Chapter 4

Approach

- Prerequisites:

  Torch7: the scientific computing framework I used for working with Neural Networks.

  EC2 Amazon Web Services (AWS) for training the Neural Network on a graphic card.

  The CIFAR-10 dataset to feed the network with training material.

- Description of the own approach

4.0.6 Prerequisites

Before i describe this thesis approach for the topic of this thesis a few words on the language i used for implementing the Neural Network. I used Torch7, because it is an code efficient way to implement Neural Networks and it is easily readable. In addition to that there is a whole community around Torch7 which has a solution for most of the problems you will encounter when developing Neural Networks.
What is Torch7?
Torch7 is a scientific computing framework which supports machine learning in a very efficient way as well most of its functionality is runnable on GPUs. It uses LuaJIT as an scripting language and for GPU support is has an underlying C/CUDA implementation. Since LuaJIT is an Lua extension you have the advantage of a fast and easy to learn scripting language.

I used an EC2 AWS machine to train the neural network which i can highly recommend. Since Torch7 is modularly built out of luarocks (`https://luarocks.org/`) packages it tends to break apart when you installed the wrong ones and messes up your libraries irreversible. Revert an AWS machine is done quickly, but to reinstall Torch7 needs much more time, especially the CUDA part. Another great advantage is, there are some EC2 AWS machines where Torch7 is preinstalled[1].

The image recognition part of this approach is trained on the CIFAR-10[Kri12] dataset. This dataset contains 60000 32×32 pixel width colored images divided in 10 classes, with 6000 images per class. From the 60000 images 50000 are training images and 10000 are test images. The test batch contains exactly 1000 randomly-selected images from each class which are not part of the trainings images. So the network cannot simply learn by heart it has to generalize the classes to be able to tell them apart.

[1] for example: AMI Id: ami-b36981d8. (Can be launched using g2.2xlarge instance)

4.1 Description of the Own Approach

The approach consists of two parts.
Object detection and object localization/tracking. In the first part, the detection part, we will see how to use a CNN with supervised learning to train a Neural Network to recognize different classes of images from the CIFAR-10 dataset or other datasets. This is not a new concept, but there are some minor differences between this and other approaches. In the second part, we will see this thesis approach to track objects with a deconvolution Neural Network without any further training. There is no training involved, because it is more something like an information extraction task and in this case the network won't getting better through training at something. It could, but this would than be a whole other project, which involves ground truth data.

4.1.1 Object Detection

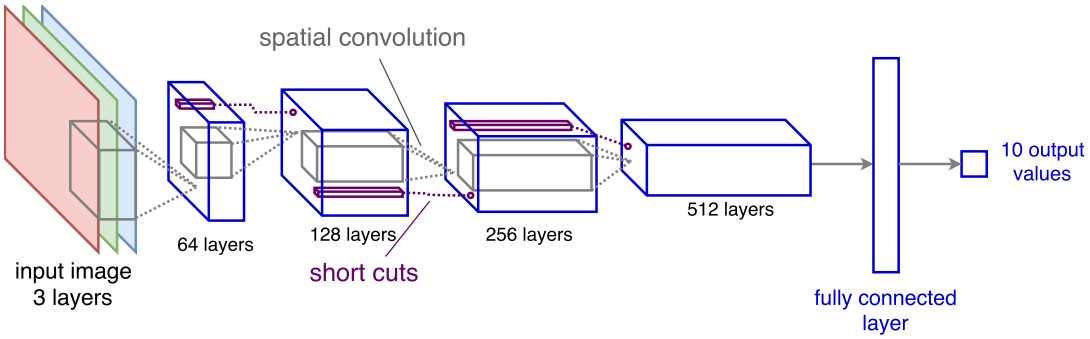


Figure 4.1: residual network of the thesis approach

In Torch7 you can easily stack different kinds of Neural Network layers, which will later be processed after another in a specific way determined by the order of how they are stacked. Most of the time we will just use a linear approach for stacking the layers, but you are also able to parallelize some layers. More on this later in this chapter. There are 4 kinds of layers which I used a lot. The convolutional layers, the batch normalization layers, the RELU layers and the pooling layers, which are all explained in chapter [3]. Here I will now explain how they fit together and will not go into detail on how they are built internally.
I used basically a residual network [HZRS15] for the object detection. The network has multiple convolutional layers with a mini-batch size from 3 to 512. The number of convolutional layers is implemented dynamically so that the user can choose how deep the network should be. But it always perform four special convolutions. It starts with a 3 to 64 mini-batch size convolution and after a dynamically number of regular 64 to 64 mini-batch size convolutions it performs a 64 to 128 mini-batch size convolutions. This is followed by again a dynamically number of 128 to 128 mini-batch size convolutions. After that follows a 128 to 256 convolution and then a dynamically number of 256 to 256 convolutional layers. At the end there is a 256 to 512 mini-batch size convolution and after that comes the fully connected part which i will cover later in this chapter. Every convolution has a filter size of 3x3, a step size of 1 and a padding of 1 with a few exceptions, see chapter [5.2].
Every convolution is followed by a batch normalization to optimize various aspects of the network. There is after every second convolutional layers a RELU layer as a

transfer function layer, which is an integral part of every Neural Network. There are two pooling layers. One at the beginning after the first convolution and one right before the fully connected part. This helps to compress the image without loosing too much key features of the image for the object detection. This helps increasing the training speed and decreasing the occupied graphics card memory a lot. It also lets the network focus more on main features, because the max-pooling operation preserves the activation values and the arrangement of the strongest represented features in the input image.

The special property of the residual network is the separation of the network data

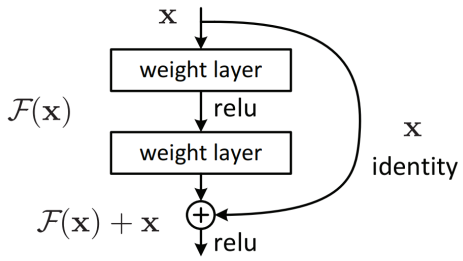


Figure 4.2: Residual learning: a building block, from [HZRS15]

forward- and backward propagation flow by adding some shortcuts which skips certain layers and are parallel to the original forward- and backward propagation flow. This helps to speed up the training effects in the front layers of the network, because by this shortcuts its like the network can train all layers simultaneously. U can see that every second layer has a short cut to the layer after the next one. In fact every layer after a ReLU layer has such a short cut. The split off and remerge function work as follows:
The split off is as simple as just duplicating the output of the last layer. Every branch holds the same values as the other branch. One branch will then flow through the next two convolutional layers as explained above the other will skip these two layers and will then remerge with the other layer. The remerge will then perform an element-wise addition on each element of the two Tensors. There is one exception two this principle. As you can see in figure 4.3 there are 3 dotted lines. This lines marks the cases, where the normal path of the network (the path which does not skip layers) changes the mini-batch size from 64 to 128, from 128 to 256 and from 256 to 512. In this cases the path which skips layers has to change its mini-batch size, too. So it performs a convolution like in the normal path where it increases the mini-batch size, but with a filter size of 1×1, so that there is practically no convolution in the spatial dimension, like in the other cases where nothing happens to the path of the short cut (which is called identity function in torch7).
The fully connected part of the Network:
After all the convolutional layers follows one fully connected layer (fc-layer) at the end of the Network, so that the network can now relate all pixels of an image on all mini-batch layers to each other and score for the recognition of the 10 object classes. The convolutional layers just relate pixels to each other which lay in the same pixel region with the size of the convolutional filter. The fully connected layer is in this sense like a convolutional filter with a filter size of the whole image. In fact you can transform a fully connected layer into a convolutional filter by increasing the filter size of the regular convolutional layer to the image size and you get a filter which performs the same operation as the fully connected layer.

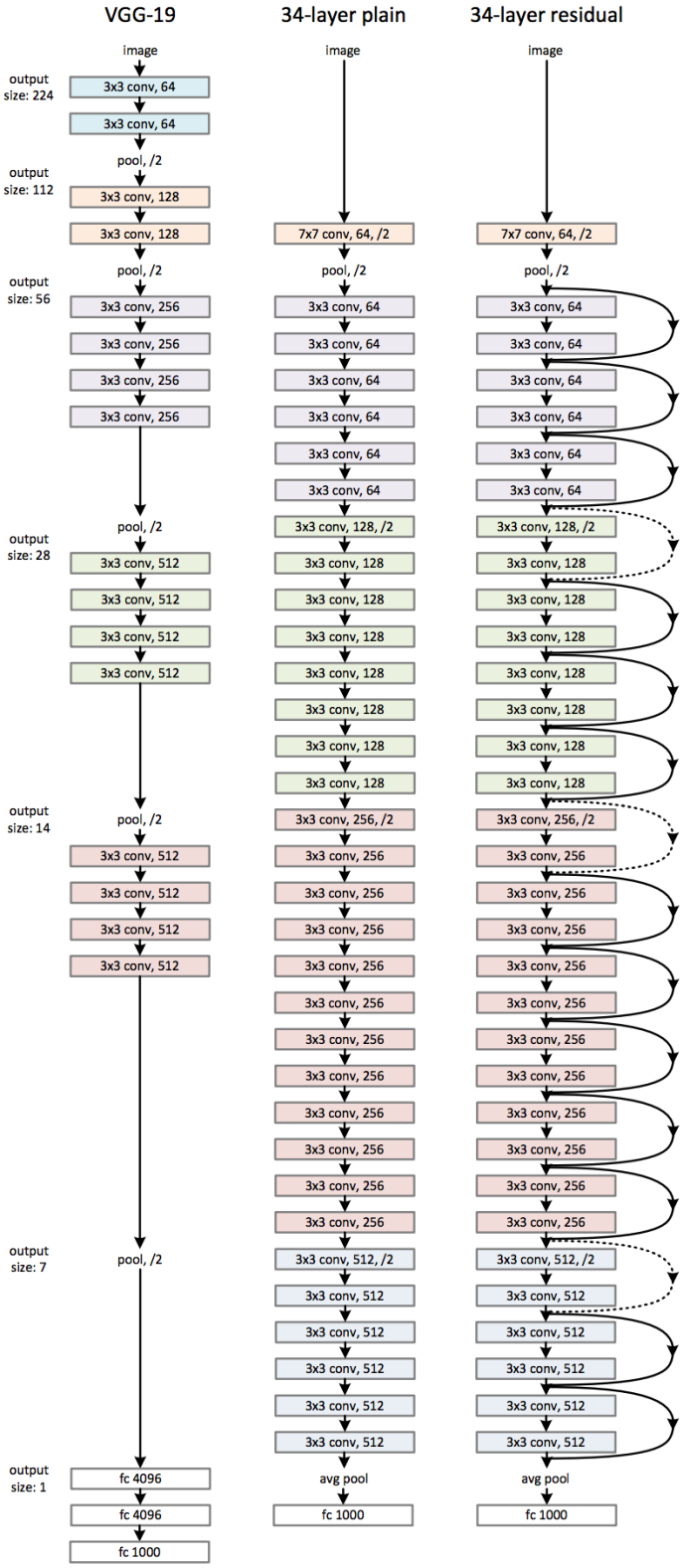


Figure 4.3: ResNet, from [HZRS15]

"Figure 3. Example network architectures for ImageNet. Left: the VGG-19 model [41] (19.6 billion FLOPS) as a reference. Middle: a plain network with 34 parameter layers (3.6 billion FLOPS). Right: a residual network with 34 parameter layers (3.6 billion FLOPS). The dotted shortcuts increase" depth dimensions. [from the same paper]

This transformation is useful because than you can revert this convolutional operation for the deconvolutional part of the network. Here the fully connected layer if used or a transformed convolutional layer would have 10 output values for the scoring for the recognition of the 10 object classes. But due to a nearly complete information loss by reverting a convolution which has over 10000 input values (from the 512 mini-batch layers and the processed image size) and just 10 output values, this thesis approach is a little bit different.
The trick is to perform a 1×1 filter size convolution which shrinks down the 512 mini-batch layers to the required 10 mini-batch layers for the object classification. Though we have 10 layers of image sized matrices of information about the recognized classes and not 10 simple and clear values, but each layer is completely separated of each other with regards to the recognized classes. This means, that no layer contains values for another layer and its associated object class. By simply summing up all values in one layer we can obtain the searched value for the layer's object class recognition score. The advantage by this approach is, that we can later easily revert the 1×1 convolution for the deconvolution process. Though the sum up process is impossible to revert, but we don't have to as we will see in the object tracking part (chapter [4.1.2]). After this follows a softmax layer.
After knowing how a specific Neural Network is build the next part to get to know is how the network learns to classify. This network uses SGD for training. We saw how SGD is working in the background chapter (chapter [4]) and in Torch7 it is very easy to use. You don't have to program SGD on your own to use it with Torch7 properly.

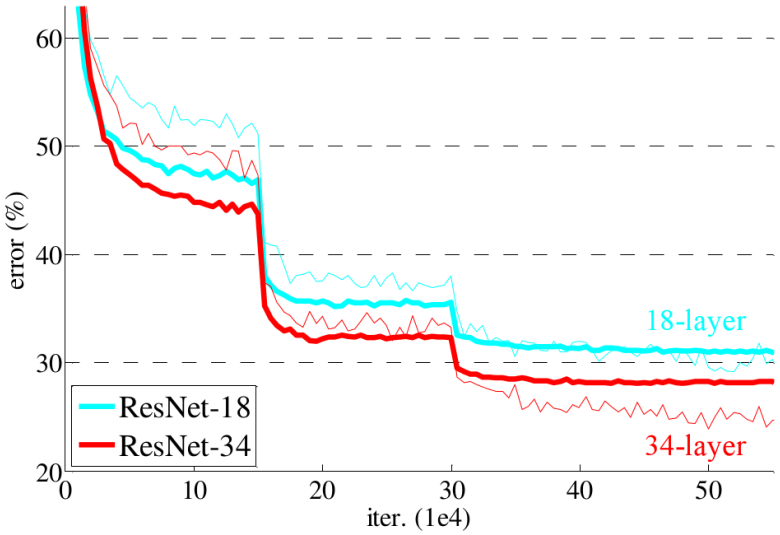


Figure 4.5: ResNet training curves (18 and 34 layers), from [HZRS15, Page 5, Fig. 4.]

But there is also this "plateau effect" problem. Did you ever wonder why most of the Neural Networks loss/epoch graphs possess this staircase-esque shape? On figure 4.5 you see a typical loss/epoch graph of a Neural Network. In this case of the original ResNet. This won't happen naturally or by the network itself. When I saw this the first time I thought that the network experienced some kind of deeper understanding in those epoch of training where it performs those jumps in the loss value. It has something human to it, to struggle for a long time learning something new and then suddenly one gets behind a concept or a subject and achieves a greater understanding. But maybe

the whole learning progress happens over a longer time and we just think that we crack the nut in the last seconds of that progress. And this is more or less what happens in the Neural Networks. Those plateaus come into being by adjusting the learning rate. Nothing magical happens here. We just decrease the learning rate so that the network learns more precisely, but this also slows down the learning [2] process. This may sound a little confusing that something is able to learn faster but more imprecisely. This effect is comparable to skimming over a text compared to reading in detail. You may find some obviously information faster, but as more riddled the information are as more futile faster reading becomes. To avoid this effect we adjust the learning rate when the loss rate plateaus. We could also readjust the learning rate constantly over time to avoid this staircase look of the loss rate, but this needs a very precisely calculation, because after a while the network could get stuck with the loss rate and didn't learn anything at all. I kept it save and choose the plateau effect method, some even adjust the learning rate by hand while training to minimize unexpected events. I got a bit off topic here, but it was important to me to show how easy someone can get misconceptions about what's happens inside a Neural Network, because they tend surprise with unempirical and remarkable effects which are often hard to reproduce, although they are based on combined but simple and consistent algorithms. Because there are actually a lot of unwanted effects which can occur in Neural Networks, I addressed a whole subchapter about that in this thesis (chapter [3.2]).

4.1.2 Object Localization

The idea behind deconvolutional based object tracking is, that a CNN holds information about the spatial position of certain features on an image. The features belonging to one certain object are scattered across the mini-batch layers of the extracted feature tensors of the different convolutional layers of the network. To extract the position of them we have to revert the CNN so that we can upsample and undo the convolutions of these features, so that we get the best possible heat map of the tracked object. In case of a ResNet there is the problem of the nonlinear layers shortcuts, that skip every second layer. My first approach was to built a deconvolutional network, that also contains layer shortcuts and that exactly revert the CNN process. To calculate one reversible convolutional layer we have to find a way to estimate the input of a layer by its output. I_x, I_y are the spatial dimensions of the input tensor and s is the total number of mini-batch layers:

$$\mathbf{T}_{\mathbf{Z}}^{(m,h,g)} = (\mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)} \circledast \mathbf{T}_{\mathbf{\Theta}}^{(l_2,l_1,v_1,u_1)}) \circledast \mathbf{T}_{\mathbf{\Theta}}^{(m,l_2,v_2,u_2)} + \mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)} \tag{4.1}$$
$$, \forall\ \mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)}, \mathbf{T}_{\mathbf{Z}}^{(m,h,g)} \in \mathbb{R}^{s \times I_y \times I_x}$$

[2] It don't slows down the learning process directly. It slows down the correction rate in the gradients due to the backpropagation. With a greater correction rate the steps in which the changes are made become also greater which may result in stepping over the aimed value for a better loss score.

If we reduce the problem to the essential operations we can transform the equation like this:

$$\begin{aligned}
&\text{unknown} : a,\ b && \text{known} : g(),\ \tilde{g}(),\ f(),\ \tilde{f}(),\ c, \\
&c = g(f(a)) + a\ , && b = g(f(a))\ , \quad a = \tilde{g}(\tilde{f}(b)). && (4.2)\\
&\Rightarrow a = c - b \Leftrightarrow b = c - a && && (4.3)\\
&\Rightarrow a = \tilde{g}(\tilde{f}(c - a)) && && (4.4)\\
&\Rightarrow a = \tilde{g}(\tilde{f}(c - \tilde{g}(\tilde{f}(c - \tilde{g}(\tilde{f}(c - \tilde{g}(...))))))) \lightning && && (4.5)
\end{aligned}$$

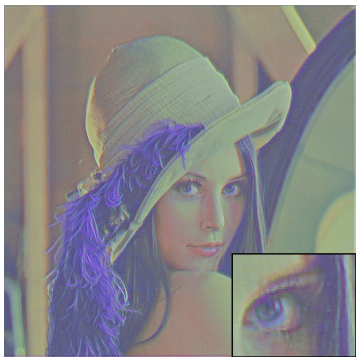
The problem is that an addition isn't a reversible operation with two variables to resolve and one known. However, since the two unknown variable are dependent on each other you can use a approximation method to resolve a.
In opposition to that the backpropagation can be calculated, because the derivation of an element wise summation of the remerging paths simplifies the equation, so that every path flowing into the summation operation gets the same gradient to pass back trough.
To find a solution for this problem I wanted to test the following conjecture: Is it possible to transform the network with the same trained weights to a linear and reversible network, without a greater loss in the classification score. A ResNet without the shortcuts is basically the same network as before, but needs more time to train its front layer weights. The main point of this transformation is, that all weights should be close to the originally trained weights of the ResNet for the same score on the same dataset, so that we can train a network with the speed of the ResNet and then use it as a linearized ResNet for the deconvolution. But why do the weights not need a bigger update after the linearizion of the network without a significant loss in the object classification score?

$$\begin{aligned}
4.1 :\ \mathbf{T}_{\mathbf{Z}}^{(m,h,g)} &= (\mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)} \circledast \mathbf{T}_{\mathbf{\Theta}}^{(l_2,l_1,v_1,u_1)}) \circledast \mathbf{T}_{\mathbf{\Theta}}^{(m,l_2,v_2,u_2)} + \mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)} \\
&\approx 2 \cdot \left((\mathbf{T}_{\mathbf{A'}}^{(l_1,y,x)} \circledast \sqrt{\tfrac{1}{2}} \cdot \mathbf{T}_{\mathbf{\Theta}}^{(l_2,l_1,v_1,u_1)}) \circledast \sqrt{\tfrac{1}{2}} \cdot \mathbf{T}_{\mathbf{\Theta}}^{(m,l_2,v_2,u_2)} \right) \qquad (4.6)
\end{aligned}$$

This approximation does only apply to the forwardpropagation of the network, and may look from a mathematically point of view strange. To weaken the weights in the kernel matrices of the convolution is like laying over the original tensor over the convoluted tensor, because a weaker convolution does less to the original tensor. I tested this approximation practically in torch7 and got very close results (± 2 % relative deviation) in the values of the output tensor, which is counterbalanceable after one epoch of training. This linearizion method has never been tested before as far as I know and may not apply to more complex branched networks.
To minimize the loss of this linearizion, the following optimization can be applied. Instead of transforming all layers with nonlinear connection between two epochs of training, we can transform the latest layer in the network, which has nonlinear connections, and do then some training before transforming the next latest nonlinear layer. If we proceed like that until all layers are linear we decrease the total loss of this transformation, because every change in the front layers of the network increases the deviation exponentially through every layer which forwards the deviation after that. So if we cut loose the shortcuts after a zipper opening principle from back to front, we stress the last layers less with growing changes in previous layers. However I did the transformation in one step because the loss was small enough to be neglected after one epoch of training.
The next step after the linearizion is to revert the network to form a Deconvolutional

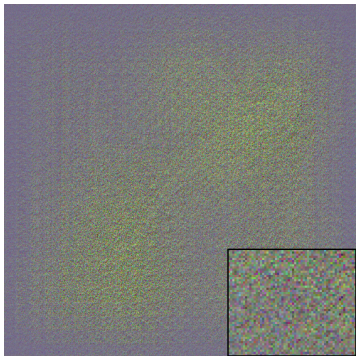
Neural Network. We do this by inverting every convolutional layer and rearrange them in reverse order from back to beginning. The first layer to be inverted is the fully connected layer. We already spitted and transformed the fc-layer to two convolutional layers, to make the inversion possible. The last convolution can be let untouched, because we just care for sending the input of this layer back to the network. The first of those two convolutions has to be inverted. This is been done, by transposing the kernel matrix (tensor) and adopting the padding, like we saw in chapter [3.1.4]. The reason why not replace the fc-layer with only one equivalent convolution, that does the whole fc-layer operation in one step, is because of the information compression which is not irreversible.

(a) Lenna (input image)
image size 3×512×512

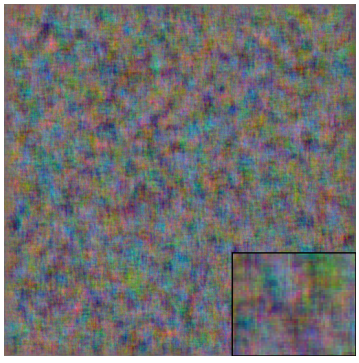


(b) fc in two steps:
kernel size (1) = 1×1
kernel size (2) = 512×512;
stride = 1; padding = 1;
mini-batch transformations:
$64 \overset{(1)}{\rightarrow} 10\ (\overset{(2)}{\rightarrow} 10)\ \overset{(1)}{\curvearrowright} 64$
10×512×512 bottleneck

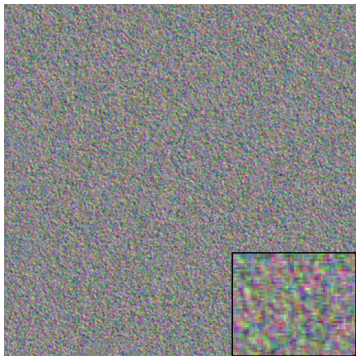
(c) example with typical CNN layer kernel size:
kernel size (1) = 7×7
kernel size (2) = 505×505;
stride = 1; padding = 1;
mini-batch transformations:
$64 \overset{(1)}{\rightarrow} 10\ (\overset{(2)}{\rightarrow} 10)\ \overset{(1)}{\curvearrowright} 64$
10×505×505 bottleneck



(d) example with 20%
kernel size (1) = 100×100
kernel size (2) = 21×21;
stride (1) = 20, (2) = 1;
padding = 1;
mini-batch transformations:
$64 \overset{(1)}{\rightarrow} 10\ (\overset{(2)}{\rightarrow} 10)\ \overset{(1)}{\curvearrowright} 64$
10×21×21 bottleneck



(e) example with near 100%
kernel size (1) = 502×502
kernel size (2) = 10×10;
stride = 1; padding = 1;
mini-batch transformations:
$64 \overset{(1)}{\rightarrow} 10\ (\overset{(2)}{\rightarrow} 10)\ \overset{(1)}{\curvearrowright} 64$
10×10×10 bottleneck



(f) fc in one step:
kernel size = 512×512;
stride = 1; padding = 1;
mini-batch transformations:
$64 \rightarrow 10 \curvearrowright 64$
10×1×1 bottleneck

Figure 4.6: image decompression pattern through deconvolution

The images in figure 4.6 show untrained convolution and deconvolution operations with a surrounding 3→64 and 64→3 mini-batch layer depth uplifting to simulate a typical fc-layer situation. in figure 4.6f we can see, that the information loss due to the

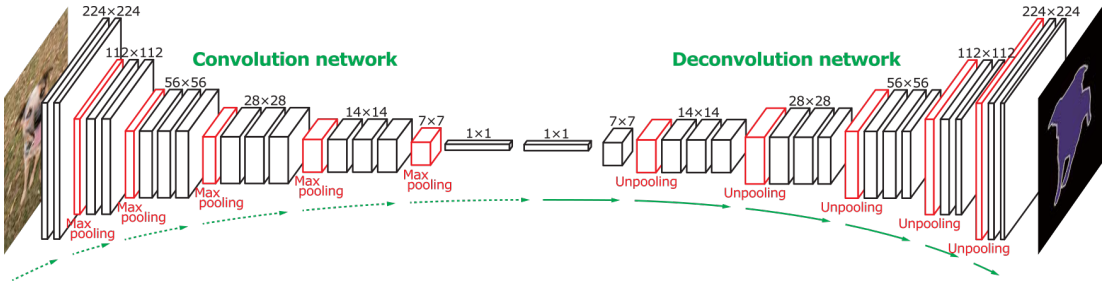
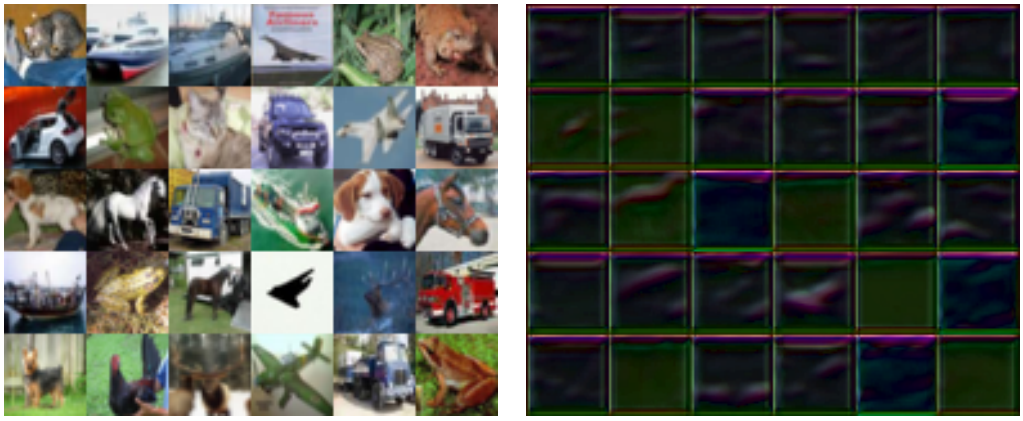


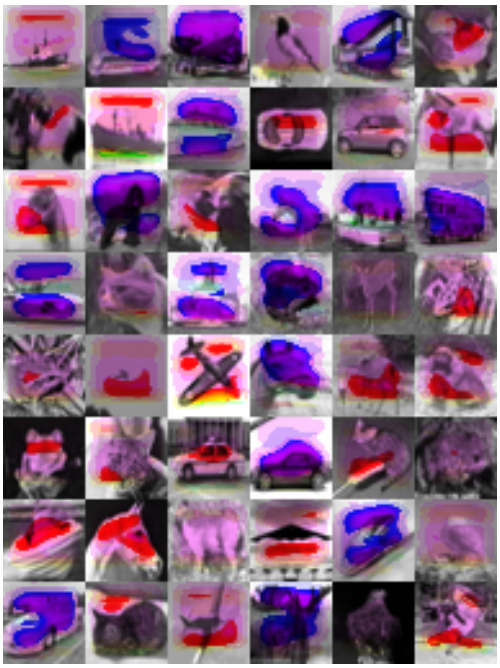
Figure 4.7: convolutional network with appended deconvolutional network,
from [NHH15, Page 3, Fig. 2.]



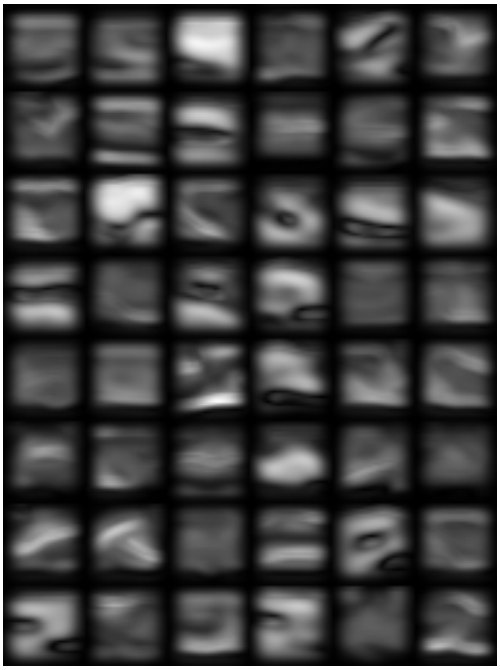
(a) first 30 images of CIFAR-10 dataset (b) processed CIFAR-10 images

Figure 4.9: object localization with option 1
(all layers to zero except for layer with highest object detection)

convolution and deconvolution process via a 10 value large bottleneck is 100%. In figure 4.6b we see the reassembled image after the backtransformation of the two step convolution. I choose the two step method for this thesis approach. The other images in figure 4.6 show reassembled images of interstages between the two methods. They show, as smaller the bottleneck is as less data is restoreable. So we should do as least compression as possible until we reach the final 10 mini-batch layers of the convolution for the scoring. Another idea to optimize the restoring process of the original image is to train the deconvolutional network to display an image which is closer to the input image. The framed images at the bottom left of each image in figure 4.6 shows a magnified version of Lenna's eye region in order to make the different decompression patterns visible. This patterns are comparable to the inference patterns through bad stride, pad, filter size combinations in figure 3.23, but in comparison to the checkerboard artifact effect this patterns are not filterable, because this decompression patterns have no reiterativeness. You can see in the frequency domain representation (figure 4.15a) of the image in figure 4.6c, that there is beside a random noise no detectable inference pattern which we can filter out.
In figure 4.15c you can see one example for checkerboard artefact's through bad chosen



(a) grain merge with original image and output of deconvolution

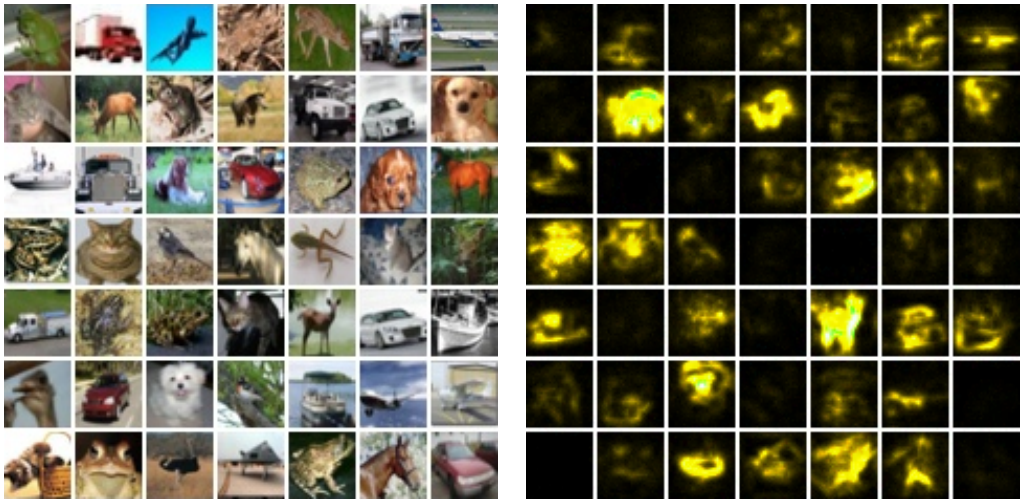


(b) difference between object to zero and untouched tensor$\mathbf{T_{C_u}}$

Figure 4.11: object localization with option 2

the coloration shows whether the classification was made through positive or negative values; positive means that one features was dominant and was responsible for the classification, negative means the lacking of other feature classes lead to a classification.

convolution parameters. Figure 4.15d shows the image with filtered inference patterns, which we achieved by blurring out the main inference distortions at the red and blue circles (figure 4.15e) in the frequency domain representation of the image.
After the inversion of the fc-layer we proceed with the inversion of the remaining convolutional-, batch-, pool- and ReLU layers as explained in chapter [3.1.4], so that we can form the whole deconvolutional process as shown in figure 4.7.
In order to evaluate now the responsible input pixels for the object recognition, we have to manipulate the output tensor $\mathbf{T_C}$ of the convolutional part before sending it back into the deconvolutional part of the network. We have several options for manipulating this tensor $\mathbf{T_C}$ with differently good results. The simplest option is to set all mini-batch layers to zero except for the one layer with the highest object detection score. Every mini-batch layer in the tensor $\mathbf{T_C}$ contains only information about the detection of one object class, so if we pass just the one layer with the highest score to the deconvolution and fill the rest layers with zeros, we should get at the output of the deconvolutional part of the network something similar to the input image, where all pixels are black or darkened without the pixel areas of the detected object. Unfortunately this whole con- and deconvolution process cannot reassemble most of the pixels reasonable enough, without the informations of all mini-batch layers together, so that the human eye would make sense of the output. We only receive something like this figure 4.9b:
The next option is to set just the layer in tensor $\mathbf{T_C}$ which recognized the object to zero and hand it over to the deconvolution. In addition to that, we send another untouched tensor $\mathbf{T_{C_u}}$ through the deconvolution and calculate the difference between these two processed outputs of the whole network. This methods provides a better



(a) CIFAR-10 clip (b) heat map

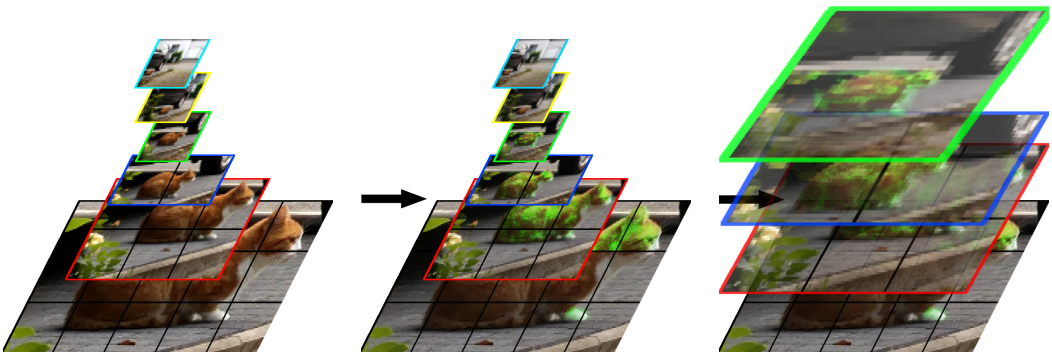
Figure 4.13: object localization with option 3

contrast between areas where an object stimulates the detection and where nothing stimulates the detection. This is the method i chose for this thesis approach. See for more output images in chapter [5.3]:
The third option takes nonlinear effects through the transfer functions into account and tries to invoke different activation values for different features due to a stochastic method.
For that we need to send several tensors $\mathbf{T_C}$ through the deconvolution. Each tensor $\mathbf{T_C}$ will have randomized values in the layer which correlates to the network's detected object. At the output of the deconvolution we will keep track of the pixels which changes more often for different input tensors $\mathbf{T_C}$. The result is a heat map, for the pixels areas which oscillate the most, which indicates a high input stimuli correlation to the recognized object. This method provides the best input stimuli detection, but needs also the most computing resources due to several deconvolution processes for one image. Because the gaol of this thesis approach is to lay the basics for real time object tracking in big pictures, which needs a lot processing time, this method isn't applicable for that conditions.

4.2 Object Tracking

The goal of object localization was to lactate different classes of objects in different images – one object in one image at a time. The goal of object tracking is to locate one specific object in a big image which multiple objects on it. To do that we disassemble a bigger image (for example a 720p image) into our tiny 32×32 pixel images our network can process. However we could process slightly bigger images with small modifications on our network, too, but no regular sized images like 720p images. The problem is if we just disassemble the big input image in 32×32 pixel images, the object could expand over more of these tiny images fragments and the network may not detect its features. Therefore we disassemble the image in different sizes (32×32,64×64,128×128,...) and then pool down all image fragments to 32×32 pixel fragments. Now the network receive a pyramid like stack of images (figure 4.16c), where all input images have a size of 32×32 pixels and capture different sized areas on the original image, see figure 4.16b.

Then the network tried to classify each image in two kinds of classes. Object detected



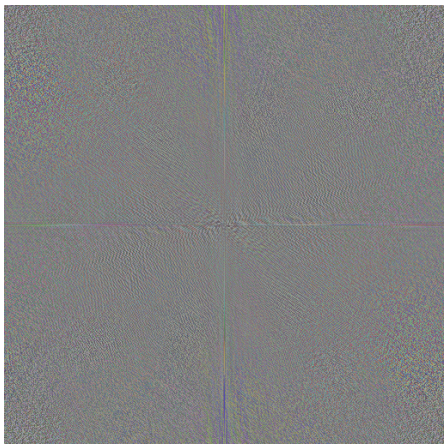
(a) (b) (c)

Figure 4.14: processing of the image "pyramid"

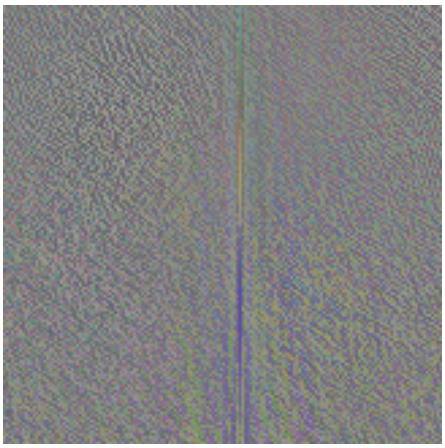
and no object detected. In case of our example image: Cat detected or no cat detected. In case of a detection the deconvolutional part of the network is activated to locate the recognized features. This output is than overlayed over the original input image fragment, to mark the detected areas of the target object on the original input image, or to extract the correct pixel locations for further applications which relay on object tracking. For that we have to put every fragment back to its original position in the original image to extract the right pixel coordinates. In case of the down pooled input fragments we have to unpool these fragments to their original size again with the marked areas on them to receive the correct original image pixel locations of the object. We end up with a stack of output images with one reassembled and marked image for each fragment size we split the original image up with. Some of these output images might not have recognized the features for a specific object, because they were to big or to small. To receive one final output image with all recognized features in all output images we simply overlie all output images with each other (figure 4.14).
To test these idea I used the CIFAR-10 dataset again, but this time I relabeled all images into two categories. Label 1 for pictures with cats on it. Label 2 with all other 9 object classes. The problem is now, that the network can choose to ignore the cats, because it receives a score of 90% correct labeled images, if it always detect label 2 on each image. To combat this I weighted the score for labeling a cat image correctly or wrongly with a 9 times higher value so that pure guessing would result in a score of 50%. If someone would form a specific tracking dataset he could also make sure that the target object class has the same amount of images than the no-target object class.

This is the whole approach of this thesis which was implemented, but this reaches not just jet the goal for a sound real time object tracking. More on this in chapter [6.1].

Figure 4.15: inference signal filter

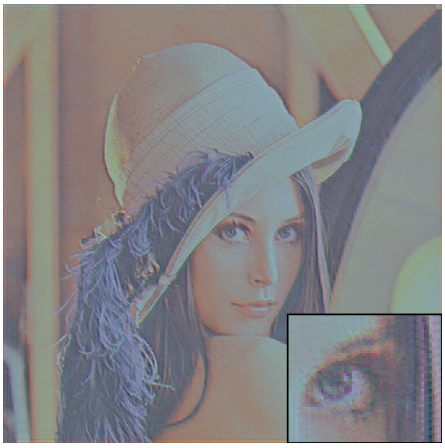


(a) (all channels) decompression patterns after FFT of 4.6c

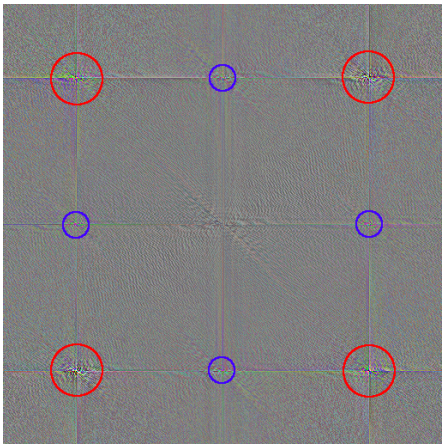


(b) bottom center zoom of 4.15a

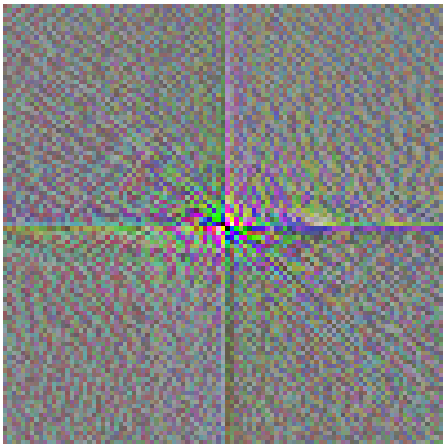
(c) convolution $\curvearrowright$ deconvolution :
kernel size = 4×4
stride (1) = 1, (2) = 3; padding = 2;
mini-batch transformations:
$3 \overset{(1)}{\rightarrow} 64 \overset{(2)}{\rightarrow} 128 \overset{(2)}{\curvearrowright} 64 \overset{(1)}{\rightarrow} 3$



(d) 4.15c with filtered inference pattern



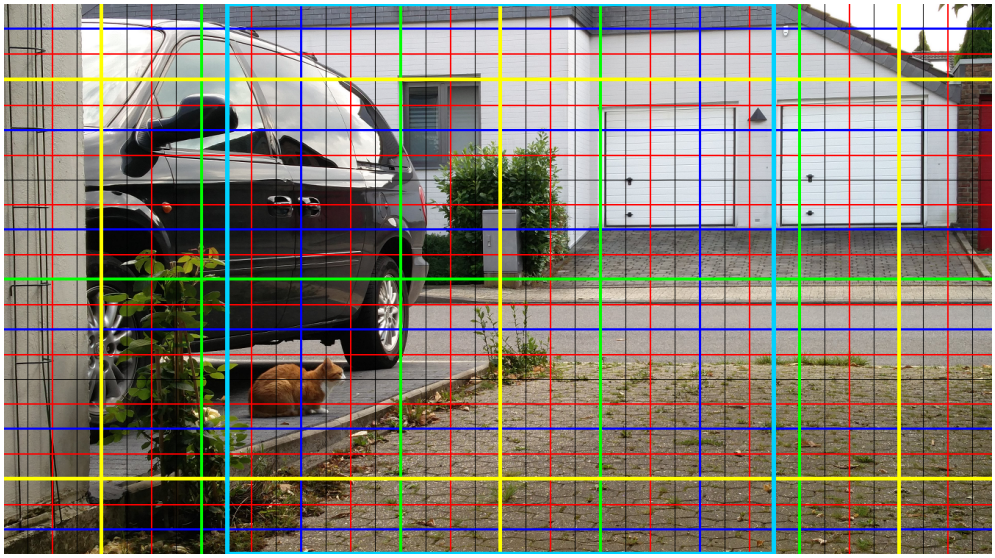
(e) all channels after FFT of 4.15c



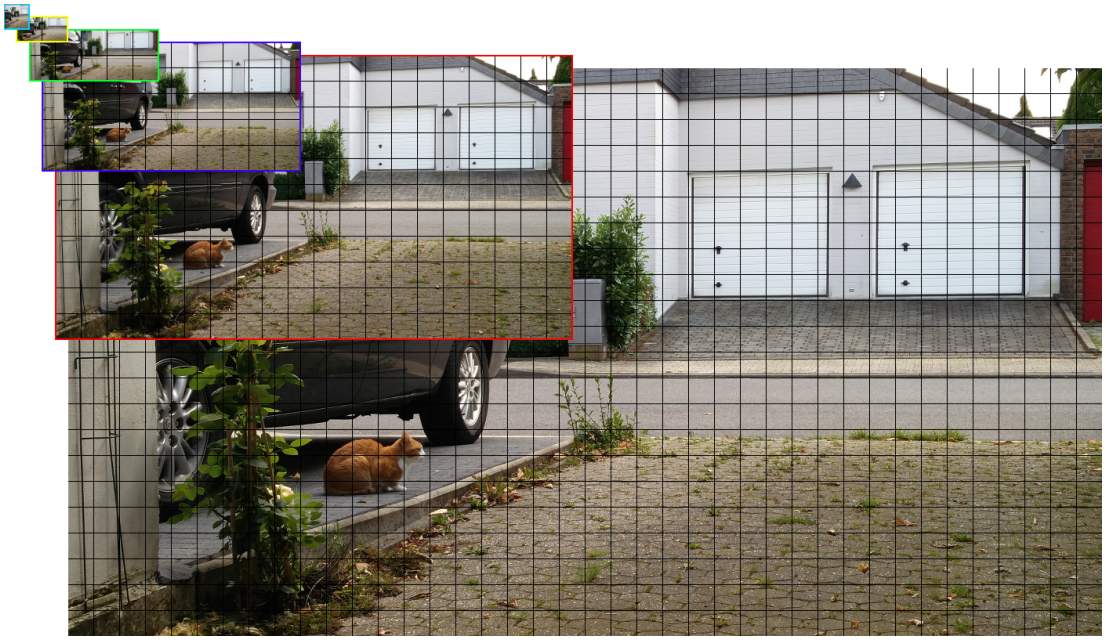
(f) bottom right corner zoom of 4.15e

Figure 4.16: image fragmentation

(a)



(b)



(c)