 3$, but it is assumpted that it was either a typo or authors forgot to mention a zero-padding.

Their input is the previous layer output. The output of the first layer is the feature map of the lowest-level features. As was already mentioned, each neuron of the next layer is connected with some local neighbourhood and with everything along the third dimension; and because the third dimension of the output is formed by the stack of filters/kernels of the first layer, each second layer neuron is connected to all detected features in some location and its neighbourhood. The result? Output feature map from the second layer contains higher-level features (simple combinations of the low-level ones, like triangles or squares; combinations of some edges, curves, etc.). The next layer will output again higher-level features and in the end, we may have very specific features like cars, reflective heliports or art deco swimming pools. This principle is illustrated in figure 2.5 using a deconvolutional network, a technique described in chapter 2.3.3.

2.2.2 ReLU layers

Since the real data we are using to train our CNNs are mostly non-linear, it is useful to introduce some non-linearity into the network. In the past, functions like a hyperbolic tangent or sigmoid were used but Rectified Linear Unit (ReLU) has been found to be trained faster and mitigate the vanishing gradient problem (a problem of slow training of low layers due to the exponential decrease of the gradient).

ReLU output is defined by a function:

$f(x) = max(0, x)$

After applying the ReLU function to the input values, we have a feature map where all negative values of the input were changed to zero. This output is called *rectified feature map*.

2.2.3 Pooling layers

Even though the input size reduction might already be included in convolutional layers, it is very common to include other layers with this purpose. Because of this purpose, they are called *pooling layers* or *subsampling layers* (they can do both downsampling or upsampling).

Pooling layers work again with a kernel. But this time, the stride is bigger than one which is quite uncommon for convolutional layers. It means that when we use a pooling layer with a kernel of size 2×2 and stride 2, the output will be half in first

two dimensions (the third one is preserved). One pooling like this reduces parameters by 75 percent.

Aside from the parameter reduction, there are two more positive effects. Because of the detail mute, it reduces a threat of overfitting and it also strengthens the shift invariance. The advantage of pooling layers is also the fact that they do not introduce any new parameters to the network.

The function for the kernel can vary, but the most-used one is max-pooling[2] having the advantage that it does not matter where in the region was the value detected. Other customary approaches apply average (compared with max-pooling in [4]), $L2$ [24] or Stochastic pooling [44]. Also used kernel size and stride vary, the most common ones are 2×2 with stride 2 and 3×3 with stride 2.

Also, architectures without pooling layers are not so uncommon today. One research on this approach can be seen in [36]. It introduces an architecture called *all convolutional net* where the subsampling may be done by increasing the stride and compares it with other approaches.

2.2.4 Normalization layers

Besides the neural exhibition, a neural inhibition is also found in the human brain. These doors of perception stay half-closed and filter or inhibit human receptions.

Normalization layers can be seen as an attempt to imitate this structure, but there are more reasons for these layers in deep learning. As was written above, input for higher (deeper) layer is the output of lower level. It means that the highest (deepest) layer input is dependent on the first layer output. Because functions of layers are changed each training step, a small change of the first layer output may have a huge effect on the last layer input and therefore also on the last layer output, which may lead to completely wrong behaviour of deeper layers. This problem is called a *covariate shift* or, following terminology from [21], an *internal covariate shift*.

This change in the distribution can be to some extent reduced by using a small learning rate and right initialization of the network. Because small learning rate radically extends the training time, other solutions were needed. Normalization layers.

[2]Choosing the biggest value from those overlaid by the kernel.

One of the most widely used approaches is called *batch normalization* [21]. In CNNs, it is common to use batches (or mini-batches) of training examples instead of one-at-a-time as the computation parallelism saves time. Batch normalization computes a mean and variance over a batch using the distribution of the summed neuron input and whiten[3] it for each training batch. According to [21], it reduced the number of training steps 14 times allowing the user to use much higher learning rate and being less careful about initialization.

Other types of normalization layers include local response normalization or $L2$ normalization.

2.2.5 Fully connected layers

As their name prompts, fully connected (FC) layers are layers where each neuron in a layer is connected to each neuron of the previous layer. Their activations can be seen simply as a matrix multiplication enhanced by a bias.

The purpose of FC layers is to take the high-level feature map as an input and return a classification vector as an output. Each value in the output refers to one class occurrence, e.g. The length of the output vector is n where n is the number of classes. FC layers are not so hard to train to use non-linear combinations of features in input which is widely used whereas the combinations of high-level features are the things we are looking for. For example, if we are looking for a platypus, the last layer output will have high values in the neurons that represent things like a duck-like snout, four legs, flat tail or a calcaneus spur; if we are looking for a jelly, we will most probably not be interested in any of these features.

Using popular *softmax* classifier[4], the output is a vector of probabilities representing each class. Other classifiers like SVM can also be used.

Fully connected layers are sometimes referred to also as *dense layers*.

2.3 Architectures of convolutional neural networks

It is generally resolved[5] that the history of successful CNNs started in 1998 when Yann LeCun and his team published the paper Gradient-Based Learning Applied

[3]Set means equal to zero and variances unit; idea proposed in [30].

[4]Softmax transforms a vector of real-valued values to a vector of values between zero and one that sum to one.

[5]Roots of CNNs lie at the end of the eighties, but the breakthrough came in nineties.

to Document Recognition [25]. Although it is just twenty years, CNNs have made tremendous progress. During this period, a plenty of various architectures was proposed, more or less successful.

Because some background of the CNN architectures evolution could help in understanding CNN fundamentals and deepens the insight into research and decisions made for this thesis, few of the most influential architectures will be mentioned in the following chapters.

In this chapter, ImageNet Large Scale Visual Recognition Challenge (ILSVRC) will be mentioned several times as it is something like a fuel for computer vision progress with which are CNNs inseparably connected. For the topic of this thesis, it should be enough to say that ILSVRC is a competition in visual recognition including many diverse tasks, for more information and results, take a look into [33] which was used also during writing this chapter.

2.3.1 LeNet-5

Nobody would argue that the fundamental architecture is the one from the paper Gradient-Based Learning Applied to Document Recognition mentioned above, the one called LeNet-5. It was proposed in [25] and due to the lack of GPU and insufficience of CPUs, its convolutional, computationally economical approach ensured the architecture a huge success and usage for example in character recognition (reading zip codes, checks).

In figure 2.2, we can see the architecture of LeNet-5. It contains many features mentioned in chapter 2.2. It consists of convolutional, pooling (subsampling) and FC layers and it also introduced the non-linearity into the network using hyperbolic tangent or sigmoids. Its two convolutional layers (C1 and C3) apply 5×5 convolutional filters with stride 1 and its two pooling layers (S2 and S4) apply 2×2 average-based filters with stride 2. The architecture is finished with two FC layers; one is needed to learn the non-linear combinations of high-level features, the other one to classify the results[6].

LeNet-5 made the first important step for CNNs and a big step for ANN-based computer vision generally. It explained that taking an input as individual values,

[6]Gaussian connections are used here instead of softmax mentioned in 2.2.5. They are based on Euclidean Radial Basis Function and described in [25].

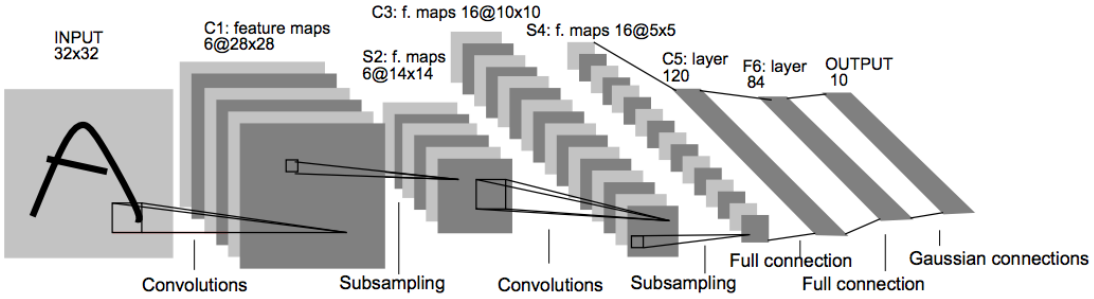


Figure 2.2: LeNet-5 architecture schema, source: [25]

besides its speed and computation problems, does not use the advantage of spatial correlations within the input.

2.3.2 AlexNet

As was mentioned, LeNet-5 became a very successful first step. Nevertheless, until the 2010s, development of CNNs was out of the main spotlight. Although there were few important events like Dan Ciresan using GPU neural nets in 2010, the main one happened in 2012, when Alex Krizhevsky came with something like a deeper version of LeNet in [23] and called it AlexNet. That year, AlexNet won the ILSVRC[7].

Krizhevsky benefited from years of progress bringing more computing power and more data to be trained on. This well-timed entree allowed him to use a deep architecture containing about sixty million parameters, ReLU layers, overlapping max pooling and dropout[8]. AlexNet also included a stack of convolutional layers instead of the previous approach, where each convolutional layer was followed by a pooling layer.

The computation was done on GPU (NVIDIA GTX 580) which allowed him to use larger datasets consisting of more and larger images as well as fastened the training. The separate GPU approach is indicated in figure 2.3 by upper and lower part of the image as two different GPU processes; it can be seen that the interaction between two GPUs happens only at certain layers.

[7]AlexNet won it by a great margin with top 5 error of 16.4 % compared to the second place with 26 %.

[8]A technique used to prevent overfitting proposed in [17]. Dropout selectively ignores individual neurons during training.

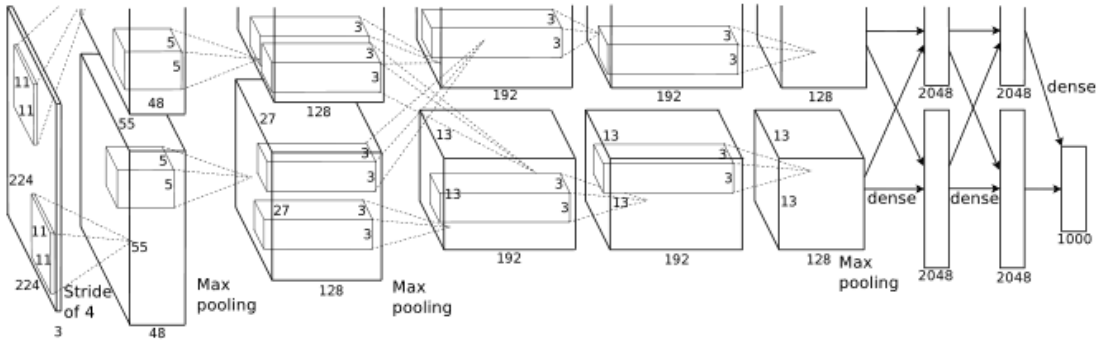


Figure 2.3: AlexNet architecture schema, source: [23]

If LeNet-5 represents the first step of CNNs, AlexNet is the first marathon. After its success, CNNs became the bleeding edge in both deep learning and computer vision.

2.3.3 ZF Net

After the success of AlexNet, the number of CNNs competing in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) noticeably increased. Matthew Zeiler and Rob Fergus built an architecture called ZF Net and won the ILSVRC 2013 with top 5 error rating of 11.2 %.

ZF Net architecture was based on AlexNet and described in [45]. Because it was believed that the 11×11 filtering in the first layer of AlexNet skipped too much information, it was replaced with a filter of size 7×7 with a decreased stride. Besides the parameter tweaking, the size of the middle convolutional layers expanded and the cross-entropy loss function was used as well as stochastic gradient descent with a mini-batch size of 128. The architecture can be seen in figure 2.4.

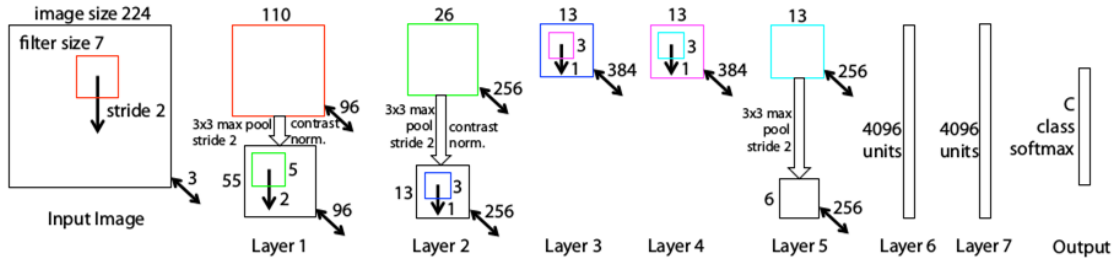


Figure 2.4: ZF Net architecture schema, source: [45]

An important feature was introducing a technique mapping features from feature maps back to pixels, due to its character this technique is called a *deconvolutional network*. During the forward pass, activations are computed at each level of CNNs; when we want to observe a certain feature of a certain layer, we pass it back through the preceding layers. In this back pass, operations are in another direction (pooling is changed to unpooling, downsampling to upsampling etc.) until the input layer is reached. It gives the user an idea of what kinds of structures are recognized in the certain feature map. A visualization of 5 layers illustrating the way from low-level features like edges or colours to high-level features like dog faces or flowers can be seen in figure 2.5.

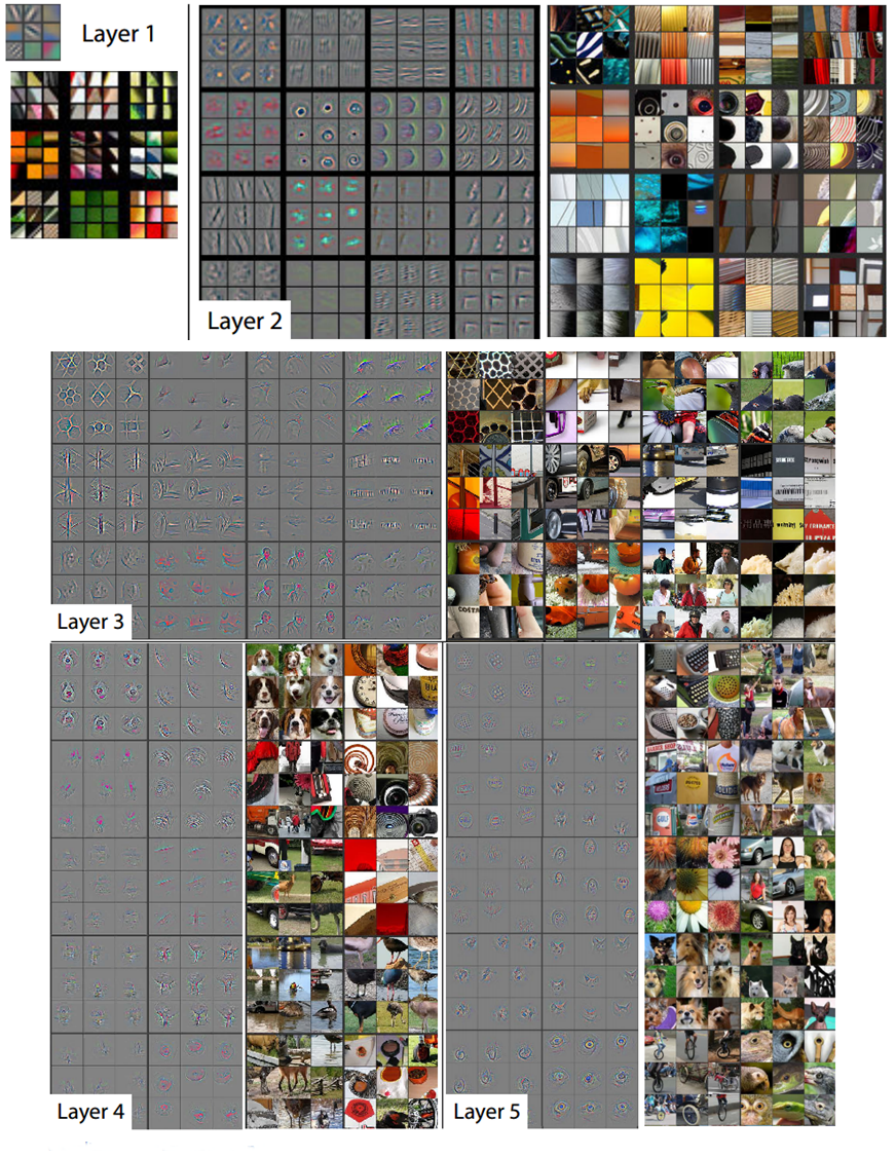


Figure 2.5: Visualization of 5 feature maps through deconvolutional networks, source: [45]

Thanks to its modifications, it was enough to train ZF Net on only 1.3 million images instead of 15 million images used with AlexNet. However, the training took 12 days instead of 5 or 6 and was stopped after 70 epochs. The training ran on a single GPU.

However, in [45], Zeiler and Fergus did more than just bring new architecture. They summarized why the time of CNNs had just come and tried to deepen the general knowledge behind these models, where especially the deconvolution visualization of feature maps can be called a missionary work.

2.3.4 VGG Net

ILSVRC 2014 brought new interesting architectures. Although VGG Net, an architecture proposed by Visual geometry group of the University of Oxford in [35], was not the winning architecture, it reached the error rate 7.3 % even with its *simplicity-in-the-first-place* architecture.

Authors experimented with different architectures with a number of layers between 11 and 19 (see figure 2.6 for their parameters) and chose 16-layer architecture homogenously using only 3×3 filters with stride and zero padding of size 1 to preserve the spatial resolution after convolution interleaved with maxpooling layers with stride 2. They found that when we use multiple convolutional layers with smaller kernels in a row, it emulates the effect of larger kernels while still retaining advantages of smaller kernels; it means that 3 layers with a kernel 3×3 emulate the effect of 1 layer with kernel 7×7, decrease the number of parameters and allows the user to implement three ReLU layers instead of one, exactly the advantage VGG Net used. It has to be said that the number of parameters was still enormous reaching almost 140 million and in the following architectures, this problem caused by FC layers had to be solved to reduce the time consumption.

VGG Net also enlarges the number of filters after each maxpool layer as can be seen in figure 2.6; the idea of decreasing spatial dimensions but increasing the third (depth) dimension was shown to be very important as well as the depth (number of layers) of the network. During the mini-batch gradient descent based training, a scale jittering was used for the data augmentation to train the model to recognize objects at different scales. The training took two to three weeks depending on the architecture and was carried out by 4 NVIDIA Titan Black GPUs.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Number of parameters (in millions)

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure 2.6: Different architectures of VGG Nets and the chosen one, source: [35]

2.3.5 GoogLeNet

And now for something completely different. Authors of GoogLeNet, the winning architecture of ILSVRC 2014 with a top 5 error rate of 6.7 %, proposed in [37] another approach, instead of at the architecture simplicity aiming at the computational simplicity.

In figure 2.7, we can see parallel blocks. Authors found that the sequential queue of layers increases a computational and memory cost a lot, so they proposed a module called *Inception*. The naïve idea behind the Inception module is illustrated in figure 2.8 and is quite simple: Why should we stack the layers in a sequence, when we can perform them in a parallel? Though the idea behind was not bad, this naïve

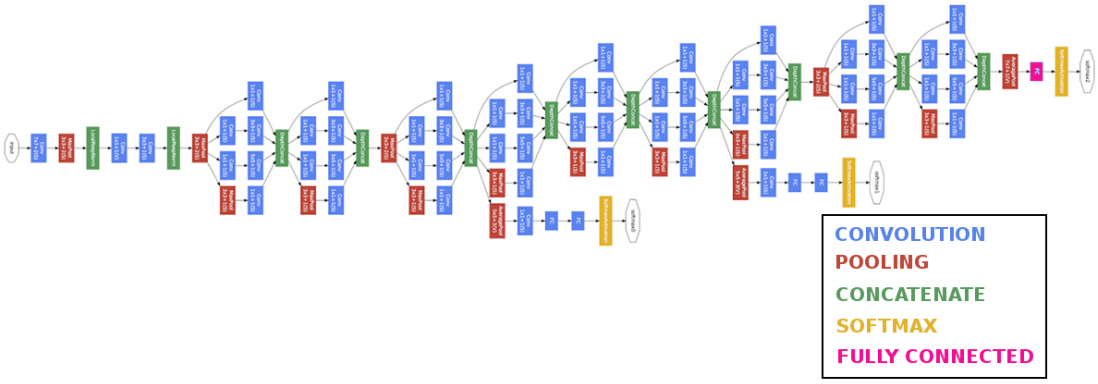


Figure 2.7: GoogLeNet network schema, source: [37]

version ended in an enormous depth (size of the third dimension) of the output after the concatenation into a single vector.

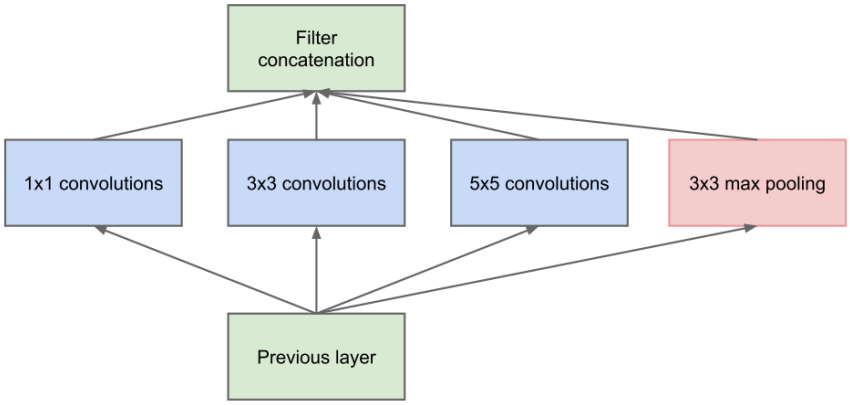


Figure 2.8: Naïve idea of Inception module, source: [37]

As can be seen in figure 2.9, authors solved the depth problem by adding 1×1 convolutional layers, sometimes referred to as *network in network*[9]. Usage of 10 1×1 filters outputs a volume with two dimensions equal to the input dimensions and the third one equal to 10. The depth of input for bigger kernels is thus *pooled* to the defined size while allowing also to use one more ReLU layer after the 1×1 convolution. This process is sometimes called a *bottleneck* (likewise the 1×1 layer is sometimes called a *bottleneck layer*).

[9]The alias of this approach was derived from the architecture called Network-in-network proposed in [26] and presenting the advantages of 1×1 convolutions.

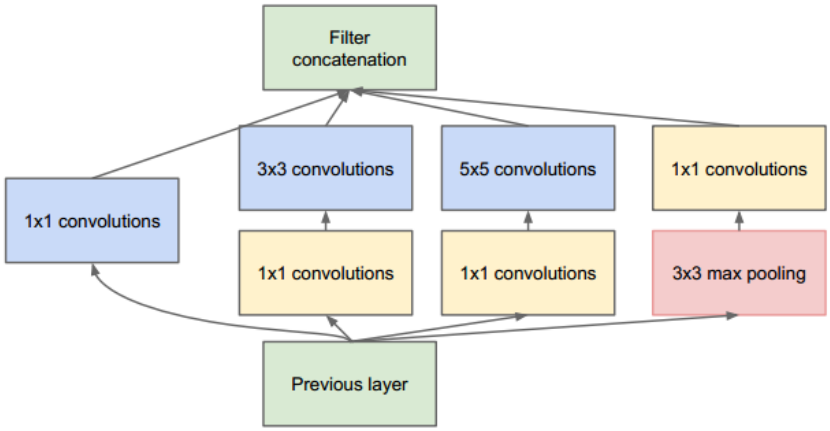


Figure 2.9: Inception module with dimensionality reduction, source: [37]

The parallelized architecture used over 100 layers while the real depth of the full network was just a fraction of this number and it used only 7 Inception modules. Authors also get rid of unnecessary FC layers and instead used an average pooling which concluded in twelve times fewer parameters than AlexNet. Their architecture also decreased the threat of overfitting. In [37], authors claimed that the network was trained *using few high-end GPUs within a week*.

Time from time, updated versions are published. Interested readers may try to find architectures like Inception V2, Inception V3 and higher.

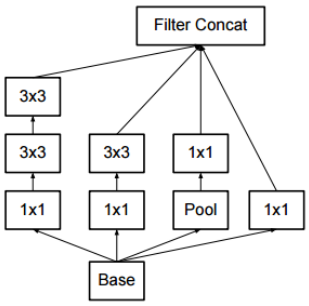


Figure 2.10: Inception V2 used the idea described already in chapter 2.3.4 about replacing 5×5 convolution with two 3×3 convolutions, source: [38]

2.3.6 ResNet

In the above-described architectures, a trend of going deeper can be noticed. Microsoft Research team noticed it too and in [15], they proposed much deeper architecture than previous ones. This architecture was called ResNet, contained 152 layers and

won ILSVRC 2015 with an error rate of 3.6 % beating even humans with their error rate of circa 5 – 10 %.

Although the depth was quite revolutionary, it was not the most innovative thing about ResNet. Neither was the usage of batch normalization as described in [21] after each convolutional layer. The most innovative thing of ResNet was a structure called a *residual block*. In 2011, Pierre Sermanet and Yann LeCun proposed in [34] a notion to *bypass* a layer. ResNet used this design with a richer mind and as can be seen in figure 2.11, they bypassed two layers.

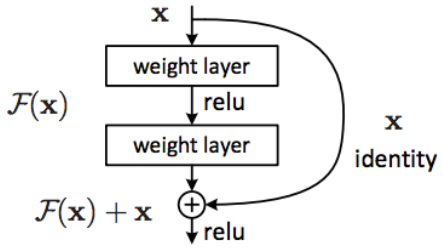


Figure 2.11: A residual block schema, source: [15]

Bypassing two layers (a term *skip connections* is not uncommon) gives much better improvements than the single layer bypass. But what exactly happens during the bypass? In traditional CNNs, only the $F(x)$ is computed. It means that the next layer does not have the real connection to the original input, but only to this transformed output, $F(x)$. When we bypass the block input x after 2 layers, we can add it to $F(x)$ which represents a change this time. By the addition, we get a mildly altered representation of the input. Authors of ResNet declared that it is easier to optimize this referenced mapping instead of the unreferenced one. Another advantage is that with addition operations, the backward propagated gradients will flow easier through the structure. Because the mapping was referenced using residuums, it is sometimes referred to as a *residual mapping*.

Authors experimented with diverse depths of the architecture counting 18, 34, 50, 101 and 152 layers. During these experiments, authors found a problem in the number of parameters in deeper architectures. However, authors found a solution by following the design of bottleneck layers described in chapter 2.3.5: They redesigned the residual block into a bottleneck block. This block reduced the number of features usually to approximately one quarter by using 1×1 convolutions and was implemented into ResNet-50 and higher.

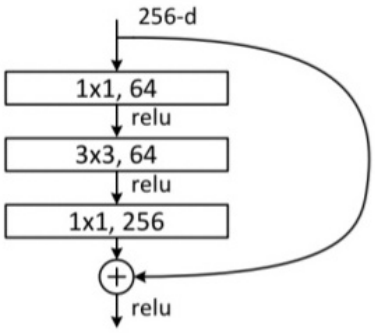


Figure 2.12: A bottleneck block schema, source: [15]

Mentioned experiments contained one more interesting event. A creation of a real monster, a 1202-layer network. Surprisingly, this network got a lower accuracy during tests. Authors argued that this is because of overfitting.

8 GPUs were used for the training of ResNet-152 and the process took something between two and three weeks.

The continuation of ResNet can be found for instance in an architecture called ResNeXt. Saining Xie and his team proposed ResNeXt in [42] and it combines ResNet with modularized parallel pathways similar to those described in chapter 2.3.5.

3 CNNs for computer vision

A paper Visualizing and Understanding Convolutional Networks by Matthew D. Zeiler and Robert Fergus [45] started with two sentences: *Large Convolutional Network models have recently demonstrated impressive classification performance on the ImageNet benchmark. However there is no clear understanding of why they perform so well, or how they might be improved.*

I tried to disperse such clouds a bit in the previous chapter, but now I would like to focus on another undertone connected with those statements. On their applications in the computer vision.

Almost everything mentioned in chapter 2 was already tied to the computer vision. The following text will briefly describe the field of computer vision itself and then introduce few tasks in that field connected with the topic of the thesis. In each task, the models that were considered during the research for the practical part of this thesis - an implementation into GRASS GIS - will be mentioned. Models are also mentioned and described to depict their evolution concluding in the selected one.

3.1 Understanding computer vision

When you see a group photo, you can easily count the number of people in the photo, you can say whether they are smiling or not, whether they are happy, sad, angry, you can even guess whether they are one family, a bunch of friends, colleagues or just random people passing by. You can do all of that in a fraction of a second without any effort. The computer vision is supposed to be a computer-aided version of this human cognition. Or in a fancier way, from [5]: *Computer vision is the transformation of data from a still or video camera into either a decision or a new representation.* The new representation might be for instance a colour shift, the decision an answer to a question like *Is there any football field in the picture?*

However, the claim that it is easy for humans does not implicate any simplicity also for computers. As is generally known and was indicated in chapter 2.1, the human brain is an extremely complex tool. The other thing is that a computer receives a visual impulse (image) in another way as is illustrated in figure 3.1. Where human sees a side mirror, a computer sees a grid of numbers. And this grid may be

completely different when the daytime, viewpoint, brightness, background or scale changes.

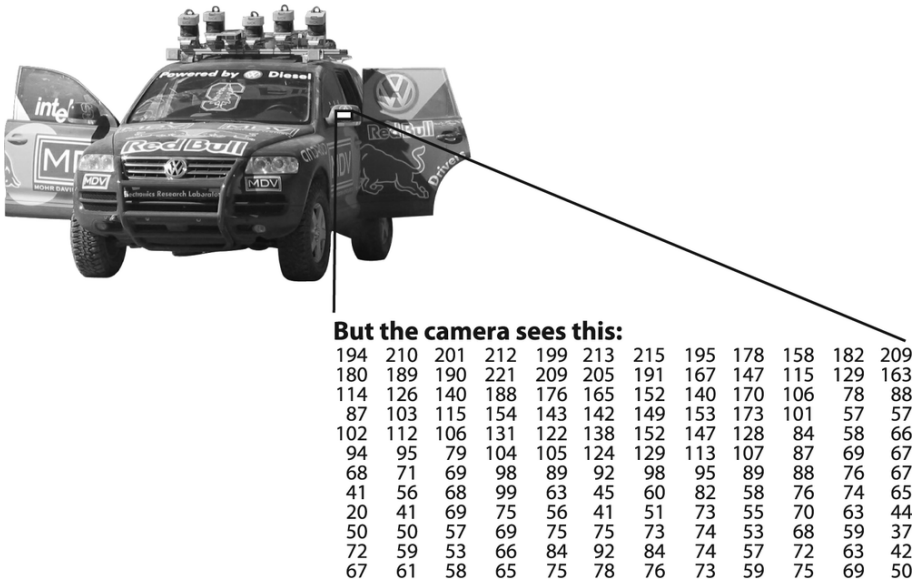


Figure 3.1: The difference between human and computer cognition, source: [5]

The task may be something like *Is there any side mirror in the picture?* This kind of tasks can be seen in the field of computer vision daily and can be extremely difficult to solve in the computer way. Although the input may include some metadata, it still has to be solved in a strict mathematical way. And because our goal is to make our computer vision system *perceive like a human*, it looks like the right place for CNNs.

Although we will focus only on classification and connected tasks like detection and segmentation, there are many more applications of the computer vision. To name a few: Autonomous cars, face recognition, fingerprint recognition, motion capture, biometrics and remote sensing.

To get a better view into the field of computer vision, it is recommended to read a richer source like [39] and [5] to get some practice.

3.2 Classification

The idea is simple, image classification is the task of assigning an image to one class. It means that the user is interested in the result of a guess of what the image contains. A good example is a camera trap. When the user is interested in moose,

he can train his model to predict moose and automatically filter all outputs from the camera trap through this validator.

The idea of training a model to sort a list of numbers is quite straightforward. Due to differences in pictures of the same object caused by influences mentioned in chapter 3.1, the classification task needs a bit of oblique, but still strictly logical thinking. It needs a data-driven approach. Instead of defining how exactly should moose look like, we feed the model with a bunch of labelled examples. It is the same process as with human children.

However, the number of classes does not have to be equal to one. The user can have a set of multiple classes and even his classifier may differ. A popular simple classifier is a binary classifier returning just 1 or 0 (representing True or False/Yes or no) for each class per image but much more widely used one is a softmax function giving a vector of probabilities for classes. This classifier also somehow more represents the human mind, as we may be unsure if there is a moose or a wapiti in the picture if it was taken with bad conditions.

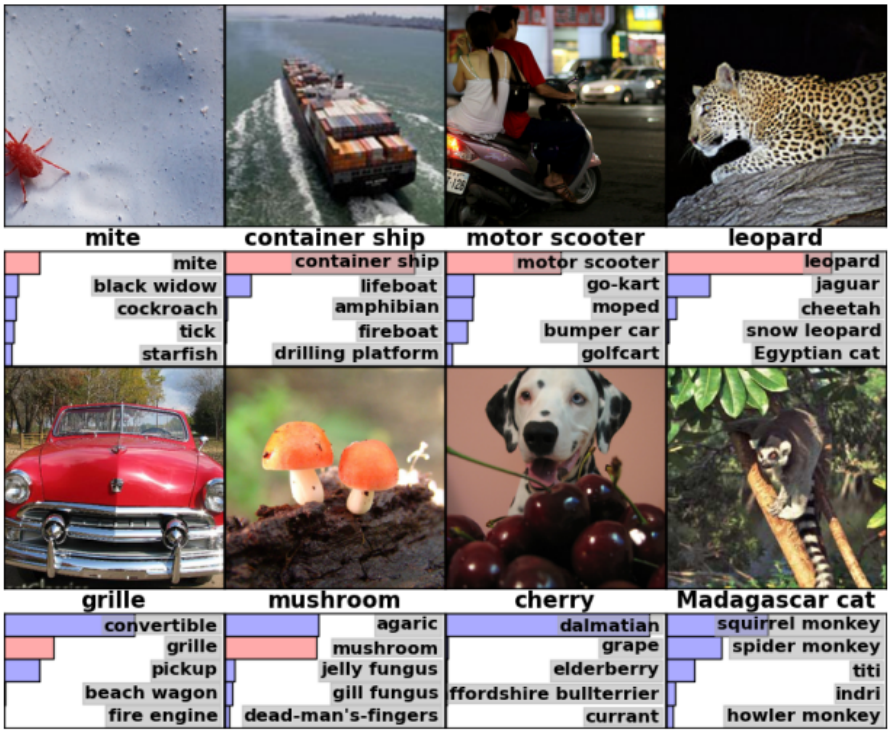


Figure 3.2: An example of the classification output with softmax function, source: [23]

The pioneering work in the field of CNN-based classification is AlexNet proposed in [23] and described in chapter 2.3.2. Another interesting implementation is the CNN-RNN framework for multi-label image classification[10] proposed in [41].

3.3 Classification with localization

Although classification with localization is sometimes ignored in similar lists as a subtype of object detection, I believe it is useful to mention it as a step between pure classification and object detection.

Classification was already explained in the previous chapter. Classification with localization uses classification in its one-class form together with bounding boxes (bounding boxes may be multiple). The goal is to draw the bounding box as tight around the object as possible and predict the class of the object, so the user ends up with two outputs:

- **Class:** Class label
- **Bounding box:** Box in the image defined by two coordinates and its width and height.

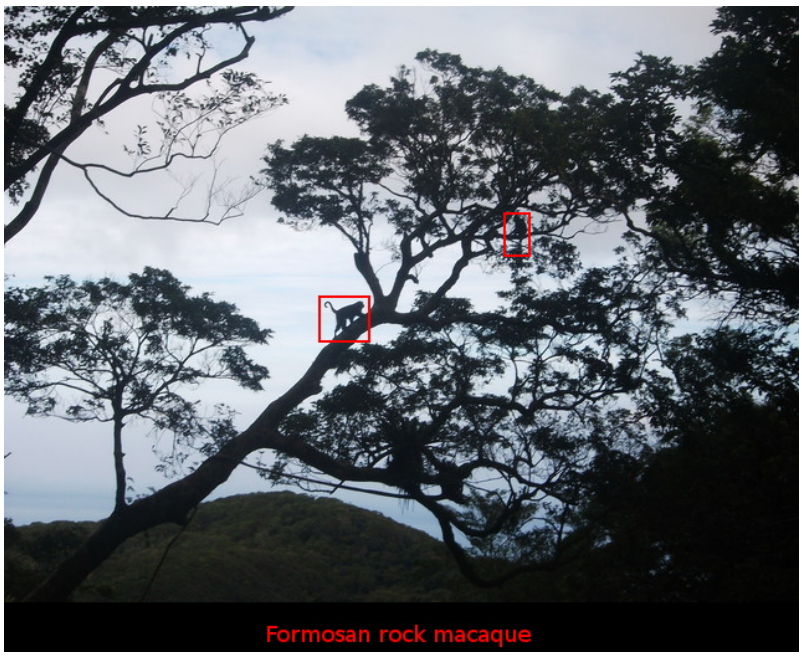


Figure 3.3: An example of classification with localization, source: author

[10]Real world images often contain more features; multi-label image classification tries to predict more than just one of them. The problem of one-label classification can be seen in the seventh image in figure 3.2.

Because multiple values are returned, this task can be considered as a kind of a regression problem. Outputs from the regression are enough to get a result as the one illustrated in figure 3.3.

3.4 Object detection

Object detection is the classification with localization for multiple classes.

Shared basics with the task from the previous chapter instigate to use the similar approach but applied to every single class individually. However, this method could be very, very inefficient. Different architectures use different practice to solve it and few of them will be presented here.

3.4.1 R-CNN

The Region-based convolutional neural network (R-CNN) is a model proposed in 2014 in [12], which combines region proposals with convolutional neural networks to detect objects in an image via bounding boxes.

The first step of the detection and also an answer to individual passes of classes is to generate category-independent region proposals containing probable objects. Instead of the whole image, those proposals are passed to a deep convolutional neural network which returns a feature vector for each region proposal. The last step is to pass this vector through a set of class-specific linear support vector machines (SVMs).

Although diverse methods may be used for the region proposals generation[11], authors of the original R-CNN paper decided to use the selective search. The selective search was proposed in [40] and mixes advantages of both an exhaustive search and segmentation. From the exhaustive search, an effort to catch all possible object locations is used; from segmentation, the idea of following the image structure to guide the sampling process is used. To deal with many diverse image conditions, the selective search uses a variety of complementary image partitions.

After regions proposition, their features must be computed. Because the network works with a fixed-size input, the region must be warped into 227×227 square RGB image, and then it can be forward propagated through an architecture composed of

[11] In the case of interest see an *objectness measure* [3] or *category-independent object proposals with diverse ranking* [8].

five convolutional layers and two fully connected layers (a modified architecture of AlexNet, see chapter 2.3.2).

The third step is scoring those extracted features (deciding which class the feature is and whether it is any class at all). This is done for each feature separately using an SVM trained for that class.

In general, the bounding box has a large overlap with the background. This is the last problem with which the algorithm has to deal. To improve the localization (make bounding boxes tighter), a linear bounding box regression method proposed in [10] is used with one modification - R-CNN applies regression on features computed by the CNN instead of on geometric features computed on the inferred deformable part model (DPM) locations.

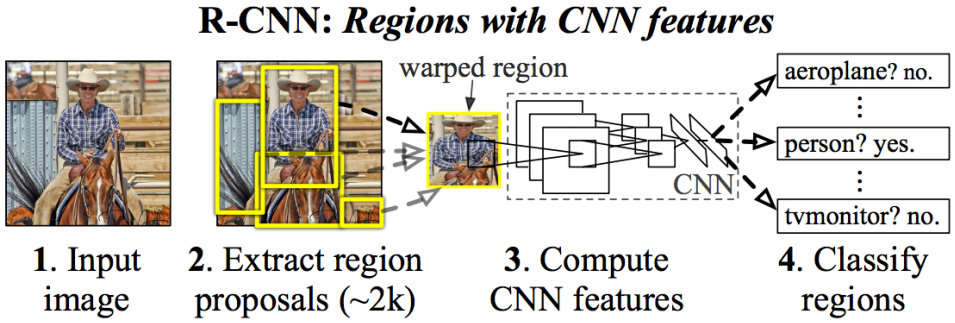


Figure 3.4: R-CNN architecture schema, source: [12]

Although R-CNN outperformed similar architectures[12], there were still some shortcomings. The biggest one was the slowness. It is caused mainly by three elements - the forward propagation through CNN (each region of every image must be passed separately), by its triplicated training (a network for generating image features, a network for the class decision and the bounding box regression model) and by the generating of bounding box proposals.

3.4.2 Fast R-CNN

Then, in the year 2015, a new architecture came to solve first two of these issues. Because the main reason for a new architecture was to speed-up R-CNN, Ross Girshick named his new architecture proposed in [11] simply Fast R-CNN. Besides

[12] [12] claims a mean average precision (mAP) of 53.3%.

the main purpose to avoid first two issues mentioned in chapter 3.4.1, it also improves its accuracy[13].

The first issue, the separate forward propagation for each region proposal, was solved by propagating the entire image to obtain a feature map before the region proposition. For each object proposal is from the feature map extracted a fixed-length feature vector by a region of interest (RoI) pooling layer.

The RoI pooling layer can be seen as a one-level spatial pyramid pooling layer, a max pooling-based downsampling algorithm proposed in [14]. Its purpose is to decompose separately each valid RoI into a fixed size 7×7 feature map. The decomposition is made by quantization each RoI to the rounded discrete grid which is filled using max pooling on the corresponding kernel of the feature map.

These feature vectors are inputs for a set of FC layers. This propagation is followed by the last, branched step, where two different outputs are obtained depending on the last layer - class probabilities from a softmax function and a bounding box defined by 4 values from a regressor.

There can be seen also a solution to the second problem named in chapter 3.4.1. Instead of three separate models, all steps are joint into only one model by appending classification (using a softmax layer instead of a separate SVM) and bounding box regression as parallel layers to the end of the model.

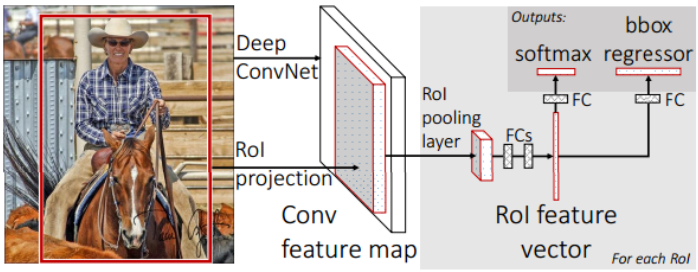


Figure 3.5: Fast R-CNN architecture schema, source: [11]

There can be raised a question whether is the SVM replacement with the softmax layer as accurate as the original approach. According to tests performed in [11],

[13][11] claims a mAP of 66% on Pascal Visual object classes (VOC) 2012. To find more info about the Pascal VOC datasets and challenges, please see [9].

the softmax layer is even slightly outperforming SVM by 0.1 to 0.8 mAP points depending on the depth of the network[14].

3.4.3 Faster R-CNN

The third speed issue mentioned in chapter 3.4.1, the region proposer based on the selective search, was solved in the year 2016 in an architecture imaginatively named Faster R-CNN, proposed in [32].

Microsoft Research team found that the feature map computed in the first part of Fast R-CNN can be used to generate region proposals instead of using slower selective search algorithm. Authors did it by including Region Proposal Network RPN after the feature maps extraction of Fast R-CNN.

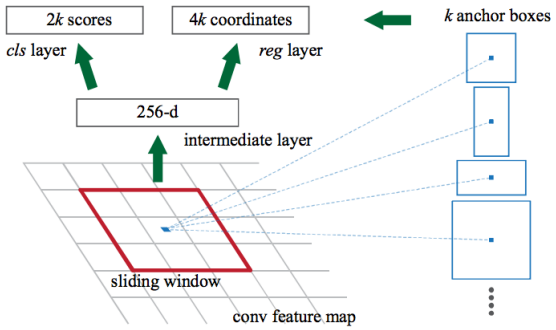


Figure 3.6: RPN schema, source: [32]

RPN approach is different than the ones used in other architectures. Instead of pyramids of images or filters, RPN uses anchor boxes - a set of rectangular bounding boxes proposals and scores created by sliding a spatial window over the entire feature map. The sliding window is a $n \times n$ fully convolutional network. The anchor boxes are defined by a scale and aspect ratio, so they are of different shapes. Powerful attributes of anchor boxes are that they are translation-invariant and multi-scale, which concludes also into a reduction of the model size.

[14]Different models were tried. Surprisingly, with a deeper network, smaller mAP difference was noticed.

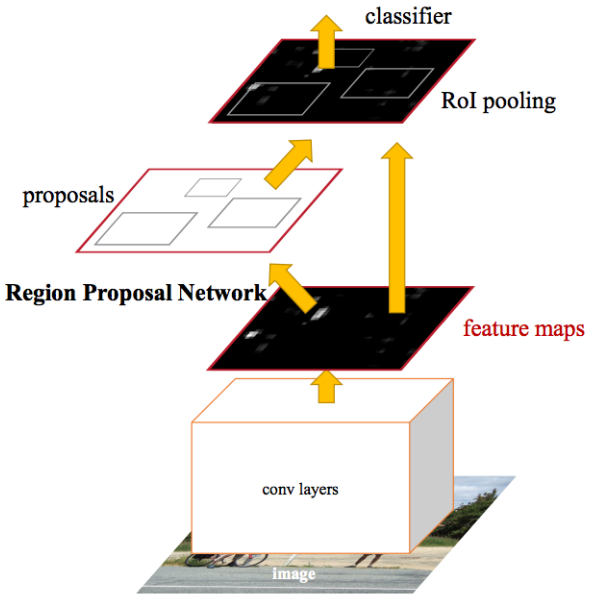


Figure 3.7: Faster R-CNN architecture schema, source: [32]

As can be seen in figure 3.7, the training scheme alternates between training (or fine-tuning) for the region proposition and for object detection. This approach with shared convolutional features quickly converges.

Besides the speed improvement, Faster R-CNN improves also the accuracy. [32] claims a mAP of 73.2% on Pascal VOC 2007 and of 70.4% on Pascal VOC 2012.

3.5 Semantic segmentation

Semantic segmentation is a labelling of each pixel in an image to a certain class without differentiating between instances of objects.

The first conception in our minds is to slide the kernel across an entire image and classify each pixel individually. However, I think that everyone surmises that this conception is not very efficient.

Two terms are connected with semantic segmentation, *encoder* and *decoder*. While the encoder is a classification network as one of those described in chapter 2.3, the decoder is a network projecting a lower resolution (a feature map) to a higher one (pixel space of the original image size), e.g. The task of the decoder is to recover the spatial information lost in the encoder. Different versions of decoders were proposed.

3.5.1 Fully convolutional network

The Fully convolutional network (FCN) was proposed by Jonathan Long and the UC Berkeley team in 2015 in [28].

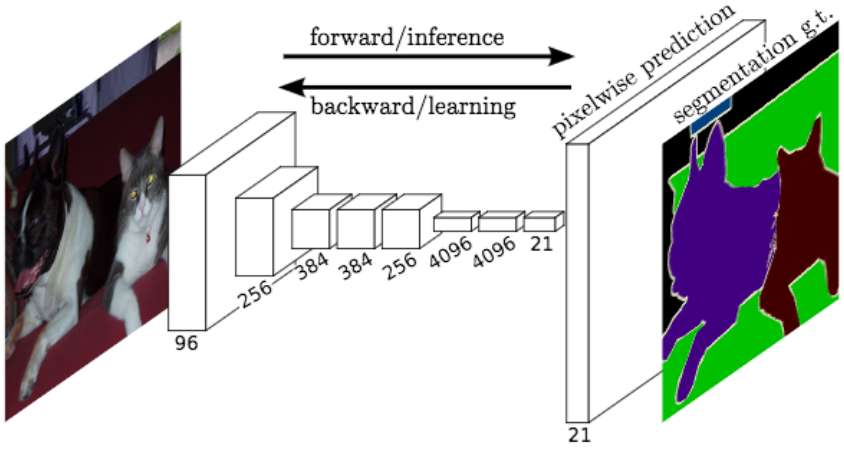


Figure 3.8: Semantic segmentation with FCN schema, source: [28]

Imagine for example VGG-16 as the backbone architecture, but the same process can be applied to any other classification architecture. Following the approach of [28], cut off the last classification layer and convert the fully connected layers into convolutional ones. 1×1 convolution was appended to predict scores for each class.

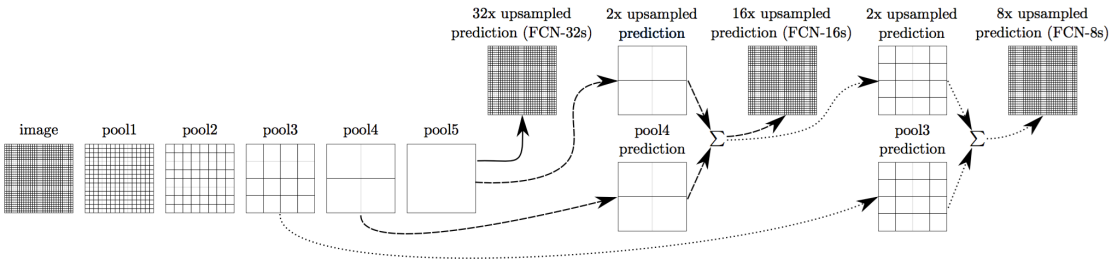


Figure 3.9: FCN skip connections schema; only pooling and prediction layers are shown, source: [28]

Everything must be then passed through the decoder; the decoder is here represented by a backward convolution, by a *deconvolution*. Deconvolution consists of upsampling using a bilinear interpolation. With a few of deconvolutional layers, the network can learn even a nonlinear upsampling. When using more layers, it is useful to fuse the output of each layer with predictions computed in the backbone

layer with corresponding resolution using similar bypass (skip connections) as that described in chapter 2.3.6.

3.6 Instance segmentation

Instance segmentation can be seen as a combination of object detection and semantic segmentation; the task is in detecting all instances of different objects and marking their pixels. In other words, the difference between semantic segmentation and instance segmentation is that instance segmentation allows marking two objects of the same class separately.

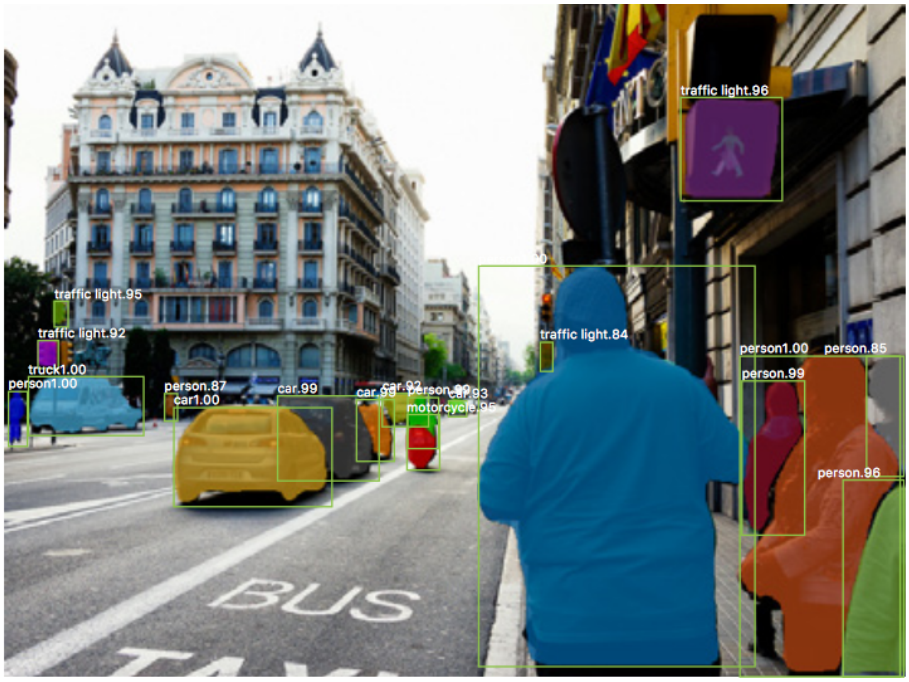


Figure 3.10: Instance segmentation example, source: [16]

3.6.1 Mask R-CNN

So we had R-CNN, Fast R-CNN and Faster R-CNN. What could be the next step? The first answer that comes to your mind is wrong - instead of expected *The Fastest* R-CNN, the Mask R-CNN was proposed in the year 2017 by Facebook AI Research FAIR in [16].

FAIR stood in a front of another question. According to [32], Faster R-CNN outperformed most of their competitors in the field of object detection. But it was still unusable for some users looking for a semantic segmentation neural network.

They had a powerful network for object detection, so the question was: Is there any way to use the current network for the semantic segmentation?

Obviously, the answer was *yes*. But instead of the semantic segmentation, they proposed a more advanced approach, the instance segmentation. To implement the instance segmentation with sufficient accuracy, there was a need for some changes in the architecture of Faster R-CNN. Following the original terminology from [16], in the following part, I will distinguish between the backbone architecture for feature extraction and the head architecture for classification, bounding box regression and mask prediction. Those will be described in following parts of this chapter along with an extra focus on a RoIAlign, a new approach in the RoI generation.

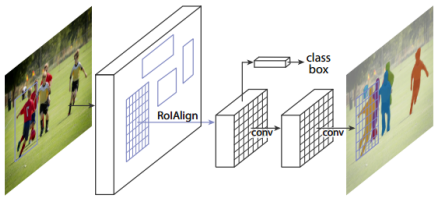


Figure 3.11: Mask R-CNN architecture schema, source: [16]

Backbone architecture

For the backbone architecture, more models can be used, but in the original paper [16], ResNet was used. ResNet was already described in chapter 2.3.6.

Moreover, authors experimented also with ResNet extended by a feature pyramid network (FPN), a top-down architecture with skip connections (in the original paper called *lateral connections*) developed for building high-level semantic feature maps at different scales proposed in [27].

FPN uses the fact that by subsampling, we obtain a different spatial resolution feature hierarchy with multi-scale, pyramidal shape. To achieve strong semantics at all scales, FPN uses bottom-up, top-down pathways and skip connections to create predictions independently on each pyramid level. The feature hierarchy is created by propagating an image through the convolutional network (bottom-up pathway). There are stacks of layers producing feature maps of the same size; they are thought of being in the same stage. One level of the FPN is created from the output of the last layer of each stage except the first stage due to its memory claims. The top-down pathway is done by upsampling feature maps from higher levels by a factor of 2 and

using nearest neighbour upsampling, and then enhancing them via skip connections with features from the bottom-up pathway as can be seen in figure 3.12.

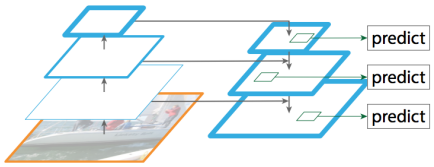


Figure 3.12: The FPN schema, source: [27]

Head architecture

In the head architecture, there was the most important change to allow the instance segmentation - including the mask branch in parallel with branches for classification and bounding box regression.

The mask branch is a pixel-to-pixel FCN predicting a mask individually for each RoI, where the mask is a binary matrix of ones (object location) and zeros (elsewhere).

In figure 3.13, we can see two implementations of the head architecture. The left one is based on ResNet-C4 (4 stage ResNet) and extends it with the compute-intensive fifth stage, the right one is based on FPN which already includes the fifth stage. In these implementations, the last convolution is 1×1 and the other ones are 3×3 and are the ones with preserved shapes; deconvolutions are 2×2 with stride 2 and increase shapes.

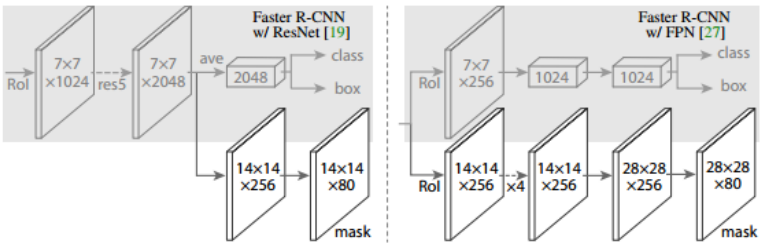


Figure 3.13: Different Mask R-CNN head architectures schema, source: [16]

RoIAlign

In figure 3.11, we can see a RoIAlign layer instead of Fast R-CNN RoI pooling from figure 3.5.

This change was needed because of spatial inaccuracy of RoI pooling due to the fact that it was not intended to be used for a pixel-to-pixel alignment. This inaccuracy is caused by quantizations and roundings and RoIAlign avoids it simply by skipping the rounding. To compute the value of the input, a bilinear interpolation at regular sampled locations (usually 4) is used.

According to [16], RoiAlign improved Mask R-CNN mask accuracy by relative 10 % to 50 %.

4 Used technologies

This chapter briefly introduces the most important technologies used during the development of modules for GRASS GIS. It means that besides GRASS GIS itself, the Python language and its libraries TensorFlow and Keras will be introduced.

4.1 GRASS GIS

Figure 4.1: GRASS GIS logo, source: `https://grass.osgeo.org/download/logos/`

The history of GRASS GIS (an acronym for Geographic Resources Analysis Support System Geographic Information System) started in the year 1982 at the U.S. Army Corps of Engineers Construction Engineering Research Laboratory, but the version 1.0 under the name GRASS was released later, in 1985. However, its way lead from the national project to the international Open GRASS Consortium in the mid-1990s. This consortium is perceived as an ancestor of today's Open Geospatial Consortium (OGC). In 1999, GRASS GIS was published under GNU General Public License (GPL).

GRASS GIS has grown to a cross-platform free and open-source GIS maintained by an international team of developers and users and licensed under the GNU GPL license allowing users to perform geospatial data management and analysis for both raster and vector data, image processing, geocoding and visualization. Within its more than 400 modules, it supports also spatio-temporal data.

For more information about GRASS GIS, please see its official website[15].

[15] `https://grass.osgeo.org/documentation/general-overview/`

4.2 Python

Figure 4.2: Python logo, source: `https://www.python.org/community/logos/`

Python is a cross-platform open-source interpreted programming language. Python was invented by a Dutch programmer and Monty Python's Flying Circus fan Guido van Rossum in the year 1991. Due to its user-friendly and readable structure, it is more and more popular.

It is also one of the most popular languages in the field of (nay) open-source GIS. Many modules for GRASS GIS are written in Python and Python is the most used language in QGIS[16] plugins. Also libraries like Fiona[17] or RasterIO[18] are Python-based, and GDAL[19] has its Python API.

For information about programming in Python, please see [31].

4.3 TensorFlow

TensorFlow is an open-source software library developed by Google Brain Team[20] primarily for the purpose of neural network research. TensorFlow uses data flow graphs, where a node represents a mathematical operation and an edge represents a tensor (a multidimensional array). A toolkit for visualization of these graphs is called TensorBoard.

TensorFlow is available for Python and C++ programming languages. Its usage optimizes mathematical expressions and as it was developed for the purpose of neural

[16] `https://qgis.org/`
[17] `http://toblerity.org/fiona/`
[18] `http://rasterio.readthedocs.io/`
[19] `http://www.gdal.org/`
[20] `https://research.google.com/teams/brain/`

Figure 4.3: TensorFlow logo, source:
`https://www.tensorflow.org/images/tf_logo_transp.png`

networks, the main focus was dedicated to this field and TensorFlow provides a lot of functions useful for deep learning.

A good starting point is the book Getting Started with TensorFlow [43]. A documentation and examples can be found also at the official website[21].

4.4 Keras

Figure 4.4: Keras logo, source: `https://keras.io/`

Keras is an open-source software library written in Python and using TensorFlow, Microsoft Cognitive Toolkit, Theano or MXNet as a backend tensor manipulation library.

Keras is again developed for the purpose of deep learning. Its powerful feature is the content of extensible objects for defining different types of layers (see chapter 2.2), models, loss functions and other tools widely used in the field of ANNs.

For a wider documentation, please see the official website[22].

[21] `https://www.tensorflow.org/`
[22] `https://keras.io/`

5 Implementation

In the field of GIS, users are dealing with a huge amount of satellite photos. Though there are many algorithms for automated classification, ANNs are a promising child which is just growing up and hogging the limelight. However, the younger sibling of ANNs is even more promising. The sibling is called CNNs.

CNNs for image processing are very strong in an object detection. The object detection in aerial photos could be very useful for example for detecting sick trees in an orthophoto of a forest or for a vehicle detection (aeroplane, ship, car, etc.). It means elements that can be represented as points. But what about searching for forests? Searching for lakes? Rivers? Oil spills? Do we really want to represent them in GIS as points in their centre of mass?

It seems that better representation is in polygons (or lines). But representing a detection bounding box as a polygon square does not seem like the right way. So there are two possible ways - semantic segmentation and instance segmentation. Three arguments were raised and discussed on behalf of instance segmentation:

- The instance differentiation could be very useful for some tasks.
- It is easier to create masks only for requested objects.
- If each pixel in a raster input will be assigned to a class, we got *semantic segmentation.*

When work on this implementation started, the paper on Mask R-CNN was just recently published ([16], March 2017). With its approach, it was a kind of bleeding edge. However, its results spoke for themselves and within a year of its publication, it was implemented in few projects following the success of the original paper. Few followers also appeared (for example [7]) and it could be interesting to examine them deeper from a geomatic point of view.

For the implementation to GRASS GIS, the Python language was chosen. There are few reasons for this decision:

- Although some GRASS GIS modules are written in C and C++, many of them are written in Python.
- Python allows me to use libraries such as TensorFlow and Keras, both very useful and widely used in the field of deep learning.

Mask R-CNN tools created for the practical part of the thesis consist of two modules. `i.ann.maskrcnn.train` allows the user to train a Mask R-CNN model on his own dataset, `i.ann.maskrcnn.detect` to use that model to detect features in georeferenced files. Both modules are licensed under GNU GPL 2 license.

Along with these modules, a library of Mask R-CNN tools was designed. This library is heavily based on a Python implementation of Mask R-CNN written by Waleed Abdulla from Matterport, Inc.[23] Matterport, Inc. published their implementation under the MIT License [2]. The MIT License is a license granting the permission to use the code, copy it, modify it, publish and even to sell it free of charge and is compatible with GNU GPL 2 (or newer) [1] of GRASS GIS. Scripts in the library are also under the MIT License and moreover, Waleed Abdulla himself agreed with the usage and modifications of his code for purposes of the GRASS GIS usage. The Matterport, Inc. Mask R-CNN implementation can be found in their GitHub repository[24].

The Matterport implementation of Mask R-CNN was chosen because of many reasons. Besides its license compatibility, it is quite robust and ready for modifications leading to another implementation, so it saved thousands of lines of code. But the main motivation behind its usage is that there is a plenty of people interested in this project, proposing their ideas and testing it. And these people are experienced in fields of computer vision and deep learning, so besides the base of GRASS GIS users, there are always people from another field working with core functions of the model. Even Abdulla himself is very active in answering people's questions and open-minded when discussing other people improvement proposals. I found it very useful and consider it as the icing on the cake of open source software.

The following text will briefly describe the structure of the mentioned modules together with their workflow. Because aspects of the Mask R-CNN architecture were already mentioned in chapter 3.6.1, this facet will be a bit overshadowed and the main focus will be given to the code implementation. The library will be also introduced altogether with notes on my modifications connected with this thesis to distinguish them from Abdulla's code.

[23] `https://matterport.com/about/`

[24] `https://github.com/matterport/Mask_RCNN`

5.1 Mask R-CNN library

The library consists of four Python modules:

- `config.py`: The configuration file for the model. It will be described in chapter 5.1.1.
- `model.py`: The core of the Mask R-CNN model. It builds up the model. It will be described in chapter 5.1.2.
- `parallel_model.py`: Contains the ParallelModel class, a subclass of the standard Keras model allowing the parallelized computation. Because this file is in the original state written by Waleed Abdulla without any modification, it will not be described further in the text.
- `utils.py`: Utilities for the model. They will be described in chapter 5.1.3.

Files `config.py`, `model.py`, and `utils.py` will be described in the following text. Because these files have quite ample inner documentation, only the most important functions will be described.

5.1.1 config.py

In the Matterport implementation, `config.py` is the configuration class setting model attributes like the learning rate, RPN anchor scales and aspect ratios (described in chapter 3.4.3). It is recommended not to use this class directly but to subclass it; in the subclassed class, the user should override model attributes to fit his future model.

Instead of overriding the `ModelConfig` class, I implemented an initialization method. The `__init__` method is automatically called when a class object is being constructed and allows to construct it in a specific state; in the `ModelConfig` class, `__init__` sets model attributes either to a default or a user-defined state. The attribute value pass is made through parameters of `i.ann.maskrcnn.train` and `i.ann.maskrcnn.detect` modules.

The `ModelConfig` class also contains the `display` method to display the model attributes.

5.1.2 model.py

`model.py` builds up the model using tools and features provided by Keras and TensorFlow. Purposes of classes and functions included in this file are diverse and can be summed as follows:

- Building the ResNet backbone.
- Building the RPN.
- Building RoIAlign layers.
- Building head architectures.
- Building the complete Mask R-CNN model and putting everything together.
- Building detection layers.
- Defining loss functions.
- Miscellaneous functions and utilities connected to the model, like batch normalization (see chapter 2.2.4), data formatting and generating (building up targets, loading ground truth masks) or bounding boxes normalization.

The file is almost without any modification. The only modifications in compare with Waleed Abdulla's original code were made to handle errors that can raise during masks loading; however, all of the functions and classes from `model.py` described below were written by Waleed Abdulla, to see the modifications please take a look at the code where the authorship is explicitly written.

ResNet backbone

The essential function for the building of the backbone architecture is the one called `resnet_graph`. It follows the architecture described in chapter 2.3.6. Its workflow is illustrated in pseudocode 5.1. Some features were simplified in the pseudocode and it uses two functions `identity_block` and `convolutional block`. These features will be described in the following text.

```
1 layers = intended layers
2 layers.add(zero padding 3x3)
3 layers.add(convolution 7x7)
```

```
layers.add(batch normalization)
layers.add(ReLu)
layers.add(maximum pooling)
layers.add(convolutional block 64x64x256)
layers.add(2 identity blocks 64x64x256)
layers.add(convolutional block 128x128x512)
layers.add(3 identity blocks 128x128x512)
layers.add(convolutional block 256x256x1024)
if architecture == 'resnet50':
    layers.add(5 identity blocks 256x256x1024)
elif architecture == 'resnet101':
    layers.add(22 identity blocks 256x256x1024)
return layers
```

Pseudocode 5.1: Building the ResNet backbone architecture

The real function does not return complete layers, but it returns them in stages $C1$, $C2$, $C3$, $C4$, $C5$ as can be seen in pseudocode 5.8, where this function is called `build_resnet_backbone`. Each of these stages represents the state of art before each convolutional block addition, which is the last layer before changing dimensions of inputs or outputs. It is important for the FPN as was mentioned in chapter 3.6.1 and illustrated in the model building in pseudocode 5.8.

Functions `identity block` and `convolutional block` are very similar and both builds the bottleneck block illustrated in figure 2.12. The only difference is that the `convolutional block` function also implements a 1×1 convolution in the shortcut connection as it is necessary to change the shape of the input to the one used in the block. The rest of their implementation is more or less the same and is illustrated in pseudocode 5.2 (the convolution should be applied in the output connection step). It uses filters given to each call of the function in the ResNet pseudocode.

```
original_input = original_input_tensor
block = intended block of layers
block.add(convolution 1x1)
block.add(batch normalization)
block.add(ReLu)
block.add(convolution 3x3)
block.add(batch normalization)
block.add(ReLu)
block.add(convolution 1x1)
```

```
block.add(batch normalization)
block.connect_outputs(block, original_input)
block.add(ReLu)
return block
```

Pseudocode 5.2: identity_block

RPN

The RPN is built by two functions, `build_rpn_model` and `rpn_graph`. However, these functions build only the model, e.g. the sliding window and its behaviour, anchors are generated in `utils.py` as described in chapter 5.1.3. Even in this split approach, it follows the idea from chapter 3.4.3.

Inputs for the `rpn_graph` function are a feature map, number of anchors per location and anchor stride and returns anchor class logits, probabilities and bounding boxes refinements. The workflow of `rpn_graph` is illustrated in pseudocode 5.3. `build_rpn_model` creates a model which firstly feed the `rpn_graph` function and then returns the above-mentioned values.

```
feature_map = input_feature_map
logits_number_of_filters = 2 * number of anchors per location
bbox_number_of_filters = 4 * number of anchors per location
shared_layer = convolution 3x3 on feature_map
rpn_class_logits = convolution 1x1 on shared_layer with
   logits_number_of_filters
rpn_probabilities = softmax on rpn_class_logits
rpn_bbox_refinements = convolution 1x1 on shared_layer with
bbox_number_of_filters
return rpn_class_logits, rpn_probabilities, rpn_bbox_refinements
```

Pseudocode 5.3: rpn_graph

An important class for the RPN is the `ProposalLayer` class. It takes anchor probabilities, bounding box refinements and anchors themselves as inputs, trims them to smaller batches while taking into account top anchors and applies refinements to the anchor boxes.

```
probs = anchor probabilities
deltas = anchor refinements
anchors = anchors
threshold = threshold for probabilities
```

```
5  top_anchors = names_of_anchors_with_top_probs(probs, how_many=min
       (6000, len(probs)))
6  probs_batch = batch_slice(probs, top_anchors)
7  deltas_batch = batch_slice(deltas, top_anchors)
8  anchors_batch = batch_slice(anchors, top_anchors)
9  boxes = apply_refinements(anchors_batch, deltas_batch)
10 proposals = [boxes, probs_batch]
11 proposals.apply_threshold(threshold)
12 return proposals
```

Pseudocode 5.4: ProposalLayer

RoIAlign

As was described already in chapter 3.6.1, RoIAlign is more or less the RoIPooling algorithm from the chapter 3.4.2 without rounding. The implementation is briefly sketched in pseudocode 5.5.

```
1  pool_shape = shape of regions
2  image_shape = shape of the image
3  boxes = list of RoIs
4  feature_maps = list of feature maps
5  h, w = compute_heights_and_widths_boxes(boxes)
6  image_area = image_shape[0] * image_shape[1]
7  roi_level = minimum(5, 4 + log2(sqrt(h * w) / (224 / sqrt(
       image_area))))
8  pooled = list()
9  for level in range(2, 6):
10     roi_level_i = 1 where roi_level == level, 0 elsewhere
11     level_boxes = gather(boxes, indices=roi_level_i)
12     pooled.append(crop_and_resize(original_image=feature_maps[level
           -2], what_process=level_boxes, shape=pool_shape, method='
           bilinear'))
13 pooled.rearrange_to_match_the_order(boxes)
14 return pooled
```

Pseudocode 5.5: RoIAlign

It implements the RoIAlign algorithm on multiple levels of the feature pyramid and in its enumerations of the $log2$ equation, it follows the ideas behind enumerations in [27] and also applies the five-levels approach. The minimum choosing at line 7

and the loop at line 9 the then follows the idea of using only layers two to five from chapter 5.1.2.

Head architectures

As can be seen in figure 3.13 and was already described in chapter 3.6.1, the head architecture is divided into two sections. The head architecture for bounding boxes and class probabilities is handled by the `fpn_classifier_graph` function and the mask architecture by the `build_fpn_mask_graph`.

`fpn_classifier_graph` takes as input RoIs, feature maps, pool size and a number of classes and returns classifier logits, probabilities and bounding boxes refinements. `build_fpn_mask_graph` takes the same input but returns only a list of masks.

```
1  rois = given regions of interest in normalized coordinates
2  feature_maps = list of feature maps from layers P2, P3, P4, P5
3  pool_size = height of feature maps to be generated from ROIpooling
4  num_classes = number of classes
5  layers = list of keras layers
6  layers.add(ROIAlign(pool_size, input=[rois, feature_maps]))
7  layers.add(convolution pool_size X pool_size)
8  layers.add(batch_normalization)
9  layers.add(ReLU)
10 layers.add(convolution 1x1)
11 layers.add(batch_normalizataion)
12 layers.add(ReLU)
13 shared = squeeze_to_one_tensor(output of layers)
14 class_logits = fully_connected_layer(input=shared,
       number_of_filters=num_classes)
15 probabilities = softmax(class_logits)
16 bboxes = fully_connected_layer(input=shared, number_of_filters=4 *
       num_classes)
17 return class_logits, probabilities, bboxes
```

Pseudocode 5.6: fpn_classifier_graph

```
1  rois = given regions of interest in normalized coordinates
2  feature_maps = list of feature maps from layers P2, P3, P4, P5
3  pool_size = height of feature maps to be generated from ROIpooling
4  num_classes = number of classes
5  layers = list of keras layers
```

```
layers.add(ROIAlign(pool_size, input=[rois, feature_maps]))
layers.add(convolution 3x3)
layers.add(batch_normalization)
layers.add(ReLU)
layers.add(convolution 3x3)
layers.add(batch_normalization)
layers.add(ReLU)
layers.add(convolution 3x3)
layers.add(batch_normalization)
layers.add(ReLU)
layers.add(convolution 3x3)
layers.add(batch_normalization)
layers.add(ReLU)
layers.add(deconvolution 2x2 with strides 2)
layers.add(convolution 1x1 with sigmoid as an activation function)
return layers
```

Pseudocode 5.7: build_fpn_maskk_graph

In the pseudocodes above, a ROIAlign object is added as the first one into layers. This object was sketched in pseudocode 5.5.

Mask R-CNN model

The centrepiece of the `model.py` file is the `MaskRCNN` class which contains methods to build the entire Mask R-CNN model by cobbling together different types of layers and to use it for training or detection.

The workflow of the method `build` is illustrated in pseudocode 5.8 and follows the architecture described in chapter 3.6.1. In the pseudocode, we can see that the head architecture differs a bit in the training and in the detection. It is due to the fact that we need loss values to be computed during the training, so we compute them from detected values and *target* values (values based on known targets from the training dataset).

```
C2, C3, C4, C5 = build_resnet_backbone()
P5, P4, P3, P2 = build_top_down_fpn_layers(C2, C3, C4, C5)
anchors = generate_anchors()
rpn = build_rpn()
rois = ProposalLayer(rpn, anchors)
if mode == 'training':
    ground_truth_values = values from the training dataset
```

```
8       bbox, classes = fpn_classifier(rois)
9       target_detection = DetectionTargetLayer(ground_truth_values)
10      mask = fpn_mask(rois from target_detection)
11      loss = loss_functions(target_detection, bbox, classes, mask)
12      model = [bbox, classes, mask, loss]
13  else:
14      bbox, classes = fpn_classifier(rois)
15      target_detection = DetectionLayer(bbox, classes)
16      mask = fpn_mask(rois)
17      model = [bbox, classes, mask]
18  return model
```

Pseudocode 5.8: Mask R-CNN.build

In the pseudocode, we can see few classes and functions. Although their purposes are quite evident, some of them can be seen in different pseudocodes. Function `build_resnet_backbone` was already described in pseudocode 5.1, subsequent function `build_top_down_fpn_layers` is fairly straightforward process connecting layers as in chapter 3.6.1, `generate_anchors` will be described in 5.12, `build_rpn` can be seen in pseudocode 5.3, `ProposalLayer` in pseudocode 5.4, `fpn_classifier` represents the `fpn_classifier_graph` from pseudocode 5.6 and `fpn_mask` is function `build_fpn_mask_graph` from pseudocode 5.7.

5.1.3 utils.py

The most important part of the `utils.py` file is the `Dataset` class. It is also the only part of the `utils.py` code that was modified for the needs of GRASS GIS usage (the other changes are just minor refactorings).

The `utils.py` also contains a lot of functions. Only a few of them will be mentioned as all of them have sufficient documentation in the code.

Dataset

The `Dataset` class is the base class for dataset classes and images. It contains informations about them including their names, identifiers and in the case of images also paths to them.

One of the written methods is the one called `import_contents`, which feeds the `Dataset` object with classes and images. The workflow is illustrated in pseudocode 5.9. Inputs for the method are:

- List of classes names intended to be learned
- List of directories containing training images and masks
- Name of model

The `add_class` method in pseudocode 5.9 import a class into the `Dataset` object dictionary altogether with a unique identifier; an important part is containing the background as the first class with identifier 0 (in the pseudocode represented simplifiedly by the `saved_class` dictionary). The `add_images` line is a loop over all images with the predefined extension contained in a given directory importing them altogether with their identifier and path into the `Dataset` object list.

```
classes = list of classes names intended to be learned
directories = list of directories containing training images and
   masks
saved_classes = {'BG': 0}
for i in classes:
    add_class
for directory in directories:
    add_images
```

Pseudocode 5.9: import_contents

Another important method written for the needs of the GRASS GIS modules is the one called `get_mask`. The workflow of the method is illustrated in pseudocode 5.10. It returns an array containing boolean masks (True for the mask, False elsewhere) for each instance in the picture, an array of class identifiers corresponding each instance in the masks array and an error message. If an error happened during the process of masks loading, the load is skipped for all masks in the directory.

```
masks_list = list of mask files within the directory
first_mask = masks_list[0]
masks_array = array containing first_mask transformed to bool
classes_list = list containing class of the first mask
for new_mask in masks_list[1:]:
    concat_mask = new_mask transformed to bool
    concatenate masks_array with concat_mask
    append class of new_mask into classes_list
    if any problem happened:
        return None, None, 1
```

```
return masks_array, classes_list, 0
```

Pseudocode 5.10: get_mask

From the rest of `Dataset` class methods, one more will be mentioned. `prepare` must be called before the usage of the `Dataset` object as it prepares it for use. The preparation is done through setting object parameters like a number of classes, classes names and identifiers or number of images. This setting is based on information got during the `import_contents` call.

Bounding boxes tools

Because bounding boxes are not required to be provided altogether with masks in the training dataset, the function `extract_boxes` is used to compute bounding boxes from masks. The function searches for the first and last horizontal and vertical positions containing mask along all channels and returns them as an array. It means that each pixel of the mask is contained in the returned horizontal-vertical bounding box and it is also as tight as possible.

A function used to compute the IoU is called simply `compute_iou`. Its workflow is illustrated in pseudocode 5.11. The handling of no intersection is also implemented in the function, but for better reading, it is not included in the pseudocode.

```
predicted_box_area = area of predicted box
groundtruth_box_area = area of given mask
y1 = the bigger one from the upper coordinates of the predicted and
    ground truth bboxes
y2 = the smaller one from the lower coordinates of the predicted
   and ground truth bboxes
x1 = the bigger one from the left coordinates of the predicted and
   ground truth bboxes
x2 = the smaller one from the right coordinates of the predicted
   and ground truth bboxes
intersection = (x2 - x1) * (y2 - y1)
union = predicted_box_area + groundtruth_box_area - intersection
iou = intersection / union
return iou
```

Pseudocode 5.11: compute_iou

With the comparison of ground truth boxes and the predicted ones is connected also the function `box_refinement`. It computes differences between ground truth

and predicted coordinates of bounding boxes and returned them as the information of the inaccuracy bounding box inaccuracy.

Pyramid anchors tools

The theory of scales and pyramids was already described in chapters 3.4.3 and 3.6.1. Two functions are connected with the generation of the anchors at different levels of a feature pyramid. The called one is `generate_pyramid_anchors` which loops over scales. In the loop, the `generate_anchors` function is called to generate anchors of ratios for a given set of scales.

The workflow of the `generate_anchors` function is illustrated in pseudocode 5.12. It takes scales and ratios of anchors, feature map shape and anchors and feature map strides as inputs. It uses these inputs to compute heights and widths of different anchors (can be seen in figure 3.6) and to compute a grid of anchors centres. This grid together with their heights and widths defines the returned value, anchors.

```
scales = array of scales
ratios = array of ratios
feature_map_shape = [height, width]
anchor_stride = stride of anchors on the featuremap
feature_stride = stride of the featuremap
heights = scales divided by a square root of ratios (each by each)
widths = scales multiplied by square root of ratios (each by each)
shifts_y = grid from 0 to shape[0] with stride anchor_stride
shifts_y = shifts_y * feature_stride
shifts_x = grid from 0 to shape[1] with stride anchor_stride
shifts_x = shifts_x * feature_stride
anchors_centers = stack of [shifts_y, shifts_x] in each combination
anchors_sizes = [heights, widths]
anchors = [anchors_centers - 0.5 * anchors_sizes, anchors_centers +
    0.5 * anchors_sizes]
return anchors
```

Pseudocode 5.12: generate_anchors

5.2 i.ann.maskrcnn.train

Before a child can recognize an object, child's parents must teach him how does the object look like, what is its name, its common colours and other attributes.

The same applies to an ANN - the model must be trained before it can detect or predict objects. The training is done in the `i.ann.maskrcnn.train` GRASS GIS module. The module is written in the file `i.ann.maskrcnn.train.py`. This file was written as part of the practical section of the thesis. Its workflow is with few simplifications illustrated in figure 5.1.

The flowchart contains few already-mentioned functions and classes, specifically the `ModelConfig` class and its `display` method from chapter 5.1.1, the `MaskRCNN` class from chapter 5.1.2 and the `Dataset` class and its methods `import_contents` and `prepare` from chapter 5.1.3.

The last step in this flowchart, a method `model.train()`, has actually two different forms depending on the usage of initial weights. The first form is applied for a training from a scratch and trains all layers. The second one consists of three smaller segments; firstly training layers 5 and higher, then fine-tuning layers 4 and higher and the last and biggest segment is fine-tuning the whole architecture. It is shown in the flowchart in figure 5.2 and the idea behind this behaviour is that it is impractical to train the first layers including low-level features, while changes have a huge impact on deeper levels and those features should be more or less the same for any object.

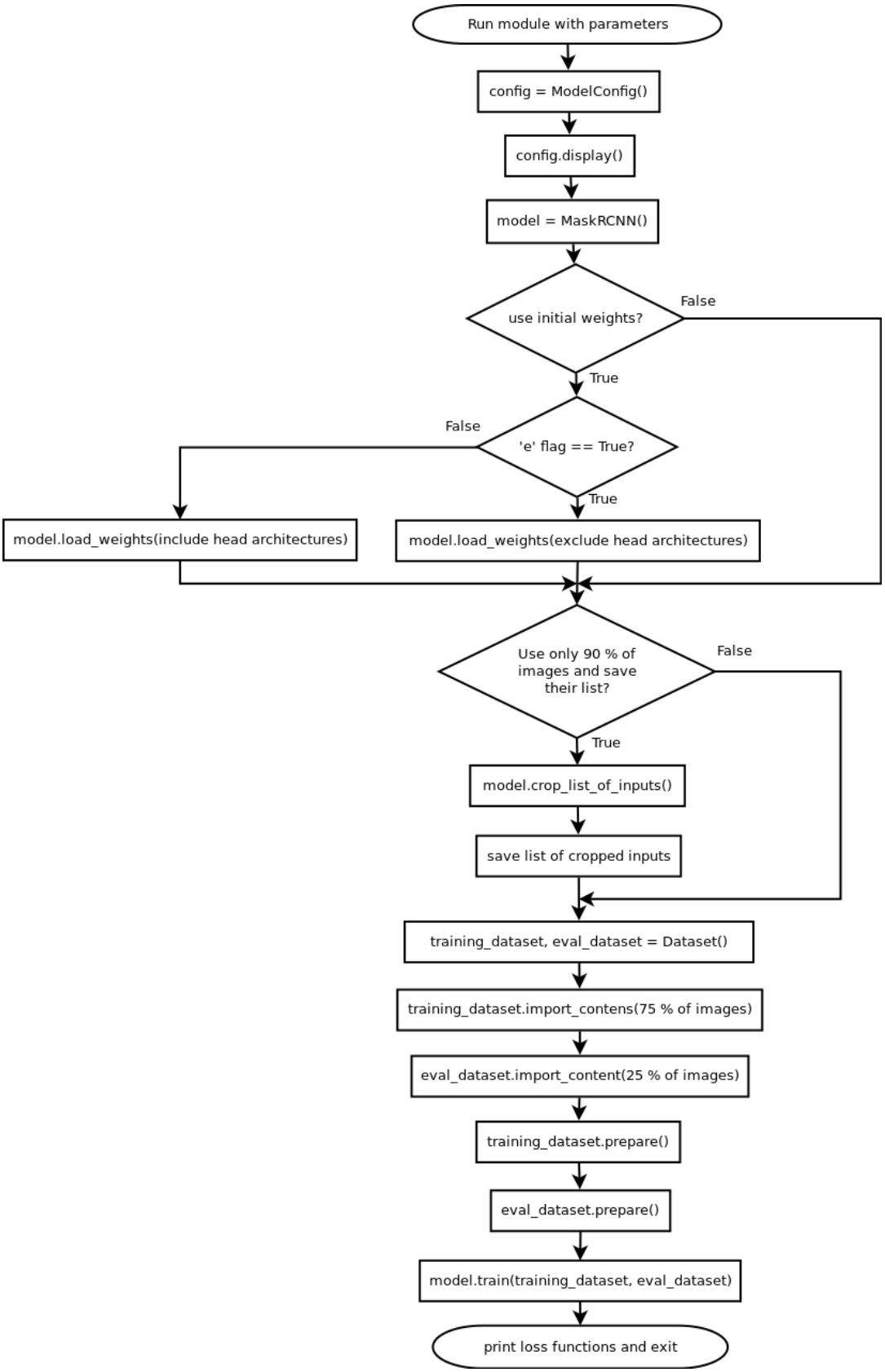


Figure 5.1: Flowchart of the i.ann.maskrcnn.train module, source: author

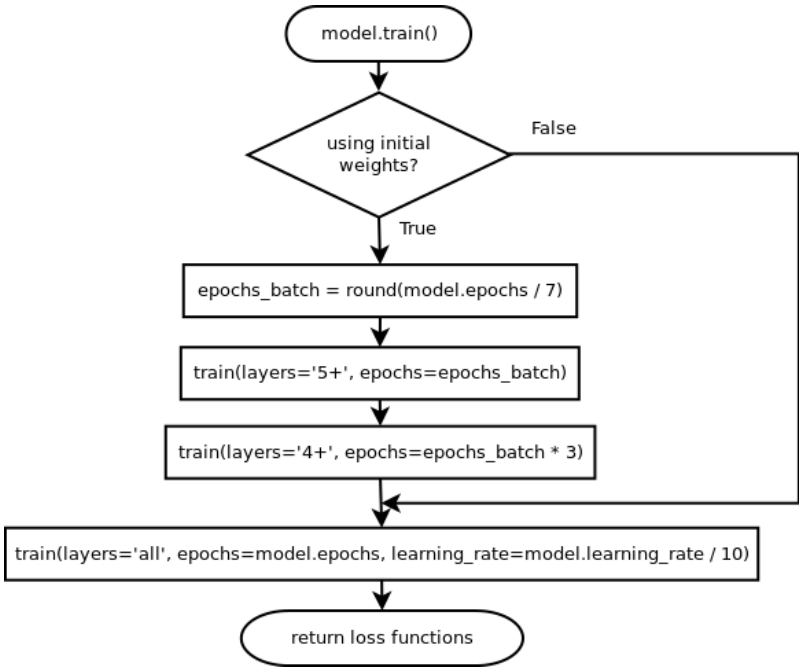


Figure 5.2: Flowchart of different training branches inside the i.ann.maskrcnn.train module, source: author

`i.ann.maskrcnn.train` also contains its own manual, help and a graphical user interface (GUI) to help a user's understanding of the module. The manual can be seen in appendix A.2.

5.3 i.ann.maskrcnn.detect

In case we have a trained model, either trained by us or provided by someone else, we can use it to detect features or objects in maps. Such maps have to be raster maps imported to GRASS GIS or georeferenced raster files. The output from the module consists of a set of vector maps for each class. Although the model is to some extent scale-invariant, it is recommended to provide rasters in similar resolution to the one used in training images. The GRASS GIS module providing detection is `i.ann.maskrcnn.detect`. The module is contained in the file `i.ann.maskrcnn.detect.py`. This file can be seen as the second part of the practical section of the thesis. Its workflow is with some simplifications illustrated in figure 5.3.

As in the `i.ann.maskrcnn.train` module, the flowchart contains few already-mentioned functions and classes, specifically the `ModelConfig` class from chapter 5.1.1 and the `MaskRCNN` class from chapter 5.1.2.

However, more unmentioned functions can be seen in the flowchart. Function `parse_instances` imports masks of detected instances into GRASS GIS (except for images with external georeferencing; they are just saved into a temporary folder). These masks are rasters of the same size as the input maps/images where mask pixels have the value of class ID and the rest are zeros. In case the georeferencing is external, a flag `-e` should be used and the function `external_georeferencing` is called to copy georeferencing files to the directory with masks and import them into GRASS GIS.

Rasters are then cropped and vectorized to get a separate vector map for each class. The process of cropping and vectorizing consists of a bunch of GRASS GIS modules and is illustrated in figure 5.4.

The illustrated process works only with detected classes (e.g. when no instance of sties was detected, the loop corresponding to this class will be skipped). The term `vector_map` used in the flowchart corresponds to the same string as the term `raster_map`, but is used to emphasize the difference in the map type; it is also the reason why we can use `v.patch` and `g.remove` with `vector_maps` - because we already have the list of raster maps. The motivation behind the inner loop in the flowchart is a detection made on multiple rasters; classes from them are extracted and vectorized separately and then patched together for each class.

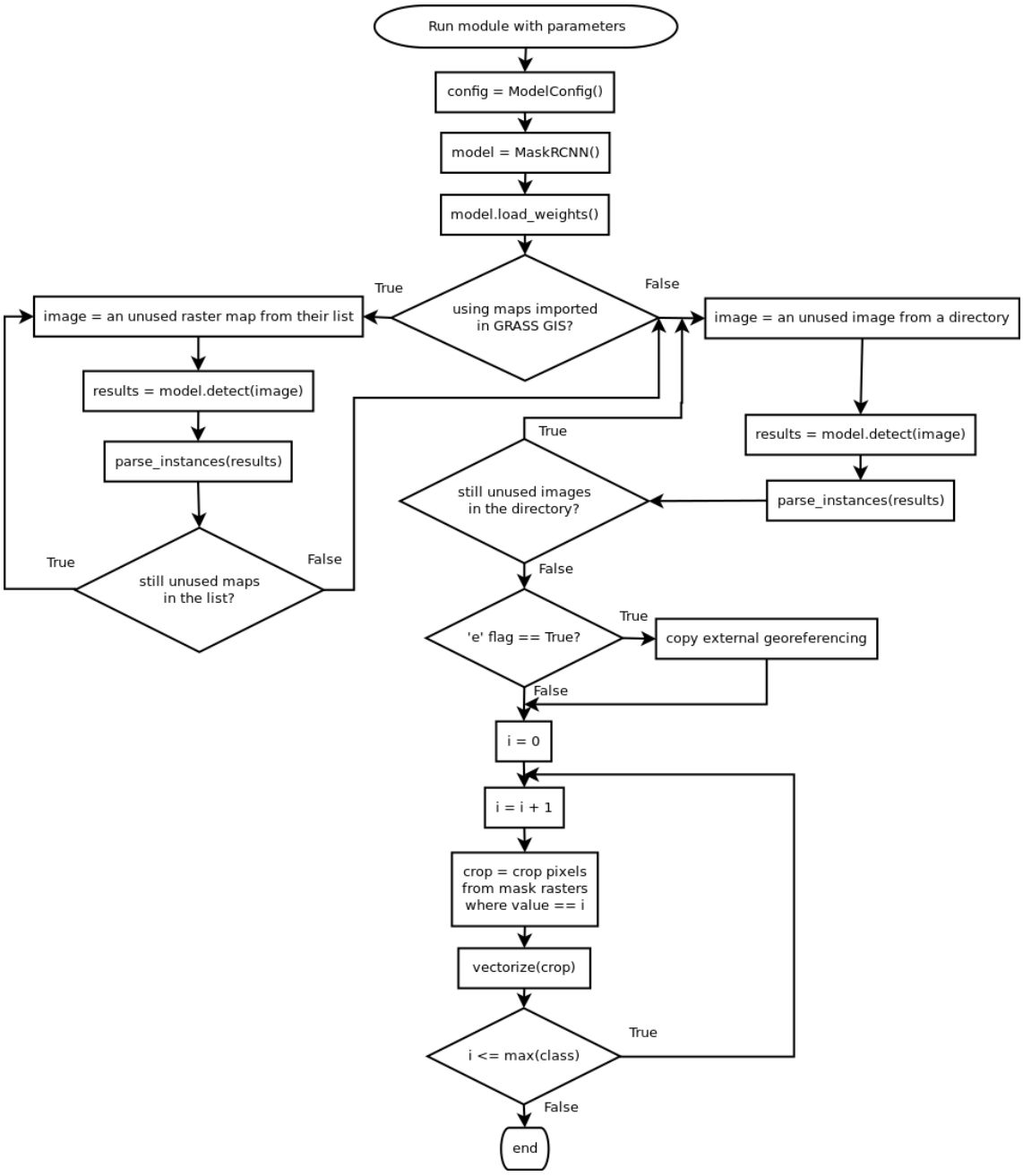


Figure 5.3: Flowchart of the i.ann.maskrcnn.detect module, source: author

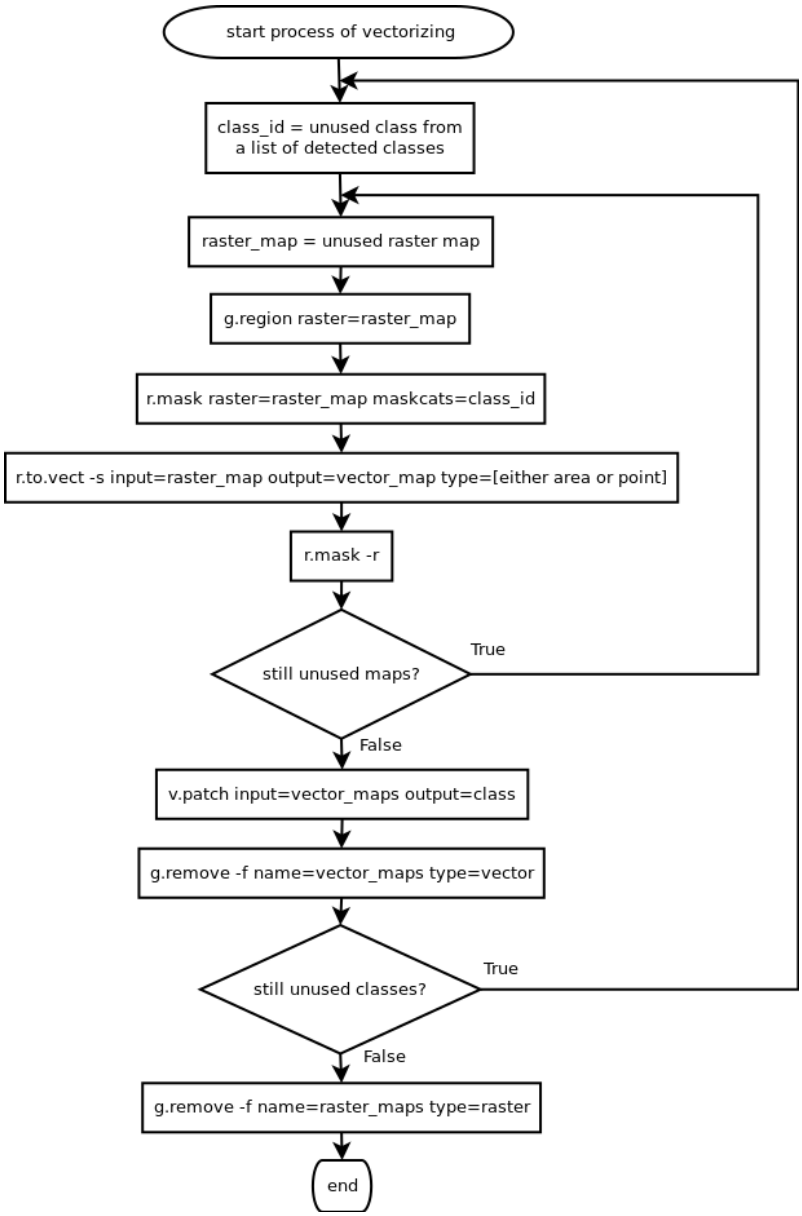


Figure 5.4: Flowchart of the vectorization and classes extraction in the i.ann.maskrcnn.detect module, source: author

The output of `i.ann.maskrcnn.detect` represents detected masks in a vector form. The output type can be either areas or points (corresponding to a centre of the mask).

`i.ann.maskrcnn.detect` also contains its own manual, help and a GUI to help a user's understanding of the module. The manual can be seen in appendix A.3.

5.4 GRASS GIS patch

Although the support for Python 3 in GRASS GIS is a widely discussed topic and although much effort has been done to for it, it is still incomplete. This lack is handled in modules, but there is still a problem in setting the GRASS environment. Thus, a patch and a recompilation of GRASS are needed to allow the user to set the environment right. The patch decodes bytes-typed string into a UTF-8-typed string and consists of the following *diff*. It can be seen in appendix A.1.

6 Conclusion

The goal of the thesis was to find and implement a suitable ANN architecture into GRASS GIS. Because ANNs and especially CNNs are shaking with the field of computer vision, many GRASS GIS users and developers were interested in implementation options into this system.

The first part of the thesis was dedicated to a theoretical background behind CNNs. It is followed by their various applications in the field of computer vision.

The second part of the thesis was dedicated to the implementation of Mask R-CNN modules into GRASS GIS. It starts with a brief introduction of some of the used tools and follows with explanations of the most important parts of the code.

The research flows in the background of the theoretical part and is briefly summarized at the beginning of the chapter on the implementation.

Developed modules are ready to use and as can be seen in appendices, I already made few tests. However, because the training costs a lot of time (at least few days on a GPU machine, even weeks on a machine without GPU), only a limited amount of tests was made. Modules were proposed for testing also to other GRASS GIS developers and users and they found their results to be satisfying, sometimes even for data, where other methods of classification in GRASS GIS failed (a shadow over a field, different colours, etc.).

Developed modules are available from the GitHub repository[25].

Even though modules work, there are still some possible extensions and even some issues.

The biggest issue is in insufficient support for Python 3 in GRASS GIS. This issue is solved by a patch attached as an e-attachment and should be solved in GRASS GIS during this year.

Possible extensions are:

- Support more training data annotation structures. The current state works with `*.jpg` images and `*.png` masks for each instance. The next step could be a JSON-based structure used in MS COCO[26].

[25] `https://github.com/ctu-geoforall-lab-projects/dp-pesek-2018`

[26] `http://cocodataset.org/#format-data`

- Support training on images in GRASS GIS (using raster map as images, vector areas as masks).
- Support more channels than three.
- Support more types of the backbone architecture. Now only ResNet 50 and ResNet 101 are supported.
- Diversify head architecture types. The current one is just very basic one.
- Implement image augmentation for the training.

The last thing worthy of mention is the high gear of progress. After only six years of research in the field of CNNs, black is white and up is down. Everything has changed and results unbelievable six years back are standards nowadays. Almost every week, there is a new architecture. Even during work on this project, many interesting architectures were proposed, to name a few, [7] or [22].

Developed modules are the first step of GRASS GIS in the direction to neural networks. Many functions could be useful for developing more modules. It would be pleasant to start a new trend and see more modules using CNNs and sharing libraries. Unsupervised learning is an open call.

List of abbreviations

AI	Artificial intelligence
ANN	Artificial neural network
CNN	Convolutional neural network
DPM	Deformable part model
FAIR	Facebook AI research
FC	Fully connected
FCN	Fully convolutional network
FPN	Feature pyramid network
GIS	Geographic information system
GPL	General public license
GRASS	Geographic resources analysis support system
GUI	Graphical user interface
ILSVRC	ImageNet large scale visual recognition challenge
IoU	Intersection over union
mAP	Mean average precision
OGC	Open geospatial consortium
R-CNN	Region-based convolutional neural network
ReLU	Rectified linear unit
ResNet	Residual network
RNN	Recurrent neural network
RoI	Region of interest
RPN	Region proposal network

SVM Support vector machine

VOC Visual object classes

References

[1] GNU General Public License. URL: http://www.gnu.org/licenses/gpl.html.

[2] MIT License. URL: https://opensource.org/licenses/MIT.

[3] ALEXE, Bogdan; DESELAERS, Thomas and FERRARI, Vittorio. Measuring the objectness of image windows. *IEEE transactions on pattern analysis and machine intelligence*. 2012, 34, n. 11, pp. 2189–2202.

[4] BOUREAU, Y-Lan; PONCE, Jean and LECUNN, Yann. A Theoretical Analysis of Feature Pooling in Visual Recognition. In: *Proceedings, 27th International Conference on Machine Learning ICML*. 2010.

[5] BRADSKI, Gary and KAEHLER, Adrian. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[6] CARANDINI, Matteo; HEEGER, David J. and MOVSHON, Joseph A. Linearity and normalization in simple cells of the macaque primary visual cortex. *Journal of Neuroscience*. 1997, 17, n. 21, pp. 8621–8644.

[7] CHEN, Liang-Chieh et al. MaskLab: Instance Segmentation by Refining Object Detection with Semantic and Direction Features. *CoRR*. 2017, abs/1712.04837. URL: http://arxiv.org/abs/1712.04837.

[8] ENDRES, Ian and HOIEM, Derek. Category-independent object proposals with diverse ranking. *IEEE transactions on pattern analysis and machine intelligence*. 2014, 36, n. 2, pp. 222–234.

[9] EVERINGHAM, Mark et al. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision IJCV*. 2015, 111, n. 1, pp. 98–136.

[10] FELZENSWALB, Pedro F. et al. Object Detection with Discriminatively Trained Part-Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2010, 32, n. 9, pp. 1627–1645.

[11] GIRSHICK, Ross. Fast R-CNN. In: *International Conference on Computer Vision (ICCV)*. 2015.

[12] GIRSHICK, Ross et al. Rich feature hierarchies for accurate object detection and semantic segmentation. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014.

[13] GOODFELLOW, Ian; BENGIO, Yoshua and COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] HE, Kaiming et al. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2015, 37, n. 9, pp. 1904–1916.

[15] HE, Kaiming et al. Deep Residual Learning for Image Recognition. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[16] HE, Kaiming et al. Mask R-CNN. In: *International Conference on Computer Vision (ICCV)*. 2017.

[17] HINTON, Geoffrey E. et al. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*. 2012, abs/1207.0580. URL: `http://arxiv.org/abs/1207.0580`.

[18] HUBEL, David H. and WIESEL, Torsten N. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*. 1959, 148, n. 3, pp. 574–591.

[19] HUBEL, David H. and WIESEL, Torsten N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*. 1962, 160, n. 1, pp. 106–154.

[20] HUBEL, David H. and WIESEL, Torsten N. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*. 1968, 195, n. 1, pp. 215–243.

[21] IOFFE, Sergey and SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, abs/1502.03167. URL: `http://arxiv.org/abs/1502.03167`.

[22] KIRILLOV, Alexander et al. Panoptic Segmentation. *CoRR*. 2018, abs/1801.00868. URL: http://arxiv.org/abs/1801.00868.

[23] KRIZHEVSKY, Alex; SUTSKEVER, Ilya and HINTON, Geoffrey E. ImageNet classification with deep convolutional neural networks. In: *Advances in neural information processing systems*. 2012. pp. 1097–1105.

[24] LE, Quoc V. Building high-level features using large scale unsupervised learning. In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013.

[25] LECUN, Yann et al. Gradient-Based learning applied to document recognition. *Proceedings of the IEEE*. 1998, 86, n. 11, pp. 2278–2324.

[26] LIN, Min; CHEN, Qiang and YAN, Shuicheng. Network In Network. *CoRR*. 2013, abs/1312.4400. URL: http://arxiv.org/abs/1312.4400.

[27] LIN, Tsung-Yi et al. Feature pyramid networks for object detection. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[28] LONG, Jonathan; SHELHAMER, Evan and DARRELL, Trevor. Fully convolutional networks for semantic segmentation. In: *IEEE conference on computer vision and pattern recognition CVPR*. 2015. pp. 3431–3440.

[29] LOPEZ, Marc M. and KALITA, Jugal. Deep Learning applied to NLP. *CoRR*. 2017, abs/1703.03091. URL: http://arxiv.org/abs/1703.03091.

[30] ORR, Geneviève and MÜLLER, Klaus-Robert. *Neural Networks: Tricks of the Trade*. Springer, 1998.

[31] PILGRIM, Mark. *Dive Into Python*. Apress, 2004. ISBN 978-1-59059-356-1.

[32] REN, Shaoqing et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In: *Advances in Neural Information Processing Systems (NIPS)*. 2015.

[33] RUSSAKOVSY, Olga et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision IJCV*. 2015, 115, n. 3, pp. 211–252.

[34] SERMANET, Pierre and LECUN, Yann. Traffic sign recognition with multi-scale convolutional networks. In: *The International Joint Conference on Neural Networks IJCNN*. IEEE, 2011. pp. 2809–2813.

[35] SIMONYAN, Karen and ZISSERMAN, Andrew. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. 2014, abs/1409.1556. URL: `http://arxiv.org/abs/1409.1556`.

[36] SPRINGERBERG, Jost T. et al. Striving for Simplicity: The All Convolutional Net. *CoRR*. 2014, abs/1412.6806. URL: `http://arxiv.org/abs/1412.6806`.

[37] SZEGEDY, Christian et al. Going deeper with convolutions. In: *IEEE Conference on Computer Vision and Pattern Recognition CVPR*. CVPR, 2015.

[38] SZEGEDY, Christian et al. Rethinking the inception architecture for computer vision. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016. pp. 2818–2826.

[39] SZELISKI, Richard. *Computer vision: algorithms and applications*. Springer, 2010.

[40] UIJLINGS, Jasper R. R. et al. Selective Search for Object Recognition. *International Journal of Computer Vision IJCV*. 2013, 104, pp. 154–171.

[41] WANG, Jiang et al. CNN-RNN: A unified framework for multi-label image classification. In: *IEEE Conference on Computer Vision and Pattern Recognition CVPR*. IEEE, 2016. pp. 2285–2294.

[42] XIE, Saining et al. Aggregated Residual Transformations for Deep Neural Networks. *CoRR*. 2016, abs/1611.05431. URL: `http://arxiv.org/abs/1611.05431`.

[43] ZACCONE, Giancarlo. *Getting Started with TensorFlow*. Packt Publishing, 2016. ISBN 978-1-78646-857-4.

[44] ZEILER, Matthew D. and FERGUS, Robert. Stochastic pooling for regularization of deep convolutional neural networks. In: *Proceedings of the International Conference on Learning Representation ICLR*. 2013.

[45] ZEILER, Matthew D. and FERGUS, Robert. Visualizing and understanding convolutional networks. In: *European conference on computer vision ECCV*. Springer, 2014. pp. 818–833.

Appendix

A User manual

HTML pages in the style of common GRASS GIS user manuals were created. In this chapter, they will be attached. Firstly, they will introduce Mask R-CNN tools generally and then each module separately.

A.1 Mask R-CNN tools

DESCRIPTION

Mask R-CNN tools allow the user to train his own model and use it for a detection of objects, or to use a model provided by someone else. It can be seen as a supervised classification using convolutional neural networks.

The training is done using module *i.ann.maskrcnn.train*. The user feeds the module with training data consisting of images and masks for each instance of intended classes and gets a model. For difficult tasks and when not using a pretrained model, the training may take even weeks; in case of a good pretrained model and powerful PC with GPU support, the training could get good results after 1 day and even less.

When the user has a trained model, it can be used for the detection. Module *i.ann.maskrcnn.detect* detects classes learned during the training and extracts from given images vectors corresponding to detected objects. Objects can be extracted either as areas or points.

DEPENDENCIES

*i.ann.maskrcnn.** modules contain a lot of external python dependencies. To run modules, it is necessary to have them installed. Modules use Python 3, so please install Python 3 versions.

- NumPy
- Pillow
- SciPy
- Cython
- scikit-image

- OSGeo
- TensorFlow
- Keras
- h5py

After dependencies are fulfilled, modules can be installed locally using *g.extension* module:

```
g.extension extension=i.ann.maskrcnn url=path/to/the/i.ann.maskrcnn/
    folder
```

After submitting the thesis, modules will be added to GRASS GIS official AddOns. It should be possible to install them directly by this command:

```
g.extension extension=i.ann.maskrcnn
```

GRASS GIS PATCH

Unfortunately, python3 is not fully supported by GRASS GIS yet. To use environment setting flags like *–overwrite*, the user has to update his GRASS GIS with the following patch:

```
===================================================================
--- lib/python/script/core.py    (revision 72644)
+++ lib/python/script/core.py    (working copy)
@@ -746,7 +746,7 @@
         elif var.startswith(b'opt_'):
             options[var[4:]] = val
         elif var in [b'GRASS_OVERWRITE', b'GRASS_VERBOSE']:
-            os.environ[var] = val
+            os.environ[var.decode("utf-8")] = val.decode("utf-8")
         else:
             raise SyntaxError("invalid output from g.parser: %s" %
                 line)
```

A.2 i.ann.maskrcnn.train

NAME

i.ann.maskrcnn.train - Train your Mask R-CNN network.

SYNOPSIS

i.ann.maskrcnn.train

i.ann.maskrcnn.train --help

i.ann.maskrcnn.train [**-esbn**] **training_dataset**=name [**model**=string]
classes=string[,string,...] **logs**=name **name**=string [**epochs**=value]
[**steps_per_epoch**=value] [**rois_per_image**=value]
[**images_per_gpu**=value] [**gpu_count**=value]
[**mini_mask_size**=value[,value,...]] [**validation_steps**=value]
[**images_min_dim**=value] [**images_max_dim**=value]
[**backbone**=string] [**--overwrite**] [**--help**] [**--verbose**] [**--quiet**] [**--ui**]

Flags:

-e

Pretrained weights were trained on another classes / resolution / sizes

-s

Do not use 10 % of images and save their list to logs dir

-b

Train also batch normalization layers (not recommended for small batches)

-n

No resizing or padding of images (images must be of the same size)

--overwrite

Allow output files to overwrite existing files

--help

Print usage summary

--verbose

Verbose module output

--quiet

Quiet module output

--ui

Force launching GUI dialog

Parameters:

training_dataset=name **[required]**

Path to the dataset with images and masks

model=string

Path to the .h5 file to use as initial values

Keep empty to train from a scratch

classes=string[,string,...] **[required]**

Names of classes separated with ","

logs=name **[required]**

Path to the directory in which will be models saved

name=string **[required]**

Name for output models

epochs=value

Number of epochs

default: 200

steps_per_epoch=value

Steps per each epoch

default: 3000

rois_per_image=value

How many ROIs train per image

default: 64

images_per_gpu=value

Number of images per GPU

Bigger number means faster training but needs a bigger GPU

default: 1

gpu_count=value

Number of GPUs to be used

default: 1

mini_mask_size=value[,value,...]

Size of mini mask separated with ","

To use full sized masks, keep empty.

Mini mask saves memory at the expense of precision

validation_steps=value

Number of validation steps

Bigger number means more accurate estimation of the model precision

default: 100

images_min_dim=value

Minimum length of images sides

Images will be resized to have their shortest side at least of this value

Has to be a multiple of 64

default: 256

images_max_dim=value

Maximum length of images sides

Images will be resized to have their longest side of this value

Has to be a multiple of 64

default: 1280

backbone=string

Backbone architecture

options: resnet50, resnet101

default: resnet101

DESCRIPTION

i.ann.maskrcnn.train allows the user to train a Mask R-CNN model on his own dataset. The dataset has to be prepared in a predefined structure.

DATASET STRUCTURE

Training dataset should be in the following structure:

- dataset-directory
  - imagenumber
    - imagenumber.jpg (training image)
    - imagenumber-class1-number.png (mask for one instance of class1)
    - imagenumber-class1-number.png (mask for another instance of class1)

  - ...
- imagenumber2
  - imagenumber2.jpg
  - imagenumber2-class1-number.png (mask for one instance of class1)
  - imagenumber2-class2-number.png (mask for another class instance)
  - ...

The described structure of directories is required. Pictures must be *.jpg files with 3 channels (for example RGB), masks must be *.png files consisting of numbers between 1 and 255 (object instance) and 0s (elsewhere). A mask file for each instance of an object should be provided separately distinguished by the suffix number.

NOTES

If you are using initial weights (the *model* parameter), epochs are divided into three segments. Firstly training layers 5+, then fine-tuning layers 4+ and the last segment is fine-tuning the whole architecture. Ending number of epochs is shown for your segment, not for the whole training.

The usage of the *-b* flag will result in an activation of batch normalization layers training. By default, this option is set to False, as it is not recommended to train them when using just small batches (batch is defined by the *images_per_gpu* parameter).

If the dataset consists of images of the same size, the user may use the *-n* flag to avoid resizing or padding of images. When the flag is not used, images are resized to have their longer side equal to the value of the *images_max_dim* parameter and the shorter side longer or equal to the value of the *images_min_dim* parameter and zero-padded to be of shape `images_max_dim` × `images_max_dim`. It results in the fact that even images of different sizes may be used.

After each epoch, the current model is saved. It allows the user to stop the training when he feels satisfied with loss functions. It also allows the user to test models even during the training (and, again, stop it even before the last epoch).

EXAMPLES

Dataset for examples:

- crops
  - 000000
    - 000000.jpg
    - 000000-corn-0.png
    - 000000-corn-1.png
    - ...
  - 000001
    - 000001.jpg
    - 000001-corn-0.png
    - 000001-rice-0.png
    - ...

Training from scratch

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=corn,rice logs=/home/user/Documents/logs name=crops
```

After default number of epochs, we will get a model where the first class is trained to detect corn fields and the second one to detect rice fields.

If we use the command with reversed classes order, we will get a model where the first class is trained to detect rice fields and the second one to detect corn fields:

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=rice,corn logs=/home/user/Documents/logs name=crops
```

The name of the model does not have to be the same as the dataset folder but should be referring to the task of the dataset. A good name for this one (referring also to the order of classes) could be also this one:

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=rice,corn logs=/home/user/Documents/logs name=rice_corn
```

Training from a pretrained model

We can use a pretrained model to make our training faster. It is necessary for the model to be trained on the same channels and similar features, but it does not

have to be the same ones (e.g. model trained on swimming pools in maps can be used for a training on buildings in maps).

A model trained on different classes (use *-e* flag to exclude head weights):

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=corn,rice logs=/home/user/Documents/logs name=crops model=/
    home/user/Documents/models/buildings.h5 -e
```

A model trained on the same classes:

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=corn,rice logs=/home/user/Documents/logs name=crops model=/
    home/user/Documents/models/corn_rice.h5
```

Fine-tuning a model

It is also possible to stop your training and then continue. To continue in the training, just use the last saved epoch as a pretrained model:

```
i.ann.maskrcnn.train training_dataset=/home/user/Documents/crops
    classes=corn,rice logs=/home/user/Documents/logs name=crops model=/
    home/user/Documents/models/mask_rcnn_crops_0005.h5
```

A.3 i.ann.maskrcnn.detect

NAME

i.ann.maskrcnn.detect - Detect features in images using a Mask R-CNN model.

SYNOPSIS

i.ann.maskrcnn.detect

i.ann.maskrcnn.detect --help

i.ann.maskrcnn.detect [**-esbn**] **images_directory**=name
images_format=string **model**=string **classes**=string[,string,...]
[**masks_output**=name] [**output_type**=string] [**--overwrite**] [**--help**]
[**--verbose**] [**--quiet**] [**--ui**]

Flags:

-e

External georeferencing in the images folder

--overwrite

Allow output files to overwrite existing files

--help

Print usage summary

--verbose

Verbose module output

--quiet

Quiet module output

--ui

Force launching GUI dialog

Parameters:

band1=name

Name of raster maps to use for detection as the first band (divided by ",")

band2=name

Name of raster maps to use for detection as the second band (divided by ",")

band3=name

Name of raster maps to use for detection as the third band (divided by ",")

images_directory=name

Path to a directory with images to detect

images_format=string

Format suffix of images

model=string **[required]**

Path to the .h5 file containing the model

classes=string[,string,...] **[required]**

Names of classes separated with ","

masks_output=name

Directory where masks will be saved

output_type=string

Type of output

options:area, point

default: area

DESCRIPTION

i.ann.maskrcnn.detect allows the user to use a Mask R-CNN model to detect features in GRASS GIS raster maps or georeferenced files and extract them either as areas or points. The module creates a separate map for each class.

NOTES

The detection may be used for raster maps imported in GRASS GIS or for external files (or using both). To use raster maps in GRASS GIS, you need to pass them in three bands following the order used during the training, e.g. if the training has been made on RGB images, use *band1=*.red</em*, *band1=*.green* and *band3=*.blue*. To pass multiple images, put more maps into *band** parameters, divided by ",".

The detection may be used also for multiple external files. However, all files for the detection must be in one directory specified in the *images_directory* parameter. Even when using only one image, the module finds it through this parameter.

When detecting, you can use new names of classes. Classes in the model are not referenced by their name, but by their order. It means that if the model was trained with classes *corn,rice* and you use *i.ann.maskrcnn.detect* with classes *zea,oryza*, zea areas will present areas detected as corn and oryza areas will present areas detected as rice.

If the external file is georeferenced externally (by a worldfile or an *.aux.xml* file), please use *-e* flag.

EXAMPLES

Detect buildings and lakes and import them as areas

One map imported in GRASS GIS:

```
i.ann.maskrcnn.detect band1=map1.red band2=map1.green band3=map1.blue
    classes=buildings,lakes model=/home/user/Documents/logs/
    mask_rcnn_buildings_lakes_0100.h5
```

Two maps (map1, map2) imported in GRASS GIS:

```
i.ann.maskrcnn.detect band1=map1.red,map2.red band2=map1.green,map2.
    green band3=map1.blue,map2.blue classes=buildings,lakes model=/home
    /user/Documents/logs/mask_rcnn_buildings_lakes_0100.h5
```

External files, the georeferencing is internal (GeoTIFF):

```
i.ann.maskrcnn.detect images_directory=/home/user/Documents/
    georeferenced_images classes=buildings,lakes model=/home/user/
    Documents/logs/mask_rcnn_buildings_lakes_0100.h5 images_format=tif
```

External files, the georeferencing is external:

```
i.ann.maskrcnn.detect images_directory=/home/user/Documents/
    georeferenced_images classes=buildings,lakes model=/home/user/
    Documents/logs/mask_rcnn_buildings_lakes_0100.h5 images_format=png
     -e
```

Detect cottages and plattenbaus and import them as points

```
i.ann.maskrcnn.detect band1=map1.red band2=map1.green band3=map1.blue
    classes=buildings,lakes model=/home/user/Documents/logs/
    mask_rcnn_buildings_lakes_0100.h5 output_type=point
```

B Examples

In the following chapter, few results obtained during tests of modules will be presented. Firstly a module trained to detect football and tennis pitches, then a module trained to detect buildings.

B.1 Pitches

The model for detecting football and tennis pitches was trained on a Debian server with 16 CPUs Intel Xeon E5540 (8 CPUs were in the use) and with memory 49 GBs. The processor base frequency of Intel Xeon E5540 is 2.53 GHz and the cache is 8 MB. The training used a model trained on the MS COCO dataset as a pre-trained model and the training took one month reaching loss function of 0.9568.

The training dataset consisted of Bing maps[27] tiles with zoom level 18 (resolution 0.6 m per pixel) and masks corresponding to above-mentioned pitches. The training dataset consisted of almost 54000 images (plus masks).

When the training was stopped, the loss function was about 0.86.

Detection of pitches worked very well on images with the same shape and resolution (0.6 m/pixel) as training images (figures B.1 and B.2). For images with worse resolution (1.19 m/pixel), the detection sometimes contained a bit of background for tennis pitches (figures B.4). However, there was a problem in detecting football pitches, as the model sometimes detected wrongly also other green fields (figure B.3). This problem is considered to be caused by the fact that training data contained also amateur football pitches consisting only of a rectangular green field and two goals.

[27] `https://www.bing.com/maps`

Figure B.1: An example of the detection on a picture containing football pitches

Figure B.2: An example of the detection on a picture containing tennis pitches

Figure B.3: An example of the detection on a picture containing football pitches

Figure B.4: An example of the detection on a picture containing football and tennis pitches

The permission to the usage of Bing map tiles was given to me specifically for purposes of this thesis. Unfortunately, the permission to share the training dataset in e-attachments could not be given to me.

B.2 Buildings

Another model was trained to recognize buildings. Training data were from the same source, with the same resolution, but this time only 2400 images (plus masks) was used.

The training ran for 2 days on a machine using NVIDIA Tesla K80 GPU. NVIDIA Tesla K80 has memory 24 GBs, effective clock speed 2.5 Hz with 875 MHz boost clock and 560 MHz core clock. The training ran from the scratch.

Results are worse than in the pitches detection. In the figure from the last epoch, figure B.10, we can see a tennis pitch and road detected as a building, the building with a black roof was not recognized and only a small part of the tribune was detected. These problems may be caused by some of the following effects:

- More than twenty times smaller dataset.
- Bigger diversity in building types (compared to tennis pitches)
- Training from a scratch and for a shorter time

Even though the training took a shorter time, it reached much smaller loss function (0.5019 for the last epoch) than in the pitches training. A figure of detection with this loss function is shown in figure B.10.

To illustrate a progress of the training, the same area will be shown also during older epochs with loss function depicted in their captions. The progress shows also interesting moment, where an epoch with loss function 0.6280 (figure B.9) seems to behave better than the one with loss function 0.5019 (it detects also the small building on the left and a piece of a building at the bottom of the picture and it does not detect the street on the right; the tribune was detected as two individual buildings).

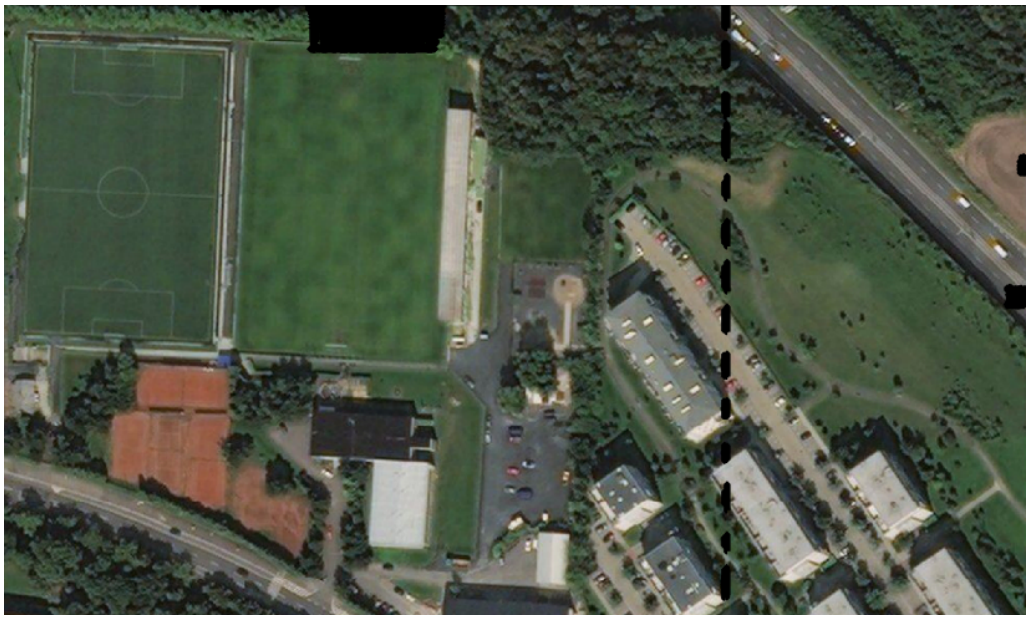


Figure B.5: An example of the detection; epoch 1, loss function 35.0102

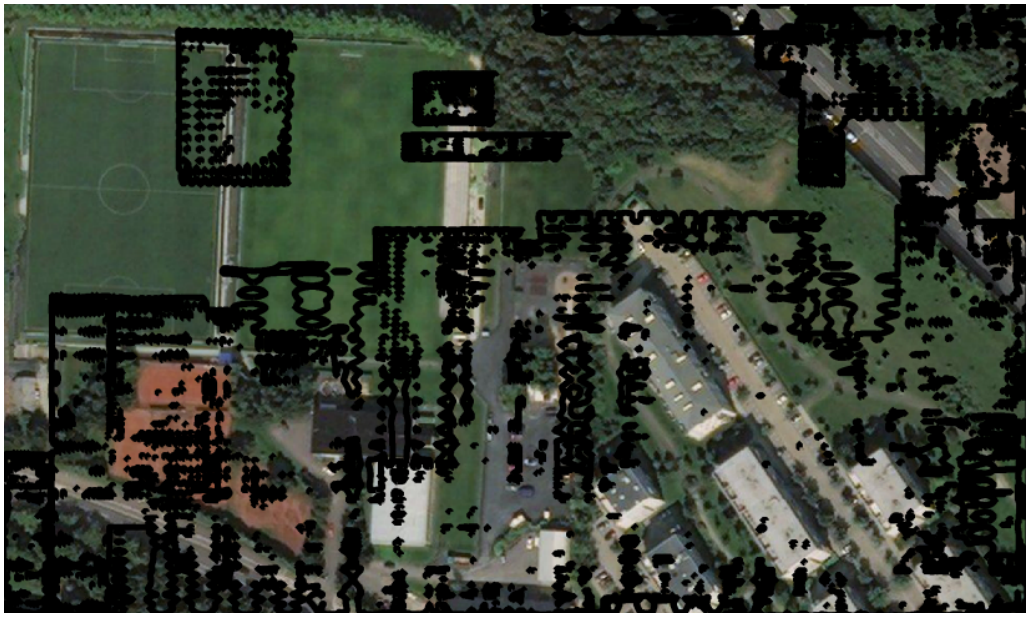


Figure B.6: An example of the detection; epoch 10, loss function 5.8694

Figure B.7: An example of the detection; epoch 50, loss function 1.3638

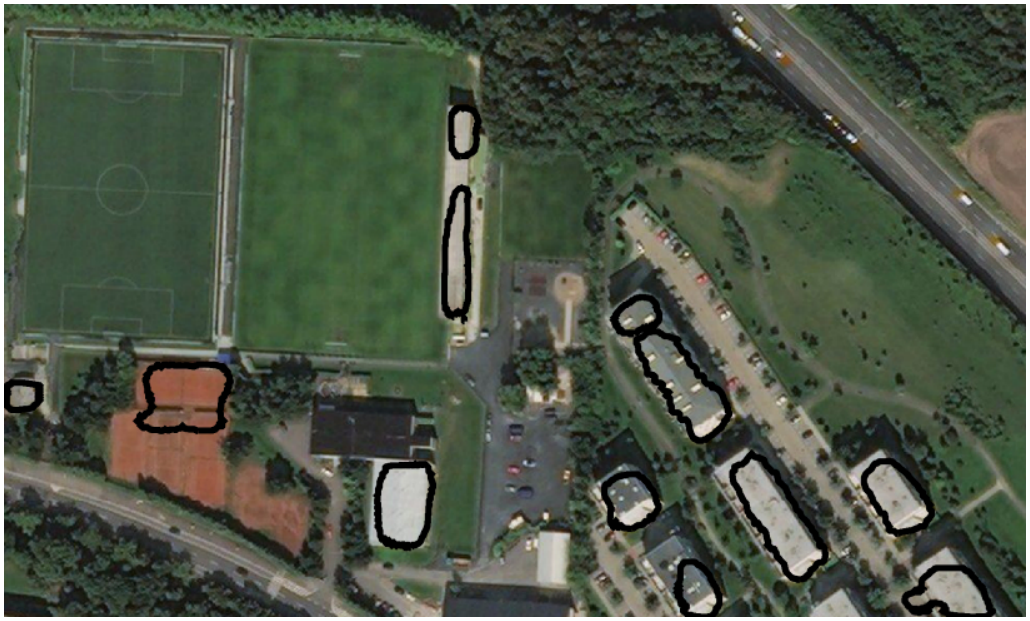


Figure B.8: An example of the detection; epoch 100, loss function 0.8245

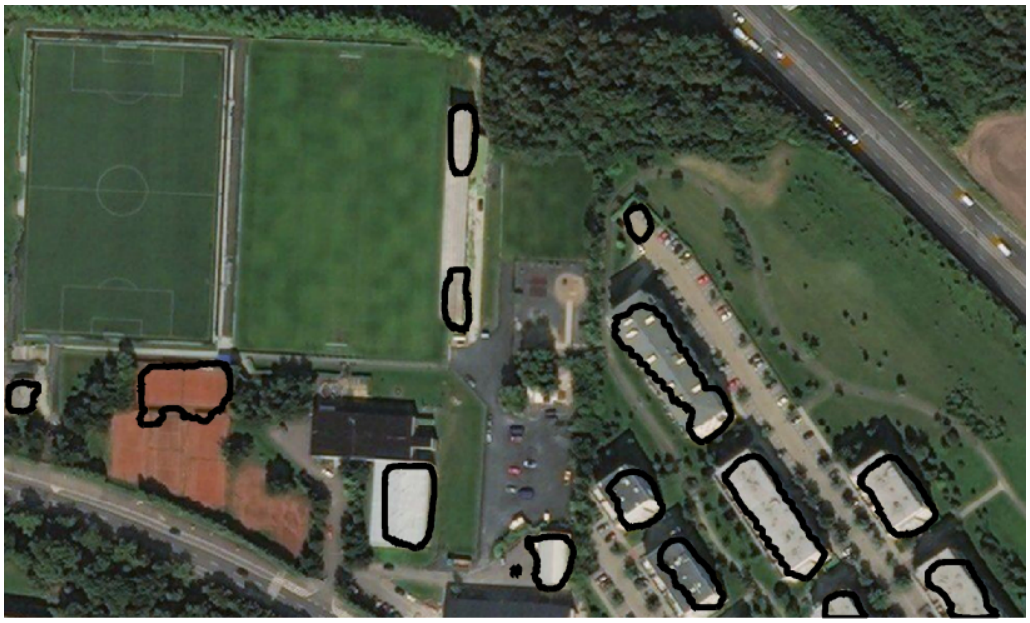


Figure B.9: An example of the detection; epoch 150, loss function 0.6280

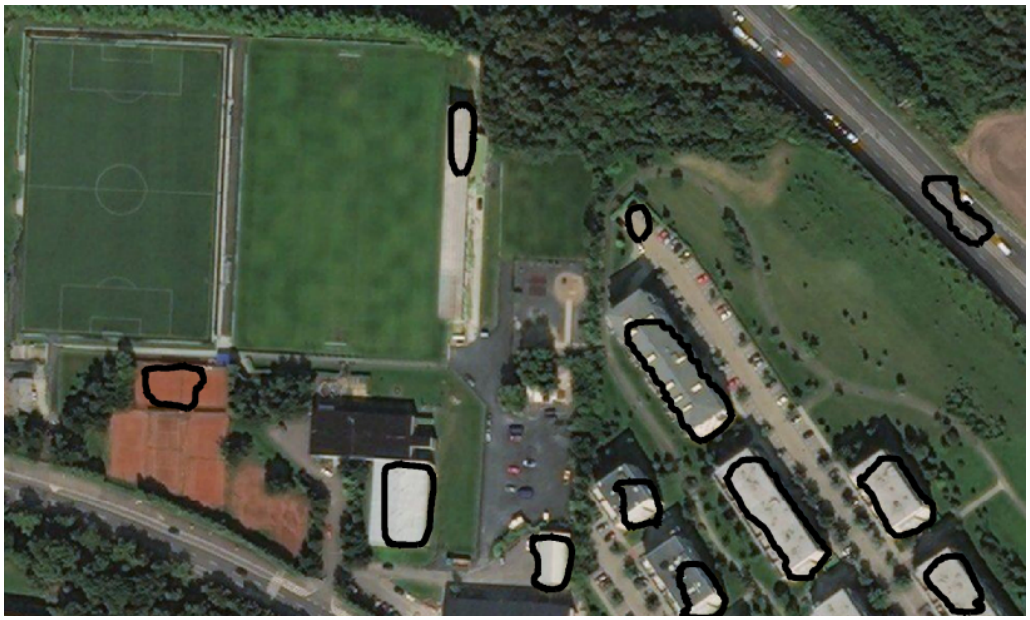


Figure B.10: An example of the detection; epoch 180, loss function 0.5019

C E-attachments

E-attachment of this thesis consists of following features:

- The source code of GRASS GIS modules and library
- GRASS GIS patch for Python 3