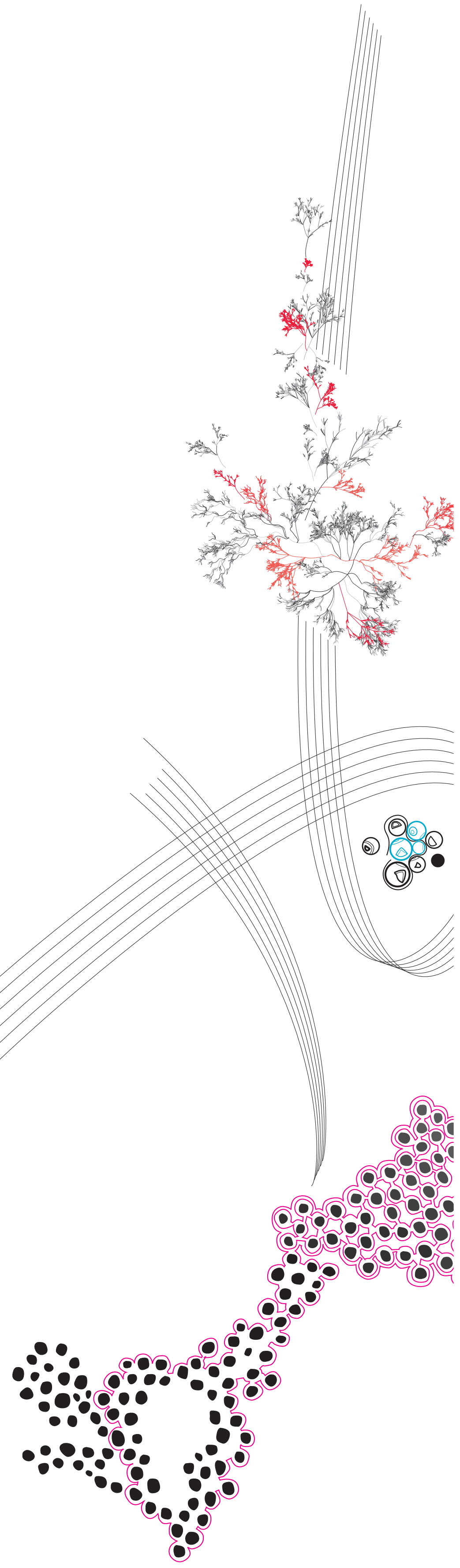


BSc Thesis Applied Mathematics

Saddle-to-saddle behaviour in shallow diagonal linear neural networks

Ylona van de Kerk

Supervisor: H. Meijer, L. Spek

July, 2024

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This thesis has been written as part of my bachelor Applied Mathematics at the University of Twente.
I want to thank my supervisors Hil Meijer and Len Spek for suggesting this interesting topic and for their guidance, insightful feedback and our discussions about this problem.

Saddle-to-saddle behaviour in neural networks

Ylona. van de Kerk[*]

July, 2024

Abstract

Neural networks are usually trained with the gradient descent method, which leads to minima of the loss function that generalize well to new data. However, it is still unclear why this method works so well. It has been observed that for a neural network with small weight initialization, the minimisation of the loss function makes very little progress for some time, until a sharp transition to a lower value occurs, which corresponds to a new feature that is learned. This incremental learning process corresponds to the jumping from saddle to saddle point of the loss function. The authors of [2] describe this saddle-to-saddle behaviour of the gradient flow in a shallow diagonal linear neural network in the limit of vanishing initialization without restrictive assumptions on the data. Motivated by this, this study determines the saddle points of the loss function and investigates the influence of these points on the minimum solution found with gradient descent in a shallow diagonal linear neural network with small weight initialization. We found that the equilibrium points of the gradient flow correspond to sparse vectors that minimize the loss function over its non-zero coordinates. Furthermore, we conclude that the direction of the jumps does not correspond linearly with the expected direction indicated by the eigenvector corresponding to the largest positive eigenvalue of the linearized gradient flow.

Keywords: saddle-to-saddle behaviour, shallow diagonal linear neural networks, weight initialization

[*]Email: y.vandekerk@student.utwente.nl

Contents

1 Introduction

Neural networks are trained with training data, which consists of examples of input/output pairs. The training is done by searching through the family of possible equations relating input to output to find the one that describes the training data most accurately. That is, we minimize the degree of mismatch between the outcome of the neural network and the training data, called the loss of this neural network. The loss depends on the parameters, also called weights, of the neural network. The gradient descent method is widely used to find the parameters that minimize the loss function. It leads to minima that generalize well to new data [4]. However, it is still unclear why this method works so well. One way to understand why the gradient descent method leads to certain minima, is to describe the full trajectory. It has been observed that learning curves of the gradient descent method with small initialisations across the training of a neural network are piecewise constant [1]. This means that the minimisation of the loss function makes very little progress for some time, until a sharp transition to a lower value occurs, which corresponds to a new feature that is learned. This incremental learning process corresponds to the jumping from one saddle point of the loss function to another saddle point [2].

Recent papers study this saddle-to-saddle behaviour of linear neural networks in the limit of vanishing initialization. The authors of [2] describe the trajectory of the gradient flow over a 2-layer diagonal linear network in the limit of vanishing initialization without restrictive assumptions on the data. The change in the parameters over time is called the gradient flow. The authors of [2] describe in full detail the jumping of the gradient flow from one saddle of the linear loss function to another until reaching a minimum solution.

The goal of this research is to describe the jumping from saddle to saddle of the gradient flow in detail, to be able to define the influence of these saddle points on the minimum of the loss function, obtained with gradient descent. This influence is studied by characterizing the direction of the jumps at each saddle point.

First, we discuss some theory related to this thesis, such as general information about neural networks, the gradient descent algorithm and the gradient flow. Next, the saddle-to-saddle behaviour of a shallow, diagonal, linear neural network is discussed. First, this is done analytically for simple neural networks and afterwards we look at the numerical analysis for wider neural networks.

2 Neural networks

This section uses [4] to give an introduction to the concept of neural networks and to explain the gradient descent method. Chapter 2-4 are used for section 2.1 and chapter 6 for 2.2. Furthermore, in this section, the gradient flow is introduced which we will use extensively in the next sections.

2.1 Introduction to neural networks

Neural networks are functions $\mathbf{y}_{net} = f[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs $\mathbf{x}$ to multivariate outputs $\mathbf{y}_{net}$. We assume that the input and output are vectors of a predetermined and fixed size and that the elements of each vector are always ordered the same way (structured data).

A neural network is trained with training data, which consists of examples of input/output pairs. Training a neural network is done by searching through the family of possible equations relating input to output to find the one that describes the training data most accurately. In other words, fitting mathematical models to the training data.

A regression model outputs a single real number. Supervised learning models map input data to an output prediction. When the inputs are passed through the network, and the output is computed, this is called inference.

An important aspect of a neural network is its parameters. Different parameter values change the outcome of the model. The model equation describes a family of possible relationships between inputs and outputs, and the parameters specify the particular relationship. Training/learning a model means finding the parameters that describe the true relationship between input and outputs the best.

The loss of a neural network is defined as the degree of mismatch between the outcome of the model that uses parameters ϕ, and the training data. It is denoted as $L[\phi]$.

When we train the model, we are seeking,

$$\hat{\boldsymbol{\phi}} = argmin_{\boldsymbol{\phi}}[L[\boldsymbol{\phi}]]. \tag{1}$$

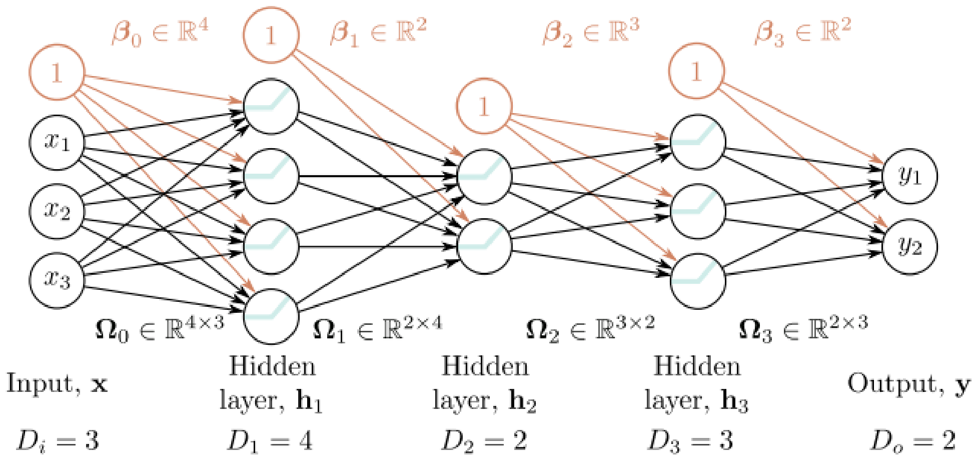
This $\hat{\boldsymbol{\phi}}$ is called a minimizer of $L[\boldsymbol{\phi}]$.

A common loss function is the quadratic loss:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{n} (f[x_i, \boldsymbol{\phi}] - y_i)^2, \tag{2}$$

where I denotes the number of training data samples. This loss function is usually chosen because the calculation of its partial derivatives is straightforward.

A neural network can consist of many different layers, which each consist of a number of nodes. Figure 1 shows an example of a neural network.



$\boldsymbol{\beta}_0 \in \mathbb{R}^4$
$\boldsymbol{\beta}_1 \in \mathbb{R}^2$
$\boldsymbol{\beta}_2 \in \mathbb{R}^3$
$\boldsymbol{\beta}_3 \in \mathbb{R}^2$
1
x_1
x_2
x_3
y_1
y_2
$\boldsymbol{\Omega}_0 \in \mathbb{R}^{4\times 3}$
$\boldsymbol{\Omega}_1 \in \mathbb{R}^{2\times 4}$
$\boldsymbol{\Omega}_2 \in \mathbb{R}^{3\times 2}$
$\boldsymbol{\Omega}_3 \in \mathbb{R}^{2\times 3}$
Input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{y}$
$D_i = 3$
$D_1 = 4$
$D_2 = 2$
$D_3 = 3$
$D_o = 2$

FIGURE 1: An example of a deep neural network with one input layer $\boldsymbol{x}$ of width 3, three hidden layers $\boldsymbol{h_i}$ for $i = 1, 2, 3$ of width 4, 2 and 3 respectively, and one output layer $\boldsymbol{y}$ of width 2. For each layer j, layer $j+1$ is computed by multiplying layer j with weights $\boldsymbol{\Omega_j}$ and adding bias $\boldsymbol{\beta_j}$. ([4] section 4.4.1 figure 4.6)

Neural networks map input to output by these three steps in each layer:

1. compute linear functions of the previous layer,

2. pass the result through an activation function $a[\cdot]$,

3. weigh the resulting activations with parameters and add an offset.

This offset that is added to each layer is called the bias of that layer (denoted by β_i in figure 1 for each layer i), and the weights of a layer are parameters with which the values of the previous layer are multiplied (denoted by Ω_i in 1 for each layer i).

A common choice for activation function is the ReLU function, defined as follows:

$$a[z] = ReLU[z] = \begin{cases} 0, & \text{if } z < 0, \\ z, & \text{if } z \geq 0. \end{cases} \tag{3}$$

The layers apart from the input and the output layer are called hidden layers. The network depicted in 1[4] for example has three hidden layers. A neural network that has one hidden layer is called a shallow network. A network with more than one hidden layer is called a deep neural network. Each layer has a number of nodes. For example, the first layer in figure 1 has three nodes and the first hidden layer has four nodes. The number of nodes of the input layer is denoted by D_i, the number of nodes of the hidden layers are denoted by D_j for each hidden layer j, and the number of nodes of the output layer is denoted by D_o.

The width of a neural network is the number of hidden units in each layer, the depth is the number of hidden layers, and the capacity is the total number of hidden units.

The number of layers of a neural network, denoted by K, and the number of hidden units in each layer are hyperparameters, since they are chosen before we learn the model parameters.

A neural network with K layers and input data $\mathbf{x}$ can be represented as:

$$\begin{aligned} \mathbf{h_1} &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h_2} &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h_3} &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\ \mathbf{y}_{net} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K, \end{aligned} \tag{4}$$

where $\mathbf{h}_j$ denotes the values of the nodes of hidden layer j and $\mathbf{y}_{net}$ denotes the network output.

The parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^{K}$ determine the particular outcome of the model. We can also write (4) as a single function:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}[...\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}]]...]]. \tag{5}$$

A neural network is trained by repeating the following two steps:

1. compute the derivatives (gradient) of the loss with respect to the parameters,

2. adjust the parameters based on the gradient to decrease the loss.

After many iterations, we aim to converge to a minimum of the loss function.

2.2 Gradient descent

To train a neural network, an optimization algorithm is used to find parameters $\hat{\boldsymbol{\phi}}$ as described in equation (1). We initialize the parameters experimentally and adjust them repeatedly in such a way that the loss decreases.

One way to adjust the parameters is to use the gradient descent algorithm, which is defined as follows: Start with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, ..., \phi_N]^T$ and iterate the following two steps:

1. compute the gradient of the loss with respect to the parameters:

$$\nabla L[\boldsymbol{\phi}] = \left[\frac{\partial L}{\partial \phi_0}, \frac{\partial L}{\partial \phi_1}, \ldots, \frac{\partial L}{\partial \phi_N}\right]^T, \tag{6}$$

2. update the parameters according to the rule:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha \nabla L[\boldsymbol{\phi}], \tag{7}$$

where the positive scalar α, called the learning rate, determines the magnitude of the change.

The gradient of the loss function is computed at the current position and indicates the direction of steepest ascent. So with gradient descent, we move a small distance in the direction of steepest descent. At the minimum of the loss function, the gradient will be zero, and the parameters will stop changing.

For n training samples, the gradient of the loss function with respect to its parameters can be computed by summing up the gradient for each data sample:

$$\nabla L[\boldsymbol{\phi}] = \sum_{i=1}^{n} \nabla l_i[\boldsymbol{\phi}], \tag{8}$$

where l_i denotes the loss with respect to one data sample:

$$l_i = (f[\mathbf{x_i}, \boldsymbol{\phi}] - y_i)^2. \tag{9}$$

Loss functions for linear regression problems always have a single well-defined global minimum. Since they are convex, this minimum is reached with gradient descent. Loss functions for most nonlinear models, including both shallow and deep networks, are non-convex. Which means that we may end up at local minima, or at a saddle point. With gradient descent, the final destination is entirely determined by the starting point.

In the appendix, section 7.1, we explain the backpropagation algorithm, which is used to calculate the gradient of the loss function for more complex neural network. This study focusses on shallow diagonal linear neural networks, for which the computation of the gradient is straightforward, so the backpropagation algorithm is not needed. However, it is explained in the appendix if the reader wants to extend this research to more complex neural networks.

2.3 Gradient flow

When we perform gradient descent, we apply the parameter update rule:

$$\boldsymbol{\phi_{t+1}} = \boldsymbol{\phi_t} - \alpha \nabla L[\boldsymbol{\phi}], \tag{10}$$

where $\boldsymbol{\phi_t}$ represents the parameters at time t and α is the learning rate. For the case of the linear regression model, the loss function is convex, and so we are guaranteed to reach the global minimum eventually if the learning rate is sufficiently small.

If we use an infinitesimally small learning rate, $\alpha \to 0$, we get:

$$\frac{\boldsymbol{\phi_{t+1}} - \boldsymbol{\phi_t}}{\alpha} = -\nabla L[\boldsymbol{\phi}], \tag{11}$$

which can be approximated by:

$$\frac{d\boldsymbol{\phi}}{dt} = -\nabla L[\boldsymbol{\phi}]. \tag{gradient flow}$$

This ODE is an example of gradient flow and tells us how the parameters change over time in the limit of $\alpha \to 0$.

Looking at this equation, we see that for zeros of the gradient of the loss function, the parameters do not change in time. This means that for a non-convex loss function, equilibrium points, $\boldsymbol{\phi}$, that do not correspond to the global minimum of the loss function, delay the parameter optimization, since $\frac{d\boldsymbol{\phi}}{dt} = 0$. This results in a constant loss function for the time spend in the neighbourhood of these saddle points. Only after many iterations, we may escape the neighbourhood of these saddle points. It has been observed that if the parameters of the network are initialized small, the loss function is piecewise constant with respect to the time [2], which means that eventually we will escape the neighbourhood of a saddle point. Our goal in this report is to locate these points and analyse the behaviour of the parameter optimization in the region around these points. This behaviour of a neural network that has a piecewise constant loss function is called the saddle-to-saddle behaviour of a neural network.

3 Methodology

In this section, we introduce the problem setup, which we use to look into the saddle-to-saddle behaviour of a particular type of neural network, namely diagonal linear neural networks. These networks are relatively simple, but they are ideal surrogate models to gain a deeper understanding of complex phenomenons such as the saddle-to-saddle dynamics [2]. The problem set-up discussed in this section is the same as the experimental setup described in section 2.1 and appendix A of [2].

We study a linear regression problem, with n data samples, such that $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ for every sample $i \in [0, n]$. We denote $x \in \mathbb{R}^{d \times n}$ as the input and $y \in \mathbb{R}^n$ as the output data. The neural network maps the input x to the linear function px, that is, $x \mapsto \langle p, x \rangle$. We use the quadratic loss for the parameter optimization:

$$L(p) = \frac{1}{2n} \sum_{i=1}^{n} (\langle p, x_i \rangle - y_i)^2, \tag{12}$$

where x_i, y_i are the input and output of training sample i. This loss function is convex, so a well-defined global minimum can be found with the gradient descent algorithm, as mentioned in section 2.2.

To study ODE (gradient flow) a 2-layer linear neural network of width d is considered as shown in figure 2, where $diag(v) \in \mathbb{R}^{d \times d}$, for $v \in \mathbb{R}^d$, are the inner weights and $u \in \mathbb{R}^d$ are the outer weights. The activation function is the identity function. The input vector $x \in \mathbb{R}^d$ is mapped to an output $y_{net} \in \mathbb{R}$, such that $y_{net} = \langle u, diag(v)x \rangle$. This corresponds to $p = u \odot v$, where $\odot$ denotes elementwise multiplication.

The quadratic loss function is described as follows:

$$F(u, v) = \frac{1}{2n} \sum_{i=1}^{n} (\langle u, diag(v)x_i \rangle - y_i)^2, \tag{13}$$

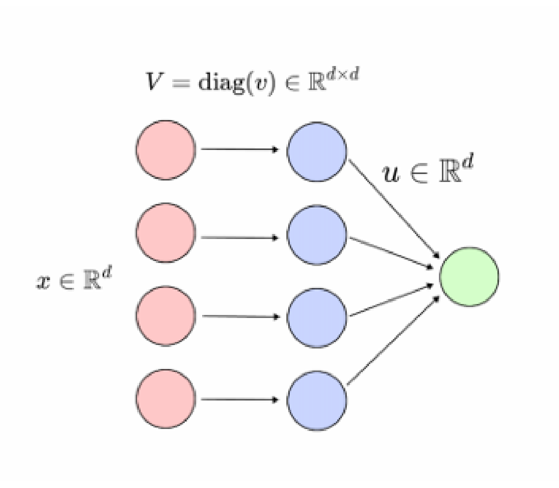


FIGURE 2: An example of a 2-layer diagonal neural network with width $d = 4$, inner weights $v \in \mathbb{R}^4$ and outer weights $u \in \mathbb{R}^4$. ([3] slide 9)

where x_i, y_i are the input and output data of a training sample i. This function is minimized by applying the gradient descent algorithm with a fixed learning rate denoted by α.

Note that the loss function $F(u, v)$ is non-convex, unlike the loss function $L(p)$, so we may not end up at a global minimum, when we apply the gradient descent method to $F(u, v)$.

Proposition 1 in [2] section 2.1 states that the saddles points of F correspond to sparse vectors that minimize L over its non-zero coordinates. The goal is to determine these saddles points of F and study the influence of these points on the parameter optimization. Note that these saddle points of F correspond to equilibrium points of (gradient flow).

The initialization of u and v, and therefore also, p is discussed in section 3.1. The data samples are generated as described in appendix A of [2]: $y_i = \langle x_i, p^* \rangle$ for every sample i, where $x_i = \mathcal{N}(0, H)$ for the identity covariance matrix H, and $p^* \in \mathbb{R}^d$ is a global minimum of the loss function.

3.1 Weight initialization

As we are interested in the saddle-to-saddle behaviour of linear neural networks in the limit of vanishing initialization, we initialize the weights of our neural network as follows with small initialization scale γ:

$$v_0 = \mathbf{0} \in \mathbb{R}^d, u_0 = \sqrt{2}\gamma\mathbf{1} \in \mathbb{R}^d_{>0},$$

which results in $p_0 = \mathbf{0} \in \mathbb{R}^d$ independently of u_0. The limit $\gamma \to 0$ is taken to approach a zero weight initialization.

This initialization leads to a zero gradient of the loss function at time $t = 0$, which means that the parameters do not change in time. That is why many iterations of the gradient descent algorithm are needed for the parameters to change significantly. The authors of [2] 'speed up' time in the neighbourhood of the saddle points of the loss function by the bijection $t \to \ln(\frac{1}{\gamma})t$. This function that accelerates time is denoted as $\tilde{t}_\gamma(t)$. The accelerated time defines the learning rate of the gradient descent method as $\ln(\frac{1}{\gamma})\alpha$ in the neighbourhood of the saddle points. Outside this neighbourhood, the time is not accelerated, so the learning rate is defined by α. Using this accelerated time, we escape these saddle points of the loss function, such that we can find a global minimum solution, p^*_γ (γ is written in subscript to emphasize the dependency of p on γ).

The authors of [2] show in section 2.2 that in the limit of $\gamma \to 0$, p^*_γ converges to the minimum l_1-norm solution

$$p^*_{l_1} = argmin_{y_i=\langle x_i,p\rangle} \|p\|_1. \tag{14}$$

The l_1-norm of a vector is the sum of the absolute value of the vector's elements, i.e., $\|p\|_1 = \sum_{j=1}^d |p_i|$.

In the next sections, we will locate this minimum solution and look at the equilibrium points visited before we arrived at this solution, for several shallow diagonal linear networks.

4 Analytical analysis 2-layer linear neural network

In this section, we look at simple shallow diagonal linear neural networks and analyse the behaviour of (gradient flow) around its equilibria by computing these points and the linearization of (gradient flow) analytically.

4.1 Case $d = 1$ and $n = 1$

For simplicity, we first look at the case $n = d = 1$, and compute the equilibrium points of (gradient flow) analytically to understand how they are computed and how many such points exists. We have that $u, v, x, p, y \in \mathbb{R}$ and the weights are $\boldsymbol{\phi} = [v, u]^T$, such that $p = uv$. We assume that x, y are non-zero. The loss function is defined as

$$F(u, v) = \frac{1}{2}(uvx - y)^2, \tag{15}$$

and (gradient flow) can be written as

$$\begin{bmatrix} \frac{dv}{dt} \\ \frac{du}{dt} \end{bmatrix} = -\nabla F(u, v) = \begin{bmatrix} -(uvx - y)ux \\ -(uvx - y)vx \end{bmatrix} = \begin{bmatrix} -u^2vx^2 + yux \\ -uv^2x^2 + yvx \end{bmatrix}. \tag{16}$$

This is a nonlinear system of differential equations, which is autonomous.

An autonomous, non-linear system of differential equations can be analysed using its linearization at the equilibria. For this, we compute the Jacobian of $-\nabla F(u, v)$ and substitute the equilibrium points to define their stability.

Equilibrium points of system (16) are points, ϕ, such that the right-hand side is the zero vector, so we need to find the zeros of $-\nabla F(u, v)$.

It can be seen that $[v, u] = [0, 0]$ is an equilibrium point, which corresponds to $p = 0$. If we require $v \neq 0, u = 0$, $[v, u]$ is an equilibrium point if,

$$yvx = 0$$

The only solution of this equation is $v = 0$, i.e., the trivial solution $[0, 0]$. For a non-trivial solution, we require $u \neq 0, v \neq 0$. In this case $[v, u]$ is an equilibrium point if it satisfies

$$uvx - y = 0$$

That is $u = \frac{y}{vx}$, $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$ is an equilibrium point of (16).

For the linearization of (16), we compute the Jacobian of $-\nabla F(u,v)$ as follows,

$$J(u,v) = \begin{bmatrix} -u^2x^2 & -2uvx^2 + yx \\ -2uvx^2 + yx & -v^2x^2 \end{bmatrix}. \tag{17}$$

Substituting $\boldsymbol{\phi} = [0,0]^T$ yields,

$$J(0,0) = \begin{bmatrix} 0 & yx \\ yx & 0 \end{bmatrix}. \tag{18}$$

Its eigenvalues are $-xy$ and xy with multiplicity 1, and eigenvectors $[-1,1]^T$ and $[1,1]^T$ respectively. Therefore, we can conclude that $\boldsymbol{\phi} = [0,0]^T$ is a saddle point of $-\nabla F(u,v)$. In the neighbourhood of $\boldsymbol{\phi} = [0,0]^T$ the general solution to (gradient flow) can be written as,

$$\boldsymbol{\phi}(t) = C_1 e^{-xyt} \begin{bmatrix} -1 \\ 1 \end{bmatrix} + C_2 e^{xyt} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \tag{19}$$

where C_1 and C_2 are arbitrary constants. Now that we determined this general solution, we can predict the direction of the jump from the saddle point $\boldsymbol{\phi} = [0,0]^T$. If $xy > 0$ we expect the jump to be linearly in the direction of $[1,1]^T$. That is, as t increases, we expect to jump to a point, $\boldsymbol{\phi}$, for which $v = u$. If $xy < 0$ we expect the jump to be linearly in the direction of $[-1,1]^T$. That is, as t increases, we expect to jump to a point, $\boldsymbol{\phi}$, for which $-v = u$.

Substituting $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$ in the Jacobian yields,

$$J\Big(\frac{y}{vx}, v\Big) = \begin{bmatrix} -\frac{y^2}{v^2} & yx \\ yx & -v^2x^2 \end{bmatrix}. \tag{20}$$

Its eigenvalues are 0 and $-\frac{v^4x^2+y^2}{v^2}$ with multiplicity 1, and eigenvectors $[-\frac{v^2x}{y}, 1]^T$ and $[\frac{v^2x}{y}, 1]^T$ respectively. To define the stability of $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$, we use Lyapunov's Theorem. The loss function $F(u(t), v(t))$ is continuously differentiable and positive definite on $\mathbb{R}^{2d}$, with its minimum at $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$. It follows that,

$$\begin{aligned} \frac{d}{dt}F(u(t),v(t)) &= \frac{\partial F}{\partial u}\frac{du}{dt} + \frac{\partial F}{\partial v}\frac{dv}{dt} \\ &= (uvx - y)vx \cdot -(uvx-y)vx + (uvx-y)ux \cdot -(uvx-y)ux \\ &= -((uvx-y)vx)^2 - ((uvx-y)ux)^2 \\ &< 0, \end{aligned} \tag{21}$$

for $\boldsymbol{\phi} \neq [v, \frac{y}{vx}]^T$. So $\frac{d}{dt}F(u(t),v(t))$ is negative definite, which means that $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$ is an asymptotically stable minimizer of $F(u,v)$.

4.2 Case d=1 and n> 1

Now that we have looked at the simple case with one data sample, we extend it to multiple, n, samples. The loss function then becomes

$$F(u,v) = \frac{1}{2n}\sum_{i=1}^{n}(uvx_i - y_i)^2, \tag{22}$$

and (gradient flow) can be written as,

$$\begin{bmatrix} \frac{dv}{dt} \\ \frac{du}{dt} \end{bmatrix} = -\nabla F(u,v) = \frac{1}{2n}\sum_{i=1}^{n} \begin{bmatrix} -(uvx_i - y_i)ux_i \\ -(uvx_i - y_i)vx_i \end{bmatrix}. \tag{23}$$

Next, we define equilibrium points of (23). It can be seen that $\boldsymbol{\phi} = [0,0]^T$ is an equilibrium point and that for a non-trivial point $v \neq 0$ and $u \neq 0$. In this case, $[v,u]$ is an equilibrium point if it satisfies

$$\frac{1}{2n}\sum_{i=1}^{n}(uvx_i - y_i) = 0$$

That is $u = \frac{y_i}{vx_i}$ for every sample i. So $\boldsymbol{\phi} = \sum_{i=1}^{n}[v, \frac{y_i}{vx_i}]^T$, for every sample i, is an equilibrium solution of (23).

For the linearization of (23), we use the Jacobian computed for the case $n = 1$. We compute this Jacobian for every data sample and sum them up to linearize the ODE for multiple data points. That is,

$$J(u,v) = \frac{1}{2n}\sum_{i=1}^{n} J_i(u,v) = \frac{1}{2n}\sum_{i=1}^{n} \begin{bmatrix} -u^2x_i^2 & -2uvx_i^2 + y_ix_i \\ -2uvx_i^2 + y_ix_i & -v^2x_i^2 \end{bmatrix}. \tag{24}$$

Substituting the equilibrium point $\boldsymbol{\phi} = [0,0]^T$ yields,

$$J(0,0) = \frac{1}{2n}\sum_{i=1}^{n} J_i(0,0) = \frac{1}{2n}\sum_{i=1}^{n} \begin{bmatrix} 0 & y_ix_i \\ y_ix_i & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{2n}\sum_{i=1}^{n} y_ix_i \\ \frac{1}{2n}\sum_{i=1}^{n} y_ix_i & 0 \end{bmatrix}. \tag{25}$$

Its eigenvalues are $\lambda_1 = -\frac{1}{2n}\sum_{i=1}^{n} x_iy_i$ and $\lambda_2 = \frac{1}{2n}\sum_{i=1}^{n} x_iy_i$ with multiplicity 1, and eigenvectors $[-1,1]^T$ and $[1,1]^T$ respectively. Therefore, $\boldsymbol{\phi} = [0,0]^T$ is a saddle point of network with $d = 1$ and multiple data samples. In the neighbourhood of $\boldsymbol{\phi} = [0,0]^T$ the general solution to (gradient flow) can be written as,

$$\boldsymbol{\phi}(t) = C_1e^{\lambda_1 t}\begin{bmatrix} -1 \\ 1 \end{bmatrix} + C_2e^{\lambda_2 t}\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \tag{26}$$

where C_1 and C_2 are arbitrary constants. Now that we determined this general solution, we can predict the direction of the jump from the saddle point $\boldsymbol{\phi} = [0,0]^T$. If $\frac{1}{2n}\sum_{i=1}^{n} x_iy_i > 0$ we expect the jump to be linearly in the direction of $[1,1]^T$. That is, as t increases, we expect to jump to a point, $\boldsymbol{\phi}$, for which $v = u \neq 0$. If $\frac{1}{2n}\sum_{i=1}^{n} x_iy_i < 0$ we expect the jump to be linearly in the direction of $[-1,1]^T$. That is, as t increases, we expect to jump to a point, $\boldsymbol{\phi}$, for which $-v = u \neq 0$.

Substituting the equilibrium point $\boldsymbol{\phi} = \sum_{i=1}^{n}[v, \frac{y_i}{vx_i}]^T$ into the Jacobian yields,

$$J(u,v) = \frac{1}{2n}\sum_{i=1}^{n} J_i(\frac{y_i}{vx_i}, v) = \frac{1}{2n}\sum_{i=1}^{n} \begin{bmatrix} -\frac{y_i^2}{v^2} & y_ix_i \\ y_ix_i & -v^2x_i^2 \end{bmatrix} \tag{27}$$

$$= \begin{bmatrix} -\frac{1}{2n}\sum_{i=1}^{n} \frac{y_i^2}{v^2} & \frac{1}{2n}\sum_{i=1}^{n} y_ix_i \\ \frac{1}{2n}\sum_{i=1}^{n} y_ix_i & -\frac{1}{2n}\sum_{i=1}^{n} v^2x_i^2 \end{bmatrix}. \tag{28}$$

The computation of the eigenvalues of this matrix gets very complicated once we increase n. Therefore, we look into the numerical computation of the values of this matrix in the next section to be able to classify this equilibrium point.

In the appendix, section 7.2, we look at a shallow diagonal linear neural network with $d = 2$ and compute its equilibrium points.

4.3 General case

For a shallow diagonal linear network with n data samples and width d, the network is represented by $y \in \mathbb{R}^n$, $x \in \mathbb{R}^{d\times n}$ and $u, v, p \in \mathbb{R}^d$, such that $\boldsymbol{\phi} = [v, u]^T$, $p = u \odot v$ and the loss function is written as:

$$F(u, v) = \frac{1}{2n}(\langle u, v \odot x\rangle - y)^2. \tag{29}$$

ODE (gradient flow) can be written as:

$$\begin{bmatrix} \frac{dv_1}{dt} \\ \vdots \\ \frac{dv_d}{dt} \\ \frac{du_1}{dt} \\ \vdots \\ \frac{du_d}{dt} \end{bmatrix} = -\frac{1}{2n}\sum_{i=1}^{n} \begin{bmatrix} (u_1v_1x_{1,i} + u_2v_2x_{2,i} + \cdots + u_nv_nx_{n,i} - y)u_1x_{1,i} \\ \vdots \\ (u_1v_1x_{1,i} + u_2v_2x_{2,i} + \cdots + u_nv_nx_{n,i} - y)u_dx_2 \\ (u_1v_1x_{1,i} + u_2v_2x_{2,i} + \cdots + u_nv_nx_{n,i} - y)v_1x_{1,i} \\ \vdots \\ (u_1v_1x_{1,i} + u_2v_2x_{2,i} + \cdots + u_nv_nx_{n,i} - y)v_dx_{2,i} \end{bmatrix}, \tag{30}$$

where $x_{j,i}$ denotes element j of the ith sample vector x.

It can be seen that $\boldsymbol{\phi} = \mathbf{0}$ is a trivial equilibrium point of (30), which corresponds to $p = \mathbf{0}$. It can also be seen that, similar to the previous cases we analysed, a non-trivial equilibrium point requires $u_j \neq 0$ and $v_j \neq 0$, which correspond to $p_j \neq 0$, for at least one value $j \in [0, d]$.

5 Numerical analysis 2-layer linear neural network

Now that we have analysed the ODE (gradient flow) analytically for simple neural networks, we look into the numerical analysis of these networks to compare our results. Furthermore, we look at neural networks with a higher width that use more data points and analyse the behaviour of (gradient flow) around the equilibria by looking at its linearization.

This section uses numerical results which are made by implementation of a shallow diagonal linear neural network in the programming language Python.

5.1 Implementation shallow diagonal linear neural network

A shallow diagonal linear neural network is implemented, for which the width, the number of data points as well as the step size of the gradient descent method and the weight initialization can be varied with.

First, all hyperparameters are set. These are:

- d, the width of the input and hidden layer,
- T, number of iterations,
- α, step size of gradient descent algorithm,

- n, number of data points,
- γ, weight initialization scale,
- margin, threshold for the size of $\nabla F(u, v)$.

Next, the data points are generated as described in section 3. We set a seed, such that we always get the same generated data points. The weights of the network are initialized according to section 3.1. Algorithm 1 describes the whole implementation, where next to the loss of every iteration also the gradient is computed, and which is used in the gradient descent step. If the gradient in one iteration is sufficiently small (maximum element smaller than the threshold), the weights corresponding to that iteration become our estimate of the equilibrium point. The exact value of this point is computed by solving for zeros of the gradient with our estimation. This is done by the scipy.optimize.root function in Python from the library Scipy. This function takes as input a system of equations that we need to find the roots of and an initial guess of the solution. It outputs the values of the variables that satisfy the system of equations and are close to the initial guess. Once we have computed this equilibrium point, we compute the Jacobian of $\nabla F(u, v)$ and substitute the equilibrium point to define its stability. Once the gradient surpasses the threshold, the weights change and we 'move' towards another zero of $\nabla F(u, v)$. The moment the gradient gets smaller again, we have arrived at this point, and we repeat the process of computing this equilibrium point. In algorithm 1 the size of the gradient is kept track of by the boolean plateau. If the gradient is small and plateau is true, we solve for the equilibrium point. Plateau is set to false in this case, such that we know not to solve for the equilibrium point again if the weights did not change. Once the gradient surpasses the threshold, boolean plateau is set to true, so that we know to solve for the equilibrium point once the gradient gets small again. Note that the accelerated time as described in section 3.1 is not implemented.

The loss function is plotted against the number of iterations as well as the values of the elements of p. The equilibrium points as well as the eigenvalues and eigenvectors of the linearization around these equilibrium points are printed. Furthermore, we keep track of the iteration at which the maximum element of the gradient is less than the margin, that is, the iteration at which we arrive at a plateau.

Algorithm 1 Computation equilibrium points

```
Set hyperparameters
Generate data points
Initialize weights
Compute network output
Compute initial loss value
plateau ← True
for t ← $1, T$ do
    Compute current values of neural network
    Compute gradient, $\nabla F(u, v)$
    if max $(\nabla F(u, v)) >$ margin then
        plateau ← True
    else if max $(\nabla F(u, v)) \leq$ margin and plateau == True then
        plateau ← False
        Compute zero of $\nabla F(u, v)$ using $\boldsymbol{\phi}_t$ as estimation
        Compute the Jacobian of $\nabla F(u, v)$ and substitute the computed equilibrium point
        Compute eigenvalues and eigenvectors of the Jacobian
    end if
    Update weights according to gradient descent step
    Compute the loss value
    Compute $p_t = u_t \odot v_t$
end for
```

5.2 Comparison analytical analysis

5.2.1 Case d=1 and n=1

First, we analyse the case $n = 1$ and $d = 1$ and check our results with the analytical analysis explained in section 4.1. Table 1 shows the values of the hyperparameters and the generated data. As explained in section 3.1 the weights are initialized as $v_0 = 0$ and $u_0 = \sqrt{2}\gamma\mathbf{1} = 1.4142 \cdot 10^{-100}\mathbf{1}$.

T	α	n	γ	margin	p^*	x	y
1000000	0.001	1	10^{-100}	10^{-4}	0.7153	1.3316	0.9525

Table 1: Values hyperparameters and generated data for a 2-layer diagonal linear neural network n=d=1.

Figure 3 shows the loss function over time as well as the values of p over time. Indeed, we observe that for small weight initialization, the loss function is piecewise constant. The weights contributing to the plateaus in the loss function are computed, and the results are as follows.

The first equilibrium point is $\boldsymbol{\phi} = [v, u]^T = [0, 0]^T$. The Jacobian at this point is,

$$J(0,0) \approx \begin{bmatrix} 0 & 1.2683 \\ 1.26836 & 0 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors,

$$\begin{cases} \text{eigenvalue } -1.2683, & \text{with eigenvector } [-0.7071, 0.7071]^T, \\ \text{eigenvalue } 1.2683, & \text{with eigenvector } [-0.7071, -0.7071]^T. \end{cases}$$

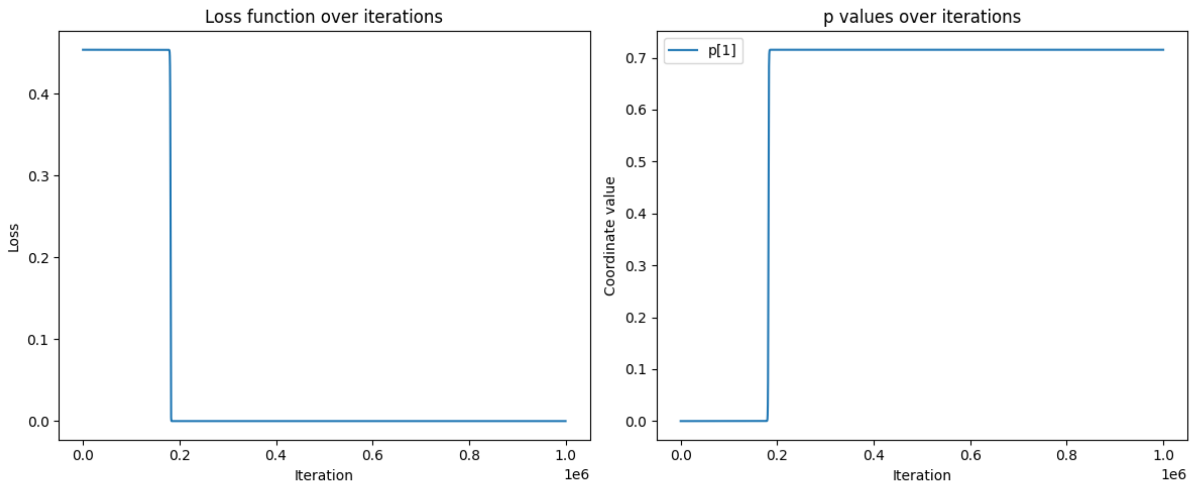


FIGURE 3: 2-layer diagonal neural network n=d=1, Left: the loss function over time, Right: the coordinates of p over time.

The second point is $\boldsymbol{\phi} = [v, u]^T = [0.8457, 0.8457]^T$, with Jacobian,

$$J(0.8457, 0.8457) \approx \begin{bmatrix} -1.2683 & -1.2683 \\ -1.2683 & -1.2683 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors,

$$\begin{cases} \text{eigenvalue } 0, & \text{with eigenvector } [0.7071, -0.7071]^T, \\ \text{eigenvalue } -2.5366, & \text{with eigenvector } [0.7071, 0.7071]^T. \end{cases}$$

These results agree with our analysis is section 4.1. The equilibrium points are $[0, 0]^T$ and $[v, \frac{y}{vx}]^T$, where $v = 0.8457$ in this case. The equilibrium point $\boldsymbol{\phi} = [0, 0]^T$ is indeed a saddle point, since the eigenvalues are of different sign. The point $\boldsymbol{\phi} = [v, \frac{y}{vx}]^T$ is indeed an asymptotically stable minimizer, since the eigenvalues are nonpositive and $p^* = uv = 0.7153$. So we can conclude that the loss function moves from a saddle point to a minimizer. Furthermore, it is interesting to note that $u = v$ for both equilibrium points which causes the Jacobian to be symmetric, so its eigenvalues are real. In section 4.1 we characterized the direction of the jump from $\boldsymbol{\phi} = [0, 0]^T$. The results agree with this jump direction determined in the analytical analysis, since for this example $xy > 0$ and we jumped linearly in the direction of $[1, 1]^T$ from $[0, 0]^T$.

5.2.2 Case d=1 and n=4

We analyse the case for more than one data point, by running the algorithm for a value of $n > 1$. The hyperparameters are the same as the previous case, table 1. Table 2 shows the values of the generated data.

p^*	0.6213
x	$[1.3316, 0.7153, -1.5454, -0.0084]^T$
y	$[0.8274, 0.4444, -0.9602, 0.0052]^T$

TABLE 2: Generated data for a 2-layer diagonal linear neural network n= 4 d=1.

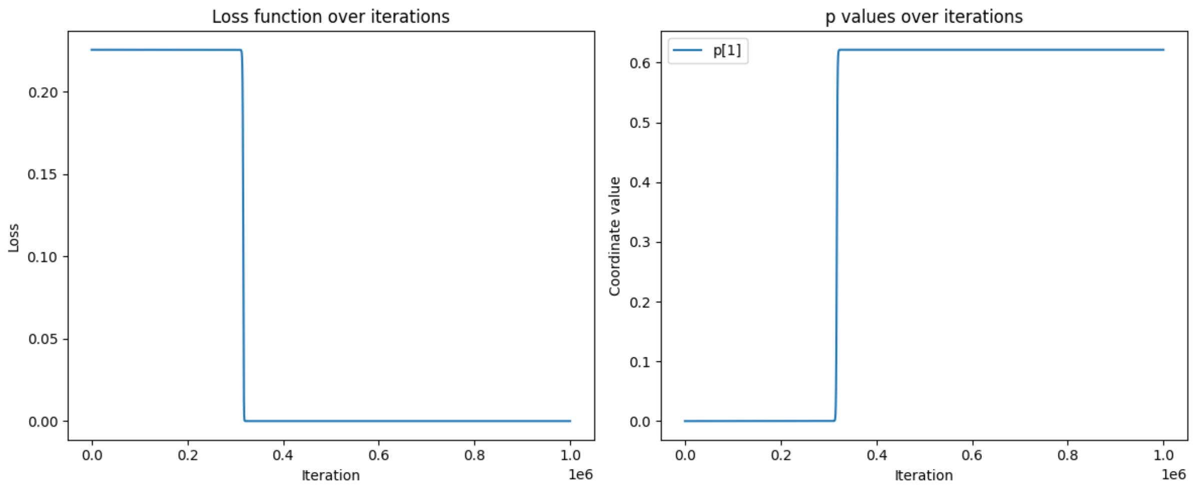


FIGURE 4: 2-layer diagonal neural network n=4 d=1, Left: the loss function over time, Right: the coordinates of p over time.

Figure 4 shows the loss function for the case of 4 data points.

The first point equilibrium point is $\boldsymbol{\phi} = [0, 0]^T$, with Jacobian

$$J(0,0) \approx \begin{bmatrix} 0 & 1.2683 \\ 1.2683 & 0 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors

$$\begin{cases} \text{eigenvalue } -2.0821, & \text{with eigenvector } [-0.7071, 0.7071]^T, \\ \text{eigenvalue } 2.0821, & \text{with eigenvector } [-0.7071, -0.7071]^T. \end{cases}$$

The second equilibrium point is $\boldsymbol{\phi} = [0.7882, 0.7882]^T$, with Jacobian,

$$J(0.7882, 0.7882) \approx \begin{bmatrix} -2.0821 & -2.0821 \\ -2.0821 & -2.0821 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors,

$$\begin{cases} \text{eigenvalue } 0, & \text{with eigenvector } [0.7071, -0.7071]^T, \\ \text{eigenvalue } -4.1642, & \text{with eigenvector } [-0.7071, -0.7071]^T. \end{cases}$$

These results can also be compared to our analytical analysis. Indeed, the equilibrium point $\boldsymbol{\phi} = [0, 0]^T$ is a saddle point and $\boldsymbol{\phi} = [v, \sum_{i=1}^{n} \frac{y_i}{vx_i}]^T$ is an asymptotically stable minimizer, since the eigenvalues are nonpostive and $p^* = uv = 0.6213$. Furthermore, it is also interesting to note that for different values of n, $u = v$. The results agree with our characterization of the jump direction in the analytical analysis, section 4.2, since for this example $\frac{1}{2n} \sum_{i=1}^{n} x_i y_i = 0.3629 > 0$ and we jumped linearly in the direction of $[1, 1]^T$ from $[0, 0]^T$.

In the appendix, section 7.3, we compare the results of our implementation for a diagonal linear neural network with $d = 2$ and $n = 1$, with the analytical analysis explained in 7.2.

5.3 Neural networks with higher width

In this section, we look at wider 2-layer diagonal linear networks with more data points, for which it was complicated to calculate and classify the equilibrium points analytically.

For several cases, we vary with the width of the layers, the number of data points, and give the resulting loss function, equilibrium points and the type of these points.

5.3.1 Case d=2 and n=2

Figure 5 shows the loss function as well as the coordinates of p, for hyperparameters shown in table 3 and generated data shown in table 4.

T	α	d	n	γ	margin
500000	0.01	2	2	10^{-50}	10^{-4}

TABLE 3: Values hyperparameters for a 2-layer diagonal linear neural network n=2 d=2.

p^*	$[0.6213, -0.7201]^T$
x	$[[1.3316, 0.7153]^T, [-1, 5454, -0.0084]^T]$
y	$[0.3123, -0.9542]^T$

TABLE 4: Generated data for a 2-layer diagonal linear neural network n= 2 d=2.

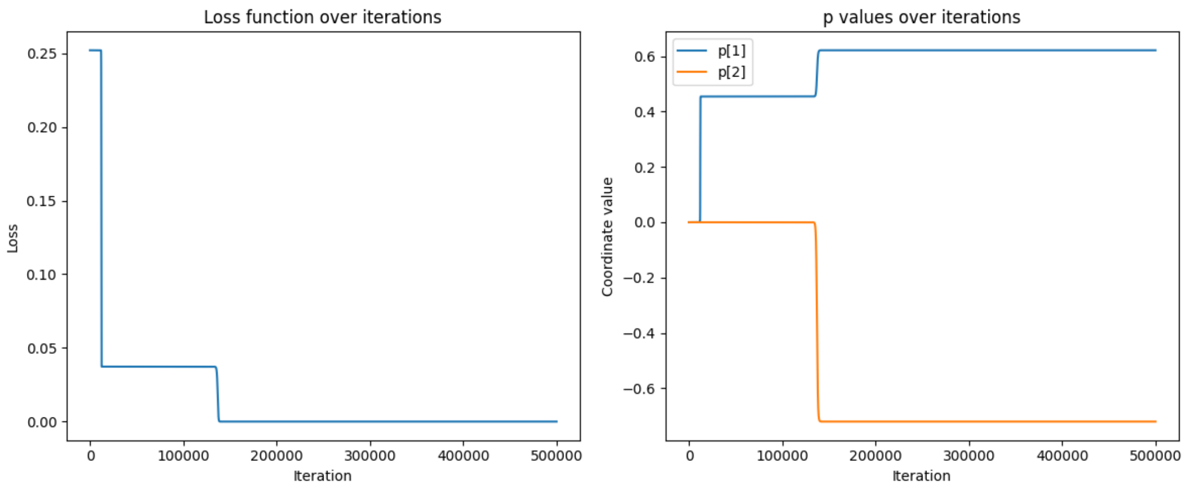


FIGURE 5: 2-layer diagonal neural network n=2 d=2, Left: the loss function over time, Right: the coordinates of p over time.

We observe that the loss function is a piecewise constant function with 3 plateaus. Table 5 shows the weights contributing to these plateaus, the type of these equilibrium points and the iteration at which the maximum element of the gradient is less than the margin for each plateau.

In the jump from the first saddle to the second, only the first coordinate of p gets activated. In the jump from the second saddle to the minimum, also the second coordinate gets activated. Also note that $|u| = |v|$ for all equilibrium points, similar to our observation

Equilibrium Point ϕ	**Type**	**Iteration**
$[0,0,0,0]^T$	Saddle	1
$[0.6740,0,0.6740,0]^T$	Saddle	12693
$[0.7882,-0.8486,0.7882,0.8486]^T$	Minimizer	140737

TABLE 5: Classification of equilibrium points and the iteration at which we arrive at the equilibrium points for a 2-layer diagonal linear neural network n=2 d=2.

in the previous section.

Table 6 shows the eigenvectors corresponding to the largest positive eigenvalue at each saddle point equilibrium point. We observe that for the first jump, from p_0 to p_1, the expected direction corresponds linearly with the actual direction of the jump. That is, if we move a distance, c, in the direction of the eigenvector corresponding to the largest positive eigenvalue, we arrive at the next saddle point, $p_{i+1} = p_i + cs_i$ for both $i = 1, 2$ (in this case, $c = \frac{0.45}{0.50} = 0.9$). The linearization at p_1 has two positive eigenvalues of almost the same value:

$$\begin{cases} \text{eigenvalue } 0.1948, & \text{with eigenvector } [-0.50, 0]^T, \\ \text{eigenvalue } 0.2082, & \text{with eigenvector } [0, -0.50]^T. \end{cases} \tag{31}$$

This means that both coordinates will change value in the jump from p_1 to p^*, only the second one will change more than the first. This corresponds with our result.

Equilibrium point ϕ_i $= [v_i, u_i]^T$, $p_i = u_i \odot v_i$	**Eigenvector** s_i
$p_0 = [0,0]^T$	$s_0 = [0.50, 0]^T$
$p_1 = [0.45, 0]^T$	$s_1 = [0, 0.50]^T$
$p^* = [0.62, -0.72]^T$	-

TABLE 6: The eigenvector, s_i, corresponding to the largest positive eigenvalue at saddle point $[v_i, u_i]$, which corresponds to $p_i = u_i \odot v_i$ for a 2-layer diagonal linear neural network $n = 2$ $d = 2$.

5.3.2 Case d=7 and n=5

Figure 6 shows the resulting plot. Table 7 shows the values of the hyperparameters and table 8 shows the generated data.

T	α	d	n	γ	margin
500000	0.001	7	5	10^{-100}	10^{-4}

TABLE 7: Values hyperparameters for a 2-layer diagonal linear neural network n=5 d=7.

We observe in figure 6 that the loss function has 5 plateaus. Table 9 shows the weights contributing to these plateaus, the type of these equilibrium points and the iteration at which the maximum element of the gradient is less than the margin for each plateau. Looking at the coordinates that get activated with each jump from an equilibrium point, we see that first only the third coordinate gets activated. In the second jump also the fifth

p^*	$[10, 20, 0, 0, 0, 0, 0]^T$
x	$[[1.3316, 0.7153, -1, 5454, -0.0084, 0.6213, -0.7201, 0.2655]^T,$ $[0.1085, 0.0041, -0.1746, 0.4330, 1.2030, -0.9651, 1.0283]^T,$ $[0.2286, 0.4451, -1.1366, 0.1351, 1.4845, -1.0798, -1.9777]^T,$ $[-1.7434, 0.2661, 2.3850, 1.1237, 1.6726, 0.0991, 1.3980]^T,$ $[-0.2712, 0.6132, -0.2673, -0.5493, 0.1327, -0.4761, 1.3085]^T]$
y	$[27.6214, 1.1713, 11.1891, -12.1123, 9.5516]^T$

TABLE 8: Generated data for a 2-layer diagonal linear neural network n= 5 d=7.

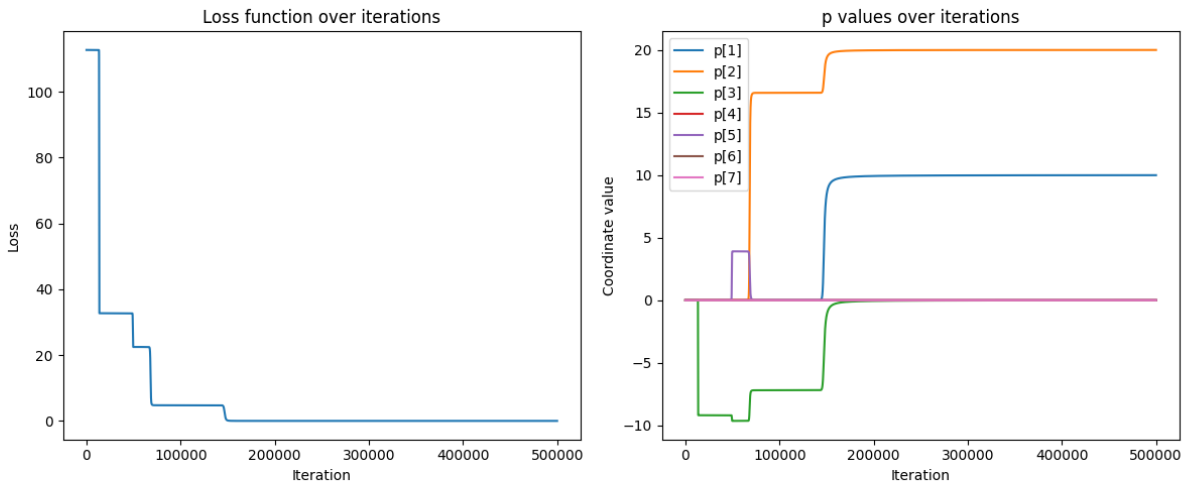


FIGURE 6: 2-layer diagonal neural network n=5 d=7, Left: the loss function over time, Right: the coordinates of p over time.

Equilibrium Point ϕ	**Type**	**Iteration**
$[0, 0, 0, 0, 0, 0, 0$ $0, 0, 0, 0, 0, 0, 0]^T$	Saddle	1
$[0, 0, -3.0318, 0, 0, 0, 0$ $0, 0, 3.0318, 0, 0, 0, 0]^T$	Saddle	13798
$[0, 0, -3.1054, 0, 1.9762, 0, 0$ $0, 0, 3.1055, 0, 1.9762, 0, 0]^T$	Saddle	50388
$[0, 4.0706, -2.6823, 0, 0, 0, 0$ $0, 4.0706, 2.6823, 0, 0, 0, 0]^T$	Saddle	82615
$[3.1623, 4.4720, -0.0012, 0, 0, 0, 0$ $3.1623, 4.4720, 0.0012, 0, 0, 0, 0]^T$	Minimizer	572728

TABLE 9: Classification of equilibrium points and the iteration at which we arrive at the equilibrium points for a 2-layer diagonal linear neural network n=5 d=7.

coordinate gets activated, but at the third jump this same coordinate gets deactivated again. After the last jump the first and second coordinate are active, and it looks like the third coordinate gets deactivated in this last jump, but if we zoom in we see that it is still active, see figure 7.

Five plateaus are seen in figure 6, but only four are recognized by our algorithm with the chosen hyperparameters. The last plateau, around iteration 150000, is not calculated, so cannot be classified. A reason for this could be that the transition is not sharp enough,

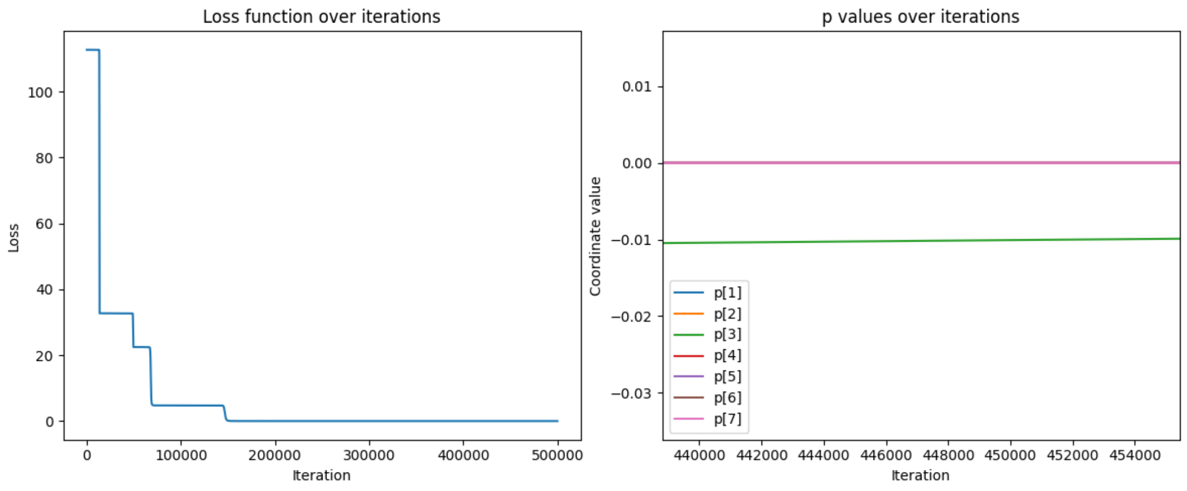


FIGURE 7: 2-layer diagonal neural network n=5 d=7, Left: the loss function over time, Right: the coordinates of p over time zoomed in at the fifth plateau.

i.e., the gradient does not surpass the threshold. Therefore, increasing the margin to 10^{-3} has been tried, but also in this case, the equilibrium point did not get computed. However, when we increased the number of iterations significantly, we are able to find the minimizer. With T set to 1000000, the last equilibrium point is classified as an asymptotically stable minimizer, see table 9. So the loss function jumps from four saddles to a minimizer at which the first, second and third coordinate are active. It is also observed that $|u| = |v|$ for all equilibrium points, similar to the previous cases.

Table 10 shows the eigenvectors corresponding to the largest positive eigenvalue at each saddle point equilibrium point. We observe that only for the first two jumps, the

Equilibrium point $\boldsymbol{\phi_i} = [v_i, u_i]^T$, $p_i = u_i \odot v_i$	**Eigenvector** s_i
$p_0 = \mathbf{0}$	$s_0 = -0.50\boldsymbol{e_3}$
$p_1 = -9.19\boldsymbol{e_3}$	$s_1 = 0.50\boldsymbol{e_5}$
$p_2 = -9.64\boldsymbol{e_3} + 3.91\boldsymbol{e_5}$	$s_2 = 0.50\boldsymbol{e_7}$
$p_3 = 1.65\boldsymbol{e_2} - 7.19\boldsymbol{e_3}$	$s_3 = 0.50\boldsymbol{e_7}$
$p^* = 10.00\boldsymbol{e_1} + 20.00\boldsymbol{e_2}$	-

TABLE 10: The eigenvector, s_i, corresponding to the largest positive eigenvalue at saddle point $[v_i, u_i]$, which corresponds to $p_i = u_i \odot v_i$ for a 2-layer diagonal linear neural network $n = 5$ $d = 7$.

expected direction corresponds linearly with the actual direction of the jump. That is, if we move a distance, c, in the direction of the eigenvector corresponding to the largest positive eigenvalue, we arrive at the next saddle point, $p_{i+1} = p_i + cs_i$ for $i = 1, 2$. For the last two jumps, the eigenvector does not correspond linearly with the direction of the jumps.

5.4 Non-zero weight initialization

In this section, we look at the loss function in the case that the weights are not initialized to zero. Instead of $v_0 = \mathbf{0}$, we set it to the weight-initialization scale $v_0 = \gamma\mathbf{1}$.

d=7 and n=5

With hyperparameters $T = 1000000$, $\alpha = 0.01$, the margin $= 10^{-4}$ and, similar to the previous case, $p^* = [10, 20, 0, 0, 0, 0, 0]$, we find the minimizers denoted in table 11 for different values of γ. The loss at all these minimizers is zero.

$v_0 = \gamma\mathbf{1}$	Minimizer $\boldsymbol{\phi}$						
γ	$\begin{bmatrix} v_1 \\ u_1 \end{bmatrix}$	$\begin{bmatrix} v_2 \\ u_2 \end{bmatrix}$	$\begin{bmatrix} v_3 \\ u_3 \end{bmatrix}$	$\begin{bmatrix} v_4 \\ u_4 \end{bmatrix}$	$\begin{bmatrix} v_5 \\ u_5 \end{bmatrix}$	$\begin{bmatrix} v_6 \\ u_6 \end{bmatrix}$	$\begin{bmatrix} v_7 \\ u_7 \end{bmatrix}$
10^{-50}	$\begin{bmatrix} 3.1621 \\ 3.1621 \end{bmatrix}$	$\begin{bmatrix} 4.4713 \\ 4.4713 \end{bmatrix}$	$\begin{bmatrix} -0.0012 \\ 0.0012 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$\frac{1}{10}$	$\begin{bmatrix} 3.1435 \\ 3.1459 \end{bmatrix}$	$\begin{bmatrix} 4.4689 \\ 4.4705 \end{bmatrix}$	$\begin{bmatrix} -0.3655 \\ -0.3789 \end{bmatrix}$	$\begin{bmatrix} 0.2836 \\ 0.3141 \end{bmatrix}$	$\begin{bmatrix} -0.0108 \\ 0.0032 \end{bmatrix}$	$\begin{bmatrix} 0.3134 \\ 0.2518 \end{bmatrix}$	$\begin{bmatrix} 0.1346 \\ 0.1849 \end{bmatrix}$
1	$\begin{bmatrix} 2.7458 \\ 3.0597 \end{bmatrix}$	$\begin{bmatrix} 4.3476 \\ 4.5327 \end{bmatrix}$	$\begin{bmatrix} -1.1612 \\ 1.7114 \end{bmatrix}$	$\begin{bmatrix} 0.8205 \\ 1.5908 \end{bmatrix}$	$\begin{bmatrix} -0.1777 \\ 0.1430 \end{bmatrix}$	$\begin{bmatrix} 2.0119 \\ 0.5520 \end{bmatrix}$	$\begin{bmatrix} 0.2537 \\ 1.3989 \end{bmatrix}$
2	$\begin{bmatrix} 2.1233 \\ 3.4828 \end{bmatrix}$	$\begin{bmatrix} 4.0702 \\ 4.8180 \end{bmatrix}$	$\begin{bmatrix} -1.1172 \\ 2.8516 \end{bmatrix}$	$\begin{bmatrix} 0.7903 \\ 2.8424 \end{bmatrix}$	$\begin{bmatrix} -0.4516 \\ 0.4030 \end{bmatrix}$	$\begin{bmatrix} 3.8570 \\ 0.4290 \end{bmatrix}$	$\begin{bmatrix} 0.1995 \\ 2.7850 \end{bmatrix}$

Table 11: The minimizers of the loss function for different values of γ, for a 2-layer diagonal linear neural network with $n = 5$, $d = 7$, $T = 1000000$, $\alpha = 0.01$ and the margin $= 10^{-4}$

We observe that for a small weight initialization, the minimizer is indeed a sparse vector, unlike the minimizer for a larger weight initialization. If we increase γ further to $\gamma = 7$ the loss function diverges. In figure 8 the loss function is shown up to iteration 300 for $\gamma = 5$. We see that we jump straight to the minimum, and the coordinates of p oscillate during this jump, after which they converge.

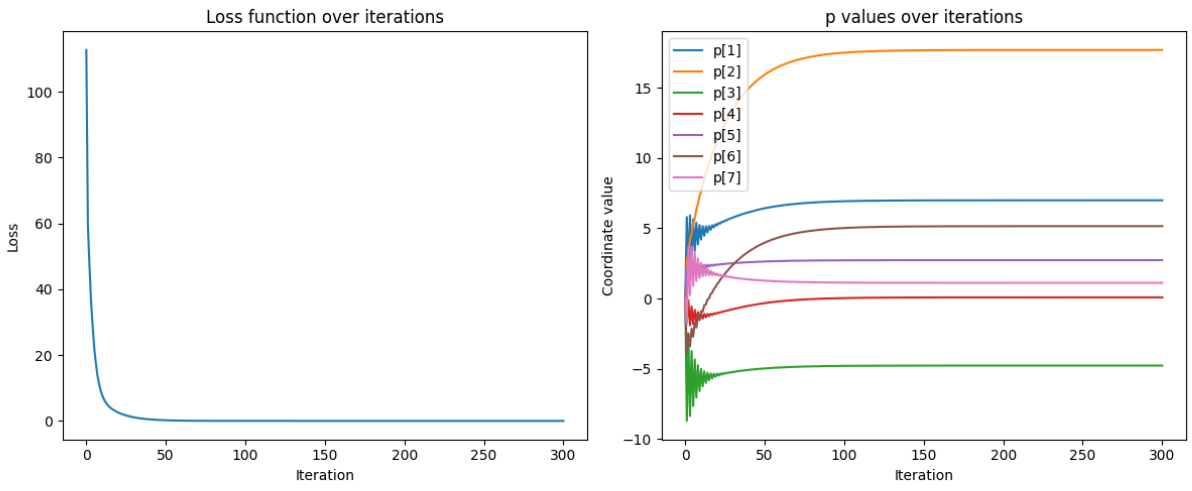


FIGURE 8: 2-layer diagonal neural network n=5 d=7 $\gamma = 5$, Left: the loss function over time, Right: the coordinates of p over time with oscillation during the first few iterations.

6 Discussion & Conclusion

In this paper, we have looked at the saddle-to-saddle behaviour of the gradient flow in shallow diagonal linear neural networks and how the saddle points influence the minimum solution of the loss function found with gradient descent.

From the analytical analysis, we saw that equilibrium points of (gradient flow) are sparse vectors and that for each equilibrium point the stability can be determined by looking at the linearization of (gradient flow) around this point. Furthermore, we characterized the jump directions for several simple diagonal linear neural networks. We checked correctness of our implementation of a shallow diagonal linear neural network using the results of the analytical analysis. This implementation was used to determine the equilibrium points of (gradient flow) for wider neural networks. We saw that the loss function jumps from saddle points to an asymptotically stable minimizer in the case of small weight initialization, and that at each jump, certain coordinates activate or deactivate.

One of the goals of this study was to characterize the direction of the jumps. In the numerical analysis, we studied these directions for two examples of shallow diagonal linear neural networks. We observed that for these examples, the direction of the first two jumps is linear in the expected direction, that is, the direction of the eigenvectors corresponding to the largest positive eigenvalues. However, this cannot be concluded for the other jumps. It would be an interesting topic for further research to characterize these jumps.

Unfortunately, there was not enough time to study the jump times during this research. It would be interesting to characterize the time at which we jump from one saddle to another saddle or minimizer. If we can predict the time we spend at a saddle point, we can 'speed up' the time by the bijection described in section 3.1 in this time interval and switch to the normal learning rate α for the times in between the saddles. Therefore, we recommend this topic for further research.

In this study, we did not look at the saddle-to-saddle behaviour for non-linear neural networks or more complex neural networks, which have a higher width and depth. We recommend these studies for further exploration.

References

[1] Dimitris Kalimeris, Gal Kaplun, Preetum Nakkiran, Benjamin Edelman, Tristan Yang, Boaz Barak, and Haofeng Zhang. SGD on Neural Networks Learns Functions of Increasing Complexity. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., May 2019.

[2] Scott Pesme and Nicolas Flammarion. Saddle-to-Saddle Dynamics in Diagonal Linear Networks, October 2023. arXiv:2304.00488 [cs, math].

[3] Scott Pesme and Nicolas Flammarion. Saddle-to-Saddle Dynamics in Diagonal Linear Networks. *Slides presentation Neurips*, 2023.

[4] Simon J.D. Prince. *Understanding Deep Learning.* The MIT Press, December 2023.

7 Appendix

7.1 Backpropagation

This section uses chapter 7 of [4] to explain the backpropagation algorithm. Applying the gradient descent method to the loss function of a neural network gives rise to the issue of calculating the gradients efficiently. This is difficult since large neural networks have millions of parameters and the derivative needs to be computed for every parameter at every iteration of the training algorithm.

The backpropagation algorithm is an efficient method for computing all the partial derivatives of the loss function for every parameter. It consists of a forward pass and a backward pass. In the forward pass, we compute and store all the intermediate values of the network and the network output. In the backward pass we calculate the derivates of with respect to each parameter, starting at the last layer of the network, reusing previous calculations, and we move toward the input layer.

We consider a neural network with K layers as described in equations (4). We want to compute $\frac{\partial l_i}{\partial \boldsymbol{\beta_k}}$ and $\frac{\partial l_i}{\partial \boldsymbol{\Omega_k}}$, for the parameters $\boldsymbol{\phi}_k = \{\boldsymbol{\beta_k}, \boldsymbol{\Omega_k}\}$ at every layer $k \in \{0, 1, ..., K\}$ and for each data sample i. When we use the activation function $a[\cdot]$, the backpropagation algorithm can be described in the following way.

The forward pass is defined by computing all intermediate values and the network output:

$$\mathbf{f_0} = \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i, \tag{32}$$

$$\mathbf{h_k} = \mathbf{a}[\mathbf{f_{k-1}}], \tag{33}$$

$$\mathbf{f_k} = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k, \tag{34}$$

for $k = 1, 2, ..., K$. Where $\mathbf{f_k}$ denotes the pre-activation of layer k.

For the backward pass, we start with the derivative $\frac{\partial l_i}{\partial \mathbf{f}_K}$ of the loss function l_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

for $k = K, K-1, ..., 1$

$$\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k}, \tag{35}$$

$$\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T, \tag{36}$$

$$\frac{\partial l_i}{\partial \mathbf{f}_{k-1}} = a'[\mathbf{f}_{k-1}] \odot \boldsymbol{\Omega}_k^T \frac{\partial l_i}{\partial \mathbf{f}_k}, \tag{37}$$

where $\odot$ denotes elementwise multiplication, and $a'[\cdot]$ is the derivative of the activation function. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\frac{\partial l_i}{\partial \boldsymbol{\beta}_0} = \frac{\partial l_i}{\partial \mathbf{f}_0}, \tag{38}$$
$$\frac{\partial l_i}{\partial \boldsymbol{\Omega}_0} = \frac{\partial l_i}{\partial \mathbf{f}_0}\mathbf{x_i^T}. \tag{39}$$

We calculate these derivatives for every training example and sum them together to retrieve the gradient for the gradient descent parameter update.

The backpropagation algorithm is efficient, since the most complicated computational step in both the forward and backward pass is the matrix multiplication (by $\boldsymbol{\Omega}$) which only requires additions and multiplications. However, it is not efficient in memory, since the intermediate values in the forward pass must all be stored. This can limit the size of the model we can train.

7.2 Analytical analysis case $d = 2$

For simplicity, we again look at the case $n = 1$ and $d = 2$, and compute the equilibrium points of (gradient flow) analytically to understand how they are computed and how many such points exists. We have that $y \in \mathbb{R}$ and $x, u, v, p \in \mathbb{R}^2$. We assume that x, y are non-zero.

We denote the vectors x, u, v, p as:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} u_1 v_1 \\ u_2 v_2 \end{bmatrix} = u \odot v,$$

such that the loss function is defined as follows:

$$F(u, v) = \frac{1}{2}(u_1 v_1 x_1 + u_2 v_2 x_2 - y)^2. \tag{40}$$

Representing the parameters of the neural network as a vector,

$$\boldsymbol{\phi} = [v_1, v_2, u_1, u_2]^T. \tag{41}$$

ODE (gradient flow) can be written out as:

$$\begin{bmatrix} \frac{dv_1}{dt} \\ \frac{dv_2}{dt} \\ \frac{du_1}{dt} \\ \frac{du_2}{dt} \end{bmatrix} = \begin{bmatrix} -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) u_1 x_1 \\ -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) u_2 x_2 \\ -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) v_1 x_1 \\ -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) v_2 x_2 \end{bmatrix}. \tag{42}$$

The next step is to find the equilibrium points of (42). A trivial equilibrium point is, $\boldsymbol{\phi} = [0, 0, 0, 0]^T$, which corresponds to $p_i = 0, i = 1, 2$.

If we require one parameter, for example u_1, to have non-zero value, all elements of the right-hand side of (42) are zero, except for the first one. Setting this equation to zero, we get,

$$u_1 x_1 y_1 = 0.$$

Since the data points are already given, the only solution is $u_1 = 0$, which is the trivial solution.

Looking at (42) it can be seen that a non-trivial equilibrium point, requires u_1 and v_1, u_2 and v_2, or all parameters to have a non-zero value. Note that these cases correspond to the cases $p_1 \neq 0$, $p_2 \neq 0$, and $p_i \neq 0, i = 1, 2$.

The first case, $u_1, v_1 \neq 0$, reduces the equations to find the zeros of $-\nabla F(u, v)$ to the following:

$$\begin{aligned}&-(u_1v_1x_1 - y)u_1x_1 = 0,\\ &-(u_1v_1x_1 - y)v_1x_1 = 0.\end{aligned}$$

The solution to this system of equations need to satisfy:

$$u_1v_1x_1 = y,$$

which is $u_1 = \frac{y}{v_1x_1}$ (can also be written as $v_1 = \frac{y}{u_1x_1}$).

The equilibrium point corresponding to the second case, $u_2, v_2 \neq 0$, is derived in the same way and is defined as, $u_2 = \frac{y}{v_2x_1}$ (can also be written as $v_2 = \frac{y}{u_2x_1}$).

For the last case, $u_1, u_2, v_1, v_2 \neq 0$, substituting the weights does reduce the equations to find the zeros of$-\nabla F(u, v)$, but the solutions need to satisfy the following equation:

$$u_1v_1x_1 + u_2v_2x_2 = y.$$

The solution can be written in multiple ways, for example $u_1 = \frac{y-u_2v_2x_2}{v_1x_1}$.

Now that we have defined the equilibrium points of ODE (42), we define their stability using the linearization of the ODE around these points. This requires the computation of the Jacobian of $-\nabla F(u, v)$. Afterwards, we substitute the solution in the Jacobian and compute its eigenvalues, to define if the solution is stable or unstable.

The Jacobian of $-\nabla F(u, v)$ is defined as follows:

$$J(u_1, u_2, v_1, v_2) = -\begin{bmatrix} u_1^2x_1^2 & u_1u_2x_1x_2 & u_1v_1x_1^2 + x_1q & u_1v_2x_1x_2 \\ u_1u_2x_1x_2 & u_2^2x_2^2 & u_2v_1x_1x_2 & u_2v_2x_2^2 + x_2q \\ u_1v_1x_1^2 + x_1q & u_2v_1x_1x_2 & v_1^2x_1^2 & v_1v_2x_1x_2 \\ u_1v_2x_1x_2 & u_2v_2x_2^2 + x_2q & v_1v_2x_1x_2 & v_2^2x_2^2 \end{bmatrix}, \quad (43)$$

where $q = u_1v_1x_1 + u_2v_2x_2 - y$.

If we evaluate the Jacobian at the equilibrium points we found in the previous section, we can classify them. The equilibrium point $\boldsymbol{\phi} = [0, 0, 0, 0]^T$, corresponds to

$$J(0, 0, 0, 0) = -\begin{bmatrix} 0 & & -x_1y & 0 \\ 0 & 0 & 0 & -x_2y \\ -x_1y & 0 & 0 & 0 \\ 0 & -x_2y & 0 & 0 \end{bmatrix}, \quad (44)$$

which eigenvalues are $-x_1y, x_1y, -x_2y$ and x_2y, with eigenvectors $[-1, 0, 1, 0]^T$, $[1, 0, 1, 0]^T$, $[0, -1, 0, 1]^T$ and $[0, 1, 0, 1]^T$ respectively. We see that for every data sample the eigenvalues will have different signs, two positive and two negative values. This means that $\boldsymbol{\phi} = [0, 0, 0, 0]^T$ is a saddle point.

In the neighbourhood of $\boldsymbol{\phi} = [0,0,0,0]^T$ the general solution to (gradient flow) can be written as,

$$\boldsymbol{\phi}(t) = C_1 e^{-x_1 y t}\begin{bmatrix}-1\\0\\1\\0\end{bmatrix} + C_2 e^{x_1 y t}\begin{bmatrix}1\\0\\1\\0\end{bmatrix} + C_3 e^{-x_2 y t}\begin{bmatrix}0\\-1\\0\\1\end{bmatrix} + C_4 e^{x_2 y t}\begin{bmatrix}0\\1\\0\\1\end{bmatrix}, \tag{45}$$

where C_1, C_2, C_3 and C_4 are arbitrary constants. Now that we determined this general solution, we can predict the direction of the jump from the saddle point $\boldsymbol{\phi} = [0,0,0,0]^T$. If $|x_1| > |x_2|$ and $x_1 y > 0$ we expect the jump to be linearly in the direction of $[1,0,1,0]^T$. That is, as t increases, we expect to jump to a point for which $v_1 = u_1 \neq 0$ and $v_2 = u_2 = 0$. If $|x_1| > |x_2|$ and $x_1 y < 0$ we expect the jump to be linearly in the direction of $[-1,0,1,0]^T$. That is, as t increases, we expect to jump to a point for which $-v_1 = u_1 \neq 0$ and $v_2 = u_2 = 0$. If $|x_1| < |x_2|$ and $x_2 y > 0$ we expect the jump to be linearly in the direction of $[0,1,0,1]^T$. That is, as t increases, we expect to jump to a point for which $v_2 = u_2 \neq 0$ and $v_1 = u_1 = 0$. If $|x_1| < |x_2|$ and $x_2 y < 0$ we expect the jump to be linearly in the direction of $[0,-1,0,1]^T$. That is, as t increases, we expect to jump to a point for which $-v_2 = u_2 \neq= 0$ and $v_1 = u_1 = 0$.

The Jacobian evaluated at $\boldsymbol{\phi} = [v_1, 0, \frac{y}{v_1 x_1}, 0]^T$, corresponds to

$$J\Big(\frac{y}{v_1 x_1}, 0, v_1, 0\Big) = \begin{bmatrix}-\frac{y^2}{v_1^2} & 0 & x_1 y & 0\\ 0 & 0 & 0 & 0\\ x_1 y & 0 & -v_1^2 x_1^2 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}. \tag{46}$$

The eigenvalues are,

$$\begin{cases} 0, & \text{with multiplicity 3,}\\ -\frac{v_1^4 x_1^2 + y^2}{v_1^2}, & \text{with multiplicity 1.}\end{cases}$$

These eigenvalues are nonpositive. Similar to the previous section, we use Lyapunov's Theorem to define the stability of $\boldsymbol{\phi}$. The loss function $F(u(t), v(t))$ is continuously differentiable and positive definite on $\mathbb{R}^d$ and has its minimum at $\boldsymbol{\phi} = [v_1, 0, \frac{y}{v_1 x_1}, 0]^T$, since $F[\boldsymbol{\phi}] = 0$.

For $u, v \in \mathbb{R}^d$ and $n = 1$,

$$\begin{aligned}\frac{d}{dt}F(u(t), v(t)) &= \sum_{i=1}^d \frac{\partial F}{\partial u_i}\frac{du_i}{dt} + \sum_{i=1}^d \frac{\partial F}{\partial v_i}\frac{dv_i}{dt}\\ &= \sum_{i=1}^d (u_1 v_1 x_1 + u_2 v_2 x_2 - y) v_i x_i \cdot -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) v_i x_i +\\ &\quad \sum_{i=1}^d (u_1 v_1 x_1 + u_2 v_2 x_2 - y) u_i x_i \cdot -(u_1 v_1 x_1 + u_2 v_2 x_2 - y) u_i x_i\\ &= -\sum_{i=1}^d ((u_1 v_1 x_1 + u_2 v_2 x_2 - y) v_i x_i)^2 - \sum_{i=1}^d ((u_1 v_1 x_1 + u_2 v_2 x_2 - y) u_i x_i)^2\\ &< 0,\end{aligned} \tag{47}$$

for $\boldsymbol{\phi} = [v, u]^T$ for which $(u_1v_1x_1 + u_2v_2x_2 - y \neq 0$. So $\frac{d}{dt}F(u(t), v(t))$ is negative definite, which means that $\boldsymbol{\phi} = [v_1, 0, \frac{y}{v_1x_1}, 0]^T$ is an asymptotically stable minimizer of $F(u, v)$.

The Jacobian evaluated at $\boldsymbol{\phi} = [0, v_2, 0, \frac{y}{v_2x_2}]^T$ is,

$$J\Big(0, \frac{y}{v_2x_2}, 0, v_2\Big) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{y^2}{v_2^2} & 0 & x_2y \\ 0 & 0 & 0 & 0 \\ 0 & x_2y & 0 & -v_2^2x_2^2 \end{bmatrix}. \tag{48}$$

The eigenvalues are,

$$\begin{cases} 0, & \text{with multiplicity 3,} \\ -\frac{v_2^4x_2^2+y^2}{v_2^2}, & \text{with multiplicity 1.} \end{cases}$$

Using Lyapunov's Theorem again, denoted in equation (47), we conclude that $\boldsymbol{\phi} = [0, v_2, 0, \frac{y}{v_2x_2}]^T$ is an asymptotically stable minimizer of $F(u, v)$.

The Jacobian at $\boldsymbol{\phi} = [v_1, v_2, \frac{y-u_2v_2x_2}{v_1x_1}, u_2]^T$ is

$$J\Big(\frac{y - u_2v_2x_2}{v_1x_1}, u_2, v_1, v_2\Big) = -\begin{bmatrix} \frac{(y-u_2v_2x_2)^2}{v_1^2} & \frac{u_2x_2(y-u_2v_2x_2)}{v_1} & x_1(y - u_2v_2x_2) & \frac{v_2x_2(y-u_2v_2x_2)}{v_1} \\ \frac{u_2x_2(y-u_2v_2x_2)}{v_1} & u_2^2x_2^2 & u_2v_1x_1x_2 & u_2v_2x_2^2 \\ x_1(y - u_2v_2x_2) & u_2v_1x_1x_2 & v_1^2x_1^2 & v_1v_2x_1x_2 \\ \frac{v_2x_2(y-u_2v_2x_2)}{v_1} & u_2v_2x_2^2 & v_1v_2x_1x_2 & v_2^2x_2^2 \end{bmatrix}. \tag{49}$$

$$\begin{cases} 0, & \text{with multiplicity 3,} \\ -\frac{(u_2v_1x_2)^2+(u_2v_2x_2)^2-2u_2v_2x_2y+v_1^4x_1^2+(v_1v_2x_2)^2+y^2}{v_1^2}, & \text{with multiplicity 1.} \end{cases}$$

Similar to the previous cases, we can conclude, using Lyapunov's theorem, that, $\boldsymbol{\phi} = [v_1, v_2, \frac{y-u_2v_2x_2}{v_1x_1}, u_2]^T$ is also an asymptotically stable minimizer of $F(u, v)$.

7.3 Numerical analysis case d=2

We analyse the case $n = 1$ and $d = 2$ and check our results with the analytical analysis explained in section 7.2. Table 12 shows the values of the hyperparameters and the generated data.

T	α	n	γ	margin	p^*	x	y
1000000	0.001	1	10^{-100}	10^{-4}	$\begin{bmatrix}-1.5454\\-0.0084\end{bmatrix}$	$\begin{bmatrix}1.3316\\0.7153\end{bmatrix}$	-2.0638

Table 12: Values hyperparameters and generated data for a 2-layer diagonal linear neural network n=1 d=2.

Figure 9 shows the resulting plot.

The weights contributing to the plateaus in the loss function are computed, and the results are as follows.

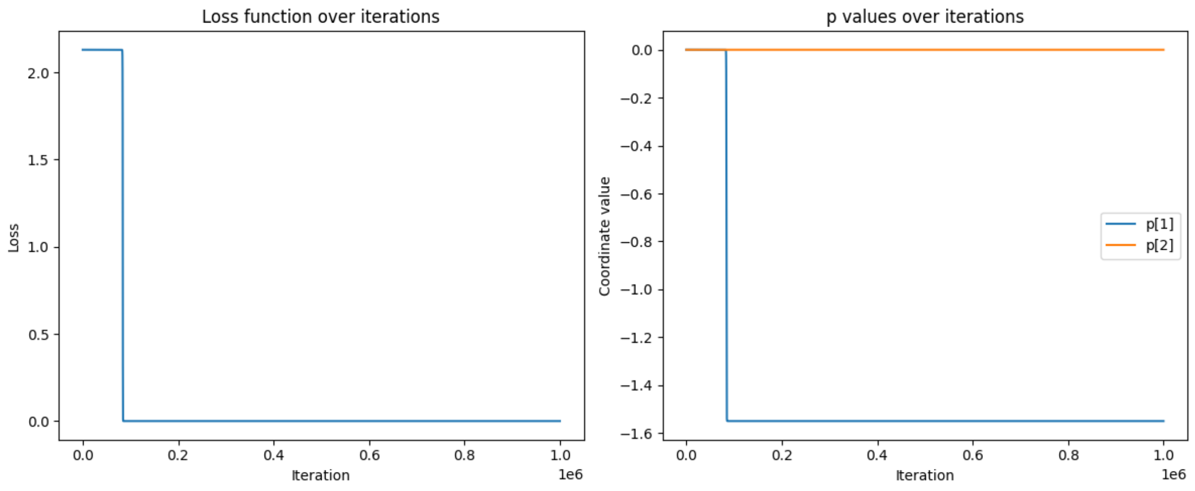


FIGURE 9: 2-layer diagonal neural network n=1 d=2, Left: the loss function over time, Right: the values of the elements of p over time.

The first equilibrium point is $\boldsymbol{\phi} = [0,0,0,0]^T$, with Jacobian

$$\begin{bmatrix} 0 & 0 & -2.7482 & 0 \\ 0 & 0 & 0 & -1.4762 \\ -2.7482 & 0 & 0 & 0 \\ 0 & -1.4762 & 0 & 0 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors,

$$\begin{cases} \text{eigenvalue } -2.7482, & \text{with eigenvector } [-0.7071, 0, -0.7071, 0]^T, \\ \text{eigenvalue } 2.7482, & \text{with eigenvector } [0.7071, 0, -0.7071, 0]^T, \\ \text{eigenvalue } -1.4762, & \text{with eigenvector } [0, 0.7071, 0, 0.7071]^T, \\ \text{eigenvalue } 1.4762, & \text{with eigenvector } [0, -0.7071, 0, -0.7071]^T. \end{cases}$$

The second point is $\boldsymbol{\phi} = [-1.2450, 0, 1.2450, 0]^T$, with Jacobian

$$\begin{bmatrix} -2.7482 & 0 & 2.7482 & 0 \\ 0 & 0 & 0 & 0 \\ 2.7482 & 0 & -2.7482 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

which has eigenvalues and corresponding eigenvectors,

$$\begin{cases} \text{eigenvalue } -5.4963, & \text{with eigenvector } [0.7071, 0, -0.7071, 0]^T, \\ \text{eigenvalue } 0, & \text{with eigenvector } [0.7071, 0, 0.7071, 0]^T, \\ \text{eigenvalue } 0, & \text{with eigenvector } [0, -0.7071, 0, 0.7071]^T, \\ \text{eigenvalue } 0, & \text{with eigenvector } [0, -0.7071, 0, -0.7071]^T. \end{cases}$$

These results agree with our analysis is section 7.2. The equilibrium points are $[0,0,0,0]^T$, which is a saddle point, and $[v_1, 0, \frac{y}{v_1 x_1}, 0]^T$, which is an asymptotically stable minimizer, $p_1^* = u_1 v_1 = -1.5454$. So we can conclude that the loss function moves from a saddle point to a minimizer, and only the first coordinate of p gets activated for these generated data points. Furthermore, it is interesting to note that $|u| = |v|$ for both equilibrium points, which causes the Jacobian to be symmetric, so its eigenvalues are real. The results agree with our characterization of the jump direction in the analytical analysis, 7.2, since for this example $|x_1| > |x_2|$ and $x_1 y = -2.7482$, so the jump from $[0, 0, 0, 0]^T$ should be linearly in the direction $[-1, 0, 1, 0]^T$, which corresponds to the actual jump direction.