Universiteit Leiden
Opleiding Informatica

Visualizing Features Learned

by

Convolutional Neural Networks

Name: J.G. Kalmeijer

Date: 22/06/2016

1st supervisor: Dr. W.J. Kowalczyk
2nd supervisor: Dr. W.A. Kosters

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Visualizing Features Learned by Convolutional Neural Networks

Leiden University

J.G. Kalmeijer

June 27, 2016

Abstract

A pattern recognition pipeline consists of three stages: data pre-processing, feature extraction, and classification. Traditionally, most research effort is put into extracting appropriate features. With the advent of GPU-accelerated computing and Deep Learning, appropriate features can be discovered as part of the training process. Understanding these discovered features is important: we might be able to learn something new about the domain in which our model operates, or be comforted by the fact that the model extracts "sensible" features. This work discusses and applies methods of visualizing the features learned by Convolutional Neural Networks (CNNs). Our main contribution is an extension of an existing visualization method. The extension makes the method able to visualize the features in intermediate layers of a CNN. Most notably, we show that the features extracted in the deeper layers of a CNN trained to diagnose Diabetic Retinopathy are also the features used by human clinicians. Additionally, we published our visualization method in a software package.

Contents

1. Introduction

Today Convolutional Neural Networks (CNNs) are the dominating classifier on many image recognition tasks [1]. A predecessor of the CNN was already introduced in a 1980 paper [8]. Perhaps the most popular CNN is LeNet-5 [19], a network specialized in character recognition. For a long time this remained the only application of CNNs because of computational constraints. In 2011 computational performance was substantially improved by using GPUs [3], and in 2012 the best performance on multiple image datasets was substantially improved through the use of CNNs and GPUs [4].

CNNs are part of the "Deep Learning" approach to machine learning, where the learning algorithms try to learn high level abstractions or *features* of the data. In a traditional pattern recognition pipeline most research effort is put into finding such appropriate features. With Deep Learning and CNNs these features are discovered by the algorithm.

Manually crafting features has the advantage that the machine learning pipeline is very transparent. The model does not have access to hidden patterns during training, and thus will not use them. With Deep Learning features are not explicitly given to the algorithm, so it is no longer obvious what features are used.

Consider the task of recognizing faces. It might be that during data collection, the face of person A is always featured on a blue background, and the face of person B is always featured on a red background. If we manually extract features, we would not use the color of the background as a feature. Still, this feature might be used by a Deep Learning model. Thus, we might be interested in understanding the features learned by a CNN simply to validate the approach the network took to solving the task.

Another reason we might be interested in the features learned by a CNN is because they can make us aware of features that are significant to a certain domain, but that we were previously unaware of. If we manage to visualize the features learned by a CNN, we might be able to discover new knowledge.

This work looks at how to visualize the features in the intermediate layers of a CNN. We first introduce CNNs (Chapter 2). Next, we discuss existing methods for visualizing features learned by deep neural networks, and an

extension to one of these methods (Chapter 3). We show the results of our proposed extensions on two datasets (Chapters 4 and 5), and summarize and discuss our findings (Chapter 6).

This report is the result of a Master Thesis project for the master Computer Science at the Leiden Institude of Advanced Computer Science. The work was completed under supervision of Dr. W.J. Kowalczyk and Dr. W.A. Kosters. The main contributions of the paper are: 1) extending a method for computing salient areas in the input image as proposed by Simonyan, Vedaldi, and Zisserman in [25] to be applicable to convolutional layers; 2) showing that—by using this extension—a CNN trained to detect Diabetic Retinopathy uses features that are similar to those used by trained clinicans to diagnose the disease; 3) publishing source code for generating the visualizations of CNNs [16].

2. Convolutional Neural Networks

Consider the task of counting the number of visible spots in an image of a die. A simple approach would be to predefine a template of a dot on a die and structurally look for matches of these templates in the input image. A possible structural approach would be to slide the template over the image, writing a 1 whenever the template matches the area it is currently covering, and a 0 otherwise. We could then solve the task by counting the number of 1s in this binary map (Fig. 2.1).

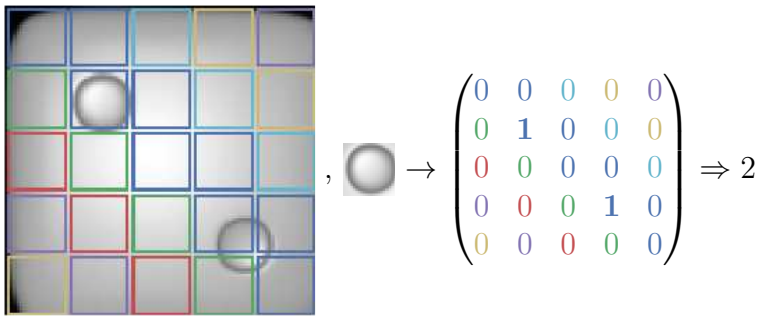


Figure 2.1: A naïve approach to solving the problem of counting the number of visible spots in an image of a die. The colors indicate the relationship between the alignment of regions of the input image (left) and the template (center). Everywhere the template matches the region sufficiently, we place a 1, and everywhere else a 0. This produces a binary map (right). In practice, it is desirable to have the regions overlap.

This approach is somewhat naïve since it assumes our single template matches all kinds of spots. If the spots occur in different scales and rotations, we might instead employ a variety of templates and create multiple binary maps. In this case, counting the number of occurrences is too naïve; multiple templates may match the same dot. Instead, we could use the maps as input for a Multi-Layer Perceptron (MLP) and try to learn how to optimally combine these maps. Here we assumed that we know a priori what templates to use. Depending on the task, this might not be obvious at all. Instead, we may also

parameterize the templates or *filters* and try to *learn* optimal filters. This is exactly what Convolutional Neural Networks (CNNs) try to do.

CNNs are a specialized version of neural networks that generally perform well on data that has a known grid topology, because this topology is hard-coded into the network architecture. MLPs have only a single layer type, called a *dense* or *fully connected* layer. These layers take an input vector and multiply this vector with a weight matrix, creating an activation vector. Next, a non-linearity is applied to this activation vector, creating the output vector (which may again be the input vector for a subsequent layer). In a dense or fully connected layer each output node is fully connected with the input nodes. This is not the case with convolutional layers, because two assumptions are made about the data: first, the data is assumed to have some grid topology (e.g., time series, images), so we expect relevant patterns to occur in small neighborhoods in the grid; second, we assume that the occurrence of a pattern is interesting irrespective of the location of the pattern in the grid.

These two assumptions allow us to use *weight* or *parameter sharing* and *sparse connectivity*. Both of these substantially reduce the number of parameters in the neural network, allowing us to stack many convolutional layers on top of each other without overfitting. In Section 2.1 we describe the convolutional layers, and most notably the convolutional operator in more detail. The non-linearities that follow a convolutional layer are described in Section 2.2. Another important layer type is a subsampling or *pooling* layer, which we describe in Section 2.3. We conclude by defining the concept of a *receptive field* in Section 2.4.

2.1 The Convolution Layer

Consider an MLP that operates on images of size 512×512. An MLP with 10 hidden nodes would have $2\,621\,440$ $(= 512 * 512 * 10)$ parameters in the first layer, suggesting that an MLP is not a good approach. Convolutional layers do not fully connect the neurons. Instead, they use something called *kernels* or *filters*. The number of parameters in such a filter is independent of the input size. The kernel might be 11×11 in size and if 64 kernels are used, then the number of parameters is only $7\,744$ — a mere 2% of the parameters used by the first dense layer of the MLP.

Kernel convolutions are an important concept in image processing [6]. A kernel is a small matrix that when convolved with an image, can, for example, detect edges in that image (Fig. 2.2). The convolution operator does this by producing a new matrix that has large values where edges occur in the original image, and small values where they do not. The exact function performed depends on the element values of the kernel. In convolutional networks these kernels are parameterized, so that they may be optimally configured for the task at hand.

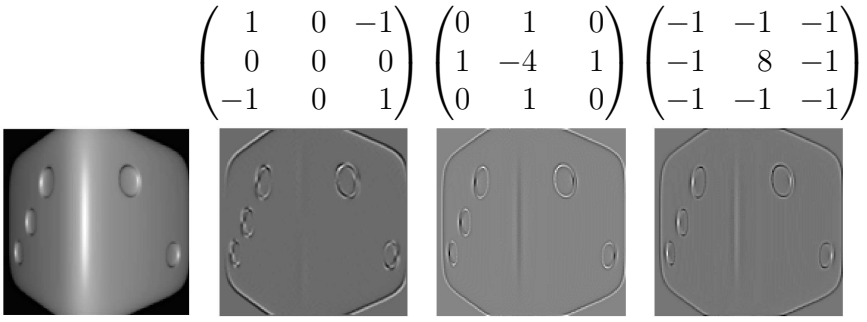


Figure 2.2: Examples of kernels (top row) that, when convolved with an image (bottom left), detect certain types of edges in that image (bottom row).

A convolution is achieved by centering the kernel (which is of odd width and height) on each pixel of the processed image, then, for each pixel, the corresponding entries in the kernel and image are multiplied, followed by summing. The resulting matrix is called a *feature map* (Fig. 2.3 shows an example).

More formally, let K be a $k \times \ell$ matrix representing the kernel, let I be a $p \times q$ matrix representing the image, with $p \geq k$ and $q \geq \ell$, then elements of the feature map F at position (i, j) can be defined as:

$$F(i,j) = (I * K)(i,j) = \sum_{m=1}^{k} \sum_{n=1}^{\ell} I(i-m+1, j-n+1) K(m,n) \qquad (2.1)$$

Notice that, even when the feature map is translated by $(p-1)/2$ and $(q-1)/2$ this definition still does not exactly correspond with centering the kernel on top of a pixel in the image, and taking the sum of the element wise product of the overlapping region — instead, the kernel is first flipped horizontally

$$I = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 7 & 8 & 9 & 0 & 7 \\ 1 & 2 & 3 & 4 & 5 & 8 \\ 6 & 7 & 8 & 9 & 0 & 9 \\ 1 & 2 & 3 & 4 & 5 & 0 \end{pmatrix}, K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

$$I * K = \begin{pmatrix} \mathbf{159} & 204 & 209 & \mathbf{230} \\ 234 & 279 & \mathbf{244} & 278 \\ 159 & \mathbf{204} & 209 & 274 \end{pmatrix}$$

Figure 2.3: Example of a *valid* convolution of a 6×6 input I with a 3×3 kernel K, producing a 4×4 feature map. Colors mark the region of the input considered for computing an output element.

and vertically (for example, by left and right multiplying the kernel with the anti-diagonal identity matrix).

Instead of the convolution operator, we typically use the *cross-correlation* operator (denoted $\star$). If we first flip the filter K both horizontally and vertically and call this K', we get:

$$F(i,j) = (I * K)(i,j) = (I \star K')(i,j) = \sum_{m=1}^{k} \sum_{n=1}^{\ell} I(i+m-1, j+n-1) K'(m,n) \tag{2.2}$$

This operator corresponds better to our intuition, where we view the kernel as a template of the pattern we are looking for.

The cross-correlation operator is only used in the context of CNNs specifically. In the context of image processing the convolution operator is typically preferred to cross-correlation, since it is commutative (and cross-correlation is not); when using multiple kernels to process an image the kernels can first be convolved together, followed by a single convolution with the image. In CNNs this optimization is not possible, since we need the intermediate activations (i.e., elements of the feature maps) for backpropagation, and the activations are typically transformed using a non-linearity (Section 2.2). From hereon we assume that convolution layers calculate cross-correlation, unless stated otherwise.

Special care must be taken when performing convolutions at the boundaries of the input. There are multiple ways to handle the boundaries. For one, we could consider only *valid* convolutions [20, 18, 24]. That is to say, convolutions where every element in K overlaps with some element in I. In this case, pixels at the boundaries of the image are somewhat ignored (being relevant to perhaps only a single element in the feature map, while pixels in the center of the image are part of the computation of $k \times \ell$ elements). This type of convolution produces a feature map of dimensions $(p - k + 1) \times (q - \ell + 1)$.

Another approach is to require that each pixel is part of the same number of convolutions. This is typically called a *full* convolution [20, 18, 24]. In this case we pad the matrix I with $k - 1$ elements on both sides of the first dimension and $\ell - 1$ elements on both sides of the second dimension. As padding a neutral value should be chosen (such as 0 for standardized data). This produces a feature map of dimensions $(p + k - 1) \times (q + \ell - 1)$.

A very convenient form of padding is one where the dimensions of the input image and the output feature map are identical. This is called *same* convolution [20, 18, 24]. In this case we pad with $(k - 1)/2$ elements on both sides of the first dimension and $(\ell - 1)/2$ elements on both sides of the second dimension. This approach simplifies network architecture design, as layers can simply be stacked upon each other indefinitely, while also preserving at least some of the information at the boundaries of the input.

Notice how each element in the feature map F depends on the same parameterized kernel K. This means all entries in F *share parameters*. Therefore, we have only $k \times \ell$ parameters, no matter how large the input is. These parameters have two interesting relationships with the parameters of an MLP. First, if we were to make the kernel the same size as the image, a convolutional layer performs the same computation as a dense layer (assuming a single hidden node and kernel). Second, if we constrain the weight matrix of a dense layer to have zero values at the right positions, a dense layer would perform the same computation as a convolutional layer; a convolutional layer can thus be interpreted as a dense layer with enforced *sparse connections*.

So far we have only considered the case where the image is a 2-D matrix, and a single 2-D kernel is used. In practice a single convolutional layer contains f different three-dimensional kernels (denoted by a 4-D tensor $\mathbf{K}$ with dimensions $f \times k \times \ell \times c$), that are convolved with a three-dimensional input $\mathbf{V}$ with dimensions $p \times q \times c$. At the first convolutional layer, p represents the

height of the input, q the width of the input, and c the number of channels of the input (e.g., one channel for each RGB component, or a single channel for a grayscale image). For a *same* convolution, this produces a $p \times q \times f$ tensor $\mathbf{Z}$.

The definition of $\mathbf{Z}_{i,j,k}$ is as follows:

$$\mathbf{Z}_{i,j,k} = \sum_{u=1}^{c} \sum_{v=1}^{k} \sum_{w=1}^{\ell} \mathbf{V}_{u,j+v-1,k+w-1} \mathbf{K}_{i,u,v,w} \tag{2.3}$$

Depending on the type of padding used not all entries may be valid.

In practice a *stride* (s_1, s_2) is sometimes used during the computation of the convolution. Remember that conceptually computing the convolution is as simple as centering the kernel on each pixel, doing the multiplications and summing the result. After obtaining the entire feature map we can perform downsampling, by for example keeping only every third row, and every other pixel in those rows. Instead of discarding many computations, we can simply perform only the computations for every third row and every other pixel in those rows. This would correspond to a stride of $(3, 2)$, and reduces the number of computations by a factor of 6.

If we first define a function r that generates a sequence of stride indices when given *start*, *end*, and *stride* as follows:

$$r(start, end, stride) = (start, start + stride, \ldots, \left\lfloor \frac{end - start + 1}{stride} \right\rfloor \times stride),$$

then Eq. (2.3) can be modified to account for strides as follows:

$$\mathbf{Z}_{i,j,k} = \sum_{u=1}^{c} \sum_{v \in r(1,k,s_1)} \sum_{w \in r(1,\ell,s_2)} \mathbf{V}_{u,j+v-1,k+w-1} \mathbf{K}_{i,u,v,w} \tag{2.4}$$

The output of the convolutional layer is typically transformed using a non-linearity. We discuss the commonly used non-linearities in the next section.

2.2 Non-Linearities

CNNs use, just like MLPs, nonlinear activation functions. Without them, the network would simply compute a linear function of the inputs, regardless of

the number of layers or *depth* of the network (since a linear composition of linear functions is again a linear function). Typically, an activation function is associated with a neuron on the network. We call the input to an activation function the neuron's *activation* and denote this using z, while the corresponding output is called the neuron's *output* and is denoted using h.

A variety of activations exists. Historically, the *sigmoid* (Eq. (2.5)) and the tanh (Eq. (2.6)) have been popular activation functions:

$$\sigma(x) = 1/(1 + e^{-x}) \tag{2.5}$$

$$\tanh(x) = 2\sigma(2x) - 1 = (e^{x} - e^{-x})/(e^{x} + e^{-x}) \tag{2.6}$$

The downside of these *sigmoidal* activation functions is that they saturate: if the activation of a neuron becomes too large or small, the gradient tends to zero. Since the gradient is backpropagated through the network, a saturated neuron in a deep layer can disturb the training process of all the neurons that are indirectly connected to it. An additional problem of the sigmoid function is that it is not zero centered, which can cause saturation problems in the layer that follows the sigmoid output.

The sigmoidal activation functions have been replaced by the Rectified Linear Unit or *ReLU* (Eq. (2.7)) and its descendants. ReLUs have been found to greatly speed up the convergence of stochastic gradient descent when compared to sigmoidal functions [17]. Additionally, the ReLU is computationally less expensive to calculate than the sigmoidal functions.

$$ReLU(x) = \max(0, x) \tag{2.7}$$

Unfortunately the gradient of the ReLU is zero for certain inputs. The *Leaky ReLU* (Eq. (2.8)) attempts to solve this by having a nonzero slope α (and thus also a nonzero gradient). Sometimes this slope is parameterized and learned during training, creating a *Parameterized ReLU* [10].

$$Leaky\ ReLU(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x >= 0)(x), \quad 0 < \alpha < 1 \tag{2.8}$$

Here $\mathbb{1}$ denotes the indicator function.

Another interesting activation function is the *maxout* [13] non-linearity. The maxout activation learns a piecewise linear function by combining activations

and taking their maximum. Maxout in CNNs aggregates over the activations of n feature maps, i.e., if $\mathbf{Z}_{i,j,k}$ is the activation at position (i, j) in the k-th feature map, and $n = 2$, then the output of the ℓ-th maxout node would be $\max(\mathbf{Z}_{i,j,2l-1}, \mathbf{Z}_{i,j,2l})$.

More formally maxout can be defined as a non-linearity that takes a vector of activations $\vec{z} \in \mathbb{R}^d$ as input, and produces a single output:

$$maxout(\vec{z}) = \max_{i \in [1,d]} z_i \tag{2.9}$$

The maxout non-linearity thus aggregates over the third dimension of the activations of a convolutional layer. This is similar to *pooling*, where we aggregate over the two spatial dimensions.

2.3 Pooling Layers

Pooling layers aggregate or *pool* their inputs along the spatial dimension. Much like convolution layers they can be thought of as sliding a window over the input, and computing a function that aggregates the values within the window. We call the size of the sliding window the *pool size*. Just like convolutional layers a pooling layer operates with a certain *stride*. The purpose of pooling is to make the network *invariant* to small translations of the input, and to reduce the number of computations. Pooling only makes sense if we assume that the exact location of a feature is not important to the task we are trying to solve; it does not matter if a feature occurs at position (i, j) or at position $(i + 1, j)$.

The most popular pooling method is *max pooling*, where we simply take the maximum of the sliding window. Taking the average, L^2 norm, or weighted average based on the distance from the central pixel are other popular methods. Typically the stride in a pooling layer is larger than 1, causing the layer to also perform downsampling. Fig. 2.4 shows an example of the different pooling methods.

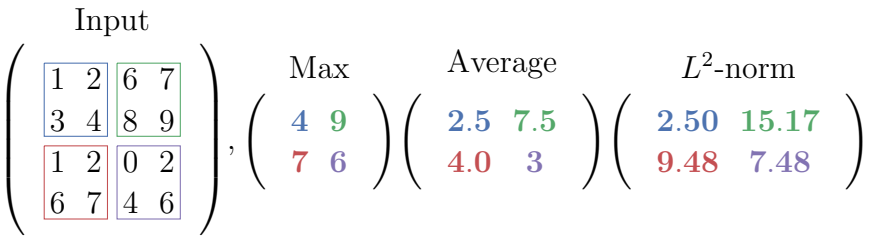


Figure 2.4: Examples of 2×2 pooling with a stride of 2 on a 4×4 input. Entries in the resulting matrices are computed by applying the aggregation operator shown on top of the resulting matrix, to on the correspondingly colored submatrix of the input. For example, $\mathbf{15.17} \approx \sqrt{\mathbf{6^2 + 7^2 + 8^2 + 9^2}}$.

2.4 Receptive Field

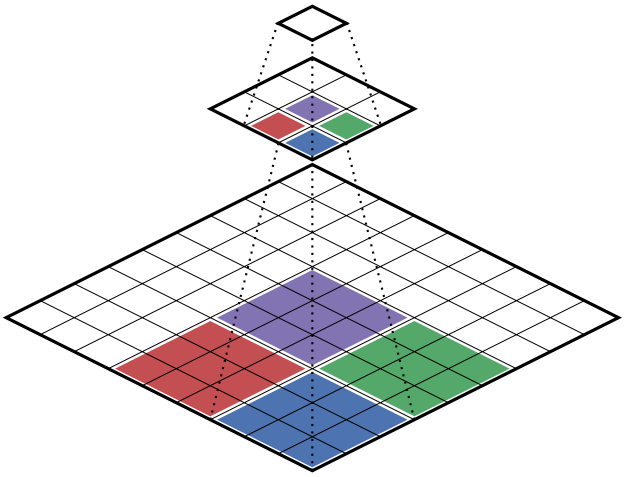
The region of the input to which a neuron in a CNN is connected is called that neuron's *receptive field*. The receptive field size is the same for all neurons in a given layer. In the first layer, the receptive field size is simply the size of the neuron's filter; as the network depth increases a neuron is aggregating over a larger and larger input area (Fig. 2.5). Clearly, since the size of our input images is limited, so is the sensible depth of a CNN.

The receptive field size is only interesting in convolution and pooling layers — after a dense layer each neuron is fully connected to the input. Both the convolution and pooling layer use a neighborhood size, i.e., the filter size or the pooling size, and a stride. The receptive field size can be determined by starting at the layer of which we wish to know the receptive field size, and working our way back towards the input. Consider for example Fig. 2.5. To generate a single output in the deepest layer, we need 2×2 outputs in the previous layer, since this is the deepest layer's filter size. To generate the first of those 2×2 outputs in the first layer, we need just the neighborhood size for the first output, and any subsequent outputs require $stride_{0,1}$ and $stride_{0,2}$ inputs across the first and second dimension. This makes the receptive field size 6×6 ($= (2-1) * 3 + 3$ across both dimensions).

More formally: the receptive field size rfs for the ℓ-th layer can be computed with the following formula, where $size_{i,k}$ and $stride_{i,k}$ respectively denote the

neighborhood size and stride of the layer at depth i in the k-th dimension:

$$rfs(\ell)_k = \sum_{i \in [0,\ell]} size_{i,k} \prod_{j \in [0,i-1]} stride_{j,k} - \sum_{i \in [0,\ell-1]} stride_{i,k} \tag{2.10}$$



$stride_1 = (1, 1), size_1 = 2 \times 2$

$stride_0 = (3, 3), size_0 = 3 \times 3$

Figure 2.5: Illustration of how the depth of the network affects the receptive field. The neuron in the deepest layer uses a filter of size 2×2. Because the layer below uses a filter of size 3×3 and a stride of $(3, 3)$, the receptive field size of the deepest neuron is 6×6.

3. Methods

We would like to visualize the features learned by a CNN. A naive approach would be to take the data on which the CNN is trained, and for each class c visualize the examples in the data of which the network is the most certain that they should be classified as c. We might display a top n for each class, and try to manually detect commonalities within this top n and assume that this is what the network looks for.

Clearly, there are some problems with this. Firstly, it is very sensitive to human bias. We might already understand the classification problem very well, and would be inclined to assume that if the network performs well, it is computing a function similar to the one computed by human classifiers. Secondly, it does not make it any easier to discover new knowledge about a problem; we could have obtained the same insights by simply inspecting samples of each class, or by talking to the experts that labeled the class.

A better approach localizes the areas in the input that the network considers important, which we will call *salient* areas. Another good approach would be to construct artificial input that the network strongly considers important. We will call such inputs *exemplars*. These two approaches are not only applicable to the network's final output, but can also be used to gain insight in what regions individual kernels consider important, hopefully shedding light on the individual features that the network uses to classify an image and the effect of network depth.

3.1 Visualizing Exemplars

We previously mentioned how selecting a few examples that maximize the network's posterior output for a certain class provides us with limited insight. Ideally, we would like to be more general. The literature suggests two approaches. The first is Activation Maximization, where we try to find the input that maximizes a neuron's activation. The second is a Deconvolutional Network, where we try to invert the operations performed by a CNN, allowing us to project a filter back into input space.

3.1.1 Activation Maximization

Consider the situation where, in the final layer, a network computes the posterior for class c on some input x, and that the network has some parameters θ: $P(c|x;\theta)$. During training, we try to find values for θ such that some error is minimized on the training set.

When finding the exemplars, we can reverse the situation. Instead of assuming that the input x is fixed, we assume that the parameters θ are fixed and that we would like to find the optimal x^* such that the posterior is maximized:

$$x^* = \underset{x \text{ such that } ||x||=\rho}{\arg\max} P(c|\theta;x), \tag{3.1}$$

where ρ is a norm constraint to prevent unbounded solutions, and $||\cdot||$ is the L^2-norm.

The above procedure is not only limited to neurons in the top most layer. Erhan et al. introduced and used this method called *Activation Maximization* in [7] to visualize what was learned by Deep Belief Networks and Stacked Denoising Autoencoders in layers preceding the final layer.

They using suggest gradient ascent to find solutions, and characterizing the unit by either the different local optima found, the best local optima found, or the average of the local optima found. They also note that in practice they find the same optima when starting from different random initializations.

Proposed Extension

Deep Belief Networks and Stacked Denoising Autoencoders are fully connected, so each intermediate neuron considers the entire input. We suggest that this method can be extended to be applicable to the neurons in intermediate layers of CNNs by simply optimizing a neuron's receptive field. Remember that in a convolution layer many neurons share the same filter, so these neurons will naturally have the same exemplar input. It is therefore more natural to speak of the exemplar input of the filter or feature.

3.1.2 Deconvolutional Networks

Deconvolutional Networks are proposed by Zeiler and Fergus in [28] to map filter activity in the intermediate layers back to input pixel space. A Deconvolutional Neural Network is attached to a CNN, providing for each layer in the CNN a reversing layer that maps features to pixels (instead of pixels to features).

The procedure for visualizing a specific filter using a Deconvolutional Neural Network is as follows: first perform a forward pass on an input image using the CNN. Then, set all feature maps in the filter's layer to zero, with the exception of the feature map of the filter itself. Now project back to input space by performing a downward pass. During the downward pass the pooling operations are inverted by *unpooling* and convolutional operations are inverted by *rectifying* followed by *filtering* with a flipped filter. We now describe the inverse operations in more detail.

Consider the following example, where a matrix represents the activations that are about to pass through a max pool layer with a filter size of 2, and a stride of 2:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{pmatrix}$$

On the first iteration of the pooling operation the following submatrix is considered:

$$\begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}$$

Here the maximum is 6, located on position $(2, 2)$. The 6 is saved in the first position of the pooling result, while the position of the maximum, a so-called *switch*, is also recorded (this is what allows an approximate construction of the inverse of the pooling operation).

Eventually we end up with the following pooling result and switches:

$$\begin{pmatrix} 6 & 8 \\ 9 & 6 \end{pmatrix}, \big((2,2),(2,4),(3,1),(4,4)\big)$$

If we then start with a matrix of the same size as the original input, and enter our recorded maxima on the switch positions recorded and zeros everywhere

else we get the following approximation:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 8 \\ 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 \end{pmatrix}$$

This concludes the unpooling operation.

To invert the convolution the Deconvolutional Neural Net first rectifies the reconstructed output, followed by convolutional filtering using a transposed version of the filter.

Because the Deconvolutional Neural Network needs an inverse layer for each layer in the CNN the approach is limited to layers for which such an approximate inverse can be found. Zeiler and Fergus propose such inverse layers for max pooling layers, and convolutional layers that use rectified linear units.

3.2 Visualizing Salient Areas

In contrast to the previous approaches that try to visualize the filter in the original input space, the following two approaches try to determine what a filter considers important in the input by creating a saliency map. A saliency map S assigns to each pixel in the input image a real valued saliency score; for an $n \times m \times c$ input image (where c is the number of channels, e.g., 1 for grayscale and 3 for RGB images) an $n \times m$ saliency map is produced.

3.2.1 Gradient Based

The Gradient Based method [25] is proposed by Simonyan, Vedaldi, and Zisserman as a method of determining the relative importance of each input pixel according to some node in a neural network. They start with the observation that if the activation of each node would be a simple linear function of the input image I, the activation $h_n(I)$ of a node n would look as follows:

$$h_n(I) = \vec{w}' I + b, \tag{3.2}$$

where $\vec{w}$ represents the linear weights and b a bias term, and the matrix representing the image is assumed to be unrolled into a vector.

If we assume that the elements of I are generally of similar magnitude then the magnitude of the elements of $\vec{w}$ can be interpreted as the relative importance of the corresponding input pixel.

Obviously the assumption that h_n is a linear function of I does not hold for CNNs. However, h_n can be approximated with a linear function in the neighborhood of some image I_0 by computing the first order Taylor expansion:

$$h_n(I) \approx \vec{w}'I + b = \left.\frac{\partial h_n}{\partial I}\right|_{I_0} I + h_n(I_0), \tag{3.3}$$

In other words, the gradient of a neurons activation with respect to a certain input at a certain point can be interpreted as an approximation of the importance of the pixels in input space. For a $n \times m \times c$ image, the gradients will take the same form. To get the saliency map one can simply take the maximum over the third dimension.

Proposed Extension

All neurons in the intermediate layers that are part of the same feature map share their parameters (see Eq. (2.2), page 9). It is these parameters that primarily determine the pattern that is recognized. Since a feature map can contain many different neurons all looking for the same pattern, it makes more sense to create a saliency map with respect to the feature map itself than with respect to the individual neurons. The method proposed in [25] is only suitable for scalar valued functions. Because of this we suggest aggregating over the values in the feature map using either the max or sum as aggregation function. The feature that produces feature map F can be visualized by computing the saliency maps for the following functions:

$$F_{max} = \max_{i,j} F(i,j) \tag{3.4}$$

$$F_{sum} = \sum_i \sum_j F(i,j) \tag{3.5}$$

The saliency maps S of the pixels in an image I_0 thus become:

$$S_{max} = \left.\frac{\partial F_{max}}{\partial I}\right|_{I_0} \tag{3.6}$$

$$S_{sum} = \left.\frac{\partial F_{sum}}{\partial I}\right|_{I_0} \tag{3.7}$$

3.2.2 Image Occlusion

One approach to constructing a saliency map is by systematically removing or *occluding* parts of the image and observing the change in neuron's output. Zeiler and Fergus used this method to show that their model trained on the ImageNet dataset "is truly identifying the location of the object in the image," and not "[...] just the surrounding context" [28]. By occluding (in turn) both the object in the image and the context in which the image occurs while observing the effects on the posterior output they were able to conclude that the object was the most relevant part of the input.

The computation of a saliency map using the occlusion method reminds us much of the computation of a feature map in a convolution layer. An occlusion kernel of size $q \times q$ containing all zeros is centered on top of a pixel (i, j), and the corresponding values in the original image are replaced by the values in the kernel, creating a new perturbed image. The reason for choosing all zeros is that this is a neutral value for zero centered data (and CNN input is typically zero centered). A neuron's output on this perturbed image is then observed, and can be contrasted with the output on the original image: if the output has dropped, then the occluded region was positively salient to the neuron; if the neuron of a specific class has risen, then the occluded region was negatively salient to the neuron; if the output remains the same, then the neuron is indifferent to the occluded region. This process is repeated for every pixel, and the systematical occlusion visualized in Fig. 3.1.

The downside of this method is the computational complexity; for a 512×512 image, roughly 260 000 feed forward passes are required. For neurons in the intermediate layers only the receptive field needs to be occluded, which can be much smaller—but there can be many intermediate neurons. Running time can be reduced at the cost of accuracy by introducing a stride, much like the stride used by convolutional layers.

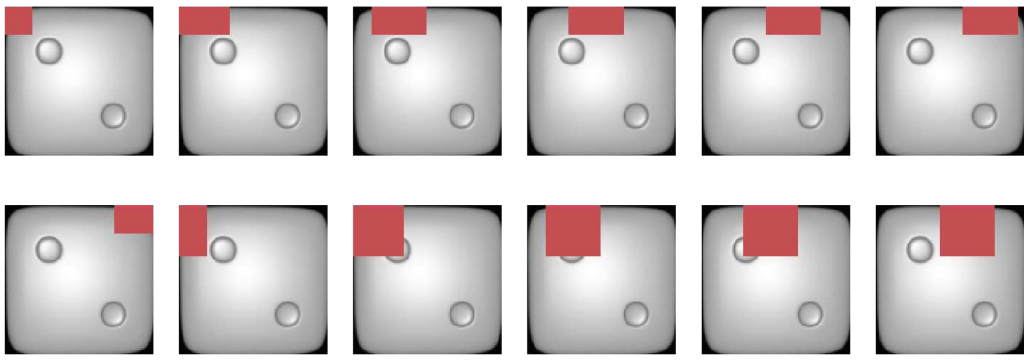


Figure 3.1: Example of systematic occlusions of a 111×111 image with an occlusion kernel of 41×41 and a stride of (17, 17). The occlusion kernel is shown in red. The total number of occlusions is 49 (12 are shown).

3.3 Conclusion

We described two existing approaches to visualizing the exemplars of a filter. Exemplars are inputs that strongly activate a given filter. These methods are called Activation Maximization [7] and Deconvolutional Networks [28].

We also described two existing approaches to visualizing the salient areas of a filter. Salient areas are regions in a specific image that a filter considers important. These methods are called the Gradient Based method [25] and the Occlusion method [28]. Our contribution is an extension to the Gradient Based method: by using an aggregation function we claim that it can be used to visualize the features in intermediate layers of the network as well as the features in the final layer.

4. Experiment: Diabetic Retinopathy

Diabetic Retinopathy (DR) is an eye disease associated with long-standing diabetes, and the leading cause of blindness in the working age population of the developed world. The World Health Organization estimates that 347 million people world wide have the disease. Trained clinicians diagnose the disease by evaluating digital color fundus photographs of the retina. "It is estimated that in 2002 diabetic retinopathy accounted for about 5% of world blindness, representing almost 5 million blind" [22]. It should come as no surprise then that automated grading of the DR presence in retinal fundus images is well studied.

On the 17th of February 2015 Kaggle started a competition on this problem [14]. They made available 88 702 fundus photographs of which 35 126 were labeled on a five grade scale. In total 661 teams competed over 7 months for a total prize pool of $100 000, of which $50 000 was awarded to first place, and $30 000 and $20 000 to second and third place respectively. The goal of the competition was to maximize the quadratic weighted kappa on a test set of 53 576 photographs. Kaggle of course kept the labels of this test set private. By the end of the competition top submissions were performing on par with human experts [15].

To receive the prize money winners need to document their solutions and deliver the source code used to generate the solution within 14 days. Presumably because of this teams are eager to share code and documentation at the end of the competition, even if they did not win any prize money. From this documentation we can conclude that all top teams used Deep CNNs to solve this task [27, 2, 9, 21, 5]. Training Deep CNNs is not an easy task, but fortunately De Fauw provided the values of the learned parameter values of the network with which he finished 5th place [5] (scoring a kappa of 0.83—1st place scores a kappa of 0.85). It is this model which we will experiment with.

The remainder of this chapter covers background information on DR. We will describe the different gradations, what different features are present in a fundus photograph, and how those features are used to determine the presence and severity of DR. We then describe the CNN architecture suggested

by De Fauw, apply visualizations techniques to determine what aspects and features of the fundus image this network considers important, and compare those features with the features used to diagnose DR by clinicians. We lastly summarize our findings.

4.1 Background Information

We first take a brief look at the main features of a fundus image, followed by features that are specific to DR, and then discuss the classification scheme used in the Kaggle DR challenge.

4.1.1 Main Features of a Fundus Image

The two most prominent features of a fundus image are the *macula* and the *optic disk*. The macula contains the *fovea*—which contains the largest number of cone cells and thus is important for good vision—and is located in the center of the fundus image if the patient looks straight into the camera. The optic disk is located in the direction of a patients nose, and is the main entry point for the major blood vessels that supply the retina. Figure 4.1 shows an annotated fundus image.

The fundus images used in the Kaggle challenge are graded according to the International Clinical Disease Severity Scale [26]. This grading scheme focuses on the following features:

Microaneurysms small areas of balloon-like swelling in the retina's blood vessels (Fig. 4.2);

Hemorrhages blood from a ruptured blood vessel (Figs. 4.2 and 4.3);

Hard Exudates lipid residues of serous leakage from damaged capillaries (Fig. 4.2);

Venous beading bulges in the walls of the veins (Figs. 4.2 and 4.3);

Venous loops looping blood vessels;

Neovascularization protrusions and outgrowth of capillary buds from pre-existing blood vessels (Fig. 4.3).

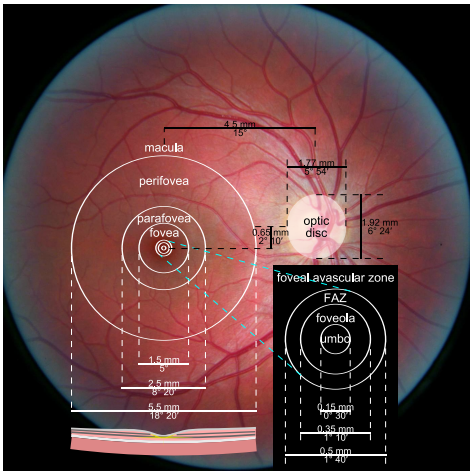


Figure 4.1: A fundus image in which the macula, fovea and optic disc are annotated (source [12]).

The location of such features matters. If exudates are present near the fovea, this means the disease is more severe than if they are far away from the fovea.

4.1.2 Grading

The fundus images used in the Kaggle challenge are graded according to the International Clinical Disease Severity Scale [26]. This scale has five levels:

1. No Diabetic Retinopathy (DR)
2. Mild Non-Proliferative Diabetic Retinopathy (NPDR)
3. Moderate NPDR
4. Severe NPDR
5. Proliferative DR

The first level is characterized by the absence of any DR features; the second by the presence of a few microaneurysm; the third by the presence of microaneurysms, intraretinal hemorrhages or venous beading. The fourth level requires that all previously mentioned features are present but in larger quantities. The fifth level is characterized by neovascularization, and by the pres-

Figure 4.2: Annotated fundus image classified as NPDR. Thin arrows: hard exudates; thick arrow: blot intraretinal hemorrhage; triangle: microaneurysm (source: [11]).

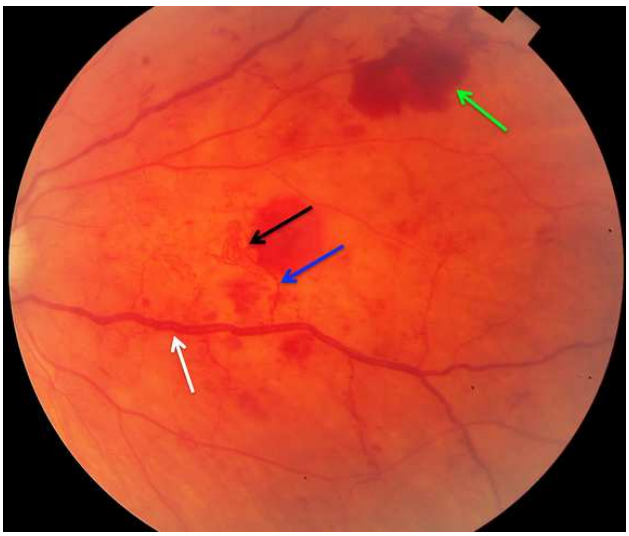


Figure 4.3: Annotated fundus image classified as PDR. White arrow: venous beading; blue arrow: stalk of neovascularization; black arrow: fan at the end of stalk; green arrow: preretinal hemorrhage (source: [23]).

ence of DR features at the fovea.

4.2 Used Architecture

The architecture and parameter settings used was provided by De Fauw [5] and is shown in Fig. 4.4. The model consists of 31 layers total, of which 23 are parameterized, giving a total of 20 923 690 parameters (nearly 20 times the size of the training and test set combined—although the train and test set were artificially augmented). Such a deep network was common among the competition's victors: networks of depths 25 [21]; 21, 25, 27 [9]; 21 [2]; and 21, 22, 23, 25, 27 [27] were used.

The network operates in batches of even size, because it considers a patient's left and right eye at the same time. Considering both eyes is crucial for good performance (classifying each eye of a patient with the class of the patient's other eye leads to a quadratic weighted kappa score of 0.85, which also happens to be the final score of the number one team—although the test and training set were split by patient, not by eye). The network also considers the resolution of the original fundus image as a feature.

All convolutional layers use leaky rectifiers with leakiness set to 0.5 as their non-linearity, while all but the last dense layer use maxout as their non-linearity, and the last dense layer uses a softmax non-linearity.

All convolutional layers use filters of size 3 and a stride of 2, with the exception of the first convolutional layer which uses filters of size 7. The max pooling layers use filters of size 3 and a stride of 2, so that the pooling areas overlap. Whenever maxout is used it computes the maximum of 2 linear functions. Multiple dropout layers are used with a dropout probability of 0.5.

4.3 Visualizing Salient Areas

We use the Gradient Based and Occlusion visualization methods described in Chapter 3 to visualize the features learned by the network. Because of the depth of the network we visualize the features at the five pooling layers only. The pooling layers consist of 32, 32, 64, 128, and 256 features respectively,

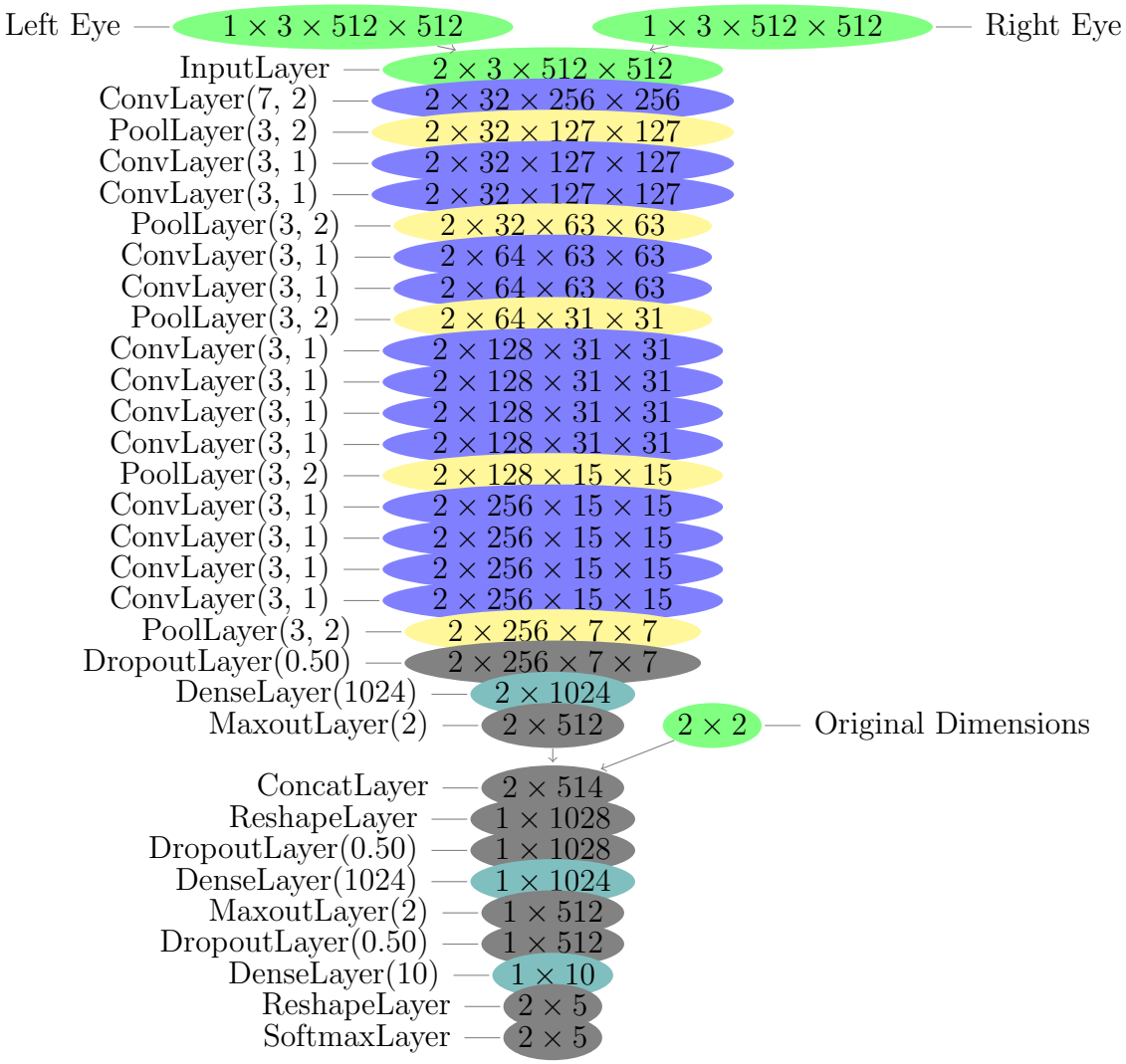


Figure 4.4: The network used in [5] to place 4th in the Kaggle competition. It takes as input the preprocessed fundus photographs of the left and right eye of a patient, as well as the original input dimensions of said fundus photographs. In the topmost layers the representations of the left and right eye of the same patient are combined. The complete network has a total of 20 923 690 parameters (nearly 20 times the size of the training and test set combined). For a convolutional layer ConvLayer and pooling layer PoolLayer the two arguments are the filter size and stride. For the maxout layer MaxoutLayer the argument is the number of features to take the maximum over. For the dense layer DenseLayer the argument is the number of hidden nodes in the layer.

and each visualization is done with respect to a single input image. As such it is infeasible to show all visualizations in printed format, and we merely highlight and summarize. For each of the features in the pooling layer we visualize the feature on the 9 fundus photographs that most strongly activate that feature, hoping this will summarize the feature.

Each of the methods produces a saliency map, i.e., for every pixel a real value indicating the importance of said pixel. To provide sufficient context we use the following visualization procedure. First, we use Gaussian smoothing with a small kernel on the saliency map. Then, a watershed algorithm is used to threshold to smoothed saliency map. This creates a binary map, marking each pixel as either important or unimportant. We then project this binary map on top of the original input by increasing the brightness in important areas. Additionally we delineate all important areas using a green marker. A blue marker is used to mark the receptive field of the neuron that is visualized.

We inspect the visualizations to see how the features learned by the filters of the network contrast to the features used by clinicians to diagnose DR. It might be that we see interesting features emerge simply because we are biased. To prevent this bias we contrast the visualizations of the trained network with the visualizations of a randomly initialized network that has the same architecture.

For each of the pooling layers we describe the learned features and highlight some interesting results. We summarize the learned features in Table 4.1 on page 37.

4.3.1 The First Pooling Layer

The first pooling layer (with a pool size of 3 and stride of 2) occurs after a single convolutional layer with a kernel size of 7 and a stride of 2 and contains 32 features total. The receptive field is already of size 11. This layer recognizes no features directly related to DR. The topmost activations are virtually always occurring at the edges of the fundus, suggesting that the network is detecting simple edges at this stage (Fig. 4.5). The randomly initialized network also always has the topmost activations occurring at the edge of the fundus.

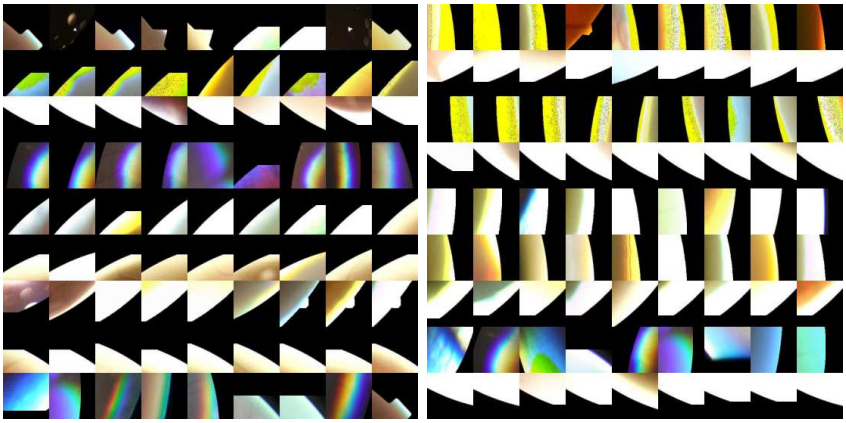


Figure 4.5: First pooling layer features of the trained network (left) and randomly initialized network (right). All features are focused on recognizing the fundus edge. The trained network contains a single feature that seems focused on blood vessels, while a random feature is detecting text.

4.3.2 The Second Pooling Layer

The second pooling layer (again, with a pool size of 3 and a stride of 2) occurs after two convolutional layers with a kernel size of 3 and a stride of 1 and contains 32 features total. The receptive field size is 35×35. Both the features of the trained network and the features of the randomly initialized network are most interested in images that have overexposed edges. Only in 2 out of 32 features the network detects something other than fundus edges; namely the optic disk (Fig. 4.6).

4.3.3 The Third Pooling Layer

The third pooling layer (with a pool size of 3 and a stride of 2) occurs after two convolutional layers with a kernel size of 3 and a stride of 1 and contains 64 features total. The receptive field size is 83×83. In this layer the features of the random network and the trained start to differ. The random network is still only recognizing simple edges, while the trained network detects white glare spots, the blood vessels of the optic disk, and microaneurysms. The filters are not yet consistent; features that detect microaneurysms also detect the blood vessels of the optic disk (Fig. 4.7).

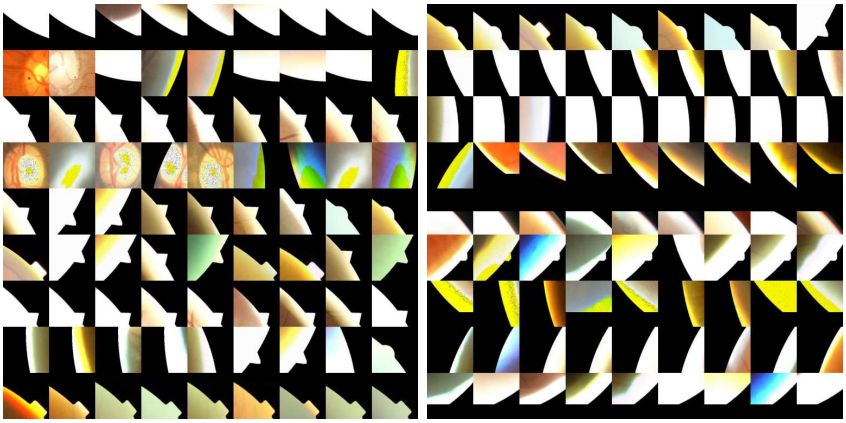


Figure 4.6: Second pooling layer features of the trained network (left) and randomly initialized network (right). Both the features of the trained network and the features of the randomly initialized network are most interested in images that have overexposed edges. Only in 2 out of 32 features the network detects something other than fundus edges; namely the optic disk.

4.3.4 The Fourth Pooling Layer

The fourth pooling layer (with a pool size of 3 and a stride of 2) occurs after four convolutional layers with kernel size 3 and stride of 1 and contains 128 features total. The receptive field size is 243×243. Microaneurysm detectors are present and seem consistent (Fig. 4.8). Hard exudates are sometimes confused with the optic disk (Fig. 4.9). The fovea can be mistaken for a hemorrhage (or the other way around) (Fig. 4.10). A specialized feature for detecting glare spots exists (Fig. 4.11). Some features recognize multiple DR features (Fig. 4.12), while others recognize image artifacts (Fig. 4.13). The optic disk is also detected (Fig. 4.14).

The random features are still primarily recognizing fundus edges, and activating most strongly on images with many artifacts.

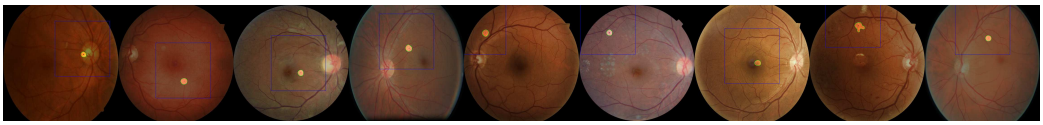


Figure 4.8: Example of a microaneurysm detector in the fourth pooling layer.

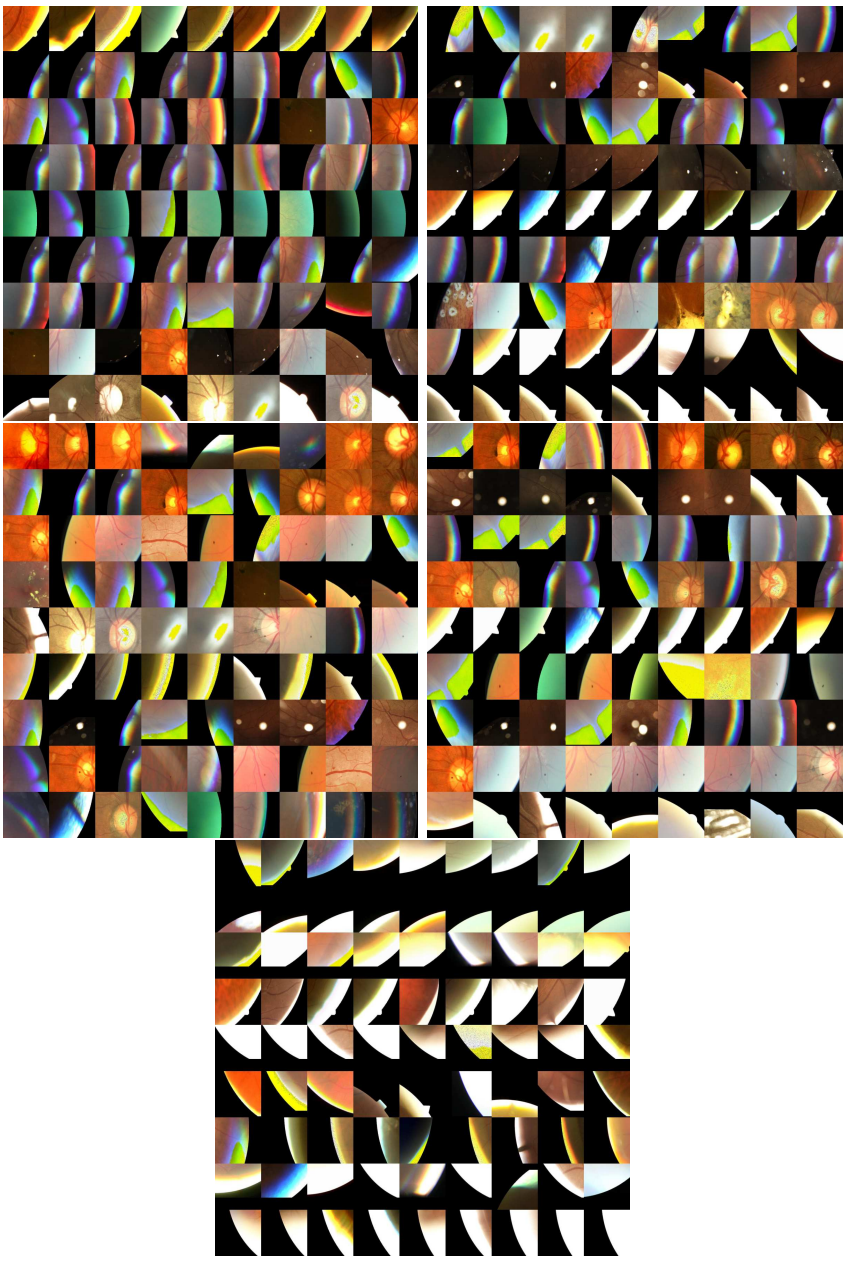


Figure 4.7: Sample of the third pooling layer features, of both the trained network (top 4 images), and the randomly initialized network (bottom center). The random network is still only recognizing simple edges, while the trained network detects white glare spots, the blood vessels of the optic disk, and microaneurysms.

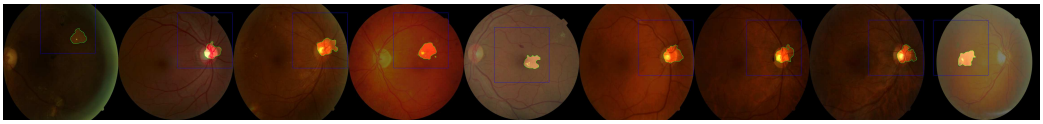


Figure 4.9: Hard exudates are sometimes confused with the optic disk.

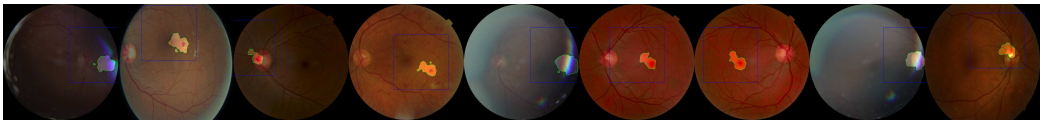


Figure 4.10: The fovea can be mistaken for a hemorrhage.

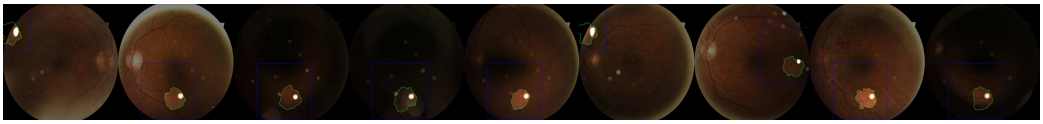


Figure 4.11: A specialized feature for detecting glare spots exists.

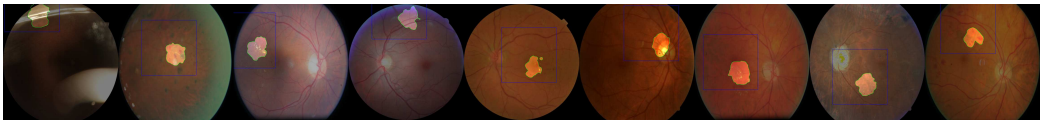


Figure 4.12: Some features detect multiple features of DR.

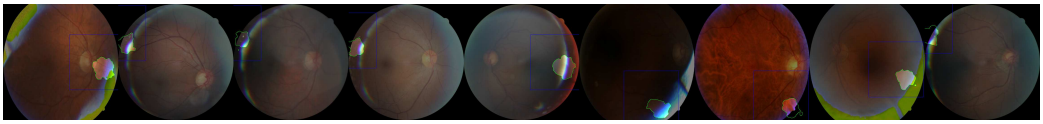


Figure 4.13: Some features recognize image artifacts.

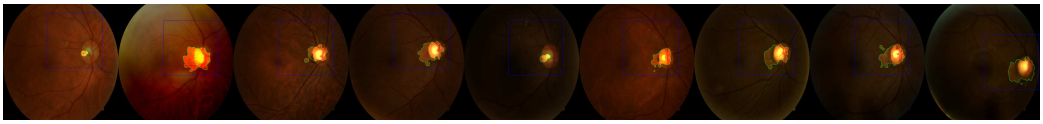


Figure 4.14: The optic disk is detected.

4.3.5 The Fifth Pooling Layer

The fifth and final pooling layer (pool size 3, stride 2) occurs after 4 more convolutional layers with kernel size 3 and stride 1. It contains 256 features total, and has a receptive field size of 563×563 (the original input image is 512×512, but each convolutional layer adds some padding).

Microaneurysm detectors are still present, although they occasionally mix in hemorrhages as well (Fig. 4.15). Hard exudate detectors are now more prevalent, occurring 8 times (Fig. 4.16). Hemorrhages are detected individually (Fig. 4.17). While in the previous layers detection was often localized to a single spot within the receptive field, now more complex shapes excite a neuron. This allows the detection of clusters of hemorrhages (Fig. 4.18).

Not all features are sensible at this level. For example, we have no idea what the feature in Fig. 4.19 detects. Blood vessels are now also detected (Fig. 4.20), but it is hard to say what makes them significant; these blood vessels show no signs of beading or neovascularization. Some features detected are hard to classify, but obviously indicate that something is wrong (Fig. 4.21).

Even at the deepest layer the features of the random network still seem centered around detecting edges in overexposed images.

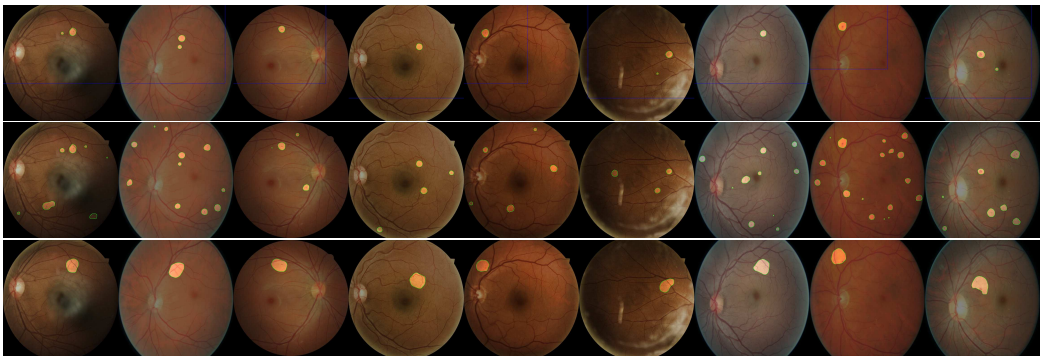


Figure 4.15: Example of a microaneurysm detector in the fifth pooling layer. The visualization methods are: Gradient Based with Max Aggregation, Gradient Based with Sum Aggregation, and Occlusion using an 11×11 occlusion window.

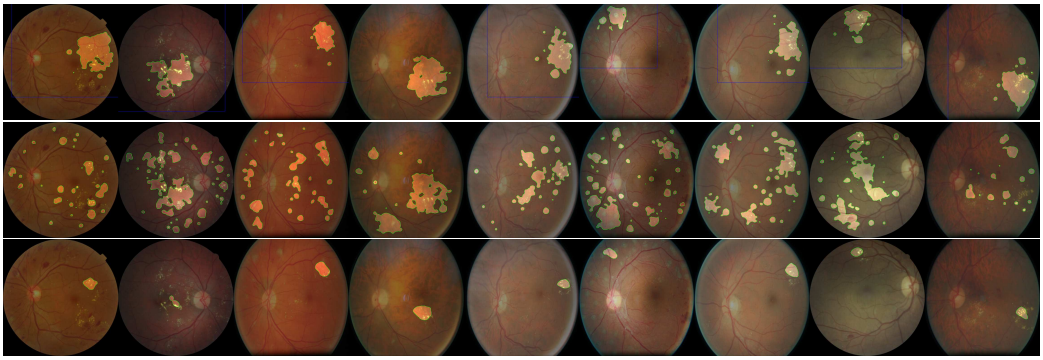


Figure 4.16: Example of a hard exudate detector in the fifth pooling layer.

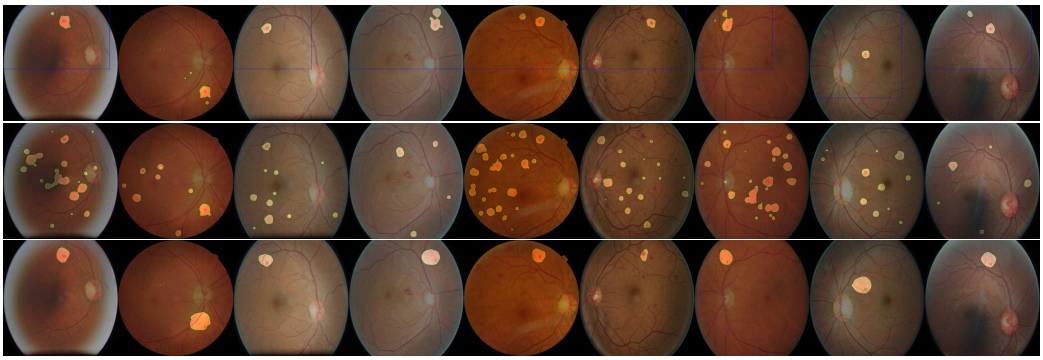


Figure 4.17: Example of a hemorrhage detector in the fifth pooling layer.

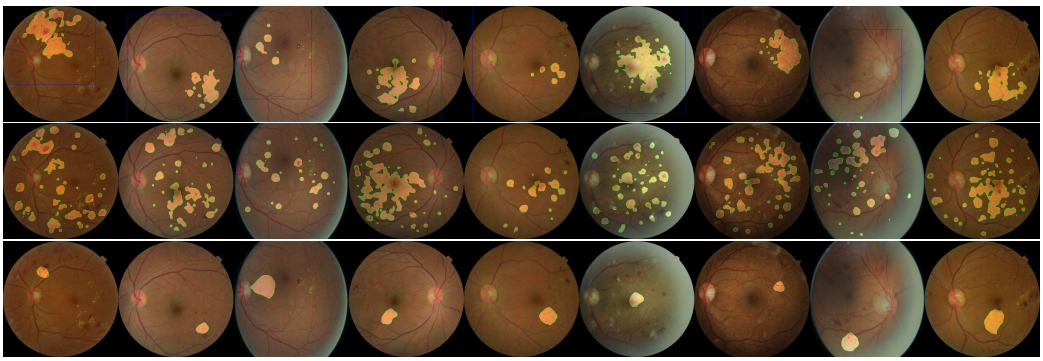


Figure 4.18: Example of a cluster of hemorrhages detector in the fifth pooling layer.

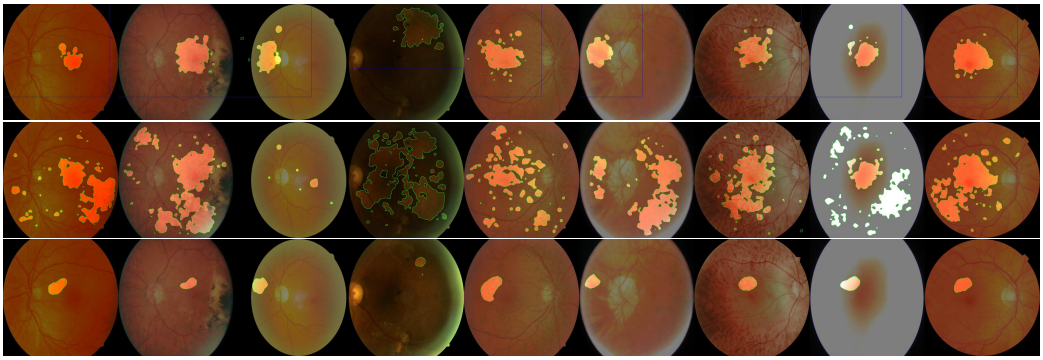


Figure 4.19: Not all features are easy to understand

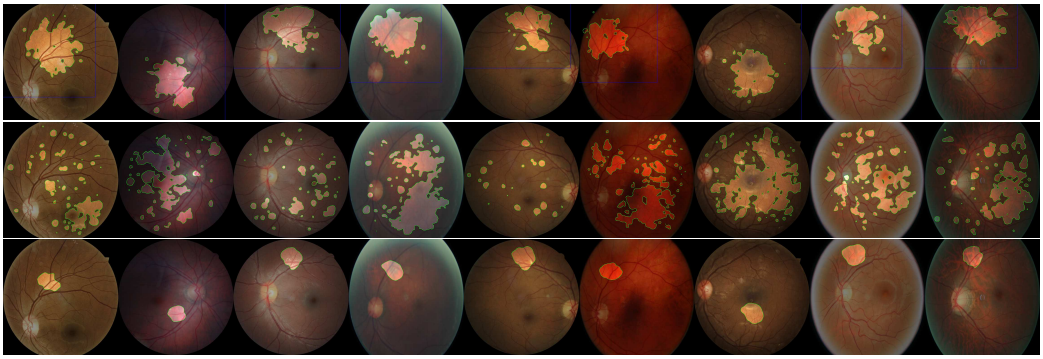


Figure 4.20: Example of a blood vessel detector in the fifth pooling layer.

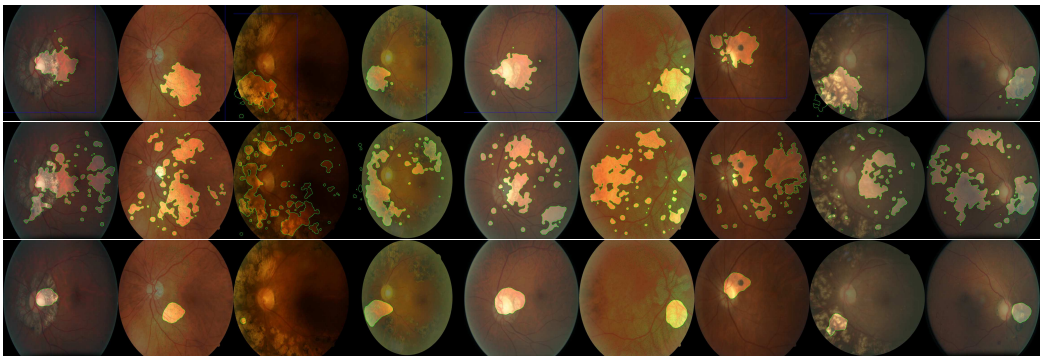


Figure 4.21: Example of detectors that are difficult to classify as DR features, but that seem to indicate disease.

					Frequency of Features				
Pooling Layer	Depth	Receptive Field Size	Number of Features	Hard Exudates	Micro-aneurysms	Hemorrhages	Optic Disk	Blood Vessels	Other
First	1	11×11	32						32
Second	4	35×35	32						32
Third	7	83×83	64		9		14		27
Fourth	12	243×243	128	1	17	2	1		29
Fifth	17	563×563	256	8	16	7		2	17

Table 4.1: Summary of the characteristics of the pooling layers and their recognized features. Frequency of features was determined by manually labeling the results of either 50 feature visualizations, or by labeling visualizations for all features in a layer if there were fewer than 50 features in that layer. Features were labeled as "Other" when they seemed to either recognize features unrelated to the task (typically image artifacts), or when they seemed to recognize a mixture of features.

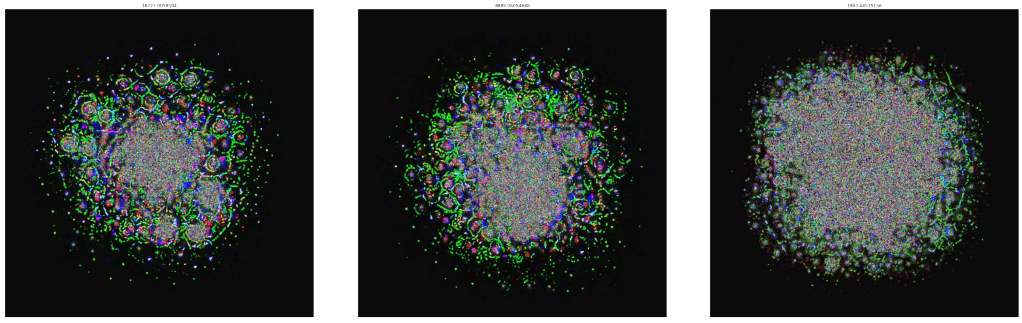
4.4 Visualizing Exemplars

We visualized the exemplars (inputs that most strongly activate a given feature) using the Activation Maximization method (Section 3.1.1). To choose a starting point for gradient descent we used three different initialization methods: maximum, mean, and random initialization. Each feature considers only part of the input: the receptive field. During maximum initialization we took a stratified sample of the dataset, we then computed the feature maps of the sample for a given feature, and for these feature maps we looked for the most strongly activated neuron. We used the input to this neuron's receptive field as a starting point. During mean initialization we computed the mean image of the entire dataset and used a patch from this mean image as starting point. During random initialization we generated random input with the size of the receptive field such that the input's mean was zero and the input's standard deviation one.

We visualized the exemplars by destandardizing the maximizers found by the Activation Maximization procedure. Figure 4.22 shows three exemplars for features recognizing different DR features. We find the results difficult to interpret. When we visualize the intermediate (non-converged) results of the Activation Maximization procedure the results seem more sensible: a blood vessel feature indeed marks the blood vessels (Fig. 4.23), while a microaneurysm marks microaneurysms (Fig. 4.24). We found that the difference between the activation of a neuron on regular input and a maximized input is large: large activations on the training and test data would still be smaller than 10; activations on optimized inputs can be as large as 10 000. Activation Maximization was previously used in the context of Deep Belief Networks, which used a sigmoid activation function [7]. The non-saturating nature of the rectified linear unit (ReLU) could explain the difference in the quality of the produced visualizations.

We found that mean and maximum initialization are easier to converge from, but no method consistently converges to a better local maximum. We found the local maximum of any initialization to be as much as a factor 2 larger than for the local maximum of a different initialization method. Visually the results seem similar for each of the initialization methods. Figure 4.25 shows an example. The visualizations seem to become more shapely or sparse as depth increases. In Fig. 4.26 we show exemplars of features at the third,

fourth, and fifth pooling layer. As the depth increases the filter starts ignoring information at the edges of the receptive field, even though all convolutional layers used *same* padding (which tries to prevent this).



(a) Exemplar for a feature recognizing microaneurysm (see also Fig. 4.15, page 34)

(b) Exemplar for a feature recognizing hemorrhages (see also Fig. 4.17, page 35)

(c) Exemplar for a feature recognizing hemorrhages (see also Fig. 4.20, page 36)

Figure 4.22: Examples of exemplars for different types of features.

4.5 Conclusion

At each layer the network extracts certain features using the network filters. We showed for each filter in the lower pooling layers the top 9 inputs that most strongly activated the filter. At the first two layers the filters are focused on detecting fundus edges, while deeper layers combine these individual filters to detect much more interesting shapes, such as hard exudates or microaneurysms. We saw that randomly combining filters does not lead to the emergence of interesting new features; when we visualized the features of a randomly initialized network, features in deeper layers were still the most strongly activated by the fundus edges.

At lower levels the filters seem sensitive to outliers. The inputs that most strongly activate a filter are typically part of an image that is overexposed or contains some other artifacts. As the network increases in dept it becomes less sensitive to these outliers, suggesting the network has learned to filter outliers. We were unable to determine how the network does this.

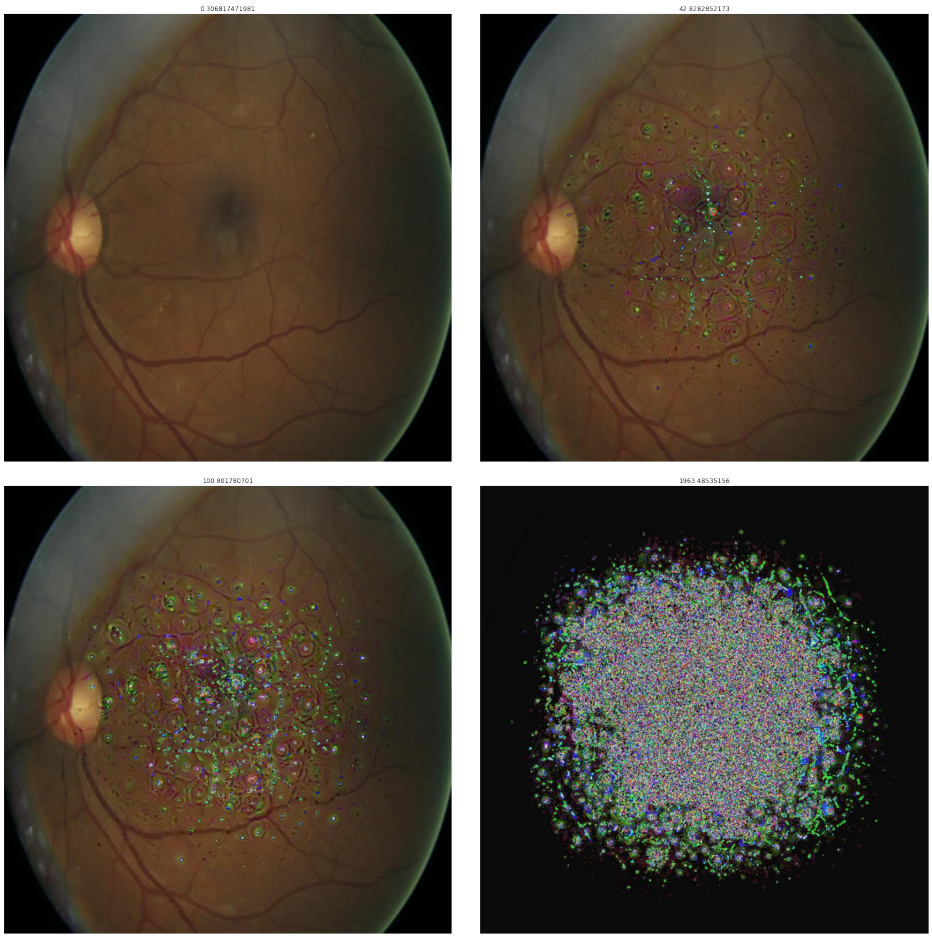


Figure 4.23: Intermediate results during the Activation Maximization procedure on a feature recognizing blood vessels when using maximum initialization. Notice how initially blood vessels are marked, but as the algorithm converges only some blood vessels at the boundaries of the input remain marked.

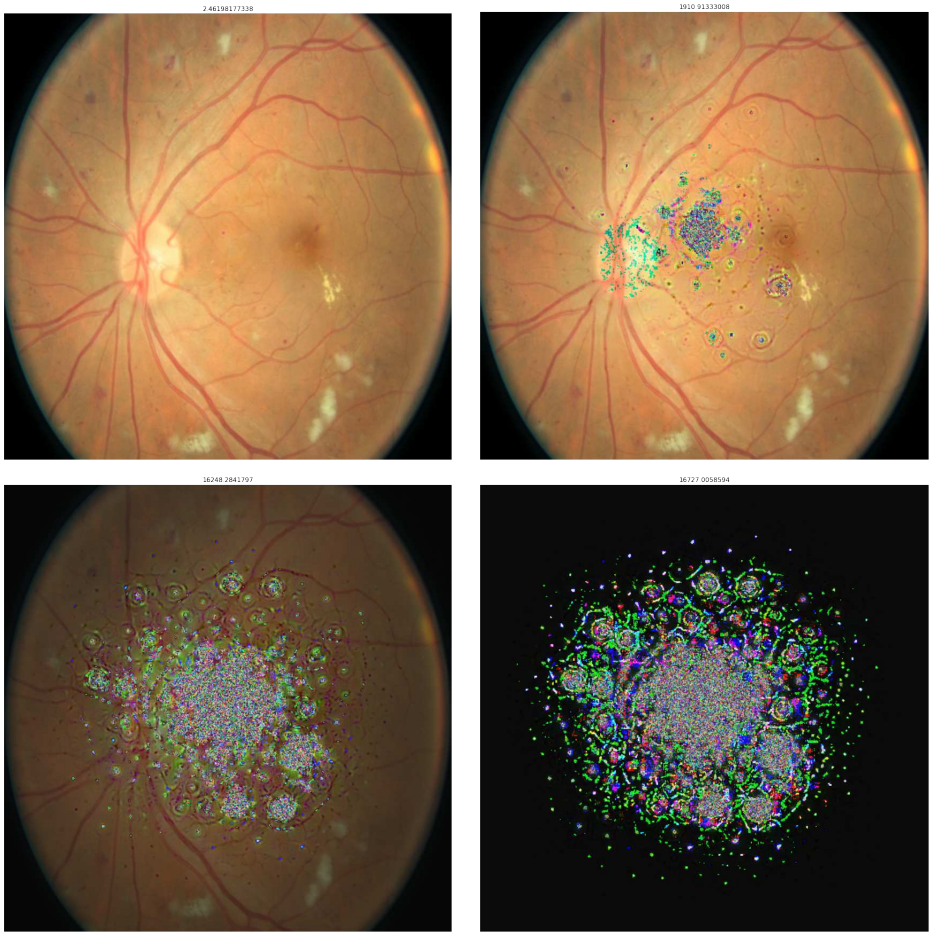
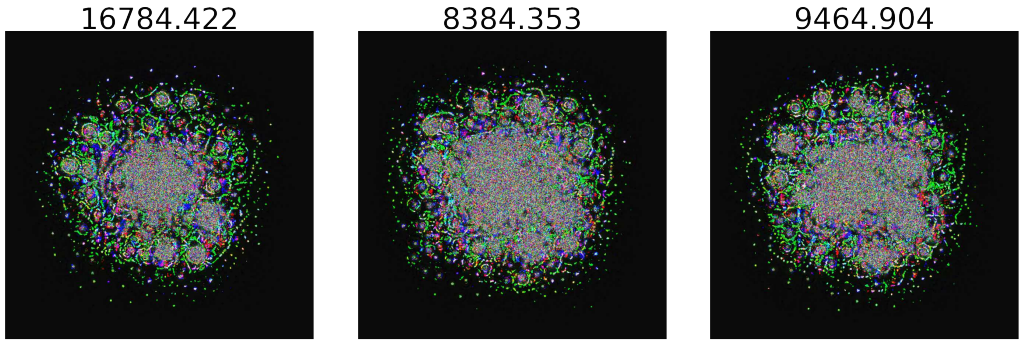
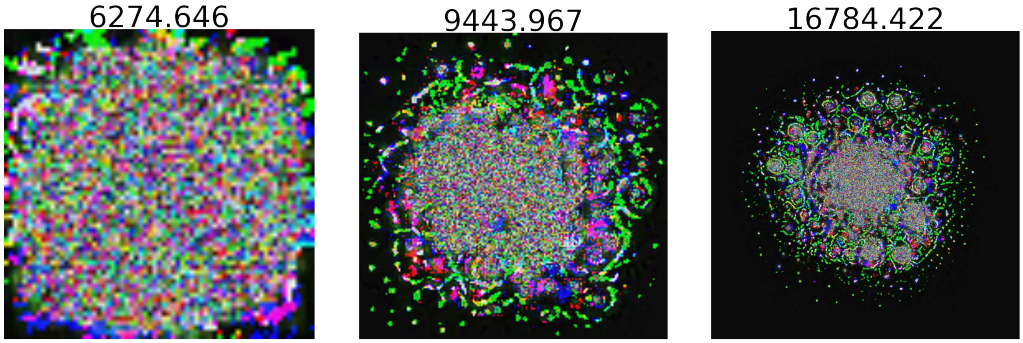


Figure 4.24: Intermediate results during the Activation Maximization procedure on a feature recognizing microaneurysms when using maximum initialization. Notice how the microaneurysm at the boundaries of the input are marked.



(a) Maximum (b) Mean (c) Random

Figure 4.25: Exemplars found by different initialization methods. Despite the large discrepancy in activation value (shown on top), the results still seem visually similar.



(a) Third Pooling Layer (b) Fourth Pooling Layer (c) Fifth Pooling Layer

Figure 4.26: Exemplars for different layers. The exemplars of the fourth and fifth pooling layer were both found to recognize microaneurysms by the method in the previous section.

By using the salient areas methods we were able to localize the important aspects of the input much more strongly than we would have been able if we were to just zoom in on the receptive field. Especially in deeper layers, where the receptive field size is large, showing just the input that is part of the receptive field as characterizing a feature is misleading: we found that it is often a very specific and localized part of the receptive field that is most relevant to a feature.

The salient areas found by all three methods, Gradient Based Max Aggregation, Gradient Based Sum Aggregation, and Occlusion, were similar. The Max Aggregation and Occlusion method both responded strongly to a very specific region only. This allows one to quickly grasp what a feature is about. The Sum Aggregation can subsequently be used to show some more details. The Gradient Based methods seemed to show more details than the Occlusion method, but this might be due to our choice of occlusion window size (11×11). By far the biggest drawback of the Occlusion method is the running time. Generating 9 Gradient Based maps for each feature in all five pooling layers takes in the order of hours on a GeForce GTX 980 Ti, while generating Occlusion based maps takes roughly a week.

We showed that most features used by clinicians to diagnose DR are learned by the network as the networks depth increases. We are unable to conclude whether this depth is a *requirement* to learn such features, or that a more shallow network would also be able to learn such features. It might be the case that in deep networks there are not enough constraints on the earlier layers to force them to learn features specific to this task. However, since *all* the well performing networks of Kaggle's Diabetic Retinopathy competition are very deep, there is some evidence to suggest that the depth is required.

Not all DR features are recognized in equal quantities. We found that most filters focused on spots—i.e., hemorrhages, microaneurysms and hard exudates. Blood vessel related filters are less abundant, and where they are present it is not possible to tell what specific features of the blood vessels such a filter is interested in. This might be a limitation in the domain knowledge of the author, the visualization method, the network, or the dataset.

We found that most features used by clinicians to diagnose DR are learned by the network. This has both a positive and negative interpretation. Positively, it is comforting that the network has learned to perform the task similarly to how humans perform it (alternatively the network could instead have

found some "nonsensical" patterns that are also related to the target variable and used these for classification instead). Negatively, we have not discovered anything new about Diabetic Retinopathy.

The fact that we were unable to learn about new features relevant to DR shows a limitation with our experimental setup. Humans have predetermined the features, and classified the fundus images according to these features. The network correctly learns these features and this classification scheme. A setup in which the network has to determine features in an unsupervised manner could lead to more knowledge discovery.

We found that characterizing a feature by visualizing the input that produces the strongest activation leads to results that are difficult to interpret. This method was used to visualize the features learned by a Deep Belief Network in previous work [7]. We found that optimal inputs have a significantly stronger activation than samples from the training data. This can not be the case for the Deep Belief Network, since it uses sigmoid activations. It might be that the unbounded nature of the ReLU is the cause of the poor visual results. We tested this hypothesis by starting from a dataset sample and running the optimization method for a few iterations. After inspecting features of which we determined their meaning by visualizing the salient areas, we thought it to be plausible that the same meaning could be derived from the non-converged Activation Maximization visualization, suggesting that the unbounded nature of the ReLU allows the input to become too different from the original input space, causing the results to become incomprehensible.

5. Experiment: Die360

For this experiment we train a variety of neural networks on the task of counting the number of spots on the most prominent face of a die. We generated the data by creating 2-D projections under different rotations of a 3-D die model and called this the Die360 dataset. This is an interesting task; we already know CNNs can handle translations on the input very well because this is hard-coded in the architecture. This task is instead focused on rotations of the input. We also consider how well MLPs perform on this task.

For humans, there seems only one obvious way to solve this task: by counting the number of spots on the most prominent face of the die. However, there is a relationship between the number of spots on the most prominent face and the number of spots on less prominent faces. Such relationships could also be considered by the network.

The dataset is described in Section 5.1. The architectures of the trained neural networks are described Section 5.2. The exemplars and salient areas are visualized in Section 5.3 and Section 5.4. Section 5.5 concludes.

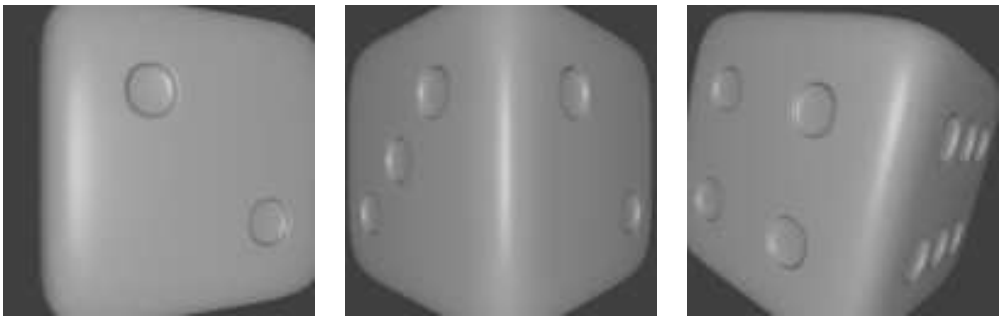
5.1 The Dataset

The dataset was generated by taking a 3-D model of a die and creating 2-D projections of this die under different rotations. We rotate around all three axes, so if we allow one degree rotations, we end up with $360^3 = 46\,656\,000$ different projections. However, we restricted the task to classifying each projection as the number of spots on the "most prominent" face, where the "most prominent" face is defined as the bottom-right face, and only images are included where this bottom-right face is visually most prominent.

To understand the concept of visually most prominent, consider Fig. 5.1. In Fig. 5.1a we see that the bottom-right face is the most visually prominent, so this projection is part of the dataset, with corresponding class label "two". In Fig. 5.1b we see that it has become ambiguous as to which face is most visually prominent, so this projection is discarded. Fig. 5.1c is an example of a projection that is also discarded, since the most prominent face is not the

bottom-right face.

After discarding the ambiguous and not visually prominent samples we are left with 49 200 samples. The samples are greyscale images of size 32×32.



(a) A projection where the most prominent face is the face containing two spots.

(b) A projection that is discarded because the visually most prominent face is ambiguous.

(c) A projection that is discarded because the bottom-right face is not the most visually prominent.

Figure 5.1: Examples of accepted and discarded projections, clarifying the concept of right-most "visually prominent" face. Examples are shown in higher resolution than the actual networks were trained on (128×128 and 32×32 respectively).

5.2 Used Architectures

We train both MLPs and CNNs. The setup for any MLP is always a network consisting of two dense layers, with either 5, 10, or 20 hidden units and the sigmoid non-linearity in the first layer, and 6 output units (one for each class) and the softmax non-linearity in the output layer. We denote these networks with `d<num hidden>`.

The CNNs consist of either 1, 2, 3, or 4 convolution layer and pooling pairs, followed by a fully connected layer with 6 output nodes (one for each class) and the softmax non-linearity. All convolutional layers use 3 filters with filter size 3×3, a stride 1 and *same* padding. All pooling layers use a pool size of 2×2 and a stride 2. We denote these networks with `c<num layers>`.

network	number of parameters	validation accuracy
c1	4 644	$0.9971^{+0.0011}_{-0.0025}$
c2	1 272	$0.9966^{+0.0006}_{-0.0015}$
c3	492	$0.9945^{+0.0023}_{-0.0040}$
c4	360	$0.9764^{+0.0110}_{-0.0124}$
c9	996	$0.9991^{+0.0009}_{-0.0012}$
d5	5 161	$0.9966^{+0.0017}_{-0.0011}$
d10	10 316	$0.9990^{+0.0004}_{-0.0007}$
d20	20 626	$0.9995^{+0.0002}_{-0.0002}$

Table 5.1: Number of parameters, average validation accuracy, and distance from the mean to the minimum and maximum validation accuracy for each network configuration. A convolutional neural network with `<num layers>` convolutional layers is denoted using `c<num layers>`. A two-layer perceptron with `<num hidden>` hidden nodes is denoted using `d<num hidden>`. The reason `c9` has so many parameters is because it alternates three convolutional layers with a pooling layer, while all other convolutional networks alternate each convolutional layer with a pooling layer.

Additionally we train one more CNN variant, this time alternating three convolutional layers (again, 3 filters with filter size 3×3, stride 1, and same padding in every layer), with a pooling layer with a pooling size of 2×2 and stride 2. This alternation is repeated three times (for 9 convolutional layers total). We denote this network with `c9`.

We train our networks on a 80/20% train/validation split using stochastic gradient descent with appropriate training parameters (i.e., momentum and learning rate settings that perform well based on experimental results). This task is fairly simple, as the training and validation set are very similar. All networks score at least 97%. Table 5.1 shows the performance details and the number of parameters of each network.

5.3 Visualizing Exemplars

For each of the convolutional networks we visualized the exemplars. The results are shown in Table 5.2, page 49. For the `c9` network the first feature in the first pooling layer strongly recognizes a spot-like shape. The other networks do not recognize a similar shape until the second pooling layer. This is probably because the `c9` network has already applied 3 convolutional layers at the first pooling layer, while the other networks have applied only 1 convolutional layer at that point. For most exemplars in the deeper layers, the exemplars have zero values at the bottom-right corner, indicating that that part of the input is ignored. We were unable to come up with a suitable explanation. The convolutional layers use *same* padding, suggesting that if the padded input would be ignored, the exemplar should be centered, rather than be top-left aligned.

5.4 Visualizing Salient Areas

We can order the images corresponding to each class by choosing an ordering such that for each image and its successor, the images are projections where the rotational parameters of the 3-D model differ a single degree. For a given class this gives us an ordered sequence of images. We can then visualize the salient area of a feature for each of the images in the sequence, creating an animation.

The visualization can differ significantly between two images that follow in sequence (Fig. 5.2). This suggests that the features are not invariant to rotations. A similar effect is observed for the softmax output, where the posterior that a network predicts for a certain class can be greatly different between two subsequent images (Fig. 5.5).

Much like with the Diabetic Retinopathy dataset we used the Gradient Based method to construct the saliency maps for features in the pooling layers (Figs. 5.3 and 5.4).

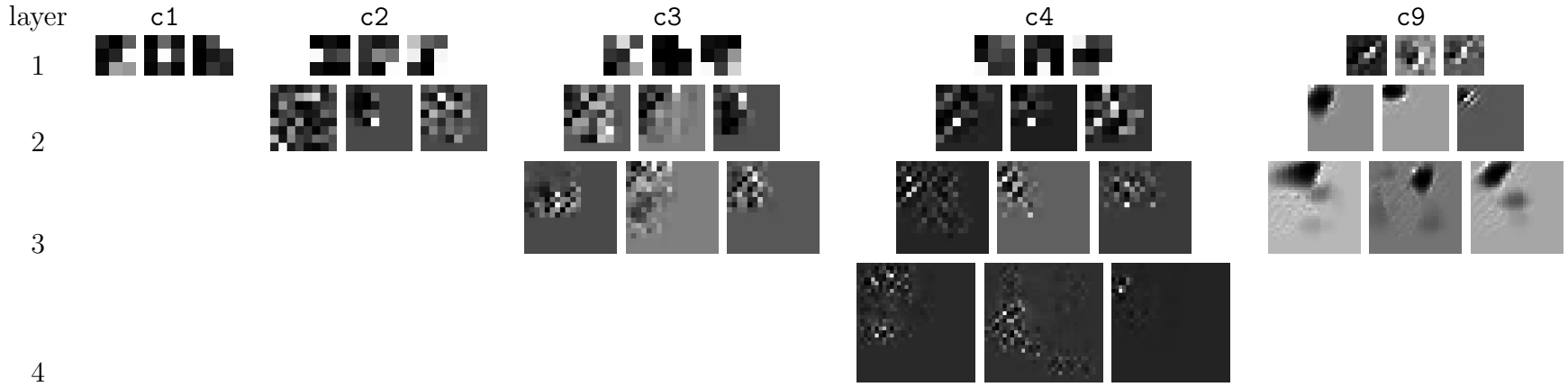


Table 5.2: Exemplars for the features of the pooling layers of different CNN architectures trained on the Die360 dataset. For the `c9` network a pooling layer is preceded by multiple convolutional layers. Because of this the receptive field is bigger, and hence the exemplars are of higher resolution. Three exemplars are shown at each layer, as each convolutional layer has three filters.

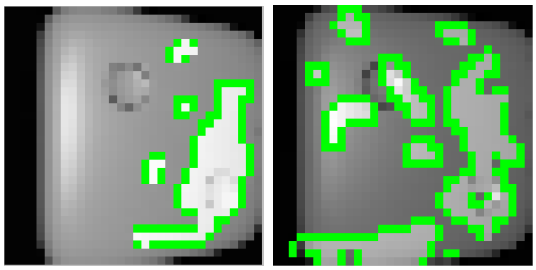
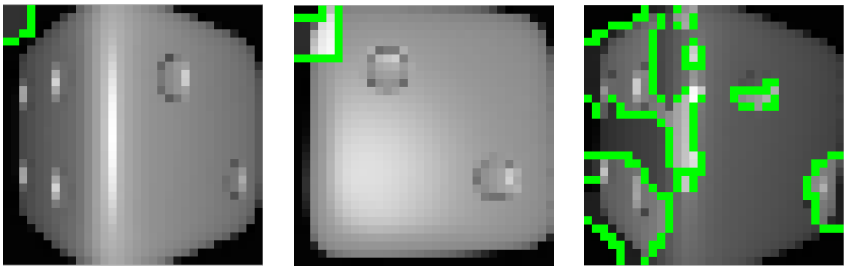
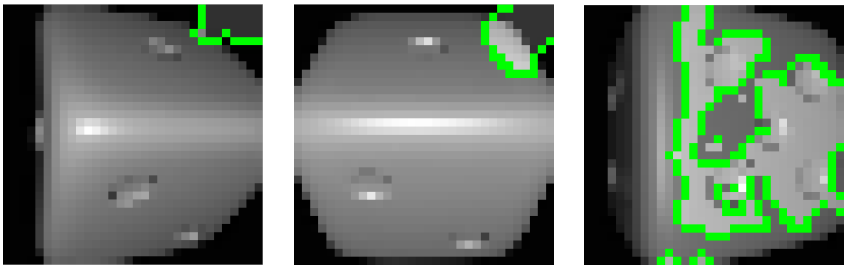


Figure 5.2: Visualization of the same feature on two images where the rotational parameters that generated the images differ by a single degree. A small change in rotation can lead to a large change in the visualization.



(a) Background (b) Face edge (c) Spots (using Sum aggregation)

Figure 5.3: The features in the first pooling layer seem consistent across all networks. Most features are either recognizing the background, or the edge of the die face. Only the `c9` network recognizes spots directly.



(a) Background (b) Face edge (c) Spots (using Sum aggregation)

Figure 5.4: Features still seem mostly focused on background or edge of the die face. In the second pooling layer the `c4` network recognizes spots directly, while the `c9` network no longer does.

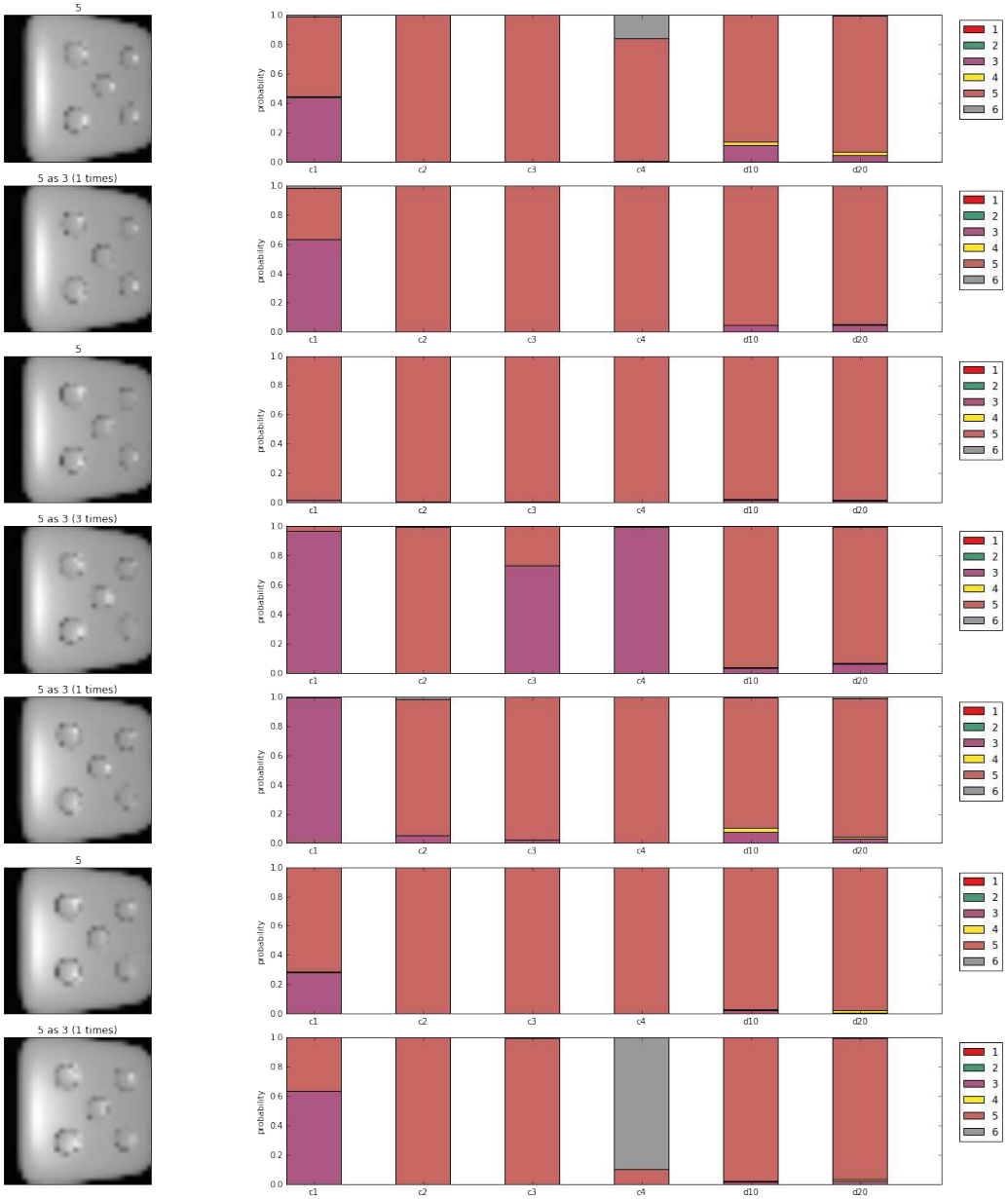


Figure 5.5: Image with the largest disagreement (center, row 4) and a stacked bar plot of the posteriors predicted by each network, together with neighboring images and their respective posteriors. Notice how, despite being visually very similar, a small rotation can lead to a big jump in the network's prediction.

5.5 Conclusion

We found that the Die360 dataset is an easy dataset to learn, presumably because for every image in the dataset there are many images much like it. For example, for nearly every given image there are 6 images for which the rotational parameters by which they were generated differ by a single degree from the rotational parameters used to generate the given image. With a 80/20 train/validation split, it is likely that for every image in the test set, at least one very similar image will end up in the training set.

We found that the networks with the most parameters performed the best on this task (which is to be expected because of the similarity of the validation and training set), and these were typically dense networks. However, we also found that increasing the depth of the convolutional networks was also a very effective method of improving performance, even scoring 100% accuracy on one of the validation folds.

We found that some features are focused on detecting background, possibly because the amount of background can contain information about the rotation of the die (Figs. 5.3a and 5.4a). Other features considered the face edge (Figs. 5.3b and 5.4b), but under different rotational conditions would also recognize spots, or follow the highlighting caused by the lamp in the 3-D scene in which the die was rendered. Spot detectors also occurred, but would be unable to distinguish between the spots on the most prominent face and the spots on other faces (Figs. 5.3c and 5.4c).

6. Summary and Conclusion

This work tries to answer the question: "How to visualize the features learned by Convolutional Neural Networks (CNNs)?". We considered two different approaches: 1) visualizing exemplars, and 2) visualizing salient areas in the original input. The main contributions of this work are: 1) extending a method for computing salient areas in the input image as proposed by Simonyan, Vedaldi, and Zisserman in [25] to be applicable to convolutional layers; 2) showing that—by using this extension—a CNN trained to detect Diabetic Retinopathy (DR) uses features that are similar to those used by trained clinicians to diagnose the disease; 3) publishing source code for generating the visualizations of CNNs [16].

We applied the different visualization approaches on networks trained on two different tasks: classifying DR and classifying the number of dots on the most prominent face of a die. For the DR task we used a CNN provided by the team that finished 5th place in the corresponding Kaggle competition. For the Die360 task we trained a variety of different networks on a synthetic dataset.

The literature suggests two methods for visualizing exemplars: Activation Maximization and Deconvolutional Networks. Activation Maximization uses gradient ascent to find inputs that most strongly activate a given feature, while Deconvolutional Networks try to reverse the process of a CNN: rather than projecting an input into some different representation, they try to project the new representation back into the input space. Due to the limitations of this method we did not apply it.

We did not have success with applying the Activation Maximization method. This method was suggested to visualize the features of Deep Belief Networks. These networks use sigmoid activation functions: a function that quickly saturates, so gradient ascent converges quickly. We observed a very large difference between the magnitude of the activations on the training data, and the magnitude of the activations on the optimized input. We hypothesize that because the rectified linear unit is non-saturating the optimized input can become too different from the training set input to be interpretable. This hypothesis was in part confirmed by initializing the optimization from a training sample, and then stopping early in the optimization process. In

that case, it seemed easier to distinguish what kind of feature was recognized.

We extended the Gradient Based method to also be applicable to intermediate layers of a CNN, by using an aggregation function on the feature map activations. We found that using the maximum or the sum as aggregation function worked well. The maximum aggregation function was less distracting, while the sum aggregation function gave more detailed results. We found that visually the results were very similar to the Occlusion method, but the Gradient Based method shows more details and scales much better to larger inputs.

By visualizing the salient areas we found that on the DR problem the features learned by the network become very similar to the features used by clinicians to diagnose DR as the depth of the network increases. For example, of the 50 inspected features in the third pooling layer a total of 27 features could not be classified as one of the features used by clinicians to diagnose DR, while in the fifth and final pooling layer only 17 out of 50 could not be classified as such.

6.1 Discussion

This work focused on different visualization methods. The visualizations produced often seemed sensible, because we found what we were looking for. Unfortunately this work lacks a good methodology to objectively evaluate the visualizations, nor were we able to provide a formal argument why one method is better than the other. It also seems that without prior knowledge of the expected features, extracting novel information based on the visualizations of a CNN is still a daunting task, even with the methods presented in this work.

6.2 Applications

By visualizing the features the classification process of the CNN becomes slightly more transparent. For example, the CNN trained on the DR task may be used to assist optometrists in grading the severity of DR. But using

the methods in this work, we could, instead of simply reporting classification confidences, highlight areas that the network considers important. This way the network might make the optometrist aware of aspects that he had not considered yet.

6.3 Future Work

We did not compare networks of different depths on the DR task, although we found that the interesting features do not emerge until the last and penultimate pooling layer. Future work could try to find CNN architectures that have different depth, but perform as well or better as the network architecture used in this work. By using a more shallow network one could investigate whether depth is a requirement for interesting features to emerge, or that the more interesting features do not emerge in the lower layers of deeper networks because it is simply not required to learn interesting features until the final layers. By using a deeper network one could investigate if more difficult to recognize features, such as blood vessels, become more prevalent.

In this work we looked at the DR problem. The CNN had to learn to grade DR according to some standard invented by humans. The relevant patterns were predetermined by humans and encoded in a labeled dataset. Because of this it is not very surprising that the CNN indeed learns features used by humans to grade. Future work could look at a task where the class is an objective measurement, for example the quality of sight. We could then try to learn what visual features seem related to poor sight, possibly suggesting new methods to improve eye sight, or to prevent deterioration of sight.

Additionally future work could look at unsupervised neural networks, and visualize the features learned by these networks, hopefully improving our understanding of the unsupervised task. This seems the most challenging future work; a specific task gives a certain context to evaluate the visualizations in. This context is missing for unsupervised learning, so the results might be even more difficult to interpret.

Bibliography

[1] Rodrigo Benenson. "Classification Datasets Results". URL: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (visited on 06/17/2016).

[2] Robert Bogucki. "Diagnosing Diabetic Retinopathy with Deep Learning". 2015. URL: http://deepsense.io/diagnosing-diabetic-retinopathy-with-deep-learning/ (visited on 05/17/2016).

[3] Dan Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. "Flexible, High Performance Convolutional Neural Networks for Image Classification". *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 22. 2011, pp. 1237–1242.

[4] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-Column Deep Neural Networks for Image Classification". *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2012, pp. 3642–3649.

[5] Jeffrey De Fauw. "Detecting Diabetic Retinopathy in Eye Images". 2015. URL: http://jeffreydf.github.io/diabetic-retinopathy-detection/ (visited on 05/17/2016).

[6] Paulo Diniz, Eduardo da Silva, and Sergio Netto. "Digital Signal Processing". 2010, pp. 397–422. URL: http://dx.doi.org/10.1017/CBO9780511781667.

[7] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Visualizing Higher-Layer Features of a Deep Network". *University of Montreal* 1341 (2009).

[8] Kunihiko Fukushima. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". *Biological Cybernetics* 36 (1980), pp. 193–202.

[9] Ben Graham. "Kaggle Diabetic Retinopathy Detection Competition Report". 2015. URL: https://github.com/btgraham/SparseConvNet/blob/kaggle_Diabetic_Retinopathy_competition/competitionreport.pdf (visited on 05/17/2016).

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on Ima-

geNet Classification". *CoRR* abs/1502.01852 (2015). URL: http://arxiv.org/abs/1502.01852.

[11] Ahmed Al-Hinai, Mohammed Al-Abri, and Rayah Al-Hajri. "Diabetic Papillopathy with Macular Edema Treated with Intravitreal Bevacizumab". *Oman Journal of Ophthalmology* 4 (2011), pp. 135–138. DOI: 10.4103/0974-620X.91270. URL: http://doi.org/10.4103/0974-620X.91270.

[12] Danny Hope. "Photograph of the Retina of the Human Eye, with Overlay Diagrams Showing the Positions and Sizes of the Macula, Fovea, and Optic Disc". 2014. URL: https://en.wikipedia.org/wiki/Macula_of_retina#/media/File:Macula.svg.

[13] Ian Goodfellow and David Warde-farley and Mehdi Mirza and Aaron Courville and Yoshua Bengio. "Maxout Networks". *International Conference on Machine Learning (ICML)* 28 (2013), pp. 1319–1327.

[14] Kaggle Inc. "Diabetic Retinopathy Detection". 2015. URL: https://www.kaggle.com/c/diabetic-retinopathy-detection/ (visited on 05/17/2016).

[15] Kaggle Inc. "Winning Models from @CHCFNews Diabetic Retinopathy Comp are on par with Human Performance!" 2015. URL: https://twitter.com/kaggle/status/626148867961147392 (visited on 05/17/2016).

[16] Jan Kalmeijer. "CNN-Vizlib — A Python Package for Visualizing the Features Learned by Convolutional Neural Networks". 2016. URL: https://github.com/jkalmeij/cnn-vizlib (visited on 06/22/2016).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "Imagenet Classification with Deep Convolutional Neural Networks". *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[18] LISA lab. "Conv — Ops for Convolutional Neural Nets — Theano 0.8.2 Documentation". URL: http://deeplearning.net/software/theano/library/tensor/nnet/conv.html#theano.tensor.nnet.conv.conv2d (visited on 06/07/2016).

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE* 86 (1998), pp. 2278–2324.

[20] MathWorks. "2-D Convolution — MATLAB conv2 — MathWorks Benelux". URL: http://nl.mathworks.com/help/matlab/ref/conv2.html (visited on 06/07/2016).

[21] Team o_O. "Kaggle Diabetic Retinopathy Detection". 2015. URL: https://github.com/sveitser/kaggle_diabetic (visited on 05/17/2016).

[22] World Health Organization. "Priority Eye Diseases". URL: http://www.who.int/blindness/causes/priority/en/index5.html (visited on 05/20/2016).

[23] The Retina Reference. "Severe Diabetic Retinopathy — Venous Beading, Neovascularization Arising from A Retinal Venule, and Preretinal Hemorrhag". URL: http://www.retinareference.com/diseases/74ef99876961fce8/images/9a29e58ff5/image.jpg.

[24] SciPy. "scipy.signal.convolve2d — SciPy v0.17.1 Reference Guide". URL: http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html (visited on 06/07/2016).

[25] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". *CoRR* abs/1312.6034 (2013). URL: http://arxiv.org/abs/1312.6034.

[26] Lihteh Wu, Priscilla Fernandez-Loaiza, Johanna Sauma, Erick Hernandez-Bogantes, and Marissé Masis. "Classification of Diabetic Retinopathy and Diabetic Macular Edema". *World J Diabetes* 4 (2013), pp. 290–294. DOI: 10.4239/wjd.v4.i6.290. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3874488/.

[27] Jun Xu, John Dunavent, and Raghu Kainkaryam (Team Reformed Gamblers). "Summary of our Solution to the Kaggle Diabetic Retinopathy Detection Competition". 2015. URL: https://www.kaggle.com/blobs/download/forum-message-attachment-files/2815/Team_Reformed_Gamblers_Solution_Summary_v2.pdf (visited on 05/17/2016).

[28] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". *CoRR* abs/1311.2901 (2013). URL: http://arxiv.org/abs/1311.2901.