

On the generalization gap of feedforward ReLU networks with low sensitivity

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF NUMERICAL ANALYSIS



Author:

András Csértán
Computer Science MSc

Supervisors:

Dániel Rác
research assistant, SZTAKI
PhD student, ELTE TTK

Levente Lócsi
senior lecturer, ELTE IK

Budapest, 2023

Contents

1	Introduction and related work	2
2	Preliminaries	4
2.1	Formalization of learning	4
2.2	Neural networks	9
3	Tangent sensitivity	11
3.1	Motivation	11
3.2	Bounding the generalization gap	12
4	Experiments	15
4.1	Jax	16
4.2	Calculating the tangent sample sensitivity	17
4.3	Results	19
5	Discussion	21
5.1	Future work	22
	Acknowledgements	24
	Bibliography	24

Chapter 1

Introduction and related work

Neural Networks have been used extensively in the past decade to solve many tasks related to machine learning, such as image classification, object detection, speech recognition, etc. However, there are still very few theoretical guarantees that these networks generalize well to unseen data. A good example of this is adversarial attacks [1], where a small change of the input, which is usually undetectable by an independent observer, causes the network to mislabel that input. Another phenomenon is overfitting, where the trained network performs well on the training set, but its true error is relatively high. With the introduction of the Neural Tangent Kernel (NTK) [2], there has been a lot of research related to overparameterized neural networks. This led to several theoretical results related to the convergence of gradient-based optimization methods [3, 4]. It has been shown empirically that these networks can be trained to have zero training error, but they still generalize well to unseen data [5]. This phenomenon suggests an implicit bias in the case of gradient-based optimization methods: the optimization algorithm prefers solutions which generalize well, while interpolating the training set [6, 7].

A possible approach to give theoretical guarantees on the generalization abilities of models in learning theory is the framework of Probably Approximately Correct (PAC) bounds [8], which was applied to neural networks in [9] by bounding the Rademacher complexity. These bounds usually depend on the norms of the layerwise weight matrices and not the actual weight structure of the network.

With the advancement in computing power, it is now possible to train very large neural networks which gives possibilities to empirically examine different properties of these networks [10]. However, it is also important, that the implementation of

these experiments is efficient so that we can perform a large number of experiments in a reasonable amount of time. There are many machine learning frameworks that provide implementations of neural networks out of the box, such as TensorFlow [11], PyTorch [12], Jax [13], etc. In this study, we will use Jax because it provides efficient implementations of automatic differentiation, vectorization, and just-in-time compilation, which are all important for our experiments.

This study covers our paper [14] which will be presented at the NeurIPS 2023 Workshop: Optimization for Machine Learning. My main contribution was providing an efficient implementation of the experiments, which was essential to obtain the empirical results presented in the paper. See Section 4.2 for the details.

The rest of this study is organized as follows. In Chapter 2 we will give a brief introduction to the basic concepts of machine learning and neural networks. We will also introduce how to measure the generalization ability of a model, and how can we give theoretical bounds on it. In Chapter 3 we will introduce the notion of tangent sensitivity, first defined in [15]. We will also show how it can be used to bound the generalization gap of a neural network. In Chapter 4 we will present the results of our experiments which show that the tangent sensitivity has a strong correlation with the generalization gap of a neural network. Finally, in Chapter 5 we will summarize our results and give some ideas for future research.

Chapter 2

Preliminaries

2.1 Formalization of learning

The phrase *machine learning* is used to describe a wide range of algorithms that can be used to solve various tasks. In this study, we will focus on the task of supervised learning and within that the case of classification problems. For an introduction to machine learning and these concepts, see e.g. [16]. In the following, we will formalize the notion of learning in the case of classification problems.

Let us consider a set of objects denoted by X and the task of predicting some unknown properties of the elements of X . These properties can be characterized by some label set Y . For example, X can be the set of all images, and the property of interest can be whether a single image contains a dog or not. In this case, we can let $Y := \{0, 1\}$, where 0 means the absence of a dog, and 1 means that the image does contain a dog. Our goal is to find an algorithm that can correctly label any image. We solve this problem by finding a function $h : X \rightarrow Y$, where for each $x \in X$ and the corresponding $y_x \in Y$

$$h(x) = y_x.$$

From now on, we will consider only the task of binary classification (i.e. $Y = \{0, 1\}$), and assume that the set of objects X can be modeled with a data distribution $\mathcal{X}$ on $\mathbb{R}^{n_{\text{in}}}$. We will call n_{in} the input dimension. We will search for h in the form of $h = c \circ f$, where $f : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ is a model function and $c : \mathbb{R} \rightarrow \{0, 1\}$ is a thresholding function.

Usually, we fix c and try to measure the ‘goodness’ of a function f with the help of an *elementwise loss function* $l : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}_0^+$, where $l(y', y)$ measures the loss

of predicting y' when the true label is y for a single data point. Intuitively, we want $l(y', y)$ to be small if $y' := f(x)$ is close to y and large otherwise. The following are some standard choices for l .

- **0 – 1 loss:** $l(y', y) = \begin{cases} 0 & \text{if } c(y') = y \\ 1 & \text{if } c(y') \neq y \end{cases}$
- **Squared error:** $l(y', y) = |y' - y|^2$
- **Cross-entropy:** $l(y', y) = -y \log(y') - (1 - y) \log(1 - y')$

For the cross-entropy loss function we need to transform $f(x)$ to the interval $[0, 1]$. This could be done by applying the *sigmoid* function $\sigma : \mathbb{R} \rightarrow (0, 1)$ to $f(x)$, where $\sigma(x) = \frac{1}{1+e^{-x}}$.

In the following we will use the concept of elementwise loss function from above to define a loss value in terms of the model function f . To formalize this, let us assume that the data comes from a distribution $\mathcal{X} \times \mathcal{Y} = \mathcal{D}$ over $X \times Y = \mathbb{R}^{n_{\text{in}}} \times \{0, 1\}$. Then the *true error* of a function $f : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ for a fixed loss function l w.r.t. $\mathcal{D}$ is

$$L_{\mathcal{D}}(f) := \mathbb{E}_{(x,y) \sim \mathcal{D}} [l(f(x), y)].$$

The problem is that we usually don't have access to the distribution $\mathcal{D}$, but only to a finite set of data $S = ((x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)) = (z_1, z_2, \dots, z_m)$ sampled from $\mathcal{D}$. For a fixed dataset S the *empirical loss* or *empirical risk* of a function $f : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ is defined as

$$L_{\text{emp}}^{(S)}(f) := \frac{1}{m} \sum_{i=1}^m l(f(x_i), y_i).$$

The definitions above follow the notation of [16].

The question is: if we only know $L_{\text{emp}}^{(S)}(f)$, can we say anything about $L_{\mathcal{D}}(f)$? We address this problem in the form of giving an upper bound on $L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f)$, called the *generalization gap* of f (w.r.t dataset S and distribution $\mathcal{D}$). In this study we consider the framework of *probably approximately correct* (PAC) bounds. Approximate, because we do not insist on the true error $L_{\mathcal{D}}(f)$ to be equal to the empirical error $L_{\text{emp}}^{(S)}(f)$, it is sufficient for us to prove that $L_{\mathcal{D}}(f) < L_{\text{emp}}^{(S)}(f) + \varepsilon$ for some $\varepsilon \in \mathbb{R}^+$. The bound is probable, because S is a random sample from $\mathcal{D}^m$ and it can happen that the empirical loss on S does not tell much about the underlying

distribution $\mathcal{D}$, for example it can happen that the sample S consists of the same data point (x, y) m times. Because of this, we want the above approximate bound to hold with a probability of at least $1 - \delta$ for some $\delta \in (0, 1)$ over the sample S . More formally, PAC-bounds are usually of the form

$$\forall \delta \in (0, 1) \exists \varepsilon_\delta : \mathbb{P}_{S \sim (\mathcal{D})^m} [L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f) < \varepsilon_\delta] \geq 1 - \delta.$$

So if we can find a function f which has low empirical risk and can prove that a bound like the above exists, it is possible to give theoretical guarantees that f will perform well on unseen data too. This is one of the reasons why *Empirical Risk Minimization* (ERM) [17] is used to choose the appropriate function f . However, if there is no restriction on the class of functions from which we can choose, it is relatively easy to find a function that has zero empirical risk. To see this, consider the following function:

$$f^*(x) := \begin{cases} y' & \text{if } \exists (x', y') \in S \text{ s.t. } x = x' \\ 0 & \text{otherwise} \end{cases}.$$

Basically f^* memorizes S and if it encounters an input x which is not present in S , it predicts 0 by default. Although f^* has zero empirical risk, it does not generalize well to unseen data.

This issue is solved by restricting the class of functions $\mathcal{F}$ from which we can choose. For example, $\mathcal{F}$ can be the set of linear predictors, neural networks with a fixed number of parameters, etc. The choice of $\mathcal{F}$ usually represents our prior knowledge of the problem in hand. If we choose a rather simple function class, we might not be able to capture the nature of some more complex distributions, but at the same time, the chosen function f^* will be less prone to overfitting. In the case of ERM (w.r.t. training set S), f^* is any function in $\mathcal{F}$ that minimizes the empirical risk, i.e.

$$L_{\text{emp}}^{(S)}(f^*) = \min_{f \in \mathcal{F}} L_{\text{emp}}^{(S)}(f).$$

One way to measure the complexity of a set of functions is the *Vapnik-Chervonenkis (VC) dimension* [18]. Informally, the VC dimension of a function class $\mathcal{F}$ is the maximum positive integer n for which there exists a fixed set of points, where for any binary labeling of these points, there exists a function $f \in \mathcal{F}$ that can perfectly label the points. For a formal definition, see e.g. [16].

Note that the VC dimension of $\mathcal{F}$ does not depend on the sample S . However, it is still possible to give bounds on the generalization gap of a function $f \in \mathcal{F}$ using the VC dimension, see e.g. [16]. To give tighter bounds, we can use complexity measures that make use of S . Such a measure of complexity is for example the *Rademacher complexity* [16].

Next, we will derive an upper bound on the expectation of the supremum of the generalization gap, following the proof of Lemma 26.2 in [16]. First, if $S = (z_1, z_2, \dots, z_m)$ is a finite sample from $\mathcal{D}$, then

$$L_{\mathcal{D}}(f) = \mathbb{E}_{S \sim \mathcal{D}} [L_{\text{emp}}^{(S)}(f)]. \quad (2.1)$$

Let $S' = (z'_1, z'_2, \dots, z'_m)$ be another sample of the same size m from $\mathcal{D}$, then using (2.1) and the fact that the supremum of expectation is greater or equal than the expectation of the supremum, we get

$$\begin{aligned} & \mathbb{E}_S \left[\sup_{f \in \mathcal{F}} (L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f)) \right] \\ &= \mathbb{E}_S \left[\sup_{f \in \mathcal{F}} \left(\mathbb{E}_{S'} [L_{\text{emp}}^{(S')}(f)] - L_{\text{emp}}^{(S)}(f) \right) \right] \\ &\leq \mathbb{E}_{S, S'} \left[\sup_{f \in \mathcal{F}} \left(L_{\text{emp}}^{(S')}(f) - L_{\text{emp}}^{(S)}(f) \right) \right] \\ &= \frac{1}{m} \cdot \mathbb{E}_{S, S'} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m (f(z'_i) - f(z_i)) \right) \right] \\ &= \frac{1}{m} \cdot \mathbb{E}_{S, S', \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot (f(z'_i) - f(z_i)) \right) \right] \\ &\leq \frac{1}{m} \cdot \mathbb{E}_{S, S', \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot f(z'_i) \right) + \sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m -\sigma_i \cdot f(z_i) \right) \right], \end{aligned}$$

where $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m)$ is sampled independently and identically distributed (i.i.d.) from the Rademacher distribution, i.e. $\mathbb{P}[\sigma_i = 1] = \mathbb{P}[\sigma_i = -1] = \frac{1}{2}$. For more details on the second last equation, see the proof of Lemma 26.2 in [16]. Using the fact that the probability of σ is the same as of $-\sigma$:

$$\begin{aligned}
 & \frac{1}{m} \cdot \mathbb{E}_{S, S', \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot f(z'_i) \right) + \sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m -\sigma_i \cdot f(z_i) \right) \right] \\
 &= \frac{1}{m} \cdot \mathbb{E}_{S', \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot f(z'_i) \right) \right] + \frac{1}{m} \cdot \mathbb{E}_{S, \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot f(z_i) \right) \right] \\
 &= \frac{2}{m} \cdot \mathbb{E}_{S, \sigma} \left[\sup_{f \in \mathcal{F}} \left(\sum_{i=1}^m \sigma_i \cdot f(z_i) \right) \right].
 \end{aligned}$$

First, we bounded the generalization gap with the expected difference of two samples of the same size, then we bounded this with the expression above. This gives the motivation for the following definition:

Definiton 1 (See Definition 26.1 in [16]). Let $A \subset \mathbb{R}^m$ be a set of vectors and $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_m) \subset \{-1, +1\}^m$ sampled i.i.d. from the Rademacher distribution, i.e. $\mathbb{P}(\sigma_i = -1) = \mathbb{P}(\sigma_i = 1) = \frac{1}{2}$ ($i = 1, 2, \dots, m$) Then the *Rademacher complexity* of A is defined as

$$\mathcal{R}(A) := \frac{1}{m} \mathbb{E}_{\sigma} \left[\sup_{a \in A} \sum_{i=1}^m a_i \cdot \sigma_i \right].$$

If we let $\mathcal{F} \circ S := \{(f(z_1), f(z_2), \dots, f(z_m)) \mid f \in \mathcal{F}\} \subset \mathbb{R}^m$, then the bound on the generalization gap above can be written in the following compact form (See Lemma 26.2 in [16]):

$$\mathbb{E}_S \left[\sup_{f \in \mathcal{F}} (L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f)) \right] \leq 2 \cdot \mathbb{E}_S [\mathcal{R}(\mathcal{F} \circ S)]. \quad (2.2)$$

With the notion of Rademacher complexity, we can introduce the following theorem which is essential in the proof of Theorem 2 in Chapter 3.

Theorem 1 (See Theorem 26.5 in [16]). *Let $\mathcal{F}$ be a function class, $\mathcal{D}$ a distribution over $X \times Y$ and l a loss function. Moreover, assume that*

$$\exists c \in \mathbb{R}, \forall (x, y) \in X \times Y, \forall f \in \mathcal{F} : l(f(x), y) \leq c..$$

Then $\forall f \in \mathcal{F}, \forall \delta \in (0, 1) :$

$$\mathbb{P}_{S \sim \mathcal{D}^m} \left[L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f) \leq 2 \cdot \mathcal{R}(l \circ \mathcal{F} \circ S) + 4c \sqrt{\frac{2 \log \left(\frac{4}{\delta} \right)}{m}} \right] \geq 1 - \delta.$$

where $\mathcal{R}(l \circ \mathcal{F} \circ S)$ is the Rademacher complexity of the set

$$\left\{ (l(f(x_1), y_1), l(f(x_2), y_2), \dots, l(f(x_m), y_m)) \in \mathbb{R}^m \mid f \in \mathcal{F} \right\}.$$

Note that here the bound on $L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f)$ is for a fixed S in contrast to the bound in (2.2), where we took expectation over S . The upper bound consists of two terms, the second is a constant depending on the loss function, the size of S , and the confidence level δ . Regarding the first term, calculating the Rademacher complexity of a set is a hard task, but it is possible to give upper bounds on it for specific function classes [16].

2.2 Neural networks

From now on, we will only consider the class of feedforward, parametric neural networks with the ReLU activation function. A neural network consists of an input layer, one or more hidden layers and an output layer. In a single layer there are multiple units ('neurons'), where each unit calculates a weighted sum of the output of the previous layers' units and applies a nonlinear function to it. See figure 2.1 for a visual representation.

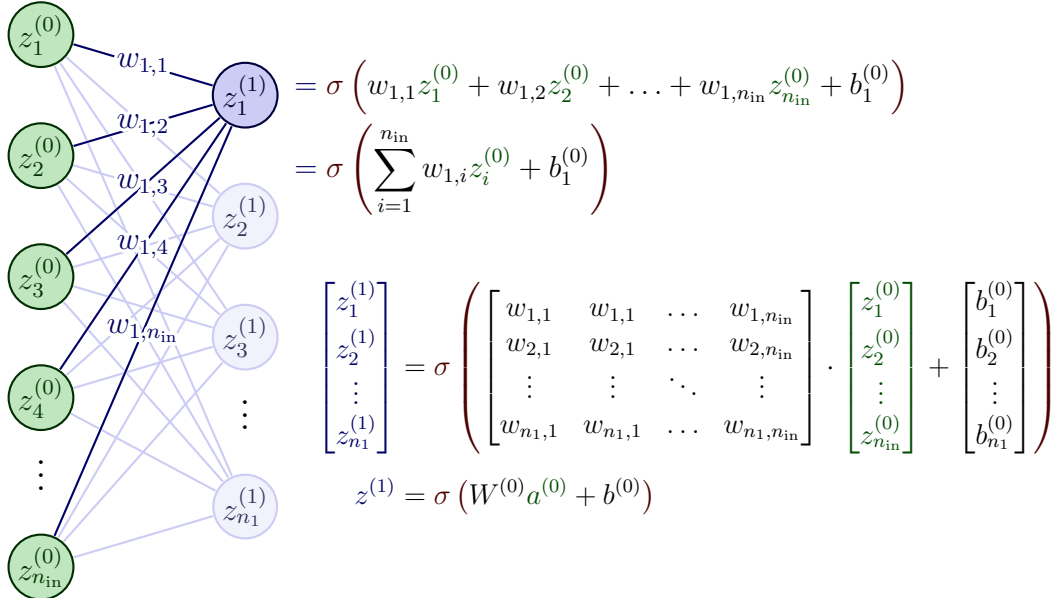


Figure 2.1: Visual representation of the first layer of a neural network. Adapted from [19].

Let L denote the number of hidden layers and n_i the number of units in the i th layer ($i = 1, \dots, L$). Also let $n_0 := n_{\text{in}}$ and $n_{L+1} := n_{\text{out}}$, the input and output

dimensions, and define for $i = 1, \dots, L + 1$

$$z^{(i)} := \sigma \left(W^{(i)} \cdot z^{(i-1)} + b^{(i)} \right) \quad \left(z^{(i-1)} \in \mathbb{R}^{n_{i-1}}, W^{(i)} \in \mathbb{R}^{n_i \times n_{i-1}}, b^{(i)} \in \mathbb{R}^{n_i} \right),$$

where σ is a nonlinear function. We will use the ReLU activation function, where

$$\sigma(x) = \max(0, x).$$

Let $P := \sum_{i=0}^L (n_i + 1) \cdot n_{i+1}$ denote the number of parameters of the neural network. Then a neural network is a function

$$f : \mathbb{R}^P \times \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}^{n_{\text{out}}}, \quad f(\theta, x) := z^{(L+1)}.$$

For a fixed P , let $\mathcal{F}_P$ denote the class of feedforward ReLU networks with output dimension $n_{\text{out}} = 1$ and parameter count P .

The training of a neural network is finding a parameter vector $\theta \in \mathbb{R}^P$ for which the empirical loss of $f_\theta : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$, $f_\theta(x) := f(\theta, x)$ w.r.t. the training set $S_{\text{train}} = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$ is sufficiently small. It is usually carried out by the method of gradient descent, in which we choose an initial $\theta_0 \in \mathbb{R}^P$ and in each turn we update θ according to the rule in (2.3). On how to choose θ_0 see e.g. [20]. Because we optimize w.r.t. the parameter vector θ , we will use the notation $\mathcal{L}_f^{(S_{\text{train}})}(\theta) := L_{\text{emp}}^{(S_{\text{train}})}(f_\theta)$. Then the update rule for θ is

$$\theta_t := \theta_{t-1} - \eta(t) \cdot \nabla_\theta \mathcal{L}_f^{(S_{\text{train}})}(\theta_{t-1}) \quad (t = 1, \dots, T, T \in \mathbb{N}) \quad (2.3)$$

where $T \in \mathbb{N}$ is the number of iterations and $\eta : \mathbb{N} \rightarrow \mathbb{R}^+$ is the *learning rate* function; $\eta(t)$ determines the length of the step we take in iteration t . Note that

$$\nabla_\theta \mathcal{L}_f^{(S_{\text{train}})}(\theta) = \frac{1}{N} \cdot \sum_{i=1}^N \frac{\partial l}{\partial f} \nabla_\theta f(\theta, x_i),$$

which plays an important role in the proof of Theorem 2.

Chapter 3

Tangent sensitivity

3.1 Motivation

Let us assume that we found a function that can classify any input from the training set with relatively high accuracy. What happens if we add some small noise to the input? One might expect that this should not have a large impact on the performance of the model. However, this is not always the case. For example, neural networks with very high accuracy on well-known datasets are prone to adversarial attacks [1]; it is possible to perturb the input in a way that the change is undetectable by an independent observer, but the model mislabels it nevertheless. So it is desirable that these small perturbations do not affect the classification abilities of the network, so the model is stable in a sense [7].

In the case of optimizing via gradient descent, we can also examine how the resulting parameter vector θ_T changes if we perturb the input a little. This gives the motivation for the following. Let f be a feedforward ReLU network and let us consider a Gaussian perturbation around the input x of the form

$$\phi(x) = x + \delta(x),$$

where $\delta(x) \sim \mathcal{N}(0, \sigma \mathbf{I})$ ($\sigma \in \mathbb{R}$) is a random variable. Then the expected change in the gradient of f w.r.t. the parameter vector θ (which has a significant impact on the optimization trajectory of the gradient descent algorithm) is related to the

following [15]:

$$\begin{aligned} \mathbb{E}_{\delta(x)} [\|\nabla_{\theta} f(\theta, x) - \nabla_{\theta} f(\theta, \phi(x))\|_2^2] &\sim \mathbb{E}_{\delta(x)} \left[\left\| \frac{\partial}{\partial x} \nabla_{\theta} f(\theta, x) \cdot \delta(x) \right\|_2^2 \right] \leq \\ &\leq \sigma \cdot \left\| \frac{\partial}{\partial x} \nabla_{\theta} f(\theta, x) \right\|_2^2. \end{aligned}$$

This gives the motivation for the following definition:

Definiton 2 (Tangent sensitivity [15]). Let $f : \mathbb{R}^P \times \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ be a feedforward, parametric neural network, where P is the number of parameters, n_{in} is the input dimension of the network. Then the *Tangent Sample Sensitivity* of f is

$$\text{TS}_f : \mathbb{R}^P \times \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}^{P \times n_{\text{in}}}, \quad \text{TS}_f(\theta, x) := \frac{\partial^2}{\partial x \partial \theta} f(\theta, x) \quad (\theta \in \mathbb{R}^P, x \in \mathbb{R}^{n_{\text{in}}}).$$

The *Tangent Sensitivity* is the expectation of the Tangent Sample Sensitivity, i.e.

$$\text{TS}_f(\theta) = \mathbb{E}_{x \sim \mathcal{X}} \text{TS}_f(\theta, x).$$

Tangent sensitivity is also interesting because its connection to the distribution of activation regions in ReLU networks. See [15] for more details.

3.2 Bounding the generalization gap

From now on, we will consider a fixed training set S_{train} of size N sampled from $\mathcal{D}$. Let us denote by $\text{GD}(\theta_0, \eta, T)$ the parameter vector $\theta_T \in \mathbb{R}^P$ obtained by the gradient descent algorithm with initial parameter $\theta_0 \in \mathbb{R}^P$, learning rate function $\eta : \mathbb{N} \rightarrow \mathbb{R}^+$, and number of iterations $T \in \mathbb{N}$. From (2.3) we have

$$\text{GD}(\theta_0, \eta, T) = \theta_T = \theta_0 - \sum_{t=1}^{T-1} \eta(t) \cdot \nabla \mathcal{L}_f^{(S_{\text{train}})}(\theta_t)$$

For a fixed $\theta_0 \in \mathbb{R}^P$, let $\text{Traj}(\theta_0) \subset \mathbb{R}^P$ denote the subset of parameters which are reachable from θ_0 with the gradient descent algorithm, i.e.

$$\text{Traj}(\theta_0) := \left\{ \theta \in \mathbb{R}^P \mid \exists T \in \mathbb{N}, \exists \eta : \mathbb{N} \rightarrow \mathbb{R}^+ : \theta = \text{GD}(\theta_0, \eta, T) \right\},$$

where we used the notation in [14].

Before we state the main theorem, let us introduce the necessary assumptions regarding the loss function, the network, and the optimization algorithm.

Assumption 1. *The loss function is of the form $l(f(x), y) = \ell(f(x) - y)$, where ℓ is K_L -Lipschitz. Moreover, the loss function can be bounded from above by a constant $B \in \mathbb{R}$.*

Assumption 2. *For a fixed $\theta_0 \in \mathbb{R}^P$ and $\varepsilon > 0$, let $B_\varepsilon(\theta_0)$ denote the ε -ball around θ_0 and let $U_{\theta_0, \varepsilon} := \text{Traj}(\theta_0) \cap B_\varepsilon(\theta_0)$. We assume that the Frobenius norm of the tangent sample sensitivity matrix is bounded on $U_{\theta_0, \varepsilon}$ for every $\theta_0 \in \mathbb{R}^P$ w.r.t. the training set input $S_{\text{train}}^{(x)} = (x_1, x_2, \dots, x_N) \in (\mathbb{R}^{n_{\text{in}}})^N$, i.e.*

$$\exists C_{\text{TS}} > 0, \forall \theta_0 \in \mathbb{R}^P, \forall \theta \in U_{\theta_0, \varepsilon} : \sup_{x \in S_{\text{train}}^{(x)}} \|\text{TS}_f(\theta, x)\|_F \leq C_{\text{TS}}.$$

$U_{\theta_0, \varepsilon}$ consists of those parameters in the neighborhood of the initial parameter θ_0 which are reachable with the gradient descent algorithm. In light of the above assumption, let us define the following set of functions:

$$\mathcal{F}_{\theta_0, C, \varepsilon} := \left\{ f(\theta, \cdot) \in \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R} \mid \theta \in U_{\theta_0, \varepsilon}, \sup_{x \in S_{\text{train}}^{(x)}} \|\text{TS}_f(\theta, x)\|_F \leq C \right\}.$$

Assumption 3. *Assume that the following bounds hold:*

- $\sup_{x \sim \mathcal{X}} |f(\theta_0, \cdot)| \leq K_{\theta_0}$
- $\sup_{x \sim \mathcal{X}} \|x\| \leq K_x,$
- $\sup_{x \sim \mathcal{X}} \|\nabla_\theta f(\theta_0, x)\| \leq K_{\nabla_0}.$

Theorem 2 (See Theorem 5 in [14]). *Let the assumptions above hold for a fixed $\theta_0 \in \mathbb{R}^P$ and $\varepsilon > 0$. Moreover, assume that $\exists C_{\text{GD}} > 0, \forall T \in \mathbb{N}, \forall \eta : \mathbb{N} \rightarrow \mathbb{R}^+ :$*

$$\sum_{t=1}^{T-1} \eta(t) \frac{1}{N} \sum_{i=1}^N \frac{\partial l}{\partial f}(\theta_{t-1}, x_i) \leq C_{\text{GD}}.$$

Then $\forall \delta \in (0, 1)$ with probability at least $1 - \delta$ over the choice of S of size m , $\forall f(\theta_0, \cdot) \in \mathcal{F}_{\theta_0, C_{\text{TS}}, \varepsilon} :$

$$L_{\mathcal{D}}(f) - L_{\text{emp}}^{(S)}(f) \leq K_L K_{\theta_0} + \frac{C_1}{\sqrt{m}} + K_L H(\theta) + B \sqrt{\frac{2 \log\left(\frac{4}{\delta}\right)}{m}},$$

where $C_1 = 2K_L K_x K_{\nabla_0} C_{\text{TS}} C_{\text{GD}}$ and $H(\theta)$ is an error term depending on the linear approximation of the network function f around the initialization point θ_0 .

The critical terms in the theorem are C_{TS} , C_{GD} and $H(\theta)$, because the other terms do not depend on the optimization strategy. $H(\theta)$ is related the following. Let $h_f : \mathbb{R}^P \times \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ be the error term in the linear approximation of the network function f around the initialization point θ_0 , i.e.

$$f(\theta, x) = f(\theta_0, x) + \nabla_{\theta} f(\theta_0, x)(\theta - \theta_0) + h_f(\theta, x).$$

Then for every θ for which $f(\theta, \cdot) \in \mathcal{F}_{\theta_0, C_{\text{TS}}, \varepsilon}$, the Rademacher complexity of the set

$$\left\{ (h_f(\theta, x_1), h_f(\theta, x_2), \dots, h_f(\theta, x_m)) \right\}$$

can be upper bounded by $H(\theta)$. Note that in the case of ReLU networks, the norm of the Hessian $H_{\theta} f(\theta, x)$ tends to zero as the width tends to infinity [21]. Hence the error term $H(\theta)$ also tends to zero. For the complete proof of the theorem, see Appendix B. in [14].

Currently there is no theoretical guarantee for the value of C_{TS} . However, it is conjectured that the tangent sensitivity and the generalization ability of the network has an even stronger connection than it is indicated by the theorem. In Chapter 4 we show empirically that the empirical generalization gap (the difference between the error on the test set and the error on the training set, see Chapter 4 for more details) is strongly correlated with the mean tangent sample sensitivity on the test set. It means it is possible to approximate the behavior of the loss during training on an unlabeled sample.

Chapter 4

Experiments

To reinforce the theoretical results in the previous chapter, we performed experiments on the MNIST [22] and CIFAR-10 [23] datasets. We used the Jax [13] machine learning framework to implement the neural network architectures and the experiments.¹

Theorem 2 gives an upper bound on the generalization gap of a feedforward ReLU network. However, due not having access to the underlying data distribution $\mathcal{D}$, we cannot calculate the generalization gap directly. Instead, we can approximate it with the empirical generalization gap, which is defined as follows.

Let S_{train} and S_{test} be two datasets of size N_{train} and N_{test} respectively, sampled from $\mathcal{D}$. Then the *empirical generalization gap* of a function $f : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}$ (w.r.t. loss function l) is

$$L_{\text{emp}}^{(S_{\text{test}})}(f) - L_{\text{emp}}^{(S_{\text{train}})}(f) = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} l(f(x_i), y_i) - \frac{1}{N_{\text{train}}} \sum_{j=1}^{N_{\text{train}}} l(f(x_j), y_j).$$

Note that these values are usually calculated during the training process to monitor the performance of the network on the training and test sets.

Calculating the tangent sample sensitivity of a neural network is a computationally expensive task. Consider a neural network with P parameters and n_{in} input dimensions. Then the tangent sensitivity is a $n_{\text{in}} \times P$ matrix, which has $n_{\text{in}} \cdot P$ elements. For example, if we have a neural network with $P = 10\,000\,000$ parameters and input dimension $n_{\text{in}} = 3000$ and we assume that we store the data in 32-bit floating point format, then the tangent sample sensitivity matrix will take up approximately

¹The source code is available at https://github.com/danielracz/tansens_public.

112 GB of memory. In our most extreme example, we had to calculate this for images from the CIFAR-10 dataset which have input dimension $n_{\text{in}} = 3072$, for a neural network with structure $[3072, 3000, 3000, 3000, 10]$ which has $P = 27\,258\,010$ parameters. Moreover, to approximate the tangent sensitivity, we need to perform this computation for every data point in the test set, which consists of 10 000 images in case of the CIFAR-10 and MNIST datasets. However, note that in Theorem 2, we only need to compute the Frobenius norm of the tangent sample sensitivity matrix, which is a much cheaper operation. Using the built-in primitives of Jax and other ideas we managed to speed up the computation, and more importantly, reduce the memory usage.

4.1 Jax

Jax [13] is a machine learning framework for Python developed by Google. Its most important features are the following:

- **automatic differentiation:** With the built-in `grad` function, we can easily compute the gradient of any Python or NumPy [24] function and their compositions. It supports both forward and reverse mode automatic differentiation.
- **automatic vectorization:** Jax can automatically vectorize functions with the `vmap` function. This is very useful for evaluating a function on a large number of inputs, e.g. when computing the loss function on a batch of data.
- **just-in-time (JIT) compilation:** Jax can compile Python code to XLA [25] code with the `jit` function, which can be executed on a CPU, GPU, or TPU. XLA stands for Accelerated Linear Algebra and it is a domain-specific compiler which was developed to speed up TensorFlow [11] programs and it is also used in Jax.

Jax has two fundamental functions for implementing automatic differentiation: Jacobian-vector products (JVP) and vector-Jacobian products (VJP). Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a function and $x \in \mathbb{R}^n$ be a vector. Then the JVP is an efficient way to calculate

$$\text{jvp}_f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad \text{jvp}_f(x, v) := \nabla f(x) \cdot v$$

for any $v \in \mathbb{R}^n$. The VJP is an efficient way to calculate

$$\text{vjp}_f : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad \text{vjp}_f(x, w) := \nabla f(x)^T \cdot w$$

for any $w \in \mathbb{R}^m$. For more details, see [26]. We will use these functions to efficiently compute the Frobenius norm of the tangent sample sensitivity matrix.

4.2 Calculating the tangent sample sensitivity

My main contribution to [14] was providing an efficient implementation for calculating the Frobenius norm of the tangent sample sensitivity matrix. Because our goal was to examine the behavior of overparameterized neural networks, the computation had to be successful for large neural networks such as the example at the beginning of this Chapter. In this section we will describe the ideas behind the implementation in detail.

To approximate the Frobenius norm of the tangent sensitivity matrix, we used the following formula:

$$\|\text{TS}_f(\theta)\|_F \approx \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \|\text{TS}_f(\theta, x_i)\|_F. \quad (4.1)$$

To calculate the Frobenius norm of the tangent sample sensitivity matrix for a single data point, consider the following. Let $g_\theta : \mathbb{R}^{n_{\text{in}}} \rightarrow \mathbb{R}^P$, $g_\theta(x) := \nabla_\theta f(\theta, x)$. Then the Frobenius norm of the tangent sample sensitivity of f at x can be computed as

$$\|\text{TS}_f(\theta, x)\|_F = \sqrt{\text{Tr} \left(\nabla g_\theta(x) \cdot (\nabla g_\theta(x))^T \right)},$$

where $\text{Tr}(A) = \sum_{i=1}^n a_{ii}$ is the trace of matrix $A \in \mathbb{R}^{n \times n}$.

To efficiently calculate $\nabla g_\theta(x) \cdot (\nabla g_\theta(x))^T$ we will use the following idea inspired by [27]. Consider the version of the jvp and vjp functions where the second argument is a matrix instead of a vector²:

$$\begin{aligned} \text{JVP}_f : \mathbb{R}^n \times \mathbb{R}^{n \times k} &\rightarrow \mathbb{R}^{m \times k}, & \text{JVP}_f(x, V) &:= \nabla f(x) \cdot V \\ \text{VJP}_f : \mathbb{R}^n \times \mathbb{R}^{m \times k} &\rightarrow \mathbb{R}^{n \times k}, & \text{VJP}_f(x, W) &:= \nabla f(x)^T \cdot W. \end{aligned}$$

²We can obtain these functions in Jax by `vmapping` the original `jvp` and `vjp` functions over the second argument.

Then we can compute the value of $\nabla g_\theta(x) \cdot (\nabla g_\theta(x))^T$, without calculating the Jacobian $\nabla g_\theta(x)$, as

$$\text{JVP}_{g_\theta}(x, \text{VJP}_{g_\theta}(x, I)) = \nabla g_\theta(x) \cdot \left((\nabla g_\theta(x))^T \cdot \mathbf{I} \right) = \nabla g_\theta(x) \cdot (\nabla g_\theta(x))^T,$$

where $\mathbf{I} \in \mathbb{R}^{n \times n}$ is the identity matrix.

To further speed up the computation of (4.1), we do not have to calculate and store $\|\text{TS}_f(\theta, x_i)\|_F$ for every x_i in the test set. Instead, we store in every iteration the sum of the first i terms and at the end we divide this sum by N_{test} . To use the `jit` transform of Jax to speed up our function using XLA³, we cannot use `if` statements or loops in our function. To overcome this, we use the `scan` function of Jax, which is a functional programming primitive that can be used to implement loops. The `scan` function takes a function f and an initial value $init$ and a list of values xs and returns a tuple $(carry, ys)$, where $carry$ is the last value of the accumulator and ys is the list of values computed by f . The built-in `scan` function of Jax is roughly equivalent to the following Python code:

```

1 def scan(f, init, xs):
2     carry = init
3     ys = []
4     for x in xs:
5         carry, y = f(carry, x)
6         ys.append(y)
7     return carry, ys

```

This leads us to the final implementation of the function that calculates the mean Frobenius norm of the tangent sample sensitivity matrices for a batch of data.

³See e.g. [28] for a performance comparison of Jax+jit and native NumPy.

```

1 @partial(jax.jit, static_argnames=('f',))
2 def mean_tansens_norm(f, W, inp):
3     def df(W, x, out):
4         f_out = jax.tree_util.Partial(f, out=out)
5         return jax.jacrev(f_out, argnums=1)(W, x).flatten()
6
7     def ts_norm(W, x, out):
8         g = jax.tree_util.Partial(df, x=x, out=out)
9         J_JT_op = lambda v: jax.jvp(g, (W,), jax.vjp(g, W)[1](v))[1]
10        return jnp.sqrt(jnp.trace(jax.vmap(J_JT_op)(jnp.eye(x.size))))
11
12    def scan_fun(carry, x):
13        ts_partial = jax.tree_util.Partial(ts_norm, W, x)
14        return carry + jax.lax.map(ts_partial, jnp.arange(10)), None
15
16    init = jnp.zeros(10)
17    return jax.lax.scan(scan_fun, init, inp)[0] / inp.shape[0]

```

4.3 Results

By Section 4.2 we can efficiently calculate the average norm of the tangent sensitivity matrix on the test set. Note that we need to calculate this value for every epoch during the training process, which can take several hours, even on high-end GPUs. We used NVIDIA A100 40 GB Tensor Core GPUs to perform the experiments. We trained feedforward ReLU networks with 3 hidden layers of different sizes on the MNIST and CIFAR-10 datasets for 60 epochs with squared loss. We used batches of size 32 and the Adam optimizer [29] with a fixed learning rate of 10^{-5} . In the experiments we evaluated the norm of the tangent sensitivity for each individual output neuron of the neural network⁴.

It follows from the proof of Theorem 5 in [14] that the actual bound depends on the term

$$\sum_{t=1}^{T-1} \eta(t) \frac{1}{N} \sum_{i=1}^N \frac{\partial l}{\partial f} \text{TS}(\theta_t, x_i) x_i.$$

However, the empirical results suggest that there is an even stronger connection between the generalization gap and the norm of the tangent sensitivity. It is con-

⁴Both the MNIST and CIFAR-10 datasets have 10 different classes.

jectured that the behavior of the tangent sensitivity around a parameter vector θ is closely related to the generalization ability of the model at θ . Figures 4.1 and 4.2 demonstrates this idea. Note that in Figure 4.2 the learning curve shows signs of the double descent phenomenon [21] and the correlation holds in this case as well.

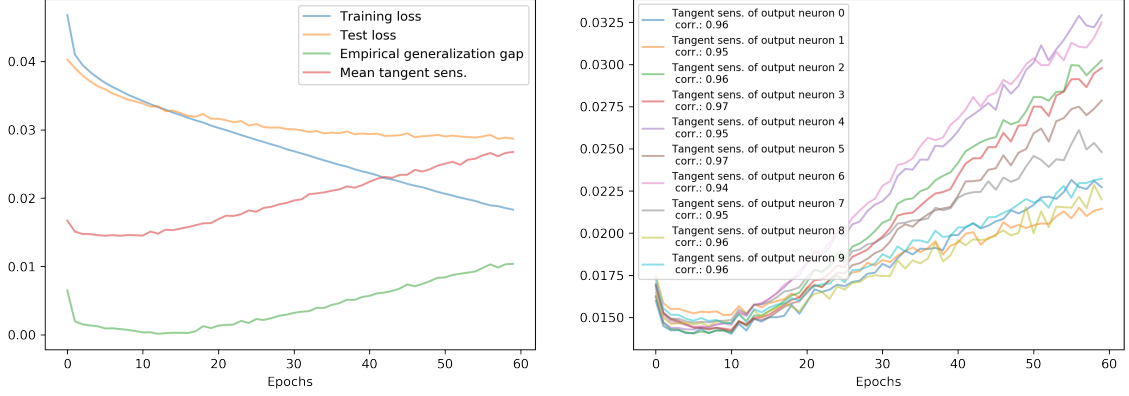


Figure 4.1: Pearson correlation [30] between the empirical generalization gap and the average norm of the Tangent Sensitivity on the test dataset for a 3×3000 -wide fully connected ReLU net trained on CIFAR-10. The y-axes are based on the actual loss values. The norm values of tangent sensitivity were linearly scaled for presentational purposes. Figure and caption from [14].

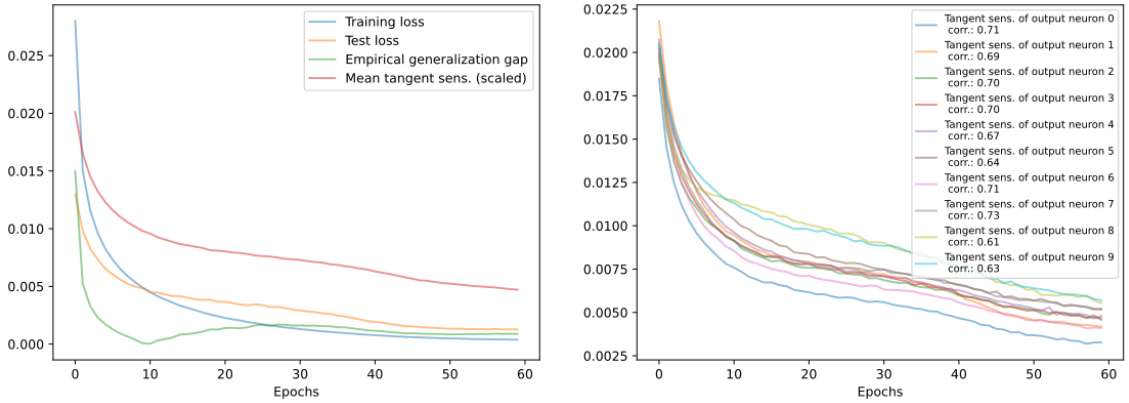


Figure 4.2: Correlation between the empirical generalization gap and the average norm of the Tangent Sensitivity on the test dataset for a 3×3000 -wide fully connected ReLU trained on MNIST. Figure and caption from [14].

Chapter 5

Discussion

In this study, we examined the generalization properties of feedforward ReLU networks using the notion of tangent sensitivity. We showed an efficient way to calculate the Frobenius norm of the tangent sensitivity matrix which plays an important role in the bound in Theorem 2. We also provided empirical evidence that the tangent sensitivity has a strong correlation with the empirical generalization gap of the network. These results indicate that the tangent sensitivity can be used to explain the generalization properties of neural networks in case of wide networks. This gives motivation for further research on this topic.

In Chapter 4, I presented in detail my contribution to our paper [14]. I provided efficient implementations related to the calculation of the norm of the tangent sensitivity of neural networks. The main challenge was to optimize the code to work in the case of overparameterized networks, where the number of parameters is in the order of millions. This implementation was used to carry out the experiments presented in the paper. The empirical results not only revealed some intriguing properties of the tangent sensitivity, but also provided further research ideas.

Note that the bound in Theorem 2 depends on the Frobenius norm of the tangent sensitivity along the optimization path. However, empirical results indicate that its value at some parameter configuration may also be in a tight relationship with the generalization gap at that point. We plan to investigate the connection between the behavior of the tangent sensitivity and the convergence of the optimization algorithm, or the convergence of the tangent sensitivity itself.

We would like to point out that the error term $H(\theta)$ in Theorem 2 depends on the width of the network. In the case of lower width networks, the correlation

between the tangent sensitivity and the generalization gap is not as strong as in case of wide networks. However, the correlation with the per-class loss is still noticeable. See Figure 5.1 for the details.

5.1 Future work

There are plenty of directions to extend the results in this thesis, both theoretically and experimentally. We would like to examine the tangent sensitivity for other architectures, such as convolutional neural networks, ResNet [31], recurrent neural networks, etc. It would also be interesting to examine the tangent sensitivity on other datasets, such as ImageNet [32] and to consider other optimization algorithms, loss functions, and activation functions. We would like to further explore the connection of the tangent sensitivity and the activation regions of ReLU networks and the behavior of these measures in the case of random relabeling of the data, where the network usually memorizes the training set [33, 15]. It would be interesting to investigate the connection of the tangent sensitivity in the infinite width limit to the Neural Tangent Kernel (NTK) Machine [2].

As for the implementation part, we will need to extend the code to support the architectures and datasets described above. Jax and the higher level frameworks built on top of it (e.g. [34]) do not have as much functionality implemented as PyTorch or TensorFlow. However, as described in Section 4.1, Jax has a lot of benefit over these frameworks, so we plan to use it in the future as well.

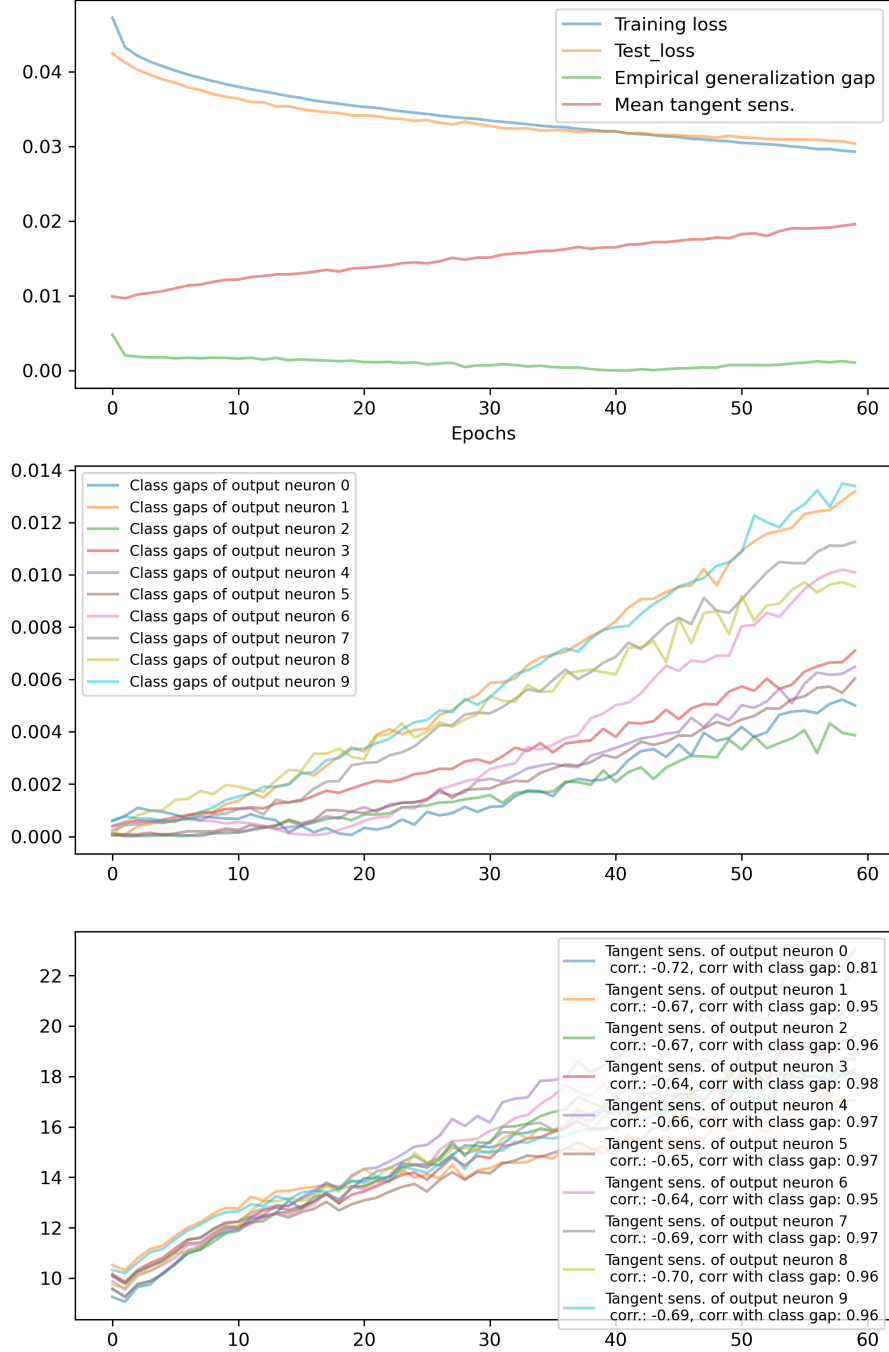


Figure 5.1: Correlation between the empirical generalization gap and the average norm of the Tangent Sensitivity on the test dataset for a 3×700 -wide fully connected ReLU trained on CIFAR-10. Figure and caption from [14].

Acknowledgements

This research was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

I would like to thank the Informatics Laboratory at HUN-REN SZTAKI¹ and the Eötvös József College for Advanced Studies² for their support during my work.

¹<https://sztaki.hun-ren.hu/en/science/departments/infolab>

²<https://eotvos.elte.hu/>

Bibliography

- [1] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples”. In: *arXiv preprint arXiv:1412.6572* (2014).
- [2] Arthur Jacot, Franck Gabriel, and Clément Hongler. “Neural tangent kernel: Convergence and generalization in neural networks”. In: *Advances in neural information processing systems* 31 (2018).
- [3] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. “Gradient descent provably optimizes over-parameterized neural networks”. In: *arXiv preprint arXiv:1810.02054* (2018).
- [4] Chaoyue Liu, Libin Zhu, and Mikhail Belkin. “Loss landscapes and optimization in over-parameterized non-linear systems and neural networks”. In: *Applied and Computational Harmonic Analysis* 59 (2022), pp. 85–116.
- [5] Chiyuan Zhang et al. “Understanding deep learning (still) requires rethinking generalization”. In: *Communications of the ACM* 64.3 (2021), pp. 107–115.
- [6] Gal Vardi. “On the implicit bias in deep-learning algorithms”. In: *Communications of the ACM* 66.6 (2023), pp. 86–93.
- [7] Spencer Frei, Gal Vardi, Peter L Bartlett, and Nathan Srebro. “The Double-Edged Sword of Implicit Bias: Generalization vs. Robustness in ReLU Networks”. In: *arXiv preprint arXiv:2303.01456* (2023).
- [8] David A McAllester. “PAC-Bayesian model averaging”. In: *Proceedings of the twelfth annual conference on Computational learning theory*. 1999, pp. 164–170.
- [9] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. “Spectrally-normalized margin bounds for neural networks”. In: *Advances in neural information processing systems* 30 (2017).

- [10] Luciano Floridi and Massimo Chiriatti. “GPT-3: Its nature, scope, limits, and consequences”. In: *Minds and Machines* 30 (2020), pp. 681–694.
- [11] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [12] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).
- [13] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: <http://github.com/google/jax>.
- [14] Dániel Rácz, Mihály Petreczky, **András Csértán**, and Bálint Daróczy. *Optimization dependent generalization bound for ReLU networks based on sensitivity in the tangent bundle*. 2023. arXiv: 2310.17378 [cs.LG].
- [15] Bálint Daróczy. “Gaussian perturbations in relu networks and the arrangement of activation regions”. In: *Mathematics* 10.7 (2022), p. 1123.
- [16] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [17] Vladimir Vapnik. “Principles of risk minimization for learning theory”. In: *Advances in neural information processing systems* 4 (1991).
- [18] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 1999.
- [19] *TikZ Neural network examples*. https://tikz.net/neural_networks/. Accessed: 2023-11-27.
- [20] Meenal V Narkhede, Prashant P Bartakke, and Mukul S Sutaone. “A review on weight initialization strategies for neural networks”. In: *Artificial intelligence review* 55.1 (2022), pp. 291–322.
- [21] Mikhail Belkin. “Fit without fear: remarkable mathematical phenomena of deep learning through the prism of interpolation”. In: *Acta Numerica* 30 (2021), pp. 203–248.
- [22] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: <http://yann.lecun.com/exdb/mnist/>.

- [23] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. MIT and NYU, 2009.
- [24] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [25] Amit Sabne. “Xla: Compiling machine learning for peak performance”. In: (2020).
- [26] *Jax, The Autodiff Cookbook*. https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html. Accessed: 2023-11-30.
- [27] *Efficient way to compute Jacobian x Jacobian.T*. <https://stackoverflow.com/questions/63559139/efficient-way-to-compute-jacobian-x-jacobian-t>. Accessed: 2023-11-28.
- [28] *JAX vs NumPy - Benchmark*. <https://www.tensorops.ai/post/should-i-switch-from-numpy-to-jax-1>. Accessed: 2023-12-01.
- [29] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [30] *SciPy, Pearson correlation coefficient*. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html>. Accessed: 2023-12-01.
- [31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [32] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.
- [33] Boris Hanin and David Rolnick. “Deep relu networks have surprisingly few activation patterns”. In: *Advances in neural information processing systems* 32 (2019).
- [34] Jonathan Heek et al. *Flax: A neural network library and ecosystem for JAX*. Version 0.7.5. 2023. URL: <http://github.com/google/flax>.