VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# DETECTION OF FIRE IN VIDEO

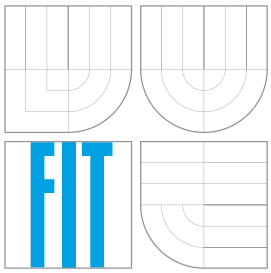
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ POLEDNÍK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# DETEKTOR OHNĚ VE VIDEU

DETECTION OF FIRE IN VIDEO

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ POLEDNÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. ADAM HEROUT, Ph.D.

BRNO 2015

# Abstrakt

Tato diplomová práce se zabývá detekcí ohně ve videu pomocí farební analýzy a strojového učění, konkrétně hlubokých konvolučních neurónových sítí, použitím nástroje Caffe. Cílem je vytvoření velké sady dat, která může sloužit jako základní prvek detekce založené na strojovém učení a vytvoření detektoru použitelného v reálné aplikaci. Pro účely projektu byla navrhnuta a vytvořena sada nástrojů pro tvorbu sekvencí s ohněm, jejich segmentaci a automatickou anotaci spolu s velkou trénovací sadou krátkých sekvencí umělo vymodelovaného ohně.



# Abstract

This thesis deals with fire detection in video by colour analysis and machine learning, specifically deep convolutional neural networks, using Caffe framework. The aim is to create a vast set of data that could be used as the base element of machine learning detection and create a detector usable in real application. For the purposes of the project a set of tools for fire sequences creation, their segmentation and automatic labeling is proposed and created together with a large test set of short sequences with artificial modelled fire.



# Klíčová slova

Detekce ohně, zpracování obrazu, video sekvence, strojové učení hlbokými konvolučními neuronovými sítěmi, počítačové vidění, Caffe, modelování ohně, kompozice scény s ohněm

# Keywords

Fire detection, image processing, video sequence, machine learning by deep convolutional neural networks, computer vision, Caffe, fire modelling, fire scene compositing


# Citace

Tomáš Poledník: Detection of Fire in Video, diplomová práce, Brno, FIT VUT v Brně, 2015

# Detection of Fire in Video

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Ing. Adama Herout, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Poledník
May 27, 2015

## Poděkování

Děkuji panu Doc. Ing. Adamovi Heroutovi, Ph.D. za čas, materiály a odbornou pomoc, kterou mi věnoval při řešení práce.

# Contents

# Chapter 1

# Introduction

This work serves as an alternative method to ordinary fire detection using short-range smoke and heat sensors. The main goal is to try a modern way of detection in image and video based on machine learning with deep convolutional neural networks (DCNNs). As DCNNs are new in this field and are considered extremely powerful (for instance thanks to the results from the work of Krizhevsky et al. [1]), it would be interesting to see how much they would succeed in fire detection. Fire is not an easy thing to record, so finding a way to create such records artifically is another motivation. It is therefore a challenge — creating a real fire detector that never saw a real fire.

The problem can be divided into two core tasks. The first one involves creating a large dataset of fire and non-fire samples and using these to train a model of deep convolutional neural network. The second task is to make a software detector that analyses image (video) input, tests its content on the pretrained model and returns a result. The proper solution would be a detection that finds fire in a video, if present, as soon as possible — within 1 to 15 seconds. My personal goal was for the number of true positives to exceed 95%.

I recorded several static scenes with a camera. The best way I found to gain realistic fire effect was to use a 3D modelling tool, specifically Blender[1]. With it, I modelled the scene with a burning fire, rendered it and pasted it onto the frames of the recording. My solution involves experiments on images and videos based on two approaches to created fire sequences and their initiation into a model's training — per image and per segment and the comparison of their results. I created three standalone universal applications, a compositor, a sampler and a database generator, which were used to create and apply training data according to a given approach. The model was created, trained and tested using Caffe framework [2] and my detector. I also employed fire's colour analysis as an addition to the detection. The detector's per segment approach trained only on modelled fire showed that it can be used for real fire detection with 100% true positives.

Analysis of different works on fire detection, which were used during this work's creation, can be found in Chapter 2. The basic concepts of DCNNs used for fire detection are described in Chapter 3. Chapter 4 describes fire data creation. The detector's description is contained in Chapter 5 and the experiments can be found in Chapter 6. Chapter 7 sums up the achieved results and dicusses the ways of continuation.

This thesis is a continuation of my term project where I proposed and implemented fire rendering and compositing from Chapter 4. Most of the existing approaches to fire detection in Chapter 2 also come from this project.

[1] Blender Online Community: http://www.blender.org/

# Chapter 2

# Existing Approaches to Fire Detection in Video

There are many methods for fire detection in image and video. This chapter contains the summary of those that I used during this work's creation. Every method's detection is based on one or more visible qualities of fire. These include its colour, shape, motion, frequency, spatial change and smoke generation. The summary is divided into multiple sections by the detected qualities. A complete analysis of fire's behavioural and visual traits can be found in Qunitiere's Principles of Fire Behaviour [3].

## 2.1 Colour

Fire's colour belongs to the most frequently detected fire traits as it is the most distinguishable.

### Fire Colour Analysis Using RGB and HSV Colour Models

Chen et al. [4] detect colour properties of fire described in Qunitiere's book [3]. The main property is fire's transition from red, through yellow and up to white colour. As white represents a very frequent colour, it is frequently excluded from detection. The watched colours are therefore red and yellow. When using RGB colour model, these colours correspond to channel values given by Rule 2.1.

$$\begin{aligned} B(x,y) &< G(x,y) \\ G(x,y) &< R(x,y), \end{aligned} \tag{2.1}$$

where $R(x,y)$, $G(x,y)$ and $B(x,y)$ represent the values of colour components $R$, $G$ and $B$ of a pixel at position $(x,y)$.

During the night, when fire becomes the only light source, $R$ component becomes the most distinguishable. Chen et al. [4] propose comparison of $R$ component's value to a needed minimal value, a threshold $R_T$, which they acquired through numerous experiments. As background lighting can also change and adapt, which may affect all the colour components of RGB model, HSV's $S$ component is also observed. This component also has its preset threshold $S_T$ which represents the saturation of a pixel when $R = R_T$. If $R$ component's value rises, saturation falls — this can be expressed using rule $S \geq ((255-R)\cdot S_T/R_T)$. If all of these conditions are met, the observed pixel is classified as fire. Otherwise it is a non-fire pixel.

### Fire's Colour Analysis Using a Combination of Normal Distribution Models in RGB

A trained colour model is used for detection in the work of Töreyin et al. [5]. Precalculated distribution of possible fire colours in RGB model space is compared to the colour of each pixel. The decision model is created using a combination of several normal distributions trained by a large set of manually annotated images. If a pixel's colour's value falls to at least one of the two given distributions, it is classified as a fire pixel.

### Analysis of Fire's Grayscale Representation and its Colour Using HSV Model

Analysis of fire's colours, together with their specific locations inside a fire area, is carried out by Liu et al. [6]. Liu et al. observed that each of fire's colours can be distiguished by its typical relative position. This can be seen in Figure 2.1. The flame's centre (core) is always the brightest spot. Moving from the centre this colour becomes yellow, orange and red. There may be a situation when fire has more than just one core which is also shown in Figure 2.1.

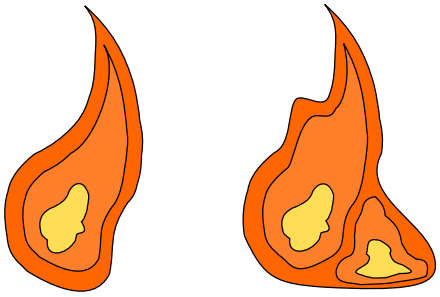


Figure 2.1: Outlines of different coloured areas in a fire with one core on the left and a fire with two cores on the right.

The first step, when looking for fire regions, is marking the brightest — whitest — areas which could present the fire's cores (Liu et al. refer to them as seeds). This marking is done in a grayscale image. Every seed is then expanded up until it covers all the surrounding pixels which can be classified as fire (white, yellow or red colour) with a high probability. The probability density functions are modelled using a combination of Gaussian distributions in HSV model colour space taken from the work of Yang et al. [7]. The last step is checking the pixels which form the outlines of the whole fire regions. If at least half of these are not classified as fire by the trained model, the entire region is excluded from the detection. This erases big white spots from the detection.

### Pixel Colour Classification Using YCbCr Model

The use of YCbCr model is the main part of Çelik's and Demirel's work [8]. They reuse the colour detection rules stated in the work of Chen et al. [4]. Fire in Figure 2.2 in RGB space shows the behaviour of individual colour channels. Red component's value is always higher than green component's value which is always higher than the blue component. As

for the colour's intensity, the red channel's intensity is always the highest and green and blue go after it.

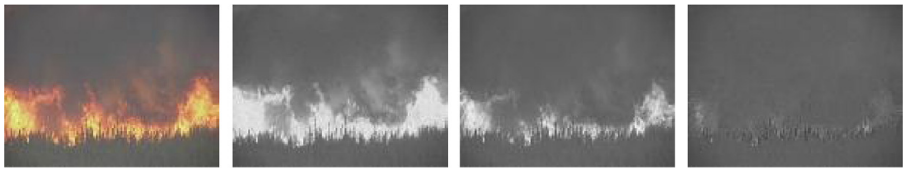


Figure 2.2: RGB image containing fire in the left column and its individual channels $R$, $G$, $B$ in this order. Image adapted from article [8].

Even though these rules apply most of the time, they are very sensitive to the scene's lighting. If this lighting changes, they continually become useless. As RGB's components do not carry the necessary information about the colour's intesity, needed when analysing the change of lighting, Çelik and Demirel incorporate YCbCr model into their method.

The rules for fire pixel classification for RGB model $R \geq G > B$ from work [4] can be rewritten to $Y > Cb$ and $Cr > Cb$ when using YCbCr. This means that fire represents an area of the highest luminance and its red colour dominates. The average values of $Y$, $Cb$ and $Cr$ components of the whole image are also valuable pieces of information for the detection. Fire region is always the brightest and most of the time its red colour prevails over the red colour of the rest of the scene. This means that fire pixel's $Y$ and $Cr$ components will always be higher than image's average. On the other hand, $Cb$ component's value is always below average.

Figure 2.3 shows that fire's $Cr$ component's values are mostly white, while $Cb$'s are predominantly black. The result of their subtraction is always a positive integer which is higher than a specified threshold. This threshold value was acquired by analysing a large testing set of images — the value that achieved the best results was $\tau = 40$.

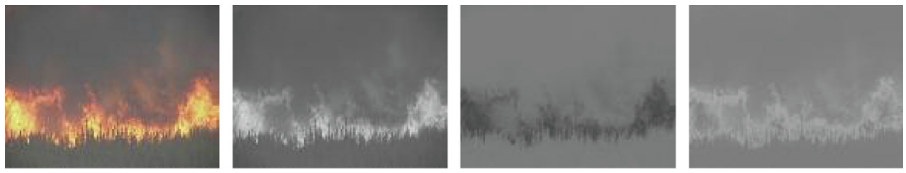


Figure 2.3: YCbCr image containing fire in the left column and its individual channels $Y$, $Cb$, $Cr$ in this order. Image adapted from article [8].

To decrease the number of false detections Çelik and Demirel also used statistical analysis of fire's colour trained by their test set. The output was a distribution of $Cb$ and $Cr$ components' values modelled by 3 polynomials shown in Figure 2.4. If a pixel's component's values fall in the area bounded by these polynomials, it is considered a fire pixel.

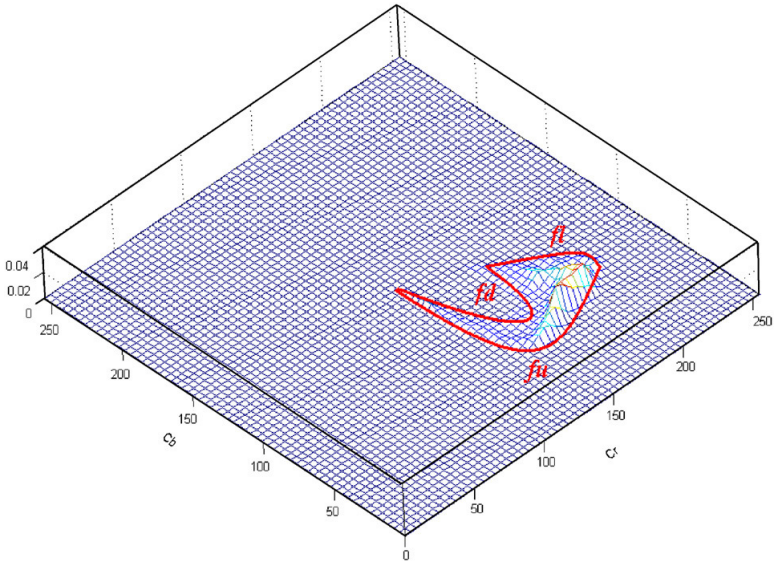



Figure 2.4: 3-dimensional display of the $Cb - Cr$ channel plane (YCbCr colour space) and $fu(Cr)$, $fl(Cr)$ and $fd(Cr)$ polynomials that restrict the values of these channels for a colour of fire. Image adapted from article [8].

### Fire Colour Analysis Using Pretrained Rules for RGB Components

Çelik et al. [9] use colour classification based on a trained decision model in RGB space. They also modified the rule from the work of Chen et al. [4], Rule 2.1, adding $R(x, y) > R_{mean}$ to it, where $R_{mean}$ is the average value of $R$ component of all pixels on fire's background. The effects of lighting's change can also be eliminated by using different set of rules pretrained by a large testing set containing scenes with various types of lighting.

## 2.2 Shape

### Assembling Fire's Shape Representation Using Fourier Transform

Fire's shape is one of the detected features in the work of Liu et al. [6]. The shape of every fire region is given by its border. This border is represented by one-dimensional signal composed out of $N$ points — pixels of fire region's border. Every point $z_i : i \in 1..N$ is expressed in a form of a complex number $z_i = x_i + j_{y_i}$, where $x_i$ as the x-coordinate of a point in an image and presents the real component and $y_i$ is the imaginary component. Fire's shape can therefore be described using a Fourier Descriptor created out of coefficients $a_k$. These are received as output of Discrete Fourier Transform (DFT) according to the article of Persoon and Fu [10].

## 2.3 Motion

These methods segment the processed frame into foreground, which represents moving objects, and background.

### Moving Region Detection Using Adaptable Method of Frames Subtraction

Töreyin et al. [5] detect moving regions using a modified method of background subtraction. This method was first described by Collins et al. [11]. To decide whether a pixel belongs to the foreground, its intensity component $I$ of HSI colour model is compared and subtracted in two subsequent frames. If the result value reaches given intensity threshold, it is considered foreground. This threshold's value adapts itself after every frame using the rules described in the work of Collins et al. [11]. The intensity values of foreground pixels in every new frame adapt as well. Their actual value is partially balanced by their previous one.

### Creation and Modification of Background Model from Colour Component's Models

Çelik et al. [9] detects fire's motion by creating and updating a model of background. This method was first described in the work of Wren et al. [12]. In order for this method to work, the monitored scene must be filmed with a static camera.

RGB colour model is used for this method. Video's background is modelled using normal distribution. Every colour channel of every pixel is modelled individually. The complete colour model of each pixel is calculated from the models of its colour channels and can be defined as follows.

$$p(I(x,y)) = p_R(I_R(x,y))p_G(I_G(x,y))p_B(I_B(x,y)), \tag{2.2}$$

where $p_R$, $p_G$ a $p_B$ represent models of pixel's distributions of $R$, $G$ and $B$ components, while $I_C(x,y)$ represents the value of channel $C; C \in R,G,B$ at position $(x,y)$. $p(I(x,y))$ is an approximation of probability density function of pixel's values at that position.

As everything is modelled using normal distribution, at the beginning all the models need to be initialized with some values of mean and standard deviation. Therefore, first frames need to be used for a training phase when the models can learn about the scene.

After the training phase motion detection can start. If at least two of pixel's channels fall outside of the trained model's values, pixel is classified as foreground.

As dynamics of the scene may be affected by any change in lighting, models' parameters are updated every new frame using a predefined weight. This weight states how much the new frame affects the model.

## 2.4 Frequency

### Temporal Wavelet Analysis

History of fire pixel frequencies is watched by Töreyin et al. [5]. Spatial wavelet analysis is used for frequency scanning. It can detect flickering (oscillation) of fire pixels. Qunitiere's Principles of Fire Behaviour [3] mentions that frequency of fire's oscillation is between 1 and 10 Hz.

Temporal wavelet analysis uses the value of RGB's $R$ component ,which presents fire's dominant colour, to measure the frequency of pixel's oscillation. For the analysis to be able to capture a pixel's frequency between 1 and 10 Hz, the video's frame rate must be at least 20 frames per second. In case this frame rate cannot be reached, the analysis may fail to detect the correct frequency. Every pixel is subjected to wavelet analysis individually. The

analysis output is a set of one-dimensional signals representing temporal change of every pixel. High frequency detection is provided by high-pass filters. If a flicker appears, the output of the high-pass is a non-zero high-frequency signal.

## 2.5 Spatial Change

### Fire Border's Temporal Shape Change Detection

In every single frame Liu et al. [6] calculate the coefficients of Fourier series of the border of every fire region created out of $N$ points. After that it is observed how these coefficients change in time. Liu et al. presume fire's basic shape — its low-frequency components — stays the same. The lower coefficients of Fourier series, for instance $a_{-1}$, $a_0$ or $a_1$, will exhibit only a little change or none. However, locally (at points very close to the fire's border) fire changes very rapidly. High coefficients, for instance $a_{-N/2}$, $a_{(-N+1)/2}$ or $a_{N/2}$, will exhibit large change and therefore can be detected. Examples of this detection's output can be seen in Figure 2.5.

Figure 2.5: Experimental results of fire outlines detection. Image adapted from article [6].

### Flame's Spatial Change Analysis by its Growth and Disorder Observation

Chen et al. [4] analyse spatial change of flames when burning. This change is affected by both the flame's rate of growth and its random movements. The degree of disorder is analysed by comparing the amount of fire pixels in a sequence of frames. The result is acquired by subtracting the amounts of every pair. This value is normalized by the amount of fire pixels of the previous frame and then compared to a predefined threshold. This threshold was acquired experimentally.

The rate of growth is calculated the same way as its disorder. Instead of comparing the amounts of fire pixels in subsequent frames, the comparison is done between a frame at the beginning of a time interval of specified length and the current frame. If the amount of fire pixels is higher in the current frame for $N$ consecutive frames, the analysed pixels will be considered fire.

### Spatial Wavelet Analysis

Töreyin et al. [5] implement spatial change monitoring. Their aim is to find information about fire region's spatial change using wavelet analysis. Spatial change is caused mainly by turbulent and random motion of pixels which is one of fire's features. Any other object should exhibit only very subtle or no random motion. The analysis is very similar to temporal wavelet analysis. As an image represents a two-dimensional signal, it needs to be subjected to two-dimensional wavelet transform. High-pass and low-pass filters are applied to the transform's output. These exclude components that are outside fire's frequency band. Spatial change ratio of a flame's region is gained by summation of squares of all the fire region's pixels and dividing them by their count. If this computed value raises above certain threshold, the region is considered fire.

An example of a spatial wavelet transform's output can be found in Figure 2.6.

Figure 2.6: Marked fire area and its spatial wavelet transformation's output on the right. Image adapted from article [5].

## 2.6 Smoke

### Using Chromatic and Dynamic Analysis for Smoke Detection

Detection of gray smoke is a part of the work of Chen et al. [4]. Smoke's gray colour can be split into two groups — light-gray and dark-gray colours. As all of the RGB components are of the same value when analysing the colour they do not bear much information. For more effective classification $I$ (intensity) component of HSI model is used. Both colour groups are given their starting and ending values — thresholds — of their $I$ components. These thresholds were acquired experimentally. This way the smoke pixels can be separated from the rest of the image. The result are two rules, $L_1 \leq I \leq L_2$ and $D_1 \leq D_2$, where $L_1$ and $L_2$, $D_1$ and $D_2$ are the thresholds of light-gray and dark-gray groups. Dynamic analysis is carried out the same way as when analysing fire's dynamics.

### Smoke Detection by Image Separation

Tian et al. [13] use a technique of image separation to detect smoke in video. A background model is constructed and based on this model, video frame's background is separated from the image leaving only a possible smoke component. This component is defined by its partial transparency $\alpha$ which, when reaching values $\alpha > 0$, makes it a smoke candidate. Candidate's features are extracted and sent to a pretrained model for classification.

# Chapter 3

# Using Deep Convolutional Neural Networks for Fire Detection

In this chapter the basic concepts of deep convolutional neural networks are described together with examples of their use.

These neural networks were primarily used for feature construction on 2D images but have since been used for many different problems. Their performance was demonstrated on tasks such as hand-written digit classification by LeCun et al. [14] resulting in LeNet convolutional network and colour image classification in articles [15, 1, 16] resulting in AlexNet and GoogleNet. Long et al. [17] used them for pixel-wise image segmentation and object detection. In the work of Ji et al. [18] they were also used for human action recognition on video.

## 3.1 Network's Architecture

As stated by van Doorn [19] deep convolutional neural networks are feedforward (every layer's outputs are connected only to inputs of its adjacent layer) neural networks with many hidden layers. Convolutional neural networks differ from normal neural networks in that neurons in a layer are connected to the following layer only sparsely. These connections are related to a neuron's relative position in its layer.

According to article [17], each layer of a convolutional network is 3-dimensional $w \times h \times d$ array, where $w$ and $h$ present the width and height and $d$ is either the channel depth (for the input layer) or feature depth. The first layer (input) has size of $w \times h$ and $d$ colour channels for images as demonstrated by Krizhevsky et al. [1]. LeCun [14] originally used $1 \times 1 \times d$-dimensional vectors containing information about the handwritten digits as the network's input.

Krizhevsky et al. designed a network that is formed by 5 convolutional layers and 3 fully-connected layers and The whole architecture can be seen in Figure 3.1.

Overview of the most used convolutional neural network components follows.

### Use of Backpropagation

As described in article [19], backpropagation is a method of applying a change in model's weights by propagating the output errors back through feedforward architecture of the network. Neural network's training controlled by backpropagation runs a training sample

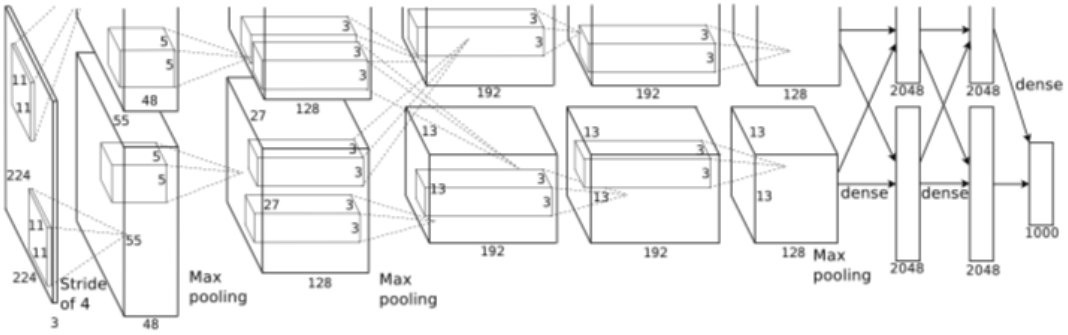



Figure 3.1: Deep convolutional neural network structure used in the work of Krizhevsky et al. [1] for ImageNet classification. The network consists of 5 convolutional layers and 3 fully-connected layers ending with a softmax classifier that produces distribution over 1000 labelled classes. The first convolutional layer filters $224 \times 224 \times 3$ images with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels in the input image. Second convolutional layer takes first layer's normalized and pooled outputs and filters them with 256 $5 \times 5 \times 48$ kernels. The third has 384 $3 \times 3 \times 256$ kernels that are connected to normalized and pooled outputs of the second layer. The third, fourth and fifth convolutional layers are connected without pooling or normalization. The fourth layer has 384 and the fifth 256 $3 \times 3 \times 192$ kernels. Fully-connected layers have 4096 neurons each. Image adapted from article [1].

through the network and outputs a result based on the used classifier — feedforward computation. Classification error is computed based on the result and the expected (partial) result and this error runs backwards accross the architecture changing weights of neurons in every layer starting with the first hidden layer — backpropagation. The weights are updated based on these factors:

1. resulting error
2. learning rate
3. neuron's activation gradient

### Convolutional Layer

Convolutional network's main components are its convolutional layers. Description according to Jia et al. [2] follows. Every such layer consists of a set of learnable kernels (filters) which are patches of the input's width and height and of the same depth as the input. During forward pass every filter convolves with the input image and creates a feature map. Feature maps of all the filters are stacked and create the output volume. Network learns filters that are activated when a specific feature is present in the input. Each neuron is connected only to a local region of the input. This region presents a **receptive field**. An example of local region connections can be seen in Figure 3.2.

This layer uses 3 hyperparameters:

**Depth** controls the number of neurons of the convolutional layer which connect to the same local region of the input just like multiple hidden neurons in a simple neural

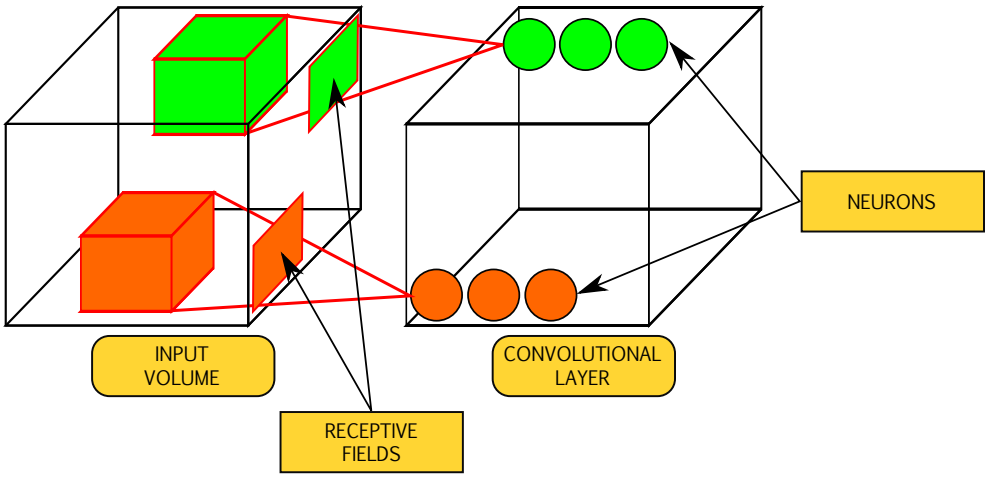



Figure 3.2: Example visualisation of connections between neurons of convolutional layer and local regions (their receptive fields) of the input volume. Local regions cover the whole depth of the input volume.

network can be connected to the same input.

**Stride** controls the intervals at which to apply filters on the input. With lower strides receptive fields overlap more.

**Padding** is used to pad the input with zeros around its border. Padding allows control over the size of the convolutional layer's output

The width and height of the layer's output volume can be computed using the Equation 3.1.

$$W_{out} = (W_{int} - F + 2P)/S + 1, \tag{3.1}$$

where $W_{out}$ is the output volume's width, $W_{in}$ is the input volume's width, $F$ is a filter kernel's size, $P$ is the zero-padding value and $S$ is the stride. In case this number is not an integer, the choice of parameters is wrong as receptive fields cannot overlap equally. This equation is analogous for the height of the output volume. The output depth's equals the number of filters in this layer.

Krizhevsky et al. [1] used input images with the size of $256 \times 256$ pixels and depth of 3 channels — $R$, $G$ and $B$. For these dimensions they chose filters of size $11 \times 11$ pixels and the same depth. The appearance of the first convolutional layer's filters from this work is shown in Figure 3.3.

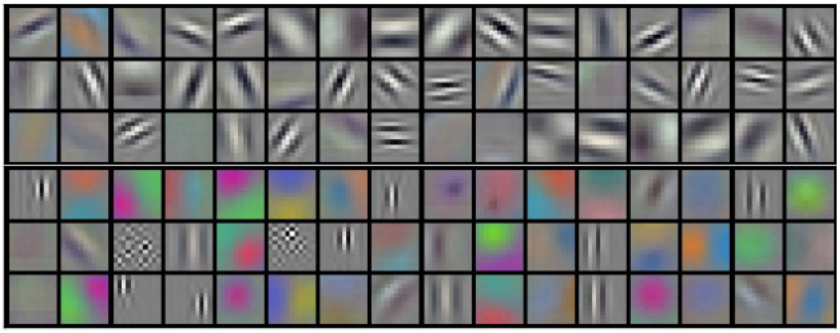


Figure 3.3: 96 trained convolutional kernels of size $11 \times 11$ with the depth of 3 colour channels ($R$, $G$ and $B$) and stride of 4 from the first convolutional layer in the network proposed by Krizhevsky et al. [1]. These filters were acquired after hundreds of thousands of iterations over a large dataset. Image adapted from article [1].

### Pooling Layer

The main purpose of pooling is dimension reduction as stated by van Doorn [19]. The most used operations for reduction are max-pooling, which gives the maximum value from the input set and average pooling that results in the average of all input values.

Based on the article [1] the pooling layers can be described as a grid of pooling units that are $s$ (stride) pixels apart from each other. Each unit covers a neighbourhood of neurons of size $z \times z$ whose centre is located at the covering unit's location. General pooling layers pool the outputs of neighbouring groups of neurons in the same kernel map and these do not overlap over the neighbourhoods of the adjacent pooling units. Common use of pooling involves setting $s = z$ which presents local pooling. Krizhevsky et al. [1] suggest overlapping pooling by setting $s < z$. The pooled output has the same dimensions a non-overlapping pooling would result in. Article states this change reduces the error rate by 0.4% and slightly reduces overfitting. An example of non-overlapping max-pooling can be seen in Figure 3.4.

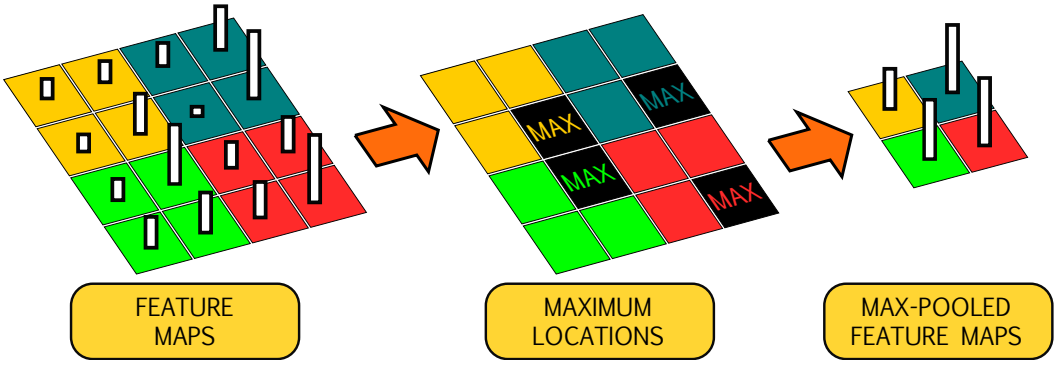



Figure 3.4: Example of non-overlapping max-pooling.

### Fully-connected Layer

As stated in article [14] this layer's neurons have full connections to all activations of the last layer. They become the final reasoning component of the network. They also no longer conform to spatial location just like the other layers and can be viewed as 1-dimensional.

### Rectified Linear Unit (ReLU)

Krizhevsky et al. [1] propose an alternative neuron model to make the convolutional network train faster. Neuron's output is generally modelled as a function $f$ of its input $x$, where $f(x) = tanh(x)$ ($tanh$ activation function) or $f(x) = (1 + e^{-x})^{-1}$ (sigmoid activation function). Having neurons where this saturating non-linearity is replaced with a function of $f(x) = max(0, x)$ allows faster learning compared to networks with $tanh$ units only. Article [1] states 25% fewer iterations required for training of the ImageNet network. Neurons with this activation function are referred to as Rectified Linear Units (ReLUs). These units do not require input normalizaton to prevent them from saturation. If there is at least one input higher than 0, learning will happen on the particular neuron. Rectifier linear units are also not bound like a basic sigmoidal function and can reach any non-negative real number as is pointed out by van Doorn [19].

### Neuron Input Normalization

Article [1] also describes the use of an input normalization scheme to prevent neurons from saturation. ReLUs do not require this kind of normalization but the network still contains it for the sake of generalisation. The process is expressed by Equation 3.2.

$$b^{i}{}_{x,y} = a^{i}{}_{x,y} / \left( k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^{j}{}_{x,y})^2 \right)^{\beta}, \tag{3.2}$$

where $b^{i}{}_{x,y}$ is a neuron's response-normalized activity, $a^{i}{}_{x,y}$ is the activity of a neuron set off by applying kernel $i$ at position $(x, y)$ and then applying ReLU non-linearity, the sum passes over $n$ kernel maps that inhabit the same spatial position and $N$ is the total number of kernel maps in the network's layer. All other parameters present constants set in correspondence to the input dataset.

### Output Classification Layer

Softmax layer serves as a final classification method in architectures proposed by articles [1, 16, 14]. It is used to produce distribution over all class labels of the input set. As pointed out by van Doorn [19], softmax is a linear classifier that uses logatithmic distribution.

## 3.2 Network's Overfitting and Its Prevention

Srivastava et al. [20] describes overfitting as adapting too much to the input training data. The architectures designed by Krizhevsky et al. [1] and Szegedy et al. [16] have millions of parameters. These allow the network to learn very deep and different features and complicated relationships between their inputs and outputs. However, with only a limited set of training data many of these relationships will be formed only by noise. This leads to overfitting.

### Input Data Augmentation

To prevent overfitting one of the most common method is to artifically enlarge the size of the input data. Krizhevsky et al. [1] suggest three such forms of data augmentation. The first is generating smaller $227 \times 227$-pixel crops from the original $256 \times 256$-pixel input images. Another is their mirroring about the vertical axis. The third method is alteration of $R$, $G$ and $B$ channels' intensities without destroying the image's identity. This adjustment is based on assumption that natural object's identity in image is invariant to intensity and colour illumination change.

### Dropout

Srivastava et al. [20] suggest combining the predictions of more different models (given by all possible settings of parameters) and using a weighted arithmetic mean with weights given by the probabilities from the training data. Such operation leads to improvement of machine learning method's performance. However, combining predictions of more different models suffers from a higher chance of overfitting that can be resolved using dropout.

This technique sets the output of every neuron in any hidden layer to zero with 50% probability. These *dropped out* neurons no longer contribute to the network's forward pass and are not visited during backpropagation. Every time a new input is analysed a different architecture is used yet the updated weights will be shared across all the architectures. Krizhevsky et al. [1] also use this technique to limit the possible co-adaptations of connected neurons in the fully-connected layers as these neurons can no longer rely on the presence of the surrounding neurons.

## 3.3 Fully Convolutional Neural Networks

Long et al. [17] propose a deep convolutional neural network for dense pixel-wise image segmentation and object detection. They extend the net to an arbitrary-sized inputs. This is done by converting the fully-connected layers to convolutional. As convolutional networks are translation-invariant, their base components (convolution, activation functions, pooling) operate only on local input regions and therefore only work with relative spatial coordinates. This is the only difference from the fully-connected layers. Other than that, their function is identical. Conversion is done by turning the fully-connected layer closest to the input layer to convolution with the filter size set to the size of the input volume. The result is a set of coarse output maps covering the whole image with a certain stride. In case the receptive fields overlap significantly, forward pass and backpropagation work much more efficiently when operating on the entire image instead of single patches. Then they connect the coarse output maps to pixels using interpolation. Long et al. [17] perform this transition on the network of GoogLeNet from article [16].

# Chapter 4

# Fire Video Rendering and Composition

The proposed method of fire detection using convolutional neural networks requires a vast number of testing data. Each testing sample must be a short sequence of fire burning in a real-life environment. Using the techniques described below I created 2000 frames of fire animation footage which can be used for detection training and testing. Examples of the created frames can be found in Appendix B.

## 4.1 Fire Video Sequences Creation

The main disadvantage of fire detection is that fire on its own represents a not so common phenomenon. Its size, shape, dynamics and colour are very variable. It is not an easy task to find an extensive quality resource for fire images let alone fire animations. In this thesis I used a different approach to normal video creation.

Instead of filming a real-life fire, a normal scene without fire was filmed with a static camera. The camera's resolution was $1920 \times 1080$ pixels. For every scene, its 3D model is created in Blender version 2.71+. The notion is to create a fire simulation in the modelled environment (an animation of burning fire) and render it using Blender's Cycles ray tracing renderer. Then extract it, together with all of its lighting effects and shadows it creates in the scene, and paste all of these onto the frames of the filmed scene.

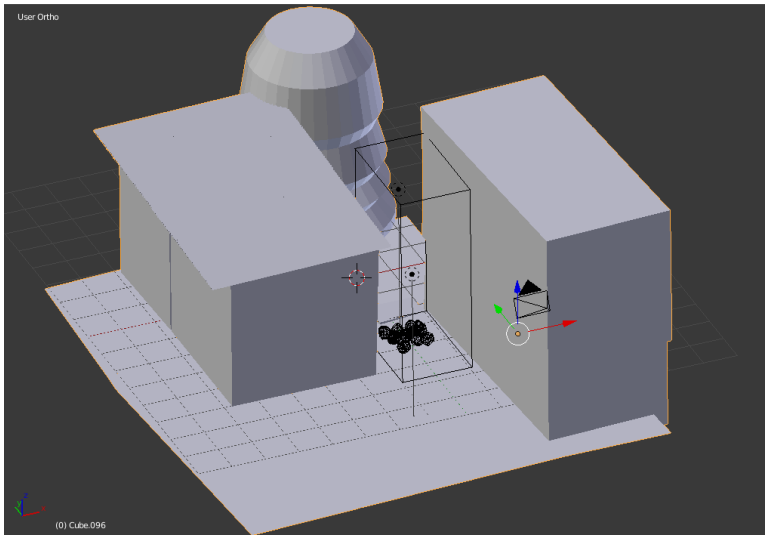
I have filmed a total of 10 static scenes that capture different kinds of environments of both exterior and interior. I chose 2 of these scenes and created their correspoding 3D models.

An illustration of the created outdoor scene, Scene 1, can be seen in Figure 4.1. Figure 4.1(a) contains the view of the complete 3D model in a Blender window. The scene represents a crude depiction of the real recorded setting without too much detail. For instance, the conifer tree in the background in Figure 4.1(b) is modelled only by its unique conical shape. The main part that must conform to reality the most is the lighting and shadows. Therefore, for the scenes to be as close to the original as possible, I tried to preserve the original lighting from the recording. Figure 4.1(c) displays the rendered scene in the same view the original video captures. Views of all the other created scenes and their comparisons with the original images can be found in Appendix A.

To add fire to this scene, Blender's fire simulation and physics engine are used. Fire is simulated at multiple locations in the scene model and a different random seed for the

simulation is always used. This guarantees that fire animations never look the same. Using this approach several different fire sequences can be created from one scene.

In order for the simulation to be rendered, it needs to baked — calculated — first. In Blender's graphical user interface this is done automatically. Fire does not spread over the entire scene but occupies only a part of it — its domain. Calculations are restricted to fire's domain which is displayed in Figure 4.1(a) as a wireframe rectangular box. This reduces the required processing power for the simulation.




(a)

(b)

(c)

Figure 4.1: Demonstration of the created Scene 1 and its comparison to the original image: a - scene's 3D model view in Blender, b - original image used as modelling template, c - rendered scene.

The main source of information about fire modelling in Blender were BlenderDiplom and Blender Cookie[1] fire and volume tutorials. As Blender does not support any predefined fire materials or models I have created my own fire material — fire shader. The setup for this material in Blender can be seen in Figure 4.2. All the qualities of fire, that I considered important, are adjustable in the material settings shown in the bottom-right corner of Figure 4.2. These fire parameters together with their default values are:

**Smoke scale** The default value is 1.0 which fills the domain with smoke of similar size to the flame.

[1]Blender tutorials:
BlenderDiplom: http://blenderdiplom.com/en/
Blender Cookie: http://cgcookie.com/blender/

**Smoke density** The value of smoke's density also affects the fire's flame as they are spread together in the same domain. The smaller the number the more transparent (see-through) the smoke and flame become. This value should be kept above 4.0.

**Fire scale** Its value affects the scale of the flame's bounds which go from the brightest in the centre to the reddish colour on the boundaries. The brightest spot is scaled with the value of this parameter. It could also be taken as the fire's temperature. Its default value is 4.0.

**Fire density** This paramter tells how much flame is actually generated. When this density is increased, the fire becomes brighter. The default value is 2.0.

**Fire gamma** Gamma sets the fire's contrast. Its default value is 1.0.

**Fire hue** Hue affects the actual colour of the burning flame and its default value is 0.5 which gives the fire its usual white-yellow-red appearance.

To add even more variety to the scene and its fire I used Blender's physics engine which goes hand in hand with its simulation. Real fire does not burn uniformly and is also affected by its environment, e.g. blowing wind or random fluctuations. I added these kinds of elements to the scene completely randomly for a more realistic effect. Wind was simulated by Blender's wind force element. This made the fire lean and burn in the direction of the wind.

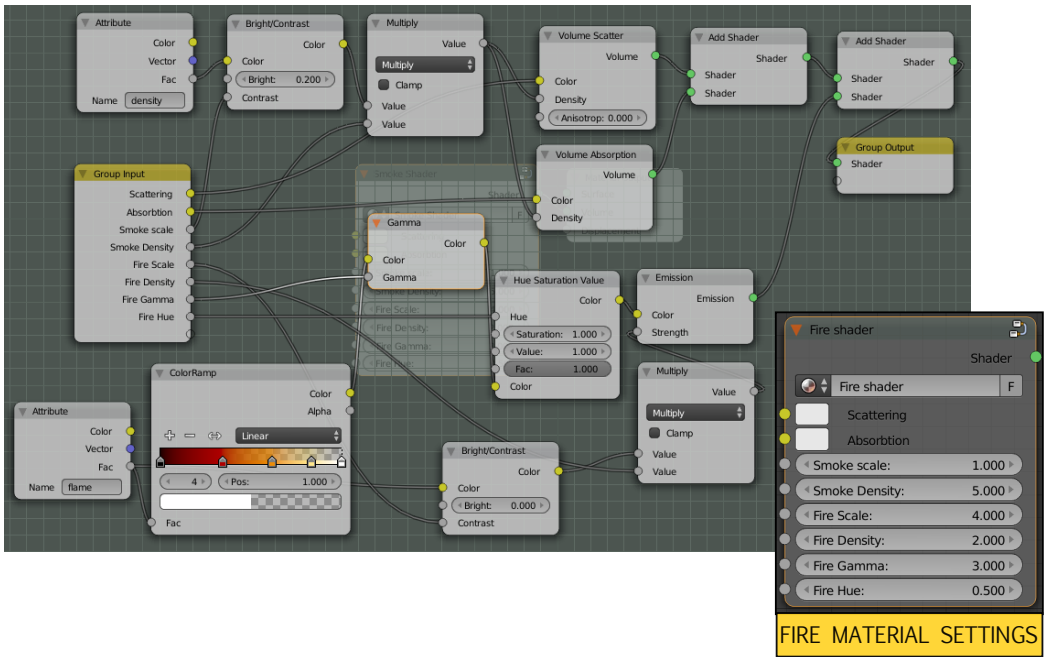
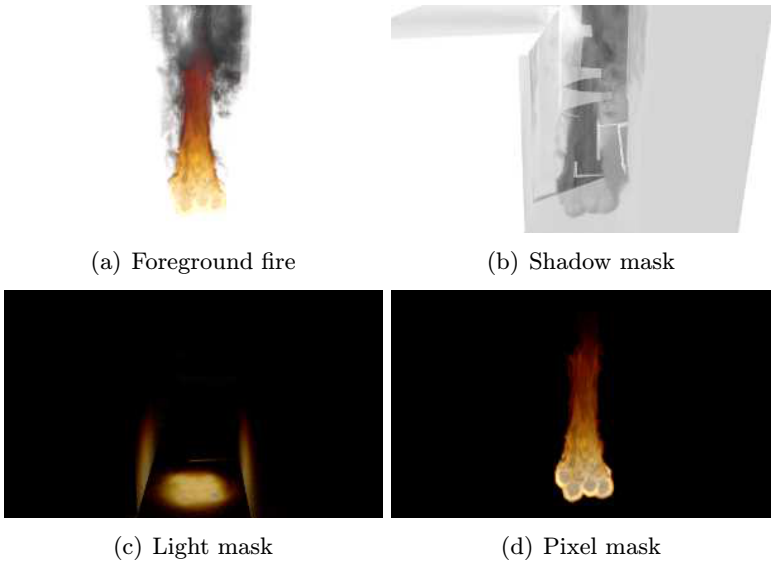



Figure 4.2: Fire material node setup in Blender.

To create a realistic scene, it must be possible to apply the generated results on the original video frames. This means that the shadow and lighting effects together with the fire foreground must be transfered from the rendered scene to the real one. The fire scene data is, therefore, rendered into multiple images — fire and smoke foreground, scene's shadows,

scene's fire light emission — and is later composited into images of a sequence. Part of the rendering process is the creation of a fire pixel mask which is used as an annotation for the detection. All of the Blender outputs are shown in Figure 4.3.



(a) Foreground fire (b) Shadow mask

(c) Light mask (d) Pixel mask

Figure 4.3: Display of rendered data from a random sequence frame.

For the generation of outputs mentioned above I used Blender's own compositor system. During the rendering process there are 4 different layers rendered for each of the animation's frames. Every layer contains different parts of the modelled scene and stores multiple pieces of information about them (e.g. shadow maps, light maps, diffuse). These layers are:

1. foreground layer which contains only the scene's fire,
2. complete scene layer without fire,
3. complete scene layer with fire,
4. complete scene layer without fire that preserves the fire's light and shadow effects on its surroundings.

The final composited outputs are created as follows.

- Foreground mask is created from layer number 1.
- Shadow mask is constructed by subtracting shadow masks from layers 2 and 3 and inverting the final image.
- Light (emission) mask is created by subtracting direct diffuse lighting components of layers 4 and 3.
- Pixel mask is rendered as light emission component of layer number 3.

For a large set of testing sequences batch rendering using the command-line interface (CLI) is an absolute necessity. As Blender does not support simulation baking within its CLI (it is only accessible in the interactive window), I created a simple script using Blender's

*Python* application interface (API). The script runs every simulation contained in the scene before rendering process and writes the result of every frame into the `baked` folder in the working directory. This folder is later referenced when rendering. The script is a part of the thesis's DVD content.

## 4.2 My Own Image and Video Compositor

Rendered fire animations from Section 4.1 need to be composited with the frames of a filmed scene. To have as much control over fire images creation as possible, I programmed my own compositor using OpenCV library [21]. I created it universally so it can be used for different kinds of compositing and not just for the creation of fire images.

The compositor uses a simple *in position* approach when compositing images. After the creation of a new Compositor object it needs te be initialized with a background image. The background can be either an OpenCV `Mat` object, path to a file or the compositor can substitute the background with its own checkerboard image. Background image is supposed to contain 3 channels of 8-bit depth. In case of loading a different image, for instance a grayscale, OpenCV, if possible, automatically converts this image to this specified format. After having a background loaded it is copied to the compositor's resulting image and the composition can start. The resulting image is rewritten after every operation — *in position* approach. Therefore, it serves as an accumulator.

The compositor includes functions that cooperate well with the data from the previous section. All of them operate on per element basis. The most important functions are:

**Masking**
: Before using any operation on the compositor's background a write-enable mask can be added. When loading a mask from an image file it is always interpreted as binary. White pixel means write-enable and all the other colours are write-disable. Pixels with write-enable flag active are affected by called operations while disabled are not. Masks can be cleared — leaving all pixels as enabled for writing — or filled — masking the entire resulting image. Loading and clearing masks presents a way to change only specific areas of the composition.

**Simple math operations with images**
: Simple addition, subtraction and multiplication use the same image format as background loading. The compositor's resulting image is always the first parameter. The second parameter for the operation is its input image.

  Image multiplication starts by converting both images to a floating-point depth and multiplying them per element. The result must be converted back to the original 8-bit channels. As the maximum value of the images' channels grew from 255 (8-bit range) to 65025 ($255 \cdot 255$) they must be normalized before conversion by multiplying every one with $1.0/65025 \cdot 255$.

  The simple operations, such as addition or subtraction, are always saturated. This means that if the result's channel value gets above (or below) its 8-bit maximum (minimum) it is reassigned that extreme value. This behaviour is provided by OpenCV.

**Image alpha blending**
: The last operation the compositor is capable of is simple alpha blending. This operation, unlike all the other compositor's operations, requires the input image to contain

alpha channel. Both of the images are added together using Equation 4.1 as is written in OpenCV documentation[2].

$$Result(x, y) = (1 - \alpha) \cdot Result(x, y) + \alpha \cdot Input(x, y), \tag{4.1}$$

where $Result(x, y)$ represents the resulting image's pixel at position $(x, y)$, $Input(x, y)$ represents the operation's input image's pixel at $(x, y)$ and $\alpha$ is the normalized value of input's alpha channel at $(x, y)$. Alpha normalization represents converting the alpha-channel's 8-bit $0 - 255$ range to a floating point range between 0.0 and 1.0. The alpha-addition is also done per element.

The result of composition may be written to an image file or shown in an interactive window. The controls for using this window are described in Appendix C, Section C.

As the compositor works only on images, for composition of multiple video frames I implemented a separate loader. Loader includes a command-line interface for easy use with the rendered data, functions for calling the compositor's operations and it can also read and write videos. The loader also features video cutting capabilities. The complete manual to using the interface together with examples can also be found in Appendix C, Section C. The compiled application and its source codes are included in the DVD content.

## 4.3 Fire Sequence and Real-Life Scene Composition

Using the rendered fire sequence from Section 4.1 and my compositor from Section 4.2 the final fire video can now be created. The process of a single fire frame creation and compositing can be seen in Figure 4.4 with numbered steps $1 - 5$.

1. In the top right corner there is the original image extracted from a video sequence. The video loader initializes the compositor with this frame as its new background.

2. At first, this image is masked to forbid the compositor to change the areas that are not supposed to change in the video. In case of the example Figure 4.4 this mask is used on the bush which is very close to the camera in the extracted frame. I created this mask manually and it is the same for all the frames.

3. Masking is followed by multiplication with a shadow mask. Shadows, which are stored in a grayscale image, darken parts of the frame.

4. After that fire light emission mask is added that lights up pixels just like a real fire would when burning. Simple addition is used for this step. Emission masks should be mainly used on scences which are poorly lit. Adding them to a sunny scene would be counterproductive.

5. On top of that an image containing the rendered fire is added using alpha-blending. This image preserves alpha channel and acts as a simple overlay.

All of these components combined create a fire frame. A short sequence of these fire images together with their corresponding fire pixel masks serve as a single testing sample.

[2]OpenCV alpha-blending: http://docs.opencv.org/doc/tutorials/core/adding_images/adding_images.html

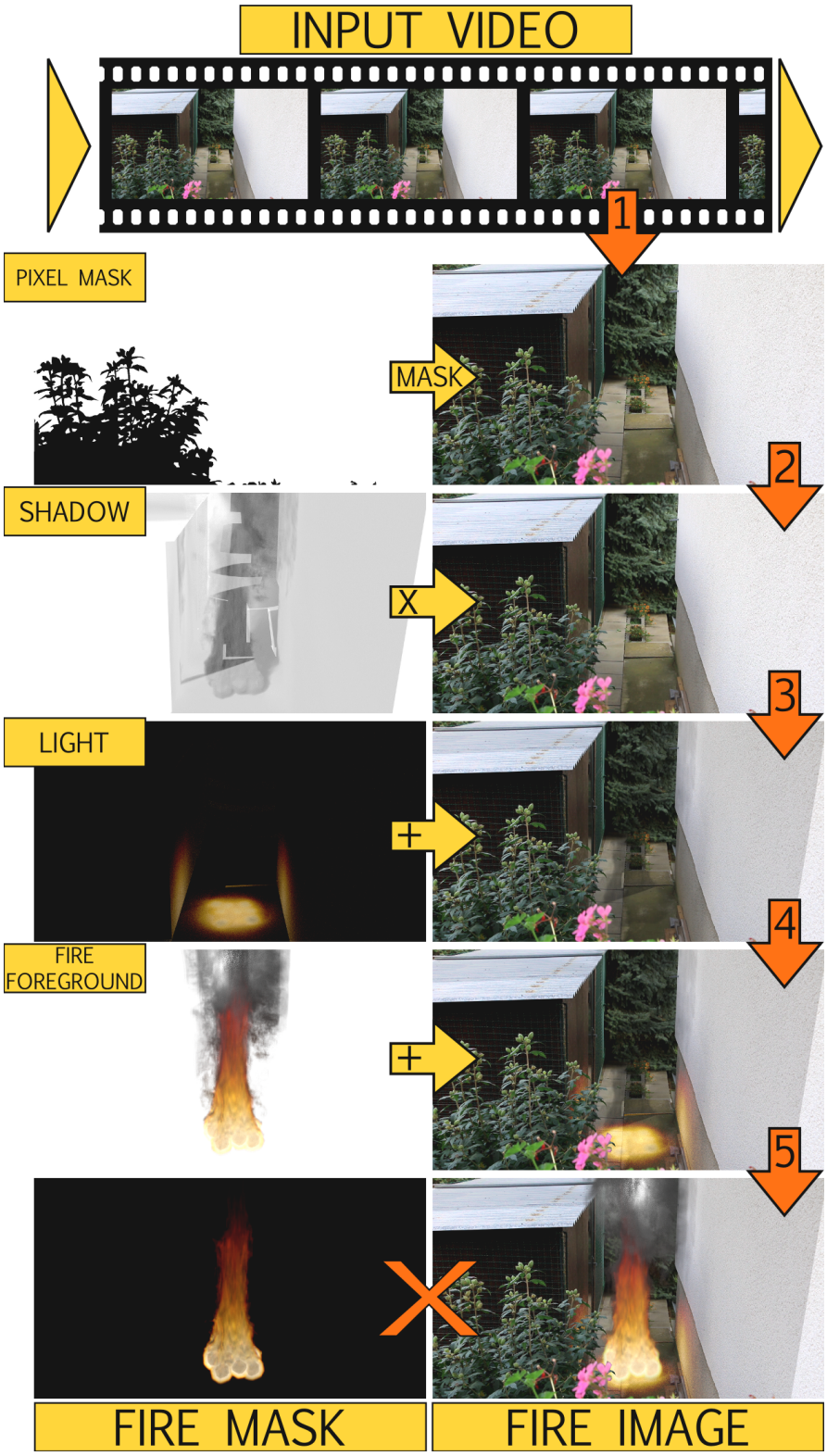



Figure 4.4: Depiction of a fire image creation: 1 - video frame image extraction, 2 - pixel mask application, 3 - shadow mask multiplication, 4 - emission mask addition, 5 - fire foreground addition (= fire image generation), X - fire mask and fire image output.

# Chapter 5

# The Fire Detector

The proposed method of fire detection using deep convolutional neural networks has not been used in of the methods that I studied.

There are different tools that allow working with these network models. There is Theano [22] for *Python*, Torch7 [23] built on *Lua* and Caffe Deep Learning Framework [2] for *Python* and *C++*. As Caffe presented the fastest results in deep learning with deep convolutional neural networks at the beginning of my work and I used *C++* for the development, I chose Caffe.

Building a detector that would easily communicate with Caffe lead me to different design choices. I had to preprocess my fire data and put them in a suitable form. For that I designed and implemented two other applications — a sampler and a database generator.

## 5.1 Fire Samples Labeling and Extraction

The resulting fire sequences from Chapter 4 serve as training and testing (validation) data for the detector's model. The detector is supposed to learn patterns and parameters from its input and recognise these during fire detection. I propose two different approaches to the detector's training - per the entire image and per segment. Per image is a special case of per segment approach. These approaches may require segmentation of a larger images to smaller ones. If fire occupies only a minimal part of the input image (sequence) it is better to make it a non-fire sample. For video analysis, every sample may have its depth that corresponds to a certain number of frames. To sample the data in these ways I created a separate sampler.

The sampler takes 2 sequences of images as input — fire images and pixel labels of these images. The required labels are binary masks (black and white) that can be generated from fire masks from Section 4.2 using my compositor. A starting frame is chosen from this sequence and a subsequence of set sample depth beginning with this frame is analysed for samples. An example of a label sample with dimensions of $5 \times 5 \times 4$ from such a subsequence is shown in Figure 5.1.

The sampling is controlled by two matrices with the same width and height as the original input image:

**sum matrix** where every pixel equals the sum of positively labelled (fire) pixels (pixels of black colour) at the same position $(x, y)$ in a sequence of image labels. An example of this matrix generated from sample in Figure 5.1 is shown in Figure 5.2,

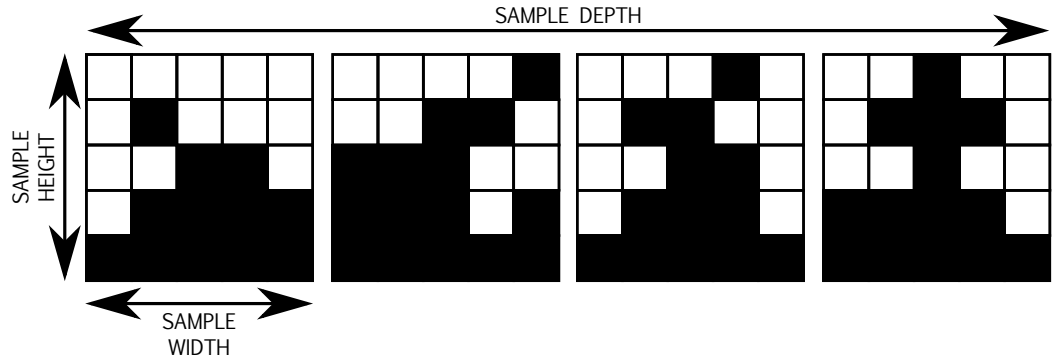



Figure 5.1: Example of a $5 \times 5 \times 4$ sample represented by its labels (black = positive (fire) pixel, white = negative (non-fire) pixel).

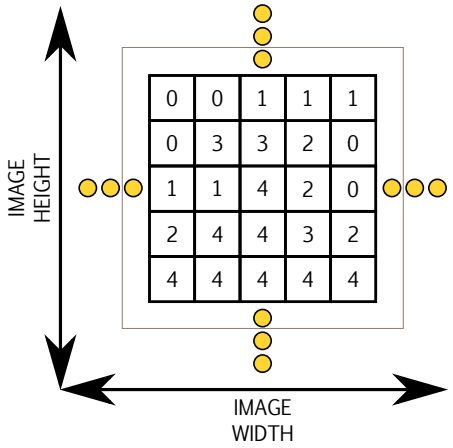



Figure 5.2: The **sum matrix** of a sequence of image labels from the example sample from Figure 5.1. Each pixel value equals the sum of positively labelled (fire) pixels (pixels of black colour) at the same position $(x, y)$ in a sequence of image labels.

**sample matrix** which stores the number of sample's positively labelled pixels in the sample's leftmost and upmost pixel in the original image. An example is shown in Figure 5.3. This matrix is generated from the **sum matrix**. After it is created, only a simple traverse over its coordinates is required to find samples that have enough fire pixels to be considered positive (fire) or negative (non-fire) samples.

The sampler contains different settings that influence the generation of samples. These include:

- sample dimensions,
- required percentage of fire pixels in a sample's volume to be negative/positive,
- the maximum number of required negative or positive samples,
- percentages of maximum number of samples within a subsequence,

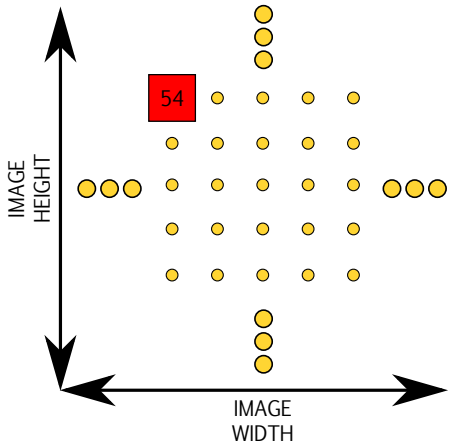



Figure 5.3: The **sample matrix** which stores the number of sample's positively labelled pixels in the sample's leftmost and upmost pixel in the original image. It was generated from the **sum matrix** in Figure 5.2.

- random or constant step between sequence's frames when picking candidates for subsequences' first frames,
- random or constant step between filtered samples when picking candidates from sample lists,
- positive/negative samples ratio (used within a subsequence),
- postprocessing options (e.g. resizing) after generation.

An example of one iteration of the sampling process can be seen in Figure 5.4.

1. In the left part of Figure 5.4 there is an example of a fire sequence that contains both the labels and the composited fire images. This sequence is traversed and a starting frame of a subsequence of given sample depth is chosen according to the sampler's settings (e.g. randomly).
2. The subsequence's frames are selected and analysed separately.
3. Subsequence's **sum matrix** and its **sample matrix** are created. Using these matrices lists of all negative and positive samples available on the given subsequence are generated (neutral samples are discarded). These lists are then filtered according to the current settings, e.g. the required ratio of positive/negative samples and their maximum number.
4. All the samples that passed the filtration are stored or just saved in a form of a text file.

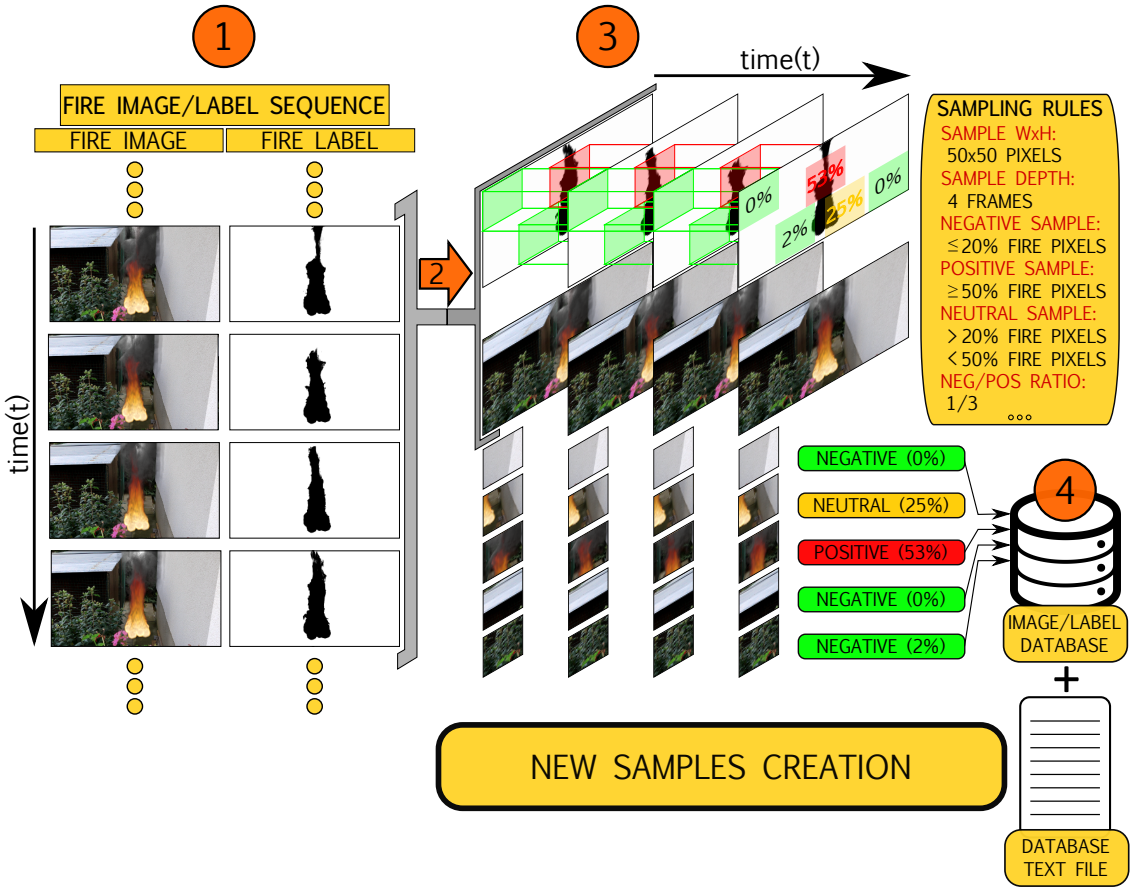



Figure 5.4: Depiction of samples' creation: 1 - fire frames and labels traverse, 2 - sample frames selection, 3 - frame labels analysis according to the sampling rules and location of possible samples of given dimensions, 4 - positive and negative samples extraction and their storage in the database.

The sampler itself allows direct extraction and saving of found samples from an image sequence but also creates a database text file which I designed. When using this file, the sampler does not need to save any image samples and only stores their paths in the text file. This file can be worked with in the next application – the database generator.

The sampler is usable as a command-line application and requires the OpenCV library [21]. The complete manual can be found in Appendix C, Section C. The compiled application and its source codes are included in the DVD content.

## 5.2 Samples Database Generation and Data Creator

Caffe [2] uses standalone files for network's definition and accepts different kinds of input training or testing data.

One of the most used formats is Caffe's *Datum* object which stores data in a serializable structure. The best way to train a network is to use a database backend for instance

*LMDB*[1] or *LevelDB*[2] which store all input samples in a single container. Caffe provides many helpful scripts that allow conversion of a simple directory with images to a database of *Datum* objects. However, for different data structures that contain more channels than a single image, e.g. a video sequence, there is no easy conversion. To solve this problem I designed my own database generator and multi-frame *Datum* creator.

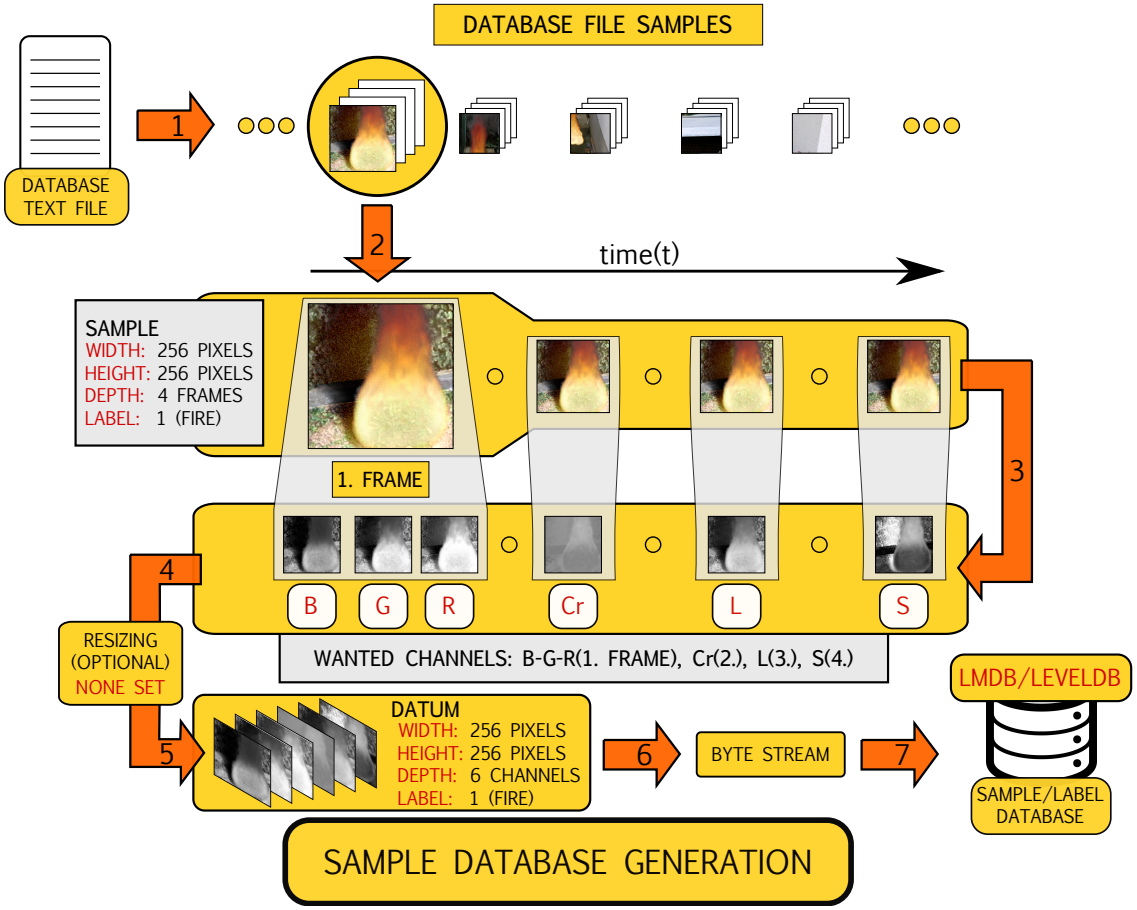



Figure 5.5: Example depiction of samples' data to *Datum* conversion and database generation: 1 - loading database text file, 2 - sample selection, 3 - concatenation of required $B$, $G$, $R$ (1. frame), $Cr$ (2. frame), $L$ (3. frame) and $S$ (4. frame) channels from every sample (in this order), 4 - optional resizing, 5 - *Datum* conversion, 6 - *datum*'s serialization to byte stream, 7 - storing the byte stream in a database.

The generator's settings allow the creation of multi-frame samples by appending channels of frames set in the configuration file to the *Datum* object. For this task I designed a separate *data creator* which takes care of conversion between multiple frames and a *Datum* object. Even though I intended to use the generator for RGB images only, I wanted to create a universal tool capable of encoding different image information into a single data sample. Therefore, the *data creator* is capable of adding different channels from every frame including:

[1]LMDB website: http://symas.com/mdb/
[2]LevelDB website: http://leveldb.org/

- $R$, $G$, $B$ channels of RGB colour model,
- $Y$, $Cb$, $Cr$ channels of YCbCr model,
- $H$, $S$, $V$, $L$ channels of HSV and HLS models,
- grayscale channel after conversion from RGB,
- binary value of 0 if grayscale channel is higher than given threshold and 1 otherwise.

Database text file created by the sampler from Section 5.1 is parsed and samples (with their labels) are retrieved. Based on the given configuration, the generator traverses the loaded samples, converts them to *Datum* objects and stores them in a database. Figure 5.5 shows an example of this process configured to store $B$, $G$, $R$ channels from the first frame, $Cr$ from the second, $L$ from the third and $S$ from the fourth from every file sample.

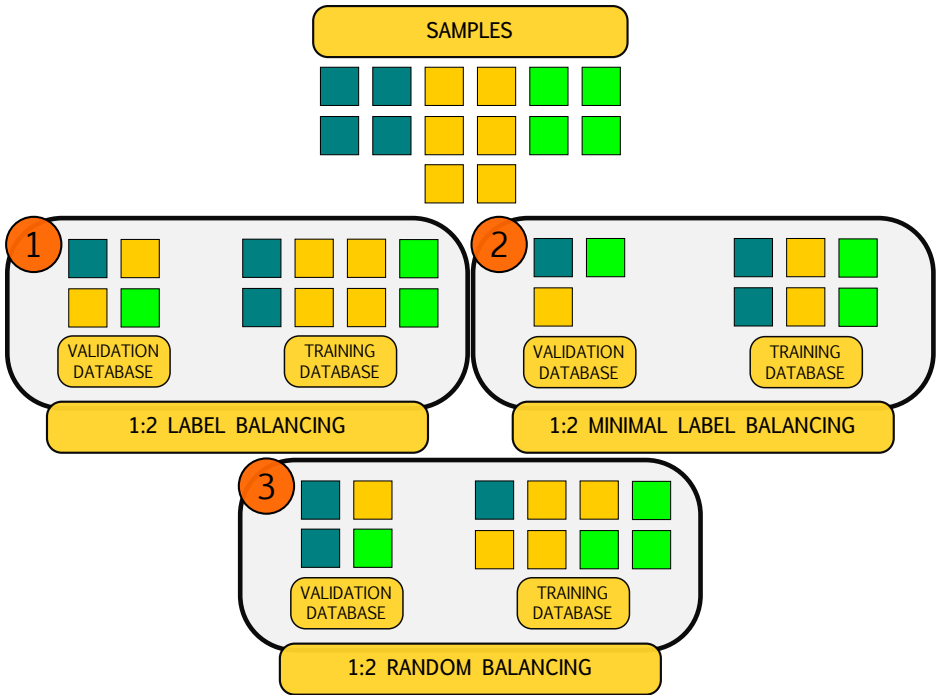



Figure 5.6: Showcase of generator's methods for balancing input samples between the training and validation databases with a training/validation ratio equal to 2:1. These methods are: 1 - *label balancing* which splits every class of the same label based on the ratio, 2 - *minimal label balancing* that (after splitting) limits every class's number of elements to the amount in class with minimal number of elements (separately for both databases), 3 - *random balancing* that randomly splits the input samples into two databases according to the set ratio. When balancing, elements that would break the configured training/validation ratio are discarded (e.g. odd number of samples).

The database generator is meant to create one or two databases — validation and training — which can then be used during Caffe's training procedure. It allows the creation of a database using either *LMDB* or *LevelDB* backend which are supported by Caffe. There are also options to resize the input samples before committing them into the database,

shuffling and splitting them using a given ratio. It can also be set to balance the amounts of samples in both databases. There are three types of balancing techniques available which are shown in Figure 5.6.

The designed *data creator* is also included in the detector itself and can be configured just like in this generator. During detection *data creator* converts the analysed frames' channels into a *Datum* and sends them to the deep convolutional neural network trained with data of the same configuration. Such a tool allows for faster experiments as the detector and generator work together by sharing the same data preprocessing.

The database generator presents a command-line application and requires the OpenCV library [21] and Caffe library [2] to work. The complete manual can be found in Appendix C, Section C. The compiled application and its source codes are also included in the DVD content.

## 5.3 Fire and Non-fire Classification

The main part of the detector is its classifier part. This part is formed by classification using a trained model of deep convolutional neural network. The detector also includes an addition — a colour analyser — that serves as a preprocessing to the classification.

### Deep Convolutional Neural Network Model Classification

The detector uses the same structure of the deep convolutional neural network that was proposed in the work of Krizhevsky et al. [1] for ImageNet classification challange [24] which is shown in Figure 3.1. In the Caffe environment, this net structure is referred to as Caffenet. It contains every modification proposed by Krizhevsky et al. in Section 3.1 and also includes using dropout for fully-connected layers and data augmentation methods like mirroring and cropping to artificially enlarge the input dataset as described in Section 3.2. The difference is switched order of pooling and normalization layers. The size of input for this network is set to $224 \times 224$-pixel crops of images with 3 colour channels. Because of this restriction I always scaled the analysed image (frame) to $256 \times 256$-pixels while leaving Caffe to extract the crops. This applies to both training and testing. The specified size should capture just enough details.

I made a few changes to the net's structure. As my aim is to find fire, there are only two labels in my dataset — fire and non-fire image (segment). As Caffenet was originally created for classification task that involved hundreds of classes, number of outputs of the last convolutional layer needed to be changed to 2 — the training process would be a lot longer without this change.

For images, this model is extremely versatile. For video detection it must be adapted by changing the corresponding number of input channels to that of multiple analysed frames. The *data creator* described in previous section is used for this task.

The convolutional net's input is a multi-frame (multi-channel) image of the specific size and its output is a number that represents the probability of classifying this image as fire.

### Fire's Colour Analysis

As a secondary improvement I added fire pixel's colour analyser into the detection. The analyser is used prior to the network's model classification. Its main task is not to improve the number of positive detections but instead to discard the regions in the input image

that do not contain at least a small portion of fire pixels. This could decrease the number of false detections. I adapted a part of the implementation from my bachelor's thesis [25] that was aimed at detecting fire by its colour and motion. I used the rules for segmentation of fire's colour described in the articles [4, 8, 9]. A demonstration of fire pixels' detection results on a fire image after applying different colour rules can be seen in Figure 5.7.

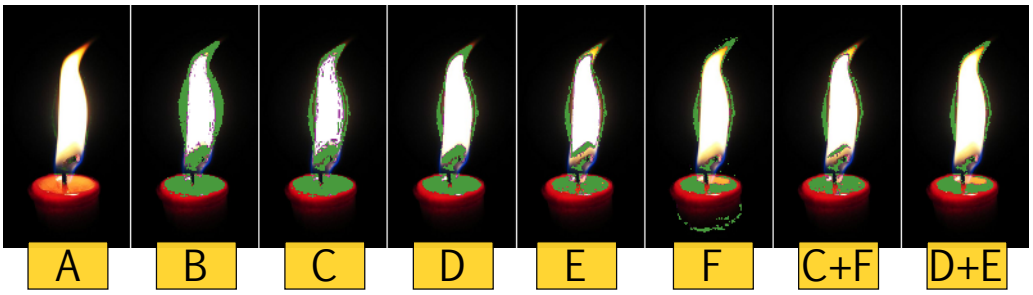



Figure 5.7: Demonstration of fire pixel's colour detection results based on different colour rules from the works [4, 8, 9]: A - original image, B - results of rules from article [4], C - results of rules from article [8] omitting proposed colour polynomials, D - results of rules from article [9] omitting pretrained model, E - results of rules from article [9] using pretrained model only, F - [8] results of using colour polynomials only, C+F - result of combination of all rules from article [8], D+E - combination of all rules from article [9]. Image adapted from my bachelor's thesis [25].

My designed detector implements all of these rules. Configuration allows setting the percentage of area of an analysed segment that needs to be occupied with fire pixels for the segment to be pushed into the deep convolutional network model for classification. Median filter can also be applied to partially remove the noise of lonely fire pixels in the image before checking this percentage.

# Chapter 6

# Fire Detector's Testing and Experiments

The detector presents a command-line application and requires both the OpenCV library [21] and Caffe library [2]. The complete manual can be found in Appendix C, Section C. The compiled application and its source codes are included in the DVD content. Demonstration of detector's output based on these experiments can also be found in the DVD content.

I experimented with my detector to find the best configuration of its individual parts. Tests were conducted for both images and for videos.

All these experiments were conducted on a laptop with Nvidia GTX 980 GPU.

## 6.1 Image Experiments

Dataset samples in the image experiments are RGB images only (stored in BGR order).

### Training, Validation and Testing Datasets

Experiments test the detection's results of per $128 \times 128$-pixel and $256 \times 256$-pixel segment approaches and per image approach in single images.

I used 3 — training, validation and testing — sets of data for the experiments. For convolutional network's training I used the composited 3D fire images. One sample of per image approach corresponds to one unsegmented image. As the net's structure is created to accept any kind of image and resizes them during processing, any image of a varied size is considered a sample. For per image training I picked 900 fire images from the composited 2000. I picked 800 images from the rest of these for the model's validation. I omitted 300 images that looked quite similar to the already used ones. Then I collected 1280 real non-fire images of mostly city buildings from LabelMe[1] database. All of these images were of a varied size starting at $640 \times 480$ pixels. 780 of these are used for per image training (as non-fire samples) and 500 are used for validation.

For per segment approach I sampled the 900 training and 800 validation fire images and 780 and 500 non-fire images using these main options set for samples generation:

- sample depth is always equal to 1,
- positive sample has more than or equal to 50% of its area formed by fire pixels,

[1] LabelMe website: http://labelme.csail.mit.edu/

- negative sample has less than or equal to 20% of its area formed by fire pixels.

I generated 5500 fire samples and 5500 non-fire samples of $128 \times 128$ pixels. I used my generator application from Section 5.2 to join these samples, shuffle them randomly, balance them according to their labels and split them in half into two databases each containing 2250 fire samples and 2250 non-fire samples. These databases were used for per segment training and validation. Then I generated the same amount of samples of $256 \times 256$ pixels and stored them in two databases.

The described validation data is meant to measure the neural network's accuracy on a similar but not the same dataset as the training data. I chose splitting the training and validation databases in a ratio 1:1 as some of the rendered images displayed only slight differences and I wanted to prevent overfitting by training with very similar data.

My goal is to create a detector of a real fire. Therefore, for real fire tests I used separate images downloaded from LabelMe which included 50 random real fire images and 50 non-fire images. I mixed these with 100 fire images and 400 non-fire images from my bachelor's thesis [25]. All of these form the testing set for image experiments.

### Experimenting with Colour Analyser

In my bachelor's thesis [25] I already tested the effectiveness of colour rules and concluded that for fire detection in image the best way is to combine all of the described rules from Figure 5.7 which boasted the best results but was also very limiting. The quality of my input dataset back then allowed the use of this combination as it contained mostly colourful and high resolution images. On the other hand, for colour detection in video the best combination consisted of rules $B$, $C$, $D$, and $F$ from Figure 5.7. Video dataset in my bachelor's thesis [25] contained fire videos downloaded from the Internet with very variable resolution and mostly low video quality — these were used to simulate low-quality surveillance system footage. To detect fire here a less restrictive combination of colour rules had to be employed. For my experiments I chose this colour rule combination for videos. I also wanted to test the least bounding colour rule $B$ shown in Figure 5.7 proposed in article [4]. My experiments include 3 different settings of the colour analyser:

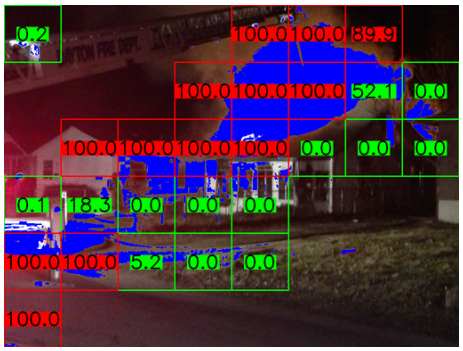
1. completely turning it off,
2. using the best colour rules combination for video already tested in my bachelor's thesis [25],
3. using only the least restrictive colour rule $B$ from Figure 5.7 based on the tests from my previous thesis.

As per image detections were very few there was no point in connecting the colour analyser to their classification. For per segment approach however, the result could be improved by discarding false regions. Segments of $128 \times 128$ pixels were most suitable for these tests. In order for a segment to be considered fire, at least 1% of its area must be formed by pixels of fire's colour. If it is not, it is ignored. I chose this small amount just to throw away regions of completely different colours.

Demonstration of the detector's results working with $128 \times 128$-pixel segments and fire's colour analysis applied can be seen in Figure 6.2.


(a) Detection using 1. setting (no colour analysis)


(b) Detection using 2. setting ($B$, $C$, $D$ and $F$ combination)




(c) Detection using 3. setting ($B$ only)

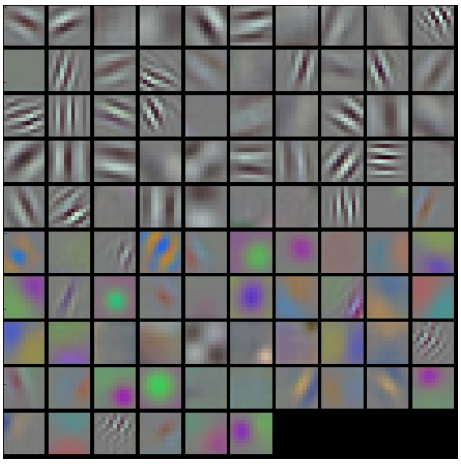
Figure 6.1: Detector's results on real fire images applying colour classification to discard regions without fire pixels. Blue colour shows the blobs of fire's colour found in the input. Many false detections are thrown away when using 2. or 3. settings.

### Training with Initialized Model's Weights

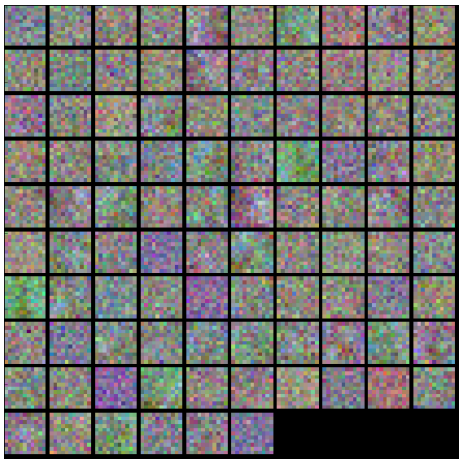
One of the advantages of Caffe is the capability to stop and restart the learning procedure of model's parameters (weights) almost anytime during training (by saving model's intermediate states). When conducting new training, the learnt weights from previous training can simply be transferred to the new model by initializing its weights to these values. The only condition is having layers named the same in both models. Krizhevsky et al. [1] created a model that took hundreds of thousands of iterations to train on their large dataset. This much time taught the filters to recognise RGB images quite well. This can be seen in Figure 6.2(b) where filters of first convolutional layer look quite *clean* and contain minimal random noise.

Compared to millions of images used for training of Caffenet, my dataset seems small. I wanted to see how much the pretrained weights would change the detector's results. I used the reference pretrained Caffenet model that is a part of Caffe's distribution. I initialized the weights of my model from this trained model except for the last fully-connected layer that contains the probability vector for all trained classes.

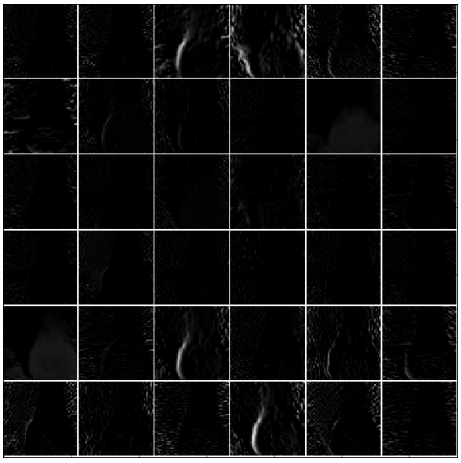
(a)



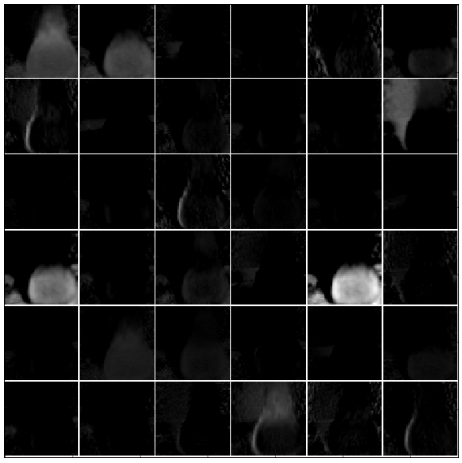
(b)



(c)



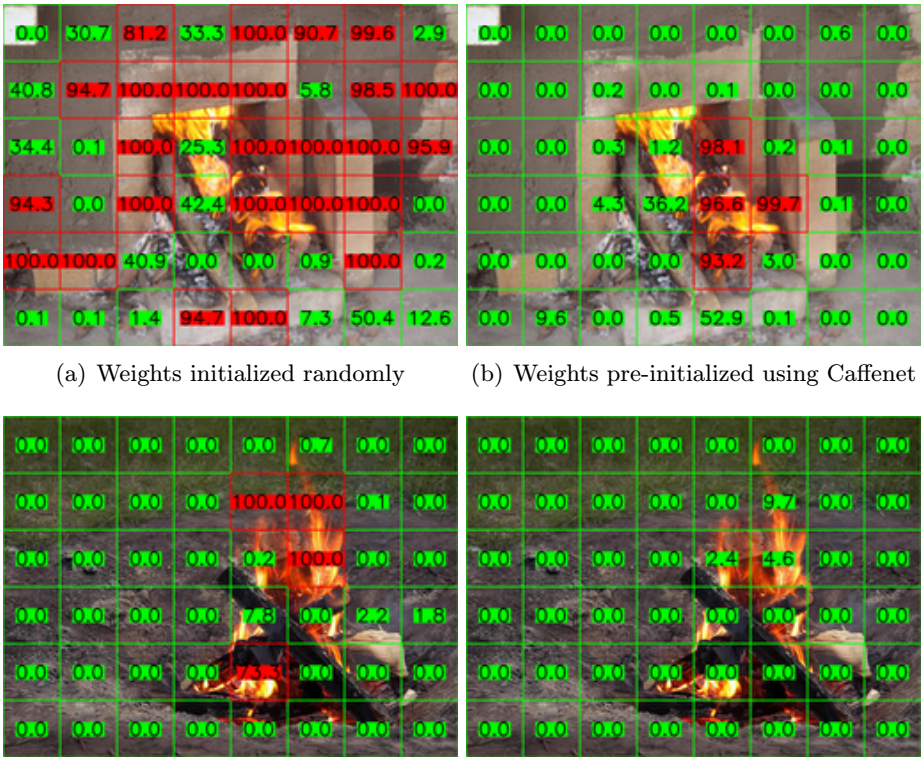
(d)



(e)

Figure 6.2: Demonstration of difference between two models by comparing the filters of first convolutional layer and their outputs: a - fire image which is filtered by kernels of the first convolutional layer (b, c) resulting in the outputs (d, e), b - filters of a model whose weights have been initialized by using a pretrained model for RGB image recognition and then updated during training with my own dataset, c - filters of a model randomly initiliazed and built entirely from scratch by my own input dataset, d - outputs of first convolutional layer in model with pretrained weights, e - outputs of first convolutional in model with randomly initialized weights.

Figure 6.2 shows how the pretrained weights change the contents and behaviour of filters




(a) Weights initialized randomly (b) Weights pre-initialized using Caffenet

(c) Weights initialized randomly (d) Weights pre-initialized using Caffenet

Figure 6.3: Examples of detector's output when using two different models: model with weights that were randomly initialized and then updated by training with my own dataset (a, c) and model with weights pre-initialized by weights of a trained Caffenet model before training with my own dataset (b, d)

(kernels) in comparison to a model with randomly initialized weights. Figure 6.2(c) displays very noisy filters. As the weights were initialized randomly, the input dataset was too small to change them during training into a cleaner form like the filters in Figure 6.2(a) built with millions of training images.

Model with weights initialized with weights of a pretrained model does achieve slightly higher probabilities of detected fire segments in a some input images and less false fire segments are detected. This can be seen in Figure 6.3(b). However, it fails to detect anything in others just like in Figure 6.3(d). The randomly initialized weights seem to be more capable of detecting fire and therefore are the preferred choice in the next experiments. Weight initialization had minimal effect on per image tests.

## 6.2 Video Experiments

### Training, Validation and Testing Datasets

During tests on images I found that $128 \times 128$-pixel segments exhibit superior performance to 256-pixel segments or per image approach. Therefore, video experiments test the video detection's results of per $128 \times 128$-pixel approach.

The samples are generated just like in image experiments but with depth always equal

to the analysed frame count. As depth is also a variable here these tests required generation and training of multiple models — creation of multiple training and validation databases. Samples are always extracted from a sequence of 3D composited images used in image tests but they are analysed to depth. For every model in this section, validation and training databases were generated separately. 3 different models were created. The amounts of samples in each database used for models' training and validation equals 2250 fire and 2250 non-fire samples just like in image tests (these are however deeper samples).

I also added the per segment models from image tests to final evaluation to compare them with the deep ones.

As for real fire testing data I used the videos from my bachelor's thesis [25] and cut them to multiple shorter sequences. This way I acquired 100 fire and 100 non-fire videos for testing.

### Video Tests Evaluation Method

For a successful fire detection, at least one fire area must be marked with a probability higher than 60%. Fire in a video, if present, must be marked within 1 to 15 seconds. With framerate set to 24 fps, fire must be detected until the first 360 have frames passed since the fire started appearing on the screen. If fire starts and the detector does not alarm its presence by marking at least one fire region then the result becomes a false negative.

Figure 6.4 shows how video tests are evaluated on fire videos. Non-fire videos follow the same scheme except the interval $\langle T_F, T_F + dt)$ from Figure 6.4 does not contain fire, $T_E$ is always equal to $T_F + dt$ and for a true negative, no alarm must be sounded. Starting frame of each video was chosen manually when I picked a frame when even the longest detection over 40 frames was already initialized.

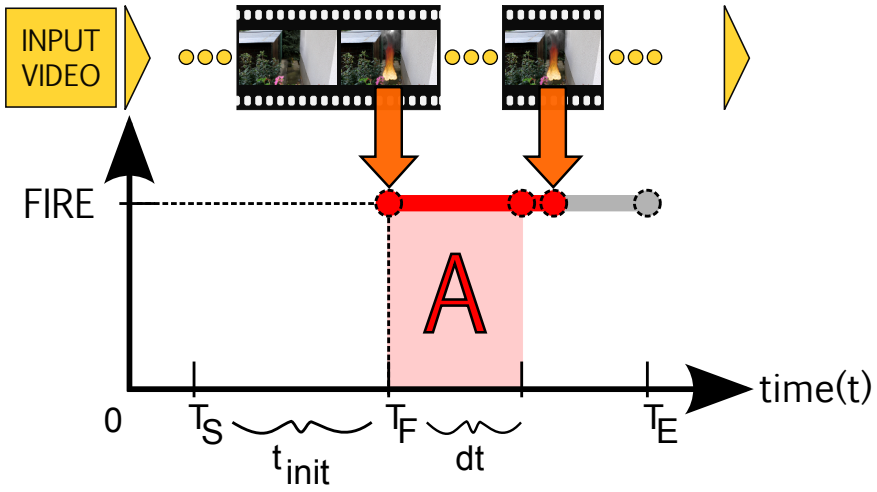



Figure 6.4: Depiction of per video experiments' evaluation on fire videos: $T_S$ is the starting time of video test (it does not need to equal to 0 as sometimes fire started minutes after the first frame and most of these frames were skipped), $t_{init}$ marks the inverval $\langle T_S, T_F \rangle$ and is always set to be 40 (the longest time window included in the tests) frames long, $T_F$ is the time of first fire frame in the video which starts the timer of 360 remaining frames (these always also include fire) or 15 seconds marked as interval $d_t = \langle T_F, T_F + dt)$, $T_E$ is the video's last frame. A - required alarm sounding area.

Real fire and non-fire videos from my bachelor's thesis [25] satisfied all the conditions of this evaluation method.

### Experimenting with Colour Analyser

These tests are the same as those for single images but only the first frame of sample's depth is analysed for colour.

### Experimenting with Video's Window Size and Stride

One of the key elements when classifying video is the access to multiple frames. This temporal feature could pose a valuable information when classifying fire based on its motion features.

Every video sample can be described by a time window that has two parameters:

**Size** which presents the number of frames which the window includes.

**Stride** that tells which $n$-th next frame is analysed in the time window, where $n = stride$.

Both parameters are shown in Figure 6.5 for demonstration. Size (sample's depth) and stride present the detector's input configuration parameters.

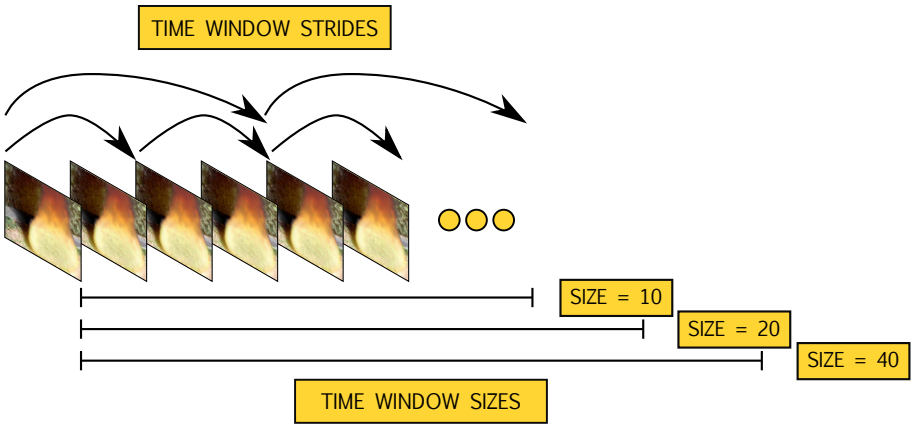
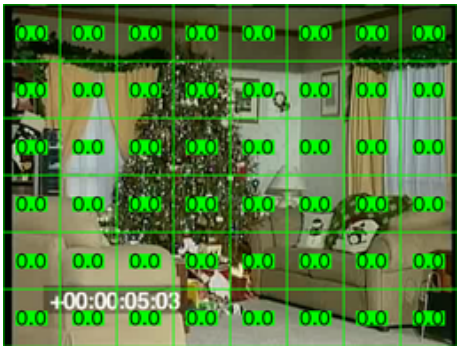



Figure 6.5: Demonstration of time window's sizes and strides.

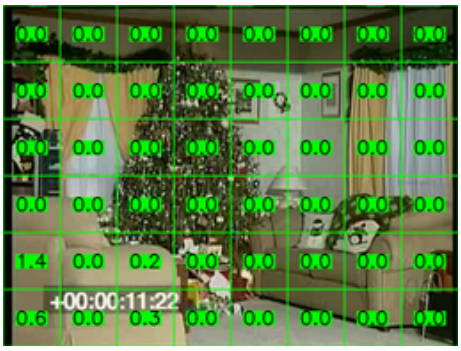
For video experiments I chose 4 different configurations of these two parameters:

1. window's size of 1 (this allows the use of image models in video tests),
2. window's size of 10 and stride of 1,
3. window's size of 20 and stride of 2,
4. window's size of 40 and stride of 4.

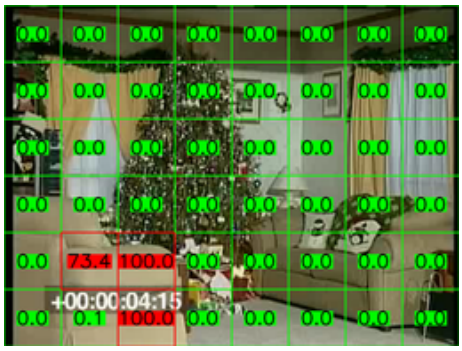
I chose these numbers this way so that there are always (except for image models) 10 frames contained in one sample. Figure 6.6 shows examples of the detector's output in the same frame of one input video.




(a) Window size 10, stride 1.




(b) Window size 20, stride 2.




(c) Window size 40, stride 4.

Figure 6.6: Examples of detector's output when using three different window size/stride configurations.

Changing the windows size had only a slight effect on final classification. From Figure 6.6 it can be seen that false detections occured in the longer time window. This is exactly the opositte result of what I expected.

## 6.3 Tests Evaluation

### Model's Parameters Settings

The net's initial learning rate was set to 0.001. In case of a higher number (faster learning) the net's weights would reach too high numbers and the net would require a change in topology. I set the number of training epochs (number of passes over the entire training data) to about 50 as higher numbers did not make a difference in validation accuracy. The only difference was that the training process took almost twice as much time. This might have been caused by the limited number of scenes that are very familiar and the net learns them too well.

### Image Tests Evaluation

The training set sizes together with the required training time for 3 models is shown in Table 6.1. The accuracy of these models tested on the validation set is presented in Table 6.2. The results show that per segment approaches are more capable of distinguishing modelled fire. The results of real fire tests can be seen in Table 6.3.

Table 6.1: Training information showing the comparison of per segment and per image approaches. Left column contains the sizes of segments or images used as input training samples (per image data contains samples of variable size). Fire and Non-fire samples present the numbers of these samples and Training time is the time (in hours) required to train the model. All the training fire samples contain only 3D modelled fire.

| **Per segment** | | | |
|---|---|---|---|
| **Sample size** | **Fire samples** | **Non-fire samples** | **Training time** |
| *128 × 128* | 2250 | 2250 | 10 hours |
| *256 × 256* | 2250 | 2250 | 10 hours |
| **Per entire image** | | | |
| *VARIED* | 900 | 780 | 5 hours |

As can be seen in Table 6.3, per image approach was the least successful. This was probably caused by many different details around the fire in an image that the classifier fails to distinguish. Per segment approach was far better scoring 100% true positives, specifically the 128-pixel model without colour analysis. Colour analysis decreases the number of false positives but also decreases the true positives. Models with initialized weights with values from a pretrained model never caught up with their randomly initialized counterparts. This might be caused by the nature of fire which is quite random.

Examples of the correct detector's per segment output on fire images can be seen in Figure 6.7. For these examples I chose the form of a heatmap to represent detections. $256 \times 256$ segments sometimes failed to find fire in comparison to smaller segments. This can be seen in Figure 6.9. Smaller segments caused more false detections though which can be seen at the example of a fire station in Figure 6.8(a). These errors were mostly caused by scenes that contained colour blobs similar to fire. An example of a detected fire per segment in a scene that contains many fire patterns and colours is shown in Figure 6.8(b). None of the methods recognised this non-fire environment.

### Video Tests Evaluation

The training set sizes together with the required training time for 5 models (the first 2 are reused from image tests) is shown in Table 6.4. The accuracy of these models tested on the validation set is presented in Table 6.5. The results of real fire tests can be seen in Table 6.6.

Table 6.6 shows that in video, the most successful was a model trained on $128 \times 128 \times 20$ (with stride 2) samples. It detected fires in all videos which is desirable. Quite close to it was the deeper model with a little more false positive detections. The deeper models are therefore the best suited for fire detection in video.

256-pixel segment models are not suitable for video as they found the smallest number of true positives. Number of analysed segments in the image matters as denser analysis

Table 6.2: Validation results comparing the approaches of per segment and per image. Left column contains the sizes of segments or images used as input validation samples (per image data contains samples of variable size). Fire and Non-fire samples present the numbers of these samples and Accuracy is the accuracy of the model's classification on the validation data. All the validation fire samples contain only 3D modelled fire and are different from the training data. This table shows how well the models can classify artifical fire.

| **Per segment** | | | |
|---|---|---|---|
| **Sample size** | **Fire samples** | **Non-fire samples** | **Accuracy** |
| *128 × 128* | 2250 | 2250 | 99% |
| *256 × 256* | 2250 | 2250 | 98% |
| **Per entire image** | | | |
| *VARIED* | 800 | 500 | 89% |

might visit more fire segments. The conducted experiments only visited each region in a grid-like manner.

Video examples can be found in the DVD content for every proposed model.

Table 6.3: Real fire image tests comparing the results of per image and per segment approaches. Left column contains the sizes of segments or images used as real input testing samples of an approach (per image data contains samples of variable size) — (I) presents initialization of this model by pretrained caffe model's weights. Second column contains the used colour analyzer settings — colour settings (N - none used, 1 - best combination of colour rules proposed in my bachelor's thesis [25], 2 - use of the least restrictive colour rule from article [4]). Third and fourth column, Fire images and Non-fire images, present the numbers of these images used for testing. TP (true positives) presents the percentage of correctly detected fire in 150 real fire images (only stating that there is/is not fire in the image by marking at least one fire area with a probability higher than 60%). FP (false positives) gives the percentage of false detections in 450 non-fire images. Processing time (P. t.) is the time required to process one sample of Sample size from the input image.

| **Per segment** | | | | | | |
|---|---|---|---|---|---|---|
| **Sample size** | **C. s.** | **Fire images** | **Non-fire images** | **TP** | **FP** | **P. t.** |
| *128 × 128* | N | 150 | 450 | 100% | 31% | 0.009 seconds |
| *128 × 128* | 1 | 150 | 450 | 96% | 20% | 0.009 seconds |
| *128 × 128* | 2 | 150 | 450 | 98% | 26% | 0.009 seconds |
| *256 × 256* | N | 150 | 450 | 81% | 8% | 0.009 seconds |
| *256 × 256* | 1 | 150 | 450 | 80% | 8% | 0.009 seconds |
| *256 × 256* | 2 | 150 | 450 | 80% | 8% | 0.009 seconds |
| *128 × 128*(I) | N | 150 | 450 | 94% | 19% | 0.009 seconds |
| *128 × 128*(I) | 1 | 150 | 450 | 88% | 19% | 0.009 seconds |
| *128 × 128*(I) | 2 | 150 | 450 | 91% | 19% | 0.009 seconds |
| *256 × 256*(I) | N | 150 | 450 | 78% | 7% | 0.009 seconds |
| *256 × 256*(I) | 1 | 150 | 450 | 75% | 6% | 0.009 seconds |
| *256 × 256*(I) | 2 | 150 | 450 | 77% | 7% | 0.009 seconds |
| **Per entire image** | | | | | | |
| *VARIED* | N | 150 | 450 | 4% | 3% | 0.010 seconds |


(a) Original fire image 1

(b) Original fire image 2

(c) Per segment $128 \times 128$

(d) Per segment $128 \times 128$

(e) Per segment $256 \times 256$

(f) Per segment $256 \times 256$

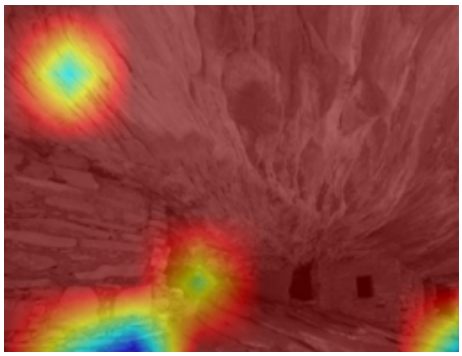
Figure 6.7: Fire detector's results on real fire images of burning houses 1 (a, c, e) and 2 (b, d, f): a, b - original fire images, c, d - heatmap output of the detector trained using $128 \times 128$ pixel samples, e, f - heatmap output of the detector trained using $256 \times 256$ pixel samples, legend: heatmap colours ranging from dark blue (0% probabilty of fire's presence) to dark red (100% probability of fire)

(a) Original non-fire image 1



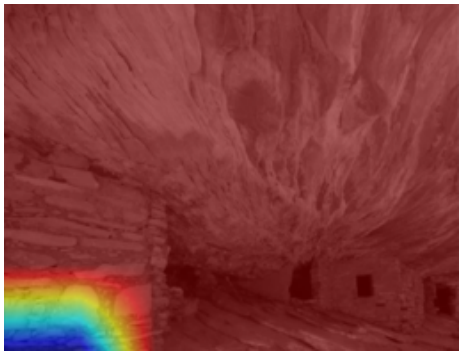
(b) Original non-fire image 2

(c) Per segment 128 × 128



(d) Per segment 128 × 128

(e) Per segment 256 × 256



(f) Per segment 256 × 256

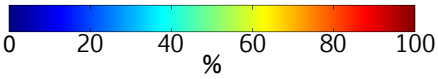
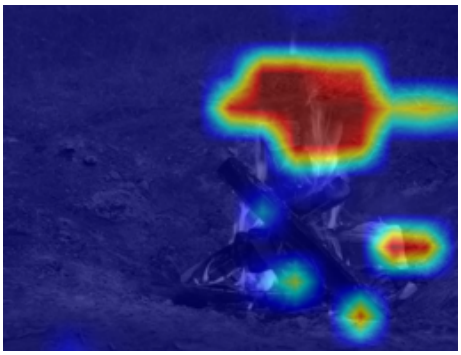



Figure 6.8: Fire detector's results on real non-fire images of a red building (a, c, e) where the 128×128 segment approach causes a false detection and a cave (b, d, f) that consits of similar patterns and colours of fire which cause many false detections: a, b - original non-fire images, c, d - heatmap output of the detector trained using 128 × 128 pixel samples, e, f - heatmap output of the detector trained using 256 × 256 pixel samples, legend: heatmap colours ranging from dark blue (0% probabilty of fire's presence) to dark red (100% probability of fire)

(a) Original fire image



(b) Per segment $128 \times 128$

(c) Per segment $256 \times 256$

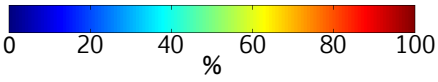



Figure 6.9: Fire detector's results on a real fire image where the $256 \times 256$ segment approach fails to detect fire: a - original fire image, b - heatmap output of the detector trained using $128 \times 128$ pixel samples, c - heatmap output of the detector trained using $256 \times 256$ pixel samples, legend: heatmap colours ranging from dark blue (0% probabilty of fire's presence) to dark red (100% probability of fire)

Table 6.4: Training results showing the comparison of different models in video tests. Left column contains the sizes of segments used as input training samples ($width \times height \times windowsize(depth)$). Fire and Non-fire samples present the numbers of these samples and Training time is the time (in hours) required to train the model. All the training fire samples contain only 3D modelled fire.

| **Per segment** | | | |
|---|---|---|---|
| **Sample size** | **Fire samples** | **Non-fire samples** | **Training time** |
| *128 × 128 × 1* | 2250 | 2250 | 10 hours |
| *256 × 256 × 1* | 2250 | 2250 | 10 hours |
| *128 × 128 × 10* | 2250 | 2250 | 17 hours |
| *128 × 128 × 20* | 2250 | 2250 | 17 hours |
| *128 × 128 × 40* | 2250 | 2250 | 17 hours |

Table 6.5: Validation results showing the comparison of different models in video tests. Left column contains the sizes of segments used as input validation samples ($width \times height \times windowsize(depth)$). Fire and Non-fire samples present the numbers of these samples and Accuracy is the accuracy of the model's classification on the validation data. All the validation fire samples contain only 3D modelled fire and are different from the training data. This table shows how well the models can classify artifical fire.

| **Per segment** | | | |
|---|---|---|---|
| **Sample size** | **Fire samples** | **Non-fire samples** | **Accuracy** |
| *128 × 128 × 1* | 2250 | 2250 | 99% |
| *256 × 256 × 1* | 2250 | 2250 | 98% |
| *128 × 128 × 10* | 2250 | 2250 | 99% |
| *128 × 128 × 20* | 2250 | 2250 | 100% |
| *128 × 128 × 40* | 2250 | 2250 | 99% |

Table 6.6: Real fire tests comparing the results of different models in video tests. Left column contains the sizes of segments or images used as real input testing samples ($width \times height \times windowsize(depth)$). Second column contains the used colour analyzer settings — colour settings (N - none used, 1 - best combination of colour rules proposed in my bachelor's thesis [25], 2 - use of the least restrictive colour rule from article [4]). Third and fourth column, Fire videos and Non-fire videos, present the numbers of these images used for testing. TP (true positives) presents the percentage of correctly detected fire in 100 real fire videos (only stating that there is/is not fire in the video by marking at least one fire area with a probability higher than 60%). FP (false positives) gives the percentage of false detections in 100 non-fire videos. Processing time (P. t.) is the time required to process one sample of Sample size from the input image.

| **Per segment** | | | | | | |
|---|---|---|---|---|---|---|
| **Sample size** | **C. s.** | **Fire videos** | **Non-fire videos** | **TP** | **FP** | **P. t.** |
| *128 × 128 × 1* | N | 100 | 100 | 95% | 88% | 0.009 seconds |
| *128 × 128 × 1* | 1 | 100 | 100 | 81% | 40% | 0.009 seconds |
| *128 × 128 × 1* | 2 | 100 | 100 | 94% | 87% | 0.009 seconds |
| *256 × 256 × 1* | N | 100 | 100 | 81% | 60% | 0.009 seconds |
| *256 × 256 × 1* | 1 | 100 | 100 | 68% | 0% | 0.009 seconds |
| *256 × 256 × 1* | 2 | 100 | 100 | 81% | 45% | 0.009 seconds |
| *128 × 128 × 10* | N | 100 | 100 | 90% | 60% | 0.011 seconds |
| *128 × 128 × 10* | 1 | 100 | 100 | 82% | 20% | 0.011 seconds |
| *128 × 128 × 10* | 2 | 100 | 100 | 88% | 60% | 0.011 seconds |
| *128 × 128 × 20* | N | 100 | 100 | 100% | 60% | 0.011 seconds |
| *128 × 128 × 20* | 1 | 100 | 100 | 88% | 20% | 0.011 seconds |
| *128 × 128 × 20* | 2 | 100 | 100 | 100% | 55% | 0.011 seconds |
| *128 × 128 × 40* | N | 100 | 100 | 100% | 60% | 0.011 seconds |
| *128 × 128 × 40* | 1 | 100 | 100 | 88% | 20% | 0.011 seconds |
| *128 × 128 × 40* | 2 | 100 | 100 | 100% | 60% | 0.011 seconds |

# Chapter 7

# Conclusion

This thesis deals with fire detection in image and video. I studied several existing methods of fire detection in image and video and created their summary. I proposed a method of fire detection using colour analysis and machine learning by deep convolutional neural networks similarly to the works of Krizhevsky et al. [1] and Ji et al. [18]. This method is unique in this field as none of the fire detection methods, that I studied, uses it.

As quality fire video sources are scarce I created my own fire videos using Blender version 2.71+ and my own compositor application. For data preparation I created 2 other applications — a sampler and a database generator which can also be used in some other project. The detector is implemented using Caffe Deep Learning Framework [2] and OpenCV library [21]. I used C++ for implementaion. Experiments test the detector on image and video datasets using 2 approaches during training — per image and per segment.

I filmed and created 3D models of 2 outdoor scenes. I created 8 different fire simulations in these scenes. Then I rendered 2000 frames of fire animations and composited them with the filmed scenes into complete fire videos. I used them to train 7 different models using 2 approaches to detection. These were validated on similarly modelled fire and non-fire images and videos and all scored $98 - 100\%$ accuracy. For real fire tests in both image and video, per segment approach with $128 \times 128$ segments reached 100% correct fire detections. I reached and surpassed my goal of 95% of true positives detected. Reaching 100% is the most important outcome for a fire detector. False positives reached 31% for this model in images and 55% in videos. Time required to process a all segments in an image(frame) was in hundreds of miliseconds.

The things and ideas that I consider the most viable are:

- use of a modern method of deep learning for fire detection,
- training of the fire detector based entirely on unreal 3D modelled fire,
- successful experiments on images and video.

As a continuation of this work, more varied scenes (e.g. city environments) for fire sequences creation could be added and rendered which could improve the trained model. As my solution's speed was not my primary goal, faster solution could be achieved by using Caffe's net surgery (converting the last fully connected layers to convolutional) and using the network to apply a sliding window detection itself. This would also achieve dense classification on any input image with a variable size.

# Bibliography


[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[3] J. Qunitiere, *Principles of fire behavior*. Career Education Series, Delmar Cengage Learning, 1998.

[4] T.-H. Chen, P.-H. Wu, and Y.-C. Chiou, "An early fire-detection method based on image processing," in *Image Processing, 2004. ICIP '04. 2004 International Conference on*, vol. 3, pp. 1707–1710 Vol. 3, Oct.

[5] B. U. Töreyin, Y. Dedeoğlu, U. Güdükbay, and A. E. Çetin, "Computer vision based method for real-time fire and flame detection," *Pattern Recogn. Lett.*, vol. 27, pp. 49–58, Jan. 2006.

[6] C.-B. Liu and N. Ahuja, "Vision based fire detection," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 4, pp. 134–137 Vol.4, 2004.

[7] M.-H. Yang and N. Ahuja, "Gaussian mixture model for human skin color and its application in image and video databases," in *Proc. SPIE: Storage and Retrieval for Image and Video Databases VII*, vol. 3656, pp. 458–466, 1999.

[8] T. Çelik and H. Demirel, "Fire detection in video sequences using a generic color model," *Fire Safety Journal*, vol. 44, no. 2, pp. 147–158, 2009.

[9] T. Çelik, H. Demirel, H. Özkaramanli, and M. Uyguroglu, "Fire detection using statistical color model in video sequences," *J. Vis. Comun. Image Represent.*, vol. 18, pp. 176–185, Apr. 2007.

[10] E. Persoon and K. S. Fu, "Shape Discrimination Using Fourier Descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, pp. 388–397, Mar. 1986.

[11] R. Collins, A. Lipton, and T. Kanade, "A System for Video Surveillance and Monitoring," in *Proceedings of the American Nuclear Society (ANS) Eighth International Topical Meeting on Robotics and Remote Systems*, April 1999.

[12] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland, "Pfinder: Real-Time Tracking of the Human Body," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, pp. 780–785, 1997.

[13] H. Tian, W. Li, L. Wang, and P. Ogunbona, "A Novel Video-Based Smoke Detection Method Using Image Separation," in *Multimedia and Expo (ICME), 2012 IEEE International Conference on*, pp. 532–537, July 2012.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[15] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Computer Science Department, University of Toronto, Tech. Rep*, vol. 1, no. 4, p. 7, 2009.

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[17] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *CoRR*, vol. abs/1411.4038, 2014.

[18] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional Neural Networks for Human Action Recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, pp. 221–231, Jan 2013.

[19] J. van Doorn, "Analysis of deep convolutional neural network architectures," 2014.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[21] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[22] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU Math Expression Compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like Environment for Machine Learning."

[24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," 2014.

[25] T. Poledník, "Detekce ohně v obraze a videu," bakalářská práce, Brno, FIT VUT v Brně, 2013.

# List of Appendices



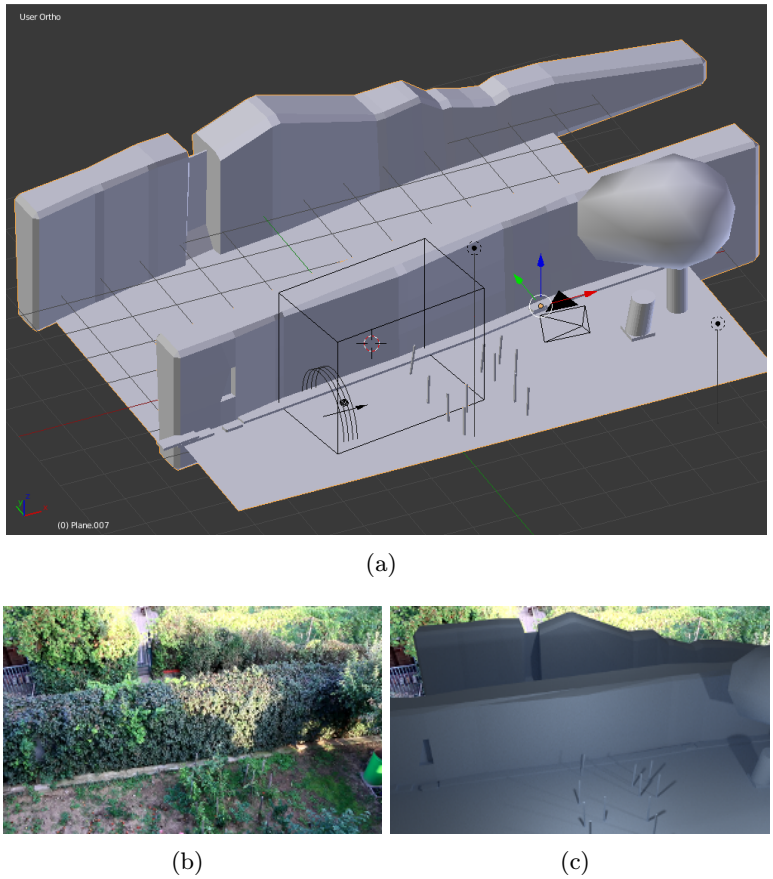
## DVD content

- Source codes of the compositor, sampler, generator and detector applications with example configuration files in `/src/` folder in directories of the same name.
- Source codes of the required version of Caffe library in `/src/Caffe/`.
- Compiled sampler and compositor applications for CentOS 5.8 and generator and detector applications for Ubuntu 14.04 in `/bin/` folder in directories of the same name.
- Created Blender 2.71+ 3d models together with *Python* (`.py`) and Windows batch (`.bat`) script files needed for batch rendering in `/models/` folder.
- Scripts for easy manipulation with the rendered data in `/models/scripts` folder.
- Examples of created rendered images in `/composite/images/` folder.
- Examples of created and composited videos in `/composite/movies/` folder.
- Examples of detected fire in images in `/test/images/` folder.
- Examples of detected fire in videos in `/test/videos/` folder.
- This thesis in PDF format in `/thesis/` folder.
- Source codes of this thesis in format for LaTeX system with images in `/thesis/src/` folder.

# Appendix A

# Demonstration of the Created 3D Scenes

The first scene's demonstration can be found in Figure 4.2 in Chapter 4. Other 3D models created for fire scenes used for the detection testing are shown in Figure A.1.




(a)



(b)



(c)

Figure A.1: Demonstration of the created Scene 2 and its comparison to the original image: a - scene's 3D model view in Blender, b - original image used as modelling template, c - rendered scene.

# Appendix B

# Examples of the Composited Fire Images

Figures B.1 and B.2 show examples of composited fire images from different scenes.

(a)

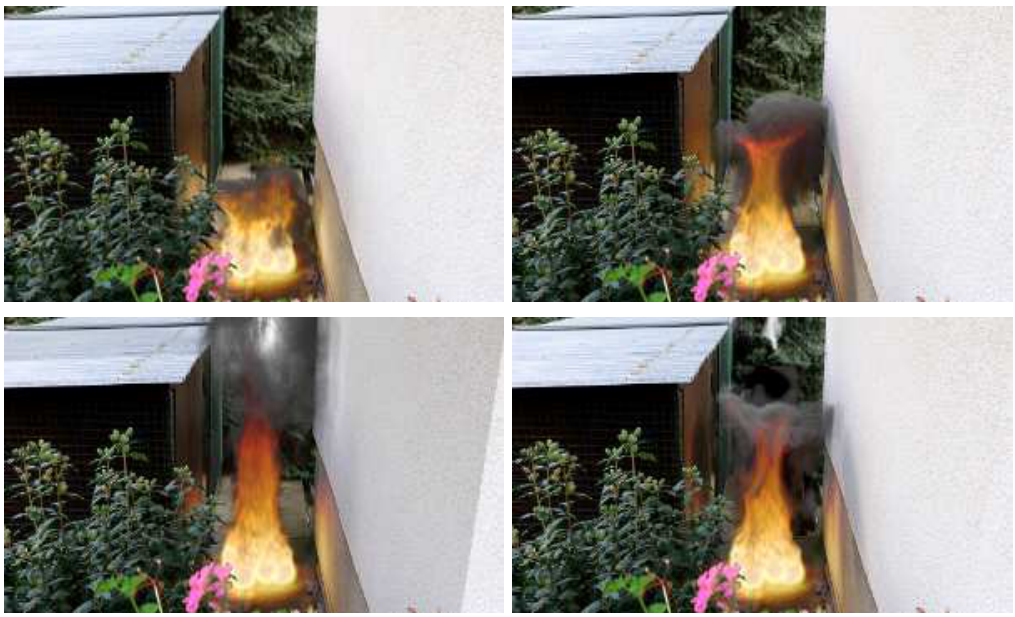


Figure B.1: Examples of composited fire images from Scene 1: a - original video frame image before composition.

(a)

Figure B.2: Examples of composited fire images from Scene 2: a - original video frame image before composition.

# Appendix C

# Manuals to the Applications

For compilation of the applications there is a Makefile prepared in the root source `src/` folder which runs the Makefiles of the corresponding applications. To compile the compositor and sampler applications, only standard C++ libraries and OpenCV library [21] of version 2.39 or higher are required. Generator and detector also require Caffe [2] and Cuda libraries whose paths must be set in their own Makefiles. All of the programs are supposed to be worked with using their command-line interfaces. The sampler, generator and detector applications use configuration files named **sa_config.cfg**, **ge_config.cfg** and **de_config.cfg**. The names of the final binary files are **compositor**, **sampler**, **generator** and **detector**. Manuals with. More detailed information with examples and commentaries can be found in the help (parameter `-h/--help`) messages and configuration files of the particular application.

## Manual to the Compositor Application

Even though the compositor class is a standalone object and can be simply included in any C++ and OpenCV project, it is mainly meant to be used with its command line interface (CLI). The CLI parameters are as follows.

```
compositor [PARAMETER] ARGUMENT
```

Compositing program uses 2 modes to analyze input:

1. reading input and showing the composited output in an interactive window (currently available only in image mode),
2. reading input and writing the composited output to a file.

Parameter description:

`-i/--image`

- composite a single image
- if parameter `-w/--write` is not specified, program creates a new interactive window with compositor's output
- created window is controlled using keyboard keys:
  - 'm' - show original
  - 'n' - show compositor's output
  - 'ESC' - cancel compositor and exit

- if this parameter is used together with `-v/--video` and `-w/--write`, video frames are written to .png images

`-v/--video`

- composite video with a set of specified images
- parameter `-w/--write` must be currently also specified when using this option
- if this parameter is used together with `-i/--image` and `-w/--write` options, video frames are written to `.png` images

`-w/--write path_to_output_file`

- when using images, instead of creating a new window for presentation, save to file
- argument: path to output file
- in image mode the program creates an interactive new window by default and visualizes the composited image
- when applied to a video with a format supported by OpenCV, program creates a new video from the compositor's output with the same width, height, codec type the input video has and framerate of 24 fps
- when applied to both the image and the video mode (`-v` and `-i` specified at the same time), output video is written to `.png` images, every image is named using `argument` + `frame_number` + `.png` naming scheme
- when applied to an image with a format supported by OpenCV, program creates a new image from the compositor's output with the same width and height
- support for given image is given by its extension
- support for given video is given by its extension (container) and codec

`-g/--graphic output_window_size`

- set the interactive window to be of a fixed input size or a scaleable size
- argument: output window size string
- argument can be either `fixed` or `scale`
- `fixed` sets the size of the output window to that of the input image and forbids its change
- `scale` sets the size of the output window to a smaller size than the size of the input and allows its change
- default value is `fixed`

`-m/--mask_im path_to_write_mask_image_file`

- set path to the write mask of the program argument image (or video)
- argument: path to file
- this image serves as a regular mask which allows a change of only those pixels which are of white colour on mask
- when used with video, the same mask is applied on all frames

`-l/--light_mask_im path_to_light_mask_image_file`

- set path to the light (emission) mask for composition with the program argument image
- argument: path to file
- when used with video, the path string is supposed to look like: `./Im_name.png`; the compositor takes the value of the argument and prepends a number (starting from the number given by `-b/--begin_frame` parameter argument and ending at `-e/--end_frame` argument) in front of its suffix beginning with a dot, for example the light mask for frame number 1 (frames are numbered from zero) is going to be `./Im_name1.png` (when `-b` argument equals zero)
- when used with images, the path is used as is with no additions

`-s/--shadow_mask_im path_to_shadow_mask_image_file`

- set path to the shadow mask for composition with the program argument image
- argument: path to file
- when using video, naming scheme is the same as with the `-l` parameter

`-f/--foreground_im path_to_foreground_image_file`

- set path to the foreground for composition with the program argument image
- argument: path to file
- when using video, naming scheme is the same as with the `-l` parameter
- the image is supposed to contain alpha channel which is used for simple blending with the argument image or video frame

`-c/--convert_to_BW_mask`

- conversion of final composited image to black-and-white image
- black colour becomes white and all other colours change to black

`-r/--repeat`

- when compositing a video which contains more frames than there are between `-b` and `-e` arguments, the first frame is used again right after the last frame
- this behaviour is repeated over the entire video

`-b/--begin_frame number_of_first_frame`

- the starting frame number which is added to the image paths mentioned above
- argument: number of the first frame (positive integer or zero)
- the default value is 0

`-e/--end_frame number_of_last_frame`

- the ending frame number which is added to the image paths mentioned above
- argument: number of the last frame (positive integer or zero)
- the default value is 250

`-p/--parse_from frame_number`

- video frame from which the composited images are applied
- argument: frame number
- the default value is 250

`-n/--number_frames frame_count`

- number of frames of the final video when written
- argument: written video frame count
- the default value is 0 (meaning all input video frames)

`-h/--help`

- print this help

`argument`

- the program requires one argument depending on used parameters `-i/` or `-v`:
  `-i/--image` : path to image file with format supported by OpenCV
  `-v/--video` : path to video file with format supported by OpenCV
  `-i` and `-v` : path to video file with format supported by OpenCV
- the image given by this argument is not supposed to contain alpha channel and when it does, it is ignored

The `-r/--repeat`, `-b/--begin_frame`, `-e/--end_frame`, `-p/--parse_from` and `-n/--number_frames` options are only usable in video or the combined mode. When used with a simple image, these options are ignored.

Examples of usage:

- `./compositor im1.jpg -f ./fire/ImFire1.png`
  Show alpha blended `ImFire1.png` on `im1.jpg` opening it in a window with `fixed` window size.
- `./compositor -i -w im2.jpg -s ./sh_mask/ImSh21.png im1.jpg`
  Write the compositor output (image) of `im1.png` multiplied by `ImSh21.png` shadow mask to file `im2.jpg`.
- `./compositor -l ./fire_em_mask/Im0052.png -f .fire/Im0052.png`
  `-s ./fire_sh_mask/Im0052.png -m ./mask.png`
  `-g scale ./MVI_5380.png`
  Show composition of image `MVI_5380.png`, with masked parts given by image `mask.png`, and light mask, shadow mask and foreground named `Im0052.png` in their correspoding subdirectories.
- `./compositor -v -w video2.mp4 -m ./mask.png -f ./fire/ImFire.png`
  `-s ./sh_mask/ImSh.png -b 2 video1.avi`
  Write the compositor output (video) of `video1.avi` to file `video2.mp4` with the zeroth frame composited with `./fire/ImFire2.png` foreground, `./sh_mask/ImSh2.png` shadow mask, first frame composited with `./fire/ImFire3.png` foreground, `./sh_mask/ImSh3.png` shadow mask etc.

If none of the `-i/--image` or `-v/--video` parameters is included, the program always analyzes images by default. The order of parameters is interchangeable. The frames are numbered from zero. All parameters and their arguments can be omitted. The program requires 1 argument described above.

OpenCV allows opening video stream out of an image file, however as there is no delay between frames (there are no frames) when writing to output, most video players will not be able to read it. This operation is however permitted as some players might be able to do so. This applies to both the compositor and the detector application described below.

## Manual to the Sampler Application

```
sampler [PARAMETER] ARGUMENT
```

The sampler program can work in 5 modes (which are set in the configuration file) when sampling input images:

1. sampling images according to the sampling rules using images and labels,
2. generating only negative samples according to the sampling rules using images and labels,
3. generating only positive samples according to the sampling rules using images and labels,
4. generating only negative samples while omitting the sampling rules,
5. generating only positive samples while omitting the sampling rules.

The sampler is mainly controlled by its configuration file and command line parameters only control the base.

Parameter description:

`-i/--in_first_num input_file_first_number`

- set first number suffix of input file name
- argument: input image (label) file's first number
- other parts of the file's name can be found in the configuration file

`-o/--out_first_num output_file_first_number`

- set first number suffix of output image file name
- argument: output file's first number
- other parts of the file's name can be found in the configuration file
- for every next output file, this number is incremented by 1
- the default value is 0

`-d/--depth sample_depth`

- set sample's depth
- argument: sample's depth

`-n/--number_images number_of_input_images`

- set the number of input images (labels) to determine the suffix number of the last one
- argument: number of input images

`-w/--write path_to_output_directory`

- set path to output directory
- argument: path to output directory
- all outputs set in the configuration file are written here
- the default value is `sa_config.cfg` int the working directory

`-s/--settings path_to_configuration_file`

- set path to configuration file
- argument: path to configuration file
- by default the program looks for `sa_config.cfg` in the working directory

`-e/--erase_old_db`

- erasing the contents of the database text file before writing new entries into it
- by default the program just appends new entries starting from the last line

`-h/--help`

- print this help

`argument`

- the program requires one argument:
  - path to directory with input image and label directories
    whose names can be set in the configuration file

Examples of usage:

- `./sampler -i 0 -d 6 -n 249 ./samples`
  Sample sequence using a configuration file `sa_config.cfg` starting with input file ending with number 0, analysing the depth of 6 frames and considering the last file's number (in the sequence) to be 248 ($249(imagescount) + 0(firstnumber) - 1$).

The order of parameters is interchangeable. All parameters without a default value are required. The argument must follow right after its parameter. The program requires 1 argument described above. Samples can be generated as images and database generation instruction file or only as this file.

## Manual to the Generator Application

```
generator [PARAMETER] ARGUMENT
```

Parameter description:

`-l/--list`

- list the contents of the input database text file (amounts of different labels) and check for settings errors without generating anything
- when this parameter is specified, processing ends just before creating the databases

`-v/--validation path_to_validation_database`

- create a validation database with this path
- argument: path to output validation database
- the generated database does not need to be used for validation, it is just a database that contains the generated data

`-t/--training path_to_training_database`

- create a training database with this path
- argument: path to output training database
- the generated database does not need to be used for training

`-d/--db_backend backend_name`

- set the backend to be used for database storing
- argument: backend name
- argument can be one either `lmdb`, `leveldb`

`-r/--random_shuffle`

- shuffle the order of loaded file entries (samples) randomly
- by default no shuffling is performed

`-b/--balance balancing_method`

- balance the amount of entries stored in both databases
- argument: balance option string
- argument can be one of `labels`, `labels_max` and `all`
- `labels` balances the databases according to the first method (1) depicted in Figure 5.6, `labels_max` according to 2. method and `all` according to 3. method
- `labels` and `all` can be performed only when both training and validation databases are specified, while `labels_max` also works on a single database
- balancing always discards elements that break the training/validation ratio (given by `-n` parameter)
- by default no balancing is perfomed (this defaults to `all` balancing without discarding)

`-n/--number_train number_to_training_before_validation`

- sets how many entries are stored in the training database before storing an entry in the validation database
- argument: trainining/validation ratio, e.g. 2 means 2:1 train./val. ratio
- this option can only be used if both training and validation databases are specified
- when working with balancing, samples are split according to this ratio and still follow the rules of balancing options
- without balancing, no discarding is performed and the remaining samples out of the ratio are put into the training database
- by default, if no balancing is performed, training/validation ratio is 1:1

`-s/--settings path_to_configuration_file`

- set path to configuration file
- argument: path to configuration file
- by default the program looks for file `ge_config.cfg` in the working directory

`-h/--help`

- print this help

`argument`

- the program requires one argument:
  path to database instruction text file
  (e.g. the one generated by the sampler)

Examples of usage:

- `./generator -d lmdb -v ./val -t ./train -b all ./lmdb_inst.txt`
  Generate balanced sets of entries (both databases contain the same amount of samples) for `lmdb` training and validation databases in `train` and `val` directories in the working directory from text file `lmdb_inst.txt` using configuration file `ge_config.cfg`.

If only one database is specified, all the entries in the input file are stored in it (without considering the `labels_max` balance option). The order of parameters is interchangeable. All parameters without a default value are required. The argument must follow right after its parameter. The program requires 1 argument described above. The generation of databases is controlled by its configuration file and command line parameters only control the base.

## Manual to the Detector Application

```
detector [PARAMETER] ARGUMENT
```

The detector program uses 2 modes to analyze input:

1. reading input and showing the detection output in an interactive window,

2. reading input and writing the detection output to a file.

Parameter description:

`-i/--image`

- detect fire in image
- if parameter `-w/--write` is not specified, program creates a new interactive window with detector's output
- created window is controlled using keyboard keys:
  'm' - show original
  'n' - show detector's output
  'ESC' - cancel detector and exit

`-v/--video`

- detect fire in video
- if parameter `-w/--write` is not specified, program creates a new interactive window with detector's output
- created window is controlled using keyboard keys:
  'SPACE' - pause/play video,
  / cancel frame-by-frame mode
  'a' - if possible, shorten the delay between consecutive frames by 1ms
  's' - extend the delay between frames by 1ms
  'd' - start frame-by-frame mode,
  'ESC' - cancel detector and exit
- in the frame-by-frame mode every key except for ESC and SPACEBAR forwards the video by 1 frame

`-c/--camera`

- detect fire in camera stream
- this option uses the same controls as video
- when stopping/restarting video, the frames may appear inconsistent
- camera must be connected and recognised by OpenCV

`-w/--write path_to_output_file`

- when using images, instead of creating a new window for presentation, save to file
- argument: path to output file
- in image mode the program creates an interactive new window by default and visualizes the fire detection
- this parameter may only be used with image and video file detection, not with camera stream
- when applied to a video with a format supported by OpenCV, program creates a new video from the detector's output with the same codec type the input video has, framerate of 24 fps and the width and height given by the configured size in the settings

- when applied to an image with a format supported by OpenCV, the program creates a new image from the detector's output with the width and height given by the configured size in the settings
- support for given image is given by its extension
- support for given video is given by its extension (container) and codec

`-g/--graphic output_window_size`

- set the interactive window to be of a fixed input size or a scaleable size
- argument: output window size string
- argument can be either `fixed` or `scale`
- `fixed` sets the size of the output window to the configured size and forbids its change
- `scale` sets the size of the output window to a smaller size than the configured size and allows its change
- default value is `fixed`

`-s/--settings path_to_configuration_file`

- set path to configuration file
- argument: path to configuration file
- by default the program looks for file `de_config.cfg` in the working directory

`-p/--polyimage path_to_polynom_image_file`

- set path to polynomial image required for use of YCrCb polynom colour rule
- argument: path to image file containing polynom boundaries
- by default the program looks for file `poly.png` in the working directory

`-n/--net path_to_net_definition_file`

- set path to Caffe net definition (deploy) file
- argument: path to net definition file
- by default the program looks for file `deploy.prototxt` in the working directory

`-m/--model path_to_trained_dcnn_model_file`

- set path to Caffe trained model (caffemodel) file
- argument: path to model file
- by default the program looks for file `model.caffemodel` in the working directory

`-h/--help`

- print this help

`argument`

– the program requires one argument depending on used parameters `-i/` or `-v`:
`-i/--image` : path to image file with format supported by OpenCV
`-v/--video` : path to video file with format supported by OpenCV
`-c/--camera` : index of camera device recognised by OpenCV (e.g. 0)

Examples of usage:

- `./detector image1.jpg`
  Detect fire in `image1.jpg` opening it in window with fixed image size using configuration file `de_config.cfg`, polynomial image file `poly.png`, net `deploy.prototxt` and model file `model.caffemodel`

- `./detector -i -w image2.jpg image1.png`
  Write detection output (image) of `image1.png` to file `image2.jpg` using configuration file `de_config.cfg`, polynomial image file `poly.png`, net `deploy.prototxt` and model file `model.caffemodel`

- `./detector --video -p palo.png -m c_20000.caffemodel video1.avi`
  Detect fire in `video1.avi` opening it in window with fixed video size using configuration file `de_config.cfg`, polynomial image file `palo.png`, net `deploy.prototxt` and model file `caffe_train_iter_20000.caffemodel`

- `./detector -v -w video2.mp4 video1.avi`
  Write detection output (video) of `video1.avi` to file `video2.mp4` using configuration file `de_config.cfg`, polynomial image file `poly.png`, net `deploy.prototxt` and model file `model.caffemodel`

- `./detector -c 12`
  Detect fire in stream from device with index 12 (index is unique to OpenCV and can be determined only by using OpenCV's interface functions) opening it in window with fixed camera input size using configuration file `config.cfg`, polynomial image file `poly.png`, net `deploy.prototxt` and model file `model.caffemodel`

If none of the `-i/--image`, `-v/--video or -c/--camera` parameters is included, the program always analyzes images by default. The order of parameters is interchangeable. All parameters and their arguments can be omitted. The program requires 1 argument described above.