University of Groningen

Dept. of Artificial Intelligence

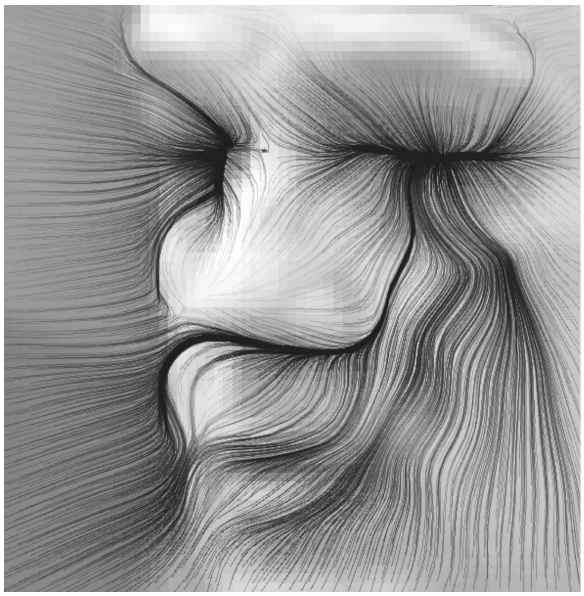
Master Thesis

Using Guided Autoencoders on Face Recognition

Author:
M.F. Stollenga
s1539906
m.stollenga@gmail.com

Supervisor:
dr. M.A. Wiering

Second Supervisor:
prof. dr. L.R.B. Schomaker



May 10, 2011

Abstract

In this master thesis we will create guided autoencoders (GAE) and apply them to face recognition. GAEs are agents that interact with images, using a novel combination of autoencoders and reinforcement learning.

They perceive part of an image through a window, use an autoencoder to encode it, and react to what they see by moving the window. GAEs are trained to find and encode specific parts of the face – in our case the eyes, nose and mouth. We use the LFWC (cropped Labeled Faces in the Wild) dataset which is very varied and has many uncontrolled variables.

We train GAEs using the CACLA reinforcement learning algorithm which can deal with continuous states and actions. To create a state, GAEs evaluate their separately trained autoencoder on what is visible through their window. The resulting state guides their actions. We show that GAEs are able to navigate the complex landscapes of the face images using only local information, which is quite remarkable.

The experiments show that GAEs can find their goals if they are initialized relatively close to their goal. If we add position information to the encodings, the performance increases greatly. We also compare deep stacked autoencoders and shallow autoencoders. Surprisingly, deep GAEs do not outperform shallow GAEs on this task. The GAEs are finally used to classify the gender of faces and whether a person is smiling or not. They are able to do classification, but do not rival state-of-the-art systems. Their flexibility however allows them to be extended easily to improve performance.

In summary, the GAEs are currently not able to perform better on classification than the state of the art. However, their ability to navigate complex images and their flexibility makes them promising tools for face recognition and computer vision.

Contents

Chapter 1

Introduction

Imagine looking at a car in the street that is obscured by a tree. Somehow you are able to recognize the car from the complex input presented to you, with absolute certainty. How does the brain go about processing this input? Although this riddle has not been answered yet, and is in fact quite a mystery, we do observe two important ways of dealing with input by the brain. Firstly, research suggests that our brain uses deep structures of many layers to process the incoming input and create higher level representations of them. These deep representations allow the brain to take all information it perceives at a lower level and combine it. This allows the brain to reconstruct the obscured car by taking the low level information about the obscured car and use higher level representations to combine this information and recognize the car, regardless of the missing information.

Secondly, instead of passively receiving input, the brain interacts heavily with the world it perceives. If for example a part of the car we are looking at is obscured by a tree, the eyes quickly move their focus to the parts that are visible, actively trying to somehow make sense of the input. By guiding actions that change perception, the brain is able to increase the quality of the input it gets by using the input itself, getting the most out of the information.

The main goal of this thesis is to create a face-recognition algorithm that can form higher level representations and allows for interaction with the data. We will build guided autoencoders (GAE) that try to guide themselves to parts of the face that they are trained to encode. The guiding will be facilitated by constantly building representations and acting on them. We will finally test GAEs in a classification experiment.

1.1 Introduction

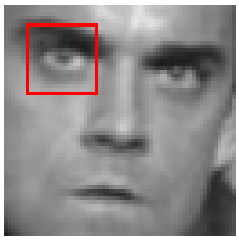
The field of Artificial Intelligence has always been focused on recreating the amazing learning capabilities of human beings. The ability to learn is seen as the major component of our intelligence. The field of machine learning (ML) is particularly interested in automated learning. ML sets out to find practical algorithms that can be run on computers and can predict previously unseen data by first being presented (and learn from) other data. ML algorithms have had many great practical successes (e.g. the Google Pagerank algorithm is a type of ML-algorithm) and can be seen as very successful in that domain. However these algorithms do not directly interact with or form deep representations of the input data, but rather use a shallow representation of the data. The goal of this research is to create an algorithm that is able to do this, by combining existing ML-frameworks with a reinforcement learning (RL) algorithm.

Figure 1.1: The face images of the LFWC database show a lot of variation. The lighting and pose can be very different, and faces can even be obscured by hands or a pair of glasses.

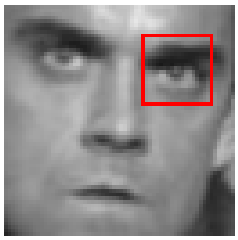
1.2 Face Recognition

The goal of the system built in this thesis is to use interaction and deep representations to deal with data. We hypothesize that such a system can deal with complex data. There is no need for such a system if we use a simple dataset, so we looked for a dataset that has a highly varied collection of faces. We chose to use the LFWC dataset, which is a cropped version of the "Labeled Faces in the Wild" (LFW) dataset [Huang et al., 2007]. It consists of about 13000 images of faces of 64 by 64 pixels, taken from magazines and the Internet. They are cropped in such a way that background information is minimized. The dataset is specifically designed to be difficult and has many uncontrolled variances. Figure 1.1 shows examples of the varied dataset, showing the variances in pose, expression and same lighting conditions. There is still human selection in the faces, as they are photos taken by photographers that try to capture the face well.

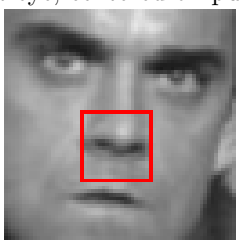
But compared to most other face-datasets, this dataset is complex. A comparison of face databases in [Tolba et al., 2005] shows that most databases have a limited number of people in it. The images are almost always taken with exactly the same setup – same camera and lighting conditions. And if there are variables that are changed, they are changed very carefully. For example, when changing the pose in a dataset, care is taken to vary the pose to exact angles with respect to the camera. All of this contributes to create a distribution of samples that are created by a process where the subject cooperates and follows exact instructions from a protocol. The control over the process shows up in the distribution of samples – if it wouldn't there would be no need for a protocol – and results in an artificial dataset. Solving face recognition for such a dataset hides away problems we encounter in real world face recognition.



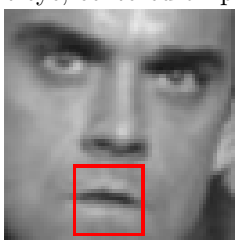
left eye, centered on pupil



right eye, centered on pupil



nose, centered on the bottom of the nose



mouth, centered in the middle of the mouth

Figure 1.2: The eyes, the nose and the mouth are annotated by hand.

1.2.1 Normalization

The pictures are gray-scale images. Every pixel has a value in the range of [0..255]. These values are too high for the neural networks that we will be using, so we need to normalize the images.

These neural networks can output values between -1.0 and 1.0. To keep the pixel values in this range we normalize the images such that their mean value is around $\mu = 0.0$ and a standard deviation of $\sigma = 0.3$. If the data is modeled well by a normal distribution, 99.8% of the pixel values should have a value in the range of $[-0.9..0.9]$.

1.2.2 Annotation

Our algorithm needs to know the position of the eyes, nose and mouth to be trained correctly. We annotated these parts by hand in a subset of 1300 face-images as follows:

eyes Both eyes are annotated at the position of the pupil.

nose The nose is annotated at the bottom of the nose. This position of the nose is more stable than e.g. the tip of the nose under different poses.

mouth The mouth is annotated at the middle of the mouth.

The resulting annotated dataset is used in the rest of our thesis. In section 2.3.2 we will extract patches of 20 by 20 pixels to create a new dataset to create models of the individual parts. This is shown in Figure 1.2, where the rectangles show the size of the patches taken.

1.3 Concepts and Systems

To build and test the GAEs we use several systems and concepts. In this section we describe them quickly. More detailed descriptions are given in the next chapters.

1.3.1 Machine Learning

First we will define the formal classification task. Let $x^i \in \mathbb{R}^m$ be an input vector of length m and let $y^i \in C$ be its corresponding class from possible classes C. We then have a dataset of input/output pairs: $D = \{(x^1, y^1), (x^2, y^2), ..., (x^n, y^n)\}$. We want to estimate a function $f : \mathbb{R}^m \to C$ that has a maximum accuracy on unseen test examples. The goal of the learning algorithm is to correctly classify new inputs, the system has never seen before.

1.3.2 Deep Learning

The bulk of current machine learning algorithms use a shallow representation to model the data. A shallow representation is characterized by an evaluation function that does not use many mathematical operations to distinguish different classes. For example the main calculation in the classification function for a Support Vector Machine (SVM) with a radial basis function (a commonly used setup) is: $\exp(-\gamma\|x_i - x_j\|^2)$. The formula in itself is not important, but the key observation is that the number of calculations made are small. In contrast, deep learning uses deep nets of (non-linear) calculations (for example neural networks with several layers) that allow for more complex calculations.

Shallow learning algorithms use a shallow representation because it is believed that the representation in the algorithm should be as simple as possible. This prevents overfitting according to Occam's razor, but more importantly it is argued that the real underlying function the algorithm tries to approximate, actually is such a shallow function. In [Bengio, 2009] a case is made against these ways of thinking, arguing that complex (deep) learning systems do not necessarily overfit and misrepresent the real data. In fact the paper describes theoretically how a problem gets exponentially more difficult for a shallow representation, if the real problem is just one layer deeper than the representation. It should also be noted that the human brain, which is the best learning system we know, uses deep representations (e.g. the many layers in our visual system).

Still one of the main problems in deep learning algorithms is how to learn them effectively. Simple approaches to learning deep systems often end up in local minima that under-perform compared to any shallow learning algorithm. But Hinton and Salakhutdinov [2006] found a way to learn deep networks efficiently by stacking autoencoders. This will be explained in section 3.2.1.

1.3.3 Feed-forward Neural Networks

Neural networks are omnipresent in the field of Artificial Intelligence. In short they are non-linear functions that map from a (possibly) multidimensional space to another (possibly) multidimensional space. These functions are defined by a set of weight-values, which define the mapping of the function. These weight-values form the flexible part of the neural network and define its behaviour.

Backpropagation We can train a neural network by finding the right values for the weights of the network. There are many ways to do this, but the most famous and widely used learning algorithm is called backpropagation [Rumelhart et al., 1986].

The idea is to present the neural network with a training example of which the desired output is known. Then we evaluate the network and calculate the difference between the desired output and the actual output of the network: the error. This error can be defined as a function of the weight-values of the network. Now comes the trick: we can calculate the gradient of this function. This tells us which 'direction' we have to alter the weight-values to maximize the error. By reversing this vector, we get the direction to minimize the error.

This gives us an algorithm to learn the right values for the weights:

1 Present the neural network with an example.

2 Evaluate the network and calculate the error: the difference between the desired and actual output.

3 Calculate the gradient of the error toward the weight values. The calculation of the gradient will be described in section 2.1.2.1.

4 Adjust the weight values in the opposite direction of the gradient, proportional to a certain learning rate.

5 Repeat until the desired effect is achieved.

1.3.4 Autoencoders

Autoencoders are a special kind of artificial neural networks. Their input- and output-layer have the same size and there is a smaller hidden layer in between. The autoencoder is presented a pattern and its goal is to reconstruct that pattern in the output – learning to map the input to itself (the 'auto' part in 'autoencoders'). The network is evaluated by evaluating the input through the hidden layer to the output layers. Because the goal is to reconstruct the input-vector in the output layer as well as possible, the network is back-propagated with the error between the reconstruction and the original pattern. The smaller sized hidden layer has to represent the larger input data. Therefore, the system learns a compressed representation of the data. The activation of the hidden layer provides a compressed representation of the data, which we call the *encoding* of the data. By using non-linear artificial neurons, e.g. sigmoid functions, an autoencoder can perform essentially a non-linear principal component analysis [Kramer, 1991].

Denoising To improve robustness of the autoencoders towards noisy input data, we can add noise to the pattern before presenting it to the network. The goal of the network is again to reconstruct the input data, but now the input data is corrupted by randomly adding noise before presenting it to the network. This process is called denoising and has been shown to increase robustness against noisy inputs [Vincent et al., 2008].

Stacking We can also put autoencoders on top of each other. This creates stacked autoencoders (SAE) that can consist of many layers. This has proved to be a successful strategy to learn deep networks [Hinton and Salakhutdinov, 2006]. We will use SAEs in this thesis to try to create deep representations of the data.

How to train SAEs will be explained in section 3.2.1. In short, first a single layer autoencoder is trained, and then the encoding of that layer serves as input to a new autoencoder, adding a layer. This process is repeated until the desired number of layers is reached.

Window GAEs perceive only part of the image. The perception of the GAE is modeled using a rectangular window. The window has a center, a width and a height. Looking through this window consists of taking the image that is contained by the window. The goal of a GAE is to move this window to the part they are trained to encode.

1.3.5 Reinforcement Learning

GAEs can be looked at as being agents that can perform actions – namely moving the window. To learn the actions of these agents we use a reinforcement learning (RL) algorithm [Kaelbling et al., 1996, Sutton and Barto, 1998]. RL algorithms can learn an agent to act in such a way that it acquires as much reward as possible. By defining the reward function in the right way, we can make the agent learn what we want.

We use the CACLA algorithm [van Hasselt and Wiering, 2007] to deal with continuous state and action spaces. It will be described in section 4.1.2.

1.3.6 Interaction with Data

Humans do not passively wait for information in our world to enter their brains. We interact with the information around us. This is true for our face-recognition too. We do not just 'receive' a single image of a face and go: 'aha this person looks happy'. Instead we move our eyes very quickly over interesting areas in a face. This is done mostly unconsciously using movements called saccades.

We actually perform several saccades per second. We do not receive any visual information when our eyes are performing these little movements. Only at the moments that the eyes are completely still, the visual information enters the brain. This allows our brain to specifically control what information enters it, in small amounts.

1.3.7 Notation

At this point we would like to describe the notation used in this thesis:

Matrix and vector elements We use $\vec{v}_i$ to denote element i from vector $\vec{v}$. We use $\mathbf{M}_{ij}$ to denote the element in row i and column j of matrix $\mathbf{M}$. To select row i from the matrix $\mathbf{M}$ we use $\mathbf{M}_i$.

Functions over Vectors We allow for applying functions $f(x) : \mathbb{R} \to \mathbb{R}$ over a vector $\vec{v}$. In that case the result is a new vector, where we evaluate function $f(x)$ element-wise: $\vec{v'} = f(\vec{v})$ where $\vec{v_i'} = f(\vec{v_i})$ for all i.

Updating The algorithms in this thesis need to update parameters. We use the following notation:

$$x \xleftarrow{\alpha} y \tag{1.1}$$

to denote:

$$x_{t+1} = (1 - \alpha)x_t + \alpha y_t \tag{1.2}$$

where x_t is the variable to update at time t, y is the value that variable x should learn, at time t, and α is a factor that controls the learning speed.

Example Say we have a variable x_0 which has some value, and we update it repeatedly with value y. If $\alpha = 1$ the variable x takes on value y in one time step. For $0 < \alpha < 1$ the variable moves towards y where the distance $|x_t - y|$ reduces proportional to $(1 - \alpha)^t$. To illustrate, a $\alpha = 0.01$ means that after 100 steps the variable x will have only moved about 63% of the distance from x_0 to value y.

Constants and Factors The experiments in this thesis use learning algorithms that use constants and factors to define how they behave. Examples of such factors are the learning rate or the portion of data used in a training algorithm. To keep things simple we use two sets of variables λ and η to denote factors and constants respectively. Factors are the variables that define behaviour at a lower level, such as the learning rate. Constants are variables that impact the initialization of parts of the algorithm. It is easier conceptually to keep these two apart.

We denote one of the factors, for example the learning rate, as follows: $\lambda_{learningrate}$. Same for the constants, for example the portion of dataset used for training: $\eta_{portion}$. In some cases the variables of the learning algorithm have standardized symbols, in which case we make an exception and use them instead (e.g. the variables of the reinforcement learning algorithm).

1.4 Guided Autoencoders and Research Question

In this thesis we will build guided autoencoders (GAE) (section 4.1.2 will give a detailed explanation of GAEs). GAEs view the image through a window and use autoencoders to create representations. They use that representation to guide themselves. GAEs move around in the image, create representations and act based on these representations – they interact with the data. This encompasses the main goal of this thesis.

Research Question Our research question is: Can guided autoencoders be used successfully to create representations of faces and classify them?

1.5 Outline

In chapter 2 we will describe the training of autoencoders and experiment with them. We will train autoencoders of the full face and then on parts of the face. We show that training on parts of the face helps to retain more detail. Then in chapter 3 we will compare training deep stacked autoencoders and shallow autoencoders on parts of the face. We show that for this problem, there is no difference in performance between shallow and deep autoencoders. Then in chapter 4 we create the framework for guided autoencoders and experiment with them. We show that GAEs are able to find their respective parts, provided that they are initialized close by. Then in chapter 5 we turn GAEs to a classification problem where they have to classify the gender of faces and guess if a person is smiling. We show that GAEs can do classification, but do not rival other classification systems at the moment. Then in chapter 6 we will give the conclusions of this thesis and discuss how to improve and extend GAEs – specifically to improve their stability and performance and create more superior classification systems.

Chapter 2

Encoding Faces

2.1 Introduction

What happens when we try to recognize a face? We point our gaze toward the face and through our eyes the image is projected on our retina. The retina sends the information to parts of the brain through the optic nerve. There it is processed to a representation that makes it 'understandable' to our brain. This representation is the key to our amazing ability to understand and compare faces.

How the face actually is represented in the brain is still studied and remains mainly a mystery. From experience we do know that the representation is very robust to the varieties of faces we encounter in everyday life. The highly dimensional and highly varying input from the world is processed to a lower dimensional description.

Outline In this chapter we will create a system that can process an image of a face, represent it in a lower dimensional vector and reconstruct it again. We will first explain how to train autoencoders. Then in the first experiment we will train an autoencoder to encode faces. We will show that this works well, but a lot of detail of the image is lost. To retain more details, in the next experiment, we will train local autoencoders to encode parts of the face.

2.1.1 Face Recognition

In this thesis we build a system that does face recognition. Face recognition should not be confused with face detection. Face detection concerns finding faces in images amongst many other objects, whereas face recognition concerns what the face looks like after it has been found. So face detection focuses on the *where*-question and face recognition on the *who*/*what*-question.

There are two approaches to study face recognition systems. Either we build a system ourselves in the computer, based on computer vision knowledge. This is called automatic face recognition. By looking at the performance of the system we can figure out which techniques work well and which do not. By figuring out which techniques work, we get an understanding of face recognition.

Another approach is studying the existing and best working face recognition system, the human brain. Studying the brain can give us fruitful insights in approaches that can then be used in automatic face recognition. The human brain is however not as accessible as a computer. Studying this system can only be done using carefully designed experiments or using brain-scans with very coarse resolutions relative to the size of neurons.

2.1.1.1 Automatic Face Recognition

There are good surveys on face-recognition research [Tolba et al., 2005, Zhao et al., 2003]. A distinction is made in [Zhao et al., 2003] between holistic and feature based methods:

Holistic methods, such as principle component analysis, take in the complete face image as input and map it into a lower dimension. They create this lower dimension by finding typical faces (e.g. eigenfaces) and describe face images as a (linear) combination of these typical faces. Using the full face as input allows these methods to use all information of the face – from its general structure to the type of eyes – and model relationships between that information. However they have no natural way to deal with changing positions or orientations of the face, other than create new typical faces for every condition. The number of typical faces needed increases exponentially with the number of variables that can change in a face, giving rise to scaling problems.

Feature based methods first look at a face image locally to extract features that describe the image. These low level features are aggregated to one representation that is then used for classification. This method makes it easier to be invariant to variability because the aggregation step provides a natural generalization. However, these methods have difficulty combining information from different parts of the face making it harder to model the global structure of the face. The method also requires more steps and is therefore more complex and less restrictive: The type of features used, the size of the features, the aggregation step – all can be done in many different ways allowing for more creativity in application but also making the search space of models bigger.

2.1.1.2 Human Face Recognition

We, humans, are all experts in face recognition. No system in the world, digital or other, can recognize faces better than us. It seems that no matter how transformed and obscured a face is, as long as there is enough information left, we can recognize it.

Human face recognition has been studied by psychologists and neuroscientists for many years. Most of this research is focused however on immediate recognition [Serre et al., 2007]. Immediate recognition comprises recognition tasks that can be evaluated in about $150ms$. This assures that the subject does not have time to use complex feedback processes to come to a result – they do not even have time to move their eyes in response to what they see. The experimenter gets very clean data of the low-level recognition process, with the cost of leaving out the more complex and perhaps interesting interactions in the brain.

Still we can learn from this research. In a paper by [Sinha et al., 2006], which has the subtitle "Nineteen Results all Computer Vision Researchers Should Know About", important results from face recognition are shown. For example, humans can recognize faces in extremely low-resolution images. Because the images we are using are 64 by 64 pixels – low resolution – it is good to know that at least humans can recognize the faces.

In [Barbeau et al., 2007] experiments show that humans are very fast in a face recognition task where the subject has to decide whether a presented face is a famous person or not. It is so fast that the researchers suggest that face-recognition is a one-way process that does not interact with the data. This is against the ideas in this thesis to increase the amount of interaction with data. However it should be noted that the images were selected in such a way that they were not very confusing. Also, the task of classifying someone as famous or not famous, is known to be very easy for humans. We expect that interaction is needed when images start to get confusing and tasks are not so clearly defined. It is at those times that we have to look at an image again in response to what we already see, in order to understand the image correctly.

2.1.2 Autoencoders

Autoencoders are a special kind of neural networks that try to recreate the pattern that is given as its input. The pattern has to pass through the hidden layers of the network before it is reconstructed at the output. The autoencoder has hidden layers that are smaller than the size of the input. This forces the neural network to represent the input in a lower dimension, i.e. compress it. This principle of compression is the basis for training autoencoders.

2.1.2.1 Multilayer Feed-forward Neural Networks

Here we formalize multilayer neural networks that will be used throughout this thesis. We denote the input to the network as $\vec{x}$ and the output of the network as $\vec{o}$.

The neural network consists of N layers and each layer is represented by a weight-matrix $\mathbf{W}^i$ and a bias vector $\vec{b}^i$, where i denotes the layer to which the matrix or bias belong to. The input layer has index $i = 0$, the output layer has index $i = N - 1$. If $n > 2$ then there are several layers in between the output and input layer, called hidden layers.

The activation $\vec{h}^i$ of every layer depends on the activation of the layer directly below it and is calculated as follows:

$$\vec{h^{i+1}}(\vec{b}^i, \mathbf{W}^i, \vec{h}^i) = \tanh\left(\vec{b}^i + \mathbf{W}^i\vec{h}^i\right) \tag{2.1}$$

where $\vec{h}^i$ is the input activation and $\vec{h}^{i+1}$ the output activation of layer i. The tanh function is a sigmoid function that makes the neural network non-linear. It can be replaced with other sigmoid functions, but in this research we will stick to the tanh function.

Equation 2.1 calculates the full output vector in one go. We also want to be able to calculate only one of the output values, in which case we write:

$$\vec{h_j^{i+1}}(\vec{b}^i_j, \mathbf{W}^i_j, \vec{h}^i) = \tanh\left(\vec{b}^i_j + \mathbf{W}^i_j\vec{h}^i\right) \tag{2.2}$$

where j is the index of the output.

We calculate the output activation $\vec{o}$ from the input activation $\vec{x}$ as follows:

1. Define the input activation for the lowest layer $\vec{h}^0 = x$.
2. Calculate the subsequent activations $\vec{h}^{1..N}$ according to formula (2.1).
3. Define the output as the last activation vector $\vec{o} = \vec{h}^N$.

Note that there are $N + 1$ activation vectors, one more than the number of layers, because we need one activation vector to define the input.

Backpropagation Now we know how to evaluate neural networks, we want to train them. A well known method to do this, and the method that will be used in this thesis, is backpropagation. It is a method that uses gradient decent to minimize a defined error measure.

Say we have an example from the dataset $(\vec{x}, \vec{y})$ where $\vec{x}$ is the input and $\vec{y}$ the corresponding desired output. Our neural network gives $\vec{o}$ as output. We define an error metric between the desired output and actual output that we want to minimize:

$$\min_{\theta} E(\vec{y}, \vec{x}, \Theta) \tag{2.3}$$

$$E(\vec{y}, \vec{x}, \Theta) = \frac{1}{2}\|f(\vec{x}, \Theta) - \vec{y}\|^2 \tag{2.4}$$

where f is the function to be trained, parametrized by Θ. In our case f is a neural network parametrized by its weight matrices and biases $\Theta = \{\mathbf{W}^{0..N-1}, \vec{b}^{0..N-1}\}$.

The next step is to calculate the gradient of this error $\frac{\partial E}{\partial \Theta}$ with respect to the parameters of the network. Using the chain-rule we can easily calculate the gradient towards the parameters of the output layer:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{2.5}$$

$$\frac{\partial \frac{1}{2}\|f(\vec{x},\Theta) - \vec{y}\|^2}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{2.6}$$

$$(\vec{o}_i - \vec{y}_i)\frac{\partial f(\vec{x},\Theta)}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{2.7}$$

$$(\vec{o}_i - \vec{y}_i)\frac{\partial \vec{h_i^N}}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{2.8}$$

$$(\vec{o}_i - \vec{y}_i)[1 - \tanh^2(\vec{b_i}^{N-1} + \mathbf{W}_i^{N-1}\vec{h}^{N-1})]\vec{h}_j^{N-1} \tag{2.9}$$

where $\mathbf{W}_{ij}$ is the element from matrix $\mathbf{W}$ on row i and column j, $\mathbf{W}_i$ is the row i from matrix $\mathbf{W}$.

These formulas flow out nicely because the weight $\mathbf{W}_{ij}^{N-1}$ only contributes to one element of the output $\vec{o}_j$. If we want to derive the gradient for a weight in the next layer $\mathbf{W}_{ij}^{N-2}$ the gradient becomes rather unwieldy because this weight contributes to all elements in $\vec{o}$:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{N-2}} = \tag{2.10}$$

$$(\vec{o}_1 - \vec{y}_1)\frac{\partial h_1^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} + (\vec{o}_2 - \vec{y}_2)\frac{\partial h_2^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} + \cdots + (\vec{o}_O - \vec{y}_O)\frac{\partial h_O^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} = \tag{2.11}$$

$$\left[(\vec{o}_1 - \vec{y}_1)\frac{\partial h_1^N}{\partial h_i^{N-1}} + (\vec{o}_2 - \vec{y}_2)\frac{\partial h_2^N}{\partial h_i^{N-1}} + \cdots + (\vec{o}_O - \vec{y}_O)\frac{\partial h_O^N}{\partial h_i^{N-1}}\right]\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} \tag{2.12}$$

It is even worse for the layer after that. This seems like our calculations grow exponentially, the deeper our weights are. But we have a solution; we already calculated $\frac{\partial \vec{h_i^N}}{\partial \mathbf{W}_{ij}^{N-1}}$ in equation 2.9, which is very similar to $\frac{\partial h_i^N}{\partial h_j^{N-1}}$ on the left side of equation 2.12:

$$\frac{\partial \vec{h_i^n}}{\partial \mathbf{W}_{ij}^{n-1}} = \left[1 - \tanh^2(\vec{b_i}^{n-1} + \mathbf{W}_i^{n-1}\vec{h}^{n-1})\right]\vec{h}_j^{n-1} \tag{2.13}$$

$$\frac{\partial h_i^n}{\partial h_j^{n-1}} = \left[1 - \tanh^2(\vec{b_i}^{n-1} + \mathbf{W}_i^{n-1}\vec{h}^{n-1})\right]\mathbf{W}_{ij}^{n-1} \tag{2.14}$$

The solution of back-propagation is to use extra variables, called deltas δ_i^n. These deltas are defined in the following way:

$$\delta_i^N = (\vec{o}_i - \vec{y}_i) \tag{2.15}$$

$$\delta_i^{n-1} = \left[\delta_1^n\frac{\partial h_1^n}{\partial h_i^{n-1}} + \delta_2^n\frac{\partial h_2^n}{\partial h_i^{n-1}} + \cdots + \delta_M^n\frac{\partial h_M^n}{\partial h_i^{n-1}}\right] \tag{2.16}$$

$$= \sum_{j=1..M}\delta_j^n\frac{\partial h_j^n}{\partial h_i^{n-1}} \tag{2.17}$$

Substituting 2.16 in 2.9 and 2.12 gives our equation to calculate the gradient of the error to the weights using deltas:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{n-1}} = \delta_i^n \frac{\partial \vec{h}_i^n}{\partial \mathbf{W}_{ij}^{n-1}} \tag{2.18}$$

Update rule We now have a process to efficiently calculate the gradient of the error towards the network parameters. Moving the parameters in this direction is the quickest way to increase the error; we want to minimize the error so we move in the opposite direction of the gradient, creating our update rules:

$$\mathbf{W}_{ij}^n \overset{\lambda_{weight}}{\longleftarrow} \mathbf{W}_{ij}^n - \frac{\partial E}{\partial \mathbf{W}_{ij}^n} \tag{2.19}$$

$$\vec{b}_i^n \overset{\lambda_{bias}}{\longleftarrow} \vec{b}_i^n - \frac{\partial E}{\partial \vec{b}_i^n} \tag{2.20}$$

Algorithm The backpropagation algorithm is as follows:

1 Define δ^N according to 2.15.

2 Calculate δ^{N-1} according to 2.16 and 2.14.

3 Update the weights $\mathbf{W}^{N-1}$ using the update rules 2.19 where the gradient is calculated using the deltas δ^{N-1}, according to 2.18 and 2.13. Formula 2.13 is already partly calculated as it is almost the same as 2.14

4 Using the new calculated deltas, go down one layer and repeat step 2 and 3 for this layer. Stop when you have reached and adjusted the lowest layer.

Initializing the weights We initialize the weights of each layer by drawing them from a uniform random distribution in the interval: $[-\frac{0.3}{\sqrt{n_{input}}}, \frac{0.3}{\sqrt{n_{input}}}]$ where n_{input} is the number of inputs to that layer [LeCun et al., 1998]. The 0.3 comes from the fact that we normalize the image data to a standard deviation of $\sigma = 0.3$, so the values are in the range of the output of the neural network – so we can also reconstruct them.

The initialization gives the weights a small needed bias; when we would initialize them at zero the outputs will also be zero and back-propagation would not work. The bias is still only small, to make sure the neural network is still activated in the linear part of its non-linear functions. At the linear part the gradients are the largest and training is the fastest [LeCun et al., 1998] (see Figure 2.1).

Choosing learning rates Choosing the right learning rates is critical; if they are too high the network will not converge and learning will be unstable, if they are too low convergence takes very long. We scale the learning rates for every layer proportional to $\frac{1}{\sqrt{n^i}}$, where n^i is the number of inputs of layer i. This is according to suggestions made in other literature [LeCun et al., 1998, Embrechts et al., 2010].

We start by using a learning factor λ_{factor} and calculate the learning rate from that:

$$\lambda_{weight} = \frac{\lambda_{factor}}{\sqrt{n^i}} \tag{2.21}$$

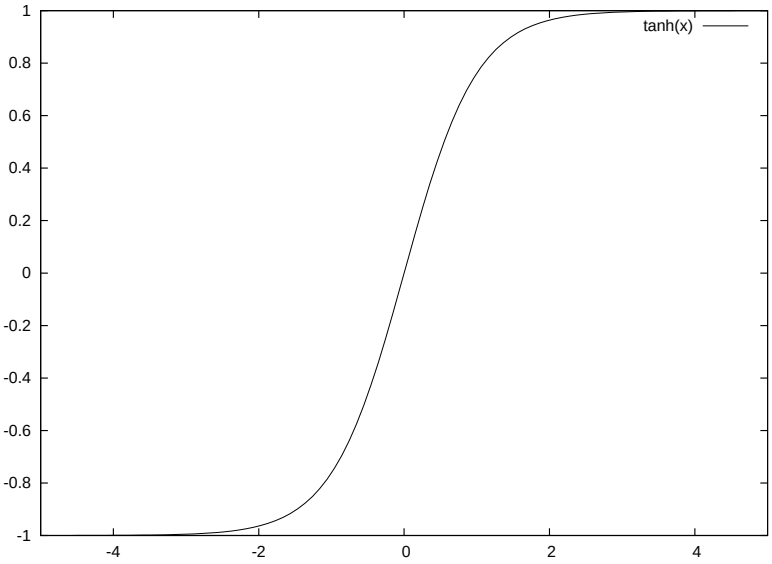


Figure 2.1: The slope of the tanh function is highest around $x = 0$. Gradients will be the highest if the weights are initialized so the function is activated in this part. This improves learning speed.

where λ_{weight} is the learning rate for the weights and λ_{bias} the learning rate for the bias. The learning rates are used to slowly adjust the parameters of the network to a good solution.

We use factor ten smaller learning rate for the bias: $\lambda_{bias} = \frac{\lambda_{factor}}{10\sqrt{n^i}}$ We do this because there are more weights than biases which means the training of the bias should settle earlier. Therefore a lower learning rate is preferable.

2.1.2.2 Learning to Encode

As mentioned in section 1.3.4, autoencoders are a special kind of neural networks that recreate the input they are given. We start out with a neural network that has as many inputs as it has outputs. A pattern $\vec{x}$ is taken from the dataset, the neural network is activated and we learn the network using back-propagation where we define the desired output to be the same vector $\vec{y} = \vec{x}$.

Denoising To help generalization we can add noise to the pattern before presenting it to the network by applying a noise function $g(\vec{x})$. The desired output is still the input vector without noise $\vec{y} = \vec{x}$.

Autoencoders that are trained with noise are called denoising autoencoders [Bengio et al., 2007], because they learn to remove the noise added over the input. Denoising autoencoders tend to be more robust, although the reasons why are not entirely clear. It is hypothesized that by pushing the pattern around in its data-space before learning, the autoencoder learns a basin of attraction from which the pattern can be reconstructed.

We use salt and pepper noise which chooses a portion λ_{noise} number of pixels at random and sets their values to white or black, basically removing the information of that pixel. We set half of the selected pixels to -0.5 for salt and the other half to 0.5 for pepper noise.

Sparse Encoding To get better generalization we can use sparse encoding. This means that we give an a-priori preference to certain encoding vectors. The effect is that the autoencoder will use

a smaller encoding space to satisfy this preference. A smaller encoding space forces the encoder to compress the data even more.

We can enforce sparse encoding by adding a term to the minimization function 2.4. A possible choice would be to minimize the l_0 norm of the encoding $\|\vec{h}\|_0$ where $\vec{h}$ is the encoding vector. It counts the number of non-null elements in the vector. This would force the encoding to use the minimum amount of vector elements – the minimum amount of bits.

Minimizing the l_0 norm however turns out to be impractical as it is non-convex and does not translate well to gradient descent learning in backpropagation. Instead we use the l_1 norm $\|\vec{h}\|_1 = \sum_i |\vec{h}_i|$ which is convex and also is a good proxy of the l_0 norm [Candes and Tao, 2005, Donoho, 2006]. The minimization function from equation 2.4 becomes:

$$E(\vec{y}, \vec{x}, \Theta) = \frac{1}{2}\|f(\vec{x}, \Theta) - \vec{y}\|^2 + \lambda_{l1}\|\vec{h}\|_1 \tag{2.22}$$

where λ_{l1} is a factor which balances $l1$ minimization and error minimization. There are dangers of using $l1$ minimization, using too high λ_{l1} can cause instabilities when learning [Bengio, 2009].

In [Wright et al., 2009] sparse representations are used on a face recognition task. They also show that $l1$ minimization creates sparse representations whereas $l2$ regularization creates dense representations. This can be seen in Figure 2.2 where the effect of the $l1$ norm is shown. The two arrows in the figure represent possible hidden layer activations, or encodings, of a certain input. $l1$ minimization would favour the left situation which codes the example closer to the axis over the right situation, using less dimensions for the encoding. $l2$ minimization would give no preference between the two situations, because it measures the length of the vectors which are the same.

Calculating performance We need a quantitative way of measuring the performance of the autoencoder. The root mean square error is a logical choice:

$$RMSE(D, \Theta) = \sqrt{\frac{1}{|D|}\sum_{(\vec{x},\vec{y})\in D} \|\vec{y} - f(\vec{x}, \Theta)\|^2} \tag{2.23}$$

where D represents the set of samples we use to measure the error, Θ are the parameters of the autoencoder and $f(\vec{x}, \Theta)$ is the autoencoder.

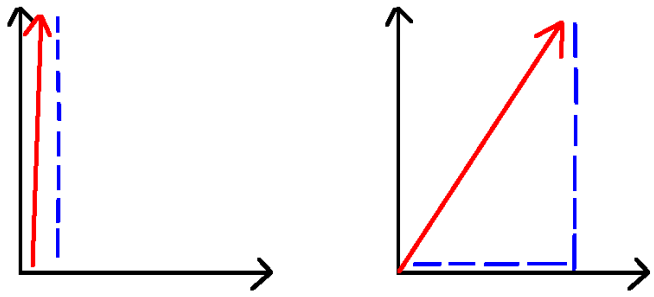


Figure 2.2: The arrow represents the encoding of an autoencoder and the dotted line is the $l1$ norm of this encoding. $l1$ minimization favours the left explanation of the data over the right one, minimizing the number of bits uses. This creates sparse representations.

2.2 Training Procedure

We will now describe the steps taken to learn the autoencoders.

2.2.1 Using the Dataset

The LFWC dataset that we are using contains about 13000 images of faces. We annotated the positions of parts of the face for a subset of 1300 faces. This annotated dataset is used in all experiments in this thesis. Using the same dataset for the experiments makes comparing results easier.

First we initialize our models:

Initialization

1. We randomly divide up the dataset in a training set and a test set. We randomly select a portion $\eta_{portion}$ of the samples for training and the rest for testing. For all experiments in this thesis, unless noted otherwise, we use $\eta_{portion} = \frac{2}{3}$.
2. We initialize the autoencoder.
3. We set the learning parameters to their values.

Then we start learning in steps called 'epochs'. In every 'epoch', every sample from our training set comes by exactly one time. For every epoch we:

1. Walk over all the samples in the training-set in a random order, add salt and pepper noise to the sample, and train the autoencoder using back-propagation. Using a random order is called stochastic gradient decent because we loop over the data stochastically, which has shown to lead to better results [LeCun et al., 1998].
2. Activate the autoencoder with all the samples from the training data and calculate the average error of the reconstruction using formula (2.23) for that epoch. This is the training error.
3. Activate the autoencoder with all the samples from the test data and calculate the average error of the reconstruction using formula (2.23) for that epoch. This is the test error.

We keep repeating these steps, advancing an epoch every time, until a defined stopping criterion is met (see below). At the end of the training we store the learned autoencoders for later evaluation.

2.2.2 Stopping Criterion

For our stopping criterion, we could use the relative increase in train performance between two epochs and compare it to a threshold:

$$\frac{RMSE(D_{train}, \Theta_t) - RMSE(D_{train}, \Theta_{t-1})}{RMSE(D_{train}, \Theta_{t-1})} > \eta_{stop} \tag{2.24}$$

where D_{train} is the training data and Θ_t are the network parameters at epoch t. The problem with this setup is that the $RMSE$ has some fluctuation which can cause the stopping criterion to be met too early. To counter this we look at the $RMSE$ of the last $\eta_{rmse\_average}$ epochs and compute the mean $RMSE$ of the first half and second half of these epochs. By using these means we have a better estimation of the actual $RMSE$ and prevent these stability issues. We use $\eta_{rmse\_average} = 20$ and $\eta_{stop} = 0.0$ throughout this thesis, unless stated otherwise. Using $\eta_{stop} = 0.0$ means that we stop learning when the error starts to increase again.

2.2.3 The Window

In the second experiment of this section and in the coming chapters, we will need a process to create patches. Patches are small sub-images of an original image. We define a window over an image, and take patches by 'looking through' that window.

The window is defined by an x_{window} and y_{window} position which describes the center of the window, and a $width_{window}$ and $height_{window}$ which describe the size of the window. In this thesis we will exclusively use windows of size $width_{window} = height_{window} = 20$ pixels. The actual size of such windows are shown in Figure 1.2.

If the center of the window comes close to the border of the image, a part of the window can lie outside the image. If we then want to get a patch from the window, we have a problem. Our solution is to define pixels outside the border of the image to be equal to the closest pixel *in* the image.

2.3 Experiments

In the following experiments we are going to train our first autoencoders on the face images from our dataset. We will look at the quality of the reconstructions of these encoders. Then we will train autoencoders on parts of the face, instead of the full face, to see if using autoencoders locally improves the amount of detail that is retained.

2.3.1 Experiment 1: Encoding faces

We will start by trying to encode the full 64 by 64 pixel pictures of the faces with an autoencoder with one hidden layer. We vary the size of the hidden layer, the $l1$ regularization λ_{l1}, the amount of noise λ_{noise} and the learning rate λ_{factor}.

Expectations What we expect is:

hidden layer size We expect that the reconstruction error will decrease as the hidden layer size increases because the autoencoder can retain more information in the hidden layer.

noise We vary the noise between 0.0 (off) and 0.1 (on). We expect better generalization when noise is added, i.e. a lower test error.

l1 We vary the $l1$ regularization between $\lambda_{l1} = 0.0$ (off) and $\lambda_{l1} = 0.1$ (on). We expect better generalization and faster learning in the (on) condition.

learning rate We vary the learning rate by varying the learning factor. We use learning factor $\lambda_{factor} = 0.01$ and $\lambda_{factor} = 0.001$. The learning rate is then calculated according to equation 2.21. We expect that the condition with lower learning rate takes longer to converge.

We also expect the variance in poses of faces to prove a problem. For example, the position of the eyes in the images will be different every time. It will be hard for the autoencoder to correct for this variance because it can not naturally account for this variation.

2.3.1.1 Results

In table 2.1 the training and test $RMSE$ are shown for the different $|\vec{h}_{encoding}|$, λ_{factor}, λ_{l1} and λ_{noise}. The results are averaged over 6 experiment.

Firstly, it is clear that the bigger the hidden layer, the lower the test and train $RMSE$ will be.

Using a lower λ_{factor} yields lower reconstruction error.

Using $l1$ regularization does not have a positive effect on test error, contrary to what was expected. It often even raises the training error, but does not reduce test error.

Adding noise increases training error while it decreases test error. This is the expected effect and it shows that adding noise increases the generalization capability of the autoencoder.

The lowest training error is achieved with 200 hidden nodes without added noise or $l1$ regularization.

The lowest test error is achieved under the same conditions, but with added noise, again showing that adding noise increases generalization.

Reconstructions Figure 2.3 shows face-reconstructions of test images under different learning rates. The quantitative results show that the lower learning rate has better reconstructions. The resulting reconstructions show differences, especially at $|\vec{h}_{encoding}| = 20$, but they are hard to interpret. We could not find clear consistent differences.

Figure 2.4 shows reconstructions of the autoencoders trained with and without added noise. The differences in reconstruction are small but noticeable. At $|\vec{h}_{encoding}| = 200$, the reconstructions trained without noise show more artifacts than those trained with noise. This is a good example of the generalization capability that added noise can have.

Finally, Figure 2.5 shows the reconstruction of faces with and without $l1$ regularization. Again the differences are small. The biggest differences are at $|\vec{h}_{encoding}| = 200$, where the reconstructions with $l1$ regularization have a little more detail.

Evolution of $RMSE$ Figure 2.6 shows the evolution of train and test $RMSE$ for different λ_{factor}. Figure 2.7 shows this for different noise conditions and Figure 2.8 for different λ_{l1} values. The $RMSE$s are averaged over 6 runs.

In all three figures we see that the differences between train and test error increases over time and is bigger for larger $|\vec{h}_{encoding}|$. This is probably because larger hidden layers give a neural network more parameters to overfit on their training data. The differences between the two noise and λ_{l1} conditions are not noticeable. The graphs for different learning rates do show differences, where the $RMSE$ decreases more gradually for lower λ_{l1}. This effect becomes more noticeable for larger $|\vec{h}_{encoding}|$.

$\lvert\vec{h}_{encoding}\rvert$	λ_{factor}	λ_{noise}	λ_{l1}	train $RMSE$	test $RMSE$
5	0.001	0.0	0.0	0.2263 +- 0.0003	0.2282 +- 0.0008
5	0.001	0.0	0.01	0.2254 +- 0.0006	0.2273 +- 0.0006
5	0.001	0.1	0.0	0.2263 +- 0.0004	0.2294 +- 0.0004
5	0.001	0.1	0.01	0.2254 +- 0.0006	0.2289 +- 0.0003
5	0.01	0.0	0.0	0.2273 +- 0.0007	0.2300 +- 0.0007
5	0.01	0.0	0.01	0.2280 +- 0.0004	0.2309 +- 0.0005
5	0.01	0.1	0.0	0.2284 +- 0.0003	0.2309 +- 0.0004
5	0.01	0.1	0.01	0.2264 +- 0.0010	0.2296 +- 0.0007
20	0.001	0.0	0.0	0.1828 +- 0.0003	0.1953 +- 0.0006
20	0.001	0.0	0.01	0.1830 +- 0.0003	0.1958 +- 0.0007
20	0.001	0.1	0.0	0.1846 +- 0.0003	0.1950 +- 0.0005
20	0.001	0.1	0.01	0.1836 +- 0.0004	0.1969 +- 0.0005
20	0.01	0.0	0.0	0.1904 +- 0.0002	0.2033 +- 0.0004
20	0.01	0.0	0.01	0.1903 +- 0.0001	0.2037 +- 0.0004
20	0.01	0.1	0.0	0.1899 +- 0.0001	0.2036 +- 0.0004
20	0.01	0.1	0.01	0.1908 +- 0.0002	0.2025 +- 0.0004
100	0.001	0.0	0.0	0.1017 +- 0.0004	0.1331 +- 0.0008
100	0.001	0.0	0.01	0.1031 +- 0.0003	0.1317 +- 0.0004
100	0.001	0.1	0.0	0.1078 +- 0.0003	0.1308 +- 0.0002
100	0.001	0.1	0.01	0.1087 +- 0.0004	0.1305 +- 0.0002
100	0.01	0.0	0.0	0.1227 +- 0.0002	0.1640 +- 0.0004
100	0.01	0.0	0.01	0.1227 +- 0.0001	0.1641 +- 0.0002
100	0.01	0.1	0.0	0.1246 +- 0.0001	0.1634 +- 0.0002
100	0.01	0.1	0.01	0.1246 +- 0.0001	0.1634 +- 0.0003
200	0.001	0.0	0.0	0.0670 +- 0.0004	0.1036 +- 0.0002
200	0.001	0.0	0.01	0.0670 +- 0.0006	0.1035 +- 0.0005
200	0.001	0.1	0.0	0.0746 +- 0.0003	0.1037 +- 0.0003
200	0.001	0.1	0.01	0.0754 +- 0.0002	0.1028 +- 0.0002
200	0.01	0.0	0.0	0.0843 +- 0.0002	0.1379 +- 0.0002
200	0.01	0.0	0.01	0.0844 +- 0.0001	0.1381 +- 0.0006
200	0.01	0.1	0.0	0.0911 +- 0.0002	0.1426 +- 0.0004
200	0.01	0.1	0.01	0.0910 +- 0.0002	0.1426 +- 0.0005

Table 2.1: Results from training an autoencoder on the images of faces. Averages over 6 runs.

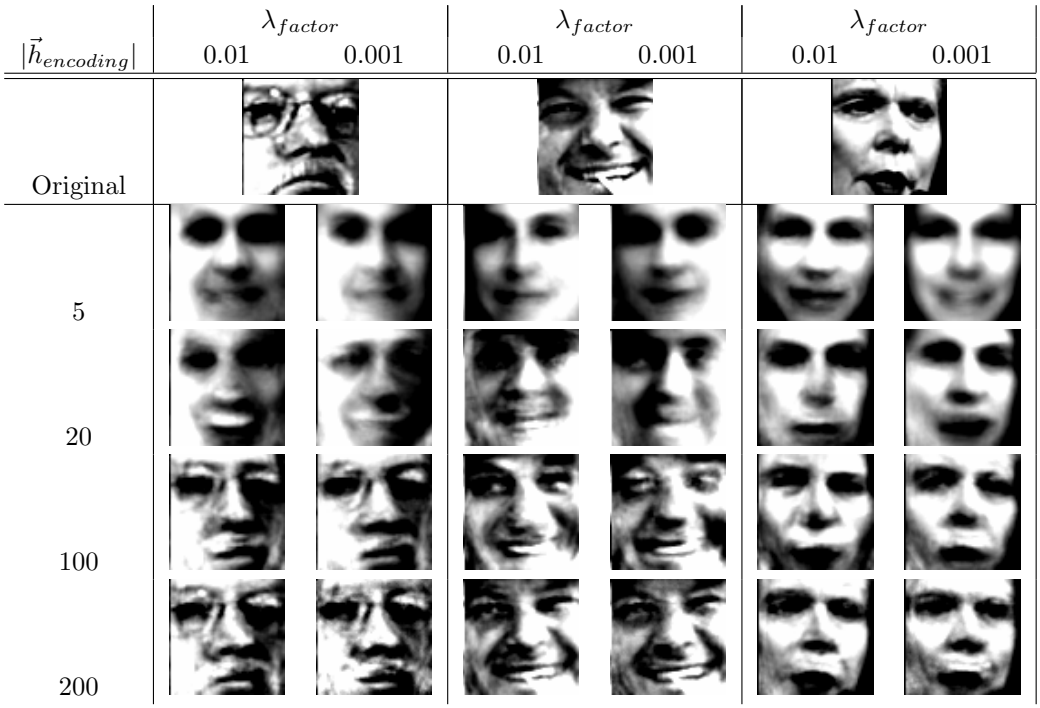


Figure 2.3: Reconstructions of the face using different learning rates.

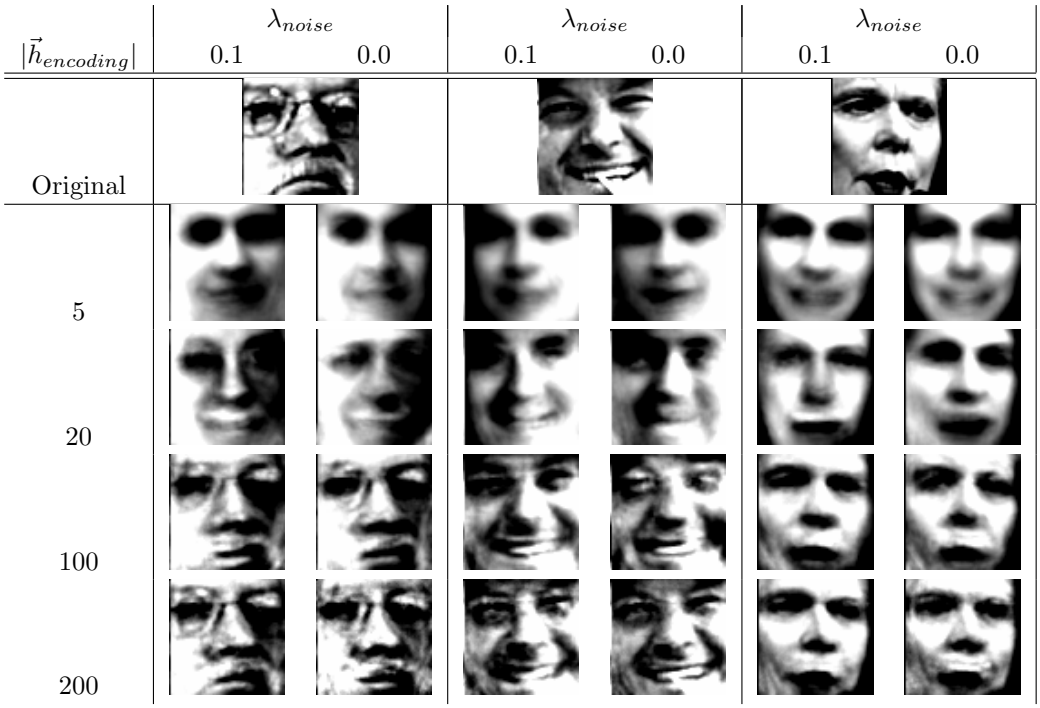


Figure 2.4: Reconstructions of the face under different noise conditions.

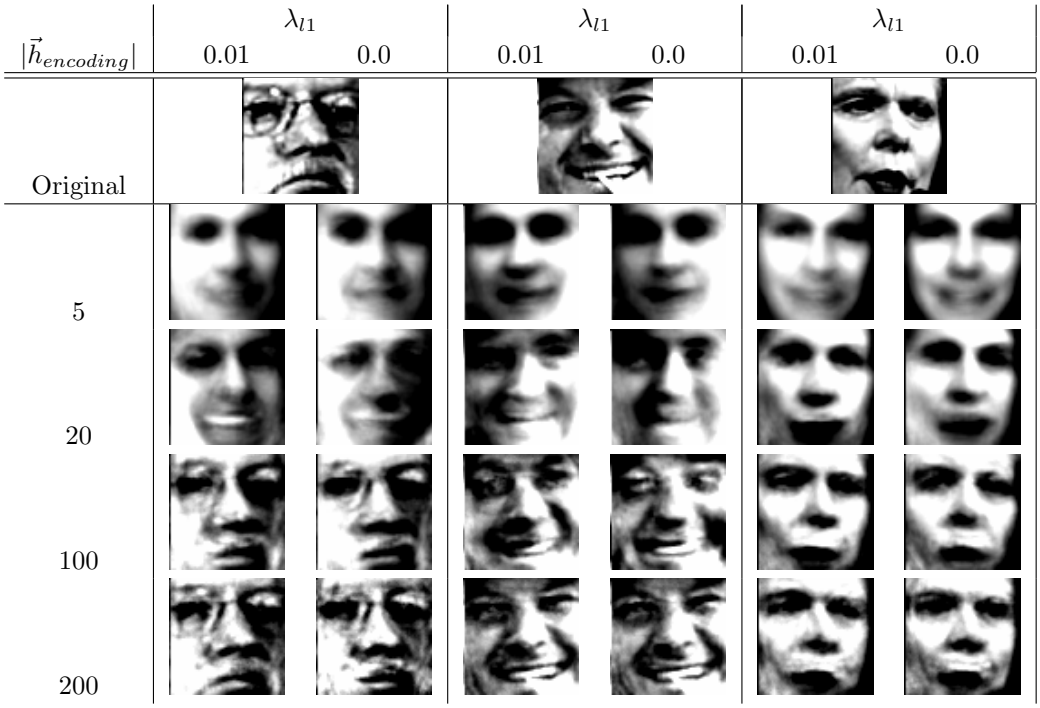


Figure 2.5: Reconstructions of the face for different l1 factors.

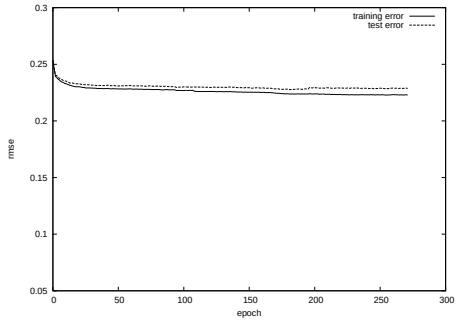
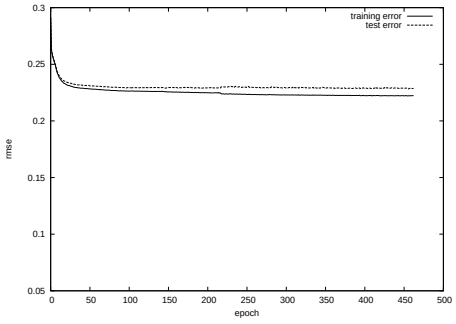
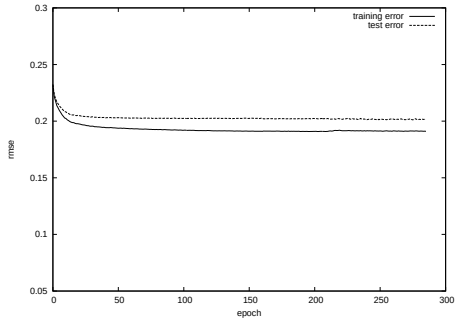
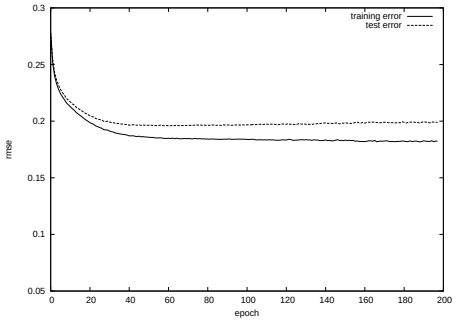
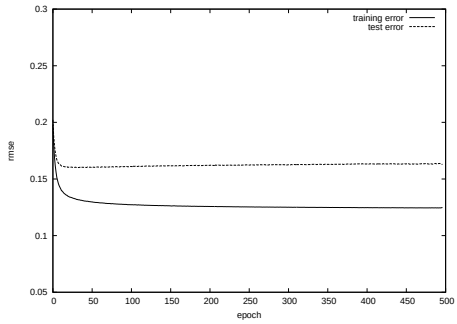
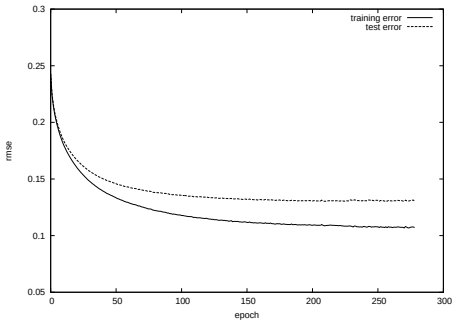
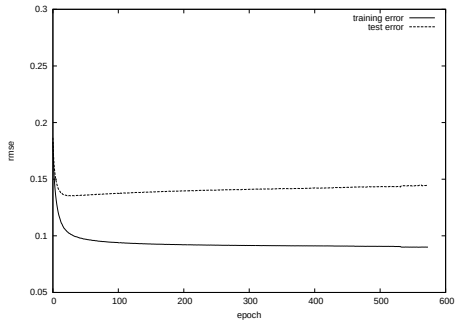
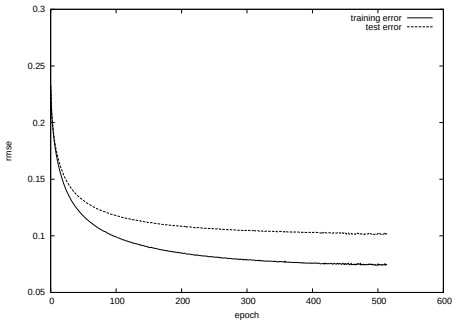
$\|h\|$	$\lambda_{factor} = 0.01$	$\lambda_{factor} = 0.001$
5		
20		
100		
200		

Figure 2.6: progress of back-propagation for face-encoding, under different learning factors

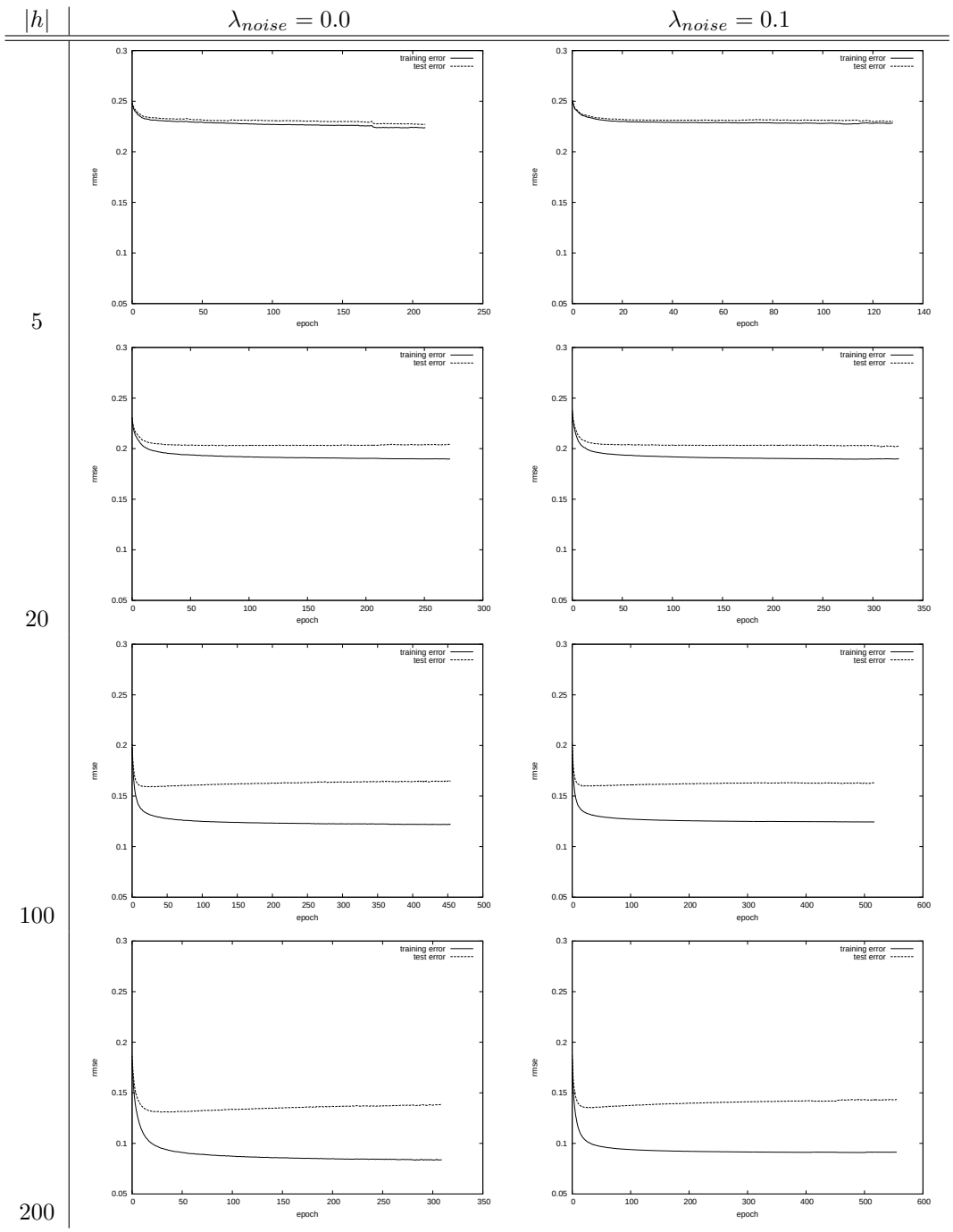


Figure 2.7: progress of back-propagation for face-encoding, under different noise conditions

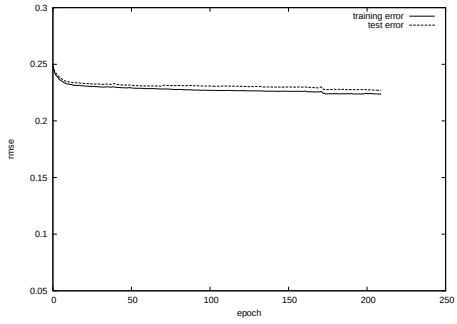
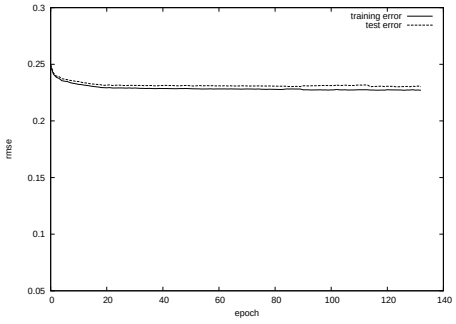
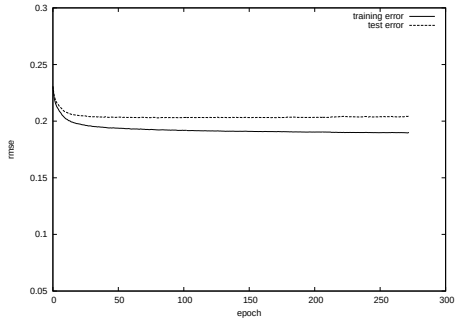
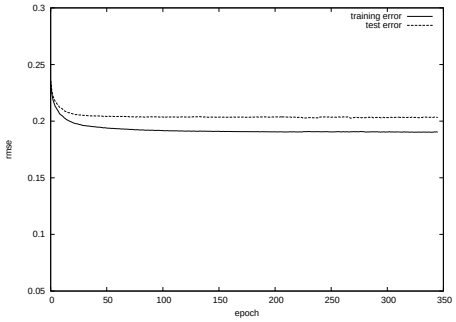
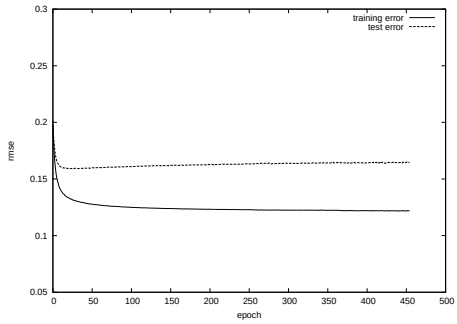
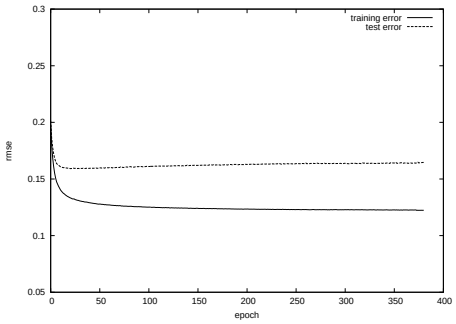
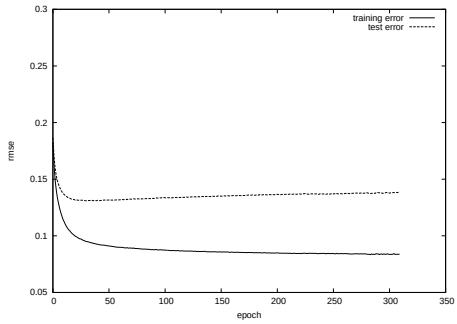
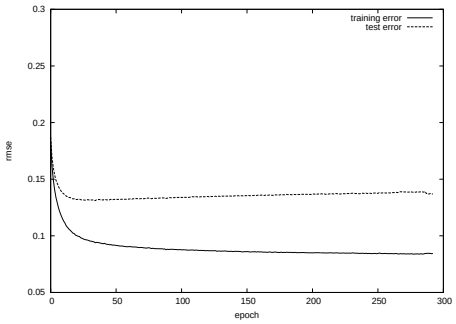
$\lvert h \rvert$	$\lambda_{l1} = 0$	$\lambda_{l1} = 0.01$
5		
20		
100		
200		

Figure 2.8: progress of back-propagation for face-encoding, under different l1 constants

To summarize The reconstructions seem quite good, but still if we look closely we see details in the original face that are missing in the reconstruction. For example, the eyes of the second face have recognizable pupils, which do much to give the person an expression. These details are gone, even when using a hidden layer size of 200. We would like to retain those details because they are important for understanding the face.

In the next experiment we will try to remedy this problem by focusing specifically on the important parts of the face with local encoders.

2.3.2 Experiment 2: Encoding parts of the face

In this experiment, instead of encoding full faces, we will locally encode parts of the face. We hope to retain more details when encoding the parts of the face separately because we take away the variance in position. For this experiment we use a learning rate of $\lambda_{learnfactor} = 0.01$, vary noise $\lambda_{noise} = [0.01, 0.1]$ and vary regularization $\lambda_{l1} = [0.0, 0.01]$.

We will train autoencoders on the left and right eye, the mouth and the nose. We create patches by putting a 20 by 20 pixel window centered at the annotation positions (see figure 1.2). We vary the hidden layer size again and expect better reconstructions for layers of larger sizes. We want to determine if using local encoding retains the details of the face better.

Creating Patches We create patches by putting the center of the window, described in section 2.2.3, on the annotated positions. We then evaluate the window to create a patch. We use small deviations from the annotated position to create a little more varied dataset. The deviations are between 0 and 2 pixels, chosen uniformly, in a random direction. Again, the annotated dataset of 1300 faces is used, which is divided in a test and training set with $\eta_{portion} = \frac{2}{3}$.

Expectations We expect the local encoder to retain more details of the parts of the face, because they have a lower dimensional input vector which is also centered around that part. Therefore the local encoder does not have to model the variances in position of parts of the face in contrast to the full face-encoder.

2.3.2.1 Results

Table 2.2 and 2.3 show the training and test $RMSE$ of the reconstructions. The experiments are averaged over 4 experiments.

According to these figures, the nose is the easiest to train, as it has the lowest reconstruction error. A possible explanation for this is that noses are less varied and generally have smoother features than the eyes. Using a bigger hidden layer lowers reconstruction error, as expected. We used two noise conditions, a higher and a lower noise condition. The high noise condition has a negative effect on both training and test error, showing that we should be careful when adding noise to our data. There are no significant differences found between the two λ_{l1} conditions.

Reconstructions Figure 2.9 shows actual reconstructions of the eye, nose and mouth. Qualitatively the reconstructions for a hidden layer of size 10 and 20 are very blurred. A size of 50 starts to produce recognizable results, but still misses some details. For example, the lower teeth that are present in one of the mouth images are not present when using size 50, but are reconstructed correctly for size 100.

The difference between a hidden layer size of 100 and 200 is small and almost not noticeable. It seems especially important for the eyes, which are almost perfectly reconstructed for size 200, but are distorted for size 100. Some seem to change orientation, looking like a left eye, instead of a right eye. We don't know what causes this change. So although in most cases there is not a big difference between using 100 or 200 hidden nodes, there are cases that depend on it. Quantitatively we see a lower test error for 200 nodes than for 100 nodes in Table 2.2 and 2.3. For this reason, we will use this layer of 200 nodes in subsequent experiments.

Figure 2.10 shows the change in error over training time. The error gradually reduces following a nice curve, as expected.

face part	hidden layer	noise	l1	train $RMSE$	test $RMSE$
eye	10	0.01	0.0	0.1252 +- 0.0004	0.1272 +- 0.0006
eye	10	0.01	0.01	0.1256 +- 0.0005	0.1275 +- 0.0006
eye	10	0.1	0.0	0.1294 +- 0.0004	0.1301 +- 0.0007
eye	10	0.1	0.01	0.1298 +- 0.0009	0.1314 +- 0.0006
eye	20	0.01	0.0	0.0937 +- 0.0003	0.0955 +- 0.0007
eye	20	0.01	0.01	0.0936 +- 0.0004	0.0938 +- 0.0006
eye	20	0.1	0.0	0.0983 +- 0.0007	0.0997 +- 0.0009
eye	20	0.1	0.01	0.0985 +- 0.0007	0.0987 +- 0.0006
eye	50	0.01	0.0	0.0565 +- 0.0002	0.0580 +- 0.0005
eye	50	0.01	0.01	0.0559 +- 0.0003	0.0564 +- 0.0006
eye	50	0.1	0.0	0.0705 +- 0.0008	0.0717 +- 0.0009
eye	50	0.1	0.01	0.0704 +- 0.0003	0.0710 +- 0.0002
eye	100	0.01	0.0	0.0400 +- 0.0003	0.0404 +- 0.0002
eye	100	0.01	0.01	0.0395 +- 0.0003	0.0400 +- 0.0007
eye	100	0.1	0.0	0.0637 +- 0.0004	0.0645 +- 0.0003
eye	100	0.1	0.01	0.0631 +- 0.0004	0.0641 +- 0.0004
eye	200	0.01	0.0	0.0338 +- 0.0003	0.0348 +- 0.0006
eye	200	0.01	0.01	0.0346 +- 0.0002	0.0358 +- 0.0003
eye	200	0.1	0.0	0.0600 +- 0.0005	0.0609 +- 0.0006
eye	200	0.1	0.01	0.0619 +- 0.0002	0.0619 +- 0.0006
eye right	10	0.01	0.0	0.1253 +- 0.0003	0.1262 +- 0.0008
eye right	10	0.01	0.01	0.1246 +- 0.0005	0.1263 +- 0.0002
eye right	10	0.1	0.0	0.1285 +- 0.0006	0.1296 +- 0.0009
eye right	10	0.1	0.01	0.1294 +- 0.0003	0.1304 +- 0.0011
eye right	20	0.01	0.0	0.0926 +- 0.0003	0.0937 +- 0.0005
eye right	20	0.01	0.01	0.0929 +- 0.0004	0.0930 +- 0.0002
eye right	20	0.1	0.0	0.0997 +- 0.0002	0.0995 +- 0.0007
eye right	20	0.1	0.01	0.0977 +- 0.0002	0.0980 +- 0.0004
eye right	50	0.01	0.0	0.0556 +- 0.0003	0.0582 +- 0.0007
eye right	50	0.01	0.01	0.0551 +- 0.0002	0.0573 +- 0.0004
eye right	50	0.1	0.0	0.0712 +- 0.0001	0.0713 +- 0.0002
eye right	50	0.1	0.01	0.0717 +- 0.0011	0.0724 +- 0.0008
eye right	100	0.01	0.0	0.0384 +- 0.0004	0.0394 +- 0.0003
eye right	100	0.01	0.01	0.0394 +- 0.0002	0.0400 +- 0.0005
eye right	100	0.1	0.0	0.0638 +- 0.0004	0.0647 +- 0.0002
eye right	100	0.1	0.01	0.0649 +- 0.0005	0.0663 +- 0.0006
eye right	200	0.01	0.0	0.0339 +- 0.0003	0.0340 +- 0.0002
eye right	200	0.01	0.01	0.0344 +- 0.0003	0.0358 +- 0.0004
eye right	200	0.1	0.0	0.0601 +- 0.0005	0.0609 +- 0.0006
eye right	200	0.1	0.01	0.0617 +- 0.0005	0.0627 +- 0.0004

Table 2.2: Results of training autoencoders on parts of the face.

face part	hidden layer	noise	l1	train $RMSE$	test $RMSE$
mouth	10	0.01	0.0	0.1240 +- 0.0006	0.1242 +- 0.0005
mouth	10	0.01	0.01	0.1231 +- 0.0005	0.1227 +- 0.0006
mouth	10	0.1	0.0	0.1254 +- 0.0003	0.1266 +- 0.0007
mouth	10	0.1	0.01	0.1270 +- 0.0005	0.1265 +- 0.0009
mouth	20	0.01	0.0	0.0898 +- 0.0005	0.0899 +- 0.0009
mouth	20	0.01	0.01	0.0894 +- 0.0003	0.0899 +- 0.0005
mouth	20	0.1	0.0	0.0929 +- 0.0004	0.0948 +- 0.0004
mouth	20	0.1	0.01	0.0932 +- 0.0007	0.0940 +- 0.0004
mouth	50	0.01	0.0	0.0552 +- 0.0003	0.0574 +- 0.0008
mouth	50	0.01	0.01	0.0559 +- 0.0007	0.0570 +- 0.0005
mouth	50	0.1	0.0	0.0674 +- 0.0008	0.0703 +- 0.0004
mouth	50	0.1	0.01	0.0670 +- 0.0001	0.0693 +- 0.0002
mouth	100	0.01	0.0	0.0405 +- 0.0004	0.0416 +- 0.0004
mouth	100	0.01	0.01	0.0395 +- 0.0002	0.0421 +- 0.0003
mouth	100	0.1	0.0	0.0601 +- 0.0006	0.0616 +- 0.0005
mouth	100	0.1	0.01	0.0613 +- 0.0002	0.0617 +- 0.0006
mouth	200	0.01	0.0	0.0345 +- 0.0003	0.0354 +- 0.0007
mouth	200	0.01	0.01	0.0358 +- 0.0007	0.0371 +- 0.0006
mouth	200	0.1	0.0	0.0571 +- 0.0002	0.0575 +- 0.0003
mouth	200	0.1	0.01	0.0588 +- 0.0003	0.0587 +- 0.0007
nose	10	0.01	0.0	0.1200 +- 0.0005	0.1210 +- 0.0005
nose	10	0.01	0.01	0.1192 +- 0.0002	0.1214 +- 0.0003
nose	10	0.1	0.0	0.1229 +- 0.0005	0.1233 +- 0.0005
nose	10	0.1	0.01	0.1231 +- 0.0005	0.1237 +- 0.0006
nose	20	0.01	0.0	0.0887 +- 0.0003	0.0897 +- 0.0002
nose	20	0.01	0.01	0.0891 +- 0.0000	0.0891 +- 0.0007
nose	20	0.1	0.0	0.0924 +- 0.0003	0.0944 +- 0.0006
nose	20	0.1	0.01	0.0923 +- 0.0002	0.0934 +- 0.0005
nose	50	0.01	0.0	0.0524 +- 0.0003	0.0531 +- 0.0002
nose	50	0.01	0.01	0.0518 +- 0.0003	0.0532 +- 0.0002
nose	50	0.1	0.0	0.0634 +- 0.0004	0.0646 +- 0.0005
nose	50	0.1	0.01	0.0634 +- 0.0002	0.0643 +- 0.0005
nose	100	0.01	0.0	0.0335 +- 0.0003	0.0348 +- 0.0005
nose	100	0.01	0.01	0.0345 +- 0.0006	0.0356 +- 0.0008
nose	100	0.1	0.0	0.0571 +- 0.0003	0.0575 +- 0.0001
nose	100	0.1	0.01	0.0564 +- 0.0003	0.0572 +- 0.0005
nose	200	0.01	0.0	0.0290 +- 0.0005	0.0299 +- 0.0002
nose	200	0.01	0.01	0.0298 +- 0.0010	0.0306 +- 0.0008
nose	200	0.1	0.0	0.0538 +- 0.0001	0.0548 +- 0.0001
nose	200	0.1	0.01	0.0558 +- 0.0002	0.0556 +- 0.0004

Table 2.3: Results of training autoencoders on parts of the face.

Figure 2.9: Reconstruction of parts of the face for several different hidden layer sizes.

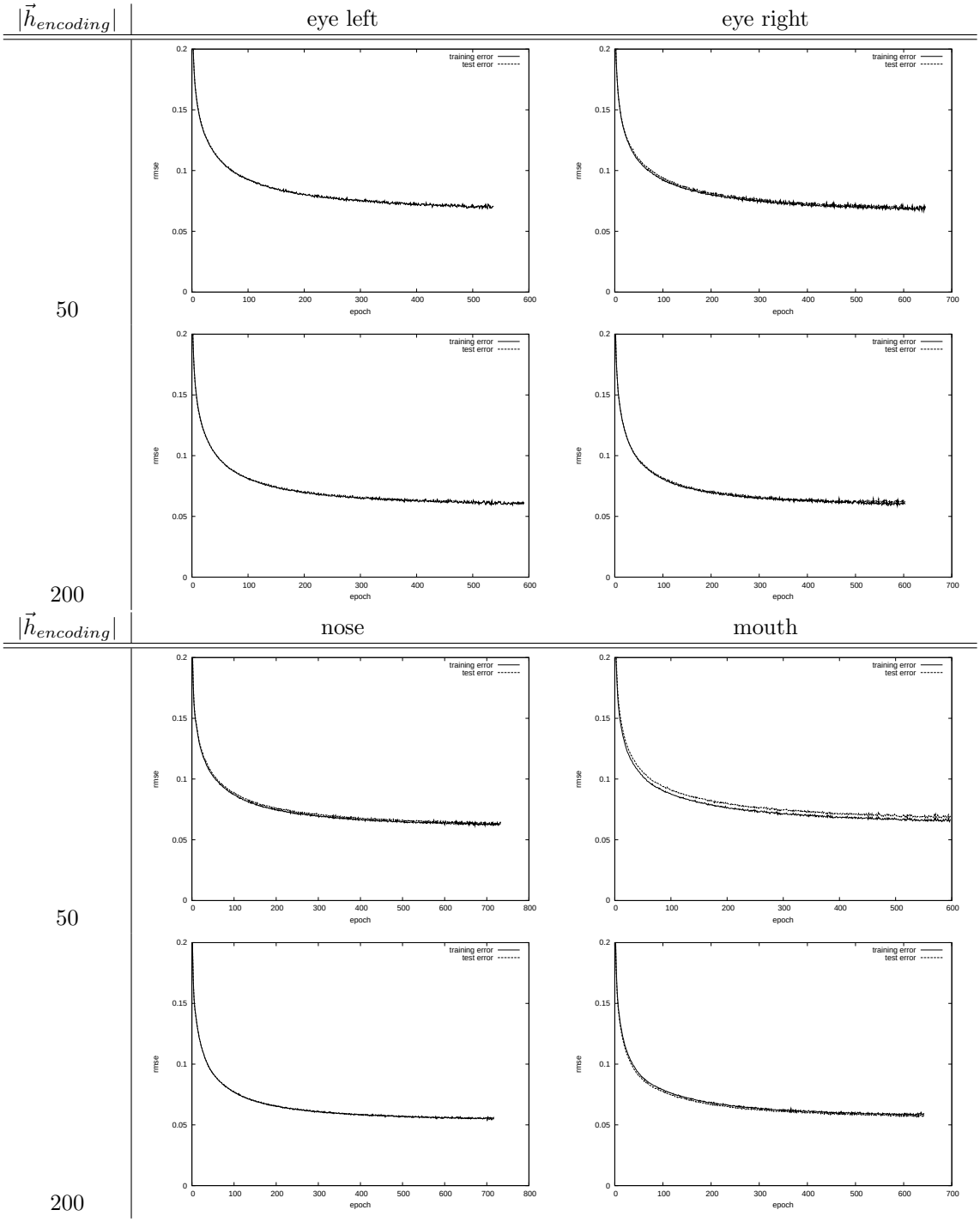


Figure 2.10: Learning progress of encoding eyes, nose and mouth.

2.4 Discussion

The results show that the variance in position of the parts of the face make it difficult for the full face encoder to retain details. A lot of information is therefore lost, which can be seen in the reconstructions of the faces that are still recognizable although important details are missing. The results from the experiments show that local autoencoders that reconstruct parts of faces retain more detail than encoders that encode the full face. The experiments also show that $l1$ minimization does not have a clear effect when training the face encoders. Adding noise however, does give us lower training errors, helping the autoencoder to generalize.

To develop the guided autoencoders, the goal of the thesis, we need the representations of the local autoencoders. However, we still need a hidden layer size of 200 to correctly reconstruct all patches, which is too big. In the next chapter we will train SAEs that gradually lower the size of the representation and hopefully are able to better reconstruct the patches using a lower dimensional representation.

Chapter 3

Deep Architectures

3.1 Introduction

In the previous chapter, we trained autoencoders to reconstruct parts of the face. These autoencoders have one hidden layer, making them shallow. In this chapter we are going to use deeper models – autoencoders with several hidden layers. Such deeper models are capable of expressing more complex functions that are out of reach for shallow models.

Deep models however were not used much in research because they were very hard to train properly. However, recent advances in machine learning combined with the fast computers of this age are starting to pull this off. In this section we will use one of these recent advances, stacked autoencoders (SAE) [Hinton and Salakhutdinov, 2006, Bengio et al., 2007], to build lower level representations of the faces in the dataset.

Outline First we will explain the ideas behind deep representations and show how to train stacked autoencoders. Then we will train stacked autoencoders on parts of the face and compare their results. We will find that the difference between deep and shallow autoencoders for this problem is negligible.

3.1.1 Deep Representations

Neural networks have been used in research for a long time, and ever since researchers have been interested to increase the number of layers in the network. There are several reasons why we would want to learn these deep networks. A deep network essentially is a hierarchical combination of many non-linear functions and can therefore perform very complex calculations. This allows deep networks to express complex functions much more efficiently than a neural network with one layer.

In [Bengio, 2009] several theoretical examples are given from other papers of functions that are represented efficiently with the right amount of layers, but start to need exponentially more nodes when the network is too shallow. Deep representations also seem to be used by the brain [Serre et al., 2007], making deep networks more biologically plausible than shallow networks.

Highly Varying Functions The class of functions that need deep representations to be represented efficiently are highly varying functions. Highly varying functions are functions $f : x \to y$ where the set of inputs $x \in X$ that are close in the output space are not necessarily close in the input space.

For example, we take a function f that takes images as input, and outputs the object contained in that image. All images that contain trees are close in output space, since they have as output 'tree'. But in input space they are not close. To see this, think of an image containing a tree, where its leaves are shown in the image as an unpredictable group of bright-green, dark-green and black

pixels because of all the light variation, occlusions and complexity present in a typical tree. If we would move that tree one pixel to the right, we would say that the image barely changed and the output of our function should still be almost the same. However, in the input space, many pixels changed dark to bright or the other way around – a big difference in input space. This is a highly varying function.

The tree example also shows *why* we need deep networks. Much of the stuff that we want to recognize are highly varying.

Training Before some recent advances, it has always been notoriously difficult to train neural networks with more than a handful of layers. The standard learning algorithm, backpropagation, did not work well on neural networks with more than 3 layers. It is not completely clear why this is the case, but three possible causes are given in [Bengio, 2009]:

- Gradient descent can easily get stuck in local minima because of the many parameters that define the network.
- Even if gradient descent can avoid local minima and can get to a low training error, it does not guarantee good generalization because of the high capacity of the network.
- There also is the problem of vanishing gradients. When propagating the error back through the network, the error is multiplied with the derivative of the tanh function and a weight. The derivative of tanh is never bigger than 1 and the weights are often low too. This decreases the size of the error that is propagated, and with many layers the error has almost vanished when it reaches the lowest layers.

Hinton and Salakhutdinov [2006] found a way to train SAEs successfully and efficiently, by learning them layer by layer: First an autoencoder is trained to create a shallow representation, and then that shallow representation itself is encoded by a new autoencoder, deepening the representation step by step.

Bengio [2009] explains the surprising effectiveness of this method as follows: Standard classification methods use some probabilistic model to train the relation between the input x and desired classification output y: $p(y|x)$. This relation between x and y is modeled directly – it does not use the information that the distribution of the input itself $p(x)$ has. But many classification problems are of the highly varying kind described above making the relation between y and x complex and modeling the distribution of $p(x)$ actually can tell us something about $p(y|x)$. SAEs do this by first learning to model $p(x)$, and use the encoding to learn $p(y|x)$ afterwards.

Drawbacks It now seems that deep networks should be better at every classification task, but this is certainly not the case. Recently, Coates et al. [2010] showed that shallow models can get state-of-the-art results in vision tasks using smart but simple modeling. We should therefore not assume that deep networks are always the best choice.

3.1.2 Training Stacked Autoencoders (SAE)

We train SAEs as follows: We start out by learning a one-hidden layer autoencoder, just like in the previous chapter. When the stopping criterion is met, we stop learning this layer.

At that point we add a new autoencoder, consisting of two layers, between the encoding and reconstruction layer. Then we use the original encoding layer to encode the input, and learn the new autoencoder to reconstruct this encoding – we present the encoding of the first layer as a data sample for the new layer. When the training of this new autoencoder has converged, we can again add a third autoencoder to reconstruct its encoding. This process can be repeated over and over again and is called stacking, which is where SAEs get their name.

When training, noise is not added over the sample directly, but over the encoding before it is used for training.

Lowering the Learning Rate To train the deep models more carefully, we will train the autoencoders a little more carefully than in the previous experiments. Instead of stopping after the increase of training error does not decrease anymore, we divide the learning rate by a factor of 4 and continue learning until the stopping criterion is met again. We repeat this process until the learning rate drops below a threshold.

Finetuning After the SAE has been trained to the desired depth, we start to finetune the model. In finetuning, we use backpropagation on the full autoencoder network. Finetuning trains all layers in the autoencoder at the same time. As was stated in the beginning of the chapter, if we would have used backpropagation on the full autoencoder on a randomly initialized neural network we would get bad results: The neural networks would get stuck in suboptimal minima and have poor generalization. But after using the process of stacking autoencoders, the neural network has a good initialization to start from.

Not using finetuning results in big training and test errors, as will be shown in the following experiment.

3.2 Experiments

Until now we only trained shallow models of faces and parts of the face – the models consisted of only one hidden layer. The reconstructions were quite good, but a lot of hidden nodes were needed. The hidden layer needed a size of about 100 for a good reconstruction of parts of the face. For reconstruction of full faces, even a size of 200 was not enough to retain all features.

So we know that we can successfully reconstruct and encode the faces, but the space we are encoding to is big. We want to represent faces in a smaller space that contains high-level features. Instead of directly encoding in a smaller space we will use SAEs. We hope that the SAEs learn higher level representations that describe the parts of the face better than shallow models.

3.2.1 Experiment 3: Stacked Autoencoders

In this experiment we will train SAEs on patches of the face. These SAEs are deep models, in contrast to the shallow models that we trained in the previous experiment. We will start again with an input layer of $n_{input} = 400$, that will get as input the pixel values of the 20 by 20 patch. We will then stack autoencoders that gradually decrease in layer size: First we encode to a 200 size layer, then that encoding will be encoded to a 100 size layer. After that 75, 50 and 30 size encoders gradually decrease the encoding, until finally a 20 size encoder creates the final encoding. The resulting SAE has layer sizes $[400, 200, 100, 75, 50, 30, 20]$, a deep model.

In Between Encodings To see if training deeper models will also gradually create deeper representations, we will also train a 20 size encoding layer at every step of training the deep encoder: We will train a $[400, 200, 20]$ autoencoder, then a $[400, 200, 100, 20]$ autoencoder, and so on. Every one of the autoencoders will be encoding to a 20 size encoding layer, and we will look at *what* these encoding layers actually encode.

Parameters The patches are again created using a window positioned over the annotated positions. We deviate the window 0 to 2 pixels, chosen uniformly, in any direction to create some position invariance.

The parameters used are: $\lambda_{l1} = 0.01$, $\lambda_{noise} = 0.01$, $\lambda_{factor} = 0.01$ and $\eta_{stop} = 0.0$. The learning factor is lowered from $\lambda_{factor} = 0.01$ to $\lambda_{factor} = 0.00004$ using the process described in section 3.2.1.

Expectations

We expect a lower reconstruction error for the deeper SAE because, in contrast to the shallow models, they slowly decrease the size of the encoding layer. This could 'introduce' them gradually to the small sized encoding.

We expect deeper level encodings to encode higher level features. 'High level' is not a clearly defined concept, but we expect that the difference will become clear when we look at the encoding.

3.2.1.1 Results

Tables 3.1 and 3.2 show the reconstruction errors for the encoders trained to different number of layers on the eyes, nose and mouth. The results are averaged over 12 experiments.

First of all we notice that, the deeper we stack autoencoders, the larger the reconstruction error becomes. This is to be expected as the size of the encoding becomes increasingly small and has less dimensions to encode the input. If we then finetune these SAEs, all reconstruction errors obtain about the same reconstruction errors – there seems to be no difference between shallow and deep autoencoders in this case. This is contrary to what we expected – we expected deeper networks to have smaller reconstruction errors.

The mouth has the smallest reconstruction errors, suggesting that it is easiest to encode, although the differences between the different parts of the face are small.

Reconstructions Figure 3.1 shows reconstructions for a shallow and a deep autoencoder. Just as the reconstruction error suggest, there are no noticeable differences in reconstructions by the deep or shallow encoder.

Layer Activations To study what the layer actually has learned, we show the activations of the hidden layers at several layers: We choose the hidden layer we want to investigate and set all activations to 0. Then we evaluate the network, creating a base reconstruction. We change the value of one of the activations from 0 to some small number and evaluate the network again. The difference of this reconstruction and the base reconstruction show the contribution of that node of the hidden layer.

Figure 3.2 shows the layer contributions for the eye encoder, Figure 3.3 the same for the nose encoder, and Figure 3.4 for the mouth encoder. The contributions of the hidden layers contain noise at the low level and become increasingly smooth when moving to deeper layers. The noise might be caused by the noise that is prevalent in the patches themselves. The lower layers respond to low complexity patterns, reacting to small specific areas strongly – showed by a white or black patch in the contributions. Also simple edge detectors can be seen at the lower layers, they react negatively and positively to two adjacent regions – showed by adjacent white and black regions in the contributions.

There is no noticeable difference between the shallow and deep 20 unit encoding layers. Again contrary to what we expected. The deeper layers do not seem to encode higher representations.

face part	layers	train RMSE	test RMSE
eye left	[400,200]	0.0329 +- 0.0002	0.0337 +- 0.0003
eye left	[400,200,100]	0.0402 +- 0.0002	0.0412 +- 0.0003
eye left	[400,200,100,75]	0.0460 +- 0.0002	0.0470 +- 0.0003
eye left	[400,200,100,75,50]	0.0714 +- 0.0008	0.0725 +- 0.0008
eye left	[400,200,100,75,50,30]	0.1300 +- 0.0015	0.1317 +- 0.0014
eye left	[400,200,20]	0.0932 +- 0.0002	0.0943 +- 0.0003
eye left	[400,200,100,20]	0.1086 +- 0.0005	0.1103 +- 0.0004
eye left	[400,200,100,75,20]	0.1233 +- 0.0007	0.1246 +- 0.0010
eye left	[400,200,100,75,50,20]	0.1604 +- 0.0028	0.1618 +- 0.0030
eye left	[400,200,100,75,50,30,20]	0.1717 +- 0.0016	0.1726 +- 0.0019
eye left	[400,200,20]finetuning	0.0901 +- 0.0001	0.0911 +- 0.0003
eye left	[400,200,100,20]finetuning	0.0900 +- 0.0002	0.0910 +- 0.0003
eye left	[400,200,100,75,20]finetuning	0.0900 +- 0.0001	0.0911 +- 0.0003
eye left	[400,200,100,75,50,20]finetuning	0.0902 +- 0.0001	0.0913 +- 0.0002
eye left	[400,200,100,75,50,30,20]finetuning	0.0902 +- 0.0002	0.0913 +- 0.0003
eye right	[400,200]	0.0322 +- 0.0002	0.0328 +- 0.0003
eye right	[400,200,100]	0.0393 +- 0.0002	0.0401 +- 0.0003
eye right	[400,200,100,75]	0.0457 +- 0.0002	0.0467 +- 0.0003
eye right	[400,200,100,75,50]	0.0717 +- 0.0007	0.0728 +- 0.0008
eye right	[400,200,100,75,50,30]	0.1275 +- 0.0015	0.1285 +- 0.0016
eye right	[400,200,20]	0.0930 +- 0.0003	0.0937 +- 0.0003
eye right	[400,200,100,20]	0.1080 +- 0.0004	0.1094 +- 0.0006
eye right	[400,200,100,75,20]	0.1233 +- 0.0005	0.1247 +- 0.0009
eye right	[400,200,100,75,50,20]	0.1601 +- 0.0026	0.1615 +- 0.0027
eye right	[400,200,100,75,50,30,20]	0.1707 +- 0.0011	0.1720 +- 0.0013
eye right	[400,200,20]finetuning	0.0896 +- 0.0002	0.0905 +- 0.0003
eye right	[400,200,100,20]finetuning	0.0895 +- 0.0001	0.0904 +- 0.0002
eye right	[400,200,100,75,20]finetuning	0.0896 +- 0.0001	0.0904 +- 0.0003
eye right	[400,200,100,75,50,20]finetuning	0.0895 +- 0.0002	0.0905 +- 0.0002
eye right	[400,200,100,75,50,30,20]finetuning	0.0895 +- 0.0002	0.0905 +- 0.0003

Table 3.1: Results of training deep autoencoders on several parts of the face. (Continued in 3.2)

face part	layers	train RMSE	test RMSE
mouth	[400,200]	0.0329 +- 0.0003	0.0333 +- 0.0003
mouth	[400,200,100]	0.0399 +- 0.0002	0.0406 +- 0.0003
mouth	[400,200,100,75]	0.0453 +- 0.0001	0.0459 +- 0.0003
mouth	[400,200,100,75,50]	0.0606 +- 0.0002	0.0611 +- 0.0004
mouth	[400,200,100,75,50,30]	0.1196 +- 0.0022	0.1198 +- 0.0020
mouth	[400,200,20]	0.0880 +- 0.0002	0.0883 +- 0.0003
mouth	[400,200,100,20]	0.0929 +- 0.0002	0.0937 +- 0.0005
mouth	[400,200,100,75,20]	0.1069 +- 0.0009	0.1072 +- 0.0006
mouth	[400,200,100,75,50,20]	0.1548 +- 0.0020	0.1550 +- 0.0019
mouth	[400,200,100,75,50,30,20]	0.1694 +- 0.0023	0.1703 +- 0.0022
mouth	[400,200,20]finetuning	0.0858 +- 0.0001	0.0863 +- 0.0004
mouth	[400,200,100,20]finetuning	0.0859 +- 0.0001	0.0861 +- 0.0002
mouth	[400,200,100,75,20]finetuning	0.0859 +- 0.0002	0.0862 +- 0.0002
mouth	[400,200,100,75,50,20]finetuning	0.0859 +- 0.0001	0.0863 +- 0.0003
mouth	[400,200,100,75,50,30,20]finetuning	0.0860 +- 0.0002	0.0865 +- 0.0002
nose	[400,200]	0.0281 +- 0.0002	0.0285 +- 0.0002
nose	[400,200,100]	0.0363 +- 0.0002	0.0367 +- 0.0003
nose	[400,200,100,75]	0.0428 +- 0.0002	0.0433 +- 0.0004
nose	[400,200,100,75,50]	0.0627 +- 0.0004	0.0631 +- 0.0006
nose	[400,200,100,75,50,30]	0.1228 +- 0.0012	0.1228 +- 0.0014
nose	[400,200,20]	0.0903 +- 0.0002	0.0908 +- 0.0005
nose	[400,200,100,20]	0.1000 +- 0.0003	0.1006 +- 0.0004
nose	[400,200,100,75,20]	0.1158 +- 0.0011	0.1162 +- 0.0014
nose	[400,200,100,75,50,20]	0.1574 +- 0.0015	0.1585 +- 0.0016
nose	[400,200,100,75,50,30,20]	0.1663 +- 0.0012	0.1666 +- 0.0014
nose	[400,200,20]finetuning	0.0865 +- 0.0002	0.0870 +- 0.0003
nose	[400,200,100,20]finetuning	0.0865 +- 0.0002	0.0871 +- 0.0003
nose	[400,200,100,75,20]finetuning	0.0863 +- 0.0002	0.0869 +- 0.0003
nose	[400,200,100,75,50,20]finetuning	0.0866 +- 0.0001	0.0869 +- 0.0003
nose	[400,200,100,75,50,30,20]finetuning	0.0865 +- 0.0002	0.0870 +- 0.0003

Table 3.2: Results, continues from 3.1.

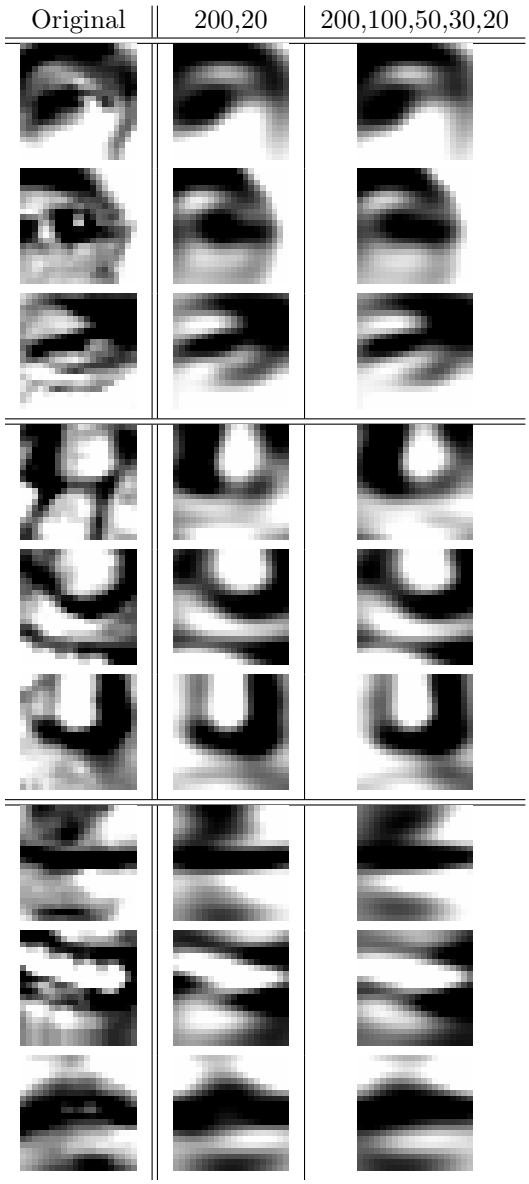


Figure 3.1: This figure shows the reconstructions of parts of the face for deep and shallow autoencoders.

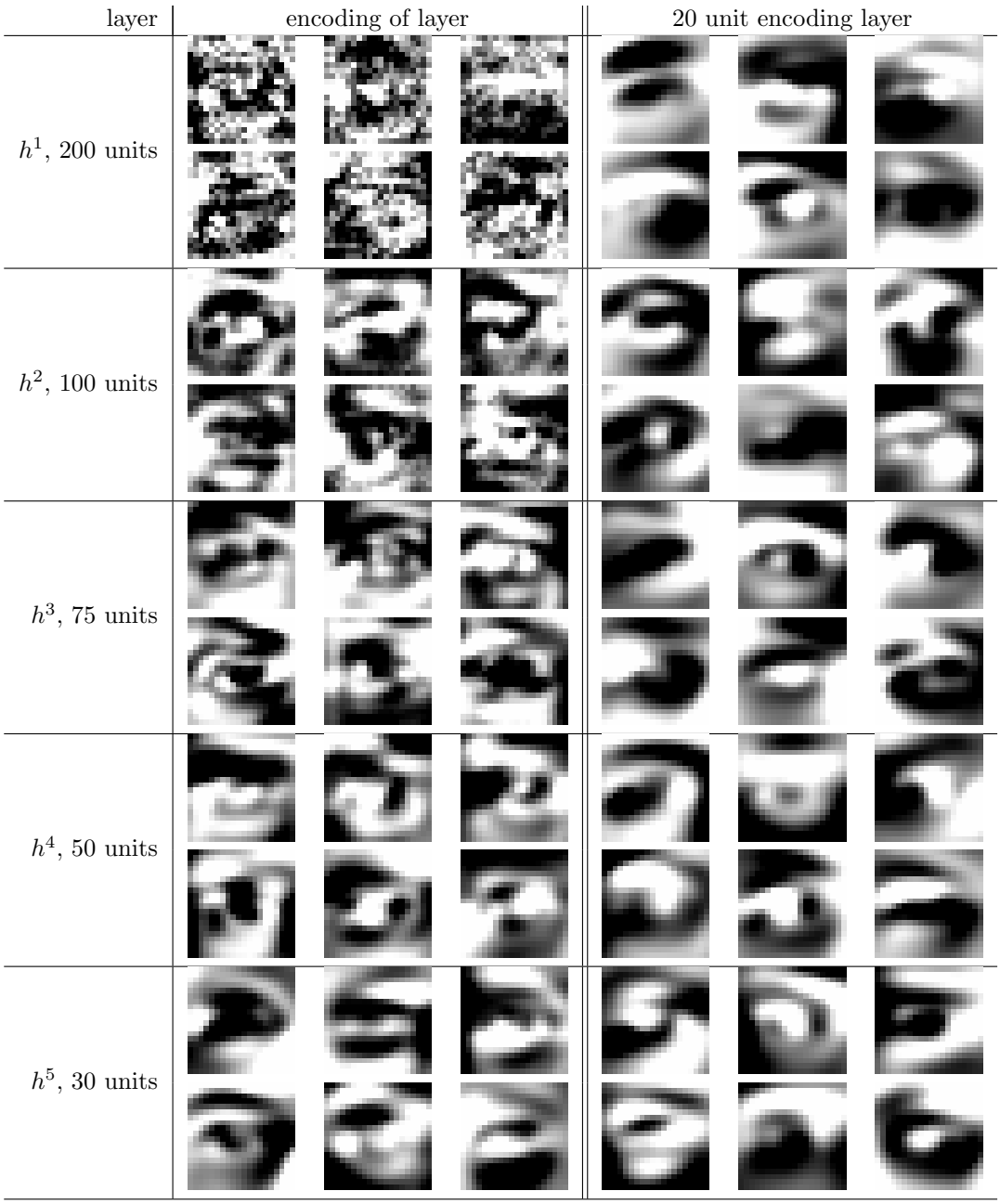


Figure 3.2: Layers of stacked autoencoders trained on eye-patches.

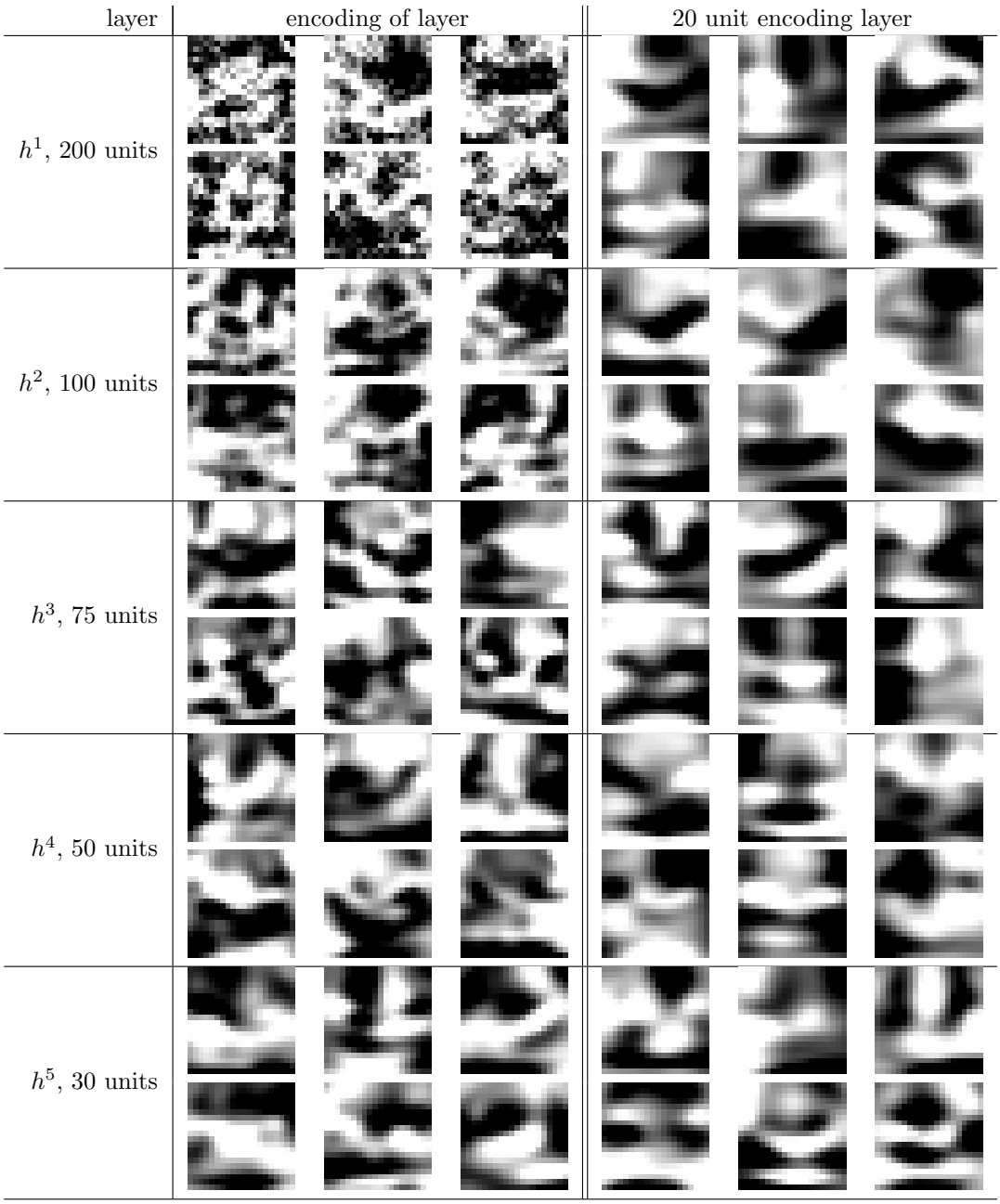


Figure 3.3: Layers of stacked autoencoders trained on nose images

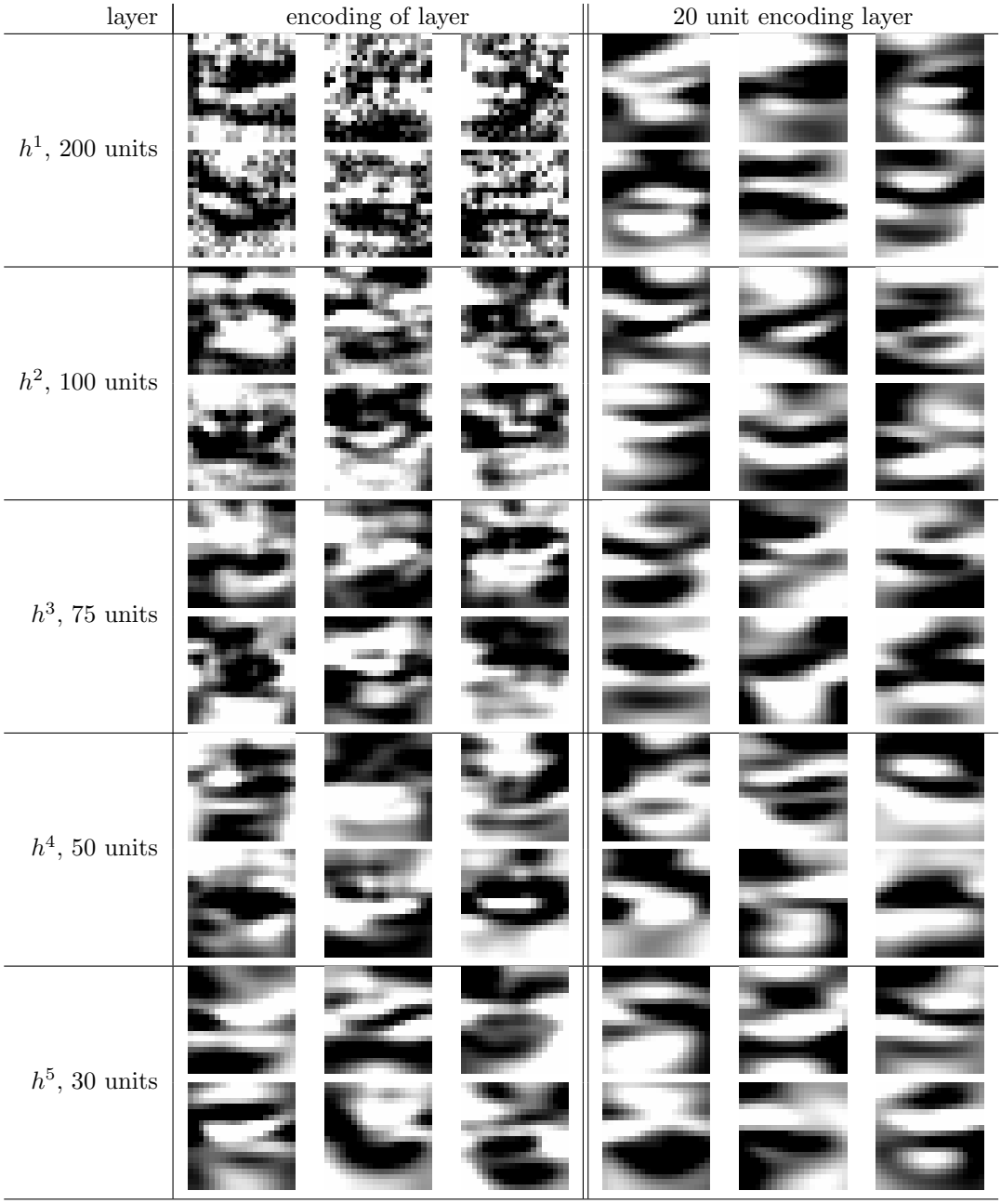


Figure 3.4: Layers of stacked autoencoders trained on mouth images

3.3 Discussion

In this chapter we compared deep and shallow autoencoders on the reconstruction of patches. The surprising result of our experiments was that both shallow and deep models perform the same.

Looking at the contributions of the nodes at every layer showed that nodes lying deeper in the model had contributions that look most like the parts the model encodes. So the models seemed to have the capability to gradually build up better representations, but this did not show in the final encodings. Several reasons for this result can be given. For one it is known that deeper models do not necessarily create better models, depending on the problem. It might be that the data we want to model just is not complex enough to benefit from deep representations.

Another explanation can be that the expressiveness of this kind of SAE is lacking. To see this, think of how a human would describe the picture of an eye-patch if he or she could only use a limited number of variables for the encoding. We would perhaps use one dimension to describe the size of the pupil, another to describe how much the eye-lid was raised, and so on – we would use high level variables to model the patch. The SAE however, despite it uses deep compositions of non-linear functions, still expresses patches as a combination of typical patches.

The combinations it can express are more complex than that of a simple linear model, but are still restricted. For example, there is no easy way to encode the translation or rotation of a patch in the SAE. To model such transformations efficiently we need three way interactions [Memisevic and Hinton, 2010]. In a three way interaction model, the activation of a node depends on the sum of products of a weight and the activation of two other nodes, instead of one node. This allows it to model transformations that SAEs can not.

In Between Finetuning We saw that with each new layer the $RMSE$ increased but after finetuning the final model all $RMSE$s were about the same. If we get to that same $RMSE$ by finetuning at the end of stacking our model, we should also be able to get that $RMSE$ at every point of the stacking. So a possible way to make better models would be to do finetuning after every layer we add. We did not do this in our experiments due to time constraints, but it would be interesting to see if that helps the final model.

Next step The next step is to extend the autoencoders in such a way that they can find the parts they need to encode on their own.

Chapter 4

Guiding

In the previous chapters we have built autoencoders that can encode parts of the face to a lower dimensional representation. However, these encoders knew the exact position of their respective parts by looking at the annotation. In this chapter we are going to create and train guided autoencoders (GAE) that can find their parts on their own.

Outline We will first explain how the GAE works and how it can be trained. Then we will do experiments to test the performance of GAEs. We first experiment with several RL parameters. Next we compare deep and shallow autoencoders for GAEs, showing that again there is no difference in performance. Next we will make the GAEs position aware and show that this increases performance. Finally we will experiment with the exploration of the actions of the GAE, where we show that increasing exploration improves performance.

4.1 Introduction

Up until now, the local autoencoders used the information from annotations to pinpoint the position of the parts they want to encode. For example, the eye encoder located itself at the exact point where the eye was annotated by the human and encoded that part. Requiring this kind of annotation for classification is unsatisfying as it removes all the advantages of automatic classification. If we still need humans to annotate new images to classify them, we might as well just ask them to classify it.

To create a fully automatic classification system we need to leave the human out of the classification loop. The goal of this chapter is to get rid of that limitation. We are going to train to guide autoencoders to their respective parts using reinforcement learning.

4.1.1 Interacting with Data

There are many imaginable ways to create a system that can find a specific part of the face in an image. There has been a lot of research in finding parts of the face - especially on finding the eyes. Face recognition systems try to find the eyes of a face image because a lot of information about the person is believed to be in it. For example, research that compared the contribution of several parts of the face to the performance of a classification task showed that the eyes contain the most information [Castrillón et al., 2010].

Approaches Some approaches to finding parts of the face can be seen as applying a function $f(\vec{x})$ over the image data x, which is evaluated and directly gives the predicted position of e.g. the eye as output. f is often a very carefully designed function comprising several different methods. As a

typical example, in a recent paper [Monzo et al., 2010], a pipeline of feature extraction and classification methods is trained to find the eye-positions. The thing to note here is that the classification is a one-way direction – the image data is given to and processed by the function and the function never looks back at it.

Other approaches use moving windows that scan the image and at every point try to decide if the part they are looking for is present. Still, the algorithm scans the image in a predetermined way and does not react to what it sees.

Our Approach Our approach will be different. We allow for interaction with the data in which a function interacts with the input $\vec{x}$ several times. This can be seen as using a function g such that $e_t = g(\vec{x}, e_{t-1})$ where x is again the image data and e_t is extra information which was calculated by function g at time t. At time $t = 0$ e_0 is initialized sensibly and g is evaluated, providing new information e, after which g is evaluated again using the new information. This goes on until a stopping criterion is met. The final prediction is done using a final processing function $h(e_{final})$ which processes the final information e_{final} and gives the prediction.

4.1.2 Guided Autoencoders (GAE)

The guided autoencoder is best viewed as an agent. An agent is an entity that resides in a state, can perform actions that (might) change this state, and gets rewards based on how good that action was.

Our agent is an autoencoder that looks at the image through a window. The state of the agent depends on the position of the window. The agent does not perceive the position directly, but uses an observation function to form the state, as will be explained below. The agent can move the window around, hopefully to position the window on the part it should look at; these are the actions of the agent. And finally, the reward can be defined based on the distance between the actual position of the window and the goal position. The bigger the distance, the less reward is given to the agent. The state, action and reward will be defined more precisely below.

The goal of the agent is simple, maximize the amount of reward. In the beginning however the agent does not have a clue what actions will give him the most reward and must learn this from experience. By trying out actions and looking at the reward the agent will learn which actions give the most reward.

4.1.3 Learning Framework

We need a formal learning framework to teach the agent to maximize its reward. This is created by defining the task as a Markov decision process and using a reinforcement learning algorithm. We will now formally define the Markov decision process and reinforcement learning (RL) algorithm [Kaelbling et al., 1996, Sutton and Barto, 1998].

4.1.3.1 Markov Decision Process

Let $s^* \in S$ denote a state taken from the state-space S. Let $s^* \in \mathbb{R}^k$ be a vector consisting of real values. Let A denote an action space. At time t an action $a_t \in A$ is chosen based on the state s_t (and not any of the previous states) that performs an operation, creating state s_{t+1}. A transition function $T : S, A \to S$ defines what state the system will be in after a certain action is chosen. The agent will be given a reward r_t based on this new state. The goal of the agent is to learn to choose the actions that maximize the expected long term reward.

Guiding To model our agent, we have to define possible states s, actions a and define the reward function R:

State The state s_{pos} of the agent is defined by the position of the autoencoder. Our agent does not see this state directly, but makes an observation using an observation function: $s = f_{observe}(s_{pos})$ where $f_{observe}$ is the observation function. This function allows the agent to 'look' through the window to the image. We will experiment with two observation functions:

position unaware encoder The position unaware GAE evaluates its autoencoder at the position s_{pos}. The resulting encoding will be its observed state vector s. This guiding agent does not perceive the position itself.

position aware encoder The position aware encoder uses the same encoding vector, with its x and y position added to it. This makes the agent aware of its position. The values of the position are normalized in the ranges of $[-0.5..0.5]$, where the upper-left of the image is defined by position $(-0.5, -0.5)$ and the bottom-right by $(0.5, 0.5)$. This keeps the position values in the range of other elements in the vector.

The use of the position in the observed state-vector might be viewed in two ways. On one side it can be viewed as 'cheating' as we humans do not posses absolute x and y positions of our gaze either. When we use the absolute position, we are actually using information given by the human that selected the face we are looking at, which would be against our principles.

Humans do however keep track of where they are looking at, and we use this information to guide our eye-movements – we have a general idea of which part of an object we are looking at. Also, a face-detection algorithm could have provided the selected faces, without human intervention, which would take the human out of the loop but still allows access to position information.

Action The actions a define the change in position of the center of the window. For example, an action vector with the values (0.1, -0.03) would move the center 0.1 units to the right and 0.03 units to the top. One unit is defined as the length of the image – a step of 1 unit is needed to jump from the left side to the right side of the image. The length of the steps is however constrained to 2.5 pixels. If we don't do this the agent becomes unstable and quickly moves out of the image.

Reward Choosing the reward function R is the hardest, as the ways to calculate a reward are limitless. For example, the eye-positions in the images are annotated. As we would like to find these positions, we give a higher reward the closer the center of the window is getting to the pupil. It seems natural to define the reward as a Gaussian with the center over the eye-position:

$$r_t = R(\vec{v_{guess}}) = \frac{1}{\sqrt{2\pi}} e^{-\frac{\|\vec{v_{goal}} - \vec{v_{guess}}\|}{2\sigma^2}} \tag{4.1}$$

This creates a smooth function that has an output in the range of [0..1]. It gives high reward at the goal position, but still a little bit reward as the agent closes in to this position. Unless stated otherwise, we use a σ of one pixel. This creates a small area of reward making sure the agents can't easily collect reward by moving closely around the goal position. The effect is that the agents have to be more precise.

4.1.3.2 Reinforcement Learning

RL learns a policy function $\pi : S \rightarrow A$ that chooses the actions based on the current state in such a way that the expected total reward $E[R|s_t] = \sum_t \gamma^t r_t$ is maximized. Here $0 \leq y \leq 1$ is a future reward discounting factor that biases the agent to get higher rewards sooner than later.

The standard RL algorithm learns the policy π by learning a value function $V : S \rightarrow R \in \mathbb{R}$ where R is the expected total reward given that the optimal policy π^* is followed from this state: $E[R|s, \pi] = V(s)$. The value function can be learned using the following function:

$$V_{t+1}(s_t) \xleftarrow{\beta} r_t + \gamma V_t(s_{t+1}). \tag{4.2}$$

A policy is defined from the value function V by choosing the action that puts the system in the state with the maximum expected reward: $\pi(s, a) = argmax_a V(T(s, a))$.

This algorithm requires that the transition function T is known. For most real applications however, this function is not known as they are both non-deterministic and not fully observable. To solve this problem, we define a function that maps from both a state and an action to the expected reward: $Q : S, A \rightarrow R \in \Re$ which is related to the expected total reward by: $E[R|s, \pi, a] = Q(s, a)$. Now the policy can be defined without the transition function T: $\pi(s) = argmax_{a'} Q(s, a')$.

This Q function needs to be learned. To do this, we observe that the expectation function can be defined recursively as follows: $E[R|s_t, \pi, a_t] = r_t + \gamma E[R|s_{t+1}, \pi]$. Where $E[R|s_{t+1}, \pi] = Q(s_{t+1}, \pi(s_{t+1}))$. Substitution then gives us: $Q(s_t, a_t) = r_t + \gamma Q(s_{t+1}, argmax_{a'} Q(s_{t+1}, a'))$. To prevent radical changes in Q-values by noise in the system, a learning factor λ is added to gently move the current Q-value to the new Q-value:

$$Q(s_t, a_t) \xleftarrow{\lambda} [r_t + \gamma Q(s_{t+1}, argmax_{a'} Q(s_{t+1}, a'))]. \tag{4.3}$$

This gives us the update rule which we can use to learn the optimal policy for our agent.

4.1.3.3 CACLA

To use the Q-function to choose an action, we have to evaluate the function for every possible action, and choose the one that has the highest expected reward. This is fine if the number of our actions is small, or finite at the least. But in this thesis we will have to deal with real-valued action- and state-spaces. This gives us an infinite number of actions to choose from, making the Q-function useless.

To deal with real-valued actions, we use an alternative RL algorithm, called CACLA [van Hasselt and Wiering, 2007], which extends Q-learning in two ways:

- Firstly, CACLA uses a neural network to approximate the value function, which is learned using the update formula 4.2. The neural network allows us to deal with continuous state spaces. It gives an approximation of the expected total reward. The neural network has one change to it: the output layer does not use the non-linear *tanh* function. This helps modeling the value-function better.

- Secondly, there is an action function $Ac : S \rightarrow A$ which maps from the current state to the optimal action, called the actor. It is learned using information from the value-function: If action a is chosen, bringing the system from state s_t to state s_{t+1}, we can calculate the expected reward: $r_t^* = V(s_t) - \gamma V(s_{t+1})$. Now we can calculate the difference between the actual reward r_t and this expected reward, also called the TD-error:

$$\delta_t = r_t - r_t^* = r_t + \gamma V(s_{t+1}) - V(s_t). \tag{4.4}$$

If $\delta_t > 0$ the observed reward was higher than expected. This means the chosen action performed better than we expected using our current policy. In that case, we want to update the actor towards the performed action. If $\delta_t <= 0$, we do not update the actor:

$$Ac_{t+1}(s_t) \xleftarrow{\alpha} a_t \text{ if } \delta_t > 0. \tag{4.5}$$

The conditional is called the critic, as it decides when the actor is learned. The Ac function is also approximated by a neural network that has a linear output layer, just as the value function.

Exploration In the training phase we do not have a learned actor Ac to choose our actions, so we need a way of choosing them. The critic classifies if an action a does better than expected, and the actor only learns something when that is the case. It is vital that we choose the actions a sensibly in such a way that we explore new actions, but also use what is already learned. We do this by sampling actions from a normal distribution centered around the predicted optimal action from the actor:

$$a_t \sim \mathcal{N}(Ac(s_t), \lambda_{exploration}) \tag{4.6}$$

This ensures that we take into account what is currently learned by the actor, but still allows for exploration.

4.1.4 Training Procedure

Just as the training of autoencoders, we train the GAE over several epochs. In one epoch, all training samples are visited once, in a random order. The training procedure for one epoch is as follows:

1. Get a sample – a face image – from the training set.
2. Put the guider in a starting state by setting it at the annotated goal position and pushing it away a certain amount (see below).
3. Let the guider take n_{steps} steps and learn from experience according to the procedure described above (section 4.1.3.3).
4. Move on to the next sample, until all samples are used and one epoch has passed.

4.1.4.1 Initializing the GAE

The GAE consists of the following components which are initialized as follows:

- *(Stacked) Autoencoder* We take autoencoders trained in previous experiments and use them for the GAE.
- *Neural Network for Value Function* The value function is approximated by a neural network and will be initialized as described in chapter 2.
- *Neural Network for Actor* The actor is a approximated by a neural network and will be initialized as described in chapter 2.

4.1.4.2 Gradually increasing search space

The agent has to learn a good value function V before it can start to make sensible decisions on learning the actor Ac. As can be seen in function 4.2 the update function of the value-function of state s_t is defined in terms of the value-function at state s_{t+1}. In other words, if the expected value at s_{t+1} is high, this will increase the expected value at s_t. The quality of this update depends on

how well the value-function is already modeling the expected value at s_{t+1}. The quality of updates given to $V(s_{t+1})$ again depend on the quality of $V(s_{t+2})$, and so on.

The problem is that in the beginning the value-function is a randomly initialized neural-network with no sensible output whatsoever. This gives a problem with updating the function. The only updates that have any meaning are the updates with high rewards r such that the reward dominates the updating value in function 4.2.

We use this information when learning by setting our begin-state s_0 very close to the goal position, guaranteeing that the rewards will be high, and slowly increasing the begin-distance from the goal position over time. This slowly introduces the value-function to the reward space and increases the average quality of value-updates, making learning more effective.

More formally, we sample our starting position from a uniform distribution centered around the goal position with deviation σ_{start}:

$$\vec{s}_{start,x} \sim \mathcal{U}(\vec{s}_{goal,x} - \sigma_{start}, \vec{s}_{goal,x} + \sigma_{start}) \tag{4.7}$$

$$\vec{s}_{start,y} \sim \mathcal{U}(\vec{s}_{goal,y} - \sigma_{start}, \vec{s}_{goal,y} + \sigma_{start}) \tag{4.8}$$

In our experiments, we start with a small area by setting σ_{start} to 2 pixels, and we gradually increase it to a size of 20 pixels over $n_{increase\_epochs}$. The area never covers the complete image and the guider is thus not trained to find its way back from every point on the image. By focusing the training on a smaller area we hope to increase the quality of the guiders.

Extended Training After $n_{increase\_epochs}$ have passed, we can do extended training for better results. We will do this for all experiments in this chapter, except for the first one where it is omitted due to computation constraints. In extended learning, we keep learning using the final σ_{start} and keep track of performance of the system by measuring its performance as described below (section 4.2.2). We look specifically at the increase in distance after pushing the GAE away 10 pixels from its goal position. We keep training until the performance decreases.

4.2 Experiments

We will now use the GAEs in the following experiments to see how well they perform.

4.2.1 Autoencoder for GAE

As stated before, we will take autoencoders trained in previous experiments and use them for our GAEs. Unless stated otherwise, we will take the deep autoencoders trained in experiment 3, consisting of layers: $[400, 200, 100, 75, 50, 30, 20]$.

4.2.2 Measuring Performance

To measure the performance we use a simple method. We start by taking the goal position and push it away an $\lambda_{testdeviation}$ amount of pixels from this position in a random direction (care is taken that the position does not lie outside of the image). We then allow the GAE to take n_{step} steps to find the goal position. We then look at the position where the guiding agent ends up and use it to calculate a performance measure.

A first logical choice would be to use the average between the end-state of the guider – where it ends up – and the goal position. This is a simple measure to compare the performance between guiders, but by aggregating all distances it also removes a lot of information. We can not see where the guiding encoders end up. The average can also be skewed by outliers – guiding encoders that take a wrong path and wander completely off.

To get a better idea of *where* the guiders end up, we also look at the specific distances: We count the guiders that end up within 2, 5 and 10 pixels of the goal position. This gives an idea what proportion of GAEs really find their parts and how many completely wander of.

Flow Faces For a qualitative view of the behaviour of the guided encoders we create flow faces. These are created by taking an image from the dataset and applying a grid over the image. The points in this grid are starting-points for our guided encoder and from there we trace lines following the path of the encoder. To get smooth lines we draw the line with some transparency and decrease transparency as the path progresses, giving a sense of direction of the path.

We also give small random perturbations to the starting positions to get a more natural distribution of starting positions. Otherwise the rigidness of the grid would create artifacts when the lines are drawn. The flow-faces give a good feeling of where the attractors in the images lie and what heuristics the guided encoders use to find their objectives.

4.2.3 Experiment 4: Finding the right RL parameters

There are several parameters that define the behaviour of the reinforcement algorithm. To test their effects, we will first train a GAE to find the left eye.

Parameters We are going to vary the α, β and γ parameters and look at their effect on the final performance. We set $n_{increase\_epochs} = 100$ and don't use extended training. We use $\lambda_{exploration} = 1.3$ pixels and use $n_{steps} = 30$. These settings for $\lambda_{exploration}$ and n_{steps} will be used throughout this chapter. We split up the dataset again in a test and train set, with $\eta_{portion} = \frac{2}{3}$.

Expectations We expect that lowering the γ value trains agents that use less steps to get to the goal, as future rewards are more discounted. This can however create more erratic behaviour which might prove unstable and give worse results. We expect the β value to be more of influence than the α value: A value for β that is too low or too high will result in too slow or too fast and unstable learning of the value function. Without a good value-function, the predictions of the critic are useless making learning of the actions ineffective no matter how well the α value was chosen.

4.2.3.1 Results

Tables 4.1 and 4.2 show the results of testing GAEs on the test set, with different RL parameters. The results are averaged over 6 runs.

We see that in almost half of the cases the GAE find the goal position within 2 pixels distance after being pushed away 10 pixels. This drops to around 7% after being pushed away 20 pixels. This large drop is most likely due to the fact that the window reaches 10 pixels wide from the center, and the GAE therefore can not see its goal after being pushed 20 pixels.

If we look at the effect of γ, we see that a lower gamma produces better results. A lower γ means that the future reward is more discounted and the agent is more time-constraint to get reward. This forces the agent to move quicker to the reward. This makes sure the agent does not take detours that take many steps. This possibly reduces the chance that the agent gets lost in the image explaining the better results.

Figure 4.1 shows flow faces for $\gamma = 0.90$ and $\gamma = 0.99$. The area that is attracted towards the eye – the basin of attraction – is bigger for $\gamma = 0.90$. The effects of α and β are not very clear. A lower α seems to give better results, because the actions are trained more carefully, but high standard deviations make comparisons hard. For the following experiments we will use $\alpha = 0.003$, $\beta = 0.003$ and $\gamma = 0.90$ as this seems to perform well.

Attractors In these first examples of flow faces, we can see the typical behaviour of GAEs. There seem to be attractors in the image, that attract GAEs that are close. These attractors correspond to certain features, that differ depending on the nature of the part that we trained on. The path towards the attractor is not necessarily a straight line – more complex flows occur that make sure the GAEs approach the attractors from a certain angle. We also see that most attractors pull the GAEs away from the goal part, causing it to wander off.

			$deviation = 10px$			
gamma	alpha	beta	increase	$< 2px$	$< 5px$	$< 10px$
0.9	0.003	0.003	-5.3079 +- 0.0803	0.5023 +- 0.0102	0.2111 +- 0.0055	0.1107 +- 0.0036
0.9	0.003	0.01	-5.1768 +- 0.0994	0.4935 +- 0.0057	0.2084 +- 0.0070	0.1184 +- 0.0050
0.9	0.003	0.03	-5.2419 +- 0.1131	0.5046 +- 0.0058	0.2084 +- 0.0061	0.1111 +- 0.0066
0.9	0.01	0.003	-5.0788 +- 0.0964	0.4939 +- 0.0055	0.2000 +- 0.0053	0.1211 +- 0.0021
0.9	0.01	0.01	-5.0810 +- 0.1517	0.5015 +- 0.0073	0.1958 +- 0.0046	0.1253 +- 0.0070
0.9	0.01	0.03	-5.0709 +- 0.0783	0.4931 +- 0.0084	0.1989 +- 0.0064	0.1188 +- 0.0091
0.9	0.03	0.003	-4.9078 +- 0.0677	0.4716 +- 0.0075	0.2073 +- 0.0038	0.1272 +- 0.0064
0.9	0.03	0.01	-4.4982 +- 0.2357	0.4517 +- 0.0128	0.2038 +- 0.0055	0.1307 +- 0.0060
0.9	0.03	0.03	-5.1297 +- 0.0540	0.4847 +- 0.0044	0.2149 +- 0.0042	0.1188 +- 0.0012
0.95	0.003	0.003	-5.1665 +- 0.0994	0.4904 +- 0.0065	0.2042 +- 0.0067	0.1284 +- 0.0073
0.95	0.003	0.01	-5.3190 +- 0.0753	0.4916 +- 0.0082	0.2188 +- 0.0046	0.1245 +- 0.0074
0.95	0.003	0.03	-5.1911 +- 0.1051	0.4789 +- 0.0095	0.2153 +- 0.0040	0.1222 +- 0.0087
0.95	0.01	0.003	-5.0346 +- 0.0350	0.4778 +- 0.0052	0.2119 +- 0.0058	0.1215 +- 0.0060
0.95	0.01	0.01	-4.8612 +- 0.1494	0.4625 +- 0.0076	0.2096 +- 0.0065	0.1372 +- 0.0097
0.95	0.01	0.03	-5.0720 +- 0.1762	0.4743 +- 0.0150	0.2165 +- 0.0051	0.1284 +- 0.0074
0.95	0.03	0.003	-5.0554 +- 0.1430	0.4801 +- 0.0069	0.2172 +- 0.0016	0.1218 +- 0.0044
0.95	0.03	0.01	-4.9843 +- 0.1197	0.4594 +- 0.0117	0.2211 +- 0.0107	0.1310 +- 0.0081
0.95	0.03	0.03	-4.8793 +- 0.1022	0.4732 +- 0.0100	0.2100 +- 0.0045	0.1215 +- 0.0064
0.99	0.003	0.003	-4.9767 +- 0.1368	0.4552 +- 0.0058	0.2149 +- 0.0052	0.1375 +- 0.0015
0.99	0.003	0.01	-4.8314 +- 0.1312	0.4640 +- 0.0079	0.2226 +- 0.0075	0.1203 +- 0.0083
0.99	0.003	0.03	-5.0693 +- 0.0947	0.4548 +- 0.0118	0.2487 +- 0.0065	0.1322 +- 0.0078
0.99	0.01	0.003	-4.5771 +- 0.1213	0.4391 +- 0.0092	0.2241 +- 0.0061	0.1337 +- 0.0070
0.99	0.01	0.01	-5.1351 +- 0.1164	0.4690 +- 0.0070	0.2264 +- 0.0080	0.1222 +- 0.0023
0.99	0.01	0.03	-4.9154 +- 0.0777	0.4962 +- 0.0051	0.2077 +- 0.0083	0.1138 +- 0.0029
0.99	0.03	0.003	-4.7115 +- 0.1315	0.4513 +- 0.0088	0.2157 +- 0.0050	0.1349 +- 0.0045
0.99	0.03	0.01	-4.7334 +- 0.2984	0.4391 +- 0.0193	0.2349 +- 0.0055	0.1253 +- 0.0067
0.99	0.03	0.03	-4.9533 +- 0.1468	0.4705 +- 0.0120	0.2180 +- 0.0050	0.1345 +- 0.0083

Table 4.1: Results of trained GAEs trained to find the left eye are shown for varying parameters. The GAEs are tested by being pushed away 10 pixels in a random direction. The increase of distance in pixels is shown under 'increase' (lower is better).

			$deviation = 20px$		
gamma	alpha	beta	$< 2px$	$< 5px$	$< 10px$
0.9	0.003	0.003	0.0651 +- 0.0018	0.0391 +- 0.0031	0.0444 +- 0.0015
0.9	0.003	0.01	0.0743 +- 0.0043	0.0368 +- 0.0058	0.0387 +- 0.0037
0.9	0.003	0.03	0.0640 +- 0.0062	0.0452 +- 0.0037	0.0368 +- 0.0025
0.9	0.01	0.003	0.0613 +- 0.0052	0.0368 +- 0.0029	0.0460 +- 0.0028
0.9	0.01	0.01	0.0720 +- 0.0021	0.0444 +- 0.0026	0.0494 +- 0.0027
0.9	0.01	0.03	0.0709 +- 0.0065	0.0414 +- 0.0021	0.0406 +- 0.0026
0.9	0.03	0.003	0.0697 +- 0.0045	0.0395 +- 0.0038	0.0479 +- 0.0044
0.9	0.03	0.01	0.0885 +- 0.0082	0.0433 +- 0.0042	0.0494 +- 0.0038
0.9	0.03	0.03	0.0605 +- 0.0081	0.0368 +- 0.0036	0.0421 +- 0.0025
0.95	0.003	0.003	0.0605 +- 0.0026	0.0467 +- 0.0047	0.0395 +- 0.0027
0.95	0.003	0.01	0.0552 +- 0.0043	0.0322 +- 0.0039	0.0375 +- 0.0033
0.95	0.003	0.03	0.0651 +- 0.0050	0.0414 +- 0.0031	0.0533 +- 0.0054
0.95	0.01	0.003	0.0651 +- 0.0031	0.0406 +- 0.0023	0.0421 +- 0.0044
0.95	0.01	0.01	0.0571 +- 0.0069	0.0368 +- 0.0057	0.0540 +- 0.0034
0.95	0.01	0.03	0.0770 +- 0.0054	0.0425 +- 0.0060	0.0429 +- 0.0040
0.95	0.03	0.003	0.0617 +- 0.0038	0.0487 +- 0.0052	0.0467 +- 0.0037
0.95	0.03	0.01	0.0513 +- 0.0042	0.0364 +- 0.0033	0.0375 +- 0.0025
0.95	0.03	0.03	0.0835 +- 0.0065	0.0479 +- 0.0047	0.0525 +- 0.0069
0.99	0.003	0.003	0.0452 +- 0.0030	0.0276 +- 0.0025	0.0444 +- 0.0022
0.99	0.003	0.01	0.0705 +- 0.0026	0.0398 +- 0.0043	0.0590 +- 0.0042
0.99	0.003	0.03	0.0448 +- 0.0058	0.0375 +- 0.0050	0.0333 +- 0.0024
0.99	0.01	0.003	0.0567 +- 0.0048	0.0307 +- 0.0025	0.0517 +- 0.0037
0.99	0.01	0.01	0.0525 +- 0.0057	0.0379 +- 0.0032	0.0452 +- 0.0039
0.99	0.01	0.03	0.0563 +- 0.0047	0.0364 +- 0.0036	0.0333 +- 0.0026
0.99	0.03	0.003	0.0678 +- 0.0055	0.0460 +- 0.0047	0.0510 +- 0.0072
0.99	0.03	0.01	0.0533 +- 0.0072	0.0372 +- 0.0052	0.0379 +- 0.0020
0.99	0.03	0.03	0.0490 +- 0.0022	0.0375 +- 0.0045	0.0375 +- 0.0042

Table 4.2: Results of trained GAEs are shown for varying parameters. The GAEs are tested by being pushed away 20 pixels in a random direction.

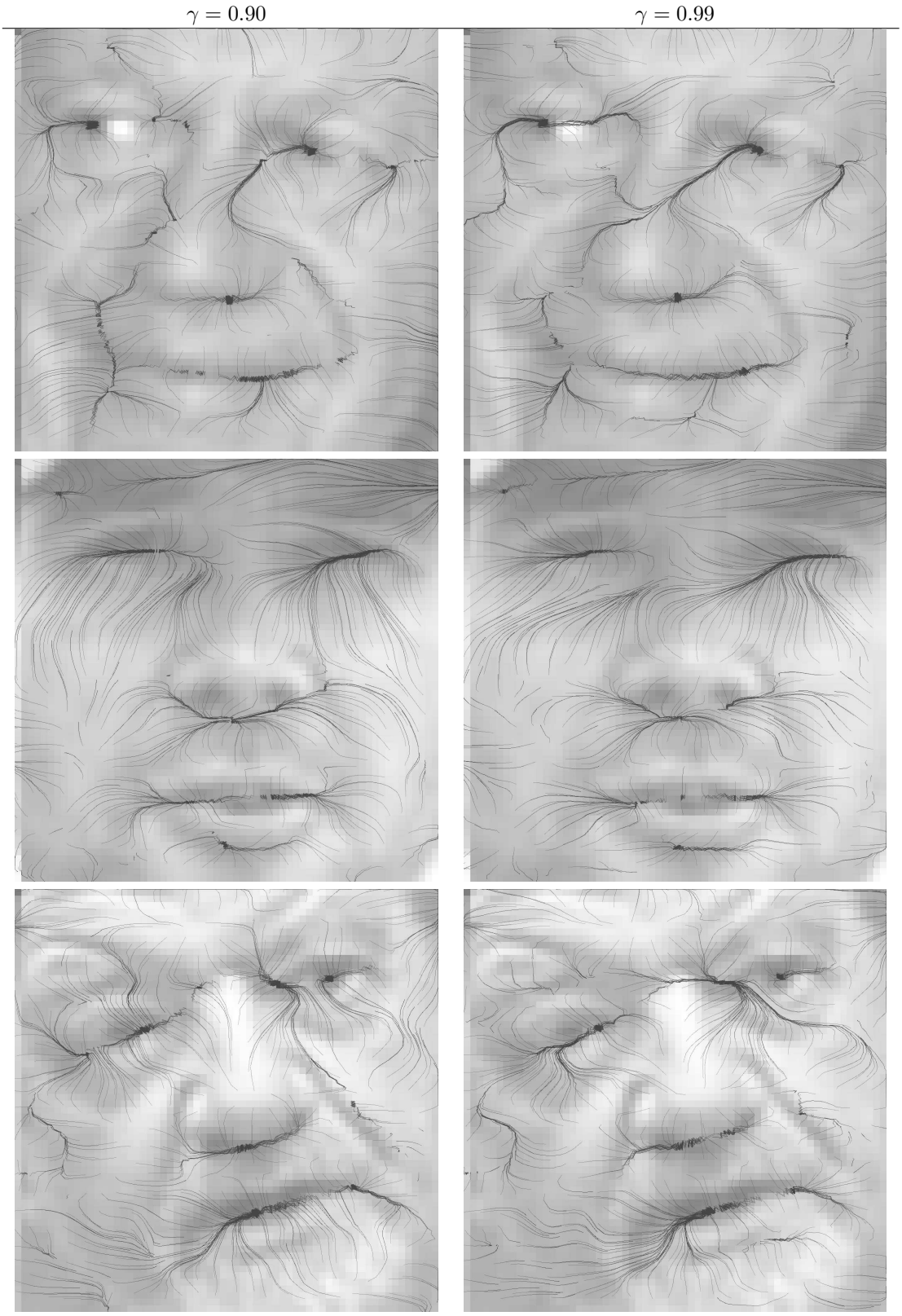


Figure 4.1: Several typical flow images trained to find the left eye, with different discount factors.

4.2.4 Experiment 5: Deep vs Shallow Guided Autoencoders

In the previous chapter we did not find differences between the encodings of deep and shallow autoencoders. It is however possible that there were differences in the encoding that we were unable to see with the tools we used. We therefore want to compare both autoencoders in a guiding experiment to see if the encoding perhaps leads to different results. We will at this point also train the GAEs on all face parts and compare their performance.

Autoencoders We will use the autoencoders trained in experiment **??**. We take shallow autoencoders with layers: $[400, 200, 20]$, deeper autoencoders with layers:$[400, 200, 100, 75, 20]$ and the deepest autoencoders with layers: $[400, 200, 100, 75, 50, 30, 20]$.

Parameters We use $\alpha = 0.003$, $\beta = 0.003$ and $\gamma = 0.90$. We also use extended learning.

Expectations We expect that this experiment will confirm our conclusion that our deep autoencoders do not produce better encodings and expect the same performance in the different conditions.

4.2.4.1 Results

The results are shown in table 4.3. The results are averaged over 3 runs.

The first thing we notice is that there is no significant difference in performance between the deep and shallow guided encoders, as we expected. We notice big differences in performance between experiments on the eye, mouth and nose. In the eye experiment, in the cases where the guider is pushed away 10 pixels, roughly between 44% and 48% find back the position of the pupil within 2 pixels or less. For the nose guider this is only roughly between 21% and 23% of the cases, and for the mouth guider even between 5% and 7%.

A possible cause is that the eye has a very well defined position, the pupil, making both annotation precise and guiding easier. For the nose it is harder, we chose the bottom of the nose as annotation position as it is the most stable point of the nose. Still it is not as precise as the pupil of the eye. The case for the mouth is the hardest, as the annotation position, the center of the mouth, is really hard to annotate precisely. In more than half of the cases, about 54%, the guider that is supposed to find the mouth wanders of in the wrong direction.

When we push away the encoders 20 pixels from their goal position, the performance lowers significantly. Please note that the windows reach 10 pixels in width and length taken from the center of the window. So pushing them away for 20 pixels does not allow them to even see their goal position. Still in about 10% of the cases the guiders that have to find the eye reach their goal within 5 pixels. For the nose guider this number is less than 2% and for the mouth guider less than 1%.

Flow Faces The flow faces of deep and shallow GAEs are shown in Figure 4.2 (left eye), 4.3 (right eye), 4.4 (nose) and 4.5 (mouth).

By looking at the flow faces we can make general interpretations of the strategies used by the guiders. These interpretations should not be seen as scientifically proven, but are useful for describing the perceived behaviour of the guiders.

The paths of the eye guiders show a preference for parts that are darker than the surroundings, for example shadows around the nose. This darker path is then followed until the guider finds a spot that is locally dark and resembles an eye. We note that the eye guiders are very distracted by eye-brows which can look very much like eyes in these low resolution pictures.

The nose guider seems to follow paths that are locally light as the area around the nose is ofter darker because of shadows. When the path becomes suddenly dark, it stops as this is often the

bottom of the nose. However in many cases it is hard to decide when to stop because of noses that are not very noticeable, and often the guider stops too early. Also if the guider does not start on a part of the nose, the chance of not finding the nose at all seems to be very high.

The mouth guider clearly has the most trouble of finding its goal. As long as it starts in the area that we would call the mouth, it finds its way to the center, but quickly wanders away otherwise. In the third face from the top, it does not even seem to find the mouth. The features of that particular mouth are very unclear however, which might explain this behaviour.

	$deviation = 10px$			
auto encoder	increase	$< 2px$	$< 5px$	$< 10px$
eye left				
$[400, 200, 100, 75, 20]$	-5.2619 +- 0.1088	0.4531 +- 0.0087	0.2429 +- 0.0072	0.1277 +- 0.0076
$[400, 200, 100, 75, 50, 30, 20]$	-5.0988 +- 0.0966	0.4398 +- 0.0127	0.2404 +- 0.0091	0.1413 +- 0.0035
$[400, 200, 20]$	-5.2165 +- 0.0491	0.4713 +- 0.0056	0.2332 +- 0.0068	0.1195 +- 0.0029
eye right				
$[400, 200, 100, 75, 20]$	-5.1772 +- 0.0536	0.4570 +- 0.0051	0.2156 +- 0.0055	0.1384 +- 0.0061
$[400, 200, 100, 75, 50, 30, 20]$	-5.1108 +- 0.1078	0.4513 +- 0.0088	0.2286 +- 0.0108	0.1344 +- 0.0051
$[400, 200, 20]$	-5.1870 +- 0.0924	0.4695 +- 0.0097	0.2115 +- 0.0059	0.1277 +- 0.0043
mouth				
$[400, 200, 100, 75, 20]$	3.1952 +- 0.3231	0.0542 +- 0.0046	0.1126 +- 0.0058	0.1637 +- 0.0106
$[400, 200, 100, 75, 50, 30, 20]$	2.9813 +- 0.4049	0.0501 +- 0.0047	0.1147 +- 0.0066	0.1885 +- 0.0134
$[400, 200, 20]$	2.1524 +- 0.1726	0.0618 +- 0.0033	0.1178 +- 0.0060	0.1990 +- 0.0116
nose				
$[400, 200, 100, 75, 20]$	-0.1575 +- 0.1144	0.2038 +- 0.0059	0.1504 +- 0.0065	0.0741 +- 0.0028
$[400, 200, 100, 75, 50, 30, 20]$	-0.3041 +- 0.1071	0.2089 +- 0.0091	0.1632 +- 0.0044	0.0649 +- 0.0022
$[400, 200, 20]$	-0.6423 +- 0.0356	0.2222 +- 0.0027	0.1801 +- 0.0067	0.0549 +- 0.0020

Table 4.3: Results of experiments with deep and shallow GAEs.

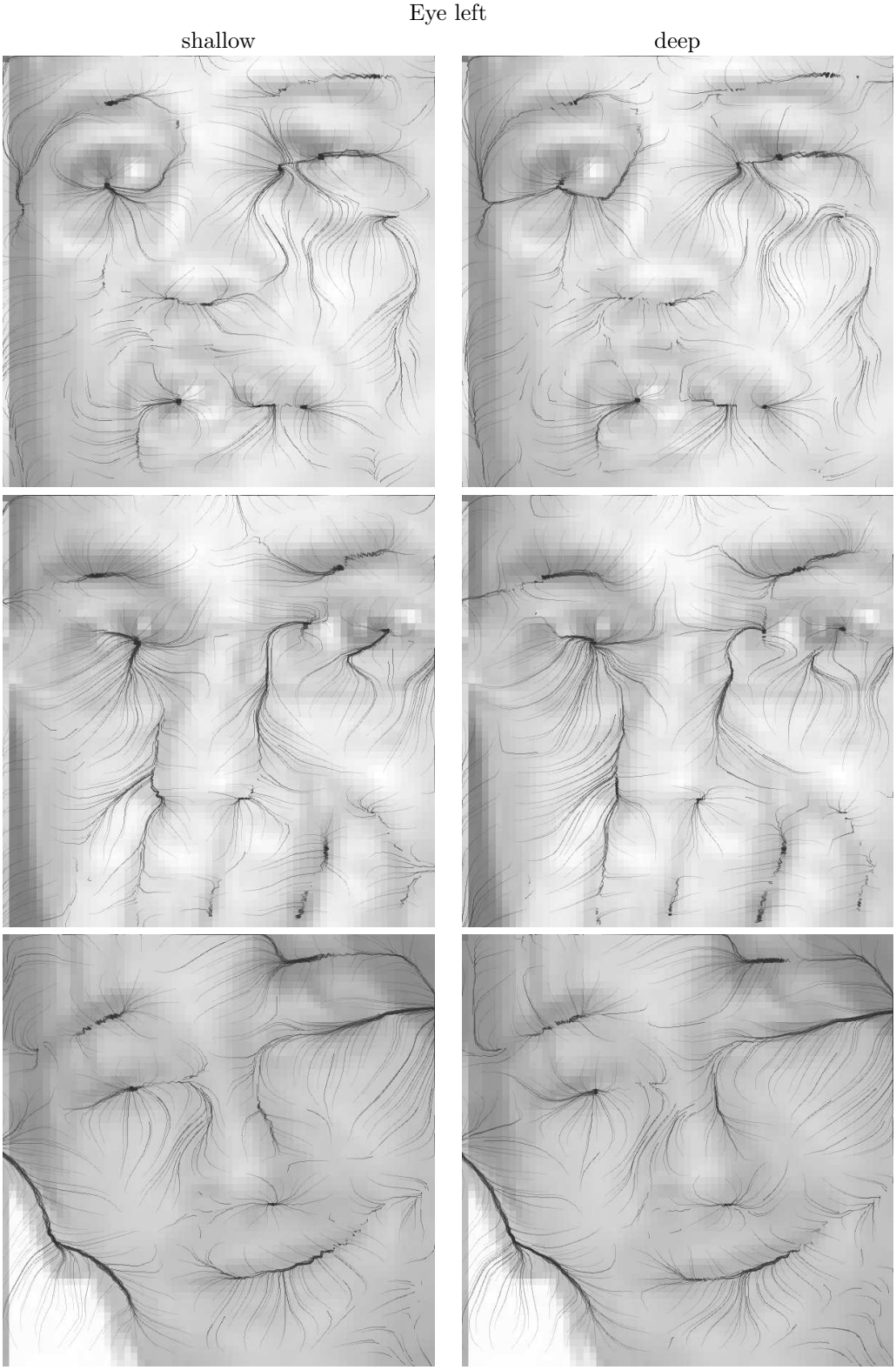


Figure 4.2: Flow images of deep and shallow autoencoders, trained to find the left eye.

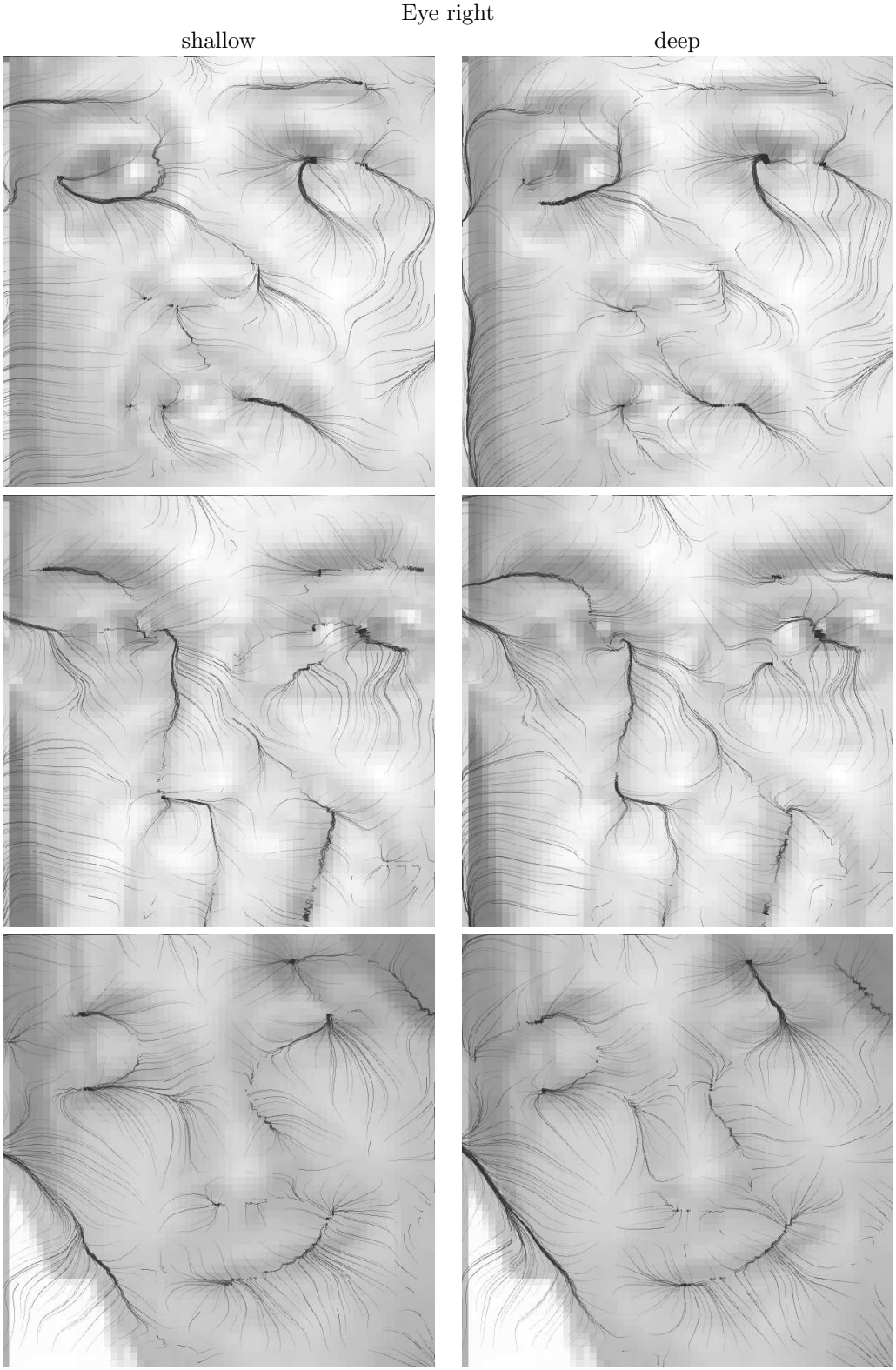


Figure 4.3: Flow images of deep and shallow autoencoders, trained to find the right eye.

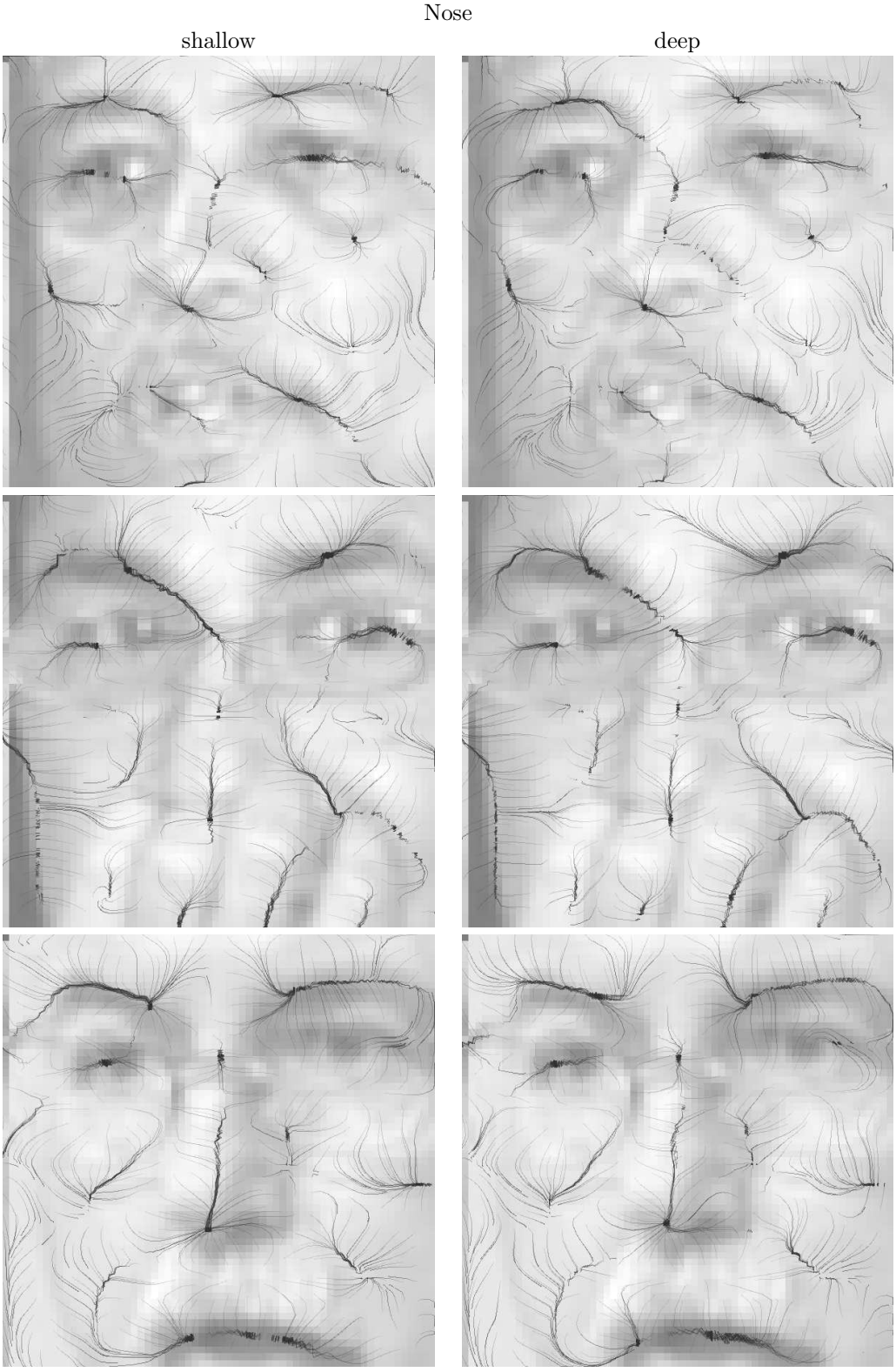


Figure 4.4: Flow images of deep and shallow autoencoders, trained to find the nose.

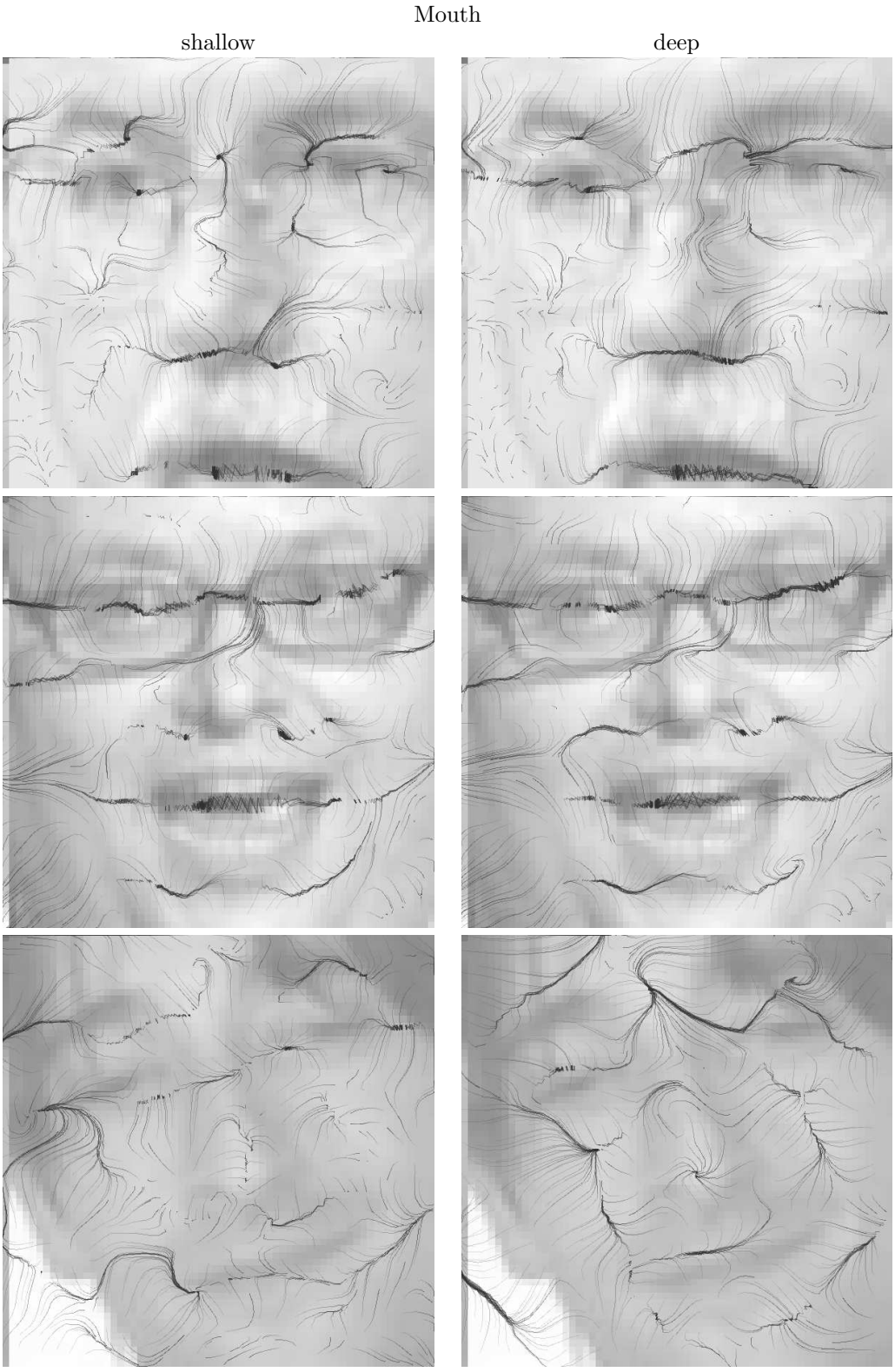


Figure 4.5: Flow images of deep and shallow autoencoders, trained to find the mouth.

4.2.5 Experiment 6: Position Aware GAE

The results from the previous experiment show that local GAEs have a capability of finding their goals, but there are many pitfalls. In this experiment we are going to add the information of the position of the window with respect to the image, into the encoding. This makes the GAE aware of its position, creating a position aware GAE.

Adding Position The encoding will be expanded with two values, one for the x-position and one for the y-position. A value of $x = -0.5$ denotes that the center of the window is exactly on the left border of the image, a value $x = 0.5$ denotes the right border of the image. The values $y = -0.5$ and $y = 0.5$ denote the upper and lower border of the image respectively.

Parameters We use the same parameters as the previous experiment. Furthermore we use the deep autoencoder and use extended learning.

Expectations We expect that the position aware GAE dominates the position unaware autoencoder. Also we expect a big improvement in the ability of the guiders to find back the goal after being pushed away 20 pixels in which case it can't perceive the goal directly, by using the position information.

4.2.5.1 Results

Table 4.4 shows the results of position aware and unaware guiders. The results are averaged over 2 runs.

Adding position clearly increases that chances of the guiders to find their goals. Also as expected, the changes of finding the goal from a 20 pixel distance increase significantly. The position aware guiders however have problems finding the nose. The causes of this are unclear, but a possible explanation is that the position of the nose is more sensitive to differences in pose than other parts. Therefore the position of the nose is not as consistent and possibly so variable that it confuses the GAE.

Flow Faces Comparisons of flow faces are shown in figure 4.6 (left eye), figure 4.7 (right eye), figure 4.8 (nose) and figure 4.9 (mouth).

The position aware encoders seem to have a better 'sense' of where they should go when trying to find the eyes. Again, finding the nose is unexpectedly hard. The basin of attraction for the mouth guider increases with position awareness, but still gets confused by other parts, such as the nose.

		$deviation = 10px$			
face part	use pos	increase	$< 2px$	$< 5px$	$< 10px$
eye	0	-4.9680 +- 0.0761	0.4345 +- 0.0109	0.2582 +- 0.0123	0.1272 +- 0.0066
eye	1	-7.1261 +- 0.0643	0.5969 +- 0.0099	0.2418 +- 0.0141	0.1027 +- 0.0058
eye right	0	-5.3816 +- 0.1192	0.4736 +- 0.0118	0.2268 +- 0.0062	0.1280 +- 0.0034
eye right	1	-6.8508 +- 0.0785	0.5670 +- 0.0105	0.2444 +- 0.0074	0.1088 +- 0.0074
mouth	0	2.8096 +- 0.3123	0.0487 +- 0.0050	0.1253 +- 0.0049	0.2077 +- 0.0091
mouth	1	-1.0840 +- 0.1841	0.1180 +- 0.0044	0.2103 +- 0.0092	0.2264 +- 0.0104
nose	0	-0.1207 +- 0.1833	0.1939 +- 0.0156	0.1609 +- 0.0089	0.0716 +- 0.0054
nose	1	-0.4761 +- 0.0479	0.1920 +- 0.0057	0.1674 +- 0.0074	0.0751 +- 0.0102

		$deviation = 20px$		
face part	use pos	$< 2px$	$< 5px$	$< 10px$
eye	0	0.0709 +- 0.0043	0.0502 +- 0.0040	0.0448 +- 0.0025
eye	1	0.3540 +- 0.0190	0.1651 +- 0.0122	0.1100 +- 0.0049
eye right	0	0.0448 +- 0.0023	0.0291 +- 0.0027	0.0475 +- 0.0026
eye right	1	0.3126 +- 0.0374	0.1452 +- 0.0178	0.0920 +- 0.0079
mouth	0	0.0011 +- 0.0007	0.0057 +- 0.0005	0.0264 +- 0.0042
mouth	1	0.0349 +- 0.0044	0.0782 +- 0.0118	0.1092 +- 0.0139
nose	0	0.0126 +- 0.0047	0.0165 +- 0.0037	0.0218 +- 0.0022
nose	1	0.0103 +- 0.0020	0.0077 +- 0.0014	0.0180 +- 0.0013

Table 4.4: Results training position aware and position unaware GAEs.

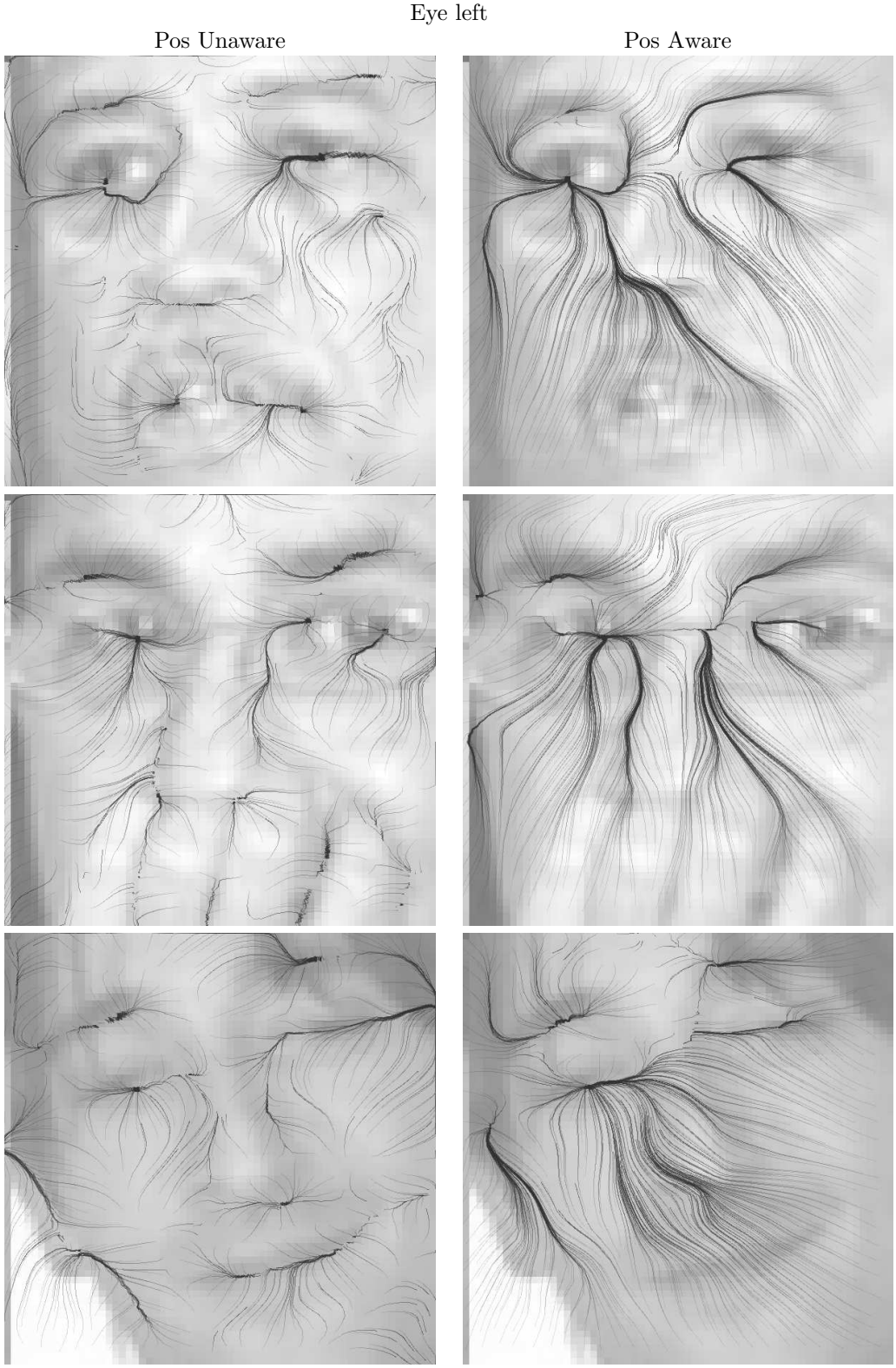


Figure 4.6: Flow images of position-aware and -unaware autoencoders, trained to find the left eye.

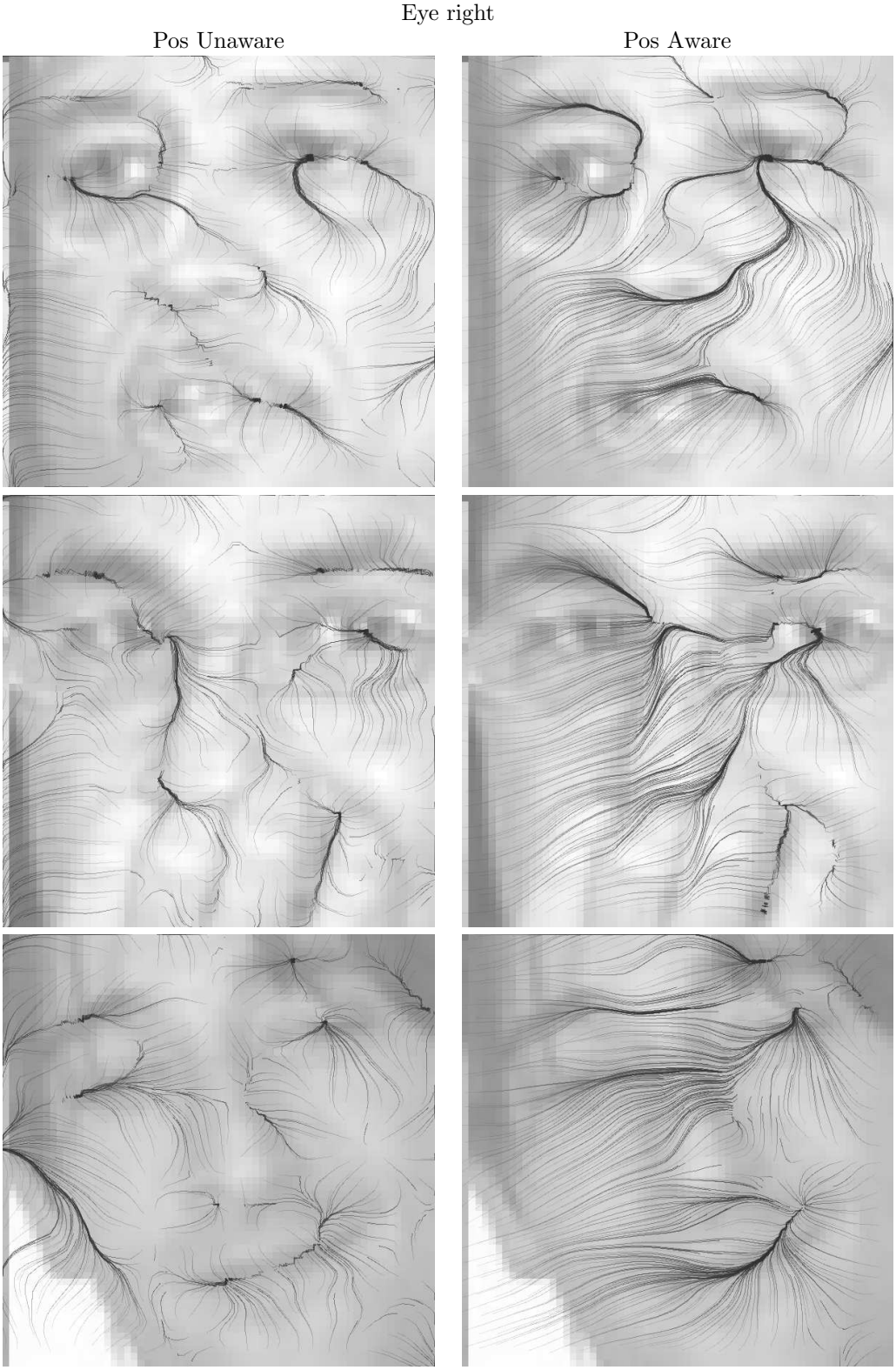


Figure 4.7: Flow images of position-aware and -unaware autoencoders, trained to find the right eye.

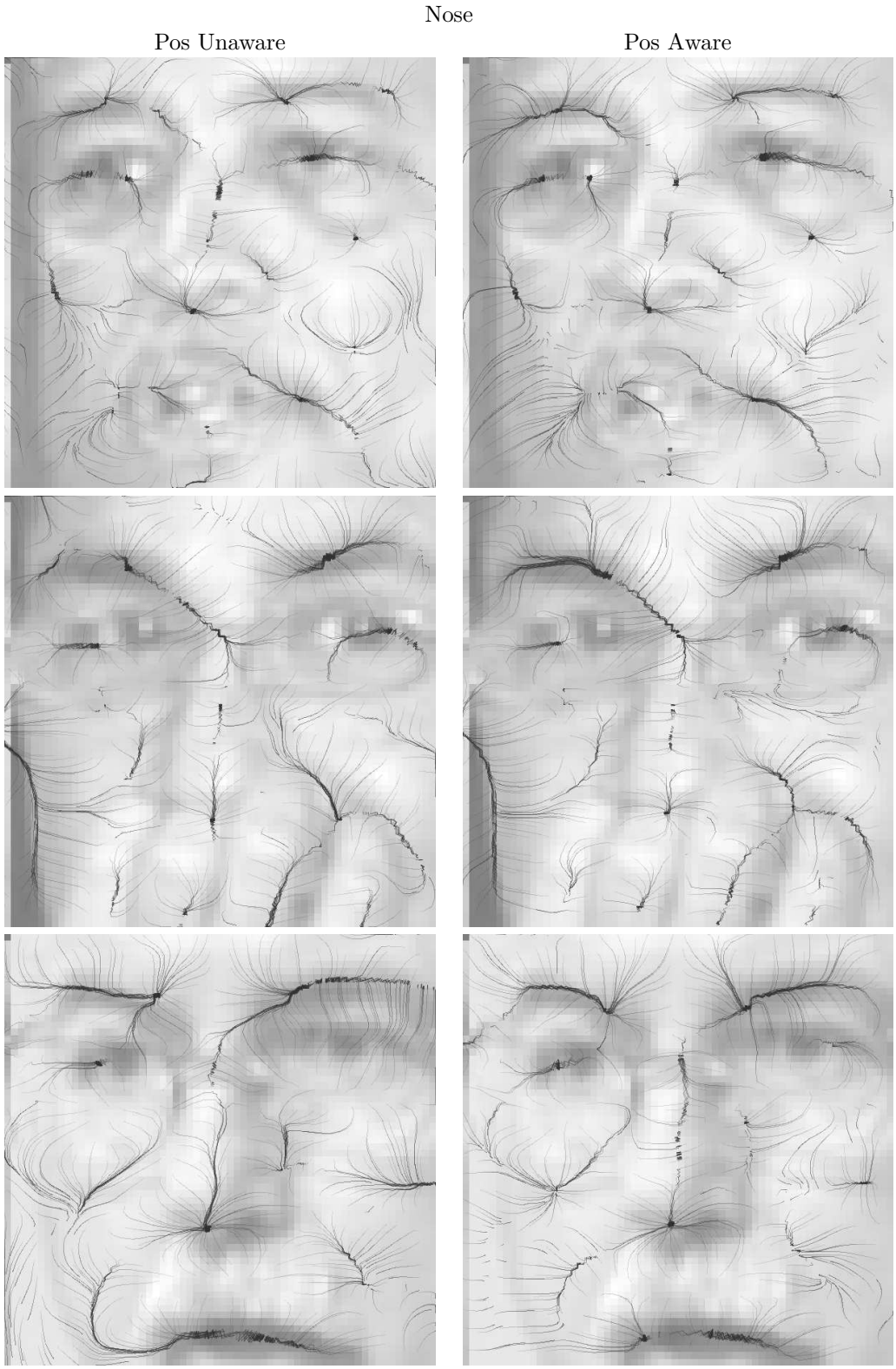


Figure 4.8: Flow images of position-aware and -unaware autoencoders, trained to find the nose.

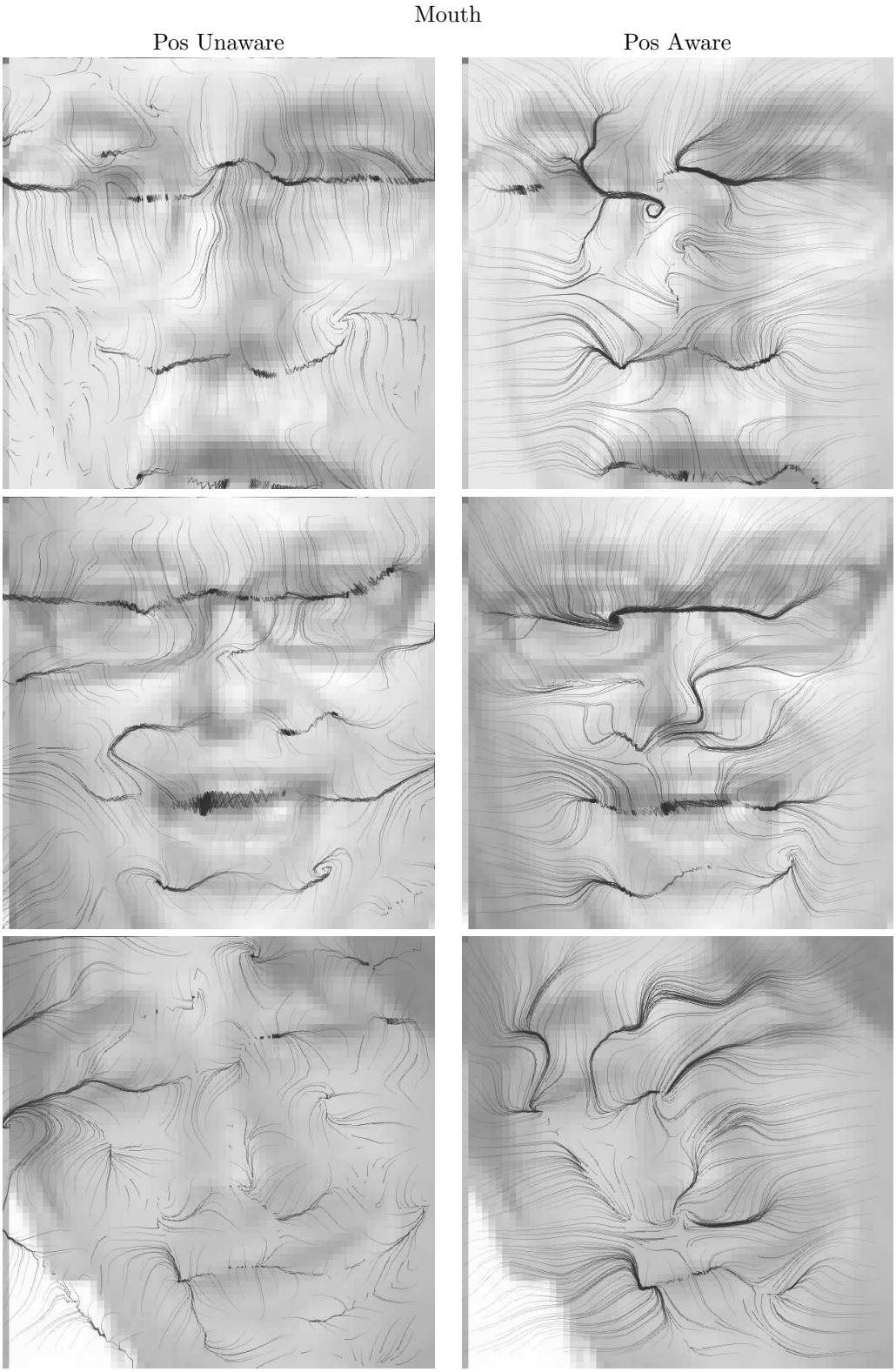


Figure 4.9: Flow images of position-aware and -unaware autoencoders, trained to find the mouth.

		$deviation = 10px$			
face part	exploration	increase	$< 2px$	$< 5px$	$< 10px$
eye	0.01	-4.2600 +- 0.1504	0.3577 +- 0.0098	0.2621 +- 0.0151	0.1636 +- 0.0085
eye	0.02	-5.2184 +- 0.1074	0.4347 +- 0.0097	0.2640 +- 0.0088	0.1308 +- 0.0048
eye	0.04	-5.6198 +- 0.0814	0.4584 +- 0.0047	0.2839 +- 0.0077	0.1025 +- 0.0034

		$deviation = 20px$		
face part	exploration	$< 2px$	$< 5px$	$< 10px$
eye	0.01	0.0487 +- 0.0034	0.0431 +- 0.0029	0.0473 +- 0.0039
eye	0.02	0.0619 +- 0.0053	0.0473 +- 0.0031	0.0521 +- 0.0030
eye	0.04	0.0496 +- 0.0037	0.0450 +- 0.0025	0.0379 +- 0.0014

Table 4.5: Effect of exploration on the performance of the guided encoder trained to find the eyes.

4.2.6 Experiment 7: Effect of Exploration

In this last guiding experiment we will investigate the effect of exploration on the GAE. We want to investigate the effect of the exploration parameter. We will vary the parameter $\lambda_{exploration} = [0.7, 1.3, 2.6]$ pixels and show the results on a position unaware GAE.

4.2.6.1 Results

Table 4.5 shows the results of this experiment. The results are averaged over 4 experiments. We see a positive relation between increasing the exploration and the performance of the guider. Possible reasons for this is that small explorations only explore the action space very locally to the existing behaviour. If the GAE uses bigger exploration however, it is easier for the GAE to get out of existing behaviours.

4.3 Discussion

In this chapter we trained and tested GAEs. In the first experiment we trained a GAE to find the left eye for several RL parameters. We saw that using a lower γ increases performance, while the effect of β and α seemed less clear.

In the second experiment we compared GAEs using deep and shallow autoencoders and trained them on all face parts. We saw no differences in performance between deep and shallow GAEs, as was expected from the results in the previous chapter.

The GAEs trained on the eyes performed the best. The nose and mouth GAE performed worse than the eye GAE, where the mouth GAE even increased distance from its goal position on average. A possible explanation, as noted in the result section of that experiment, are the less precise annotations. The position of the eyes can be defined precisely by defining its position as the center of the pupil. The nose and mouth do not have such clear features. For the nose we could think that the tip of the nose is a clear feature, and this was tried during annotation, but it became clear that this is a highly variable feature visibly. For the mouth it is even harder. We used the center of the mouth to define the position, but the center is recognized by humans by using the symmetry of the mouth – at the center itself there is no clear visual mark. This makes it very hard for a GAE to recognize the center and move in that direction.

In the third experiment we made the GAEs aware of its position. This increased the performance of the GAEs for all parts, except the nose GAE. A hypothesis we gave was that the nose position is so varied that it is not helping, but confusing the GAE. We should also note that the GAEs are not pushed further away than 20 pixels when training. Maybe position becomes more useful for GAEs if they are pushed further away.

In the fourth and last experiment we looked at the effect of exploration on performance. Increasing exploration increased performance, possibly because it is easier for a GAE to get out of 'habits'.

Qualitative Results Despite the problems of finding some parts of the face, the flow faces show us very interesting and promising results. The fact that GAEs can learn gradients in the difficult and varied images, and guide themselves to their goal parts is amazing on its own. This is especially true when we take into account that a GAE only sees very local information. Also, their autoencoders are trained to encode the specific goal part, but are not trained on any of its surroundings. Still the encodings give enough information to guide the GAEs.

Another interesting observation is that the form of the paths together convey information about the image. The basins of attraction for example could be used to divide up the image in interesting parts. The flow of the paths in this basin could be used to identify obstacles and interesting parts of that basin. We will look at ways to use this information in the discussion (chapter 6).

Chapter 5

Face Recognition

At this point we have trained shallow and deep autoencoders and extended them to GAEs that can find parts of the face. In this chapter we will create a strategy to use GAEs to classify properties about the faces.

5.1 Introduction

In this chapter we will conduct the final experiments to turn our GAEs to useful classification.

Outline We will first explain how the classes are annotated and how we combine the individually trained GAEs to one classifier. Then we experiment with the classifier, showing that out of the box the GAEs are not great classifiers, probably due to instability issues.

5.1.1 Annotation of Classes

We annotated two properties of the faces for classification. The first is the gender – whether someone is male or female – which is known to be a difficult problem. The second property is 'smiling' where we classify if someone is smiling or not. Annotation of the images is done by one subject.

The classification of smiling proved quite difficult. First of all, there are many types of faces that could be considered 'smiling' in different ways: The class is a natural dichotomy. However, we still need a consistent method of classification. Secondly, it can be very hard to see if someone smiling in some of the images, because of unclear pictures. To reduce these problems, we specified smiling a little more precise: We annotated a picture as a smile when we could clearly see a confident smile. If a picture could perhaps have a smile, but is was debatable, we annotated them as 'no smile'.

The annotation of the gender was easier, as that class is naturally a dichotomy. Still there were pictures where the gender was debatable. In such cases we gave our best guess.

5.1.2 Finding-Strategies

We tested the capabilities of the GAEs in the previous chapter. But the process we used still used annotation information – we initialized the guiders at the annotated positions, pushed them away and let them find their way back. We want to do classification now, which means we need a method that does not use the annotation position at all – we want to get the human out of the loop.

Our solution is as follows: We will first give a good estimation of the position of the part we want to find. This estimation will either be based on the average position or a guess of a trained neural network (see below) and puts the guided encoder in a good start position. From that position, the GAE will try to guide itself to the part.

Starting Position The previous experiments showed that GAEs are easily let astray if they start of too far from their goal parts. Therefore we need to put the GAE in a good starting position. We will use two estimations of the starting position:

Average We will estimate the average position of the individual parts and use that as a starting position. This will in many cases put the guider close to their parts, but of course is not adaptive in any way.

Neural Network We will train a neural network that gets the full image of the face as input and outputs the estimated position as output. This method can adapt to the orientation of the face.

5.2 Experiments

5.2.1 Experiment 7: Position estimates

Before we do classification, we will test how good our part finding method performs. We will show the average distance toward the goal for the mean and neural network estimated, before and after using the guiders. We will use both position aware and position unaware guiders for a full comparison.

Neural Network Estimate A neural network will provide estimates for the position of the parts. We chose the following layer sizes for the neural network: $[4096, 10, 2]$. It will be trained for 300 epochs using a learning rate of $\lambda_{factor} = 0.001$. These parameters proved to work well. We will train a separate neural network for each part.

Expectations We expect the neural network estimate with guiding to give the best results. Just using the mean without guiding is expected to give the lowest results. In both cases we expect guiding to give better results than not using guiding.

5.2.1.1 Results

Table 5.1 shows the average end distance toward the desired goal under different strategies, tested on the test set. In all cases, except the mouth, using guiding increases performance and the neural network estimate is better than the mean estimate. The neural network estimates are closer to the goal position than the mean, as expected. The mouth GAE already proved difficult to train in previous experiments, and this is no exception. Except for the result for the mouth, our expectations are met.

Pos Unaware				
Part	mean	mean + guiding	nn	nn + guiding
eye left	0.0509	0.0385	0.0385	0.0361
eye right	0.0537	0.0374	0.0415	0.0369
nose	0.0765	0.0477	0.0468	0.0401
mouth	0.0760	0.1090	0.0491	0.0974
Pos Aware				
Part	mean	mean + guiding	nn	nn + guiding
eye left	0.0509	0.0337	0.0394	0.0317
eye right	0.0537	0.0336	0.0412	0.0318
nose	0.0765	0.0471	0.0466	0.0402
mouth	0.0760	0.0910	0.0488	0.0810

Table 5.1: Results of different finding schemes to find the parts of the face are shown.

5.2.2 Experiment 8: Classification

In this final experiment we will classify properties of the face. We use the final estimates of the parts positions, as they were found in the previous experiment. Then we use the encoding of the GAEs from the different parts and concatenate them to one long vector: $\vec{encoding}_{full} = \vec{encoding}_{left\_eye} + \vec{encoding}_{right\_eye} + \vec{encoding}_{nose} + \vec{encoding}_{mouth}$. For the position unaware GAEs the encodings have length 20, thus the final encoding has 80 values. For the position aware GAEs the encodings have length 22, thus the final encoding has 88 values. We use the 1300 images from the annotated dataset.

Classifier The final encoding will then be used as input for a classifier that maps to one of the final classes. The classes that we want to estimate are dichotomies. We need to translate them to the real-values output of the neural network. We will translate the classes to 1 and -1 to denote both instances – for example male and female – when we train the network. When we want to do classification, we will estimate the class corresponding to 1 when the output is above zero, and the other class otherwise.

We will use a k-nearest-neighbour (kNN) classifier and a support vector machine (SVM) for the final classification. We train the kNN for $k = [1, 2, 3, 4]$ using 10-fold cross validation and choose the k that has the highest performance. We train the parameters for the SVM using a logarithmic grid search, again using 10-fold cross validation, and use the best performing parameters.

Dummy and Raw test For comparison, we show the results of two other classification schemes. One is the dummy-classifier which always guesses the most frequent class. The other is the raw classifier that is trained using the kNN and SVM as the other systems, but has the raw pixel values of the whole face as input-vector.

Expectations The quality of the full encoding depends on how close the GAE got to its desired part. If the GAE wanders of it will encode some random part of the face which will screw up the classification. We therefore expect the classification results to correlate with the results from the previous experiment. We expect the position aware GAEs to outperform the position unaware ones.

5.2.2.1 Results

In table 5.2 the results of the classification experiment are shown. The position aware GAEs positioned on the annotations using an SVM gave the highest performance on both gender and smile recognition.

Gender Recognition For gender classification, using the NN as a starting point and using guiding worked best for position unaware GAEs using SVM. For position aware GAEs, using the mean as starting point proved better. If guiding was not used, the classification did not score higher than the raw classifier.

Smile Recognition For smile recognition, using the NN without guiding worked best in both position aware and unaware GAEs. Surprisingly, the position unaware encoders trained using SVM have better performance than the position aware encoders. Also using guiding decreased performance in all cases for SVM classification. A possible explanation is that the bad guiding performance of the mouth GAE kills performance.

NN vs. Mean Also using the neural network as a first guess does not necessarily outperform using the mean as a starting point. In the 'smile'-classification task, using guiding actually lowers the performance.

Gender		
dummy	83.08	
	4-NN	SVM
raw	84.91	90.85
Pos Unaware	4-NN	SVM
correct	87.14	91.92
mean	85.64	90.08
mean + guide	87.18	90.92
NN	87.40	90.31
NN + guide	86.98	91.23
Pos Aware	4-NN	SVM
correct	88.03	92.15
mean	85.18	90.00
mean + guide	87.45	91.38
NN	87.09	90.77
NN + guide	87.70	91.31
Smile		
dummy	66.92	
	3-NN	SVM
raw	74.55	83.54
Pos Unaware	3-NN	SVM
correct	75.57	84.69
mean	72.39	79.31
mean + guide	72.09	76.54
NN	71.12	81.54
NN + guide	71.42	77.92
Pos Aware	3-NN	SVM
correct	75.30	84.85
mean	72.03	78.69
mean + guide	72.29	78.38
NN	70.45	82.15
NN + guide	72.82	80.08

Table 5.2: Results of classification experiments are shown.

5.3 Discussion

The classification performance of our systems are not as we expected. Especially the fact that the use of guiding has almost no impact on the recognition performance is unexpected. We think the main problem is that the guiding is too variable and unstable in its current state. The classifier does not get consistent encodings because the GAEs can get lost, and with no method to detect the quality of the guiding result it is defenseless against this unreliability. In the last chapter we will discuss methods to get rid of this instability.

The low performance of the classification scheme can also be attributed to several other causes.

The GAEs create local features – they have no way to do holistic processing. It only sees the nose, mouth and eyes and does not have a view on the full face. This reduces the capabilities of the system. For example, it cannot form an interpretation of the general form of the face. In the next chapter we will discuss an extension that can amend the classification scheme to use holistic processing.

We know that the mouth GAE is very unreliable and is probably the cause of lowering the performance for smile-detection – in which the mouth is especially important.

It is not clear why in gender classification the GAEs that start from the mean perform better than GAEs that start from the NN guess, while this is the other way around before using guiding.

Chapter 6

Conclusion / Discussion

In this chapter we will look back at the experiments, discuss the implications of the results and point out directions for future research.

6.1 Summary

Chapter 2 In the first experiment we trained autoencoders to encode and reconstruct images of faces. Reconstructing faces seemed to work, but a lot of detail was lost.

To retain more detail, we focused on reconstructing smaller parts of the face. The second experiment reconstructed parts of the face – the eyes, nose and mouth – that were annotated by hand. Patches of 20 by 20 pixels were taken at these positions and used as input for the autoencoders. As expected, these reconstructions were able to hold more detail than the first experiment. Firstly, they were centered over the part they wanted to reconstruct and did not have to model translations. Secondly, they could 'focus' on reconstructing a single part, reducing the variabilities that had to be encoded.

We experimented with sparsity using $l1$ minimization and used denoising to improve generalization. In our experiments, $l1$ minimization did not seem to have a positive effect, contrary to what was expected. Denoising did improve generalization.

Chapter 3 In this chapter we tried to model more complex variabilities by using SAEs. These SAEs are deep models that can express complex variabilities using compositions of non-linear functions. We trained SAEs on the patches, but contrary to what we expected, both normal 1-layer autoencoders and SAE performed equally well.

Chapter 4 In this chapter we created GAEs (GAE) – autoencoders that look through a window to the image and can move that window over the image. GAEs were trained to guide themselves to the parts of the face that they were trained to encode, using a reinforcement learning algorithm called CACLA.

We trained two kinds of GAEs: those that could only perceive the image through their window, but were oblivious to where they were in the image. And those that could see through their window and also were aware of their position in the image. As expected, the latter GAE performed the best.

The results showed that GAEs have the capability to find their parts. Especially the performance on finding the eyes is encouraging. However, GAEs can also be let astray. Especially the performance of the mouth GAE was worse than expected.

We also compared deep GAEs – that used SAEs – and shallow GAEs – that had one-layer autoencoders. Because the reconstruction performance of both models already was shown to be the

same in chapter 3, we did not expect better results for the deep models. The results even showed that shallow models performed slightly better.

In spite of some problems, the GAEs can calculate gradients based on the local information presented to them that in many cases guided them to their goals. This is quite amazing given the difficulty of the landscape they have to navigate through.

Chapter 5 In this chapter we did the final experiments that combined the models trained in previous chapters and turned them to a classification problem. Using purely the encoding of the GAEs allowed us to do classification, however the performance can not rival state-of-the-art face recognition.

6.2 Discussion

We will now discuss our findings and give directions for future research.

6.2.1 Main Results

We have done many experiments in this thesis. We describe the main results here.

- When training AEs, using $l1$ minimization to increase sparsity did not improve reconstructions in our experiments
- Adding noise to samples improved generalization of AEs.
- Looking at the flow faces we see GAEs learn gradients that create basins of attraction.
- Adding position to the encoding improves performance greatly in finding the parts, except for the nose.
- The results show that GAEs can navigate themselves to their goals quite well, when initialized properly.
- Using deep SAEs did not outperform shallow autoencoders in our experiments.
- Using a NN to provide an initial guess outperforms using the mean position, except for the mouth GAE.
- Direct classification with GAEs is possible but does not rival state-of-the-art systems.

Main Contributions The goal of this thesis was to create better representations of the data by increasing interaction between the model and the data. We achieved this goal by creating GAEs, a novel combination of reinforcement learning and autoencoders. The autoencoder of the GAE takes the data perceived through a window and encodes it to a smaller representation. Then the RL algorithm uses this representation to train the GAE to guide itself to its goal.

Amazingly, GAEs can learn to navigate themselves through the complex landscapes of a face image using only local information perceived through the window. This shows that AEs and RL can be a powerful combination. The fact that the autoencoders are only trained to reconstruct their parts and not its surroundings makes it even more impressive.

Advantages The GAE is a very general and flexible model – it does not have many parts:

A window which gives it a view on the data.

An AE which enables the GAE to represent the data from the window efficiently.

An Actor and Value function approximator which allow it to guide itself.

These parts can easily be replaced by new parts with other capabilities. The function approximators, the autoencoders and the window can all be replaced or expanded, as we will describe below.

Disadvantages

- To train GAEs we need annotated images where the goal parts are located by humans. This tedious work might be reduced by automated discovery of goals, as we will describe below.
- The results show that the GAEs need good initial guesses of the goal parts to guide themselves correctly. Otherwise they are pulled away from their goals by basins of attraction.
- Our current classification scheme using GAEs just outperforms a neural network trained with the raw images as input.

We will describe possible ways to reduce or remove these disadvantages below.

Other findings We also showed that using the deep representations of the SAE did not outperform shallow AEs on reconstruction of patches and guiding of GAEs. This was a surprising result, as we expected that the gradual reduction of information by the layers would allow GAEs to express 'higher' representations about the data.

Research Question Our research question was :"Can guided autoencoders be used successfully to create representations of faces and classify them?". From our results we conclude that this goal was achieved within certain limits. The GAEs are in general able to successfully create representations, provided that they are initialized close to their parts. Classification using GAEs is also possible, although their performance does not improve the state of the art.

6.2.2 Future Work

GAEs are very flexible models. We will discuss future directions of research to improve GAEs using this flexibility.

6.2.2.1 Lessons from the Experiments

The results from the experiments give us some clear directions that are highly likely to improve the performance of GAEs.

Adjusting the Window The performance of the mouth GAE was the lowest. We hypothesized causes of this. Firstly the mouth is annotated using a point in the middle of the mouth. There is however no clear feature that described the middle of the mouth, and the autoencoder is in many cases too small to encompass the full mouth. Secondly, the mouth can be very vague in this difficult dataset which makes the small window even more problematic. Luckily the window of the GAE is flexible and can easily be increased in size. In case of the mouth, a wider window that is able to fully encompass the mouth can also learn to watch for the corners of the mouth and find the annotation correctly.

The annotation itself can also be extended to describe the face more carefully. Instead of annotating the mouth with a point, we could describe it with a line. This would create a more descriptive and 'fair' annotation of the face, at the cost of more annotation time.

Encoding Surroundings Currently the AEs of the GAE are specifically trained to encode patches of a certain part. This makes sure that the encoding is precise when the GAE is very close to its goal, but it can give problems when the GAE is out of reach from its goal. Then it finds itself in parts of the image on which the AE is never trained before. We could increase the encoding of such parts by training the AEs on a more diverse set of patches that also include patches taken randomly from the image. This is likely to improve guiding in the surroundings around the goal parts.

Other Classification Schemes The current classification scheme using GAEs just outperforms a NN trained on the raw image data. We think that the reason why the performance of this simple raw classifier comes close to our system is that the raw classifier can process information holistically. Using the full face allows it to find dependencies that the GAEs can not see, such as the general shape of the face and shape of the jaw.

We could take two approaches to decrease this problem for GAEs. We can simply add more GAEs and train them on other parts of the face, such as the chin, creating a better covering of the face. We could also extend our classification scheme where we use the positions of the GAEs to normalize the image and put parts at fixed positions in the image. This distorted image could then be encoded by an extra autoencoder, which can process the full face holistically. Such a system would use the GAEs to get rid of the variance problems we see when we naively try to encode the raw image data. The resulting holistic encoding could be extended with the local GAE encodings of the parts. Such an encoding is likely to prove a good representation of the face.

Using the Value function The reinforcement learning algorithm did not only learn the GAEs how to act, but also learned them the amount of future reward they can expect – the value-function. This value function can be used to our advantage by evaluating it at the position where the GAE ends up. If it wandered of, it probably got stuck in some attractor that does not quite look like the desired part of the face. The value-function generally has a lower expectation at such points, and the output of the function could be used to test the 'goodness' of this final state. In the event of a low output from the final function, our classifier can decide to try again or just ignore the encoding.

Using Context The GAEs for the eyes, mouth and nose act completely independently. This makes GAEs very locally focused. We know that humans benefit from using as much information as possible, also using the context. In other words, we do face recognition holistically [Sinha et al., 2006].

We can incorporate context information by allowing the agents to interact with each other. In such a multi-agent infrastructure, one GAE which has found an eye could give other agents a good guess where the other eye might be. When the two eyes are found, good estimates of the position of the mouth and nose can be given. And an eyebrow GAE could prevent an eye GAE to confuse the eyebrow for an eye – something that occurs often with the current system. Such a system could have a better performance, because it can deal with situations that are now problematic.

6.2.2.2 Flexibility of GAEs

As said before, GAEs are very flexible because their parts can easily be changed or extended. We present possible promising extensions to GAEs.

Extending the Window The current GAEs are a little constraint. They can only translate the window they are looking through. It is however easy to extend the model to more exotic operations. The window could for example be allowed to turn, shear, grow or shrink. This could provide a natural way to create invariance to exactly these transformations, which would make implementing invariance very simple. We experimented with other transformations in our model, but quickly

found out that they are hard to control. This seems to be mainly because it is hard for the model to learn stabilizing feedback on some transformations.

If we allow for rotation transformations, for example, the GAE needs to model its actions in such a way that when it is in the desired position and it would turn to the right, the GAE wants to turn left, and vice versa. If the input space does not easily allow for such modeling, it will be like balancing yourself without a vestibular. We argued in a previous discussion (section 3.3) about possible models that could model rotations and translations more easily. We think that such models are needed to extend GAEs with such operations.

So in general, actions can be added easily, but care should be taken that they are implemented in such a way that they stay stable.

Extending the Autoencoder Currently, the pixel values are fed as raw data to the autoencoder. Image data however, is known to have a lot of local variabilities that can be modeled easily on a local scale, but become problematic if we want to model all pixels at once.

Instead of directly feeding the raw data, we could use a convolutional neural network that models these local variations. Such a network would model the data more naturally which makes it easier for a GAE to model. Convolutional networks are more biologically plausible and have been shown to produce very good results. They have even been used recently to improve the state-of-the-art [Cireşan et al., 2011]

Adaptive Reward Function Besides changing all the parts of the GAE itself, we can also experiment with the reward function. Actually, in exploring the possibilities of training GAEs in this research, we implemented a system that did not use annotations to train the AE of GAE. Instead, the reward was defined based on how well it could reconstruct the patch it was currently looking at. A good reconstruction provided bigger reward. However the most uninteresting and smooth parts are the easiest to reconstruct. To force the GAEs to encode interesting parts, the reward was divided by the 'easiness' of the patch, giving almost no reward to patches without features. The resulting system was too unpredictable to be used in this thesis. We think however that a looking more closely at other research in automated discovery, we can make such an approach work [Schmidhuber, 1991].

6.2.2.3 Currently Unused Information and Unexplored Applications

If we look at the flow faces, we see a lot of information about the image in the paths of the GAEs. There are several way to use that information.

We first have to identify the basins of attraction. This can be done by initializing the GAEs at a raster on the image and storing in these pixels where the position where GAEs end up. Two adjacent pixels belong to the same basin of attraction if they end up close to each other. These basins of attraction can then be characterized using certain features, such as the size of the basin or the average curvature of the paths in the basin. This information could be used to split up the image in regions which has applications in image segmentation and recognition.

Tracking Another application of GAEs is tracking over video data. Once a GAE has positioned itself on a certain part of the image, this part will be close but slightly moved in the next image. Because the last position is a good initialization for the new image, GAEs are likely to perform well in such tasks.

6.2.2.4 A General Perspective

Until now, we looked at GAEs as tools to find annotated parts of the image, where these parts correspond to complex features such as 'noses' and 'eyes' that were thought up by humans. We could however look at GAEs from a more general perspective. We already argued that GAEs can

be used to divide up the image into basins of attraction. We can take all these basins and use the encoding at their attractor as the characterization of that basin. This creates a group of descriptors of interesting parts, resembling the SIFT algorithm [Lowe, 1999]. As different GAEs find different features, we would repeat this process for several types of GAEs.

The question now becomes which GAEs to use and how to train them. If we want to create such a general system and compare it to the SIFT algorithm, we cannot require annotations of all kinds of parts of the image. There are ways to train GAEs without annotations, such as the one described above, which uses an adaptive reward function. We could train several GAEs using such a system, where we enforce diversity in the GAEs so they specialize on certain features. Such an approach allows GAEs to be applied to any object recognition system. To see if such an approach works, more research needs to be done.

6.3 Conclusion

We combined autoencoders and reinforcement learning to a novel system called guided autoencoders. GAEs actively interact with the data and can be trained to find gradients using only local information. These gradients create basins of attraction that guide the GAE through the complex landscapes of the face images. This ability is unique to GAEs.

The GAEs were trained to find the eyes, nose and mouth of faces. They perform well on the task of finding the eyes, while finding the mouth proved to be more difficult. The eye, nose and mouth GAEs can be combined to do classification tasks. However, they do not rival state-of-the-art systems yet.

Their ability to navigate complex landscapes and their flexibility, make GAES promising systems. We have pointed to several extensions that are easy to implement and likely to improve performance. More experimentally, GAEs can also easily be applied to other tasks such as image segmentation and tracking. And finally, we proposed a generalization of GAEs to general feature extractors that resemble the SIFT algorithm and allows GAEs to be applied to any image recognition task.

To conclude, GAEs are able to guide through complex images. Although their classification performance is not ready to rival the state-of-the-art, the flexibility of GAEs allows them to be extended easily to improve performance and do other tasks than classification. With further research, we think that GAEs can be improved and become valuable systems in face recognition and computer vision in general.

Bibliography

Emmanuel J. Barbeau, Margot J. Taylor, Jean Regis, Patrick Marquis, Patrick Chauvel, and Catherine Lie geois Chauve. Spatio temporal dynamics of face recognition. *Cerebral Cortex*, 2007.

Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1): 1–127, 2009.

Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems 19*, systems19(d):153, 2007.

Emmanuel Candes and Terence Tao. Decoding by linear programming. *IEEE Transactions on Information Theory*, 51(12):22, 2005. URL `http://arxiv.org/abs/math/0502327`.

Modesto Castrillón, Oscar Déniz, Daniel Hernández, and Javier Lorenzo. A comparison of face and facial feature detectors based on the viola–jones general object detection framework. *Machine Vision and Applications*, 2010.

D C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and J Schmidhuber. High-performance neural networks for visual object classification. *Applied Sciences*, page 12, 2011. URL `http://arxiv.org/abs/1102.0183`.

Adam Coates, Honglak Lee, and Andrew Y. Ng. An analysis of single-layer networks in unsupervised feature learning. *Advances in Neural Information Processing Systems*, 2010.

David L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1614066`.

Mark J. Embrechts, Blake J. Hargis, and Jonathan D. Linton. Augmented efficient backprop for backpropagation learning in deep autoassociative neural networks. *Computational Intelligence*, pages 18–23, 2010.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. URL `http://www.ncbi.nlm.nih.gov/pubmed/16873662`.

Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical report, University of Massachusetts, Amherst, 2007.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. doi: 10.1.1.134.2462. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.2462`.

M. A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991. URL `http://math.stanford.edu/research/comptop/references/kr.pdf`.

Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Muller. *Neural Networks: Tricks of the trade*, chapter Efficient BackProp. Springer, 1998.

D G Lowe. Object recognition from local scale-invariant features. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 2([8):1150–1157 vol.2, 1999. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=790410`.

Roland Memisevic and Geoffrey E. Hinton. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural Computation*, 22(6):1473–1492, 2010. URL `http://www.ncbi.nlm.nih.gov/pubmed/20141471`.

David Monzo, Alberto Albiol, Jorge Sastre, and Antonio Albiol. Precise eye localization using hog descriptors. *Machine Vision and Applications*, 2010.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X. URL `http://portal.acm.org/citation.cfm?id=104279.104293`.

Jürgen Schmidhuber. Curious model-building control systems. In *In Proc. International Joint Conference on Neural Networks, Singapore*, pages 1458–1463. IEEE, 1991.

Thomas Serre, Gabriel Kreiman, Minjoon Kouh, Charles Cadieu, Ulf Knoblich, and Tomaso Poggio. A quantitative theory of immediate visual recognition. *Progress in Brain Research*, 165(06):33–56, 2007. URL `http://www.ncbi.nlm.nih.gov/pubmed/17925239`.

Pawan Sinha, Benjamin Balas, Yuri Ostrovsky, and Richard Russell. Face recognition by humans: Nineteen results all computer vision researchers should know about. *Proceedings of the IEEE*, 94 (11), 2006.

Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.7692`.

A. S. Tolba, A. El-Baz, and A. El-Harby. Face recognition: A literature review. *International Journal of Signal Processing*, 2(2):88–103, 2005.

Hado van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007.

P. Vincent, H. Larochelle, Y. Bengio, and P. A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008. URL `http://portal.acm.org/citation.cfm?id=1390156.1390294`.

John Wright, Allen Y Yang, Arvind Ganesh, S Shankar Sastry, and Yi Ma. Robust face recognition via sparse representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31 (2):210–227, 2009. URL `http://www.ncbi.nlm.nih.gov/pubmed/19110489`.

W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys*, 35(4):399–458, 2003. URL `http://portal.acm.org/citation.cfm?id=954339.954342`.