

École polytechnique de Louvain

Study of the generalization abilities of affine neural networks used in regression tasks trained with gradient flow

Author: **Pietro GAVAZZI**

Supervisors: **Benoît LEGAT, (Laurent JACQUES)**

Readers: **Estelle MASSART, Christian HAFNER, Giovanni GRAP-PIGLIA, Adeline COLSON**

Academic year 2025–2026

Master [120] in Data Sciences Engineering

Abstract

Multi-layer perceptrons and other neural networks have been used very successfully in many supervised learning tasks. These performances remains largely unexplained to this day.

There exist a large literature who study the properties of linear neural networks, a more simplified form of multi-layer perceptrons, to discover insights on the reasons of the performances of multi-layer perceptrons.

Building on these works, the author selects one paper in this literature and analyses it through the length of a concept called implicit regularisation.

He then introduces affine neural networks, a new type of neural network which is an intermediate step in terms of complexity between linear neural networks and multi-layer perceptrons. He shows how the analysis of the generalisation of the properties of linear neural networks to affine neural networks gives new insights on multi-layer perceptrons that linear neural networks do not give.

Contents

0	Introduction	1
I	State of the art	3
1	Supervised learning theory	4
1.1	Introduction to supervised learning	4
2	Bias-variance	11
2.1	Dummy problem	11
2.2	The bias-variance decomposition	14
2.3	Other framework of understanding in use	18
3	Regularisation	19
3.1	Ridge regularisation	19
3.2	Implicit regularisation	21
4	Neural Networks	26
4.1	Introduction to Neural Networks	26
4.2	Neural networks in supervised learning	30
5	Linear neural networks and implicit regularisation	32
5.1	Modelling of linear neural networks	32
5.2	Gradient flow for linear neural networks	33
5.3	Implicit regularisation for linear neural networks with small initialisation	33
II	Contribution of the author	38
6	Gradient flow of mlps	39
6.1	Matrix form of mlps	39

7	Affine Neural networks	42
7.1	Motivation for affine Neural Networks	42
7.2	Modelling affine neural networks	43
7.3	Linear and affine cases for linear regression and linear neural networks	46
7.4	Numerical test on the alignment property in affine neural networks	46
7.5	Conclusion	48
A	Formal Definitions	50
A.1	Risk of a function and Hypothesis Space	50
A.2	Supervised learning task	51
A.3	Risk of supervised learning algorithm	51
B	Bias Variance Decomposition	53
C	Bias of the training error	55
C.1	Description of our problem	55
C.2	Distribution shift in best of two values for parameter θ	56
D	Generalisation framework	60
D.1	Theory behind generalisation	60
D.2	Conclusion	61
E	Gradient flow OLS	62
E.1	Statement of the problem	62
E.2	Decomposition into column space and kernel	63
E.3	Gradient flow in the subspaces	63
E.4	Solving the gradient flow in the parallel component	64
E.5	Properties of matrix exponential	64
E.6	Computing the integral term	66
E.7	Final solution	66
F	Derivation of the gradient of linear Neural Networks	68
F.1	Proof part 1: useful results	68
F.2	Proof part 2: derivation of the formula of the gradient	69
G	Derivation of the gradient of affine Neural Networks	71
G.1	Proof part 1: useful results	71
G.2	Proof part 2: derivation of the formula of the gradient	72
H	Proof of lemma 1 for linear Neural Networks	74
I	Lemma 1 disproven for affine Neural Networks	75

List of Figures

2.1	Regression polynomial degree 1 on two different datasets	13
2.2	Regression polynomial degree 12 on two different datasets	14
2.3	Bias-variance decomposition of the Risk	16
2.4	Regression polynomial degree 1 on two different datasets	17
3.1	Ridge regression polynomial degree 12 on two different datasets . .	21
3.2	Big picture	23
3.3	Linear regression optimisation path	24
4.1	Symbolical representation of a perceptron	27
4.2	Symbolical representation of a feedforward neural network with one hidden layer.	28
4.3	Symbolical representation of a feedforward neural network with two hidden layers.	29
6.1	ReLU linear network network	41
C.1	Training error 5000 training datasets of 5 examples	56
C.2	Distribution shift for the value of the training error	57
C.3	Training error 5000 training datasets of 15 examples	58

Listings

1.1	Datasets generation	6
1.2	Training	6
1.3	Test	8

Chapter 0

Introduction

If we assume that we have some input X and some target Y , 2 random variables following a joint probability distribution $\mathcal{D}_{(X,Y)}$, supervised learning is the art of finding the best possible function f that predicts the target as well as we can using the input $f(X) \approx Y$. For this we design a supervised learning procedure that finds this function using a dataset with a limited number of realisations of some inputs and their corresponding targets.

Supervised learning is a very experimental art: because Hoeffding's inequality guarantee the performance of our result the great majority of the practitioners use a heuristical understanding to come up with new procedures. While we can have guarantees that they perform well on given tasks, we often have a lack of understanding of the inner workings and the reasons they perform.

One commonly used heuristic is the bias-variance trade-of heuristic, who comes from the bias-variance decomposition Geman et al. [1992].

The first thing this decomposition shows is that any specific supervised learning problem has a best performing function called the conditional expectation function : $\mathbb{E}[Y|X]$.

Then it also illustrates that a given problem the performance of a supervised learning procedure can be decomposed into 2 term.

The first term is called the bias: it measures how different the conditional expectation function is from the function the supervised learning procedure find in expectation over all realisation of the training dataset.

The second is called the variance, it measures how the function the supervised learning procedure finds vary with the different realisation of the training dataset.

This lead to the bias-variance trade-off heuristic: the belief that if we increase pool of hypothesis functions the supervised learning procedure has access to (the hypothesis space), then the bias term would decrease but the variance term would increase. This is because while in expectation we could find a function close to the conditional expectation function, the choice of function would vary with different

realisation of the training dataset.

This traditional understanding was dented by the advent of neural networks.

Neural networks are parametric model loosely inspired by the working of the human brain. Their use in supervised learning has had surprising results that challenged the bias-variance trade-off heuristic, especially the double descent phenomenon Belkin et al. [2019]: this results seemed to indicate that when the supervised learning had access to a larger amount of functions, the variance would decrease.

The bias variance trade-off heuristic can still be saved by the concept of "implicit regularisation" introduced in Gunasekar et al. [2017]. Supervised learning procedures do not pick all the function in the hypothesis space: they generally perform gradient descent on parametric functions, and the choice of the parametrisation organise the functions of the hypothesis space in such a way that the supervised learning algorithm will choose in a reduced set of functions. The result of Belkin et al. [2019] could then perhaps be explained by some implicit regularisation?

These recent discoveries motivated the studies of optimisation paths of linear neural networks, a more simplified form of neural network. The contribution of the author lies in the observation that we could conceive a middle step in complexity between linear neural networks and more complex neural networks called affine neural networks. The study of affine neural networks gives us better insight on the optimisation path of more general types of linear neural networks.

As a proof of concept, the author analysed the results of a paper studying linear neural networks Marion and Chizat [2024]. He interpreted these results with the concept of implicit regularisation, then partially generalised them for affine neural networks. The author hopes this can motivate more studies on affine neural networks.

Part I

State of the art

Chapter 1

Supervised learning theory

In this chapter we will give an introduction to supervised learning.

We will see that if we assume that we have some input X and some target Y , 2 random variables following a joint probability distribution $\mathcal{D}_{(X,Y)}$, supervised learning is the art of finding the best possible function f that predicts the target as well as we can using the input $f(X) \approx Y$.

This is formalised as the problem of minimising the expected loss over the joint distribution, what we will call risk minimisation.

In supervised learning, we only have access to a set of realisations of inputs and targets, these realisations are called data points or data.

We will see that a supervised learning algorithm finds a function by fitting a parametric model minimising the empirical risk on the training set, called training error. This is an approximation of the the real risk computed with a subset of the data points.

We will then give some guarantee on the precision of the training error as approximation of the risk by using Hoeffding's inequality, we will then motivate the use of the testing error to approximate the risk of the function found by the supervised learning algorithm.

We will finally give the mathematical definition of the risk that will be used to quantify the performances of supervised learning algorithms.

This will be an informal introduction to SL. If the reader is interested in a more formal development, he can find it in Appendix A.

1.1 Introduction to supervised learning

Definition of the risk of a function

Supervised Learning is a discipline whose origins can be traced back very far even if it was not called that way back then.

In a well known machine learning textbook Hardt and Recht [2022] it is told that the British astronomer Edmond Halley calculated in 1692 the price of annuities with church registries. This can be considered a SL problem in a modern context.

While this problem is very interesting for historical reasons, it has some technical details that are not suited for a toy problem. Hence, to develop SL we will use a more simplified problem inspired by this first one.

Suppose you want to create a machine that predicts as well as possible the maximum age a person taken at random in the population is expected to live. If the only thing we know about the person is their current age, how will that be done in practice?

First, let's explain what we mean by "the maximum age a person taken at random in the population is expected to live". We can define a tuple for every person $(X \in \mathbb{R}_+, Y \in \mathbb{R}_+)$ where X is the current age of the person and Y is the age of this person when he dies.

The X are called the "inputs" and the Y are called the "targets". When X is a vector, each individual component is called a "feature".

If we take a person at random in the population, it means that we sample our tuple from a distribution: $(X, Y) \sim \mathcal{D}_{(X,Y)}$.

Now, what we want to do can be explained as predicting Y given X : this means finding a function $f : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ so that $f(X) \approx Y$.

Finally, to specify what we mean by $f(X) \approx Y$, we can define a loss function $\mathcal{L} : (\mathbb{R}_+, \mathbb{R}_+) \rightarrow \mathbb{R}_+$ so that $f(X) \approx Y \implies \mathcal{L}(f(X), Y) \approx 0$ and if they are very different $\mathcal{L}(f(X), Y) \gg 0$. Such a function $\mathcal{L}$ is called a "Loss Function".

The aim will thus be to reduce the expected loss, also called the "Risk" $\mathcal{R}$ which is defined in Equation 1.1.

$$\min_f \mathcal{R}(f) := \min_f \mathbb{E}_{(X,Y) \sim \mathcal{D}_{(X,Y)}} \mathcal{L}(f(X), Y) \quad (1.1)$$

There exists many loss functions, but during all the duration of the master thesis, we will use the mean square difference loss $\mathcal{L}(f(X), Y) = (f(X) - Y)^2$. The risk is defined in Equation 1.2.

$$\min_f \mathcal{R}(f) := \min_f \mathbb{E}_{(X,Y) \sim \mathcal{D}_{(X,Y)}} (f(X) - Y)^2 \quad (1.2)$$

The aim of SL is thus to find the function that minimises the Risk. We have a dataset of examples to achieve this aim, which are realisations of the jointly distributed random variable $(X, Y) \sim \mathcal{D}_{(X,Y)}$, we can call these realisations "data points" or "data".

SL is the procedure that learns the function minimising the risk given a dataset of data.

The SL procedure consists generally of 3 steps: datasets generation, training and testing. We will implement these steps in the python programming language and comment the code.

1.1.1 Datasets generation step

```
# ----- data generation -----

X, y = generate_data(nb_data= 300)
ages = np.arange(len(life_exp))

# ----- train / test split -----
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

Listing 1.1: Datasets generation

With the code presented in Listing 1.1, we generate an array $\mathbf{X}$ which contains the current ages of many people taken at random in the population, and the array $\mathbf{y}$ containing the corresponding ages of death.

In this case we can generate as many data points as we want, in other problems however, we are limited to a finite number. We will assume for this toy problem that we are limited to 300 data points.

The purpose of this datasets generation step is to divide these data points into two sets, a training set $\mathcal{S}_N = \{(X_i, Y_i)\}_{i=1}^N$ containing N data points that will only be used during training and a test set $\mathcal{T}_M = \{(X_i, Y_i)\}_{i=1}^M$ containing M data points that will only be used during testing.

The aim of separating the data as such will be explained when discussing the Test step in Section 1.1.3.

1.1.2 Training step

```
model = LinearRegression()
model.fit(X_train, y_train)
mean_train_error = np.mean((y_train - model.predict(X_train)
    )**2)
```

Listing 1.2: Training

During the training step, also called optimisation step, we define a parametric model with `model = LinearRegression()`, we then fit the chosen model to the set of data points in the training set with `model.fit(X_train, y_train)`.

What is called a "model" or "parametric model" in this case is a "parametric function" $f : (\mathbb{R}_+, \Theta) \rightarrow \mathbb{R}_+$. To distinguish the "input" from the "parameter" we write $f_\theta(X) := f(X, \theta)$ where $\theta \in \Theta$ is the parameter and $X \in \mathbb{R}_+$ is the input.

"Fitting the model" or "fitting the parameters" refers to an optimisation process where we train our model to minimize what is called the "empirical risk" calculated with our training set, also called "training error" as defined in Equation 1.3.

$$\hat{\mathcal{R}}_{S_N}(f_\theta) := \frac{1}{N} \sum_{i=1}^N \mathcal{L}(Y_i, f_\theta(X_i)) \quad (1.3)$$

The training error is an approximation of the true risk $\hat{\mathcal{R}}_{S_N}(f_\theta) \approx \mathcal{R}(f_\theta)$. We then find the best parameter $\hat{\theta}$ for our model by using a solver on the empirical risk as defined in Equation 1.4, and this gives us the function found by the supervised learning algorithm $f_{\hat{\theta}}$.

$$\hat{\theta} = \text{Optimiser} \left(\underset{\theta}{\operatorname{argmin}} \hat{\mathcal{R}}_{S_N}(f_\theta) \right) = \text{Optimiser} \left(\underset{\theta}{\operatorname{argmin}} \sum_{i=1}^N \mathcal{L}(Y_i, f_\theta(X_i)) \right) \quad (1.4)$$

The output of the optimiser is not, in all generality, the minimum of the empirical risk. But in particular, for our toy problem, we will fit a linear model on a regression problem with mean square loss, and will find a global minimum $(\hat{\theta}_0, \hat{\theta}_1)$ as defined in Equation 1.5.

$$(\hat{\theta}_0, \hat{\theta}_1) = \underset{\theta_0, \theta_1 \in \mathbb{R}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N (Y_i - \theta_0 - X_i \theta_1)^2. \quad (1.5)$$

Is the training error a good approximation of the risk? If we assume losses are bounded between 0 and a constant upper bound C , we can use Hoeffding's inequalities to bound the difference between the Risk and the empirical risk with arbitrary probability, like shown in Equation 1.6.

$$\begin{aligned} \Pr \left(\mathbb{E}_{S_N}[\hat{\mathcal{R}}_{S_N}(f_\theta)] - \hat{\mathcal{R}}_{S_N}(f_\theta) \geq \delta \right) &\leq e^{-\frac{2N\delta^2}{C^2}} \\ \Pr \left(\mathbb{E}_{S_N}[\hat{\mathcal{R}}_{S_N}(f_\theta)] - \hat{\mathcal{R}}_{S_N}(f_\theta) \leq -\delta \right) &\leq e^{-\frac{2N\delta^2}{C^2}} \end{aligned} \quad (1.6)$$

The proof of equation 1.6 can be found in Duchi

The main result of Hoeffding's Inequality is that the difference between the risk and the empirical risk decreases exponentially with respect to the number of samples in the training dataset N .

An important thing to notice is that if θ is chosen independently from S_N and in view of the fact that the training error is a sum of independent identically distributed losses $\mathcal{L}(X_i, Y_i)$, we can guarantee $\mathbb{E}_{S_N}[\hat{\mathcal{R}}_{S_N}(f_\theta)] = \mathcal{R}(f_\theta)$.

1.1.3 Test step

```
mean_test_error = np.mean((y_test - model.predict(X_test))
**2)
```

Listing 1.3: Test

To analyse the performance of our trained model $f_{\hat{\theta}}$, we calculate the empirical risk on our test dataset and not our training dataset, why?

Let's analyse the training error on the trained parameter as defined in Equation 1.7.

$$\hat{\mathcal{R}}_{S_N}(f_{\hat{\theta}}) := \frac{1}{N} \sum_{i=1}^N \mathcal{L}(Y_i, f_{\hat{\theta}}(X_i)) \quad (1.7)$$

The difference with Equation 1.3 is that f_{θ} becomes $f_{\hat{\theta}}$.

When calculating the training error for $\hat{\theta}$ we do not consider every possible realisation of the training dataset but only those realisations where $\hat{\theta}$ is the result of our optimiser. This induces a bias and leads to the fact that:

$$\mathbb{E}_{S_N}[\hat{\mathcal{R}}_{S_N}(f_{\hat{\theta}})] \neq \mathcal{R}(f_{\hat{\theta}}).$$

This phenomena is closely related to the "regression to the mean" explained in Barnett et al. [2005].

A more in depth explanation on why the training error is not a good approximation of the risk on the trained parameter can be found in Appendix C.

To have a good estimator of the risk we use the empirical risk calculated with our test set, also called "test error" as defined in Equation 1.8.

$$\hat{\mathcal{R}}_{\mathcal{T}_M}(f_{\hat{\theta}}) := \frac{1}{M} \sum_{i=1}^M \mathcal{L}(Y_i, f_{\hat{\theta}}(X_i)) \quad (1.8)$$

Because we did not use the test set to find our best parameter, the expectation of the test error is equal to the risk.

1.1.4 Some thoughts of the author on the essence of supervised learning

The aim of a SL procedure is to learn a function that minimizes the risk. If we do not change the dataset generation step and the testing step, the test error will always be a good measure to compare different adaptations of the training step.

This makes SL a very experimental art by essence: you can change the parametric model, the optimiser, the training algorithm's objective function, compare the performances, and find the best performing variation.

It is not uncommon to try a bunch of variations guided only by heuristics, and simply pick the one that works best.

Why one model performed better than the other one? Does it even matter to understand why it works if you can prove it performs well regardless of our understanding?

In practice, many papers treating SL are dedicated to come up with new algorithms with heuristics, testing the performances and then publish them if they are better than the state-of-the-art model on a given task.

There are fewer papers that try to develop formal approaches to understand the performances of SL algorithms. In the next chapter, we will try to understand the performances of SL algorithms via what will be called the "bias-variance tradeoff heuristics", which is one attempt to develop a theoretical understanding of supervised learning algorithms.

1.1.5 Definition of risk for a SL algorithm

Until now, we defined the risk in Equation 1.1 as a function of the trained model $\mathcal{R}(f_{\hat{\theta}})$. The problem with this definition is that we cannot evaluate the performance of a SL algorithm with it.

The model trained by our SL algorithm is determined by the training dataset S_N , so if we change the training dataset the value of the risk changes too.

We would like to evaluate a SL algorithm regardless of the specific realisation of the training dataset it is trained on.

Therefore, let us first define the risk of a SL algorithm, but first what is a SL algorithm formally ?

A SL algorithm learns a function $f : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ with a training dataset $S_N \in (\mathbb{R}_+, \mathbb{R}_+)^N$ (in Listing 1.2 this corresponds to the execution of `model.fit()`).

Let $\mathcal{H}$ be the set of all functions $\mathbb{R}_+ \rightarrow \mathbb{R}_+$, a SL algorithm can be formally defined as a functional $\mathcal{F} : (\mathbb{R}_+, \mathbb{R}_+)^N \rightarrow \mathcal{H}$ that associates at each possible realisation of the dataset $S_N = \{(X_i, Y_i)\}_{i=1}^N \in (\mathbb{R}_+, \mathbb{R}_+)^N$ a specific function $f \in \mathcal{H}$.

For sake of clarity if $\mathcal{F}(S_N) = f$, let's define $\mathcal{F}_{S_N}(X) := f(X)$.

Armed with these notations, we can now define the risk of a SL algorithm $\mathcal{R}(\mathcal{F})$ in Equation 1.9 where S_N is our training dataset.

$$\mathbb{E}_{X,Y}[\mathcal{L}(Y, \mathcal{F}_{S_N}(X))]. \quad (1.9)$$

But now we need to take the expectation of the risk on all possible realisations of the training dataset S_N , so the final formula for the risk is defined in Equation 1.10.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_{S_N}[\mathbb{E}_{X,Y}[\mathcal{L}(Y, \mathcal{F}_{S_N}(X))]]. \quad (1.10)$$

This definition of the risk works for all losses, but because we will only use mean square loss we will keep the notation $\mathcal{R}(\mathcal{F})$ for the risk with this particular loss.

Chapter 2

Bias-variance

In this chapter we will study the case of minimum square loss to show that the risk of a supervised learning algorithm can be decomposed into 3 terms, in which only 2 are dependent on the supervised learning algorithm: the bias term and the variance term. We will also discover that, for a given problem, the best function a supervised learning algorithm can learn is a function called the "conditional expectation function" (CEF): $f(X) = \mathbb{E}[Y|X]$, which is the function that gives us 0 bias.

We will then study empirically the bias and variance with a polynomial regression on the toy problem of our first chapter, and discover that increasing the degree of the polynomial lowers the bias but increases the variance.

We will explain it with the concept of expressive power of polynomial models, a loose way to describe how qualitatively wide the set of functions in the hypothesis space is.

This will allow us to motivate a commonly used heuristic in supervised learning called the bias-variance trade-off heuristic: which is that widening the hypothesis space by having more complex parametric models increases the variance because the optimiser has the choice between a larger pool of functions but lowers the bias because the optimiser will be able find in expectation a better approximation of the CEF with an extended hypothesis space, and vice versa if we reduce it.

2.1 Dummy problem

While developing the bias-variance decomposition, we will use polynomial parametric models to illustrate the different concepts.

2.1.1 Polynomial models

We will define polynomial models of degree d as parametric functions that scale every monomial or combination of monomials of the features of degree d proportionally to a parameter and then output the sum, as defined in Equation 2.1.

$$\text{Poly}_\theta^d(X) = X^d \cdot \theta \quad (2.1)$$

And where θ is the vector of parameter and where X^d is an augmented vector of features containing the different monomials and combination of monomials of the features. For example if $X = (x_1, x_2)$ and $d = 2$ this vector would be:

$$X^{d=2} = [1 \quad x_1 \quad x_2 \quad x_1x_2 \quad x_1^2 \quad x_2^2] \quad (2.2)$$

For every realisation of the training dataset $\mathcal{S}_N$, we can define the augmented matrix of inputs $X_{\mathcal{S}_N}^d$ and the augmented vectors of target $Y_{\mathcal{S}_N}$.

$$X_{\mathcal{S}_N}^d := \begin{bmatrix} X_1^d \\ \vdots \\ X_i^d \\ \vdots \\ X_N^d \end{bmatrix} \quad Y_{\mathcal{S}_N} := \begin{bmatrix} Y_1 \\ \vdots \\ Y_i \\ \vdots \\ Y_N \end{bmatrix} \quad (X_i, Y_i) \in \mathcal{S}_N \quad (2.3)$$

To fit polynomial models, we will use the pseudo-inverse method to find the least squares solution of our training error minimisation problem.

The trained parameter $\hat{\theta}^d$ are found using the formula given in equation 2.4

$$\hat{\theta} := X_{\mathcal{S}_N}^\dagger Y_{\mathcal{S}_N} \quad (2.4)$$

Where $X_{\mathcal{S}_N}^\dagger$ is the Moore–Penrose pseudo-inverse of the augmented matrix of inputs $X_{\mathcal{S}_N}^d$.

2.1.2 Illustration of bias and variance of polynomial regression

With the bias-variance decomposition, bias and variance can be given formal definitions, and the risk can be decomposed as a sum of the bias term, the variance term and a term that does not depend on the model.

But before defining them formally, two regressions will be shown to give the intuition behind the decomposition.

We will see in the next section that the function which minimises the risk for a regression problem with squared difference loss is the Conditional Expectation

Function (CEF) : $f(X) = \mathbb{E}[Y|X]$. In our toy problem, the CEF is the life expectancy at age X .

Let's analyse the trained models of degree 1 and 12 on 2 different realisations of our training dataset and then compare the results with the life expectancy.

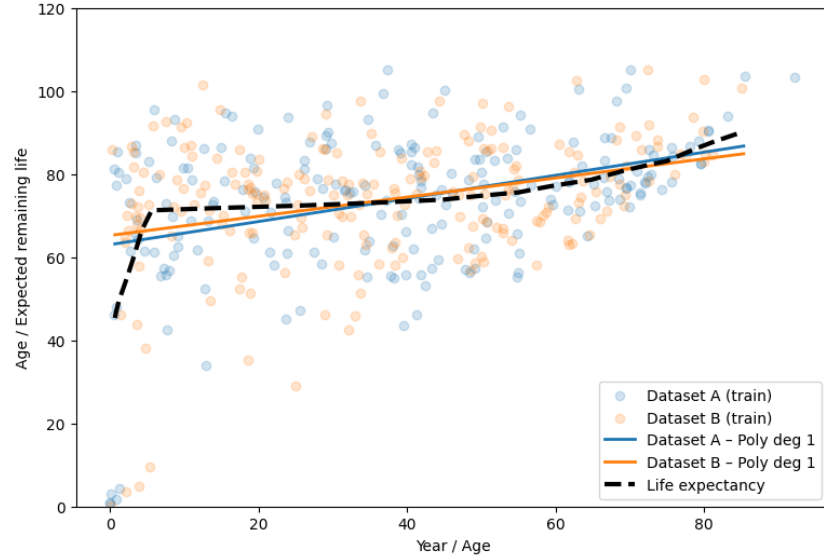


Figure 2.1: Regression polynomial degree 1 on two different datasets

As you see in Figure 2.1 both models find practically the same straight line, so the variance of the model when changing the dataset is very low.

But a straight line cannot approximate the curve of the life expectancy very well, so whatever we do the bias of the model will be high.

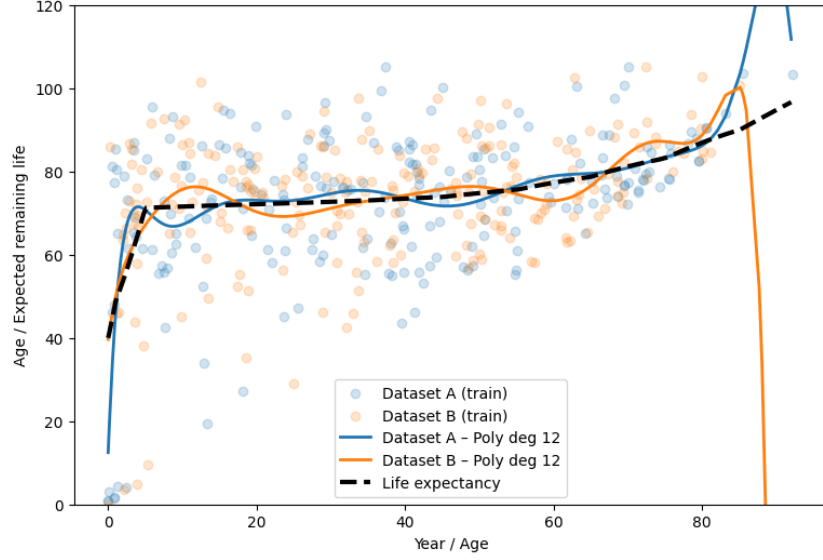


Figure 2.2: Regression polynomial degree 12 on two different datasets

Now let's compare with a higher degree model.

As you can see in Figure 2.2 the prediction of both models vary more from one another, especially in regions where there is less data like when $X \geq 80$. We say that the variance of the model when changing the dataset is very high.

But if we take the best possible polynomial of degree 12, it can better approximate the CEF. And we will see that for polynomial regression we will always converge to the best possible approximation of degree 12 in expectation over all realisations of the training dataset. So the bias will be lower than with a polynomial of degree 1.

2.2 The bias-variance decomposition

2.2.1 Theory behind bias-variance decomposition

We will reproduce the bias-variance decomposition that can be found in Geman et al. [1992]. We will show the bias-variance decomposition for squared loss. The proof can be found in Annex B.

The risk as defined in Equation 1.10 can be decomposed as shown in Equation 2.5.

$$\mathcal{R}(\mathcal{F}) = \text{Bias}(\mathcal{F}) + \text{Variance}(\mathcal{F}) + \text{IE} \quad (2.5)$$

The Irreducible Error (IE), as defined in Equation 2.6, is the expected difference

between Y and the CEF $\mathbb{E}_Y[Y|X]$. This CEF is dependent on the specific problem, and it is given by our observation of some phenomenon in the world.

The IE does not depend on our SL algorithm $\mathcal{F}$ and thus cannot be reduced.

$$\text{IE} = \mathbb{E}_X[\mathbb{E}_Y[(Y - \mathbb{E}_Y[Y|X])^2]] \quad (2.6)$$

The bias term, as defined in Equation 2.7, represents the difference between the expected model over all training datasets and the CEF.

$$\text{Bias}(F) = \mathbb{E}_X[(\mathcal{F}_{S_N}^-(X) - \mathbb{E}_Y[Y|X])^2] = \mathbb{E}_X[(\mathbb{E}_{S_N}[\mathcal{F}_{S_N}(X)] - \mathbb{E}_Y[Y|X])^2] \quad (2.7)$$

The Variance term, as defined in Equation 2.8, represents the expected difference between the output of the model trained with a particular training dataset and the output of the expected trained model on all realisation of the training dataset. This variance term comes from the variance of the training dataset: if there was only one possible realisation of the training dataset this term would be equal to 0.

$$\text{Variance}(\mathcal{F}) = \mathbb{E}_X[\mathbb{E}_{S_N}[(\mathcal{F}_{S_N}(X) - \mathcal{F}_{S_N}^-(X))^2]] = \mathbb{E}_X[\mathbb{E}_{S_N}[(\mathcal{F}_{S_N}(X) - \mathbb{E}_{S_N}[\mathcal{F}_{S_N}(X)])^2]] \quad (2.8)$$

From this decomposition we can deduce that the best possible SL algorithm learns the CEF $\mathcal{F}_{S_N}(X) = \mathbb{E}_Y[Y|X]$ whichever the value of the training dataset.

Indeed if this is the case the bias and the variance are equal to 0 as you can see in Equation 2.9.

$$\begin{aligned} \text{Bias}(\mathbb{E}_Y[Y|X]) &= \mathbb{E}_X[(\mathbb{E}_{S_N}[\mathbb{E}_Y[Y|X]] - \mathbb{E}_Y[Y|X])^2] \\ &= \mathbb{E}_X[(\mathbb{E}_Y[Y|X] - \mathbb{E}_Y[Y|X])^2] = \mathbb{E}_X[0] = 0 \\ \text{Variance}(\mathbb{E}_Y[Y|X]) &= \mathbb{E}_X[\mathbb{E}_{S_N}[(\mathbb{E}_Y[Y|X] - \mathbb{E}_{S_N}[\mathbb{E}_Y[Y|X]])^2]] \\ &= \mathbb{E}_X[\mathbb{E}_{S_N}[(\mathbb{E}_Y[Y|X] - \mathbb{E}_Y[Y|X])^2]] = \mathbb{E}_X[\mathbb{E}_{S_N}[0^2]] = 0 \end{aligned} \quad (2.9)$$

2.2.2 The bias-variance trade-off heuristic

Empirical observations

With these formal definitions of bias and variance, we can make numerical tests and compute the empirical estimation of bias and variance for different polynomial models.

What we empirically observe is a U-shaped curve for the risk as a function of the degree of the polynomial (as illustrated in Figure 2.3).

When decomposing the risk into bias and variance, on polynomials of lower degree, the bias term dominates while the variance is negligible.

As the degree increases, the bias decreases faster than the variance increases, so the total risk decreases.

Then it comes a moment, after the optimal degree, where the variance increases faster than the bias decreases, so the risk increases.

This is called the bias-variance trade-off, because empirically we observe that the choice of the degree for polynomial regression is subject to the arbitration of a trade-off between its impact on the bias and its impact on the variance.

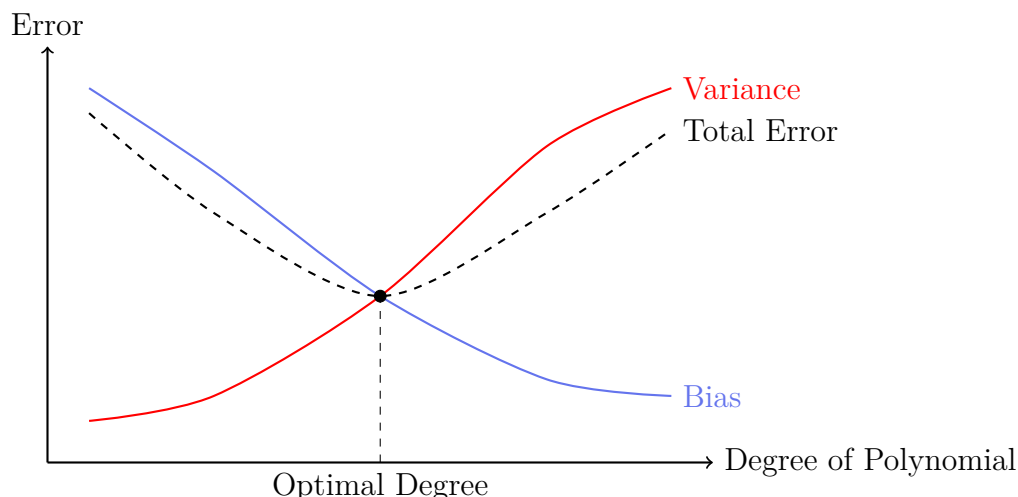


Figure 2.3: Bias-variance decomposition of the Risk

Motivations and descriptions of the bias-variance trade-off heuristic

Although the bias-variance trade-off being a well known phenomenon in polynomial interpolation and other supervised learning procedures, the author was surprised not to find any formal description of the phenomenon.

A well known machine learning textbook, Hastie et al. [2009], attributes the increase in variance and the decrease in bias to what he calls "model complexity". Model complexity is never formally defined but from the examples given we can describe it as some measure of how qualitatively wide the set of functions is. For the rest of the document we will speak of "expressive power" of the model.

Let's call $\mathcal{H}_d$ the set of all functions that can be represented with polynomials of degree d (the hypothesis space of the model).

Then we have $\mathcal{H}_{d-1} \subset \mathcal{H}_d$ because polynomials of degree $d - 1$ are also by definition polynomials of degree d .

Therefore, the number of functions a polynomial model of lower degree can represent is lower than the number of functions a polynomial model of higher degree can represent.

Because polynomials of degree d can represent a wider variety of functions, we say they have a bigger "expressive power".

With this knowledge, we can explain the evolution of the bias term.

For parametric models with higher expressive power, because we have a choice between more functions, we can reasonably expect that the optimiser will in expectation find a better approximation of the CEF, so the bias will decrease for a large variety of SL problems.

In the case of polynomial regression, theorem 2 of White [1980] even proves in most cases we will find in the expectation over all realisations of the training datasets the polynomial that best approximates the CEF.

The explanation for the evolution of the variance term remains more elusive.

Although there is no rigorous proof even for polynomial regression, the increase in variance is attributed to the optimiser having the choice between a larger pool of functions in the hypothesis space (or in other words, the fact that the expressive power of the parameter model is higher).

Because the optimiser has more choice, it will vary its choice of function to better fit the specific realisation of the training dataset it is trained on, and this will lead to widely different predictions with different realisations of the training dataset. This is especially true in regions with low data density or in regions with high target variance.

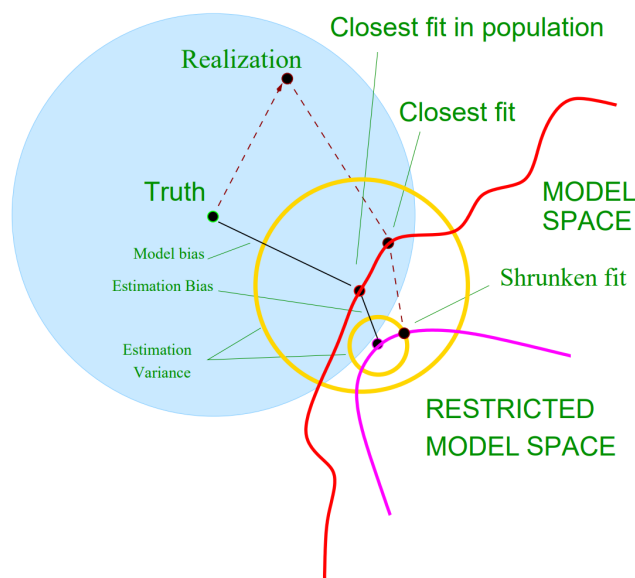


Figure 2.4: Regression polynomial degree 1 on two different datasets

This situation is illustrated in the textbook Hastie et al. [2009] by figure 2.4.

What he calls the "truth" is the conditional expectation function.

But we do not have access to the CEF, only to one realisation of the training dataset: that is represented by the dot he called "realization", we can also loosely interpret it as the function that perfectly interpolates the training dataset and has 0 training error.

If we take the particular case of our example, the "model space" is $\mathcal{H}_d$ and the "restricted model space" is $\mathcal{H}_{d-1}$. For each model space we have a closest fit to the CEF (what is called "closest fit in population"), the distance of this point to the truth is a representation of the amount of bias.

But depending on the particular realisation of the training dataset the "closest fit" will change.

The expected distance over all realisations of the training dataset between the closest fit and the closest fit in population is the variance, it is represented by a yellow circle whose surface is proportional to the amount of variance.

If the increase in amount of bias when going from $\mathcal{H}_d$ to $\mathcal{H}_{d-1}$ is smaller than the decrease in amount of variance, then $\mathcal{H}_{d-1}$ is a better parameter model for this problem and vice-versa if the inverse is true.

2.3 Other framework of understanding in use

Although the author understands supervised learning through the framework of the bias-variance decomposition, the majority of researchers seem to have studied supervised learning through the concept of "generalisation".

In this framework we try to bound for a particular training algorithm the value of the risk given the value of the training error, as shown in equation 2.10.

$$\mathcal{R}(\mathcal{F}_{S_N}) \leq \hat{\mathcal{R}}_{S_N}(\mathcal{F}_{S_N}) + \mathcal{O}(C_{\mathcal{F}}) \quad \text{with probability } 1 - \epsilon \quad (2.10)$$

$\mathcal{O}$ refers to the "big o operator" used to describe the order of approximation, the $C_{\mathcal{F}}$ is called the capacity and is a value dependent on the supervised learning algorithm. From the author's understanding it is often dependent on the expressive power of the hypothesis space: the more expressive the looser the bound.

The study of generalisation is motivated by the observation that it is relatively easy to obtain a low training error, but it is much more difficult to guarantee the training error of the fitted model is a good approximation of the risk.

An introduction to this framework is provided in Appendix D.

Chapter 3

Regularisation

In the previous chapter we introduced the bias variance trade-off heuristic.

In this chapter we will analyse ridge regression as an example of regularisation technique. Regularisation techniques are direct applications of this heuristic: they can be seen as clever ways to reduce the size of the hypothesis space so that the reduction in variance is much higher than the increase in bias.

In the previous chapter we stated that extending the hypothesis space increased the variance because "the optimiser has the choice between a larger pool of functions". In this chapter, we will question this statement and we will see that over-parametrised regression trained with gradient flow tends to implicitly reduce the hypothesis space. This reduction is proportional on the rank of the matrix of inputs. This will be the first example of what we will call implicit regularisation.

Most ideas discussed in this section comes from Srebro [2022], most precisely from the 2 first two parts of this lecture done by Professor Nathan Srebro. The author took large freedoms to present his understanding of the subject.

3.1 Ridge regularisation

We have seen in the previous sections that polynomials with higher degree tend to have a higher variance.

This increase in variance is attributed to the fact that polynomials can represent more functions, and thus when varying the training dataset they will learn different functions all the time.

But suppose now that for a problem we suspect that the CEF has some non-linear behaviour, and we would like to learn this non-linear behaviour with a polynomial model without having the big variance that comes with this model.

It is possible through what we call the "regularisation" of the model.

Many types of regularisations exist, but the essence lies in reducing the number of functions represented by a model.

One popular way to regularise polynomial models is through a method called Tikhonov regularisation (also called Ridge Regression).

Instead of trying to optimise the empirical risk, we try to minimise the ridge empirical risk defined as shown in Equation 3.1.

$$\min_{\theta} \hat{\mathcal{R}}_{S_N}^{\alpha}(\text{Poly}_{\theta}^d) = \min_{\theta} \frac{1}{N} \left(\sum_{i=1}^N \left((X_i^d \cdot \theta - Y_i)^2 + \alpha \|\theta\|_2^2 \right) \right) \quad (3.1)$$

With α being the parameter that determines how much the squared norm of θ weighs in the empirical risk equation.

The solution is given by equation 3.2 where I_N is the identity matrix

$$\hat{\theta} = ((X_{S_N}^d)^T \cdot X_{S_N}^d + \alpha I_N)^{-1} \cdot (X_{S_N}^d)^T \cdot Y_{S_N} \quad (3.2)$$

In this case we can show that the solution of ridge regularisation is the solution of a constraint optimisation problem as defined in Equation 3.3.

$$\min_{\|\theta\|_2^2 \leq c} \hat{\mathcal{R}}_{S_N}(\text{Poly}_{\theta}^d) = \min_{\|\theta\|_2^2 \leq c} \frac{1}{N} \sum_{i=1}^N (X_i^d \cdot \theta - Y_i)^2 \quad (3.3)$$

Where c is the parameter that determines how much the squared norm of θ weighs in the empirical risk equation.

The author did not find a peer-reviewed paper discussing the link between ridge regression and constraint generalisation.

The proof for this result used in this section was found in Ayah [2018].

Both α and c are linked with the solving the constraints optimisation problem with the KKT conditions on the Lagrangian.

We can always find c given α and vice versa: c is the norm of the solution we found with ridge regression with a given α , and the α is equal to the λ parameter of the KKT conditions on the Lagrangian of the constraint optimisation problem given by c .

So changing the α parameter, like changing the degree of the polynomial, is a way to change the size of the input space.

The bias-variance trade-off heuristic tells us that increasing α will decrease the variance, which is illustrated by the comparison of the results of ridge regression on figure 3.1 with the least squares regression on figure 2.2.

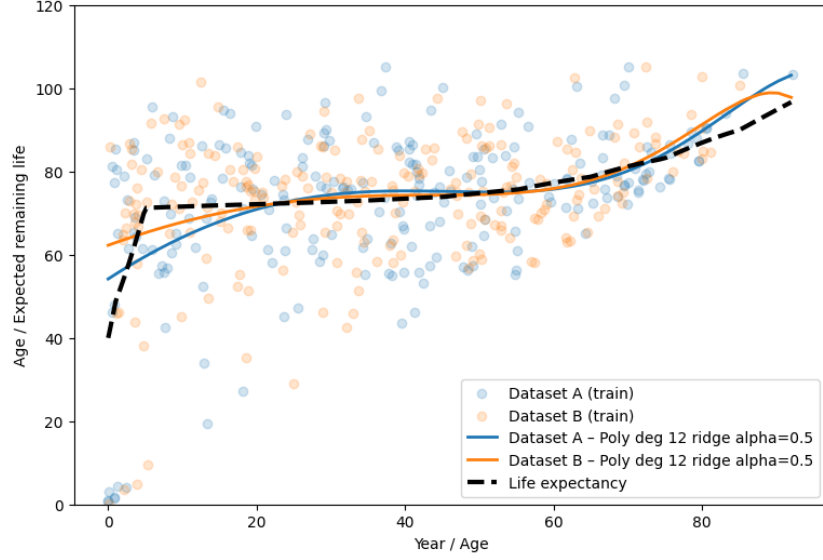


Figure 3.1: Ridge regression polynomial degree 12 on two different datasets

It is generally expected for models with high expressive power to need some kind of "regularisation" to perform well in SL.

3.1.1 Gradient flow

We will study the training of parametric models with gradient flow.

Gradient flow describes the dynamics of gradient descent with vanishing step size.

More precisely, the variation of the value of the parameters of a parametric model during training follows the ordinary differential equation defined in Equation 3.4.

$$\frac{d}{dt}(\Theta) = -\nabla_{\Theta} \left(\hat{R}_{\mathcal{S}_N}(\Theta) \right) \quad (3.4)$$

Where Θ is the vector containing all the parameters and $\nabla_{\Theta} \left(\hat{R}_{\mathcal{S}_N}(\Theta) \right)$ is the gradient of a particular empirical training error.

3.2 Implicit regularisation

Until now we did not mention the role of the optimiser on the performance of supervised learning algorithm

Most supervised learning models minimise the empirical risk by using some variant of the gradient descent method on the parameters of a specific parametric model.

This can have an impact on the effective hypothesis space of the model, the reason is illustrated in figure 3.2

The effect of a parametric model is that it maps the set of functions in the hypothesis space into a parametric space $\mathcal{H} \rightarrow \Theta$ (we will assume for the rest of the document that this mapping is $\mathbb{R}^n$).

This induces a mapping from the parameter space to a value of the risk, and given a training dataset it also implies a mapping from the parameter space to a value of the empirical risk. To find the best function that minimises the empirical risk, we perform gradient descent.

Gradient descent begins with a first iterate, then updates the iterate going in the direction of steepest descent.

The important observation is that the parameter space Θ is a vector space, and that organises the functions in $\mathcal{H}$ in an ordered way.

Suppose we have identified 2 groups of functions: "steep functions" and "flat functions". For linear models, this can represent functions whose graph is a steep hyperplane or flat hyperplane respectively.

It is possible that a particular modelling places the functions in these groups closer together in the parameter space. Then, when starting with a particular initial iterate, gradient descent could, depending on the empirical risk function, finish with an output in the group of flat functions or in the group of steep functions. This is a kind of "implicit regularisation".

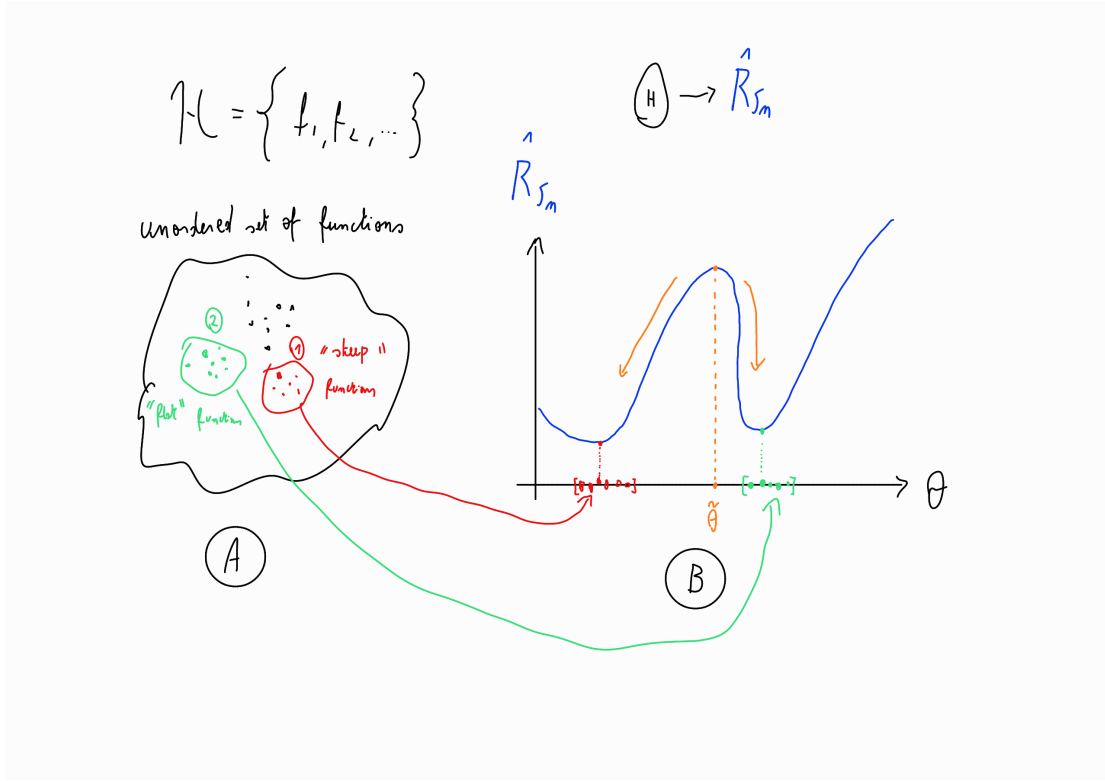


Figure 3.2: Big picture

Implicit regularisation refers to a regularisation process that biases the result of the optimisation step of the supervised learning algorithm towards certain kind of solutions instead of others, without it being explicitly encoded in the objective of the optimisation procedure.

We can see this phenomenon in action in overdetermined linear regression trained with gradient flow.

3.2.1 Solution of linear regression trained with gradient flow

Let $X_{S_N} \in \mathbb{R}^{N \times d}$ and $Y_{S_N} \in \mathbb{R}^N$ be the input matrix and target vector where each line corresponds respectively to an input and a target in the training dataset. Let $\theta \in \mathbb{R}^d$ be a vector of parameters.

We want to solve the gradient flow on the problem given by the empirical risk equation:

$$\hat{R}_{S_N}(\theta) = \frac{1}{N} \|X_{S_N} \cdot \theta - Y_{S_N}\|_2^2 \quad (3.5)$$

The training dynamics are illustrated by Figure 3.3.

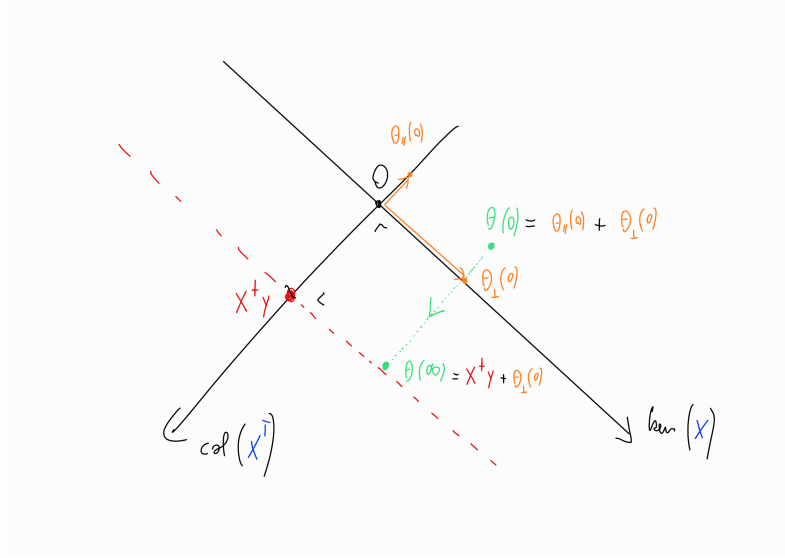


Figure 3.3: Linear regression optimisation path

Let $\theta(0)$ be the first iterate of the gradient flow, then we can always decompose this first iterates into:

$$\theta(0) = \theta_{\parallel}(0) + \theta_{\perp}(0), \quad \theta_{\parallel}(0) \in \text{col}(X_{\mathcal{S}_N}^{\top}), \quad \theta_{\perp}(0) \in \ker(X_{\mathcal{S}_N}). \quad (3.6)$$

When $X_{\mathcal{S}_N}$ is full column rank, then the kernel of $X_{\mathcal{S}_N}$ contains only the 0 vector and the parameter vector $\theta(\infty)$ converges to the unique minimum empirical risk solution $X_{\mathcal{S}_N}^{\dagger} \cdot Y_{\mathcal{S}_N}$ where $X_{\mathcal{S}_N}^{\dagger}$ is the Moore-Penrose pseudo-inverse of $X_{\mathcal{S}_N}$.

When $X_{\mathcal{S}_N}$ is not full column rank (this corresponds to an over-parametrised problem), there exists an infinite amount of minimum empirical risk solutions: all vectors $\theta \in \{X_{\mathcal{S}_N}^{\dagger} \cdot Y_{\mathcal{S}_N} + \theta_{\perp} | \theta_{\perp} \in \ker(X)\}$. In Figure 3.3 these optimal solutions lie in the red dashed line.

For any initial iterate, the solution will converge to

$$\theta(\infty) = \theta_{\parallel}(\infty) + \theta_{\perp}(0) = X_{\mathcal{S}_N}^{\dagger} \cdot Y_{\mathcal{S}_N} + \theta_{\perp}(0). \quad (3.7)$$

The proof for these results can be found in Appendix E.

3.2.2 Implicit regularisation in linear regression trained with gradient flow

A first kind of implicit regularisation happens when we choose an initial iterate $\theta(0)$ with no part in the kernel of $X_{\mathcal{S}_N}$: $\theta_{\perp}(0) = 0$. This can always be satisfied by starting with $\theta(0) = 0$. Then the gradient flow converges to the optimal solution

with the minimum norm $X_{\mathcal{S}_N}^\dagger \cdot Y_{\mathcal{S}_N}$. Nowhere in the training algorithm we explicitly penalise the norm of the solution, this regularisation is implicit.

But also for other initial iterates, we can consider some implicit regularisation operates. We have defined already "flat functions" as functions whose graph corresponds to "flat" hyperplanes and "steep functions" as functions whose graph correspond to a steep hyperplane.

The parameter model mapping orders the functions of our hypothesis space in such a way that parameters corresponding to flat functions are positioned close to $\theta = 0$ ($\|\theta\|$ is small) whereas parameters corresponding to steep functions are positioned far from the $\theta = 0$ ($\|\theta\|$ is big) .

Given a specific training dataset, if we take the part of our initial iterate in the kernel of $X_{\mathcal{S}_N}^T: \theta_\perp(0)$, the output of our supervised learning algorithm will keep its corresponding behaviour. It will keep its "steep" behaviour if we take a large initial iterate whereas he will keep its "flat" behaviour if we take an initial iterate close to the origin.

Because we have no input with some value in the kernel of $X_{\mathcal{S}_N}$ by definition, we can probably state as an heuristic that the part of the input in the kernel of $X_{\mathcal{S}_N}$ should have no impact on the prediction of the target. This is done choosing $\theta(0) = 0$ as our first iterate.

Chapter 4

Neural Networks

In this chapter we will briefly introduce Neural Networks (NNs), in particular multi-layer perceptrons (MLP), and their use in SL. We will see that they are learned with some variation of gradient descent.

We will then discuss the expressive power of neural network first with the universal approximation theorem, then with the number of linear regions.

Finally we will discuss the application of the bias-variance trade-off heuristic, the double-descent phenomena that shows it does not work very well: this suggests the existence of an implicit regularisation that makes so that gradient descent to learn in practice a reduced set of the set of functions a neural network can represent.

4.1 Introduction to Neural Networks

NNs are a type of parametric model loosely mimicking the working of the human brain.

In this master thesis we will only study Fully Connected (FC) neural networks. The basis of this parametric model is the perceptron model.

4.1.1 Perceptron

$$\text{Perceptron}(x) = h(\theta_0 + \sum_{i=1}^n \theta_i x_i) \quad (4.1)$$

A perceptron parametric model is a function $p : \mathbb{R}^n \rightarrow \mathbb{R}$. It is computed in two steps:

- We first compute a weighted sum of the inputs (including adding a constant parameter called bias).

- We then pass this result through a non-linear function called activation function h .

A popular activation function h is the Rectified Linear Unit (ReLU).

$$\text{ReLU}(x) = \max(x, 0) \quad (4.2)$$

These perceptrons can also be symbolically represented as neurons like in Figure 4.1.

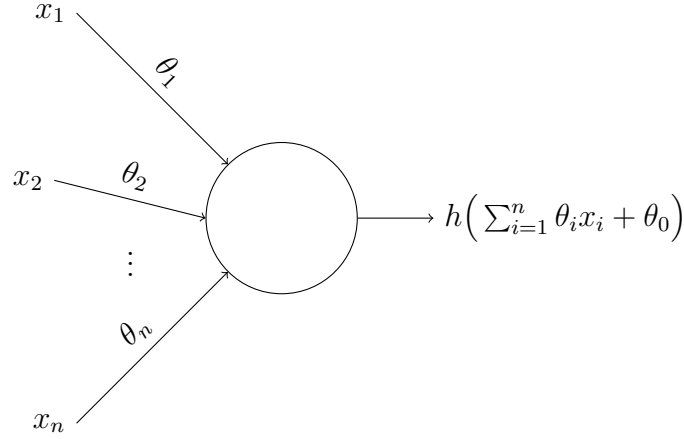


Figure 4.1: Symbolical representation of a perceptron

The weights and the biases are the parameters of the perceptrons.

4.1.2 Deep and shallow neural networks

A perceptron alone can only represent ramp functions, but we can do a weighted sum of many ramp functions to construct a shallow NN.

$$\text{shallow}(x) = \theta_0^1 + \sum_{k=1}^W \theta_k^1 f_k(x), \quad (4.3)$$

$$f_i(x) = h\left(\theta_0^{0,i} + \sum_{k=1}^n \theta_k^{0,i} x_k\right), \quad i \in \{1, \dots, W\}.$$

A shallow NN with ReLU activation function is just a linear combination of W ramp functions.

It can symbolically be represented as a graph like in Figure 4.2.

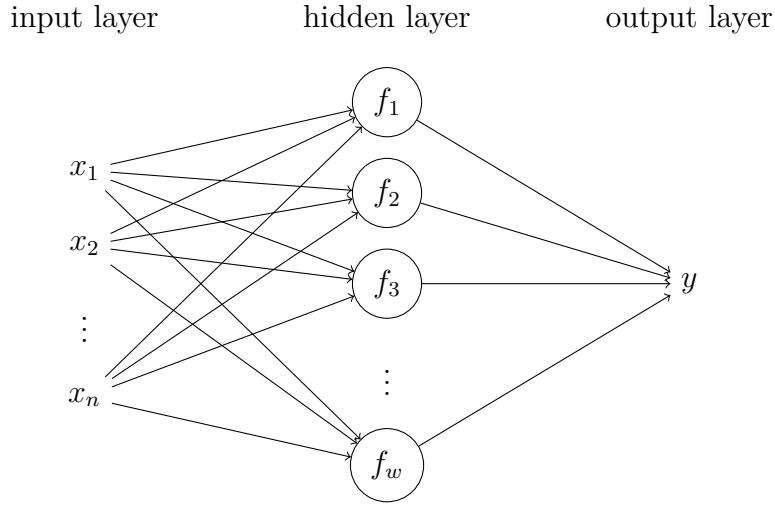


Figure 4.2: Symbolical representation of a feedforward neural network with one hidden layer.

You can see that every edge corresponds to a weight and every node corresponds to either an input, the output or an activation function. There is also a bias parameter for every node representing an activation function, having this symbolical representation can thus allow us to easily compute the number of parameters.

Shallow NNs are also called NNs with one hidden layer. This layer is called hidden in the sense that we don't know from the output of the NN the output of the different functions f_i .

A variation that can be done is to feed the output of the different f_i functions to a new layer of functions, and then a linear combination of the output of this new layer would make the output of this neural network.

What was just described is a 2-hidden-layer NN which is illustrated in Figure 4.3.

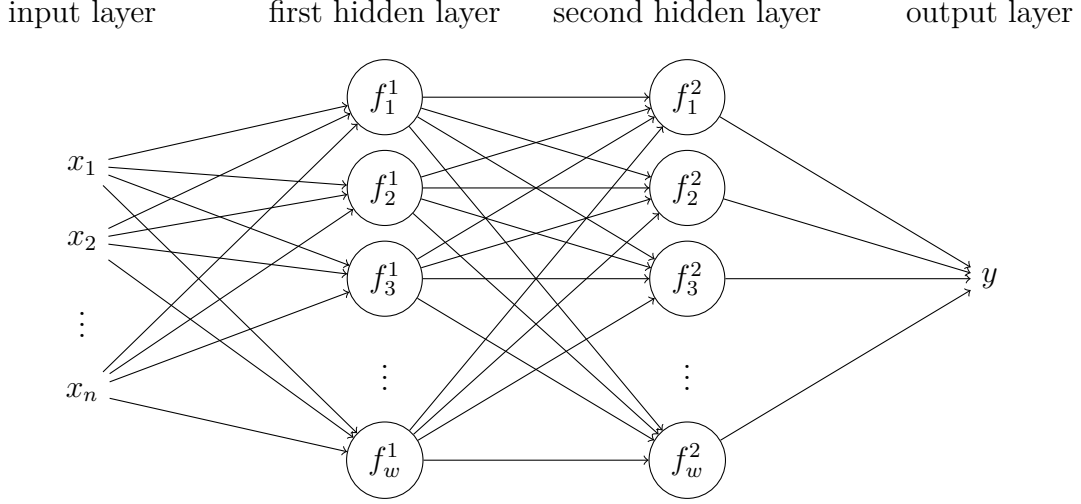


Figure 4.3: Symbolical representation of a feedforward neural network with two hidden layers.

In all generality, a FC NN with ReLU activation functions with $L-1$ hidden layers, or of depth L , follows this formula of Equation 4.4.

These neural networks are also called multi-layer perceptrons (MLP)

$$\text{deep}(x) = \theta_0^L + \sum_{k=1}^{W_{L-1}} \theta_k^L f_k^L(x),$$

$$f_i^j(x) = \text{ReLU} \left(\theta_0^{j-1,i} + \sum_{k=1}^{W_{j-1}} \theta_k^{j-1,i} f_k^{j-1}(x) \right), \quad j \in \{1, \dots, L-1\}, \quad i \in \{1, \dots, W_j\},$$

$$f_i^0(x) = x_i, \quad i \in \{1, \dots, n\}. \quad (4.4)$$

4.1.3 Neural network's expressive power

It has been widely demonstrated that neural networks, with varied types of activation functions, can approximate any possible continuous functions, these types of results are called "universal approximation theorems" (the first one to prove one of them was Cybenko [1989]).

A universal approximation theorem has in particular been proven for NNs with ReLU activation functions Lu et al. [2016].

These results hold for networks of any depth and while varying the width of the hidden layers.

For a specific depth or length, shallow and deep NNs with ReLU activation functions learn piecewise linear functions, and the number of linear regions R is given by theorem 5 of Montúfar et al. [2014] and can be found in equation 4.5.

$$R \geq \prod_{l=1}^{L-2} \left\lfloor \frac{W_l}{W_0} \right\rfloor^{W_0} \sum_{j=0}^{W_0} \binom{W_{L-1}}{j}. \quad (4.5)$$

As also explained in the paper, the number of linear regions grows polynomially with the width but exponentially with the depth.

In light of this result, we can say that the expressive power of mlps grows with its length and its width.

4.1.4 Learning of neural networks

Back-propagation and stochastic gradient descent

It is possible to relatively efficiently calculate the partial derivatives of NN output by their parameters $\theta_{i,j}^k$ through the back-propagation algorithm (introduced at first in Rumelhart et al. [1986]).

Calculation of partial derivatives is only relatively efficient because the cost of calculating them on a training error grows linearly with the size of the training set.

Because of that, NNs are generally learned with some stochastic gradient descent method.

These methods are variations of gradient descent which use an approximation of the gradient of the empirical risk using only part of the training data.

While this training has complex dynamics, for the rest of this master thesis we will analyse the training of NNs using gradient flow defined in section 3.1.1.

Generalisation of this master thesis to a training with gradient descent and then a training with stochastic gradient descent are left for future works.

4.2 Neural networks in supervised learning

4.2.1 Bias variance trade-off heuristic for neural networks and double descent phenomenon

Using the bias-variance trade-off heuristic, the fact that deep NNs can represent many functions should be interpreted as much as a blessing as a curse.

On the one end, for NNs to be successful in many SL tasks, we expect them to be able to approximate a wide variety of CEFs.

But on the other hand, we expect models with high expressive power to have high variance and to thus need regularisation to have good performances.

The surprising fact is that empirical evidence shows that for a large amount of problems, we observe very big NNs perform well in SL tasks without needing any explicit regularisation method to decrease their variance.

More precisely, when increasing the number of layers and the depth of the layers the performance of NNs increase.

An influential paper that described this phenomenon called it "double descent" (Belkin et al. [2019]), because if the number of hidden layers is small the test error follows the U-curve described in Figure 2.3, but it comes a point where the test error decreases again (in many problems it is past the interpolation point: when the number of parameter is equal to the number of data points). An important observation to make is that the double descent phenomena describe the fact that deep neural networks perform better than shallow neural networks.

This suggest the existence of some type of implicit regularisation that reduce the variance of the supervised learning procedure use to train big neural networks.

Chapter 5

Linear neural networks and implicit regularisation

What we want to achieve in the study of more simplified NNs, like linear NNs, is to come up with results that can be generalised to more complex NNs.

In this chapter, we will present the results of Section 4 of Marion and Chizat [2024], and will discuss to which extent they can also be seen as implicit regularisation.

We will end by explaining the difficulties of coming up with an implicit regularisation for linear neural network, and the further studies that have to be done.

In the next chapter, we will see that some results that cannot be considered implicit regularisation for linear neural networks can be considered implicit regularisations for MLPs.

5.1 Modelling of linear neural networks

Linear NNs are NNs without activation function and without bias as defined in Equation 5.1.

$$\begin{aligned} \text{linear}_\Theta(x) &= \sum_{k=1}^{W_{L-1}} \theta_k^L f_k^L(x), \\ f_i^j(x) &= \sum_{k=1}^{W_{j-1}} \theta_{k,i}^{j-1} f_k^{j-1}(x), \quad j \in \{1, \dots, L-1\}, \quad i \in \{1, \dots, W_j\}, \\ f_i^0(x) &= x_i, \quad i \in \{1, \dots, n\}. \end{aligned} \tag{5.1}$$

To help us study linear NNs, it can be useful to rewrite a linear network in its matrix form as defined in Equation 5.2.

$$\begin{aligned} \text{linear}_\Theta(x) &:= x \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L \\ \Theta^j &:= \begin{bmatrix} \theta_{0,0}^j & \dots & \theta_{0,W_j}^j \\ \vdots & \ddots & \vdots \\ \theta_{W_{j-1},0}^j & \dots & \theta_{W_{j-1},W_j}^j \end{bmatrix} \end{aligned} \quad (5.2)$$

Notice that the matrix form used in this document is different than the matrix form used in Marion and Chizat [2024]: in their representation the matrices of Equation 5.2 are transposed.

Notice also that the last matrix Θ^L is a column vector, in general proofs presented will assume this fact.

5.2 Gradient flow for linear neural networks

The risk for linear NNs on least squares regression is defined in Equation 5.3.

$$R(\Theta) = \|Y - \text{linear}_\Theta(X)\|_2^2 \quad (5.3)$$

Where Y and X are given data matrices.

The formula for the gradient was given by the paper is repeated in Equation 5.4.

$$\begin{aligned} \nabla_{\Theta^j}(R(\Theta)) &= \nabla_{\Theta^j} \left(\|Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L\|_2^2 \right) \\ &= 2 \cdot (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot (\text{linear}_\Theta(X) - Y) \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T \end{aligned} \quad (5.4)$$

Because we will later study the gradient flow of affine NNs, it can be useful to develop a proof in such a way that can be easily generalised for more complex NNs: the author's proof can be found in Appendix F.

5.3 Implicit regularisation for linear neural networks with small initialisation

Although it was not the main objective of Marion and Chizat [2024], we will revisit the results of its section 4: "analysis of gradient flow from small initialisation" and interpret them as implicit regularisation.

We will compare the results we have with the implicit regularisation in linear regression.

5.3.1 Main results for linear neural networks

Weight balancing lemma

Without assuming anything more, the parameter matrices of linear neural networks trained with gradient flow on minimum square loss are subject to what will be called the "weight matrices balancing lemma" (this corresponds to lemma 1 in Marion and Chizat [2024]).

The proof can be found in Appendix H.

$$\frac{d}{dt} \left(\Theta^{j+1} \cdot (\Theta^{j+1})^T - (\Theta^j)^T \cdot \Theta^j \right) = 0 \quad \forall t \geq 0 \quad \forall j \in \{1, \dots, L-1\} \quad (5.5)$$

This lemma plays an important role because at any point in the training path, the weights will be subject to this invariance.

Small weight initialisation and weight alignment lemma

For proving the rest of the results, the paper assumes:

$$R(\Theta(0)) \leq R(0) \text{ and } \nabla_{\Theta} R(\Theta(0)) > 0 \quad (5.6)$$

It also assumes that before training by gradient flow, the first iterate of each parameter matrix to be of small Frobenius norm: what we mean precisely is that we want to define $\epsilon > 0$ arbitrarily small, and we want that the first iterate satisfies Equation 5.7.

$$3 \cdot \left(\max_{1 \leq j \leq L} \|W^j(0)\|_F^2 + 2 \cdot \sum_{i=1}^{L-1} \|(\Theta^i(0))^T \cdot \Theta^i(0) - \Theta^{i+1}(0) \cdot (\Theta^{i+1}(0))^T\|_2 \right) \leq \epsilon \quad (5.7)$$

Satisfaction to conditions of equation 5.6 can be easily obtained by generating the first iterate of all but 1 arbitrarily chosen parameter matrix using random Gaussian entries with same variance, and setting the last parameter matrix to 0.

Satisfaction to condition of Equation 5.7 is obtained by reducing the variance until the condition is satisfied.

Under these 2 assumptions, the parameter matrices of linear neural networks during training are subject to what we will call the "weight matrices alignment lemma" (which corresponds to lemma 2 in Marion and Chizat [2024])

The weight matrices alignment lemma consists of 3 results that can be found in equation 5.8.

$\forall t \geq 0 :$

$$\begin{aligned}
& \text{(i)} \quad \forall k \in \{1, \dots, L\}, \quad \|\Theta^k(t)\|_F^2 - \|\Theta^k(t)\|_2^2 \leq \varepsilon, \\
& \text{(ii)} \quad \forall j, k \in \{1, \dots, L\}, \quad \left| \sigma_k^2(t) - \sigma_j^2(t) \right| \leq \varepsilon, \\
& \text{(iii)} \quad \forall k \in \{1, \dots, L-1\}, \quad \left\langle v_k(t), q_{k+1}(t) \right\rangle^2 \geq 1 - \frac{\varepsilon}{\sigma_{k+1}^2(t)},
\end{aligned} \tag{5.8}$$

Where $\Theta^k(t)$ is the weight matrix of layer k at time t , $\|\cdot\|_F$ is the Frobenius norm, $\|\cdot\|_2$ the spectral norm, $\sigma_k(t)$ the largest singular value of $\Theta^k(t)$, and $q_k(t), v_k(t)$ corresponding to respectively the first left and the first right singular vectors.

q was used for the left singular vector instead of the more widely used u because u and v were too similar letters that could be confused between each other.

Identity (i) of Equation 5.8 means that at any time the matrices are nearly rank 1 or rank 0.

Identity (ii) of Equation 5.8 bounds the difference in spectral norm between two weight matrices, and assuming the matrices are nearly rank 1 or 0 this also implies a bound on the difference of their Frobenius norms.

Identity (iii) of Equation 5.8 shows that for two consecutive weight matrices the right first singular vector of the first matrix is nearly aligned with the first left singular vector of the second one.

NB1: These results hold for any value of ε , but the interpretation only makes sense for small values of ε

NB2: To come up with these results, we assumed $\|\Theta^L\|_F^2 - \|\Theta^L\|_2^2 = 0$, this assumption is satisfied because Θ^L is a 1 column matrix.

While this was not the main objective of the paper, this lemma has some interesting implication on the value of all parameters of the solution during training and at convergence.

Let us define Θ^S to be the vector which is the product of all parameter matrices Θ^i as shown in equation 5.9.

$$\Theta^S := \prod_{i=1}^L \Theta^i \tag{5.9}$$

There are many ways to multiply matrices so that they multiply to Θ^S . We are interested in the minimal sum of square coefficient solution, which is the result of the problem of equation 5.10.

$$C = \min_{\Theta^i} \sum_{i=1}^L \|\Theta^i\|_F^2 \quad \text{subject to} \quad \prod_{i=1}^L \Theta^i = \Theta^L \tag{5.10}$$

Then the sum of square coefficient is optimal up to an order of $\epsilon^{\frac{1}{L}}$

$$\sum_{i=1}^L \|\Theta^i\|_F^2 \leq C + \mathcal{O}(\epsilon^{\frac{1}{L}}) \quad (5.11)$$

The proof can be found in annex J. This proof is original and can be considered a contribution from the author.

PL condition

Using the results of Equation 5.8, Marion and Chizat [2024] claims that the regression problem satisfy PL condition, in the sense that there exists a constant $\mu > 0$ such that for all $t \geq 1$:

$$\sum_{k=1}^L \left\| \nabla_{\Theta}^k (R(\Theta)) \right\|_F^2 \geq \mu (R(\Theta) - R_{\min}),$$

Where R is the empirical risk, $R_{\min}$ is its minimum value and $\Theta = \Theta(t)$ are the weights.

The author was not able to verify well enough the proof to verify the interpretation of "for all $t \geq 1$ " but he assumes it implies at least: "at some point during training this condition is satisfied".

Consequently, the gradient flow converges exponentially fast to a global minimiser of the risk. This implies a region of convergence for the product of all parameter matrices Θ^P .

$$\Theta^P(\infty) \in \text{Opt} = \{X^\dagger y + \theta | \theta \in \ker(X)\} \quad (5.12)$$

Implicit regularisation of linear neural networks

For linear neural networks, to understand how they are subject to an implicit regularisation, we only need to understand the gradient flow of the product matrix. This is because knowing the product matrix gives us sufficient information to define the linear function the neural networks represent.

For linear neural networks, we only have studied the case where the initial iterate is subject to the conditions of equations 5.6 and 5.7.

And even in this case, we have only proven that the solution converges to the global optimum, so it seems we haven't found any explicit regularisation for linear neural networks

The study of the gradient flow map of figure 3.3 and how it generalizes for the product matrix in linear neural network could be an interesting study by itself, but

the implicit bias of linear neural networks in itself is not the end-goal. What we are interested in is how much this implicit bias informs us on the implicit bias of mlps.

In the next chapter, we will see how the result of equation 5.11, if they hold for mlps, constitute an implicit regularisation of mlps.

Part II

Contribution of the author

Chapter 6

Gradient flow of mlps

In the previous chapter we discussed some results in linear neural networks, in this chapter we will discuss how we can use our knowledge of linear neural networks too deduce some implicit regularisation on mlps if these results hold.

We will in particular show that some gradient descent paths in mlps are exactly the same as particular gradient descents paths in linear neural networks.

We will then motivate the need to study affine neural networks by suggesting that for a property to be generalisable from linear neural network to mlp it must first be generalisable to affine neural networks.

6.1 Matrix form of mlps

As we discussed in the previous chapter, the result of equation 5.11 was not an implicit regularisation of linear neural networks, because it did not impact the product matrix.

Nonetheless we will see it is a implicit regularisation if these results hold for mlps. To understand that let's write a 2-layer linear neural network modified adding a relu activation function on the first layer. This way we make it more similar to the general form of neural network described in equation 4.4.

$$\begin{aligned} \text{linear}_{\Theta}(x) &:= \text{ReLU} \left(x \cdot \begin{bmatrix} \theta_{1,1}^1 & \theta_{1,2}^1 \\ \theta_{2,1}^1 & \theta_{2,2}^1 \end{bmatrix} \right) \cdot \begin{bmatrix} \theta_{1,1}^2 \\ \theta_{2,1}^2 \end{bmatrix} \\ &= \begin{bmatrix} \text{ReLU} \left(x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \right) & \text{ReLU} \left(x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \right) \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1}^2 \\ \theta_{2,1}^2 \end{bmatrix} \end{aligned} \quad (6.1)$$

This allows us to identify 4 linear zones in the input-output map that are illustrated by figure 6.1.

Zone 1 is the zone where $x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \leq 0$ and $x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \leq 0$, so because of this $\text{linear}_\Theta(x) = 0$.

Zone 2 and 3 are zones where respectively $x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \geq 0$ and $x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \leq 0$, and $x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \leq 0$ and $x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \geq 0$. the output is determined by the linear neural networks described by the following equations:

$$\begin{aligned} \text{zone}_{2\Theta}(x) &:= x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1}^2 \end{bmatrix} \\ \text{zone}_{3\Theta}(x) &:= x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \cdot \begin{bmatrix} \theta_{2,1}^2 \end{bmatrix} \end{aligned} \quad (6.2)$$

We see that the relu function changes the matrix-structure of the original neural network by removing columns in the matrix where the output of the relu function is 0 then removing the corresponding line in the next matrix.

Zone 4 is the zone where $x \cdot \begin{bmatrix} \theta_{1,1}^1 \\ \theta_{2,1}^1 \end{bmatrix} \geq 0$ and $x \cdot \begin{bmatrix} \theta_{1,2}^1 \\ \theta_{2,2}^1 \end{bmatrix} \geq 0$, the output is equal to the sum of the output of the linear neural network of zone 2 and zone 3., but he can also be described by the following equation:

$$\text{zone}_{4\Theta}(x) := x \cdot \begin{bmatrix} \theta_{1,1}^1 & \theta_{1,2}^1 \\ \theta_{2,1}^1 & \theta_{2,2}^1 \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1}^2 \\ \theta_{2,1}^2 \end{bmatrix} \quad (6.3)$$

The different zones in the input space are determined by the parameter, so during training when the parameters are updated a particular input in the training dataset can jump from one zone to the other.

But for the sake of argument let assume a particular supervised learning problem where all the data points of the training set are in zone 4, and that during training they do not change zone, then the training of the network is equivalent to the training of the linear network describing zone 4, all the results of chapter 5 hold and in particular the result of equation 5.11 hold.

In the previous chapter, we said that results of equation 5.11 did not constitute an implicit bias for linear neural networks because the only aspect influencing the output was the product matrix. When we add relu activation to a linear neural network, because the parameters of the linear neural networks in zone 2 and 3 can be found in zone 4, the same result constitute an implicit bias for this kind of networks.

The suspected implicit regularisation works as such: if we begin with small weight initialisation (this corresponds to a "flat" piecewise linear function), because we keep the weights minimised during training, we will converge to a "flat" solution.

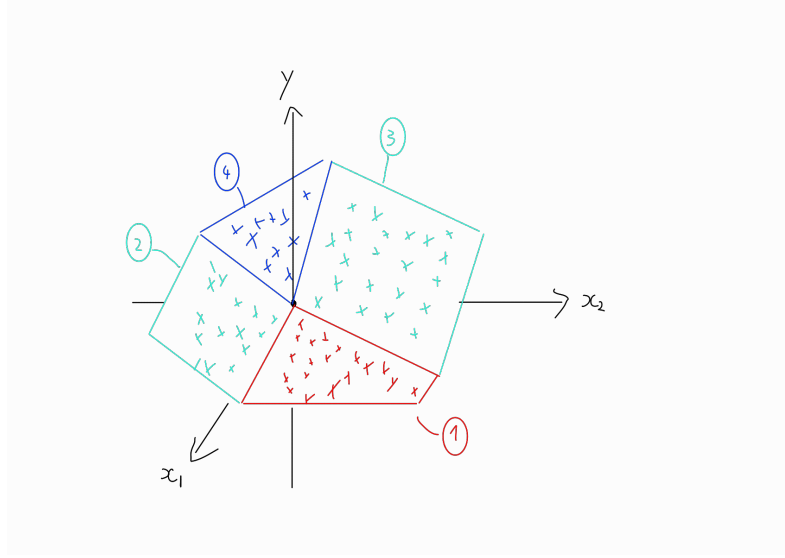


Figure 6.1: ReLU linear network network

To further this aim, the study of the generalisation of properties 5.8 and 5.11 for linear neural network adding relu activation functions could be an interesting work in itself. Another possibility would be to relax the conditions of equations 5.6 and 5.7 to come up with bounds on the final solution (because for now we only considered first iterates close to 0).

The author decided to try to generalise these properties in a different way. Linear neural networks can only represent linear functions, whose graph are planes passing through the origin. To represent affine functions in general, the author will study a new kind of neural network: affine neural networks.

Chapter 7

Affine Neural networks

In this chapter we will introduce affine neural networks, a middle link between linear neural networks and mlps.

We will then investigate if the results of chapter 5 hold for affine neural networks. We will show in particular that the weight matrices balancing lemma, upon which all other shown properties of linear neural networks rest upon, does not hold in general for affine neural networks.

We will then discuss the result of a numerical test on a particular training of affine neural networks and discuss what it implies on the implicit regularisation of mlps.

7.1 Motivation for affine Neural Networks

As already discussed in Section 4.1.3, mlps represent piecewise linear functions.

It is important to know that although we call them piecewise "linear" functions, what we really mean by that is piecewise "affine" functions.

This distinction is important because linear neural networks can only represent linear functions, in figure 6.1 they would be represented by hyperplanes passing through the origin.

Hyperplanes not passing through the origin cannot be represented by linear neural networks, and mlps represent "connected segment of hyperplanes" not necessarily passing through the origin.

An important observation is that mlps can be perfectly described as affine neural networks in the same way as described in chapter 6 for linear neural networks with relu activation function. The different affine regions share the same parameters.

If we would be able to show similar behaviour for affine neural networks as in the linear regression case with equations 5.5 and 5.11, it can be a first step to understand the implicit regularisation of mlps.

7.2 Modelling affine neural networks

Affine NNs are a generalisation of linear NNs where we add the biases as defined in Equation 7.1.

$$\begin{aligned}
 \text{affine}_{\Theta}(x) &= \theta_0^{L-1} + \sum_{k=1}^{W_L} \theta_k^L f_k^L(x) \\
 f_i^j(x) &= \theta_{0,i}^{j-1} + \sum_{k=1}^{W_{j-1}} \theta_{k,i}^{j-1} f_k^{j-1}(x), \quad j \in \{1, \dots, L-1\}, \quad i \in \{1, \dots, W_j\}, \\
 f_i^0(x) &= x_i \quad i \in \{1, \dots, n\}.
 \end{aligned} \tag{7.1}$$

The author did not find any paper studying specifically affine NNs.

To study the properties, it is necessary to develop the necessary mathematical framework.

To that aim, it can be useful to represent affine NNs in their matrix multiplication form like for linear NNs.

Assuming $W_0 = n$ and defining $x^* := \begin{bmatrix} 1 & x \end{bmatrix}$ we have:

$$\begin{aligned}
 \text{affine}_{\Theta}(x) &:= x^* \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L \\
 \Theta^j &:= \begin{bmatrix} 1 & \theta_{0,1}^j & \dots & \theta_{0,W_j}^j \\ 0 & \theta_{1,1}^j & \dots & \theta_{1,W_j}^j \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \theta_{W_{j-1},1}^j & \dots & \theta_{W_{j-1},W_j}^j \end{bmatrix} \quad \forall j \in \{1, \dots, L-1\} \\
 \Theta^L &:= \begin{bmatrix} \theta_0^L \\ \theta_1^L \\ \vdots \\ \theta_{W_{L-1}}^L \end{bmatrix}
 \end{aligned} \tag{7.2}$$

Let's call every matrix Θ^j with $j \in \{1, \dots, L-1\}$ an "affine matrix".

To add a bias, or an intercept in a linear regression model, it is common use to add a column of 1 in your data matrix.

We use the same trick here, and we add to each affine matrix with the exception to the last a column with 1 to the first line and the appropriate number of 0 in the next lines.

This allows to do the following decomposition:

$$\begin{aligned}
\Theta^j &:= \begin{bmatrix} 1 & \theta_{0,1}^j & \cdots & \theta_{0,W_j}^j \\ 0 & \theta_{1,1}^j & \cdots & \theta_{1,W_j}^j \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \theta_{W_{j-1},1}^j & \cdots & \theta_{W_{j-1},W_j}^j \end{bmatrix} = \begin{bmatrix} c^j & \tilde{\Theta}^j \end{bmatrix} = \begin{bmatrix} 1 & \tilde{\Theta}_b^j \\ 0^j & \tilde{\Theta}_s^j \end{bmatrix} \in \mathbb{R}^{(W_{j-1}+1) \times (W_j+1)} \\
c^j &:= \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{(W_{j-1}+1) \times (1)} \\
\tilde{\Theta}^j &:= \begin{bmatrix} \theta_{0,1}^j & \cdots & \theta_{0,W_j}^j \\ \theta_{1,1}^j & \cdots & \theta_{1,W_j}^j \\ \vdots & \ddots & \vdots \\ \theta_{W_{j-1},1}^j & \cdots & \theta_{W_{j-1},W_j}^j \end{bmatrix} \in \mathbb{R}^{(W_{j-1}+1) \times (W_j)} \\
0^j &:= \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{(W_{j-1}) \times (1)} \\
\tilde{\Theta}_b^j &:= [\theta_{0,1}^j \quad \cdots \quad \theta_{0,W_j}^j] \in \mathbb{R}^{(1) \times (W_j)} \\
\tilde{\Theta}_s^j &:= \begin{bmatrix} \theta_{1,1}^j & \cdots & \theta_{1,W_j}^j \\ \vdots & \ddots & \vdots \\ \theta_{W_{j-1},1}^j & \cdots & \theta_{W_{j-1},W_j}^j \end{bmatrix} \in \mathbb{R}^{(W_{j-1}) \times (W_j)}
\end{aligned} \tag{7.3}$$

We can give names to these different matrices:

- $\tilde{\Theta}^j$ can be called the parameter matrix, it is composed of :
 - $\tilde{\Theta}_b^j$ the bias vector, because it is the vector containing the biases
 - $\tilde{\Theta}_s^j$ the slope matrix, containing the rest of the parameters

7.2.1 Gradient flow equation for affine neural networks

Let's now represent the gradient flow of the learning of an affine network in a regression problem with least square error loss.

$$\frac{d}{dt}(\tilde{\Theta}^j) = -\nabla_{\tilde{\Theta}^j}(R(\Theta)) \quad (7.4)$$

We notice that contrary to the linear case, it is not the whole matrix that will be modified but only the parameter matrix.

This motivates the following operator:

$$\text{Inc}(\tilde{\Theta}) := \begin{bmatrix} 0 & \tilde{\Theta} \end{bmatrix} \quad (7.5)$$

This way when we can define the "incorporated gradient"

$$\nabla_{\tilde{\Theta}^j}^*(R(\Theta)) := \text{Inc}(\nabla_{\tilde{\Theta}^j}(R(\Theta))) : \quad (7.6)$$

This incorporated gradient allows us to write the gradient flow for all the affine matrices:

$$\frac{d}{dt}(\Theta^j) = -\nabla_{\tilde{\Theta}^j}^*(R(\Theta)) \quad \forall j \in \{1, \dots, L-1\} \quad (7.7)$$

The gradient flow for the last matrix uses the normal gradient and not the incorporated gradient.

Incorporated gradient for all affine matrices

The incorporated gradient of the affine matrix is thus a little different than in the linear case, it is shown in Equation 7.8. The proof is found in annex G.

$$\begin{aligned} \nabla_{\tilde{\Theta}^j}^*(R(\Theta)) = \\ 2 \cdot \text{Inc} \left((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot (\text{affine}_{\Theta}(X) - Y) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T \right) \end{aligned} \quad (7.8)$$

Discussion on the incorporated gradient

We can notice that we break a symmetry that was present in the linear case: the incorporated gradient of a weight matrix is dependent on the biases and slopes of the affine matrices before it but only on the slope parameters of the affine matrices after it.

This plays an important role for disproving the weight matrices balancing lemma.

7.2.2 Weight matrices balancing lemma for affine neural network

For the results in Section 5.3.1 for linear NNs, we assumed weight matrices balancing lemma as defined in Equation 5.5 did hold.

For affine NNs, however, the time derivatives of Equation 5.5 is not 0 but is as defined in Equation 7.9.

$$\frac{d}{dt} \left(\Theta^{j+1} (\Theta^{j+1})^T - (\Theta^j)^T \Theta^j \right) = \Delta + \Delta^T \quad (7.9)$$

With Δ defined in Equation 7.10

$$\Delta = 2 \cdot (\Theta^j)^T \cdots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y) \right) \cdot (\tilde{\Theta}_s^L)^T \cdots (\tilde{\Theta}_s^{j+2})^T \cdot \begin{bmatrix} (\tilde{\Theta}_b^{j+1})^T & 0 \end{bmatrix} \quad (7.10)$$

We now notice that this derivative is equal to 0 only if the value of the learned biases are equal to 0.

7.3 Linear and affine cases for linear regression and linear neural networks

In the statistical study of linear regression (in particular the statistical study of Ordinary Least Squares), affine and linear cases are the same because of the augmented matrix trick.

We can easily make the mistake of assuming it is also the case for linear and affine neural networks but it is not the case.

7.4 Numerical test on the alignment property in affine neural networks

We will conclude this chapter with a numerical test: we will do a qualitative assessment of the results of the training of an affine NNs with gradient descent and highlight the potential dynamics at play through the properties of linear and affine NNs.

A compiled notebook of the numerical tests can be found in this repository: <https://github.com/Pietro-Gavazzi/Affine-Neural-Networks-Eigen-Decomposition>.

For the numerical tests we trained an affine model on a least squares regression problem.

The data was generated with a linear model:

$$Y = \beta_0 + X\beta_{\text{true}} + \eta \quad (7.11)$$

Where the values of X , β_{true} and η are chosen following independent normal distributions and $\beta_0 = 1$.

The β_{true} has 4 features and 6 data-points were generated: if we wanted to train a linear regressor we would be in an over-determined case.

For the network, we trained a 4 layer affine NN of width 4 (in an matrix representation we have 3 5x5 and 1 5x1 weight matrix).

Let's test identities in Equation 5.8 on the matrices of the affine NN at convergence:

- for Θ_i with $i \leq L - 1$ we have:
 - The second singular value always are slightly less than 1 and the norms of the bias vectors are slightly greater than 0.
 - The first singular values are nearly of equal size: $\sigma_i = (1.655 \pm 0.010)$
 - The other singular values are close to 0.
- for Θ_L :
 - The only singular value is equal to $\sigma_L = 3.137 \pm 0.005$
- v_i and q_{i+1} are nearly aligned between all matrices but between the last one and the preceding one:
 - $v_1^T q_2 = -1 \pm 0.005$
 - $v_2^T q_3 = 0.993 \pm 0.005$
 - $v_3^T q_4 = -0.502 \pm 0.005$

Can we explain these results with the results seen before?

At the beginning of training, all parameters are initialised close to 0.

In the first iterations, the gradients for all parameters are close to 0 but the gradient for the bias of the last matrix Θ^L : it is immediately updated.

Remember that the Δ terms in the derivation of weight matrices balancing lemma are equal to 0 if the biases are equal to 0 as can be seen in Equations 7.9 and 7.10.

This means that for all the matrices but the last ones, weight matrices balancing lemma holds, further investigation must be conducted to verify if the other results in Section 5.3.1 hold for affine NNs.

If we increase the depth of the affine neural network, we can see the effect on more matrices and thus on a bigger proportion of the parameters.

7.5 Conclusion

The double descent phenomenon for mlps of Belkin et al. [2019] describes the fact that deep neural networks performed better than shallow neural networks, our numerical results showed that the weight balancing lemma and the matrix alignment lemma do not hold only for the last weight matrices in affine neural networks but does in practice hold for all the other matrices, suggesting a stronger effect the more weight matrices there are.

As explained in chapter 6, these two results can potentially play an important role in the implicit regularisation of mlps, even if further investigation must be done to verify how these results generalise when taking initial iterates far from 0, and when adding a relu activation function.

Perhaps we will discover that it is not those results who are the most important to describe the gradient path of mlps, but this might be an interesting try.

While he finds these results interesting by itself, the author consider himself new in the study of linear neural networks. There exist a large literature that study linear neural networks, and as stated by Marion and Chizat [2024], there is also a large literature in matrix completion that is closely related to the study of linear neural networks.

The author hopes to have provided a path to generalise these result to affine neural networks, which he believes can show new insight on the workings of mlp's

Thanks

A great thanks to my family and Lucie for their emotional support.

Thanks to Guido for the chess matches in the late evenings after work.

A particular thank to Lucie for her help by correcting my bad English.

Appendix A

Formal Definitions

A.1 Risk of a function and Hypothesis Space

Imagine we want to predict the value of some variable Y as well as possible, where we only know the value of some other variable X .

More formally, we say we have some random variables $(X \in \mathcal{X}, Y \in \mathcal{Y})$ following some joint probability distribution $\mathcal{D}_{(X,Y)}$, we write $(X, Y) \sim \mathcal{D}_{(X,Y)}$.

The X is called the "input" and Y is called "target". When X is a vector each individual component is called a "feature".

"Predicting the value of Y " means finding the function $\hat{f} : \mathcal{X} \rightarrow \mathcal{Y}$ that given a value X predicts the best possible value for Y .

More precisely, "predicting best" is formally defined as taking the minimum of what we call the "risk". The risk is the expectation of some "loss function" $\mathcal{L} : (\mathcal{Y}, \mathcal{Y}) \rightarrow \mathbb{R}_+$ which takes as input Y and its prediction $\hat{f}(X)$, and outputs some real positive number.

The risk is thus formally defined in Equation A.1.

$$\mathcal{R}(f) = \mathbb{E}_{(X,Y) \sim \mathcal{D}_{(X,Y)}}[\mathcal{L}(Y, f(X))] \quad (\text{A.1})$$

In theory f can be any function $\mathcal{X} \rightarrow \mathcal{Y}$, but in practice this problem is very difficult, so we restrict ourself to some defined set of functions $\mathcal{H} = \{h : \mathcal{X} \rightarrow \mathcal{Y}\}$, called the "hypothesis space".

We can thus define our problem as finding the function $\hat{f}$ within $\mathcal{H}$ that minimizes the risk $\mathcal{R}(f)$ as defined in Equation A.2.

$$\hat{f} = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \mathcal{R}(f) = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \mathbb{E}_{(X,Y) \sim \mathcal{D}_{(X,Y)}}[\mathcal{L}(Y, f(X))]. \quad (\text{A.2})$$

A.2 Supervised learning task

We have introduced the risk formula in Equation A.1, however it can not be computed in practice because we do not know the real distribution $\mathcal{D}_{(X,Y)}$.

For "supervised learning" we assume we have access to some "sample dataset" $\mathcal{S}_N = \{(X_i, Y_i)\}_{i=1}^N \in (\mathcal{X}, \mathcal{Y})^N$, which is a dataset of N identically distributed random variables $(X_i, Y_i) \sim \mathcal{D}_{(X,Y)}$ called "samples", following the joint probability distribution $\mathcal{D}_{(X,Y)}$. We also generally assume the samples to be independent.

We will assume that the empirical mean of the loss over the dataset is a good approximation of the expected value of the loss on a random sample from the distribution as illustrated in Equation A.3.

$$\mathbb{E}_{(X,Y)}[\mathcal{L}(Y, f(X))] \approx \frac{1}{N} \sum_{i=1}^N \mathcal{L}(Y_i, f(X_i)) \quad (\text{A.3})$$

We will call this empirical mean the "empirical risk" $\hat{\mathcal{R}}_{\mathcal{S}_N}(f)$ which is dependent on the sample dataset, it is defined in Equation A.4.

$$\hat{\mathcal{R}}_{\mathcal{S}_N}(f) := \frac{1}{N} \sum_{i=1}^N \mathcal{L}(Y_i, f(X_i)). \quad (\text{A.4})$$

Approximating the Risk $\mathcal{R}$ with the Empirical Risk $\hat{\mathcal{R}}_{\mathcal{S}_N}$ is very naive. However, one of the advantages is that a procedure to find the function that minimizes risk can be naturally derived from it as shown by the modification of Equation A.2 into Equation A.5.

$$\tilde{f} = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \hat{\mathcal{R}}_{\mathcal{S}_N}(f) = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^N [\mathcal{L}(Y_i, f(X_i))]. \quad (\text{A.5})$$

With this formulation, the problem depends on a sum of data point and not an expectation, it can thus be solved with an optimisation algorithm.

In all generality, we don't always find the minimum of the empirical risk, what we have is the result of an optimisation algorithm on the objective of equation A.5 so the result of our SL algorithm $\hat{f}$ is as defined in Equation A.6.

$$\hat{f} := \operatorname{Optimiser} \left(\underset{f \in \mathcal{H}}{\operatorname{argmin}} \hat{\mathcal{R}}_{\mathcal{S}_N}(f) \right) = \operatorname{Optimiser} \left(\underset{f \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^N [\mathcal{L}(Y_i, f(X_i))] \right) \quad (\text{A.6})$$

A.3 Risk of supervised learning algorithm

Going back to the general case of supervised learning from Equation A.2 and repeated below.

$$\hat{f} = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \mathcal{R}(f) = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \mathbb{E}_{(X,Y)}[\mathcal{L}(Y, f(X))].$$

We also had a sample dataset $\mathcal{S}_N$ that determined through a learning procedure the best function $f \in \mathcal{H}$ as defined in Equation A.6. To define the risk of a SL algorithm, we will explicitly define the training procedure of f using $\mathcal{S}_N$.

We define the "Supervised Learning Algorithm" of equation A.6 as a function $\mathcal{F} : (\mathcal{X}, \mathcal{Y})^N \rightarrow \mathcal{H}$, a function that associate at each possible realisations of the dataset $\{(X_i, Y_i)\}_{i=1}^N \in (\mathcal{X}, \mathcal{Y})^N$ a specific function $f \in \mathcal{H}$.

For sake of clarity if $\mathcal{F}(\mathcal{S}_N) = f$, let's define $\mathcal{F}_{\mathcal{S}_N}(X) := f(X)$. (example of $\mathcal{F}$: training with gd of linear estimator, $\mathcal{F}$ is linear estimator)

Putting this function $\mathcal{F}_{\mathcal{S}_N}$ as argument in the definition of the risk of a function redefines Equation A.3 into Equation A.7.

$$\mathbb{E}_{X,Y}[\mathcal{L}(Y, \mathcal{F}_{\mathcal{S}_N}(X))]. \quad (\text{A.7})$$

However, this formula does not reduce to a scalar because we still have a dependence to $\mathcal{S}_N$ which is a random variable. In other words, the definition of the "Risk of a Supervised Learning Algorithm" is given by Equation A.8.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_{X,Y}[\mathbb{E}_{\mathcal{S}_N}[\mathcal{L}(Y, \mathcal{F}_{\mathcal{S}_N}(X))]]. \quad (\text{A.8})$$

Appendix B

Bias Variance Decomposition

We will do the proof of Geman et al. [1992]

To make the bias-variance decomposition, we assume that the domains are defined as $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{Y} = \mathbb{R}$, and that the loss function is the mean square error $\mathcal{L}(f(X), Y) = \|f(X) - Y\|_2^2$.

Given the definition of the risk defined in Equation A.8, the law of iterated expectations is used to derive Equation B.1.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_X[\mathbb{E}_Y[\mathbb{E}_{\mathcal{S}_N}[\mathcal{L}(Y, \mathcal{F}_{\mathcal{S}_N}(X))|X]]]. \quad (\text{B.1})$$

We can swap the expectation of the training dataset $\mathcal{S}_N$ and the expectations on X and Y assuming this expectation is finite (Fubini's theorem) which provides Equation B.2.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[\mathbb{E}_Y[\mathcal{L}(Y, \mathcal{F}_{\mathcal{S}_N}(X))|X]]]. \quad (\text{B.2})$$

We define the Conditional Expectation Function (CEF) as: $\mathbb{E}[Y|X] := \bar{Y}$. Furthermore, we add and subtract inside the square and we develop as shown in Equation B.3.

$$\begin{aligned} \mathcal{R}(\mathcal{F}) &= \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[\mathbb{E}_Y[((Y - \bar{Y} - \mathcal{F}_{\mathcal{S}_N}(X) + \bar{Y}))^2|X]]] \\ &= \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[\mathbb{E}_Y[(Y - \bar{Y})^2|X] + \mathbb{E}_Y[(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{Y})^2|X] \\ &\quad - 2\mathbb{E}_Y[(Y - \bar{Y})(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{Y})|X]]] \end{aligned} \quad (\text{B.3})$$

The first term does not depend on the model $\mathcal{F}$, it consists on what we call the Irreducible Error (IE) which is defined in Equation B.4.

$$\text{IE} = \mathbb{E}_X[\mathbb{E}_Y[(Y - \bar{Y})^2]] \quad (\text{B.4})$$

The third term is equal to 0 because $\mathbb{E}_Y[(Y - \bar{Y})|X] = \bar{Y} - \bar{Y} = 0$. Finally the remaining term does not depend on Y . In other words, Equation B.3 simplifies to Equation B.5.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[(\mathcal{F}_{\mathcal{S}_N}(X) - \mathbb{E}[Y|X])^2]] + \text{IE} \quad (\text{B.5})$$

Already here we can deduce that the best theoretically possible SL algorithm is the one that for all training datasets finds the CEF.

Now, to derive the bias variance decomposition, we add and subtract $\bar{\mathcal{F}}_{\mathcal{S}_N}(X) = \mathbb{E}_{\mathcal{S}_N}[\mathcal{F}_{\mathcal{S}_N}(X)]$. $\bar{\mathcal{F}}_{\mathcal{S}_N}(X)$ correspond to the expected value of $\mathcal{F}_{\mathcal{S}_N}(X)$ on all possible datasets $\mathcal{S}_N$.

$$\begin{aligned} \mathcal{R}(\mathcal{F}) &= \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{\mathcal{F}}_{\mathcal{S}_N}(X) + \bar{\mathcal{F}}_{\mathcal{S}_N}(X) - \mathbb{E}[Y|X])^2]] + \text{IE} \\ &= \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{\mathcal{F}}_{\mathcal{S}_N}(X))^2 + (\bar{\mathcal{F}}_{\mathcal{S}_N}(X) - \mathbb{E}[Y|X])^2 \\ &\quad + 2(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{\mathcal{F}}_{\mathcal{S}_N}(X))(\bar{\mathcal{F}}_{\mathcal{S}_N}(X) - \mathbb{E}[Y|X])]] + \text{IE} \end{aligned} \quad (\text{B.6})$$

The third term is equal to 0 because $\mathbb{E}_{\mathcal{S}_N}[(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{\mathcal{F}}_{\mathcal{S}_N}(X))] = 0$

We thus have the final bias variance decomposition in Equation B.7 where the first term is the variance, and the second term is the bias.

$$\mathcal{R}(\mathcal{F}) = \mathbb{E}_X[\mathbb{E}_{\mathcal{S}_N}[(\mathcal{F}_{\mathcal{S}_N}(X) - \bar{\mathcal{F}}_{\mathcal{S}_N}(X))^2]] + (\bar{\mathcal{F}}_{\mathcal{S}_N}(X) - \mathbb{E}[Y|X])^2 + \text{IE} \quad (\text{B.7})$$

Appendix C

Bias of the training error

This appendix is not meant as a formal proof, but it is the study of a simple supervised learning algorithm example to disprove a misconception.

One popular way to describe why the training error is not an approximation of the risk is that when we optimise to find the best parameter, this is done using all the samples in the training dataset. So, when we use this best parameter to calculate the training error, this induces a dependence between all the losses and thus we can no longer use the bound provided by Hoeffding's inequality.

While this can be partly the reason, this would lead us to think that if we trained our supervised learning algorithm on a training algorithm comprised of only one sample, the resulting training error would be a good approximation of the risk: and this is not the case. The main reason is similar to the "regression to the mean" phenomena described in Barnett et al. [2005].

C.1 Description of our problem

In our example of supervised learning, the X_i will follow a uniform distribution between 0 and 1 and the $Y_i = X_i + \epsilon_i$ where the ϵ_i will follow a normal with mean $\mu_\epsilon = 0$ and standard deviation $\sigma_\epsilon = 0.1$. We will each time work with training and testing datasets composed of 5 samples of X_i, Y_i .

We will run the supervised learning algorithm 5000 times, each time with a new training dataset and testing dataset and compare the training error with the test error. The training datasets will be labelled S_5^i , $i \in \{1, \dots, 5000\}$ and the testing datasets will be labelled T_5^i , $i \in \{1, \dots, 5000\}$.

All supervised learning algorithm that will be presented are variation in the training of the parametric model $f_\theta = \theta \cdot X$.

C.2 Distribution shift in best of two values for parameter θ

Let's first test analyse a simple training algorithm where we have the choice between two discrete values of θ : $\theta \in \{0.5, 1.7\}$. Our algorithm will consist on choosing the best performing value of θ on the training error as defined in Equation C.1.

$$\mathcal{F}_{S_N}(X) = \hat{\theta}X, \quad \hat{\theta} = \arg \min_{\theta \in \{0.5, 1.7\}} \frac{1}{N} \sum_{(X_i, Y_i) \in S_N} (Y_i - \theta X_i)^2 \quad (\text{C.1})$$

We will distinguish $\hat{\theta}$ when θ is the parameter with minimum risk

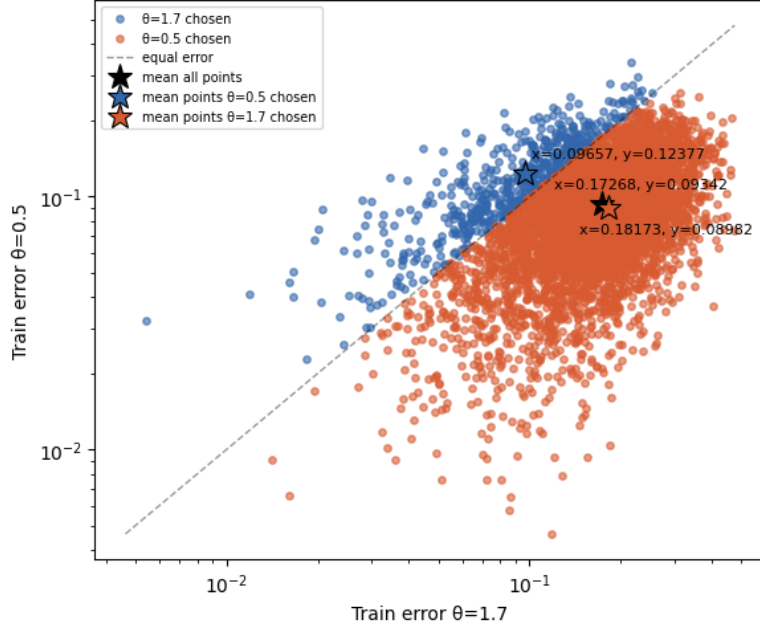


Figure C.1: Training error 5000 training datasets of 5 examples

On Figure C.1 we illustrate each of the 5000 example as a point where the horizontal coordinates are the training errors setting $\theta = 1.7$ and the vertical coordinates are the training errors setting $\theta = 0.5$ (nb: the plot is in logarithmic scale so that the points are more evenly spread out). The black star represent the mean value of all the attempts.

If we take the whole 5000 examples, both the mean training error taking $\theta = 1.7$ and $\theta = 0.5$ (respectively the horizontal and vertical component of each point) can be considered unbiased approximation of the risk because the value of θ is chosen without regard of the value of the risk. The coordinates of the black star are thus

unbiased estimations of the risk (when $\theta = 1.7$ for the x coordinates and $\theta = 0.5$ for the y coordinate). Because one training error is calculated with 5 data-points, the mean of all training errors is equivalent to a training error with $5 * 5000 = 25000$ data points.

To illustrate the reason why training and testing error are not the same, it can be useful to separate the case where $\hat{\theta} = 0.5$ is chosen and the case where 1.7 is chosen.

When $\hat{\theta} = 1.7$, the training error is not the horizontal component of one point taken at random out of the 5000 but we will only consider the points where the training error for $\theta = 1.7$ is lower than the training error for $\theta = 0.5$, thus where the horizontal component is lower than the vertical component, thus all the blue points over/left of the diagonal line.

These points do not follow in all generality the same distribution as the whole 5000 points, in fact because they are all left of the diagonal line, as long as there exist at least one point where $\hat{\theta} = 0.5$ the training error for $\hat{\theta} = 1.7$ will be in average (represented by the blue star) less than the true risk.

Similarly, When $\hat{\theta} = 0.5$, the training error is not the vertical component of one point taken at random out of the 5000 but we will only consider the points where the training error for $\theta = 0.5$ is lower than the training error for $\theta = 1.7$, thus where the vertical component is lower than the horizontal component, thus all the orange points under/right of the diagonal line.

These points do not follow in all generality the same distribution as the whole 5000 points, in fact because they are all under of the diagonal line, as long as there exist at least one point where $\hat{\theta} = 1.7$ the training error for $\hat{\theta} = 0.5$ will be in average (represented by the orange star) less than the true risk.

These distribution shift are represented the histograms in figure C.2. The vertical lines represent the empirical means of the different distributions.

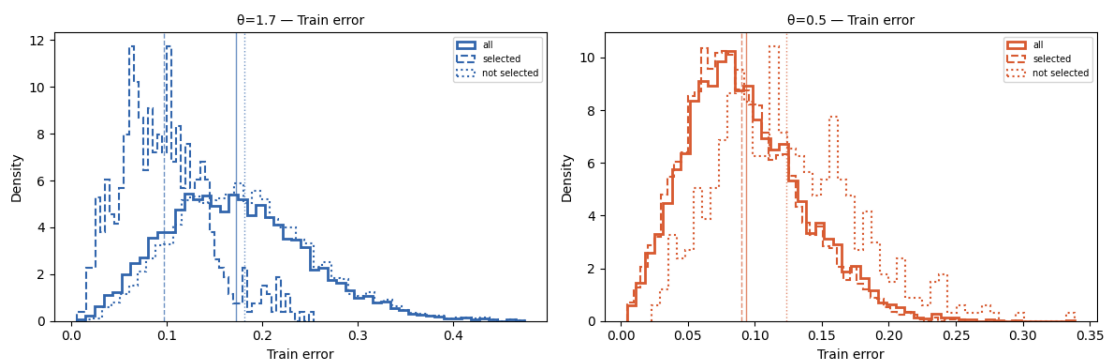


Figure C.2: Distribution shift for the value of the training error

$\theta = 1.7$ is selected around 10.62% of the time whereas for the rest of the time $\theta = 0.5$ has a lower training error. Knowing this information we can calculate the mean training error with the x coordinate of the blue star and the y coordinate of the orange star: $\frac{10.62}{100} * 0.9657 + \frac{89.38}{100} * 0.08982 = 0.090534$.

But to find the Risk of the training algorithm, you have to have an unbiased estimation of the training error for both coordinates, and that should be done also considering vertical training errors with $\theta = 1.7$ in examples where $\hat{\theta} = 0.5$ and training errors with $\theta = 0.5$ in examples where $\hat{\theta} = 1.7$. Using the mean training errors with all 5000 training datasets and both values for θ (the coordinates of the black star) we can estimate the risk of this learning algorithm to be equal to $\mathcal{R}(\mathcal{F}_{\text{dummy}}) \approx \frac{10.62}{100} * 0.17268 + \frac{89.38}{100} * 0.09342 = 0.10184$ which is nearly equal to the mean of the testing errors (setting $\theta = \hat{\theta}$) which are $\mathcal{R}(\mathcal{F}_{\text{dummy}}) \approx 0.10153$. These two are thus our best approximation for the risk of our training algorithm (technically the second one is the only unbiased approximation because we did use the training datasets to find the percentage distributions of the values of $\hat{\theta}$).

C.2.1 Change in size of the training dataset

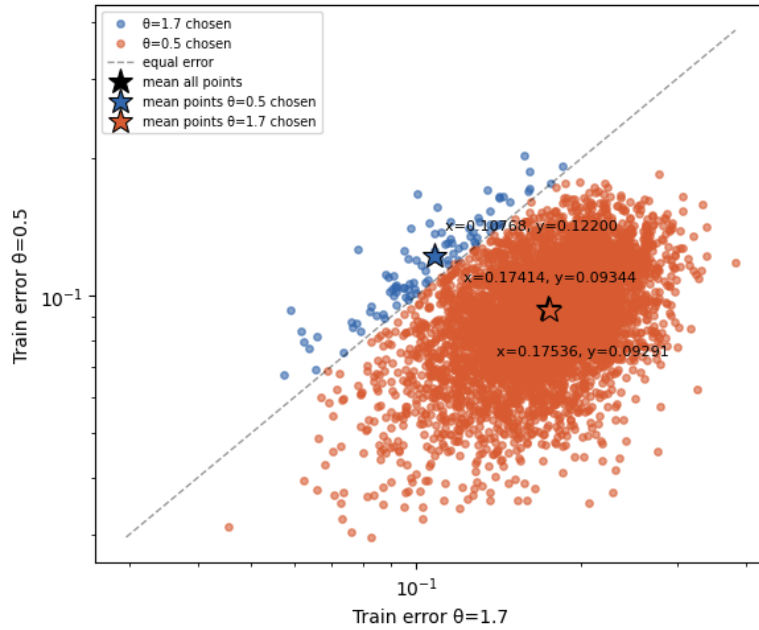


Figure C.3: Training error 5000 training datasets of 15 examples

In Figure C.2.1 we did the same experiment but changed the size of the training dataset from 5 data-points to 15 data-points. Unsurprisingly because the size of the

training set is bigger the points will be more centred on the real risk (approximated by the black star). The training will more often pick the best parameter $\hat{\theta} = 0.5$ (98.2% of the time) but because he sometimes choose $\hat{\theta} = 1.7$ this introduces a bias in the training error of the model.

more choices for the value of $\hat{\theta}$

Instead of choosing between 2 values of theta we could choose between 3 values of theta. If the try to map the training errors, we would have a similar graph as Figure C.1 but in 3 dimensions, with all the space divided in 3 areas, and tie the probability to be in one specific area out of 3 (which is the probability of choosing one specific value) will be probably less than in one specific area out of 2, but the distribution of length in any of the 3 direction (the value of the training error setting the value of θ) will not necessarily be the same if we take the points in all the 3 areas (an unbiased estimation of the risk) or if we take only one of the 3 areas (the training error when a particular $\hat{\theta}$ is set).

When we have the choice between an infinite amount of values (for example: $\theta \in \mathbb{R}$, the probability of taking one specific value for $\hat{\theta}$ will be 0 and we will handle probability densities, but the distribution shift will still exist for every specific value of θ .

Appendix D

Generalisation framework

D.1 Theory behind generalisation

Most papers talking about generalisation try to bound the value of the risk in probability using the training error and some "capacity" term that depends on the supervised learning algorithm, often dependent on the hypothesis class.

These leads to bounds like Equation D.1

$$\mathcal{R}(\mathcal{F}(S_N)) \leq \hat{\mathcal{R}}_{S_N}(\mathcal{F}(S_N)) + \text{Capacity}_{\mathcal{F}}(\epsilon) \quad \text{with probability } p \geq 1 - \epsilon \quad (\text{D.1})$$

You can easily write these kind of bounds in the form of equation D.2.

$$\Pr \left(\mathcal{R}(\mathcal{F}(S_N)) - \hat{\mathcal{R}}_{S_N}(\mathcal{F}(S_N)) > \text{Capacity}_{\mathcal{F}} \right) < \epsilon \quad (\text{D.2})$$

The main intuition behind these bounds flows from the observation that if there is only one function in the hypothesis class, then the training error would be a unbiased estimator of the risk and thus we can use Hoeffding's inequality to come up with this bound.

For our algorithm in Equation C.1, the bounds can be written like Equation D.5

$$\Pr \left(\mathcal{R}(f_{\theta}) - \hat{\mathcal{R}}_{S_N}(f_{\theta}) \geq \delta \right) \leq e^{-\frac{2N\delta^2}{C^2}} \quad \theta \in \{0.5, 1.7\} \quad (\text{D.3})$$

We can use the union bound so that we guarantee the inequality for both hypothesis during training.

$$\Pr \left(\left(\mathcal{R}(f_{0.5}) - \hat{\mathcal{R}}_{S_N}(f_{0.5}) \geq \delta \right) \cup \left(\mathcal{R}(f_{1.7}) - \hat{\mathcal{R}}_{S_N}(f_{1.7}) \geq \delta \right) \right) \leq 2e^{-\frac{2N\delta^2}{C^2}} \quad (\text{D.4})$$

So when we select the best out of both parameter, this inequality still holds.

$$\Pr \left(\min_{\hat{\theta} \in \{0.5, 1.7\}} \mathcal{R}(f_{\hat{\theta}}) - \hat{\mathcal{R}}_{S_N}(f_{\hat{\theta}}) \geq \delta \right) \leq 2e^{-\frac{2N\delta^2}{C^2}} \quad (\text{D.5})$$

The capacity in this case is thus dependent on the number of hypothesis in the hypothesis space. For infinite sized hypothesis classes, generalizations of this procedure exist using properties of the hypothesis classes. We thus try to define a "capacity measure" of the class in such a way to have the best possible bounds.

The reason why the training error is not a good estimation of the risk while the testing error is is due to the distribution shift when training illustrated in Figure C.1, so if we understand this distribution shift better we can come up with better bounds.

D.2 Conclusion

In this chapter, we had a brief introduction into the theory of generalisation.

While it is the most used framework in the literature, the author does better understand the problem through the lens of the bias-variance trade-off, which will be the framework used in the rest of the document

Nevertheless the study of the training error-risk gap is so widely used that it seemed appropriate to indicate it with a small chapter, whose depth is proportional to the current understanding of the author

Some keywords to research on this framework are "pac learning", "vc-dimension", "class capacity", "capacity control" and "no free lunch theorem"

Appendix E

Gradient flow OLS

E.1 Statement of the problem

We want to solve the gradient flow on the problem given by the risk equation:

$$\hat{R}(\theta) = \frac{1}{2} \|X\theta - y\|^2. \quad (\text{E.1})$$

Where $X \in \mathbb{R}^{N \times d}$ is a matrix of rank $r \leq N, r \leq d$ and $y \in \mathbb{R}^N$ a vector.

This implies a gradient flow for the parameter vector $\theta \in \mathbb{R}^d$ given by :

$$\dot{\theta} := \frac{d\theta}{dt} = -\nabla_{\theta} \hat{R}(\theta) = -X^{\top}(X\theta - y) = -X^{\top}X\theta + X^{\top}y. \quad (\text{E.2})$$

The matrix X can also be decomposed into his reduced Singular Value Decomposition of X given by:

$$X = Q_r \Sigma_r V_r^T \quad (\text{E.3})$$

- $Q_r \in \mathbb{R}^{N \times r}$ has orthonormal columns, in particular $Q_r^T Q_r = I_r$ where I_r is the identity matrix.
- $\Sigma_r \in \mathbb{R}^{r \times r}$ is diagonal with positive singular values.
- $V_r \in \mathbb{R}^{d \times r}$ has orthonormal columns, in particular $V_r^T V_r = I_r$ where I_r is the identity matrix.
- $r = \text{rank}(X)$.

The author uses $Q\Sigma V^T$ as singular value decomposition instead of the more traditional $U\Sigma V^T$ for the singular value decomposition and argue that this notation

is more efficient because when doing handwriting calculation U and V happens to be very similar letters that can easily be mistaken for one another.

E.2 Decomposition into column space and kernel

Let us decompose θ into components along the column space of $X^\top$ and the kernel of X :

$$\theta = \theta_{\parallel} + \theta_{\perp}, \quad \theta_{\parallel} \in \text{col}(X^\top), \quad \theta_{\perp} \in \text{ker}(X). \quad (\text{E.4})$$

To that aim, we define the projection matrices:

$$P := P_{\text{col}(X^\top)} = X^\top (XX^\top)^\dagger X, \quad M := P_{\text{ker}(X)} = I - P, \quad (\text{E.5})$$

Where $(XX^\top)^\dagger := Q_r \Sigma_r^{-1} Q_r^\top$ is the Penrose pseudo-inverse of $(XX^\top)$, this allows us to compute the projection matrix even if the rank r is strictly less than N ($X^\top$ is not full column rank).

You can verify using the reduced singular-value decomposition (SVD) of $XX^\top$ that

$$PX^\top = X^\top (XX^\top)^\dagger XX^\top = X^\top, \quad XP = XX^\top (XX^\top)^\dagger X = X \quad (\text{E.6})$$

With these matrices we can have a computable formula for both components:

$$\theta_{\parallel} := P\theta, \quad \theta_{\perp} := M\theta \quad (\text{E.7})$$

.

E.3 Gradient flow in the subspaces

Projecting the ODE onto these subspaces gives the following formula's for the gradient flow:

$$\dot{\theta}_{\parallel} = -X^\top X \theta_{\parallel} + X^\top y, \quad \dot{\theta}_{\perp} = 0 \quad (\text{E.8})$$

proof:

$$\begin{aligned}
\dot{\theta}_{\parallel} &= P\dot{\theta} \\
\dot{\theta}_{\parallel} &= -PX^{\top}X\theta + PX^{\top}y \\
\dot{\theta}_{\parallel} &= -X^{\top}X\theta + X^{\top}y \\
\dot{\theta}_{\parallel} &= -X^{\top}X(P+M)\theta + X^{\top}y \\
\dot{\theta}_{\parallel} &= -X^{\top}XP\theta + X^{\top}y \\
\dot{\theta}_{\parallel} &= -X^{\top}X\theta_{\parallel} + X^{\top}y
\end{aligned}$$

$$\begin{aligned}
\dot{\theta}_{\perp} &= M\dot{\theta} \\
\dot{\theta}_{\perp} &= -MX^{\top}X\theta + MX^{\top}y \\
\dot{\theta}_{\perp} &= 0
\end{aligned}$$

E.4 Solving the gradient flow in the parallel component

Using a well known result from control theory, we know the vector linear ODE for $\theta_{\parallel}$:

$$\dot{\theta}_{\parallel} = -X^{\top}X\theta_{\parallel} + X^{\top}y \quad (\text{E.9})$$

admits a closed-form solution using the matrix exponential:

$$\theta_{\parallel}(t) = e^{-X^{\top}Xt}\theta_{\parallel}(0) + \int_0^t e^{-X^{\top}X(t-\tau)}X^{\top}y d\tau. \quad (\text{E.10})$$

Substituting SVD into the solution:

$$X^{\top}X = V_r\Sigma_r^2V_r^{\top}, \quad X^{\top}y = V_r\Sigma_rU_r^{\top}y. \quad (\text{E.11})$$

$$\theta_{\parallel}(t) = e^{-V_r\Sigma_r^2V_r^{\top}t}\theta_{\parallel}(0) + \int_0^t e^{-V_r\Sigma_r^2V_r^{\top}(t-\tau)}V_r\Sigma_rU_r^{\top}y d\tau. \quad (\text{E.12})$$

E.5 Properties of matrix exponential

To progress further, let's review some properties of matrix exponentials.

The definition of a matrix exponential is the following:

$$e^{At} := \sum_{k=0}^{\infty} \frac{1}{k!} (At)^k \quad (\text{E.13})$$

If A is diagonalisable ($A = B\Lambda B^{-1}$ with Λ diagonal) we have:

$$e^{At} = B e^{\Lambda t} B^{-1} \quad (\text{E.14})$$

proof:

$$\begin{aligned} e^{At} &:= \sum_{k=0}^{\infty} \frac{1}{k!} (At)^k \\ &= \sum_{k=0}^{\infty} \frac{1}{k!} (B\Lambda B^{-1}t)^k \\ &= \sum_{k=0}^{\infty} \frac{1}{k!} (B\Lambda t B^{-1})^k \\ &= B \left(\sum_{k=0}^{\infty} \frac{1}{k!} (\Lambda t)^k \right) B^{-1} = B e^{\Lambda t} B^{-1} \end{aligned} \quad (\text{E.15})$$

Computing the exponential of a diagonal matrix reduces to computing the exponential of each diagonal component:

$$e^{\Lambda t} = \begin{bmatrix} e^{\lambda_1 t} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & e^{\lambda_n t} \end{bmatrix} \quad (\text{E.16})$$

proof:

$$\begin{aligned} e^{\Lambda t} &= \sum_{k=0}^{\infty} \frac{1}{k!} (\Lambda t)^k \\ &= \sum_{k=0}^{\infty} \frac{1}{k!} \begin{bmatrix} \lambda_1 t & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n t \end{bmatrix}^k \\ &= \begin{bmatrix} \sum_{k=0}^{\infty} \frac{1}{k!} (\lambda_1 t)^k & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sum_{k=0}^{\infty} \frac{1}{k!} (\lambda_n t)^k \end{bmatrix} \\ &= \begin{bmatrix} e^{\lambda_1 t} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & e^{\lambda_n t} \end{bmatrix} \end{aligned} \quad (\text{E.17})$$

E.6 Computing the integral term

This part of the proof was done with the help of chat gpt.

Let's first compute the integral term:

$$\begin{aligned}
\int_0^t e^{-V_r \Sigma_r^2 V_r^\top (t-\tau)} V_r \Sigma_r U_r^\top y \, d\tau &= \int_0^t e^{-V_r \Sigma_r^2 (t-\tau) V_r^\top} V_r \Sigma_r U_r^\top y \, d\tau \\
&= \int_0^t V_r e^{-\Sigma_r^2 (t-\tau)} V_r^\top V_r \Sigma_r U_r^\top y \, d\tau \\
&= V_r \left(\int_0^t e^{-\Sigma_r^2 (t-\tau)} \Sigma_r \, d\tau \right) U_r^\top y.
\end{aligned} \tag{E.18}$$

Performing the change of variable $s = t - \tau$ (so $ds = -d\tau$) gives:

$$V_r \left(\int_0^t e^{-\Sigma_r^2 (t-\tau)} \Sigma_r \, d\tau \right) U_r^\top y = V_r \left(\int_t^0 e^{-\Sigma_r^2 s} \Sigma_r (-ds) \right) U_r^\top y = V_r \left(\int_0^t e^{-\Sigma_r^2 s} \Sigma_r \, ds \right) U_r^\top y \tag{E.19}$$

Since $e^{-\Sigma_r^2 s} \Sigma_r$ is diagonal, We can decompose the matrix element-wise:

$$\int_0^t e^{-\Sigma_r^2 s} \Sigma_r \, ds = \begin{bmatrix} \int_0^t \sigma_1 e^{-\sigma_1^2 s} \, ds & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \int_0^t e^{-\sigma_r \sigma_r^2 s} \, ds \end{bmatrix} = \begin{bmatrix} [-e^{-\sigma_1^2 s}]_0^t & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & [-e^{-\sigma_r^2 s}]_0^t \end{bmatrix} = I - e^{-\Sigma_r^2 t}. \tag{E.20}$$

Hence the integral term simplifies to:

$$\int_0^t e^{-X^\top X (t-\tau)} X^\top y \, d\tau = V_r (I - e^{-\Sigma_r^2 t}) \Sigma_r^{-1} U_r^\top y. \tag{E.21}$$

E.7 Final solution

Combining both terms, we obtain the explicit solution for the gradient flow in the parallel component:

$$\theta_{\parallel}(t) = V_r e^{-\Sigma_r^2 t} V_r^\top \theta_{\parallel}(0) + V_r (I - e^{-\Sigma_r^2 t}) \Sigma_r^{-1} U_r^\top y. \tag{E.22}$$

As $t \rightarrow \infty$, the exponential term vanishes, and we obtain the minimum-norm solution:

$$\theta_{\parallel}(\infty) = V_r \Sigma_r^{-1} U_r^\top y = X^\dagger y, \tag{E.23}$$

where $X^\dagger$ is the Moore-Penrose pseudo-inverse of X .

Therefore, the gradient flow converges to a solution that lies in the column space of $X^\top$ and has minimal norm in the kernel of X :

$$\theta(\infty) = \theta_{\parallel}(\infty) + \theta_{\perp}(0) = X^\dagger y + \theta_{\perp}(0). \quad (\text{E.24})$$

This completes the derivation of the gradient flow dynamics for OLS.

Appendix F

Derivation of the gradient of linear Neural Networks

F.1 Proof part 1: useful results

To make the proof, we will use the "bracket notation" for matrices $X \in \mathbb{R}^{m \times n}$ as defined in Equation F.1.

$$X = \begin{bmatrix} X_{11} & \cdots & X_{1,n} \\ \vdots & \ddots & \vdots \\ X_{m1} & \cdots & X_{m,n} \end{bmatrix} = [X_{i,j}]_{i,j=1}^{m,n} \quad (\text{F.1})$$

Before calculating the gradient, we will develop some results needed for the proof.

For any vectors $a \in \mathbb{R}^m$, $b \in \mathbb{R}^n$ and any matrices $X \in \mathbb{R}^{m \times n}$ and $H \in \mathbb{R}^{m \times m}$, the results in Equations F.2 and F.3 hold true.

$$\nabla_X (a^T \cdot X \cdot b) = \left[\frac{\partial}{\partial X_{i,j}} \left(\sum_{k=1}^m \sum_{l=1}^n a_k b_l X_{k,l} \right) \right]_{i,j=1}^{m,n} = [a_i b_j]_{i,j=1}^{m,n} = a \cdot b^T \quad (\text{F.2})$$

$$\begin{aligned}
\nabla_X (b^T \cdot X^T \cdot H \cdot X \cdot b) &= \nabla_X \left(\left(\left[\sum_{l=1}^n X_{k,l} b_l \right]_{k=1}^m \right)^T \cdot H \cdot \left[\sum_{q=1}^n X_{p,q} b_q \right]_{p=1}^m \right) \\
&= \left[\frac{\partial}{\partial X_{i,j}} \left(\sum_{k=1}^m \sum_{l=1}^n \sum_{p=1}^m \sum_{q=1}^n X_{k,l} b_l H_{k,p} X_{p,q} b_q \right) \right]_{i,j=1}^{m,n} \\
&= \left[\left(\sum_{p=1}^m \sum_{q=1}^n b_j H_{i,p} X_{p,q} b_q \right) + \left(\sum_{k=1}^m \sum_{l=1}^n b_j H_{k,i} X_{k,l} b_l \right) \right]_{i,j=1}^{m,n} \quad (\text{F.3}) \\
&= \left[b_j \left(\sum_{p=1}^m H_{i,p} \left(\sum_{q=1}^n X_{p,q} b_q \right) + \sum_{k=1}^m H_{k,i} \left(\sum_{l=1}^n X_{k,l} b_l \right) \right) \right]_{i,j=1}^{m,n} \\
&= \left(\left[\sum_{p=1}^m H_{i,p} \left(\sum_{q=1}^n X_{p,q} b_q \right) + \sum_{k=1}^m H_{i,k}^T \left(\sum_{l=1}^n X_{k,l} b_l \right) \right]_{i=1}^m \right) \cdot b^T \\
&= (H + H^T) \cdot X \cdot b \cdot b^T
\end{aligned}$$

F.2 Proof part 2: derivation of the formula of the gradient

Now that we have developed all the tools necessary for the proof, let's begin the derivation:

$$\begin{aligned}
\nabla_{\Theta^j} (R(\Theta)) &= \nabla_{\Theta^j} \left(\|Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L\|_2^2 \right) \\
&= \nabla_{\Theta^j} \left((Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L)^T \cdot (Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) \right) \\
&= \nabla_{\Theta^j} \left(-2 \cdot (Y^T \cdot X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) + ((\Theta^L)^T \dots (\Theta^2)^T \cdot (\Theta^1)^T X^T X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) \right)
\end{aligned}$$

Let's now derive the 2 terms independently.

Using Equation F.2, the first term becomes:

$$\begin{aligned}
\nabla_{\Theta^j} - 2 \cdot (Y^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L) &= \nabla_{\Theta^j} (a^T \cdot \Theta^j \cdot b) \\
&= a \cdot b^T = -2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot Y \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T) \\
a^T &:= -2 \cdot Y^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \\
b &:= \Theta^{j+1} \dots \Theta^L
\end{aligned}$$

Using Equation F.3, the second equation becomes:

$$\begin{aligned}
& \nabla_{\Theta^j} \left(((\Theta^L)^T \dots (\Theta^{j+1})^T \cdot (\Theta^j)^T \cdot (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L) \right) \\
&= \nabla_{\Theta^j} (b^T \cdot (\Theta^j)^T \cdot H \cdot \Theta^j \cdot b) = (H + H^T) \cdot \Theta^j \cdot b \cdot b^T = 2 \cdot H \cdot \Theta^j \cdot b \cdot b^T \\
&= 2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T) \\
&= 2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot \text{linear}_{\Theta}(X) \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T)
\end{aligned}$$

$$H := (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1}$$

$$b := \Theta^{j+1} \dots \Theta^L$$

NB : H is a symmetric matrix

Now adding terms we have the final equation of the gradient as defined in Equation 5.4.

$$\begin{aligned}
\nabla_{\Theta^j} (R(\Theta)) &= -2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot Y \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T) \\
&\quad + 2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot \text{linear}_{\Theta}(X) \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T) \\
&= 2 \cdot ((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot (\text{linear}_{\Theta}(X) - Y) \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T)
\end{aligned}$$

Appendix G

Derivation of the gradient of affine Neural Networks

G.1 Proof part 1: useful results

First we need to adapt our matrix analysis results for incorporated gradient of product affine matrices.

Assuming $X = \begin{bmatrix} c & \tilde{X} \end{bmatrix} \in \mathbb{R}^{(m) \times (n+1)}$ is an affine matrix, $a \in \mathbb{R}^m$ and $b \in \mathbb{R}^{n+1}$ are some vectors, we have:

$$\begin{aligned} \nabla_X^* (a^T \cdot X \cdot b) &= \nabla_X^* \left(a^T \cdot \begin{bmatrix} c & \tilde{X} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_{1,\dots,n} \end{bmatrix} \right) = \nabla_X^* (a^T (c \cdot b_0 + \tilde{X} \cdot b_{1,\dots,n})) \\ &= \text{Inc} \left(\nabla_{\tilde{X}} (a^T \cdot c \cdot b_0 + a^T \cdot \tilde{X} \cdot b_{1,\dots,n}) \right) = \text{Inc} (a \cdot b_{1,\dots,n}^T) \end{aligned} \quad (\text{G.1})$$

Where b_0 is the first line of b and $b_{1,\dots,n}$ is the vector of the rest of the lines b .

For the second result, Assuming we have $H \in \mathbb{R}^{m \times m}$:

$$\begin{aligned} \nabla_X^* (b^T \cdot X^T \cdot H \cdot X \cdot b) &= \nabla_X^* \left(\begin{bmatrix} b_0 & b_{1,\dots,n}^T \end{bmatrix} \cdot \begin{bmatrix} c^T \\ \tilde{X}^T \end{bmatrix} \cdot H \cdot \begin{bmatrix} c & \tilde{X} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_{1,\dots,n} \end{bmatrix} \right) \\ &= \nabla_X^* \left((b_0 \cdot c^T + b_{1,\dots,n}^T \cdot \tilde{X}^T) \cdot H \cdot (c \cdot b_0 + \tilde{X} \cdot b_{1,\dots,n}) \right) \\ &= \nabla_X^* \left(b_0 \cdot c^T \cdot H \cdot c \cdot b_0 + b_{1,\dots,n}^T \cdot \tilde{X}^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} \right. \\ &\quad \left. + b_0 \cdot c^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} + b_{1,\dots,n}^T \cdot \tilde{X}^T \cdot H \cdot c \cdot b_0 \right) \\ &= \text{Inc} \left(\nabla_{\tilde{X}} (b_{1,\dots,n}^T \cdot \tilde{X}^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} + b_0 \cdot c^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} + b_{1,\dots,n}^T \cdot \tilde{X}^T \cdot H \cdot c \cdot b_0) \right) \\ &= \text{Inc} \left(\nabla_{\tilde{X}} (b_{1,\dots,n}^T \cdot \tilde{X}^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} + b_0 \cdot c^T \cdot H \cdot \tilde{X} \cdot b_{1,\dots,n} + b_0 \cdot c^T \cdot H^T \cdot \tilde{X} \cdot b_{1,\dots,n}) \right) \end{aligned} \quad (\text{G.2})$$

We have 3 terms: for the first one we can use the result of Equation F.2 and for the second and third the result of Equation F.3.

$$\begin{aligned}
&= \text{Inc} \left((H + H^T) \cdot \tilde{X} \cdot b_{1,\dots,n} \cdot b_{1,\dots,n}^T + b_0 \cdot c \cdot H^T \cdot b_{1,\dots,n}^T + b_0 \cdot c \cdot H \cdot b_{1,\dots,n}^T \right) \\
&= \text{Inc} \left((H + H^T) \cdot (\tilde{X} \cdot b_{1,\dots,n} + c \cdot b_0) \cdot b_{1,\dots,n}^T \right) \\
&= \text{Inc} \left((H + H^T) \cdot \begin{bmatrix} c & \tilde{X} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_{1,\dots,n} \end{bmatrix} \cdot b_{1,\dots,n}^T \right) \\
&= \text{Inc} \left((H + H^T) \cdot (X \cdot b) \cdot b_{1,\dots,n}^T \right)
\end{aligned} \tag{G.3}$$

The fact we need to remove the first line of the vector b in these formulas motivates us to see how we can remove the first line of a product of matrices.

Let's analyse the product of affine matrices :

$$\Theta = \Theta^1 \cdot \Theta^2 \dots \Theta^n = \begin{bmatrix} 1 & \tilde{\Theta}_b^1 \\ 0^1 & \tilde{\Theta}_s^1 \end{bmatrix} \cdot \Theta^2 \dots \Theta^n = \begin{bmatrix} \begin{bmatrix} 1 & \tilde{\Theta}_b^1 \end{bmatrix} \cdot \Theta^2 \dots \Theta^n \\ \begin{bmatrix} 0^1 & \tilde{\Theta}_s^1 \end{bmatrix} \cdot \Theta^2 \dots \Theta^n \end{bmatrix}$$

So if we want to remove the first line the equation becomes:

$$\begin{aligned}
&\begin{bmatrix} 0^1 & \tilde{\Theta}_s^1 \end{bmatrix} \cdot \Theta^2 \dots \Theta^n = \begin{bmatrix} 0^1 & \tilde{\Theta}_s^1 \end{bmatrix} \cdot \begin{bmatrix} 1 & \tilde{\Theta}_b^2 \\ 0^2 & \tilde{\Theta}_s^2 \end{bmatrix} \cdot \Theta^3 \dots \Theta^n \\
&= (0^1 \cdot \begin{bmatrix} 1 & \tilde{\Theta}_b^2 \end{bmatrix} + \tilde{\Theta}_s^1 \cdot \begin{bmatrix} 0^2 & \tilde{\Theta}_s^2 \end{bmatrix}) \cdot \Theta^3 \dots \Theta^n = \tilde{\Theta}_s^1 \cdot (\begin{bmatrix} 0^2 & \tilde{\Theta}_s^2 \end{bmatrix} \cdot \Theta^3 \dots \Theta^n) \\
&= \tilde{\Theta}_s^1 \cdot \tilde{\Theta}_s^2 \dots \tilde{\Theta}_s^n
\end{aligned}$$

For the last equality we assume that the last matrix is the matrix of the last layer of an affine linear NN, and thus $\Theta^n = \begin{bmatrix} \Theta_b^n \\ \Theta_s^n \end{bmatrix}$

G.2 Proof part 2: derivation of the formula of the gradient

Now that we have all the required tools, lets derive the formula of the gradient:

$$\begin{aligned}
\nabla_{\Theta^j}^* (R(\Theta)) &= \nabla_{\Theta^j}^* \left(\|Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L\|_2^2 \right) \\
&= \nabla_{\Theta^j}^* \left((Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L)^T \cdot (Y - X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) \right) \\
&= \nabla_{\Theta^j}^* \left(-2 \cdot (Y^T \cdot X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) + ((\Theta^L)^T \dots (\Theta^2)^T \cdot (\Theta^1)^T X^T X \cdot \Theta^1 \cdot \Theta^2 \dots \Theta^L) \right)
\end{aligned}$$

Like in the linear case, we have the 2 different terms that can be simplified using Equations :

$$\begin{aligned}
\nabla_{\Theta^j}^* \left(-2 \cdot Y^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L \right) &= \nabla_{\Theta^j} \left(a^T \cdot \Theta^j \cdot b \right) \\
&= a \cdot b_{1,\dots,n}^T = -2 \cdot \left((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot Y \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T \right) \\
a^T &:= -2 \cdot Y^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \\
b &:= \Theta^{j+1} \dots \Theta^L \\
b_{1,\dots,n} &= \tilde{\Theta}_s^{j+1} \dots \tilde{\Theta}_s^L
\end{aligned}$$

The second equation becomes :

$$\begin{aligned}
\nabla_{\Theta^j}^* \left(\left((\Theta^L)^T \dots (\Theta^{j+1})^T \cdot (\Theta^j)^T \cdot (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L \right) \right) \\
&= \nabla_{\Theta^j}^* \left(b^T \cdot (\Theta^j)^T \cdot H \cdot \Theta^j \cdot b \right) = (H + H^T) \cdot \Theta^j \cdot b \cdot b_{1,\dots,n}^T = 2 \cdot H \cdot \Theta^j \cdot b \cdot b_{1,\dots,n}^T \\
&= 2 \cdot \left((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \cdot \Theta^j \cdot \Theta^{j+1} \dots \Theta^L \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T \right) \\
&= 2 \cdot \left((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot \text{linear}_{\Theta}(X) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T \right)
\end{aligned}$$

$$\begin{aligned}
H &= (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot X \cdot \Theta^1 \dots \Theta^{j-1} \\
b &:= \Theta^{j+1} \dots \Theta^L \\
b_{1,\dots,n} &= \tilde{\Theta}_s^{j+1} \dots \tilde{\Theta}_s^L
\end{aligned}$$

Now adding both terms we have the final equation of the gradient:

$$\begin{aligned}
\nabla_{\Theta^j}^* (R(\Theta)) &= \\
2 \cdot \text{Inc} \left((\Theta^{j-1})^T \dots (\Theta^1)^T \cdot X^T \cdot (\text{affine}_{\Theta}(X) - Y) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T \right) &\quad (\text{G.4})
\end{aligned}$$

Appendix H

Proof of lemma 1 for linear Neural Networks

In this section we will proceed to prove lemma 1, this is done because we will see that this result fails to generalise for affine NNs.

NB: this proof is taken from Arora et al. [2018].

Using the product rule for time derivatives we have:

$$\begin{aligned}\frac{d}{dt} \left(\Theta^{j+1} (\Theta^{j+1})^T - (\Theta^j)^T \Theta^j \right) &= G_1 + G_2 - G_3 - G_4 \\ G_1 &= \frac{d}{dt} \left(\Theta^{j+1} \right) (\Theta^{j+1})^T \\ G_2 &= \Theta^{j+1} \frac{d}{dt} \left((\Theta^{j+1})^T \right) \\ G_3 &= \frac{d}{dt} \left((\Theta^j)^T \right) \Theta^j \\ G_4 &= (\Theta^j)^T \frac{d}{dt} \left(\Theta^j \right)\end{aligned}$$

You can notice that $G_1 = G_2^T$ and $G_4 = G_3^T$.

Now, replacing the time derivative with the gradient flow we have for G_1 :

$$G_1 = \left(2 \cdot (\Theta^j)^T \dots (\Theta^1)^T \cdot \nabla_{\hat{\theta}} R(\hat{\theta}) \cdot (\Theta^L)^T \dots (\Theta^{j+2})^T \right) \Theta^{j+1}$$

On the other way, G_4 :

$$G_4 = \left(\Theta^j \right)^T \left(2 \cdot (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot \nabla_{\hat{\theta}} R(\hat{\theta}) \cdot (\Theta^L)^T \dots (\Theta^{j+1})^T \right)$$

Hence, we can see $G_1 - G_4 = 0$, and this also implies $G_2 - G_3 = 0$, thus proving the lemma.

Appendix I

Lemma 1 disproven for affine Neural Networks

The beginning of the proof stays the same :

$$\begin{aligned}\frac{d}{dt} \left(\Theta^{j+1} (\Theta^{j+1})^T - (\Theta^j)^T \Theta^j \right) &= G_1 + G_2 - G_3 - G_4 \\ G_1 &= \frac{d}{dt} \left(\Theta^{j+1} \right) (\Theta^{j+1})^T \\ G_2 &= \Theta^{j+1} \frac{d}{dt} \left((\Theta^{j+1})^T \right) \\ G_3 &= \frac{d}{dt} \left((\Theta^j)^T \right) \Theta^j \\ G_4 &= (\Theta^j)^T \frac{d}{dt} \left(\Theta^j \right)\end{aligned}$$

We still have $G_1 = G_2^T$ and $G_4 = G_3^T$

But now when we replace the time derivative with the incorporated gradient we have:

$$\begin{aligned}G_1 &= 2 \cdot \text{Inc} \left((\Theta^j)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y) \right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+2})^T \right) \cdot (\Theta^{j+1})^T \\ &= 2 \cdot \begin{bmatrix} 0 & (\Theta^j)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y) \right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+2})^T \end{bmatrix} \cdot \begin{bmatrix} 1 & 0^T \\ (\tilde{\Theta}_b^{j+1})^T & (\tilde{\Theta}_s^{j+1})^T \end{bmatrix} \\ &= 2 \cdot (\Theta^j)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y) \right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+2})^T \cdot \begin{bmatrix} (\tilde{\Theta}_b^{j+1})^T & (\tilde{\Theta}_s^{j+1})^T \end{bmatrix}\end{aligned}$$

And now G_4 becomes :

$$\begin{aligned}
G_4 &= \left(\Theta^j\right)^T \text{Inc} \left(2 \cdot (\Theta^{j-1})^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y)\right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T\right) \\
&= 2 \cdot \text{Inc} \left(\left(\Theta^j\right)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y)\right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+1})^T\right) \\
&= 2 \cdot \left(\Theta^j\right)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y)\right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+2})^T \dots \begin{bmatrix} 0 & (\tilde{\Theta}_s^{j+1})^T \end{bmatrix}
\end{aligned}$$

Therefore $G_1 - G_4$ is not equal to 0 in all cases any-more:

$$G_1 - G_4 = 2 \cdot (\Theta^j)^T \dots (\Theta^1)^T \cdot \left(X^T \cdot (\text{affine}_\Theta(X) - Y)\right) \cdot (\tilde{\Theta}_s^L)^T \dots (\tilde{\Theta}_s^{j+2})^T \cdot \begin{bmatrix} (\tilde{\Theta}_b^{j+1})^T & 0 \end{bmatrix} \quad (\text{I.1})$$

Let's define $\Delta := G_1 - G_4$, we have thus and we find Equation 7.9

Appendix J

Proof of optimality

The proof will first find an infimum for the sum of square coefficient, then prove show that assuming weight matrices alignment lemma the sum of square coefficient differ from this infimum at most by an order of $\epsilon^{\frac{1}{L}}$

This order can potentially be reduced by reduced by a clevere use of property (iii) of this lemma: this is left for future work

For all linear neural network such that:

$$\prod_{i=1}^L \Theta^i = \Theta^P \quad (\text{J.1})$$

We have:

$$\sum_{i=1}^L \|\Theta^i\|_F^2 \geq L \cdot (\|\Theta^P\|_2^2)^{\frac{1}{L}} := C \quad (\text{J.2})$$

proof:

$$\begin{aligned} \sum_{i=1}^L \|\Theta^i\|_F^2 &= L \cdot \left(\frac{1}{L} \sum_{i=1}^L \|\Theta^i\|_F^2 \right) \\ &\geq L \cdot \left(\prod_{i=1}^L \|\Theta^i\|_F^2 \right)^{\frac{1}{L}} \\ &\geq L \cdot \left(\left\| \prod_{i=1}^L \Theta^i \right\|_F^2 \right)^{\frac{1}{L}} \\ &= L \cdot (\|\Theta^P\|_F^2)^{\frac{1}{L}} \\ &= L \cdot (\|\Theta^P\|_2^2)^{\frac{1}{L}} \\ &:= C \end{aligned}$$

In order we use firs the property that the geometric mean is always lower than the arithmetic mean of positive numbers (AM-GM inequality), then the inequality

of the product of the norm and the norm of the product, finally the L2 norm is equal to the frobenius norm from the fact Θ^P is a column vector.

$$\min_k \sigma_k := \sigma_{\min}(\mathbf{J.3})$$

First using weight matrices alignment lemma we have:

$$\begin{aligned} \sigma_{\min} \leq \sigma_i \leq \sigma_{\min} + \epsilon \\ \iff \sigma_i = \sigma_{\min} + \mathcal{O}(\epsilon) \end{aligned} \tag{J.4}$$

Now let's prove an intermediary result that will be used later:

$$(\min_i \sigma_i)^2 := \sigma_{\min}^2 \leq \frac{C}{L} + \mathcal{O}(\epsilon^{\frac{1}{L}}) \tag{J.5}$$

where the $\mathcal{O}$ operator is used for "big o operator" used to describe the order of approximation.

proof:

$$\begin{aligned} \|\Theta^P\|_2^2 &= \left\| \prod_{i=1}^L \Theta^i \right\|_2^2 \\ &= \left\| \prod_{i=1}^{L-2} \Theta^i \cdot (\Theta^{L-1} \cdot \Theta^L) \right\|_2^2 \\ &= \left\| \prod_{i=1}^{L-2} \Theta^i \cdot Q^{L-1} \cdot \Sigma^{L-1} \cdot (V^{L-1})^T \cdot q^L \cdot (\sigma^L) \cdot (v^L)^T \right\|_2^2 \quad \text{using property (i) and (iii) of alignment} \\ &\geq \left\| \prod_{i=1}^{L-2} \Theta^i \cdot Q^{L-1} \cdot \begin{bmatrix} (1 - \frac{\epsilon}{(\sigma_{\min} + \epsilon)})\sigma_{L-1}\sigma_L & 0 & \dots \\ 0 & -\mathcal{O}(\epsilon) & \\ \vdots & & \ddots \end{bmatrix} \cdot (v^L)^T \right\|_2^2 \\ &\geq \left\| \prod_{i=1}^{L-2} \Theta^i \cdot Q^{L-1} \cdot \begin{bmatrix} \sigma_{\min}^2 - \mathcal{O}(\epsilon) & 0 & \dots \\ 0 & -\mathcal{O}(\epsilon) & \\ \vdots & & \ddots \end{bmatrix} \cdot (v^L)^T \right\|_2^2 \\ &\geq \|Q^1\|_2^2 \cdot \left\| \begin{bmatrix} \sigma_{\min}^L - \mathcal{O}(\epsilon) & 0 & \dots \\ 0 & -\mathcal{O}(\epsilon) & \\ \vdots & & \ddots \end{bmatrix} \cdot (v^L)^T \right\|_2^2 \\ &= (\sigma_{\min}^L)^2 - \mathcal{O}(\epsilon) \end{aligned}$$

Using the definition of C:

$$\begin{aligned}
C &:= L \cdot (\|\Theta^S\|_2^2)^{\frac{1}{L}} \geq ((\sigma_{\min}^L)^2 - \mathcal{O}(\epsilon))^{\frac{1}{L}} \\
&\iff C \geq L \cdot (\sigma_{\min}^2 - \mathcal{O}(\epsilon^{\frac{1}{L}})) \\
&\iff \sigma_{\min}^2 \leq \frac{C}{L} + \mathcal{O}(\epsilon^{\frac{1}{L}})
\end{aligned}$$

We can now show that assuming parameter matrix alignment lemma, the sum of squares coefficient differ from the theoretical minimum up to a factor on the order of $\epsilon^{\frac{1}{L}}$

$$\sum_{i=1}^L \|\Theta^i\|_F^2 \leq C + \mathcal{O}(\epsilon^{\frac{1}{L}}) \quad (\text{J.6})$$

proof :

$$\begin{aligned}
\sum_{i=1}^L \|\Theta^i\|_F^2 &\leq \sum_{i=1}^L (\|\Theta^i\|_2^2 + \mathcal{O}(\epsilon)) \\
&\leq L(\sigma_{\min}^2 + \mathcal{O}(\epsilon)) \\
&\leq L\left(\frac{C}{L} + \mathcal{O}(\epsilon^{\frac{1}{L}})\right) \\
&= C + \mathcal{O}(\epsilon^{\frac{1}{L}})
\end{aligned}$$

Because C is also a lower bound for the sum of squares coefficients, then this equation also holds interpreting C as the minimum possible value of the sum of squares coefficients.

Bibliography

Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992. doi: 10.1162/neco.1992.4.1.1.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine learning practice and the bias–variance trade–off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019. doi: 10.1073/pnas.1903070116. URL <https://www.pnas.org/content/116/32/15849>.

Suriya Gunasekar, Blake E. Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nathan Srebro. Implicit regularization in matrix factorization. In *Advances in Neural Information Processing Systems 30*, pages 6152–6160, Long Beach, CA, USA, 2017. Curran Associates, Inc. URL <https://proceedings.neurips.cc/paper/2017/file/58191d2a914c6dae66371c9dc91b41-Paper.pdf>.

Pierre Marion and Lénaïc Chizat. Deep linear networks for regression are implicitly regularized towards flat minima. In *Advances in Neural Information Processing Systems 37*, Vancouver, Canada, 2024. Neural Information Processing Systems Foundation, Inc. URL <https://proceedings.neurips.cc/paper/2024/hash/8c976a95df6a229551cd28c76627edc9-Abstract.html>. NeurIPS 2024 Main Conference Track.

Moritz Hardt and Benjamin Recht. *Patterns, Predictions, and Actions: A Story About Machine Learning*. Princeton University Press, Princeton, NJ, 2022. ISBN 9780691233734. URL <https://mlstory.org/>.

John Duchi. CS229 Supplemental Lecture notes Hoeffding’s inequality.

Adrian G. Barnett, Jolieke C. van der Pols, and Annette J. Dobson. Regression to the mean: What it is and how to deal with it. *International Journal of Epidemiology*, 34(1):215–220, February 2005. ISSN 0300-5771. doi: 10.1093/ije/dyh299.

- Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, 2nd edition, 2009. doi: 10.1007/978-0-387-84858-7. URL <https://doi.org/10.1007/978-0-387-84858-7>.
- Halbert White. Using Least Squares to Approximate Unknown Regression Functions. *International Economic Review*, 21(1):149–170, 1980. ISSN 0020-6598. doi: 10.2307/2526245.
- Nathan Srebro. Applying statistical learning theory to deep learning, 2022. URL <https://www.youtube.com/playlist?list=PLEIq5bchE3R1XjmCpiKc960umHySw3neU>. Lecture at the Summer School on Statistical Physics of Machine Learning, Les Houches (July 4–29, 2022).
- Suzy Ayah. Equivalence of constrained and unconstrained form for ridge regression, Jul 2018. URL <https://suzyahyah.github.io/optimization/2018/07/20/Constrained-unconstrained-form-Ridge.html>. Accessed: 2026-01-06.
- George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989. doi: 10.1007/BF02551274.
- Zhi-Qin Lu, Hanming Pu, Fei Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *Neural Networks*, 78:1–10, 2016. doi: 10.1016/j.neunet.2016.12.006.
- Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in Neural Information Processing Systems 27*, pages 2924–2932, Montréal, Canada, 2014. Curran Associates, Inc. URL <https://proceedings.neurips.cc/paper/2014/hash/5422-on-the-number-of-linear-regions-of-deep-neural-networks.pdf>.
- David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. doi: 10.1038/323533a0.
- Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 244–253, Stockholm, Sweden, July 2018. PMLR.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl