Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor Thesis

Neural Network Learning Visualization

Alikhan Anuarbekov

Supervisor: Ing. Vladimír Kunc

May 21, 2021

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Anuarbekov** Jméno: **Alikhan** Osobní číslo: **483420**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vizualizace učení neuronových sítích

Název bakalářské práce anglicky:

Neural network learning visualization

Pokyny pro vypracování:

Učení neuronových sítí a podobných modelů s vysoko-dimenzionální parametrizací je často velmi neprůhledná úloha – dva modely schopné naučit se stejnou rozhodovací funkci se mohou učit zcela odlišně v závislosti na rozdílné vnitřní parametrizaci. Cílem bakalářské práce je proto provést rešerši stávajících metod vizualizací učení a obtížnosti učení neuronových sítí a re-implementovat vhodnou metodu popsanou v literatuře (případně navrhnout rozšíření stávající či vlastní). Dále je cílem implementovanou vizualizační metodu aplikovat na prozkoumání hypotéz, že některé stávající architektury popsané v literatuře dosahují lepších výkonů právě díky snadněji optimalizovatelné krajině. Student se zaměří zejména na skip-connection u sítě ResNet a dále na transformativní adaptivní aktivační funkce (TAAF). Výstupem práce je jak teoretická rešerše, tak praktická implementace prototypu metody spolu s experimenty používající implementovanou metodu ke zkoumání problematiky učení neuronových sítí.

Seznam doporučené literatury:

1. Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. NIPS, 2018.
2. Dianne Cook and German Valencia. A Slice Tour for Finding Hollowness in High-Dimensional Data. Journal of Computational and Graphical Statistics, Volume 29, 2020.
3. Ian J. Goodfellow, Oriol Vinyals. Qualitatively characterizing neural network optimization problems. ICLR 2015
4. Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry Vetrov, and Andrew Gordon Wilson. 2018. Loss surfaces, mode connectivity, and fast ensembling of DNNs. NIPS. 2018.
5. Stanislav Fort, Stanislaw Jastrzebski. Large Scale Structure of Neural Network Loss Landscapes. NeurIPS, 2019.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Mgr. Vladimír Kunc, Intelligent Data Analysis FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **04.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Ing. Mgr. Vladimír Kunc podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 21. května 2020 ....

Acknowledgments

I want to thank my supervisor, Ing. Kunc, for his help in my work throughout the semester. I am also grateful to prof. Železný for the proposal of the topic of the work. Additionally, I would like to thank CESNET Metacentrum for the offer of the computing power for my practical part of the work. Finally, I want to thank my family and friends for their support.

Abstrakt

Tato bakalářská práce se zabývá analýzou a vizualizací učení neuronových sítí ve formě ztrátové funkce. První část této práce je teoretický základ hlubokého učení. Druhá část je zkoumání povrchu ztrátové funkce v závislosti na architektuře neuronové sítě.
Hlavním zaměřením bakalářské práce jsou feedforward networks, zejména jejich použití v úlohách klasifikaci obrázku a regrese. Korektnost a zobecnitelnost algoritmů jsou měřeny pomocí validační přesnosti, a potom jsou porovnány s tvarem povrchu ztrátové funkce. Bylo zjištěno, že zkoumané metody na vizualizaci povrchu ztrátové funkce jsou vhodné jen pro nalezení významných chyb, jako je špatná volba architektury sítě nebo typu ztrátové funkce, ale zároveň jsou neefektivní pro vizualizaci pokročilých metod jako je zřetězování DenseNetu.

Klíčová slova: neuronové sítě, hluboké učení, vizualizace, ztratová funkce

Abstract

This bachelor thesis's subject aims to visualize and analyze neural networks' training as a loss function. The first part of the thesis is the theoretical basis of deep learning. The second part is researching the loss surface shape depending on the deep learning network architecture.
The thesis primarily focuses on feedforward neural networks, particularly on their usage in image classification and regression tasks. The generalizability is measured in the form of validation accuracy and is compared to the loss surface curvature. It was found that the examined loss surface visualization methods are suitable for significant problem detection, such as lousy architecture or loss function choice, yet are ineffective for the evaluation of the more advanced method such as DenseNet concatenating.

Keywords: neural network, deep learning, visualisation, loss function

Contents

Introduction

1 Machine learning **2**
1.1 Definition . . . 2
1.2 Training process . . . 3
1.3 Optimizer . . . 3
1.3.1 Gradient Descent . . . 3
1.3.2 Adam optimizer . . . 4

2 Neural Networks **5**
2.1 Perceptron . . . 5
2.2 Activation function . . . 6
2.2.1 Static activation functions . . . 6
2.2.2 Adaptive activation functions . . . 7
2.3 Backpropagation . . . 7
2.4 Cost function . . . 8
2.5 Convolutional Neural Networks . . . 8

3 Advanced Neural Network Techniques **10**
3.1 Gradient Explosion and Vanishing . . . 10
3.2 Batch normalisation . . . 10
3.3 Skip connections and Residual learning . . . 11
3.4 Densely Connected CNN . . . 12

4 Visualisation **13**
4.1 Grand Tour visualisation . . . 13
4.2 Loss Surface Visualisation in General . . . 13
4.3 Filter-wise normalized directions . . . 14
4.4 Visualisation with PCA components . . . 15
4.5 Use of methods . . . 15
4.5.1 PCA components . . . 15
4.5.2 Random trajectories . . . 16
4.6 Sharp vs Flat discussion . . . 17

5 Implementation and methods **18**
5.1 Dataset overview . . . 18
5.2 Experiments . . . 18
5.3 Visualisation methods . . . 19
5.4 Models overview . . . 19
5.5 Visualization details . . . 19

6 Experimental part **20**
6.1 ResNet110, Random trajectories with Filter normalization . . . 20
6.2 ResNet110, Validation dataset . . . 23
6.3 ResNet50 . . . 24
6.3.1 ResNet50, Training . . . 24

6.3.2 ResNet-50 PCA visualization . . . 24
6.3.3 ResNet-50(noskip) Training . . . 25
6.3.4 ResNet-50(noskip) PCA visualization . . . 25
6.3.5 ResNet50, Random trajectories with Filter normalization . . . 26
6.4 ResNet56, Random trajectories with Filter normalization . . . 28
6.5 DenseNet, Cifar/ImageNet . . . 30
6.6 ResNet50/DenseNet121 with ImageNet . . . 32
6.7 TAAF, Preparations . . . 34
6.8 TAAF, Preparations 2 . . . 35
6.9 TAAF vs Plain, 3 layers, 6000 neurons . . . 36
6.10 TAAF vs Plain, 3 layers, 9000 neurons . . . 38
6.11 Brief overview . . . 41

7 Conclusion **43**

8 Appendix **50**
8.1 Pytorch library . . . 50
8.2 Tensorflow library . . . 51
8.3 Used platforms . . . 52
8.4 High-Level overview . . . 52

Introduction

Motivation

The question of Artificial intelligence existed for thousands of years and it is one of the fundamental parts of materialist philosophy. We can find a giant constructed of bronze, depicting a human-made artificial life, in Ancient Greek mythology[40][35], as well as the robots from Karel Čapek's R.U.R.

René Descartes analyzed the difference between the artificial machine and human consciousness in his work [42]. He stated:
"If touched in a particular part it may ask what we wish to say to it; if in another part it may exclaim that it is being hurt, and so on. But it never happens that it arranges its speech in various ways, in order to reply appropriately to everything that may be said in its presence, as even the lowest type of man can do."
Therefore, he formulated the central question of today's discussions on Strong and Weak A.I.

However, the actual starting point of machine intelligence can be considered the 1950s with Alan Turing's proposition of the Turing Test, a replacement for the philosophical question "Can machines think?"[6].

Today, after 70 years of progress, the A.I. has become a significant part of our life. Started from a camera tracking field, machine learning is making its way to the more complex tasks. Car driving, face recognition, and more automatized fields are occupied by it[30]. Even though we are far from the "Strong A.I.", this field's progress is gaining momentum, and the discussions are more relevant than ever[38].

One of the modern artificial intelligence paradigms is the Deep neural network architecture that made its way to the top in 2006 with the state-of-art solution to the image classification task. Since then, neural networks have achieved further success and became one of the primary disciplines in the modern machine learning field[17][18].

However, the process of training a neural network often gives unexpected results. The architecture and parametrization cause a massive impact on network behavior. Commonly, we observe a vast difference between their approaches to solve a problem, even if their decisions are nearly the same. To understand the cause of this behavior, we will dive into the network's training process. More precisely, we will analyze the loss function - the foundation stone of the training process and compare the loss function's surface of the different architectures.

We will cover the theoretical basics, the training process, including some advanced techniques, and the ways of visualizing it in detail. After that, we implement two loss surface visualization methods and test the original papers' statements on these methods.

Chapter 1

Machine learning

The *neural network* algorithms are a part of a larger **Machine Learning** field. Because of that, we should firstly dive into the machine learning field, and step by step, come to the neural networks.

1.1 Definition

What is machine learning?

The definition is:
*"Machine Learning Algorithm is capable of learning from its **Experience(E)** towards a **Task(T)**, and its **Performance(P)** improves with an **Experience(E)**."* [22]

- **Example** is data received from the real world. (Image, Audio, etc.)
- **Features** are a valuable part of an Example data. (Pixels, Audio waves, etc.)
- **Result** is an output that the algorithm is supposed to return. (Image description, text form of an Audio, etc.)

Task T

Task is a process of analyzing an *Example* and calculating a *Result* based on *Features*.
The primary types of *Task* analyzed in this paper are:

- **Regression** - Approximation of a continuous function defined by points.[34]
  The goal is - for a given input data vector, return an output data vector.
- **Classification** - Classification of an image from the label set[52].
  The goal is - for a given input data vector, return an output labels' probabilities.

Performance P

Performance is an explicitly defined function representing an error or a success towards the *Example* and its correct *Result*. Practically it is represented as a scalar function, and it is called a **loss function** or **cost function**.

Experience(E) and Dataset

We divide the entire dataset of examples into three categories:

- **Training set** - examples for training purposes
- **Test set** - examples to calculate the performance
- **Validation set** - examples to optimize hyper-parameters(see section 1.2).

Experience is a knowledge of an algorithm after analyzing a collection of training examples.

1.2 Training process

Every machine learning algorithm is being trained in the following steps:

1. Start with some *Experience*
2. Analyze a given *Example*
3. Improve our *Experience* based on *Performance* loss function
4. Repeat

Hyperparameters

Even though the algorithm can learn on its own, we should define a structure that cannot be changed during the training process. These settings are called **Hyperparameters**. They can also be optimized, but our algorithm cannot perform this, and we would need external interference[49]. For example: type of **loss function**, number of **neurons in network**(see chapter 2), **gradient step size**(see section 1.3.1), etc.

Parameters

Everything changeable during the process of training is a part of parameters. Therefore, every single parameter is considered as a variable of a composite function.
Based on this approach, the optimization can be defined.

1.3 Optimizer

Below is the detailed description for the third step of the training process (see section 1.2) for the scalar functions.

1.3.1 Gradient Descent

One of the key methods to find the function's local minimum is the **Gradient Descent**.

Gradient

The gradient is defined as a vector of a function's partial derivatives:

$$\nabla_{\vec{x}} f(\vec{x}) = (\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n})^T, \quad \vec{x} \in R^n$$

We know that the *gradient* represents a slope of the function or, in other words, the direction of the most rapid increase of function's value based on its linear approximation in a given point.

Negative Gradient direction

So, to certainly decrease the value of the function, we can go in the opposite direction:

$$\vec{x}_{\mathbf{new}} = \vec{x} - \epsilon \nabla_{\vec{x}} f(\vec{x})$$

ϵ = length of a step

The length of a step should be optimized as a hyperparameter since, at a close distance to the local minimum, it can step over it and possibly converge to another local minimum.

By repeating these steps, we will reach a point near the local minimum, which has better performance.
In our case, we should calculate this direction across all the input examples.

Stochastic Gradient descent

However, the problem is that if we calculated the gradient across all thousands of examples, it would take a long time to train our algorithm.
Because of that we divide our training examples into the **mini-batches** $B = \{Input^{(i)}, Input^{(i+1)}, ...Input^{(i+m)}\}$, where **m** can be any small number. The new approach is:

1. Calculate the direction for a chosen mini-batch
2. Move the point
3. Take the next mini-batch

That way, it would minimize it not necessarily the shortest way, but it would take significantly less time to calculate the direction.

1.3.2 Adam optimizer

Adam optimizer is an SGD with an adaptive gradient and step size choosing[16]. The main difference is the form of the minibatch gradient applying. Instead of sequentially applying gradients, it calculates **momentum**, or, in other words, the **Exponentially Weighted Average(EWA)** of the previous gradients to approximate the actual gradient of the total dataset.

Let us say we have **t** minibatches with indices 1 to t.

$$\vec{g}_t = \nabla_{\vec{x}} f_t(\vec{x}) = \text{minibatch gradient(with index t)}$$

$$\vec{m}_t = \beta_1 \vec{m}_{t-1} + (1 - \beta_1)\vec{g}_t = \text{momentum of the mean}$$

$$\vec{m}_t = \frac{\vec{m}_t}{(1 - \beta_1^t)} = \text{descaling to correct } \beta_1 \text{ scaling}$$

$$\vec{v}_t = \beta_2 \vec{v}_{t-1} + (1 - \beta_2)\vec{g}_t^{\,2} = \text{momentum of the variance}$$

$$\vec{v}_t = \frac{\vec{v}_t}{(1 - \beta_2^t)} = \text{descaling to correct } \beta_2 \text{ scaling}$$

And finally:

$$\text{momentum gradient of the minibatch} = \vec{G}_t = \frac{\vec{m}_t}{\sqrt{\vec{v}_t} + 0.000001^*}$$

So, we just apply:

$$\vec{x}_{\mathbf{new}} = \vec{x} - \epsilon \vec{G}_t$$

ϵ = length of a step
β_1, β_2 = constant hyperparameters

* - added to prevent zero division

As the Machine learning introduction is given, we can define a more concrete Machine learning algorithm type, neural networks, in the next chapter.

Chapter 2

Neural Networks

This chapter dives deeply into the neural networks, one of the most used machine learning algorithm nowadays. Main source is a book [18].
Although the neural networks are not limited by feedforward networks only, this thesis focuses mainly on them. The reason for this is the upcoming experiments that focus on the classification and profile reconstruction of gene expression tasks that are primarily solved by feedforward networks.

Feedforward networks

The *feedforward networks'* main idea is that they approximate some function f^*, which maps every input $\vec{x}$ to a category y.

- The algorithm defines a function: $y = f(\vec{x}, \vec{\theta})$
  where $\vec{\theta}$ is a set of a parameters.
- The *feedforward* means that the input $\vec{x}$ is going through the intermediate computations and finally becomes an output y.
- The *network* means that function can be written in the form of a chain of functions:
  $f(\vec{x}) = f^{(1)}(f^{(2)}(f^{(3)}(\vec{x})))$

  This representation allows us to build a non-linear function f and divide all subfunctions into the layers.

2.1 Perceptron

Perceptron is one of the first artificial neuron structures. Each perceptron can be viewed as the intermediate function that creates a *feedforward network*.

- A perceptron is an object that holds some real **value** in $\langle 0, 1 \rangle$ bound.
- It can have **inputs** $x_1, x_2, x_3, ...$ in $\langle 0, 1 \rangle$ bound.
- Each input is associated with some real **weight** parameter $w_1, w_2, w_3, ...$ that was predefined for perceptron.
- It can connect to other perceptrons, so its **value** is passed as **output** to the connected perceptrons.
- Originally perceptron was defined that way:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

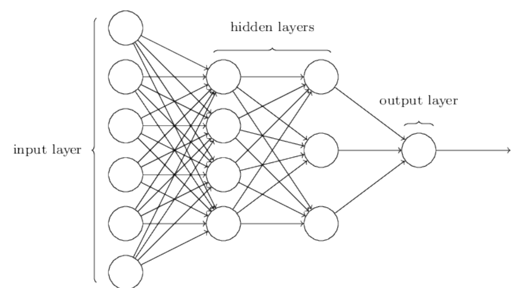


Figure 2.1: Neural Network made of artificial neurons
(Taken from [18])

This function is called the **Step function**, and the **threshold** parameter is any real value predefined for perceptron.

We divide all artificial neurons into 3 layers:

Input layer - neurons without connected input. In practice, these neurons represent input *example* data. [22]
Output layer - neurons without connected output. In practice, these neurons represent network's *result*.
Hidden layer - other neurons. In practice, these neurons represent network's hidden heuristic. These neurons can be further divided into several layers, as it is shown in figure 2.1.

2.2 Activation function

However, the problem is that the **Step function** is not continuous. Thus, we can hardly calculate the derivative of the network's final output, especially if we connect multiple perceptrons into the network.

Because of that, we want to generalize the definition of the perceptron:

- The artificial neuron will calculate its output:

$$\text{output} = f(\vec{w} \cdot \vec{x} + b) \tag{2.1}$$

$\vec{w}$ = vector of weights, $\vec{x}$ = vector of inputs, $b = -threshold$ = **bias**, $f(\dot{})$ = **activation function**, $\vec{z} = \vec{w} \cdot \vec{x} + \vec{b}$ = weighted input vector

2.2.1 Static activation functions

One of the most used *activation functions* are the linear **sigmoid function**;

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{2.2}$$

the **hyperbolic tangent** and the non-linear **ReLU function**:

$$\sigma(z) = tanh(z) \qquad \sigma(z) = max\{0, z\}. \tag{2.3}$$

LReLU(Leaky Rectified Linear Unit) was proposed in recent studies[10] and was shown to be superior to the original ReLU due to the linearity on both sides[10]:

$$\sigma(z) = \begin{cases} z, & \text{if } y \leq 0 \\ az, \quad a \in (0,1) & \text{if } y < 0 \end{cases}$$

On top of that, **ELU(Exponential Linear Unit)** activation shows significant advantages compared to both ReLU and LReLU. Faster convergence rate and competitive results on the core datasets such as ImageNet and Cifar make this activation a considerable alternative to the common **ReLU + Batch Normalization**(see 3.2) combination[20]:

$$\sigma(z) = \begin{cases} z, & \text{if } y \leq 0 \\ az, \quad a \in (0,1) & \text{if } y < 0 \end{cases}$$

Moreover, we consider using the **Softmax function** in the network's *output layer* to convert it into the labels' probability.:

$$softmax(z_i) = \frac{e^{cz_i}}{\sum_j e^{cz_j}}, \qquad c = \text{predefined constant} \tag{2.4}$$

2.2.2 Adaptive activation functions

However, the activation functions are not necessary non-parametrized as they were presented in the previous cases. The activation function's parametrization implies that we train the parameters through Gradient Descent, and, thus, it becomes an **Adaptive activation function**.

For example, paper [13] proposes a variation of a **Leaky ReLU function** with a trainable parameter instead of a constant and achieves a human-level surpassing performance on ImageNet. Similarly, the [3] proposes an **auto-tuning sigmoid** to control the shape of the activation function.

Transformative adaptive activation function

The primary adaptive function analyzed in this thesis is the **Transformative adaptive activation function (TAAF)** [48]:

Assume that we plan to apply a TAAF to every neuron in a layer:

$\vec{y} =$ output of the layer and input of the TAAF, $f(\cdot) =$ basic non-parametrized activation function

Then, we define TAAF as follows:

$$g(f, \vec{y}) = \alpha \cdot f(\beta \cdot \vec{y} + \gamma) + \delta$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ are trainable parameters added to a neuron

2.3 Backpropagation

Suppose we consider our neural network to be a combination of the artificial neurons with any activation function. In that case, the *gradient* in the **Optimization step**(see section 1.2) will be calculated this way[26][2]:

Assume that the neural network's output error is the Cost function2.4 composed of the layers or, in other words, from all the network parameters and input vector $\vec{x}$..

$$C = \text{cost function} = C(w_1, ..., w_n, b_1, ..., b_k, .., \vec{x})$$

As we mentioned in the 1.3.1, the gradient should be computed as a partial derivative vector. Therefore, the partial derivative will have a form of:

$$\Delta C = (\frac{\partial C}{\partial w_1}, ... \frac{\partial C}{\partial w_n}, ..., \frac{\partial C}{\partial b_k}, ..)_{|\vec{x}}$$

Finally, since we consider the feedforward networks only, the layers are applied in a specific order.

$$C(w_1, ..., w_n, b_1, ..., b_k, .., \vec{x}) = C(l^{(1)}(l^{(2)}(....l^{(p)}(\vec{x}))))$$

Consequently, we can use the composed function formula to compute the derivative of the functions of the layers between the partial parameter layer and the output:

$$y_j = \text{output of the layer } l^j$$

$$p = \text{last layer index}$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial \vec{y}_p} \cdot \frac{\partial \vec{y}_p}{\partial w_i} = \frac{\partial C}{\partial \vec{y}_p} \cdot \frac{\partial \vec{y}_p}{\partial \vec{y}_{p-1}} \cdot \frac{\partial \vec{y}_{p-1}}{\partial w_i} = \frac{\partial C}{\partial \vec{y_p}} \cdot \frac{\partial \vec{y_p}}{\partial \vec{y}_{p-1}} \cdot ... \cdot \frac{\partial \vec{y}_i}{\partial w_i}$$

In the end, every mentioned partial derivative can be defined in a modular way.
The name **"Backpropagation"** implies the application of the derivatives from the last to the first layers.

2.4 Cost function

As we introduced the **cost function** in section 1.1, now we will see which ones are the most useful in modern neural networks.

One of the most commonly used functions is the **Mean Squared Error(MSE)**:

$$MSE = \frac{1}{2} \cdot \frac{1}{m} \sum_{i=1}^{m} (estimation_i - real_i)^2$$

$\overrightarrow{estimation}$ - calculated result, $\overrightarrow{real}$ - correct result, m - number of test examples

We use $\frac{1}{2}$ for future gradient purposes

The MSE function is primarily used in the regression tasks, and often the **Cross-Entropy** is preferred in the classification task case:

$$C = -\frac{1}{m} \sum_{i=1}^{m} [real_i \ln(estimation_i) + (1 - real_i) \ln(1 - estimation_i)]$$

The reason behind this was studied in research [9] that clearly shows that the MSE is worse than Cross-Entropy in the case of poorly initialized weights.

Finally, we will introduce the **Mean Absolute Error(MAE)**, the cost function used essentially in regression tasks, particularly in the TAAF experiments in the forthcoming chapters6.50.

$$MAE = \frac{1}{m} \sum_{i=1}^{m} (estimation_i - real_i)$$

Since the TAAF experiments' dataset belong to the vector-to-vector[39] problem type, paper [46] shows that the MAE gives lower loss values.

2.5 Convolutional Neural Networks

It is important to note that usually images are analyzed by a specific neural network architecture called **Convolutional Neural Networks**.
It has three main concepts that allows us to get the highest results within small amount of time on images:

- **Local receptive fields** = every neuron from hidden layer gets an input from a small region of previous layer and analyses the data locally. See figure 2.2
- **Shared weights and biases** = every neuron in each hidden layer has the same weights and biases as other neurons in it.
- **Pooling layers** = on the top of two concepts above, we will use another type of hidden layer, which is used to compress and re-scale the data from previous hidden layer. See figure 2.3

Locality and Shared parameters

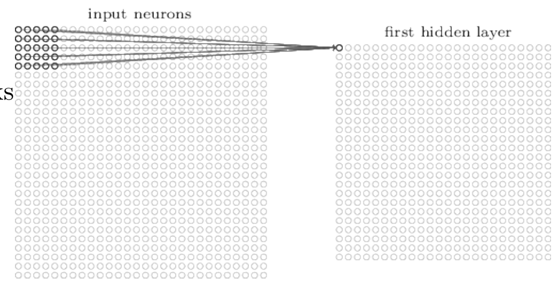


Figure 2.2: Local receptive fields example (Taken from [18])

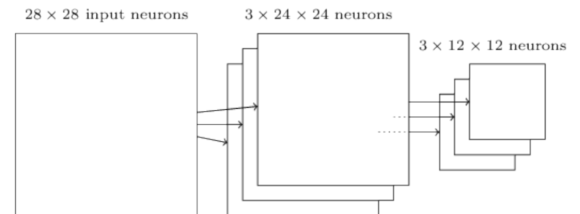


Figure 2.3: Pooling layer example (Taken from [18])

In a fully connected layer, we would have a parameter between each input pixel and output neuron. Convolutional Neural networks solved this problem by sharing the same weights across all neurons in the *convolutional layer*. The weights of the layer are called the **filter**.

Since an image is a depiction of an object, we can assume that pixels far away have nearly zero connection between themselves. Because of this, the calculations and analysis of the entire image at once is too complex for one layer to accomplish.

As an example, see picture 2.2.
The filter is used to extract features from the image by applying it to every 5x5 area. That way, it can analyze the local field of pixels and determine the simple features from this area. Then, when all the features for every area are gathered, the resulting output image will represent the feature's spatial location.
Finally, the Convolutional layer's essential superiority is its spatial invariance - since we use the same parameters, the feature's detection does not depend on its position.

Intermediate images

As we can see from figure 2.2, one hidden layer of convolutional neural network can be seen as another image and thus can be re-scaled to simplify its content. We can implement this operation as another layer called the **Pooling layer**.

This approach is especially handful if the resulting output is much simpler than the number of pixels on the image. For example, if our goal is to determine the digit from the image, we can do so even without using all pixels, but just by calculating the maximum or average of the pixel area[18], [37].

Chapter 3

Advanced Neural Network Techniques

3.1 Gradient Explosion and Vanishing

From [4], we can define a problem in the backpropagation process:

Under certain conditions[32], the gradient tends to either:

- **Explode** = first layers' weights become too large and tend to oscillate
- **Vanish** = first layers' weights become too small, and the backpropagation takes a prohibitive amount of time or does not find it at all.

The type of backpropagation error exponentially depends on the size of the weights.

3.2 Batch normalisation

As stated in [15], with the rapid change of the network's input and parameters, it is hard to maintain control over the training process.

With the change of the parameters, the distribution of data passed to the next layers changes. As a result, every layer should adapt to new inputs, even though they are just scaled old ones. This problem is called a **covariate shift**[15]. In order to solve this problem, we can normalize the input of each layer. This normalization is called **Batch Normalization**[15], [31]:

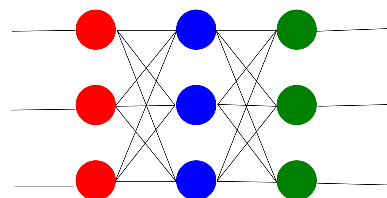


Figure 3.1: Example of neural network

For a layer of neural network, which takes an d-dimensional input $\vec{x} = (x^{(1)}, x^{(2)}, ..., x^{(d)})$, we will normalize each dimension:

$$x^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

where **E** = **mean** and **Var** = **variance** are computed throughout the whole training set.

As an example - in figure 3.1 the blue layer will consider the red layer as input $\vec{x}$.

That way, we can significantly speed up the convergence, but it is impractical to calculate a global mean and variance. Because of that, we use **Batch Normalizing Transform** with empirical data from mini-batch[15],

[31]:

For a mini-batch $B = (x_1, ..., x_m)$:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i = \textbf{empirical} \quad \textbf{mean}$$

$$\sigma_B = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B) = \textbf{empirical} \quad \textbf{variance}$$

$$Xnew_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B + \epsilon}} = \textbf{new} \ \ \textbf{normalized} \ \ \textbf{dimension} \tag{3.1}$$

where ϵ is a small arbitrary regulatory number.
The 3.1 equation will be calculated for every $Xnew_i^{(k)}$ separately.

After these operations, we add two more parameters, γ and β, to our layer so that it can detect the distribution change across minibatches and not move the optimal weights[15], [31]:

$$y_i = \gamma Xnew_i + \beta = BatchN_{\gamma,\beta}(X_i)$$

where y_i is an output passed to the next layer. Parameters γ and β are trained as well as weights and biases.
That way, we will converge faster and not make a *covariate shift* mistake. Moreover, since we have placed our values in [0, 1] bound, we have solved the *Gradient explosion* case.

3.3 Skip connections and Residual learning

The network structure we discussed previously can be seen as a **plain network**, which means that every layer communicates with its neighbor layers only. However, *the Gradient Explosion and Vanishing* problem is being caused due to this sequential applying of the functions.

To solve this, we can connect the non-neighbor layers in order for them to communicate without the precision loss between them.
That way, we would use a new structure of neural networks called **Residual Network(ResNet)**[14][50].

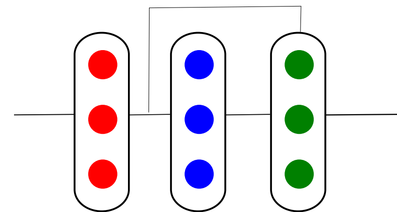


Figure 3.2: Example of the Skip connection

In a plain network we are computing an input sequentially:

$$\vec{x}^{l+1} = \sigma(W^{l+1} * \vec{x}^{l} + \vec{b}^{l+1})$$

where
$\vec{x}^{\mathbf{l}}$ = input from previous layer,
$\mathbf{W}$ = weight matrix, $\vec{b}$ = bias vector,
σ = activation function

But in the **ResNet** we will compute it:

$$\vec{x}^{l+1} = \sigma(W^{l+1} * \vec{x}^{l} + \vec{b}^{l+1})$$

$$\vec{z}^{l+2} = W^{l+2} * \vec{x}^{l+1} + \vec{b}^{l+2}$$

$$\vec{x}^{l+2} = \sigma(\vec{z}^{l+2} + \vec{x}^{l})$$

for every 2 layers of the network, where we change only the last input. Similarly, it can be done for larger distances.
For an illustration see figure 3.2, where the Red layer is standing for L-th layer.

As a result, the ResNet networks are not suffering from the *Gradient Vanishing*[14][50].

3.4 Densely Connected CNN

The last network architecture used in experiments is the Densely Connected Neural Networks(DenseNets)[23]. Before we dive into the architecture theory, we should add some knowledge to the Convolutional layers from section 2.5.

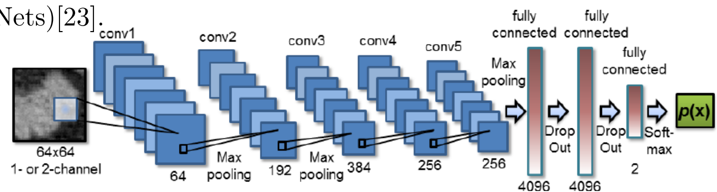


Figure 3.3: Example of Convolutional Network (Taken from [19])

For example, imagine the input to our network - typically, it is an RGB image, e.g., 2D matrix divided into three channels or, in other words, filters.

Now, let's say that we want to extract various features that from this image. If we were to use the method as we know it, we should have built a network with multiple convolutional or even multiple fully connected layers. However, this is an over-parametrized approach. The natural convolution layers are using multiple filters - stacking many convolution layers and using the same input for all of them. As a result, we get many features with minimum parameters used. Finally, the Densely Connected Networks' main idea is combining the Residual

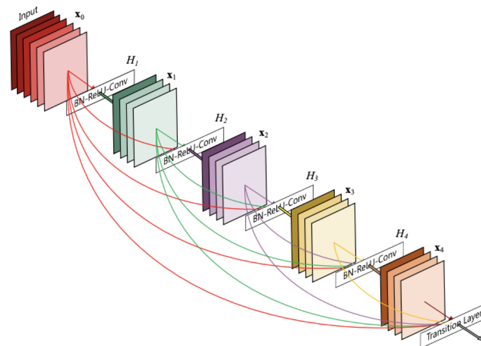


Figure 3.4: Building block of DenseNet
(Taken from [23])

Network approach with the Convolution stacking technique. But, unlike in ResNets, we are concatenating them instead of summing - which means that we are stacking all filters from previous layers and not changing them. As a result - we increase layer size, but not as high as parameters count grow in ResNet.

Chapter 4

Visualisation

This chapter is a short review of the different neural network training visualization methods.

4.1 Grand Tour visualisation

The first approach in the Neural Network training visualization is the change of the output values during the training[7].

Let the output data is a set of **K** label probabilities, which is a **P**-dimensional vector. To represent it in a low-dimensional 2D space, we should take a vector $\vec{\mathbf{O}}$ as the origin and then choose two base vectors.

To continuously choose the proper parameters, the author proposes a method of rotating all 2D subspaces[7]. Another way is to choose the parameters manually depending on a particular strategy[5]:

- *Equal division of 2D space between **K** labels*:
  The origin will be the vector with equal probabilities and two vectors are chosen from the set of possible positions.
- *Linear techniques(PCA components)*:
  Analyzed in 4.4
- *Non-Linear techniques(Sammon's projection)*[1]:
  Defining a cost function that tries to preserve the inter-label distances. The algorithm minimizes it with the Gradient descent.

Figure 4.1: Example of Grand Tour visualization
(Taken from [47])

As an example, see figure 4.1, where the different color represents different labeling of the data. We can see the geometrical arrangement of the labels.

4.2 Loss Surface Visualisation in General

The second approach is to visualize the loss function's surface, which is more significant to the training process. The reason behind this is straightforward - backpropagation plays a crucial role in the training process because it defines the gradient and, therefore, the training path of the parameters.

As we have mentioned in section 2.3, the backpropagation is made with respect to the weights and biases. These parameters create a very high-dimensional vector θ of variables, which is impossible to visualize in its original dimension.
Because of that, we have several approaches to the visualization of the loss function.

1D visualisation

One of the straightforward approaches is to analyze the surface between two vectors of parameters: $\vec{\theta}$ and $\vec{\theta}^*$. We can do so by defining a surface function[27][12]:

$$f(\alpha) = (1 - \alpha)\vec{\theta} + \alpha\vec{\theta}^* \tag{4.1}$$

representing all points across the line between two vectors. Then, we can draw a graph with an α on one axis and the value of the loss function in the $f(\alpha)$ point on the other, e.g. $y = \text{CostFunction}(f(\alpha))$

However, this approach does not have enough expression power to prove our hypothesis about the surface curvature fully. It was proved in [27] that the sharpness/smoothness(see 4.6) of the 1D curve is misleading and can be easily reshaped using simple normalization methods. Thus, we should use a more advanced visualization technique.

2D visualisation

To visualize a surface in the 2-dimensional space, we must choose an origin vector $\vec{\theta}^*$ and two directions $\vec{\delta}$ and $\vec{\eta}$. After that, we can define a function to get all points on the 2D surface around the origin[27][12]:

$$f(\alpha, \beta) = (\vec{\theta}^* + \alpha\vec{\delta} + \beta\vec{\eta}) \tag{4.2}$$

After that, we can set two axes to α and β, and the third axis will be the value of the loss function at the f(α, β) point, e.g. $y = CostFunction(f(\alpha, \beta))$
This method is just the multiple usages of the previous 1D interpolation if no other techniques are used. We will consider two approaches to make use of this method - **PCA**(see 4.4) and **Filter normalization**(see 4.3).

Scale invariance

As we introduced the normalization methods in section 3.2, we can say that any network's parameter can be re-scaled using the batch-normalization[15].

Even though we will get the same distribution for any network, we cannot be sure what units to choose to represent the network surface. For example - the large weights will be normalized to small numbers, and we would need a tiny step to catch the surface. In comparison, the small weights will not be changed too much, and we will compute a lot of unnecessary points' value on the surface.[27]

4.3 Filter-wise normalized directions

As stated in the previous sections, we should consider the scale-invariant loss function's properties and tune the direction length accordingly. Concerning this problem, we can use a **Filter-Wise normalization** method to catch the required curvature in the (-1, 1) range of the direction.

Let us assume that we have a network with $\vec{\theta}$ parameters vector. Afterward, we generate a direction $\vec{d}$ for the parameters vector with the same shape. To correctly choose a length, we will use the following normalization:

For each **layer i**:

- If the layer's parameters have a dimension greater than 3:

$$\vec{d}_{i,j} = \frac{\vec{d}_{i,j}}{\|\vec{d}_{i,j}\|}\|\vec{\theta}_{i,j}\| \tag{4.3}$$

  where $d_{i,j}$ is the element associated with j-th **filter**[a] of i-th layer.

- Otherwise:

$$\vec{d}_{i,j} = 0 \tag{4.4}$$

[a]For example, layer with a total shape of (16, 3, 3) have 16 matrices/**filters** of (3, 3).

That way, we will get vectors with the correct length to solve the *Scale invariance* problem[27].

4.4 Visualisation with PCA components

PCA Components

Theoretical basis

We want to project vectors from **m-dimension** to **k-dimension** so that the *variance* is maximal.
Solution:[11][28]

1. Lets denote matrix A = $\{\vec{a}_1, ..., \vec{a}_n\}^T$, where every $\vec{a}_i \in R^m$ is a given vector.

2. Because the solution can be affine, we will use their mean as origin:
$\vec{a}_{mean} = \frac{1}{n}\sum_i \vec{a}_i$,
$\vec{a}_i := \vec{a}_i - \vec{a}_{mean}$

3. We want to find a maximal variance matrix B = $\{\vec{b}_1, ..., \vec{b}_k\}^T$, so that the span of all $\vec{b}_i$ creates a k-dimension subspace with all vectors from A.

4. We will calculate eigendecomposition of $A^T A = W \Lambda W^T$, which is the Covariance matrix of a data matrix A^T. In case that eigendecomposition not exists, we can use Singular decomposition(SVD).

5. We will choose k largest eigenvalues from Λ :
$Y = (\Lambda_{n-k+1}, ..., \Lambda_n)$.
And then ortogonalize it, Y = ortog(Y).

6. Finally, we know that $B = a_{mean} * \mathbf{1^T} + AYY^T$

Application of PCA components

Lets assume that $\vec{\theta}_i$ represents the parameters at epoch i and the training ended at epoch n = $\{\vec{\theta}_0, ..., \vec{\theta}_n\}$.
Let matrix M = $\{(\vec{\theta}_0 - \vec{\theta}_n), ..., (\overrightarrow{\theta_{n-1}} - \vec{\theta_n})\}$ and we use PCA Components(with k = 2) to find two directions from this matrix.

That way we will visualise the maximum variation possible for the 2D.

4.5 Use of methods

4.5.1 PCA components

As shown in the previous chapter, the PCA components are the linear approximation of the given points into a lower dimension. There exist other non-linear methods of approximation, such as t-SNE [4], but for simplicity, we consider the PCA components only.

As papers propose, the PCA trajectories can be used to visualize either the path of the optimizer[21], [27] or the spatial representation of the output([43]). In most cases, the optimizer path's shape can be described as a curve approaching the local minimum with a bit of oscillation near the minimum. For an illustration, see 4.2 However, instead of visualizing the path, our goal is to investigate the converged point as a whole, not the individual case. Moreover, as [8] states, the data ordering in a dataset increases convergence speed. The consequences are obvious - even a reshuffling of data tangibly changes the optimizer path. Accordingly, we will make an assumption - most of the paths with identical origin and leading to the selected minimum occupy the

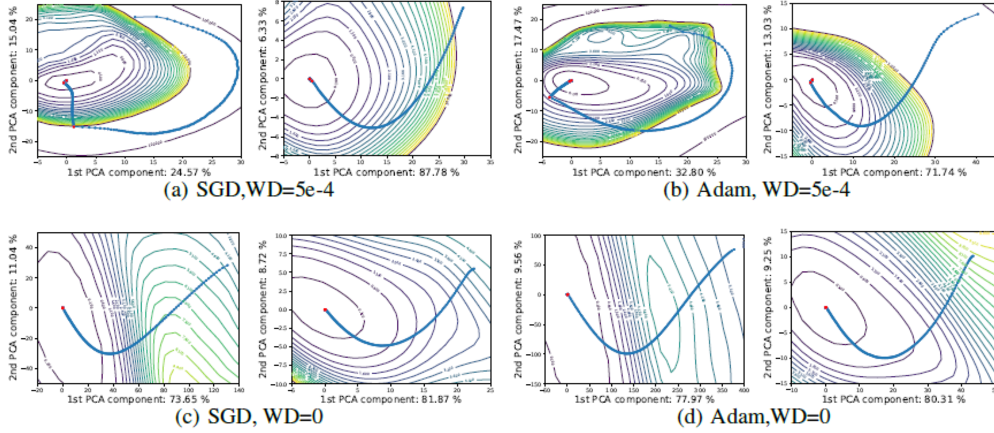


Figure 4.2: Example of PCA trajectory path
(Taken from [27])

area visualized by PCA trajectories.

Finally, we can formulate the following properties of a surface that we want to witness by PCA trajectories:

- **Sharpness** or **Flatness**(see section 4.6)
- **Convexity** or **Non-convexity** - as mentioned above, the optimizer paths are sensitive to curvature, data sampling, and other changes in the training process. So, we can state that the non-convex surroundings of the minimum act as a hazardous obstacle. In the case of a lousy learning rate, the optimizer can quickly move to the area with an opposite gradient direction and converge to an entirely different minimum. Nevertheless, as concluded in [27], the 2D approximation's convexity is not enough to prove the convexity of the original space. Positively, the non-convexity implies the total non-convexity of the original space.

We summarize them under the term **efficiency** of the minimum.

4.5.2 Random trajectories

The PCA trajectories' main goal was to visualize the approximation of the optimizer's path as a subspace. Nevertheless, the subspace of the optimizer's path is not enough to confirm the local minimum's efficiency. Even though the PCA's subspace confirms the efficiency in the selected subspace, the actual point may be located around the non-convex area and, therefore, be ineffective. Considering that, we should try visualizing the endpoint's surroundings to examine the found minimum's efficiency.

The main reason to use random trajectories instead of PCA or any other fixed direction is to observe parameter subspace that we do not consider in those methods. For example, the network can perform well for the optimized path, but the surroundings of the minimum can be entirely ineffective near the point.

Length normalization

However, the main question is how to choose the random direction and it's length if we don't have any information about it. To do so, we have such method:

- Create a $\vec{\theta}$-dimensional **Gaussian direction vector** $\vec{\mathbf{d}}$ = *Normal distribution* generalized in the high-dimension
- Apply **Filter normalization**[27] - that way we will consider only the part of the layer where we have matrices instead of scalars. In this case, we will be zeroing the parts of random trajectory in case of bias, batch normalization, and other non-matrix types of parameters. Because of this, at each point, the Batch normalization and other non-convolutional scaling are remaining the same. Thus, we removed the scale invariance problem[27].

Finally, the more detailed reasons for the filter normalization method are described in [27]. Specifically, the critical advantage of this method is the more accurate correlation of the sharpness with generalization error.

4.6 Sharp vs Flat discussion

The motivation for the subsequent loss surface comparison is the discussion of sharp versus flat curvature generalizability [24], [25], [27], [29].

The discussion is being unsolved due to the solid arguments on both sides. In comparison, the paper [24] stated that the small-batch SGD converges to the flat minima correlated with better generalizability. Afterward, the paper [25] argued that the generalizability is not directly related to the curvature, and there are techniques to solve the generalizability problem [33].
Finally, the paper [27] introduced new methods to show the correlation between flatness and generalizability, thus continuing the discussion.

In the end, we stick to the middle way - we test the introduced methods and check the correlation of flatness and generalizability(see 6.1-6.6). Afterward, we conclude the results and evaluate the method's correctness.

Theoretical summary

We have summarized the theoretical basics, the different architectures, and several advanced techniques used in modern Neural Networks. With all the theoretical background set, we can dive into the implementation of visualization in the next chapter.

We will focus on the Loss Surface visualization techniques and implement it in the Python.

Chapter 5

Implementation and methods

In this chapter, we outline the experiments and provide details about the methods used in the next chapter.

5.1 Dataset overview

- **Heterogeneity–aware dataset**[1] - Gene expression data defined as a regression problem. More precisely, we use half of the inferred genes, e.g., pre-processed version from [48].
  Size: 52,407 samples(train) + 17,469 samples(validation) of input size 942 and output size of 4759.
- **Cifar10**[2] - Classification dataset of hand-written digits, e.g., images of numbers 0-9.
  Size: 50,000(Train) + 10,000(Validation) color images of size 32x32 split into 10 labels.
- **ImageNet**(ILSVRC 2012)[3] - A complex dataset that is made of natural images. It includes a wide variety of labels such as tiger shark, digital clock, board. The total number of labels is 1000.
  Size: 1,281,167(Train) + 50,000(Validation) images of sizes between 75x56 and 4288 x 2848.

5.2 Experiments

For all experiments, we will consider validation accuracy as a generalizability representation.

- **ResNet-Cifar10**= Experiments in these sections will replicate the results from [27] and compare the loss surface of the different ResNet implementations(see 5.4) and visualize the impact of the skip connections.
  Dataset used - **Cifar10**.
- **DenseNet-Cifar10** = The experiment in this section compares the impact of *DenseNet*'s concatenating (see 3.4) with the skip connections on the same dataset.
  Dataset used - **Cifar10**.
- **DenseNet/ResNet-ImageNet** = The experiment in this section analyzes the previous *DenseNet* and *ResNet* architectures with the more complex dataset.
  Dataset used - **ImageNet**.
- **TAAF x Dense** = Experiments in these sections analyze the differences between the TAAF and static activations. We are trying to inspect the hypothesis - whether adding four parameters per neuron results in a more generalizable loss surface curvature or not.
  We will test it on two network scales:
  - 3 layers, 6000 neurons
  - 3 layers, 9000 neurons

  Dataset used - **Heterogeneity–aware dataset**.

[1] https://cbcl.ics.uci.edu/public_data/D-GEX/
[2] https://www.cs.toronto.edu/~kriz/cifar.html
[3] http://www.image-net.org/

5.3 Visualisation methods

We will focus on two loss surface visualization methods in our experiments:

- *PCA components*(**PCA**) - the primary use of this method is to show the fittest linear approximation of the surface in the optimizer space.
  The motivation is to visualize the surface around the path of the optimizer[27].
- *Random trajectories with filter normalization*(**filnorm**) - the primary use of this method is to analyze the surface around the converged local minima.
  The motivation is to assess its effectiveness and generalizability[25].

5.4 Models overview

Below is the overview of the models taken from an external repository and the custom re-implemented ones. The brief conclusion of the architectures' performance and the parameters count are summarized in section 6.11.

External models

- TAAF(TensorFlow) [48]
- ResNet50(TensorFlow)[51]
- ResNet110(PyTorch)[41]
- DenseNet cifar(TensorFlow) [36]
- DenseNet121, ResNet50(TensorFlow-in-built) [4]

Implemented models

- ResNet56(TensorFlow) - re-implementation of ResNet110 from [41]

5.5 Visualization details

The implementation of the loss surface visualization [5] contains three primary parameters:

- **Scale** = represents the area of the loss surface analyzed. It is defined as a square with a radius R, in coordinates $[-R, R] \times [-R, R]$. Throughout the experiment visualizations, we denote:
  - R=2 or $[-2, 2] \times [-2, 2]$ as **2xScaled**
  - R=1 is not denoted with scale modification.
- **Resolution** = represents the number of points used to approximate the area chosen. It can be viewed as a discretization of an area's axes. Throughout the experiment visualizations, we denote:
  - For example, 480 points per axes(x,y) as $\mathbf{480} \times \mathbf{480}$
- **XY_Scale** = represents the scaling of the XY plane, while the Z-axis scale remains the same or log-scaled[6]. The XY-Scaling can be either auto-performed by the **MatPlotLib** library or manually tuned in the **VTK visualization** libraries. Both variants are being used in most cases.

[4]`https://www.tensorflow.org/api_docs/python/tf/keras/applications`
[5]`https://gitlab.fel.cvut.cz/anuarali/loss-surface-bp`
[6]Log scale discussed later

Chapter 6

Experimental part

This chapter summarizes the experiments and their results.

6.1 ResNet110, Random trajectories with Filter normalization

Our next experiments replicate the result from the [27] with the **ResNet** Loss Surface.
More precisely, we consider multiple implementations of the ResNet architecture and compare their impact on the loss surface curvature.

ResNet110, Training part

Below is a summary of training of the ResNet110 from [41] repository.

Required hardware:

- 1 CUDA60 GPU

Time taken:

- **2 hours 01 minutes 13 seconds**.

Final results:

- Optimal epoch: **200**.
- Loss function = **MSE**
- loss: **0.0187**.
- accuracy: **87.40%**.
- validation loss: **0.02737**.
- validation accuracy: **81.49%**.

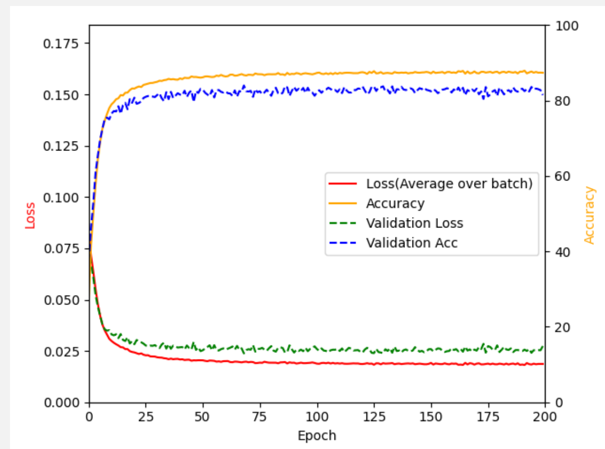


Figure 6.1: ResNet110 training

Important note

The subsequent experiments compare the surfaces of the ResNet implementations with their noskip versions using the random trajectories only. The reason for this is the computational cost of the visualization.

ResNet110(Noskip), Training part

Repeat the previous approach with skip connections removed from the model.

Required hardware:

- 1 CUDA60 GPU

Time taken:

- **2 hours 02 minutes 42 seconds**.

Final results:

- Optimal epoch: **69**.
- Loss function = **MSE**
- loss: **0.0776**.
- accuracy: **31.08%**.
- validation loss: **0.0792**.
- validation accuracy: **31.29%**.

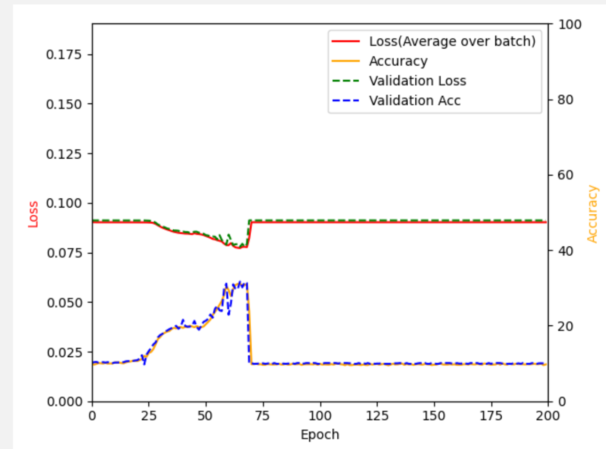


Figure 6.2: ResNet110(Noskip) training

Again, removing the skip connections shows weight explosion and model degradation over time.

Visualization

Now we will generate the filter normalized random trajectories and observer the loss surface of the models. As a relative point, we will choose epoch 69, as it is the best accuracy score in ResNet110(Noskip) model, and we can observe the surface where the models are moving at that epoch.

Visualization part(ResNet110/ResNet110(Noskip))

Required hardware:

- 20x cuda60 GPU
- 20x cuda60 GPU

Visualization area:

- Scale=1, Resolution:200x200
- Scale=4, Resolution:300x300[a]

Time taken:

- **12 hours 02 minutes 00 seconds**.
- **18 hours 12 minutes 12 seconds**.

[a]Larger size used to demonstrate more detailed visualization

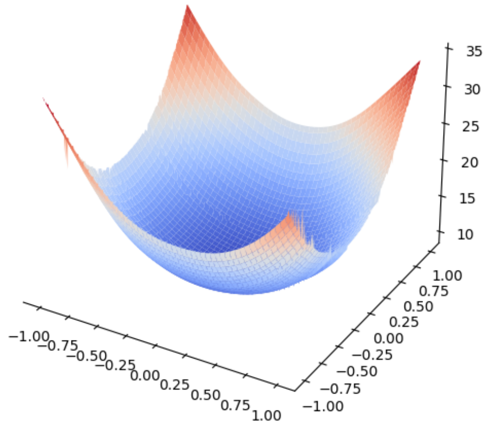


Figure 6.3: Resnet110, ,
log scale, scale=1, 200x200

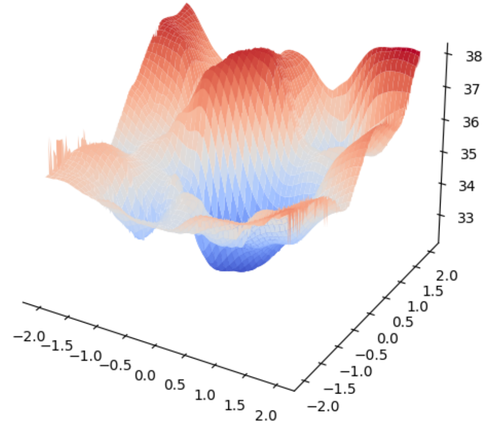


Figure 6.4: ResNet110(Noskip),
log scale, scale=1, 200x200

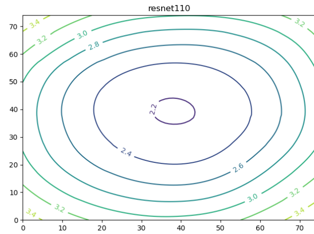


Figure 6.5: ResNet110,
Contour of 6.3

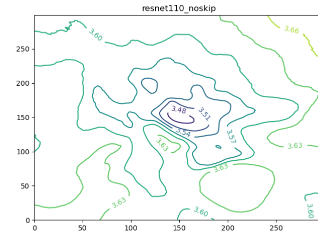


Figure 6.6: ResNet110(Noskip),
Contour of 6.4

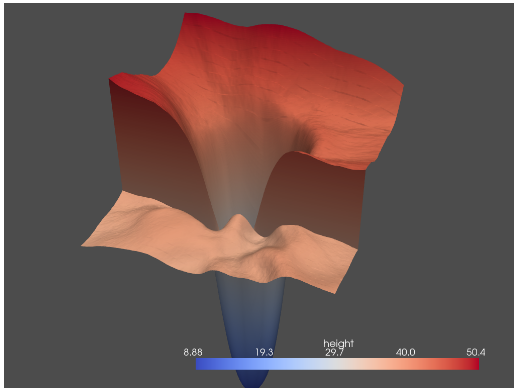


Figure 6.7: ResNet110, light + shaders,
log scale, scale=3, 300x300[a,b]

[a] https://youtu.be/r-gC66QALqs
[b] https://youtu.be/novNA_qAiBA

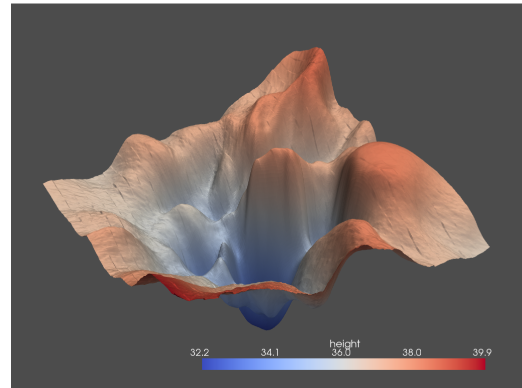


Figure 6.8: ResNet110(Noskip), light + shaders,
log scale, scale=3, 300x300[a,b]

[a] https://youtu.be/AkY8hLnc-EE
[b] https://youtu.be/dhuTU-sxZ0I

6.2 ResNet110, Validation dataset

The following plots are the same trajectories from the previous section but calculated on the validation dataset. The main goal of the following experiment is to show the similarity between the train and validation loss surface.

Visualization part(ResNet110/ResNet110(Noskip))

Required hardware:

- 20x cuda60 GPU

Visualization area:

- Scale=4, Resolution:300x300[a]

Time taken:

- **15 hours 11 minutes 18 seconds**.

[a]Larger size used to demonstrate more detailed visualization

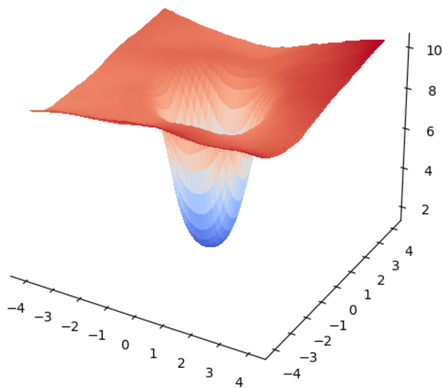


Figure 6.9: ResNet110,
loss surface of validation dataset

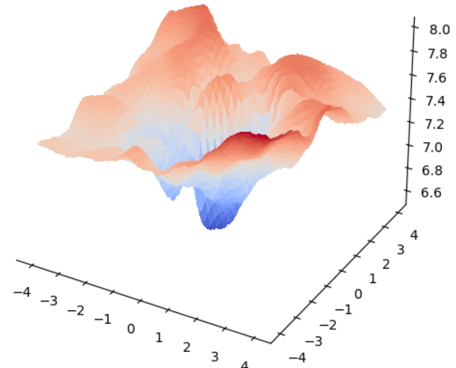


Figure 6.10: ResNet110(Noskip),
loss surface of validation dataset

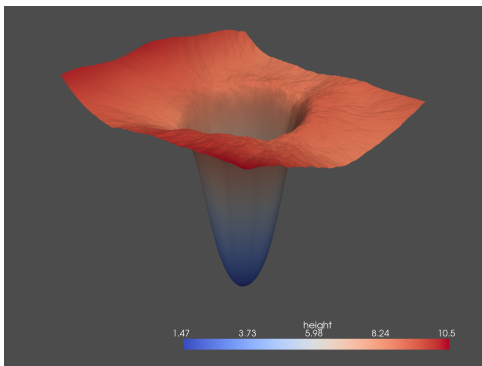


Figure 6.11: ResNet110,
same as 6.9[a]

[a]https://youtu.be/0i7ksxGJp60

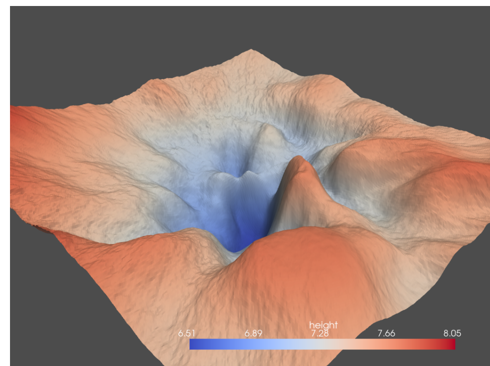


Figure 6.12: ResNet110(Noskip),
same as 6.10[a]

[a]https://youtu.be/s_0RZGU-hRA

6.3 ResNet50

The next ResNet architecture, **ResNet-50** from [51], was designed for the more complex dataset, *ImageNet*. Therefore, the architecture has more layers than needed for the image analysis and can be viewed as a visualization of a lousy architecture choice.

6.3.1 ResNet50, Training

Training part

Required hardware:

- 1 cuda60 GPU, 1 CPU, 4 GB RAM

Time taken:

- **1 hours 45 minutes 02 seconds**.

Final results after **150** epochs:

- Loss function = **Cross Entropy**
- loss: **0.0020**.
- accuracy: **98.11%**.
- validation loss: **0.0437**.
- validation accuracy: **58.25%**.

For an illustration, see graph 6.13.

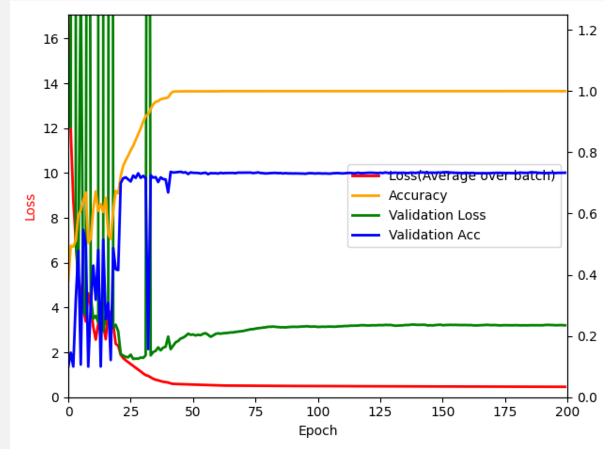


Figure 6.13: ResNet50 training

6.3.2 ResNet-50 PCA visualization

Visualization part

After obtaining the checkpoints, we will run the visualization part:

Hardware needed:

- 10 machines * [1 cuda60 GPU, 1 CPU, 10GB RAM]

Visualization area:

- Scale=1, Resolution:200x200

Calculation of an area took:

- **12 hours 53 minutes 07 seconds**.

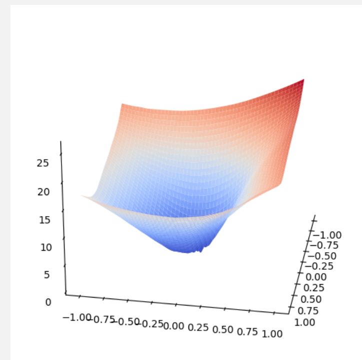


Figure 6.14: ResNet50, PCA, with log scale[a]

[a] Log scale is discussed later

6.3.3 ResNet-50(noskip) Training

Training part

Repeat the previous approach with skip connections removed from the model.

Required hardware:

- 1 cuda60 GPU, 1 CPU, 10GB RAM

Time taken:

- **1 hours 46 minutes 05 seconds**.

Final results:

- Loss function = **Cross Entropy**
- loss: **0.0052**.
- accuracy: **66.87%**.
- validation loss: **0.1011**.
- validation accuracy: **29.82%**.

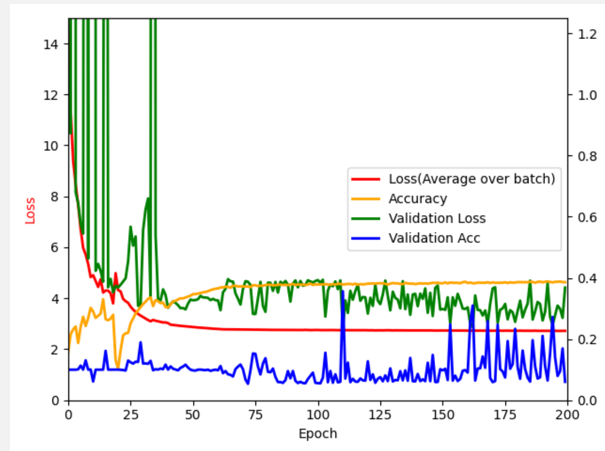


Figure 6.15: ResNet50 with no skip connection training

The result is **150** calculated checkpoints with **20,774,666** parameters value in each of them. For an illustration, see graph 6.15.

As we see from the training graph, removing the skip connections from the ResNet50 caused a massive impact on the graph. The loss value and accuracy are degrading. The hypothesis of skip connection's role in the networks with dozens of hidden layers is working in our case.

6.3.4 ResNet-50(noskip) PCA visualization

Visualization part

Required hardware:

- 10 machines * [1 cuda60 GPU, 1 CPU, 10GB RAM]

Visualization area:

- Reduced size: 200x200

Calculation of an area took:

- **13 hours 01 minutes 25 seconds**.

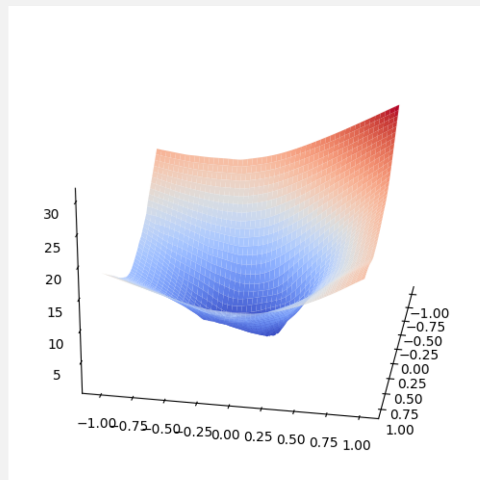


Figure 6.16: ResNet50(Noskip), PCA[a], with log scale[b]

[a]We used the scaling to the length of path from origin to the most distant checkpoint
[b]Log scale is discussed later

6.3.5 ResNet50, Random trajectories with Filter normalization

Visualization part

- 10 machines * [1 cuda60 GPU, 1 CPU, 10GM RAM]
- 20 machines * [1 cuda60 GPU, 1 CPU, 10GM RAM]

Visualization area:

- Resolution size: 200x200, Scale: 1x1
- Resolution size: 300x300, Scale: 4x4**

Calculation of an area took:

- **15 hours 11 minutes 12 seconds**.
- **16 hours 3 minutes 3 seconds**.

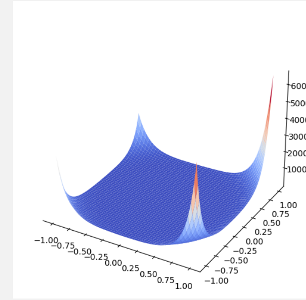


Figure 6.17: 200x200 Resnet50,[-1,1]

* We will present other visualizations after the repairing in the next subsection
** Larger size used to demonstrate more detailed visualization

Repairing of the plot

Nevertheless, the flat surface on the bottom is misleading. The values there are not constant, so the surface we see is deformed due to the other reason.
After analyzing the values in the plot, we can quickly determine that the problem here lies in the number order - the minimum is **103**, and the maximum number is **3.2e5**.

To solve these problems, we will use several techniques, such as:

- **Log scale** - to visualize the surface without the order deformations, we can use a logarithm of values. That way, we will preserve the function's surface in terms of value difference - the more significant values will have a larger value and vice versa.
- **Value threshold** - we will set a certain threshold, and we will clamp all values above it so that the function will not be a pointing arrows.
  We can see them as flat red surfaces on the sides of the plot.

As an example see 6.18.

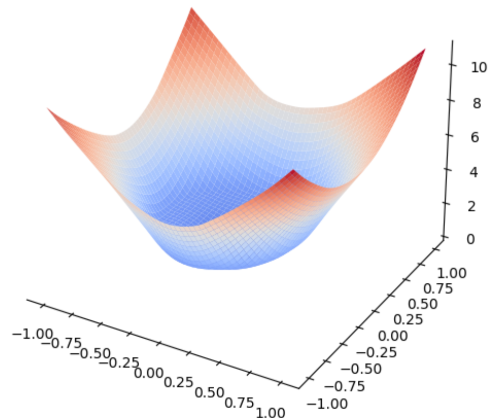


Figure 6.18: Logarithm of the 6.14

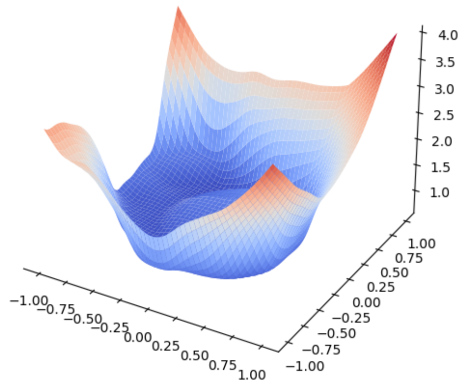


Figure 6.19: ResNet50(Noskip), Filter norm, log scale, scale=1, 200x200

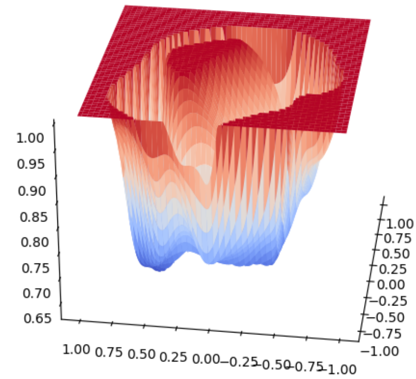


Figure 6.20: ResNet50(Noskip), Filter norm, threshold=7 + log scale, same as 6.19

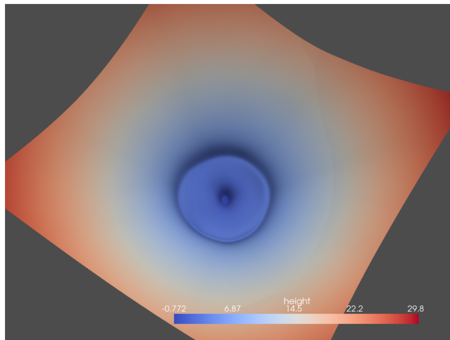


Figure 6.21: ResNet50, light + shaders, same as 6.18[a]

[a]https://youtu.be/CLhJFhy6vmM

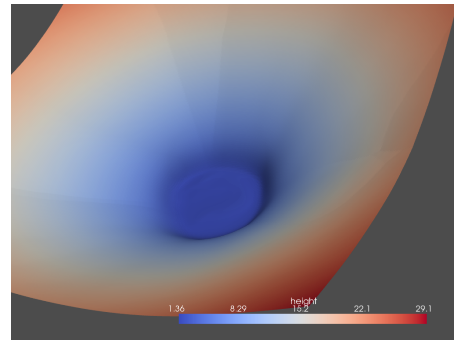


Figure 6.22: ResNet50(Noskip), light + shaders, same as 6.19[a]

[a]https://youtu.be/0pZH-153f3Q

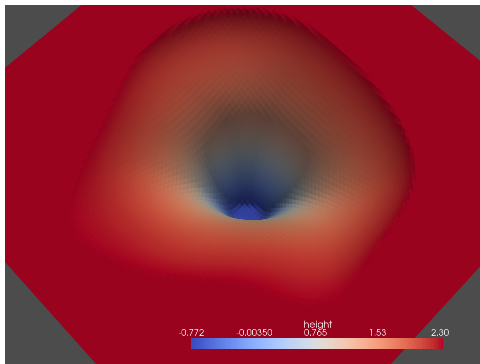


Figure 6.23: ResNet50, light + shaders, threshold=7 + log scale[a]

[a]https://youtu.be/qe009WdDjMc

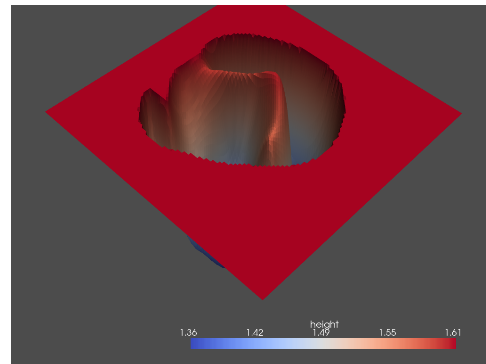


Figure 6.24: ResNet50(Noskip), light + shaders, threshold=7 + same as 6.20[a]

[a]https://youtu.be/YmfdkWz1dHU

Observations

As we see, the more complex the architecture gets, the more information we lose with the linear approximation in PCA components. Because of this, the comparison of the surfaces relies more on the random trajectories than the PCA.

Nonetheless, we perceive the results as similar to the previous experiments. With the mentioned threshold technique, we can zoom in and inspect the impact of the skip connections. The skip connections conversely remove the deformations and make the surface flatter. As mentioned beforehand, the flat surface is more likely to become more generalizable[27][25][24].

6.4 ResNet56, Random trajectories with Filter normalization

ResNet56, Training part

Below is a summary of training of the ResNet56.
Required hardware:

- 1 CUDA60 GPU

Time taken:

- **1 hours 01 minutes 25 seconds**.

Final results:

- Loss function = **Cross-Entropy**
- loss: **0.00004**.
- accuracy: **99.49%**.
- validation loss: **0.0332**.
- validation accuracy: **71.97%**.

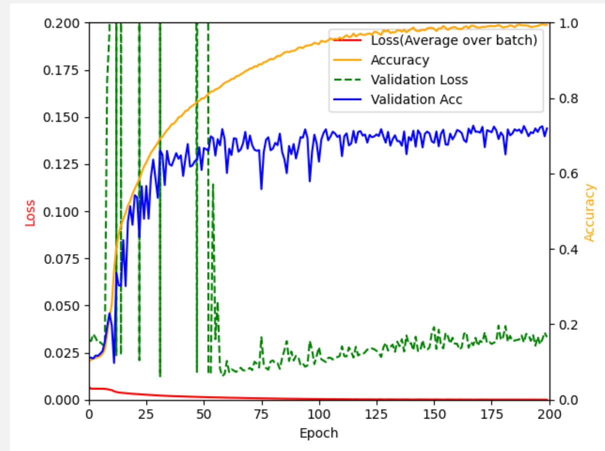


Figure 6.25: ResNet56 training

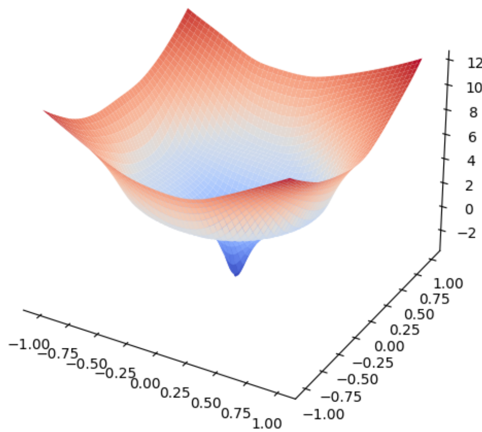


Figure 6.26: Resnet56, 200x200, log scale + threshold

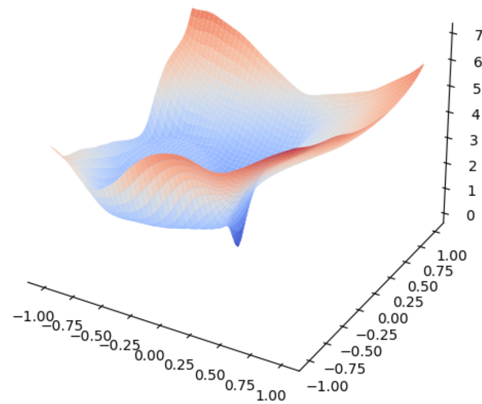


Figure 6.27: ResNet56(Noskip), 200x200, log scale + threshold

ResNet56(Noskip), Training part

Below is a summary of training of the ResNet56.

Required hardware:

- 1 CUDA60 GPU

Time taken:

- **1 hours 05 minutes 23 seconds**.

Final results:

- Loss function = **Cross-Entropy**
- loss: **0.0013**.
- accuracy: **80.98%**.
- validation loss: **0.02736**.
- validation accuracy: **52.49%**.

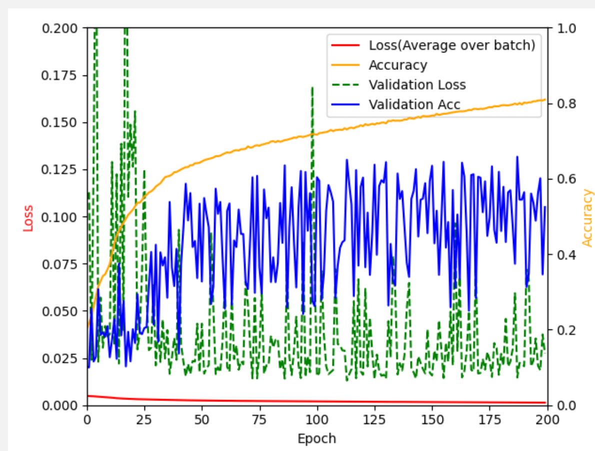


Figure 6.28: ResNet56(Noskip) training

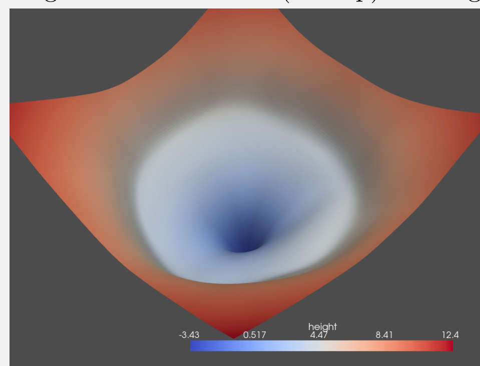


Figure 6.29: 6.26 with light + log scale[a]

Visualization part(ResNet56/ResNet56(Noskip))

Visualizations:

- ResNet56
- ResNet56(Noskip)

Required hardware:

- 10x cuda60 GPU
- 10x cuda60 GPU

Time taken:

- **11 hours 52 minutes 23 seconds**.
- **11 hours 55 minutes 42 seconds**.

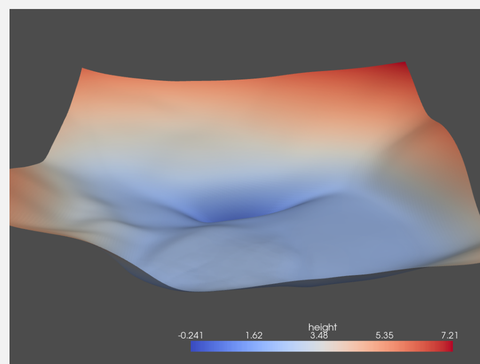


Figure 6.30: 6.27 with light + log scale[b]

[a] `https://youtu.be/tiPlHAu-2q8`
[b] `https://youtu.be/r4N37djM45w`

Summary of ResNet

After the visualization of all ResNet architectures with the Cifar10 dataset, we can summarize the following conclusions:

- The flattening effect of the ResNet's skip connections was witnessed on all scales and models.
- All of the ResNet models were prevailing over the noskip versions on both loss and accuracies.
- The MSE function gives the most deformations and is the most sensitive static activation function to the surface curvature.

Concluding note - since the MSE is the most sensitive to deformations, we may assume that it is not suitable for classification tasks. More specifically, the generalizability in a deformed surface may suffer due to the sharp minima.
Also, since we have an accuracy used as a generalizability representation, now is the time to compare the accuracy performance from 6.11 with the curvature.
The most prominent evidence for the accuracy-flatness correlation is the ResNet110 plots(81% vs 31.29%), and it holds for train and validation datasets. Next, ResNet50(58.25% vs 29.82%)and ResNet56(71.97% vs 52.49%), even though the curvature changes are visible only near the minimum, also supports the correlation, but can not fully prove the correlation.For a brief overview of the correlation, see 6.11.

6.5 DenseNet, Cifar/ImageNet

DenseNet cifar, Training part

Below is a summary of training of the DenseNet from [36] repository.
Required hardware:

- 1 CUDA60 GPU

Time taken:

- **4 hours 05 minutes 33 seconds**.

Final results:

- Optimal epoch: **150**.
- Loss function = **Cross-Entropy**
- loss: **0.0578**.
- accuracy: **99.22%**.
- validation loss: **0.9348**.
- validation accuracy: **84.60%**.

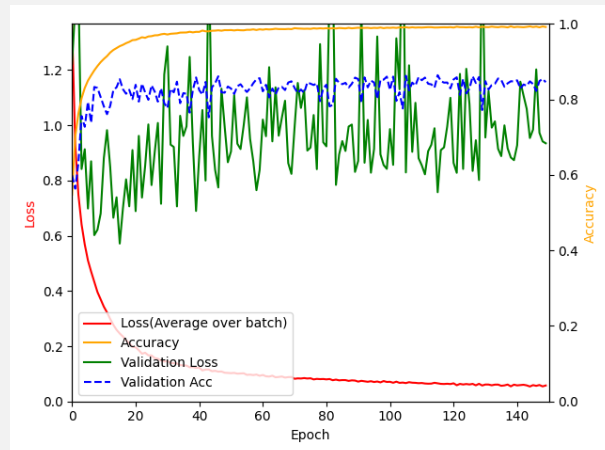


Figure 6.31: DenseNet cifar training

Visualization part(DenseNet-cifar)

Required hardware:

- 20x cuda60 GPU
- 20x cuda60 GPU

Time taken:

- **13 hours 44 minutes 26 seconds**.
- **13 hours 41 minutes 58 seconds**.

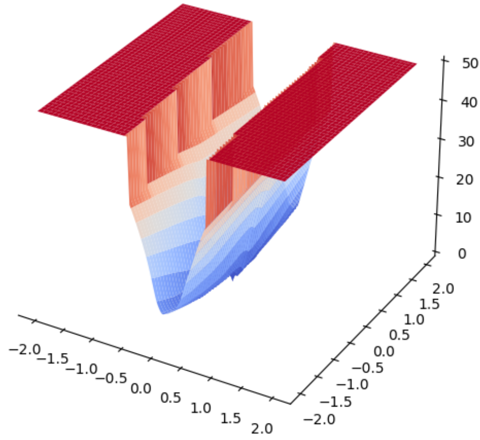


Figure 6.32: DenseNet, PCA, [200x200], 2x scaling

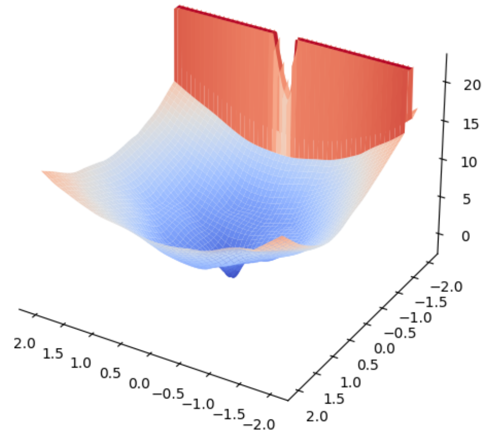


Figure 6.33: DenseNet, filnorm random, [200x200], 2x scaling

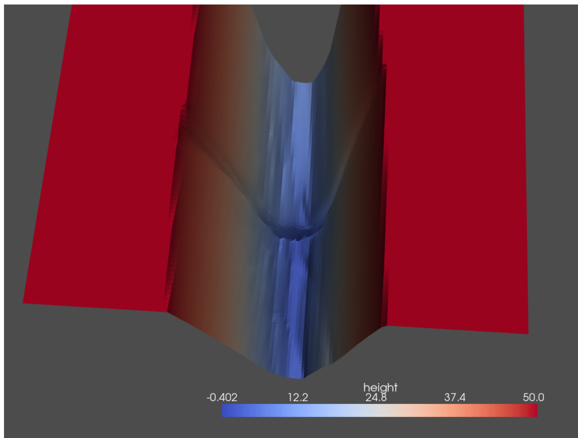


Figure 6.34: DenseNet, PCA, light + threshold=exp(50), same as 6.32[a]

[a]https://youtu.be/SJfUXEHdPl4

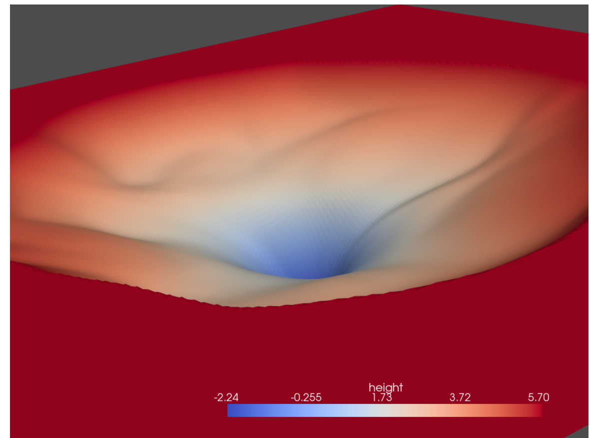


Figure 6.35: DenseNet, filnorm, light + threshold=300, same as 6.33[a]

[a]https://youtu.be/ohJocdL0xLg

Observations

From the filter-normalized visualization above, we can see that the curvature of the DenseNet architecture is slightly different from the ResNet architecture curvatures. The notable differences are small non-convexities that occur as the surface decreases near the minimum.
However, in comparison with the ResNet's noskip versions, these non-convexities are primarily negligible and do not affect the direction of the gradient as the previous deformations observed.

6.6 ResNet50/DenseNet121 with ImageNet

As we tested the network architectures on one of the commonly used datasets - Cifar10, we can visualize a more complex dataset. However, as the complexity grows, the time needed for the training grows with it. Because of this, we use pre-trained models available in the TensorFlow library.
Moreover, since the validation data are extensive, we can not visualize with a high resolution as in previous visualizations.

The primary difference compared to previous visualizations is that we cannot possibly visualize an original train loss in a reasonable time. The complexity of the ImageNet dataset is beyond our computational powers, and thus we will focus on the validation dataset only.

DenseNet121

Parameter number:

- 7,978,856

ImageNet validation accuracy(checked):

- 0.7232

Hardware needed:

- 20x cuda60 GPU

Visualized are:

- Scale = 1, Resolution = 60x60

Time taken:

- 21 hours 37 minutes

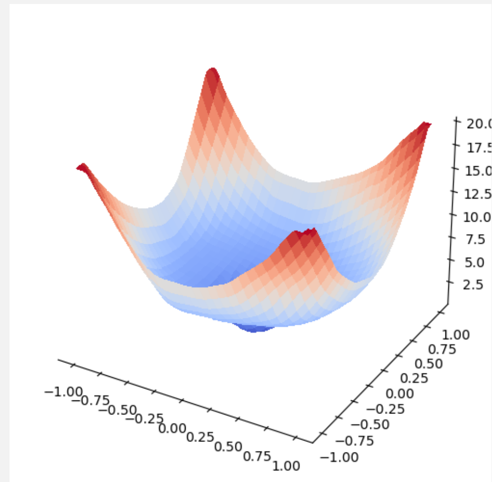


Figure 6.36: DenseNet121, ImageNet validation loss

ResNet50

Parameter number:

- 25,636,712

ImageNet validation accuracy(checked):

- 0.8011

Hardware needed:

- 20x cuda60 GPU

Visualized are:

- Scale = 1, Resolution = 60x60

Time taken:

- 23 hours 22 minutes

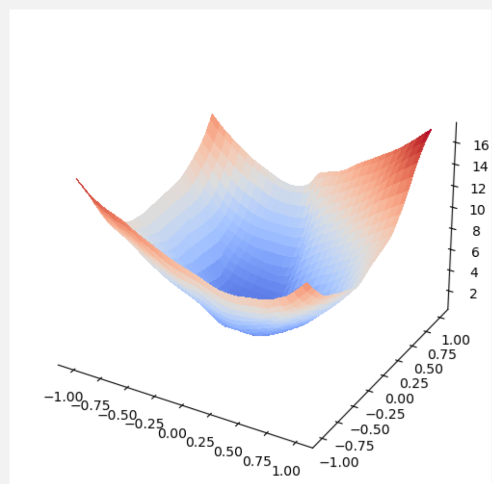


Figure 6.37: ResNet50, ImageNet validation loss

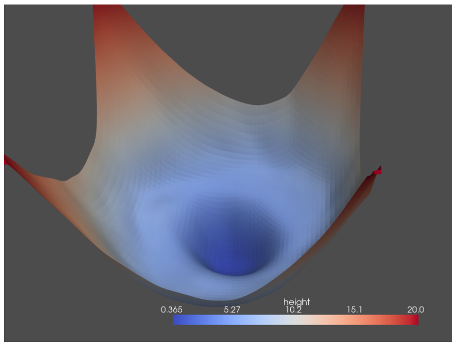


Figure 6.38: DenseNet121, Filter norm, scale=1, 60x60[a]

[a]https://youtu.be/uH8ghzdTEfU

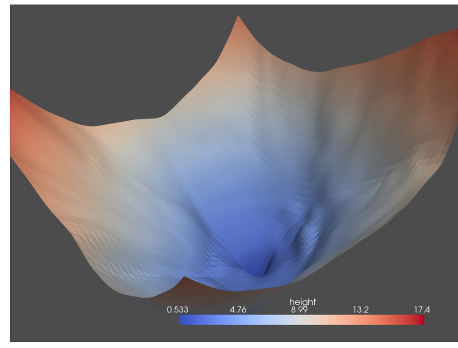


Figure 6.39: ResNet50, Filter norm, scale=1, 60x60[a]

[a]https://youtu.be/_5RX0Bv1wg8

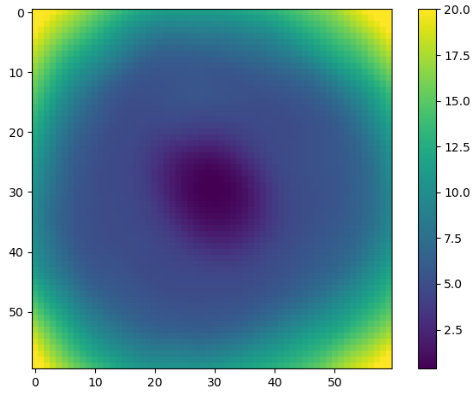


Figure 6.40: DenseNet121, Heatmap, scale=1, 60x60

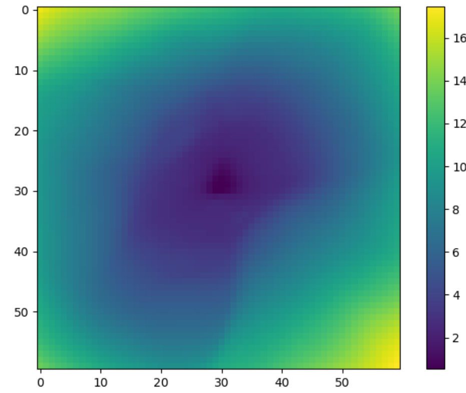


Figure 6.41: ResNet50, Heatmap, scale=1, 60x60

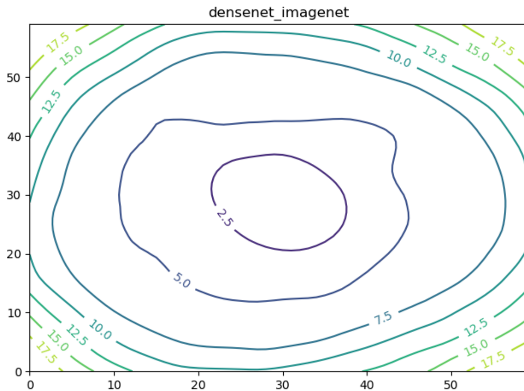


Figure 6.42: DenseNet121, ContourMap, scale=1, 60x60

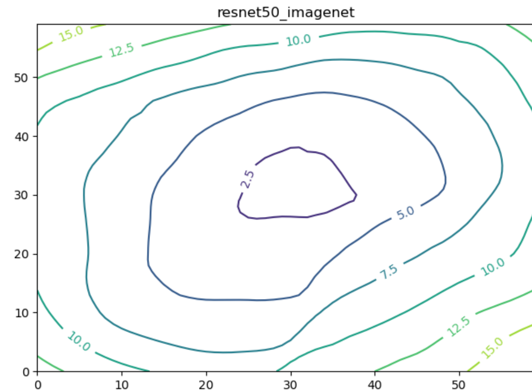


Figure 6.43: ResNet50, ContourMap, scale=1, 60x60

Observations

As we have seen previously(see 6.2), the validation dataset surface reflects the loss surface of the training loss. On top of that, all the assumptions of the optimal minimum hold for the following pre-trained networks.
We can observe that the method proposed in [27] correlates with the generalization error and works even with the validation loss surface.

6.7 TAAF, Preparations

Important Note

For all the following TAAF experiments, we use the sigmoid-based TAAF activation function

Experiment details

Our next experiment is going to show the basic functionality of the loss surface visualization.

Network: *TAAF*, two layers with 300 neurons each[48].
Dataset: a subset of the *D-GEX(GEO)*[1]dataset, taken from [48].
Visualization technique: *PCA*(151 checkpoints with 664408 parameters each)

The main goal is to visualize the surface of the small network and show the main techniques used in the upcoming experiments.

Required hardware from the MetaCentrum:

1. 4 machines with 8 avx512dq CPU each
2. 7 machines with 8 avx512dq CPU each

Visualization details:

- First approach = **271x271**, scale = **1**
- Second approach = **811x811**, scale = **2**

Calculation of an area took:

1. **22 hours 22 minutes 38 seconds**.
2. **26 hours 38 minutes 05 seconds**.

The results are figures 6.44 and 6.45

As we see, with a relatively small number of parameters in comparison with the next experiments, we get a smooth surface.

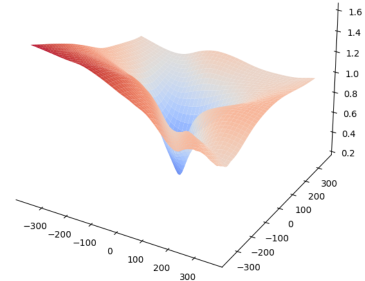


Figure 6.44: 271x271 loss surface of [48] [calculated on MetaCentrum platform]

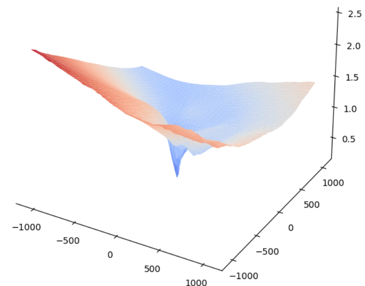


Figure 6.45: 811x811 loss surface of [48] [calculated on MetaCentrum platform]

[1] https://www.ncbi.nlm.nih.gov/gds

6.8 TAAF, Preparations 2

The following plots show us the importance of scaling and resolution increase.
Hardware needed:

- 10 machines * [1 cuda60 GPU, 1 CPU, 8GB RAM]
- 20 machines * [1 cuda60 GPU, 1 CPU, 8GB RAM]

Visualization area:

- Scale=1, Resolution:210x210 (plots 6.46, 6.48)
- Scale=2, Resolution:480x480 (plots 6.47, 6.49)

Calculation of an area took:

- **12 hours 53 minutes 07 seconds**.
- **13 hours 45 minutes 12 seconds**.

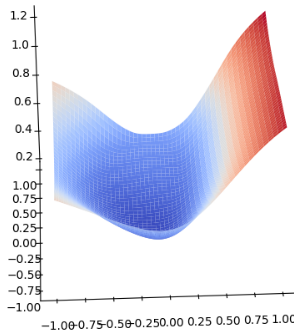


Figure 6.46: TAAF[2x300], PCA

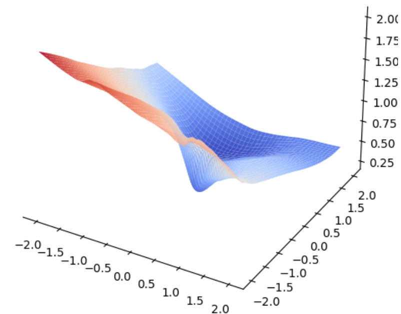


Figure 6.47: TAAF[2x300], PCA, 2x scaling

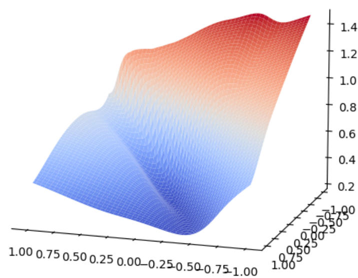


Figure 6.48: TAAF[2x300], filnorm

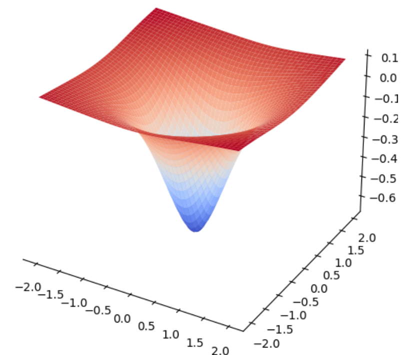


Figure 6.49: TAAF[2x300], filnorm, 2x scaling

However, as we see from the plots, it is hard to analyze the surface in 3D. Therefore, we convert the same surface to the heatmap + contour maps in the coming TAAFxDense sections. Also, the log scale is applied to all of the plots.

6.9 TAAF vs Plain, 3 layers, 6000 neurons

Now, as we have seen the visualization of a small network, we can visualize a more practical network.
The two following experiments compare the loss surfaces of the networks with and without the *TAAF* activation functions.
We will call the non-TAAF activations network as **Plain**, e.g., fully-connected hidden layers with a sigmoid activation function.

Training part

Parameter count:

- **106,297,036** (*TAAF*)
- **106,228,759** (*Plain*)

Required hardware:

- 1 cuda60 GPU, 1 CPU, 24 GB RAM

Time taken:

- **3 hours 45 minutes 02 seconds**.

Final results after **500** epochs:

- Loss function = **MAE**
- loss: **0.1324**(TAAF), **0.1293**(Plain).
- validation loss: **0.1607**(TAAF), **0.1813**(Plain).

For an illustration, see graph 6.50.

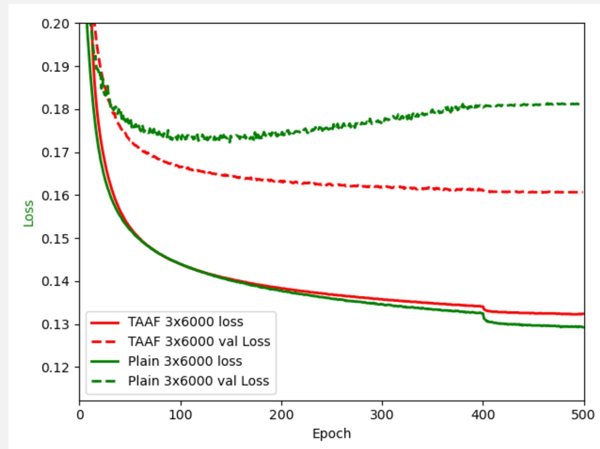


Figure 6.50: TAAF vs Plain, 3x6000

As we see, with the relatively low number of parameters added, the network's performance increases significantly. The dominance of the parametrized *sigmoid TAAF* over the default *sigmoid* is apparent.

Visualization part

Hardware needed:

- 10 machines * [1 cuda60 GPU, 2 CPU, 28GB RAM]

Visualization area(same for all):

- Scale=2, Resolution:200x200

Calculation of an area took:

- **17 hours 12 minutes 05 seconds**(PCA)
- **18 hours 01 minutes 11 seconds**(filnorm)

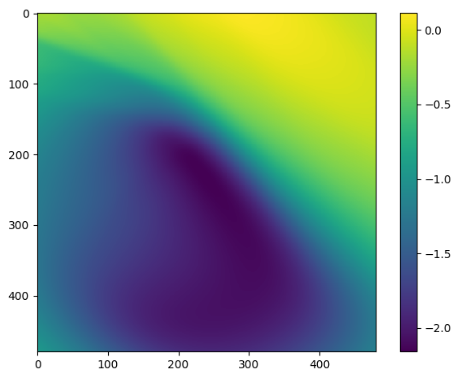


Figure 6.51: TAAF[3x6000] PCA, 2x scaled

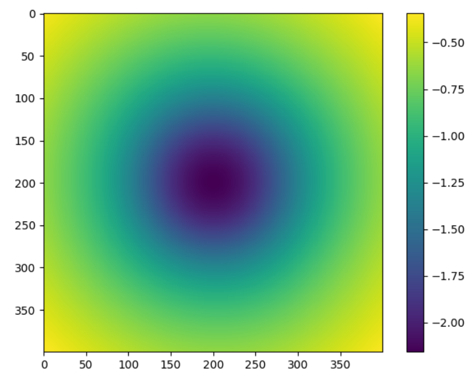


Figure 6.52: TAAF[3x6000] filnorm, 2x scaled

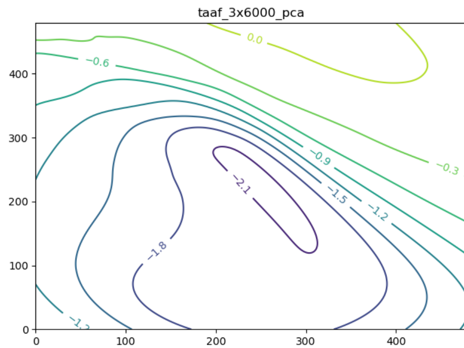


Figure 6.53: TAAF[3x6000] PCA Contour

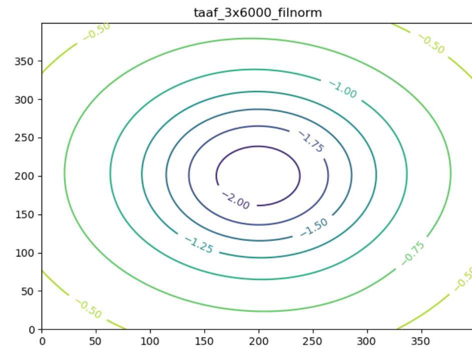


Figure 6.54: TAAF[3x6000] filnorm Contour

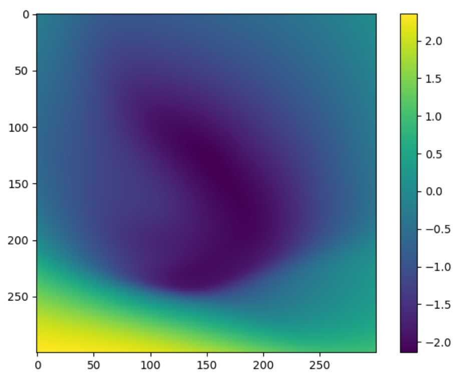


Figure 6.55: Plain[3x6000] PCA, 2x scaled

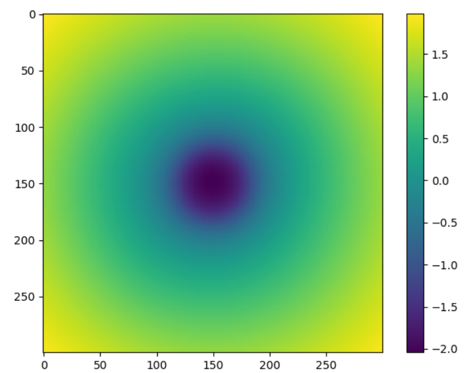


Figure 6.56: Plain[3x6000] filnorm, 2x scaled

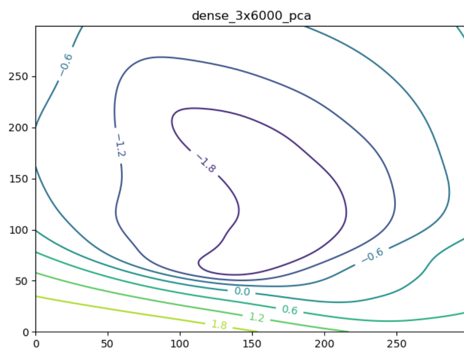


Figure 6.57: Plain[3x6000] PCA Contour

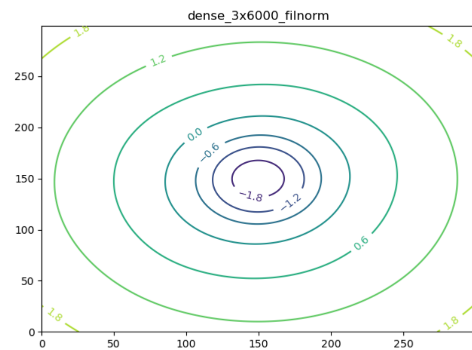


Figure 6.58: Plain[3x6000] filnorm Contour

6.10 TAAF vs Plain, 3 layers, 9000 neurons

Training part

Parameters count:

- **213,436,036** (*TAAF*)
- **213,340,759** (*Plain*)

Required hardware:

- 1 cuda60 GPU, 1 CPU, 24 GB RAM

Time taken:

- **3 hours 45 minutes 02 seconds**.

Final results after **500** epochs:

- Loss function = **MAE**
- loss: **0.1251**(TAAF), **0.1206**(Plain).
- validation loss: **0.1625**(TAAF), **0.1819**(Plain).

For an illustration, see graph 6.59.

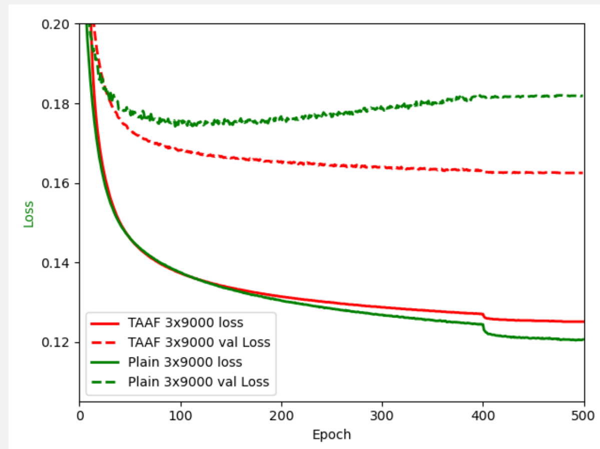


Figure 6.59: TAAF vs Plain, 3x9000

Again, the dominance of the TAAF activations over the default sigmoid is seen. However, we can observe that the performance has not improved and partially degraded compared to the 6000 neurons. The possible reasons for this are the overfitting issue and the poor starting point issues.
Unfortunately, due to the resource contraints, we cannot reproduce the same experiment multiple times and analyze the reasons for this behavior more profoundly.

Visualization part

Hardware needed:

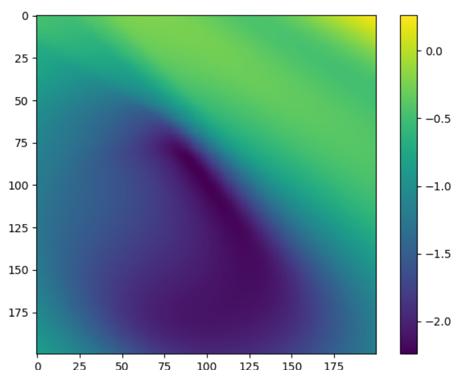
- 10 machines * [1 cuda60 GPU, 2 CPU, 28GB RAM]

Visualization area(same for all):

- Scale=2, Resolution:200x200

Calculation of an area took(Average):

- **19 hours 33 minutes 21 seconds**(PCA)
- **19 hours 37 minutes 14 seconds**(filnorm)



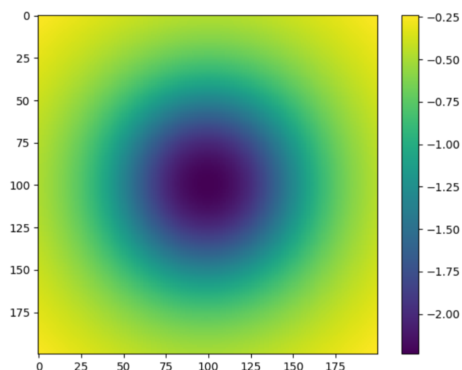
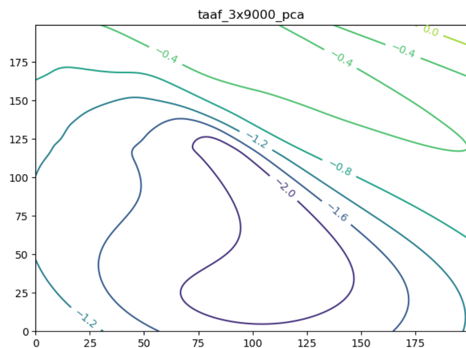


Figure 6.60: TAAF[3x9000] PCA, 2x scaled

Figure 6.61: TAAF[3x9000] filnorm, 2x scaled



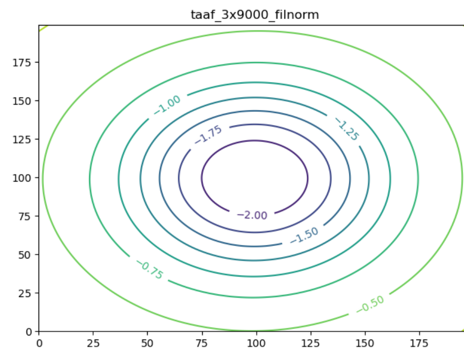
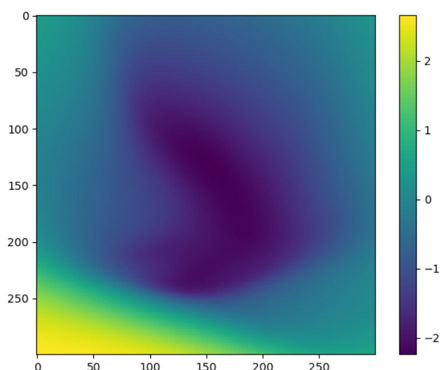


Figure 6.62: TAAF[3x9000] PCA Contour

Figure 6.63: TAAF[3x9000] filnorm Contour



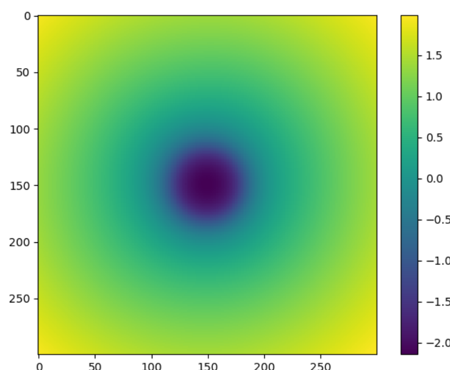
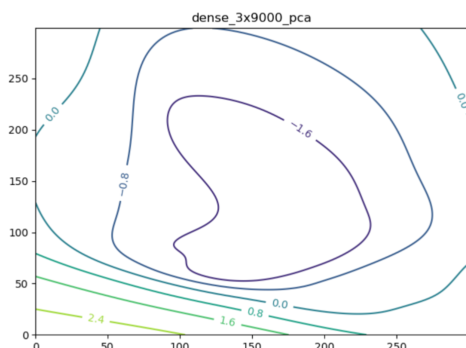


Figure 6.64: Plain[3x9000] PCA, 2x scaled

Figure 6.65: Plain[3x9000] filnorm, 2x scaled



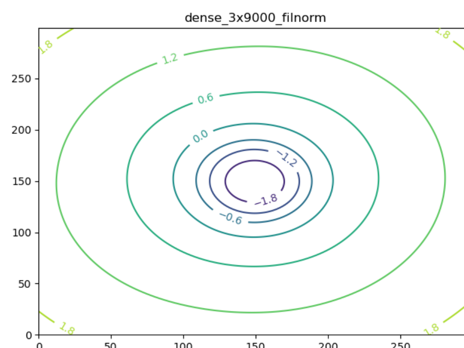


Figure 6.66: Plain[3x9000] PCA Contour

Figure 6.67: Plain[3x9000] filnorm Contour

TAAF summary

As we see, the network scaling to the 9000 neurons has not changed the curvature of a surface compared to the 6000-neuron version.
Unfortunately, no other information can be gathered from the plots. The filter-normalized random trajectory plots show us the same curvature between TAAF and FC networks.
Moreover, the PCA plots show us nearly the same plot when reduced to the unit scale, e.g., cut 1/4 of each axis. From this observation follows that the paths of the optimizers follow the same curvature. The only exciting observation encountered is the degrading phenomenon after the scaling. Unluckily, it had no effects on the plot.

So, if we were to conclude the TAAF regression section, we see that the loss surface method is not suitable for this type of regression task with the networks chosen. The experiments show us the necessity of further analysis with more sophisticated methods to measure the impact of the TAAF.

6.11 Brief overview

** All CPUs are avx512dq(TensorFlow requirement)*
*** [—] is 1 machine resources*

Computation time comparison				
Model name **Experiment details**	**Training time**	**Training resources**	**Visualisation time**	**Visualisation resources**
TAAF(Prep), PCA, 271x271, 811x811	0h 32m	1 CPU* (MetaCentrum)	22h 22m 26h 38m	7x [8 CPU*]** (MetaCentrum)
TAAF(3x6000) **PCA(200x200)**	3h 45m	1x CUDA60 GPU (MetaCentrum)	17h 12m	10x [CUDA60 GPU]** (MetaCentrum)
TAAF(3x6000) **filnorm(200x200)**	3h 45m	1x CUDA60 GPU (MetaCentrum)	18h 01m	10x [CUDA60 GPU]** (MetaCentrum)
TAAF(3x9000) **PCA(200x200)**	3h 45m	1x CUDA60 GPU (MetaCentrum)	19h 33m	10x [CUDA60 GPU]** (MetaCentrum)
TAAF(3x9000) **filnorm(200x200)**	3h 45m	1x CUDA60 GPU (MetaCentrum)	19h 37m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet50 **PCA, 200x200**	3h 01m	1x CUDA60 GPU (MetaCentrum)	12h 53m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet50(Noskip) **PCA, 200x200**	3h 36m	1x CUDA60 GPU (MetaCentrum)	13h 01m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet50 **FiltNorm, 200x200**	3h 01m	1x CUDA60 GPU (MetaCentrum)	15h 11m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet50 **FiltNorm, 300x300**	3h 01m	1x CUDA60 GPU (MetaCentrum)	16h 03m	20x [CUDA60 GPU]** (MetaCentrum)
ResNet50(Noskip) **FiltNorm, 200x200**	3h 36m	1x CUDA60 GPU (MetaCentrum)	15h 11m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet50(Noskip) **FiltNorm, 300x300**	3h 36m	1x CUDA60 GPU (MetaCentrum)	16h 03m	20x [CUDA60 GPU]** (MetaCentrum)
ResNet56 **FiltNorm, 200x200**	1h 01m	Google Colab GPU	11h 52m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet56(Noskip) **FiltNorm, 200x200**	1h 05m	Google Colab GPU	11h 55m	10x [CUDA60 GPU]** (MetaCentrum)
ResNet110 **FiltNorm, 200x200**	2h 01m	Google Colab GPU	12h 02m	20x [CUDA60 GPU]** (MetaCentrum)
ResNet110 **FiltNorm, 300x300**	2h 01m	Google Colab GPU	18h 02m	20x [CUDA60 GPU]** (MetaCentrum)
ResNet110(Noskip) **FiltNorm, 200x200**	2h 02m	Google Colab GPU	12h 03m	20x [CUDA60 GPU]** (MetaCentrum)
ResNet110(Noskip) **FiltNorm, 300x300**	2h 02m	Google Colab GPU	18h 03m	20x [CUDA60 GPU]** (MetaCentrum)
DenseNet(Cifar) **PCA, 200x200**	4h 05m	1x CUDA60 GPU (MetaCentrum)	13h 44m	20x [CUDA60 GPU]** (MetaCentrum)

Model accuracies comparison			
Model name	**Epochs**	**Cifar10 Train Accuracy**	**Cifar10 validation Accuracy**
ResNet50	150	98.11%	58.25%
ResNet50(Noskip)	150	66.87%	29.82%
ResNet56	200	99.49%	71.97%
ResNet56(Noskip)	200	80.98%	52.49%
ResNet110	200	87.40%	81.49%
ResNet110(Noskip)	200	31.08%	31.29%
DenseNet(Cifar)	150	99.22%	84.60%

TAAF vs Plain regression comparison			
Model name	**Epochs**	**GEO Train Loss**	**GEO validation Loss**
TAAF[**3x6000**]	500	0.1324	0.1607
Plain[**3x6000**]	500	0.1293	0.1813
TAAF[**3x9000**]	500	0.1251	0.1625
Plain[**3x9000**]	500	0.1206	0.1869

ImageNet model comparison		
Model name	**Parameter count**	**validation accuracy**
ResNet50	25.64 mil	80.11%
DenseNet121	7.98 mil	72.32%

Chapter 7

Conclusion

Through the work, we have successfully analyzed the theoretical and practical parts of Neural Networks.
We have started with the Machine learning theory and introduced some basic methods for optimizing the learning algorithm like *Stochastic Gradient Descent*.

Next, we narrowed it to the Neural network field, starting from the **Perceptron**, one of the first Neural network architecture. With the step-by-step progressing, we have shown the reasons behind specific changes in the architecture. On top of that, we introduced several techniques to solve a few critical problems during Neural network training.

After that, we have reviewed the ways to visualize the loss surface of the neural network and identified their advantages and disadvantages. On top of that, we introduced the fundamental properties that we want to observe on our visualized plots.

Finally, we dived into the work's practical part, where we successfully implemented the **PCA** and **filter-normalized random** trajectory visualizations. We have tested it on **TAAF**, **ResNet**, and **DenseNet** networks and came to the following conclusion:

1. Loss surface visualization technique can be used for specific tasks only. More precisely - it visualizes the curvature of the surface and hence, is needless in the case of methods that have little effect on the curvature. The examples of these disadvantages were the experiments with TAAF that showed us tiny deformations only(viz. fig 7.1, 7.2).

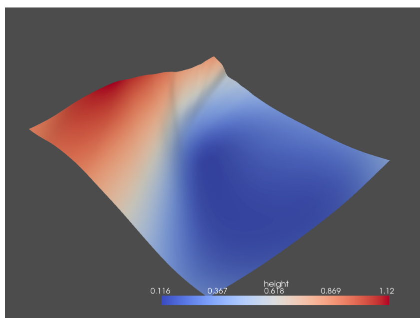


Figure 7.1: TAAF, 3x6000, light[a]

[a]https://youtu.be/c2qQ04Egqso

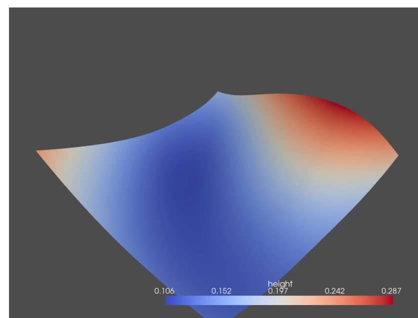
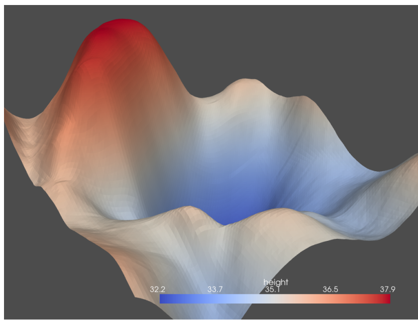


Figure 7.2: Plain, 3x9000, light[a]

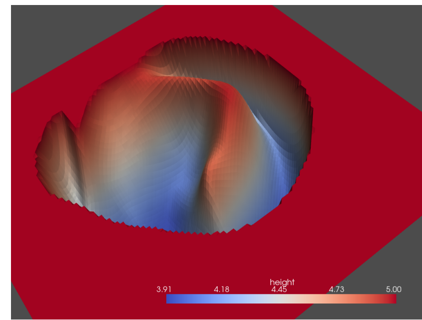
[a]https://youtu.be/OjAVh5SwHAI

2. Curvature alone can not say anything about the generalizability of the local minimum. However, it can serve as one of the criteriums to evaluate the total generalizability, combined with the more precise methods.
   Primarily, the local minimum can be evaluated as non-optimal if a highly non-convex area surrounds it. The most notable example is the ResNet110 (Noskip) surface and, less representative sample, ResNet50.

The non-optimality of these surfaces lies in deformations - As we see, these deformations are large enough to change the optimizer's direction completely. For an illustration see figure 7.3.



ResNet110(Noskip)



ResNet50(Noskip)

Figure 7.3: Most notable surfaces

Furthermore, the MSE loss function used in ResNet110 shows us why it is wrong for the classification tasks. The visualizations of the larger areas show us that even the skip connections are not capable of repairing the non-convex deformations of the surface. For an illustration see figure 7.4, 7.5

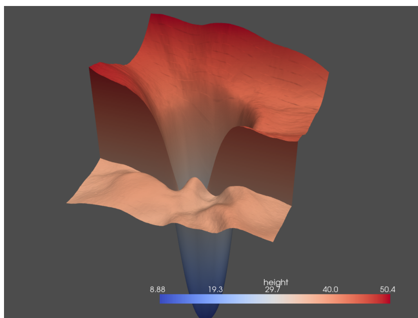


Figure 7.4: ResNet110

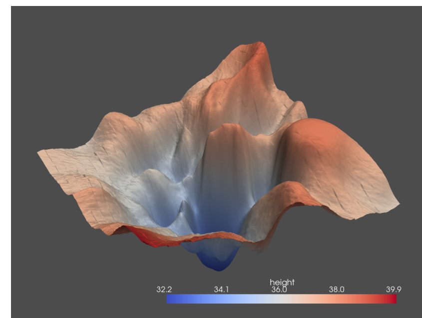


Figure 7.5: ResNet110(Noskip)

3. *PCA trajectories* are suitable for small architectures only. After the specific limit, the PCA's ability to approximate the high-dimensional surface becomes less and less effective. The proof for this is the attempt to visualize the PCA trajectory plot for the ResNet50, a highly parametrized network(viz fig 7.6, 7.7).

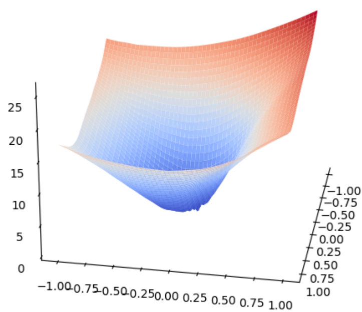


Figure 7.6: ResNet50, PCA

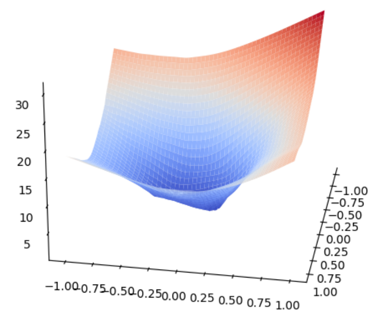


Figure 7.7: ResNet50(Noskip), PCA

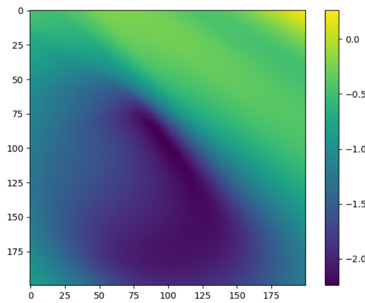


Figure 7.8: TAAF, 3x9000

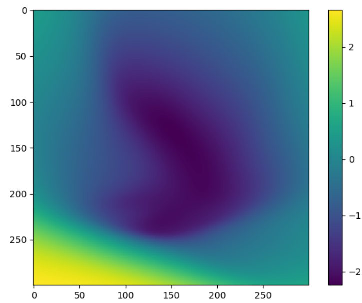


Figure 7.9: Plain, 3x9000

Alternatively, over-parametrized TAAF and Plain networks with 3x9000 sizes each also prove the point(viz. fig 7.8, 7.9).

As we see, the PCA shows us the same picture for entirely different networks in terms of performance. Therefore, the linear approximation gives zero information in this case.

4. Finally, the evaluation of the method from [27], e.g., *filter-normalized random trajectories*.
   As we progress through the architectures, we observe that the more complex architecture and more over-parametrized ones are less sensitive to the curvature change, while lousy choices such as the MSE loss function give us the most deformations visible.

   For example, the DenseNet performance is hard to determine using this method. It has a non-convex deformation, but the total shape remains flat. For an illustration see figure 7.10, 7.11

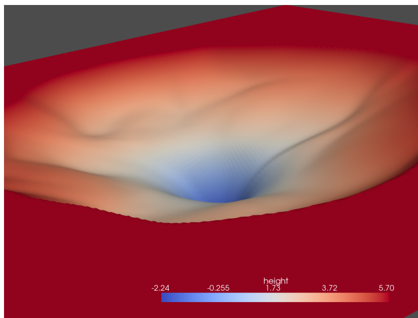


Figure 7.10: DenseNet, cifar

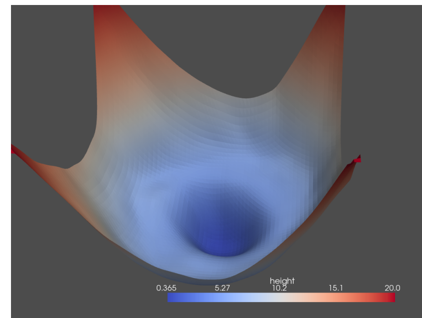


Figure 7.11: DenseNet, ImageNet

Even though the research here is not enough to fully prove the statement, the observations clearly show the following correlation - the method's effectiveness decreases with the more advanced method such as DenseNet concatenation, task-specific architecture(ResNet50 with ImageNet), and TAAF activation functions.

To conclude - the visualization of the loss surface of the neural network is one of the possible visualization ways to evaluate the generalizability. However, the more advanced the architectures become, the less noticeable the problems are. Because of this behavior, new methods of evaluating neural network performance are needed.

Bibliography

[1] J. W. Sammon, “A nonlinear mapping for data structure analysis”, *IEEE Transactions on computers*, vol. 18, no. 5, pp. 401–409, 1969. [Online]. Available: http://syllabus.cs.manchester.ac.uk/pgt/2017/COMP61021/reference/Sammon.pdf, (visited 30.03.2021).

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Representations by Back-propagating Errors”, *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0, (visited 06.04.2021).

[3] T. Yamada and T. Yabuta, “Neural network controller using autotuning method for nonlinear functions”, *IEEE Transactions on Neural Networks*, vol. 3, no. 4, pp. 595–601, 1992. DOI: 10.1109/72.143373.

[4] S. Hochreiter and J. Schmidhuber, “Long short-term memory”, *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, (visited 30.03.2021).

[5] L. Yang, “3D Grand Tour for Multidimensional Data and Clusters”, in *Advances in Intelligent Data Analysis*, D. J. Hand, J. N. Kok, and M. R. Berthold, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 173–184, ISBN: 978-3-540-48412-7. [Online]. Available: https://cs.wmich.edu/~yang/explorer/16420173.pdf, (visited 31.03.2021).

[6] A. P. Saygin, I. Cicekli, and V. Akman, “Turing test: 50 years later”, *Minds and Machines*, vol. 10, Oct. 2000. DOI: 10.1023/A:1011288000451. [Online]. Available: https://www.researchgate.net/publication/2435828_Turing_Test_50_Years_Later, (visited 30.03.2021).

[7] M. Huh and K. Kim, “Visualization of multidimensional data using modifications of the grand tour”, *Journal of Applied Statistics*, vol. 29, pp. 721–728, Jul. 2002. DOI: 10.1080/02664760120098784.

[8] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning.”, in *ICML*, A. P. Danyluk, L. Bottou, and M. L. Littman, Eds., ser. ACM International Conference Proceeding Series, vol. 382, ACM, 2009, pp. 41–48, ISBN: 978-1-60558-516-1. [Online]. Available: http://dblp.uni-trier.de/db/conf/icml/icml2009.html#BengioLCW09, (visited 28.04.2021).

[9] P. Golik, P. Doetsch, and H. Ney, “Cross-entropy vs. squared error training: A theoretical and experimental comparison”, Aug. 2013, pp. 1756–1760. [Online]. Available: https://www.researchgate.net/publication/266030536_Cross-Entropy_vs_Squared_Error_Training_a_Theoretical_and_Experimental_Comparison, (visited 30.03.2021).

[10] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models”, in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. [Online]. Available: https://ai.stanford.edu/~amaas/papers/relu%5C_hybrid%5C_icml2013%5C_final.pdf, (visited 30.03.2021).

[11] J. Shlens, “A tutorial on principal component analysis”, *CoRR*, vol. abs/1404.1100, 2014. arXiv: 1404.1100. [Online]. Available: http://arxiv.org/abs/1404.1100, (visited 30.03.2021).

[12] I. Goodfellow, O. Vinyals, and A. Saxe, “Qualitatively characterizing neural network optimization problems”, in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6544, (visited 30.03.2021).

[13] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”, in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition”, *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385, (visited 30.03.2021).

[15] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167, (visited 30.03.2021).

[16] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization”, in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980, (visited 30.03.2021).

[17] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning”, *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539. [Online]. Available: https://www.researchgate.net/publication/277411157_Deep_Learning, (visited 20.05.2021).

[18] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/, (visited 30.03.2021).

[19] H. Roth, L. Lu, A. Farag, H.-C. Shin, J. Liu, E. Turkbey, and R. Summers, “Deeporgan: Multi-level deep convolutional networks for automated pancreas segmentation”, vol. 9349, Jun. 2015, ISBN: 978-3-319-24552-2. DOI: 10.1007/978-3-319-24553-9_68, (visited 30.03.2021).

[20] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus)”, Jan. 2016, (visited 30.03.2021).

[21] Eliana Lorch, “Visualizing Deep Network Training Trajectories with PCA”, *In ICML Workshop on Visualization for Deep Learning*, 2016. [Online]. Available: https://icmlviz.github.io/icmlviz2016/assets/papers/24.pdf, (visited 17.04.2021).

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org (visited 30.03.2021).

[23] G. Huang, Z. Liu, and K. Q. Weinberger, “Densely connected convolutional networks”, *CoRR*, vol. abs/1608.06993, 2016. arXiv: 1608.06993. [Online]. Available: http://arxiv.org/abs/1608.06993, (visited 30.03.2021).

[24] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On large-batch training for deep learning: Generalization gap and sharp minima”, *CoRR*, vol. abs/1609.04836, 2016. arXiv: 1609.04836. [Online]. Available: http://arxiv.org/abs/1609.04836, (visited 22.04.2021).

[25] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, “Sharp minima can generalize for deep nets”, *CoRR*, vol. abs/1703.04933, 2017. arXiv: 1703.04933. [Online]. Available: http://arxiv.org/abs/1703.04933, (visited 17.04.2021).

[26] S. He. (2017). The 1986 Backpropagation Paper, [Online]. Available: https://sijunhe.github.io/blog/2017/03/27/reading-notes-the-1986-backpropagation-paper/. (visited 06.04.2021).

[27] H. Li, Z. Xu, G. Taylor, and T. Goldstein, “Visualizing the loss landscape of neural nets”, *CoRR*, vol. abs/1712.09913, 2017. arXiv: 1712.09913. [Online]. Available: http://arxiv.org/abs/1712.09913, (visited 30.03.2021).

[28] S. Mishra, U. Sarkar, S. Taraphder, S. Datta, D. Swain, R. Saikhom, S. Panda, and M. Laishram, “Principal component analysis”, *International Journal of Livestock Research*, p. 1, Jan. 2017. DOI: 10.5455/ijlr.20170415115235, (visited 30.03.2021).

[29] B. Neyshabur, S. Bhojanapalli, D. Mcallester, and N. Srebro, “Exploring generalization in deep learning”, in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/10ce03a1ed01077e3e289f3e53c72813-Paper.pdf, (visited 28.04.2021).

[30] I. Poola, “How artificial intelligence in impacting real life every day”, *International Journal for Advance Research and Development*, vol. 2, pp. 96–100, Oct. 2017. [Online]. Available: https://www.researchgate.net/publication/321348028_How_Artificial_Intelligence_in_Impacting_Real_Life_Every_day, (visited 31.03.2021).

[31] J. Bjorck, C. P. Gomes, and B. Selman, “Understanding batch normalization”, *CoRR*, vol. abs/1806.02375, 2018. arXiv: 1806.02375. [Online]. Available: https://arxiv.org/abs/1806.02375, (visited 20.05.2021).

[32] B. Hanin, "Which neural net architectures give rise to exploding and vanishing gradients?", in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018. [Online]. Available: https://arxiv.org/abs/1801.03744, (visited 20.05.2021).

[33] E. Hoffer, I. Hubara, and D. Soudry, *Train longer, generalize better: Closing the generalization gap in large batch training of neural networks*, 2018. arXiv: 1705.08741 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1705.08741, (visited 02.05.2021).

[34] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, and R. Horaud, "A comprehensive analysis of deep regression", *CoRR*, vol. abs/1803.08450, 2018. arXiv: 1803.08450. [Online]. Available: http://arxiv.org/abs/1803.08450, (visited 17.04.2021).

[35] A. MAYOR, *Gods and Robots: Myths, Machines, and Ancient Dreams of Technology.* Princeton University Press, 2018, ISBN: 9780691183510. [Online]. Available: http://www.jstor.org/stable/j.ctvc779xn.

[36] Asrst. (2019). DenseNet (tf-keras) on CIFAR-10, [Online]. Available: https://www.kaggle.com/asrsaiteja/densenet-tf-keras-on-cifar-10/notebook. (visited 30.03.2021).

[37] V. Christlein, L. Spranger, M. Seuret, A. Nicolaou, P. Král, and A. Maier, "Deep generalized max pooling", *CoRR*, vol. abs/1908.05040, 2019. arXiv: 1908.05040. [Online]. Available: https://arxiv.org/abs/1908.05040, (visited 19.05.2021).

[38] J. Flowers, "Strong and weak ai: Deweyan considerations", in *AAAI Spring Symposium: Towards Conscious AI Systems*, 2019. [Online]. Available: http://ceur-ws.org/Vol-2287/paper34.pdf, (visited 31.03.2021).

[39] J. Qi, J. Du, S. M. Siniscalchi, and C. Lee, "A theory on deep neural network based vector-to-vector regression with an illustration of its expressive power in speech enhancement", *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 27, no. 12, pp. 1932–1943, 2019. DOI: 10.1109/TASLP.2019.2935891.

[40] A. Shashkevich. (2019). Greek myths have some scary ideas about robots and a.i., [Online]. Available: https://www.futurity.org/artificial-intelligence-greek-myths-1999792/. (visited 31.03.2021).

[41] W. Yang. (2019). pytorch-classification, [Online]. Available: https://github.com/bearpaw/pytorch-classification. (visited 31.03.2021).

[42] (2020). A History of Artificial Intelligence, [Online]. Available: https://ahistoryofai.com/descartes-2/. (visited 31.03.2021).

[43] A. F. Carlos M. Alaız and J. R. Dorronsoro, "Visualization of the feature space of neural networks", *ESANN 2020 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2020. [Online]. Available: https://www.esann.org/sites/default/files/proceedings/2020/ES2020-130.pdf, (visited 28.04.2021).

[44] DATAI Team. (2020). Deep learning tutorial for beginners, [Online]. Available: https://www.kaggle.com/kanncaa1/deep-learning-tutorial-for-beginners. (visited 30.03.2021).

[45] ——, (2020). Pytorch tutorial for deep learning lovers, [Online]. Available: https://www.kaggle.com/kanncaa1/pytorch-tutorial-for-deep-learning-lovers. (visited 30.03.2021).

[46] J. Qi, J. Du, S. M. Siniscalchi, X. Ma, and C.-H. Lee, "On mean absolute error for deep neural network based vector-to-vector regression", *IEEE Signal Processing Letters*, vol. 27, pp. 1485–1489, 2020, ISSN: 1558-2361. DOI: 10.1109/lsp.2020.3016837. [Online]. Available: http://dx.doi.org/10.1109/LSP.2020.3016837.

[47] Ursula Laa, Dianne Cook. (2020). A slice tour for finding hollowness in high-dimensional data, [Online]. Available: https://distill.pub/2020/grand-tour/. (visited 31.03.2021).

[48] Vladimir Kunc, Jiří Kléma. (2020). TAAF-D-GEX, [Online]. Available: https://github.com/kunc/TAAF-D-GEX. (visited 30.03.2021).

[49] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice", *CoRR*, vol. abs/2007.15745, 2020. arXiv: 2007.15745. [Online]. Available: https://arxiv.org/abs/2007.15745, (visited 19.05.2021).

[50] Z. Zhang, J. Ren, Z. Zhang, and G. Liu, "Deep latent low-rank fusion network for progressive subspace discovery", *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, Jul. 2020. DOI: 10.24963/ijcai.2020/383. [Online]. Available: http://dx.doi.org/10.24963/ijcai.2020/383, (visited 30.03.2021).

[51] S. Bhattacharyya. (2021). Understand and Implement ResNet-50 with TensorFlow 2.0, [Online]. Available: https://towardsdatascience.com/understand-and-implement-resnet-50-with-tensorflow-2-0-1190b9b52691. (visited 30.03.2021).

[52] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, *Deep learning based text classification: A comprehensive review*, 2021. arXiv: 2004.03705 [cs.CL], (visited 17.04.2021).

Chapter 8

Appendix

This chapter will summarize implementation details.

8.1 Pytorch library

Tutorials [45] and [44] are being summarized below as an example of Neural network interface in Python language

Code overview

Initializers

```
# Neural Network wrapper class
class CustomModel(nn.Module):
    def __init__(self, dim1, dim2, dim3):
        # Create an empty wrapper with all variables
        super(CustomModel, self).__init__()

        # initialize every hidden layer function #

    def forward(self, X):
        # compute an output from X with hidden layer functions #


# Initialize a function y = XA + b
# Random parameters in <-sqrt(1/dim1), sqrt(1/dim1)>
# To access parameters: A = f.weights,    b = f.bias
f = nn.Linear(dim1, dim2)

# Initialize a function y = max(x, 0)
nn.ReLU()

# Initialize a function y = tanh(x)
nn.Tanh()

# Initialize a CrossEntropy(Softmax(out, real)) loss function
nn.CrossEntropyLoss(out, real)

# Initialize a MSE function loss function
nn.MSELoss()

# Optimization gradient function
# Parameters are Variables, learning_rate = number
torch.optim.SGD(parameters, learning_rate)
```

Training process

```
# Create objects
model = CustomModel(input_dim, hidden_dim, output_dim)
error = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.2)

# Train from some train data
# (image, label) is mini-batch
for (image, label) in train_data:
    optimizer.zero_grad() # Clear gradient
    outputs = model(train) # Calculate output
    loss = error(outputs, labels) # Calculate loss
    loss.backward() # Calculate gradient of loss function
    optimizer.step() # Move variables
```

8.2 Tensorflow library

The next library is more high-level and flexible for model definition because machine learning's main functionality is wrapped and prepared for us.

Initializers

```
import tensorflow as tf

# Wrapper objects of Neural Network
tf.keras.Model()  # Empty wrapper
tf.keras.Sequential() # Network without skip-connections

# Layers are added to wrapper
tf.keras.layers.Layer() # Empty Layer
tf.keras.Flatten(input_shape) # Reshape 2D data into 1D
tf.keras.Dense(shape) # Fully connected layer (W + bias)
model.add(Layer) # Add layer to model

# Optimizers
tf.keras.optimizers.SGD(l_rate) # Stochastic Gradient Descent
tf.keras.optimizers.Adam(l_rate) # Adam modifier

# Loss functions
tf.keras.losses.MeanSquaredError() # MSE
tf.keras.losses.BinaryCrossentropy() # Cross-E for 1D output

# Activation functions
tf.keras.tf.keras.activations.relu(x) # Non-linear ReLu
tf.keras.activations.sigmoid(x) # Linear sigmoid
tf.keras.activations.softmax(x) # SoftMax -> (0, 1)
```

Training process

```
model = Sequential() # Define model
# Add layers
model.add(Dense(10, activation='relu',input_shape=(n_features,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='sgd', loss='mse') # Compile model
model.fit(X, y, epochs=100, batch_size=32) # Train model
```

8.3 Used platforms

As the starting point, we use the **Google Colab** platform for the implementation of basic functionality. Unfortunately, this platform is merely capable of calculating thousands of loss surface points. Mainly, it provides only a 2 CPU hardware that is incapable of proper parallelization of this task.
For example, 8.1, which is a primitive loss surface, has been calculated in nearly a half-hour with 2 CPU parallelization.

Thus, we will move our calculations to the CESNET MetaCentrum[1] to achieve a more prominent picture of the Loss Surface.

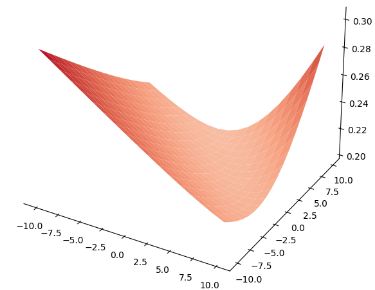


Figure 8.1: 20x20 loss surface of [48] [calculated on Google Colab platform]

8.4 High-Level overview

Training process

The first part is to train a model and save intermediate results of parameters.

```
# Any model definition and compile
model = model_initialization()

# Prepare save callback and define filepath with formatting
mcc = tf.keras.callbacks.ModelCheckpoint(filepath, ..)

# Create an array of all callbacks
callbacks = [mcc, ...]

# Train a model and add callbacks to it
# Saving of the parameters happens during training
model.fit(callbacks=callbacks, ..)
```

Visualization process

The second part is to compute the Loss Surface from saved parameters and model.

```
# Load our train data and model
loss_vis_obj = LossVisualisation(model, optimizer, loss_func, train_x, train_y)

# Load our checkpoints
# Calculate PCA components (already in it)
loss_vis_obj.load_from_checkpoints(...)
# OR generate random trajectories
loss_vis_obj.generate_rand_trajectories(...)

# Calculate an area of Loss Surface
area = loss_vis_obj.compute_mesh(...)

# Visualize calculated area
loss_vis_obj.visualize_3d()
```

[1] https://metavo.metacentrum.cz/

Random trajectories calculation

Layer normalization

random_traj.py

```
def generate_rand_trajectories(model):
    a = model.trainable_variables
    res = []
    for v in a:
        # Generate Gaussian random vector
        rand = tf.random.normal(v.shape)
        # Layer normalization
        rand = tf.math.scalar_mul(1/(tf.norm(rand, axis=None)) + 1e-10, rand)
        rand = tf.math.scalar_mul(tf.norm(v, axis=None), rand)
        # Add them to array(per filter)
        res.append(rand)
```

Filter normalization

random_traj.py

```
def filter_normalization(direction : np.ndarray, layer_variables: np.ndarray):
  rand_final = []
  for d, w in zip(direction, layer_variables):
    d = d * (1 / np.linalg.norm(d) + 1e-10) # |d| = 1
    w_size = np.linalg.norm(w) # get |w|
    d = d * w_size # |d| = |w|

    d = np.reshape(d, -1)
    rand_final = np.concatenate((rand_final, d), axis=0)
  return rand_final

def get_random_filnorm_trajectories(model):
  rand_dir = []
  orig_dir = []
  for param in model.trainable_variables:
    param_np = param.numpy();
    shape = param.shape
    d = None
    if param_np.ndim <= 1:
      d = np.zeros(shape=shape)
      d = np.reshape(d, -1)
    else:
      rd = np.random.normal(size=shape)
      d = filter_normalization(rd, param_np)
    rand_dir = np.concatenate((rand_dir, d), axis=0)

    param_f = np.reshape(param_np, -1)
    orig_dir = np.concatenate((orig_dir, param_f), axis=0)
  print('Final dimensions:')
  print(rand_dir.shape)
  return rand_dir, orig_dir
```

Full code is available on gitlab[2]

[2] https://gitlab.fel.cvut.cz/anuarali/loss-surface-bp