FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Marek Dobranský

Object detection for video surveillance using the SSD approach

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Jakub Lokoč, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial inteligence

Prague 2019

I would like to express my gratitude to RNDr. Jakub Lokoč, Ph.D., the supervisor of this thesis, for his patient guidance, feedback and all advice he has given me.

I also want to thank my family and girlfriend for their continued support and encouragement during my studies and especially during the time spent working on this thesis.

Title: Object detection for video surveillance using the SSD approach

Author: Marek Dobranský

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Lokoč, Ph.D., Department of Software Engineering

Abstract: The surveillance cameras serve various purposes ranging from security to traffic monitoring and marketing. However, with the increasing quantity of utilized cameras, manual video monitoring has become too laborious. In recent years, a lot of development in artificial intelligence has been focused on processing the video data automatically and then outputting the desired notifications and statistics. This thesis studies the state-of-the-art deep learning models for object detection in a surveillance video and takes an in-depth look at SSD architecture. We aim to enhance the performance of SSD by updating its underlying feature extraction network. We propose to replace the initially used VGG model by a selection of modern ResNet, Xception and NASNet classification networks. The experiments show that the ResNet50 model offers the best trade-off between speed and precision, while significantly outperforming VGG. With a series of modifications, we improved the Xception model to match the ResNet performance. On top of the architecture-based improvements, we analyze the relationship between SSD and a number of detected classes and their selection. We also designed and implemented a new detector with the use of temporal context provided by the video frames. This detector delivers enhanced precision while meeting real-time requirements.

Keywords: object detection, video surveillance, deep neural networks, SSD architecture

Contents

Introduction

In recent years, security cameras have become widely used for indoor and outdoor surveillance. Covering more and more public space in cities, the cameras start to serve various smart city purposes ranging from security to traffic monitoring and marketing. However, with the increasing quantity of utilized cameras and recorded streams, manual video monitoring and analysis becomes too laborious. Hence, developments in artificial intelligence and the broader availability of computing power are necessary to benefit from such rich data sources. Instead of monitoring by people, the goal is to train an effective and efficient artificial intelligence model to process the video data automatically and then output the desired notifications and statistics.

The automatization in video surveillance has evolved rapidly in the last decades. Not so long ago, state-of-the-art detection relied on motion detection based on the principle of frame difference. Frame difference methods either compare successive frames with each other or use background subtraction to detect changes in the video. This is a fast and simple method for detecting moving objects and can be nowadays commonly found embedded directly in the security cameras. However, the frame difference approach is prone to false detections as it is sensitive to every movement regardless of the object type. Hence, it is no longer used as a standalone detector.

An object detector with classification capability can not only enhance the detection by providing the means for class-based filtering but also removes a significant deficiency of frame differencing methods, detection of stationary objects. Modern detectors gain this capability by relying on feature vectors for a class description. The first such algorithm with competitive real-time results was a face detector by Viola et al.. A few years later, another significant step in detection came from Dalal and Triggs using the histogram of oriented gradients for human detection. The last breakthrough in object detection that has become the dominant approach of the current decade is the application of deep learning. In the ImageNet Large Scale Visual Recognition Challenge, deep learning approaches have been winning consistently since 2012 and surpassed human performance in 2015.

Goals

The goal of this thesis is to review state-of-the-art deep learning models for object detection and implement a real-time detection model based on the SSD approach. The objective is not only to re-implement a selected model but to improve that model for purposes of surveillance while using the information gathered from the review. This thesis aims to test proposed improvements experimentally.

On top of the optimization of the SSD, we aspire to design and implement a second model, a real-time video detector with enhanced precision by exploiting the temporal information provided by the video.

The focus of this thesis is limited to the detection component of a more extensive video detection pipeline. We do not concern ourselves with optimizing pre- and post-processing operations. Although many interesting tasks are based on object detection, e.g., tracking and re-identification, they are not in the scope of this thesis.

Thesis Structure

Chapter 1 defines the metrics needed for the evaluation of classification and object detection models, and notation used throughout this thesis.

In chapter 2, we present a brief overview of the history of convolutional neural networks used for image classification and region-based object detectors.

Chapter 3 is dedicated to providing an in-depth review of real-time detectors. We began by explaining the transition from region-based detectors to one-stage detectors and then described two such networks, namely SSD and YOLO. The second part of the chapter is focused on reviewing two methods for video detection with the use of temporal information, i.e., Tube-CNN and Temporal SSD.

We present our contributions in chapter 4. We propose a set of improvements of SSD aimed at real-time detection for video surveillance and test them experimentally. Our contributions include a comparison of SSD implemented on a multitude of base networks (ResNet, Xception, NASNet), a test of the relationship between detector performance and a number of detected classes, and an improved version of Xception modified to suit the needs of SSD detector. Our final contribution is an extension of SSD detector by the addition of three-dimensional convolutions using the temporal dimension, called a Single Shot Detector with Temporal Convolution (SSDTC).

Chapter 5 provides details on the methodologies used in this work. It also presents supplementary results acquired during the experiments.

1. Preliminaries

In the last few years, there has been a rapid development in the field of deep learning application to computer vision and object detection. With each model being an iterative improvement on its predecessors, we review a broad spectrum of neural networks to properly understand the structure and the reasoning behind current state-of-the-art models. This amount of information compels us to leave an explanation of the inner workings of neural networks out of the scope of this thesis. We, therefore, expect prior knowledge from the reader in the area of deep learning, namely in understanding standard modules such as fully connected, convolutional and pooling layers, activation functions, soft-max classifier, batch normalization and principle behind backpropagation and loss functions. All required knowledge can be obtained from the book by Goodfellow et al. [11].

A large part of this thesis is focused on describing and comparing classification and object detection models, in order to do so, this chapter is dedicated to describing used evaluation metrics and notations.

1.1 Evaluation metrics

In order to evaluate and compare multiple approaches, we require clearly defined evaluation metrics. A state-of-the-art model presented without full specification of used evaluation metrics makes comparing multiple results impossible. Thankfully, there are competitions and challenges with precisely defined rules that are often used to make such comparisons.

The *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC)[1] is most often used to benchmark classification. Object detection is commonly evaluated in two challenges: the *PASCAL Visual Object Classes Challenge*[2], and the *COCO Object Detection Task*[3]. Each challenge also provides a public dataset, *ImageNet*, *PASCAL VOC*, and *COCO* respectively.

In this section, we define metrics used to evaluate those challenges, but also other, not considered qualities. Most notably, none of the mentioned challenges is evaluated based on the speed of the model. Since our work is focused on real-time video analysis, we are interested in finding a balance between accuracy and the number of images processed per second (fps). One more factor to consider would be the physical size of the model, usually

[1]`http://www.image-net.org/challenges/LSVRC/`
[2]`http://host.robots.ox.ac.uk/pascal/VOC/`
[3]`http://cocodataset.org/`

represented by the number of parameters and directly impacting the amount of needed memory.

1.1.1 Classification

The most common and intuitive evaluation metric for classification problems is the ratio of correctly classified samples. This ratio is referred to as a *classification accuracy*. However, a complement to the accuracy, *top-1 error*, is also often used. In ILSVRC, alongside top-1 error, a *top-5 error* is used as another major criterion. The top-5 error represents the fraction of test samples in which the correct label does not appear in the top 5 predicted results.

1.1.2 Object Detection

Evaluating a localization and classification of multiple objects in an image is a more complex task than classification, mainly because there is no simple one-to-one mapping between ground-truths and predictions. A ground-truth data are a set of N boxes with labels, and detector generates a set of M boxes with labels and class confidence values.

Because predicted boxes do not perfectly match ground-truths, a matching algorithm is needed to decide whether a prediction is true positive or false positive. Matching is usually done by computing *intersection over union* (IoU) value for each pair of ground-truth and predicted boxes, and then selecting positive detections based on predetermined threshold.

$$\mathrm{IoU} = \frac{\mathrm{Area}(\mathrm{Prediction} \cap \mathrm{Ground\ truth})}{\mathrm{Area}(\mathrm{Prediction} \cup \mathrm{Ground\ truth})}$$

With predictions sorted into true positives (TP), false positives (FP) and false negatives (FN) (no predictions matching a ground-truth box) we are able to calculate *precision* and *recall*.

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$\mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Now we can define the first metrics used for object detection. Note that all following metrics depend on precision and recall, and therefore depend on the IoU threshold.

Precision-Recall Curve (PRC)

PRC is a plot showing the relationship between values of precision and recall. It is plotted separately for each class and reveals how a change in confidence influences the precision and recall values.

Every point on PRC represents a chosen confidence threshold used for determining positive predictions for a given class. The curve does not show this threshold. Instead, it shows precision and recall received by applying this threshold.

An object detector of a particular class can be considered reliable if its precision stays high while recall increases, which means that predictions with lower confidence score can be considered good predictions.

Average Precision (AP)

Comparing curves is not an easy task, particularly if they cross each other frequently, as it often happens with PR curves. However, we can use the area under the PR curve as numerical metrics, called *average precision.* It is usually calculated by interpolation, either on all data points or a small number of equally spaced points (earlier versions of PASCAL VOC Challenge use 11).

Interpolation equation for all points:

$$\sum_{r=0}^{1}(r_{n+1} - r_n)p_{interp}(r_{n+1})$$

with

$$p_{interp}(r_{n+1}) = \max_{\tilde{r} \geq r_{n+1}} p(\tilde{r})$$

where $p(\tilde{r})$ is precision at recall $\tilde{r}$.

Mean Average Precision (mAP)

Comparing two models class by class is impractical, especially with the classifiers for hundreds of classes. Therefore, the most often used metrics for object detectors is a *mean average precision.* As the name suggests, it is a mean of AP across all classes.

Most common notation, mAP@[0.5] means mAP with IoU threshold 0.5. The mAP can also be averaged over multiple IoU thresholds, mAP@[.5, .95] denotes the average mAP from IoU 0.5 to 0.95 usually with step 0.05.

We use mAP@[0.5] for all evaluations in this thesis.

1.1.3 Inference Time

Inference time is a significant factor to consider for the model in real-time application use. With the state-of-the-art models, images are often processed under a second, often in a few milliseconds. Such small numbers can be hard to comprehend. Therefore a more intuitive metric is used. That being the number of processed *frames per second* (fps). However, unlike precision metrics, the fps values are heavily dependant on hardware, software framework, batch size and amount of pre- and post-processing included in the measurement. Hence, only the measurements performed in the identical hardware and software environment are suitable for comparison.

1.2 Notation and Convention

To avoid any confusion, we define the standard notation used throughout this thesis. We tried to follow the general convention, but there can be some variance compared to other works.

Data

To represent the size of the multidimensional data (tensors), we use bracket notation with dimensional sizes separated by $\times$ sign (not to be confused with multiplication).

A two-dimensional matrix represented as $[\mathbf{a}\times\mathbf{b}]$, is most often used to represent spatial dimensions of an image or a feature map.

In three dimensional data, denoted as $[\mathbf{a}\times\mathbf{b}\times\mathbf{c}]$, the first two numbers represent the spatial dimension and c represents the number of channels, e.g., standard RGB image has three channels. For convenience, we may use $[\mathbf{a}\times\mathbf{c}]$ notation if a and b are equal, while explicitly stating the fact.

The next dimension used in neural networks is the batch size n. We will be adding this dimension as the last one to our notation, $[\mathbf{a}\times\mathbf{b}\times\mathbf{c}\times\mathbf{n}]$.

We will also be using three-dimensional convolutional layers, that produce five-dimensional tensors. However, we will not add this dimension d to the end of the list, but instead, keep channels and the batch size at the end: $[\mathbf{a}\times\mathbf{b}\times\mathbf{d}\times\mathbf{c}\times\mathbf{n}]$.

We sorted the values to allow for unambiguous reference of data without listing every dimension, provided the context of two (2d) or three-dimensional (3d) convolution. For example, data passing between two-dimensional convolutional layers is in the form of four-dimensional tensor, but we will often use only spatial dimensions or spatial and channel dimension to describe given tensor.

Convolution

Since convolutional layers are usually chained together, and always work with the data with *channel* dimension, we do not need to state the channel depth of the input data explicitly. We know that if previous convolution outputs data with the size of [a×b×c], and we need to apply $k{\times}k$ kernel, the actual size of the kernel is [k×k×c]. However, we must expressly state the number of channels outputted from the convolution to receive the full information. This number determines how many times we perform the convolution operation with different kernels.

The output of a convolutional layer is commonly called a *feature map*. This output can be either viewed as a set of c two-dimensional maps or a single three-dimensional multi-channel feature map. In this thesis, we are not interested in singular feature maps outputted by the convolutional layer but rather look at them as a unified three-dimensional map with spatial and channel dimensions.

Unless stated otherwise, the default parameters for the convolutional layer is stride of one, dilation of one, and sufficient padding to allow the spatial dimensions of the output feature map to stay equal to the input ones. The effects of these parameters are illustrated on fig. 1.1.

- **Conv. k×k×c** : 2d convolutional layer with kernel size $k{\times}k$ and output channel depth of c, while using default values for padding, stride and dilation.
- **Conv. k×k×c /s P:p D:d** : 2d convolutional layer with stride s, padding p and dilation d.
- **Conv3d. k×k×k$_3$×c P:0** : 3d convolutional layer with kernel spatial size $k{\times}k$ and temporal size k_3, and output channel depth of c and zero padding in all dimensions.
- **Conv3d. k×k×k$_3$×c P:p,p,p$_3$** : 3d convolutional layer with spatial dimensions padded by p elements and third dimension padded by p_3 elements.

Other Layers

- **Max-pool. k×k /s P:p** : maximum pooling with kernel size $k{\times}k$, stride s and padding p. Default value for stride and padding is 1.
- **Fully connected / FC - N** : fully connected layer with N neurons

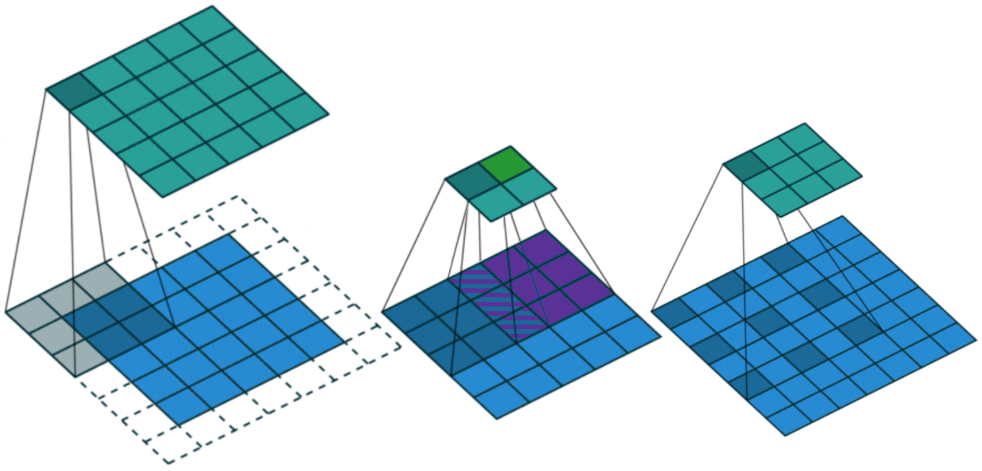


Figure 1.1: Effect of padding, stride and dilation on two-dimensional convolutional layer (left to right). Blue maps represent inputs, and cyan maps outputs. Images adopted from Dumoulin and Visin [8].

2. Neural Networks for Image Processing

In recent years, there has been an upsurge in the use of neural networks. We can partially attribute it to the evolution of hardware allowing for the implementation of network models with multiple layers. Deep neural networks (DNN) are now finding their use in many applications, e.g., classification, time series prediction, and optimization tasks, and are replacing many older machine learning methods. Some of the benefits of neural networks include their ability to find and learn complex non-linear relationships in provided data and subsequent generalization to unseen data. However, some applications do not allow for so-called 'black box' models and require logical reasoning behind the decisions.

Image processing is one of the fields where DNNs are heavily utilized and outperform traditional machine learning approaches with big margins. Uses of DNNs for image processing vary from classification and object detection to auto-encoders for noise removal, and generative networks. A surge in DNN based image processing started in the 2010s with an implementation of a multi-layer convolutional network trained on GPU [6].

In this chapter, we take a closer look at how are DNNs applied in object detection. The goal of such a detector is to localize and classify multiple objects of interest in the given image. There are multiple ways of localization, such as semantic segmentation, which categorizes individual pixels, or key-point and skeleton detections. However, we are interested in a more straightforward, axis-aligned bounding box (bbox) predictions. Each of the predicted bounding boxes needs a corresponding class prediction.

Considering the importance of classification in object detection, we dedicate a significant portion of this chapter to describing multiple classification network models. Many detection models directly utilize or are inspired by those classification networks.

2.1 Classification Networks

A fundamental building block for a modern state-of-the-art image classification network is a convolutional layer. Accordingly, we call this type of networks a convolutional neural networks (CNN) [11, ch. 9]. CNN based classifier is a network that given the input image, extracts a feature map from this image and then applies classification layers to produce a confidence score for each possible class. Usually, the soft-max function is applied to the confidence score to get a probability distribution.

We use this section to take a walk through the history of classification CNNs and outline some of the most influential models. Some of those models are still used, and others are responsible for inspiring the next generations of even better networks.

2.1.1 AlexNet (2012)

AlexNet designed by Krizhevsky et al. [16] is the first CNN that won the ILSVRC challenge over traditional computer vision and machine learning approaches. It created a foundation on which today's state-of-the-art models are built, and set a new standard for image recognition. AlexNet is created from a stack of five convolutional layers interleaved by max-pooling layers, followed by two fully connected layers and a softmax layer. A local response normalization [16, section 3.3] is applied after first two convolutional layers. AlexNet also popularized the use of ReLU non-linearity in CNNs.

2.1.2 VGG (2014)

The network architecture, mostly known as VGG, by Simonyan and Zisserman [26], is built on the deep CNN concept behind AlexNet. It managed to prove the feasibility of even deeper network utilizing small convolution filters.

Each of the VGG's convolutional filters employs a 3×3 kernel with the depth of the feature map gradually increasing through the network. The convolutions are followed by three fully connected layers and softmax layer, see fig. 2.1.

Multiple versions of the VGG architecture can be constructed, depending on the number of convolutional layers. The most popular is the 16 layer version, titled VGG16.

Today the VGG network is considered to be a general architecture for a classification network due to its linear architecture with a decreasing area of the features, and an increasing number of channels.

2.1.3 Inception (2014)

Predating architectures suggest that increasing the number of layers and layer size, leads to better precision. Szegedy et al. introduced Inception v1 [27], also known as *GoogLeNet*, with the goal of increasing precision while improving utilization of computing resources.

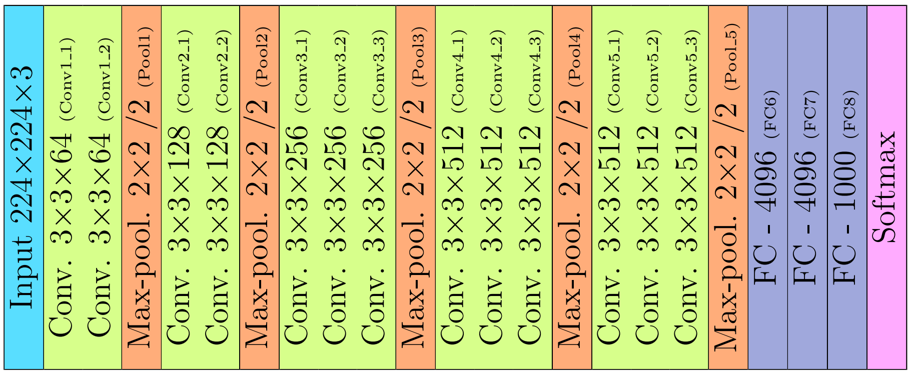


Figure 2.1: Architecture of VGG network version D, commonly called VGG-16. Other versions in [26, table 1].

Although stacking more convolutional layers improves the accuracy, an increasing computational cost of those layers quickly overpowers the benefits. To avoid the aforementioned cost, Inception introduces the concept of sparsity in convolutional layers. The sparsity is achieved by using *inception modules* that approximate a sparse structure by utilizing multiple convolutions with different kernel sizes and concatenating the outputs together (see fig. 2.2).

To reduce the computational cost further, each convolution is preceded with additional 1×1 convolution, used for a dimensionality reduction. An alternate path in the *inception module* is provided by max-pooling operation and concatenating it to the output.

Inception begins with a sequence of convolution, pooling, and local response normalization operations. This 'stem' is followed by a chain of nine *inception modules*, topped by a fully connected soft-max classifier. Two auxiliary classifiers are added to intermediate layers of the network to help propagate gradients and provide regularization during the training.

Inception v2, v3 (2015)

A set of improvements to the Inception network is introduced in later versions of the network. Most notably a factorization of convolution layers in Inception v2 and v3, Szegedy et al. [28]. Factorization replaces larger convolutions with a network of many smaller ones. They found this method very effective, e.g., replacing a 5×5 convolution with two layers of 3×3 results in a relative gain of 28% and replacing 3×3 layer with 3×1 and subsequent 1×3 layer is 33% cheaper.

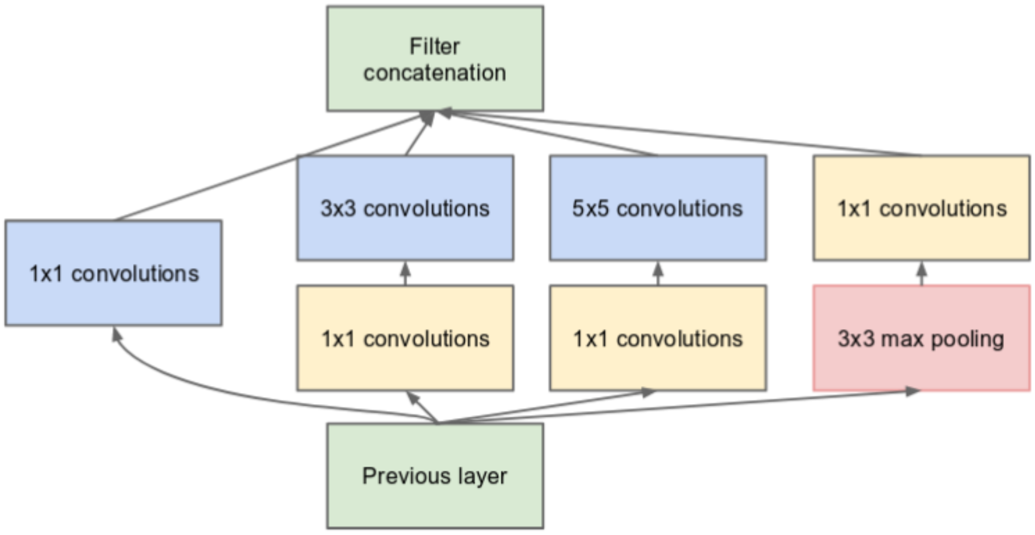


Figure 2.2: Inception module, picture from [27, figure 2].

2.1.4 ResNet (2015)

A trend of adding more layers to CNNs to achieve better accuracy has pushed the limit towards networks with hundred or more layers. Theoretically, adding more layers to a model should produce equal or better results, based on the fact that shallow model is the subspace of the deeper one. Therefore, additional layers can learn to forward the data. In practice, however, observations suggest that this is not the case, and very deep networks can experience eventual degradation. A solution to this problem was proposed by Kaiming He and Sun [14] in the ResNet architecture by directly introducing identity functions to the network.

Basic principles of ResNet are directly inspired by the VGG. Most of the convolutional layers use 3×3 filters and follow two simple rules: keep the number of filters the same, unless changing the output size and double the filters if the feature size is halved.

Newly introduced residual connections bypass each pair of the convolutional layers and forms *residual blocks*. These connections can be an identity function or, if the input and output feature map depths of the residual block do not match, a 1×1 convolution can be used.

We can see the high level architecture of this model in fig. 2.3 (left). Each of four *Layers* represents a sequence of multiple residual blocks, exact numbers of blocks can be found in [14, table 1]. Previously described *residual block* with two convolutional layers, known as *Basic block*, is used for smaller ResNet models (ResNet18, ResNet34). Deeper ResNet models (ResNet50,

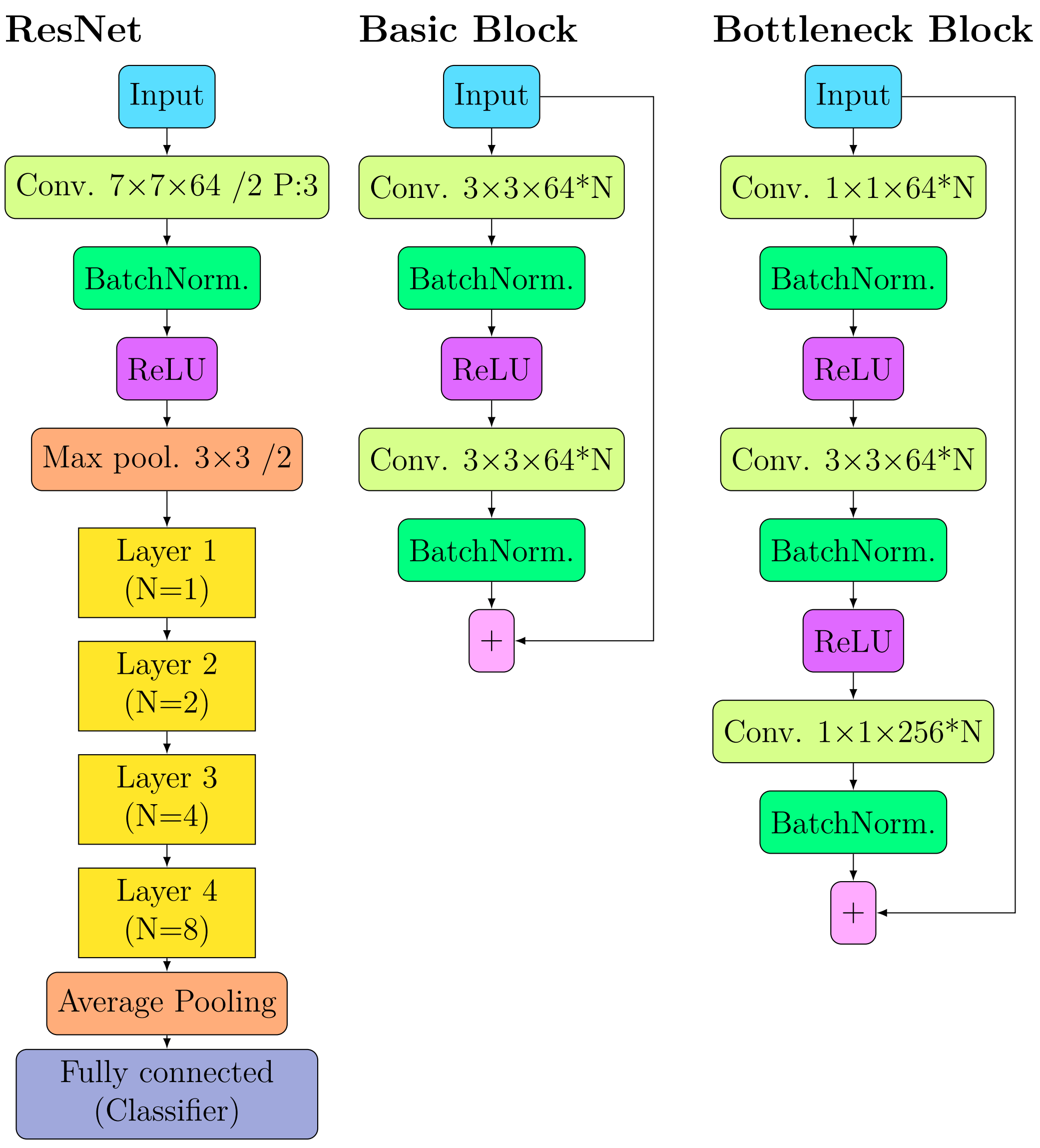


Figure 2.3: Architecture of the ResNet network and residual blocks. Each of the four *Layers* is created by stacking multiple residual blocks.

ResNet101, ResNet152) use the *Bottleneck block* with three convolutional layers. In *Bottleneck block*, the 1×1 layers are responsible for reducing and then restoring dimensions, allowing for faster 3×3 layer with reduced input and output dimensions. Thanks to the efficient architecture, the 152-layer ResNet has lower computational complexity than the 16-layer VGG network.

2.1.5 Xception (2017)

Xception architecture by Chollet [5], is heavily inspired by previous architectures, mainly Inception and ResNet. It is built on the hypothesis claiming: "the mapping of cross-channel correlations and spatial correlations in the feature maps of convolutional neural networks can be entirely decoupled." This hypothesis expands upon the hypothesis underlying Inception architectures. Therefore the name 'Extreme Inception.'

The hypothesis is realized in the form of *depthwise separable convolution* layers (shortly separable convolution). Depthwise separable convolution consists of two steps: a *depthwise convolution* and *pointwise convolution.* A depthwise convolution is a convolution performed independently over each channel, i.e., a convolution without changing the number of channels. The second step is a pointwise convolution that uses 1×1 kernel to map the output of depthwise convolution into new channel space.

The model is formed by linearly stacking separable convolution layers with the addition of residual connections as seen on fig. 2.4. Convolutional layers, non-linearity, and poolings are structured into residual blocks similarly to ResNet architecture.

2.1.6 NASNet (2017)

This architecture stands out from others mentioned because Zoph et al. [34] used a machine learning algorithm to design the network. It is a result of a *AutoML*[1] project. Unlike manually designing the network by trial and error, AutoML searches the space of all possible models, e.g., using reinforcement learning and evolutionary algorithms. This approach is limited by the computational cost and therefore limited to small datasets.

NASNet is a result of taking an architecture designed for small *CIFAR-10*[2] dataset by AutoML and using it to create larger model for ImageNet dataset. The model is composed of two types of learned cells, a *Normal Cell*

[1]`https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html`

[2]`https://www.cs.toronto.edu/~kriz/cifar.html`

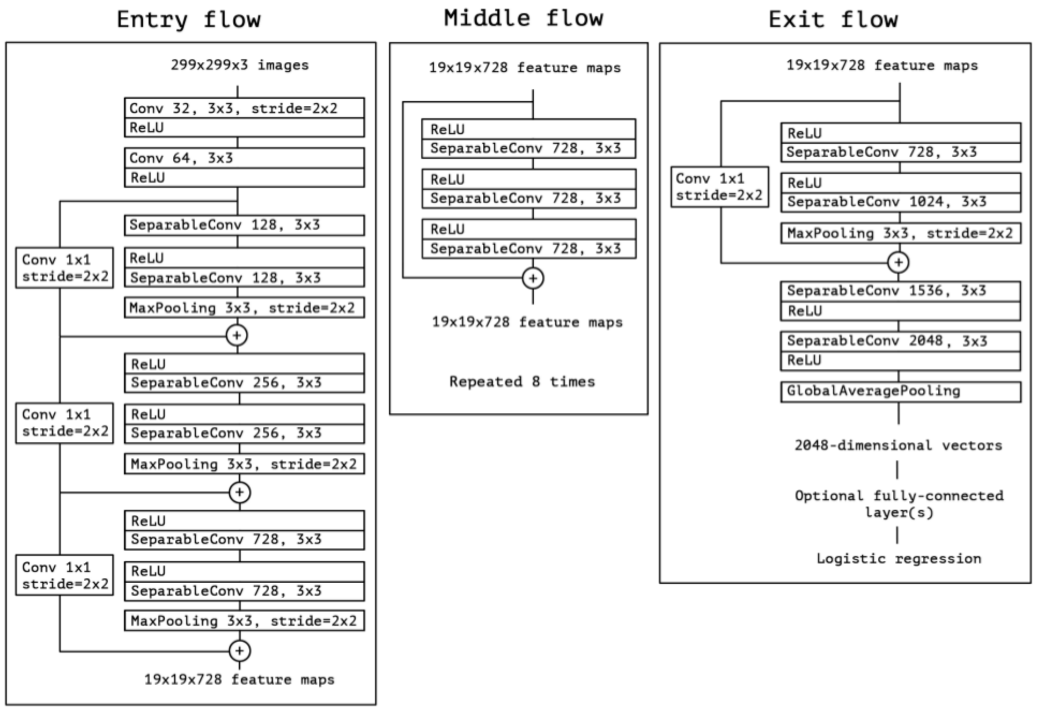


Figure 2.4: Structure of *Xception* architecture. Taken from [5, fig. 5].

and *Reduction Cell* (see fig. 2.5). A general structure of the network is then created by alternating a Reduction Cell and *N* Normal Cells.

2.1.7 Classifier Comparison

At the beginning of this chapter, we mentioned that the classification networks are often compared based on performance on the ImageNet dataset. Newer models, like Xception, NASNet, and modifications of ResNet reach excellent accuracy. However, there is a large discrepancy in their performance considering inference speed. In fig. 2.6 we provide an overview of fps–accuracy relationship taken from an independent benchmark by Bianco et al. [2]. Although the experiment was performed with batch size 1, we expect a universal increase of fps with a bigger batch and only small changes to the relative performance of different models. We can observe a clear trade-off between speed and accuracy for the classification task.

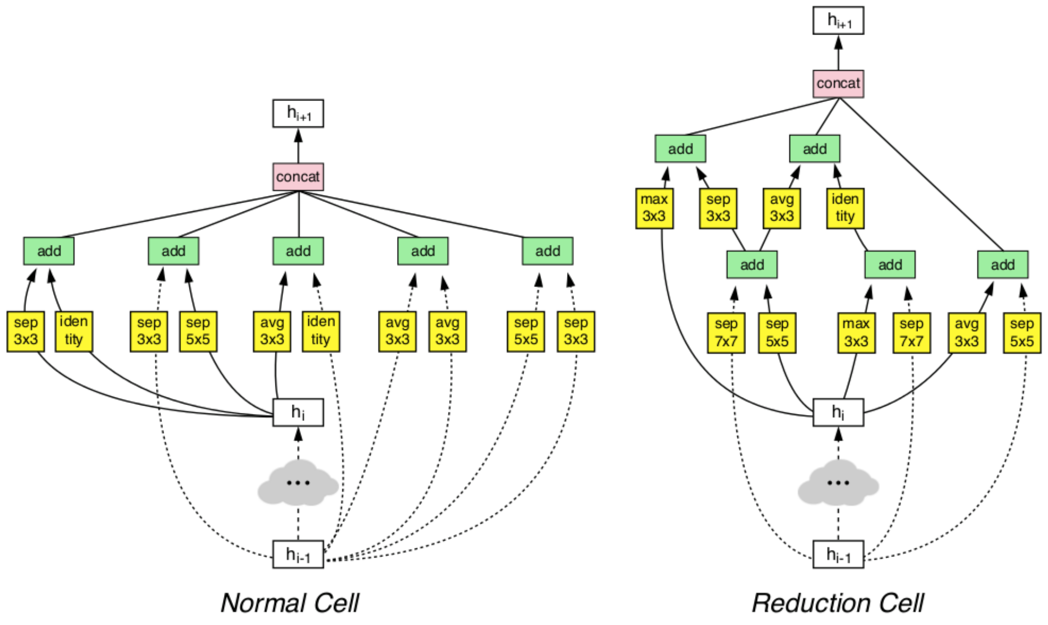


Figure 2.5: Modules used in *NASNet-A*, designed by AutoML. Image from `ai.googleblog.com/2017/11/automl-for-large-scale-image`.

2.2 Detection Networks

The goal of the detection network is to localize all objects of chosen categories. There is a simple logical step from classification task to detection. It is to classify selected regions in the image as one of the classes or as a background. A classification applied to every possible box in the image would undoubtedly produce great detection results but at the extreme computational cost. This section describes a family of algorithms based on this simple idea while managing limited computational resources.

We will see that this family of so-called *region based* approaches can reach state-of-the-art results, with increasing efficiency by each generation. Considering precision, Faster R-CNN is still used as one of the most reliable detectors. However, although it can process a few frames per second, we do not consider it to be a truly real-time detector and applicable for demanding tasks, such as video surveillance. We devote section 3.2 to detection networks performing in real-time constraints.

2.2.1 R-CNN (2014)

Region-based Convolutional Network (R-CNN) by Girshick et al. [10] is the first member of the family of region-based detection models. The foundation

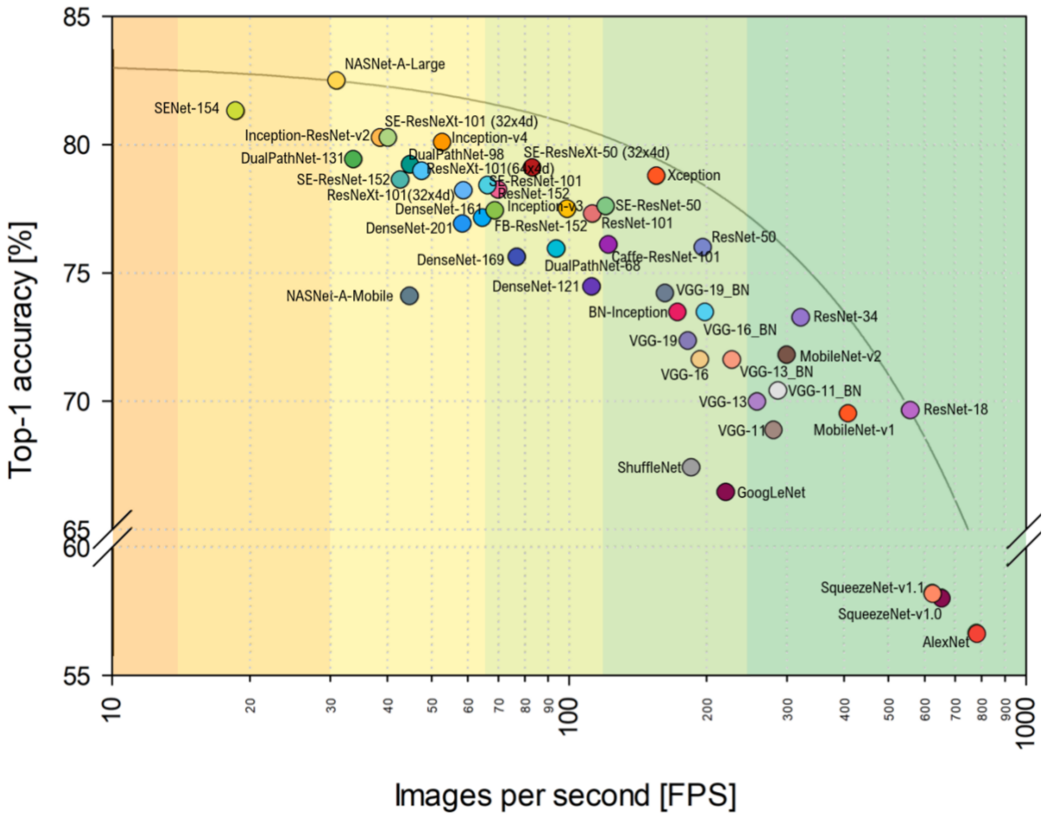


Figure 2.6: Benchmark of state-of-the-art classification deep neural networks on *ILSVRC* dataset. Performed on NVIDIA Titan X GPU with batch size 1. Taken from [2, fig. 3]

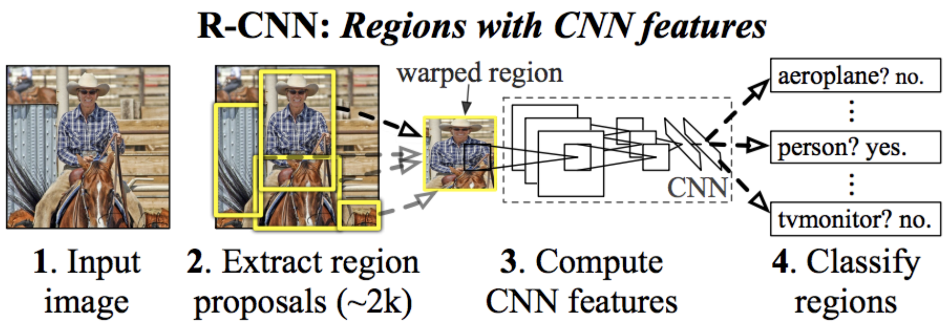


Figure 2.7: R-CNN architecture. Taken from [10, fig. 1].

idea is simple: select regions in the picture and classify each region. This approach leads to a combination of three modules: region proposal algorithm, feature extraction using CNNs on those regions and subsequent classification.

A naive approach would use a sliding window and classify each cutout of the image. However, examining all the windows for different sizes and aspect ratios of possible objects would be extremely slow. R-CNN solves this problem by applying a region proposal algorithm that selects about 2000 most likely locations of objects. Regions are selected using the *Selective search* (SS) [30] algorithm and serve as candidates for bbox predictions. In addition, bounding box regression can be trained to improve bbox prediction accuracy.

Each region is processed separately by a CNN into a feature map (the original architecture uses Alexnet, but any classification network can be substituted). Finally, each feature map can be scored. R-CNN uses class specific linear support-vector machines instead of a soft-max classification provided by CNNs. Figure 2.7 illustrates the architecture.

2.2.2 Fast R-CNN (2015)

Even though R-CNN was a major step in the right direction, its performance is far from real-time. Girshick [9] introduces Fast R-CNN with a series of innovations to its predecessor, aimed at improving speed and accuracy. Provided benchmark on the NVIDIA K40 GPU suggests improvements from 47 seconds per image using R-CNN with VGG16 feature extractor, to 320 milliseconds with Fast R-CNN using the same feature extractor (not including time for SS proposals).

Similarly to R-CNN, this architecture also utilizes region proposal al-

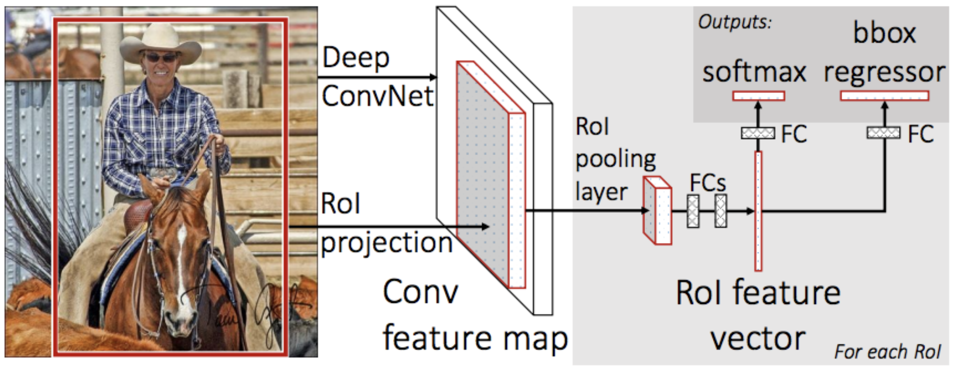


Figure 2.8: Fast R-CNN architecture. Taken from [9, fig. 1]

gorithm and a CNN to produce a feature map. A significant drawback of R-CNN was computing feature map for each region, despite overlaps. Fast R-CNN processes whole input image into a feature map, and then, using a *region of interest (RoI) pooling layers*, extracts a feature vector for each region. All extracted feature vectors are pooled to the same size and are passed through a series of fully connected layers, leading to the softmax classifier and bounding box regression layer. An illustration of this process can be seen on fig. 2.8.

2.2.3 Faster R-CNN (2015)

Faster R-CNN by Ren et al. [24] expands on the Fast R-CNN with the aspiration to achieve real-time performance. Fast R-CNN managed to build a fast feature extraction and subsequent classification usable in a real-time environment. However, it is heavily slowed down by a region proposal SS algorithm. Faster R-CNN expands on the idea of sharing resources and replaces SS with *region proposal network* (RPN). RPN is built on top of a feature map generated by the feature extractor. As suggested, the feature map is shared between RPN and object detection. This approach is able to achieve 5 fps, which can find limited use in a real-time environment. Whole architecture can be seen on fig. 2.9.

Region Proposal Network

RPN is designed as a small, sliding-window network, with negligible cost compared to the feature extractor. It is composed of 3×3 convolutional layer with 512 filters and two sibling 1×1 convolutional layers for region regression and classification. Classification in RPNs determines whether proposed region contains an object or a background (*cls* score). Region regression part of the network is tied to the concept of *anchors* (the anchor is a predefined box centered at a location of sliding-window). Assuming k anchors with different sizes and aspect ratios are used, regression produces 4k relative parameters and classifier 2k scores. Regression parameters are used to modify the position and size of their corresponding anchor. The number of regions is then reduced by eliminating proposals with high overlap using a *non-maximum suppression* (NMS) based on *cls* score [20]. After NMS, top-N ranked proposal regions are used for detection.

To calculate loss and train RPN, a matching between ground-truth boxes and generated region proposals needs to be determined. A positive label is assigned to two kinds of regions: the one with the highest IoU overlap with ground-truth box; regions that have IoU higher than 0.7 with any ground-truth box. A negative label is assigned to a non-positive box if its IoU is lower than 0.3 for all ground-truth boxes. Rest of the boxes do not contribute to training.

4-step alternating training:

1. train RPN with feature extractor initialized by ImageNet pre-trained model
2. train separate Fast R-CNN using proposals generated by RPN from step 1
3. train RPN with feature extractor initialized by weights learned by the detector in step 2, fine-tune only layers unique to RPN
4. using the model from step 3, fine-tune layers unique to Fast R-CNN

Thanks to the modular architecture, R-CNN family networks can exploit any CNN as a feature extractor. Therefore Faster R-CNN can achieve state-of-the-art detection results exploiting the latest advances in classification networks and is often used as a benchmark of their performance.

2.2.4 Mask R-CNN (2017)

Previous R-CNN based architectures used bounding boxes to localize individual objects. He et al. [12] adds a localization based on semantic segmentation, where the goal is to classify each pixel into a category.

Mask R-CNN is built upon Faster R-CNN and combines both bounding box localization and semantic segmentation by predicting segmentation

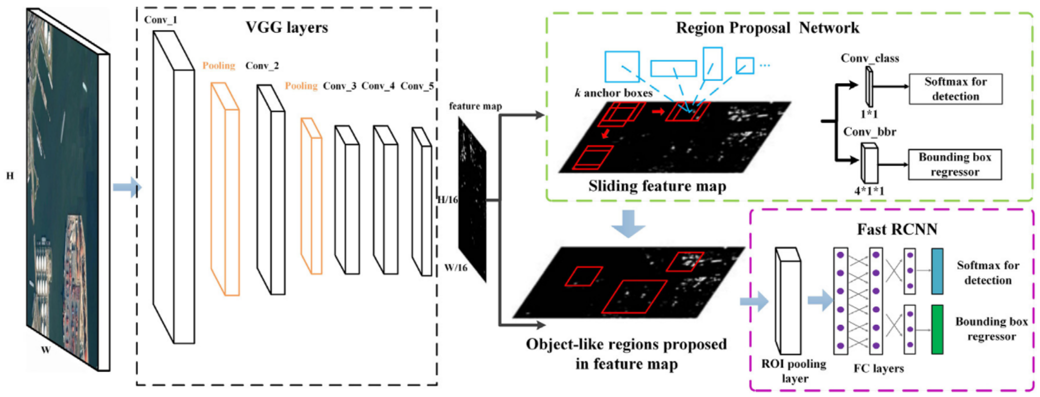


Figure 2.9: The architecture of Faster R-CNN. From `https://researchgate.net/figure/The-architecture-of-Faster-R-CNN_fig2_324903264`.

masks for each RoI. The product of this approach is a bounding box and class for each object, together with a binary segmentation mask. Unlike the semantic segmentation on whole input, applying it on RoIs allows for *instantiated segmentation* where selected pixels correspond to a given instance of a class.

Implementation and architecture are very similar to Faster R-CNN, with two exceptions. One of them is already described, fully convolutional segmentation branch which works in parallel with classification and bounding box regression heads. The other difference is a replacement of RoI pooling layer with *RoI alignment layer*. The problem of RoI pooling for this purpose is quantization of floating-number RoI to discrete feature map grid and consequent imprecision. RoI alignment mitigates this problem by using bi-linear interpolation to compute exact values of features at four sampled locations in each of RoIs locations and aggregating the results. High-level architecture and the RoI alignment layer are visualized on fig. 2.10.

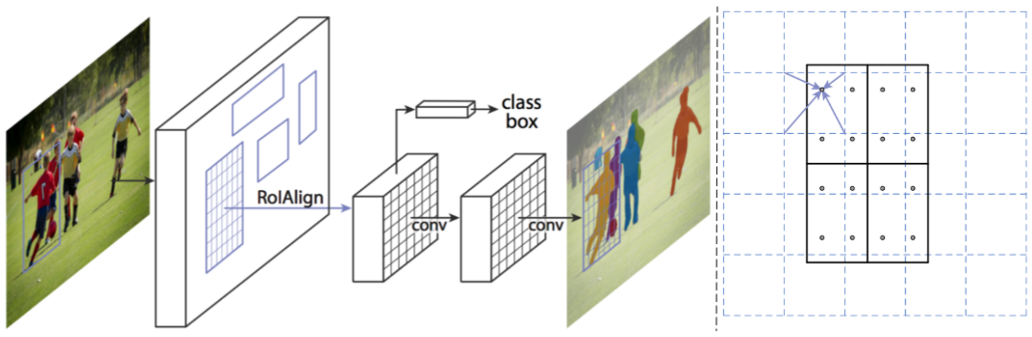


Figure 2.10: Left: The architecture of Mask R-CNN. Right: RoI align, grid represents feature map, solid lines an RoI and the dost are the sampling points. From [12, fig. 1, 3]

3. Related Work

In this chapter, we provide an overview of the methods used for detections in the video. There are two types of approaches that can be used for video detection, a high-speed single frame detector, or a detector designed for a sequence of frames. Although there is no standardized definition of how many fps are real-time, we can consider 25fps as a minimum. However, in practice, we need as much fps as possible not only to keep up with one video stream but to process multiple video streams on one GPU.

Currently, the detectors designed for image detection hold an edge over detectors with temporal information, mostly because available methods for image detection are faster than temporal methods. However single-frame detectors are unstable and require a lot of post-processing for smooth tracking in video.

3.1 One-Stage Detection

From our experience, the step from the region-based networks to one-stage detectors can be hard to comprehend. To ease the transition, this section offers a thorough explanation of the one-stage detector building process based on the knowledge of Faster R-CNN network (section 2.2.3).

At this point, we do not concern ourselves with the optimal model but rather present general ideas. We start by defining the input and desired output of such a detector and then explain how to implement it.

Detection Input

Similarly to Faster R-CNN, we start by processing the input image into a feature map. This process can be done by any feature extraction network, e.g., a classification network up to the fully connected layer. Such a network produces a feature map of [n×n×c] size that will serve as a base for our detector.

Desired Output

To perform a bounding-box detection, we need a parametric representation of both bbox and classification. Since bbox is a rectangle, it can be specified by four parameters: a center position *(X,Y)*, width *W* and height *H*.

These values can be either relative with respect to predefined anchor boxes or absolute values.

For a standard soft-max classification, a confidence score for each of the C classes is required. However, the network also needs the ability to classify the bbox as a background. An additional parameter with background confidence score B can provide the means by being either viewed as a boolean value deciding the presence of an object, or a part of a soft-max classification.

We ended up defining a set of C+5 parameters [c_1, ..., c_C, B, X, Y, W, H]. In the region-based network, this would represent an output for one region. However, there are no given regions in a one-stage detector, or should we say, every position on a feature map is a region. Given the map [n×n×c], we expect to predict described vector of C+5 parameters for [n×n] cells. This gives us an expected tensor of [n×n×(C+5)] size.

Anchor Boxes

In region proposal networks, we saw the use of multiple anchor boxes. Experiments suggest that it is easier for the network to accept a set of proposed boxes and refine them, rather than regress absolute values by itself. It also helps the generalization if the network can associate boxes with certain aspect rations with certain classes.

There are multiple options to include this feature in our model. For each position, we can either predict K anchor box parameters and one set of confidence scores or a separate classification for each box. In the first case, we receive the prediction tensor of [n×n×(K*4+C+1)] shape and in the latter case of [n×n×(K*(C+5))] shape. For simplification, as the details of the implementation are not necessary, let us call this number of parameters P.

Detection

Now that we know what the input and output of the detector are, we can design the network layers to meet the defined criteria. We will look at both fully connected and convolutional options.

Before we start, however, we feel the need to emphasize, that the detector does not have any information about the purpose of output values. It is designed to generate P values for given feature map position, but the information about the association with anchor boxes and classes is provided only during the training process by the correct organization of the ground-truth data and the loss function.

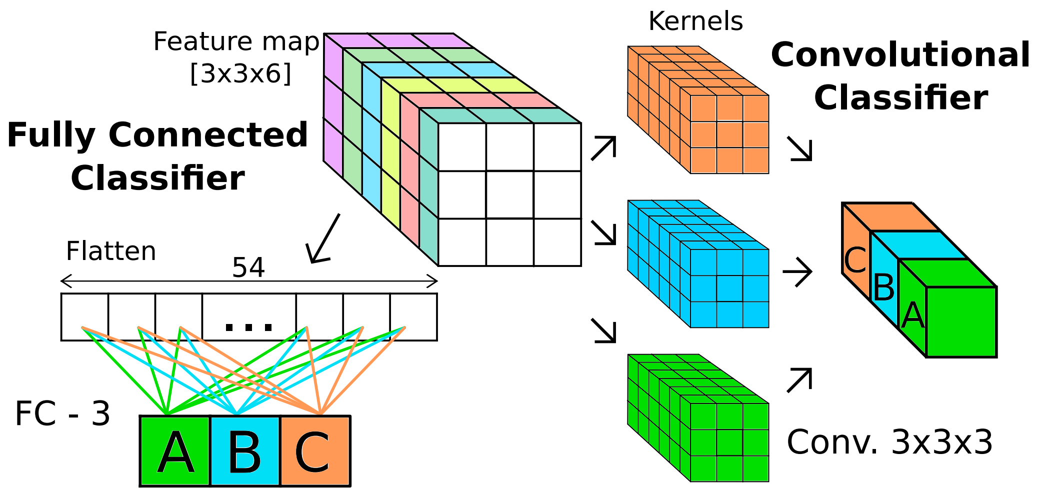


Figure 3.1: A convolutional layer with kernel size equal to feature map size and no padding performs an equivalent operation as fully connected (FC) layer. While the FC layer is parametrized by the number of neurons and produces the output of size [N], convolution is parametrized by the number of output channels and therefore produces [1×1×N] shaped tensor. Kernels are color-coded to match the produced channel.

Fully Connected Detector

The simplest means of getting from [n×n×c] map to [n×n×P] values using a fully connected layer is to connect each input value to each output value. It is a functional but clearly a very inefficient solution that requires training of n^4cP parameters. An improvement can be made using an additional fully connected layer with a smaller number of neurons (b) and reducing the number of trainable parameters to $n^2b(c + P)$.

Notice that this detector is not only unaware of the order of P parameters, but the association of parameters with the feature map position is also not present. As a matter of fact, each predicted parameter is calculated with the information from the entire feature map, and it is only via the training process that the spatial relations are formed.

Convolutional Detector

Before we start with a convolutional detector, we need to demonstrate the concept by showing the possibility of using a convolutional layer as a classifier equivalent to a fully connected layer. In a classification network with C classes, a layer with C neurons is connected to the last feature map with

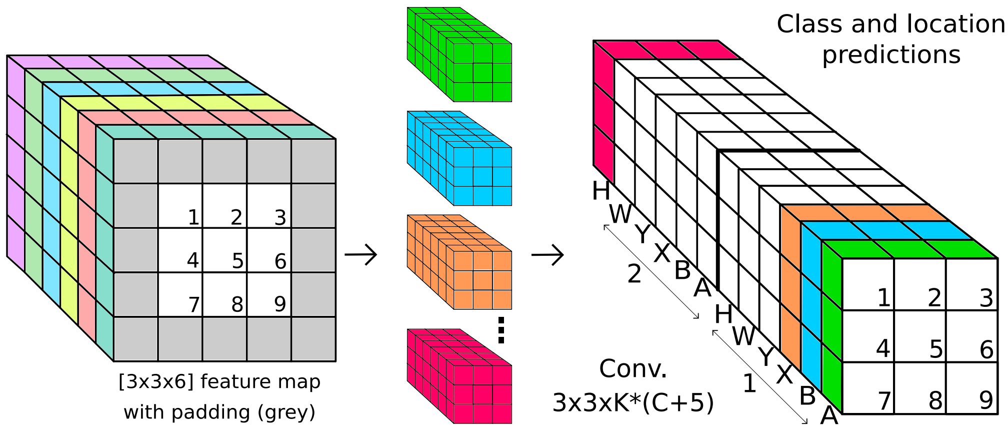


Figure 3.2: A single convolutional layer can generate multiple class and location predictions. For every convolutional window it predicts $K = 2$ classified bounding boxes. Each prediction composes of four location parameters (height H, width W and center position X, Y), predictions for $C = 1$ class (A) and object confidence score (B). Kernels of convolution are color-coded to match the produced channel.

[n×n×c] shape. This layer takes an input of all n^2c values and outputs a class distribution. On the other hand, let us deploy a convolutional layer with a kernel size of $n \times n$ and C output channels. From the definition, we know that the actual size of the kernel includes the channel depth equal to the depth of input data and the output number of channels specify the number of such kernels. Now we can see that the application of one convolutional kernel, is in this case equal to one fully connected neuron. The only difference in both approaches is the shape of input and output data. This comparison is illustrated on fig. 3.1.

Now that we know that the convolution can perform the same tasks as a fully connected layer, no matter whether it is classification or regression, we can change the naive fully connected detector to a naive convolutional detector. Of course, this has no benefits other than proving the point.

However, if we decrease the size of the convolutional filter to $k \times k$, where $k \ll n$, we can use the convolution as a spatially localized detector. The resulting convolutional operation performs detection equivalent to a fully connected detector for each of [n×n] positions on a feature map, using only local information and producing the P parameters for each position. We illustrate the convolutional detector on fig. 3.2.

Thanks to the convolution, we reduced the number of trainable parameters from n^4cP in naive approach to k^2cP, where k is usually a small number, e.g. $k = 3$, and number of multiplications to n^2k^2cP. The other benefit is that convolution performs equivalent detection at each position on the image, where a fully connected detector can acquire some positional biases.

3.2 Real-Time Detectors

In section 2.2 we took a look on a series of two-stage detectors, with separate region proposals and classification. A few years later, a new type of detectors has been developed, with faster one-stage design. One-stage detectors combine classification and bbox regression into a single pass of the network and are able to achieve real-time performance without problems. Also, the trade-off between speed and precision is continually diminishing in favor of one-stage approach. In this section, we show off two popular approaches *You Only Look Once* and *Single Shot MultiBox Detector*.

3.2.1 YOLO: You Only Look Once (2016)

Building on the success of neural network detectors from R-CNN family. Redmon et al. [23] introduced a new approach to object detection. They unify networks for localization and classification into a new single network that predicts both bounding box positions and class probabilities in a single evaluation. This approach also simplifies the training process, as YOLO can be directly trained end-to-end.

Thanks to straightforward single-pass architecture YOLO claims to perform at 45 frames per second on Titan X GPU. Although it has to sacrifice some precision compared to region proposal methods, it out-performs other real-time systems of its time [25].

Detection

Prediction in YOLO works in a grid-based system. It divides the image into $S{\times}S$ grid with each cell responsible for detecting the object centered in that cell. Each cell produces predictions for B bboxes and one set of class confidence predictions.

Bbox prediction is composed of four positional parameters and confidence score. Center coordinates relate to the grid cell while width and height are represented relative to the whole image. Confidence score reflects IoU with ground-truth box. Class confidence prediction represents the conditional probability of a said class, given the presence of the object in that cell.

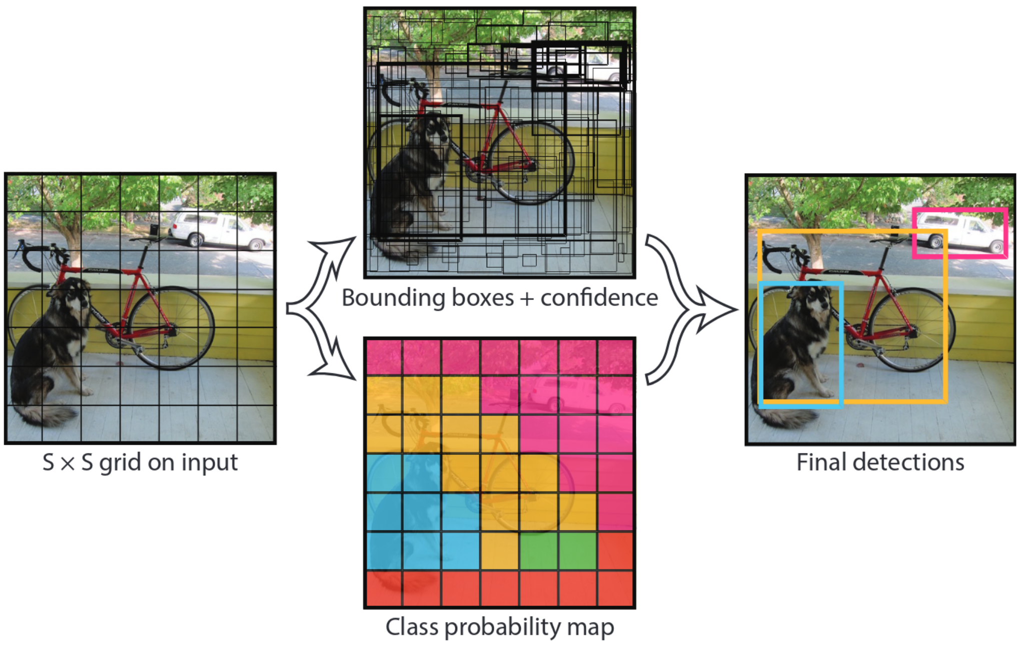


Figure 3.3: Detection process of YOLO. From [23, fig. 2].

Final confidence for each box is the product of both conditional class probabilities and the individual box confidence predictions. We can see the illustration of this process on fig. 3.3.

Architecture

YOLO is designed as a single network that takes the input image and outputs bbox and class predictions. Design of the network is inspired by Inception classification network. Although it does not use inception modules, it relies on the 1×1 reduction layers to speed up 3×3 convolutions. YOLO uses 24 convolutional layers followed by two fully connected layers. Full architecture is shown on fig. 3.4.

Various other versions and modifications are possible. A smaller and faster version, called Fast YOLO, has a similar architecture but uses only 9 convolutional layers. Another possibility to improve YOLO is to replace the custom architecture with a more common feature extractor from a classification network. YOLO build on top of a VGG16 achieves better precision at the cost of half of the frames per second.

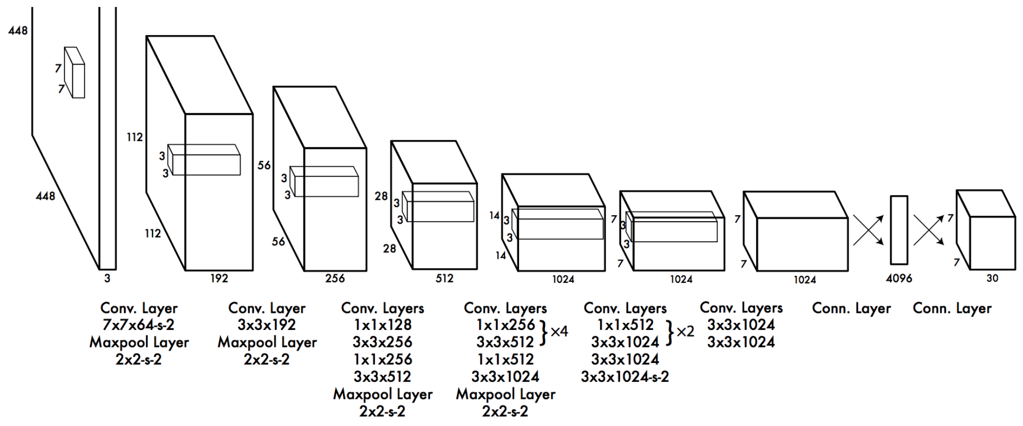


Figure 3.4: YOLO architecture for evaluating PASCAL VOC. It uses 7 by 7 grid with 2 bboxes per cell. Detecting 20 categories, the output's shape is [7×7×30]. From [23, fig. 3].

Training

Although the network can be trained end-to-end, it is common for CNN to pre-train on ImageNet dataset. This is also the case for YOLO. First, convolutional layers are pre-trained on the ImageNet dataset, then the detection layers are added, and the whole network is trained for detection. The model is optimized using the sum-squared error between predictions and ground-truths. The loss function is a sum of three parts, classification loss, localization loss, and bbox confidence loss.

Classification loss

$$\mathbf{L_{cls}} = \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{class}} (p_i(c) - \hat{p}_i(c))^2$$

Localization loss

$$\begin{aligned}\mathbf{L_{loc}} = {} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right]\end{aligned}$$

Confidence loss

$$\mathbf{L_{cnf}} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}}(C_i - \hat{C}_i)^2 + \lambda_{\text{noobj}} \sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}}(C_i - \hat{C}_i)^2$$

where $\mathbb{1}_{i}^{\text{obj}}$ denotes if object appears in cell i and $\mathbb{1}_{ij}^{\text{obj}}$ denotes that the jth bounding box predictor in cell i is responsible for that prediction.

The gradient of cells that do contain the object can be overpowered with the cells that do not. Therefore, the loss from negative confidence predictions is decreased by $\lambda_{\text{noobj}} = 0.5$. And to emphasize the bbox predictions, localization loss is increased using $\lambda_{\text{coord}} = 5$. In the localization loss, we can see that the center coordinates are handled differently to width and height. The square root of width and height is used to equalize the impact of the absolute value of error in small and large boxes.

Properties

A primary virtue of YOLO is its speed for real-time applications, and its simple architecture allows for easy training and end-to-end optimization. YOLO's detection layer is provided with context from the whole image which leads to less false detections than in region proposal methods.

On the other hand, a significant problem with YOLO's grid-based detection system is a limitation to one class per cell. This limitation results in the inability to detect multiple objects in close proximity, such as people in the crowd.

YOLO also suffers from multiple problems with precise localization. It learns to detect arbitrary shapes, which can be hard to generalize to objects in new and unusual aspect ratios. Also, it predicts the bboxes on the heavily down-sampled image which leads to further imprecision.

YOLO v2 (2017)

The second version of YOLO architecture by Redmon and Farhadi [22], introduces a series of improvements to the original network. Compared to original 45fps on [448×448] image, v2 achieves 59fps on [480×480] images.

A selection of changes to YOLO architecture:

- *batch normalization* [13] improves precision and helps regularization
- *fully convolutional architecture* shortens the inference time
- class and confidence predictions are no longer tied to grid locations, instead five *anchor boxes* are used

- detection on *multiple feature maps* with different sizes concatenated into channels

3.2.2 SSD: Single Shot MultiBox Detector (2016)

SSD is another real-time detector, aiming to outperform Faster R-CNN by using a single network with one-time evaluation. Liu et al. [18] presented this model just months after YOLO. Even though these networks are built on similar principles, there are multiple key differences. SSD manages to outperform Faster R-CNN and YOLO both in speed and precision.

Detection

One of the main features of SSD is that the detector network is fully convolutional and does not utilize any fully connected layers. Predictions are therefore generated for every position of a convolutional window. SSD adopts similar concept to Faster R-CNN's anchor boxes, this time called default or prior boxes. For each position of the feature map, multiple prior boxes with different aspect ratios are proposed. By default, six boxes are used. Contrary to YOLO, both bbox and classification predictions are made for each position and each prior box.

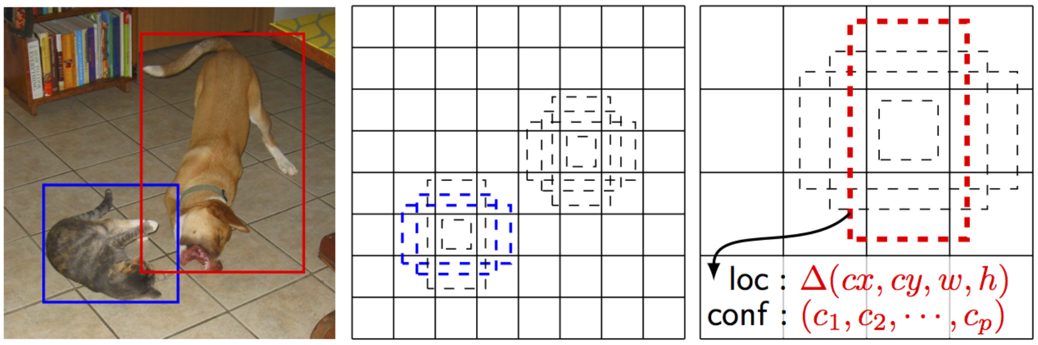
Bboxes are predicted relative to prior box locations which are themselves relative to a feature map location. There is no bbox or region confidence value. Instead, SSD uses an additional background class in classification predictions. Considering B prior boxes and C classes, SSD generates [m×n×(B*(C+5))] parameters on feature map of [m×n] size.

SSD detects objects on multiple feature maps at different scales, to accommodate detection of different sized objects. Moreover, this allows detectors on each level to focus on predicting a smaller range of bbox sizes. For the illustration of this process see fig. 3.5.

Architecture

The SSD architecture can be described as a set of three modules. A base network, extra convolutional layers and detection layers.

- **Base network**'s task is to take the input image and produce a feature map. To this end, a feature extractor build from classification network is an ideal candidate. SSD's base is built from VGG16 network with some modifications. First of all, all fully connected layers are removed and replaced with another pair of convolution layers. Pool5 layer is also changed from 2×2/2 to 3×3/1 pooling.


loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$

(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 3.5: SSD detection. (a) Input image with ground-truth boxes. (b) and (c) Predictions based on prior boxes on multiple scales of feature maps.

- **Extra layers** serve the purpose of providing more feature maps on decreasing scale to the detector. Smaller feature maps aggregate more information to a smaller area and allow for the detection of larger objects with small convolutional window. On the other hand, information about small objects can be lost. Therefore the use of gradually decreasing series of feature maps. *Extra layers* are implemented as a sequence of convolution layers connected to the end of the *base*.

- **Detection layers** are the final layers of the network. There is a pair of classification and localization convolutions for each feature map. Considering SSD300, where 300 stands for the width and height of input image. Detection is performed on 6 feature maps of sizes [38, 19, 10, 5, 3, 1], using [4, 6, 6, 6, 4, 4] prior boxes respectively, producing 8732 predictions per class. First, two of those feature maps are pulled from the VGG network, and the *extra layers* provide the rest. All detection layers are implemented using 3×3 convolutions with an appropriate number of filters, as seen on fig. 3.6.

Training

SSD pre-trains the *base network* on ImageNet dataset, and after that removes the classification layers and replaces them with *extra* and *detection* layers. The model is than trained for detection end-to-end. SSD utilizes *smooth L1* loss for localization and *cross-entropy* loss for classification. The final loss is the sum of those two components.

Before the training, we need to figure out which prior boxes match the ground-truth annotations. For each ground-truth box, two criteria are used:

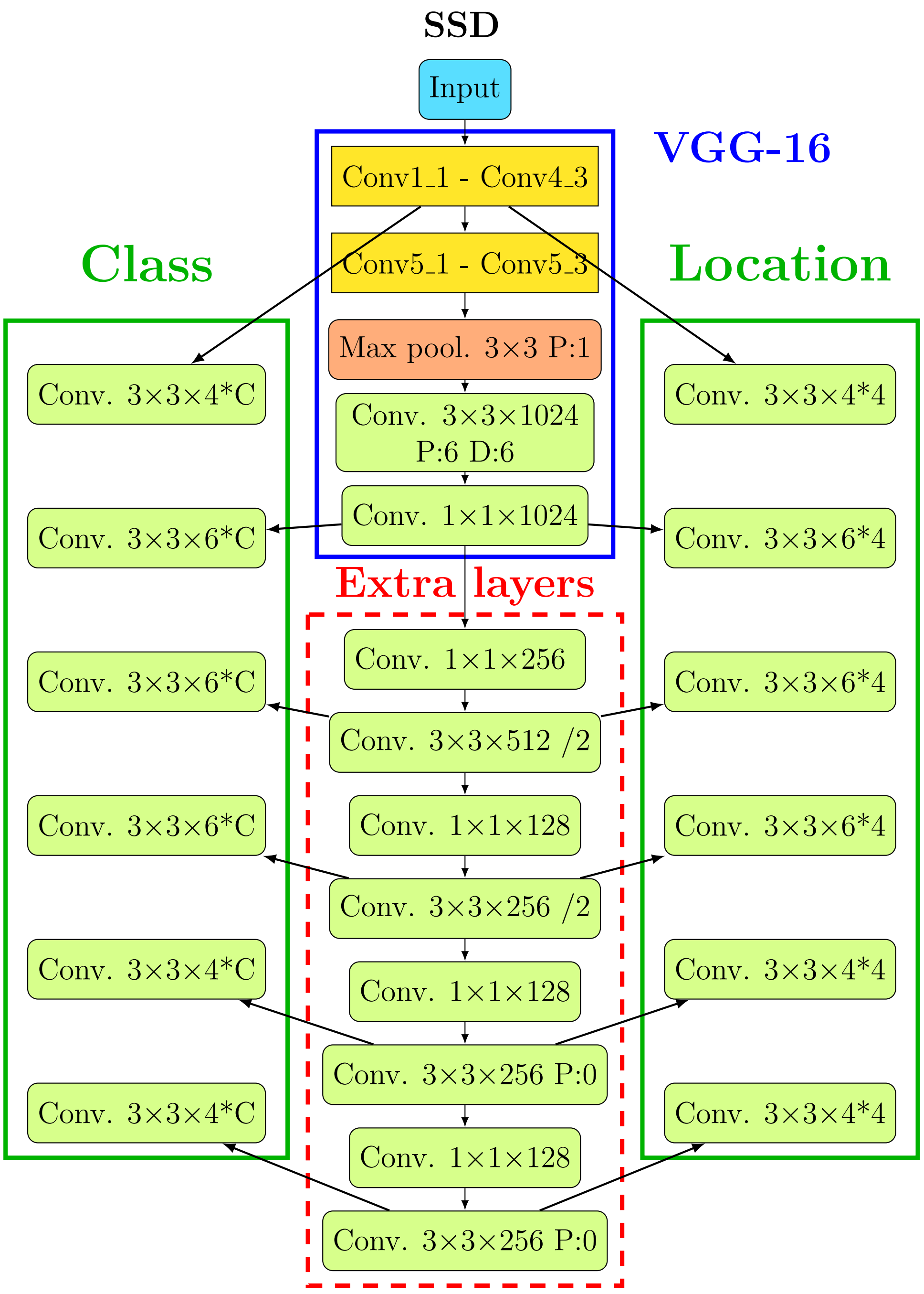


Figure 3.6: SSD architecture based on modified VGG16 network. VGG's three fully connected layers are replaced with two new convolutional layers, and Pool5 layer is changed from 2×2/2 to 3×3/1 pooling. Detection on the first and last two feature maps uses four prior boxes, while the rest uses six boxes. See more details on VGG in section 2.1.2.

a prior box with highest IoU is selected, and then, the ground-truth box is also matched to all prior-boxes with IoU higher than a threshold (0.5). Let the $x_{ij}^p = 0, 1$ be an indicator that ith prior box matches jth ground-truth box with class p.

To keep the balance between positive and negative samples, *hard negative mining* algorithm is employed. Only top negative samples, with the highest confidence score, are chosen. The goal is to keep the ratio of positives and negatives below 1:3.

Localization loss expresses the error between predicted boxes (l) and ground-truths (g). Predictions are generated in respect to corresponding prior boxes. Therefore, after matching the boxes, a ground-truths also need to be represented in respect to prior box (d), with center (c_x, c_y) and width (w) and height (h).

$$\mathbf{L}_{\text{loc}}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in (c_x, c_y, w, h)} x_{ij}^k \text{smooth}_{L1}(l_i^m - \hat{g}_j^m)$$
$$\hat{g}_j^{c_x} = (g_j^{c_x} - d_i^{c_x})/d_i^w \qquad \hat{g}_j^{c_y} = (g_j^{c_y} - d_i^{c_y})/d_i^h$$
$$\hat{g}_j^w = log(\frac{g_j^w}{d_i^w}) \qquad \hat{g}_j^h = log(\frac{g_j^h}{d_i^h})$$

Confidence loss or classification loss, is the softmax loss over class confidences (c).

$$\mathbf{L}_{\text{cls}}(x, c) = -\sum_{i \in Pos}^{N} x_{ij}^p log(\hat{c}_i^p) - \sum_{i \in Neg} log(\hat{c}_i^0) \quad \text{where} \quad \hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p exp(c_i^p)}$$

Total loss is then a weighted sum of both losses. The weight parameter α is set to 1. The loss is also divided by the number of matched prior boxes to keep it independent of the number of objects.

$$\mathbf{L}(x, c, l, g) = \frac{1}{N}(\mathbf{L}_{\text{cls}}(x, c) + \alpha \mathbf{L}_{\text{loc}}(x, l, g))$$

3.3 Detection with Temporal Information

In this section, we present two recent models designed specifically for detecting objects in the video. Videos can provide more information to the detector, compared to a set of single independent frames, thanks to the addition of temporal information. Theoretically, using multiple consecutive video frames can provide a large improvement regarding detector instability, and occlusions.

A time-space series of detected bounding boxes is usually referred to as a tube or a tublet. If such a series is generated at once for every object, it can provide smooth tracking and reduce the need for post-processing matching of detections. We use the term *chunk* to refer to a series of consecutive video frames, to differentiate from the batch of independent images.

However, adding another dimension to the detection model has a performance impact. It is also much harder to create a dataset suitable for training of detectors with a temporal dimension because a whole series of frames need to be precisely annotated. Of course we can find a few public datasets e.g. *ImageNet VID*[1], *YouTube-8M*[2] and some smaller ones like *HollywoodHeads*[3].

This section presents two approaches to adding temporal information to an object detector. First presented method adopts the approach of region proposal methods and the second one is based on a single-stage detector.

3.3.1 Tube-CNN (2018)

The architecture of this detector is very similar to Fast-RCNN with the added temporal dimension. The main idea is based on region/tube proposals and the following classification network for those regions. Vu et al. [32] proved that using a temporal information provided by continuous video frames enhances a precision of the detector. On the other hand, this approach adds more complexity to slower than real-time Faster-RCNN detector. The resulting network achieves only low single digit frame per second values, depending on the configuration.

Tube Proposal Network

Tube proposal generation begins with a feature extraction on a chunk of video frames. After the processing of all frames individually, feature maps are stacked together in the temporal dimension. Then, volumetric convolutional layers (conv3d) are applied to produce a feature volume.

Analogously to Faster-RCNN, each position in this feature volume is used to create tube proposals using K anchor tubes. The proposal for each anchor consists of the *objectness* and position parameters. The *objectness* score

[1]`http://image-net.org/challenges/LSVRC/2017/#vid`
[2]`https://research.google.com/youtube8m`
[3]`https://www.di.ens.fr/willow/research/headdetection`

reflects the probability of the presence of the same objects as opposed to a background.

IoU value for tubes is defined as a minimum of spatial IoU at the ends of tubes. The number of proposals is reduced by eliminating tubes with high overlap using a non maximum suppression based on *objectness* score.

A provided data for the training usually consists of a series of ground-truth boxes for individual frames. During the training of TPN, a series of ground-truth boxes is approximated by a ground-truth tube. Those tubes are then matched against the tube proposals using an IoU threshold. A tube proposal network is designed only to consider tubes corresponding to linear motion in order to limit the complexity.

Detection Network

Tube-CNN, as the name suggests, is a fully convolutional network with the task to extract feature maps from the incoming chunk of images, and using the given tube proposals, classify the tubes and refine the object positions.

Same as in TPN, the feature extraction is done independently on each frame using some general extractor, e.g., ResNet. Feature maps are then stacked to form a spatio-temporal feature volume. Unfortunately, the provided research paper does not clarify whether the proposal and detection networks share the same extraction layers or not.

A Tube-of-Interest (ToI) pooling is employed on feature volume to select the sub-volume corresponding to each tube proposal. The selected volume is then max-pooled into a fixed-size feature and subsequently used as an input for a classifier. A general convolutional classifier with soft-max activation is used.

The regression branch of the network predicts the exact position of the object in the first and last frame of the tube. Both regressions begin with RoI pooling on the corresponding feature maps and continue with convolutional layers to produce the positional parameters.

The detection process is illustrated in fig. 3.7.

3.3.2 TSSD (2019)

Chen et al. combine the SSD with recurrent networks [11, chpt. 10], namely ConvLSTM cells introduced by Xingjian et al. [33], to create a *temporal single-shot detector* (TSSD) [4]. They also proposed a tracking module with *Online Tubelet Analysis* working on top of the TSSD. Compared to SSD's 45fps, TSSD achieves 27fps on ImageNet VID dataset.

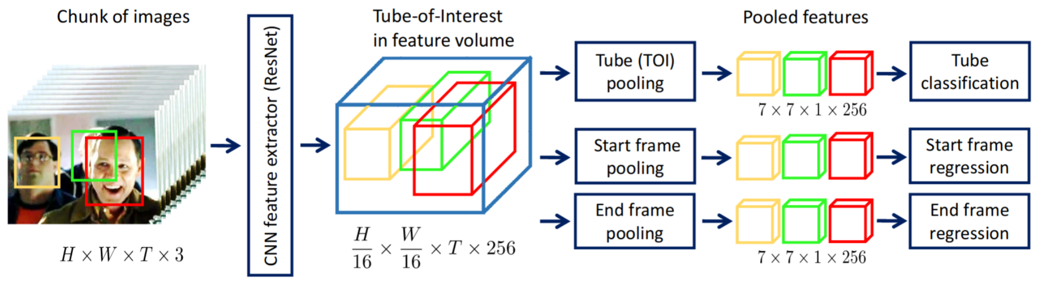

Chunk of images
$H \times W \times T \times 3$
CNN feature extractor (ResNet)
Tube-of-Interest in feature volume
$\frac{H}{16} \times \frac{W}{16} \times T \times 256$
Tube (TOI) pooling
Start frame pooling
End frame pooling
Pooled features
$7 \times 7 \times 1 \times 256$
$7 \times 7 \times 1 \times 256$
Tube classification
Start frame regression
End frame regression

Figure 3.7: Architecture of Tube-CNN for object detection. Taken from [32, fig. 2].

Architecture

TSSD is based on a standard backbone of SSD implemented on fully convolutional VGG16 base with extra layers (see section 3.2.2). This base provides six feature maps that are used for classification and bbox regression in SSD. However, TSSD applies one more layer on the feature maps before detection. This is where the temporal information comes into effect, using the aforementioned convolutional LSTM cells. Two *Attentional ConvLSTM* (AC-LSTM) cells are deployed, one for the bottom three feature maps and one for the top three. Only a small adjustment has been made to the underlying VGG network, that being lowering the number of channels in the second feature map to 512, to equalize the channels in all low-level feature maps. The outputs of both cells are then used for classification and regression in the same way the SSD would do. Details of TSSD implementation, including details on AC-LSTM, can be seen on fig. 3.8.

Features contributing to positive object detections are usually unevenly distributed in feature maps and through the scales. Authors proposed the Attentional ConvLSTM cell with the goal of background and scale suppression. The temporal attention module in AC-LSTM provides the rest of the cell with object-aware features.

Training

Similarly to SSD, the loss function of TSSD has multiple objectives weighted by α, β, γ and ξ constants.

$$\mathbf{L} = \frac{1}{N}(\alpha \mathbf{L}_{\text{loc}} + \beta \mathbf{L}_{\text{cls}}) + \gamma \mathbf{L}_{\text{att}} + \xi \mathbf{L}_{\text{asc}}$$

Where $\mathbf{L}_{\text{cls}}$ and $\mathbf{L}_{\text{loc}}$ are defined according to SSD (section 3.2.2) and N is the number of matched boxes.

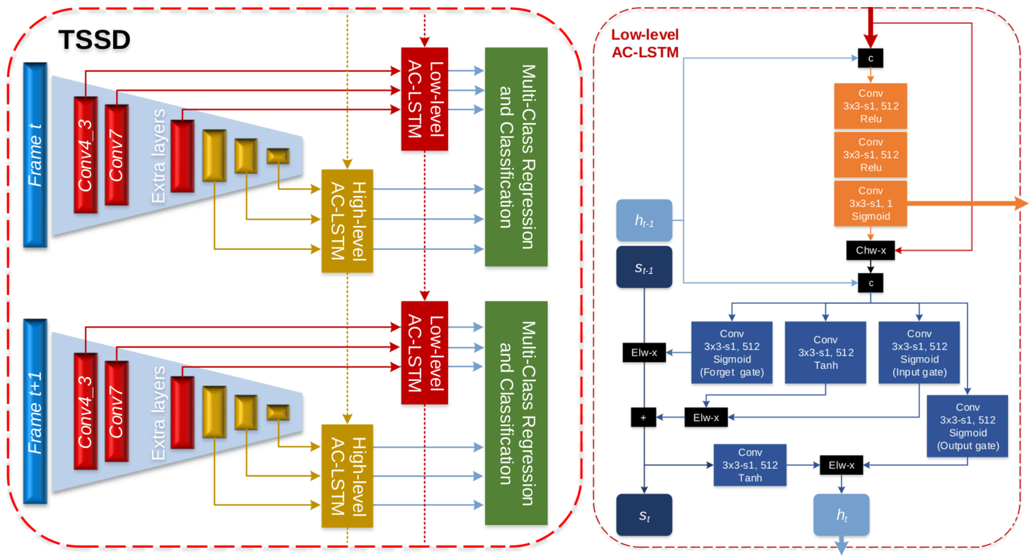


Figure 3.8: Architecture of TSSD (left) and AC-LSTM cell (right). *c* denotes concatenation; *Chw-x*, *Elw-x* represent channel-wise and element-wise multiplication, respectively; + is element-wise summation. From [4, fig. 2, 3].

Attention loss is calculated as a binary cross-entropy loss between ground-truth attention map A_g and prediction maps A_{sc} for each scale sc. A_g is a binary map with ones on positions inside ground-truth boxes. Each predicted attention map is up-scaled (A_{psc}^{up}) to match the dimensions of the A_g.

$$\mathbf{L}_{\text{att}} = \sum_{sc=1}^{6} \mu(-A_{psc}^{up} log(A_g) - (1 - A_{psc}^{up}) log(1 - A_g))$$

Video frames are generally temporally consistent. Therefore we also expect consistency in detections on short sequences of frames. We can suppress fluctuations in temporal detections by defining association loss. We use class-discriminative score list (sl) to represent detection in a frame. sl sums up top-k predictions after application of NMS.

$$\mathbf{L}_{\text{asc}} = (\sum_{t=1}^{seq} sl_t - sl_{avg})/seq$$

Where sl_t denotes the score list in frame t, sl_avg denotes the average list in a sequence and seq represents the sequence length.

4. Analyzing the Single Shot Detector

Our goal is to improve SSD detector proposed by Liu et al. and adjust it to fit the needs of video surveillance. We look for possible improvements by analyzing the network and other performance impacting factors. We take an especially close look at the underlying convolutional network in SSD and compare multiple alternatives. The other important factor we consider is that the SSD was designed for detection on individual frames instead of a continuous video stream.

We base our work on the implementation of SSD by Liu et al.. However, since the performance of the neural network is heavily dependant on the used framework and hardware, and the precision depends on the training data, we had to re-implement and train the baseline SSD for our comparisons.

Although we did not manage to reach the same precision as authors (41.2 mAP@[0.5]), we are aware of deficiencies in our implementation, that if resolved, should equalize the results. To speed up the training process we used fast and simple data augmentation algorithms and trained our models end-to-end without freezing the weights of pre-trained feature extractor and subsequent fine-tuning. We leave both of those issues to future work and instead focus on the relative comparison of models in the equivalent environment.

4.1 Feature Extraction Network

Looking at the structure of the network, we found the best candidate for improvement to be the underlying feature extractor. The feature extractor provides the data on which SSD performs detection and is, therefore, the integral part of the model. SSD uses relatively old VGG16 network that compared to more modern CNNs lacks in speed and precision. The feature extractor is in the context of object detectors often called the *base network*.

To explore the options, we started by implementing the SSD on multiple base networks and training them on the COCO dataset to analyze the impact on SSD's performance. We decided to implement SSD on three post-VGG networks, namely ResNet, Xception and NASNet.

We chose ResNet because it is a well-known network with a simple design and easy scalability. Xception got our attention for its performance in the benchmark by Bianco et al. (section 2.1.7), placing it around the optimal spot between speed and precision. NASNet, or precisely NASNet-A-Mobile, was chosen out of curiosity for its unique, machine learning designed structure.

We will be referring to ResNet networks by their number of layers, e.g., ResNet50. Also, Xception will be called Xception version A, or shortly XceptionA, to avoid confusion with versions we are going to introduce later. To emphasize the base of SSD network, we will be calling it *base*-SSD, e.g., VGG16-SSD.

4.1.1 Connecting SSD to Classification CNNs

To implement the SSD on other base networks, we first needed to decide how to create the interface between the networks. We needed to define which features of SSD and the base network architectures we wanted to keep unchanged and which would have to be adjusted.

SSD uses six feature maps, two extracted from the VGG16 network and four from extra layers. For input image of [300×300] pixels, the feature map sizes are: [38×38×512], [19×19×1024], [10×10×512], [5×5×256], [3×3×256] and [1×1×256]. We decided to preserve the spatial resolution of those feature maps as close as possible to the original, without changing the structure of base networks. Meaning, we did not keep the number of channels equal to SSD's. This approach certainly poses some risks. Not enough channels could negatively impact the precision, and too many channels would surely have an impact on the detection speed.

To find the most suitable layers for feature map extraction, we started with the strategy of finding the deepest possible layer with feature size as close as possible for every original feature map. This approach proved itself to be very straightforward since feature map sizes in convolutional networks decrease in resolution with the increasing depth, and the reduction is usually made by halving the size. After exhausting the network, we added the *extra layers* as needed, similarly to VGG16-SSD. The final feature map sizes are listed in table 4.1.

After determining the layers for feature extraction, we implemented the rest of the SSD without alteration. Each feature map is fed into both classification and localization layers with the corresponding scale, as described in fig. 3.6.

ResNet-SSD

Different sizes of ResNet network can be created by parameterizing its high level, four *Layer* architecture. Ideally, we wanted a single ResNet-SSD architecture that would support 34, 50 and 101 layer version without alterations.

VGG-16	ResNet34	ResNet50/101	XceptionA	NASNet*
[38×512]	[38×128]	[38×512]	[37×256]	[28×264]
[19×1024]	[19×256]	[19×1024]	[19×728]	[14×528]
[10×512]	[10×512]	[10×2048]	[10×2048]	[7×1056]
[5×256]	[5×512]	[5×512]	[5×512]	[4×512]
[3×256]	[3×256]	[3×256]	[3×256]	[2×256]
[1×256]	[1×256]	[1×256]	[1×256]	

Table 4.1: Feature map sizes used in SSD's detection. The dimensions are calculated for the input image of [300×300] pixels, except for NASNet, which only accepts [224×224] inputs. First values represent spatial dimensions of a square feature map and the second ones represents the number of produced channels.

Thankfully, this is exactly how the ResNet is designed, with consistent spatial feature map sizes between high-level *Layers*. Those sizes are [75×75], [38×38], [19×19] and [10×10]. The latter three are exactly matching the VGG16-SSD prototype.

After being provided with three feature maps by the network, we removed poolings and fully connected classifier and replaced it with an appropriate set of *extra layers* to produce the other three maps. The architecture of ResNet34-SSD, with highlighted detection feature maps is illustrated on fig. 4.1.

XceptionA-SSD

Similarly to ResNet, Xception can also be viewed in multiple layers of abstraction, the highest-level structure with *entry*, *middle* and *exit flow*, or lower-level twelve *block* structure with preceding and tailing convolutional layers (see fig. 2.4). After we removed the classifier from the *exit flow*, the three parts of the network provided feature maps of [19×19], [19×19] and [10×10] sizes. The latter two feature maps met our expectations. However, we had to dive deeper inside *entry flow* to find larger feature map, analogous to the original [38×38] map. The best option was to select the feature map produced by the second *block* of the network, with the [37×37] size.

We managed to extract three feature maps for detection from the networks, meaning that we had to add enough *extra layers* for the generation of remaining three maps. The illustration of XceptionA-SDD can be seen on fig. 4.1.

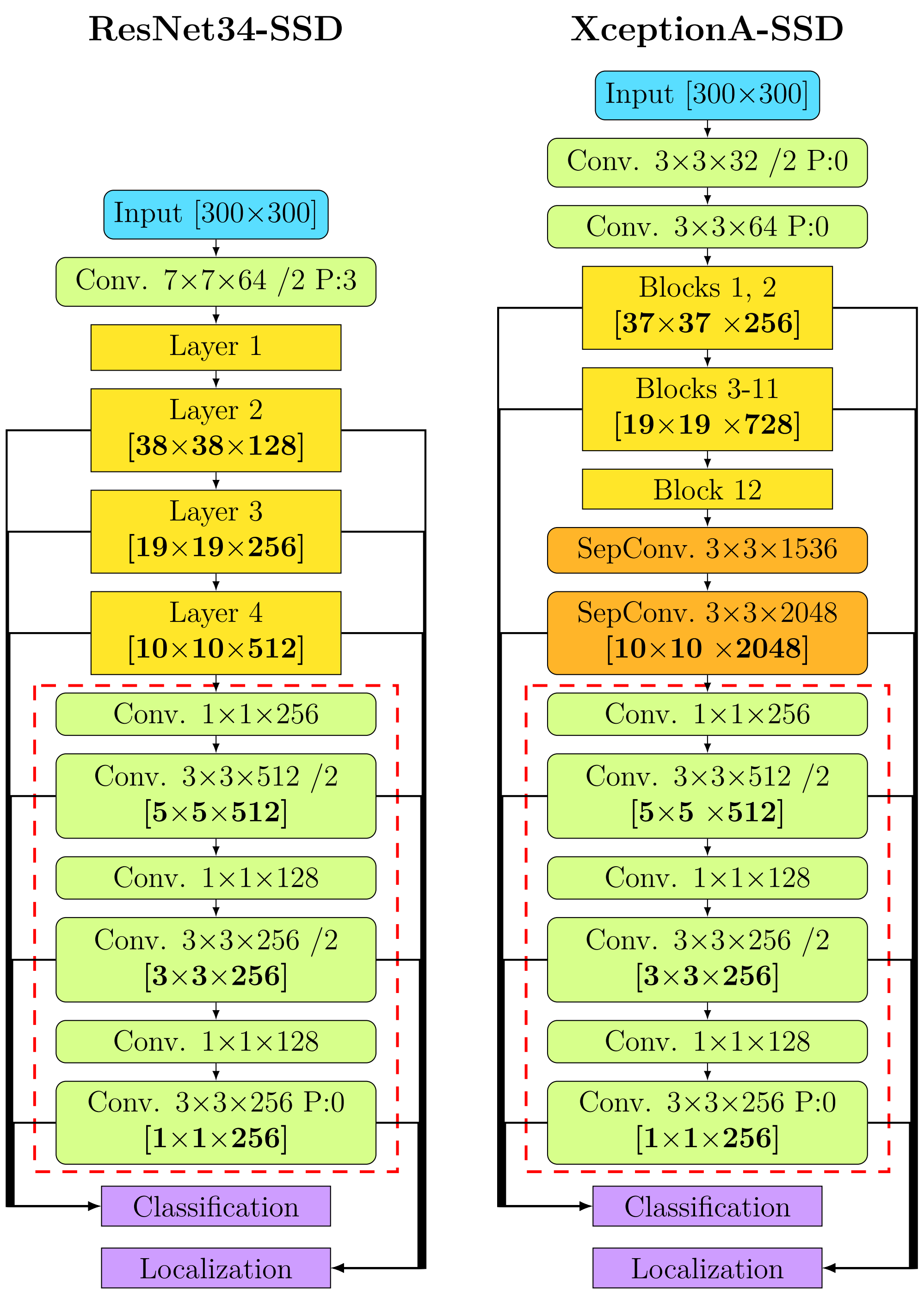


Figure 4.1: Resnet34-SSD(left) and XceptionA-SSD (right). For details on *Layers* and *Blocks* see fig. 2.3 and fig. 2.4 respectively. Extra layers are highlighted with a red dashed rectangle.

NASNet-SSD

Transition of SSD to NASNet-A-Mobile has proven itself to be the most complex of the three. In the beginning, we ran into the restriction on input image size. Due to the complex structure of the network with a lot of branching and subsequent concatenations and additions we decided against modifying the NASNet part of the detector and continuing with the given input size of [224×224] pixels.

Nevertheless, the input size was not the only problem. Since the mobile version of NASNet uses only four *Reduction Cells*, two of which are subsequent, and the stacks of *Normal Cells* do not change the spatial dimensions of feature maps, there are only three available sizes of feature maps. We had no option other than choosing to use feature maps of [28×28], [14×14] and [7×7] sizes. Keeping with the general formula for adding *extra layers*, i.e., using 3×3 convolutions with stride 2, we decided against advancing beyond [2×2] feature map and reduce the number of extracted feature maps to five. See fig. 4.2 for illustration.

4.1.2 Performance Results

In order to compare the performance of our SSD networks, both in terms of precision and speed, we trained them on the COCO dataset. We tested the prototype VGG16-SSD, and SSDs implemented on the ResNet34, the ResNet50, the ResNet101, the XceptionA, and the NASNet-A-Mobile. After the training, we tested the networks for speed and precision and plotted the results to fig. 4.3. The results show that VGG16-SSD is the slowest one. However, in terms of precision, it surpasses both the XceptionA and the NASNet SSDs.

With the knowledge gained from implementing the NASNet-SDD and the acquired results, we believe that the network, or at least its mobile version, is not suited for this task. The results exposed the significance of the trade-offs we were forced to implement, in order to build SSD on this network.

Although the results of the XceptionA-SSD were disappointing, we believed that the simple structure of the network would allow us to make the necessary changes and outperform VGG16. We formulated the hypothesis that the main reason for the poor precision is the fact that the first feature map is extracted from too shallow of a layer. The depth of the network at the point of the extraction is only six convolutional layers, granted some are split into depthwise separable convolution, we believed it is not enough for the first feature map of the detector. We will test our hypothesis and try to rectify the shortcomings of Xception-SSD in section 4.3.

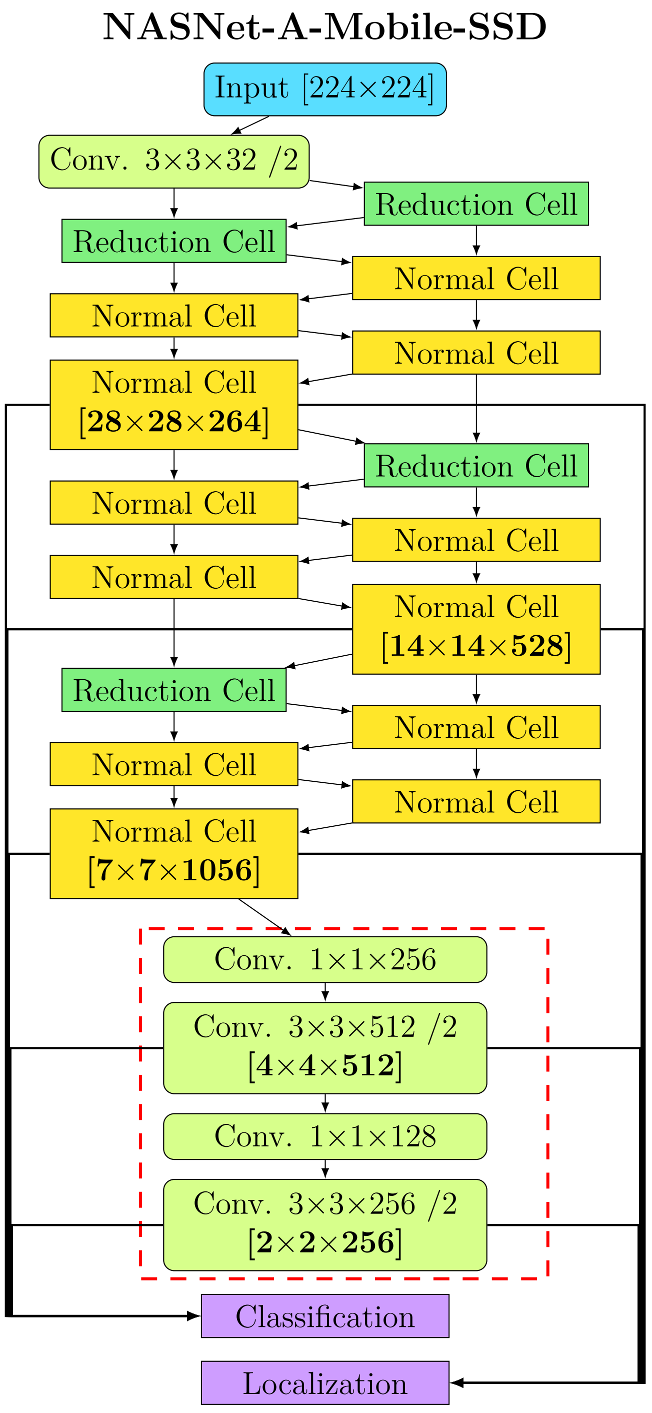


Figure 4.2: NASNet-SSD based on NASNet-A-Mobile. For detailed description of cells see section 2.1.6. *Extra layers* are highlighted with a red dashed rectangle.

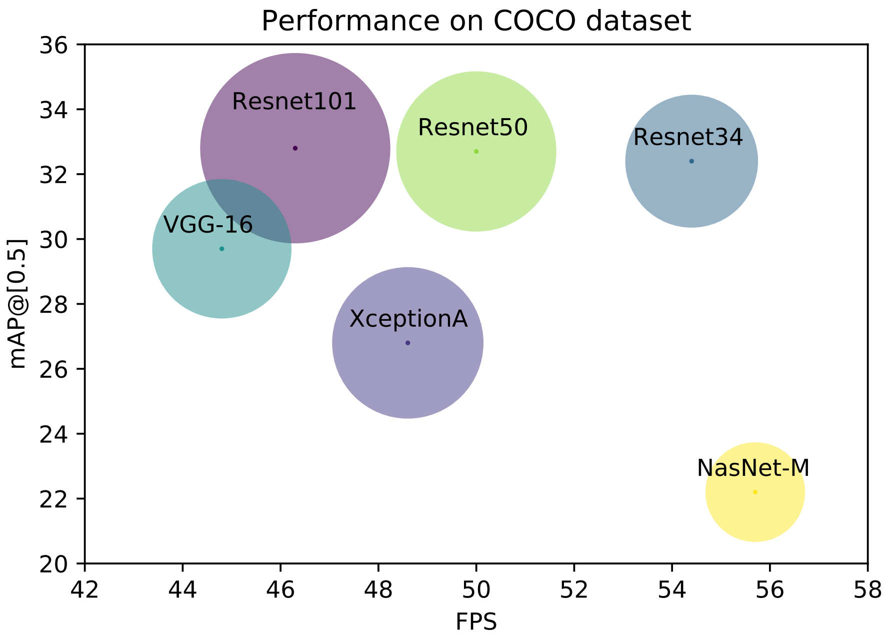


Figure 4.3: Performance of SSD detector on multiple base networks. Circle radii demonstrate relative difference of network parameter counts.

4.2 Training Data and Classes

The principal difficulty in solving problems with neural networks is the lack of available data. Although some networks can be trained using reinforcement learning, object detection is one of those that requires supervision. However, the creation of a problem-specific dataset requires a lot of human resources. The other option is to use public datasets like the COCO dataset.

The problem of such datasets is that if they include every required class, they also often include many classes that are not needed for a particular application. For surveillance, we are only interested in classes such as people and vehicles.

We compared the performance and precision of SSD trained on full COCO dataset and a subset for surveillance. We expected significant performance improvement by lowering the number of detected classes. However, we were not sure how would this change impact precision. We hypothesized that by removing classes with similar features to the detected ones, there is a possibility of increasing the error by means of false detections.

Surveillance Dataset Classes

- person
- bicycle
- motorcycle
- car
- bus
- truck
- train

4.2.1 Precision Impact

Before we get to the comparison of the training results, we will explain our hypothesis on an example. Consider that we want to detect people in a ZOO and we have access to the dataset with people and animals. Would it be better to train the network only to detect people or to detect both humans and animals, and then filter out the animals in post-processing? In the latter case, the network would ideally detect people as people, animals as animals, and everything else as a background. However, in the case where we train the network only to recognize people, how would such a network classify a monkey? Would similar features prevail and shift the classifier towards the person, or would negative examples of a monkey in dataset be strong enough for teaching the network that it is not a person?

The problem we describe is, of course, part of a broader difficulty with non-exhaustive datasets. Our question is, therefore: If we have an available set of annotations for the classes we are not interested in, but that can be present in our input images and share similarities with detected classes, would it not be better to learn to detect those classes?

Results

Table 4.2 shows that in our case, it is possible to remove unnecessary classes from the dataset without impacting the performance. It is worth mentioning that we tried to precautiously counter-measure our hypothesis by creating the Surveillance dataset by only filtering the annotations and keeping all images of the original dataset.

We tested the approach on multiple architectures, and the results do not conclusively favor one dataset over the other. However, based on this experiment, we are not able to conclusively confirm or deny our hypothesis. Even if we wanted to make a conclusion for the COCO dataset using the provided training and validation data, the experiment is still dependant on the choice of classes.

	COCO	Surveillance
ResNet34	47.3	47.3
ResNet50	46.6	48.7
ResNet101	47.2	45.7
XceptionA	39.4	37.8
NASNet	36.4	36.9

Table 4.2: Mean average precision of Surveillance classes. Comparing networks trained on all 80 classes of COCO dataset and Surveillance subset of COCO.

4.2.2 Performance Impact

We expected a proportional increase in performance with the elimination of unwanted classes. We saw in fig. 3.6 that the channel depth of the classification layers in SSD is dependent on the number of classes. Shallower classification layers are not only processed faster in the neural network but also produce less data that will result in faster post-processing, mainly, non-maximum suppression.

We illustrate the effect that the number of detected classes has on performance in fig. 4.4. Although the relationship of frames per second and classes is hyperbolic, on the interval between 7 and 80 classes, we approximate the loss of 0.8fps with each additional class on ResNet50-SSD. Considering the dataset can contain tens or hundreds of classes, we showed that the filtering out the unwanted classes could produce significant performance benefit.

4.3 Modifying Xception

We managed to successfully re-implement SSD on other architectures and boost the speed of surveillance by removing unnecessary classes. We can confidently say that ResNet50-SSD with 48.7% mAP and 108fps or ResNet34-SSD with 47.3% mAP and 125fps outperform original SSD on VGG16 with 46.1% mAP and 42fps. However, we believed that the underwhelming result of XceptionA-SSD could be used as a stepping stone and the results could be pushed further with modifications to the Xception architecture.

We already mentioned the hypothesis that the major factor limiting the precision of XceptionA-SSD is the extraction of [37×37] feature map after the second block of the network. To rectify this problem, we decided to make adjustments to the network, in such a way that we could extract [37×37] feature map after *block 7*, and keep the [19×19] map after *block 11* as previously dictated by network architecture. The reasoning behind choosing *block*

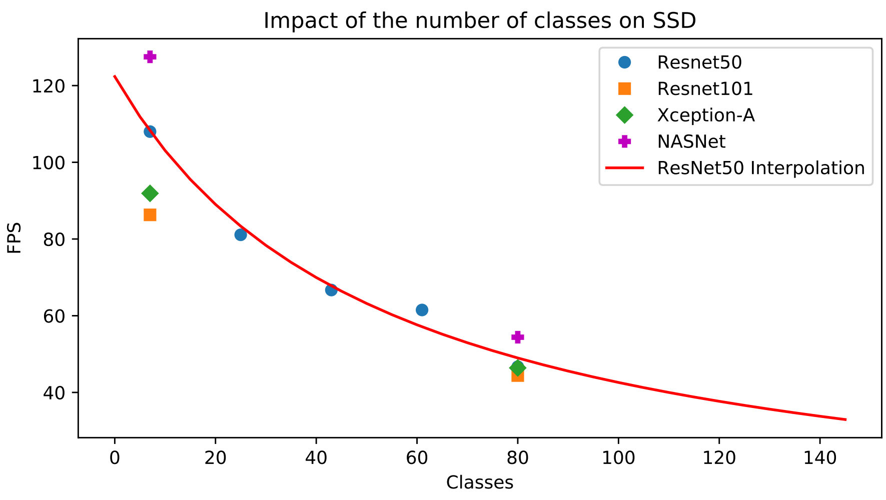


Figure 4.4: Impact of the number of classes on SSD performance. Inference time values ($1/fps$) are linearly interpolated, the result is therefore hyperbolic approximation of frames per second.

7 was that it is deep enough in the network and leaves four blocks between the feature map extractions. It is perhaps a bit arbitrary decision and blocks 6 or 8 would do just as well, or better. In order to achieve the correct sizes of feature maps, we moved the max-pooling with stride 2 from *block 3* to *block 8*. This resulted in outputs of *blocks 2-7* to be the [37×37] size and *block 8* to output [19×19] feature map.

The proposed change to the pooling, and thus feature map sizes results in the increased complexity of the network as the [19×19×728] features now grew to the [37×37×728] size. To remedy this, we decreased the number of channels for concerned blocks to 256. We also decided to continue with trimming layers in the rest of the network. In the end, we got the [37×37] map with 256 channels, [19×19] map with 512 channels and [10×10] map with 1024 channels. The architecture with highlighted changes is illustrated on fig. 4.6.

Our modification not only helped to improve Xception-SSD to overcome VGG16, but we also outperformed ResNet50-SSD's 48.7% mean average precision with 49.8 % mAP. However, despite our best efforts to save computation, we managed to outperform XceptionA only slightly, by achieving 105fps. The relative performance of XceptionH based SSD to other SSDs using the Surveillance dataset can be seen on fig. 4.5.

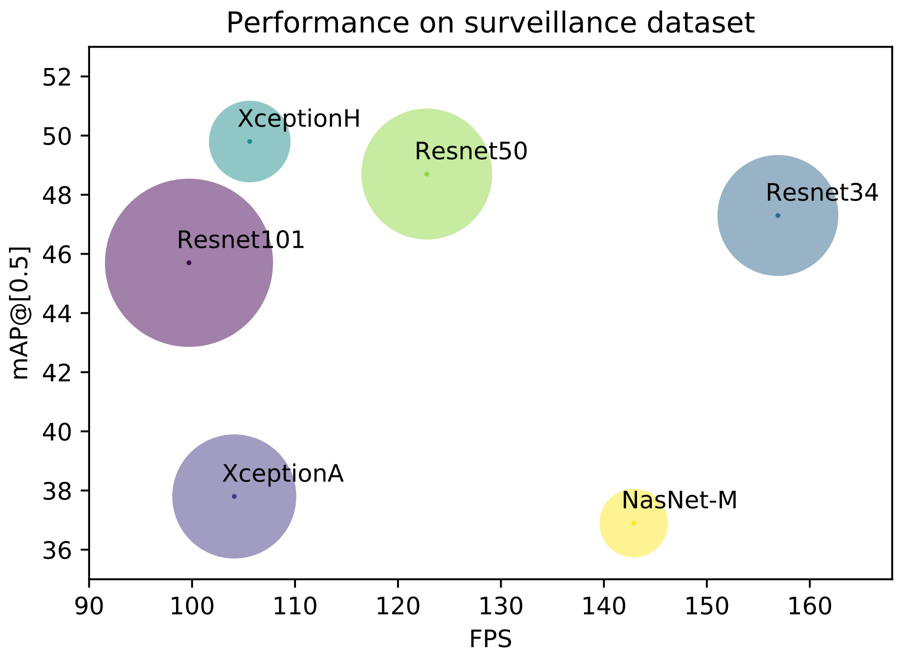


Figure 4.5: Performance of SSD detectors on multiple base networks on Surveillance dataset. Circle radii demonstrate relative difference of network parameter counts.

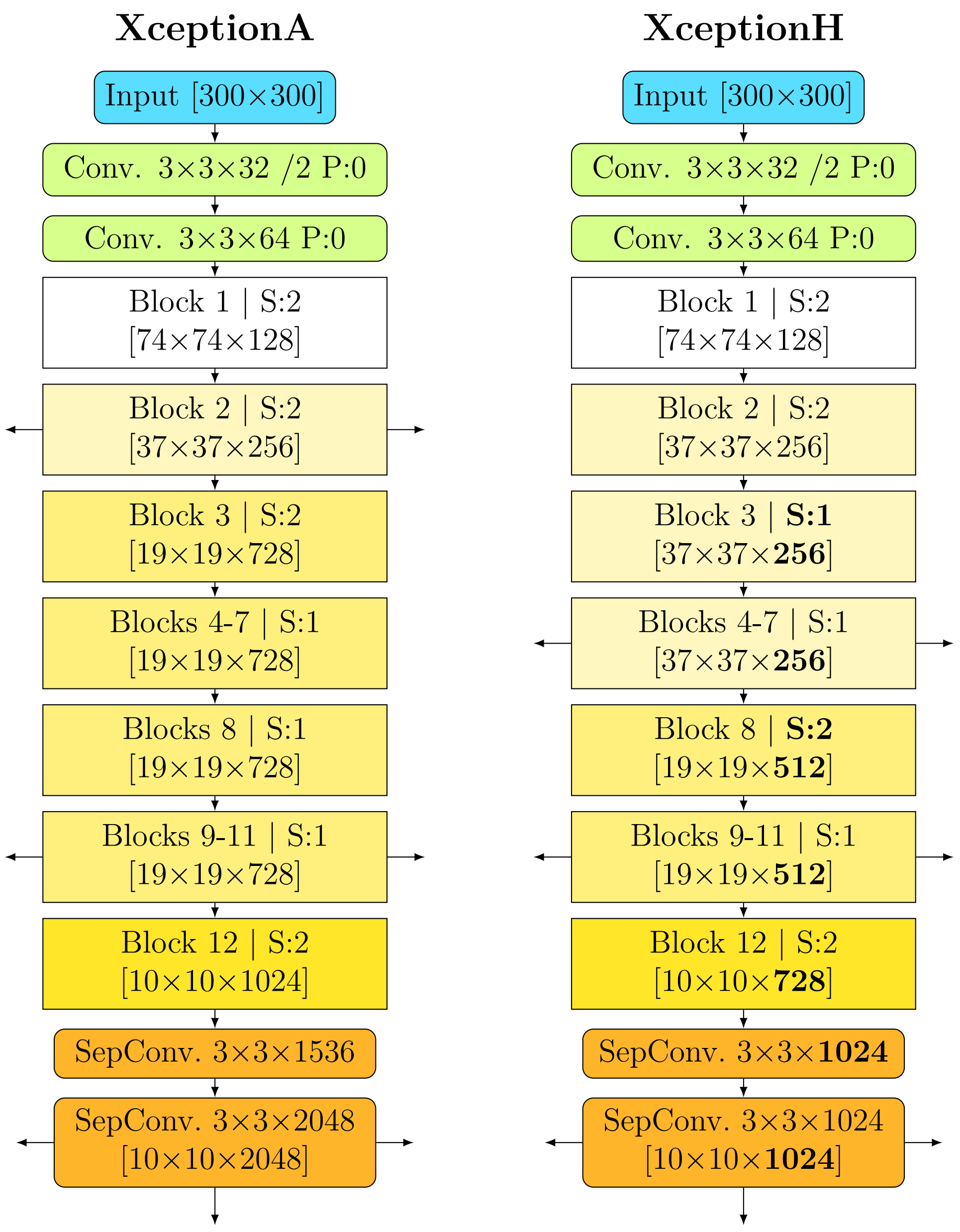

XceptionA
Input [300×300]
Conv. 3×3×32 /2 P:0
Conv. 3×3×64 P:0
Block 1 | S:2 [74×74×128]
Block 2 | S:2 [37×37×256]
Block 3 | S:2 [19×19×728]
Blocks 4-7 | S:1 [19×19×728]
Blocks 8 | S:1 [19×19×728]
Blocks 9-11 | S:1 [19×19×728]
Block 12 | S:2 [10×10×1024]
SepConv. 3×3×1536
SepConv. 3×3×2048 [10×10×2048]
XceptionH
Input [300×300]
Conv. 3×3×32 /2 P:0
Conv. 3×3×64 P:0
Block 1 | S:2 [74×74×128]
Block 2 | S:2 [37×37×256]
Block 3 | **S:1** [37×37×**256**]
Blocks 4-7 | S:1 [37×37×**256**]
Block 8 | **S:2** [19×19×**512**]
Blocks 9-11 | S:1 [19×19×**512**]
Block 12 | S:2 [10×10×**728**]
SepConv. 3×3×**1024**
SepConv. 3×3×1024 [10×10×**1024**]

Figure 4.6: XceptionH architecture (right) compared to XceptionA (left). Changes are highlighted with bold font. Connection to SSD's detection layers are indicated by the arrows on the sides, *extra layers* are appended to the bottom of the network. *S* represents stride of the block, implemented using max-pooling. Blocks are also color-coded based on the feature map size. Extra, classification and localization layers are unchanged from fig. 4.1. For details on *Blocks* see fig. 2.4.

4.4 SSDTC: SSD with Temporal Convolution

Since our main priority is video surveillance, we wanted to explore the options of video detection, exploiting the additional information a video can provide over still images. In section 3.3 we have already examined two approaches of utilizing the continuity of video frames to achieve higher precisions. One approach used architecture similar to Faster R-CNN with use of temporal tubes-of-interest. The other one used convolutional LSTM cells to harness the temporal information inside a modified SSD. However, the major drawback of both approaches was their inference speed.

Inspired by the two approaches, we decided to implement our version of video detector with temporal information. To this end, we chose to expand on the SSD. We already have an understanding of the model, and the one-stage detectors are currently the only relevant choice considering speed. We also wanted to avoid adding complex, time-consuming structures like LSTM cells used in TSSD. Instead, we decided to use three-dimensional convolutional layers (conv3d), to aggregate the information from multiple images. We named our approach *Single Shot Detector with Temporal Convolution* (SSDTC).

Of course, we need to prove our concept by training it on some data and comparing it to unmodified SSD. Since we require a dataset with consecutive frames, we decided to start with the HollywoodHeads dataset. This set annotates heads in sections of movies. Considering that dataset has only one class and previous results of SSD testing suggests ResNet34-SSD would be more than sufficient for this task, we trained it as a baseline. Consequently, we also based SSDTC on ResNet34-SSD.

Architecture

In standard SSD, the detection is performed on a batch of independent images. We expect to perform detection on a single batch, or a chunk, of consecutive frames.

We start by extracting the feature maps, as we would in SSD, independently on each frame with the same network (ResNet34 with *extra layers*). Starting with the chunk $n = 16$, we get 16 sets of feature maps. For the ease of explanation, consider only the first feature map of [38×38×c] size. Stacking those maps from the chunk, we get a temporal feature volume of [38×38×c×n] size, where c is the number of channels.

At this point, we apply the temporal convolutional layers, realized using conv3d layers. We apply two conv3d layers on each feature volume, both followed by batch normalization and ReLU activation. The first one, *Conv3d.*

1×1×3×ch operates only on temporal and channels dimensions. The subsequent one works with all dimensions of feature volume, applying *Conv3d. 3×3×3×(2*ch)* layer.

No padding is used in the temporal dimension of conv3d layers, therefore we only receive *n-4* detections for *n* input frames. This may seem inefficient because five frames are needed for detection on one frame, but the impact of this constant overhead can be minimized with bigger chunks.

After the temporal layers, we can again view the created feature volume as an array of independent feature maps corespondent to frames and apply SSD's detection layers.

In simple terms, we add temporal information from neighboring frames to each feature map, before executing the detection. This allows for reuse of most of the SSD's architecture and simple transition on other base networks. The illustration of SSDTC's architecture can be seen on fig. 4.7. To our best knowledge, this is a unique combination of SSD and three-dimensional convolution for object detection with temporal information.

Results

A previously mentioned, ResNet34-SSD was trained on the same dataset to serve as s baseline for comparison. This SSD achieved the precision of 81.16% while performing at 156 frames per second.

Our SSDTC architecture managed to reach the precision of 86.73% with processing speed of 148 fps. However, considering that detection is not performed for every processed frame, the effective speed of the network is 111 frames per second with the chunk size of 16 frames. In fig. 4.8, we can see that SSDTC helped to remove many false detections and stabilized the detections.

Meanwhile, the Tube-CNN method we reviewed in section 3.3.1 reports the precision of 86.8% while reaching less than 10 fps. Therefore, we consider this result a successful proof of our concept. We managed to reach the precision of temporal region-based network while exploiting the speed of one-stage detector.

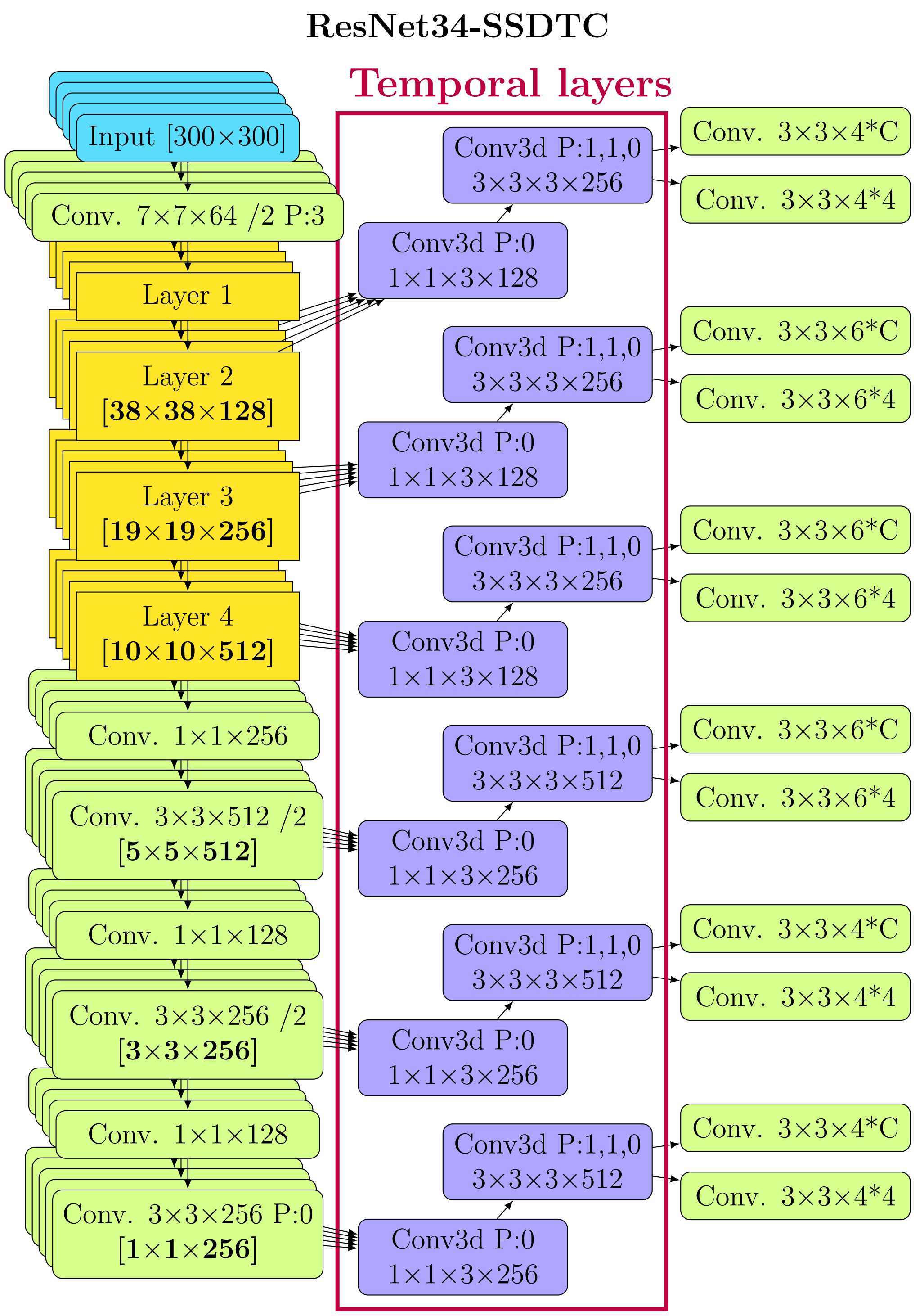


Figure 4.7: SSDTC architecture on ResNet34 base. All temporal layers are followed by batch normalization and ReLU activation functions, no padding is used in temporal dimension.

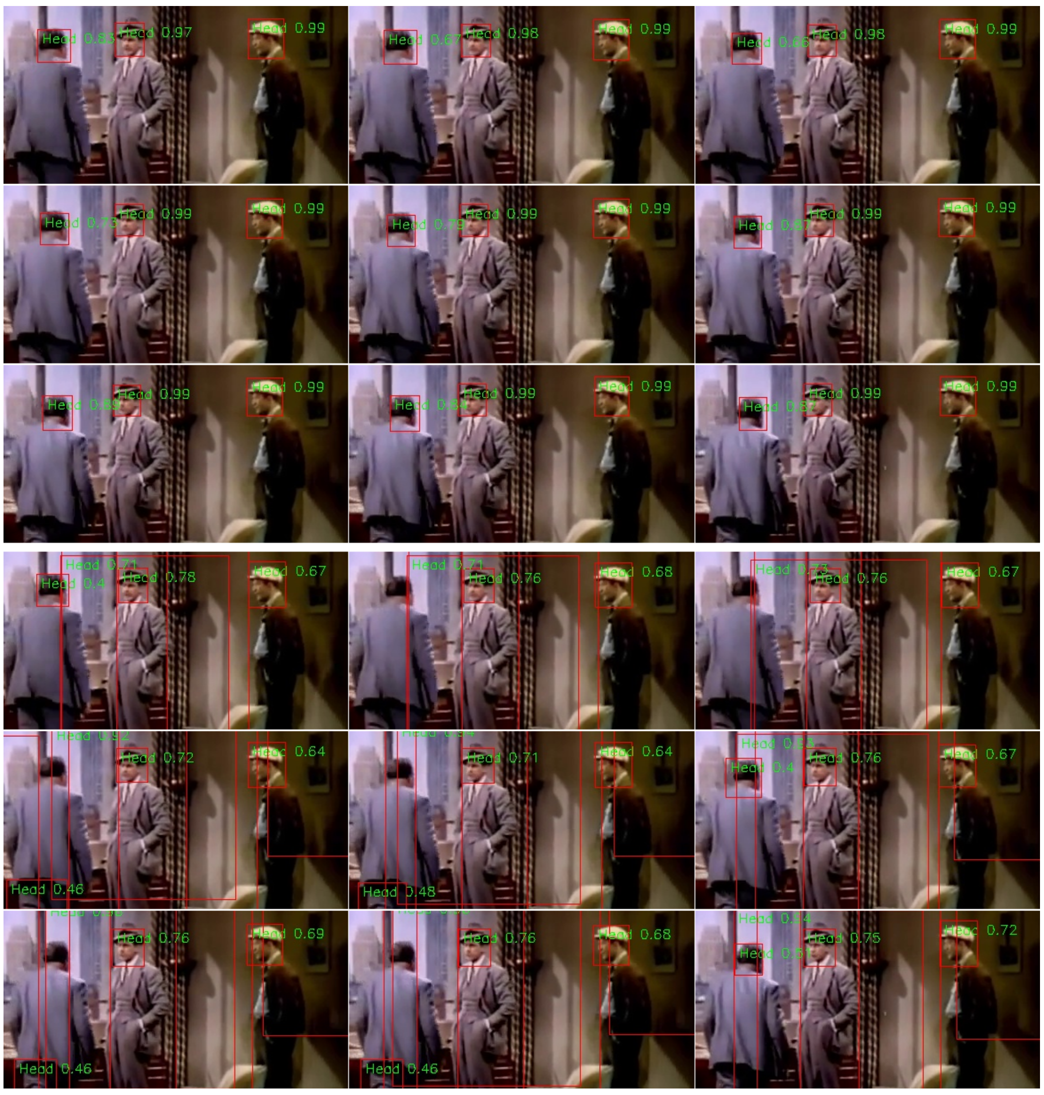


Figure 4.8: Comparison of detections by SSDTC (top) and SSD (bottom). Performed on the sequence of nine frames, ordered from left to right, top to bottom. Frames were cropped after detection.

5. Experiments

In this chapter, we provide details on the techniques used to gather the data presented in the previous chapter. We also present additional experimental results.

5.1 SSD Training

Liu et al. implemented SSD model[1] from their research paper using the *Caffe*[2] deep learning framework. We decided against performing our experiments in this, by now outdated framework, and implemented our version in newer *PyTorch*[3] framework (more detailed technical specification of used libraries is available in appendix A).

We started by implementing a common framework for SSD detectors, that would support SSD models with many modifications. Although SSDTC inference is possible in the same framework, we had to create a separate environment for the training (more on SSDTC in section 5.4).

The main characteristics of our framework are the use of *SGD* optimizer [3] and the loss function defined for SSD, also called MultiBox loss (section 3.2.2). Another essential element of the detection pipeline, used for inference, is the non-maximum suppression algorithm. The parametrization of these modules can be seen in table 5.1. It is also worth mentioning that for the sake of consistency, all experimental models were trained for 401 epochs with a batch size of 32, using those default parameters.

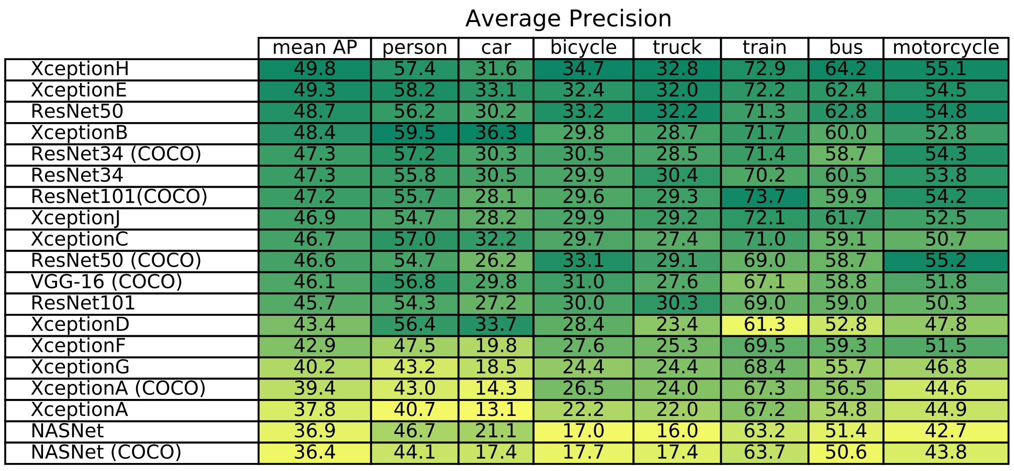
[1] `https://github.com/weiliu89/caffe/tree/ssd`

[2] `http://caffe.berkeleyvision.org/`

[3] `https://pytorch.org/`

MultiBox loss		SGD optimizer		NMS	
IoU threshold	0.5	learning rate	10^{-3}	IoU threshold	0.5
positive/negative sample ratio	1:3	momentum	0.9	confidence threshold	0.2
		weight decay	5×10^{-4}		

Table 5.1: A selection of default values for most important parameters in our framework. We used these values for training and evaluation in all experiments. Note that the learning rate decreases during the training.



Average Precision

	mean AP	person	car	bicycle	truck	train	bus	motorcycle
XceptionH	49.8	57.4	31.6	34.7	32.8	72.9	64.2	55.1
XceptionE	49.3	58.2	33.1	32.4	32.0	72.2	62.4	54.5
ResNet50	48.7	56.2	30.2	33.2	32.2	71.3	62.8	54.8
XceptionB	48.4	59.5	36.3	29.8	28.7	71.7	60.0	52.8
ResNet34 (COCO)	47.3	57.2	30.3	30.5	28.5	71.4	58.7	54.3
ResNet34	47.3	55.8	30.5	29.9	30.4	70.2	60.5	53.8
ResNet101(COCO)	47.2	55.7	28.1	29.6	29.3	73.7	59.9	54.2
XceptionJ	46.9	54.7	28.2	29.9	29.2	72.1	61.7	52.5
XceptionC	46.7	57.0	32.2	29.7	27.4	71.0	59.1	50.7
ResNet50 (COCO)	46.6	54.7	26.2	33.1	29.1	69.0	58.7	55.2
VGG-16 (COCO)	46.1	56.8	29.8	31.0	27.6	67.1	58.8	51.8
ResNet101	45.7	54.3	27.2	30.0	30.3	69.0	59.0	50.3
XceptionD	43.4	56.4	33.7	28.4	23.4	61.3	52.8	47.8
XceptionF	42.9	47.5	19.8	27.6	25.3	69.5	59.3	51.5
XceptionG	40.2	43.2	18.5	24.4	24.4	68.4	55.7	46.8
XceptionA (COCO)	39.4	43.0	14.3	26.5	24.0	67.3	56.5	44.6
XceptionA	37.8	40.7	13.1	22.2	22.0	67.2	54.8	44.9
NASNet	36.9	46.7	21.1	17.0	16.0	63.2	51.4	42.7
NASNet (COCO)	36.4	44.1	17.4	17.7	17.4	63.7	50.6	43.8

Table 5.2: Average precision of all tested networks on Surveillance classes. COCO indicates that the network was trained on COCO dataset, otherwise only Surveillance data were used for training.

5.2 Measurements

To make our work comparable to other similar studies and future works, we explain methods used for both precision and performance measurements.

5.2.1 Precision

Rather than implementing a copy of the precision evaluation, our precision measurements were taken using an external tool. Using the external tool, independent on our implementation, allows us to easily compare our results with other works with little to no modifications. We used the implementation by Padilla [21] that mirrors the evaluation process of the PASCAL VOC Challenge. The test setup used default parameters, meaning that the interpolation of AP was calculated using all data points and the IoU threshold was set to 0.5.

We present all the measurements taken on Surveillance dataset while performing multiple experiments described in this thesis in table 5.2.

5.2.2 Inference Speed

The absolute values of the inference speed measurement would not provide any information without the knowledge of the environment in which they have been taken. In this section, we provide the details of both software and hardware environments used for measurements.

Testing was done by processing a total of 10 000 images in batches of 16. This process was timed, and the average *fps* value was calculated. Since we do not consider scaling and cropping the images to be part of the network, we did not need to include this process in the measurement. The set of [300×300] pixel images was pre-loaded into memory before the timer was started. On the other hand, non-maximum suppression is a critical part of the algorithm and is included in the measurement.

All our testing was done on the following hardware:

- AMD EPYC 7401P CPU @ 2GHz ×24
- NVIDIA GeForce GTX 1080 Ti
- 128GB DDR4 RAM

5.3 Improving the Xception-SSD

In section 4.3 we introduced a hypothesis about Xception$A$-SSD. Based on this hypothesis, we presented the improved Xception$H$ model. However, as suggested by the name, it was not our first modification, and we needed some trial-and-error testing to achieve this result.

We will describe every iteration of the Xception-SSD model we trained and the reasoning behind the particular modifications. For clarity, we will refer to the Xception$X$ models in this section only by their version letters. The performance of mentioned models on Surveillance dataset is plotted on fig. 5.1, and more details on precision in table 5.2. Also, the feature map sizes inside the models are shown in table 5.3.

Versions B, C, D

We examine this trio at once, since version C adds modifications to version B, and version D further modifies version C. Notably, we trained these models in parallel and therefore had no results from concurrent models to inform on the design.

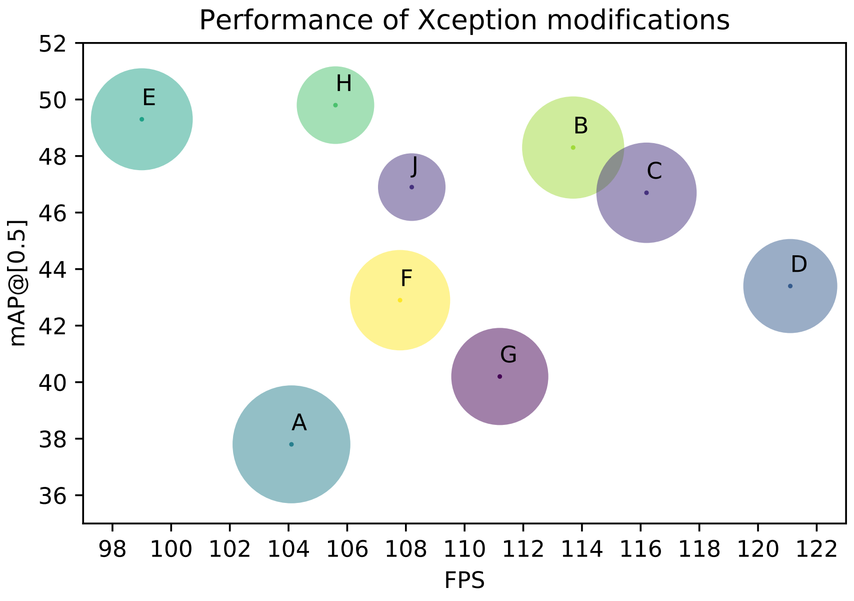


Figure 5.1: Performance of multiple Xception modification on Surveillance dataset. Circle radii demonstrate relative difference of network parameter counts.

	A	B (C, D)	E (F, G)	H	J
B1	[74×128]	[74×128]	[74×128]	[74×128]	[74×128]
B2	**[37×256]**	[37×256]	[37×256]	[37×256]	[37×256]
B3	[19×728]	[19×256]	[37×256]	[37×256]	[37×256]
B4-6	[19×728]	[19×256]	[37×256]	[37×256]	[37×256]
B7	[19×728]	**[19×256]**	**[37×256]**	**[37×256]**	**[37×256]**
B8-10	[19×728]	[19×728]	[19×728]	[19×512]	[19×512]
B11	**[19×728]**	**[19×728]**	**[19×728]**	**[19×512]**	**[19×512]**
B12	[10×1024]	[10×1024]	[10×1024]	[10×728]	[10×512]
S1	[10×1536]	[10×1536]	[10×1536]	[10×1024]	[10×512]
S2	**[10×2048]**	**[10×2048]**	**[10×2048]**	**[10×1024]**	**[10×512]**

Table 5.3: A table of feature map sizes on the output of layers of the Xception networks. Bx stand for Xception blocks, S1 and S2 stand for separable convolution layers that follow the block structure (see fig. 2.4). The first number represents spatial dimensions of a square feature, expecting [300×300] input, and the second one represents the number of channels. The highlighted feature maps are used for the detections. The versions C, D, and F, G share the feature maps sizes with their parent versions, if the given layers are present, but do not match the highlighted feature extraction.

Starting with the assumption that the problem of the version *A* is the position of the first feature extraction for detection, we moved the extraction from *block 2* to *block 7*. We also reduced the number of output filters on *blocks 2 to 7* from 728 to 256. As a result, the first detection is performed on a [19×19×256] feature map as opposed to the [37×37×256] map of *A*. The main factor here, it that the feature map is extracted after the data pass five additional blocks.

Due to a reduction of feature map size, we considered this modification a half-measure in our plan to move the first feature extraction deeper into the network. However, it proved to be a successful step in the right direction. The network has significantly gained in both speed and precision values.

Both versions *C* and *D* were designed for observation of the impact of the removal of parts of the network. We designed the modifications in such a way that they would not affect other layers. For clarity, the removals of blocks and layers do not alter the numbering scheme.

Version *C* omits *blocks 5, 6 and 7* and extracts the [19×19] feature map after *block 10* instead of *block 11*. However, we did not observe a satisfying performance boost to justify received loss of precision.

Version *D* was designed to test the need for six detection layers, mainly the layers for detecting large objects. It is based on *C* and removes the feature layers produced by *extra layers*. Although the observed performance boost from *C* to *D* was more significant than the one from *B* to *C*, it was also coupled with a major precision penalty.

Versions E, F, G

Similarly to the previous trio, these versions are also based on each other, with the main architectural change being applied in *E*. Versions *F* and *G* were designed to perform independent experiments.

Version *E* is the one, where we finally implemented our original intention of moving the first extracted feature map of [37×37] size to a deeper layer. Implementation-wise, the only difference between *B* and *E*, is the removal of max-pooling with stride 2 from *block 2* and placing it to *block 8*. This movement results in the increase of feature size to the desired [37×37] map up to *block 8*.

We observed a noticeable drop-off in performance compared to version *B* and a slight increase in precision. Although the number of parameters in the network is the same, the change from [19×19] to [37×37] requires about four times more computation per convolutional layer.

Version *F* modifies *E* in the same way, *C* modified *B*. It skips the *blocks 5, 6 and 7* and extracts the second feature map after *block 10*. Although the received performance boost in relation to *E* is significant, the precision is also significantly impacted in a negative way.

Version *G* repeated the experiment from version *D*, but instead of removing the last three detection feature maps, we removed every other one, thus keeping the first, third and fifth ones. It is based on version *F*, and again we see the precision loss we cannot justify by performance gain.

Version H

Since we managed to gain the best precision result with version *E*, we decided to try and increase its performance. To this end, we designed the version *H*. However, tests *C*, *F*, *D* and *G* showed that the removal of blocks or detection layers from the network is detrimental for the result. Therefore we decided for a less radical solution of trimming the channel depth of the network. We ended up trimming the [19×19] feature map to 512 channels and [10×10] map to 1024 channels.

Experiments show that this adjustment not only put the performance of the model halfway between *E* and *B* but also slightly boosted the precision.

Version J

After the success of version *H*, we decided to try the limits of channel removal approach. We started with the setup of *H*, and set the number of channels of every layer following *block 7* up to *extra layers*, to 512.

Version *J*, showed us, that this is also not a viable solution. The results are underwhelming in both, precision and performance.

Conclusion

In conclusion, we managed to receive the best precision results from the version *H*. Our first implementation, *A*, managed only 37.8% mAP on Surveillance dataset. Meanwhile version *H* achieved 49.8% mA on the same data while keeping the performance equivalent.

5.4 SSDTC Implementation and Training

We have already described the architecture of SSDTC in section 4.4; however, the actual implementation and training process proved to be more complicated than SSD. In this section, we go through the challenges brought by SSDTC.

As mentioned, we based the SSDTC on ResnNet34-SSD. We used the SSD initialized by the weights we learned on Surveillance dataset. We did not need the detection layers of SSD, so we removed them from the model. The resulting ResNet34 with *extra layers* served as a feature extractor for both SSDTC and also for training the baseline SSD on HollywoodHeads dataset. For training of both networks, we froze the weights in the extractor and trained only temporal and detection layers. It is important to note that every image processed by SSDTC, passes through the same extractor.

The SSDTC design we created, has different input data requirements for training and inference. To properly train, it needs a large number of smallest possible chunks in a batch for the network. However, one large chunk is the most effective way for inference and the most natural way as we expect a continuous video stream.

This, however, poses conflicting requirements for the implementation of the network, namely in the passing of data between the modules. For the purposes of explanation, lets call the ResNet34 network with *extra* layers the *Extractor*, the two conv3d layers the *Temporal module*, and the localization and classification layers the *Detection module*. We will also consider the detection process for only one of six feature map layers of the Extractor.

During training, the network starts with feature extraction on seemingly independent $n*c$ frames in a batch (c being the minimal possible chunk, in our case 5, and n the batch size). Coming to the temporal layers, the feature maps need to be reshaped to represent n chunks of size c to allow for conv3d layers. After the temporal layers, the feature maps come out in the batch size of n, but due to the nature of temporal layers, the chunk size is reduced to 1. This data then has to be reshaped again, to remove the temporal dimension, and create a batch of size n for the detection layers.

On the other hand, during inference, the input is a single chunk of consecutive frames, that is equivalent to a single batch. This batch can be simply passed thorough feature extraction layers, and the only modification needed for the temporal layers is to encapsulate the whole tensor in additional dimension to represent a temporal batch of size one. After the temporal convolution, we remove the encapsulation and continue with the detection; however, the batch size in this step is smaller than at the beginning.

We can see the difference between training and inference feature sizes in table 5.4. This behavior forced us into two separate implementations for inference and training. Although both implementations share the same network models, the differences are in the handling of the data between the modules. In the table, we can see that during training, it is the *chunk dimension* that gets eliminated in temporal layers, and during the inference, it is the *batch dimension.*

Module	Training	
	Input	Output
Extractor	[$H_0 \times W_0 \times 3 \times 5$*N]	[$H_1 \times W_1 \times F_1 \times 5$*N]
Temporal	[$H_1 \times W_1 \times F_1 \times 5 \times N$]	[$H_2 \times W_2 \times F_2 \times 1 \times N$]
Detector	[$H_2 \times W_2 \times F_2 \times N$]	
	Inference	
	Input	Output
Extractor	[$H_0 \times W_0 \times 3 \times C$]	[$H_1 \times W_1 \times F_1 \times C$]
Temporal	[$H_1 \times W_1 \times F_1 \times C \times 1$]	[$H_2 \times W_2 \times F_2 \times C-4\times 1$]
Detector	[$H_2 \times W_2 \times F_2 \times C$-4]	

Table 5.4: Data shapes and sizes on the input and output of the SSDTC modules. The output of the Detector complies with SSD definitions (section 3.2.2).

Although it is possible to use both versions for training and inference, each has a significant disadvantage if not used as intended. The inference module does not allow for the use of the batch normalization in conv3d layers, because it operates with temporal volume in a batch size one. The fact that the ability to use batch normalization is vital for the training was also experimentally tested. Without normalization, we were not able to overperform standard SSD with the temporal version. There is a better chance for the training module to be modified for efficient inference, although with some performance hit inherent from our implementation.

Conclusion

Our work can be summarized in five segments. We started by reviewing related work and other relevant image processing models based on deep learning. We continued by weighting the options for improving SSD, where we decided to replace VGG feature extractor by a more modern network. Then we took a look at the relationship between detected classes and detector performance. In the fourth part, we returned to improving SSD, this time we focused on one base network, and instead of using the classification network as is, we make a series of adjustments aimed at improving the performance. We dedicated the final part of this work to designing and implementing a version of SSD detector with the use of temporal information.

Model review

We began by briefly summarizing the development of classification networks by presenting a selection of models while describing their architecture and historical significance. After we covered the bases of image processing, we moved on to region-based object detection and compiled a brief overview of R-CNN family of networks.

Since the focus of this thesis is to optimize the SSD detector for video surveillance, we presented an in-depth examination of this detector. However, it would not be right to present SSD without mentioning its contemporary counterpart, YOLO. Because both SSD and YOLO are one-stage detectors and our experience shows that the one-stage approach can be challenging to comprehend, we compiled a detailed examination of this approach.

In order to gain more extensive knowledge about video detection and inspiration for our work, we also examined a pair of video detectors exploiting the temporal information in the video.

Feature Extraction Network

We know, from the research paper introducing SSD, that the precision of the model can be improved by implementing a sophisticated data augmentation algorithm. Since augmentation algorithms slow down the training process, we decided to advance with a fast and simple augmentation and instead focus on the relative comparison of model performances. To obtain a benchmark baseline, we began our work by re-implementing SSD in PyTorch framework and trained it on COCO dataset.

Our first steps towards optimizing SSD led to a search for a replacement for the underlying outdated VGG classifier. We tried out ResNet networks, Xception and NASNet-Mobile. The testing revealed that all versions of SSD based on ResNet are capable of outperforming standard SSD, but Xception and NASNet only outperform VGG in terms of speed. Although this result suggested ResNet as a clear winner, achieving 32.7% mAP and 50fps over original 29.7% mAP and 45fps, we were not satisfied with all the results and further pursued the possibilities for improvement, mainly for the Xception-SSD.

Classes

Before we continued with testing of architecture modifications, we decided to take a side-step toward the training data. As stated before, our baseline SSD is trained on COCO dataset. However, we are not interested in every class provided by COCO. As a matter of fact, we are only interested in seven of those eighty classes.

Before we moved to this smaller dataset, we took this opportunity and performed two tests concerning the impact of limiting the number of detected classes. The first experiment was based on our hypothesis, according to which the removal of the unwanted classes from the training process can harm the precision of the detector. The second test was to observe the relationship between the number of detected classes and the speed of the network.

The results of the precision test on were inconclusive for our test setup and did not favor one dataset over the other. However, the inference speed test clearly shows the benefits of a lower number of classes. With ResNet50-SSD, we managed to speed up the network from 50 fps on 80 classes to 123 fps on 7 classes. Based on the results of those tests, we decided to continue further testing with this limited dataset.

Model Modifications

Although ResNet was the best performer in our first tests, we decided to try and improve the Xception-based SSD, since Xception looked promising in classification benchmarks and our SSD implementation showed disappointing precision. We already ruled out the option of keeping both SSD and Xception unmodified. We performed some experiments with modified detection but did not receive any positive results. Therefore we decided to adjust the feature extraction part of the network to fit the needs of SSD better.

We took an experimental approach and designed multiple versions of Xception to test. After a few iterations, we arrived at the Xception version H. This version managed to reach 49.8% mAP and perform on par with

ResNet50-SSD's 48.7% mAP while being about 15% slower.

Although this experiment did not result in the model that would manage to outperform ResNet substantially, we learned a lot about the relationship between classification models and detection ones. The fact is that not every state-of-the-art classification network is fit to serve as a feature extractor for an object detector like SSD, mainly if said detector uses multiple feature maps at different scales. Some networks like NASNet-Mobile may be downright unfit, and others need adjustment to fit the needs of the detector.

Temporal Detection

In the final chapter of this thesis, we turned our attention to video detectors with the use of temporal information provided by the continuity of surveillance video. Based on the knowledge we gained from the experiments with SSD and reviewing the temporal models, we designed our version of the temporal detector. We based it on ResNet34 feature extractor, with SSD-like detection, extended by three-dimensional temporal convolution for consecutive frames. We named our approach *Single Shot Detector with Temporal Convolution* (SSDTC).

Testing our design, we observed a significant precision improvement over regular ResNet34-SSD on HollywoodHeads dataset. SSD reached 81.2% mAP, while SSDTC managed to achieve 86.8%. This result is comparable to 86.7% of reviewed Tube-CNN. Furthermore, Tube-CNN performs at less than 10 fps which is significantly less than our speed of 111 fps.

We believe our model to be a successful proof of concept and see a lot of opportunities for future work to further enhance its performance.

Future Work

Although our experiments were successful in both improving the SSD detector and proving the viability of more advanced temporal detection, it left a lot of room for further improvements. Here, we provide a few suggestions for future work.

- Explore further optimizations of ResNet architecture for SSD.
- Improve the precision by expanding the dataset and implementing a more sophisticated and varied augmentation algorithm for training.
- Explore the options of redesigning the SSDTC architecture to avoid repetitive processing of frames.

- Measure the impact of temporal window size in SSDTC and a possibility of adding padding in the temporal dimension of three-dimensional convolution.

- Speed up the detection through implementation improvements. For example, implement pre- and post-processing operations on GPU or re-implement the network in a fast framework optimized for inference, e.g., TensorRT.

Applications

In the introduction, we mentioned many interesting and useful applications of object detectors. In this section, we present a selection of additional applications that directly use object detection and build on top of it to generate more complex information about the processed video.

Tracking Thanks to the nature of the video, we expect to be able to locate and track objects in time, observe their direction and speed. The purpose of the tracking is to connect detections from individual frames into a set of continuous trajectories. A tracker can use the centralized bounding-box positions and match those points into a trajectory using a recursive probability estimation, i.e., the Kalman filter [15].

Re-Identification Re-identification is an extension of the tracking problem to objects passing between fields of view of multiple cameras. Regions inside bounding boxes predicted by an object detector can be used by the DeepReID [17] or other similar algorithms, to find the matching object.

Search An automatic annotation of video frames by an object detector can also be used for interactive video retrieval [19] in large video collections. For example, the ability of the detector to learn to recognize a set of classes with bounding boxes can be used for sketch based search [29] [1].

Bibliography

[1] Giuseppe Amato, Paolo Bolettieri, Fabio Carrara, Franca Debole, Fabrizio Falchi, Claudio Gennaro, Lucia Vadicamo, and Claudio Vairo. Visione at vbs2019. In *International Conference on Multimedia Modeling*, pages 591–596. Springer, 2019.

[2] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] Xingyu Chen, Junzhi Yu, and Zhengxing Wu. Temporally identity-aware ssd with attentional lstm. *IEEE transactions on cybernetics*, 2019.

[5] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[6] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[7] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In Cordelia Schmid, Stefano Soatto, and Carlo Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition (CVPR '05)*, volume 1, pages 886–893, San Diego, United States, June 2005. IEEE Computer Society. doi: 10.1109/CVPR.2005.177. URL `https://hal.inria.fr/inria-00548512`.

[8] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[9] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

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

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[14] Shaoqing Ren Kaiming He, Xiangyu Zhang and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, 2016. IEEE Computer Society. ISBN 978-1-4673-8851-1.

[15] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82 (Series D):35–45, 1960.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Wei Li, Rui Zhao, Tong Xiao, and Xiaogang Wang. Deepreid: Deep filter pairing neural network for person re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 152–159, 2014.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[19] Jakub Lokoč, Werner Bailer, Klaus Schoeffmann, Bernd Muenzer, and George Awad. On influential trends in interactive video retrieval: video browser showdown 2015–2017. *IEEE Transactions on Multimedia*, 20 (12):3361–3376, 2018.

[20] Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 850–855. IEEE, 2006.

[21] Rafael Padilla. Object-Detection-Metrics: Object Detection Metrics Release v0.2, January 2019. URL `https://doi.org/10.5281/zenodo.2554189`.

[22] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[23] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[24] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[25] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[26] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[28] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[29] Thanh-Dat Truong, Vinh-Tiep Nguyen, Minh-Triet Tran, Trang-Vinh Trieu, Tien Do, Thanh Duc Ngo, and Dinh-Duy Le. Video search based on semantic extraction and locally regional object proposal. In *International Conference on Multimedia Modeling*, pages 451–456. Springer, 2018.

[30] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

[31] Paul Viola, Michael Jones, et al. Rapid object detection using a boosted cascade of simple features. *CVPR (1)*, 1:511–518, 2001.

[32] Tuan-Hung Vu, Anton Osokin, and Ivan Laptev. Tube-cnn: Modeling temporal evolution of appearance for object detection in video. *arXiv preprint arXiv:1812.02619*, 2018.

[33] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810, 2015.

[34] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

A. Implementation

This appendix provides an overview of the implementation we have used for the experiments. We describe the distribution of the source code into modules and specify the runtime environment requirements. We also provide a short guide into running the code and training the network.

A.1 Code structure

The logical structure of the implementation is illustrated in fig. A.1. The figure shows different modules used for training and inference, including the shared core. This entire structure is common to both SSD and SSDTC, except for SSDTC training, where the `ssdtc.py` replaces the `ssd.py` file.

Physically, the code is organized in the `ssd/` folder as follows:

lib/ SSD and SSDTC implementations with loss function, prior box generation module and other utility functions.

lib/data/ Dataset classes for loading the data and annotations for both COCO and HollywoodHeads datasets and an augmentation module.

lib/models/ Detector network models.

train/ Training scripts.

test/ Inference and evaluation scripts.

A.2 Runtime Environment

We performed our experiments on a remote server with the use of *Docker*[1] containers for deployment. The use of the GPUs requires an addition of the *nvidia-docker*[2]. Unfortunatelly, *nvidia-docker* is currently supported only on Linux platforms, of which we tested *Ubuntu 16.04* and *18.04*.

The following list of the software requirements is stringent because of the *nvidia-docker* limitations. On the other hand, the hardware limitations are hard to specify, the faster the GPU and CPU the better. CPU is responsible for feeding the data into the network and their augmentation. The speed of

[1]`https://www.docker.com/`

[2]`https://github.com/nvidia/nvidia-docker/wiki/Installation-(version-2.0)`

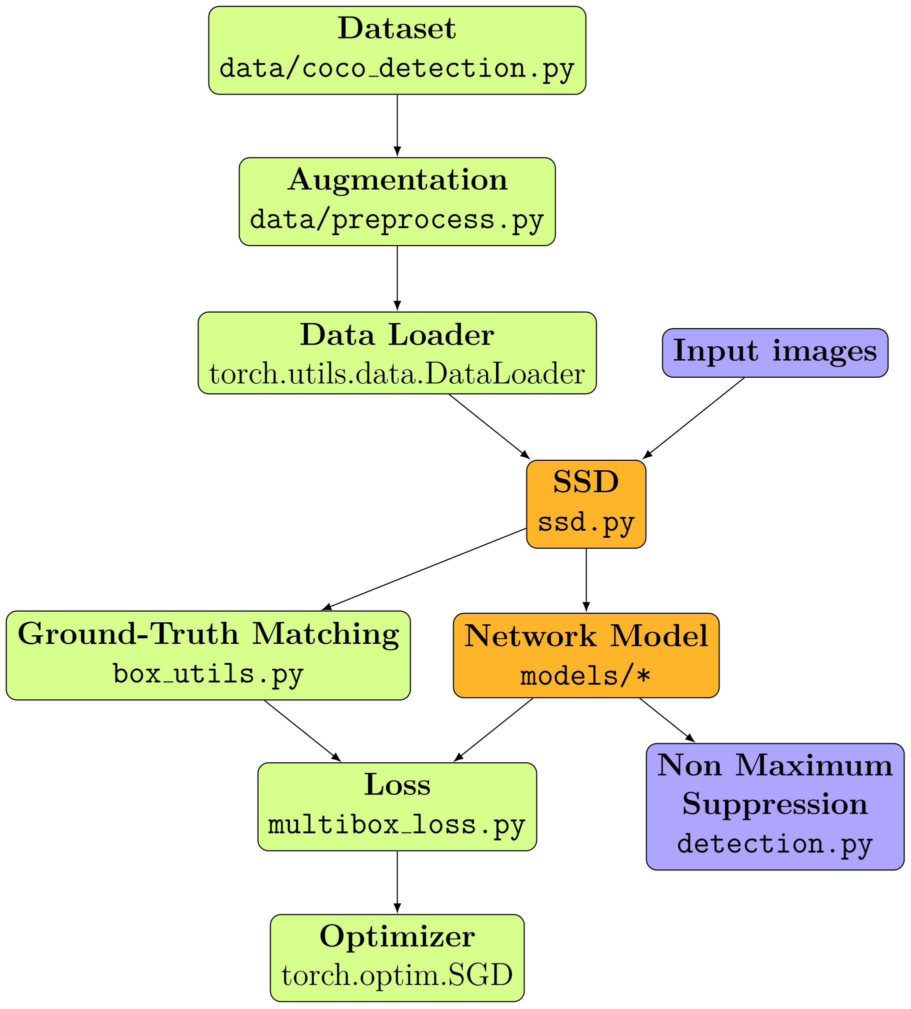


Figure A.1: Implementation pipeline of SSD modules. Training pipeline (green) and inference pipeline (purple) with shared core elements (orange).

the storage medium then limits the CPU and so on. Nonetheless, the optimal setup should be bottlenecked by the GPU. The size of the GPU memory is a limiting factor for the model size and the batch size.

Software requirements	
NVIDIA driver	>=396
docker-ce	18.06.1
nvidia-docker	2.0

Minimal HW	**requirements**
NVIDIA GPU	6 GB
RAM	16 GB
CPU	8 threads

We strongly recommend using this project via a Docker container, however, it is possible to follow the steps inside provided Docker files to set up the native environment (`docker/ssd` and `docker/pytorch`). The unavoidable fact is, that compatible versions of *PyTorch*, *CUDA Toolkit*[3], *cuDDN* library[4] and operating system are needed. Our implementation uses *PyTorch 0.4.1* with *CUDA 9.2*, *cuDDN 7* in a *Ubuntu 16.04* based Docker image.

We streamlined the process of building the Docker container into running a single `docker_build.sh` script.

With the image built, we can start the container:

```
docker run -it --ipc=host --runtime=nvidia -v
   data_loc:/data -v save_loc:/save ssd:v1.0
```

We recommend mounting the directory with training or input data and the directory for output data into the Docker container.

A.3 Training and Evaluation

In `ssd/train/` and ssd/test/, we provide all scripts used to get the results in this thesis and a few additional inference modules.

Training

Assuming the previous steps were followed, we can change into a *SSD* directory inside the running Docker container and train the network by executing the following:

```
python3 -m ssd.train.train_resnet_small --size 50
   --export /save/ --loc /data/ --resume imgnet
```

[3]`https://developer.nvidia.com/cuda-toolkit`
[4]`https://developer.nvidia.com/cudnn`

Separate scripts are provided for each model and each dataset with almost identical parameters. The most important is to specify dataset location (`--loc`), location for saving the trained weights (`--export`) and the loaded checkpoint (`--resume`). The ResNet models are further specified by `--size` parameter and Xception by `--type` parameter. It is also important to set batch size (`--batch`) depending on the available memory.

Evaluation

Evaluation script is run very similarly to the training one, except for the need to specify the network type as the parameter:

```
python3 -m ssd.test.eval --net resnet --size 50 --
    loc /data/ --resume weights --batch 16
```

The evaluation is not set up to calculate the precision by itself, but instead to export detections and ground-truth boxes into `detections` and `groundtruths` folders. The exported data are then evaluated by external algorithm (`https://github.com/rafaelpadilla/Object-Detection-Metrics`).

Inference

We have prepared a two-part inference module. One part processes the video and outputs a file with the detections. The other one takes the video and the generated file and produces a new video with detected boxes drawn on. Those files are `test/detect_video.py` and `test/tag_video.py` and the user guide can be found inside those files.

List of Figures