Stockholms universitet

Unveiling the Double Descent Phenomenon of Residual Networks in Deep Learning

Stefan Miletic

Handledare: Chun-Biu Li
Examinator: Josefin Ahlkrona
Inlämningsdatum: 2024-5-20

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Acknowledgements

I would like to offer my sincerest gratitude to my supervisor Chun-Biu Li for his extraordinary dedication to the project and invaluable expertise in the field. The countless hours spent together at the seminars have made this project an unforgettable learning experience.

Abstract

In deep learning, the residual networks (also referred to as ResNets) were developed and specialized for the purposes of image recognition. Ever since the first appearance of ResNets, people were experimenting with various architectural designs and benchmark tests, including training them on famous data sets such as CIFAR-10. One particular architecture is called ResNet-18, and in some cases, when trained on CIFAR-10, it can exhibit a surprising phenomenon called the "double descent". The double descent phenomenon is a term used to describe the behavior of a model's, or its family's, classification accuracy curve, also known as the test error curve. In this thesis, we reproduce the epoch-wise double descent of ResNet-18, trained on CIFAR-10, as discussed in [8]. Furthermore, we apply some additional experiments directly on the model's output layer in hopes to shed some light on the understanding of the root cause of double descent.

Contents

1 Introduction

One of the main challenges in deep learning deals with image recognition and classification [3, 11]. Some examples are recognition of handwritten text and classification of various objects or animals. In classification, perhaps the most famous data set is the **CIFAR-10** introduced in [1]. This data set contains thousands of images of various animals and vehicles. Most of the modern deep learning models train on CIFAR-10 as a benchmark test for their performance. One of these models is the residual neural network called **ResNet-18**, first introduced in [4], which has shown high accuracy. However, under certain circumstances, ResNet-18 exhibits a particularly interesting and surprising behavior called the **double descent** [8]. This phenomenon will be our subject of study. The disposition of the thesis is as follows:

i. First, we introduce some fundamentals of information theory, namely the notions of *entropy* and *cross-entropy*. This theory is used to develop the *cross-entropy loss* which is popularly used in classification problems for the purposes of measuring performance. The cross-entropy loss is an example of a *loss function* used by optimization algorithms in order to *train* models to perform well. Since model training is heavily dependent on the choice of a loss function and an optimization algorithm, we introduce optimization theory as well. The main algorithm which is discussed is the *gradient descent* and its popular variants in deep learning. Next, we give a step-by-step construction of a general neural network model called a *fully connected feedforward network*. We then explain how these models are trained and discuss some practical and theoretical aspects of network training, including the phenomenon of *double descent*.

ii. Second, we define the operation of *convolution* and show how it can be used to construct *convolutional layers* and *networks*. These types of networks are particularly useful in image recognition and classification. Furthermore, we also introduce the idea of *residual blocks* and *skip connections* which is used in the development of *residual networks*. We then present a famous family of models which unifies all of these concepts into a single architecture, namely ResNet-18.

iii. Finally, we reproduce the epoch-wise double descent discussed in [8] and perform additional studies directly on the output layer of ResNet-18. Our goal is to try to understand how the output layer geometry changes under the phenomenon of double descent, and hopefully shed some light on its root cause.

2 Prerequisites

In order to welcome a general reader to the broad field of deep learning, this special chapter is dedicated to some fundamental concepts necessary to get started. The material introduced here serves as a bridge to the main subjects of this thesis, namely the application of *residual neural networks* in classification discussed in Section 3.

2.1 Entropy and Cross-Entropy

One of the key ideas in information theory is **entropy**. When transmitting information, we somehow want to quantify how much of that information (usually encoded in bits) is useful to the recipient. Ideally, learning that an unlikely event has occurred should be more informative than learning that a likely event has occurred. For example, learning that our favorite *Formula 1* racer has participated in a race is less informative than learning that he has won that race. Furthermore, we also want two independent events to convey more information than either one of those isolated, i.e. we want the additive property on this quantification of information we seek. For example, learning that our mentioned racer has won two races should convey more information than learning that he has won one of those two races (assuming that the races are independent of each other).

One function that satisfies all of the mentioned properties we seek is the logarithm. We demonstrate the idea with another classic example — weather forecasting. Suppose hypothetically that there is a town on Earth where every day it is either rainy or sunny with equal probability. Encode the rainy weather by $r = 1$ and sunny by $s = 0$, i.e. by bits, and let their probabilities be $P(r) = P(s) = 0.5$. Now, when forecasting tomorrow's weather, our *uncertainty* reduces by a factor of 2 since before knowing the outcome there were two equally likely weather events. More importantly, no matter how we choose to encode the weather information, e.g. by longer strings such as "rainy" and "sunny" instead of bits 1 and 0, we will still only communicate 1 bit of useful information. Here, 1 arises as the binary logarithm of 2 — the number of equally likely events. We will explain this in more detail below.

Suppose now, more generally but still a bit unrealistically, that there are 16 weather events, each occurring with equal probability of 1/16 in our imaginary town. When forecasting the weather now, our uncertainty will reduce by a factor of 16, which is equal to 2^4. This exponent 4, which was 1 in the first example, since $2 = 2^1$, is how we can calculate the number of "useful" bits, i.e. by taking the binary logarithm of the *uncertainty factor*. The idea easily generalizes to arbitrary many weather events, say n, (still assumed to

be occurring with equal probabilities), and computing the number of "useful" bits is done by taking $\log_2(n)$.

Finally, we want to further generalize by allowing the events to attain arbitrary probabilities. Suppose now that our weather events, or any events essentially, are labeled by $E_{i\in I}$ and occur with probabilities $P(E_i) = p_i$ such that $\sum_i p_i = 1$. Then the number of bits tied to sending the information of an event E_i occurring is calculated by taking $\log_2(1/p_i) = -\log_2(p_i)$.

We are now ready to define entropy, but first, let us properly define the notion of "useful" bits which is called **self-information**.

Definition 2.1 (Self-information)**.** *Given a random event E occurring with probability $P(E) = p$, the number of **bits** associated to the occurrence of E, i.e. the self-information of E, is defined as*

$$I(E) \coloneqq -\log_2 P(E) = -\log_2 p.$$

*One **bit** of information is therefore defined as the amount of information gained by observing an event with probability* 0.5*.*

With this definition and previous examples in mind, we define **entropy** (also known as Shannon entropy) as follows:

Definition 2.2 (Entropy)**.** *Given a discrete probability distribution P over a random variable x, the entropy of P (or x) is defined as*

$$H(P) \coloneqq \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log_2 P(x)] = -\sum_i p_i \log_2 p_i,$$

where p_i are the probabilities of the outcomes of x.

We see from the definition of entropy that it essentially computes the average amount of information in an event drawn from P. Sometimes, above definitions use the natural logarithm instead, in which case the units are called *nats* instead of *bits*. Bits are also sometimes called *shannons* after the famous information theorist Claude Shannon, and entropy itself is also called Shannon entropy. In the case of continuous random variables, entropy is also called *continuous* or *differential entropy* [3].

There is another important measure closely related to entropy, and that is the **cross-entropy**. Going back to the example of weather forecasting, suppose that we want to transmit the information of one of the 16 weather events (occurring with equal probabilities) by efficiently encoding them in bits. Then each event would be represented by a string of 4 bits, i.e. 0000, 0001, 0010 etc. In this case, since the events occur with equal probabilities,

the cross-entropy is simply defined as the average message length, which is 4 and is equal to the entropy. This is the ideal scenario, however, if the probabilities of the underlying events are different, the cross-entropy is usually larger than the entropy, but also has an analogous generalization as was the case with entropy. The goal is then to efficiently encode the information in a sense that its cross-entropy does not diverge a lot from its entropy. For example, suppose that 14 out of 16 weather events occur with probability 0.01 and the remaining two with probability 0.43. Then encoding each event with 4 bits would assign the probability of $1/2^4 = 1/16$ to an event whose true[1] probability is $0.01 = 1/100$. There may also exist scenarios where some messages are longer than others, in which case the predicted distribution may not even add up to 1. Here is the definition of cross-entropy when the events occur with arbitrary probabilities:

Definition 2.3 (Cross-entropy)**.** *Given two discrete probability distributions P and Q over the same random variable x, the cross-entropy of Q relative to P is defined as*

$$H(P, Q) = \mathbb{E}_{x\sim P}[I_Q(x)] = -\mathbb{E}_{x\sim P}[\log_2 Q(x)] = -\sum_i p_i \log_2 q_i,$$

where $I_Q(x)$ is the self-information of x with respect to Q, and p_i and q_i are respective probabilities of the outcomes of x.

The key difference between entropy and cross-entropy is the fact that the self-information is computed with respect to Q instead of P when evaluating cross-entropy. We also mentioned earlier that we want to choose our encodings such that cross-entropy does not diverge a lot from entropy. This is an established measure known as the **Kullback-Leibler (KL) divergence** or **relative entropy** [3].

The takeaway concept for us in this subsection will be the definition of cross-entropy. We will see later when we introduce neural networks and network training how cross-entropy can be used to formulate a loss function. This *cross-entropy loss* will play a very important part later on and will be one of our main subjects of study and experimentation in the *double descent* phenomenon.

2.2 Optimization and Learning

The optimization theory plays a central part in deep learning as it allows us to *train* networks in order to improve their performance. More precisely, for any

[1]Here, we have used self-information to estimate the true probability distribution.

network model $\boldsymbol{y}(\boldsymbol{x};\boldsymbol{\theta})$, there is some *cost*, or *loss*, function E associated to it which tells us how well the model performs. In deep learning, the variable $\boldsymbol{\theta}$ represents a set of parameters that define a network model $\boldsymbol{y}$. A model $\boldsymbol{y}(\boldsymbol{x};\boldsymbol{\theta})$ can then be treated both as a function of inputs $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$. The term *training* refers to minimizing a loss function associated to $\boldsymbol{y}$ with respect to $\boldsymbol{\theta}$, which leads to some key ideas in optimization theory [9]:

Given a real valued loss function $E\colon \mathbb{R}^d \to \mathbb{R}$, where d is the dimension of the input, the objective is to find $\boldsymbol{\theta}^* \in \mathbb{R}^d$ such that

$$E(\boldsymbol{\theta}^*) \leq E(\boldsymbol{\theta}) \text{ for all } \boldsymbol{\theta} \text{ in some neighborhood of } \boldsymbol{\theta}^*.$$

Any such $\boldsymbol{\theta}^*$ is called a *local minimizer* of E, and if the neighborhood mentioned above is the whole $\mathbb{R}^d$, then $\boldsymbol{\theta}^*$ is also called a *global minimizer*. In practice, there could be many different local minimizers of E, and finding a global minimizer is often too expensive or not necessary. We are usually satisfied with a minimizer which gets the job done, i.e. which drives the network model to a desired accuracy.

In the field called *supervised learning*, we usually have an empirically observed data set of input/output pairs $D = \{(\boldsymbol{x}_i, \boldsymbol{t}_i)\}_{i\in I}$, indexed by some i from a finite set of indices I, which is used to construct E. This D is independent of $\boldsymbol{y}$ and is obtained by gathering some data associated to the problem we want to solve. For example, if we want $\boldsymbol{y}$ to recognize images $\boldsymbol{x}_i$ of cats and dogs, the data set D would be a collection of images of cats and dogs, with their matching labels $\boldsymbol{t}_i$, indicating whether $\boldsymbol{x}_i$ is a cat or a dog. In this sense, E can be seen to depend on D as well and not just $\boldsymbol{\theta}$, leading to a particular form of optimization called *learning* [3]. To be more concrete, we are interested in minimizing the so called *empirical risk* defined as

$$E(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} E_i(\boldsymbol{y}(\boldsymbol{x}_i;\boldsymbol{\theta}), \boldsymbol{t}_i), \tag{2.1}$$

where N is the number of elements in D, i.e. $N = |D|$, and E_i is a per-example loss function, or mismatch, associated to the pair $(\boldsymbol{y}(\boldsymbol{x}_i;\boldsymbol{\theta}), \boldsymbol{t}_i)$, with respect to $\boldsymbol{\theta}$. Hence for each pair $(\boldsymbol{x}_i, \boldsymbol{t}_i) \in D$ we evaluate $\boldsymbol{y}(\boldsymbol{x}_i;\boldsymbol{\theta})$ and compute $E_i(\boldsymbol{y}(\boldsymbol{x}_i;\boldsymbol{\theta}), \boldsymbol{t}_i)$. The sum of all such E_i is then averaged, creating a loss function $E(\boldsymbol{\theta})$ which needs to be minimized.

The optimization theory provides algorithms that can minimize (2.1). Perhaps the most famous such algorithm is the **gradient descent (GD)** (also called **steepest descent**) [9]. The idea is to iteratively approach a local minimizer by going in the direction of negative gradient:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla E(\boldsymbol{\theta}). \tag{2.2}$$

Here, α is the so called *step size* or *learning rate*, and E is assumed to be differentiable in a neighborhood of the initial $\boldsymbol{\theta}_0$. However, due to the fact that D can be extremely big, computing the gradient $\nabla E(\boldsymbol{\theta})$ is not always an easy task and may require extensive computational powers. For this reason, the GD is often not considered the best choice to minimize (2.1) and other methods are developed. The most popular alternative in deep learning is the **stochastic gradient descent (SGD)**. The key insight behind SGD is the fact that the gradient

$$\nabla E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} E_i(\boldsymbol{y}(\boldsymbol{x}_i; \boldsymbol{\theta}), \boldsymbol{t}_i) \tag{2.3}$$

of (2.1) can be seen as an expectation, which then can be approximated from a small set of samples [3]. Hence in each step in (2.2) we sample a small *minibatch* (or more formally a subfamily) $D' = \{(\boldsymbol{x}_i, \boldsymbol{t}_i)\}_{i \in K \subseteq I}$ from the *batch* D, where K is a smaller index subset of I often having between 1 and a few hundred pairs $(\boldsymbol{x}_i, \boldsymbol{t}_i)$, say $|D'| = M \ll N = |D|$, which are then used to approximate

$$\boldsymbol{g} \coloneqq \frac{1}{M} \sum_{i=1}^{M} \nabla_{\boldsymbol{\theta}} E_i(\boldsymbol{y}(\boldsymbol{x}_i; \boldsymbol{\theta}), \boldsymbol{t}_i) \approx \nabla E(\boldsymbol{\theta}). \tag{2.4}$$

The size of the minibatch D' is kept fixed in each step of the iteration and the SGD can be expressed as

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \boldsymbol{g}, \tag{2.5}$$

where, of course, α is the learning rate and $\boldsymbol{g}$ is sampled in each step.

The story of stochastic optimizers does not end with the SGD. There exist further generalizations that also employ the so called *adaptive learning rates* and the concept of *momentum*. Some famous examples are the **AdaGrad**, **RMSProp** and **Adam**[2], which have become standard in most deep learning applications [3]. To be consistent with [8], we will use Adam in Section 4.

2.3 Neural Networks

Inspired by the biological architecture of real neural networks, the *artificial neural networks* (ANNs), or briefly *neural networks* (NNs) play a central and fundamental role in deep learning. A neural network can be thought of as a

[2]See the original paper [7] for more details about the Adam optimizer, including the convergence analysis.

series of function compositions where a certain input information is propagated through the network's layers in order to obtain some final transformed output. For example, the input may be a picture of a cat or dog and the output a value between 0 and 1 indicating the probability that the provided input was indeed a cat (or dog). The reason why NNs are so important is because they allow us, given enough computational resources and time, to approximate any continuous function to arbitrary precision under certain conditions. This result is better known as the **universal approximation theorem** [2, 3], and when we take into account that functions can model almost any real world phenomena, e.g. image or speech recognition, it becomes clear why deep learning is so important.

In simplest form, we are given some input vector $\boldsymbol{x} \in \mathbb{R}^d$ of dimension d and start by introducing parameters called **weights** $\boldsymbol{w} \in \mathbb{R}^d$ and **biases** $b \in \mathbb{R}$ in order to transform $\boldsymbol{x}$ to $\boldsymbol{w}^T\boldsymbol{x} + b$. This is a straight-forward linear transformation of $\boldsymbol{x}$ to some constant which we will denote by $l = \boldsymbol{w}^T\boldsymbol{x} + b$. Suppose now that we want to use our initial input $\boldsymbol{x}$ to obtain n different new constants, say $\boldsymbol{l} = (l_1, \ldots, l_n)$, and let us call this $\boldsymbol{l}$ our first layer. Then for each l_i in this first layer we may want to use different sets of weights and biases, say $\boldsymbol{w}_i$ and b_i, to construct $l_i = \boldsymbol{w}_i^T\boldsymbol{x} + b_i$. More compactly, this whole process can be described in terms of linear algebra as $\boldsymbol{l} = W^T\boldsymbol{x} + \boldsymbol{b}$, where W is a weight matrix whose i'th column is $\boldsymbol{w}_i$, i.e. $W^T = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n)^T$, and $\boldsymbol{b} = (b_1, \ldots, b_n)$.

The next step is to introduce an **activation function** which is applied to the first layer $\boldsymbol{l}$ in order to introduce nonlinearities. For the most part, these functions should be differentiable since we will be interested in computing the gradient of some cost function associated to the underlying model. Generally, it is important to allow for nonlinear models since the majority of problems are highly nonlinear.

Continuing with our construction, an activation function, say $h(\cdot)$, is applied component-wise to $\boldsymbol{l}$ to obtain $h(\boldsymbol{l}) = (h(l_1), \ldots, h(l_n))$, or compactly

$$h(\boldsymbol{l}) = h(W^T\boldsymbol{x} + \boldsymbol{b}). \tag{2.6}$$

This is the way we define the first activated layer of our network model and its components, as well as components of any other layer, are called *nodes* or *neurons*.

As one might have already guessed, the next step is to treat (2.6) as the new input and proceed to the construction of the second layer. Let us relabel our $\boldsymbol{l}$ to $\boldsymbol{l}^{(1)}$ to indicate that it is the first layer and define the second layer as $\boldsymbol{l}^{(2)} = W_2^T h(\boldsymbol{l}^{(1)}) + \boldsymbol{b}_2$, where W_2 and $\boldsymbol{b}_2$ may be different from W^T and $\boldsymbol{b}$ used in the construction of $\boldsymbol{l}^{(1)}$. A similar new activation function, say $h_2(\cdot)$,

is then applied to $\boldsymbol{l}^{(2)}$ to obtain

$$h_2(W_2^T h(\boldsymbol{l}^{(1)}) + \boldsymbol{b}_2). \tag{2.7}$$

We can start observing from (2.7) how layers begin to stack as

$$h_2(W_2^T h(\boldsymbol{l}^{(1)}) + \boldsymbol{b}_2) = h_2(W_2^T h(W^T \boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{b}_2). \tag{2.8}$$

By generalizing (2.8) to m layers $\boldsymbol{l}^{(1)}, \ldots, \boldsymbol{l}^{(m)}$ and activations $h_1, \ldots, h_m$, a general network model $\boldsymbol{y}(\boldsymbol{x})$ will look like this:

$$\begin{aligned} \boldsymbol{y}(\boldsymbol{x}) &= h_m(W_m^T h_{m-1}(\boldsymbol{l}^{(m-1)}) + \boldsymbol{b}_m) \\ &= h_m(W_m^T h_{m-1}(W_{m-1}^T h_{m-2}(\boldsymbol{l}^{(m-2)}) + \boldsymbol{b}_{m-1}) + \boldsymbol{b}_m) \\ &\vdots \\ &= h_m(W_m^T h_{m-1}(\ldots h_1(W^T \boldsymbol{x} + \boldsymbol{b}) \ldots) + \boldsymbol{b}_m). \end{aligned} \tag{2.9}$$

The final layer is called the output layer, and the layers between the input and output layers are called *hidden* layers. If the number of hidden layers is greater than one, the model is usually called a *deep* neural network.

Some standard activation functions for a general layer $\boldsymbol{z} = (z_1, \ldots, z_n)$ are the **rectified linear unit** (ReLU) defined by $z_i \mapsto \max\{0, z_i\}$ and the **softmax function** defined by $z_i \mapsto e^{z_i} / \sum_{j \neq i}^{n} e^{z_j}$. The ReLU activation is recommended for almost any feedforward model, both in classification and regression, as it efficiently introduces nonlinearities. This can be seen from its definition where the negative components of the input are rectified to zero. A rule of thumb is as follows — the more ReLU activations there are in a model, the more nonlinear function the model can approximate [3]. On the other hand, softmax is mainly used in classification problems on the output layer. It allows us to transform the output to a probability distribution over its components. For example, suppose that the output is given by $\boldsymbol{y} = (y_1, \ldots, y_n)$. Then each of the components y_i could theoretically take any real value. However, after the softmax activation, the output becomes normalized, preserving its component-wise relative magnitude due to softmax being a monotone function, and allowing the output to be transformed to a probability distribution. If the goal of the output $\boldsymbol{y}(\boldsymbol{x})$ is to classify $\boldsymbol{x}$, then each of y_i can be regarded as a relative probability that $\boldsymbol{x}$ belongs to class $i = 1, \ldots, n$. For example, if there are three classes and the unactivated output for $\boldsymbol{x}$ is $\boldsymbol{y}(\boldsymbol{x}) = (1, 2, 3)$, then $\text{softmax}(1, 2, 3) \approx (0.09, 0.245, 0.67)$, preserving the relative magnitudes of $(1, 2, 3)$ and normalizing the output to a probability distribution.

We conclude this subsection with a couple of observations. The model in (2.9) where every node is obtained by activating linear combinations of previous layer's nodes is called *a fully connected feedforward* network. The term *fully connected* indicates that every node is determined by all the nodes of the previous layer, and the term *feedforward* describes a general propagation of the input from the first layer all the way to the last one so that at no point we go from one layer to a previous one. There exist more advanced networks where inputs are allowed to propagate backwards, e.g. *recurrent networks* (RNNs), but also feedforward networks that are not fully connected, e.g. *convolutional networks* (CNNs) which will be a big part of our discussion [3].

2.3.1 Network Training

We mentioned earlier when we introduced optimization theory that network training directly increases accuracy of a given model by minimizing some sort of a loss function. To be more concrete, suppose that we are given a network model $\boldsymbol{y}$. When dealing with inputs $\boldsymbol{x}$, we usually treat $\boldsymbol{y}$ as a function of $\boldsymbol{x}$, i.e. $\boldsymbol{y}(\boldsymbol{x})$. However, this is only useful when we want to evaluate $\boldsymbol{y}$ on a fixed set of weights and biases. For example, we may feed some vector $\boldsymbol{x}_1$ to $\boldsymbol{y}$ which represents an image of a cat or dog, and the model $\boldsymbol{y}$ will return a value $\boldsymbol{y}(\boldsymbol{x}_1) \in [0, 1]$ indicating how likely the input is to represent a cat/dog. The main question then is how to teach the model to recognize images of cats and dogs. The answer is by *training* it on a known set of cats and dogs which is labeled correctly.

The first step is to treat $\boldsymbol{y}(\boldsymbol{x})$ as a function of its parameters as well, say $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a high dimensional vector consisting of all the weights and biases associated to every layer. The goal is then to find an optimal $\boldsymbol{\theta}$ which makes $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$ "recognize" cats/dogs, i.e. minimizes some loss function. This is accomplished in the following way. First we initialize a network model $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$. The initial set of parameters, say $\boldsymbol{\theta}_0$, is then set to be normally or uniformly distributed around zero with small variance, indicating that the neurons are still "weakly connected" and are yet to begin interacting with each other. When we have fixed our architecture and initialized parameters $\boldsymbol{\theta}_0$, we need to associate a loss function to $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$. One standard loss function for regression is the *mean squared error* (MSE) defined as

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_i; \boldsymbol{\theta}) - \boldsymbol{t}_i\|^2, \tag{2.10}$$

where in the context of training, N is the number of known input/output pairs $(\boldsymbol{x}_i, \boldsymbol{t}_i)$. Each input $\boldsymbol{x}_i$ can e.g. represent a point in the domain of a

function we want to approximate by our model $\boldsymbol{y}$, and the corresponding output $\boldsymbol{t}_i$ a true value of that function evaluated at $\boldsymbol{x}_i$. This kind of regime where we have a *training* set of input/output pairs provided in advance is called *supervised learning*. In contrast to that, there is also *unsupervised learning* where no training set is given to us and the model is expected to extract or predict some pattern or information from data [3].

Lastly, we apply an optimization algorithm to our chosen loss function with the initial $\boldsymbol{\theta}_0$ in order to minimize the mismatch between the model's prediction $\boldsymbol{y}(\boldsymbol{x}_i; \boldsymbol{\theta})$ and the true target value $\boldsymbol{t}_i$. This is how network training is defined — as minimization of a loss function with respect to $\boldsymbol{\theta}$. It is also important to mention that, although network models are trained on known sets of data, they are not tested on the same data. Usually some sort of a split needs to be made on the available data, e.g. a 70%-30% training/testing split, creating a new *testing* set on which the trained model's accuracy is tested. This is crucial because it allows us to monitor how the model generalizes to unseen data.

There exist many other loss functions and variants of (2.10) such as *root mean squared error* (RMSE), *mean absolute error* (MAE), but also functions such as the *cross-entropy loss*. The deciding factor of choosing an appropriate loss function lies in the nature of the task we are facing. If we are e.g. doing regression, it is usually beneficial to consider variants of MSE, and for classification problems such as the running example of cats and dogs, the cross-entropy loss might yield better results. Finally, as announced earlier, here is the definition of the **cross-entropy loss** which will be our choice in the implementations in Section 4 later:

Definition 2.4 (Cross-entropy loss for classification)**.** *Suppose that to each valid input $\boldsymbol{x}$ a model $\boldsymbol{y}(\boldsymbol{x})$ assigns a probability distribution $\boldsymbol{q} = (q_1, \ldots, q_n)$ such that $\sum_{i=1}^{n} q_i = 1$ and q_i is the probability that $\boldsymbol{x}$ belongs to class i. Moreover, suppose that $\boldsymbol{p} = (p_1, \ldots, p_n)$ is the true probability distribution associated to $\boldsymbol{x}$, i.e. there is an $i = k$ such that $p_k = 1$ and $p_i = 0$ for all $i \neq k$. Then the cross-entropy loss of $\boldsymbol{x}$ is defined as*

$$-\sum_{i=1}^{n} p_i \ln q_i = -\ln q_k.$$

We usually use the natural logarithm when computing the cross-entropy loss since it plays well with optimizers and popular activation functions such as **softmax**. Another point worth noticing is that true distributions $\boldsymbol{p}$ in the definition above have the form of the so called *one-hot* vectors, meaning that they are mostly zero and only one component is 1. For this reason, when

computing the standard cross-entropy of $\boldsymbol{q}$ relative to $\boldsymbol{p}$, only one product in the expectation contributes with a non-zero value. This is also the reason why the cross-entropy loss is sometimes called the **log loss** in classification. Notice also that when $q_k \approx 1$, the loss is close to 0, and when $q_k \approx 0$, the loss increases drastically. This is an important property because when we e.g. want to evaluate the cross-entropy loss over a minibatch of samples, having a few misclassified samples can dominate the overall loss[3].

2.3.2 Training in Minibatches

When using very large data sets, the training set is usually split into smaller subsets of equal size called *minibatches*. This technique is e.g. used in the gradient based stochastic optimizers that we mentioned earlier. Consequently, when reporting losses, people usually take the average loss over some number of minibatches.

For each minibatch of samples, a stochastic optimizer will estimate the true gradient by computing (2.4). However, even the computation of the estimated gradient in (2.4) is highly nontrivial due to the fact that a model can have millions of parameters. This issue is solved by a famous gradient estimation technique called **backpropagation**[4], which allows us to efficiently compute the gradient of a network model by a clever use of the chain rule. The computed gradient is then used to drive the network's parameters to a local minimizer in order to minimize the loss function.

Now, when the optimizer has seen all the samples from all minibatches, we usually repeat the process and start over. This is called training in *epochs*, where each epoch denotes a whole new pass over the train samples. It is important to train in epochs in order to achieve high accuracy, i.e. the optimizer has to see the entire training set many times (often hundreds of times) in order to find a satisfactory minimum. If the number of epochs is too low, there is a risk that the model will not be trained well, leading to higher losses and lower accuracy. However, the number of epochs should not be too large either because it can put additional computational strain, but also make the model perform worse over time. In fact, this issue of finding a perfect number of epochs is closely related to a concept of *overfitting and underfitting* discussed in the next subsection.

Here is a concrete example. Suppose that the data has 50k/10k train/test samples respectively (this is the case with CIFAR-10 data in Section 4). If we set the minibatch size to 100, we will end up creating 500/100 minibatches

[3]This is a very important observation that we will refer to in Section 4.

[4]See the original paper [10] for more details about backpropagation, and [11] for recent implementations.

respectively, each of which contains 100 samples in it. The optimizer will then perform 500 steps (*backward* passes), each time using 100 samples to compute a *running loss*

$$\sum_{i=1}^{100} E_i(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{t}_i). \tag{2.11}$$

The *running* loss is then averaged and reported for all the N minibatches

$$\frac{1}{N}\sum_{n=1}^{N}\sum_{i(n)=1}^{100} E_i(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{t}_i), \tag{2.12}$$

where n is a minibatch index and $i(n)$ a sample index in minibatch n. In the simplest case, we can set $N = 500$ (and $N = 100$ for the test set), i.e. calculate the running loss for each minibatch and average over the total number of minibatches.

In the context of (2.1), we can think of the sum (2.11) as each individual E_i in (2.1). In this sense, we may then relabel (2.12) as

$$E = \frac{1}{N}\sum_{n=1}^{N} E_n, \tag{2.13}$$

where

$$E_n = \sum_{i(n)=1}^{100} E_i(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{t}_i). \tag{2.14}$$

Hence per-example losses E_i in (2.1) become minibatch-wise losses E_n when we train a model in minibatches.

2.4 Underfitting and Overfitting

Some of the most interesting performance checks one can do on a network model is to look at the so called *learning curves*. These include the graphs of train and test losses as functions of epochs. For example, an ideal *healthy learning* would show a train loss monotonically decreasing to zero, and a test loss resembling a U-shaped curve. The train loss should also, for the most part, be lower than the test loss across all epochs since the model is expected to perform better on the data it has seen (train samples) than the data is has never seen (test samples).

Of particular interest is the behavior of the test loss, which is also called the **generalization error**. If the training was satisfactory, i.e. the model has reached some desired accuracy, the test loss will in most cases undergo the

so called **underfitting** and **overfitting regimes**. The underfitting regime occurs in the early stages of training when the model is still learning. This is graphically observed as a rapid decrease in both the test and train losses. Then, by proceeding with the training, the model will enter the overfitting regime where it will start fitting train samples extremely well. The train loss is then seen to asymptotically approach zero but the test loss begins to increase due to model's overfitting on the train samples and poor generalization to the test samples. Here is an illustration in Figure 1 taken from Section 4 while working on some of the experiments:

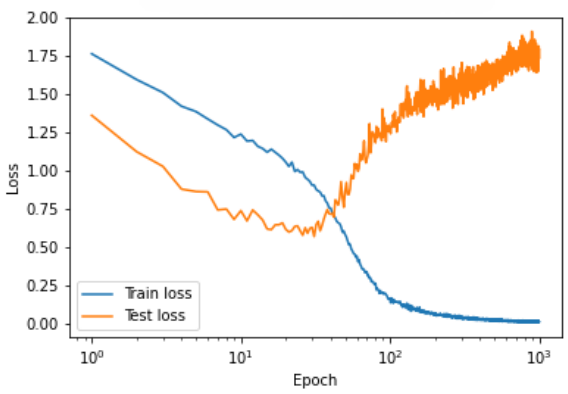


Figure 1: Learning curves of a network model, showcasing the U-shaped behavior of the test loss and convergence to zero of the train loss. The losses are the cross-entropy loss.

As we can see in Figure 1, the test loss can be seen as a U-shaped curve, while the train loss approaches zero. It is worth mentioning that the losses may not always intersect, and we usually, in simple models, get the train loss below the test loss across all epochs, even the early ones. Furthermore, the smoothness of these curves may vary and sometimes oscillate. It is important to keep in mind that each model, combined with the training details, will generally yield a different result. Also, depending on the nature of these factors, we usually get different results even for repeated runs. For example, the optimizers such as the SGD and Adam are stochastic in nature, hence the results are inherently stochastic.

The idea of underfitting and overfitting also generalizes to other notions of losses or errors. In classification, for example, we can define **errors** (not to be confused with previous losses) as $1 - C/T$, where C is the number of correctly classified test samples and T is the total number of test samples. We can then construct a family of models by gradually increasing their model

complexity in some sense, e.g. by increasing the number of nodes per layer, and finally plotting errors versus model size (or errors versus epochs for a particular model). A similar U-shaped curve may occur for the test error as capacity (or epoch) increases. A conventional practice for smaller families of models is then to pick the model which minimizes the test error right before it starts rising in the U-shaped curve. This is known as the optimal **early stopping** and is related to a concept from statistical learning called the **bias-variance tradeoff** [3]. The wisdom is that after a certain point, each model will perform worse due to overfitting. It is usually thought that larger models, due to having increasingly more parameters, begin to fit the smallest details of train samples, leading to overfitting. This will, in turn, lead to an increased error on test samples. A simple example is given in regression where we try to interpolate a finite number of points by polynomials. The larger the degree of a polynomial is, the better we fit the points, but also the larger the variance becomes. Although, the test loss remains to be U-shaped.

Interestingly enough, the test error demonstrating a U-shape in the first part of the training may sometimes descend again, leading to a phenomenon called the **double descent**. This is the scenario for most of the modern deep learning models. Figure 2 shows an example taken from Section 4, which also happens to be the corresponding error plot of the model in Figure 1:

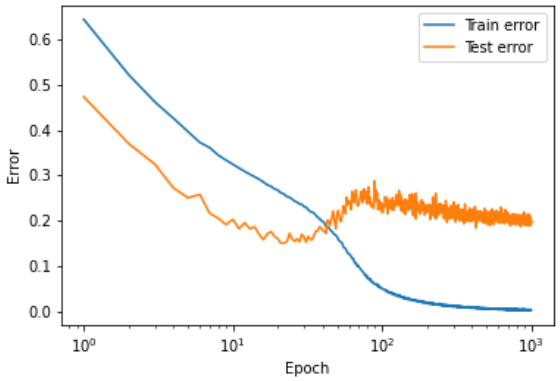


Figure 2: Error curves of a network model, showcasing the double descent behavior of the test error and convergence to zero of the train error.

We see in Figure 2 that the U-shaped error curve can indeed undergo the double descent. This is a very special phenomenon which will be explored in great detail in Section 4.2.1.

We finish this subsection with a very important observation — the choice of a generalization error can completely change our perspective on the learn-

ing curves. Figures 1 & 2 illustrate two perspectives of the **same** model, showing both the classical U-shaped curve and the special double-descent curve. It is therefore particularly interesting to ask what might lead one curve to a standard U-shape but the other curve to a double descent. This is another big question we try to answer in Section 4.2.1.

3 Advanced Neural Networks

In this section we start looking at more advanced feedforward architectures. We introduce the operation of *convolution* in order to develop *convolutional layers* and *networks*. We also discuss more abstractly about network layers and their functionality, and we try to come up with techniques to increase their efficiency. This will, in turn, lead us to the concept of *residual blocks* and *residual networks*. Finally, we combine both convolution and residual blocks to introduce a famous family of residual networks called **ResNet-18**.

3.1 Convolutional Networks

For certain tasks such as image recognition, one of the biggest advantages of convolutional networks (CNNs) is their parameter sparsity. In contrast to standard fully connected feedforward networks, CNNs employ much fewer (often multiple orders of magnitude) neurons, which leads to tremendous computational benefits. These networks are still considered feedforward since the data is never processed backwards, and the key idea behind them is the operation of convolution. Essentially, CNNs can be regarded as a special case of fully connected NNs where some parameters are set to zero and some are repeated for multiple neurons. We start by introducing the operation of discrete convolution in 1D:

Definition 3.1 (Discrete convolution)**.** *Given a pair of functions* $f, g \colon \mathbb{Z} \to \mathbb{R}$*, the discrete convolution of* $f(n)$ *and* $g(n)$ *is defined as*

$$(f * g)(n) \coloneqq \sum_{u=-\infty}^{\infty} f(u)g(n-u).$$

We can immediately see from the definition, by shifting the variable of summation to $n - u$, that convolution is commutative, i.e. $f * g = g * f$.

To give a concrete example, we may assume that f and g are defined, or non-zero, at a finite number of points. In a computer science terminology, we can think of f and g as arrays, e.g. $f = \{a_1, \ldots, a_j\}$ and $g = \{b_1, \ldots, b_k\}$. If we assume without loss of generality that $k \leq j$, the convolution of f and g

for all n would correspond to first reversing the order of elements in g, then "**sliding**" g along f, i.e.

$$(f * g) = \{a_1b_1, a_1b_2 + a_2b_1, \ldots, a_1b_k + \cdots + a_kb_1, \ldots, a_jb_k\}.$$

To make it even simpler, assume that $f = \{2, 4, 6\}$ and $g = \{0.5, 1\}$. Then

$$\{2, 4, 6\} * \{0.5, 1\} = \{2 \cdot 0.5, 2 \cdot 1 + 4 \cdot 0.5, 4 \cdot 1 + 6 \cdot 0.5, 6 \cdot 1\} = \{1, 4, 7, 6\}.$$

Note that, in regards to the summation bounds in Definition 3.1, we assume that the list elements whose indices exceed these bounds are set to 0.

In the context of CNNs, we often denote the functions to be convolved by $\boldsymbol{x}$ and $\boldsymbol{w}$, then call $\boldsymbol{x}$ an input and $\boldsymbol{w}$ a *kernel* or *filter*. The output $\boldsymbol{x} * \boldsymbol{w}$ is often called a *feature map*. We can think of $\boldsymbol{x}$ as an input which is to be propagated to the next layer, and of $\boldsymbol{w}$ as a collection of weights and biases which slides over the input and is learned by training.

The discrete convolution is easily generalized to n-dimensions. In CNNs, mostly the 2D variant is used. In this case, we can think of $\boldsymbol{x}$ as an array of arrays or matrix (e.g. a greyscale image) and of $\boldsymbol{w}$ as a smaller parameter matrix. The filter $\boldsymbol{w}$ then slides over $\boldsymbol{x}$ in some desired fashion, computing $\boldsymbol{x} * \boldsymbol{w}$.

Definition 3.2 (Discrete convolution in 2D)**.** *Given a pair of functions $f, g\colon \mathbb{Z}^2 \to \mathbb{R}$, the discrete convolution of $f(i, j)$ and $g(i, j)$ is defined as*

$$(f * g)(i, j) \coloneqq \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n)g(i - m, j - n).$$

As was the case in Definition 3.1, the 2D convolution is also commutative. However, although this property might be useful in theory, it is not generally used in CNNs. By simply redefining the summation indices we can introduce a new non-commutative convolution called *cross-correlation* [3]:

Definition 3.3 (Discrete cross-correlation in 2D)**.** *Given a pair of functions $f, g\colon \mathbb{Z}^2 \to \mathbb{R}$, the cross-correlation of $f(i, j)$ and $g(i, j)$ is defined as*

$$(f * g)(i, j) \coloneqq \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m + i, n + j)g(m, n).$$

Most CNN implementations use the cross-correlation convolution, and the term convolution is used ambiguously. The primary motivation for using cross-correlation is entirely practical. When implementing CNNs, we usually deal with 2D inputs such as images. Hence computing $f(m+i, n+j)g(m, n)$

instead of $f(m,n)g(i-m,j-n)$ for all desired pairs of (m,n) is much simpler because the kernel is kept fixed and we only need to iterate over the input.

Here is an example where the kernel is kept inside the boundaries of the input matrix and slides left-to-right and top-to-bottom (this is a standard convention in most implementations):

$$\begin{bmatrix} 1 & 4 & 5 & 2 \\ 1 & 0 & 7 & 5 \\ 1 & 5 & 2 & 6 \\ 7 & 3 & 7 & 8 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 0 \cdot 2 & 5 \cdot 1 + 5 \cdot 2 \\ 1 \cdot 1 + 3 \cdot 2 & 2 \cdot 1 + 8 \cdot 2 \end{bmatrix} = \begin{bmatrix} 1 & 15 \\ 7 & 18 \end{bmatrix}.$$

Finally, a **convolutional network** is any network containing at least one layer being transformed by the operation of convolution. Similarly to the weight and bias matrices and vectors in (2.9), there is an analogous way to express convolution in terms of matrices. In 1D, it corresponds to multiplication by a *Toeplitz matrix*, and in 2D to multiplication by a *doubly block circulant matrix* [3].

3.1.1 Pooling, Padding and Strides

There are some important practical details which need to be addressed. In the context of 2D convolution, we usually think of inputs as images, e.g. a 32×32 matrix, and of filters as smaller weight matrices, e.g. a 3×3 filter, which slide over the 32×32 image (left-to-right and top-to-bottom). In terms of Definitions 3.2 & 3.3, we set a bound on m and n in such a way so that the filter g is always entirely contained within f, i.e. we do not consider m and n for which f is not defined. The feature map can then be anything, depending on the values in the 3×3 filter and the way it slides over the image. This "way of sliding" over an image is formalized by the notions of **pooling**, **padding** and **strides**.

Starting with **pooling**, there are two major variants, namely **maximum** and **average pooling**. Suppose for a moment that the previously mentioned 3×3 filter which slides over a 32×32 image has no weights, but is rather used as a detection tool to read off values from the respective image. If we choose to do maximum pooling, then each time this detection filter slides over a 3×3 portion of the 32×32 input image, it will only pick up the maximum value from that 3×3 portion. Similarly, in average pooling, the 3×3 detection filter will pick up the average of the $3 \times 3 = 9$ pixels from that portion of the input image. This is how pooling operates.

Next, we have **padding**. This technique is straight-forward and refers to putting an input image in a "frame of zeros". To be more precise, if the input is a 32×32 image containing some values in each pixel, then padding this

image would refer to transforming it to a 33×33 image whose each edge is just zeros, and the non-edge elements are kept the same. It is also possible to apply thicker padding simply by repeating the process to a previously padded picture. This technique is particularly useful if we want to put some extra attention to the input's borders, but also if we want to increase its dimensions.

Lastly, and perhaps most importantly, we have a technique to evaluate convolution (and cross-correlation) in simple patterns called **strides**. Notice how in Definitions 3.2 & 3.3 we only compute $(f * g)$ for some choices of (i, j). If we choose to compute $(f * g)$ for all such (i, j), we will end up doing something analogous to the example we had earlier when we computed $\{2, 4, 6\} * \{0.5, 1\}$. However, we sometimes want to introduce sparsity and not consider each possible choice of (i, j), but rather each (im, jn) for some fixed m and n. These values of m and n denote the number of rows and columns traversed in each slide of g over f and are referred to as **strides**.

We conclude by a simple example of cross-correlation incorporating all three techniques. We set strides of 3 and 2 for height and width respectively, and add a single layer of padding:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 2 \times 2 \\ \max \\ \text{pooling} \end{bmatrix} = \begin{bmatrix} \max \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} & \max \begin{bmatrix} 0 & 0 \\ 2 & 3 \end{bmatrix} \\ \max \begin{bmatrix} 0 & 7 \\ 0 & 0 \end{bmatrix} & \max \begin{bmatrix} 8 & 9 \\ 0 & 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 7 & 9 \end{bmatrix}.$$

It is also important to mention that we usually want the the feature map to be in a predetermined shape. For example, if the input is a 32×32 image, maybe we also want the feature map to be a 32×32 image. This is achieved by properly tuning strides and padding. Furthermore, in practice, most of the images are represented with three **channels** known as the RGB (red, green and blue). In this case, the input $\boldsymbol{x}$ is no longer a 32×32 matrix, but a $32 \times 32 \times 3$ tensor. The operations are applied in the same manner as before but channel-wise.

3.2 Residual Networks

Suppose that we are given a network model $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$ and wish to approximate some reasonable function $f^*(\boldsymbol{x})$, e.g. a classifier or perhaps a continuous function. This task, of course, falls under the umbrella of supervised learning and might be solved by training $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta})$ on an adequate training set. Now, since the architecture of our model is fixed, there should exist some class of functions $\mathcal{F}$ such that for any $f \in \mathcal{F}$ we can achieve $\boldsymbol{y}(\boldsymbol{x}; \boldsymbol{\theta}) \approx f(\boldsymbol{x})$ by

tuning $\boldsymbol{\theta}$ via training. If we were lucky in our construction of $\boldsymbol{y}$, we will be able to approximate f^* well, i.e. $f^* \in \mathcal{F}$. However, there is no guarantee for this to be the case and we somehow want to expand our model so that it encompasses a larger family than $\mathcal{F}$. The key aspect of this idea is to allow the new model, say $\boldsymbol{y}'$, to keep its old capabilities, i.e. the class of functions it can reach, while also allowing it to expand. Hence we want $\boldsymbol{y}'$ with $\mathcal{F} \subseteq \mathcal{F}'$, where $\mathcal{F}'$ is the expanded model's family of approximatable functions [11].

One way to expand a network's $\mathcal{F}$ is to add a hidden layer, but with a small caveat — the new hidden layer should be trainable to include the identity function. This way, the network will keep the original capabilities, but also expand them, i.e. $\mathcal{F} \subseteq \mathcal{F}'$. By repeating this process and keep adding new layers that are trainable to include the identity function, we will end up creating a chain of nested function classes and a consistent way to keep expanding a model. This is the central idea behind *residual blocks* which was originally used in [4, 5] to construct *residual networks*, also called *ResNets*.

Here is how these ideas are used in the implementations. Suppose that we are given a network model, and instead of looking at the input layer getting transformed to the final output layer, we consider some *block* (or *subnetwork*) of n intermediate hidden layers which maps a hidden input layer $\boldsymbol{x}$ to some other hidden output layer $H(\boldsymbol{x})$. We incorporate the idea of nested function classes by transforming $H(\boldsymbol{x})$ to a block which ought to learn the identity function easily. This is accomplished by first reparametrizing $H(\boldsymbol{x})$ to the *residual mapping*

$$F(\boldsymbol{x}) \coloneqq H(\boldsymbol{x}) - \boldsymbol{x}, \tag{3.1}$$

and then defining the *residual block* by adding $\boldsymbol{x}$ directly to $F(\boldsymbol{x})$, i.e.

$$F(\boldsymbol{x}) + \boldsymbol{x}. \tag{3.2}$$

Note that $H(\boldsymbol{x})$ and $\boldsymbol{x}$ are assumed to be of equal dimensions. If they are not, there are simple workarounds such as e.g. passing $\boldsymbol{x}$ through a 1×1 convolutional filter with appropriate padding. This new connection between the input $\boldsymbol{x}$ and the output $F(\boldsymbol{x})$ established via addition in (3.2) is called a **skip** or **shortcut connection**. The name comes from the fact that for a residual block (3.2), $\boldsymbol{x}$ can pass both through the residual mapping, where it is affected by weights and biases, but also skip any effects of weights and biases and arrive directly at the output of the residual mapping. The idea is schematically illustrated in Figure 3:

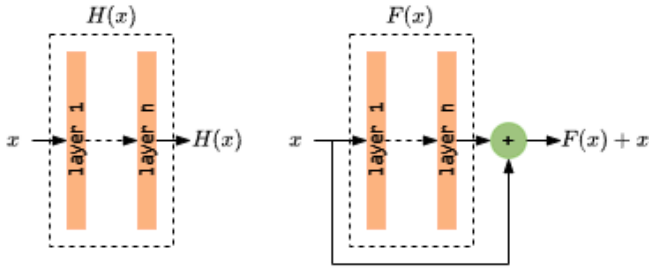


Figure 3: An illustration of a block $H(x)$ on the left and its residual block $F(\boldsymbol{x}) + \boldsymbol{x}$ on the right, where the input $\boldsymbol{x}$ in the residual block can pass both through the residual mapping $F(\boldsymbol{x}) \coloneqq H(\boldsymbol{x}) - \boldsymbol{x}$, but also through the shortcut connection.

Now, if the goal is to learn the identity, the old block $H(\boldsymbol{x})$ would have to do so directly, whereas the new residual block $F(\boldsymbol{x}) + \boldsymbol{x}$ would only have to push $F(\boldsymbol{x})$ to zero, which is generally much easier and more plausible.

Intuitively speaking, we can think of residual blocks as shortcuts used by inputs to propagate faster through a network. In contrast to standard feedforward networks where inputs need to pass layer by layer, in residual networks, inputs can skip layers by passing through skip connecting. This way, residual blocks expand the internal "infrastructure" of a standard feedforward model, which was the key idea behind ensuring the nested structure of function classes by increasing the number of hidden layers trainable to include the identity function. Finally, a **residual network** is any network that incorporates residual blocks/skip connections.

3.3 Famous Architectures

We are finally ready to introduce one of the most famous residual network architectures popularly used in image recognition, namely **ResNet-18** [4]. This architecture combines the ideas of residual blocks and convolutional layers, but also utilizes a novel technique called **batch normalization** [6]. There are several different ways to construct residual blocks in ResNet-18, leading to its several different variants. Each one of them fixes one residual block and uses it repeatedly throughout the entire network. This block is usually called a *basic block*, and the one we use in Section 4, and which is used in [8], is called a **full pre-activition block** [5]. However, before we present this block, we need to say a couple of words about batch normalization.

We will not go through many details and will simply state that the main purpose of batch normalization is to improve the training efficiency of deep

neural networks.[5] The idea is to periodically **normalize** layers. To be more concrete, suppose first that $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ is a minibatch of input samples which will be jointly used by a stochastic optimizer in the training procedure. Furthermore, suppose that $H = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N\}$ is a corresponding minibatch of activations $\boldsymbol{h}_i$ associated to a layer we want to normalize. We first compute the empirical mean and standard deviation of H:

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{h}_i, \quad \boldsymbol{\sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{h}_i - \boldsymbol{\mu})^2}. \tag{3.3}$$

For numerical stability, a small $0 < \delta \ll 1$ is usually introduced to $\boldsymbol{\sigma}$, hence it is redefined as

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{h}_i - \boldsymbol{\mu})^2}. \tag{3.4}$$

We then use $\boldsymbol{\mu}$ and this new $\boldsymbol{\sigma}$ to normalize each $\boldsymbol{h}_i \in H$ as

$$\boldsymbol{h}'_i = \frac{\boldsymbol{h}_i - \boldsymbol{\mu}}{\boldsymbol{\sigma}}. \tag{3.5}$$

Hence, a new normalized minibatch of activations $H' = \{\boldsymbol{h}'_1, \ldots, \boldsymbol{h}'_N\}$ is formed on which the network operates, as it would have done with H.[6]

Now that we have defined batch normalization, we can look at Figure 4 to see an illustration of the pre-activition block we use:

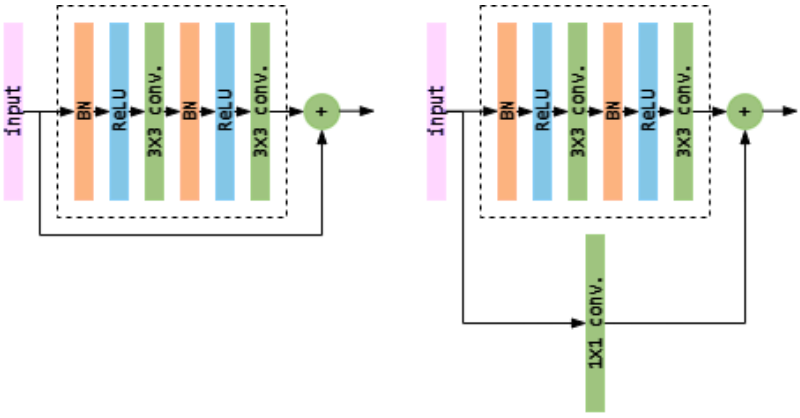


Figure 4: The pre-activation block, with and without the 1×1 convolution which reshapes the input into the desired shape for addition. These two blocks are the building components for a ResNet-18 model.

[5] See [6] for the original introduction of batch normalization and [3, 11] for more recent discussions and implementations.

[6] People usually further transform (3.5) to $\gamma_i \boldsymbol{h}'_i + \beta_i$, where the parameters γ_i and β_i are learned by training. For more details, see [3, 6, 11].

The block inside the dashed rectangle in Figure 4 corresponds to a residual mapping (3.1). By adding a shortcut connection to it, we end up creating a residual block called the **pre-activation block** [5]. However, the shape of the input sometimes does not match the shape of the output from the residual mapping. This is resolved by passing the input through a 1×1 convolutional layer with appropriate padding, transforming it into the desired shape for addition[7]. The input to the pre-activation block is usually a $H \times W \times C$ image tensor, where $H \times W$ describes the shape of the image (height $\times$ width) and C is the number of channels. In the standard RGB presentation, the number of channels is 3 (red, green and blue). Depending on the details in convolutional layers, this input will get reshaped as it passes through blocks. The way it changes in ResNet-18 is discussed below after the introduction of the full model. The layers are also color coded and stand for batch normalization (BN), 3×3 & 1×1 convolution and ReLU.

Finally, we use the two variants of the pre-activation block in Figure 4 as building blocks for a full ResNet-18 model in Figure 5:

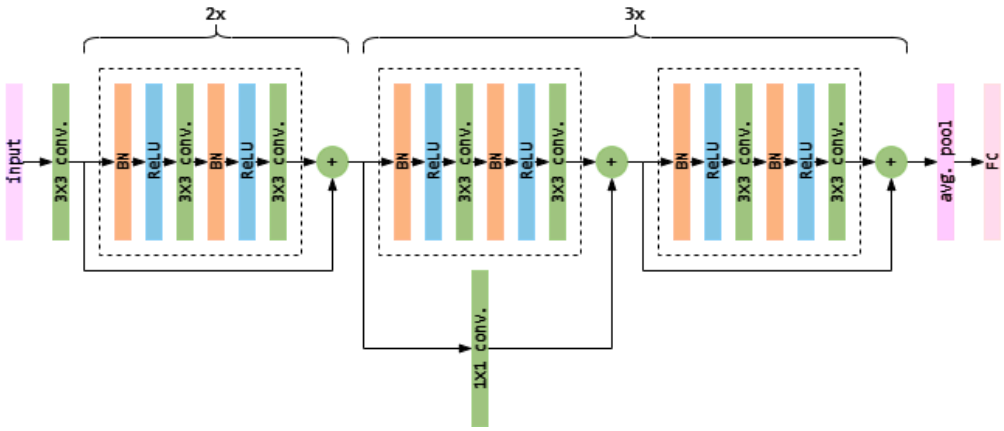


Figure 5: A ResNet-18 model incorporating pre-activation blocks. Some of the blocks will reshape the input and thus have a 1×1 convolutional layer in the shortcut connection.

We can see in Figure 5 that ResNet-18 is built by stacking basic blocks on top of each other. This is true for other[8] basic blocks as well and not just the pre-activation one we use here. Now, if we count the number of convolutional layers (excluding the 1×1 layer), there will be 18 layers in

[7] We mentioned this trick earlier when we introduced residual blocks in (3.2).

[8] See [5] for the original discussion where the layers in Figure 4 are slightly permuted and addition may come sooner, before some layers.

total. Note that we also count the initial convolutional layer and the average pool. Hence, this is where the number "18" comes from in the model's name. As one already might have suspected, there exist other types of ResNets where even more basic blocks, and thus convolutional layers, are deployed. Some examples are ResNet-34, ResNet-50, ResNet-101 and ResNet-152 [4]. Various implementation slightly tweak the initial layers and add a 7×7 convolutional layer followed by batch normalization and maximum pooling, but the standard structure of basic block stacking remains the same. In fact, the 7×7 filter was used in the original implementations [4].

Finally, we describe how the input changes as it passes through the model. The first convolutional layer, right after the input, transforms a standard RGB image $H \times W \times 3$ to $H \times W \times k$. This value of k is fixed and is usually referred to as the network's **width** parameter. Now, the first two blocks, marked by curly brackets and "2×" in Figure 5, will keep this new shape, i.e. $H \times W \times k$, hence no 1×1 convolution is needed. Then, each of the remaining three groups of two blocks will double the number of channels, but also halve the size of the image, i.e. the shape will first change to $H/2 \times W/2 \times 2k$, then $H/4 \times W/4 \times 4k$, and finally $H/8 \times W/8 \times 8k$. The output is obtained by passing this $H/8 \times W/8 \times 8k$ tensor through an average pool and transforming it via a fully connected layer to a $1 \times 1 \times n$ tensor (vector) where n is the number of classes.

One final word regarding the strides in the convolutional layers and model width; the initial convolutinal layer, followed by the first two blocks, stride their 3×3 filters by the value of 1, both horizontally and vertically. The remaining three groups of two blocks stride similarly by the value of 2. We use this in implementations in Section 4, as was done in [8]. Moreover, the standard ResNet-18 uses $k = 64$, and by saying "standard ResNet-18", we mean one with $k = 64$ with the mentioned strides and block structure in Figure 5.

Here is an example. In the standard model, given a $32 \times 32 \times 3$ image input and 10 classes, the shape will change as follows:

$$\underbrace{32 \times 32 \times 3}_{\text{input}} \to \underbrace{32 \times 32 \times 64}_{\text{first 2 blocks}} \to \underbrace{16 \times 16 \times 128}_{\text{next 2 blocks}} \to$$
$$\to \underbrace{8 \times 8 \times 256}_{\text{next 2 blocks}} \to \underbrace{4 \times 4 \times 512}_{\text{last 2 blocks}} \to \underbrace{1 \times 1 \times 10}_{\text{output}}.$$

This is the specification we have in Section 4, and which is used in [8], i.e. $32 \times 32 \times 3$ inputs, 10 classes and $k = 64$. Notice how the model first needs to expand the input to extremely high dimensions, from 3 to 64 channels, and then successively halves the image resolution but keeps doubling the number

of channels. This is not a coincidence and is one characteristic of ResNets. For further discussion, see the original paper [4].

4 Numerical Experiments

In this section we study the behavior of the standard ResNet-18 model's learning curves and output layer geometry over 1k epochs for various label noise levels on CIFAR-10 data. Our attention is set to the understanding of the **double descent** phenomenon that may occur in the test errors due to label noise variation. We will recreate the epoch-wise double descent shown in Figure 10(a) in [8]. Since there will be a good amount of terminology and technical details, all the details are summarized in Section 4.1.

4.1 Experimental Details

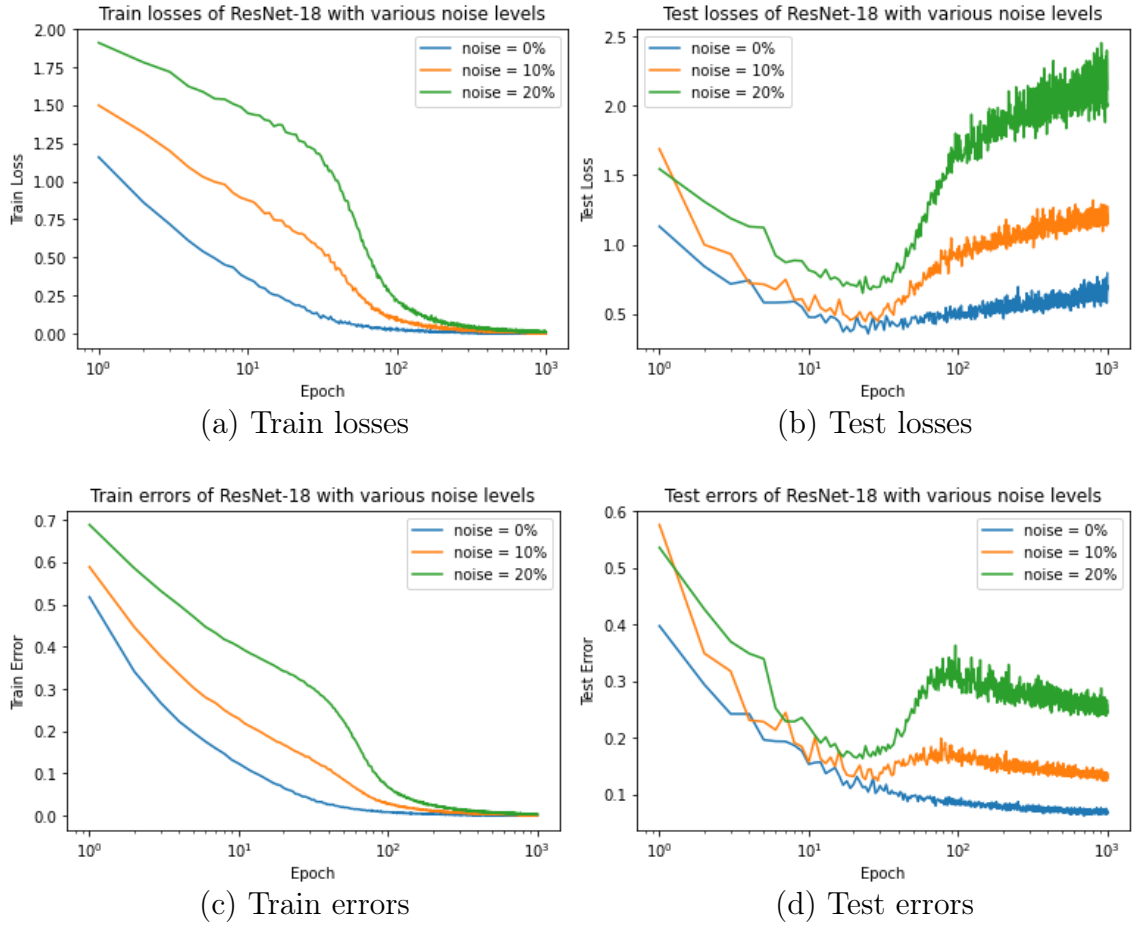
- The code to reproduce the experiments is publicly available[9].
- All experiments are conducted in Python 3.11.7 using **PyTorch** and **CUDA** on a personal computer with RTX 2060 GPU.
- The residual network architecture for all experiments is the standard **ResNet-18** model discussed in Section 3.3. The models initialize with randomized parameters in accordance with PyTorch's default settings.
- The training is conducted on **CIFAR-10** containing 50k/10k train/test $32\times32\times3$ images of 10 different classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck). Data augmentation[10] is used, including `RandomCrop(32, padding=4)` and `RandomHorizontalFlip`.
- The optimizer is **Adam** with learning rate=0.0001 and momentum terms $(\beta_1, \beta_2) = (0.9, 0.999)$.
- Errors are defined as $1 - C/T$, where C is the number of correctly classified test images and T is the total number of test images; whereas losses are defined in terms of **cross-entropy loss**.
- The models are trained for 1k **epochs** in **minibatches** of 100, creating 500/100 train/test minibatches, respectively. The reported losses are the average running train/test losses over the last 100 minibatches in an epoch.

[9] See `https://github.com/stmi98/da6007_degree_project` for more details.
[10] See [3] for more details on the dataset augmentation technique.

- Each figure in Section 4.2.1 shows 3 different runs, one for each noise level 0%, 10% and 20%. The **noise level** corresponds to the amount of train samples having wrong labels. Due to computational limitations, only 1 run is performed for each noise level. The scatter plots in Figures 7-9 show the **softmax** output of three different classes (airplanes, dogs and cats) from the full 10D output of the **test** set.

4.2 Experimental Results

We start by showing the learning curves for each noise level:



(a) Train losses (b) Test losses

(c) Train errors (d) Test errors

Figure 6: Learning curves of ResNet-18 trained on CIFAR-10 using Adam for various noise levels over 1k epochs.

The train/test error plots in Figures 6(c) & 6(d) agree with the plots in Figure 10(a) in [8], hence the experiment has been successfully recreated.

Additionally, in Figures 6(a) & 6(b), we also get to see the corresponding train/test cross-entropy losses. We make the following observations from Figure 6:

Effects of noise on train losses and errors Noisy training labels seem to slow down the training process. By introducing mislabeled images, we increase the risk for a model to misclassify an image in the early stages of training when the model is still learning. The bigger the noise, the longer it takes for a model to learn the classes. In Figures 6(a) & 6(c), all training curves seem to converge to zero, so the overall minimization of the cross-entropy loss is successful, and in the end, the models correctly classify the training set across all noise levels, given enough training time. Qualitatively speaking, both losses in Figure 6(a) and errors in Figure 6(c) seem to agree and assume similar shapes, which is expected. Note that training a classifier with hard 0 and 1 targets can be harmful to the loss function when a label is misclassified. In fact, learning with a softmax classifier may actually never predict the probabilities of exactly 0 and 1, hence label noise is sometimes injected into the training set in a technique called *label smoothing* [8].

Effects of noise on test losses and errors In contrast to training curves where noisy labels gave rise to qualitatively similar losses and errors, here, we get to observe the double descent phenomenon of errors in Figure 6(d) and the standard U-shaped curve of losses in Figure 6(b). The U-shaped behavior of test losses is standard and was explained in Section 2.4. Although we can clearly see that the noise level directly causes the double descent to occur in Figure 6(d), it is still unclear why it happens. In the absence of noise, the double descent is not observed for 1k epochs, and as the noise level rises, it becomes more prominent. In particular, we can observe that the error curve corresponding to noise level 10% manages to recover from the effects of noise in the sense that the second descent is lower than the first, i.e. the accuracy becomes better in later epochs. However, this is not the case with the error curve corresponding to noise level 20%. This particular observation was also made in [8]. Overall, as expected, the model seems to perform better in the absence of noise.

4.2.1 Output Layer Geometry

We will now try to answer the following question: Why do noisy labels make the U-shaped curves drastically increase in Figure 6(b), but also make the test

errors in Figure 6(d) undergo the double descent, as epochs progress? To try to answer this, we will look directly at the model's softmax-activated output layer, for 3 different classes, and study the effects of noise in Figure 6(d). We choose the classes of airplanes, dogs and cats for one curious reason — dogs and cats are easily confused, whereas airplanes are mostly well-separated[11]. Hence we can expect to observe in the early stages of the output layer's scatter plot how dogs and cats get mixed together, but airplanes remain isolated. We will split the **test error** curves in Figure 6(d) in **3 phases** to cover the initial descent, the ascent and the second descent:

P1: The initial descent in epochs 1-20.

P2: The ascent in epochs 40-80.

P3: The second descent in epochs 100-1k.

Let us make some important remarks before we start looking at the output layer. Each input $\boldsymbol{x}$ representing an image of an airplane, dog or cat will get transformed by the model in some way. It is hard to tell what will happen to $\boldsymbol{x}$ internally in the network architecture, but what we can tell is what the final output will strive towards. To be more concrete, recall that the final softmax-activated output of the standard ResNet-18 is a 10D vector $\boldsymbol{q} = (q_1, \ldots, q_{10})$ whose i'th element is $q_i = P(i \mid x)$, where i represents a class. In the ideal scenario, the output vector $\boldsymbol{q}$ will be a one-hot vector, as introduced in Definition 2.4. For our particular case of 3 different classes, this perfect output would simply be one of the vectors $(0,0,1), (0,1,0)$ or $(1,0,0)$. Hence, if we construct a scatter plot for all the test samples, where each sample was perfectly classified by the model, the plot would simply show just 3 points, namely $(0,0,1), (0,1,0)$ and $(1,0,0)$. However, since our model is not perfect, the best we can hope for is that the outputs will accumulate near these points. We will keep this in mind while looking at the upcoming scatter plots of the output layer.

Finally, we start looking at the scatter plots in Figures 7-9 portraying Phases 1-3. The comments are summarized below and the corresponding figures are located at the end of the section.

P1 - The first test error descent Shown in Figure 7, this is manifested as a gradual disentanglement of dogs and cats, while the airplanes remain heavily concentrated around their respective axis. •This confirms the prediction we made earlier when we assumed that the classes of dogs

[11] A concrete study can be made by looking at the so called *confusion matrix* of these classes. One can show that animals that look alike will get mixed more frequently.

and cats get confused. The entanglement is particularly accentuated in the first epoch (first row) in Figure 7. •Next, as the epochs progress, we can notice how each of the models try to push the samples towards their respective axes and the ideal points described earlier. •Regarding the different noise levels, we can also observe the effect of noisy labels. In the first column (a) corresponding to noise=0%, the scatter is much more clear when compared side-to-side with the other two noise levels. This is particularly accentuated on the dog-cat plane in the last row of the figure.

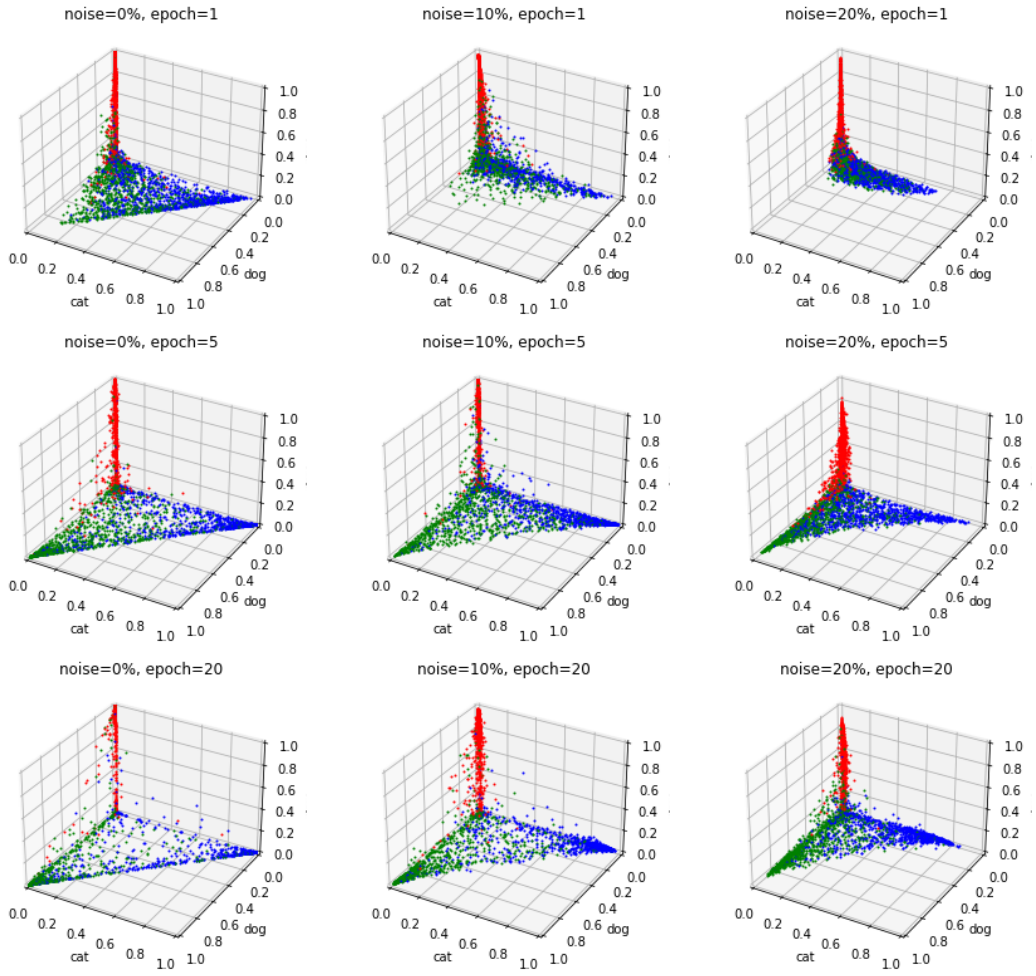
P2 - The test error ascent Shown in Figure 8, this is the most interesting phase since it captures the rise in test errors due to the introduction of noise. •In the noise=0% case, i.e. column (a), we can clearly see that the model tries to push the samples to the ideal scenario even further. When compared to the other two noise levels, there are noticeably fewer samples, indicating that the majority of them are being compressed to the ideal points. •However, as the noise level increases, we can start observing how the misclassified samples do not get fully compressed to the ideal points, but instead start forming some sort of a noisy "dust". •If we compare the third columns, i.e. columns (c), of Figures 7 & 8, it looks like the model first disentangles dogs and cats, then starts pushing the samples towards their respective axes, and as we enter Phase 2, the samples get "blown into the air" from the axes like "specks of dust". The larger the noise is, the "dustier" the scatter plot becomes. •Translating this to the cross-entropy loss in Figure 6(b), it becomes more evident why the test losses increase as the noise levels increase. It is because of this "dust" which increases the risk of a sample to get incorrectly classified. Recall that each sample gets mapped to a class corresponding to the largest probability in the 10D softmax output vector. As the number of incorrectly classified samples increases, the running cross-entropy loss will have more terms that increase. To see this, recall that the cross-entropy loss will compute $E_i = -\ln P(\text{true class} \mid i)$, where i is an input image. If the image gets pushed away from its true label, this $P(\text{true class} \mid i)$ will jump from a value close to 1, to a value close to 0, consequently increasing[12] the contribution of E_i to a loss of the form (2.11).

P3 - The second test error descent Shown in Figure 9, this is the final phase where the epochs are pushed asymptotically. •The noise=0% case continues with its compression from Phases 1 & 2 as it "brushes

[12]We mentioned this risk when we introduced the cross-entropy loss in Section 2.3.1.

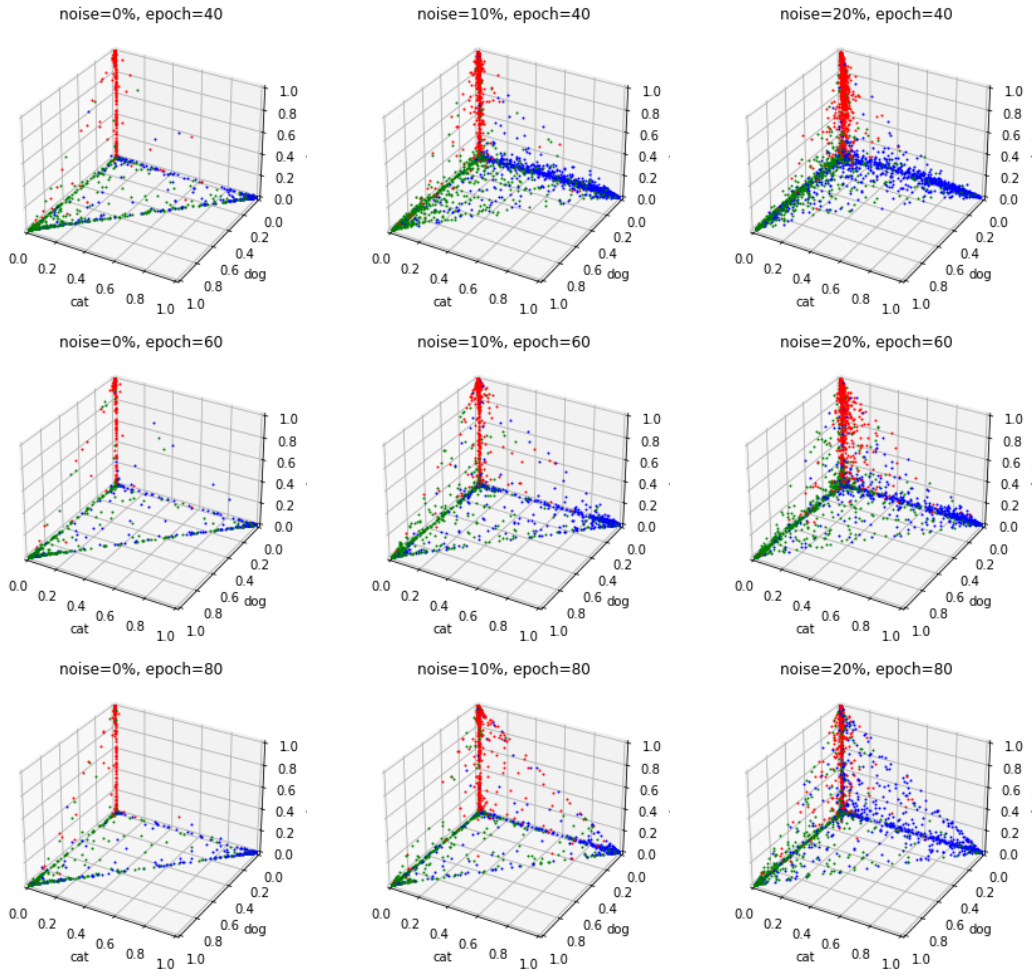
away" even the smallest occurrences of "dust specks". The associated error plot in Figure 6(d) slowly continues with its trend of convergence to zero. The lack of additional mislabeled samples does not contribute to the loss in Figure 6(b), hence the less prominent the overfitting regime of the curve is. •Finally, the other two columns of noise levels 10% & 20%, i.e. columns (b) & (c), begin to settle down as well. We can observe a clear decrease of noisy samples and the model finally starts to compress the samples. We can think of this whole phase as a **recovery** phase from the effects of noise. •It seems like the asymptotic learning will slowly, over long enough time, "brush up" the misclassified samples. It is important to mention that even a couple of misclassified samples can contribute a lot to the running cross-entropy loss. If the probability in the negative logarithm is small enough, the overall loss will keep drastically increasing. However, one such sample would not affect the error curve that drastically, hence one curve may ascend but the other one descend, which is the key insight.

We conclude this section with one interesting physical analogy which seems to connect the results in Figures 7-9 with the test errors in Figure 6. Assume that a noise-free model's **test error** will monotonically approach its theoretical minimum in finite epochs. Moreover, assume that this decrease of error is bigger for early epochs, but gets smaller as epochs progress. Call this underlying function the *error outflow*. We can then think of the introduction of noise to the learning process as the introduction of some corresponding *error inflow*. This *error inflow* may also be more prominent for the early epochs, but get smaller as epochs progress. The results from this section suggest that this hypothetical outflow function will, over long time, start dominating the inflow function, reducing the overall error. Physically, we can think about the test error as a **water level** in some container, and the water quantity as the number of misclassified samples. Then the outflow function can be thought of as a leakage from this container and the inflow function as some water inflow. The whole process could then be modeled by a simple ODE, and by proper parameter tuning represent the double descent in terms of the water level in this container. It would be interesting to know if such inflow/outflow functions exist, and if so, if they can be inferred for a fixed model with all its hyperparameters and stochasticity. Consequently, the whole double descent phenomenon could be modeled by an ODE.



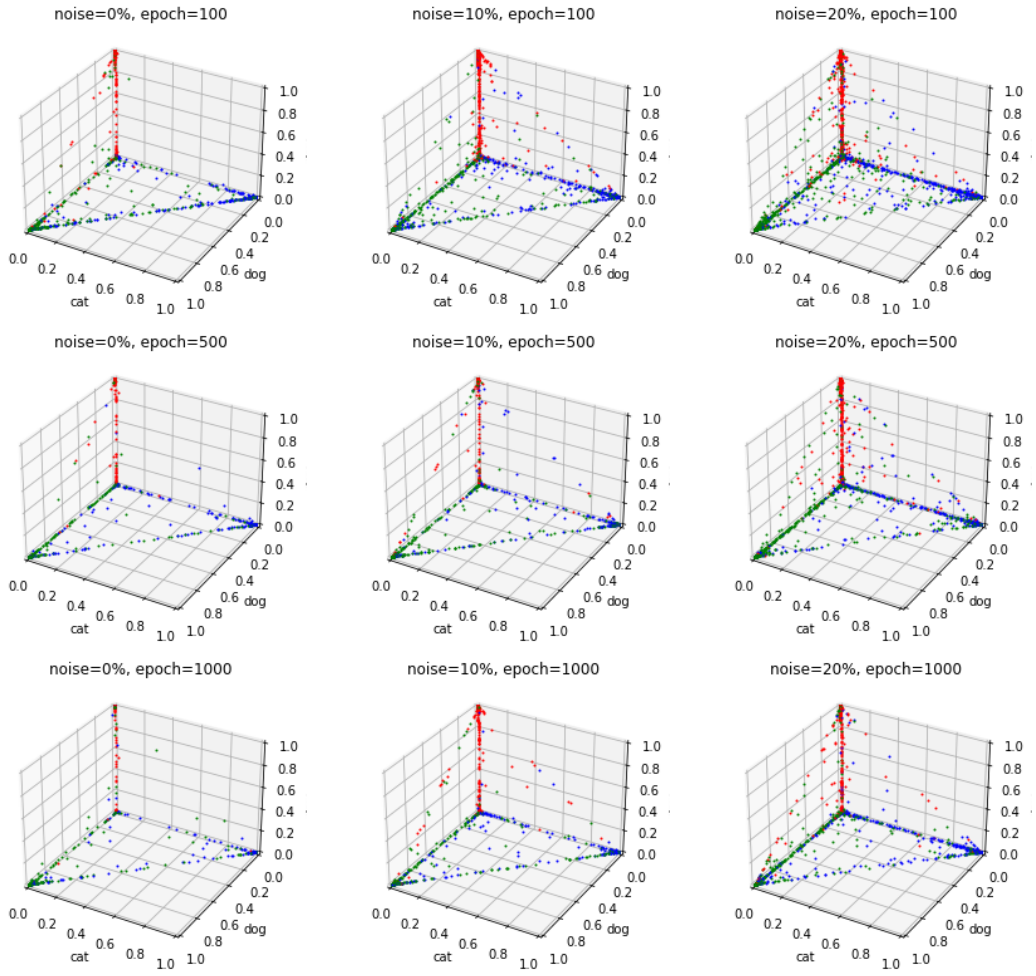
(a) noise=0% (b) noise=10% (c) noise=20%

Figure 7: Side-to-side output geometry comparison of ResNet-18 trained on CIFAR-10 using Adam for 3 different classes (red=airplane, green=dog, blue=cat). Each row/column represents various epochs/noises in **Phase 1**.



(a) noise=0% (b) noise=10% (c) noise=20%

Figure 8: Side-to-side output geometry comparison of ResNet-18 trained on CIFAR-10 using Adam for 3 different classes (red=airplane, green=dog, blue=cat). Each row/column represents various epochs/noises in **Phase 2**.



(a) noise=0% (b) noise=10% (c) noise=20%

Figure 9: Side-to-side output geometry comparison of ResNet-18 trained on CIFAR-10 using Adam for 3 different classes (red=airplane, green=dog, blue=cat). Each row/column represents various epochs/noises in **Phase 3**.

5 Discussion

We confirmed that the epoch-wise double descent phenomenon can indeed occur for residual networks. We recreated Figure 10(a) from [8] and expanded on it by also looking at the corresponding cross-entropy losses. We performed additional experiments by directly looking at the output layer geometry of the standard ResNet-18 model, and tried to make a connection between the learning curves of cross-entropy test losses and the associated test errors. The results that we have obtained suggest that learning with noisy labels directly impairs a model's accuracy. However, if the noise levels are not too high, the effects of noise will be slowly diminished over time. Most importantly, this recovery process cannot be detected from the cross-entropy loss curves since the introduction of noisy labels increases the risk of misclassification, and even a few misclassified samples can drastically increase the per-example cross-entropy loss. Consequently, even when a model starts recovering from the effects of noise, i.e. starts reducing the number of incorrectly classified samples, the cross-entropy loss will have been increasing, both from the overfitting regime and the extreme contributions from misclassified samples.

It is important to mention the limitations of the performed experiments. Since there was a limited computational power available, only single runs were performed. Furthermore, we did not experiment with different hyperparameters in the optimizer, and different data sets. Heuristically, we also looked at three particular isolated classes in the model's output layer. A full scale experimentation should consider tweaking the hyperparameters, using different data sets and more noise levels, but also include multiple runs and look at more classes in the output layer.

Towards the end of Section 4.2.1, we curiously introduced one physical analogy connecting test errors with water levels in a container by using some sort of *inflow/outflow* functions in an ODE. It would be interesting to see how far this idea can be explored.

References

[1] K. Alex, *Learning multiple layers of features from tiny images*, https://www. cs. toronto. edu/kriz/learning-features-2009-TR. pdf, (2009).

[2] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of control, signals and systems, 2 (1989), pp. 303–314.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016. http://www.deeplearningbook.org.

[4] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015.

[5] ——, *Identity mappings in deep residual networks*, 2016.

[6] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015.

[7] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[8] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever, *Deep double descent: Where bigger models and more data hurt*, CoRR, abs/1912.02292 (2019).

[9] J. Nocedal and S. J. Wright, *Numerical optimization*, Springer, 1999.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, nature, 323 (1986), pp. 533–536.

[11] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into deep learning*, arXiv preprint arXiv:2106.11342, (2021).

Datalogi
Maj 2024

www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm