Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Dominik Kolonits

PREDICTION OF OBJECT MOVEMENT IN VIDEOS WITH DEEP LEARNING

SUPERVISOR

Dr. Bálint Gyires-Tóth

BUDAPEST, 2018

Table of Contents

Összefoglaló

Napjainkban a mesterséges intelligencia szerepe növekvő tendenciát mutat, egyre több területen van hatással életünkre, az ember és a digitális eszközök összefonódása egyre jellemzőbb modern társadalmunkban. Az elektronikus szerkezetek azonban már nemcsak szórakozásra, hanem a monoton és unalmas tevékenységek kiváltására, a hibalehetőségek csökkentésére, a pontosság, valamint a produktivitás növelésére is alkalmasak.

A gépi látás – mely a mesterséges intelligencia egyik fontos ága - segítségével berendezéseink képesek az azokat körülvevő világ észlelésére, automatizált vizsgálatok, valamint koordinálási feladatok elvégzésére. A gépi látás segítségével a képfeldolgozási feladatok számos esetben sokkal komplexebb módon valósíthatók meg, mint a hagyományos szenzorok segítségével, így lehetővé válik az emberi tényező kizárása, s ezáltal a minőség növelése, azonban ennek ára is van, hiszen jellemzően az ilyen rendszerek felépítése sok esetben bonyolultabb.

Az egyik igen izgalmas és jelentős érdeklődést kiváltó alkalmazási területnek a közlekedés tűnik jelenleg, az önvezető autók esetleges elterjedésében fontos szerepet fog játszani a mesterséges intelligencia is. A közlekedésben az egyik legnagyobb kihívás a jármű körül lévő objektumok felismerése, e célból különféle algoritmusokat fejlesztettek ki, mindnek hátránya azonban, hogy valós idejű alkalmazásuk esetén erőforrásigényük magas.

Munkámban egy olyan mély tanulás (deep learning) alapú megoldást dolgozok ki, mely az erőforrásigény csökkentését tűzi ki célul. A megoldás meglévő objektumfelismerő algoritmuson alapszik, ez fogja a rendszer bemenetét szolgáltatni, s az általam kidolgozott Long Short-Term Memory (LSTM) alapú hálózat a meglévő objektumok típusából, illetve korábbi elhelyezkedésükből próbál meg a jövőbeli elhelyezkedésükről információt adni, ily módon nem válna szükségessé minden képkockára, hanem elég csak bizonyos időközönként újra lefuttatni az objektumfelismerő algoritmusokat. A megvalósítandó hálózat az LSTM hálózatok idősoros jóslásának fajtájához tartozik. Ez annyit jelent, hogy az időben korábban történt eseményekből próbálunk meg a bekövetkező jövőbeli eseményekre valamiféle előrejelzést adni. Az ilyen hálózatoknak több korábbi időpillanatban történt adatot is be lehet adni bemenetnek, így növelve az előrejelzés pontosságát, melyet természetesen

az is erősen befolyásol, hogy mennyi idővel előre szeretnénk információkat kapni a modelltől.

Az ideális modell megtalálása érdekében a legnépszerűbb metrikákat (MABO – Mean Average Best Overlap, mAP – Mean Average Precision) használom. A feladatot nehezíti a sok változó tényező, mint a tanult osztályok száma, a bemeneti adatok száma (mennyi korábbi adat szolgáljon bemenetnek), valamint hogy milyen időtávra predikáljunk előre.

Összességében tehát objektumok mozgásának előrejelzését fogom megvalósítani idősor alapú LSTM hálózattal, melyet MABO és mAP metrikákkal fogok kiértékelni. A modell által előrejelzett eredmények pedig lineáris regresszió által meghatározott eredményekkel kerülnek összevetésre.

Abstract

Nowadays, the role of artificial intelligence is increasing; it has more and more areas affecting our lives, the intertwining of humans and digital devices are becoming more and more common in our modern society. Electronic devices, however, do not serve sole entertainment purposes but are also capable of carrying out monotonous and boring activities, thus reducing defect potential, while increasing accuracy and productivity.

With machine vision - a branch of artificial intelligence -, our devices can detect the world around them, to carry out automated tests and to execute tasks needing coordination. With machine vision, these image processing tasks can be accomplished in a much more complex way than with traditional sensors, so humans can be excluded increasingly to improve the quality, but it also comes with a price, because the design of such systems is more complicated.

Creating solutions for the automotive industry is one of the most exciting areas of application, artificial intelligence will play an important role in the possible spreading of self-driving cars. One of the biggest challenges in the transportation sector is the recognition of objects around the vehicle, and various algorithms have been developed for this purpose. However, they have a disadvantage since their resource requirements are high in real-time applications.

In this work, I am trying to develop a deep learning based solution that can reduce this resource requirement. The solution is based on an existing object recognition algorithm that will provide the input for the system. The „remembering" (Long Short-Term Memory - LSTM) network I have developed will try to provide information about the future location of the objects based on their the exact class and their previous locations, so it would not be necessary to run the object recognition algorithm at all times, it would need to be re-run only after a certain period. The network that is going to be developed belongs to the time series prediction type of the LSTM networks. This means that we are trying to forecast events based on events that happened earlier in time. Such networks can take inputs from multiple earlier moments in time, thus increasing the precision of forecasting, which is naturally heavily influenced by how long in the future we want to receive information from the model.

To find the ideal model, I will use different metrics (MABO - Mean Average Bounding Box Overlap, mAP – Mean Average Precision). The task is made complex by

the many variable factors such as the number of classes to train for, the number of input data (how many previous data should serve as input), and how far to predict in the future.

All in all, I am going to predict the movement of objects with a time series based LSTM network, which I will evaluate with MABO and mAP metrics. The results predicted by the model are compared with the results determined by linear regression.

1 Introduction

Imagination has always played an important role in the life of humanity. The greatest scientific fictions of the XX. century revolved mainly around thinking, feeling robots with artificial intelligence (AI). The scientists knew of course, that artificial intelligence is quite far away from their current knowledge, however very important researches were lead at this time that serve as a foundation for the science of today. Thanks to the work of our predecessors, we have come a long way, yet the real artificial intelligence is still far from us, but there are significant steps forward almost in every field of the application of the AI. Artificial intelligence has many variants; one of those is machine learning.

Nowadays machine learning is getting more and more popular, its usage finds a way in almost every aspect of our lives, we should just think about our music listening habits, e.g. YouTube and Spotify always manage to recommend us new songs, and in the very high percentage of cases, let us be honest, we do like their recommendations. Another example could be the new way of unlocking of our phones, namely just by looking at it. This feature involves deep learning for the face detection via the camera. This use-case, object recognition and object detection, is one of the main area of the research of this field, this is mainly due to its versatile forms of applications. Object recognition is the process of identifying different objects in an image, while object detection goes a step further, it also determines the positions of the recognised objects.

Object recognition and detection can help us automate many tedious tasks, like evaluating invoices for accounting, face detection and thus ensuring security at crowd events, improving industrial processes or even for driving. The latter one is maybe the most progressive field of work today: developing a fully autonomous car is a main goal of many companies and researchers as well. An autonomous car needs to be able to recognise and detect every objects related to it and to the traffic in its surrounding, and it should be able to track them as well. This task is not trivial at all, although there are promising solutions regarding this topic. The state-of-the-art algorithms, that will be introduced in Section 2 have really good performance in recognising and detecting objects even in real-time, but they have one important downside, namely the performance, the computing capacity needed to carry out these complex tasks. Real-time object detection

usually has high computational demands that are generally satisfied by high performance Graphical Processing Units (GPUs). These components usually come with quite a high price tag and with the need of a reliable power supply with huge capacity. These criteria makes it challenging or even impossible to use these algorithms in real-time in various environments, such as in smartphones, or even in cars.

The main goal of this thesis is to develop an algorithm that can track the trajectory of objects in videos with lower computational needs. This can be achieved if not all frames are processed, but some frames are left out and neural networks delivers a predicted location of the objects for the future frames, so the processing of the frames in between becomes unnecessary.

2 Related Deep Learning techniques

2.1 Object detection algorithms

What is object detection exactly? Object detection is object classification and object localization performed jointly. This means that an object detection algorithm has two main tasks, it has to predict, what kind of object is on the image, and where that object is exactly on the image. (Figure 1)

These algorithms usually have at least five outputs per each identified class, the class label of the object, and the four coordinates identifying the so called bounding box, that can be seen in the image below as well. The coordinates of the bounding box can vary in different ways. The algorithm can return the centre of the bounding box, and the length of the sides of the box, another possibility would be to return the coordinates of the top left corner, and the length of the sides, or it can deliver the minimum and maximum coordinates of the sides of the box projected on both axes.

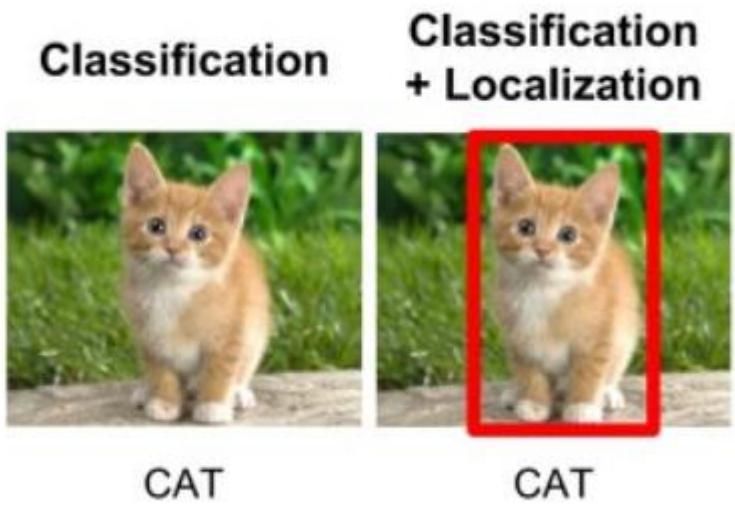

Figure 1. Object detection: object classification and localization together (Source: http://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/)

Object detection algorithms can be divided into two main categories:

- region-based algorithms
- single-shot algorithms

The most known algorithms of these categories are going to be introduced in the coming subsections.

2.1.1 Region-based object detection algorithms

The most trivial approach for object detection would be to use sliding windows of various sizes over the image, and evaluate all windows one-by-one with a CNN. That would create a given classification result for each window, after that we should create an algorithm that chooses the right windows as final results. This method is quite slow however, and that is mainly because of the many different windows we have to evaluate. R-CNN was created to help with this problem.

2.1.1.1 R-CNN

R-CNN stands for Regions with CNN (Convolutional Neural Network) features [1]. This object detection system consists of three modules.

The first generates approx. 2000 category-independent region proposals using a special method, called selective search. Selective search takes into consideration that an object can occur at any scale within the image. Furthermore, some objects have less clear boundaries then other objects. Therefore, all object scales have to be taken into account. Diversification plays an important role as well, this means that there is no single optimal strategy to group regions together (lots of variables, e.g. lighting conditions), so instead of a single strategy, which works well in most cases, selective search has a diverse set of strategies to deal with all cases. The last main feature of selective search is that it should be “reasonably fast [2].”

The second module is a large convolutional neural network that extracts a fixed-length (4096) feature vector from each region. In order to compute features for a region proposal, first we must convert the image data in that region into a form that is compatible with the CNN (its architecture requires inputs of a fixed 227 × 227 pixel size). From the many possible transformations of our arbitrary-shaped regions, the simplest one is used. Regardless of the size or aspect ratio of the candidate region, we warp all pixels in a tight bounding box around it to the required size. Prior to warping, we dilate the tight bounding box so that at the warped size there are exactly p pixels of warped image context around the original box (we use p = 16).

The third module is a set of classs-specific linear SVMs.

To sum up in a nutshell, R-CNN takes an input image, creates regions that possibly contain an object, then those regions are evaluated with the help of a CNN. The CNN used for image classification can be pre-trained Figure 2.

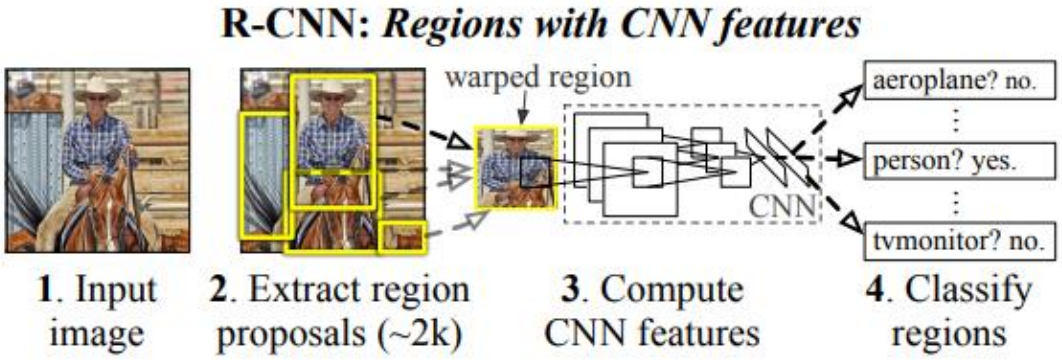

Figure 2. R-CNN workflow (Source: [1])

2.1.1.2 Fast R-CNN

Fast R-CNN stands for Fast Region-based Convolutional Network method (Fast R-CNN) for object detection [3]. This algorithm was created to improve the main problems with R-CNN, which were:

- **Training is a multi-stage pipeline**. R-CNN first finetunes a CNN on object proposals using log loss. Then, it fits SVMs to CNN features. These SVMs act as object detectors, replacing the softmax classifier learnt by fine-tuning. In the third training stage, bounding-box regressors are learned.
- **Training is expensive in space and time**. For SVM and bounding-box regressor training, features are extracted from each object proposal in each image and written to disk. With very deep networks, such as VGG16, this process takes 2.5 GPU-days for the 5k images of the VOC07 trainval set. These features require hundreds of gigabytes of storage.
- **Object detection is slow**: Detection with VGG16 takes 47s / image (on a GPU).

The network first processes the whole image with several convolutional (*conv*) and max pooling layers to produce a *conv* feature map. So instead of running a CNN for each window (~2000 times per image), we simply run it once. After that a Region of Interest (RoI) pooling layer extracts a fixed-length feature vectors from the feature map.

Each feature vector is fed into a sequence of fully connected (fc) layers that finally branch into two sibling output layers: one that produces softmax probability estimates over K object classes plus a catch-all "background" class and another layer that outputs four real-valued numbers for each of the K object classes. Each set of 4 values encodes refined bounding-box positions for one of the K classes. (Figure 3)

Fast R-CNN can use pre-trained networks as well, although they need to be modified a bit in order to work. First, the last max pooling layer is replaced by a RoI pooling layer that is configured by setting H and W to be compatible with the net's first fully connected layer. Second, the network's last fully connected layer and softmax are replaced with the two sibling layers described earlier (a fully connected layer and softmax over K + 1 categories and category-specific bounding-box regressors). Third, the network is modified to take two data inputs: a list of images and a list of RoIs in those images.

The main reasons that Fast R-CNN is faster than R-CNN are that the Conv feature map for an image is calculated once, and that the training is faster since we only need to train one network instead of three, training all network weights with back-propagation is an important capability of Fast R-CNN.

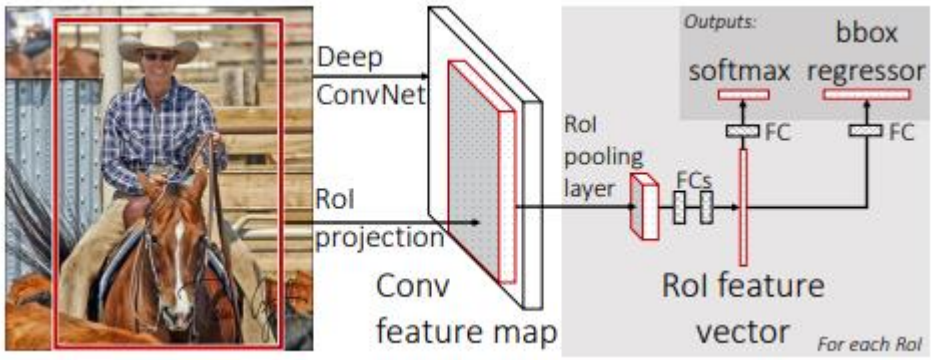

Figure 3. Fast R-CNN workflow (Source: [3])

Fast R-CNN has proven to be faster in training and testing time as well, but due to the fact, that the region proposals are the result of another model (using selective search), the improvement is not as good as it could be.

2.1.1.3 Faster R-CNN

"Our observation is that the convolutional feature maps used by region-based detectors, like Fast RCNN, can also be used for generating region proposals [4]."

The sentence above is the key to understand Faster R-CNN. Basically by integrating the region proposition into the same network that is used for classification as well makes the algorithm a lot faster, since region proposition was the main bottleneck up until this time.

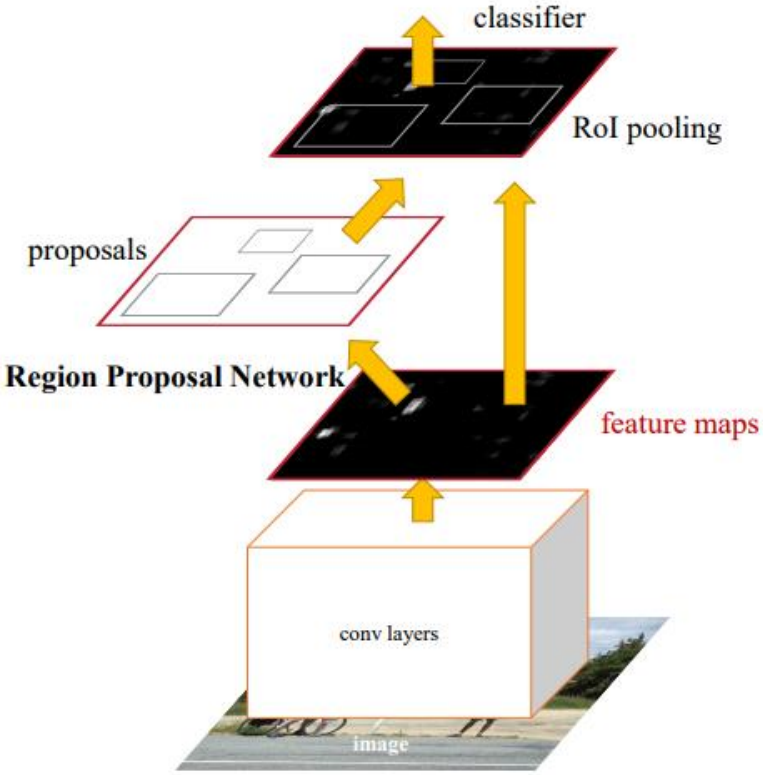

Figure 4. Faster R-CNN workflow (Source: [4])

But how does this region proposition work? For the region proposal generation a small network has to slide over the convolutional feature map output by the last shared convolutional layer. The input of this network is an n×n window over the feature map. Each sliding window is mapped to a lower-dimensional feature. This feature serves as an input two FC layers, a box regression (*reg*) and a box-classification layer (*cls*). (Figure 4)

At each sliding window position multiple region proposals are computed, the maximum number of the proposals is denoted as k. The *reg* layer has 4k outputs (4 coordinates per bounding boxes), the *cls* layer outputs 2k scores, which determines the probability whether there is an object to be found in the proposal or not.

Anchor boxes are typical bounding boxes that can be found on images, and they are associated with a scale and aspect ratio.

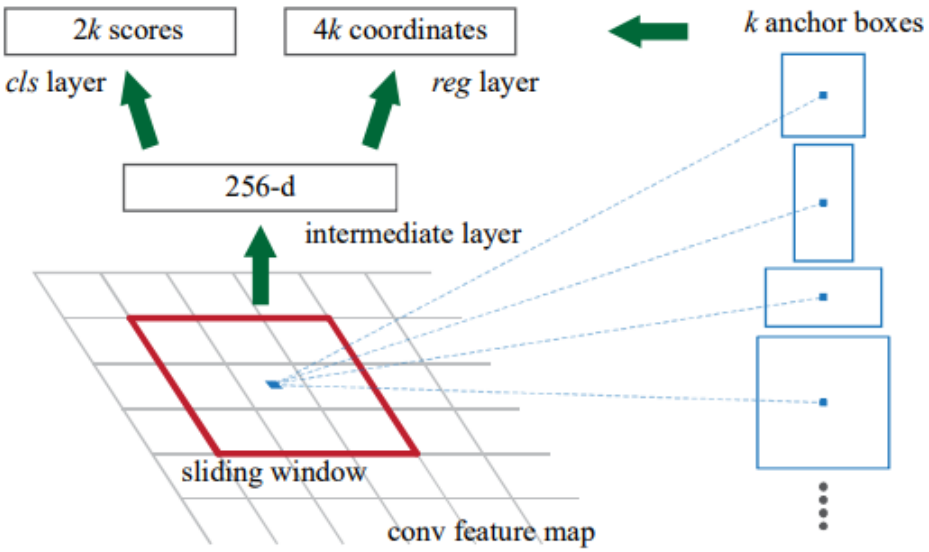

Figure 5. Region Proposal Network (RPN) (Source: [4])

So after the extraction of the feature map, we generate region proposals with the help of the RPN, and then the output of the RPN is fed into the classifier, which results in different kind of classifications and tightened bounding boxes. (Figure 5)

2.1.1.4 R-FCN

R-FCN stands for Region-based Fully Convolutional Network [5]. Although this algorithm is region based, meaning that it uses the popular two-stage object detection strategy that consists of region proposal, and region classification, it is not a direct successor of R-CNN in contrast to Fast R-CNN and Faster R-CNN.

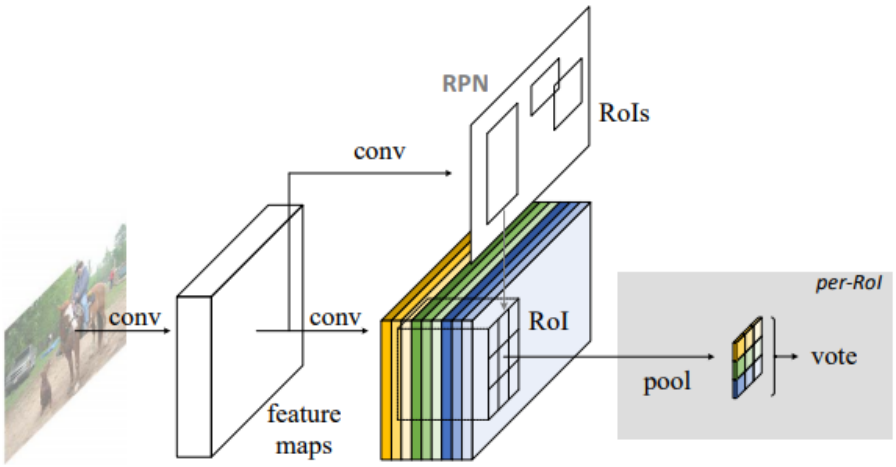

Figure 6. Overall architecture of R-FCN (Source: [5])

To incorporate translation variance into FCN, a set of position-sensitive score maps is constructed by using a bank of specialized convolutional layers as the FCN output. Each of these score maps encodes the position information with respect to a relative spatial position (e.g., “to the left of an object”). On top of this FCN, we append a position-sensitive RoI pooling layer that shepherds information from these score maps, with no weight (convolutional/fc) layers following. The entire architecture is learned end-to-end. All learnable layers are convolutional and shared on the entire image, yet encode spatial information required for object detection, see in Figure 6.

Region Proposal Network (RPN) is used for the extraction of candidate regions. Given the proposal regions (RoIs), the R-FCN architecture is designed to classify the RoIs into object categories and background. In R-FCN, all learnable weight layers are convolutional and are computed on the entire image. The last convolutional layer produces a bank of k^2 position-sensitive score maps for each category, and thus has a k^2 (C + 1)-channel output layer with C object categories (+1 for background). The bank of k^2 score maps correspond to a k × k spatial grid describing relative positions. For example, with k × k = 3 × 3, the 9 score maps encode the cases of {top-left, top-center, top-right, ..., bottom-right} of an object category, this can be seen in Figure 7.

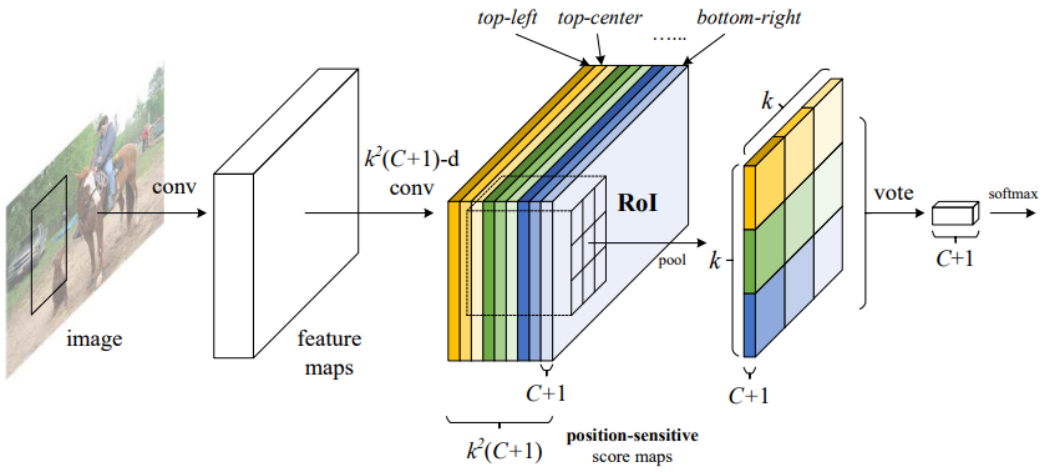

Figure 7. Key idea of R-FCN for object detection (Source: [5])

“Simply put, R-FCN considers each region proposal, divides it up into sub-regions, and iterates over the sub-regions asking: “does this look like the top-left of a baby?”, “does this look like the top-center of a baby?” “does this look like the top-right of a baby?”, etc. It repeats this for all possible classes. If enough of the sub-regions say “yes, I match up

with that part of a baby!", the RoI gets classified as a baby after a softmax over all the classes. [6]" This process is illustrated in Figure 8.

With this setup, R-FCN is able to simultaneously address location variance by proposing different object regions, and location invariance by having each region proposal refer back to the same bank of score maps. These score maps should learn to classify a cat as a cat, regardless of where the cat appears. Best of all, it is fully convolutional, meaning all of the computation is shared throughout the network.

As a result, R-FCN is several times faster than Faster R-CNN, and achieves comparable accuracy [6]."

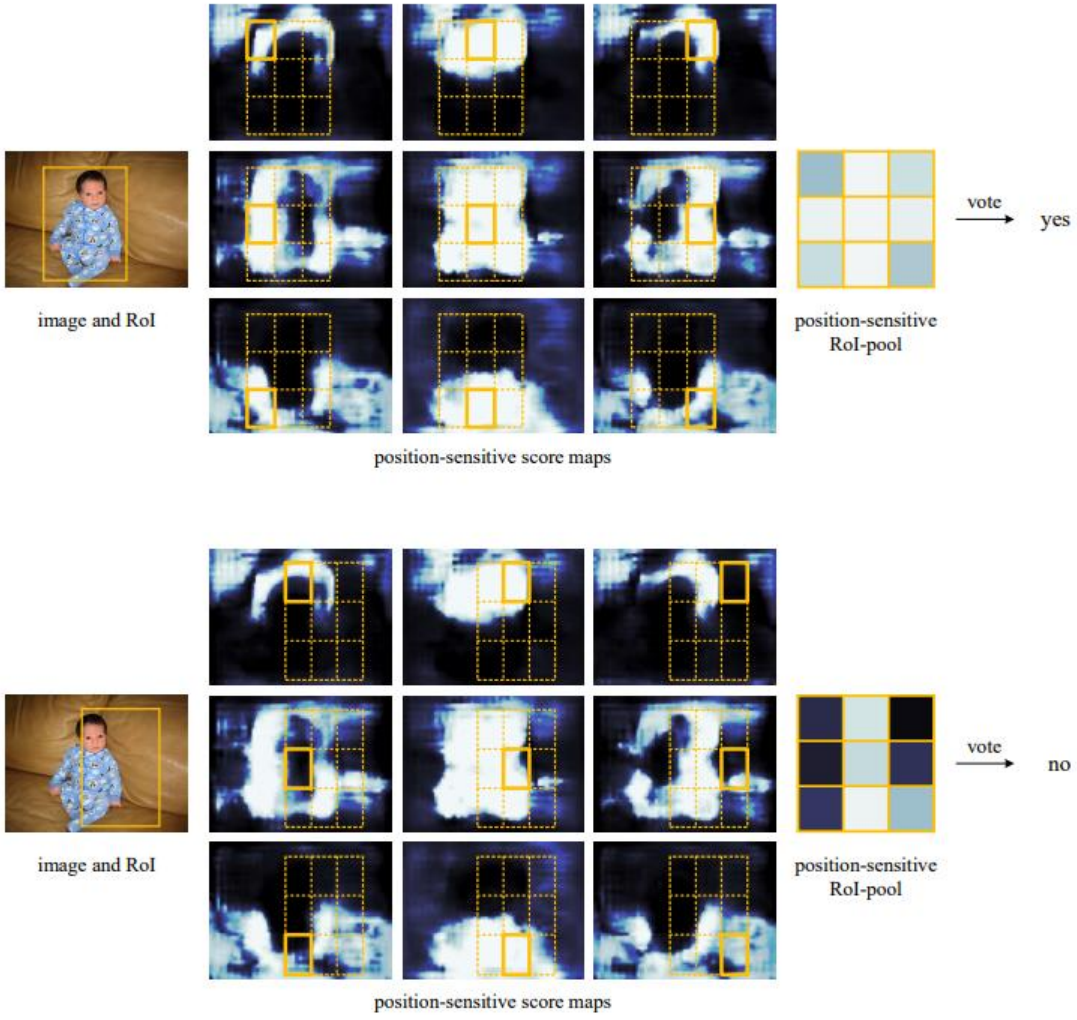

Figure 8. Visualization when a RoI (Region of Interest) does not correctly overlap the object (Source: [5])

2.1.2 Single shot object detection algorithms

Up until now we have dealt with region-based object detection algorithms. The main feature of these networks is that first they generate region proposals, after that comes the classification of those regions with the help of convolutional layers.

Single shot object detection algorithms do these steps at once, in a "single shot", they predict the classes and bounding boxes simultaneously.

2.1.2.1 Single Shot Multibox Detector (SSD)

The SSD [7] approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections. The early network layers are based on a standard architecture used for high quality image classification (truncated before any classification layers), which we will call the base network. We then add auxiliary structure to the network to produce detections with the following key features:

- **Multi-scale feature maps for detection:** We add convolutional feature layers to the end of the truncated base network. These layers decrease in size progressively and allow predictions of detections at multiple scales. The convolutional model for predicting detections is different for each feature layer.
- **Convolutional predictors for detection**: Each added feature layer (or optionally an existing feature layer from the base network) can produce a fixed set of detection predictions using a set of convolutional filters. These are indicated on top of the SSD network architecture in Fig. 10. For a feature layer of size $m \times n$ with p channels, the basic element for predicting parameters of a potential detection is a $3 \times 3 \times p$ small kernel that produces either a score for a category, or a shape offset relative to the default box coordinates. At each of the $m \times n$ locations where the kernel is applied, it produces an output value. The bounding box offset output values are measured relative to a default box position relative to each feature map location.
- **Default boxes and aspect ratios:** A set of default bounding boxes is associated with each feature map cell, for multiple feature maps at the top of the network. The default boxes tile the feature map in a convolutional manner, so that the position of each box relative to its corresponding cell is fixed. At each feature map cell, we predict the

offsets relative to the default box shapes in the cell, as well as the per-class scores that indicate the presence of a class instance in each of those boxes. Specifically, for each box out of k at a given location, we compute c class scores and the 4 offsets relative to the original default box shape. This results in a total of *(c + 4)k* filters that are applied around each location in the feature map, yielding *(c + 4)kmn* outputs for a $m \times n$ feature map. For an illustration of default boxes, please refer to Fig. 9. The default boxes are similar to the anchor boxes used in Faster R-CNN [4], however they are applied to several feature maps of different resolutions. Allowing different default box shapes in several feature maps let us efficiently discretize the space of possible output box shapes, see Figure 9.

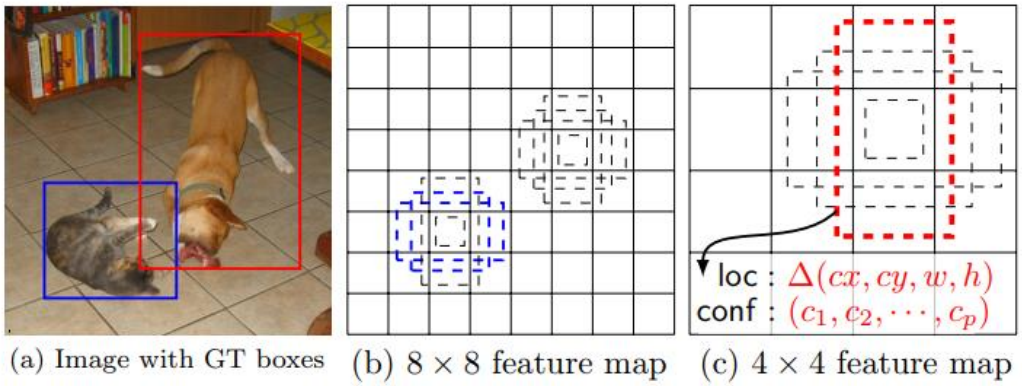

Figure 9. Default bounding boxes (Source: [7])

The workflow of SSD is basically as follows:

- Generation of several feature maps at different scales with convolutional layers
- Evaluation of the feature maps with a 3x3 convolutional filter
- For each location in each of these feature maps, a 3x3 convolutional filter is used to evaluate a small set of default bounding boxes, which are the same bounding boxes used by the anchor boxes of the Faster R-CNN.
- Prediction of the class probabilities and bounding box offsets for each box

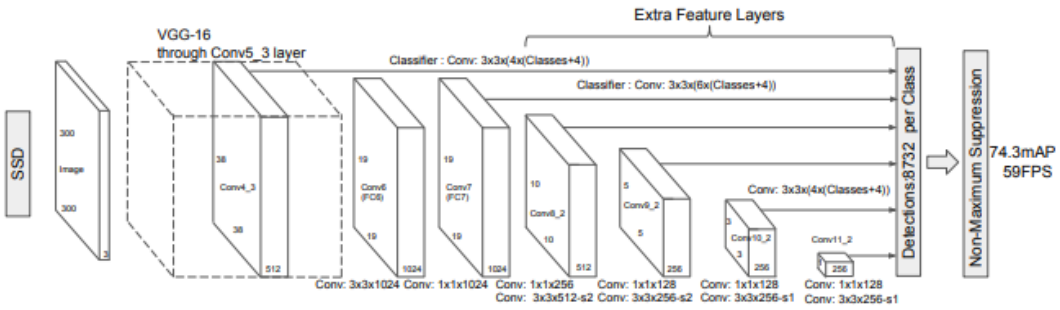

Figure 10. The architecture of SSD (Source: [7])

SSD is more accurate than previous single shot algorithms, because it outputs bounding box offset values and class probabilities at each extra feature layer (Figure 10). Since each extra feature layer is smaller than the previous one, the effective receptive field of the layers in the end of the network is getting bigger, see Figure 11.

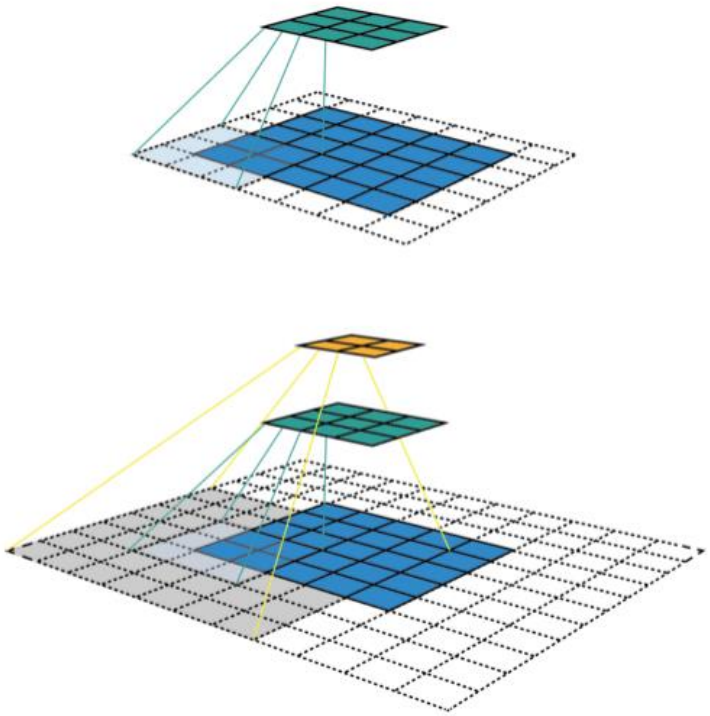

Figure 11. Changing effective receptive field after the extra feature layers (Source: https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807)

2.1.2.2 YOLOv3

YOLO (You Only Look Once) was a state-of-the-art object detection system [8], SSD was released after that and it was faster and more accurate than YOLO as well. After that the creators of YOLO started working on a better network, that was YOLOv2 [9], an

improved version of YOLO. YOLOv2 introduced many new features compared to YOLO, like batch normalization, usage of anchor boxes…etc. In the spring of 2018, after I started to work on this thesis, an even newer version of YOLO, YOLOv3 [10], came out. YOLOv3 has many developments too, but in order to have a better understanding of them, first I am describing the basic idea behind YOLO.

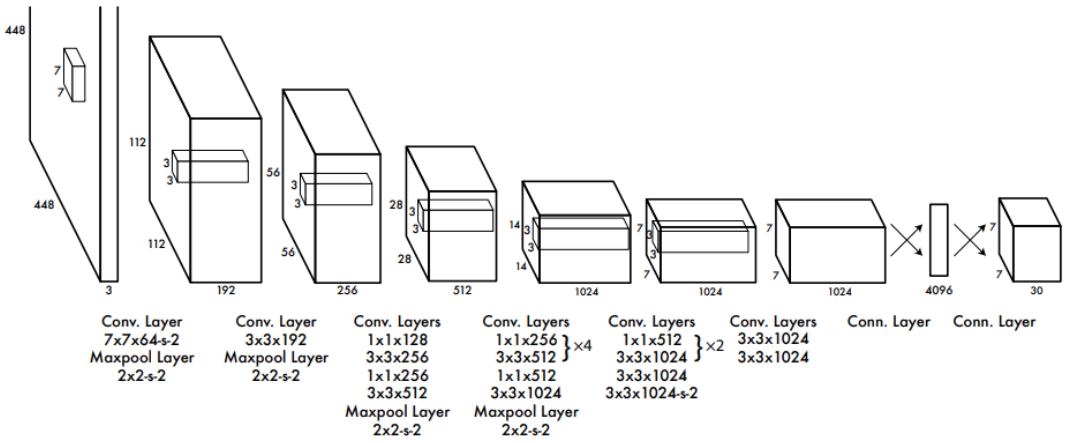

Figure 12. Original YOLO architecture (Source: [8])

The original architecture can be seen in Figure 12. First YOLO divides the image into multiple grids. The actual implementation of YOLO has different number of grids. (7x7 for training YOLO on PASCAL VOC dataset), for the illustration below 4x4 grid is used. (Figure 13)

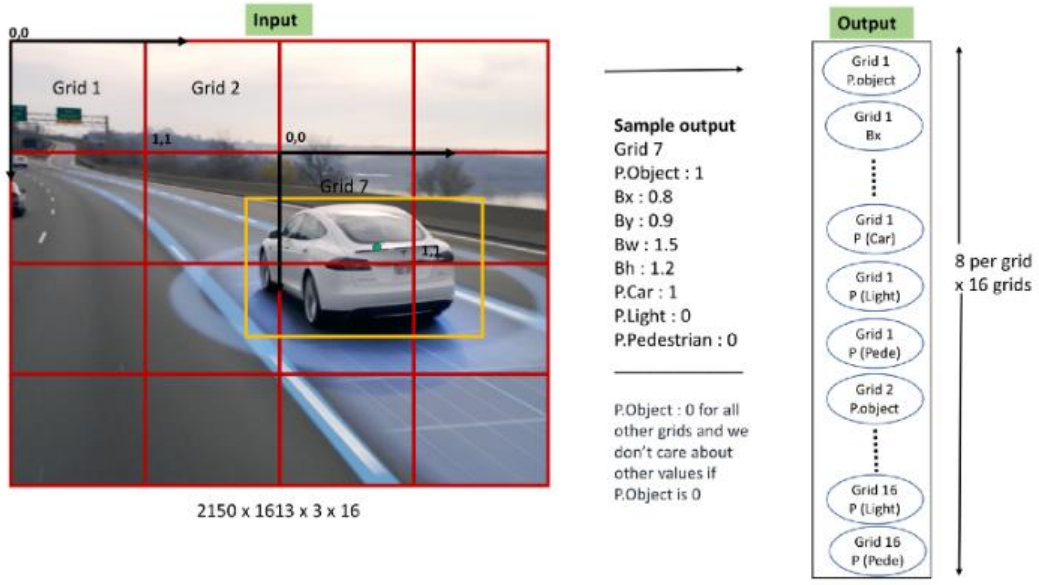

Figure 13. YOLO input and output (Source: [11])

The training data has a specific structure as well. "If *C* is the number of unique objects in our data, *S*S* is number of grids into which we split our image, then our output vector will be of length *S*S*(C+5).* For e.g. in above case, our target vector *is 4*4*(3+5)* as we divided our images into *4*4* grids and are training for 3 unique objects: Car, Light and Pedestrian." [11] The *+5* can be split into two main parts, one field carries the value whether or not the central point of an object is present in the actual grid, the remaining 4 fields give us the coordinates of the bounding boxes. This is important so that we do not count one object multiple times in different grids.

After this, a deep convolutional neural network with loss function as error between output activations and label vector is created. Basically, the model predicts the output of all the grids in just one forward pass of input image through a CNN.

In order to be able to detect multiple objects in one grid cell, anchor boxes are used. (Figure 14) In addition to having *5+C* labels for each grid cell (where C is number of distinct objects), the idea of anchor boxes is to have *(5+C)*A* labels for each grid cell, where *A* is required anchor boxes. If one object is assigned to one anchor box in one grid, other object can be assigned to the other anchor box of same grid. So that one object could not be detected multiple times, non-max suppression is applied. Non-max suppression removes the low probability bounding boxes, which are very close to a high probability bounding boxes.

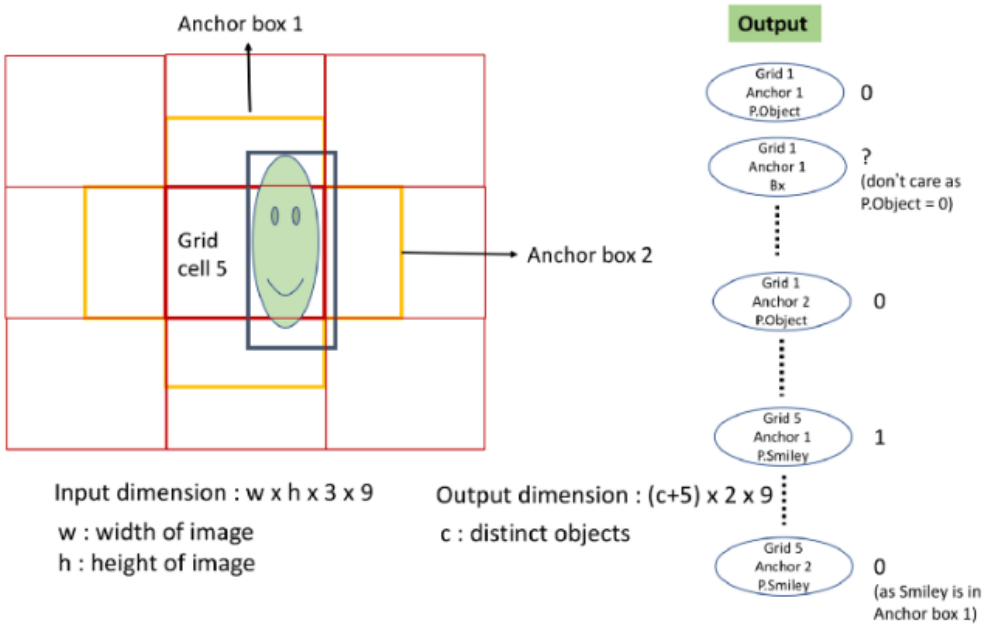

Figure 14. YOLO with anchor boxes (Source: [11])

Now that we have a basic understanding of YOLO, we can have a look at the new features and changes introduced by YOLOv3.

The most important change is that the focus in case of YOLOv3 was rather the accuracy than the speed, and this can be observed in the architecture as well.

YOLOv3 uses a variant of Darknet, a custom framework used for the training of neural networks, with a network of 53 layers trained on Imagenet. For object detection 53 extra layers after the previous 53 layer network, that is why it is called Darknet-53. All together YOLOv3 has 106 layers, which is substantially more than the 30 layers of YOLOv2, so the shift in the direction of accuracy in exchange of speed can be seen in the architecture too.

Another new feature is that YOLOv3 predicts boxes at three different scales, meaning, that now, similarly to SSD, it can detect objects of variable sizes easier than before.

The next improvement is that YOLOv3 in total uses nine anchor boxes, three anchor boxer for each scale. Now we have three scales, each divided into different grids, and for each grid we have three anchor boxes to predict, and if we sum this all up, it means that the number of predicted bounding boxes in 10647, which is ten times more, than YOLOv2 had. This is contributing to making the prediction process slower, compared to YOLOv2 too.

In previous versions of YOLO a softmax layer was used to determine which class belongs to a given anchor box, the one with the highest probability. The use of a softmax layer assumes the classes are mutually exclusive, but it is not always the case. E.g. when a dataset has both animal and dog class in it, then both labels should be predicted, this is called multilabel classification. In YOLOv3 independent logistic classifiers are used in order to achieve multilabel classification.

2.2 Long Short Term Memory networks

Long Short Term Memory[12][13] (LSTM) networks are a special kind of recurrent neural networks. Recurrent neural networks (RNN) are such networks, which contain loops, so that they can contain previous information that was fed to it, “remember” previous inputs. This way the network becomes capable of solving such tasks, which depend heavily on having contextual information as well not just only the

single one input that was actually fed at a single moment. Tasks like this are e.g. sentence creation, predicting the next word in a sentence or movement prediction based on previous frames.

LSTM networks provide a solution to the problem, that there is always a gap between an information and the time when that information is going to be useful, when it is going to influence the prediction, this is called long-term dependency. Normal RNN-s are not capable of handling too big of a gap like that, but LSTM networks are, they were created in order to handle long-term dependencies. But how do they work?

In an LSTM network, there are repeating modules, just like in case of regular RNN-s, but the main difference is in the structure of these modules. In the repeating modules of an LSTM network, there are four interacting neural layers opposed to the single layer of a regular RNN. These four layers can be represented as "gates", they decide whether an information should be kept or not, whether it should be updated or not. The information that these gates are going to influence is the cell state or cell memory, which flows straightforward from one module to another and this is partly the reason for why LSTM networks can cover larger gaps than regular RNN-s.

The first gate is the so called forget gate (Equation 1), it decides how much of the historical information we should forget.

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \tag{1}$$

In Equation 1 we can see the actual mathematical representation of the forget gate, its influence on the output is going to be explained further a bit later. The forget gate is basically a sigmoid function and the input of this function consists of two main parts, the weighted (W_f) values of the historical information ($a^{<t-1>}$) and the actual input ($x^{<t>}$) and the bias values (b_f). In every equation in this section the letter **W** is denoting the weights, the letter **a** is denoting the historical output data, **x** represents the actual input and **b** is the bias.

The second gate is the update gate (Equation 2), it shows by how much the historical value should be modified in order to keep it up to date.

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \tag{2}$$

Now we know whether or not we should update the old values, but how are we going to modify them, what are the new values?

$$\tilde{c}^{<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \quad (3)$$

In Equation 3 we see the new candidate values (**c**) for the cell state overwriting, which would overwrite the old, historical values, they are simply calculated with a **tanh** function.

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \quad (4)$$

Equation 4 represents the calculation method for the new cell state values. It is the sum of two main parts. On the one hand it uses the above described forget gate to determine how much exactly we are going to forget and on the other hand we can see the update gate, which determines by how much we are modifying the old values by the new ones.

The fourth gate is the output gate (Equation 5). This gate is playing a major role in creating the output data for an actual repeating module.

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \quad (5)$$

$$a^{<t>} = \Gamma_o * \tanh c^{<t>} \quad (6)$$

In Equation 6 we see how the output gate is put to use. For the calculation of the output value the forget gate is multiplied by the above calculated value of the cell state, modified by the **tanh** function.

To sum up, all repeating modules have three inputs (the actual input, the historical output values and the cell state values) and two outputs (the actual output, the actual cell state value). Based on this structure it can be seen that these modules can be linked easily to one another, making it possible to create bigger networks, consisting of multiple repeating modules really simple. Sometimes a single module can have a third output as well, and that would be the actual output of the calculation, which is led directly "outwards", that could be directly saved and not piped into the next module.

3 Setting objectives

After we review the different object detection algorithms, I would like to summarize once again, what the objectives of this thesis are.

I would like to build a system that consists of two main parts.

- The first main part is the chosen object detection algorithms, which is processing the images and saving the results of this detection process.
- The second part comes into play, when the object detection algorithm managed to produce enough input data for the predicting network, that, depending from the chosen network, is going to need a certain number of inputs from earlier frames. Based on the input data, the predicting network is going to output a possible future location for each object that was previously detected by the object detection algorithm.

The process described above is going to make up a constant loop, meaning that after a successful prediction we return to the chosen object detection algorithm in order to be able to feed further inputs for the prediction network. We can achieve the reduction of hardware needs through the skipping of processing certain number of frames. We can skip to process some frames, because we have already a future location for all the objects detected earlier and we do not need their exact position in between the earlier, detected and the predicted frames. The hardware and energy needs of such a system would be lower than one of a constantly running detection algorithm.

Another important point to note would be, that with this solution we could act earlier in time, based on our predictions of the future, this could be used extensively in many fields, e.g. in transportation.

4 System Design

The main goal of this thesis is to create a neural network that is capable of predicting the trajectory, the movements of various objects in videos. For this, I am using a state-of-the-art object detection algorithm, which has to deliver a classification and four bounding box coordinates for a given object in a frame.

The process of determining the state-of-the-art algorithm to be used can be seen in Figure 15. We use a dataset, which consists of videos, where very frame is annotated, then both YOLO and SSD process the videos and the results of these algorithms is processed on a metric, the exact describing of the metric is going to be discussed later, in Chapter 5.2.1

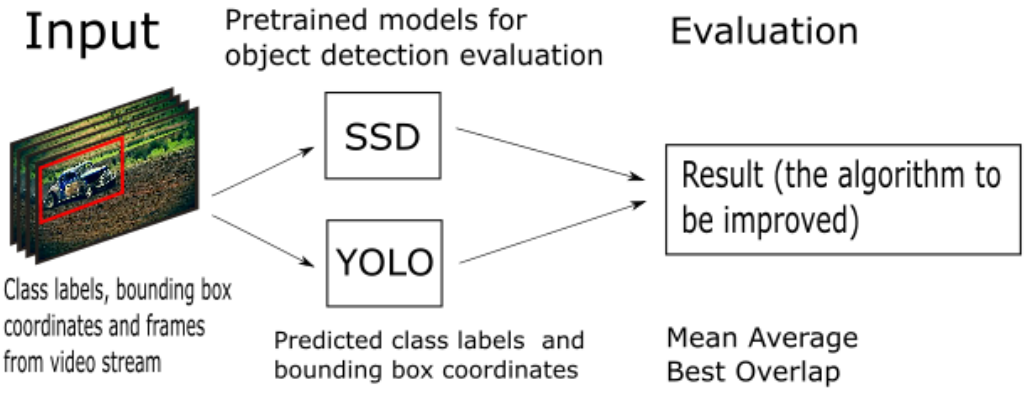

Figure 15. Evaluation process

After an appropriate object detection algorithm was chosen, the next step of the work begins. This step includes the creation of a neural network, which is capable of predicting the shift in the bounding box coordinates over *n* frames and determining the correct class label as well, shown in Figure 16.

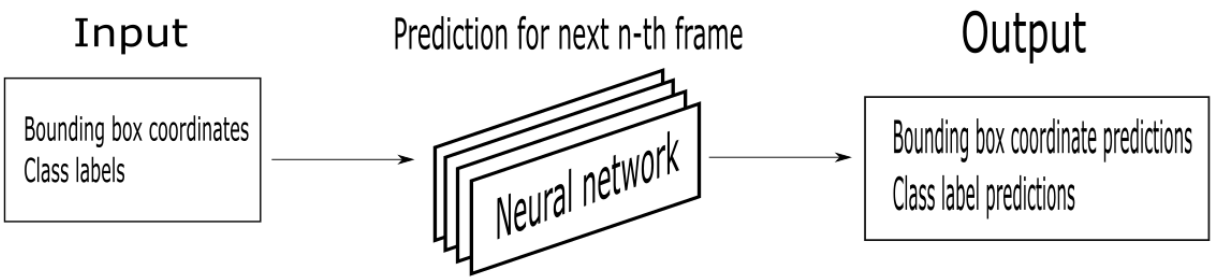

Figure 16. Prediction of the shift in the coordinates of the bounding boxes.

The full workflow is intended to work the following way (Figure 17):

- Input a video, input frames from a video
- Based on the current and previous frames, predict the class labels and the bounding box coordinates for the next *n*-th frames

In case of a video the output of the neural network can be inspected quite well.

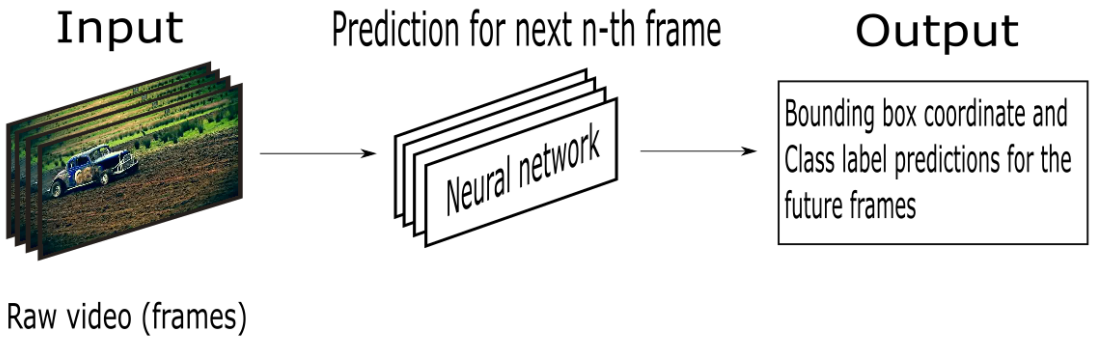

Figure 17. Complete workflow

The system for development is a workstation that runs Ubuntu (Linux) with Nvidia GeForce GTX TITAN X (with 12 GB GDDR5 memory), Intel(R) Core(TM) i7-4790 CPU@3.60GHz and 32 GB RAM installed. The main programming language, this being a development process, is going to be Python 3.6, while of course other tools, such as bash scripts, are going to be used extensively as well.

The system that is going to be developed is going to consist of two main parts.

The first main part is going to be the chosen object detection algorithm that is producing the input data necessary for the prediction network. The model of the object detection algorithm is going to be loaded into the memory at the start of the program, so that we do not have to load it each and every time we need to process an image. So the

object detection algorithm is going to be running constantly, while waiting for inputs, for images to process.

The second main part is the prediction system. This system utilises the data that was delivered by the object detection algorithm and predicts future locations of the earlier detected objects. After the prediction, the execution is going to continue with the object detection again, making this a two-step loop.

Thanks to the prediction, we can skip the processing of certain images, frames, so that the processing of a video is going to require much less hardware resources.

5 Dataset and evaluation

5.1 Dataset

The dataset used for the comparison of the different algorithms is the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2015 VID dataset. [14] Properties of the dataset:

- This dataset consists of videos of various resolution and length, the frames of the videos are also included in a separate directory structure
- Every frame of the videos is annotated. The annotations are saved in XML format
- For the determination of the class labels (Figure 18), a map file is necessary, which maps the codes of the classes to human readable class labels.
- The full dataset is approximately 100GB and is divided into training, validation and test sets. The size of these sets are respectively 51 GB, 8 GB and 14 GB
- All together 1122397 annotated frames, from which 866870 frames are annotated by hand (more than 77%)
- All frames of the videos are available as JPEG images as well.

```
n01503061 5 bird
n02924116 6 bus
n02958343 7 car
n02402425 8 cattle
n02084071 9 dog
```

Figure 18. A segment of the map file

5.2 Evaluation of the dataset

Because the whole proposed system is based on an existing object detection algorithm that can produce the necessary input data for the prediction network, the capabilities of the object detection algorithms, which were possible candidates for this role, needed to be evaluated first in order to be able to continue the developing of the rest of the system.

5.2.1 Evaluation metrics

5.2.1.1 Mean Average Best Overlap

The evaluation framework consists of a bash script, which runs SSD and YOLO for the given videos and saves the results into a given hierarchy. The other part of the framework is a Python script, which calculates the results based on the results from the object detection and from the ground truth values. The metrics used for the evaluation is the Mean Average Best Overlap (MABO). [2] To understand MABO, first we need to get to know to the Average Best Overlap (ABO), since MABO is merely a generalization of that.

"To calculate the Average Best Overlap (Equation 7) for a specific class c, we calculate the best overlap between each ground truth annotation $g_i^c \in G^c$ and the object hypotheses L generated for the corresponding image, and average:" [2]

$$ABO = \frac{1}{|G^c|} \sum_{g_i^c \epsilon G^c} max_{l_j \epsilon L} Overlap(g_i^c, l_j) \tag{7}$$

,where *Overlap* (Equation 8) is the area of the intersection of the two bounding boxes divided by the union of those:

$$Overlap\left(g_i^c, l_j\right) = \frac{area(g_i^c) \cap area(l_j)}{area(g_i^c) \cup area(l_j)} \tag{8}$$

MABO is the mean ABO over the different classes. In our case, we calculate the mean of the ABO values over the frames, then the mean of the ABO values for the different classes.

5.2.1.2 Mean Average Precision

Mean Average Precision (*mAP*)[15] is the most used metric for object detection algorithms, this is partly thanks to the Pascal Visual Object Classes Challenge[16], where it became the main metric for the evaluation.

In order to understand *mAP*, first we have to understand *recall* and *precision*.

Recall (Equation 9) gives us, how much of the relevant data we have found.

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{9}$$

Precision (Equation 10) gives us, how big of a portion of the found data was actually correct.

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{10}$$

From our prediction results, which are organized in a descending order of the confidence level, we can create the *recall-precision* curve. This curve is usually not smooth at all, so in order to get a smooth curve we take eleven points (0, 0.1, 0.2, ... , 1) from it, and interpolate a curve, and the area below that curve is the *average precision (AP).* (Equation 11)

$$AP = \frac{1}{11} \sum_{r \epsilon \{0, 0.1, \ldots, 1\}} \rho_{interp}(r) \tag{11}$$

,where

$$\rho_{interp}(r) = max_{\tilde{r} \geq r} \rho(\tilde{r}) \tag{12}$$

Equation 12 means, that we do not take the exact precision values at the observed points (*r*), but rather a maximum precision whose recall value is greater than *r*.

If we calculate the *AP* values for all the different object classes and then calculate the mean of those, we get the *mAP* value.

5.2.2 Evaluation results

As the main goal is to track the objects as precisely as possible, below are some demonstration images, which show the ground truth values and the results of the object detection algorithms:

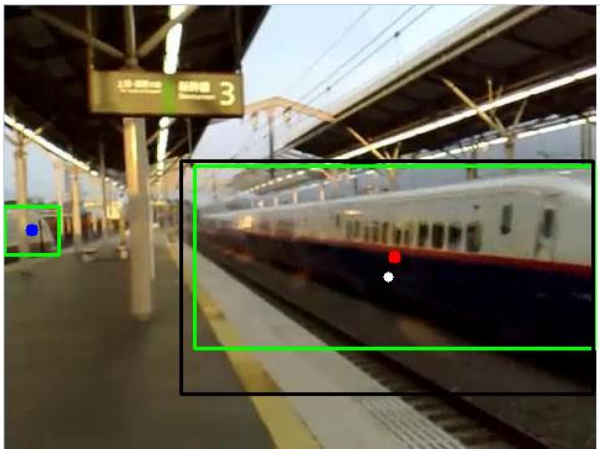

Figure 19. Ground truth bounding boxes (with green) and the middle point of those (blue and red) and the result of YOLO bounding box (black) and its middle point(white)

In Figure 19 we can see that the predicted bounding box from YOLO is quite accurate compared to the ground truth bounding box. The smaller train on the left was not found in this frame, but thank to its small ground truth bounding box this does not matter as much, since in the metric we have to compute a mean of the overlap values, so this component has a little weight in the sum.

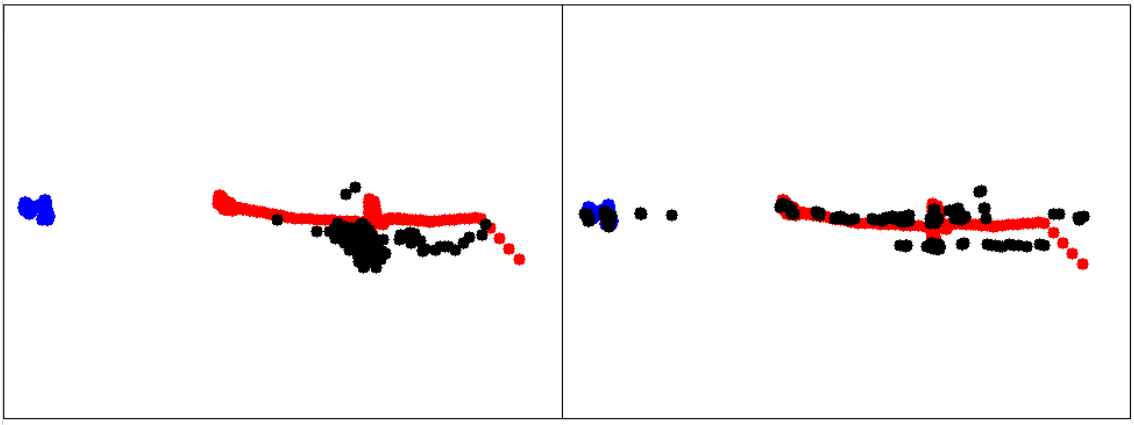

Figure 20. The movement of the middle points of the ground truth bounding boxes (blue and red) over the whole video, and the predicted center points for SSD (left, with black) and YOLO (right, with black)

In Figure 20 we can see the movement of two objects compared to the ground truth movement in a video tracked by both SSD and YOLO. The trajectory was formed

from the movement of the middle points of the bounding boxes. In this instance YOLO showed better tracking capabilities than SSD, because the results of YOLO are closer to the ground truth red line than the results of SSD, besides YOLO was capable of recognising the second train on the left hand side, while SSD missed it completely. Next we have to investigate the full, overall results of these algorithms.

The full evaluation with metrics *AP* and *ABO* as discussed above on the full dataset resulted as follows (The dataset has thirty classes, in Table 1. only those classes are present, that were recognised by either of the algorithms):

Table 1. Evaluation results of the metrics on the dataset

Metrics / **Classes**	**AP (%)**		**ABO**	
	SSD	**YOLO**	**SSD**	**YOLO**
bear	0	18.72	0	0.86
bicycle	45.09	45.90	0.69	0.73
bird	34.08	52.68	0.73	0.78
bus	63.10	64.30	0.78	0.77
car	58.53	57.87	0.75	0.80
dog	37.16	45.55	0.78	0.82
elephant	0	63.93	0	0
horse	63.39	78.67	0.80	0.84
sheep	29.54	25.12	0.74	0.86
train	67.70	71.30	0.79	0
zebra	0	69.98	0	0.83

I

In Table 2. we can see the overall results for the two algorithms, in the calculation of the values only those classes were taken into consideration that were recognised by the given algorithm:

Table 2. The results of the prediction processes

Metrics / Algorithms	mAP(%)	MABO
SSD	49.82	0.76
YOLO	54.00	0.81

As we can see in the table above, the results of YOLO are better considering both mAP and MABO, higher numbers are better in both instances.

The results of the MABO evaluation has quite a graphic meaning: it shows in how big of a ratio the ground truth and the predicted bounding boxes are overlapping.

An important side to consider is the hardware utilisation during the prediction (Table 3.).

Table 3. The hardware utilization results of the prediction processes

Hardware utilisation / Algorithms	CPU (%)	Memory (%)	Streaming Multiprocessor Utilisation (%)	GPU Memory Utilisation (%)
SSD	103.15	7.78	42.67	28.28
YOLO	103.98	1.41	42.67	19.45

The hardware utilisation results are very similar, YOLO utilises the CPU slightly more, while SSD is more reliant on both GPU and RAM memory.

The summary of the results is that YOLO can track the movements of objects better than SSD at the moment. Based on the two metrics, MABO and mAP, the results of YOLO exceed the result of SSD in both metrics. In case of mAP YOLO is better than

SSD by more than 4% and in case of MABO it is better by 0.05, meaning it can detect objects more accurately, than SSD.

There is not a significant difference between the two in the hardware utilisation results, so for the next phase of the work YOLO will be the base algorithm.

The conclusion, that YOLOv3 is more precise, than SSD, is also supported by official measurements. (Figure 21.)

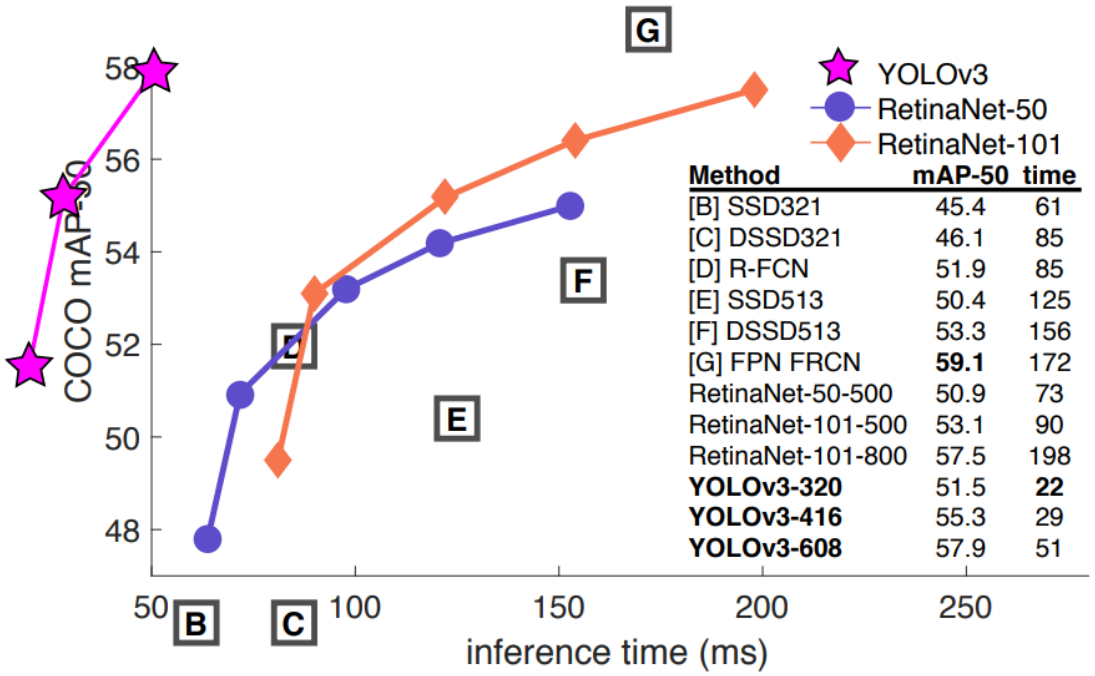

Figure 21. Speed and accuracy comparison of different algorithms[10]

6 Proposed solution

The fully realised system consists of two main parts. One part is the object detection system, which is responsible for the creation of the input data for the other main part of the system, the predicting network, which then predicts future locations for the detected objects.

6.1 Object detection system

The object detection system is a framework written in Python, which incorporates both the chosen object detection solution, YOLOv3, and the network used for prediction.

The object detection system loops over the images of a video, and then feeds these images to the object detection algorithm, alongside with the sequential number of the image. The object detection algorithm was modified to save the results in a given folder and the name of the result file is equivalent to the sequential number of the image, this makes the evaluation process much easier, because the ground truth annotations for the images in the dataset are named in the same convention.

But how do we know when to give the next image as input for the YOLO network? Naturally, YOLO has a variable processing speed and time per images, so based on that theoretically we could wait a time long enough and then feed the next image to the YOLO algorithm accordingly. This solution however would not be too efficient, so in order to achieve a higher processing speed, another solution was implemented. After the framework loads the YOLO model into the memory, another thread is started, which reads constantly the output of the subprocess running YOLO. Since the output of YOLO was modified to indicate the end of the image processing with a given string, this way we can easily see whether or not YOLO is ready for the next input.

While implementing this method, two main difficulties arose. First I did not want to start another thread, because the subprocess management in Python makes it possible to simply redirect the output of a subprocess with a pipe and then with different methods (e.g. *readline*()). This solution did not function, because YOLO does not have information constantly written to its output and this causes all methods, which would read the lines of the output, fail. This is the reason for why another thread has to be started.

Another difficulty was, that YOLO does not write its output to *stdout*, but to *stderr* and this took me some time to realise.

So up until now we have a system, which feeds images to YOLO, and then the results are saved in a given folder. Here it is important to note that the system has two free variables, which depend on the network used for prediction.

One variable determines how many results from previous moments are used as inputs for the prediction. So for example, we have a network used for prediction that expects data from five timestamps, moments. That means that ideally we should have five result files, each containing the same detected class and its different coordinates in five different images. Then we could use this data of five timestamps as input for the prediction network.

The other variable determines how far into the future, how many timestamps, moments ahead we would like to predict the location of objects. E.g. we could predict the location two timestamps ahead, this means that we could skip the processing of two images, the detection of objects in two images, since we already have a location for the object further ahead in the future. After two timestamps or frames, we have to use the object detection algorithm again.

To sum it up, the first variable determines how many result files we should have in the result folder, how many processed frames are necessary for the whole prediction process. The second variable shows us how many images, frames we should skip, because we already predicted the location of the objects for frames that come later in time.

6.2 Network for prediction

6.2.1 The preparation of the dataset used for training and validation

In the dataset, there is an annotation file in XML format to every frame. These annotations were created either by hand or were generated. The usage of XML files is not as convenient as other file formats, so I decided to generate separate training and validation files in *csv* format for every class.

Since there are cases, when not just one, but two or more objects are present in a frame, the creation of the aforementioned files was a bit difficult. Because of the presence of multiple object in a frame, it is not ideal for the further usage of these files to simply

loop over the files and write everything to the files, because then every line could contain different objects and that would make the post-processing quite complex.

Instead of doing that, I decided to create separate temporary files for every object in a video and write the data belonging to the objects with given object IDs to the appropriate temporary file. In the end of a video, the content of the temporary files are moved to the final file and concatenated simply behind one another.

The final files contain the video and frame IDs, the resolution of the given image, the object label, the coordinates of the bounding box and finally the object IDs. The coordinates of the objects are written standardised to the file too, meaning that the x and y coordinates are divided respectively by the width and height of the image.

Besides all of the above and because of the existence of generated annotations and its relevance in later usage, I decided to write the "status", whether an annotation was generated or made by hand, to the final files as well. In Figure 22 we can see a short part of a final training file, though it is not its full content, the standardised coordinates are missing from it. When using the files for training, one must pay attention, so that only such data is used, that belongs together. For example, the video ID must be the same when creating a set for training, because only within a video can we speak of coherent data. Another aspect to consider is the object ID, we cannot mix data from different objects even when they are present in the same video.

```
1  video_ID,frame_ID,class,xmin,ymin,xmax,ymax,width,height,object_ID,generation_statu
2  0000/00097000,000000,bear,524,97,1054,590,1280,720,0,0
3  0000/00097000,000001,bear,519,96,1052,593,1280,720,0,1
4  0000/00097000,000002,bear,520,94,1041,593,1280,720,0,0
5  0000/00097000,000003,bear,517,99,1036,595,1280,720,0,0
6  0000/00097000,000004,bear,512,97,1036,590,1280,720,0,0
7  0000/00097000,000004,bear,511,96,1033,590,1280,720,0,0
8  0000/00097000,000004,bear,510,96,1030,590,1280,720,0,0
9  0000/00097000,000005,bear,510,95,1025,592,1280,720,1,0
10 0000/00097000,000005,bear,511,95,1020,592,1280,720,1,0
```

Figure 22. An example of the final training file

I decided to experiment with three different data preparation methods that can be used for training and evaluation.

- *original method:* the first method uses the every line of the file, makes no difference between the annotations made by hand and the generated annotations. E.g. in Figure 22, in case of using data from five earlier frames and predicting only the next one

ahed, the first five rows would constitute a sufficient set, plus we need the sixth row, that is the result we expect from the network

- *jump-over method*: in this case, we simply jump over the lines, which were generated and not made by hand, so in this case the first six rows would make up to a set, because the second row would be jumped over, since it is a generated annotation, the seventh row would be the expected result
- *skip method*: in this method we look for consecutive rows, meaning that we only use the data for training, when there is sufficient amount, the number of earlier timestamps needed, annotations made by hand in consecutive rows. So in this case, we would need five rows, where the video and frame IDS, the object ID is the same for five rows and each row needs to be annotated by hand. This method is the strictest out of all the described methods

6.2.2 Actual network

The implementation is written in Keras, which is an open source neural network library written in Python. Keras can be used with different backends as well; in my implementation, Tensorflow was used.

The model is a sequential model, meaning that it is a linear stack of different layers, it can be completely customised in this aspect.

The first layer is an implicit input layer, it is determined by the *batch_input_shape* argument of the first added neural layer. This makes it possible, that there is no need to include in every layer an argument regarding the shape of the input, they can be inferred from the output of previous layers.

The second layer is the actual LSTM layer with 16 cells. Cells are units, which together form a layer, they have also an impact on the dimensionality of the output space. As described earlier, LSTM networks work great with object movement prediction, so they would be a good choice for this task. LSTM network can be used effectively in case of time-series [17] problems. Time-series problems represent such tasks, where we have information about events in the past and we want to predict the next event like that in the future based on the earlier events. Since we have a fully annotated dataset, we can easily create a time-series problem. The main obstacle in the generation of the training and validation datasets was that sometimes there are multiple objects in a video, so there are

multiple objects in the annotation belonging to the frames. Due to this we cannot simply read the annotations in order and assemble a training dataset, because then the annotation of the different objects were mixed together, which would result in a quite complicated post-processing. Because of this the annotation data of the object are collected separately for the objects, based on the object identification number, and they are concatenated only at the end of the video.

All that having said, we have to note that the input of this layer depends heavily on the actual configuration we have set earlier. The configuration consists of the number of timestamps, of how many earlier data we would like to base our prediction on. Besides that we can change the time of the prediction as well, by that we can change how far ahead in the future we would like to predict the position of the object. So on the one hand the input is determined by the number of timestamps, and on the other hand it is fixed. It is fixed in one aspect, because no matter how many timestamps or prediction distance we choose, we always have to input the four coordinates and an integer-label representing the given class.

The third layer is a so called *Dense* layer, which represents a fully connected neural network layer. This being the last layer, the output of this layer is going to determine the output of the whole network. Since we want to predict the future location of an object, we need to have the four bounding box coordinates as an output, so this *Dense* layer has four neurons as outputs as well.

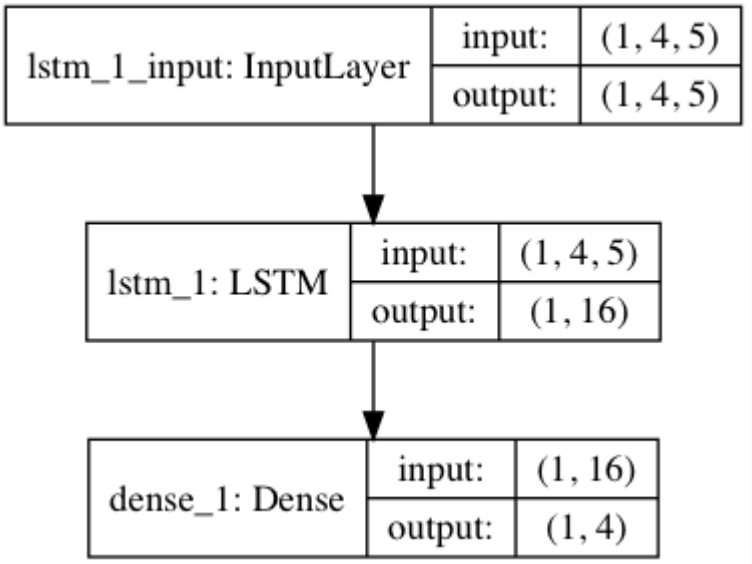

Figure 22. An example of the inputs and outputs of a network

In Figure 22. we can see an LSTM network that requires data from four earlier timestamps, and based on the five data (four coordinates and the class labels) from those timestamps it predicts the future location. In this example the prediction distance was 3, however this cannot be seen in the picture of the model, it only influences the pre-processing of the annotation and the validation data.

The training and validation data needed for the creation of the LSTM network was extracted from the original annotations. Both for the training and the validation, the input coordinates were standardised, meaning they were divided respectively by the height or the width of the images.

Different versions of the network with the same architecture as above were trained, they are going to be discussed further in detail later, here I just list the different types:

- trained for every class with the original method data preparation
- trained for the 11, known by YOLO, classes with the original method data preparation
- trained for the 11, known by YOLO, classes with the jump-over method data preparation
- trained for the 11, known by YOLO, classes with the skip method data preparation

7 Results

The results (Table 4.) of the realised solution, where the object detection algorithm YOLOv3 and a trained LSTM are working together are going to be detailed in this section. The used weights for YOLOv3 were downloaded from the official website, the LSTM model was trained on the university server.

The LSTM model used for the measurements takes inputs from five earlier frames and predicts three frames ahead in the future.

For the training, the training files, generated from the annotations of the training videos, were used, for the validation the validation files generated from the annotations of the validation videos were used. The LSTM model was trained for different number of epochs, and for every model, the validation results were saved so that a good model could be chosen easily. The validation results were created by calculating the root-mean-square error (RMSE) between the validation ground truth values and the predicted values. Four different LSTM models were tested.

The used loss function was the *mean_squared_error* loss function, and the optimizer used for the training was *Adam*.

1. *all LSTM*: The first model was trained for every class (30) and for 62 epochs, because the RMSE values calculated from the validation data were not decreasing any longer. The results of this model were at best on par, but in average worse than the results from linear regression. Since this performance was not good enough, I tried to change up a few things.
2. *original 11-classes LSTM*: The second model was trained only for the 11 classes that are known to YOLOv3 base model, so this YOLO model could not recognise the other 19 classes anyway. This model was trained for 68 epochs.

Since I was looking for further improvements, I decided to change up the data preparation part of the training script. Up until now, all of the annotation data was used for the training. As I mentioned earlier, more than 77% of the annotation data was annotated by hand.

3. *jump-over LSTM*: The first variant of the new data preparation excluded the generated, not by hand annotated annotations for the objects. So whenever a

generated annotation comes by, it is simply jumped over, and the processing continues with the oncoming annotation data. In this case it is not necessary, that the data needed for the training is located consecutively, there can be gaps between the hand annotations, those hand annotations are going to be grouped together. The model shown in Table 4. was trained for 131 epochs.

4. *skip LSTM* : The second variant of the new data preparation is even stricter than that, it looks for as many consecutive hand annotations as many timestamps we have set. So e.g. if we need information from 5 earlier timestamps, than only such annotations are included, where all of the 5 consecutive frames were annotated by hand. The used version of this model was trained for 120 epochs.

To evaluate the results of the different models, a baseline needed to be determined and implemented. For the evaluation of the results of this previously described hybrid system consisting of an object detection algorithm and an LSTM network, I used linear regression. The script implementing the linear regression works very similarly as the real system. This script loops over the result files, which are ordered into different folders, one for each video. Then based on the two free variables, the number of previous timestamps needed for input and the "distance" of the prediction, the script uses the same inputs as the real system, and with linear regression "predicts" a future location for the objects and saves the results in the same folder structure. This way we have the same structure for the real results and for the linear regression results, so the comparison of these two are quite simple.

Table 4. The results of the two methods, linear regression and LSTM prediction

Metrics / **Classes**	**mAP(%)**					**ABO**				
	all LSTM	***orig. 11 cl. LSTM***	***jump-over LSTM***	***skip LSTM***	***Linear regression***	***all LSTM***	***orig. 11 cl. LSTM***	***jump-over LSTM***	***skip LSTM***	***Linear regression***
bear	17.92	22.58	22.80	22.81	22.33	0.65	0.73	0.78	0.77	0.73
bicycle	25.87	46.96	48.39	48.35	47.24	0.58	0.67	0.67	0.67	0.66
bird	33.16	42.53	43.56	43.58	41.41	0.62	0.70	0.73	0.74	0.69
bus	46.72	55.38	55.45	55.96	54.44	0.65	0.71	0.73	0.73	0.70

car	32.66	37.61	39.25	41.78	41.83	0.65	0.68	0.74	0.74	0.71
dog	41.43	43.49	44.32	44.84	43.13	0.67	0.72	0.75	0.75	0.70
elephant	29.39	31.24	31.98	32.35	30.34	0.66	0.73	0.76	0.77	0.70
horse	54.31	55.50	55.92	56.66	54.63	0.68	0.73	0.76	0.77	0.70
sheep	14.58	14.93	15.34	15.72	14.98	0.71	0.72	0.74	0.77	0.78
train	76.34	83.53	83.89	83.64	83.83	0.72	0.73	0.74	0.74	0.72
zebra	26.89	28.78	29.12	29.28	27.35	0.70	0.70	0.72	0.72	0.72

In Table 4 we can see the test results of all four, different models. It can be seen, that the *all LSTM* model was in case of every class worse, than linear regression. Because of this, I decided to reduce the number of classes to do the training for, so the only difference between *the all LSTM* and *original 11-classes LSTM* is the number of classes that they were trained for. This only difference resulted in much better results, *original 11-classes LSTM* is already better in most cases, than linear regression. *jump-over LSTM* is the model, where the *jump-over* data preparation method was used. This model is even better, then the previous ones. The last model, *skip LSTM* is the best out of these four models, this network was trained with using the *skip* data preparation method.

The higher the numbers in case of both metrics, mAP and MABO, are, the better.

In Table 5. we can see the overall results for the five different methods, the prediction with the LSTM models and linear regression:

Table 5. The hardware utilization results of the prediction processes

Metrics / **Algorithms**	**mAP(%)**	**MABO**
all LSTM	36.30	0.66
original 11-classes LSTM	42.05	0.71
jump-over LSTM	42.73	0.74

skip LSTM	43.18	0.74
Linear regression	41.96	0.71

The overall results (Table 5) support the results, we have seen in Table 4. *All LSTM* has the worst performance, its *mAP* and *MABO* is the lowest of all. Just by reducing the number of classes to do the training for, significant increase in accuracy could be achieved. *Original 11-classes LSTM model* is better than *all LSTM* by more than 5% in *mAP*, and 0.05 in *MABO*. The first usage of the changed data preparation is in the case of *jump-over LSTM*. Just by using another method for the data preparation, we could achieve 0.68% increase in *mAP* and 0.03 in *MABO*. The best model was the *skip LSTM*, for the training of this model the *skip* data preparation method was used. This change resulted in another 045% increase in *mAP*, but no increase in *MABO*. It can be seen, that the different data preparation methods have quite a high impact on the overall performance of the network. Between the *original 11-classes LSTM model*, *jump-over LSTM* and *skip LSTM* the only difference was in data preparation and thanks to this a positive change of 1.13% in *mAP* and a 0.03 in *MABO* could be achieved.

The hardware utilisation results of the full can be seen in Table 6, this is for the best model, the *skip* model:

Table 6. The results of the prediction processes

Hardware utilisation / **Algorithms**	**CPU (%)**	**Memory (%)**	**Streaming Multiprocessor Utilisation (%)**	**GPU Memory Utilisation (%)**
Full system	12.96	4.2	2	1

8 Evaluation

The results show, that this architecture could not be successfully trained for all thirty classes, which then makes the training of YOLO for this usage unnecessary as well.

The first model, which was trained for all 30 classes, delivered worse results, then the linear regression serving as baseline. So this model should not be used, since linear regression is a simpler solution, which delivers better results.

The second model, which was trained only for the eleven classes known by YOLO from this dataset, is proven to be just as good as linear regression, so the usage of this mode could be spared too, simply by using linear regression instead.

The third model is trained only for eleven classes too, but thanks to the change in the data preparation, this model is already better, then linear regression.

The fourth model, which looks for consecutive hand annotated segments among the annotations was proven to be the best one for this particular scenario, where we have this kind of model and where we need to have 5 timestamps in order to predict 3 timestamps ahead in the future. Since the data preparation script is in this case strictest, meaning that only the hand annotated parts are used for training, which represent the reality obviously as precisely, as closely as possible, it is logical, that this model is the most successful one. Naturally, this model is better than the baseline too, so this could have some real life use too.

The hardware utilisation results of the final system are much less than the results of the standalone YOLOv3 system. This is mainly due to the more simple way of functioning, namely YOLOv3 is running object detection in three threads constantly, in order to achieve such speed as it does. By eliminating the need for these threads, the hardware needs drop quite much. The skipping of processing of certain number (3) of frames supports the drop in hardware utilisation.

9 Future work

In the nearby future, my solution for this particular problem could be developed in many ways.

In order to achieve better result, YOLOv3 could be trained for this particular dataset. The training process was indeed begun, but its results could not been evaluated, so I used the original weights, that can be downloaded from the official webpage of YOLO.

Although the current results are already better, than the baseline, there is quite much room for development. The number of earlier timestamps needed, and the prediction distance are both free variables, so many experiments could be done in this aspect too.

Otherwise, I think that the LSTM architecture could be changed too, something new could be tried out, it might result in better performance. The change of the architecture would be necessary anyway, since as we could see in the results, the current architecture is not capable of handling all of the thirty classes.

Another interesting point would be the investigation of the usage of this kind of solution with object segmentation algorithms. Those kind of algorithms are getting better and better every day, and since they deliver a more precise position of objects, it would be very much desirable to extend the solution to this field.

10 Summary

The main purpose of this thesis was to develop a solution that can predict the movement of different objects. In order to achieve this, I investigated a various number of object detection algorithms, so that I can choose the most fitting one for the purpose.

After the evaluation of the two best alternatives, YOLOv3 and SSD, on the dataset, which was surprisingly difficult to find, I decided to go with YOLOv3.

The next step was the experimentation with different time-series LSTM models and data preparation methods. I managed to find a model that is better than the baseline, linear regression solution, though thanks to the many free variables, a lot of possibilities (number of timestamps, prediction distance, architecture) are left untested.

In the end a fully operational system was developed, where YOLOv3 is loaded into the memory and prepares constantly the input data for the prediction network. After we have enough input data for the prediction, the LSTM model is predicting the future position of the already recognised objects, so that YOLO should not be run for every frame. When the predicted time surpasses, YOLO can continue recognising the object again, so that the next prediction could commence.

The results of this kind of object movement prediction are promising, though there is much room for further improvement.

References

[1] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 580-587).

[2] Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. International journal of computer vision, 104(2), 154-171.

[3] Girshick, R. (2015). Fast r-cnn. In Proceedings of the IEEE international conference on computer vision (pp. 1440-1448).

[4] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems(pp. 91-99).

[5] Dai, J., Li, Y., He, K., & Sun, J. (2016). R-fcn: Object detection via region-based fully convolutional networks. In Advances in neural information processing systems (pp. 379-387).

[6] R-FCN: https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9 (Date of visit: 2018.05.20)

[7] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, October). Ssd: Single shot multibox detector. In European conference on computer vision (pp. 21-37). Springer, Cham.

[8] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788).

[9] Redmon, J., & Farhadi, A. (2017). YOLO9000: better, faster, stronger. arXiv preprint.

[10] Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767.

[11] YOLO: https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad (Date of visit: 2018.05.20)

[12] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[13] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. IEEE transactions on neural networks and learning systems, 28(10), 2222-2232.

[14] ILSVRC: http://image-net.org/challenges/LSVRC/2015/ (Date of visit: 2018.05.20)

[15] Salton, G., & McGill, M. J. (1986). Introduction to modern information retrieval.

[16] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., & Zisserman, A. (2010). The pascal visual object classes (voc) challenge. International journal of computer vision, 88(2), 303-338.

[17] Laptev, N., Yosinski, J., Li, L. E., & Smyl, S. (2017). Time-series extreme event forecasting with neural networks at uber. In International Conference on Machine Learning (No. 34, pp. 1-5).