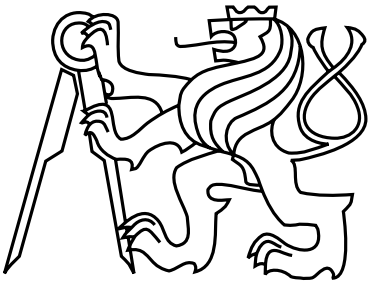
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering



Bachelor's Thesis

Human Detection from Aerial Vehicles Using Neural Networks

Andrii Zakharchenko

Supervisor: Ing. Milan Rollo, Ph.D.

Study Programme: Cybernetics and Robotics

Field of Study: Systems and Control

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zakharchenko** Jméno: **Andrii** Osobní číslo: **453318**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Systémy a řízení**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Detekce osob bezpilotními prostředky s využitím neuronových sítí

Název bakalářské práce anglicky:

Human Detection from Aerial Vehicles Using Neural Networks

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Ficenec Adam: Localization of UAVs from camera image, Diploma thesis, CTU in Prague, 2016.
[2] Hwai-Jung Hsu and Kuan-Ta Chen, 'Face Recognition on Drones: Issues and Limitations,' In Proceedings of ACM DroNet 2015, 2015.
[3] Alexander Toshev, Christian Szegedy: DeepPose: Human Pose Estimation via Deep Neural Networks. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1653-1660, 2014.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Milan Rollo, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **16.01.2018** Termín odevzdání bakalářské práce: **08.01.2019**

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Milan Rollo, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Datum převzetí zadání

Podpis studenta

Aknowledgements

I would like to thank Czech Technical University in Prague and all the instructors and professors for the knowledge that I gained here and especially for the challenges that I had to overcome.

I also take this opportunity to thank all my friends for their assistance throughout my studies. I am thankful to them for giving me the encouragements and for the wonderful times we spent together.

Finally, I want to express my deep gratitude to my family for their continuous help, support and love. I will always be grateful to them for the opportunity to explore different directions in my life and to learn all these amazing things that I have learned during these years.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Law no. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague .....

Abstract

The goal of this work is to propose a neural network model that will be suitable for detecting humans from aerial vehicles. For this task we studied artificial neural networks and especially convolutional neural networks and their suitability for the object detection. We also made an overview of existing deep learning frameworks and gathered dataset in order to train a network to detect humans from aerial images. We proposed several neural network models and trained them, however, all our models do not generalise and simply overfit to training dataset.

Contents

List of Figures

List of Tables

1 Introduction

Human detection from drones has a wide variety of usage. By enabling drones to recognise and detect people we can employ them in applications such as area patrolling, search and rescue missions, people flow analysis and many more.

The goal of this work is to create a neural network that will be able to detect people in an image that was shot from a drone. Here, by object detection we mean the following: given an image that contains an object to be detected we want to output a bounding box, a rectangle that tightly encloses that object. We will also try to create a fast network in order to process images in a real time.

Before we get to the main body of this thesis, let's firstly familiarise our self with existing methods for human detection from images and after we will present the structure of this work.

1.1 Human detection from images

1.1.1 Classical approach to object detection

Human detection is a sub-task of a more general object detection problem, so in this section we will describe the object detection task, which can be easily translated to the detection of humans. We can describe an object detection routine in several steps: firstly, we extract regions that might potentially contain objects, then we describe those regions or we can also say that we create *features*, then we classify those features as human or non-human and lastly we do post-processing, where we, for example, merge positive regions [26]. Let's further discuss each step in more details.

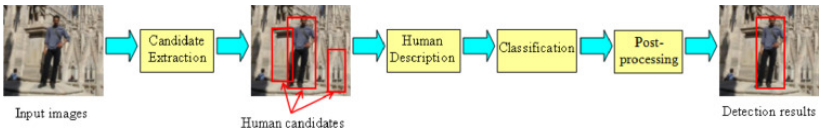


Figure 1.1: Object detection pipeline. Image from [26]

At the candidate extraction step we want to cover all possible areas of an image that can contain an object to be detected. The most simple approach to this is simply to extract regions without any prior knowledge about the object at various locations and various scales. The drawback of this method is that we will need to classify a lot of candidates later and this creates a computational bottleneck. Of course, we can add a prior knowledge about the object. For example, in human detection problem we can extract only those regions that are

taller than they are wider, thus reducing the number of proposals. However this could lead to decrease in recall since some instances of the object may not fall into this aspect ratio.

Another possible approach is to use selective search algorithm [15]. This algorithm uses image segmentation to extract candidates, it is relatively fast to compute and it creates less candidates then the "brute-force" approach presented above, which decreases computational time.

After candidate regions are created, we want to extract an information from them that will help us later classify those regions as human or non-human object. There are a lot of various features that can be extracted from an image and here we will name only few of them. The most widely used feature to represent information about shape is histogram of oriented gradients (HOG) [6] and scale invariant feature transform (SIFT) [5]. Haar-like features [2] and local binary patterns (LBP) (originally presented in [1] and used for human detection in [7]) are commonly used to represent information about texture.

Once the human descriptors are extracted from the candidate regions, the classification step is invoked to classify the candidate regions as human or non-human. For example, in [6] support vector machine (SVM) was used to make classification based on HOG features. In [4] learning algorithm based on AdaBoost with cascade architecture was used. Also, in [10] machine learning algorithm named Deformable Part-based Model (DPM) was used.

After we classified all proposed regions we may end up with multiple detections for the same object. In order to filter out those detections in post-processing stage we may use non-maximum suppression (NMS) algorithm. NMS is a key post-processing step in many computer vision applications. In the context of object detection, it is used to reduce multiple predictions in, ideally, a single bounding-box for each detected object.

There is a couple of issues with the classical approach to the object detection. Firstly, only one object descriptor is usually used (e.g HOG or SIFT) per detection algorithm and this object descriptor is hand-crafted. For example, in human detection problem people can have different poses, they can be occluded by other objects or just a part of a human body may be present in an image. On top of that, images can be taken in different light conditions and they may have different colour balance. Simply put, there is a lot of variety and this single hand-crafted object descriptor should be robust to all possible changes.

Secondly, we need to run an algorithm on multiple region proposals. Which is a big computational bottleneck.

1.1.2 Object detection with neural networks

The history of neural networks started in the middle of the 20th century and by the beginning of the 21st century all important concepts such as the backpropagation algorithm, multi-layer architecture, convolutional neural networks (CNN) and so on have been already discovered. However, the rise of the deep learning started only in 2012 when convolutional neural network called AlexNet [13] designed by Krizhevsky et al. won ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012). The task of the challenge was to correctly classify objects from ImageNet dataset. Before AlexNet the best classification error on the dataset was 0.26, while AlexNet achieved the error rate of 0.16. Later VGGNet [17], another deep CNN, achieved the error rate of 0.12. In 2015 CNN called ResNet [20] performed better than an average human and achieved the miss-classification rate of 0.036.

After CNNs showed great success in the classification task, researches started to employ neural networks in the detection task. The first successful object detection neural network was R-CNN [14]. In the Pascal VOC challenge 2012 it achieved the mean average precision (mAP) of 63 %, while the second best model was not neural network based and had the mAP of 40 %.

The idea behind R-CNN was to train it firstly to do the classification, then they adapted it to the detection domain and after that they run region proposals through the network to extract features and they also trained SVM that classifies those features. So the difference between classical approaches and R-CNN is that the last uses features created by the network, not hand-crafted features. R-CNN still uses selective search algorithm to generate region proposals and after that we need to run all those proposals through the network, also separate classification algorithm is used. All these steps were combined in a single neural network by the model named Faster R-CNN [24]. This model has the mAP of 74 % on Pascal VOC dataset and it's able to run at 5 frames per second (when R-CNN requires approximately 40 sec per image).

We will discuss object detection neural networks in greater details in chapter 4. Meanwhile, we can see that neural network based algorithms outperform classical approaches in terms of mean average precision. However, this doesn't mean that neural networks don't have drawbacks, on the contrary - a big and versatile dataset is required, as well as a lot of computing power to train a neural network. It is also harder to understand what networks learnt (how they "make decisions") compared to more simple algorithms. But the accuracy that they might have outweighs those disadvantages.

1.1.3 Results evaluation in object detection

After we trained an object detection model we want to somehow evaluate its performance. *Precision* and *recall* are most suitable metrics for this:

$$\text{precision} = \frac{TP}{TP + FP}$$
$$\text{recall} = \frac{TP}{TP + FN}$$

where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives.

Predicted bounding box is considered true positive if it matches ground truth box. Predicted boxes may not have exactly the same width and height - in order to solve this we will say that predicted box matches ground truth box if their intersection over union (IoU) is equal to or greater than 0.5.

$$\text{IoU} = \frac{\text{area}(\text{box}_1 \cap \text{box}_1)}{\text{area}(\text{box}_1 \cup \text{box}_2)}$$

If predicted bounding box does not match any ground truth box or it has IoU less than 0.5 it is considered false positive. Number of false negatives is the amount of ground truth boxes that were not predicted.

Metric called mean average precision is often used in object detection field, however it

is relevant only for multi-class detection. Since we are only concerned with the single class "human" we don't present this metric here.

1.2 Thesis structure

This thesis contains 8 chapters, including this introduction. In chapter 2 we will explain what are artificial neural networks, how they work and how they learn. Then, in chapter 3, we will explore specific type of neural networks called "convolutional neural networks" and we will examine why they are suitable for image processing. After that, in chapter 4, state-of-the-art neural network models for the object detection task will be discussed and, in chapter 5, we will make an overview of existing deep learning computer frameworks. Also, in order to train a neural network we require a data set of images that were shot from the high altitude and corresponding bounding boxes, this data set will be presented in chapter 6. We will talk about implementation details and results of this work in chapter 7. And in the end, in chapter 8, we will make final conclusions.

2 Artificial Neural Networks

2.1 What is artificial neural network?

Artificial neural networks (ANN) are computing systems inspired by biological neural networks that constitute human or animal brains. In the most general form, an ANN is a system designed to model the way in which a brain performs a particular task. Such systems learn to do tasks by considering given examples, generally without task specific programming. They have found most use in tasks where it is not possible to apply rule-based programming [3]. This makes an ANN a good candidate for human detection, since it is hardly possible to manually program a classifier which will consider all possible positions and all different appearances of humans.

An ANN is based on a collection of connected units called artificial neurons (analogous to axons in biological brain). Each connection (called synapse) between neurons can transmit a signal to another neuron. The receiving neuron can process an input signal and then send a signal to another neuron.

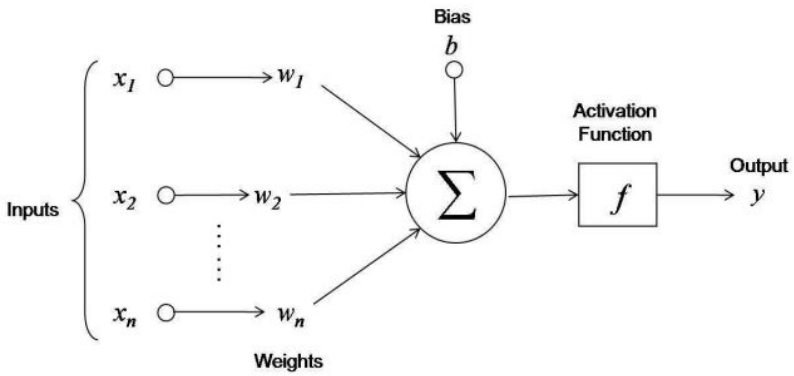


Figure 2.1: Graphical representation of artificial neuron. Image from [25]

Each neuron can have multiple inputs x_i which are multiplied by weights w_i and then summed in an adder and with added bias b sent to activation function f, see figure 2.1. We can describe a neuron by the two following equations [3]:

$$\phi = \sum_{i=1}^{n} w_i \cdot x_i + b \tag{2.1}$$

$$y = f(\phi) \tag{2.2}$$

where ϕ is the output of the adder, f is an activation function and y is the output of a neuron.

Activation function f is a non-linear function. In order to training a neural network, it is convenient to chose activation function that is differentiable. For example, sigmoid function, hyperbolic tangent function, SoftMax function and rectified linear unit are by far the most popular activation functions.

Typically, neurons are organised in layers, see figure 2.2. Signals travel from the first (input), to the last (output) layer, after traversing n hidden layers.

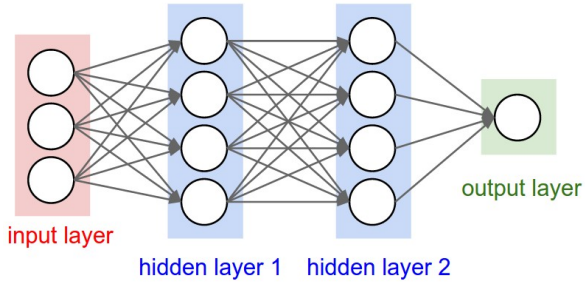


Figure 2.2: A four layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Image from [41]

These type of networks are called fully connected feed-forward networks.

2.2 Training process

To train a neural network means that we should find a set of weights Θ that will increase the accuracy of a network. To do this, firstly, we need a training set $X = \{(x_i, y_i)|i = 1...N\}$ and, secondly, we need to define an error (loss) function. An error function $E(X, \Theta)$ is a function, which defines the error between the desired output y_i and computed output $\hat{y}$ of the neural network. It is clear that we want the error to be as low as possible, so formally training task can be stated as follows[35]:

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}}\, E(X, \Theta) \tag{2.3}$$

Since this can't be solved analytically, gradient descent algorithm can be used. For this we need to compute a gradient of the error function with respect to all weights.

2.2.1 Computation of the gradient

Before deriving the gradient let's make some formal definitions:

- w_{ij}^k will denote a weight of neuron j in layer k, for incoming neuron i.
- b_j^k bias for neuron j in layer k

- M_k number of neurons in layer k
- o_i^k output of neuron i in layer k
- $a_j^k = b_j^k + \sum_{i=1}^{M_{k-1}} w_{ij}^k o_i^{k-1}$
- f is activation function and f_0 activation function in last layer.

Further, we can simplify math by incorporating bias term into weights $w_{0j}^k = b_j^k$. To do this we need to add fixed output to layer $k-1$, $o_0^{k-1} = 1$. Now we can rewrite weighted product-sum as

$$a_j^k = \sum_{i=0}^{M_{k-1}} w_{ij}^k o_i^{k-1} \tag{2.4}$$

For the sake of example mean-squared loss function will be used (2.5), we also assume the neural network with only one output.

$$E(X, \Theta) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{2.5}$$

Now we need to calculate derivatives of the loss function with respect to all weights.

$$\frac{\partial E(X, \Theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial}{\partial w_{ij}^k} \frac{1}{2} (\hat{y}_d - y_d)^2 = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial E_d}{\partial w_{ij}^k} \tag{2.6}$$

To simplify math further we can write derivatives for loss function $E_d = \frac{1}{2}(\hat{y}_d - y_d)^2$ using derivative chain rule and later substitute it back to equation 2.6:

$$\frac{\partial E_d}{\partial w_{ij}^k} = \frac{\partial E_d}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \tag{2.7}$$

The first term on the right hand side is usually called an error and denoted as:

$$\delta_j^k = \frac{\partial E_d}{\partial a_j^k} \tag{2.8}$$

The second term can be calculated from the equation 2.4:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = o_i^{k-1} \tag{2.9}$$

And from this we get:

$$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k \cdot o_i^{k-1} \tag{2.10}$$

Still term δ_j^k needs to be calculated. It will be shown that this term depends on the values of error terms in the next layer. Thus, computation of the error terms will proceed backwards from the output layer down to the input layer. This is where backpropagation, or backwards propagation of errors, gets its name.

Firstly, let's calculate derivatives of the *output layer* m. Network in this example has only one output, which means that there is only one neuron in the last layer, hence we need to calculate error term only for one neuron:

$$\delta_1^m = (f_0(a_1^m) - y) f_0'(a_1^m) = (\hat{y} - y) f_0'(a_1^m) \tag{2.11}$$

From the equations 2.7, 2.8, 2.9, 2.10 and 2.11 we can rewrite the derivative of error function w.r.t all weights in the output layer as follows:

$$\frac{\partial E_d}{\partial w_{i1}^m} = (\hat{y} - y) f_0'(a_1^m) o_i^{k-1} \tag{2.12}$$

Than we also need to compute the error term for all hidden layers k, $1 \leq k < m$:

$$\delta_j^k = \frac{\partial E_d}{\partial a_j^k} = \sum_{l=1}^{M_{k+1}} \frac{\partial E_d}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{M_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k} \tag{2.13}$$

Now, noting that $a_l^{k+1} = \sum_{j=1}^{M_k} w_{jl}^{k+1} g(a_j^k)$ we get:

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} f'(a_j^k) \tag{2.14}$$

Putting it all together, the partial derivative of E_d w.r.t a weight w_{ij}^k in the *hidden layer* is:

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = o_i^{k-1} f'(a_j^k) \sum_{l=1}^{M_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \tag{2.15}$$

2.2.2 Gradient descent algorithm

Now we have rules to compute the gradient of the error function. This gradient will be used in update rule

$$\Theta^{t+1} = \Theta^t - \alpha \frac{\partial E(X, \Theta^t)}{\partial \Theta} \tag{2.16}$$

Where Θ^t denotes the parameters of the network at step t and parameter α is often called learning rate.
Now, after we know how to compute gradient of the loss function with respect to all weights, we further use gradient descent algorithm to perform weight updates. The gradient descent

with back propagation proceeds in the following steps [30]:

input : α, $(x_i, y_i) \in X$, Θ, network
returns: Θ^*
begin
  $\Theta^t \leftarrow \Theta$;
  while *Convergence criteria is not satisfied* **do**
    Store $\hat{y}$, a_j^k, o_j^k *for every example in data set*;
    $\hat{y}_{\ i}, a_j^k, o_j^k \leftarrow$ `ForwardPass(`X, *network*`)`;
    Compute gradient of E *w.r.t* Θ;
    $\nabla E \leftarrow$ `ComputeGradient(`$\hat{y}_i$, a_j^k, o_j^k, *network*, Θ^t`)`;
    Update weights;
    $\Theta^{t+1} \leftarrow \Theta^t - \alpha \nabla E$;
    $\Theta^t \leftarrow \Theta^{t+1}$;
  end
  $\Theta^* \leftarrow \Theta^t$
end

Algorithm 1: Gradient descent algorithm

The problem with this algorithm is that we compute gradient for entire training set, which can be problematic for very large data set, since training examples can take up all available memory. To combat this problem mini-batch gradient descent algorithm is used [30]:

input : α, $(x_i, y_i) \in X$, Θ, network, batchSize
returns: Θ^*
begin
  $\Theta^t \leftarrow \Theta$;
  while *Convergence criteria is not satisfied* **do**
    for $(x_{i:i+batchSize}, y_{i:i+batchSize}) \in X$ **do**
      $\hat{y}_{\ i}, a_j^k, o_j^k \leftarrow$ `ForwardPass(`$(x_{i:i+batchSize}, y_{i:i+batchSize})$, *network*`)`;
      Compute gradient of E *w.r.t* Θ;
      $\nabla E \leftarrow$ `ComputeGradient(`$\hat{y}_i$, a_j^k, o_j^k, *network*, Θ^t`)`;
      Update weights;
      $\Theta^{t+1} \leftarrow \Theta^t - \alpha \nabla E$;
      $\Theta^t \leftarrow \Theta^{t+1}$;
    end
  end
  $\Theta^* \leftarrow \Theta^t$
end

Algorithm 2: Mini-batch gradient descent algorithm

As we can see we divide a training set in parts that are called batches and we perform a weight update for every batch. This helps to improve memory efficiency. Another benefit of this algorithm is that it performs weight updates more frequently, this way it reduces variance in weight updates which can lead to a more stable convergence [30].

2.3 Summary

In this chapter we introduced simple building blocks of artificial neural networks and presented how ANN's are trained. However, simple fully-connected networks are not suitable for image processing task, since they don't take into account spatial information and would require a lot of weights to be able to process images, which in turn leads to a slow performance. In the next chapter we will introduce convolutional neural networks that designed specifically to handle images as inputs, however their applications are not limited only to image processing.

3 Convolutional neural networks

3.1 Introduction

Convolutional neural networks (simply CNN) are special type of feedforward networks. They are also made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and follows it with an activation function. We still can employ learning algorithms studied in section 2.2. But the main difference from regular feedforward networks is that an explicit assumption about an input of CNN is made, it should have a known grid-like topology (e.g. images) [27]. This assumption helps us to efficiently encode properties of inputs by introducing convolution operation into the architecture. As we will see in next sections convolution helps us to reduce the number of parameters that need to be learned and also allows network to adapt to spatial arrangement of an input.

3.2 Convolution operation

As was said in previous section CNN employs a mathematical operation called convolution. It is defined for two functions f and g where one of them is reversed and shifted [27]:

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{3.1}$$

In CNN terminology, the first function f is often referred as the input and the second g as the kernel. The output is sometimes called a feature map. Since in practice we usually work with discrete time, it is convenient to define discrete convolution as follows [27]:

$$s(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) \tag{3.2}$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is also multidimensional array of parameters adapted by the learning algorithm. Because each element of the input and kernel must be stored explicitly, we usually assume that those functions are zero everywhere but the finite set of points for which we store the values. In practise this means that we can implement infinite summation as summation over finite set of array elements. Additionally, when processing 2D image I we want to use convolution over two dimensions at the same time for this we can use a two-dimensional

kernel function K. With observations above we can rewrite convolution operation as follows [27]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.3}$$

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{3.4}$$

As we can see above, convolution is a commutative operation. This property arises because we have flipped the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. There is a related operation called *cross-correlation* that is widely used in convolutional neural networks [27]:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.5}$$

Its main difference from convolution is that it is not commutative but does not require flipping of the kernel. Moreover,

$$S(i,j) = (I * K)(i,j) = (I \star K)(i,j) \tag{3.6}$$

where by the symbol $*$ is denoted the operation of convolution and by the symbol $\star$ operation of cross-correlation.

Convolution operation is also used in image processing. For example, we can use convolution to sharpen input image. In fig. 3.1 we can see visualisation of convolution operation. In this figure kernel with size 3x3 is applied to the input image, intensity of each pixel in 3x3 region of the input image is multiplied by parameters of a kernel and added up to generate the output.

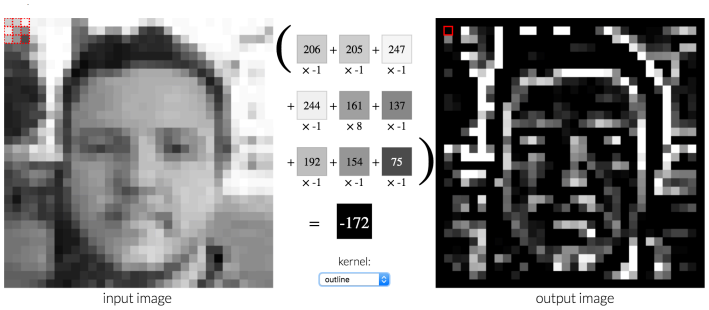


Figure 3.1: Visualisation of convolution. Image from http://setosa.io/ev/image-kernels/

3.3 Architecture of CNN

A simple convolutional neural network is a sequence of layers that perform certain transformations on the input. Three main types of layers are used to build a CNN: *convolutional layer*, *pooling layer* and *fully-connected layer*. We can combine these core building blocks to construct a convolutional neural network as shown in fig. 3.2. CNN in this figure uses two convolutional layers with kernels of size 5×5, two pooling layers (denoted as sub-sampling layer) and fully connected layer at the end to classify an input image.

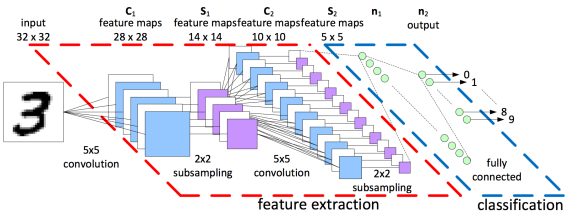


Figure 3.2: CNN architecture for a handwritten digit recognition task. Image from [11]

3.3.1 Convolutional layer

The convolutional layer (CL) is the layer where network performs convolution operation on an input. CL consists of set of learnable filters (also known as kernels), that have some predefined size $W \times H$. For example, in fig. 3.2 filters in first CL have size 5×5. When an input is fed to the layer, we slide those filters along width and height of the input performing convolution operation. The output of the layer will be a predefined number of feature maps.

Since images consist of three channels (red, green and blue) and we want to apply several filters to all of those channels to get multiple feature maps, it is convenient to describe input, output and kernels as tensors and to rewrite convolution operation in terms of tensors. First, let's consider l-th convolutional layer in the network and describe it's input, output and kernels mathematically [34]:

- Input to the l-th layer $\mathbf{I}^l$ is a tensor of the 3-rd order such that $\mathbf{I}^l \in \mathbb{R}^{H^l \times W^l \times D^l}$. Thus we need triplet of indexes $(0 \leq i^l < H^l, 0 \leq j^l < W^l, 0 \leq d^l < D^l)$ to address an element in the input. Note that zero-based indexing is used to simplify further equations.
- Output of the l-th layer $\mathbf{y}^l$ is also 3-rd order tensor, $\mathbf{y}^l \in \mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}}$. Indexes used to address elements in the output are $(0 \leq i^{l+1} < H^{l+1}, 0 \leq j^{l+1} < W^{l+1}, 0 \leq d^{l+1} < D^{l+1})$
- Kernel of the l-th layer $\mathbf{K}^l$ is 4-th order tensor, such that $\mathbf{K}^l \in \mathbb{R}^{h \times w \times D^l \times D}$. The reason for using 4D kernels is to take into account the number of channels D^l of the input $\mathbf{I}^l$. $\mathbf{K}^l = \{K^l_{i,j,d^l,d}\}$, such that $0 \leq i < h, 0 \leq j < w, 0 \leq d^l < D^l, 0 \leq d < D = D^{l+1}$

With the definitions above we can rewrite convolution operation in terms of tensors:

$$y_{i^{l+1},j^{l+1},d} = b_d + \sum_{i=0}^{h} \sum_{j=0}^{w} \sum_{d^l=0}^{D^l} K_{i,j,d^l,d} \cdot I^l_{i^{l+1}+i,j^{l+1}+j,d^l} \tag{3.7}$$

This equation is repeated for all spatial locations (i^{l+1}, j^{l+1}) and for all output channels d. In this equation the term b_d represents a bias of channel d.

It is worth noting, that after convolution operation the size of the output will be smaller than the size of the input. To calculate the size of the output next equations could be used:

$$H^{l+1} = H^l - h + 1 \tag{3.8}$$

$$W^{l+1} = W^l - w + 1 \tag{3.9}$$

If we want to control the size of the output without changing the size of the kernel, we can introduce the concept of ***zero-padding***. The idea is that we will increase the size of the input by adding zeros to the borders as shown in fig. 3.3. Parameter P will be used to denote the amount of zero-padding used. For example, in fig. 3.3 $P = 1$, since we added 1 layer of zeros.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 3.3: A zero-padded 4 x 4 matrix becomes a 6 x 6 matrix. Image from [31]

Another crucial parameter that is used in convolutional layer is ***stride***, denoted by S. Stride tells us with what step we will slide filters along the input image. For example, if $S = 1$ that means that we slide filter one pixel at a time, when $S = 2$ filter will jump 2 pixels at a time. Obviously this means, that with bigger stride we will get smaller output. To calculate the size of the output with zero-padding and stride we can use following equations:

$$H^{l+1} = \frac{H^l - h + 2P}{S} + 1 \tag{3.10}$$

$$W^{l+1} = \frac{W^l - w + 2P}{S} + 1 \tag{3.11}$$

Another important feature of a CL is an activation function that is used. It is important to add non-linearities to the network if we want to classify non-linear data. We have a big choice of activation functions, for example, as those discussed in sec. 2.1. However, the most used one in CNN is a ReLU (rectified linear unit), which is computed as $f(x) = max(x, 0)$. Advantage of the ReLU is that it's gradient is simply 0 or 1 depending on sign of x, which helps to eliminate the problem of vanishing gradient during training.

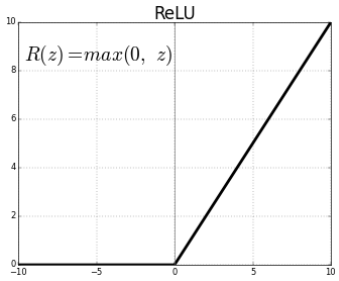


Figure 3.4: Graph of the ReLU function

To summaries on the convolution layer:

- Parameters required to set up CL:
  - Number of output channels (also number of filters) D
  - Number of input channels D^l
  - Size of kernels h and w
  - Stride S
  - Zero-padding P.
- Number of learned weights is $h \times w \times D^l \times D$ plus biases D
- Size of the output:
  - $H^{l+1} = \frac{H^l - h + 2P}{S} + 1$
  - $W^{l+1} = \frac{W^l - w + 2P}{S} + 1$

3.3.2 Pooling layer

It is common to insert a pooling layer between convolutional layers. Its function is to reduce the spatial size of feature maps produced by convolutional layers. In the figure 3.5 we can see how pooling layer operates on a feature map. It divides an input into sub-regions and it propagates further some summary statistics (max value, average value, etc.) on values inside those region. For example in fig. 3.5 max pooling was used, with this type of pooling only the highest values in sub-regions will be propagated further. It is possible to use different pooling functions, for example, the average pooling or the L2-norm.

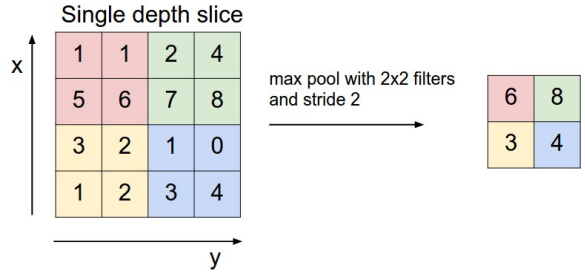


Figure 3.5: The most common downsampling operation is max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square). Image from [41]

The main reason behind using pooling layer is to make representation of input become invariant to small translations. Invariance to translation means that if we translate an input by a small amount the majority of pooled values should not change. Also, using pooling layer increases computational efficiency, since representation of input becomes smaller and smaller after each pooling layer.

To set up a pooling layer we need to know two parameters *stride* and *spatial size*. Those parameters have the same meaning as a stride and a kernel size in the convolutional layer.

3.3.3 Fully connected layer

After series of convolutional and pooling layers we usually use fully connected (FC) layer to produce classifications. Fully connected layer is the layer where all outputs of previous layer are connected to all neurons in this layer. Though it is not compulsory to use FC layer after convolutions, it can increase overall model accuracy.

3.4 Motivation for using CNN

Convolution has several features that help to improve neural networks: *sparse interaction*, *parameter sharing* and *equivariant representation* [27]:

3.4.1 Sparse interactions

In traditional neural networks, as shown in fig. 2.2, all neurons from adjacent layers are interconnected. We can describe this interaction between layers as matrix multiplication, where one matrix will represent outputs of the first layer and another matrix will represent weights of the second layer. If there are m neurons in first layer and n neurons in the second layer, matrix of weights will consist of $m \times n$ elements and matrix multiplication will have $O(mn)$ complexity. Convolutional neural networks, on the other hand, have *sparse interactions* between layers, this is accomplished by using kernel smaller than the input. So if we limit number of connections each neuron in the second layer may have to some k that is way smaller than m, weight matrix will consist of $k \times n$ elements and multiplication will have complexity of $O(kn)$.

3.4.2 Parameter sharing

Parameter sharing refers to using the same parameters in more than one place. For example, in a traditional NN each weight is used exactly once and never revisited. In a CNN, however, each element of kernel is used at every position of input. The parameter sharing used by convolution operation means that rather that learning different sets of weights for every location, we only learn one set. Parameter sharing reduce the storage requirements of the model to k parameters.

3.4.3 Equivariant representation

In the case of convolution, parameter sharing causes the system to have a feature called *equivariance* to translation. To say that a function is equvariant means that if the input changes, the output changes in the similar way. When processing time series data convolution produces a sort of time-line that shows when different features appear in the input. If we move an event later in time, the same representation of it will appear in the output, just later in time. Same goes for images - convolution creates a 2-D map of where certain features appear in the input. If me move the object in the input, its representation in feature map will move for the same amount.

3.5 Summary

Using convolutional neural networks for image processing brings several important advantages in comparison with traditional NN. Firstly, we can reduce complexity of algorithm, secondly we can reduce storage requirements to store learned parameters. To see how dramatic these improvements are, let's consider an example where 512×512 greyscale image is fed to traditional neural network with number of inputs equals to the number of pixels in input image, $N_{in} = 262144$. Let's assume that the second layer has at least the same amount of neurons N_2 as input layer. To describe interactions between the input layer and the second layer we will need $N_{in} \times N_2$ parameters or roughly 68.7 billion and matrix multiplication will have complexity $O(N_{in} \times N_2)$. Now, let's consider an example where the same image will be the input to a convolutional neural network where first layer has 10 kernels with 5×5 size. Using formula from sec. 3.3.1 we can calculate the amount of weights used in first layer: $N = h \times w \times D^l \times D + D = 260$. We can see that the number of parameters in the first layer of the CNN is significantly less comparing to a traditional NN. The fact that we need more parameters in simple feedforward neural networks means they are much more prone to overfitting than CNN. Complexity of computing convolution, given that we use stride $S = 1$ and zero-padding $P = 0$, can be computed as follows $O(h \times w \times D^l \times D \times H^{l+1} \times W^{l+1}) = O(65025000)$, which is roughly 10^3 times faster than in a traditional NN.

Another advantage is that by applying convolution we extract same features from different position, meaning that if object of interest appears in different places of image, it will just appear in a different place on a feature map produced by convolution, while in traditional NN we extract different features from different positions.

4 Convolutional neural network for object detection

4.1 Introduction

Currently there are several approaches to object detection with convolutional neural networks. We can divide these approaches into two sets: the first set would be based on region-based convolutional neural networks (simply R-CNN) models and the second group can be called "single shot models".

The core idea of R-CNN methods is to extract features using CNN and with these features classify regions that were provided externally (e.g. with selective search algorithm) or internally (see section 4.2.3) as some object. Family of R-CNN models has 3 most significant architectures. First architecture was introduced in [14] and became known simply as R-CNN. This work was crucial in terms of developing approaches for object detection with CNN, but proposed model was complicated and very slow. Then improvement to this architecture was presented in [19] and was called Fast R-CNN. This model was much faster and simpler that R-CNN, but still wasn't good enough for real-time detection. The new model was introduced in [24] to combat slow performance of Fast R-CNN and was called Faster R-CNN.

The group of single shot models is based on a following idea: giving an input image we virtually divide it into $N \times N$ grid. Each grid cell is now responsible for detection of an object, which centre lies in this cell. Term *virtually* means that we don't need to divide the image into $N \times N$ smaller images and pass those images through the network, allowing to make predictions in one forward pass. Most significant architectures in this group are You Only Look Once (YOLO) [23], Single Shot MultiBox Detector (SSD) [22] and YOLOv2 [28], which is improvement to YOLO model. The main advantage of these models is their high performance in terms of frames per second.

In following sections models that were presented above will be discussed in more details.

4.2 R-CNN family

4.2.1 R-CNN

The pipeline of R-CNN model works in following way: firstly, the selective search algorithm is applied on an image to create region proposals, then these proposals are resized and fed to CNN to extract features, after that support vector machines (SVM) are used to classify those regions as an object using features extracted by CNN. And the final step is bounding

box regression. The purpose of this step is to get more precise boxes for objects.

The main drawbacks of this approach are

- We need to pass through the model **all** region proposals produced by selective search algorithm, which can take 40 to 50 seconds per image.
- By using CNN, SVM and Bounding Box regressor we can't train the model as a single pipeline, which increases training time.
- To train the SVM we need to extract features from CNN and store them on disk.

4.2.2 Fast R-CNN

Fast R-CNN was designed to eliminate main disadvantages of R-CNN model. Again the selective search algorithm was used to provide region proposals, but now the idea is to share computations of CNN across all region proposals so it will take only one forward pass of an image. This was achieved by introducing RoI(region of interest) pooling layer. RoI pooling takes as inputs region proposals (or regions of interest) and feature maps from last convolutional layer of CNN. Regions of interest are defined by a four-tuple (x, y, h, w), where (x, y) are coordinates of top-left corner and (h, w) are height and width; pooling layer has kernel of size $H \times W$. On the feature map that represents an input image we will take a smaller region which is associated with a RoI defined by (x, y, h, w) and we will divide it into grid $h/H \times w/W$ and from each cell maximum value is taken to produce a feature map that represents this region of interest. This procedure is illustrated in fig. 4.1, bigger window represents some region of interest and smaller windows are region where max values will be taken from.

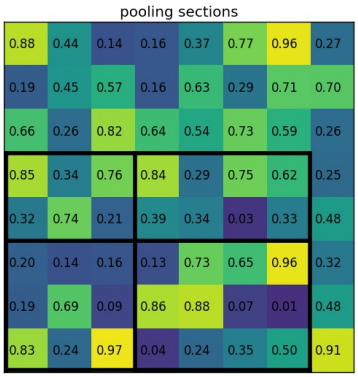


Figure 4.1: RoI pooling. Size of the region of interest doesn't have to be perfectly divisible by the number of pooling sections (in this case RoI is 7×5 and we have 2×2 pooling sections). Image from [33]

RoI pooling will output one feature map per region proposal. After RoI pooling follow two fully connected layers that are branched into two other FC layers one with softmax loss and another with bounding box regression.

Advantages of this model are that it can be trained in a single run, since SVMs are not used and time of forward pass is now about 2 seconds per image. But 2 s/image still makes this model not suitable for real-time detection. And the computational bottleneck of this model appears to be the algorithm that is used for generating region proposals.

4.2.3 Faster R-CNN

Faster R-CNN was developed to address the issue of slow generation of region proposals. The main idea was that region proposals depended on features of the image that were already calculated with the forward pass of the CNN. So why not reuse those results for region proposals instead of running a separate selective search algorithm? This was achieved by introducing Region Proposal Networks (RPN), that take the feature maps from last convolutional layer as an input and output multiple predictions of bounding boxes. Then, as in Fast R-CNN, those region proposals are fed to RoI pooling layer.

The Region Proposal Network works by passing a sliding window over the CNN feature map and at each window location, outputting k potential bounding boxes and scores that describe certainty that this box contains an object, regardless of class of that object (see fig. 4.2). These k bounding boxes are called anchor boxes, their size and aspect ratio are chosen in advance. Since last convolutional layer will produce multiple feature maps, and region proposals are generated at each location of sliding windows, a big number of proposals will be generated. Non-maximum suppression algorithm is used to filter out some proposals based on their certainty scores.

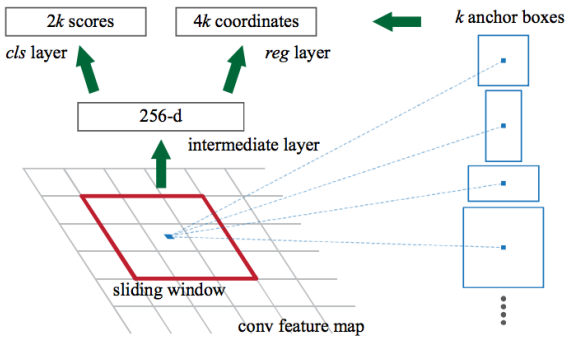


Figure 4.2: Region proposal network. Image from [24]

Faster R-CNN showed good performance and were able to process around 5 images per second, which is much more better when comparing with R-CNN and Fast R-CNN. 5 fps is close to real-time detection.

4.3 Single Shot Models

4.3.1 You only look once (YOLO) model

You only look once model divides the input image into $S \times S$ grid. If the center of an object falls into a grid cell, then that cell is responsible for detecting this object. Each cell predicts

B bounding boxes. Also, each cell predicts C class conditional probabilities $P(class_i|object)$. Each bounding box consists of 5 predictions: (x, y, h, w, p). The (x, y) two-tuple represents the center of the bounding box relative to the grid cell; (h, w) coordinates are width and height of a box relative to the whole image. And p is a confidence score, that represents how model is confident that the box contains an object and how accurate this box is.

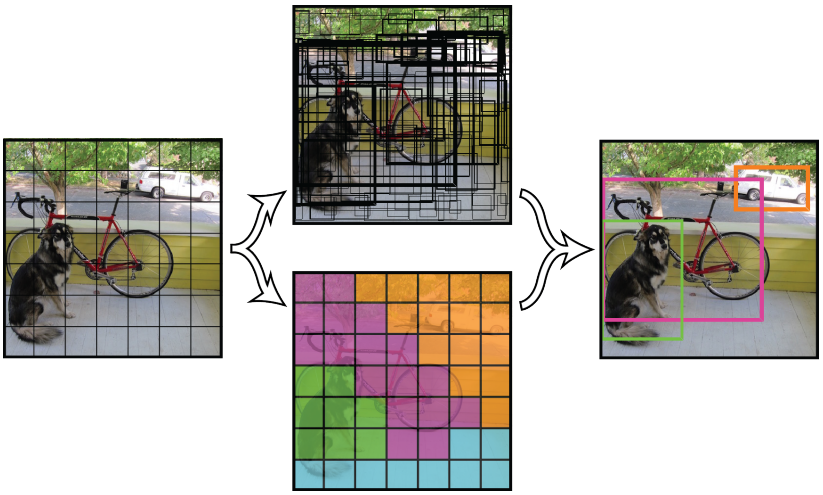


Figure 4.3: YOLO detection principle. Image from [23]

Finally all these predictions are encoded into $S \times S \times (5B + C)$ tensor. As shown in fig. 4.4, architecture of YOLO consists of several convolutional layers that are followed by a fully connected layer that is reshaped into $S \times S \times (5B + C)$ tensor. Detection procedure is performed with non-maximum suppression algorithm.

YOLO model can process 45 images per second, which makes it a good candidate for real-time object detection. However, this model has several disadvantages:

- Each cell can predict only one object, that could be problem if centres of two and more objects lie in one cell.
- Mean average precision (mAP) of YOLO model is 63.4, when Fast and Faster R-CNN have more than 70 mAP on the same dataset.

4.3.2 Single shot detector(SSD)

Single shot detector is based on the idea that all feature maps can be used to predict class and location of an object. By utilising feature maps from several different layers in a single network, it is possible to handle different sizes and shapes of objects. In fig. 4.4, we can see that several layers of a CNN are connected to detection layer.

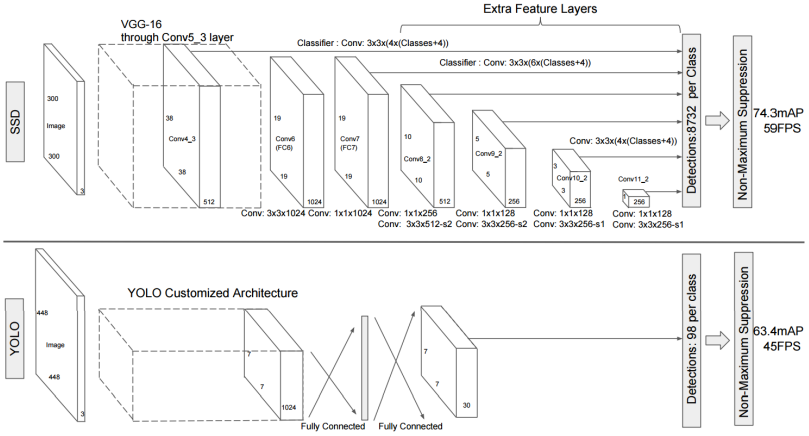


Figure 4.4: SSD architecture (top) and YOLO architecture (bottom). Image from [22]

To produce class and location prediction SSD associates k number of default boxes (similar to anchor boxes in R-CNN) at each position of a feature map. For each default box at each location in a feature map SSD predicts 4 offset coordinates (x, y, h, w) relative to default box. Further, for each box out of k at a given location c class scores are predicted. So given one feature map of size $m \times n$ we can apply $s \times s \times$ convolution kernel with $(c+4)k$ channels to predict kc sets of class scores and $4k$ sets of coordinates. This will give a total of $(c+4)kmn$ outputs that encode coordinates and confidence scores for bounding boxes. After all predictions are computed (across whole network) non-maximum suppression algorithm is applied to choose best bounding boxes.

SSD provides 74.3 mAP on Pascal VOC2007 data set, which is comparable with Fast and Faster R-CNN and at the same time can operate at 46 FPS for 300px input images, though for input images of size 500×500 its FPS drops to 19.

4.3.3 You Only Look Once version 2 (YOLOv2)

YOLOv2 was designed to eliminate limitations of the YOLO model introduced in the section above. YOLOv2 still divides an image into $S \times S$ grid and each cell is responsible for prediction of k bounding boxes. Though, now they use boxes with predefined aspect ratios, similar to Faster R-CNN and SSD, and now each box is responsible for predicting an object (not cell as in YOLO). Each of k boxes predicts 5 values $(t_x, t_y, t_w, t_h, t_0)$. Values (t_x, t_y) are parameters that are responsible for predicting coordinates of the center of the box relative to the cell. Values (t_w, t_h) are parameters that are responsible for predicting width and height of the box, however, now prediction also depends on width and height of default boxes. We can see that total number of predictions to make is $S \times S \times k(5 + c)$, where c is the number of classes.

Another improvement is that the fully connected layer was removed. This allowed to resize model on the fly and to train network with images of different sizes. This is somehow similar to SSD, which uses multiple feature maps to detect objects of various sizes, however, YOLOv2 doesn't need additional computations during forward pass. Also the passthrough layer was added to make predictions on fine grained features. Passthrough layer simply takes feature maps from earlier layer and concatenates them with low resolution features by

stacking adjacent features into different channels.

All these changes improved not only precision of the network (depending on size of the input image 69-78 mAP was reported in paper), but also frame rate. With the input of size 544×544px network was able to process 40 frames per second, while with low resolution of input (288×288px) it is able to run on 91 fps.

4.4 YOLO v3

Even though YOLO v3 [38] wasn't around when we started to work on this thesis, we still decided to include it to this overview to complete the picture of the state-of-the-art object detection models.

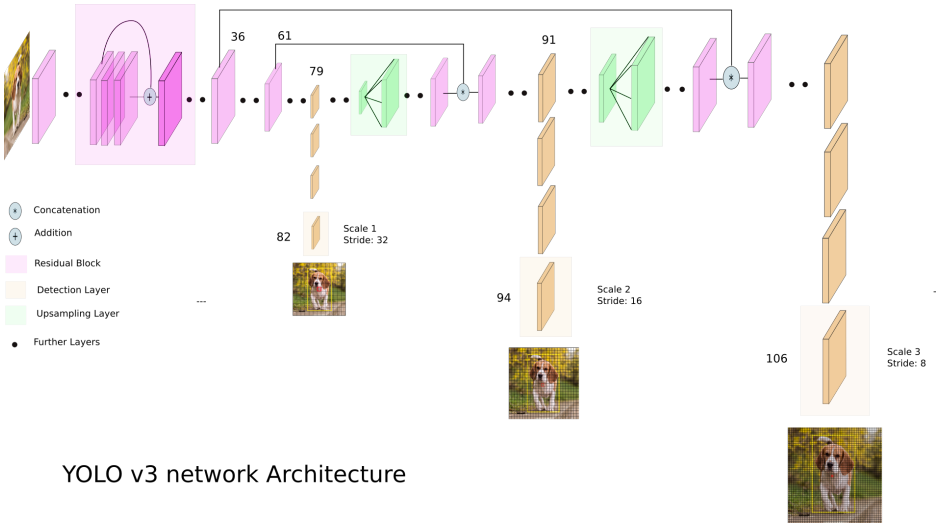


Figure 4.5: Architecture of YOLO v3. Image from [36]

In figure 4.5 we can see the architecture of YOLO v3 model, it is still a fully convolutional neural network as YOLO v2. Though v3 is bigger than its predecessors, it has 106 layers in total and utilises state-of-the-art techniques such as residual blocks, skip connections and upsampling layers. Another change is that yolo now predicts objects at three different scales. First detection layer is the 79th layer in the network and predictions are made for 13×13 grid size, second detection layer is the 91st layer with grid size of 26×26 and the 3rd detection layer is the last layer in the network with grid size of 52×52. YOLO v3 uses 9 anchor boxes in total - 3 per each detection layer. The idea behind this architecture is to address the issue of predicting small objects in previous versions. Even though 3rd version has more than hundred layers it is still relatively fast, not as fast as previous models, but it is capable to run in 51 ms on Titan X GPU

4.5 Conclusion

Since detection of objects from drones requires real-time processing it is obvious that most suited models for this are YOLO, SSD or YOLOv2. YOLO has worse accuracy compared

with sate-of-the-art systems, but its simple architecture can be a great benefit. SSD and YOLOv2 have more complicated architectures, but they also have better performance. SSD, however, has worse performance than YOLOv2 in terms of accuracy and frame rate and also its architecture is more complicated than of YOLOv2.

	R-CNN	Fast R-CNN	Faster R-CNN	YOLO	SSD	YOLOv2
mAP	58.5	70.0	76.4-73.2	63.4	76.8-74.3	78.6-69.0
FPS	0.025-0.02	0.5	5-7	45	19-46	40-91

Table 4.1: Comparison of mean average precision and frames per second rate of different models. Models were trained on Pascal VOC 2007, 2012 data sets (R-CNN was trained on VOC 2007 only). Data taken from [28]

5 Deep learning computer frameworks

5.1 Overview

A deep learning framework is a set of tools that provides building blocks for designing, training and validating deep neural networks. A big amount of such frameworks exists and they are all constantly changing. However, only few of them have been widely accepted.

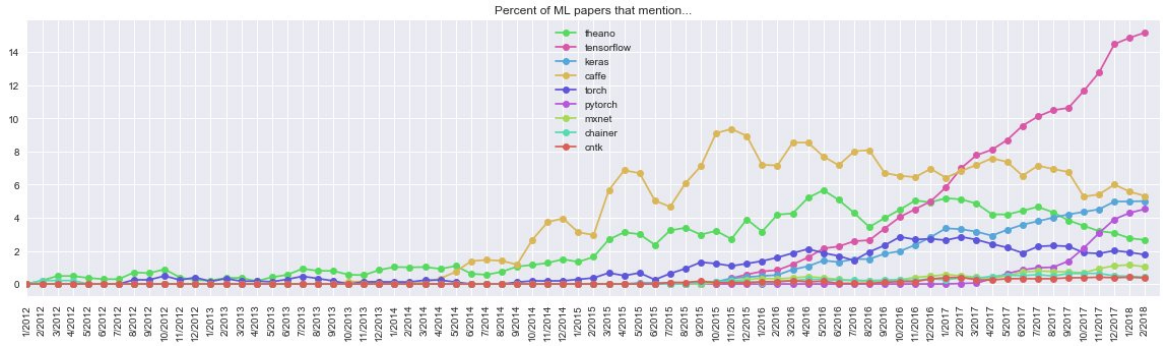
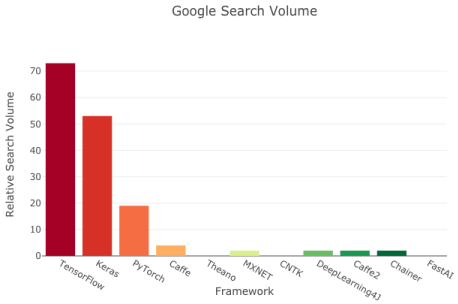


Figure 5.1: Unique mentions of deep learning frameworks in arxiv papers. Andrej Karpathy (@karpathy). 9 march 2018, 6:19 pm. on Tweeter.

For example, in Figure 5.1 percentage of arXiv articles for a given month that mention a given framework is shown. We can see that TensorFlow has the most mentions and shows a steady growth, also PyTorch, Keras and Caffe are among frequently mentioned frameworks, though Caffe is now in a decline.

The dataset from [40] contains information about popularity of 11 deep learning frameworks based on the following categories: Online Job Listings, KDnuggets Usage Survey, Google Search Volume, Medium Articles, Amazon Books, ArXiv Articles, GitHub Activity. We used code provided by the author of the dataset to create figures bellow.

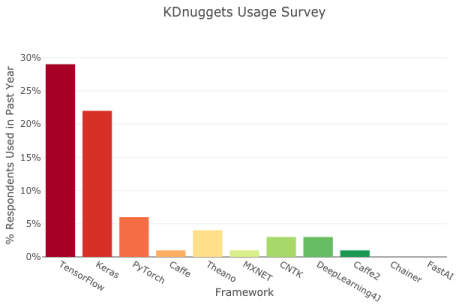
(a) Number of job listing that mention given framework.



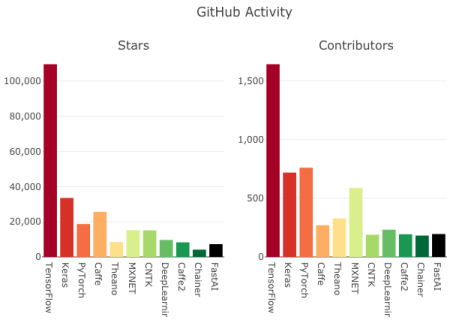
(b) Google search trends

Figure 5.2: Popularity of the frameworks among job listings and in Google search trends

Figure 5.2a shows the number of job listings that mention a given framework and Figure 5.2b shows search results from Google trends, however, Google doesn't provide absolute search numbers - it provides relative numbers. Here we can see that TensorFlow, Keras, PyTorch and Caffe are most popular frameworks among employers and they are also the most searched frameworks on Google.



(a) KDnuggets usage survey



(b) Github activity

Figure 5.3: KDnuggets survey results and GitHub activity

Figure 5.3a shows the results from KDnuggets survey named Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis [43]. KDnuggets is a popular website among data scientists. Here we can see that TensorFlow, Keras and PyTorch are among the leaders, however Caffe was surpassed by Theano, CNTK and Dl4j. And lastly, Figure 5.3b shows number of stars of a framework's github repository and number of contributors that work on those projects.

We can see that Tensorflow, Keras, PyTorch and Caffe are the most popular frameworks, let's briefly discuss each one them.

5.1.1 Caffe

The Caffe library was originally developed at UC Berkeley; it was written in C++ with a Python interface. An important distinctive feature of Caffe is that one can train and deploy models without writing any code. To define a model, you just edit configuration files or use pre-trained models from the Caffe Model Zoo, where you can find most established state-of-the-art architectures. Then, to train a model you just run a simple script [42]. Due to its older architecture Caffe is now being surpassed by newer frameworks like TensorFlow or PyTorch.

5.1.2 TensorFlow

TensorFlow is being developed and maintained by Google. It is written in C++/Python and provides Python, Java, Go and JavaScript API. TensorFlow uses *static computational graphs*, this means that we define all needed operations in advance and after that computational graph is being constructed and compiled and then executed. Unfortunately, if you want to improve the networks architecture with conditionals or loops you cannot simply use python keywords, to add nodes to the graph you should use special control flow operations provided by the TensorFlow API.

Apart from purely computational features, TensorFlow provides an extension called TensorBoard that can visualize the computational graph, plot quantitative metrics about the execution of model training or inference, and basically provide all sorts of information necessary to debug and fine-tune a deep neural network in an easier way [42].

5.1.3 Keras

Keras is a high-level neural network library written in Python by Francois Chollet, currently a member of the Google Brain team. It works as a wrapper over one of the low-level libraries such as TensorFlow, Microsoft Cognitive Toolkit, Theano or MXNet. However, Keras is being developed with an eye towards fast prototyping. It is not flexible enough for complicated models, and sometimes error messages are not easy to debug [42].

5.1.4 PyTorch

PyTorch was released by Facebook's artificial-intelligence research group for Python, based on Torch (previous Facebooks framework for Lua). It is the main representative of *dynamic graph* in contrary to TensorFlow that, as mentioned before, uses static graph. Thanks to dynamic graph, PyTorch is integrated in Python more than TensorFlow. So you can write conditionals and loops as in ordinary python program. It is also said to be a bit faster than TensorFlow. It has extensive documentation with a lot of official tutorials and examples, however, the community is still quite smaller as opposed to TensorFlow [42].

5.2 Suitability for on-board deployment

Modern UAVs use a so called system-on-a-chip (SoC) as an on-board computer. SoC is a relatively small integrated circuit that includes components of a computer, such as a CPU,

memory, I/O ports and storage.

There are several SoCs produced by NVIDIA, namely: Tegra K1, Tegra X1, Tegra X2 and Xavier. Also, there is a big family of SoCs called Intel NUC, that are produced by Intel. NVIDIA chips have a multi-core processor based on ARM architecture and also a specifically designed GPU. On the other hand, Intel NUCs have Intel's mobile processors with x86 architecture. These are the same processors that are used in regular laptops (e.g. Intel Core i7).

In order to detect objects with neural networks in real time we need a GPU to accelerate computations. Moreover, the majority of deep learning frameworks use NVIDIA's CUDA library for the GPU acceleration, which means that we have to use a GPU produced by NVIDIA.

It is possible to compile TensorFlow, Keras, PyTorch and Caffe to use on ARM systems. Thus we can consider to use NIVIDA's SoCs, especially since they have built-in graphics processing unit. However, those GPUs are not as powerful as desktop GPUs. For example, let's compare Xavier's and Tegra's X2 GPUs with NVIDIA GeForce GTX 1060 (the one that we used). Xavier's GPU is capable of 1300 GFLOPS (giga floating point operations per second) on single precision floating point numbers. Tegra X2 can perform 437-750 GFLOPS and NVIDIA GeForce GTX 1060 can perform 4372 GFLOPS. We already see that our GPU is 3 times faster that Xavier and approximately 5 to 10 times faster than Tegra X2.

Another option can be to use one of the Intel NUC systems. It shouldn't be an issue to install one of these frameworks on this system, since it has a regular x86 architecture. Then we can connect an external GPU to the Intel NUC via Thunderbolt port. However, this requires to install additional hardware on a drone, which increases its weight.

5.3 Summary

In this chapter we made an overview of existing deep learning frameworks and we concluded that TensorFlow, Keras, PyTorch and Caffe are the most popular frameworks. In the table 5.1 we provide a short summary of those tools.

We also studied suitability of the deep learning frameworks for on-board deployment. We concluded that TensorFlow, Keras, PyTorch and Caffe can be installed on systems that are used on UAVs and that the problem of running a neural network on a drone is not in the framework, but rather in computational power of on-board hardware.

For this project we chose TensorFlow framework since it can be installed on an on-board computer and due to its popularity, extensive documentation, big community and TensorBoard - off-the-shelf visualisation tool.

	Developers	Platforms	Written in	Interfaces	GPU support
Theano	Université de Montréal	Cross-platform	Python	Python (Keras)	yes
Tensorflow	Google Brainteam	Linux, macOS, Windows, Android	C++, Python, CUDA	Python (Keras), C/C++, Java, Go, JavaScript, R, Julia, Swift	yes
Keras	François Chollet	Linux, macOS, Windows	Python (requires Theano or tensorflow as backend)	Python, R	yes
Caffe	Berkeley Vision and Learning Center	Linux, macOS, Windows	C++	Python, MATLAB, C++	yes
PyTorch	Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Facebook's artificial-intelligence research group	Linux, macOS, Windows	Python, C, CUDA	Python	yes

Table 5.1: Comparison of deep learning frameworks.

6 Dataset and its analysis

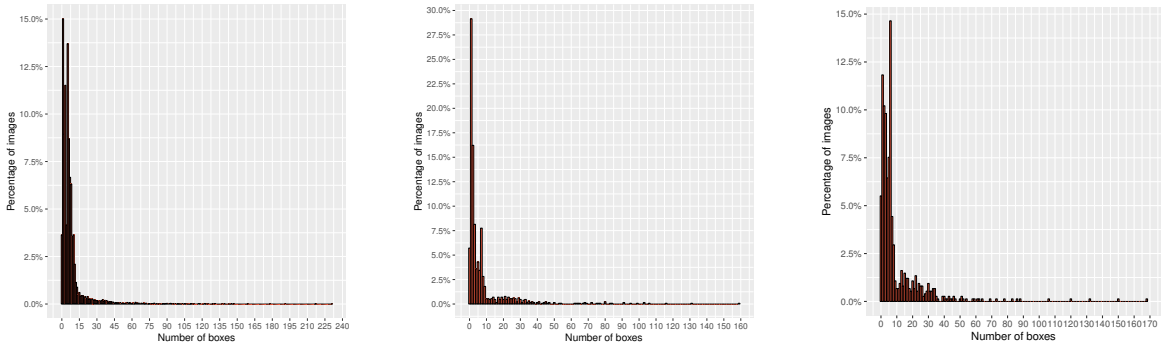
In order to train neural network to detect people from aerial footage, we need a dataset where people are depicted from the corresponding point of view and dataset should also provide bounding boxes for every person in an image.

We were able to find several publicly available datasets that provide footage from drones and from CCTV cameras, they are: Okutama-Action dataset[32], Mini-drone video dataset[18], UCF Aerial Action Dataset[8], VIRAT Video Dataset[12] and Vision Meets Drones dataset (VisDrone) [39].

Except for VisDrone, these datasets are mainly designed for action recognition task, but they still provide bounding boxes for people that are present in an image. In total, these datasets have 139 video files or approximately 450k frames and VisDrone additionally has 6470 images. We cannot use every single frame from videos to train network, since subsequent frames are very alike, which most certainly will cause the network to overfit. In order to reduce overfitting we sampled each 10th frame from videos and since VIRAT dataset has still background we sampled each 100th frame. These datasets have several other problems, namely in VIRAT and in Mini-Drone datasets some people that are present in an image are not labelled and some videos in VIRAT dataset have only labels for actions, but do not contain bounding boxes for objects. Another problem is that some bounding boxes are defined very badly - in some cases center of a bounding box was far away from the center of the object it describes. In order to filter out images that contain these types of problems we had to manually review the entire data set and remove those images. After we filtered out images with bad bounding boxes we split gathered data set in training, testing and validation subsets. **Since images are gathered from videos, during splitting we made sure that most of images that go to different subsets should be from different video sequences.** This will help us to see whether the model overfits or not. Here is the summary of gathered dataset:

	Training set	Testing set	Validation set	Total
Number of images	8693	1276	744	10713
Number of bounding boxes	74122	10006	7599	91727
Average number of boxes per image	8.53	7.84	10.21	-
Max number of boxes per image	231	159	168	-

Let's further investigate our gathered dataset. Firstly let's see distribution of number of boxes:

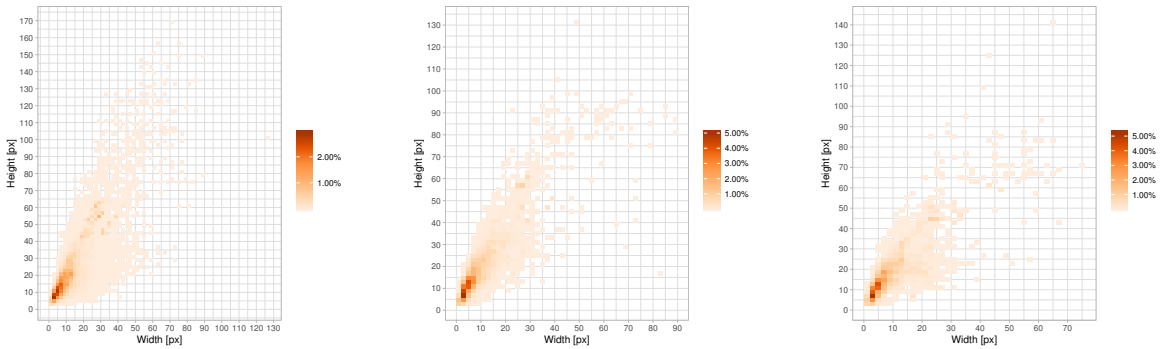


(a) Training set (b) Testing set (c) Validation set

Figure 6.1: Distribution of number of boxes per image

From the graphs above we can see that in all sets the majority of images have roughly from 0 to 15 objects and that distributions between subsets are approximately the same.

Another property of the dataset that may be interesting is the distribution of boxes by width and height. For this test we resized bounding boxes such that they correspond to images with 720×405 size with preserved aspect ratio. This means that if initial image had $4 : 3$ aspect ratio it will preserve it after we will do the resize, but will fit to the given size, which in our case has aspect ratio of $16 : 9$. In this way we will not distort bounding boxes and at the same time we will be able to compare them. From the figures below we can see that the most frequent sizes of the boxes are up to 30 pixels in height and up to 20 pixels in width.

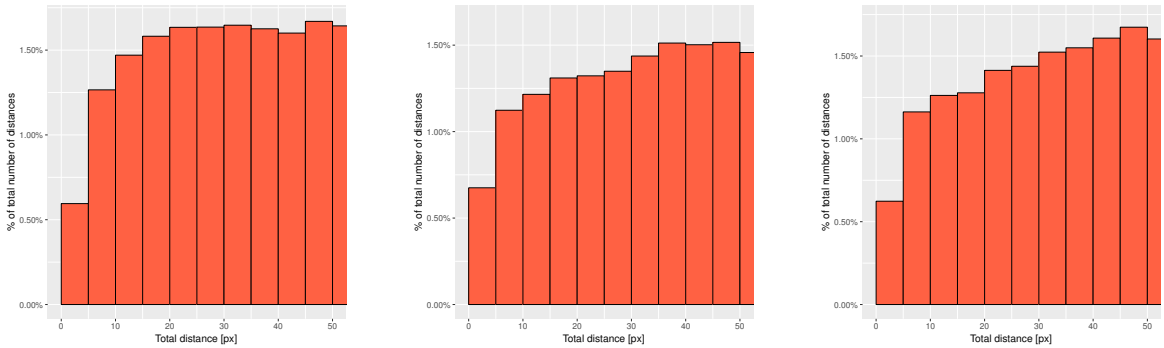


(a) Training set (b) Testing set (c) Validation set

Figure 6.2: Distribution of boxes by width and height

Since later we will use the model that is based on YOLO presented in chapter 4 it may be useful to know what is the distribution of distances between objects in our dataset. Basically, we will divide an image into a grid and each grid cell will be able to predict only one object. This kind of analysis may help us to determine suitable size of the grid. For this analysis, same as in previous point, we firstly resize images to 720×405. After that we compute euclidean distances between objects in the same image, we only compute distances once, meaning that if we calculate distance from object A to object B we will not compute

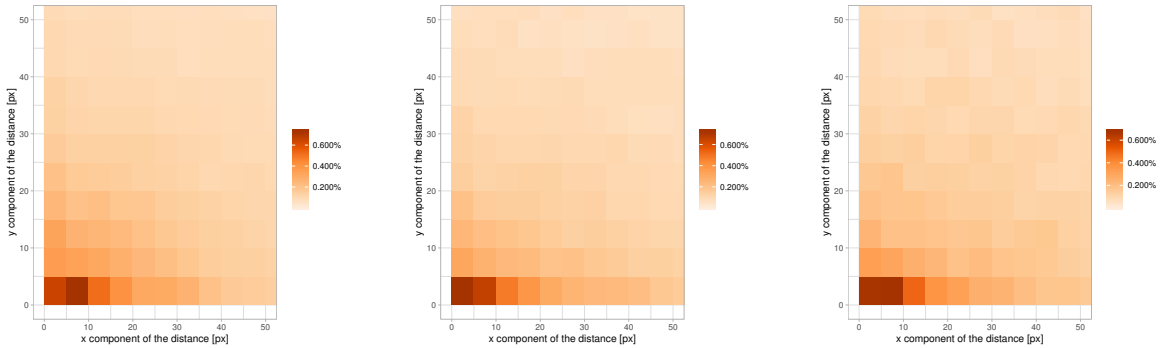
distance from B to A. In figure 6.3 we show distributions of total distances. For better view we limited x axis to 50 pixels.



(a) Training set (b) Testing set (c) Validation set

Figure 6.3: Total distances between bounding boxes

We can also look at distribution of distances by its' x and y components - this may be useful when we decide how tall and wide should grid cells be.



(a) Training set (b) Testing set (c) Validation set

Figure 6.4: Distances between bounding boxes by x and y components

In this chapter we presented the dataset that we gathered and did a simple analysis. From that analysis we can see that on average images in the dataset contain 8 to 10 human objects. Also, the majority of the objects are small relatively to image size and some objects are very close to each other.

7 Implementation and results

For the purpose of the human detection from drones we've chosen neural network based on You Only Look Once (YOLO) model described in this paper [23] and TensorFlow deep learning framework to implement it. We chose YOLO model due to its simple architecture and good precision. Our initial idea was firstly to implement YOLO v1 and after that YOLO v2 [28], but we were only able to implement model that is based on YOLO v1. You can find our code at this GitHub repository: https://github.com/ABlack-git/yolo_object_detection.

For this work we use computer with NVIDIA GeForce GTX 1060 GPU with 6GB of VRAM, Intel Core i9-7920, 2.0 Ghz processor and 32 GB of RAM to train and test proposed networks.

Our models has differences compared to YOLO v1, so let's further discuss it in more details.

7.1 General description of the model

Same as in YOLO paper we virtually divide an image into smaller rectangle regions that will be responsible for predicting humans. Since we try to detect objects that are small relatively to the image size and can be very close to each other, we will need substantially more regions than in the original paper. We can see this in Figure 7.1 below.

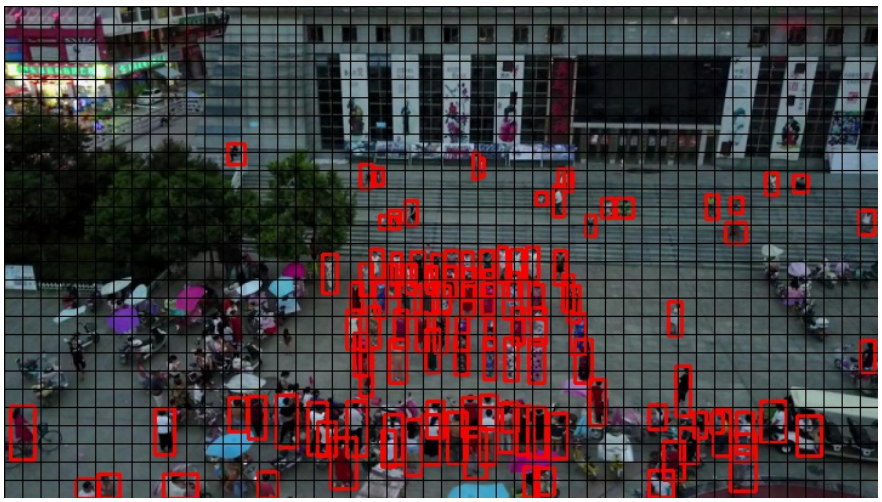


Figure 7.1: Image from VisDrone dataset with 48 by 27 grid

From Figure 6.3 we can see how close objects may be to each other in our dataset (when images are resized to 720×405). This means that if, for example, diagonal of a rectangular region will be smaller than 20 px, we won't be able to predict certain amount of objects, since there is a possibility that two or more objects will fall into the same grid cell. But on the other hand, we can't make those regions small enough because this will increase complexity of the model. Also, in the paper the model is capable of predicting several bounding boxes per grid cell, this is done in order for the model to be robust to predict boxes with different aspect ratios (e.g. the first box in a grid cell can specialise on a vertical boxes, while the second one can specialise on a horizontal boxes). Since we will use a high number of grid cells, our model predicts only one box per grid cell in order to reduce the number of learnable parameters, because large number of parameters may lead to overfitting. Also, we only detect humans so we don't need to make classification, we will just assume that if our network outputs a bounding box it will represent a human. However, we still predict coordinates of the center of a bounding box relative to the grid cell, width and height relative to image size and we also predict confidence of a bounding box as in the original paper.

7.1.1 Loss function

Due to the changes that we've maid we also need to adjust the loss function. This is the loss function that we optimise:

$$\begin{aligned} L = \lambda_1 \sum_{i=0}^{N} \mathbb{1}_i^{obj} [(x_i - \hat{x}_i)^2 + ((y_i - \hat{y}_i)^2] + \lambda_2 \sum_{i=0}^{N} \mathbb{1}_i^{obj} [(w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2] + \\ + \lambda_3 \sum_{i=0}^{N} \mathbb{1}_i^{obj} (C_i - \hat{C}_i)^2 + \lambda_4 \sum_{i=0}^{N} \mathbb{1}_i^{noobj} (C_i - 0)^2 \end{aligned} \tag{7.1}$$

where x, y are coordinates of the center of a bounding box relative to the grid cell, w, h are width and height of a predicted bounding box relative to the image, C is confidence, that is a parameter that tells how good a predicted bounding box is. Variables with hat represent ground truth labels and $\hat{C}$ we calculate as intersection over union of predicted box with ground truth box. $\mathbb{1}$ is a function that can be either 1 or 0:

$$\mathbb{1}_i^{obj} = \begin{cases} 1 & \text{if i-th cell should predict an object} \\ 0 & \text{otherwise} \end{cases}$$

From this we can see that we will adjust coordinates and confidence only of those boxes that should predict objects. Also we want to decrease confidence of those boxes that do not predict objects, thus function $\mathbb{1}_i^{noobj}$ should be 1 when there is no object to predict by i-th cell.

Parameters λ_k are constants that we picked to be $\lambda_1 = 5$, $\lambda_2 = 5$, $\lambda_3 = 1$ and $\lambda_4 = 0.1$. With this we emphasise the loss from coordinates and we decrease the loss from confidence of those boxes that do not predict objects, otherwise this loss will overwhelm other parts of the loss function. We picked these values after iteratively trying out different numbers. We initially started with all parameters being less than one and greater than zero, however with such parameters training loss wasn't decreasing at all. We also tried to keep contribution of different parts of the loss function to be more or less the same.

7.1.2 Output of the network

Output of our network is an array of floating point numbers that can be understood in the following way. Each grid cell can predict only one bounding box and each box has 5 values (x,y,w,h,C), thus every five elements in the output array will correspond to one grid cell. For example, if we have 5×5 grid, the first set of five numbers will correspond to the first cell in the first row, the second set will correspond to the second cell in the 1st row, than the 6th set of five numbers will correspond to the first cell in the second row and so on. Our implementation provides function *getPredictions()* which will return list of predicted bounding boxes. In case of 5×5 grid the network will output 25 predictions and, if an input image contains one person, the majority of them will have low confidence and we can simply discard them by setting confidence threshold. However, the network can output several boxes with high confidence that overlap for one object. We can use the non-maximum suppression (NMS) algorithm in order to filter out those boxes. NMS and confidence thresholding are part of the *getPredictions()* function. Let us show how NMS works.

```
1  Function nonMaxSuppression(boxes, confidenceThreshold, iouThreshold):
       boxes               : boxes produced by the network
       confidenceThreshold: confidence threshold
       iouThreshold        : IoU threshold
2      Delete boxes whose confidence < confidenceThreshold ;
3      sorted ← sort boxes by their confidence ;
4      output ← empty list ;
5      while sorted is not empty do
6          # pick the box with highest confidence and remove from sorted
7          currentBox ← sorted.pop() ;
8          output.add(currentBox) ;
9          # compute IOU between current box and boxes that remain in sorted
10         IoU ← computeIOU(currentBox, sorted) ;
11         Delete boxes from sorted whose IoU with currentBox > iouThreshold ;
12     end
13 return output
```

Algorithm 3: Non-maximum suppression algorithm

Basically, this means that we will firstly remove bounding boxes with low confidence, than iteratively we will pick the box with highest confidence and remove those boxes which greatly overlap with it.

7.2 Results

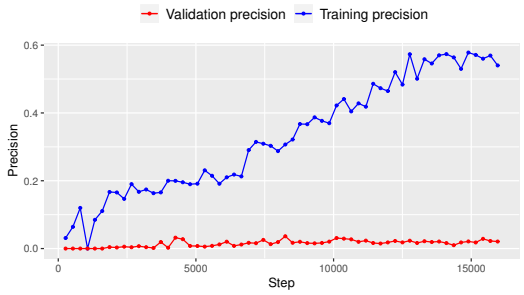
We trained two models, one that consist of 7 layers and the other of 8 layers. By our versioning system we refer to them as 6l-v2-1 and 8l-v2-1, respectively. Their architecture is described in Table A.1 and Table A.2. In both models we employ batch normalisation layers [21], leaky rectified linear units as activation function and weight decay regularisation. In the 6l-v2-1 model we use stride of 2 in the first two convolutional layers, with this we

achieve faster sub-sampling of the input. In the 8l-v2-1 model we use stride of 1 in all convolutional layers, thus we do sub-sampling only with max pooling layers. We also added another convolutional layer to this network, compared to the 7 layer model. We use 48×24 grid size with 720×480 px input size in the model with 7 layers, this will give us 15×20 px detection regions. We decided to use 720×405 px input size in the network with 8 layers, since the majority of images in the training set have aspect ratio of $16 : 9$, we also use grid size of 48×27, which will give us 15×15 px detection regions.

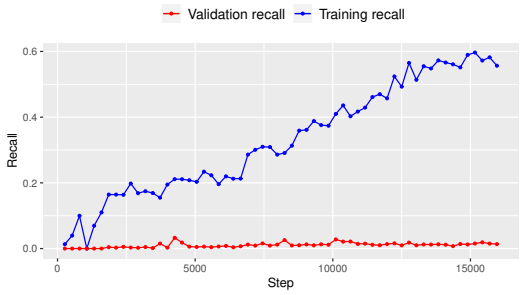
We train both networks with momentum optimiser with momentum of $\gamma = 0.9$ and weight decay of $\lambda = 0.0005$. Due to high loss in the beginning of the training, we start to train with the learning rate of $\alpha = 0.00001$ and then rise it to $\alpha = 0.0001$ and for model with 8 layers we rise it to $\alpha = 0.001$.

During training we record average precision and recall measured both on training and validation datasets. By average precision and recall we mean that at first we compute those metrics for every single image and then the average of all measurements. We do precision-recall tests only on the subset of 2000 images of the training set. Another important detail is how we deal with edge cases. If our network outputs zero bounding boxes and an input image also have zero ground truth boxes we say that precision=1 and recall=1. When our network outputs zero boxes and an input image has more than zero associated ground truth boxes we say that precision=0 and recall=0. Also, in the case when the network outputs more than zero boxes and an input image doesn't have ground truth boxes precision and recall equal zero.

We present results of those measurements in figures 7.2 and 7.3. From those figures we can see that our networks greatly overfit, since precision and recall on validation dataset stays low during entire training, while on the training set we can see that those metrics improve over time.

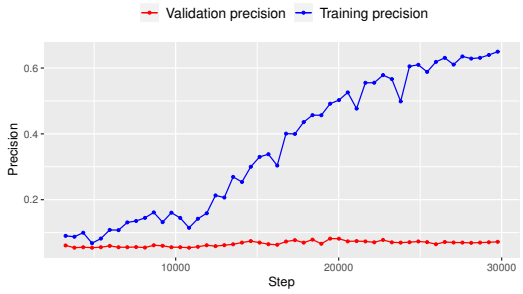


(a) Precision on validation and train sets

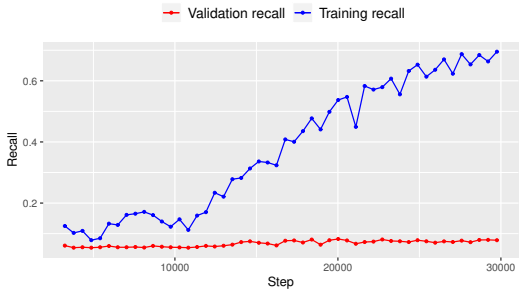


(b) Recall on validation and train sets

Figure 7.2: Precision and recall of the 6l-v2-1 model during training



(a) Precision on validation and training sets



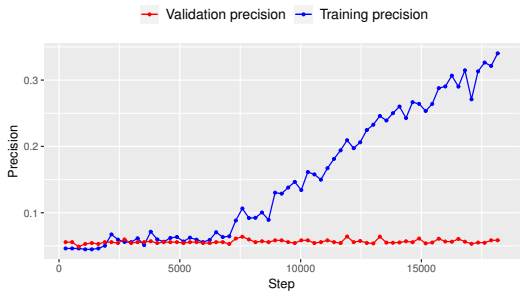
(b) Recall on validation and training sets

Figure 7.3: Precision and recall of the 8l-v2-1 model during training

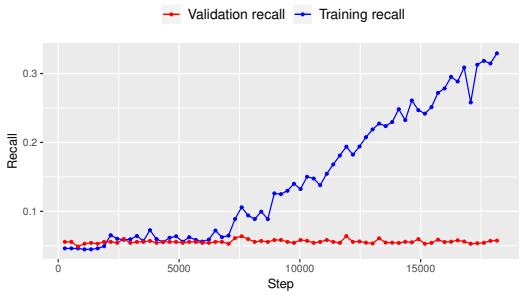
There are several reasons that may cause a network to overfit. Firstly, our training dataset is not very big, it has only 8600 images and, more over, it is not diverse, since images in the dataset were sampled from videos. Secondly, if we will look at Table A.1 and Table A.2, we can see that both those models have great number of learnable parameters: model with 7 layers has 21 million parameters and model with 8 layers has 61 million. The last fully connected layer of the model with 7 layers has 17 millions of parameters which is roughly 80 % of the total number, and the last layer of the 8 layer model has 39 millions of parameters, which is approximately 65 % of the total amount of learnable weights. Last fully connected layer has such a big number of parameters due to the size of the grid.

Big amount of weights in the combination with a small dataset can be a valid reason for network to overfit. In order to try to fight overfitting, we, firstly, train the model that also consists from 8 layers (we refer to it as 7l-v1-1), its architecture is described in Table A.3. However, we added max pool layer between last convolutional and fully connected layers and we removed the first batch normalisation layer. In total this model has close to 6 millions parameters and last fully connected layer has 2.5 millions of parameters which is close to 43 % of the total number of parameters.

We start to train this model with learning rate of $\alpha = 0.00001$ and then increase it to $\alpha = 0.0001$. We also use batch size of 32. We let it train for approximately 20 epochs, but, as shown in Figure 7.4, this model also overfits.



(a) Precision on validation and train sets

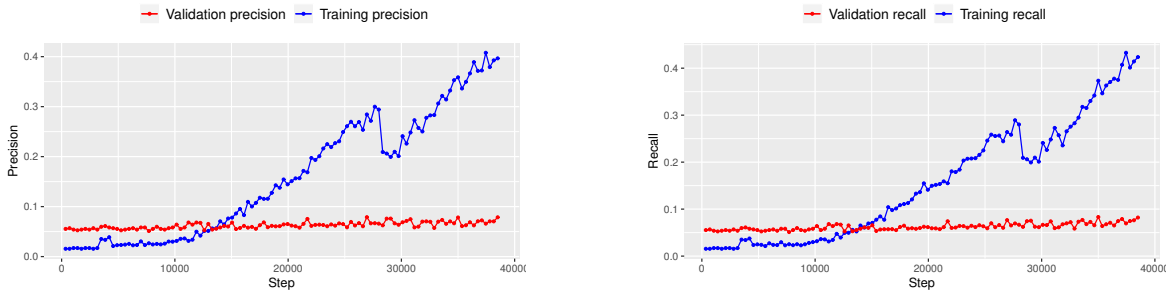


(b) Recall on validation and train sets

Figure 7.4: Precision and recall of the 7l-v1-1 model during training

Next attempt to fight overfitting is to artificially augment training set. We do this with

horizontal flipping, random translation, random scale and random changes in HSV colour space. We train model with the same architecture as described in Table A.3 on the augmented dataset. To differentiate between the previous model and the one that was trained on the augmented dataset, we call this one as 7l-v1-3. We use batch size of 64 and the same learning rate scheduling as in the previous part. We train it for about 100 epochs but even after adding artificially augmented data this model also overfits, as shown in Figure 7.5.



(a) Precision on validation and train sets

(b) Recall on validation and train sets

Figure 7.5: Precision and recall of the 7l-v1-3 model

In figures 7.4 and 7.5 we can see that in the beginning of the training precision and recall are higher on the validation dataset than on the training dataset. The reason behind this is how we defined precision and recall for the situation when there are no objects to detect in an input image and the network output zero boxes as well. Also in the training set there are approximately 3 % of images without objects, while in the validation dataset there are slightly more than 5 % of such images, which we can see in the figure 6.1. The reason that this cannot be seen in figures 7.2 and 7.3 is that at the time when we recorded those measurements corresponding models already were able to predict sufficient amount of true positives on the training set.

In the table 7.1 we show average precision (AP) and average recall (AR) for all our models on a subset of 1500 images of the training set and on the entire testing set. As we described in the section 1.1.3 we consider a prediction to be true positive if its IoU with ground truth box is equal or greater than 0.5. We also checked whether our networks detects objects on the testing set, but output boxes that are slightly off in terms of width and height. For this we recorded statistics with lower thresholds of IoU. Even though we can see that AP and AR do improve, but the improvement is insignificant. Also note, that we can not compare these models since they all have been trained for different number of epochs.

	IOU=0.5		IOU=0.3		IOU=0.2	
	AP	AR	AP	AR	AP	AR
model 7l-v1-1 on the test set	0.056	0.059	0.059	0.067	0.065	0.075
model 7l-v1-1 on the training set	0.331	0.314	0.439	0.402	0.476	0.437
model 7l-v1-3 on the test set	0.051	0.057	0.068	0.08	0.082	0.098
model 7l-v1-3 on the training set	0.463	0.539	0.513	0.589	0.546	0.623
model 6l-v2-1 on the test set	0.059	0.063	0.082	0.095	0.089	0.102
model 6l-v2-1 on the training set	0.411	0.420	0.526	0.515	0.549	0.525
model 8l-v2-1 on the test set	0.077	0.08	0.084	0.098	0.094	0.103
model 8l-v2-1 on the training set	0.599	0.612	0.701	0.694	0.731	0.733

Table 7.1: Statistics of models on training and testing datasets

In table 7.2 we show average time of prediction of our models and corresponding frames per second (FPS) rate. For this we don't count time that is needed to read image from disk and to resize it because those times greatly depend on the size of an image. Read time on our system may vary from 10 ms to 100 ms and resize time from 5 ms to 80 ms.

Model version	Average prediction time [ms]	FPS
7l-v1-1	21	48
7l-v1-3	20	50
6l-v2-1	22	45
8l-v2-1	28	35

Table 7.2: Prediction time of trained models

We spent approximately two days to train 6l-v2-1 and 8l-v2-1 models, less than 2 days to train 7l-v1-1 and about 3 days to train 7l-v1-3 model.

At this point it is unclear to us why models keep overfitting. The most reasonable explanation is that the dataset is quite small comparing to complexity of the task and also that the dataset is not versatile enough. But due to time constrains we are unable to perform further experiments.

8 Summary

In this work we, firstly, ma an overview of existing human detection methods in the section 1.1. There we showed object detection pipeline, most common algorithms for feature extraction and classification. We also presented metrics such as precision, recall and intersection over union that are used in the object detection field for evaluation.

After that, in chapter 2, we studied fully connected artificial neural networks. We showed what is a neuron in the ANN and how multilayer networks are organised. Then we explained how to train a neural network with backpropagation algorithm.

Since fully connected neural networks are not suitable for image processing, in chapter 3 we studied convolutional neural networks. We, firstly, explained what is convolution operation and how it is used in image processing. After that we showed simple building blocks of the CNN and, lastly, in the section 3.4 we explained why it is preferable to use convolutional neural networks for image processing.

In chapter 4 we made an overview of existing models of convolutional networks for object detection. We briefly studied models from R-CNN family, such as, R-CNN, Fast R-CNN and Faster R-CNN. After that we studied Single Shot Models, namely, You Only Look Once, Single Shot Detector and You Only Look Once v2. We saw that models from R-CNN family have better mean average precision but they also have worse frame rate than the single shot models.

In order to implement our own neural network we had to choose a suitable computer framework. For this reason, in chapter 5, we made an overview of existing deep learning computer frameworks. We looked at the popularity of different frameworks in different areas and we concluded that Caffe, TensorFlow, Keras and PyTorch are among the most popular frameworks. We also gave a brief description of those frameworks and we've chosen TensorFlow for our project.

In chapter 6 we presented our dataset, which was gathered from publicly available datasets, namely, Okutama-action, Mini-drone, UFC Aerial Action, VIRAT and Vision Meets Drones.

In chapter 7 we presented a general description of our model, showed the loss function and encoding of the output layer. We also described the non-maximum suppression algorithm that is used to filter out boxes with low confidence. After that we presented results of our work.

The main goal of this work was to propose a neural network model that will be able to detect people in images that were shot from an aerial vehicle. We used a model that is based on You Only Look Once neural network described in [23]. We trained several instances of the proposed model and, unfortunately, they all show signs of the strong overfitting and are

not able to generalise. Overfitting is not good on its own, of course, but on the other hand it at least shows that the model is functional. The reason behind overfitting may be a big number of parameters in the last fully connected layer and that the dataset that we were able to gather is not big and versatile enough.

During work on this project we faced several difficulties that are related to aerial object detection. Firstly, it is a lack of a big and versatile dataset in that field. Most of the datasets that we used are designed primary for action recognition. On the other hand, in the area of the ground-based object detection there are several popular datasets, for example, Microsoft Common Objects in Context (COCO) [16], The PASCAL Visual Object Classes (VOC) [9] and Open Images Dataset [37]. COCO has more than 200k labeled images and 80 different categories, while VOC has 11.5k images and 20 classes. They both provide not only bounding boxes, but also segmentation maps. Open Images has roughly 2 million images with annotated bounding boxes and 600 classes. Another problem is that objects that are shot from high altitude naturally occupy small area in the image, comparing to objects in ground-based images. This pushed us to chose a bigger grid size, which in turn resulted in a big number of weights in last fully connected layer. Smaller objects will also be less represented in feature maps after subsequent pooling.

There are several ways to improve this work. First of all, we need to extend our dataset. Since manual annotation of images is very time-consuming it may be preferable to somehow automatically generate annotation from photorealistic computer games such as GTA 5, for instance. In [29] authors propose a method to semi-automatically extract segmentation maps for images from computer games. They state than on average it takes 7 seconds to annotate an image with their approach. Another way to improve this work is to use a bit different neural network models. For example, it may be better to use YOLO v2 [28]. It is a fully convolutional neural network, which means that there are no fully connected layers in that model. This may reduce number of weights, which we get from the last layer and also this will speed up the network. Another solution may be to use YOLO v3 [38] which was among all also designed to improve detection of small objects.

A Appendix

Layer	Output size	Parameters
Input	Channels: 3 Size: 720×480 Total: 1036800	Grid size: 48×24
Conv 1	Channels: 16 Size: 360×240 Total: 1382400	Kernel size: 4×4 Stride: 2 Number of weights: 768
Batch norm	Same	Number of weights: 2764800
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 16 Size: 180×120 Total: 345600	kernel: 2x2 Stride: 2
Conv 2	Channels: 32 Size: 90×60 Total: 172800	Kernel size: 4×4 Stride: 2 Number of weights: 8192
Batch norm	Same	Number of weights: 345600
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 32 Size: 45×30 Total: 43200	kernel: 2x2 Stride: 2
Conv 3	Channels: 64 Size: 45×30 Total: 86400	Kernel size: 3×3 Stride: 1 Number of weights: 18432
Batch norm	Same	Number of weights: 172800
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 64 Size: 23×15 Total: 22080	kernel: 2x2 Stride: 2
Conv 4	Channels: 128 Size: 23×15 Total: 44160	Kernel size: 3×3 Stride: 1 Number of weights: 73728
Batch Norm	Same	Number of weights: 88320
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 128 Size: 12×8 Total: 12288	kernel: 2x2 Stride: 2
Conv 5	Channels: 256 Size: 12×8 Total: 24576	Kernel size: 3×3 Stride: 1 Number of weights: 294912
Batch norm	Same	Number of weights: 49152
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 256 Size: 6×4 Total: 6144	kernel: 2x2 Stride: 2
Conv 6	Channels: 128 Size: 6×4 Total: 3072	Kernel size: 3×3 Stride: 1 Number of weights: 294912
Batch norm	Same	Number of weights: 6144
Leaky ReLU	Same	alpha: 0.01
FC	5760	Number of weights: 17700480

Table A.1: Architecture of the 6l-v2-1 model

Layer	Output size	Parameters
Input	Channels: 3 Size: 720×405 Total: 874800	Grid size: 48×27
Conv 1	Channels: 16 Size: 720×405 Total: 4665600	Kernel size: 4×4 Stride: 1 Number of weights: 768
Batch norm	Same	Number of weights: 9331200
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 16 Size: 360×203 Total: 1169280	kernel: 2×2 Stride: 2
Conv 2	Channels: 32 Size: 360×203 Total: 2338560	Kernel size: 4×4 Stride: 1 Number of weights: 8192
Batch norm	Same	Number of weights: 4677120
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 32 Size: 180×102 Total: 587520	kernel: 2x2 Stride: 2
Conv 3	Channels: 64 Size: 180×102 Total: 1175040	Kernel size: 3×3 Stride: 1 Number of weights: 18432
Batch norm	Same	Number of weights: 172800
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 64 Size: 90×51 Total: 293760	kernel: 2×2 Stride: 2
Conv 4	Channels: 128 Size: 90×51 Total: 587520	Kernel size: 3×3 Stride: 1 Number of weights: 73728
Batch Norm	Same	Number of weights: 1175040
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 128 Size: 45×26 Total: 149760	kernel: 2×2 Stride: 2
Conv 5	Channels: 256 Size: 45×26 Total: 299520	Kernel size: 3×3 Stride: 1 Number of weights: 294912
Batch norm	Same	Number of weights: 306176
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 256 Size: 23×13 Total: 76544	kernel: 2×2 Stride: 2
Conv 6	Channels: 512 Size: 23×13 Total: 153088	Kernel size: 3×3 Stride: 1 Number of weights: 1179648
Batch norm	Same	Number of weights: 306176
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 512 Size: 12×7 Total: 43008	kernel: 2×2 Stride: 2
Conv 7	Channels: 256 Size: 12×7 Total: 21504	Kernel size: 3×3 Stride: 1 Number of weights: 1179648
Batch norm	Same	Number of weights: 43008
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 256 Size: 6×4 Total: 6144	kernel: 2×2 Stride: 2
FC	6480	Number of weights: 39813120

Table A.2: Architecture of the 8l-v2-1 model

Layer	Output size	Parameters
Input	Channels: 3 Size: 720×420 Total: 874800	Grid size: 48×21
Conv 1	Channels: 16 Size: 7360×210 Total: 1209600	Kernel size: 3×3 Stride: 2 Number of weights: 432
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 16 Size: 180×105 Total: 302400	kernel: 2×2 Stride: 2
Conv 2	Channels: 32 Size: 90×53 Total: 152640	Kernel size: 3×3 Stride: 2 Number of weights: 4608
Batch norm	Same	Number of weights: 305280
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 32 Size: 45×27 Total: 38880	kernel: 2x2 Stride: 2
Conv 3	Channels: 64 Size: 45×27 Total: 77760	Kernel size: 3×3 Stride: 1 Number of weights: 18432
Batch norm	Same	Number of weights: 155520
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 64 Size: 23×14 Total: 20608	kernel: 2×2 Stride: 2
Conv 4	Channels: 128 Size: 23×14 Total: 41216	Kernel size: 3×3 Stride: 1 Number of weights: 73728
Batch Norm	Same	Number of weights: 82432
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 128 Size: 12×7 Total: 10752	kernel: 2×2 Stride: 2
Conv 5	Channels: 256 Size: 12×7 Total: 21504	Kernel size: 3×3 Stride: 1 Number of weights: 294912
Batch norm	Same	Number of weights: 43008
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 256 Size: 6×4 Total: 6144	kernel: 2×2 Stride: 2
Conv 6	Channels: 512 Size: 6×4 Total: 12288	Kernel size: 3×3 Stride: 1 Number of weights: 1179648
Batch norm	Same	Number of weights: 24576
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 512 Size: 3×2 Total: 3072	kernel: 2×2 Stride: 2
Conv 7	Channels: 256 Size: 3×2 Total: 1536	Kernel size: 3×3 Stride: 1 Number of weights: 1179648
Batch norm	Same	Number of weights: 3072
Leaky ReLU	Same	alpha: 0.01
MaxPool	Channel: 256 Size: 2×1 Total: 512	kernel: 2×2 Stride: 2
FC	5040	Number of weights: 2580480

Table A.3: Architecture of the 7l-v1-1 and 7l-v2-3 models

Bibliography

[1] Timo Ojala, Matti Pietikäinen, and David Harwood. "A comparative study of texture measures with classification based on featured distributions". In: *Pattern Recognition* 29.1 (1996), pp. 51–59. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(95)00067-4. URL: http://www.sciencedirect.com/science/article/pii/0031320395000674.

[2] C. P. Papageorgiou, M. Oren, and T. Poggio. "A general framework for object detection". In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. Jan. 1998, pp. 555–562. DOI: 10.1109/ICCV.1998.710772.

[3] Simon Haykin. *Neural Networks. A comprehensive foundation*. Hamilton, Ontario, Canada: Pearson Education, 1999.

[4] P. Viola and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. Dec. 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517.

[5] David Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (Nov. 2004), pp. 91–. DOI: 10.1023/B:VISI.0000029664.99615.94.

[6] N. Dalal and B. Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. June 2005, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.

[7] Yadong Mu et al. "Discriminative local binary patterns for human detection in personal album". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. June 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587800.

[8] *UCF Aerial Action Data Set*. 2009. URL: http://crcv.ucf.edu/data/UCF_Aerial_Action.php.

[9] M. Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338.

[10] P. F. Felzenszwalb et al. "Object Detection with Discriminatively Trained Part-Based Models". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.9 (Sept. 2010), pp. 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167.

[11] H. Corporaal M.C.J. Peemen B. Mesman. *Speed sign detection and recognition by convolutional neural networks*. 2011.

[12] S. Oh et al. “A large-scale benchmark dataset for event recognition in surveillance video”. In: *CVPR 2011*. 2011, pp. 3153–3160.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[14] R. Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *ArXiv e-prints* (Nov. 2013). arXiv: 1311.2524 [cs.CV].

[15] J. R. R. Uijlings et al. “Selective Search for Object Recognition”. In: *International Journal of Computer Vision* 104.2 (2013), pp. 154–171. URL: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013.

[16] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *arXiv e-prints*, arXiv:1405.0312 (May 2014), arXiv:1405.0312. arXiv: 1405.0312 [cs.CV].

[17] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv e-prints*, arXiv:1409.1556 (Sept. 2014), arXiv:1409.1556. arXiv: 1409.1556 [cs.CV].

[18] T. Boult et al. “Foreword - De-identification for Privacy Protection in Multimedia 2015”. In: *2015 11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition (FG)*. Vol. 04. 2015, pp. 1–2.

[19] R. Girshick. “Fast R-CNN”. In: *ArXiv e-prints* (Apr. 2015). arXiv: 1504.08083 [cs.CV].

[20] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *arXiv e-prints*, arXiv:1512.03385 (Dec. 2015), arXiv:1512.03385. arXiv: 1512.03385 [cs.CV].

[21] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *arXiv e-prints*, arXiv:1502.03167 (Feb. 2015), arXiv:1502.03167. arXiv: 1502.03167 [cs.LG].

[22] W. Liu et al. “SSD: Single Shot MultiBox Detector”. In: *ArXiv e-prints* (Dec. 2015). arXiv: 1512.02325 [cs.CV].

[23] J. Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *ArXiv e-prints* (June 2015). arXiv: 1506.02640 [cs.CV].

[24] S. Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *ArXiv e-prints* (June 2015). arXiv: 1506.01497 [cs.CV].

[25] Cornell University. *Neural Networks and Machine Learning*. 2015. URL: http://blogs.cornell.edu/info2040/2015/09/08/neural-networks-and-machine-learning/.

[26] Philip O. Ogunbona Duc Thanh Nguyen Wanqing Li. “Human detection from images and videos: A survey”. In: *Pattern Recognition* 51 (2016), pp. 148–175. DOI: https://doi.org/10.1016/j.patcog.2015.08.027. URL: http://www.sciencedirect.com/science/article/pii/S0031320315003179.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[28] J. Redmon and A. Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *ArXiv e-prints* (Dec. 2016). arXiv: 1612.08242 [cs.CV].

[29] Stephan R. Richter et al. “Playing for Data: Ground Truth from Computer Games”. In: *European Conference on Computer Vision (ECCV)*. Ed. by Bastian Leibe et al. Vol. 9906. LNCS. Springer International Publishing, 2016, pp. 102–118.

[30] S. Ruder. “An overview of gradient descent optimization algorithms”. In: *ArXiv e-prints* (Sept. 2016). arXiv: 1609.04747 [cs.LG].

[31] Abhineet Saxena. *Convolutional Neural Networks (CNNs): An Illustrated Explanation*. 2016. URL: https://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/.

[32] M. Barekatain et al. “Okutama-Action: An Aerial View Video Dataset for Concurrent Human Action Detection”. In: *ArXiv e-prints* (2017). arXiv: 1706.03038 [cs.CV].

[33] Tomasz Grel. *Region of interest pooling explained*. 2017. URL: https://blog.deepsense.ai/region-of-interest-pooling-explained/.

[34] Jianxin Wu. *Introduction to Convolutional Neural Networks*. 2017. URL: https://pdfs.semanticscholar.org/450c/a19932fcef1ca6d0442cbf52fec38fb9d1e5.pdf.

[35] Andrew Hsu John McGonagle George Shaikouski. *Backpropagation*. 2018. URL: https://brilliant.org/wiki/backpropagation/.

[36] Ayoosh Kathuria. *Whats new in YOLO v3?* 2018. URL: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

[37] Alina Kuznetsova et al. “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale”. In: *arXiv:1811.00982* (2018).

[38] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv e-prints*, arXiv:1804.02767 (Apr. 2018), arXiv:1804.02767. arXiv: 1804.02767 [cs.CV].

[39] Pengfei Zhu et al. “Vision Meets Drones: A Challenge”. In: *arXiv preprint arXiv:1804.07437* (2018).

[40] Jeff Hale. *Deep Learning Framework Power Scores 2018*. URL: https://www.kaggle.com/discdiver/deep-learning-framework-power-scores-2018/data.

[41] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/.

[42] Sergey Nikolenko Oktai Tatanov. *NeuroNuggets: An Overview of Deep Learning Frameworks*. URL: https://medium.com/neuromation-io-blog/neuronuggets-an-overview-of-deep-learning-frameworks-8e5c164ce012.

[43] Gregory Piatetsky. *Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis*. URL: https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html/2.