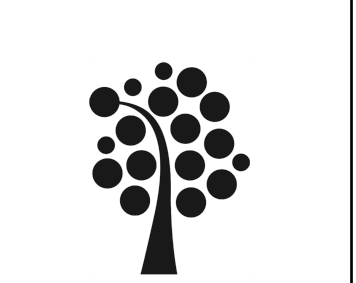

Bachelor Degree Project

Attractors of autoencoders

- Memorization in neural networks

Author: Jonas Strandqvist
Supervisor: Jesper Andersson
Semester: VT 2020
Subject: Computer Science

Abstract

It is an important question in machine learning to understand how neural networks learn. This thesis sheds further light onto this by studying autoencoder neural networks which can memorize data by storing it as attractors. What this means is that an autoencoder can learn a training set and later produce parts or all of this training set even when using other inputs not belonging to this set. We seek out to illuminate the effect on how ReLU networks handle memorization when trained with different setups: with and without bias, for different widths and depths, and using two different types of training images – from the CIFAR10 dataset and randomly generated.

For this, we created controlled experiments in which we train autoencoders and compute the eigenvalues of their Jacobian matrices to discern the number of data points stored as attractors. We also manually verify and analyze these results for patterns and behavior. With this thesis we broaden the understanding of ReLU autoencoders: We find that the structure of the data has an impact on the number of attractors. For instance, we produced autoencoders where every training image became an attractor when we trained with random pictures but not with CIFAR10. Changes to depth and width on these two types of data also show different behaviour. Moreover, we observe that loss has less of an impact than expected on attractors of trained autoencoders.

Keywords: machine learning, overfitting, memorization, neural network, autoencoder, attractor, Jacobian, eigenvalue, CIFAR10, random data, ReLU, bias

Contents

1 Introduction

Within artificial intelligence, neural networks are well-studied systems used for machine learning. They can "learn" from examples to perform certain tasks, such as classifying objects on pictures or steering the controls of a self-driving car. A neural network learns during a training phase where it adapts itself to produce better results on given training examples. After training, the neural network can predict outcomes on input data it has not processed before.

Autoencoders are special cases of neural networks, which are used to find efficient encodings of data. An autoencoder in fact first encodes given data and then directly decodes it. During the training phase, it is optimized such that the result of this encoding and decoding is as close as possible to the given training data.

Recent experiments show that autoencoder neural networks can memorize data [1]. Memory in this situation is an autoencoder's ability to reproduce data it has been trained with even when other data is used as input. To be more precise the memorized data is stored as so-called attractors, which are data points the autoencoder converges to after many iterations. This means that iterating the autoencoder on any input will eventually result in one of the data points used for training.

Within this thesis, we extend the experiments in [1] to study the behavioral changes of autoencoders with Rectified Linear Units (ReLUs) – one of the many types of units used in neural networks – when varying the number of parameters optimized during training and when changing the training data from real data to random data.

1.1 Related work

Since machine learning is a rapidly growing topic these last couple of years, there is an abundance of articles, blogs and portals exploring many types of different neural networks and autoencoders.

In the article [1], A. Radhakrishnan, M. Belkin and C. Uhler train autoencoders and compute how many data points in the training set become attractors after sufficient training. The data they work with are mainly pictures (among others from the CIFAR10 [2] data set), but also video and audio files. They perform their experiments on a variety of autoencoders with different setups, e.g. changing the size of the autoencoder or using different optimization algorithms during training. These experiments build on earlier work on attractors of autoencoders in [3]. Both articles [1] and [3] introduce eigenvalues of Jacobian matrices of autoencoders as a tool to determine attractors in a given training set, which we make use of in this thesis.

One property of attractors is that training data which is stored as attractors can be recovered by the autoencoder when using random input data. According to [1], the only other article in the literature discussing this phenomenon is [4], which presents a deep study on the memorization of a single image and the impact different architectures of autoencoders have in that setting. Since that article focuses on training with a single image, it presents a special case of the investigation in [1] and this thesis.

1.2 Motivation

It is an intuitive and wide-spread belief in the machine learning community that overparametrized neural networks memorize the training data instead of extracting general rules from it which are then applied on new input data. Here overparametrization refers to neural networks with many more trainable parameters than available training data, which

is common for deep networks. This phenomenon is sometimes referred to as *overfitting* or *overtraining*. The experiments in [1] and [3] substantiate this belief. In fact, the authors do not only provide evidence for that the training data is memorized, but that it is even stored as attractors, which means that the trained autoencoder is stable on distorted inputs close to the training data.

The latter is particularly interesting in light of *adversarial examples* which are inputs to neural networks designed such that the network produces a false prediction. Typically, adversarial examples are obtained by perturbing a data point slightly such that the perturbation causes the neural network to make a different prediction although the perturbed data is indistinguishable for human users from the original data. This cannot happen for data points which are stored as attractors in an autoencoder, since the autoencoder makes (almost) the same predictions for sufficiently small perturbations of an attractor.

We want to extend the experiments in [1] to increase the understanding for this behavior of overfitting and attractors. For instance, the authors of [1] do not explain fully which parameters in their autoencoders are optimized during training: more specifically, they do not mention if the trainable parameters are only the *weights* in an autoencoder or also its *bias vectors* (see Chapter 2 for an explanation of these notions). We want to investigate the impact of including / excluding bias on the number of stored attractors.

To give a meaningful approximation for the expected number of attractors, we perform experiments with random training data, in contrast to using structured data (e.g actual pictures) as in [1]. Although structured data is more relevant to real-world applications, it is also important to study random data to get a better understanding of the average behavior of attractors in autoencoders. This is particularly interesting for the theory of machine learning, as mathematical approaches to explain machine learning often assume sufficiently random training data.

1.3 Problem formulation

All autoencoders in this thesis are ReLU neural networks (see Chapter 2 for a definition). We decided to focus on ReLU networks because they have the easiest mathematical structure among all the types of autoencoders studied in [1]. ReLU autoencoders are piecewise linear functions, whereas the other autoencoders studied in [1] invoke either trigonometric functions or more advanced versions of ReLU (such as *leaky ReLU* or *SELU*).

In this thesis, we plan to increase the understanding of:

- the amount of attractors in ReLU autoencoders of different sizes,
- the use of Jacobian matrices and their eigenvalues as a tool to understand attractors,
- the impact on the number of attractors when training ReLU autoencoders with(out) bias, and
- the resulting difference on the number of attractors when training ReLU autoencoders with random data vs. structured data.

1.4 Outline

The relevant background about neural networks, which can be helpful for understanding this thesis can be found in Chapter 2. We then introduce autoencoders and their attractors in Chapter 3. Following comes a brief explanation of the method used to answer the problem formulation of the thesis in Chapter 4. Afterwards we discuss the design of the

experiments for the thesis in Chapter 5. Continuing, the implementation and its components will be covered in Chapter 6. We report our results from our experiments in Chapter 7 and give our analysis of the results in Chapter 8. Then we hold a brief discussion on each of the questions in the problem formulation in Chapter 9 and we conclude the thesis by presenting what has been done as well as future work in Chapter 10.

2 Neural Networks

We explain fundamental concepts used throughout this thesis. A deeper introduction to neural networks and machine learning is given, for instance, in the book 'Deep Learning' [5].

2.1 Main concepts

When we refer to a **neural network**, it is a feed-forward neural network (FNN). Such a neural network is built up of layers, and each layer contains a set of nodes (sometimes referred to as units). Each node in one layer is connected with all nodes in the next layer via edges, see Figure 2.1. Each edge has a weight and each node has a bias. One of the main attributes of an FNN is that there are no cycles, that the input nodes in the first layer do not have parents and that the output nodes in the last layer do not have children. The graph representation of an FNN is known as a directed acyclic graph. Other types of neural networks also exists, such as Recurrent Neural Networks (RNN), Self-normalizing Neural Network (SNN), or Convolutional Neural Networks (CNN) which are designed to to deal with 2D shapes and are thus commonly used in applications using pictures.

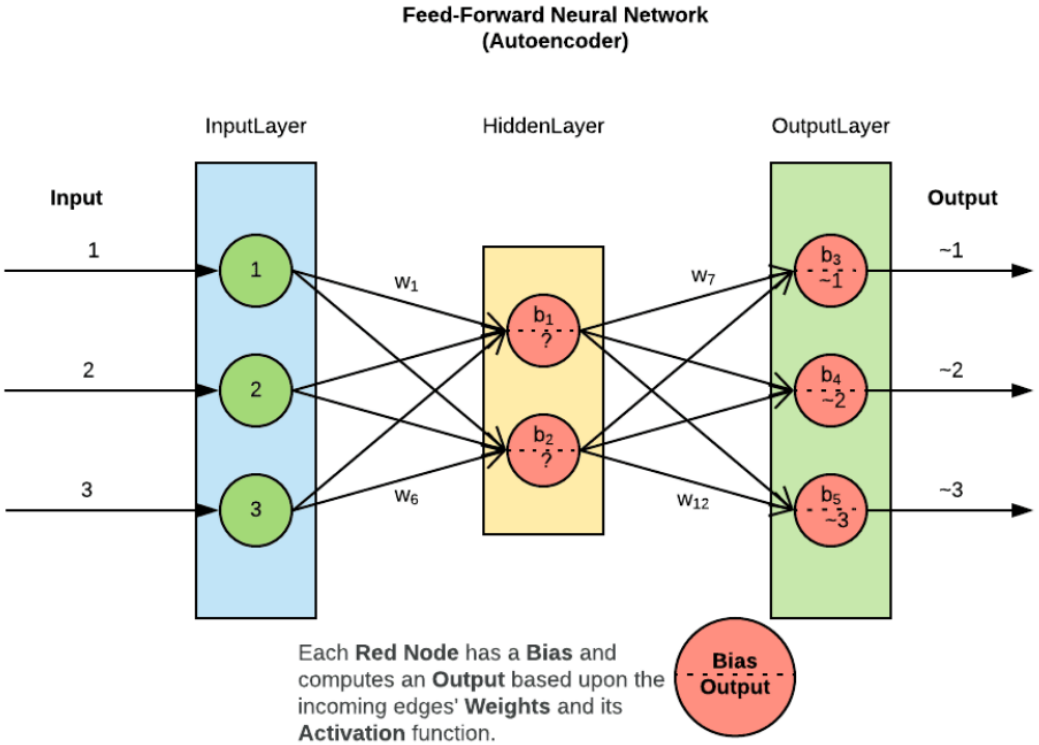

Figure 2.1: FNN

Depth and width Depth is what we refer to as how many hidden layers a neural network has between its input and output layers (depth sometimes also include the output layer, in this thesis it however does not). The width of a layer is the number of its units. For instance, the network in Figure 2.1 has depth one, its only hidden layer has width two, and its input and output layers both have width three.

Each node in the neural network, except for in the input layer, computes an output value depending on the inputs $x_1, x_2, \ldots, x_n$ along its n incoming edges: when writing b

for the bias at the node and $w_1, w_2, \ldots, w_n$ for the weights on its n incoming edges, this output value is

$$\sigma(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the *activation function* at the node; see Figure 2.2. Activation functions are used to model the behavior of cells in a brain: such a function is "the representation of the rate of action potential firing in a cell within a neural network" [6]. For each neural network used in this thesis, all nodes in that network will have the same activation function. A simple example of an activation function is just binary, choosing between 0 and 1 depending on the input.

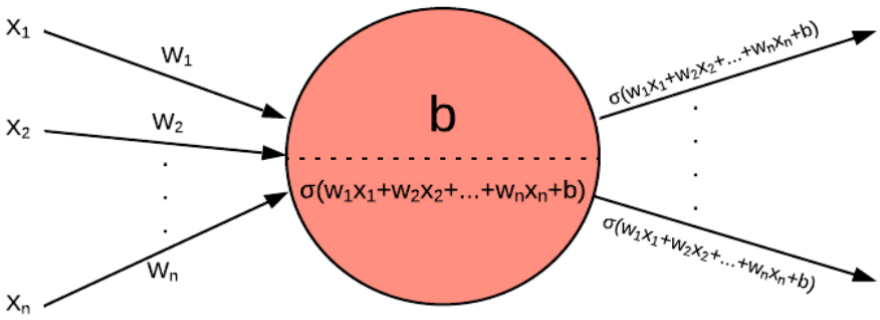

Figure 2.2: Computation by a node with activation function σ in a FNN; see Figure 2.1.

Rectified Linear Unit (ReLU) A *ReLU* is a unit with the piecewise linear activation function

$$\sigma(z) = \max\{0, z\}.$$

This is one of the most used activation functions [5]. A neural network is a *ReLU network* if all its nodes are ReLUs.

Predicting When all the weights and biases in a neural network are assigned, then the neural network can produce an output, called *prediction*, on given input data. To give an example, if the task of the network is to classify objects on pictures, Figure 2.3 would hopefully be the result.

Figure 2.3: Simple prediction.

2.2 Machine Learning

Training, also known as **fitting** within machine learning, is the process when the weights and biases in a neural network are improved to yield better predictions. For this, we need three ingredients:

1. We need to present **training data** $(X_1, Y_1), \ldots, (X_n, Y_n)$ to the network. Here each X_i is an input for the network, and Y_i is the expected answer for that input.

2. We need a measurement to decide how good the current weights and biases in our neural network are given the training data; this is the task of the **loss function**.

3. We need a way to improve the value of the loss function by updating the weights and biases in the network; this is done by the **optimizer**.

In machine learning, one often distinguishes between **supervised** and **unsupervised** learning. We describe these notions in more details.

Loss function The loss function is the way we measure how well the predictions $\hat{Y}_1, \ldots, \hat{Y}_n$ of our neural network on training inputs $X_1, \ldots, X_n$ compare to the expected outputs $Y_1, \ldots . Y_n$. The result of the loss function is what we call the *loss*. If our loss value is converging towards 0 during training it means that we are on the right track; if it becomes higher it means we are getting further away from our desired result.

In this thesis we are using *mean squared error* as our loss function which measures the average or mean $(\frac{1}{n}\sum_{i=1}^{n})$ of the squares of the errors $(Y_i - \hat{Y}_i)^2$:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Optimizer Optimization is the process of minimizing or maximizing a function, this is the exact job of the optimizer. In our case the optimizer is trying to minimize the loss function by finding the best weights and biases. This is done iteratively during the training phase over several so-called epochs: during each epoch, the optimizer processes the training set once to improve the loss.

One of the most classic algorithms for this is gradient descent which uses derivatives to find the direction in which to locally minimize the function. This is an expensive optimization for large datasets however, since the computational cost per epoch is $O(n)$ where n is the number of examples in the training set [5]. In the thesis we are using the more efficient optimization algorithm RMSprop [7], since, according to [1, Figure 3], it seems to produce the highest number of attractors in some setups of ReLU autoencoders.

Supervised learning A typical task for neural networks is classification. To give an example of training data, each training input X_i could be a picture of some animal or object, and the training output Y_i would then be the classification which type of animal or object (e.g. cat, dog, airplane, boat) the input X_i looks like, see Figure 2.4. This labeling of training input data is sometimes called 'supervisory signal', since we are supervising the learning process by telling the neural network what each input is and what we expect it to do with it. Hence we call this way of training *supervised learning*.

Unsupervised learning Unsupervised learning is when training of a neural network is done without using the typical labeling of the training data as explained under supervised learning. This is what we study in this thesis. In our setup of unsupervised learning, we "label" the training with the same data as we used for training (i.e. $X_i = Y_i$), shown in Figure 2.5. Here we allow the neural network to make sense of the data by itself. Examples of unsupervised learning tasks are density estimation, clustering and de-noising.

Figure 2.4: Labeled training data

Figure 2.5: Training data for unsupervised learning

2.3 Encoding and decoding pictures

An **autoencoder** is a special case of a feedforward neural network, with the restriction that its input and output layer have the same width. The goal of an autoencoder is to map every input to itself; meaning that its sole purpose is to reproduce the input as the output. Hence, an autoencoder learns in an unsupervised manner where each training input is the same as the expected answer, as in Figure 2.5.

In an autoencoder, the hidden layers have a smaller width than the input and output layer, as shown in Figure 2.1. So an autoencoder first encodes the data to some distorted format before reconstructing it. After reconstruction of the data to the output layer, it should then be as close as possible to the input; further explanation can be found in [5]. These networks can also be used purely for reconstructing distorted pictures, see Figure 2.6.

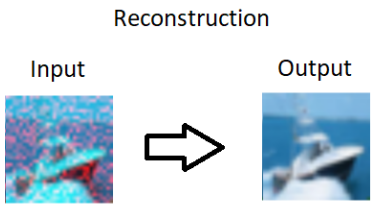

Figure 2.6: from pixel distortion

Random, structured and perturbed data In this thesis, we use different types of input or training data. With the term *structured data* we mean data that is non-generic, with some sort of relational connection between the data entries, such as actual pictures. (So this term does not refer to data which is structured to easily store it in a database.) The structured data we use in this thesis are images from the CIFAR10 data set, which is commonly used within machine learning, e.g. in [1, 3]. *Perturbed data* is structured data which has been nudged in some particular direction, making it distorted but still retaining

some of this structure. On the other hand, entries in *random data* used in this thesis do not have any relation to their neighbouring data entries, and in that sense are not structured.

These different types of data are exampled in Figure 2.7 as images.

Figure 2.7: Random, structured and perturbed image.

3 Memorization in autoencoders

In this chapter we extend the discussion on autoencoders. More specifically, we explain their attractors, following the description in the articles [1] and [3].

3.1 Iterative fixed points

An input to an autoencoder is called a *fixed point* if its prediction is equal to the input. Since the loss of an autoencoder is a metric that shows how close the data points in the training set are to their predictions, a low loss in a well-trained autoencoder means that the predictions of the training data points are *almost* the same as the original training data. However, the training data points are rarely honest fixed points in practical scenarios, since it rarely happens that these predictions are actually equal to the training data.

That is why we use the notion of *iterative fixed points* instead. Since the input and output layer of an autoencoder have the same width, we can use the predictions of the autoencoder as inputs. This means that we can iteratively apply the autoencoder on its previous prediction, and observe how the predictions converge, see Figures 3.1 – 3.3 for examples. We say that an input used for training an autoencoder is an *iterative fixed point* if its iterative predictions converge to a data point which is closer to the original input than to any other data in the training set. For instance, in the bottom of Figure 3.1, the image of the ostrich is an iterative fixed point. Although the figure only shows the results of the first 163 iterations, we had also verified that the ostrich is returned after 10,000 repeated iterations of the autoencoder.

Figure 3.1: Above: iterative predictions of perturbed ostrich image.
Below: iterative predictions of original ostrich image.

3.2 Attractors

The notion of an attractor is slightly different in this thesis than in the articles [1] and [3]. We will now explain our notion of this concept and explain the differences to the mentioned articles in Section 3.3.

We say that an input to an autoencoder is an *attractor* if it is an iterative fixed point and if, for all sufficiently small perturbations of the input, the iterative predictions of the perturbed input converge to the same data as the original input. The exampled ostrich in Figure 3.1 is not an attractor since a small perturbation was enough to converge to another picture in the training set. The ostrich image is given by 3072 numbers between zero and

one, and for the small perturbation we changed each of these numbers by ± 0.001 on average.

As can be seen while observing Figure 3.1, the difference between the perturbed and the original ostrich picture is impossible to discern for the human eye. Hence, the perturbed version of the ostrich picture is an adversarial example since it causes false iterative predictions of the autoencoder. This is why the notion of attractors is not only important for theory but also for practical scenarios, since they do not yield adversarial examples through perturbation.

Using input data outside the training set of an autoencoder, its iterative predictions will typically converge to an attractor within the training set. Figures 3.2 and 3.3 show examples for this behavior. Interestingly, the picture of the green truck in Figure 3.2 produces an image of a bird after one prediction, however through iteratively applying the autoencoder we converge to the white truck. This happens because the white truck is an attractor in the training set, whereas the bird is not. We also observe the white truck as an attractor in Figure 3.3, where the boat directly produces the truck and it also does not change when iteratively applying the autoencoder.

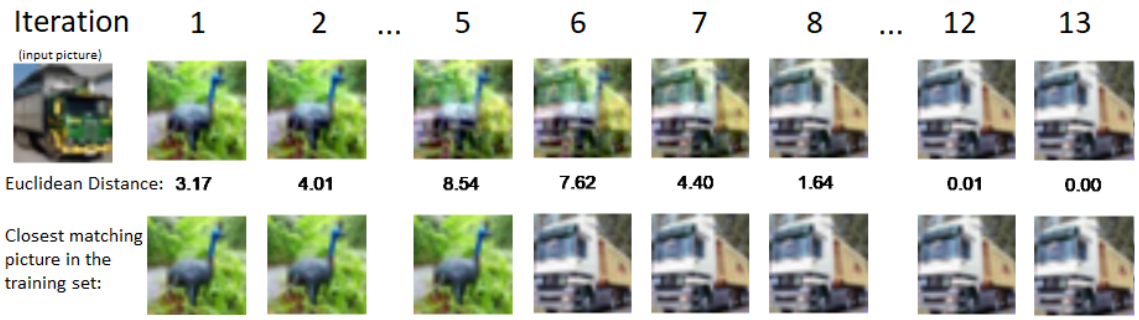

Figure 3.2: Iterative predictions of an input outside the training set converge to an attractor via a non-attractor.

Figure 3.3: Iterative predictions of an input outside the training set converge to an attractor immediately.

3.3 Jacobian matrices and their eigenvalues

Following the ideas of the articles [1] and [3], we can check if a given training data point is an attractor using the criterion presented in Table 3.1.

		largest absolute value of all eigenvalues:	
		< 1	> 1
iterative fixed point?	yes	attractor	not attractor
	no	not attractor	not attractor

Table 3.1: A training data point of an autoencoder is an attractor if it is an iterative fixed point and all eigenvalues of its Jacobian matrix are smaller than one in absolute value.

We now explain this criterion in details. The authors of the articles [1] and [3] describe attractors using eigenvalues of Jacobian matrices of the autoencoder map. More specifically, an autoencoder with fixed weights and biases can be viewed as a mathematical function: it takes an input data point and produces its prediction. In this thesis, our data points are images determined by 3072 numbers; this gives us the autoencoder function $\alpha : \mathbb{R}^{3072} \to \mathbb{R}^{3072}$. At any input data point p we can compute the Jacobian matrix $J_\alpha(p)$ of the function α. This is a square matrix of format 3072×3072. The eigenvalues of the Jacobian $J_\alpha(p)$ are typically complex numbers, but for the theory described in [1] and [3] only the highest absolute value of these eigenvalues is important: For a perfectly trained autoencoder with loss 0, the following two criteria hold (see Proposition 1 in [1] and Theorem 2 in [3]).

- A data point p is an attractor if the highest absolute value of the eigenvalues of $J_\alpha(p)$ is smaller than one.
- Moroever, p is not an attractor if this highest absolute value is larger than one.

The case where the highest absolute value of the eigenvalues is equal to one is not covered by the theory in [1] and [3], but this case is unlikely to happen in practice.

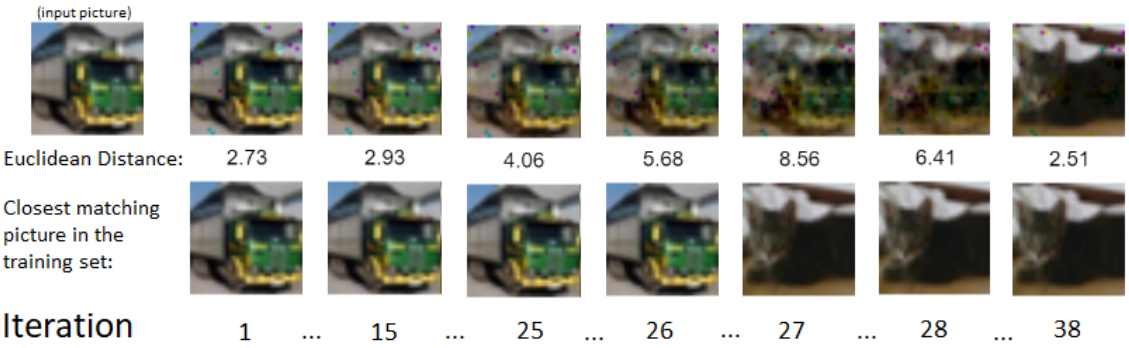

Figure 3.4: The truck is a training image which is not an iterative fixed point although the maximum absolute value of the eigenvalues of its Jacobian matrix is ≈ 0.35.

Since the loss of our trained autoencoders is strictly higher than zero, we cannot rely on that all data points with eigenvalues lower than one (in absolute value) are attractors. This is because a data point with small eigenvalues only means that the iterative predictions of small perturbations of that data point converge to the same data as the original unperturbed data point. However, the original data point might not be an iterative fixed point, e.g. it could converge to some other point in the training set, in which case it cannot

be an attractor. An example for this is shown in Figure 3.4: The truck image is not an iterative fixed point as its iterative predictions converge to the cat picture. That is why we do not want count it as an attractor, although all eigenvalues of the Jacobian at the truck image are smaller than one in absolute value, which means that small perturbations of the truck also converge to the cat when iteratively applying the autoencoder. Hence, to test if a data point is an attractor, we do not only have to check if its eigenvalues are smaller than one (in absolute value), but also that it is an iterative fixed point.

On the other hand, we can still rely on the fact that an eigenvalue higher than one (in absolute value) means that an iterative fixed point is not an attractor: small perturbations in the direction of an eigenvector corresponding to a high eigenvalue will not converge towards the original data point when iteratively applying the autoencoder. This is in fact how we perturbed the ostrich image in Figure 3.1 to produce the adversarial example.

Table 3.1 summarizes the way we check if a given training data point is an attractor.

4 Method

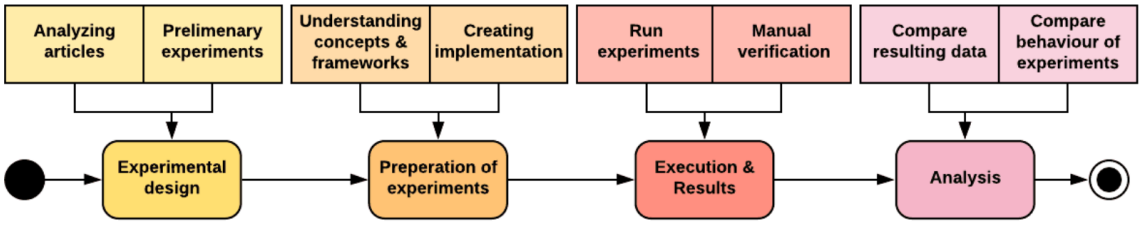

Figure 4.1: Overview of our controlled experiments.

The method used to get results for this thesis is through controlled experiments. To design our experiments, we first studied the articles [1] and [3]. We then created a preliminary implementation to run initial (informal) experiments to understand the experimental scope possible in the given time of this project. In Chapter 5, we explain how an experiment is performed and describe scope and limitations of our experiments.

To prepare the experiments, we had to understand the underlying theoretical concepts – such as attractors, Jacobian matrices and eigenvalues – as well as the frameworks used for our machine learning implementation. The latter was created with TensorFlow, an end-to-end open source platform for machine learning [8], within JavasScript and Python. We refer to Chapter 6 for details on the implementation.

During the execution phase, we ran all the defined experiments and gathered their results. In addition to our automated implementation, we manually observed the behavior of all experiments and verified the majority of the results manually. In Chapter 7, we present our results and observations. Finally, we analyzed the behaviour of our different experiments and compared their resulting data, see Chapter 8.

4.1 Preliminary experiments

When setting up the environment for our experiments we first tried replicating the base structure that could be discerned from [1]. This included running experiments that gave similar results including training autoencoders with 10 pictures instead of 100, however for our final experiments we decided on 100 pictures keeping a resemblance to the tables in [1].

As we had trouble to get TensorFlow to quickly calculate the Jacoobian matrix we also trained autoencoders on gray pictures to see if these were faster. As we at the end wrote our own computation (see Section 6.4) which was more effective, this no longer had big enough significance, thus colored pictures were used.

As the goal was to start with an as easy setup as possible we first trained models with stochastic gradient descent. Loss values and picture quality was however deemed insufficient, which was the reason for using RMSprop, the next in line in terms of complication from [1]. In addition, RMSprop led to the largest amount of attractors in ReLU autoencoders in comparison to other optimizers studied in [1, Figure 3].

When it came to the choice of activation function, ReLU autoencoders produced slightly worse pictures than some other functions (e.g. *SELU* or *Leaky ReLU*). ReLU was however deemed good enough for the experiments, as it has lower complexity.

In terms of epochs trained we went with 50,000 epochs as even training up to a million epochs only resulted in slight improvement to the loss on most autoencoders. Time was also a factor here as training more epochs simply takes longer time.

To get an understanding of which models were worth training for the thesis within the spectrum 1-512 width and 1-60 depth, autoencoders were trained at certain intervals. ReLU autoencoders got trained at depth 11, with width 256 and 512 producing similar or worse loss values than with width 128 and therefore excluded from the experiments.

Autoencoders were also trained at width 64, with higher depths such as 31 and 60 also producing worse loss values than depth 11, thus excluding higher depth autoencoders from our experiments.

A combination of increasing width to 60 and depth to 512 was also experimented with. These gave the same bad loss result as any autoencoder trained with depths 30 or higher with lower width, hence these were also not included in the scope of the experiments.

As time was also a bottleneck of this thesis and noticing that autoencoders trained using higher widths and depth (each added parameter adds time) did not yield significantly better loss values, the scope was chosen at maximum depth 11 with maximum width 128.

4.2 Threats to reliability

To make sure that the obtained results are reproducible, all code for the implementation is made available on Github at [9]. We note that the optimization algorithm for training neural networks makes random choices, which means one cannot reproduce our exact numerical values when repeating our experiments. The overall outcome of the experiments should however follow the same trends and thus be possible to replicate. That is why we also upload all trained autoencoders used in the results as TensorFlow models (which consists of the weights and biases of a trained neural network together with the setup used for training) on the web address mentioned above: loading these models will produce the results given in this thesis.

We also want to point out that our results could be influenced by our use of hardware. We are not aware of any limitations with the use of TensorFlow for training neural networks besides memory and time needed to perform training, but it is not implausible that other systems could produce different results. The machine running the experiments is an: ASUS GL702VMK with Intel(R) Core(TM)i7-7700HQ CPU @ 2.80GHz (4 Cores, 8 Threads), 16GB RAM, (Geforce GTX1060 GDDR5 6GB) on Win10 Education 64-bit.

4.3 Threats to validity

One must also admit that any implementation can produce unreliable data, as it could contain bugs or erroneous code. All code and results in this thesis are however verified manually at different levels: We tested all implemented functions on a smaller scale before starting with larger inputs. Moreover, we monitored how well our trained autoencoders perform by manual recognition of rendered prediction pictures in the browser through canvas and JavaScript; see Figure 6.4 for an example rendering. In our renderings we also show the Euclidean distance of the prediction to the closest training picture, which is for instance important when manually finding all iterative fixed points (see Section 3.1) in a given set of training images.

A crucial part of our experiments is to determine in an automated way if a training image is an iterative fixed point; see Section 5.1. According to our notion of iterative fixed points, we need to check if the iterative predictions of a training image converge to an image closer to the image we started from than to any other image in the training set. To test convergence, we would need to iterate our autoencoders an infinite amount of times, which is not plausible in practice. Hence, we had to find some bound for this thesis. After preliminary experiments, we decided that the iteration converged if the Euclidean

distance between the current prediction and the previous prediction is at most 0.0005. As convergence is not always possible, we need another limitation: we set the maximum amount of iterations to 32,000. Pictures whose predictions have not converged at this point are not considered to be iterative fixed points and marked as false. As 32,000 is not infinity, this could technically be a too low threshold. However, we verified that most images converge way before reaching 1000 iterations, and images that do not even converge after 32,000 iterations are not practically relevant for our results. See Section 6.3 for more details on the implementation.

5 Experimental design

In this chapter, we explain a typical process of an experiment and which parameters change between experiments. We also cover the scope of the experiments and the reasoning behind some decisions for limiting the scope.

5.1 Setup of an experiment

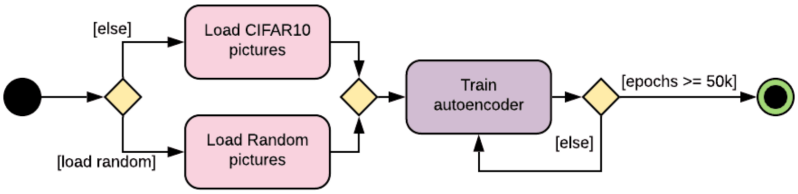

Figure 5.1: Training process.

In each experiment, we train a ReLU autoencoder within a JavaScript environment using TensorFlow. We do this using 100 images as training data. These images are either randomly generated or loaded from CIFAR10, a dataset often used in machine learning [2]. We train each autoencoder for 50,000 epochs (i.e. the optimization algorithms processes the training set 50,000 times to improve the loss): see Figure 5.1.

To check which training images are attractors of the trained autoencoder, we perform the following two parallel processes (Figure 5.2). On the one hand, we iteratively apply the autoencoder to each of its training images to test if it is an iterative fixed point. On the other hand, we compute the eigenvalues of the Jacobian matrix at each training image.

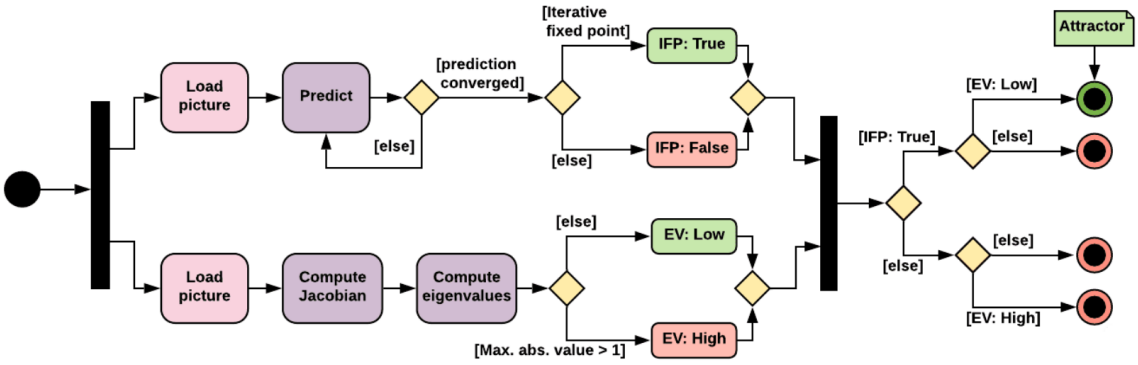

Figure 5.2: Attractor check: the 4 final states correspond to the 4 categories in Table 3.1.

5.2 Scope

The experiments that we perform for this thesis use autoencoders of certain widths and depths, which have been set after preliminary tests using widths in the range of 1 to 512 and depths between 1 and 64. These tests showed that depths above 31 produced impractically bad loss values, and widths above 128 severely increased the time it takes to train without showing any improvements on the loss values (e.g. same or worse results than with width 128). Using these observations (also described in Section 4.1) as well as influences from [1], we decided to set the maximum width for our experiments at 128 and the maximum depth at 11.

On top of this, we investigate the effect of the use of bias (which increases the number of learnable parameters during training) and different types of training images (CIFAR10 vs. randomly generated) on our autoencoders. The finalized combination of experiment setups are shown in Table 5.1. This means that the parameters that we influence in our different experiments are the following:

- bias,
- number of hidden layers,
- amount of units per layer,
- the training images.

depth \ width	128		64		32	
11	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random
6	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random
3	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random
2	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random
1	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random	cifar10 random

Table 5.1: Final set of experiments. Pictures used for training: random or CIFAR10. **Color legend**: with bias or without bias.

.

We repeat each of the 60 experiments listed in Table 5.1 four times and report their average number of attractors as well as the average and the confidence interval of their loss values (see Chapter 7). Training neural networks sometimes can lead to exceptionally bad loss values as the optimizer might get stuck in a bad local minimum. This can yield "non-functional" neural networks with predictions that are severely erroneous. If we would run each experiment only once, we could not rule out ending up with such a neural network. By repeating each experiment four times, we increase our chances to observe typical training behavior instead of only exceptional cases.

5.3 Limitations

The scope of our experiments is limited by several factors. One limitation is to only run training for up to 50,000 epochs. This specific limitation is set due to the time it takes to train a neural network. Even the smallest example that we experiment on takes 38 minutes to run, and the largest over 3 hours. Since we run each experiment four times to observe average behavior, time became an important factor throughout this project. This is also a reason why we cannot train an autoencoder for every single depth in our range from 1 to 11 and for every width between 32 and 128, and instead only run the experiments listed in Table 5.1.

Due to the amount of tuneable parameters when training neural networks, we also had to set a limitation to which of these we experiment with. Trying to optimize all these parameters is not plausible in the time frame of this thesis. Instead we use a similar setup to the ReLU autoencoders that produced the largest number of attractors according to [1, Figure 3]. The parameters we change between experiments are described in the previous section.

6 Preparation of experiments (Implementation)

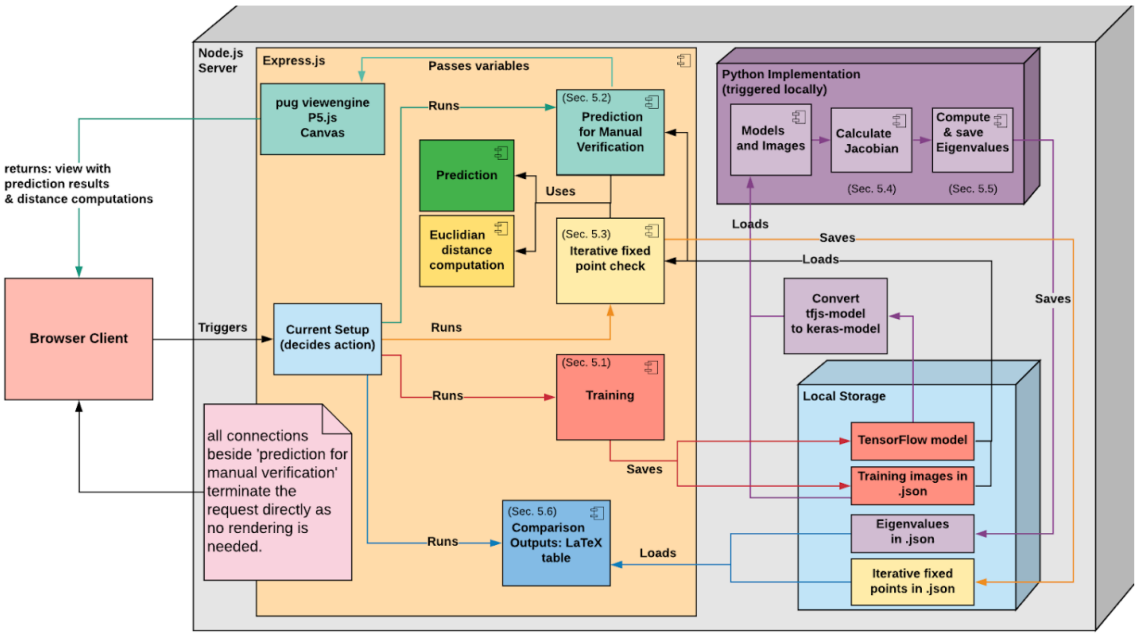

Figure 6.1: Implementation overview.

The implementation needs to take care of a couple of different things to produce the results needed for this thesis. Some computations are done with Python, but the main bulk of the application is coded in JavaScript within Node.js; an overview can be seen in Figure 6.1. The overview is an informal diagram showing how "components" interact. Each "component" in the diagram *marked with a number* has a section of its own below. The sections' numbering also show a typical order of execution.

We see that we can manually trigger Python functionality which loads TensorFlow models for autoencoders, computes Jacobian matrices and their eigenvalues (see Sections 6.4 and 6.5, also Figure 6.6), and then saves the results as json-files. We also see that a client browser needs to send a request which triggers one of the actions in the Node.js environment through an Express.js web-server. The 'action' triggered is based on what is pre-defined within the application as the client sends a request (as in, there is no actual user interface implemented). These actions are:

- training of an autoencoder (see Section 6.1 and Figure 6.2).
- prediction with a rendered view of the input, output, and closest image to the prediction in the training set as well as the Euclidean distance to the last mentioned images (see Section 6.2 and Figure 6.3).
- check for iterative fixed points in a specified model as well as saving the results as json (see Section 6.3 and Figure 6.5).
- comparison which loads data from the previous mentioned actions (computations of iterative fixed points and eigenvalues), which in turn produces a table to be used in the thesis with data for analysis (see Section 6.6 and Figure 6.7).

This overview does however not include the inner workings of the implementation. The whole implementation can be found on Github at [9]. Some details are covered in the sections below, describing the actions in the order of execution used throughout one experiment.

6.1 Training

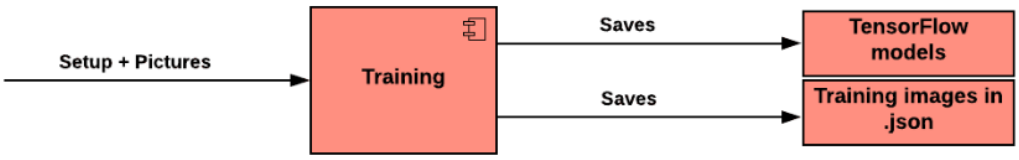

Figure 6.2: Trained autoencoders are stored as TensorFlow models along with the pictures they were trained with.

In each experiment, we train an autoencoder neural network within a JavaScript environment using TensorFlow, this is referred to as a trained *model*. Such a model is first created by deciding parameters such as what type of model, where we use a *sequential model* (simple model with a linear stack of layers [10]), and how each of the layers should be designed.

In our case all layers are of the type `dense` (meaning that all units of one layer are connected to the units in the next layer) and have the same settings for `bias` (`true` or `false`), `activation` function (in our case `ReLU`), and width (i.e., number of `units`). In each experiment this width is either `32`, `64`, or `128`, there are however two exceptions: First, when working with dense layers in TensorFlow, the first layer specified is the first hidden layer which also takes an *input shape*, determining the width of the input layer. In other words, when using dense layers, the input layer does not get implemented as its own layer; it is only specified via the input shape in the first hidden layer. In our case, the input shape is `3072` due to the format of our training images (see below). Second, as an autoencoder requires the input and output to be the same shape, we also add an output layer which has `3072` units at the end of our neural network.

The model is then compiled with an `optimizer` algorithm (set to `RMSProp` with a learning rate of `0.001`) and a `loss` function (set to `mean squared error`).

After this setup we can now *fit* or *train* the model by feeding it pairs of `training` and `target` data. Since we are working with autoencoders, the target data is the same as the training data. In our case, the training data consists of 100 images with $32 \times 32 = 1024$ pixels each. Since each pixel has three channels (RGB), every training picture is given by 3072 numbers between 0 and 1. Our pictures are either from CIFAR10 or randomly generated by picking a value between 0 and 1 for each of the 3072 numbers of a picture. The latter is done using *Math.random()* in JavaScript which picks values approximately uniformly distributed. The data points representing pictures are also stored as json-files to be used in other parts of the program (especially important for randomly generated pictures), as well as saved as png-files for manual verification.

Lastly, we also specifiy the amount of `epochs` the training should run, as discussed in Section 5.1 to 50,000 epochs. Both the model with the lowest loss during the whole training process as well as the current model when reaching 50,000 epochs will be saved permanently. This is mainly added to make sure that the model with the lowest loss is not lost during training, which is however a more common occurrence when training using the *adam* optimizer which we used only in our preliminary experiments.

6.2 Predictions for manual verification

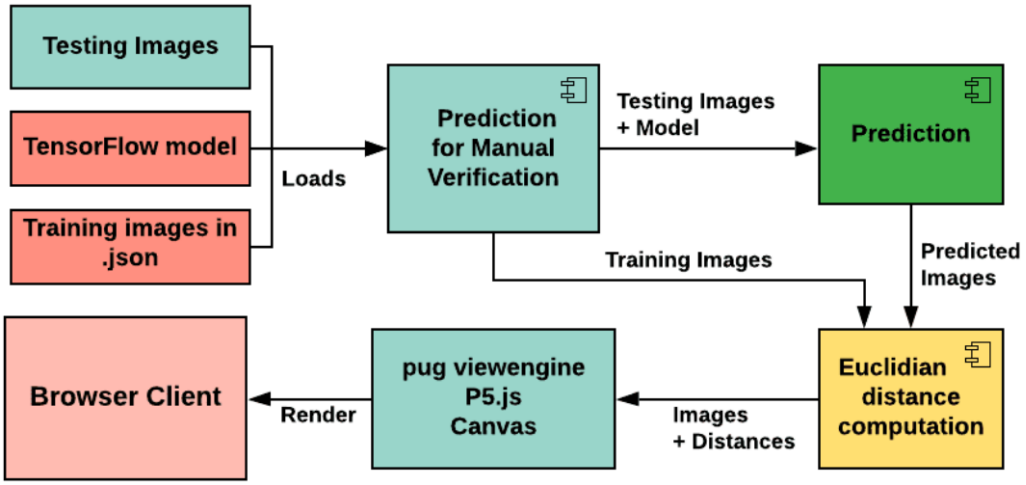

Figure 6.3: Overview of manual verification environment.

Using TensorFlow.js and a model that we trained, we can then *predict*. The main purpose of this part of the implementation is to manually verify some of the results presented in this thesis and to build an understanding of what is happening when performing certain experiments, see for instance the adversarial example in Figure 3.1.

These predictions require an input of testing data, which in our case – depending on the verification purpose – is either the same as the training data, perturbed versions of that data, or other data of the same type (in our case pictures of the same format). This part of the implementation has been combined with the view, where a rendering of each of the resulting pictures from the prediction together with its matching training and testing data sets is presented to the user for manual verification.

Figure 6.4: Example rendering of manual verification environment.

To make this work however we also have to reformat the 3072 numbers describing

each involved picture and store these as an image, since looking at thousands of numbers is not something humans excel at. These are however only stored until the next prediction is made. We also calculate the Euclidean distance between each predicted result and its closest picture in the training set.

Now to create the verification environment, a very limited Express.js web-server implementation responds to the clients request. This response includes a basic pug view engine page together with which pictures and data to load, which in turn loads a script that uses P5.js to render the data onto a HTML Canvas. The pictures depicted here are the ones trained with, the pictures fed into the prediction as well as the resulting pictures of the prediction. We also render for each prediction its closest picture in the training set as well as its Euclidean distance to that closest picture, see Figure 6.4.

6.3 Iterative fixed points

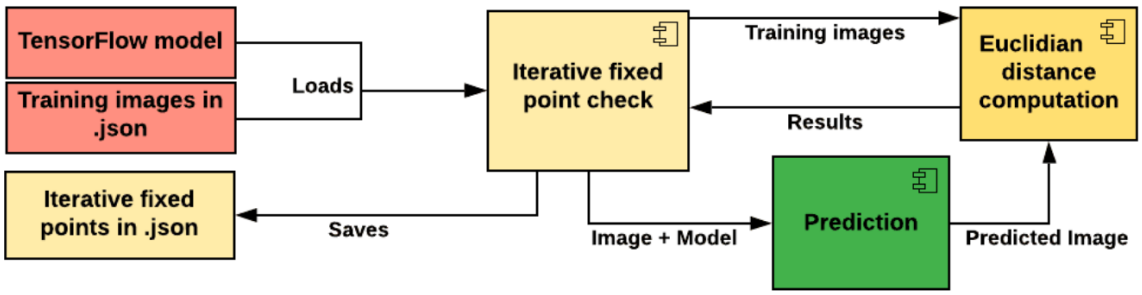

Figure 6.5: Iterative fixed point check.

One part to check if a training picture is an attractor is to test if it is an iterative fixed point. In the implementation we do this test through re-iterating predictions using the output from the last prediction as input and computing the distance between the previous and the current prediction until we reach a distance less than 0.00005. This means that the iterative predictions have reached convergence (approximately). At this point we look to see if the current prediction is closest to the original training picture and not to another picture in the training set. If this is the case, we decide that the training picture is an iterative fixed point and mark it with true; otherwise we decide that it is not and mark it with false.

If this distance of 0.00005 could not be reached within 1000 iterations of the autoencoder, we stop predicting. If the closest training image to the final prediction is not the same image we started from, we mark the image with false; otherwise we mark it with a null value and move to the next image in the training set. After finishing these iterative predictions for all training pictures, we then take all the null-marked images and repeat the above computations with an upper bound of 32,000 iterations (instead of 1000). This time however, we mark each picture that does not converge but still retains itself as the closest picture with false (instead of null), since such pictures do not converge within a practical amount of iterations (discussed in Chapter 4). The results from this iterative fixed point check are then saved in a json-file.

6.4 Jacobian matrices

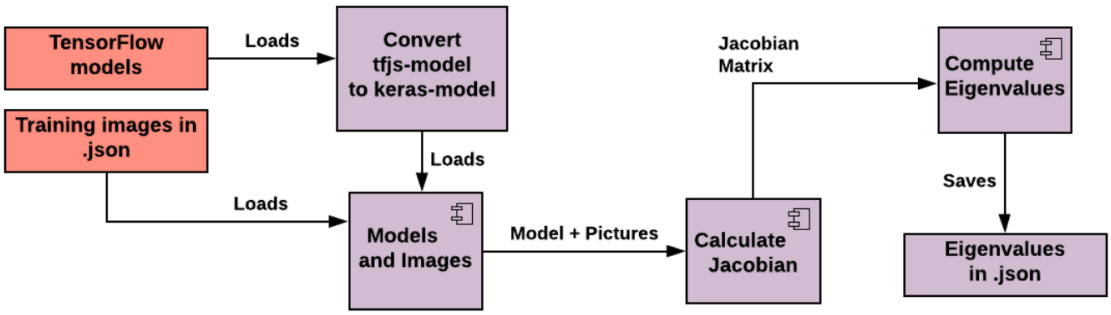

Figure 6.6: Computing eigenvalues of Jacobian matrices.

Besides the check for iterative fixed points, we compute the eigenvalues of the Jacobian matrix of the autoencoder at a training picture, to decide if that picture is an attractor or not. Before we get a hold of these eigenvalues, the Jacobian matrix needs to be computed. Originally the idea was to do this in JavaScript as well. However, a built-in functionality to obtain Jacobian matrices does not seem to exist within that environment, which was the main cause to move some of these computations to Python. In the Python environment we can use GradientTape to automatically compute Jacobian matrices after converting and loading our models from JavaScript into Python. Using GradientTape was however way too slow to be usable for this thesis since it took about 15,000 seconds on current hardware (see specification at the end of Section 4.2) to run the computation for one image and a network of depth 12 and width 256 (which was needed during our preliminary experiments). To solve this complication and be able to compute Jacobian matrices within the scope of 100 pictures per model and multiple models, this had to be implemented some other way. Since we did not find other ways to automatically compute Jacobian matrices in Python, we implemented our own algorithm to find these matrices, using the knowledge of the chain rule and the layered structure of the autoencoder with its activation functions. Our implementation applied to the same model as mentioned above solves the computation in barely 5 seconds. To verify that our new implementation works, we manually compared our results to the results delivered by GradientTape in several instances.

We now describe our algorithm. Given an input vector x to a neural network, the output vector y_1 of the first layer is computed as follows:

$$y_1 = \sigma(W_1 \cdot x + b_1),$$

where σ is the activation function, and W_1 and b_1 are the weight matrix resp. bias vector of the first layer. To explain our use of the chain rule, we re-write this expression:

$$y_1 = \sigma(\ell_1(x)), \qquad \text{where } \ell_1(x) = W_1 \cdot x + b_1.$$

According to the chain rule, the Jacobian matrix $J_1(x)$ of that expression at the input vector x can be computed by multiplying the following two matrices: $J_\sigma(\ell_1(x))$, which is the Jacobian matrix of the activation function at the vector $\ell_1(x)$, and $J_{\ell_1}(x)$, which is the Jacobian matrix of ℓ_1 at the input vector x. By basic rules of differentiaton, the latter matrix is just the weight matrix, meaning that $J_{\ell_1}(x) = W_1$. This yields that the Jacobian matrix of the first layer at the input vector x equals

$$J_1(x) = J_\sigma(\ell_1(x)) \cdot W_1.$$

Similarly, the second layer of the neural network computes

$$y_2 = \sigma(\ell_2(y_1)), \qquad \text{where } \ell_2(y_1) = W_2 \cdot y_1 + b_2$$

is given by the weight matrix W_2 and the bias vector b_2 of the second layer. As above we use the chain rule and obtain now that the Jacobian matrix of the second layer is

$$J_2(y_1) = J_\sigma(\ell_2(y_1)) \cdot W_2.$$

Applying the chain rule again, we see that the Jacobian matrix of a 2-layer neural network at the input vector x is

$$J_2(y_1) \cdot J_1(x).$$

For a neural network with k layers, its Jacobian matrix is a product of k matrices:

$$J_k(y_{k-1}) \cdot J_{k-1}(y_{k-2}) \cdots J_2(y_1) \cdot J_1(x). \tag{1}$$

Each of these k factors is computed as described above for $J_2(y_1)$ and $J_1(x)$.

We essentially implemented the above algorithm with the exception that we compute the product (1) iteratively as we work our way through the layers. In our implementation, σ is the ReLU activation function. Its Jacobian matrix $J_\sigma(z)$ at a vector z is a diagonal matrix whose entries are either zero or one, depending on which entries of z are negative or positive. It typically does not happen in practice that z has entries which are zero.

6.5 Eigenvalues

Now that we have the Jacobian matrix at each training picture, we can compute the eigenvalues through Numpy in Python by using the Linear Algebra library and its eigenvalue functionality. With this we can now retrieve all eigenvalues, and also their eigenvectors if needed. For finding the attractors we however only need to check that the absolute value of each eigenvalue is below 1, so for each training picture we save the largest absolute value of all eigenvalues in a json-file to be loaded in the JavaScript environment later on.

6.6 Attractors

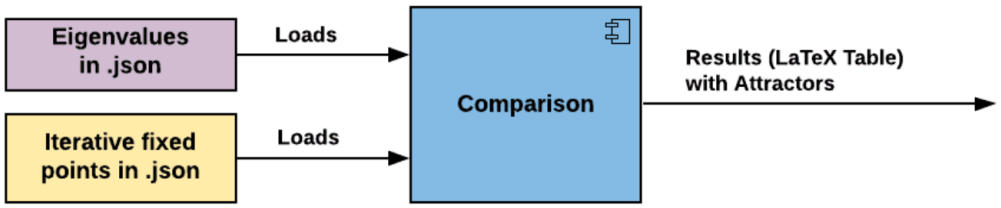

Figure 6.7: Determining attractors.

Finally, to find the attractors of a training data set, the JavasScript application can now run a function which uses the data acquired in the previously described parts of the implementation: here we load the json-file with the largest absolute values of the computed eigenvalues as well as the json-file with the results from the iterative fixed point check. Comparing the entries in these two files, we can now answer into which of the the four categories described in Table 3.1 each training image belongs.

7 Execution & Results

In this chapter, we present results gathered during the experiments executed for this thesis.

7.1 Average amount of attractors, loss and training times

For each pair of width and depth in Table 7.1, four autoencoders were trained using 100 training pictures, as described in Chapter 5. For each trained autoencoder and each training picture, we computed the eigenvalues of its Jacobian matrix and determined if the picture is an iterative fixed point. With this information we checked if that picture is an attractor, or if it falls into one of the other three cases described in Table 3.1. The average number of pictures in each of those four cases is shown in Table 7.1, where we averaged over the four experiments carried out for each investigated pair of width and depth. The corresponding average losses reached when training the autoencoders as well as the average training times are displayed in Table 7.2.

7.2 The notion of iterative fixed points

We want to investigate how well our notion of "iterative fixed points" introduced in Chapter 3 works for our experiments. Our notion says that a training image is an iterative fixed point if its iterative predictions converge to a picture which is closer to the original image than to any other picture in the training set. However, just because a training image has this property, its converged prediction does not necessarily resemble the original image. In that sense, a training image which is an iterative fixed point could be a "false positive" by human perception.

We therefore manually verified all training images that were reported as iterative fixed points during all 72 experiments using CIFAR10 pictures (reported in Tables 7.1a and 7.1b). We include rendered pictures of all iterative fixed points from experiments with depth 1 (Figure 7.1) and depth 2 (Figure 7.2). Since 47 iterative fixed points have been identified throughout our experiments with depth 3 (see Table 7.1a), we depict them in the Appendix (see Figure A.1). The iterative fixed points of our experiments with bias and and depths 6 and 11 will not be shown as they are too many and we did not find any "false positives" among them.

Remark. Aside from this, we also report for the 6000 pictures used in our experiments with bias and CIFAR10 that our implementation of the iterative fixed point check marked 63 of these with null after 1000 iterations (i.e. their iterative predictions are not converging but still closest to their starting picture after 1000 steps). We thus checked each of those 63 pictures with 32,000 iterations of the autoencoder. During this process, only 4 were found to be iterative fixed points.

As for the 12 experiments without bias in Table 7.1b, we observed that each of them has at most one iterative fixed point. We found 9 iterative fixed points in total, however all of them are the same training image. As we manually verified these images, a typical example from one of these experiments is shown in Figure 7.3. This is a typical case of a "false positive".

Legend:

ifp & <1	ifp & >1
no ifp & <1	no ifp & >1

ifp – iterative fixed point

<1 or >1 – max. abs. value of eigenvalues lower resp. higher then 1

depth \ width	128		64		32	
11	**86.5**	0	**64.25**	0.25	**16.75**	2
	11.75	1.75	32.25	3.25	55.25	26
6	**41.75**	0	**22.5**	0.5	**9**	2.5
	44.25	14	39	38	21.5	67
3	**4.25**	1.25	**2.75**	0.75	**2**	0.75
	7	87.5	3.75	92.75	2.75	94.5
2	**1.25**	1	**0.5**	0.5	**0.25**	1.25
	0.75	97	0.5	98.5	0.75	97.75
1	**0.25**	1	**0**	1.25	**0**	1.75
	0.5	98.25	0	98.75	0.25	98

(a) CIFAR10, with bias.

depth \ width	128		64		32	
11	**0.75**	0	**1**	0	**0**	0.5
	73.5	25.75	99	0	51.75	47.75

(b) CIFAR10, without bias.

depth \ width	128		64		32	
11	**100**	0	**47**	0.25	**13.25**	0.25
	0	0	52.25	0.5	75.25	11.25
6	**98.75**	0	**27**	0.25	**11.25**	2
	1.25	0	63.75	9	48.5	38.25
3	**0**	0	**3.5**	1.5	**2.5**	2
	1	99	7.25	87.75	2.75	92.75
2	**0**	0	**0.25**	1	**0.5**	1.5
	0	100	0	98.75	0.25	97.75
1	**0**	0	**0**	0.5	**0**	0.5
	0	100	0	99.5	0	99.5

(c) random pictures, with bias.

depth \ width	128		64		32	
11	**1**	0	**1**	0	**1**	0
	99	0	99	0	99	0

(d) random pictures, without bias.

Table 7.1: Average number of training images in the four categories defined in Table 3.1, see **Legend** above. In particular, the average number of attractors is highlighted.

Legend:

average loss	confidence interval of loss
average time in seconds	

depth \ width	128		64		32	
11	0,00313	±0,00056	0,00317	±0,00027	0,01054	±0,00122
	11246		6945		5782	
6	0,00403	±0,00175	0,00401	±0,00086	0,01099	±0,00034
	8241		4851		4044	
3	0,00407	±0,00117	0,00434	±0,00068	0,01180	±0,00079
	6038		3233		3043	
2	0,00586	±0,00038	0,00498	±0,00086	0,01198	±0,00190
	6028		2898		2629	
1	0,00672	±0,00191	0,00694	±0,00122	0,01417	±0,00077
	5295		2529		2358	

(a) CIFAR10, with bias.

depth \ width	128		64		32	
11	0,06696	±0,11229	0,12852	±0,02196	0,01159	±0,00121
	8857		5361		4685	

(b) CIFAR10, without bias.

depth \ width	128		64		32	
11	0,00039	±0,00012	0,02413	±0,00061	0,05296	±0,00071
	10310		7156		5904	
6	0,00060	±0,00018	0,02397	±0,00111	0,05305	±0,00088
	7754		4889		3966	
3	0,00123	±0,00078	0,02344	±0,00083	0,05216	±0,00134
	6084		3324		3043	
2	0,00215	±0,00182	0,02314	±0,00084	0,05364	±0,00166
	5983		2974		2636	
1	0,00234	±0,00086	0,03331	±0,02651	0,05573	±0,00293
	5639		2567		2240	

(c) random pictures, with bias.

depth \ width	128		64		32	
11	0,17020	±0,00479	0,17988	±0,00752	0,17258	±0,03129
	8614		5317		4823	

(d) random pictures, without bias.

Table 7.2: Averages of the loss values (above left), the confidence interval at 95% (above right), and the average training time in seconds (below) for the autoencoders in Table 7.1.

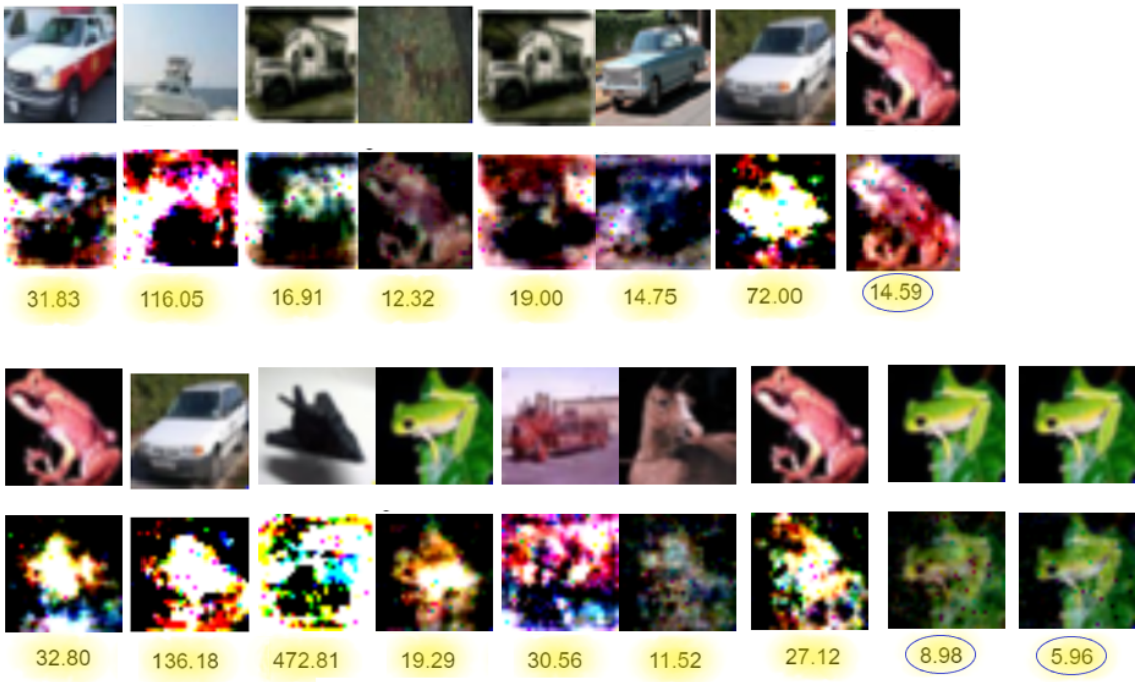

Figure 7.1: For each iterative fixed point found in the 12 experiments with bias, **depth 1** and CIFAR 10 reported in Table 7.1a, we show its converged iterative prediction and the Euclidean distance between that final prediction and the original image.
Predictions with the most resemblance to the original input image have their Euclidean distance values circled, through manual verification.

Figure 7.2: For each iterative fixed point found in the 12 experiments with bias, **depth 2** and CIFAR 10 reported in Table 7.1a, we show its converged iterative prediction and the Euclidean distance between that final prediction and the original image.
Predictions with the most resemblance to the original input image have their Euclidean distance values circled, through manual verification.

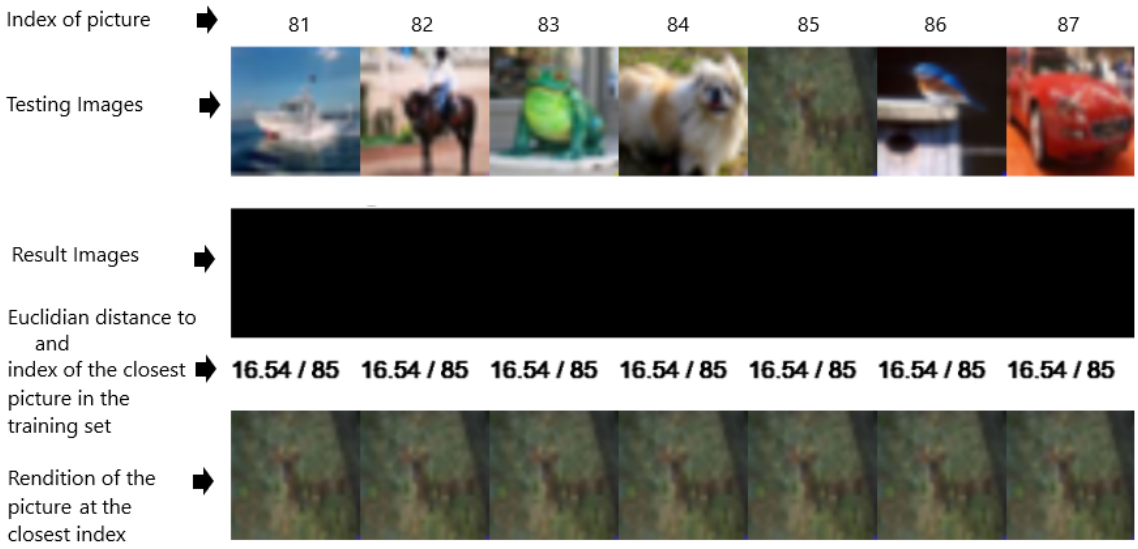

Figure 7.3: All registered iterative fixed points for all 12 experiments without bias reported in Table 7.1b have the same index; namely 85.
Above we show a rendering of training images 81–87 and their 1000th iterative predictions from one of those experiments (with width 128 and depth 11).

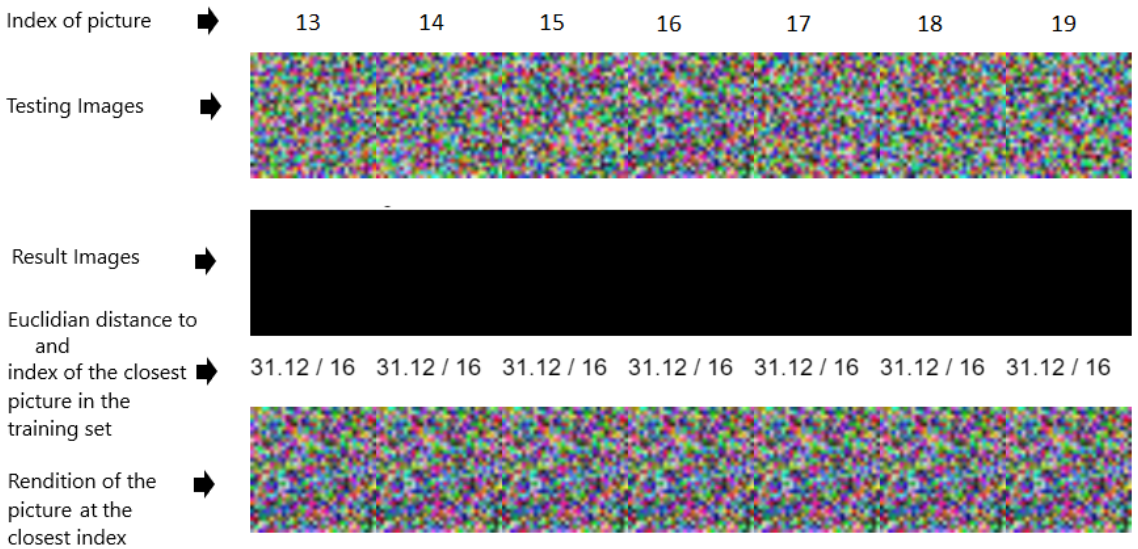

Figure 7.4: All registered iterative fixed points for all 12 experiments without bias reported in Table 7.1d have the same index; namely 16.
Above we show a rendering of training images 13–19 and their 1000th iterative predictions from one of those experiments (with width 128 and depth 11).

8 Analysis

In this chapter, we analyze the data in the tables as well as the pictures presented in the previous chapter. To get a better feeling while comparing the results, we now present the amount of learnable parameters of our autoencoders: We write w for the width, d for the depth, and $b \in \{0, 1\}$ denotes if bias is used or not. Then the number of parameters is

$$2 \cdot (3072 \cdot w) + w^2 \cdot (d - 1) + b \cdot (3072 + w \cdot d).$$

We list the amount of parameters for each investigated pair of depth and width in Table 8.1, with and without bias. There we can also see that the trained autoencoders used have enough parameters to overfit in all situations where depth is above 32, as the number of parameters in our training data equals 307200 (3072 values $\cdot$100 pictures).

depth \ width	128		64		32	
11	954752	950272	437952	434176	210272	206848
6	872192	868352	417152	413696	204992	201728
3	822656	819200	404672	401408	201824	198656
2	806144	802816	400512	397312	200768	197632
1	789632	786432	396352	393216	199712	196608

Table 8.1: Number of autoencoder parameters: **with bias** and **without bias**.

depth \ width	128	64	32
11			
6			
3			
2			
1			

8.1 Average amount of attractors, loss and training times

CIFAR10 experiments with bias As can be seen in Table 7.1a, the shallower a network becomes, the harder it is to retain attractors. This aligns with the outcomes of articles [3] and [1]. This also applies for narrow networks, i.e. a lower width reduces the number of attractors.

Interestingly, reducing the width from 64 to 32 increases the loss, whereas the losses for widths 64 and 128 are similar, see Table 7.2a. The effects of reducing the depth generally yields an increase in the loss values, which shows that trained autoencoders under these circumstances produce worse predictions. The loss value however does not give a good representation of what is to be expected for the number of memorized attractors. This observation is based on the averages for the autoencoders of width 64 and 128 reported in Tables 7.1a and 7.2a: as mentioned above, their losses are similar, but the amounts of attractors for width-64 autoencoders are at least 25% lower than for width-128 autoencoders (for all investigated depths). Loss does however have an impact on the quality of predicted pictures which in turn affects how accurately they can be analyzed programmaticly.

Another interesting observation is how few iterative fixed points have high eigenvalues. Our Table 7.1a shows, as depth and width decrease, that the point of where a picture

gets a high eigenvalue always relates to firstly losing its iterative fixed point status: The table shows a trend that high depth and width autoencoders produce many attractors, which first become non-iterative fixed points with low eigenvalues as the depth and width decrease, and finally end as non-iterative fixed points with high eigenvalues when depth and width become low.

CIFAR10 experiments without bias Running experiments without bias, we see that these do not create actual attractors, confer with Table 7.1b. In fact, the few attractors and all iterative fixed points that are listed in this table are "false positives": These images are so distorted after iteratively applying the autoencoder that the reason they show up as iterative fixed points is that the distance is still low enough not to be excluded when just being a black picture, see Figure 7.3.

As depth 11 gave the best numbers of attractors in all training scenarios with bias but did not manage to produce any actual attractors without bias, we decided that there was no point in trying to expand the experiments past the point of depth 11. This is why any other depths are missing in our reported results.

What can also be reported on training without bias is that the reached loss-values are less stable since the confidence intervals in Table 7.2b are considerably larger than in Table 7.2a. We observe that the higher width can produce better loss values than low-width autoencoders, however this had not been the case for all four width-128 autoencoders in Table 7.2b, which explains the large confidence interval. We believe that we require a much larger sample size, which we do not currently have, to confidently observe the positive impact of a higher width on the loss values in Table 7.2b.

Experiments with random images We note when looking at the results for random pictures in Table 7.1c that the general trends are similar when looking at Table 7.1a for CIFAR10 pictures. This means that depth as well as width reduction does reduce the amount of attractors. It also holds that we once again do not see many iterative fixed points that have high eigenvalues.

We now look at the differences between both tables. Most notable is the column of width 128: when training autoencoders with random images, essentially all training images are attractors for depth 6 or 11, whereas for depth at most three the training images are of "the opposite type", meaning that they have large eigenvalues and are not iterative fixed points. Generally, for all autoencoders of depth at least three in Table 7.1c it is true that training images with low eigenvalues are much more frequent than in Table 7.1a.

For our experiments with random images, depth seems to have a lower impact on the number of attractors than width does; particularly when comparing with the results of our CIFAR10 experiments where depth has a stronger impact than width. Changing the width from 128 to 64 or even 32 results in a steeper decrease of the amount of attractors in Table 7.1c than in comparison to Table 7.1a. Regarding depth, when looking at the trend over all values of the widths at depth 6 we see that there is a slight increase of attractors among the random pictures in comparison to CIFAR10 pictures. When increasing the depth to 11, we however observe a higher increase in the number of attractors for CIFAR10 then with random pictures. This in turn makes us assume that width is more important in creating attractors for random pictures without relation between their pixels, which was however the opposite when running the same experiments with structured pictures from CIFAR10 as depth was the main factor there.

Analyzing the loss values in Table 7.2c, we also see that the impact of width is greater than that of depth. For width 128, the loss values are generally better when training with

random instead of CIFAR10 images in 7.2a: Even at depth 1 with random pictures we have loss values that are lower than at depth 11 with CIFAR10. However reducing the width to 64 makes the loss far worse when training with random pictures, even worse than the worst loss for depth 1 and width 32 with CIFAR10 images. These observations do however strengthen our belief that a low loss is not the only factor to memorization in autoencoders.

Training without bias results in autoencoders which no longer manage to retain any attractors, see Table 7.1d (the few attractors in the table are false positives, see Figure 7.4). However, we see that all random pictures produced low eigenvalues, which was not the case for CIFAR10 in Table 7.2b. Moreover, the loss when training without bias using random pictures (see Table 7.2d) is much worse than when using CIFAR10 (see Table 7.2b).

Training times We also note that each time we reduce depth or width, training times get shorter as the network diminishes in size. Training times are also reduced as we remove bias since we remove learnable parameters from the optimization algorithm when doing so. Changing the type of pictures (random vs. CIFAR10) used for training does not seem to have any impact on the training time as long as the input shape is retained.

8.2 Iterative fixed points

As we observe the results on iterative fixed points in Table 7.1a compared to the rendered pictures shown in Figures 7.1 and 7.2, manual verification by human perception differs from some results in the table. This is because some of the iterative fixed points seem to be "false positives' as their iterative predictions do not have any or only minimal resemblance to the original picture. We manually verified the 17 iterative fixed points in our 12 experiments at depth 1: their pictures are shown in Figure 7.1. Roughly 82% of these iterative fixed points would not be reported as iterative fixed points by manual verification. At depth 2, in Figure 7.2, roughly 36% of the 19 pictures reported would be "false positives". Of the 47 iterative fixed points at depth 3, shown in Figure A.1, this percentage is only 19%. Moreover, as the depth increases (to 6 and 11) all iterative fixed points in the result are also classified as correct by manual verification. These observations suggest that our iterative fixed point check works well for larger depths. It can be somewhat unreliable for lower depths, which however does not have a huge impact on our results as there are either way not many iterative fixed points when using low depth.

Looking at Figures 7.1 and 7.2 we also see that we could change our algorithm to exclude pictures as iterative fixed points when their Euclidean distance to their converged iterative prediction is high, for instance the stealth plane in Figure 7.1 with the distance 472.81. However a meaningful distance to start excluding pictures is not easy to determine, since we have pictures like the red frog in the top right of Figure 7.1 that has a Euclidean distance of 14.59 and could be argued to be an iterative fixed point, although we have several other "false positives" with considerable lower distance. This discussion suggests that the Euclidean distance might not be the best measurement for picture to picture comparison. This is also supported by the deer with distance 12.32 in Figure 7.1 as manual verification tells us that its iterative prediction clearly resembles the red frog but – using the Euclidean distance for comparison – this iterative prediction is apparently closer to the deer than the red frog.

9 Discussion

9.1 ReLU autoencoders of different sizes

With this thesis we have gotten a better understanding for how width and depth affects our trained ReLU autoencoders: the analysis and our results show that reducing width or depth of the autoencoders has a negative impact on the amount of attractors. This trend has already been described in Figure 6 in [1] for SELU autoencoders (i.e. networks using SELU activations instead of ReLU), which we now complement for ReLU autoencoders with our Tables 7.1a and 7.1c. In particular, we want to point out that all four ReLU autoencoders with bias, depth 11 and width 128 that we trained using random pictures resulted in that every single training image became an attractor, although we did not even train sufficiently long to reach as low loss values as reported in [1]. These still relatively small autoencoders can serve as a good starting point for attempting a mathematical proof that random training data yields 100% attractors for sufficiently high width and depth.

We do however not have an exact configuration of when the amount of attractors start to decline when reducing width or depth, and would require a more fine grained table to describe the behaviour of the impact on our autoencoders with random pictures. Another thing that might have relevance, especially with our CIFAR10 pictures where we did not reach 100 attractors, is the amount of epochs trained, as this was limited in this thesis. Our research does not study the effect of the time trained on the numbers of attractors. We have also seen in the analysis that the loss values might be less relevant than what we originally believed when it comes to the impact on the amount of attractors. This shows that it might have been ignorant to exclude some models during preliminary experiments because of similar or worse loss values on higher depth and width models in comparison to the ones with lower depth and width (see Section 5.2).

9.2 Eigenvalues of Jacobian matrices

Using the eigenvalues, or rather the maximum absolute value of the eigenvalues, of a Jacobian matrix is a useful tool in the investigation of attractors. However, there is a slight hindrance when using this tool under our circumstances: our autoencoders just have not been able to reach a good enough loss, in contrast to the experiments in [1] and [3], where they introduce this method relying on considerably low losses. As we in this thesis work with less than exemplary autoencoders due to the limitation of time, we can only use this tool to exclude pictures that are not attractors if they have an eigenvalue with absolute value above 1. We cannot reliably use the other end of this tool to say that a picture is an attractor by seeing that the absolute values of all eigenvalues are below 1, since the latter condition only means that the perturbation of a picture will behave the same as the picture itself when predicting iteratively.

This in turn made us introduce our notion of iterative fixed points. With this notion and our implementation we could exclude many more training images from being attractors (if their iterative predictions converge to a completely different image, although all their eigenvalues were less than one in absolute value). We have however seen that our results still report some “false positives” but these are rather few and occur only in lower depth autoencoders.

9.3 The impact of bias

We have been able to investigate the impact of not using bias as we train ReLU autoencoders with CIFAR10 or with random pictures and can clearly state that bias is necessary to create attractors in the range of width and depth we investigated.

9.4 Random vs. structured pictures

Our results in Tables 7.1c and 7.1d approximate the the expected number of attractors for uniformly distributed training data. To the best of our knowledge, this has not been reported in the literature before.

We have found differences in how autoencoders trained with random or structured pictures behave in memorization with the same training parameters. These differences show in both the amount of attractors produced as well as the loss values achieved while training with our setups. They could stem from the fact that the structure of the data in CIFAR10 with patches of the same color creates 'less parameters' to learn and thus are easier to learn by smaller autoencoders. For instance, we can see several cases when comparing Tables 7.1a and 7.1c where our CIFAR10 experiments yield more attractors than training with random images. However, as most of our larger autoencoders produce better results with random data, it seems as if the structure is more of a hindrance to complete memorization.

10 Conclusion

The primary goal of this thesis is to extend experiments made in article [1] and broaden the understanding of attractors in autoencoders and the use of eigenvalues of Jacobian matrices to find these.

The first thing we did was to find an angle on which we wanted to broaden this, as the field is extensive and one cannot do all experiments at once. We restricted ourselves to the use of the RMSProp optimizer as well as ReLU autoencoders which are relevant for practical purposes but still relatively simple to understand theoretically.

To build this broadened understanding we use the method of controlled experiments. We study autoencoders of three different widths and five different depths with two types of training pictures (randomized and CIFAR10), while also analysing the impact of bias. We train and evaluate each setup four times to create averages for a total of 144 experiments which we did not only analyze the results from, but also manually verified to see how our algorithms behaved.

We see an interesting behaviour when training with random pictures in comparison to the more structured pictures in CIFAR10; not only did we manage to produce ReLU autoencoders with 100 attractors out of 100 possible training pictures, but we also saw that depth had less of an impact than width had when training with specifically random pictures. This behaviour was not the same with CIFAR10 pictures but showed that depth instead had a greater impact on the number of attractors. Another more surprising result was that the loss reached by training had less impact than expected on the number of attractors. Our experiments also verify the claims from articles [1], [3] and [4] that sufficient depth is required for memorization.

These experiments use the TensorFlow machine learning platform and was at first intended to only run within JavaScript, but as some functionality was missing for JavaScript, we included Python to do calculations with Jacobian matrices and their eigenvalues. We noticed that we had to implement these calculations ourselves as they just were not fast enough using GradientTape within Tensorflow. Specifically for the implementation it would have been easier to configure everything within the Pyhon environment to start with, as using two environments only make things more complicated.

As we analyzed the results, we found that our own notion of an iterative fixed point (based on the fixed points in article [1]) was producing some "false positives" on lower depths where memorization is flawed, however produces accurate results at depth six and up.

10.1 Future work

As both training autoencoders and computing Jacobians and their eigenvalues take quite a bit of time on a normal computer, experiments and the general scope of the thesis had to be reduced. The following problems would have been interesting, not only for this thesis but for the general understanding of the subject, are not included:

- As we decided on a fixed number of epochs to train, which could not be too large, the effect of the amount of epochs on memorization could be looked into.
- As we could not train all possible autoencoders within the range of width 1–128 and depth 1–11, running more experiments within these ranges and create a more fine grained table than Table 7.1 would give a better understanding of the effects of width and depth on the number of attractors.

- As time to train increases the larger an autoencoder becomes in width and depth, many of the larger autoencoders got excluded; this was also partly because the losses did not seem to improve in the amount of epochs trained in comparison to the lower values of width and depth. But since we observed that loss has less of an impact than we previously believed, these larger autoencoders would also be interesting to experiment on to see their results (especially for CIFAR10 where we did not reach 100 attractors).

- In this thesis, we decided to train on 100 pictures being inspired by [1]. However, a study on how the size of the training set affects the percentage of attractors would also be interesting to see.

- Another item of interest is how the structure of a picture affects the autoencoders memorization. A clear relation between larger patches of one color and a picture becoming an iterative fixed point for our autoencoders trained on CIFAR10 is visible in Figures 7.1, 7.2 and A.1. This could be investigated by only using training pictures with black background, or using one picture but with different colors.

- Optimizing our notion of iterative fixed points so that manual verification is no longer necessary to rule out "false positives" is important for larger experiments.

- It would be an enrichment for the theory of machine learning to give a mathematically rigorous proof that every data point in a sufficiently generic training set becomes an attractor for well-trained and sufficiently large ReLU autoencoders.

References

[1] A. Radhakrishnan, M. Belkin, C. Uhler, "Overparameterized Neural Networks Can Implement Associative Memory," 2019, arXiv:1909.12362.

[2] A. Krizhevsky, V. Nair, G. Hinton, "The CIFAR-10 dataset," 2009, dataset. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[3] A. Radhakrishnan, K.D. Yang, M. Belkin and C. Uhler, "Memorization in over-parameterized autoencoders," in *International Conference on Machine Learning (ICML 2019)*, 2019, arXiv:1810.10333.

[4] C. Zhang, S.Bengio, M.Hardt, M.C.Mozer, Y.Singer, "Identity Crisis: Memorization and Generalization under Extreme Overparameterization," in *International Conference on Learning Representations (ICLR 2020)*, 2019, arXiv:1902.04698.

[5] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2016.

[6] P. Saxena, D. Singh, M. Pant, *Problem Solving and Uncertainty Modeling through Optimization and Soft Computing Applications*. IGI Global, 2016.

[7] "RMSprop," tensorflow v2.1 API - docs. [Online]. Available: https://js.tensorflow.org/api/latest/#train.rmsprop

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] J. Strandqvist, "Experiment code and models," 2020, code and models at github. [Online]. Available: https://github.com/Lendzin/2dv50e_bachelorthesis.git

[10] "Sequential model," tensorflow v2.1 API - docs. [Online]. Available: https://js.tensorflow.org/api/latest/#sequential

A Appendix

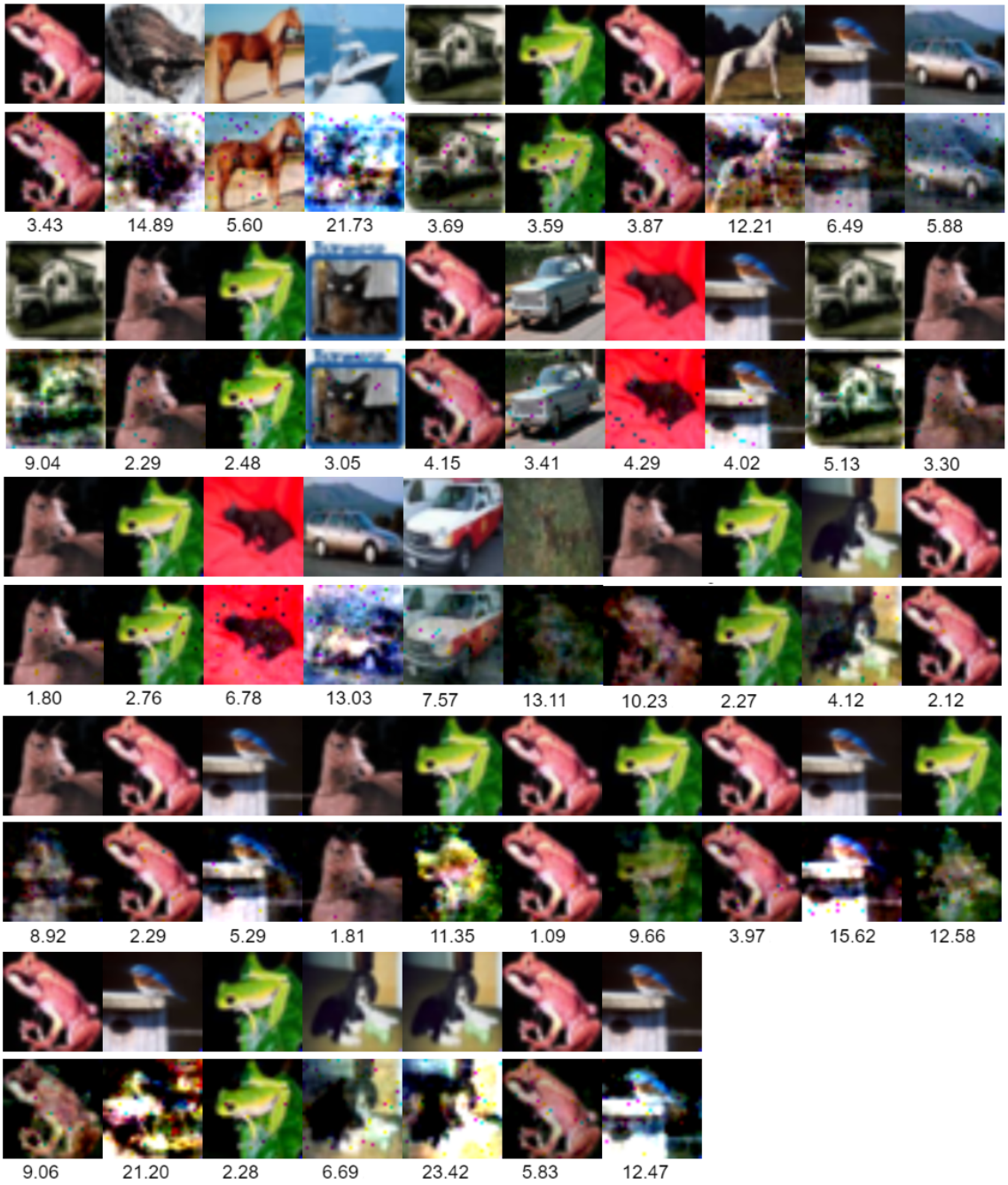

Figure A.1: Depth-3 iterative fixed points.