UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR'S THESIS no. 6347

VISUALIZATION OF CLASS ACTIVATION MAPS FOR IMAGE CLASSIFICATION

Marijan Smetko

Zagreb, June 2019.

Deep convolutional models are the principal ingredient in many computer vision applications. However, these models are often criticized for lack of interpretability, that is, inability to explain their decision to humans. Consequently, techniques for interpreting and visualizing decisions of convolutional models represent a very interesting research topic. This thesis consists of the following tasks. Choosing a framework for automatic differentiation and getting acquainted with libraries for manipulating matrices and images. Analyzing and briefly describing existing approaches for interpreting and visualizing decisions of deep models. Choosing an appropriate convolutional model, initializing it with public parameters trained on ImageNet, and fine-tuning it for classification on FGVCx. Evaluating and presenting the classification accuracy. Interpreting classification decisions on high-ranked false positives and low-ranked false negatives. Suggest suitable directions for future work. The thesis is to be accompanied with source code of developed methods, as well as with employed datasets, instructions and documentation. The literature has to be properly cited. Received assistance has to be documented.

I want to show my immense gratitude to my parents,
father Damir and mother Dubravka,
for always being supportive in everything I did my whole life.

Table of Contents

1 Introduction

Computer vision was always an active research field, and with recent successes of machine learning and deep learning, the results are getting more impressive with every new day. However, what deep learning models lack is interpretability – a neural network is given several thousand input images and output labels, and with some linear algebra and calculus and a little bit of luck, it finds a global minimum of a loss function and it can start to classify images. Yet, no one actually knows what exactly *is* what the model learns and in what way it really affects the final decision for the class of a given image. Therefore, if we are to really understand deep neural networks, we must delve deeper into the models themselves. This thesis uses a basic type of exploring, using visualizations of feature activation maps for the top predicted class and how it helps a model learn to classify various images. The images come from a publicly available, and a bit modified, FGVCx dataset. The first chapter presents convolutional models for visual recognition. The chapter introduces the notation, describes the basic building blocks, presents the typical structure of the model and explains the training procedure. The second chapter presents the main ingredients of the state-of-the-art convolutional models such as batch normalization, residual connections and adaptive training. The third chapter presents the design and various details of the developed implementation. The most interesting excerpts from the source code are carefully explained at the level of individual expressions. The fourth chapter presents experimental results. The final chapter provides the conclusions and suitable directions for the future work.

2 Convolutional neural networks

2.1 Artificial neural networks and deep learning

An artificial neural network (ANN) is an artificial intelligence model which was designed after the human brain in the middle of the 20th century. Its basic unit is the neuron, a cell which takes an arbitrary number of inputs and gives only one output by calculating a weighted sum of inputs. In the early days of the model, the output, or the activation, was a step function which fired *'1'* if the weighted sum was over some threshold, but modern era improvements use other activation functions such as the sigmoid, the hyperbolic tangent and the rectified linear function, which are all differentiable. The neurons are grouped into layers – every neuron has a separate set of weights used for the calculation of the weighted sum, and each neuron's inputs are all of the outputs of the previous layer. That kind of layer is called a Fully Connected (FC) layer[1]. The output of the whole layer is elementwise application of the layer's activation function, which is then fed as input to the next layer. The depth of the neural network is defined as the number of layers the network has, and if it is deep enough, we enter a branch of machine learning called deep learning[2].

$$\boldsymbol{x}^{\boldsymbol{T}} = [x_1\, x_2 \ldots x_n] \qquad \boldsymbol{a}^{\boldsymbol{0}} = \boldsymbol{x} \tag{2.1}$$

$$\boldsymbol{W}^{\boldsymbol{l}} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{bmatrix} \tag{2.2}$$

$$z_i^l = \sum_{j=1}^{m} w_{ij}^l a_j^{l-1} \qquad \boldsymbol{z}^{\boldsymbol{l}} = \boldsymbol{W}^{\boldsymbol{l}} \cdot \boldsymbol{a}^{\boldsymbol{l-1}} \tag{2.3}$$

$$\boldsymbol{a}^{\boldsymbol{l}} = \sigma\left(\boldsymbol{z}^{\boldsymbol{l}}\right) \tag{2.4}$$

1 FC layers are the most common, but there are layers which are not fully connected. Some of them will be presented later.

2 There is no official threshold, but typically neural networks are two to hundreds of layers deep

More formally, a neural network is a list of linear and non-linear transformations. It is said to be L layers deep if it has L repetitions of linear and non-linear transformations. The input is an *n*–dimensional column vector (2.1), and the *l*–th ($1 \leq l \leq L$) neuron layer with *n* inputs and *m* outputs is presented as a matrix $\boldsymbol{W}^{l}$ in (2.2). The weight w_{ij} represents the strength of the connection between the *i*-th neuron of this layer and *j*-th neuron of previous layer. Although this seems counterintuitive, it allows a compact notation for the weight matrix. The product (2.3) represents the linear transformation of the previous layer's activation, and the layer's output is simply an elementwise nonlinear transformation. If the neural network has L layers, then the final, L-th layer's output $\mathbf{a}^{L}$ is the output of the whole network.

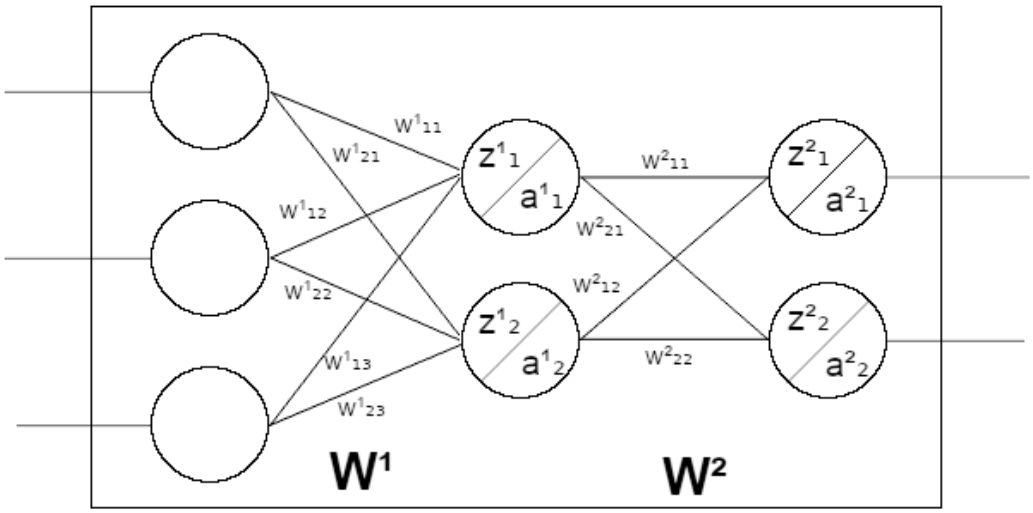

Figure 2.1: *Basic scheme of a fully connected artificial neural network with three inputs and two hidden layers. As per equation 2.3, the input is first linearly mapped from 3-dimensional to 2-dimensional space. This vector is acted upon with the first layer's activation function, according to 2.4, resulting with a vector which then becomes the input for the second hidden layer. After repeating the same procedure once again, second layer's output becomes the output of the whole network*

2.1.1 Training

To train a neural network we need to define a loss, also called cost, function $L(y_{true}, y_{pred})$ (not to confuse with the number of layers L) which tells us how far away the network's prediction is from the value it should predict. The most common loss functions are mean squared error $\left(f(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \right)$, most commonly used for regression tasks, and

cross entropy $\left(f(\boldsymbol{y},\hat{\boldsymbol{y}})=\frac{1}{n}\sum_{i=1}^{n}\left(y_i\cdot\ln(\hat{y}_i)+(1-y_i)\cdot\ln(1-\hat{y}_i)\right)\right)$, most commonly used for classification tasks. The goal of the training is to optimize the loss function to be as low as possible, and for that we need to know the derivatives of the loss function w.r.t. every prediction and the derivatives of the activation functions w.r.t. both the input vector and the weight matrix. The algorithm called backpropagation allows us to propagate the error from the end of a network through the layers back to the start, updating the weights on the walk back through the layers.

There are two important assumptions for the cost functions: a) it should be written as an average of the costs for individual examples, and, b) it must be a function of the network's outputs. In that way, we can use the loss function as a measure of "error" the network produces. Loss function is usually differentiable, and that property is exploited to calculate the vector of gradients w.r.t. activations for the last layer, the exit neurons ($\boldsymbol{\delta}^L$ vector). The key concept here is that the $\boldsymbol{\delta}$ vector for the last layer can be used not just to compute the gradient of the weights, but to also compute the $\boldsymbol{\delta}$ vector for the second-to-last layer using the chain rule of differentiation. That $\boldsymbol{\delta}$ vector is further used for the $\boldsymbol{\delta}$ vector of the directly previous layer and so on until every weight of the whole network is updated. The big downside of the backpropagation algorithm is that all of the temporary inputs need to be remembered for the correct computation of the backward gradient, which can require vast amounts of computer resources.

2.2 Convolutional neural networks (CNN)

ANNs and their derivations have proved to be extremely successful for various tasks such as financial applications, speech synthesis, and video game AI. Usages of ANNs in computer vision were an improvement over the usual machine learning methods, e.g., the support vector machine classifier, but the results were not as spectacular as expected. The problem was how the network was using its input images, or more precisely, what amount of input features was taken into consideration. An image is a rectangular grid of pixels that are only locally correlated. For example, in a typical image of a cat, the two eyes are close together and the position of a tail has little to do with the position of eyes. Feeding an image 'as is' to the ANN could potentially make it learn that some distant corner parts are correlated, when

they shouldn't be. The convolutions were invented to prevent that, and to force image to focus on the small region.

2.2.1 Convolutions

Convolutions are the core building blocks of the CNN. The convolution kernels[3] are small, usually square matrices, that are applied in a special way to a window of the input image to produce a scalar output. They are applied elementwise to the values of the pixels of the input image, as shown in figure 2.2. If the input image is grayscale, which means it only has a width and a height, the applied convolutions are two-dimensional, and if the input image is in color, which means it has a width and height for each of the three colors, the convolutions are three-dimensional. In any case, the output of the convolution operation is a 3D tensor. The purpose of the convolutions is to detect what features are present in the input image, so there are convolutions that detect edges, circle parts or any other meaningful combination of pixels. There are many convolutions per layer, and each convolution is slid across the whole image, portion by portion, and applied as described, resulting in a feature map, which is then stacked together with other feature maps to make the output tensor. Usually, the dimensions of an image get smaller through layers, and the number of convolution filters gets larger.

3 The convolution kernels are also called feature detectors or filters. These terms will be used interchangeably

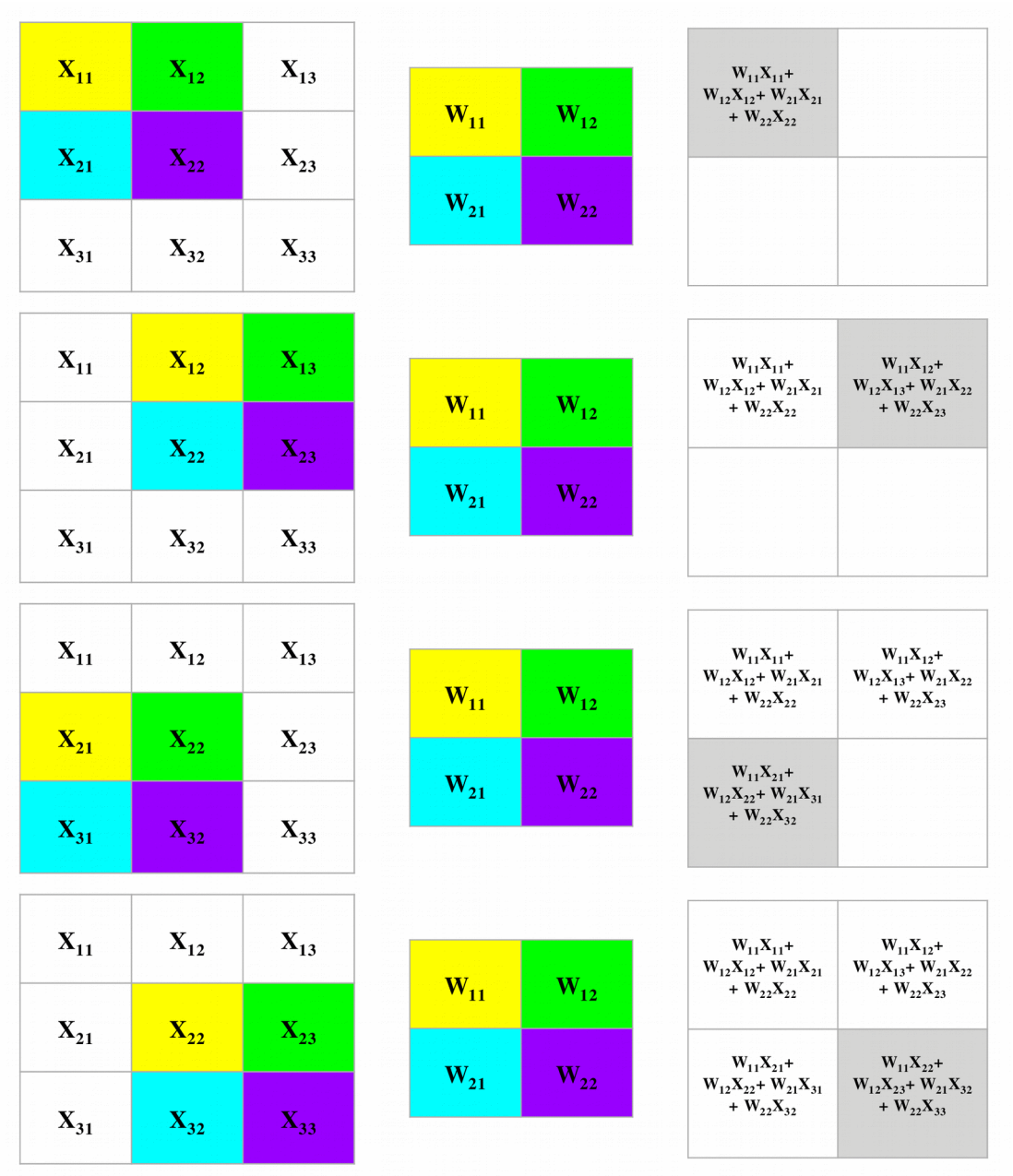

Figure 2.2:[12] *An application of 2x2 convolution filter* ***w*** *on 3x3 image* ***x****. First, the window is aligned with the top left corner of the image, and the image values underneath are multiplied with corresponding filter weights. The sum of partial products are stored in a feature map in the top left corner. The convolution window is then slid right with stride one, and the procedure is repeated, with the result being stored in the top right corner. As the window now reached the end of the image, next iteration will start below, repeating the same procedure and storing results in appropriate cells.*

More formally, a convolution kernel is an f-by-f for single channel, or f-by-f-by-c for c channel input tensor, array of weights which is applied as described to the current portion of the image to produce a scalar which indicates how strongly the feature is detected. Here f stands for the width and

height of the convolutional filter, as they are usually square. It's important to notice that, if the convolution is of the same size as the input image, it behaves just like the fully connected layer. Each successive application of the convolution kernel is on a new part of the image moved by s (which stands for 'stride') to the side or down. If we gather all the scalars produced by one convolution kernel in an array of numbers, we get a feature map for that convolution. Different convolutions have different feature maps, and they are stacked together for another important step of the CNNs – the pooling step.

2.2.2 Pooling operation

The pooling step is frequently, but not necessarily, found after the convolution step. The core pooling operation is similar to the convolution, but it is used for a different purpose. Images are usually of very large dimensions, and to process an image in reasonable time we must somehow downsample the image without losing too much information. Rescaling is very expensive so pooling was invented as a fast alternative. The idea is to have a window, just like the convolution, and to calculate the average of the numbers inside the window. It's like a convolution with every weight set to $1/f^2$, with the difference being that the weights are not learnable. Also, another important difference is that no pixel can show up in more than one pool, which is to say pooling operations segment the image in disjunctive areas, unlike convolution which can have a size of, say, 3x3 but have a stride of 2. The described pool operation is also known as the average pool (avg pool). It is mostly replaced by an alternative, max pool which simply takes the maximum value in the window, as shown in figure 2.3. It is nonlinear and works better in practice.

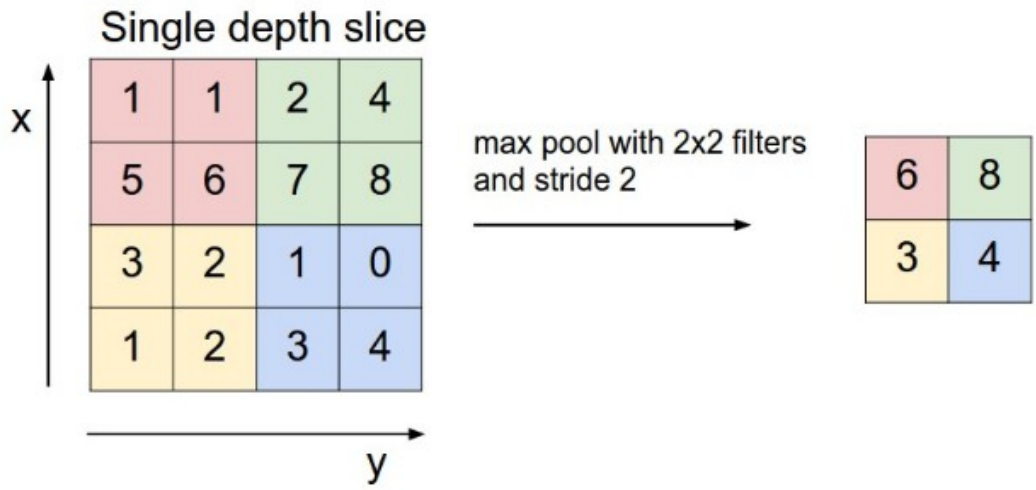

Figure 2.3[6]*:A visualization of the max pool operation. In this example, the 4x4 image is sliced in disjunct 2x2 areas, and the maximum number in respective areas are selected and stored in a new matrix. That matrix is the output of the max pool layer.*

Pooling layers reduce the memory footprint needed to process an image by downsampling, as it was already mentioned. They discard the excess information which could potentially misdirect the next layer, since it's more important for the detected features to be in relative position one to another, than to have an exact location on an image. In the cat image example, it's useful for the two detected eyes to be close to the whiskers and ears as it will allow us to maybe detect the head in the next layer. It was long thought this was crucial to the success of convolutional models, but invertible models[3] were invented that retain the whole information and can both restore the original image and have great accuracy. Nevertheless, the pooling operation is used very much in practice and yields great results.

The output of the CNN layer (the convolution and pool together) is an activation function (usually ReLU) applied to the output of the max layer. This output is the input to the next convolution layer[4] and so forth. The output of the last layer is then fed into the global average pooling module. The resulting F-dimensional vector is processed by a fully connected layer with softmax activation which then finally classifies the input image. Convolution layers here have a role of feature extraction as they are more powerful, in comparison to ordinary fully-connected networks, and, in practice, perform with greater accuracy.

4 Usual number of convolution layers is measured in tens

2.2.3 Training

The training of convolutional neural networks is very similar to the training of fully connected networks. It starts the same, by calculating the value of the loss function and backpropagating the **δ** vector to the start of the ANN, and it is then applied backwards through the convolutional layers and its activation functions. In max pool, only one component of a convolution window has a gradient, the others have 0. In avg pool, all components of the convolution window have the same gradient. This gradient is then used to update all the convolutions, and it can be used to calculate a new **δ** vector for the previous convolutional layer. The downside of this algorithm is, still, the need to remember the temporary inputs, with an amplification, since images are usually very large, just like the number of convolution operations. For reference, a famous ResNET 50 convolutional model, which was used for this thesis' implementation, has ~500MB of weight tensors by itself, but during training it requires about 7.5 GB of RAM[10].

2.3 A word on PyTorch

PyTorch is an automatic differentiation framework, written in the programming language Python. Its core feature is the dynamic creation of computational graphs in the form of directed acyclic graphs, built out of nodes which represent the basic operations such as addition, scalar multiplication, matrix multiplication and so on. PyTorch knows how to differentiate these elementary operations, and by using the chain rule it can differentiate any function with respect to any of the input variables or intermediary results. Simply invoking .backward() method on the PyTorch variable starts the chain rule propagation of derivatives over all input nodes over the edges of the graph, as every node remembers the inputs it has gotten. This proved to be very useful for the implementation of the backpropagation algorithm, and it was heavily used throughout the practical work of the thesis. On a final note, .backward() propagates back and calculates the differentials, but it does not perform an actual weight update. The update is usually done in special components called optimizers, of which the most well known is Adam. The Adam optimizer was exclusively used in the implementation of the thesis, and is explained in section 3.1.

3 Modern approaches and state-of-the-art

As the field of deep learning flourished with the exponential increase of computing power in the late 20th and early 21st century, many new techniques and procedures were invented to help combat problems models showed in their infancy. Some solutions were directed towards the problem of the vanishing gradient. It turned out some activation functions were pushing the gradient towards zero, and only a few of the last layers were updating their weights – the norm of the **δ** vector was too small for the weight updates to be meaningful, so new activation functions were invented. Other solutions turned to another frequent issue, the overfitting. The idea of training a machine learning algorithm is for it to generalize well – to make correct predictions on input data it never saw – but it's often the case a model somehow fails to generalize and instead it focuses on specificities of the training set, thus learning in a wrong way. This thesis will cover only some of the improvements which had the greatest effect in improving training and accuracy of the neural network models, namely adaptive training, batch normalization, residual connections and transfer learning.

3.1 Adaptive training

The weight updates are rarely done evaluating the whole training set, but small batches that approximate the correct gradient and allow more frequent weight updates, thus making a model converge faster. The neuron cells which contribute the most to the outcome, important neurons, usually get updated most often, and the ones who don't, usually get small updates. But, important neurons should be updated with smaller increments to stabilize the learning, and less important could be updated with bigger updates that won't ruin the stability.

To update weights differently, a new information, beside the global learning rate, is needed, and it's how a weight has been updated in the previous iterations. One way to accomplish that is to accumulate the previous updates, and then divide the **δ** vector with it. If the accumulation is big, a neuron gets a smaller update, and vice versa. This is a description of the AdaGrad algorithm, with a difference it actually accumulates squares of the gradient, and divides them with a square root.

The problem with AdaGrad is that the update magnitude decays exponentially and becomes infinitesimally small as the number of iterations

grows, which in practice means the updates stop being meaningful across the whole network. A better solution comes from Geoffrey Hinton, who modified rprop algorithm to calculate a moving average over several last epochs with exponential decay. The global learning rate is then divided by the square root of the average of squares and multiplied with the **δ** vector, hence the name, root mean square prop, or RMSProp. RMSProp calculates a gradient, it's square, the new mean and makes the update of weights. The gradient calculated in this way can make the convergence rather slow, but it can be improved with Nesterov momentum. After the gradient is calculated and the weights are updated, another gradient calculation is performed in this new point, which is then used as a correction for the first gradient. Although Nesterov momentum introduces some extra computational complexity, it allows the learning rate to be bigger due to more precise gradient, which makes the network to converge faster.

Algorithms covered so far only kept track of the mean squared magnitude of the vector, but not the direction. Algorithm that keeps track both of the mean gradient and mean momenta is called Adam, and in practice works the best. The reason is that by calculating a moving exponential decay average, perturbations of the stochastic gradients cancel out and the net result is always pointing in the general direction of the slope towards the minimum, which allows us once more to use a bigger learning rates. The gradient of one batch is, therefore, used only to update the mean so far, and in the original paper terminology, is considered as "velocity". Note that this can also be combined with Nesterov momentum, creating a variant called Nadam.

3.2 Batch normalization

When training the ANN, the input data is usually normalized to have a mean of 0 and variance of 1, so the first hidden layer's inputs always have the same distribution. But, other layers as input get the activations of the previous layers, with the distribution which is changing over time as the weights get updated throughout the epochs. A small change in the first layers can get amplified, causing a huge shift before it gets to the last layers. This is a reason to insert one batch normalization layer after every hidden layer, which then normalizes a batch of inputs from its mean and variance to some other μ_B and σ_B. They are learnable parameters that also get updated with gradient descent.

It was widely accepted that batch normalization reduces the covariate shift[7], a change in distribution, but some scholars[4] found out it's not actually the case. The covariate shift is still happening, but it's the objective function that's getting smoothed out, which allows a network to converge more easily with learning rates that don't have to be as small as before.

3.3 Residual connections

Up until now, this thesis always assumed the input of one layer is only constituted out of the output of the previous layer. That does not have to be always the case. We can combine the outputs from two or three layers before with the outputs of the layer before and feed it in this layer. The type of the connection that allows this is called residual connection, also known as the skip connection, as the outputs 'skip' a layer or more, as shown in figure 3.1.

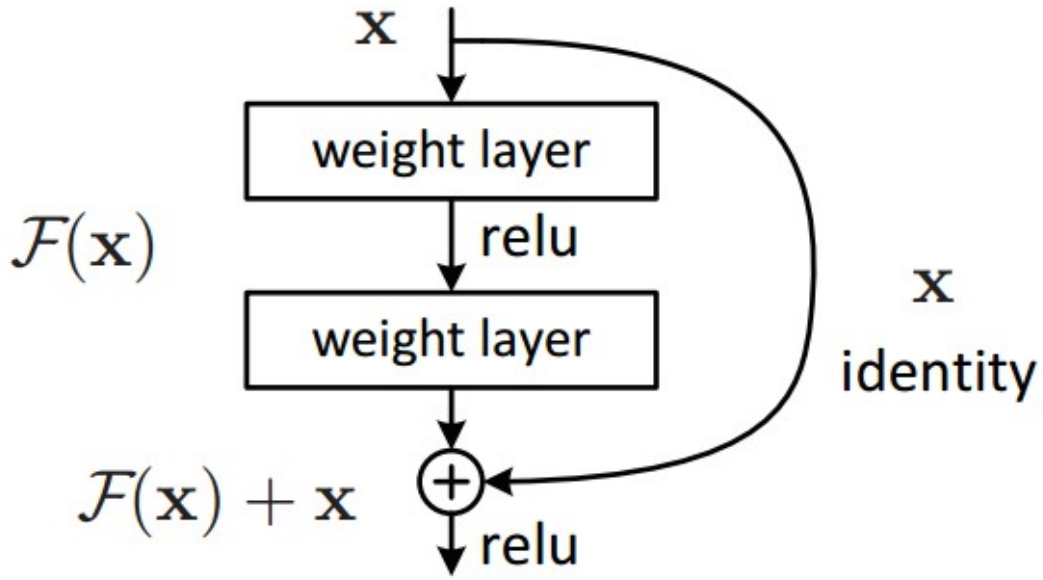

Figure 3.1*: A scheme of a residual, or a skip connection. The first weight layer processes its input with linear and nonlinear transformation, producing the output. This output is then linearly transformed, but, before applying the activation function, the previous input is added. The resulting sum is then acted upon with the layer's activation function. The output is composed of the input and the difference from the input and target output, so the layer only has to learn that difference, or a residue.*

The reason behind the name 'residual' is, since they as inputs have both the outputs from a layer and two layers before, is the previous layer only has to learn a difference, or a residue, between those two vectors. In practice they improved accuracy and generalization the most in comparison with Adam or batch normalization. The reason models with residual connections,

or ResNets for short, keep outperforming other models is that they are effectively making the network more shallow[5], allowing the information to flow further and simplifying the model. Until 2018, Resnets have been the most successful models in ImageNet training, and then were superseded by EfficientNet, which will not be discussed.

3.4 Transfer learning

Training a neural network from the scratch is very difficult due to the vast hyper-parameter space that needs to searched, even with multiple acceptable solutions. However it was observed that a network, once trained for a specific task, can further be used for another similar task, after a bit of fine tuning for the new task. A good analogy is how us humans use things we learned in one situation for another situation, e.g. the model that differentiates home cat species can be later improved to differentiate between the wild cat species. This observation is called transfer learning.

One explanation for the possibility of transfer learning in the field of computer vision is that features detected in one type of images usually exist in other types, such as borders, lines and circles. The new network that would be trained from start would have to learn those same features all over again, which is very time costly. Instead, since the feature detectors can theoretically work with new images, we can adapt the old network in much simpler and shorter way. This also usually yields greater accuracy.

The important assumption is that the old and the new image datasets are reasonably similar. Cat classifier will may not transfer successfully towards skin tumor identification. For this reason, it’s important to train a network on a general purpose dataset which covers a large number of various distinct classes, for the network to generalize well. Examples of such datasets are ImageNet, MNIST, CIFAR, Pascal VOC and many others. This way, the convolutions really learn to extract the mutual features which appear in very different contexts, thus generalizing well. Such trained network can afterwards be fine tuned for a specific use by first adapting the last layers of the network to the inputs before and ‘locking’ the earlier gradients to zeroes, and once the saturation point is reached by unlocking the whole model for training, which further increases accuracy for the specific purpose.

4 Activation map visualization model

4.1 Data

To visualize activation maps for images, it's necessary to have both the dataset of images and the trained model for image classification. This thesis is based on two works[1][2] which used The Pascal VOC 2007 dataset[11], that contains just under 10.000 images in 20 classes with an average of ~2.5 objects per image. In this thesis' implementation only one of several possible annotations was actually used. This dataset was only used as a support for the main dataset, which was the FGVCx dataset with fungi species, stripped down to 50 most numerous classes, with a little over 13.300 images, 150 for validation and the rest for train. These images were used in 2018 Kaggle competition, and in the original form feature almost 1400 various fungi species, from underground truffles, ordinary ground mushrooms, hoof mushrooms and lichen, with occasional microscope imagery or schematic drawing. Extra preprocessing includes square cropping the images around the center and rescaling them to have an exact 256x256x3 volume, as some images were grayscale.

4.2 The model

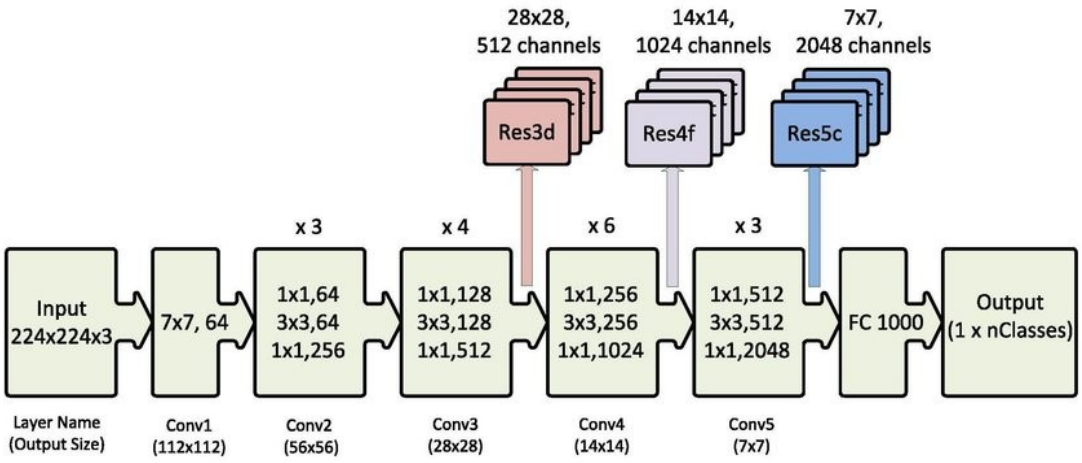

Figure 4.1: *The architecture of ResNet50 model. Input images are 224x224x3 tensors. They are first processed with 64 7x7 convolutions and 3x3 max pool, that output 56x56 sized tensor. Resulting tensor is further processed by iterations of 1x1 and 3x3 convolutions with varying count and channel numbers. After some of the iterations, downsampling is performed to gradually reduce the tensor size. In the end, entire image tensor is compressed to 2048 dimensional vector which is then fed to the ordinary ANN with* `class_count` *number of neurons that finally classifies the image.*

As it was discussed in the chapter 3.4, convolutional neural networks are commonly pretrained on a general dataset and then pretrained for the specific use. This thesis used Residual nets, specifically, ResNet50 model, whose architecture is shown in figure 4.1[5]. This model proved to be complex enough to extract meaningful features from the FGVCx images, but not too complex for it to train or progress slowly. There was no multi-label classifying, but only one multi-class model with the number of outputs equal to the number of classes of the dataset (50 for FGVCx, 20 for Pascal VOC 2007). The loss function this model was optimizing was Cross Entropy Loss. For weight updates Adam optimizer was used, but without Nesterov momentum. Initial learning rate was $8.5 \cdot 10^{-4}$ but was exponentially decayed after every epoch with varying intensity.

4.3 The implementation

```
import torch.nn as nn
from torch.utils.data import DataLoader
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms #import Compose, Normalize, ToTensor

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = models.resnet50(pretrained=True)
set_parameter_requires_grad(model, feature_extracting=True)
#"""
img_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])
])
#WORKDIR defined in the first code section
fungi_train = FungiTrainDataset(WORKDIR, img_transform)
fungi_val = FungiValDataset(WORKDIR, img_transform)
fungi_val_vanilla = FungiValDataset(WORKDIR, lambda t: t)
dataloader_train = DataLoader(fungi_train, batch_size=32,shuffle=True,
                              num_workers=8, pin_memory=True)
dataloader_val = DataLoader(fungi_val, batch_size=50, shuffle=False,
                            num_workers=3, pin_memory=True)
dataloaders = {"train": dataloader_train, "val": dataloader_val}
SECOND_LAYER = 594
THIRD_LAYER = 173
model.fc = nn.Linear(2048, fungi_train.getClassCount())
model.to(device)
optimizer = optim.Adam([param for param in model.parameters() if param.requires_grad],
                       lr=8.5e-4, weight_decay=1.2e-6)
decayed_optim = optim.lr_scheduler.ExponentialLR(optimizer, 0.975)
loss = nn.CrossEntropyLoss()
model, val_acc_history, train_acc_history = train_model(model, dataloaders, loss, decayed_optim,
                                                        device, num_epochs=10, fc_at_epoch=9)
```

***Figure 4.2**: Model training setup with Python programming language and PyTorch differentiation framework. Entire code was developed, tested and run in Google Colab environment. The dataset was stored in Google Drive and accessed with gdrive module. The model architecture is freely available to download, pretrained or not.*

5 This is an example of the network which reduces image size but increases the number of channels

To train a neural network using PyTorch, it's necessary to prepare the dataset of images, image preprocessing steps, the machine on which the training will occur, the very model, its loss function, weight optimizer and so on. As seen in figure 4.2, in line 8, immediately after the imports, the device is selected. GPU units are preferred over CPU as they can process batches in parallel which speeds up the training process, but a fallback option is needed. Next, a model object is instanced by downloading an ImageNet pretrained copy which is freely available for unrestricted usage. Lines 12 to 20 define a composition of transformations used for preprocessing every image in batch. Images, represented in memory as PIL.Image objects, are enhanced by random transformations to augment the dataset.

```
1  class FungiDataset(Dataset):
2      """
3      A class modelling a default organization of FGVCx 2018 challenge dataset
4      (https://github.com/visipedia/fgvcx_fungi_comp)
5      """
6      def __init__(self, dataset_root, transform):
7          self.root = dataset_root
8          self.transform = transform
9          self.data_json = None
10         return
11
12     def __len__(self):
13         return len(self.data_json['images'])
14
15     def __getitem__(self, i):
16         assert 0 <= i < len(self)
17         filename = self.data_json['images'][i]['file_name']
18         img = mimg.imread(self.root + filename) #returning the [0, 255]
19         if len(img.shape) != 3:
20             img = np.stack((img, img, img), axis=2)
21
22         img = self.transform(img)
23         category = self.data_json['annotations'][i]['category_id']
24         #mask = np.zeros(self.getClassCount())
25         #mask[category] = 1
26         return {"image": img, "class": category}
```

***Figure 4.3**: Image dataset representation boils down to subclassing a Dataset class, which abstracts away the details of image handling. Only two methods must be overriden, the getter of one image and its label, and the dataset size. PyTorch automagically packs multiple these images to tensors which are then given to the mode as input.*

They could be flipped horizontally, since a mirror image of a mushroom is still a mushroom, and they could have colors slighty changed, since darker image is still of the same class (or any other image property that's changed) and then transforming the resulting image to PyTorch tensor with normalization of the red, the green and the blue color channels of the image, so the input image distribution has zero mean and unit variance. This transformation composition is given as an argument in the constructor of the dataset object, whose structure can be seen in figure 4.3. In this figure, lines 15 to 26 are responsible for loading an image, ensuring it's not grayscale, and returning a dictionary with the image and corresponding image class.

Lines 25 to 28 back in figure 4.2 define DataLoader objects, which know how to communicate with PyTorch Dataset classes to load them in multiple threads to batch tensors. Other arguments are the number of images in one batch and whether or not to immediately push the resulting tensor to a GPU, and whether or not to read the dataset sequentally. In the last steps, the model is changed to reflect the classes of FGVCx dataset, and not ImageNet, and pushed to the GPU for faster computation. The final train function call is then only preceded by the definition of the optimizer component, which is Adam, and the loss function, which is CrossEntropy.

4.4 Training procedure

The train_model function accepts the following arguments: a model, dictionary of data loaders, loss (or criterion) function, optimizer, device object, the number of epochs and the number of the epoch in which updating only the last layer stops and updating the whole network begins. As the entire function has a lot of feedback prints and statistic collection, only the most interesting excerpt is showed in figure 4.4 that directly updates the model.

```
35          for phase in ['train', 'val']:
36              if phase == 'train':
37                  model.train()  # Set model to training mode
38              else:
39                  model.eval()   # Set model to evaluate mode
40
41              running_loss = 0.0
42              running_corrects = 0
43
44              # Iterate over data.
45              for i, batch in enumerate(dataloaders[phase], 1):
46                  inputs = batch["image"].to(device)
47                  labels = batch["class"].to(device)
48                  optimizer.zero_grad()
49
50                  with torch.set_grad_enabled(phase == 'train'):
51                      outputs = model(inputs)
52                      loss = criterion(outputs, labels)
53
54                      _, preds = torch.max(outputs, 1)
55
56                      # backward + optimize only if in training phase
57                      if phase == 'train':
58                          loss.backward()
59                          decayed_optimizer.optimizer.step()
```

***Figure 4.4**: Portion of the training procedure*

Lines 35 to 40 prepare the model for one of two phases, training and evaluating phase. In the training phase, the partial inputs and the gradients are remembered, as they are needed for the backpropagation algorithm. In the evaluation phase, the model is not updated and none of the forementioned tensors are needed. Next, batch tensors of images and

classes are extracted from the data loader. It's mandatory for the both input tensors, output tensors and the model to be on the same device, and lines 46 and 47 send the tensors to the same device there the model is. The most important step is in line 51 where the inputs are finally applied to the model, giving outputs which are then compared with our criterion function to estimate the loss. Backpropagating the loss in line 58 computes the derivatives all the way back to the very first layer (as explained in section 2.3) and then optimizer updates the weights via method call in line 59.

```
12 def returnCAM(feature_conv, weight_softmax, class_idx):
13     # generate the class activation maps upsample to 256x256
14     size_upsample = (256, 256)
15     bz, nc, h, w = feature_conv.shape
16     output_cam = []
17     for idx in class_idx:
18         cam = weight_softmax[idx].dot(feature_conv.reshape((nc, h*w)))
19         cam = cam.reshape(h, w)
20         cam = cam - np.min(cam)
21         cam_img = cam / np.max(cam)
22         cam_img = np.uint8(255 * cam_img)
23         output_cam.append(cv2.resize(cam_img, size_upsample))
24     return output_cam
25
26
27 class CAMapper:
28     def __init__(self, model, finalconv_name):
29         def hook_feature(module, input, output):
30             self._features_blobs.append(output.data.cpu().numpy())
31             return
32         self._features_blobs = []
33         self.net = copy.deepcopy(model)
34         self.net.eval()
35         self.net.to("cpu")
36         self.net._modules.get(finalconv_name).register_forward_hook(hook_feature)
37         params = list(self.net.parameters())
38         self.weight_softmax = np.squeeze(params[-2].data.numpy())
39         return
40
41     def process(self, image, do_transform=False):
42         height, width, _ = image.shape
43         if do_transform:
44             normalize = transforms.Normalize(
45                mean=[0.485, 0.456, 0.406],
46                std=[0.229, 0.224, 0.225]
47             )
48             preprocess = transforms.Compose([
49                transforms.ToPILImage(),
50                transforms.Resize((224,224)),
51                transforms.ToTensor(),
52                normalize
53             ])
54         else:
55             preprocess = lambda t: t
56
57         img_tensor = preprocess(image)
58         img_variable = Variable(img_tensor.unsqueeze(0))
59         logit = self.net(img_variable)
60         h_x = F.softmax(logit, dim=1).data.squeeze()
61         probs, idx = h_x.sort(0, True)
62         probs = probs.numpy()
63         idx = idx.numpy()
64         CAMs = returnCAM(self._features_blobs[0], self.weight_softmax, [idx[0]])
65         heatmap = cv2.applyColorMap(cv2.resize(CAMs[0],(width, height)), cv2.COLORMAP_JET)
66         result = image * 0.7 + cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB) * 0.3
67         return [result.astype(np.uint8)]
```

Figure 4.5: *OO approach to CAM visualization. The code was adapted from publicly available repository of the code this thesis is based on.*[9]

4.5 Activation map visualization

Activation maps are maps of the activation magnitude in the last convolutional layer. For the ResNet50 example, the last convolutional layer has 2048 feature maps which are subsequently reduced to a vector by global average poooling module. This is output is then linearly mapped to a 50 dimensional space to get the prediction vector. The activation map of the most probable class, the highest exit scalar, is plotted with a weighted sum of that neuron's input weights and 2048 convolution activations. The resulting array is the activation map of an image for the predicted class. This procedure is illustrated in figure 4.5.

The code for the generation of the images is borrowed from public repository[9] and adapted to the object oriented paradigm. Constructor of the `CAMapper` class in lines 28 to 39 takes the model as input, and registers a private list for activations to be put in. This way, every time a picture is sent through a model the activations can be accessed. The core of the procedure is in method `.process()` which takes NumPy array and gives it to the model, preprocessed or not, to produce outputs and then calls a helper function to actually a compute the class activation map of the class with the highest output. The helper function in lines 12 to 24 takes does the matrix product of weights and activations in line 18, normalizes the result matrix to be in range [0, 255] in line 22, and then rescales the resulting matrix to be the same size as the input image in line 23. Afterwards, back in the process method in line 65, a heat map is taken of the resulting CAM as to show the activation magnitudes in colors, which is finally interpolated with the input image in line 66 to produce a visualization.

5 Experimental results

The goal of this thesis was to train a neural net classifier, visualize the class activation maps in convolutional neural networks, and test whether or not the maps can be used as a localization technique.

5.1 Classification results

5.1.1 FGVCx

FGVCx dataset proved to be very hard for training an artificial neural network classifier. First of all, the dataset contains very different types of images, from ordinary mushroom images in nature, over images of mushrooms that are sliced open, to microscope images of fungi spores or even hand drawn sketches. With that variety comes another difficulty which is intrinsic to this dataset, and it's that many species of fungi share a common 'mushroom-like' figure. This is proved experimentally by calculating different top-k accuracies, where k stands for an integer of how many top probabilities are taken into consideration. It's easily seen in figure 5.1, last row, that error drops from 25% when considering only the top prediction, to less than 10% if considering top three predictions, and under 5% if considering top 5 predictions. This means the model has a general idea of what the species does the fungus belongs to, but cannot pinpoint the exact class because the classes are simply too similar one to each other. Visual representation of this data can be seen in figures 5.2 and 5.3. The data shows consistent loss dropping and accuracy raising for train, but not for val, which shows some noise around the saturation point. Significant differences between accuracy thresholds can be seen. This is further shown in confusion matrices for the train and val set in figures 5.4 and 5.5 respectively. It's also easily seen the confusion matrices don't have a strong diagonal, but have a few strong columns which correspond to the most predicted classes.

	Train				Val			
epoch	*loss*	*top1*	*top3*	*top5*	*loss*	*top1*	*top3*	*top5*
1	2.6970	29.88	51.04	61.93	2.1489	42.67	66.00	75.33
15	1.2186	63.71	84.94	91.09	1.7223	50.00	74.67	86.00
30	1.0603	68.62	87.87	93.27	1.7495	48.67	76.67	85.33
45	0.9547	71.11	90.02	94.68	1.6253	55.33	79.33	89.33
60	0.8973	73.31	90.85	95.14	1.5351	60.00	79.33	86.00
75	0.8617	74.44	91.38	95.62	1.6085	54.67	76.67	87.32

Figure 5.1: *Table of experimental data while training on FGVCx. Training loss is slowly converging, but validation loss reached a plateau around 1.6. Visual representation of this table is shown in the next pages.*

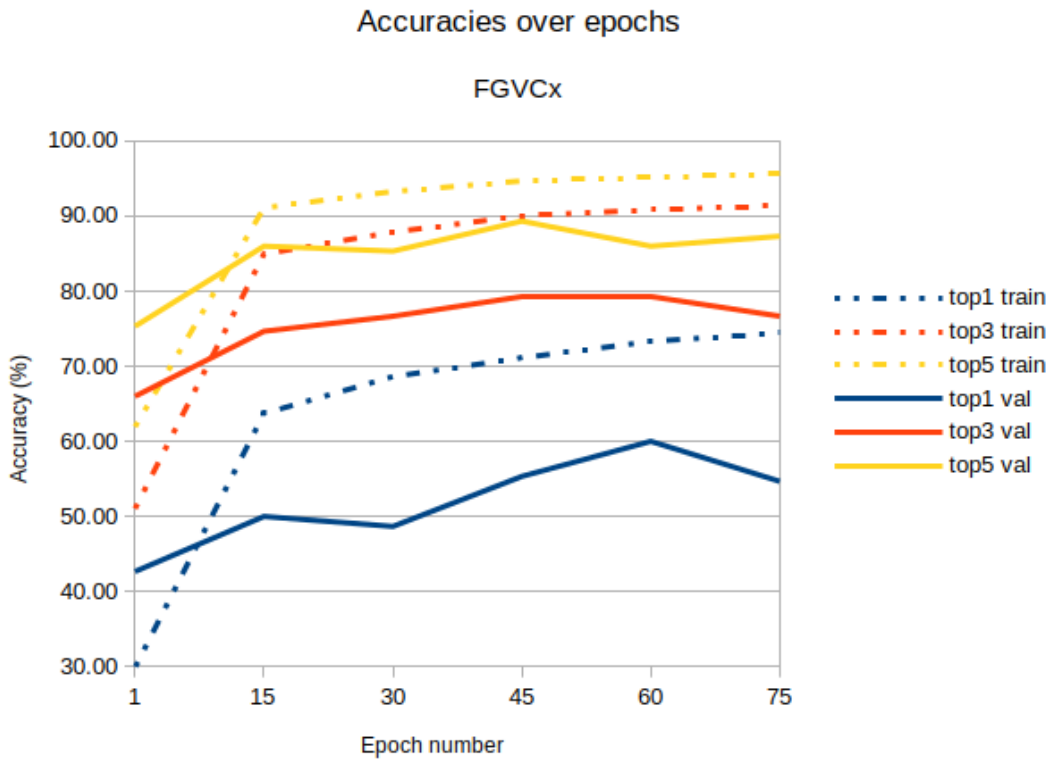

Figure 5.2: *Graph of accuracies while training on FGVCx. The gains of taking more than just the first prediction are significant. Training accuracies are rising steadily over the epochs, but validation accuracies oscillate around some value.*

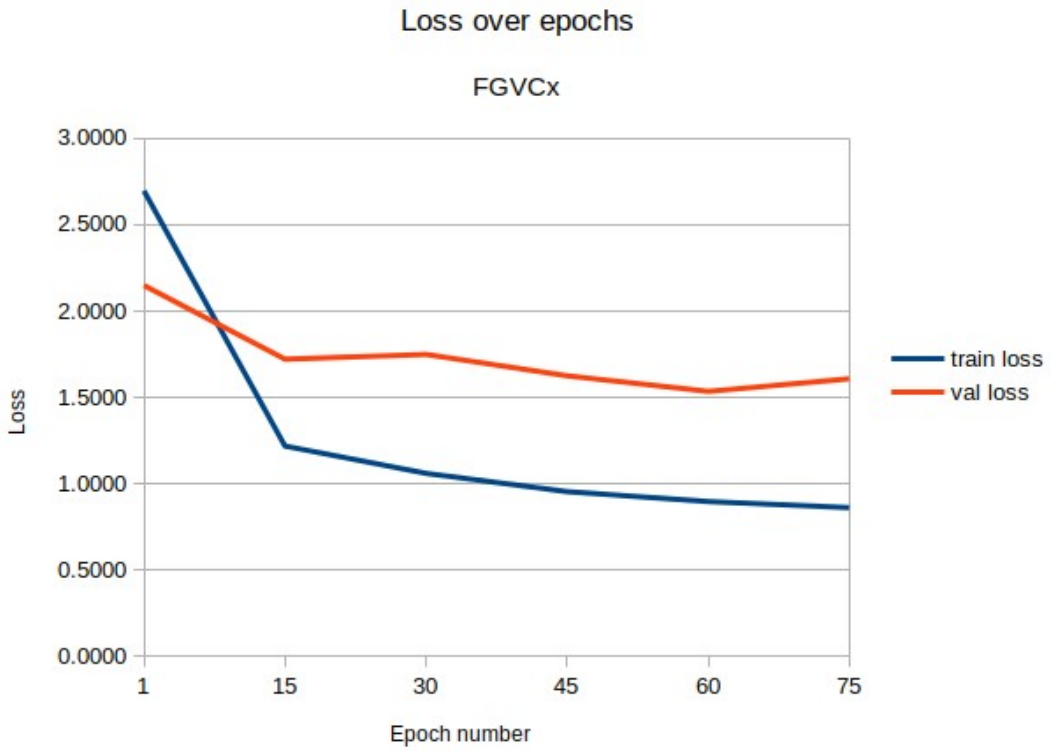

Figure 5.3: *Graph of loss while training on FGVCx. The training loss is consistently converging, but validation loss is oscillating around 1.6, which is acceptable if we compare this value with the maximum cross entropy loss for 50 classes (ln(50)=3.912.)*

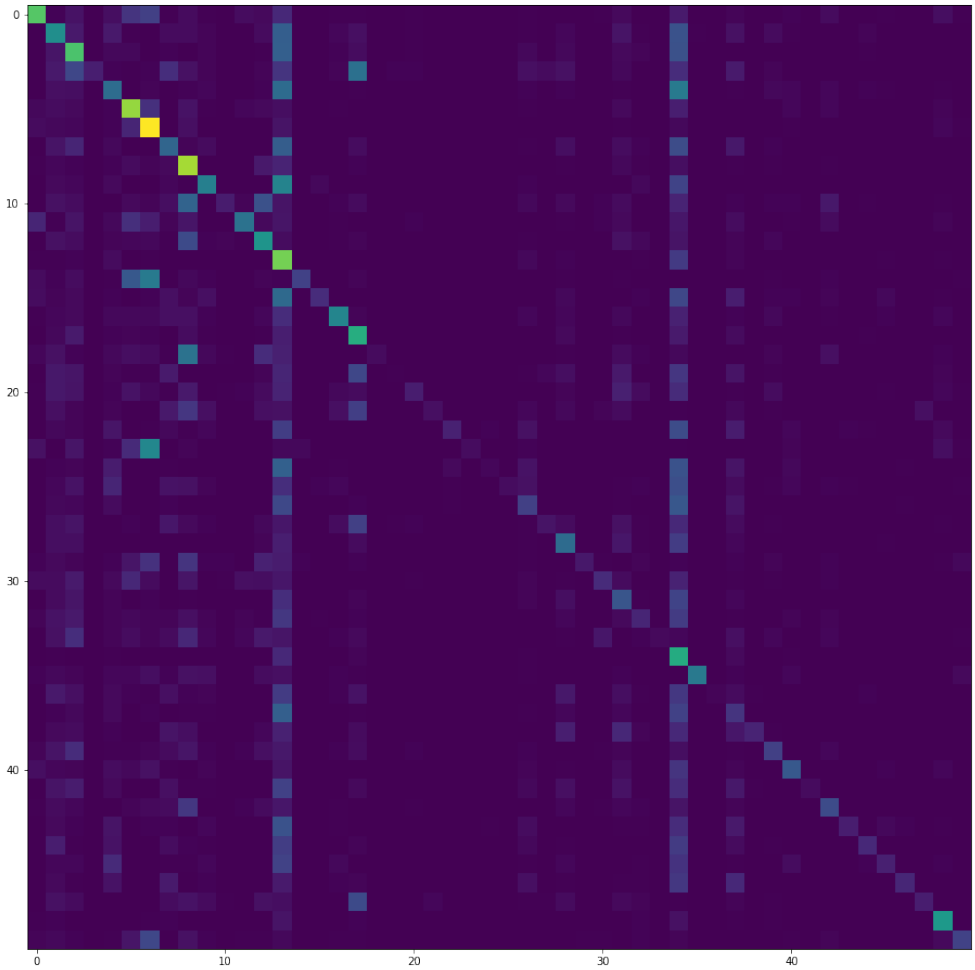

Figure 5.4: *Confusion matrix on the FGVCx train set. The numbers are not normalized per class size, but they show original numbers. Ideally, the diagonal should be strong, but many of the predictions are dispersed in other classes.*

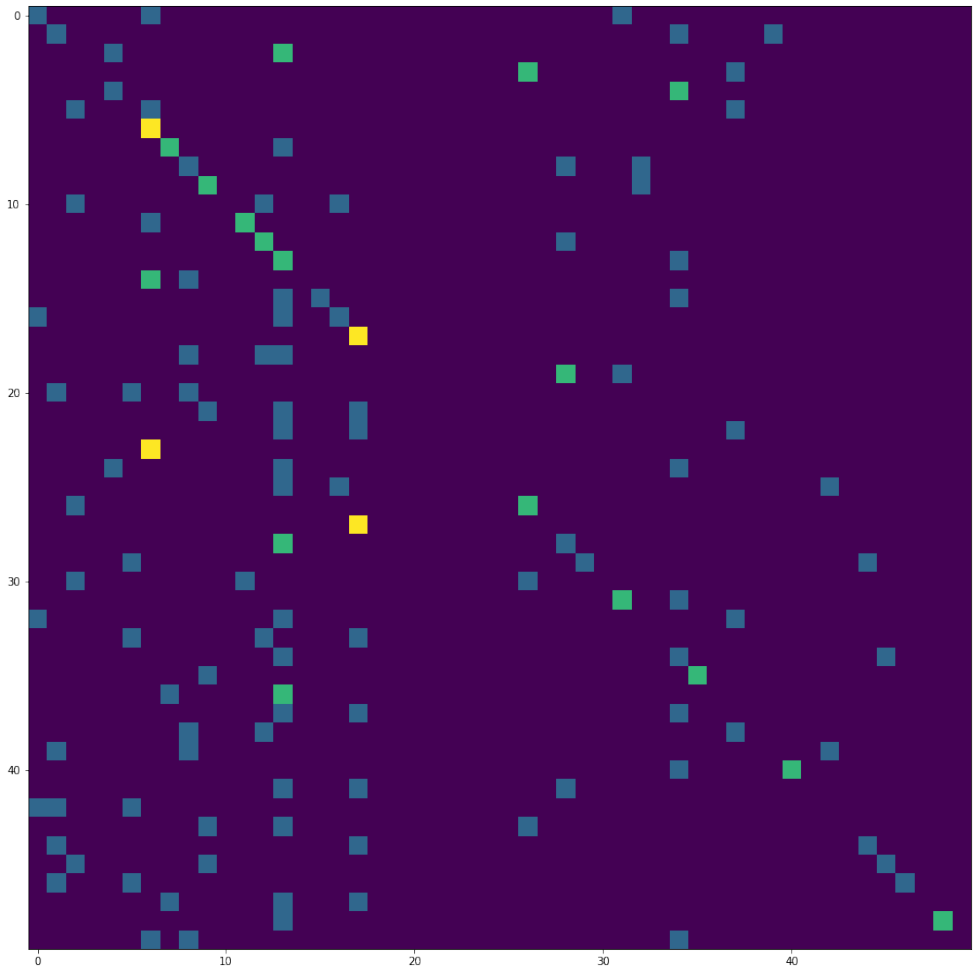

Figure 5.5: *Confusion matrix on the FGVCx val set. Although it looks similar to the figure 5.4, the size of validation is very small, which makes many squares zero.*

5.1.2 Pascal VOC 2007

Since the original paper this thesis is based on worked on Pascal VOC 2007 dataset, all of the experiments were repeated with it. Multi class learning was used, as opposed to multi label learning, which means out of many annotations, only one was used for each image in the training set. The first thing to notice is that the number of epochs is now much smaller (20 for VOC as opposed to 75 for FGVCx). Secondly, despite the smaller number of epochs, the results are much better, both quantitatively, and qualitatively, as it shall be seen in section 5.2. Here the classes are much more distinct, and taking more than the top three predictions doesn't really help the classification accuracy, as the gain is under 2%, which is shown by the figure 5.7, last row. It can also be seen that the network reaches some sort of a validation plateau after just a few epochs, while the training loss and accuracy continue their expected changes.

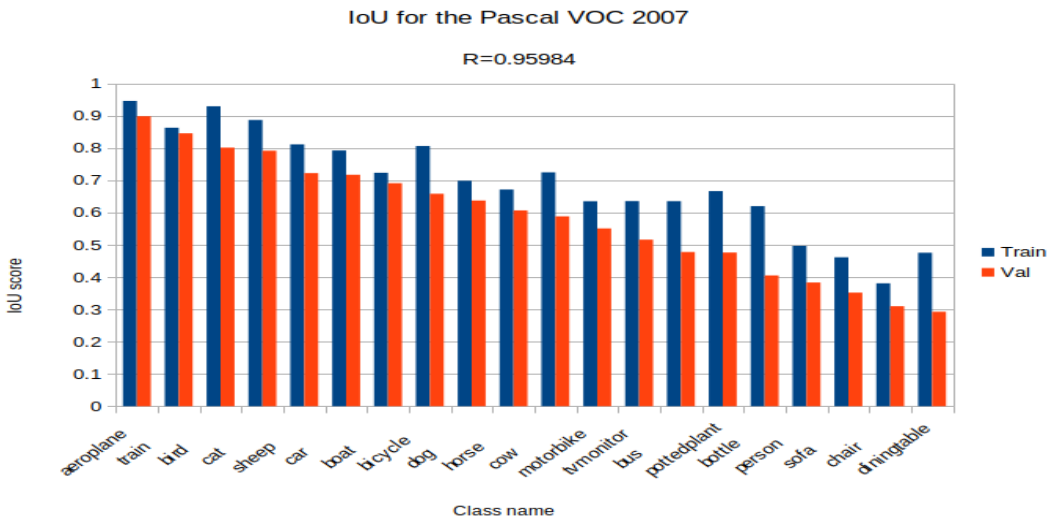

Figure 5.6: *Intersection-over-union metric on the Pascal VOC 2007 dataset generated from confusion matrix on the training and validation image set. Training IoU is consistently greater than validation IoU, but the two are very correlated.*

In this experiment the confusion matrices of the train and validation set, shown in figures 5.10 and 5.11 respectively, are very similar to each other. However, they both have noticeable first row and column, which correspond to the `person` class, and second-to-last column which corresponds to the `sofa` class, and also a very strong diagonal. This indicates the network doesn't overfit to the training set and retains its generalization capabilities, due to the distinction of the classes.

	Train				Val			
epoch	*loss*	*top1*	*top3*	*top5*	*loss*	*top1*	*top3*	*top5*
1	1.8927	47.22	70.33	81.57	1.2353	63.55	88.80	95.62
5	0.8000	74.61	94.32	97.80	0.8203	73.78	93.31	97.53
10	0.6490	78.81	96.16	99.00	0.8054	73.11	92.91	97.21
15	0.5472	83.33	97.36	99.40	0.8128	73.15	93.39	97.53
20	0.5023	83.09	97.84	99.68	0.7979	74.06	93.82	97.57

Figure 5.7: *Table of experimental data while training on Pascal VOC 2007. Here the results are much better in comparison with the FGVCx dataset*

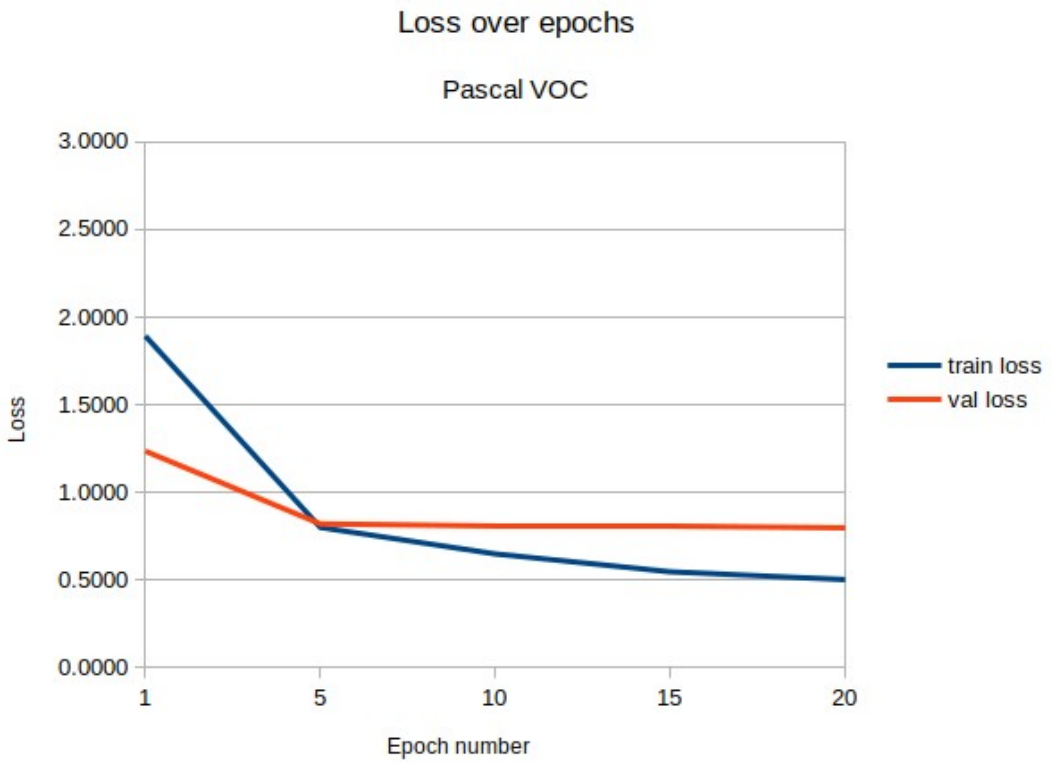

***Figure 5.8**: Graph of loss while training on Pascal VOC 2007*

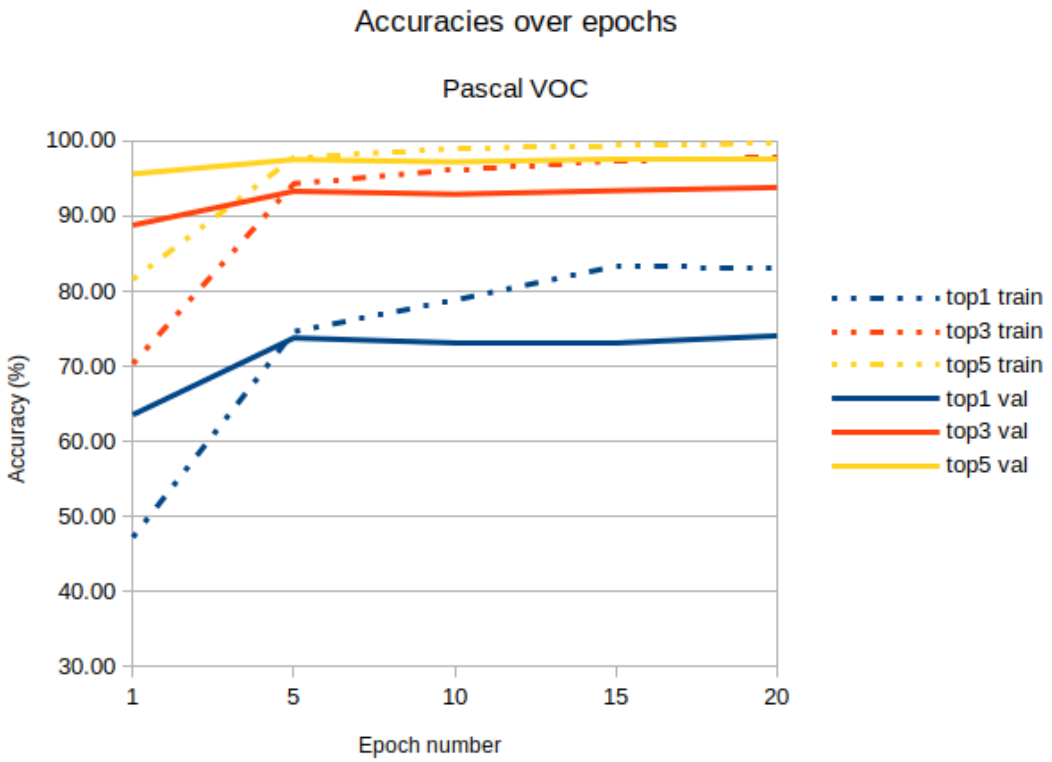

***Figure 5.9**: Graph of accuracies while training on Pascal VOC 2007. Validation accuracies reach a certain threshold after 5 epochs and don't increase, while training accuracies tend to still increase, although slow. Taking more than three top predictions does not really help the model as it's almost sure to guess the correct class in those three tries.*

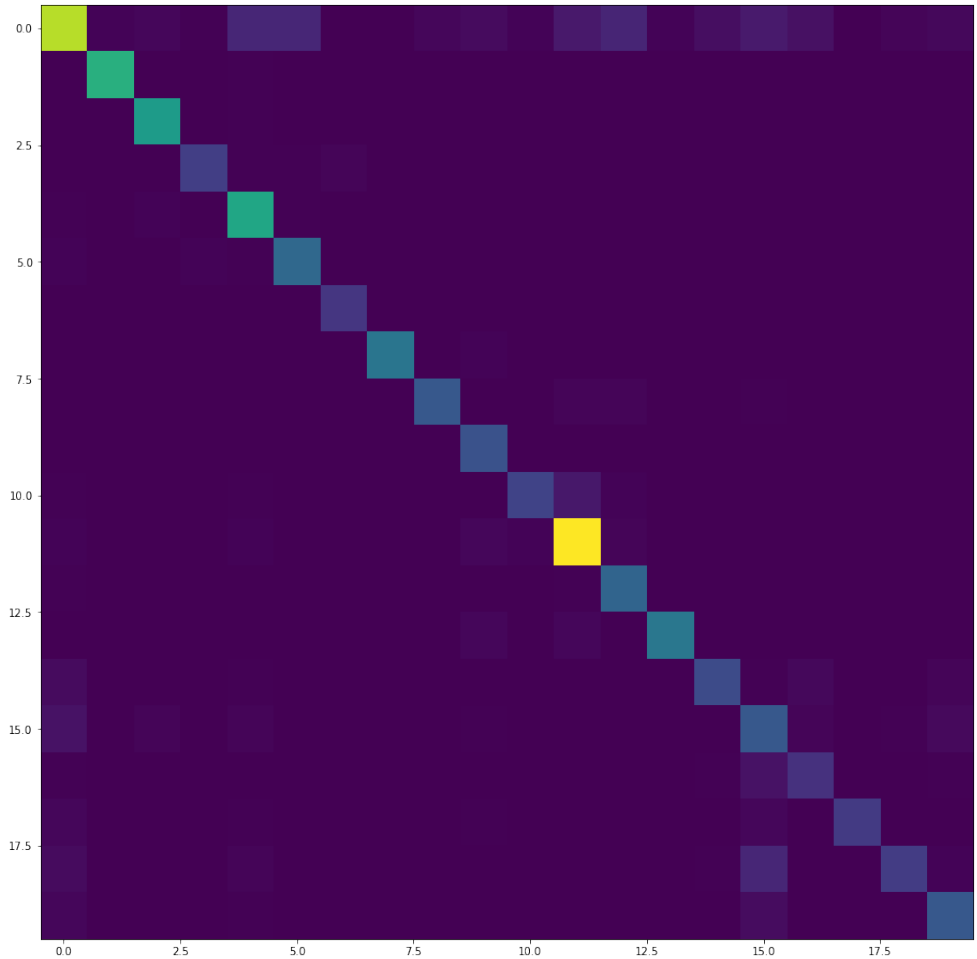

Figure 5.10: *Confusion matrix of the training set. Once again original numbers, and not normalized, are plotted. Unlike figure 5.4, here the diagonal is very strong.*

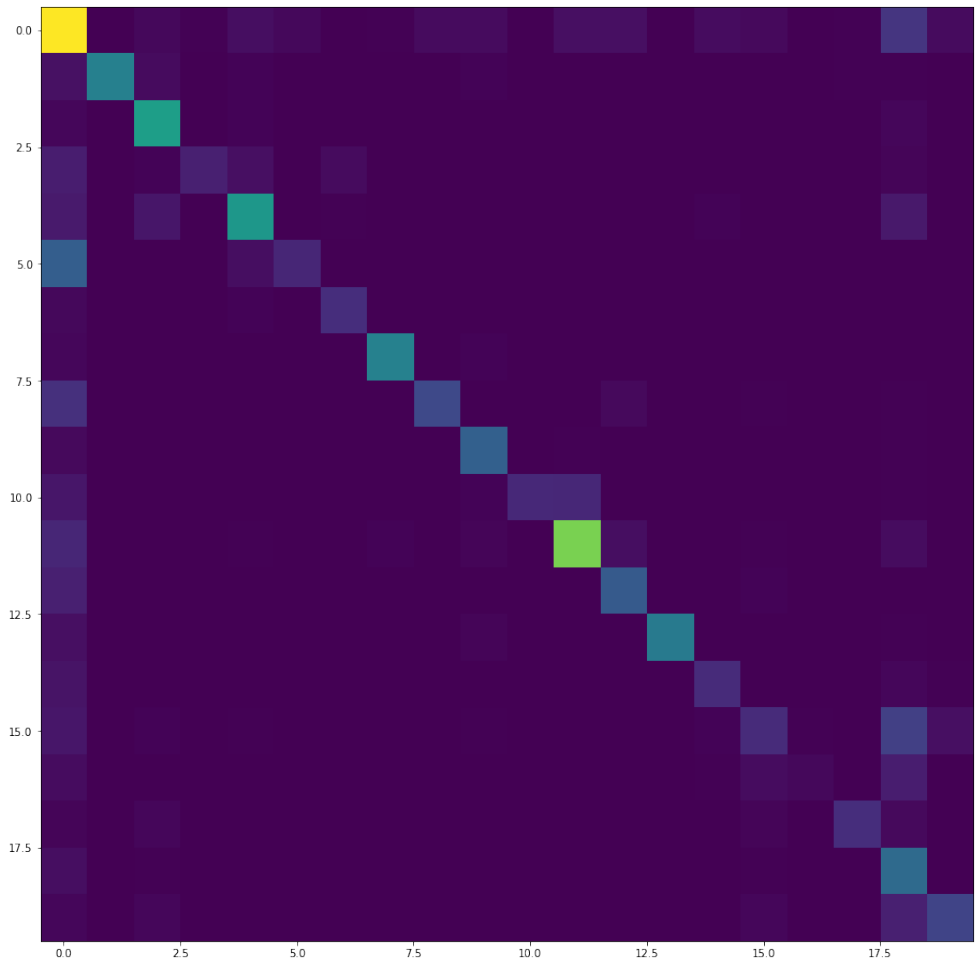

Figure 5.11: *Confusion matrix of the Pascal VOC val set. It looks similar to the figure 5.10, but more dispersed with a strong first column, the* `person` *class*

5.2 Activation maps

5.2.1 FGVCx

Class activation maps of the FGVCx dataset are shown in figure 5.12. It can be seen that the forms of the activation heat maps roughly approximate the shape and the position of the fungi. An interesting phenomenon can be spotted: some objects are shown to be surrounded by blue with red background, and some are the direct opposite. Since the heat map maps the small numbers (by relation, not magnitude) to the blue color, and greater numbers to the red color, it follows that for some classes the activation vector's weights are mostly negative. Nevertheless, the absolute magnitude approximately localizes the object regions. Also, the heat maps mostly have irregular and 'wavy', dispersed forms. This will be explained in section 5.2.2.

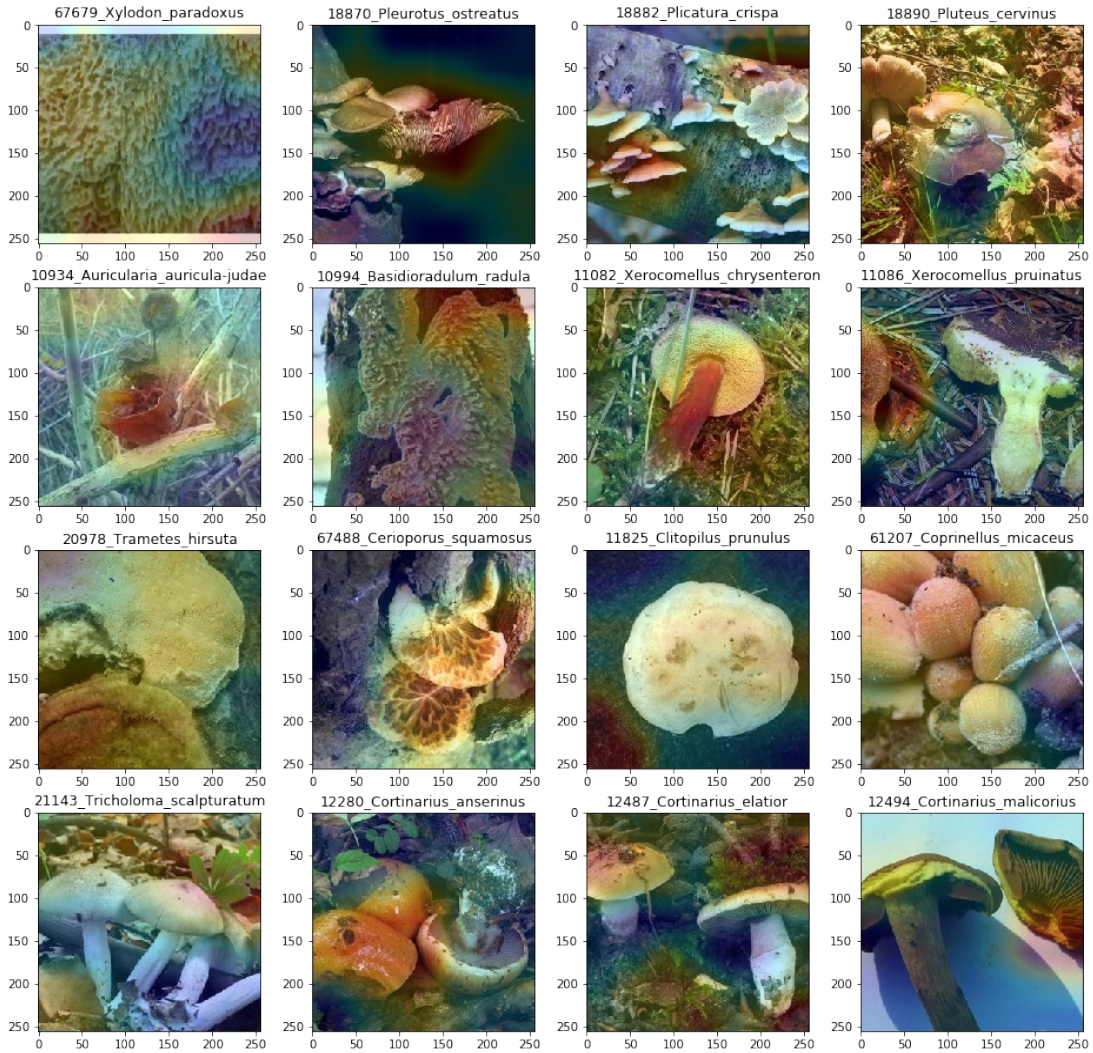

Figure 5.12: *Class activation maps for some images from FGVCx. They are mostly around the object which is annotated, yet are irregular and 'wavy'. Some objects are localized with the blue color (top left) and some with the red color (next one).*

5.2.2 Pascal VOC 2007

Class activation maps for Pascal VOC 2007, shown in figure 5.13, are much more regular and compact. This is also connected to distinction of the classes. Since Pascal VOC images are very distinct, a model can only mark a general position as important, since it's very confident in it's decision, resulting in condensed heat maps. This is different from FGVCx where a lot of classes are mutually similar and the model needs to specify what features disambiguate an input image, and where they are positioned. This is analogous to the humans specifying the features a poisonous fungus could have. These heat maps show once more the nondeterminism of activation vector's weights, as some images have the negative blue spectrum as a central region, and some have the positive red spectrum.

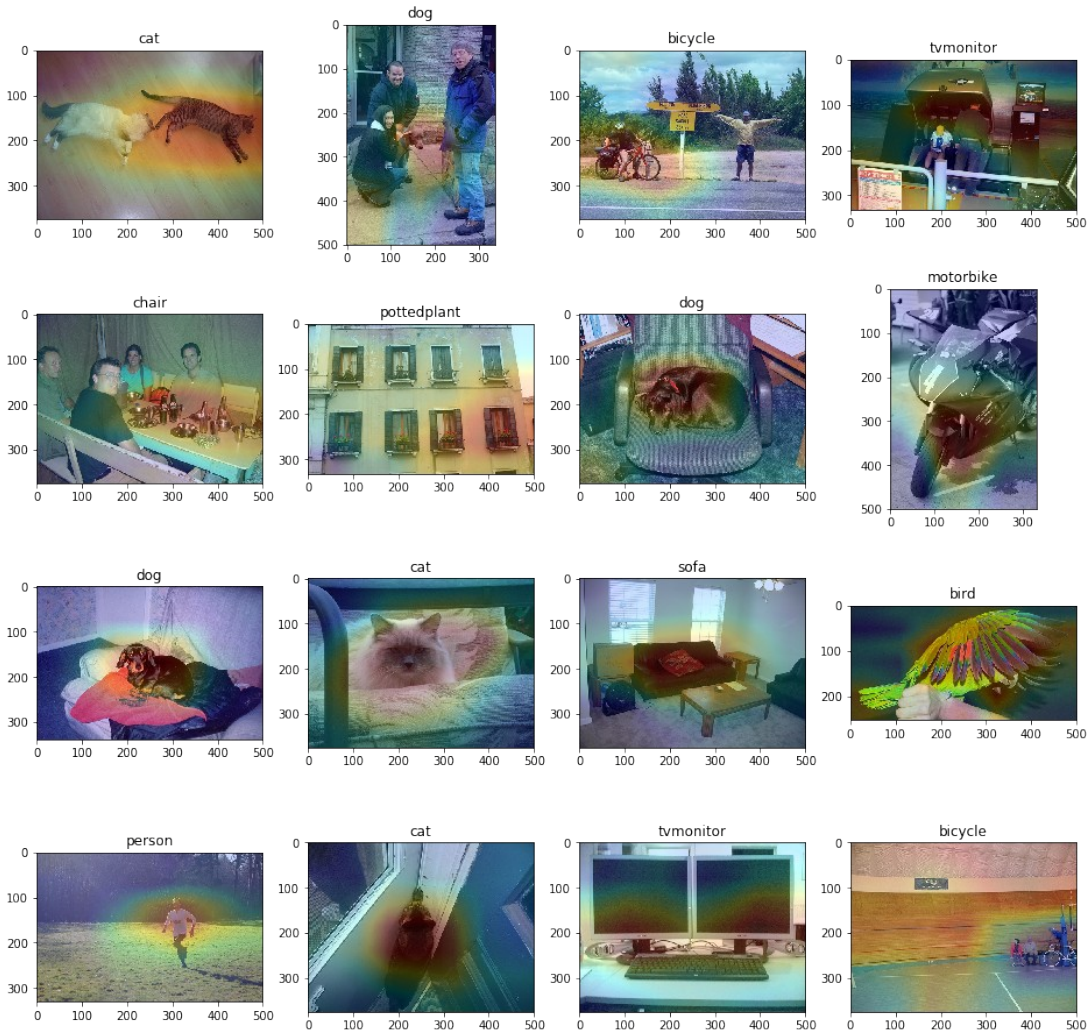

Figure 5.13: *Class activation maps for some images from Pascal VOC. The same phenomenon of nondeterministic image localization can be seen. Here the activation maps are much more compact and surround the object more precisely in comparison with activation maps of images from FGVCx.*

5.3 Lowest ranked false negatives and highest ranked false positives

The exit of a neural network classifier is the conditional probability of a class given the image. Lowest ranked false negative for a given class is the image labeled as that class, for which the model outputs probability that is lower than any probability outputted for the other images that are labeled as that very class. Likewise, the highest ranked false positive for a given class is the image that's not labeled as that class, but has the highest probability for that class among any other images that are also not in that class. These images are usually mislabeled in other datasets, or can be assigned a different class, but Pascal VOC doesn't have mislabeled images.

Lowest ranked false negatives for a subset of the Pascal VOC dataset can be seen in figure 5.14. A quick glance at the feature maps shows that they are mostly around the object they should recognize, which hint the

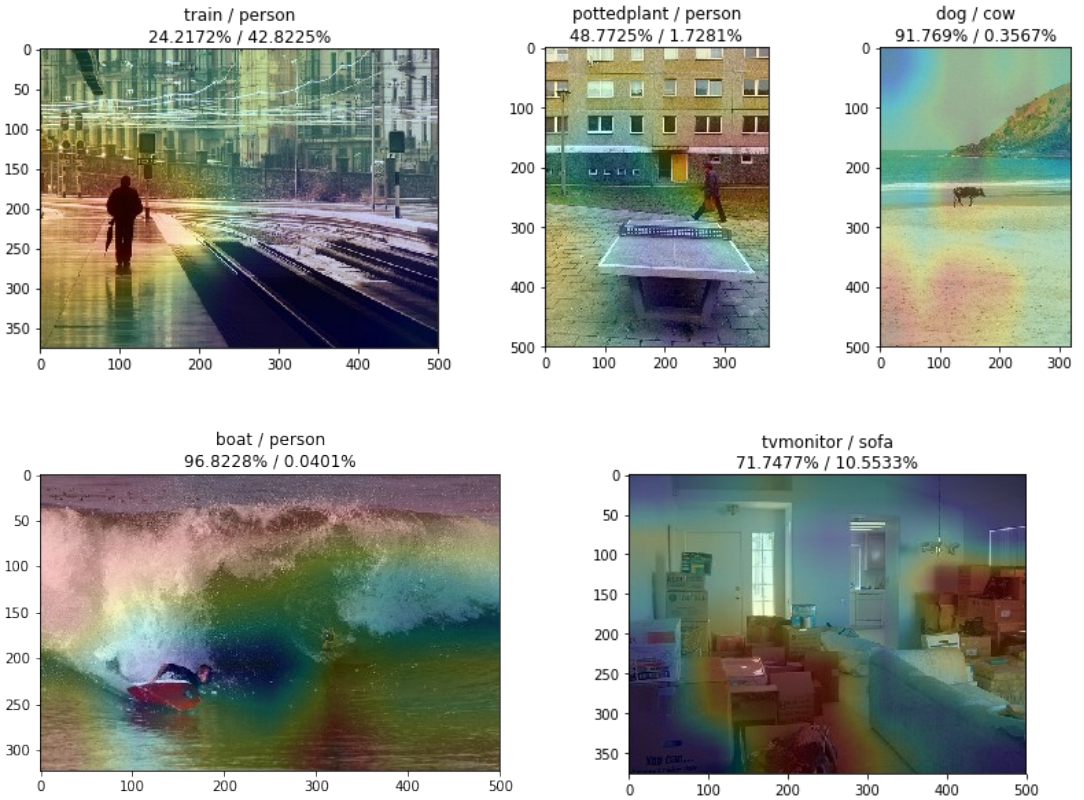

***Figure 5.14**: Lowest ranked false negatives of Pascal VOC. For every class an image was found for which the model outputs the lowest probability. Images are plotted with their activation maps, true class / predicted class respectively, and true class probability / predicted class probability respectively. Activation maps are of the class the model predicted, and not the true class. Only images that are annotated with only one class were considered for this visualization*

model's certainty in its decision. A more detailed analysis shows how some of the false classification could be considered as an honest mistake by a model, e.g. `cow` class that was annotated as a `dog`, due to images of dogs on the beach perhaps, or how the table tennis table looks like a flower pot from that angle. Other images show how some class predictions depend on the context in the image, such as the left two images for the `person` class that was labeled as a `train` or `boat`. It's easy to imagine a boat making those waves.

Highest ranked false positives, shown in figure 5.15, have similar characteristics as lowest ranked positives. The heat maps are also irregular but more scattered. Since the images are not labeled as the model predicts, localization is not very precise. Detailed observation once again shows us how the context influences model's decision, for example `tvmonitor` can really be placed in the blue region in the second image, and `train` does in fact resemble the shipyards. One could also notice that some images have smaller differences between the colors. This is due to low activations across the whole image as the model is insecure in his predictions. This can be interpreted that the trained model would successfully combat badly annotated data if they existed, and would generalize well, as he's fairly certain image is of some other class, and not the labeled one.

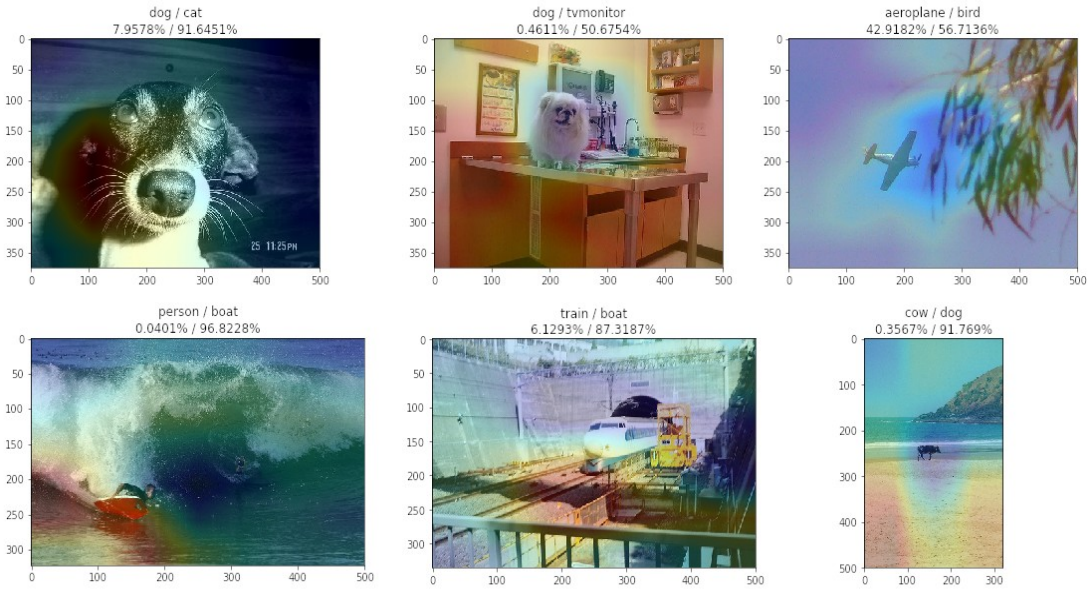

Figure 5.15: *Highest ranked false positives for a class, sorted by the predicted class. For every class only the images not labeled as that class were considered. Among them, the image with the highest model output for the selected class is shown. CAM's were plotted for the labeled class, and not for the predicted, to emphasize the model's insecurity of the real class. Many of the mispredictions can be attributed to the image context (aeroplane, boat) or image resolution (tvmonitor, cow).*

6 Conclusion

This thesis focused on convolutional neural networks as image classifiers. By doing a series of experiments on two different datasets, it explored what do the convolutional neural networks actually learn. The experimental results show that if the classes are distinct enough, learning the model will be quicker and easier, with lower loss, higher accuracy, and rounder, more compact activation maps. If the classes aren't distinct enough, though, the training will last much longer with less of a progress, and the activation maps become irregular and dispersed. The activation maps are usually near or around the object that's labeled in the image, opening up a usage for object localization. Dispersed activations usually concentrate around the features of an object that allow the model to infer the correct class, and not the whole object. Highest ranked false positives show that the model can sometimes correct a mislabeled picture as it generalizes well. Class activation maps show that the activations are closer to zero if the model is unsure about the resulting class, coloring a heat map more uniformly, as it cannot pinpoint what image features are decisive for the correct classification.

7 Literature and references

[1] Oquab M. Bottou L. Laptev I. Sivic J. *Is Object Localization for free: Weakly-supervised learning with convolutional neural networks*, pages 1-10.

[2] Zhou B. Khosla A. Lapedriza A. Oliva A. Torralba A. *Learning Deep Features for Discriminative Localization*, CVPR Las vegas 26.06.2016. pages 1-10.

[3] Jacobsen, J.H. Smeulders, A. Oyallon, E. *i-RevNet: Deep Invertible Networks.* International Conference on Learning Representations (ICLR), 2018. pages 1-11

[4] Santurkar S. Tsipras D. Ilyas A. Madry A. How does batch normalization help optimization? 29.05.2018. pages 7-9 https://arxiv.org/abs/1805.11604

[5] Veit A. Wilber M. Belongie S. Residual networks behave like ensembles of relatively shallow networks, 27.06.2016. pages 1-12 https://arxiv.org/pdf/1605.06431.pdf

[6] Anđelić A. Semantička segmentacija slika dubokim konvolucijskim modelima, bachelor's thesis, FER, June 2018.

[7] Ioffe S. Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, pages 1-11

[8] 2018 FGVCx Fungi Classification challenge, 22.03.2017. https://github.com/visipedia/fgvcx_fungi_comp.

- Accesed 04.03.2019.

[9] Sample code for the Class Activation Mapping, Bolei Zhou, 24.08.2017. https://github.com/metalbubble/CAM

- Accessed 12.05.2019.

[10] Why is so much memory needed for deep neural networks? Jamie Hanlon. https://www.graphcore.ai/posts/why-is-so-much-memory-needed-for-deep-neural-networks

- Accessed 09.06.2019.

[11] The PASCAL Visual Object Classes Challenge 2007, 07.04.2007. http://host.robots.ox.ac.uk/pascal/VOC/voc2007/index.html

- Accessed 30.05.2019.

[12] https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

8 Summary

This thesis focused on convolutional neural networks as image classifiers by doing a series of experiments on two different datasets. After a brief introduction to deep learning discipline, a short overview of the best practices was given. A pretrained model with residual connections, ResNet50, was selected, in combination with Adam optimizer. It was trained for classification in appropriate number of epochs. It was shown it could guess a correct fungi species, and it classifies general objects with 97% top three accuracy. Later, if was shown that over similarity of fungi species inhibits further learning of the network. This type of problems is not found in Pascal VOC dataset. While learning the classification of the objects, the network implicitly learned how to a) localize objects themselves and b) how to localize specific features that distinguish one class from another. It's been demonstrated with lowest rated positives and highest rated negatives of the Pascal VOC validation dataset that the network has problems with ambiguous classes but generalizes well and can correct mislabeled data.

Keywords: *convolutional neural network, deep learning, residual models, fungi, Pascal VOC, class activation maps, localization, classification*

Ovaj završni rad se fokusirao na konvolucijske neuronske mreže kao klasifikatore slika u nizu eksperimenata na dva različita skupa podataka. Nakon kratkog uvoda u područje dubokog učenja, dan je kratki pregled najboljih praksi. Odabran je predtrenirani rezidualni model ResNet50, u kombinaciji sa Adam optimizatorom. Model je treniran za klasifikaciju u odgovarajućem broju epoha. Rezultati pokazuju da model predviđa točnu vrstu gljive, te da može klasificirati općenite objekte sa preko 97%-postotnom točnosti u najbolja tri pokušaja. Kasnije, dokazano je da prevelika sličnost različitih vrsta gljiva usporava daljnje učenje i porast točnosti. Ovaj tip problema nije bio primjećen u Pascal VOC skupu podataka. Tijekom učenja klasifikacije objekata, mreža je implicitno naučila i a) lokalizirati same objekte na slici i b) lokalizirati posebne značajke koje razlikuju jedan razred od drugog. Demonstracija najlošije rangiranih pozitiva i najbolje rangiranih negativa pokazala je da mreža ima problema sa višeznačnim razredima, no generalizira dobro i može ispraviti pogrešno označene podatke.

Ključne riječi: *konvolucijska neuronska mreža, duboko učenje, rezidualni modeli, gljive, Pascal VOC, mapa razrednih aktivacija, lokalizacija, klasifikacija*