Mathematics of Neural Networks

Bart M.N. Smets

November 12, 2022

"You insist that there is something a machine cannot do. If you tell me precisely what it is a machine cannot do, then I can always make a machine which will do just that."

– John von Neumann

"As a technical discussion grows longer, the probability of someone suggesting deep learning as a solution approaches 1*."*

– comment on Youtube anno 2020

Thanks to my colleagues Gijs Bellaard, Remco Duits, Jim Portegies and Alessandro Di Bucchianico for valuable input and feedback during the writing of these lecture notes.

Contents

Introduction

The last decade has seen great experimental progress being made in machine learning, spearheaded by deep learning methods that make use of so-called **deep neural networks**. Many challenging, high-dimensional tasks that were previously beyond reach have become feasible with remarkably simple (in the algorithmic sense) techniques coupled with modern computational resources. Particularly in the fields of computer vision and natural language processing, deep learning is currently the go-to tool.

Machine learning is usually positioned under the umbrella of **artificial intelligence**. While artificial intelligence is a fairly nebulous and broad term, the term machine learning refers specifically to algorithms that can improve their performance at a given task when presented with more data about the problem. Machine learning algorithms are used in a wide variety of applications such as speech recognition and computer vision, where it is difficult or impossible to develop conventional algorithms to perform the required task. These learning algorithms generally start from very general model that is then **trained** based in sample data to learn how to perform a specific task. When the underlying model being used is a (deep) neural network then we speak about deep learning.

The idea of using computational models that are inspired by the workings of biological neurons dates as far back as McCulloch and Pitts (1943). Over the following decades more of the ingredients that we think of a standard today were added: training the network on data was tried by Ivakhnenko and Lapa (1966), Fukushima (1987) originated the ancestor of our current convolutional neural networks and LeCun, Boser, et al. (1989) introduced backpropagation as a training mechanism for neural networks.

However, non of these efforts led to a breakthrough in the use of neural networks in practice and such models were mainly regarded as a academic curiosity during this time period. This state of affairs only changed at the start of the new millennium with the appearance of programmable **GPUs** (Graphics Processing Unit), while initially designed with rendering 3D graphics in mind these devices could be leveraged for other purposes, such as by Oh and Jung (2004) for neural networks. This eventually led to such breakthroughs as on the ImageNet (Deng et al. 2009) image classification challenge by Krizhevsky, Sutskever, and G. E. Hinton (2012), where neural networks managed to dominate other, more traditional, techniques. These events can be thought of as the start of the modern deep learning era.

Despite more than a decade of impressive experimental results, theoretical understanding of why deep learning works as well as it does is lacking. This presents an opportunity both for understanding and improvement, particularly for mathematicians.

This course presents an introduction to neural networks from the point of view of a mathematician. We cover the basic vocabulary and functioning of neural networks in Chapter 1. In Chapter 2 we look at deep neural networks and the associates techniques that allow them to work. Chapter 3 covers a novel application of geometry to neural networks. We will discuss

how the theory of Lie groups and homogeneous spaces can be leveraged to endow neural networks with certain structural symmetries, i.e. make them equivariant under certain geometric transformations.

Chapter 1

The Basics

1.1 Supervised Learning

While there are different kinds of machine learning, we will be focusing on **supervised learning**. Supervised learning is a machine learning paradigm where the available data consists of a pairing of inputs with know, correct, outputs. What is unknown is exactly how the mapping between those inputs and outputs works. The goal is to infer, from the available data, the general structure of the mapping in the hope that this will generalize to unseen situations. Formally we can state the problem as follows.

The supervised learning problem Given an unknown function $f : X \to Y$ between spaces X and Y, find a good approximation of f using only a dataset of N samples:

$$\mathcal{D} = \left\{ (x_i, y_i) \right\}_{i=1}^{N} \quad \text{with} \quad y_i = f(x_i) \quad \text{for all } i = 1 \dots N.$$

The space X is also called the **feature space** and its elements are referred to as feature vectors. The space Y is also called the **label space** and its elements are referred to as labels.

Example 1.1. Let X be the space of all images of cats and dogs and f the classifier that maps to $Y = \{\text{"dog"}, \text{"cat"}\}$. Which we can express numerically as $X = [0,1]^{3\times H\times W}$ (i.e. an image of height H and width W with 3 color channels) and $Y = [0,1]^2$ (one probability for each of the two classes).

The general approach to solving this problem consists of three main steps.

1. Choose a model $F : X \times W \to Y$ parametrized by a parameter space W (W for weights, since that is how parameters in NNs are often referred to).
2. We need to quantify the quality of the model's output, so we need to choose a **loss function** $\ell : Y \times Y \to \mathbb{R}$, where $\ell(y_1, y_2)$ indicates "how different" y_1 and y_2 are.
3. Based on the dataset and the loss function choose $w \in W$ so that $F(\,\cdot\,; w)$ is the "best" possible approximation of the target function f.

This high-level approach leaves some open questions however.

- How to choose the model?
- How to choose the loss function?
- How to optimize?

None of these questions have canonical answers and are largely handled by trial-and-error and heuristics simply because we have to resolve them before we can make any progress. Regardless, these choices will have a large impact on the final result even though they are in no way supported by the available data or some form of first principle. The collective set of assumptions we make to be able to proceed with the problem is called **inductive bias** in the machine learning field.

For the purpose of this course we will of course pick neural networks as our models of choice and we will talk about those later. The second thing we have to do is chose a loss function. A **loss function** is a function $\ell : Y \times Y \to \mathbb{R}$, it works much like a metric, but it is less restrictive.

- Usually but not always $Y \times Y \to \mathbb{R}^+$
- Since we want to minimize loss, the minimum should exist so that $\min_{y \in Y} \ell(y_0, y)$ is well posed.
- Differentiable (a.e. at least) would be necessary since we want to do gradient descent.
- Metric properties like identity of indiscernibles, symmetry and triangle inequality need not hold.

Having chosen a model and a loss function we move on to optimization, or: what is the "best" choice of $w \in W$? The straightforward (but not necessarily preferred) answer: minimize the loss on the dataset, i.e. find:

$$w^* = \arg\min_{w \in W} \sum_{i=1}^{N} \ell\left(F(x_i; w), y_i\right).$$

Example 1.2 (Linear least squares)**.**

$$F(x; w) = \sum_{j=1}^{n} w_j \varphi_j(x)$$

and $\ell(y_1, y_2) = |y_1 - y_2|^2$ for some basis functions $\{\varphi_j\}_j$. Then we get the familiar **linear least square** setting.

What is the "best" $w \in W$ is not necessarily the one that minimizes the loss on the dataset, **overfitting** is often an issue in any optimization problem. Which brings us to **regularization**.

We will discuss regularization techniques for NNs more in the future. But for now we will mention that **parameter regularization** is a common technique from regression that is often used in NNs. This type of regularization is characterized by the addition of a penalty term to the data loss that discourages parameter values for straying into undesirable areas (such as becoming too large). The modified optimization problem becomes:

$$w^* = \arg\min_{w \in W} \sum_{i=1}^{N} \ell\left(F(x_i; w), y_i\right) + \lambda C(w)$$

with $\lambda > 0$ and $C : W \to \mathbb{R}^+$ to penalize complexity in some fashion.

Example 1.3 (Tikhonov regularization)**.** $W = \mathbb{R}^n$ and $C(w) = \|w\|^2$. In the context of neural networks this type of parameter regularization is also called **weight decay**.

Regression & classification Supervised learning should sound familiar by now. Indeed if $X = \mathbb{R}^n$ and $Y = \mathbb{R}^m$ then it amounts to **regression**.

Regression is a large part of supervised learning but it is more generally formulated so that is encompasses other tasks such as **classification**.

Classification is the ML designation for (multiclass) logistic regression where $X = \mathbb{R}^n$ and we try to assign each $x \in X$ to one of m classes. The numeric output for this type of problem is normally a discrete probability distribution over m classes:

$$Y = \left\{(y_1, \ldots, y_m) \in [0,1]^m \;\middle|\; \textstyle\sum_{i=1}^m y_i = 1\right\}.$$

Example 1.1 is of this type.

Remark 1.4 (Statistical learning theory viewpoint). In SLT the assumption is made that the data samples (x_i, y_i) are drawn i.i.d. from some probability distribution μ on $X \times Y$. *Think about why this is a fairly big leap.* What we are then interested in is minimizing the **population risk**:

$$R(w) := \mathbb{E}_{(x,y)\sim\mu}\left[\ell\left(F(x;w), y\right)\right] := \int_{X\times Y} \ell\left(F(x;w), y\right) d\mu(x,y).$$

The goal would be finding the parameter set that minimizes this population risk:

$$w^* = \arg\min_{w\in W} R(w).$$

But in reality we do not know μ and so we cannot even calculate the population risk, let alone minimize it. So we do the next best thing: minimize the **empirical risk**:

$$\hat{R}(w) := \frac{1}{N}\sum_{i=1}^{N} \ell\left(F(x_i;w), y_i\right).$$

The parameter set that minimize the empirical risk is called the **empirical minimizer**:

$$\hat{w} = \arg\min_{w\in W} \hat{R}(w),$$

which is the same thing as minimizing the loss on the dataset in the supervised learning setting.
When we add a regularization term as before it is called **structural risk minimization**.

$$\hat{w} = \arg\min_{w\in W} \hat{R}(w) + \lambda C(w).$$

SLT is then concerned with studying things such as bounds on $\hat{R}(\hat{w}) - R(w^*)$. For more see 2MMS80 Statistical Learning Theory.

1.2 Artificial Neurons & Activation Functions

As the name suggests, artificial neural networks are inspired by biology. Just like their biological counterparts their constituent parts are artificial neurons. The basic structure of a biological neuron is illustrated in Figure 1.1. Each neuron can send and receive signals to and from other neurons so that together they form a network.

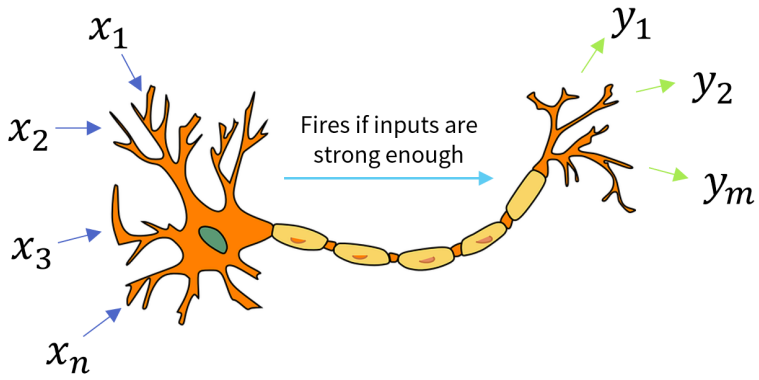


Figure 1.1: A simplified biological neuron. The dendrites on the left receive electric signals from other neurons, once a certain threshold is reached the neuron will fire a signal along its axon and through its synapses on the right relay a signal to other neurons.

In similar fashion artificial neural networks consist of artificial neurons. Each artificial neurons takes some inputs (usually in the form of real numbers) and produces one or more outputs (again, usually real numbers) that it passes to other neurons. The most common model neuron is given by an affine transform followed by a non-linear function. Let $\boldsymbol{x} \in \mathbb{R}^n$ be the input signal and $\boldsymbol{y} \in \mathbb{R}^m$ be the output signal then we calculate

$$\boldsymbol{y} = \sigma(A\boldsymbol{x} + \boldsymbol{b}), \tag{1.1}$$

where $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{b} \in \mathbb{R}^m$ and σ is a choice of **activation function**. The components of the matrix A are referred to as the **linear weights**, the vector $\boldsymbol{b}$ is called the **bias**. The intermediate value $A\boldsymbol{x} + \boldsymbol{b}$ is sometimes referred to as the **activation**. The activation function σ is also sometimes called the transfer function or the non-linearity.

Inputs and outputs need not span the real numbers. Depending on the application we could encounter:

- $\{0, 1\}$: binary,
- $\mathbb{R}^+$: non-negative reals,
- $[0, 1]$: intervals (probabilities for example),
- $\mathbb{C}$: complex,
- etc.

A historically significant choice of activation function is the **Heaviside** function, given by

$$H(x) := \mathbb{1}_{x\geq 0}(x) := \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{else.} \end{cases}$$

A neuron of the type (1.1) that uses the Heaviside function as its activation function is called a **perceptron**. Let us see what we can do with it. Let $\boldsymbol{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ then a neuron with input

$\boldsymbol{x} \in \mathbb{R}^n$ and output $y \in \mathbb{R}$ would look like

$$\mathcal{N}(\boldsymbol{x}) = H(\boldsymbol{w}^T\boldsymbol{x} + b) = \begin{cases} 1 & \text{if } \boldsymbol{w}^T\boldsymbol{x} \geq -b, \\ 0 & \text{if } \boldsymbol{w}^T\boldsymbol{x} < -b. \end{cases}$$

Which is nothing but a linear binary classifier on $\mathbb{R}^n$ since $\boldsymbol{w}^T\boldsymbol{x} = -b$ is a hyperplane in $\mathbb{R}^n$. This hyperplane divides the space into two and assign the value 0 to one half and 1 to the other.

Example 1.5 (Boolean gate)**.** We can (amongst other things) use a perceptron to model some elementary Boolean logic. Let $x_1, x_2 \in \{0, 1\}$ and let $\text{AND}(x_1, x_2) := H(x_1 + x_2 - 1.5)$ then the neuron behaves as follows.

x_1	x_2	$\text{AND}(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

The Heaviside function is an example of a **scalar** or **pointwise** activation function. Often when a use a scalar function as activation function we abuse notation to let it accepts vector (and matrices) as follows. Let $\sigma : \mathbb{R} \to \mathbb{R}$ then we allow

$$\sigma\left(\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}\right) \equiv \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_n) \end{bmatrix}.$$

We list some commonly used scalar activation functions which are also illustrated in Figure 1.2.

- **Rectified Linear Unit** (ReLU): arguably the most used activation function in modern neural networks, it is calculated as

  $$\sigma(\lambda) = \text{ReLU}(\lambda) := \max\{0, \lambda\}.$$

- **Sigmoid** (also known as logistic sigmoid or soft-step):

  $$\sigma(\lambda) := \frac{1}{1 + e^{-\lambda}}.$$

  The sigmoid was commonly used as activation function in early neural networks, which is the reason that activations functions in general are still often labeled with a σ.

- **Hyperbolic tangent**: very similar to the sigmoid, it is given by

  $$\tanh(\lambda) := \frac{e^{\lambda} - e^{-\lambda}}{e^{\lambda} + e^{-\lambda}}.$$

- **Swish**: a more recent choice of activation function that can be thought of as a smooth variant of the ReLU. It is given by the multiplication of the input itself with the sigmoid function:

  $$\text{swish}_\beta(\lambda) := \lambda\, \sigma(\beta\lambda) := \frac{\lambda}{1 + e^{-\beta\lambda}},$$

  where $\beta > 0$. The β parameter is usually chosen to be 1 but could be treated as a trainable parameter if desired. In case of $\beta = 1$, this function is also called the **sigmoid-weighted linear unit** or SiLU.

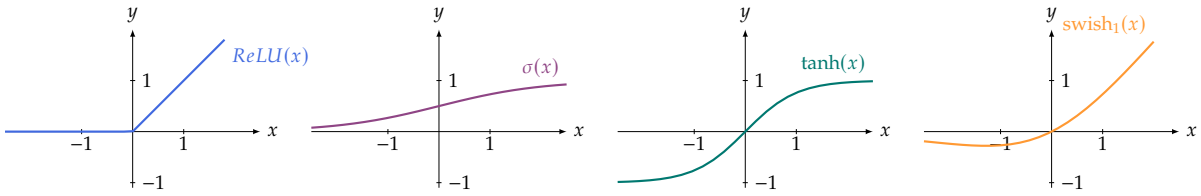


Figure 1.2: Some common scalar activation functions. From left to right: the rectified linear unit, the logistic sigmoid, the hyperbolic tangent and the swish function with $\beta = 1$.

Activation functions need not be scalar, we list some common multivariate functions.

- **Softmax**, also known as the normalized exponential function: $\text{softmax} : \mathbb{R}^n \to [0,1]^n$ is given by

$$\text{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}\right) := \frac{1}{\sum_{i=1}^{n} e^{x_i}} \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}.$$

  Softmax has the useful property that its output is a discrete probability distribution, i.e. each value is a non-negative real in the range $[0,1]$, and all the values in its output add up to exactly 1.

- **Maxpool**: here each output is the maximum of a certain subset of the inputs: the function $\text{maxpool} : \mathbb{R}^n \to \mathbb{R}^m$ is given by

$$\text{maxpool}\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}\right) := \begin{bmatrix} \max_{j \in I_1} x_j \\ \max_{j \in I_2} x_j \\ \vdots \\ \max_{j \in I_m} x_j \end{bmatrix}.$$

  Where for each $i \in \{1, \ldots, m\}$ we have a $I_i \subset \{1, \ldots, n\}$ that specifies over which inputs to take the maximum for each output. Maxpooling can easily be generalised by replacing the max operation with min, the average, the mean, etc.

- **Normalization**, sometimes it is desirable to re-center and re-scale a signal:

$$\text{normalize}\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}\right) := \begin{bmatrix} \frac{x_1 - \mu}{\sigma} \\ \frac{x_2 - \mu}{\sigma} \\ \vdots \\ \frac{x_n - \mu}{\sigma} \end{bmatrix},$$

  where $\mu = \mathbb{E}[\boldsymbol{x}]$ and $\sigma^2 = \text{Var}(\boldsymbol{x})$. There are many variants on normalization where the difference is how μ and σ are computed: over time, over subsets of the incoming signals, etc.

All the previous examples of activation functions are deterministic, but **stochastic** activation functions are also used.

- **Dropout** is a stochastic function that is often used during the training process but is removed once the training is finished. It works by randomly setting individual values of a signal to zero with probability p:

$$\Big(\text{dropout}_p(\boldsymbol{x})\Big)_i := \begin{cases} 0 & \text{with probability } p, \\ x_i & \text{with probability } 1-p. \end{cases}$$

- **Heatbath** is a scalar function that outputs 1 or -1 with a probability that depends on the input:

$$\text{heatbath}(\lambda) := \begin{cases} 1 & \text{with probability } \frac{1}{1+e^{-\lambda}} \\ -1 & \text{otherwise.} \end{cases}$$

All the activation functions we seen are essentially fixed functions, swish and dropout have a parameter but it is usually fixed to some chosen value. That means that the trainable parameters of a neural network are usually the linear weights and biases. There is however no a-priori reason why that needs to be the case, in fact we will see a class of non-linear operators with trainable parameters at the end of Chapter 3. Regardless, having parameters in the non-linear part of a network is somewhat rare in practice at the time of this writing.

1.3 Shallow Networks

While we are interested in deep networks we start out with a look at shallow networks. Because of their simplicity we can still approach them constructively and gain some intuition about neural networks in general along the way. You can also think about the study of shallow networks as the study of single layers of deep networks.

Let us consider a shallow ReLU network with scalar in- and output. Let $\boldsymbol{w} = (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) \in W = (\mathbb{R}^N)^3$ be our set of parameters for some $N \in \mathbb{N}$, then we define our model $F : \mathbb{R} \times W \to \mathbb{R}$ as

$$F(x; \boldsymbol{w}) := \sum_{i=1}^{N} c_i \, \sigma(a_i x + b_i), \tag{1.2}$$

where σ is the ReLU. Diagrammatically this network is represented in Figure 1.3.

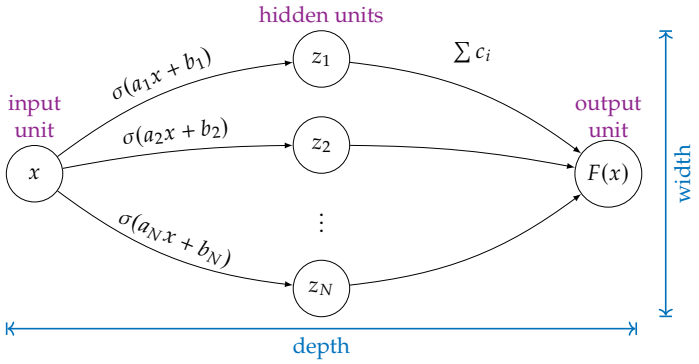


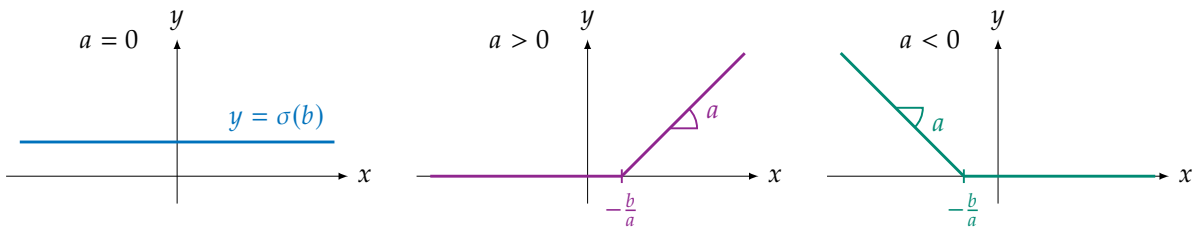
Figure 1.3: Diagrammatic representation of a shallow $\mathbb{R} \to \mathbb{R}$ neural network per (1.2). In deep learning literature the input and output of a network are often referred to as the input unit respectively the output unit. The intermediate values are often called the hidden units. What is commonly referred to as the **width** and **depth** of the network is also indicated.

We will restrict ourselves to $x \in [0, 1]$ for the time being. In practice this would not be much of a restriction since real world data is compactly supported.

Let us explore what types of functions our network can express: the output is a linear combination of functions of the type

$$x \mapsto \sigma(ax + b).$$

Which, depending on the value of a, gives us one of the following types of functions.



All three of these classes of functions are piecewise linear functions (or piecewise affine functions really), so any linear combination of these functions would again be a piecewise linear function. Hence, our model (1.2) is really just a particular parameterization for a piecewise linear function on $[0, 1]$.

Can we then represent any piecewise linear function on $[0, 1]$ by a shallow ReLU network? Let $f : [0, 1] \to \mathbb{R}$ be piecewise linear function with N pieces. We will denote the inflection points with

$$0 = \beta_1 < \beta_2 < \ldots < \beta_{N+1} = 1$$

and the slopes (i.e. the constant derivatives of each piece) with $\alpha_1, \ldots, \alpha_N$. An example of this setup is illustrated in Figure 1.4.

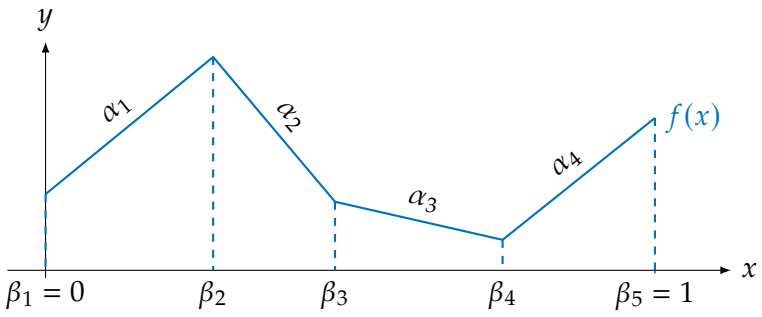


Figure 1.4: Example of a piecewise linear function on $[0, 1]$ with 4 pieces. The location of the inflection points are labeled with β's and the slope of each piece is denoted with an α.

Now choose a network of the type in (1.2) with $N + 1$ neurons:

$$F(x) := \sum_{i=1}^{N+1} c_i \, \sigma(a_i x + b_i).$$

Next we choose the parameters $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ of the network as:

$$\begin{array}{lll} a_{N+1} = 0, & b_{N+1} = 1, & c_1 = \alpha_1, \\ a_1, \ldots, a_N = 1, & b_i = -\beta_i \text{ (for } i = 1 \ldots N), & c_i = \alpha_i - \alpha_{i-1} \text{ (for } i = 2 \ldots N). \end{array}$$

This turns our model into

$$F(x) = f(0) + \alpha_1 x + \sum_{i=2}^{N} (\alpha_i - \alpha_{i-1}) \, \sigma(x - \beta_i). \tag{1.3}$$

When we examine the third term of (1.3) we see that the ReLU's will vanish each term until x reaches the appropriate threshold, i.e. $\sigma(x - \beta_i) = 0$ until $x > \beta_i$. Hence for $x \in [\beta_1, \beta_2]$ we get the function $x \mapsto f(0) + \alpha_1 x$ which exactly matches the function f in that interval. As we proceed to the $[\beta_2, \beta_3]$ interval the first term of the right hand sum becomes non-zero and the model becomes

$$\begin{aligned} x \mapsto & f(0) + \alpha_1 x + (\alpha_2 - \alpha_1)(x - \beta_2) \\ & = f(0) + \alpha_1 x - \alpha_1 x + \alpha_1 \beta_2 + \alpha_2 (x - \beta_2) \\ & = f(\beta_2) + \alpha_2 (x - \beta_2), \end{aligned}$$

which is exactly f in the interval $[\beta_2, \beta_3]$. We can keep advancing along the x-axis like this and every time we pass an inflection point a new term will activate and bend the line towards a new heading. The model can effectively be rewritten as

$$F(x) = \begin{cases} f(0) + \alpha_1 x & \text{if } x \in [\beta_1, \beta_2], \\ f(\beta_2) + \alpha_2 (x - \beta_2) & \text{if } x \in [\beta_2, \beta_3], \\ \quad \vdots & \\ f(\beta_{N-1}) + \alpha_N (x - \beta_{N-1}) & \text{if } x \in [\beta_{N-1}, \beta_N], \end{cases}$$

which matches f exactly. Hence, we may conclude that on compact intervals the shallow scalar ReLU neural networks are exactly the space of piecewise linear functions.

Piecewise linear function are a simple class of function but can be used to approximate many other classes of function to an arbitrary degree, as the following lemma shows.

Lemma 1.6. Let $f \in C([0,1], \mathbb{R})$ then for all $\varepsilon > 0$ there exists a piecewise linear function F so that

$$\sup_{x \in [0,1]} |f(x) - F(x)| < \varepsilon.$$

Proof. Let $\varepsilon > 0$ be arbitrary. Since f is a continuous function on a compact domain it is also uniformly continuous on said domain, i.e.

$$\exists \delta > 0 \, \forall x_1, x_2 \in [0,1] : |x_1 - x_2| < \delta \Rightarrow |f(x_1) - f(x_2)| < \frac{\varepsilon}{2}.$$

Now choose $N > \frac{1}{\delta}$ and partition $[0,1]$ as $x_i = \frac{i}{N}$ for $i = 0 \dots N$. Define the piecewise linear function F as

$$F(x) = \sum_{i=1}^{N} \mathbb{1}_{[x_{i-1}, x_i)}(x) \left(f(x_{i-1}) + \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} (x - x_{i-1}) \right).$$

Note that for all $i - 0 \dots N$ that $F(x_i) = f(x_i)$. Now let $x \in [0,1]$ be arbitrary, x will always fall inside some interval $[x_{i-1}, x_i)$ where we have

$$\begin{aligned} |f(x) - F(x)| &\leq |f(x) - f(x_i)| + |f(x_i) - F(x)| \\ &< \frac{\varepsilon}{2} + |F(x_i) - F(x)| \\ &< \frac{\varepsilon}{2} + |F(x_i) - F(x_{i-1})| \\ &= \frac{\varepsilon}{2} + |f(x_i) - f(x_{i-1})| \\ &< \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon \end{aligned}$$

□

Since shallow scalar ReLU networks represent the piecewise linear function on $[0, 1]$ and those piecewise linear functions are dense in $C([0, 1])$ we get the following.

Corollary 1.7. Shallow scalar ReLU neural networks of arbitrary width (1.2) are universal approximators of $C([0, 1])$ in the supremum norm.

This corollary is our first **universality** result. In the context of deep learning, universality is the study of what classes of functions can be approximated arbitrarily well by particular neural networks architectures. Notice that Corollary 1.7 has 4 ingredients:

1. the type of neural network (shallow scalar ReLU network),
2. the growth direction of the network (width),
3. the space of function to approximate ($C([0, 1])$),
4. and how the approximation is measured (with the supremum norm).

Generalizing this universal approximation result to $\mathbb{R}^n$ is not possible with just one layer, one of the reasons deep networks are necessary. In general constructive proofs such as for Lemma 1.6 are not possible/available and we will have to contend ourselves with existence results.

Universality is theoretically interesting since it tells us that neural networks can in principle closely approximate most reasonable types of functions (continuous, L^p, etc.), thus explaining to some degree why they are powerful. But universality does not consider many other important facets of neural networks in practice:

- economy of representation: how does the number of parameters scale with the desired accuracy,
- economy of finding a good approximation: just because a good approximation exist does not mean it is easy to find.

We will not discuss universality further for now. Just remember that, under some mild assumptions, neural networks are universal approximators.

1.4 Stochastic Gradient Descent

Recall the elements of supervised learning:

- a dataset $\mathcal{D} = \{(x_i, y_i) \in X \times Y\}_{i=1}^N$ for some input space X and output space Y,
- a model $F : X \times W \to Y$ for some parameter space W,
- and a loss function $\ell : Y \times Y \to \mathbb{R}$.

We can then look at the total loss over the dataset for a given choice of parameters:

$$\ell_{\text{total}}(w) := \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \ell\left(F(x; w), y\right). \tag{1.4}$$

Note how the loss is a function of the parameters w only, it could also include some additional regularization terms (as in Section 1.1) but we will leave those details aside for now.

Knowing little about F we cannot find a minimum directly, but assuming everything is differentiable we can use gradient descent:

$$w_t = w_{t-1} - \eta \nabla_w \ell_{\text{total}}(w_{t-1}),$$

where $\eta > 0$ is called the **learning rate** and where we chose some $w_0 \in W$ as our starting point.

Sadly, doing this calculation is not practical in the real world due to:

- the datasets being huge,
- the models having an enormous amount of parameters.

Just calculating ℓ_{total} is a major undertaking, calculating $\nabla_w \ell_{\text{total}}$ is entirely out of the question.

Instead, we take a divide and conquer approach. We randomly divide the dataset into (roughly) equal **batches** and tackle the problem batch by batch. Denote a batch by an index set $I \subset \{1 \dots N\}$, then we consider the loss over that single batch:

$$\ell_I(w) := \frac{1}{|I|} \sum_{i \in I} \ell\left(F(x_i; w), y_i\right). \tag{1.5}$$

Say we divide the dataset into batches $I_1, I_2, \dots, I_B$, we can then split our gradient descent step into B smaller steps:

$$w_t = w_{t-1} - \eta \nabla_w \ell_{I_t}(w_{t-1}) \tag{1.6}$$

where we again assume some starting point $w_0 \in W$. When we reach $t = B$ and have exhausted all the batches we say we have complete an **epoch**. Subsequently, we generate a new set of random batches and repeat the process. This process is called **stochastic gradient descent**, or SGD for short, the stochastic referring the the random selecting of batches.

If the batch I is uniformly drawn then

$$\mathbb{E}\left[\nabla_w \ell_I(w)\right] = \nabla_w \ell_{\text{total}}(w),$$

i.e. the gradient of a uniformly drawn random batch is an unbiased estimator of the gradient of the full dataset.

> *Remark* 1.8 (Higher order methods). Gradient descent (stochastic or not) is a first order method since it relies only on the first order derivatives. Higher order methods are rarely used with neural networks because of the large amount of parameters. A model with N parameters has N first order derivatives but N^2 second order derivatives, since neural network commonly have millions to billions of parameters calculating second order derivatives is simply not feasible.

A simple but important modification to the (stochastic) gradient descent algorithm is **momentum**. Instead of taking each step based only on the current gradient we take into account the direction we were moving in in previous steps. The modified gradient descent step is given by

$$\begin{aligned} v_t &= \mu\, v_{t-1} - \eta \nabla_w \ell_{I_t}(w_{t-1}), \\ w_t &= w_{t-1} + v_t, \end{aligned}$$

where we initialize with $v_0 = 0$ and $\mu \in [0, 1)$ is called the **momentum coefficient**. The variable v_t can be interpreted as the current velocity vector along which we are moving in the parameter

space. At each iteration our new velocity takes a fraction of the previous velocity (controlled by μ) and updates it with the current gradient. Observe that for $\mu = 0$, i.e. no momentum, we revert to (1.6). The larger we take μ the more influence the history of the gradients has on our current step.

1.5 Training

In practice the **training process** involves more than applying SGD on the whole dataset. One thing we need to keep in mind is that the dataset is the only real information we have about the underlying function we are trying to approximate, so there is no way to judge how good our approximation is outside of using the dataset. Given that we know that neural networks are universal approximators we can always find a neural networks that perfectly fits any given dataset; making overfitting a given. What we are really after is **generalization**: the ability of our neural network to give the correct output for inputs it has not seen before.

We accomplish this by splitting the dataset and only using part of it to train the network, the remaining part of the dataset (that the network has never seen) we use to test how well our network has generalized. The exact split depends on the situation but using 80% of the dataset for training and 20% for testing is a good rule-of-thumb.

Let us denote our training and testing datasets as

$$\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}} \subset \mathcal{D}.$$

We then perform SGD on the training dataset, i.e. try to minimize the training loss:

$$\ell_{\text{train}}(w) := \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \ell(F(x;w), y). \tag{1.7}$$

But we judge the performance of our network by the testing loss:

$$\ell_{\text{test}}(w) := \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \ell(F(x;w), y). \tag{1.8}$$

By monitoring the testing loss during training we are able to judge for how long we should train, how well our network generalizes and when overfitting sets in. Figure 1.5 illustrates the typical behaviour of loss curves that are encountered during training.

When we are designing a network for a given application we usually do repeated training while we play with the **hyperparameters** to try and improve the testing loss. This can lead us to overfit the hyperparameters to the given testing dataset. There are two things we can do to mitigate this:

1. split the dataset in three parts: training/testing/validation,
2. redo the split randomly every time we retrain.

In the author's opinion the second method is preferable since it is easier to implement and even if the first method is used it is still necessary to redo the split if we decide to make some changes after having used the validation dataset (since if we use it more than once we are back where we started).

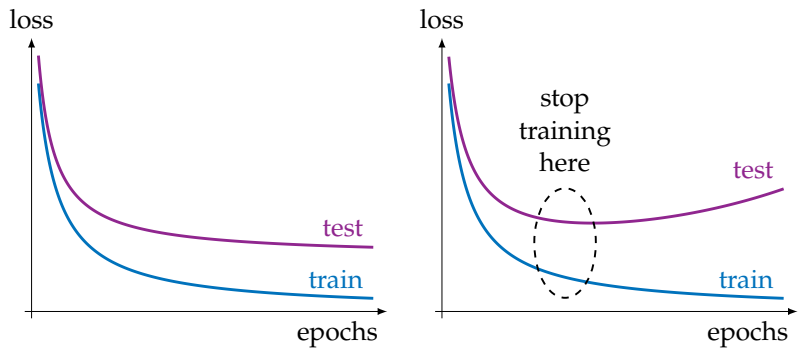


Figure 1.5: Typical progression of the training and testing loss. The training loss will generally converge to some very low value. The testing loss either behaves in a similar fashion and will converge on some higher value, as is illustrated on the left. The testing loss could also start to increase again at some point, as on the right, this indicates overfitting and tells you when to stop training.

Remark 1.9 (Hyperparameters). The hyperparameters are all parameters that we ourselves select and are not trainable parts of the model. Examples are: batch size, learning rate, size of the model, choice of activation function, ratio of the dataset split, etc.

Chapter 2

Deep Learning

The recent advances in machine learning were made largely with so called deep neural networks. The **deep** in this context refers to neural networks where the input will pass through many neurons for processing, an arrangement not dissimilar to the brain and in contrast to our previous shallow networks. Once the potential of these networks was demonstrated the term **deep learning** was coined as a synonym for doing machine learning with deep neural networks. Deep neural networks are a very broad class of models that contains a host of various architectures for different applications, see Wikipedia (2021b) for an overview. We will only cover two main architectures but much of what we will discuss is transferable to other types of networks.

While deep networks can be very powerful there are challenges in getting them to work properly and many aspects of these networks are not yet well understood. This chapter will cover what deep neural networks are and the techniques and algorithms that are necessary to make them work.

2.1 Deep Neural Networks

2.1.1 Feed Forward Networks

The quintessential example of a deep neural network is the **feed forward** neural network wherein connections between nodes do not form cycles. A feed forward network simply takes a set of shallow networks and concatenates them, feeding the output of one layer of neurons as input into the next layer of neurons. Let us formalize this construction.

Let $L \in \mathbb{N}$ and $N_0, N_1, \ldots, N_{L+1} \in \mathbb{N}$. Let $\sigma_1, \ldots, \sigma_L$ be activation functions, which we will assume to be scalar functions that we apply pointwise for the moment. Let $F_i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ be affine transforms given by

$$F_i(\boldsymbol{x}) := A_i \boldsymbol{x} + \boldsymbol{b}_i$$

where $A_i \in \mathbb{R}^{N_i \times N_{i-1}}$ and $\boldsymbol{b}_i \in \mathbb{R}^{N_i}$ for $i \in \{1, \ldots, L+1\}$. Then we call $\mathcal{N} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_{L+1}}$ given by

$$\mathcal{N} := F_{L+1} \circ \sigma_L \circ F_L \circ \cdots \circ \sigma_1 \circ F_1 \tag{2.1}$$

a **feed forward neural network**. This network is said to have L **(hidden) layers**, N_i is said to be the **width** of the layer i and the maximum of all the layer's widths, i.e. $\max_{i=1}^{L}\{N_i\}$, is also called the width of the network.

We usually label inputs with $\boldsymbol{x}$, outputs with $\boldsymbol{y}$ and if necessary the intermediate results with $\boldsymbol{z}^{(i)} \in \mathbb{R}^{N_i}$ as follows:

$$\boldsymbol{z}^{(0)} := \boldsymbol{x}, \qquad \boldsymbol{z}^{(i)} := \sigma_i \left(F_i(\boldsymbol{z}^{(i-1)}) \right),$$

for $i \in \{1, \dots, L+1\}$.

Variants to this architecture are still called feed forward networks. For example we could include multi-variate activation functions such as soft-max or max pooling, this would require us to specify the input and output widths of the layers differently but we could still write the network down as in (2.1). Another common feature is a **skip connection**: some of the outputs of a layer are passed to layers deeper down rather than (or in addition) to the next layer. An example of a feed forward network is illustrated in graph form in Figure 2.1.

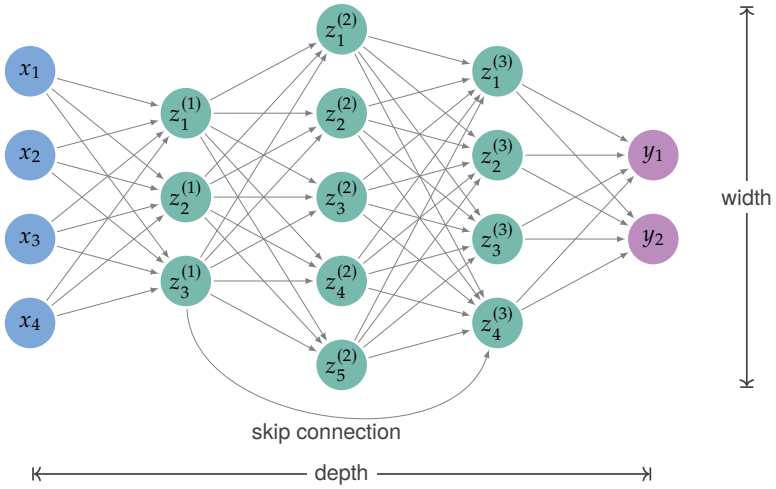


Figure 2.1: Graph representation of a feed forward neural network with 3 hidden layers. Each node is computed as the activation function applied to an affine combination of its inputs. Passing signals to deeper layers rather than the next layer is also a common feature of these types of networks and is called a skip connection.

Remark 2.1 (Recurrent neural networks). The other major class of neural networks besides feed forward are the **recurrent neural networks**, these networks allow for cycles to be present in their graphs. Recurrent neural networks are mainly used for processing sequential data such as in speech and natural language applications. See Wikipedia (2021b) for a survey.

Previously we saw how a shallow neural network with ReLU activation functions results in a piecewise linear function. If we want to model a more complex function we needed to increase the width of the network, the number of linear pieces would scale linearly with the amount of neurons.

Consider a sawtooth function as in Figure 2.2, if we wanted to model a sawtooth with n teeth with a shallow ReLU network we would need $2n$ neurons. But consider the network for 1 tooth:

$$f(x) := \mathrm{ReLU}(2x) - \mathrm{ReLU}(4x - 2) + \mathrm{ReLU}(2x - 2), \tag{2.2}$$

and concatenate it multiple times: we would also get a more and more complex sawtooth for every concatenation as is shown in Figure 2.2.

In fact we would get a doubling of the number of teeth every time we reapply f and so an exponential increase in the amount of linear pieces in the output. This is the major benefit of depth: the complexity of the functions a network can model grows faster with depth than with width.

This is not to say that we should design our networks with maximum depth and minimum width. As is usually the case in engineering there are tradeoffs to consider and any real network strikes a balance between width and depth.

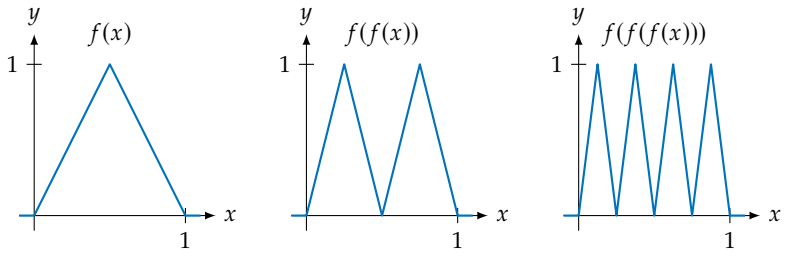


Figure 2.2: Iterative application of the function f from (2.2) causes an exponential increase in the complexity of the result as measured by the amount of linear pieces.

Remark 2.2. While in a shallow ReLU network each linear piece is independent in a deep ReLU network this is not the case. This can be understood by considering that the number of parameters in a deep ReLU network scales linearly with the number of layers while the number of pieces of the output scales exponentially, at some point there will not be enough parameters to describe any piecewise linear function with that amount of pieces. With an equivalent shallow ReLU network the output space is exactly the space of all piecewise linear functions with that amount of pieces.

2.1.2 Vanishing and Exploding Gradients

One difficulty that arises with deep networks is in training. A parameter in an early layer of the network, it has to pass through many layers before finally contributing to the loss.

Let us disregard the affine transforms of a network for the moment and focus on the activation functions. Let σ be the sigmoid activation function given by $\sigma(a) := \frac{1}{1+e^{-a}}$, then define

$$\sigma^N := \underbrace{\sigma \circ \sigma \circ \cdots \circ \sigma}_{N}.$$

For computing the derivative of σ^N we apply the chain rule to find the recursion

$$\frac{\partial}{\partial a}\sigma^N(x) = \sigma'\left(\sigma^{N-1}(a)\right)\ \frac{\partial}{\partial x}\sigma^{N-1}(a).$$

From Figure 2.3 we deduce that $0 < \sigma'(a) \leq 1/4$ so

$$\left|\frac{\partial}{\partial a}\sigma^N(a)\right| \leq \left(\frac{1}{4}\right)^N.$$

Consequently performing a gradient descent step of the form

$$a_{i+1} = a_i - \eta\,\frac{\partial}{\partial a}\sigma^N(a_i),$$

would not change the parameter very much, a problem which becomes worse with every additional layer. Worse is that $\sigma'(a)$ goes to zero quickly for large absolute values of a, as can be seen in Figure 2.3. In this regime, when $\sigma(a)$ is close to 0 or 1, we say the sigmoid is **saturated**.

Theoretically we could train for longer to compensate for very small gradients but in practice this also does not work because of how floating point math works. Specifically when working with floating point numbers we have

$$a + b = a \quad \text{if} \quad |b| \ll |a|,$$

hence a small enough update to a parameter does not actually change the parameter.

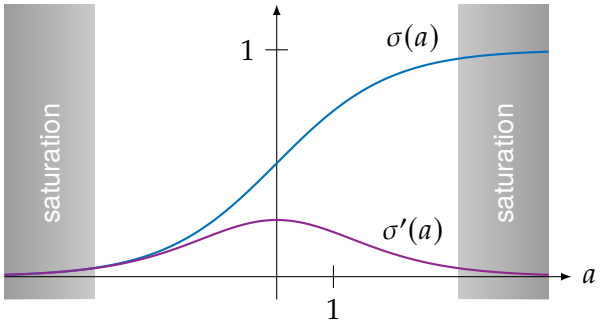


Figure 2.3: The sigmoid activation function and its derivative. Observe that the derivative is at most $1/4$ and goes to zero quickly as $a \to \pm\infty$, this is called saturation.

This behaviour of gradients becoming smaller and smaller the deeper the network becomes is called the **vanishing gradient problem**. The opposite phenomenon can also occur where the gradients become larger and larger leading to unstable training, this is called the **exploding gradient problem**.

We can contrast the behaviour of the sigmoid with the rectified linear unit in the same situation. Recall that the ReLU is defined as $\mathrm{ReLU}(x) := \max\{0, x\}$, then we define

$$\mathrm{ReLU}^N := \underbrace{\mathrm{ReLU} \circ \mathrm{ReLU} \circ \cdots \circ \mathrm{ReLU}}_{N}.$$

Calculating the derivative we get:

$$\frac{\partial}{\partial a} \mathrm{ReLU}^N(a) = \begin{cases} 1 & \text{if } a > 0, \\ 0 & \text{else,} \end{cases} \tag{2.3}$$

which does not depend on the number of layers N. Conceptually (2.3) can be interpreted as a ReLU allowing for a 1 derivative for parameters that are currently affecting the loss (regardless of which layer the parameter resides in) and giving a zero derivative for parameters that do not. The ReLU (at least partially) sidestepping the vanishing/exploding gradient problem accounts for their popularity in deep neural networks.

This is not to say that ReLUs are without issues. If an input to a ReLU is negative it will have a gradient of zero, we say the ReLU or the neuron is **dead**. Consequently all neurons in the previous layer that only (or predominantly) feed into that neuron will also have a gradient of zero (or close to zero). You could visualize this phenomenon as a dead neuron casting a shadow on the lower layers as is illustrated in Figure 2.4.

During training, which neurons are dead and where the shadow is cast changes from batch to batch. As long as most neurons are not perpetually in the shadow we should be able to properly train the network. However, if we happen to get an unlucky initialization it is quite possible that a large part of our network is permanently stuck with zero gradients, this phenomenon is called the **dying ReLU problem**.

Of course, the choice of activation function is not the only thing we need to consider. The gradients in a real network also depend on the current values of the parameters of the affine transforms. So we still need to initialize those parameters to values that do not cause vanishing or exploding gradient problem right from the start. We will get back to parameter initialization later on.

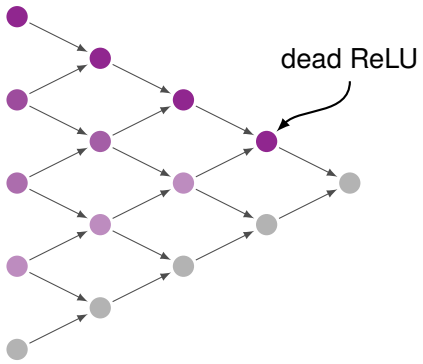


Figure 2.4: A dead ReLU (i.e. with negative argument and so zero derivative) will cause the gradients of the upstream neurons to also become zero, this effect will cascade all the way through to the start of the network. Neurons that feed into non-dead ReLUs will still have a gradient via those live connections. The purple shading of the nodes indicates the degree to which each node is affected by the dead ReLU.

2.1.3 Scaling to High Dimensional Data

If we consider all components of a matrix A_i from (2.1) to be trainable parameters then we say layer i is **fully connected** or **dense**, meaning all outputs of the layer depend on all inputs. When all layers are fully connected we say that the network is fully connected. Having all components of the matrices A_i and vectors $\boldsymbol{b}_i$ as trainable parameters is of course not required. In fact the general fully connected feed forward network from (2.1) is not often used in practice, but many architectures are specializations of this type adapted to specific applications. Specialization in this context primarily means choosing which components of the A_i's and $\boldsymbol{b}_i$'s are trainable and which are fixed to some chosen constant.

The primary reason fully connected network are not used is that they simply do not scale to high-dimensional data. Consider an application where we want to apply a transform on a 1000×1000 color image. The input and output would be elements of the space $\mathbb{R}^{3\times1000\times1000}$, a linear transform in that space would be given by a matrix

$$A \in \left(\mathbb{R}^{3\times1000\times1000}\right)^2.$$

This matrix has $9 \cdot 10^{12}$ components. Storing that many numbers in 32-bit floating point format would take a whopping 36 terabytes, and that is just a single matrix. Clearly this rules out fully connected networks for high dimensional applications such as imaging.

There are 3 strategies employed to deal with this problem:

1) sparsity,
2) weight sharing,
3) parameterization.

Sparsity mean employing sparse matrices for the linear operations. When we fix most entries of a matrix to zero we do not need to store those entries and we simplify the calculation that we need to perform since we already know the result of the zeroed out parts.

With **weight sharing** the same parameter is reused at multiple locations in the matrix. In this case we just have to store the unique parameters and not the whole matrix. Weight sharing is technically a special case of the last strategy.

With **parameterization** we let the components of the matrix depend on a (small) number of parameter. In this case we store the parameters and compute the matrix components when required.

The following matrices are an example comparing the fully connected case with the three proposed strategies:

$$\underset{\text{(fully connected)}}{\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}}, \quad \underset{\text{(sparse)}}{\begin{bmatrix} a & & \\ & b & \end{bmatrix}}, \quad \underset{\text{(shared weights)}}{\begin{bmatrix} a & b & b \\ b & a & b \\ b & b & a \end{bmatrix}}, \quad \underset{\text{(parametrized)}}{\begin{bmatrix} f_1(a,b) & f_2(a,b) & f_3(a,b) \\ f_4(a,b) & f_5(a,b) & f_6(a,b) \\ f_7(a,b) & f_8(a,b) & f_9(a,b) \end{bmatrix}},$$

we need to store 9 numbers for the fully connected matrix but only 2 numbers for each of the other matrices.

These 3 strategies are not mutually exclusive and we will look at an architecture that uses a combination of sparsity and weight sharing later on.

2.2 Initialization

For deep network, the initial values of the parameters make a significant difference to the functioning of the SGD algorithms. Not in the least because initial parameters being too small/large in magnitude would cause vanishing/exploding gradient problems and cause immediate issues for training. In this section we will develop some stochastic initialization schemes that provide a functional starting point for a network to train from. We generally use stochastic methods for initialization since we need initial parameter values in a layer to be sufficiently different. Imagine if the values were the same, then their gradients would also be the same and they would never diverge from each other, locking the network in an untrainable state. The most straightforward method of achieving this is drawing values from a probability distribution, which is currently the common practice. Usually the distributions that are used are either the normal $\mathcal{N}(0, \sigma^2)$ or the uniform $\text{Unif}[-a, a]$, where we need to pick σ^2 or a to avoid training issues such as vanishing/exploding gradient. Each group of parameters (the linear coefficients and the biases in each layer) typically get assigned their own distribution. We will look at how these distributions are currently chosen.

Remark 2.3 (Deterministic initialization scheme). If you recall, in the tutorial notebook `4_FunctionApproximationIn1D.ipynb` we developed a deterministic initialization scheme that outperformed the default stochastic scheme. We did have to use our insight about the problem as a whole (beyond what the data provided) to do it. It is quite conceivable that for a given application a deterministic scheme can be developed that gives a much better starting point for training than the current crop of stochastic schemes.

Remark 2.4 (On the importance of good initialization). A good initialization scheme allows the network to reach a higher performance level in less time. This can have important consequences for large production networks. One such network is GPT-3 (see Brown et al. 2020), which is used for natural language processing, it has 175 billion parameters and training it has reportedly cost 4 million $. Hence better initialization schemes can be of substantial economic value.

2.2.1 Stochastic Initialization

To develop a suitable probability distribution to initialize parameters with we start by looking at the input to a neuron as a vector valued random variable $X \in \mathbb{R}^n$. The output of a neuron is then a random variable $Y \in \mathbb{R}^m$ per

$$Y = \sigma\left(AX + \boldsymbol{b}\right),$$

for some constant matrix $A \in \mathbb{R}^{m\times n}$, bias vector $\boldsymbol{b} \in \mathbb{m}$ and activation function σ. Or alternatively written out per component:

$$Y_i = \sigma\left(\sum_{j=1}^{n} A_{ij}X_j + b_i\right). \tag{2.4}$$

The idea is to then initialize A and $\boldsymbol{b}$ so that the variance of the signal does not change too much from layer to layer:

$$\sum_{i=1}^{m} \mathrm{Var}(Y_i)^2 \approx \sum_{j=1}^{n} \mathrm{Var}(X_j)^2.$$

Controlling the L^2 norm of the signal variances is not necessarily the only possibility here, but it is the choice we will proceed with.

Calculating variances of functions of random variables is difficult in general. The schemes we will be looking at depend on the following approximation.

Lemma 2.5. Let X be a real valued random variable and let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function then

$$\mathrm{Var}\left(f(X)\right) \approx f'\left(\mathbb{E}[X]\right)^2 \cdot \mathrm{Var}(X),$$

assuming the variance of X is finite and f' is differentiable.

Proof. Let $m := \mathbb{E}[X]$ and approximate f by its linearization $f(X) \approx f(m) + f'(m)(X - m)$. Then we find

$$\begin{aligned}
\mathrm{Var}\left(f(X)\right) &= \mathbb{E}\left[\left(f(X) - \mathbb{E}[f(X)]\right)^2\right] \\
&\approx \mathbb{E}\left[\left(f(m) + f'(m)(X-m) - \mathbb{E}[f(m) + f'(m)(X-m)]\right)^2\right] \\
&= \mathbb{E}\left[\left(f(m) + f'(m)(X-m) - f(m) - f'(m)\,\mathbb{E}[(X-m)]\right)^2\right] \\
&= \mathbb{E}\left[\left(f'(m)(X-m)\right)^2\right] \\
&= f'(m)^2\,\mathbb{E}\left[(X-m)^2\right] \\
&= f'\left(\mathbb{E}[X]\right)^2 \cdot \mathrm{Var}(X).
\end{aligned}$$

□

Intuitively, the approximation in Lemma 2.5 can be read as: if the variance of X is small and f' is reasonably bounded then the variance of $f(X)$ will also be small.

Applying Lemma 2.5 to (2.4) yields

$$\mathrm{Var}(Y_i) \approx \sigma'\left(\sum_{j=1}^{n} A_{ij}\,\mathbb{E}[X_j] + b_i\right)^2 \left(\sum_{j=1}^{n} A_{ij}^2\ \mathrm{Var}(X_j)\right).$$

To make progress let us assume $\mathbb{E}[X_1] = \ldots = \mathbb{E}[X_n]$ and $\text{Var}(X_1) = \ldots = \text{Var}(X_n)$, then

$$\text{Var}(Y_i) \approx \underbrace{\sigma'\left(\mathbb{E}[X_1]\sum_{j=1}^{n} A_{ij} + b_i\right)^2 \left(\sum_{j=1}^{n} A_{ij}^2\right)}_{\text{ideally } \approx 1} \text{Var}(X_1).$$

When the bracketed term is approximately 1 then the variances of the outputs Y_i are about the same as those of the inputs X_j.

Let us now turn the A_{ij}'s and b_i's into random variables: $A_{ij} \sim \mu_1$ and $b_i \sim \mu_2$ for all i and j in their respective ranges and where μ_1 and μ_2 are some choice of scalar probability distributions. Ideally we would choose μ_1 and μ_2 so that

$$\mathbb{E}\left[\sigma'\left(\mathbb{E}[X_1]\sum_{j=1}^{n} A_{ij} + b_i\right)^2 \left(\sum_{j=1}^{n} A_{ij}^2\right)\right] = 1. \tag{2.5}$$

This expression allows us to put a condition on our choice of probability distributions that ensures that the variances of the signals between layers stay under control (at least at the start of training). The following examples show how (2.5) is utilized.

Example 2.6 (Sigmoid with balanced inputs)**.** Like before assume $\mathbb{E}[X_1] = \ldots = \mathbb{E}[X_n]$ and $\text{Var}(X_1) = \ldots = \text{Var}(X_n)$ and our goal is choosing a probability distribution for the linear coefficients and one for the biases. Say σ is the sigmoid activation function and we have $\mathbb{E}[X_i] = 0$, i.e. we have balanced inputs. Additionally we want balanced parameter initialization, i.e. $\mathbb{E}[b_i] = 0$ and $\mathbb{E}[A_{ij}] = 0$ for all i, j in their respective ranges. Since $\mathbb{E}[X_i] = 0$ (2.5) reduces to:

$$\mathbb{E}\left[\sigma'(b_i)^2 \left(\sum_{j=1}^{n} A_{ij}^2\right)\right] = \mathbb{E}\left[\sigma'(b_i)^2\right] \mathbb{E}\left[\sum_{j=1}^{n} A_{ij}^2\right],$$

since the b_i's and A_{ij}'s are independent. We know that $0 < \sigma'(x) \leq 1/4$ and that the maximum is achieved at $x = 0$. Hence for that first factor to not become too small we need the variance of b_i to be small since we already decided on setting $\mathbb{E}[b_i] = 0$. Of course the smallest possible variance is zero, so let us be uncompromising and fix $b_i = 0$ for all i. Thus the previous expression becomes

$$\sigma'(0)^2 \, n \, \mathbb{E}\left[A_{ij}^2\right] = \frac{n}{16} \mathbb{E}\left[(A_{ij} - 0)^2\right] = \frac{n}{16} \mathbb{E}\left[(A_{ij} - \mathbb{E}[A_{ij}])^2\right] = \frac{n}{16} \text{Var}(A_{ij}),$$

which equals 1 if

$$\text{Var}(A_{ij}) = \frac{16}{n}.$$

So we could choose our probability distribution for the linear coefficients to be the normal distribution $\mathcal{N}(0, 16/n)$ or the uniform distribution Unif $[-4\sqrt{3}/n, 4\sqrt{3}/n]$. Of course the choice of the type of distribution is free as long as the expected value is zero and the variance is $16/n$, but in practice you will usually only encounter normal or uniform distributions. In any case this choice of expected values and variances will provide some assurance that the signals will not explode or die out as they travel through the network (at least at the start of training).

Example 2.7 (ReLU with balanced inputs)**.** Again assume $\mathbb{E}[X_1] = \ldots = \mathbb{E}[X_n]$ and $\text{Var}(X_1) = \ldots = \text{Var}(X_n)$. This time we use the ReLU activation function and we are going to initialize both

our linear coefficients and biases with a uniform distribution Unif$[-a, a]$, our goal is choosing $a > 0$ in a suitable manner. Assume again that the inputs are balanced, i.e. $\mathbb{E}[X_i] = 0$, then the expression from (2.5) simplifies to

$$\begin{aligned}\mathbb{E}\left[\text{ReLU}'(b_i)^2\left(\sum_{j=1}^{n} A_{ij}^2\right)\right] &= \mathbb{E}\left[\mathbb{1}_{b_i>0}\right]\ \mathbb{E}\left[\left(\sum_{j=1}^{n} A_{ij}^2\right)\right] \\ &= \mathbb{P}(b_i > 0)\ n\ \text{Var}(A_{ij}) \\ &= \frac{n}{2}\ \text{Var}(A_{ij}) \\ &= \frac{na^2}{6},\end{aligned}$$

which equals 1 if $a = \sqrt{6/n}$ so we would draw our A_{ij}'s and b_i's from Unif $\left[-\sqrt{6/n}, \sqrt{6/n}\right]$.

2.2.2 Xavier Initialization

The initialization schemes from the previous section focused on controlling the variance of the signals going forward through the network. While this does help in controlling the vanishing/exploding gradient problem we can also look at gradients directly as they backpropagate through the network, this is the approach taken by Glorot and Bengio (2010). The first author's name is Xavier Glorot and for that reason the scheme we will be seeing is commonly referred to as Glorot or Xavier initialization (as it is in PyTorch for example).

The idea is to treat the partial derivatives of the loss function with regards to the linear coefficients and biases as random variables as well. Consider a setting with a linear activation function and no bias:

$$Y_i = \sum_{j=1}^{n} A_{ij} X_j,$$

where we assume all inputs X_j are distributed i.i.d. with zero mean. We additionally want to initialize our coefficients A_{ij} with mean zero as well. Since the A_{ij} and X_j's are independent the variance distributes over the sum. Additionally we have that

$$\text{Var}(A_{ij}X_j) = \mathbb{E}[X_j]^2\,\text{Var}(A_{ij}) + \mathbb{E}[A_{ij}]^2\,\text{Var}(X_j) + \text{Var}(A_{ij})\,\text{Var}(X_j) = \text{Var}(A_{ij})\,\text{Var}(X_j)$$

since $\mathbb{E}[X_j] = \mathbb{E}[A_{ij}] = 0$. So we work out that

$$\text{Var}(Y_i) = \sum_{j=1}^{n} \text{Var}(A_{ij})\,\text{Var}(X_j) = n\,\text{Var}(A_{ij})\,\text{Var}(X_j).$$

Hence for forward signal propagation we have $\text{Var}(Y_i) = \text{Var}(X_j)$ if

$$\text{Var}(A_{ij}) = \frac{1}{n}. \tag{2.6}$$

But we can look at the backward gradient propagation as well. Let ℓ be a loss function at the end of the network, then we can look at the partial derivatives of ℓ with respect to the inputs

and outputs as random variables as well. Call these random variables $\frac{\partial \ell}{\partial X_i}$ and $\frac{\partial \ell}{\partial Y_i}$, applying the chain rule gives us

$$\frac{\partial \ell}{\partial X_j} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial Y_i} \frac{\partial Y_i}{\partial X_j} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial Y_i} A_{ij}.$$

Now we make the same assumption about the backward gradients as we did about the forward signals, namely that the partial derivatives $\frac{\partial \ell}{\partial Y_i}$ are i.i.d. with zero mean. Then we can do the same calculation as before and find

$$\operatorname{Var}\left(\frac{\partial \ell}{\partial X_j}\right) = m \operatorname{Var}\left(A_{ij}\right) \operatorname{Var}\left(\frac{\partial \ell}{\partial Y_i}\right).$$

Hence if we want to have $\operatorname{Var}\left(\frac{\partial \ell}{\partial X_j}\right) = \operatorname{Var}\left(\frac{\partial \ell}{\partial Y_i}\right)$ for backward gradient propagation we need to set

$$\operatorname{Var}(A_{ij}) = \frac{1}{m}. \tag{2.7}$$

Now unless $n = m$ we cannot satisfy (2.6) and (2.7) at the same time, but we can compromise and set

$$\operatorname{Var}(A_{ij}) = \frac{2}{n+m}. \tag{2.8}$$

Under this choice we can use the normal distribution $\mathcal{N}(0, \frac{2}{n+m})$ or the uniform distribution $\operatorname{Unif}\left[-\sqrt{\frac{6}{n+m}}, \sqrt{\frac{6}{n+m}}\right]$ to draw our coefficients A_{ij} from.

Of course in reality we never use the linear activation function. The original Xavier initialization scheme has been expanded to include specific activation functions. For example for the ReLU by He et al. (2015), they arrive at

$$\operatorname{Var}(A_{ij}) = \frac{4}{n+m}.$$

Which intuitively makes sense: since the ReLU is zero on half its domain the variance of the coefficients needs to be increased to keep the variances of the signals/gradients constant.

Other choices of activation function lead to other multipliers being introduced to the same basic formula (2.8):

$$\operatorname{Var}(A_{ij}) = \alpha^2 \frac{2}{n+m},$$

where α is called the gain and depends on the choice of activation function.

Remark 2.8. See torch.nn.init.xavier_uniform_ and torch.nn.init.xavier_normal_ for PyTorch's implementation of these initialization schemes.

2.2.3 Those are a lot of Assumptions

In the last two sections we made a lot of assumptions to arrive at simple formulas. Some of the assumptions are even verifiably incorrect in the networks we employ. In spite of the coarse and inelegant way these initialization schemes were derived they are widely used for the simple reason that they work. They do not totally solve the vanishing/exploding gradient problem but they still significantly improve the performance of the gradient descent algorithms.

2.3 Convolutional Neural Networks

→ Visualizations of convolutions from Dumoulin and Visin (2018), also available at https://github.com/vdumoulin/conv_arithmetic.

→ Handy reference that provides an overview of CNNs: CNN cheat-sheet.

We previously saw that using fully connected networks for high dimensional data such as images is a non-starter due to the memory and computational requirements involved. The proposed solution was reducing the effective amount of parameters by a combination of using a sparse matrix and weight sharing/parameterization. How exactly we should sparsify the network and which weights should be shared or parametrized had to be determined application by application.

In this section we will look at a combination of sparsity and weight sharing that is suited to data that has a natural spatial structure, think about signals in time (1D), images (2D) or volumetric data (3D). What these types of spatial data have in common is that we treat each 'part' of the input data in the same way, e.g. we do not process the left side of an image in another way than the right side.

The type of network that exploits this spacial structure is called a **Convolutional Neural Networks**, or CNN for short. As the name suggest these networks employ the convolution operation as well as the closely related pooling operation. We will look at how discrete convolution and pooling are defined and how they are used to construct a deep CNN.

2.3.1 Discrete Convolution

Recall that in the familiar continuous setting the convolution of two functions $f, g : \mathbb{R} \to \mathbb{R}$ is defined as:

$$(f * g)(x) := \int_{\mathbb{R}} f(x - y)\, g(y)\, dy, \tag{2.9}$$

which can be interpreted as a filter (or kernel) f being translated over the data g and at each translated location the L^2 inner product is taken.

To switch to the discrete setting we will use notation that is more in line with programming languages. When we have $f \in \mathbb{R}^n$ we will use square brackets and zero-based indexing to access its components, so $f[0] \in \mathbb{R}$ is f's first component and $f[n-1] \in \mathbb{R}$ its last. We use this array-based notation since we will need to do some computations on indices and $f[i - j + 1]$ is easier to read than f_{i-j+1}.

Example 2.9. Let $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ then the familiar matrix product $y = Ax$ can be written in array notation as

$$y[i] = \sum_{j=0}^{n-1} A[i, j]\, x[j]$$

for each $i \in \{0, \ldots, m-1\}$.

Definition 2.10 (Discrete cross-correlation and convolution in 1D)**.** Let $f \in \mathbb{R}^n$ (the input) and $k \in \mathbb{R}^m$ (the kernel) then their discrete cross-correlation $(k \star f) \in \mathbb{R}^{n-m+1}$ is given by:

$$(k \star f)[i] := \sum_{j=0}^{m-1} k[j]\, f[i+j]$$

for $i \in \{0, \dots, n-m\}$.
Discrete convolution is defined similarly but with one of the inputs reversed:

$$(k * f)[i] := \sum_{j=0}^{m-1} k[m-1-j]\, f[i+j]$$

for $i \in \{0, \dots, n-m\}$.

Making an index substitution in the definition of discrete convolution makes the relation to the continuous convolution (2.9) more apparent:

$$(k * f)[i] = \sum_{j=i}^{i+m-1} k[i-j+(m-1)]\, f[j].$$

The idea here is to let the components of the kernel $k \in \mathbb{R}^m$ be trainable parameters. In that context it does not matter whether the kernel is reflected or not and there is no real reason to distinguish convolution from cross-correlation. Consequently in the deep learning field it is usual to refer to both operations as convolution. Most 'convolution' operators in deep learning software are in fact implemented as cross-correlations, as in PyTorch for example. We will adopt the same convention and talk about such subjects as convolution layers and convolutional neural networks but the actual underlying operation we will use is cross-correlation.

As in the continuous case, discrete convolution can be generalized to higher dimensions. For this course we will only look at the 2 dimensional case as extending to higher dimensions is straightforward. To make things slightly simpler we will restrict ourselves to square kernels, in practice kernels are almost always chosen to be square anyway.

Definition 2.11. (Discrete cross-correlation in 2D) Let $f \in \mathbb{R}^{h \times w}$ (the input, read h as height and w as width) and $k \in \mathbb{R}^{m \times m}$ (the kernel) then their discrete cross-correlation $(k \star f) \in \mathbb{R}^{(h-m+1)\times(w-m+1)}$ is given by

$$(k \star f)[i_1, i_2] := \sum_{j_1, j_2=0}^{m-1} k[j_1, j_2]\, f[i_1 + j_1, i_2 + j_2]$$

for $i_1 \in \{0, \dots, h-m\}$ and $i_2 \in \{0, \dots, w-m\}$.

The convolution operator "*" can be extended to 2 dimensions similarly but we will only be using cross-correlation for the remainder of our discussion of convolutional neural networks.

For a visualization of discrete convolution in 2D, see Figure 2.5.

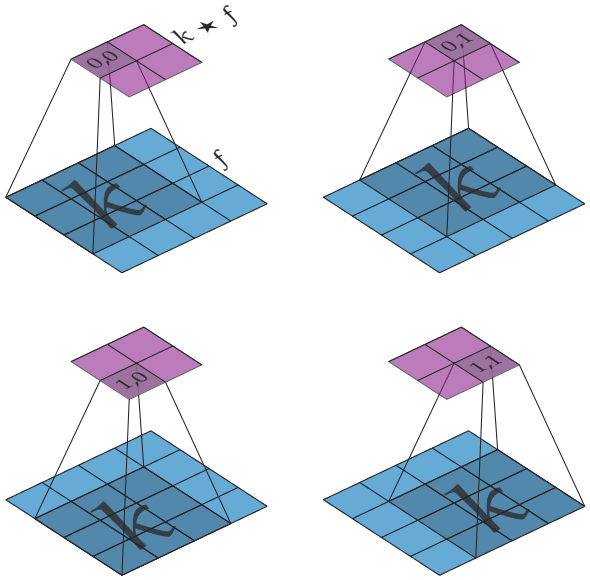


Figure 2.5: An illustration of convolution (or cross-correlation) in 2D. Here the input $f \in \mathbb{R}^{4\times4}$ (colored blue) is convolved with the kernel $k \in \mathbb{R}^{3\times3}$, which yields an output $(k \star f) \in \mathbb{R}^{2\times2}$ (colored purple).

Remark 2.12 (Index interpretation conventions). In Definition 2.10 we called the first dimension of an array $f \in \mathbb{R}^{h\times w}$ the height and the second the width. Additionally, if you look at Figure 2.5 we place the origin point $(0, 0)$ in the top left. This is different to the Cartesian convention of denoting points in the plane with (x, y) (i.e. width first, height second) and having the origin in the bottom left. Both conventions are illustrated bellow for an element of $\mathbb{R}^{3\times4}$.

Cartesian convention

0, 3	2, 3	2, 3
0, 2	2, 2	2, 2
0, 1	1, 1	2, 1
0, 0	1, 0	2, 0

y (↑), x (→)

Array convention

i_1 (↓), i_2 (→)

0, 0	0, 1	0, 2	0, 3
1, 0	2, 1	3, 2	4, 3
2, 0	2, 1	2, 2	2, 3

Of course choosing either convention does not change the underlying object, merely the interpretation of the indices and shape in colloquial terms. The array convention is almost universally adopted in software and it is used in PyTorch, for that reason we will adopt it when dealing with spatial data.

2.3.2 Padding

The convolution operations we proposed so far have the property that the output is of smaller size than the input. Depending on our goal this might or might not be desirable. If shrinking output size is not desirable we can use padding to ensure the output size is the same as the input size. The most common type of padding is zero padding, which we will look at for the 2 dimensional case.

Definition 2.13 (Zero padding in 2D)**.** Let $f \in \mathbb{R}^{h\times w}$ and let p_t, p_b, p_l, p_r, which we read as top, bottom, left and right padding respectively. Then we define $ZP_{p_t,p_b,p_l,p_r} f \in \mathbb{R}^{(h+p_t+p_b)\times(w+p_l+p_r)}$ as

$$ZP_{p_t,p_b,p_l,p_r} f[i_1, i_2] := \begin{cases} 0 & \text{if } i_1 < p_t \text{ or } i_1 \geq h + p_t \\ & \text{or } i_2 < p_l \text{ or } i_2 \geq w + p_l, \\ f[i_1 - p_t,\ i_2 - p_l] & \text{else,} \end{cases}$$

for all $i_1 \in \{0, \ldots, h + p_t + p_b - 1\}$ and $i_2 \in \{0, \ldots, w + p_l + p_r - 1\}$.

If we then have an $f \in \mathbb{R}^{h\times w}$ and we want to convolve with a kernel $\mathbb{R}^{m\times m}$ while keeping the shape of the output the same we can choose

$$p_t := \left\lfloor \frac{m-1}{2} \right\rfloor, \quad p_b := \left\lceil \frac{m-1}{2} \right\rceil, \quad p_l := \left\lfloor \frac{m-1}{2} \right\rfloor, \quad p_r := \left\lceil \frac{m-1}{2} \right\rceil,$$

then $k \star ZP_{p_t,p_b,p_l,p_r} f \in \mathbb{R}^{h\times w}$. We leave verifying this claim as an exercise. An example of this technique is illustrated in Figure 2.6.

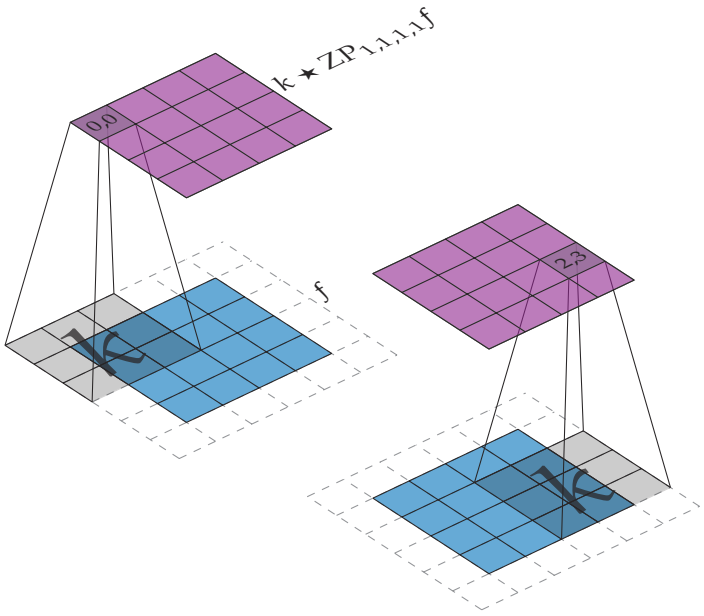


Figure 2.6: Adding padding to the input allows the output of the convolution operation to retain the size of the original input. The most common type of padding is zero-padding, where each out-of-bounds value is assumed to be zero.

Many more padding techniques exist, they only vary in how the out-of-bounds values are chosen. In PyTorch the available padding modes are listed in pytorch.org/docs/stable/generated/torch.nn.functic

2.3.3 Max Pooling

The second operation commonly found in CNN is max pooling. We already saw an activation function called max pooling previously, this is exactly what is used in a CNN but with a particular choice of subsets to take maxima over that plays well with the spatial structure of the data.

The idea is to have a 'window' slide over the input data in the same way that a convolution kernel slides over the data and then take the maximum value in each window. We will take a

look at a particular type of 2D max pooling that is commonly found in CNNs used for image processing and classification applications.

Definition 2.14 ($m \times m$ max pooling in 2D)**.** Let $f \in \mathbb{R}^{h \times w}$ and let $m \in \mathbb{N}$. Then we define $MP_{m,m} f \in \mathbb{R}^{\lfloor h/m \rfloor \times \lfloor w/m \rfloor}$ as

$$MP_{m,m} f[i_1, i_2] := \max_{\substack{0 \le j_1 < m \\ 0 \le j_2 < m}} f[i_1 m + j_1, i_2 m + j_2]$$

for all $i_1 \in \{0, \ldots, \lfloor h/m \rfloor\}$ and $i_2 \in \{0, \ldots, \lfloor w/m \rfloor\}$.

Two things to note about this particular definition. First, if h and/or w is not divisible by m then some values at the edges will be ignored entirely and not contribute to the output. We could modify the definition to resolve this but in practice m is usually set to 2 and so in the worst case we lose a single row of values at the edge, and that is only if the input has odd dimensions.

Second, we move the window in steps of m in both directions instead of taking steps of 1, this is called having a **stride** of m. Having a stride larger than 1 allows max pooling to quickly reduce the dimensions of the data. Figure 2.7 shows an example of how $MP_{2,2}$ works.

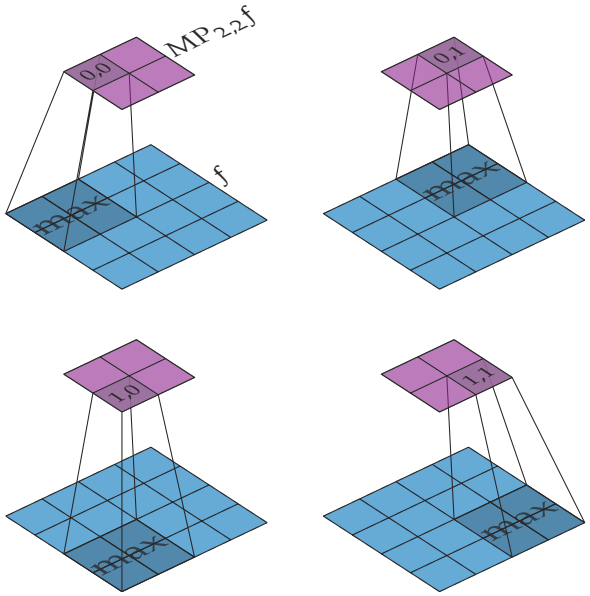


Figure 2.7: Max pooling is similar to convolution in that a window of a certain size slides over the input, instead of taking a weighted sum inside the window we take the maximum value. Usually the window is moved in strides equal to its size so that the outputs are the maxima from disjoint sets of the input. The pooling operation in the figure is usually just called 2×2 max pooling, referring to both the window size and stride used.

2.3.4 Convolutional Layers

So how are convolutions/cross-correlations used in a neural network? First of all we organize our inputs and outputs differently, instead of the inputs and outputs being element of $\mathbb{R}^n$ for some n we want to keep the spatial structure. In the 2D case it makes sense to talk about objects with height and width, so elements in $\mathbb{R}^{h \times w}$ for some choice of $h, w \in \mathbb{N}$. We will call these objects 2D maps or just **maps**. We will want more than one map at any stage of the network, we say we want multiple **channels**.

Concretely, say we have a deep network where we index the number of layers with i, then we denote the input of layer i as an element in $\mathbb{R}^{C_{i-1} \times h_{i-1} \times w_{i-1}}$ and the output of layer i as an element

of $\mathbb{R}^{C_i \times h_i \times w_i}$. We say layer i has C_{i-1} **input channels** and C_i **output channels**. We call $h_{i-1} \times w_{i-1}$ and $h_i \times w_i$ the shape of the input respectively output maps.

Example 2.15 (Image inputs)**.** If we design a neural network to process color 1080p images then $C_0 = 3$ (RGB images have 3 color channels), $h_0 = 1080$ and $w_0 = 1920$, i.e. the input to the first layer is an element in $\mathbb{R}^{3\times1080\times1920}$. For a monochrome image we would need just 1 channel.

A **convolution layer** is just a specialized network layer, hence it follows the same pattern of first doing a linear transform, then adding a bias and finally applying an activation function. The activation function in a CNN is typically a ReLU or max pooling function (or both). The linear part will consist of taking convolutions of the input maps, but there are several ways to do that, we will look at two of them.

The first, and most straightforward one, is called **single channel convolution** (alternatively know as **depthwise convolution**). With this method we assign a kernel of a certain size to each input channel and then perform the convolution of each input channel with each kernel. This gives us a number of maps equal to the number of input channels, we subsequently take point-wise linear combinations of those maps to generate the desired number of output maps.

Definition 2.16 (Single channel convolution)**.** Let $f \in \mathbb{R}^{C\times h\times w}$ and let $k \in \mathbb{R}^{C\times m\times m}$. We call C the number of input channels, $h \times w$ the input map shape and $m \times m$ the kernel shape. Let $A \in \mathbb{R}^{C'\times C}$, we call C' the number of output channels. Then we define $SCC_{k,A}(f) \in \mathbb{R}^{C'\times(h-m+1)\times(w-m+1)}$ as:

$$SCC_{k,A}(f)[c', i_1, i_2] := \sum_{c=0}^{C-1} A[c', c]\; k[c, \cdot, \cdot] \star f[c, \cdot, \cdot],$$

for all $c' \in \{0, \dots, C'-1\}$, $i_1 \in \{0, \dots, h-m+1\}$ and $i_2 \in \{0, \dots, w-m+1\}$.

This operation allows the entries of the kernel stack k and matrix A to be trainable, hence the number of trainable parameters is $C \cdot m^2 + C' \cdot C$. Figure 2.8 gives a visualization of how single channel convolution works.

An alternative way of using convolution to build a linear operator is **multi channel convolution** (alternatively known as **multi channel multi kernel** (MCMK) convolution). Instead of assigning a kernel to each input channel and then taking linear combinations we assign a kernel to each combination of input and output channels.

Definition 2.17 (Multi channel convolution)**.** Let $f \in \mathbb{R}^{C\times h\times w}$ and let $k \in \mathbb{R}^{C'\times C\times m\times m}$. We call C the number of input channels, C' the number of output channels, $h \times w$ the input map shape and $m \times m$ the kernel shape. Then we define $MCC_k(f) \in \mathbb{R}^{C'\times(h-m+1)\times(w-m+1)}$ as:

$$MCC_k(f)[c', i_1, i_2] := \sum_{c=0}^{C-1} k[c', c, \cdot, \cdot] \star f[c, \cdot, \cdot],$$

for all $c' \in \{0, \dots, C'-1\}$, $i_1 \in \{0, \dots, h-m+1\}$ and $i_2 \in \{0, \dots, w-m+1\}$.

Under this construction the kernel components are the trainable parameters and so we have a total of $C' \cdot C \cdot m^2$ trainable parameters. The multi channel convolution construction is illustrated in Figure 2.9.

Neural network frameworks like PyTorch implement the multi channel version. CNN's in literature are also generally formulated with multi channel convolutions. So why did we also introduce single channel convolutions?

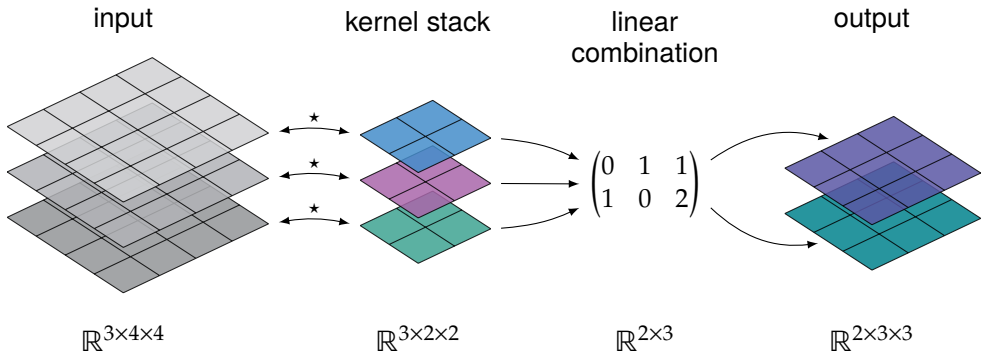


Figure 2.8: With **single channel convolution**, also called **depthwise convolution**, each input channel gets assigned a single kernel. After doing C_{in} cross-correlations we take pointwise linear combinations of the resulting maps to generate the desired number of output maps. In this example that yields a total of 18 trainable parameters (or 20 if we include a bias per output channel).

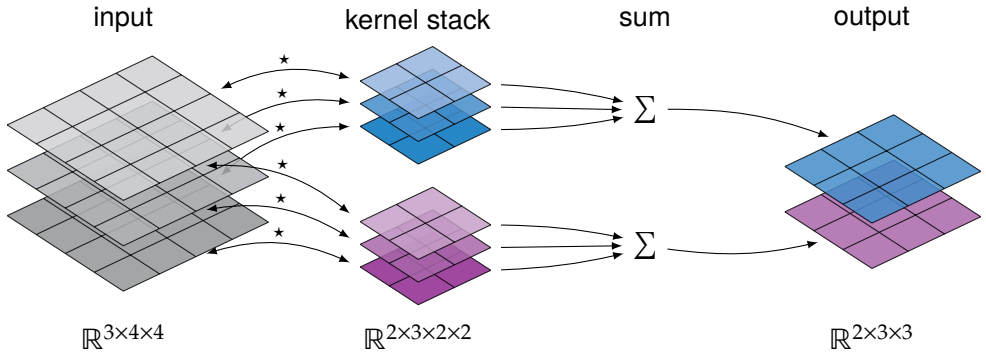


Figure 2.9: With **multi channel convolution** each output channel gets assigned a stack of kernels, where each stack has a kernel per input channel. This results in $C_{out} \cdot C_{in}$ kernels. The resulting outputs of the $C_{out} \cdot C_{in}$ cross-correlations are then summed up pointwise per output channel resulting in C_{out} output channels. In this example that yields a total of 24 trainable parameters (or 26 if we include a bias per output channel).

First of all, single and multi channel convolution are equivalent in the sense that given an instance of one I can always construct an instance of the second that does the exact same calculation. Consequently we can work with whichever construction we prefer for whatever reason without losing anything. The nice thing about single channel convolution is that there is a clear separation between the processing done inside a particular channel and the way the input channels are combined to create output channels. These two processing steps are fused together in the multi channel convolution operation. In the author's opinion this makes the multi channel technique harder to reason about, having two distinct steps that do two distinct things seems more elegant.

All that is left now to create a full CNN layer is combining one of the convolution operations, add padding if desirable and pass the result through a max pooling operation and/or a scalar activation function such as a ReLU.

2.3.5 Classification Example: MNIST & LeNet-5

Figure 2.10: The MNIST dataset (Modified National Institute of Standards and Technology dataset) consists of a large collection of 28×28 grayscale images of hand drawn digits. The goal is assigning to each image the correct 0-9 label.

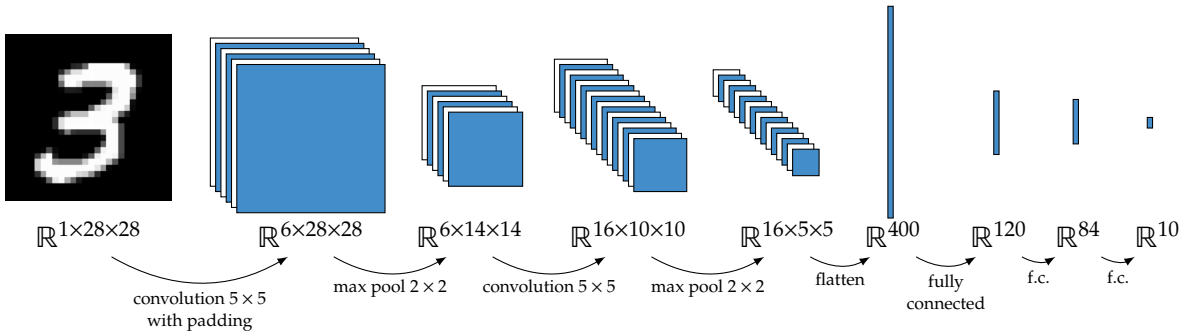


Figure 2.11: A modernized LeNet-5 network, the classic LeNet-5 network from LeCun, Bottou, et al. (1998) used sigmoidal activation functions, unusual sub-sampling layers and Gaussian connections. Replacing these with ReLUs, max pooling and fully connected layers yields a somewhat simpler network that works equally well.

2.4 Automatic Differentiation & Backpropagation

All our training algorithms are gradient based and so our ability to train a network rests on us computing those gradients. The traditional approaches we could take are:

(1) compute the derivative formulas ourselves and code them into the program,

(2) use a computer algebra system to perform symbolic differentiation and use the resulting formula,

(3) use numeric differentiation (i.e. finite differences).

None of these three options are appealing. We definitely do not want to work out the derivatives ourselves for every network we define so option (1) is out. Symbolic differentiation with computer algebra systems like Mathematica usually produces complex and cryptic expressions in non-trivial cases. Subsequently evaluating these complex expressions is usually anything but efficient, this rules out option (2). For networks with millions if not billions of parameters computing finite differences would entail millions or billions of evaluations of the network, this makes option (3) impossible to use.

What we need is a fourth technique: **automatic differentiation**, also called **algorithmic differentiation** or simply **autodiff**. We will restrict ourselves to looking at automatic differentiation in the context of feed-forward neural networks but it is a very general technique, see Baydin et al. (2018) for a broader survey of its uses in machine learning. In introducing automatic differentiation it is perhaps best to emphasizing that it is **not** symbolic differentiation nor numeric differentiation, even though it contains elements of both.

Remark 2.18 (Autograd vs. autodiff). Autograd is a particular autodiff implementation in Python. Autograd served as a prototype for a lot of autodiff implementations, including PyTorch's, for that reason in PyTorch the term **autograd** is used as a synonym for autodiff.

2.4.1 Notation and an Example

Let us say we have a network $F : \mathbb{R} \times \mathbb{R}^2 \to \mathbb{R}$ with two parameters a and b and loss function ℓ. We are interested in computing:

$$\frac{\partial}{\partial a}\ell(F(x_0; a, b), y_0)\bigg|_{(a,b)} \quad \text{and} \quad \frac{\partial}{\partial b}\ell(F(x_0; a, b), y_0)\bigg|_{(a,b)}, \tag{2.10}$$

i.e. the partial derivatives with respect to out parameters at their current values which are just real numbers. Note how we use a and b first as dummy variables to indicate the input we want to differentiate over and then again as real valued function arguments.

Partial derivative notation can get cluttered quickly, as (2.10) shows. To simplify things we are going to introduce some notational conventions. Let $f : \mathbb{R}^2 \to \mathbb{R}$ be given by

$$f(x, y) = g(h(x, y), k(x, y)), \tag{2.11}$$

where g, h and k are also functions from $\mathbb{R}^2$ to $\mathbb{R}$. We evaluate f at a fixed (x, y) by calculating:

$$u = h(x, y), \quad v = k(x, y), \quad z = g(u, v) = f(x, y),$$

which are all just real numbers and not functions like f, g, h and k are. Now we introduce the following notations for the intermediate results u and v:

$$\frac{\partial u}{\partial x} := \frac{\partial}{\partial x}h(x, y)\bigg|_{(x,y)}, \qquad \frac{\partial u}{\partial y} := \frac{\partial}{\partial y}h(x, y)\bigg|_{(x,y)},$$
$$\frac{\partial v}{\partial x} := \frac{\partial}{\partial x}k(x, y)\bigg|_{(x,y)}, \qquad \frac{\partial v}{\partial y} := \frac{\partial}{\partial y}k(x, y)\bigg|_{(x,y)},$$

and for the final output z we define:

$$\frac{\partial z}{\partial u} := \frac{\partial}{\partial u}g(u, v)\bigg|_{(u,v)}, \qquad \frac{\partial z}{\partial v} := \frac{\partial}{\partial v}g(u, v)\bigg|_{(u,v)}$$
$$\frac{\partial z}{\partial x} := \frac{\partial}{\partial x}f(x, y)\bigg|_{(x,y)}, \qquad \frac{\partial z}{\partial y} := \frac{\partial}{\partial y}f(x, y)\bigg|_{(x,y)},$$

which are all real numbers. At first glance the partial derivative of one real number with respect to another real number is nonsensical, but if we use one real number in the computation of a second one it does make sense to ask how sensitive the value of the second is with respect to the first if we changed it a little bit. This sensitivity is of course nothing but the partial derivative of the function used in the computation of the second value evaluated at the first value. But since we are not interested in the whole partial derivatives it simplifies things to keep the function and its partial derivatives implicit and use the simpler notation we just introduced.

Using this notation the chain rule applied to (2.11) can be expressed very succinctly as

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial x} \quad \text{and} \quad \frac{\partial z}{\partial y} = \frac{\partial z}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial y}. \tag{2.12}$$

We are always looking to minimize our chosen loss function, hence we are mainly interested in the partial derivatives of the loss function. We will use the following notation to further abbreviate the partial derivatives of the loss: let $\ell \in \mathbb{R}$ be the loss produced at the end of our calculation and let $\alpha \in \mathbb{R}$ be either a parameter or intermediate result used in the calculation of ℓ then we write

$$\overline{\alpha} := \frac{\partial \ell}{\partial \alpha},$$

which we refer to as a **gradient** (technically the value of the gradient of the loss function with respect to α evaluated at the current value of α, but that is quite the mouthful). So in the case of the two-parameter network F from before: for a given set of parameters $a, b \in \mathbb{R}$ we need to calculate the gradients $\overline{a}, \overline{b} \in \mathbb{R}$, which are exactly the evaluated partial derivatives from (2.10).

As an example to see how this notation works let us pick a concrete example and systematically work through differentiating it. Let $F(x; a, b) := \sigma(ax + b)$ for some choice of (differentiable) activation function $\sigma : \mathbb{R} \to \mathbb{R}$. Let $(x_0, y_0) \in \mathbb{R}^2$ be some data point and let us use the loss function $(y, y') \mapsto \frac{1}{2}(y - y')^2$. After we pick our parameter values $a, b \in \mathbb{R}$ we can simply compute the corresponding loss, doing this is called the **forward pass** and is shown on the left side in (2.13) below.

forward		backward	
	$a, b, x_0, y_0 \in \mathbb{R}$		$\overline{a} = \frac{\partial \ell}{\partial z}\frac{\partial z}{\partial a} = \overline{z}\, x_0\,, \quad \overline{b} = \overline{z}$
↓	$z = ax_0 + b$	↑	$\overline{z} = \frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial z} = \overline{y}\,\sigma'(z)$
	$y = \sigma(z)$		$\overline{y} = \frac{\partial \ell}{\partial y} = y - y_0$
	$\ell = \frac{1}{2}(y - y_0)^2$		$\overline{\ell} = \frac{\partial \ell}{\partial \ell} = 1$

(2.13)

After having evaluated the network we can start calculating the partial derivatives of the loss with respect the the parameters a and b by apply the chain rule from back to front, this is called the **backward pass** and is shown on the right side in (2.13). Some thing to note about the schematic in (2.13).

- Everything that we wrote down is a numeric computation and the whole schematic can be executed by a computer as is.
- In writing down the backward pass we did use our symbolic knowledge of how the operations in the forward pass need to be differentiated if we look at them as functions.

- Writing down the trivial $\overline{\ell} = 1$ is of course redundant, but autodiff implementations such as PyTorch's do actually start the backward pass by creating a single element tensor containing the value 1 (and it makes the schematic look symmetric).
- Some of the intermediate values computed during the forward pass are reused during the backward pass. In the schematic (2.13) the reused values have been shaded in green.

Remark 2.19. In PyTorch, the intermediate results computed during the forward pass that need to be retained for the backward pass are called **saved tensors**. In a typical neural network many of the intermediate results have to be saved for the backward pass, this is the reason that training a neural network requires a lot of memory.

Nothing we have done in the example in (2.13) is novel, we just did what we would normally do if asked to calculate these partial derivatives. Only, we wrote it down systematically in a way that we can automate.

2.4.2 Automation & the Computational Graph

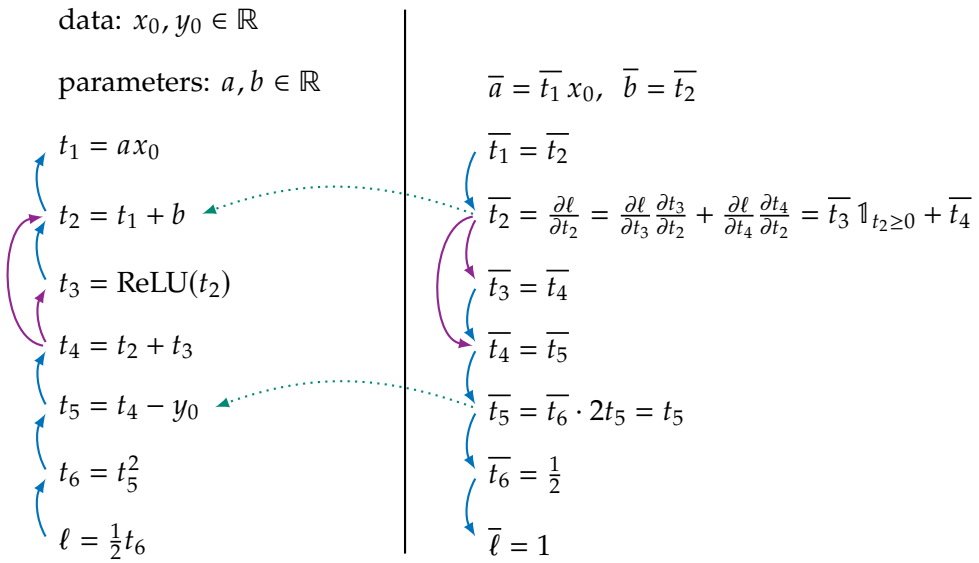
The key to automating the type of calculation in (2.13) and (2.14) is splitting it into primitive operations and tracking how they are composed. Consider the network $F(x; a, b) := \mathrm{ReLU}(ax+b)$ with loss function $(y, y') \mapsto \frac{1}{2}(y-y')^2$ evaluated for some data point $(x_0, y_0) \in \mathbb{R}^2$ and parameter values $a, b \in \mathbb{R}$. Evaluating the network one primitive operation at a time looks as follows.

forward ↓

data: $x_0, y_0 \in \mathbb{R}$	
parameters: $a, b \in \mathbb{R}$	$\overline{a} = \frac{\partial \ell}{\partial t_1}\frac{\partial t_1}{\partial a} = \overline{t_1}\, x_0, \quad \overline{b} = \frac{\partial \ell}{\partial t_2}\frac{\partial t_2}{\partial b} = \overline{t_2}$
$t_1 = a x_0$	$\overline{t_1} = \frac{\partial \ell}{\partial t_2}\frac{\partial t_2}{\partial t_1} = \overline{t_2}$
$t_2 = t_1 + b$	$\overline{t_2} = \frac{\partial \ell}{\partial t_3}\frac{\partial t_3}{\partial t_2} = \overline{t_3}\, \mathbb{1}_{t_2 \geq 0}$
$t_3 = \mathrm{ReLU}(t_2)$	$\overline{t_3} = \frac{\partial \ell}{\partial t_4}\frac{\partial t_4}{t_3} = \overline{t_4}$
$t_4 = t_3 - y_0$	$\overline{t_4} = \frac{\partial \ell}{\partial t_4} = \frac{\partial \ell}{\partial t_5}\frac{\partial t_5}{\partial t_4} = \overline{t_5} \cdot 2t_4 = t_4$
$t_5 = t_4^2$	$\overline{t_5} = \frac{\partial \ell}{\partial t_5} = \frac{1}{2}$
$\ell = \frac{1}{2} t_5$	$\overline{\ell} = 1$

↑ backward (2.14)

The backward pass consists again of numeric computations but at every step we need to know how the value computed at that line, say α, is used so we are able to compute the correct partial derivative $\overline{\alpha}$. In the case of (2.14) that is fairly straightforward as every intermediate value is only used in the next step i.e. t_i only depend on t_{i-1} and the parameters. Consequently $\overline{t_i}$ only depends on $\overline{t_{i+1}}$ and t_i. A more general example that has a slightly more complicated structure is the network $F(x; a, b) := \mathrm{ReLU}(ax + b) + (ax + b)$. We can write this network out in primitive form as well, we omit the forward/backward arrows but add a dependency graph that shows how the intermediate results depend on each other.



(2.15)

The dependency graph of the gradients $\overline{t_i}$ in (2.15) is naturally the reverse of the dependency graph of the values t_i augmented with dependencies on the results from the forward pass. Both the values t_3 and t_4 depend on the value t_2 hence $\overline{t_2}$ depends on both $\overline{t_3}$ and $\overline{t_4}$ in addition to t_2 itself. This double dependency causes the chain rule applied to $\overline{t_2}$ to have two terms, of course this generalizes to multiple dependencies.

Constructing this **computational graph** is exactly how machine learning frameworks such as PyTorch implement gradient computation. Each time you perform an operation on one or more tensors a new node is added to the graph to record what operation was performed and on which inputs the output depends. Then, when it becomes time to compute the gradient (i.e. `.backward()` is called in PyTorch) the graph is traversed back to front.

As mentioned, the graph records what operation was performed at each node. This is necessary because what backward computation needs to be performed at each node depends on what the corresponding forward computation was, this is where our symbolic knowledge needs to be added.

2.4.3 Implementing Operations

In the previous section we saw how the evaluation of a neural network (or any computation for that matter) can be expressed as a computational graph where each node correspond to a (primitive) operation. To be able to do the backward pass each node needs to know how to compute its own partial derivative(s). Previous examples (2.13) (2.14) and (2.15) only used simple scalar operations, now will explore how to implement both the forward and backward computation for a general multivariate vector-valued function.

Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable map, let $\boldsymbol{x} = [x_1 \ \cdots \ x_n]^T \in \mathbb{R}^n$ and $\boldsymbol{y} = [y_1 \ \cdots \ y_m]^T \in \mathbb{R}^m$ so that $\boldsymbol{y} = F(\boldsymbol{x})$. Let $\ell \in \mathbb{R}$ be the final loss computed for a neural network that contains F as

one of its operations. Then we can generalize our gradient notation from scalars to vector as

$$\overline{\boldsymbol{x}} := \frac{\partial \ell}{\partial \boldsymbol{x}} := \begin{bmatrix} \frac{\partial \ell}{\partial x_1} \\ \vdots \\ \frac{\partial \ell}{\partial x_n} \end{bmatrix}, \qquad \overline{\boldsymbol{y}} := \frac{\partial \ell}{\partial \boldsymbol{y}} := \begin{bmatrix} \frac{\partial \ell}{\partial y_1} \\ \vdots \\ \frac{\partial \ell}{\partial y_m} \end{bmatrix}.$$

During the forward pass the task is: given $\boldsymbol{x}$ compute $\boldsymbol{y} = F(\boldsymbol{x})$. During the the backward pass the task is: given $\boldsymbol{x}$ and $\overline{\boldsymbol{y}}$ compute $\overline{\boldsymbol{x}}$, we call this backward operation $\overline{F} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$. Let us see what it looks like

$$\begin{aligned}
\overline{\boldsymbol{x}} &= \begin{bmatrix} \frac{\partial \ell}{\partial x_1} \\ \vdots \\ \frac{\partial \ell}{\partial x_n} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{j=1}^m \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial x_1} \\ \vdots \\ \sum_{j=1}^m \frac{\partial \ell}{\partial y_j} \frac{\partial y_j}{\partial x_n} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{j=1}^m \overline{y_j} \frac{\partial y_j}{\partial x_1} \\ \vdots \\ \sum_{j=1}^m \overline{y_j} \frac{\partial y_j}{\partial x_n} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \overline{y_1} \\ \vdots \\ \overline{y_m} \end{bmatrix} \\
&= J(\boldsymbol{x})^T \overline{\boldsymbol{y}},
\end{aligned}$$

where $J(\boldsymbol{x})$ is the Jacobian matrix of F evaluated in $\boldsymbol{x}$. So not unexpectedly we end up with the general form of the chain rule.

Example 2.20 (Pointwise addition)**.** Let $F : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ be defined by $F(\boldsymbol{a}, \boldsymbol{b}) := \boldsymbol{a} + \boldsymbol{b}$, with $\boldsymbol{a} = [a_1\ a_2]^T \in \mathbb{R}^2$ and $\boldsymbol{b} = [b_1\ b_2]^T$. We can equivalently define F as $F : \mathbb{R}^4 \to \mathbb{R}^2$ as follows:

$$\boldsymbol{c} = F(\boldsymbol{a}, \boldsymbol{b}) = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix}.$$

Since this is a linear operator the Jacobian matrix is just the constant matrix above. The backward operation $\overline{F} : \mathbb{R}^4 \times \mathbb{R}^2 \to \mathbb{R}^4$ is then given by

$$\begin{bmatrix} \overline{\boldsymbol{a}} \\ \overline{\boldsymbol{b}} \end{bmatrix} = \overline{F}(\boldsymbol{a}, \boldsymbol{b}, \overline{\boldsymbol{c}}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \overline{c_1} \\ \overline{c_2} \end{bmatrix} = \begin{bmatrix} \overline{c_1} \\ \overline{c_2} \\ \overline{c_1} \\ \overline{c_2} \end{bmatrix} = \begin{bmatrix} \overline{\boldsymbol{c}} \\ \overline{\boldsymbol{c}} \end{bmatrix}$$

for a gradient $\overline{\boldsymbol{c}} = [c_1\ c_2]^T \in \mathbb{R}^2$.

So to implement this operation we do not have to retain the inputs $\boldsymbol{a}$ and $\boldsymbol{b}$, we can implement the backward calculation by simply copying the incoming gradient $\overline{\boldsymbol{c}}$ and passing the two copies up the graph.

Example 2.21 (Copy)**.** Let $F : \mathbb{R}^n \to \mathbb{R}^{2n}$ be given by

$$\begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix} = F(\boldsymbol{a}) = \begin{bmatrix} a_1 \\ \vdots \\ a_n \\ a_1 \\ \vdots \\ a_n \end{bmatrix},$$

with $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^n$. This operation makes two copies of its input, it is clearly linear with Jacobian

$$J = \begin{bmatrix} I_n \\ I_n \end{bmatrix},$$

where I_n is a unit matrix of size $n \times n$. Given gradients $\overline{\boldsymbol{b}}, \overline{\boldsymbol{c}} \in \mathbb{R}^n$ of the outputs the gradient $\overline{\boldsymbol{a}}$ of the input is calculated as:

$$\overline{\boldsymbol{a}} = \overline{F}(\boldsymbol{a}, \overline{\boldsymbol{b}}, \overline{\boldsymbol{c}}) = \begin{bmatrix} I_n & I_n \end{bmatrix} \begin{bmatrix} \overline{\boldsymbol{b}} \\ \overline{\boldsymbol{c}} \end{bmatrix} = \overline{\boldsymbol{b}} + \overline{\boldsymbol{c}}.$$

Example 2.22 (Inner product)**.** In this example the Jacobian is not constant and we do have to retain the inputs to be able to do the backward pass. Let $F : \mathbb{R}^{2n} \to \mathbb{R}$ be given by

$$c = F(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=1}^{n} a_i b_i,$$

for $\boldsymbol{a} = [a_1 \ \cdots \ a_n]^T$ and $\boldsymbol{b} = [b_1 \ \cdots \ b_n]^T \in \mathbb{R}^n$. Then the Jacobian matrix evaluated at $[\boldsymbol{a} \ \boldsymbol{b}]^T$ is given by

$$J(\boldsymbol{a}, \boldsymbol{b}) = \begin{bmatrix} b_1 & \cdots & b_n & a_1 & \cdots & a_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}^T & \boldsymbol{a}^T \end{bmatrix}.$$

Given a (scalar) gradient $\overline{c} \in \mathbb{R}$ of the output we compute the gradients of $\boldsymbol{a}$ and $\boldsymbol{b}$ as follows:

$$\begin{bmatrix} \overline{\boldsymbol{a}} \\ \overline{\boldsymbol{b}} \end{bmatrix} = \overline{F}(\boldsymbol{a}, \boldsymbol{b}, \overline{c}) = \begin{bmatrix} \boldsymbol{b}^T & \boldsymbol{a}^T \end{bmatrix}^T \overline{c} = \overline{c} \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{a} \end{bmatrix},$$

which depends on the input values $\boldsymbol{a}$ and $\boldsymbol{b}$.

In (2.14) and (2.15) we deconstructed the loss function $(y, y') \mapsto \frac{1}{2}(y - y')^2$ into three primitive operations. As a consequence the backward pass has a step where we first multiply with 2 and then multiply with $1/2$, this is not efficient of course. Hence deconstructing our calculation into the smallest possible operations is not always advisable. In this example we will express the L^2 loss function as single operation instead.

Example 2.23 (L^2 loss)**.** Let $F : \mathbb{R}^{2n} \to \mathbb{R}$ be given by

$$\ell = F(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{2} \sum_{i=1}^{n} (y_i - y_i')^2$$

for $\boldsymbol{y} = [y_1 \ \cdots \ y_n]^T$ and $\boldsymbol{y}' = [y_1' \ \cdots \ y_n']^T \in \mathbb{R}^n$. Then the Jacobian matrix evaluated at $[\boldsymbol{y} \ \boldsymbol{y}']^T$ is given by

$$J(\boldsymbol{y}, \boldsymbol{y}') = \begin{bmatrix} y_1 - y_1' & \cdots & y_n - y_n' & y_1' - y_1 & \cdots & y_n' - y_n \end{bmatrix} = \begin{bmatrix} (\boldsymbol{y} - \boldsymbol{y}')^T & (\boldsymbol{y}' - \boldsymbol{y})^T \end{bmatrix}.$$

The backward operation has signature $\overline{F} : \mathbb{R}^{2n} \times \mathbb{R} \to \mathbb{R}^{2n}$. The second argument of $\overline{F}$ is a given gradient $\overline{\ell} \in \mathbb{R}$, which is trivially $\overline{\ell} = 1$ if the output ℓ is the final loss we are interested in minimizing. We then compute the gradients as follows:

$$\begin{bmatrix} \overline{\boldsymbol{y}} \\ \overline{\boldsymbol{y}'} \end{bmatrix} = \overline{F}(\boldsymbol{y}, \boldsymbol{y}', \overline{\ell} = 1) = \begin{bmatrix} (\boldsymbol{y} - \boldsymbol{y}')^T & (\boldsymbol{y}' - \boldsymbol{y})^T \end{bmatrix}^T \overline{\ell} = \begin{bmatrix} \boldsymbol{y} - \boldsymbol{y}' \\ \boldsymbol{y}' - \boldsymbol{y} \end{bmatrix}.$$

If $\boldsymbol{y}$ is the output of the neural network and $\boldsymbol{y}'$ is the data point then we are only interested in $\overline{\boldsymbol{y}}$ and we would only compute $\boldsymbol{y} - \boldsymbol{y}'$. This is equivalent to the computation in (2.14) and (2.15) but avoids the redundant multiplications.

2.5 Adaptive Learning Rate Algorithms

The learning rate is crucial to the success of the training process. In general the loss is highly sensitive to some parameters but insensitive to others, just think about parameters in different layers. Momentum alleviates some of the issues but introduces problems of its own and adds another hyperparameter we have to tune.

Assigning each parameters its own learning rate (and continually adjusting it) is not feasible. What we need are automatic methods. This has lead to the development of **adaptive learning (rate) algorithms**. The idea is to set the learning rate dynamically per-parameter at each iteration based on the history of the gradients. We will look at three of these methods: **Adagrad**, **RMSProp** and **Adam**.

Let us recall the setting. We have a parameter space $W = \mathbb{R}^N$ and some initial parameter value $w_0 \in W$. Let I_t be the batch at iteration $t \in \mathbb{N}$ and ℓ_{I_t} its associated loss function. We will abbreviate the current batch's gradient as:

$$g_t := \nabla \ell_{I_t}(w_t).$$

The SGD update rule with learning rate $\eta > 0$ is then:

$$w_{t+1} = w_t - \eta g_t.$$

With momentum the update rule becomes:

$$\begin{aligned} v_t &= \mu v_{t-1} - \eta g_t, \\ w_{t+1} &= w_t + v_t, \end{aligned}$$

where $v_0 = 0$ and $\mu \in [0, 1)$ is the momentum factor.

2.5.1 Adagrad

Adagrad (Adaptive Gradient Descent) was one of the first adaptive learning rate methods introduced by Duchi, Hazan, and Singer (2011).

Let $t \in \mathbb{N}_0$, I_t the batch index set at iteration t and ℓ_{I_t} its associated loss. Let w_0 be the initial parameter values and abbreviate $g_t := \nabla \ell_{I_t}(w_t)$.

The Adagrad update is defined per-parameter as:

$$(w_{t+1})_i = (w_t)_i - \frac{\eta}{\sqrt{\sum_{k=0}^{t} (g_k)_i^2 + \varepsilon}} (g_t)_i.$$

Or when we take all the operations on the vectors to mean component-wise operations we can write more concisely:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\sum_{k=0}^{t} g_k^2 + \varepsilon}} g_t.$$

At each iterations we slow down the effective learning rate of each individual parameter by the L^2 norm of the history of the partial derivatives with respect to that parameter. As $(g_t)_i$ may be very small or zero we add a small positive ε (say 10^{-8} or so) for numerical stability.

The benefit of this method is that choosing η becomes less important, the step size will eventually decrease to the point that it can settle into a local minimum.

On the other hand, the denominator $\sqrt{\sum_{k=0}^{t} (g_k)_i^2}$ is monotonically decreasing and will eventually bring the training process to a halt whether a local minimum has been reach or not.

2.5.2 RMSProp

Root Mean Square Propagation or RMSProp is an adaptive learning rate algorithm introduced by Tieleman, G. Hinton, et al. (2012).

RMSProp uses the same basic idea as Adagrad, but instead of accumulating the squared gradients it uses a weighted average of the historic gradients. Let $\alpha \in (0, 1)$:

$$v_t = \alpha v_{t-1} + (1 - \alpha) g_t^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} g_t,$$

where again all the arithmetic operations are applied component-wise. The α factor is called the **forgetting factor**, **decay rate** or **smoothing factor**. The suggested hyperparameter values to start with are $\eta = 0.001$ and $\alpha = 0.9$.

2.5.3 Adam

The Adaptive Moment Estimation method, or Adam, was introduced by Kingma and Ba (2017). In addition to storing an exponentially decaying average of past square gradients v_t like RMSProp, Adam also keeps a running average of past gradients m_t, similar to momentum.

Let $\beta_1, \beta_1 \in (0, 1)$,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(g_t)^2,$$

with $m_0 = v_0 = 0$. The vectors m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance), hence the name of the method.

As we start with $m_0 = v_0 = 0$ the estimates are biased towards zero. Let us see how biased and whether we can correct that bias. By induction we have:

$$m_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i,$$

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} (g_i)^2,$$

and so:

$$\mathbb{E}[m_t] = (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\mathbb{E}[g_i],$$
$$\mathbb{E}[v_t] = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbb{E}[(g_i)^2].$$

Assuming $\mathbb{E}[g_i]$ and $\mathbb{E}[(g_i)^2]$ are stationary (i.e. do not depend on i) we get:

$$\mathbb{E}[m_t] = \left((1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\right)\mathbb{E}[g_t],$$
$$\mathbb{E}[v_t] = \left((1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\right)\mathbb{E}[(g_t)^2].$$

Which simplifies to:

$$\mathbb{E}[m_t] = (1-\beta_1^t)\mathbb{E}[g_t],$$
$$\mathbb{E}[v_t] = (1-\beta_2^t)\mathbb{E}[(g_t)^2].$$

Therefore, the bias-corrected moments are:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

The update rule of Adam is then given by:

$$w_{t+1} = w_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon},$$

where again all arithmetic is applied component-wise.

The default hyperparameter values suggested by Kingma and Ba (2017) are $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\varepsilon = 10^{-8}$.

2.5.4 Which variant to use?

Which of these algorithms should you use? If your data is sparse (very high dimensional data usually is) then the adaptive learning rate methods usually outperform plain SGD (with or without momentum). Kingma and Ba (2017) showed that due to its bias correction Adam performs marginally better late in the training process. In that regard Adam is a safe option to try first. Nonetheless trying different algorithms to see which one works best for any given problem can be worthwhile.

Fig. 2.12 illustrates the different behaviour of the methods we have discussed in some artificial loss landscapes. These landscapes model some of the problems the algorithms may encounter and show some of the strengths and weaknesses of each method.

There are many more SGD variants we have not discussed, you can look at the `toch.optim` namespace in the PyTorch documentation to see the available algorithms.

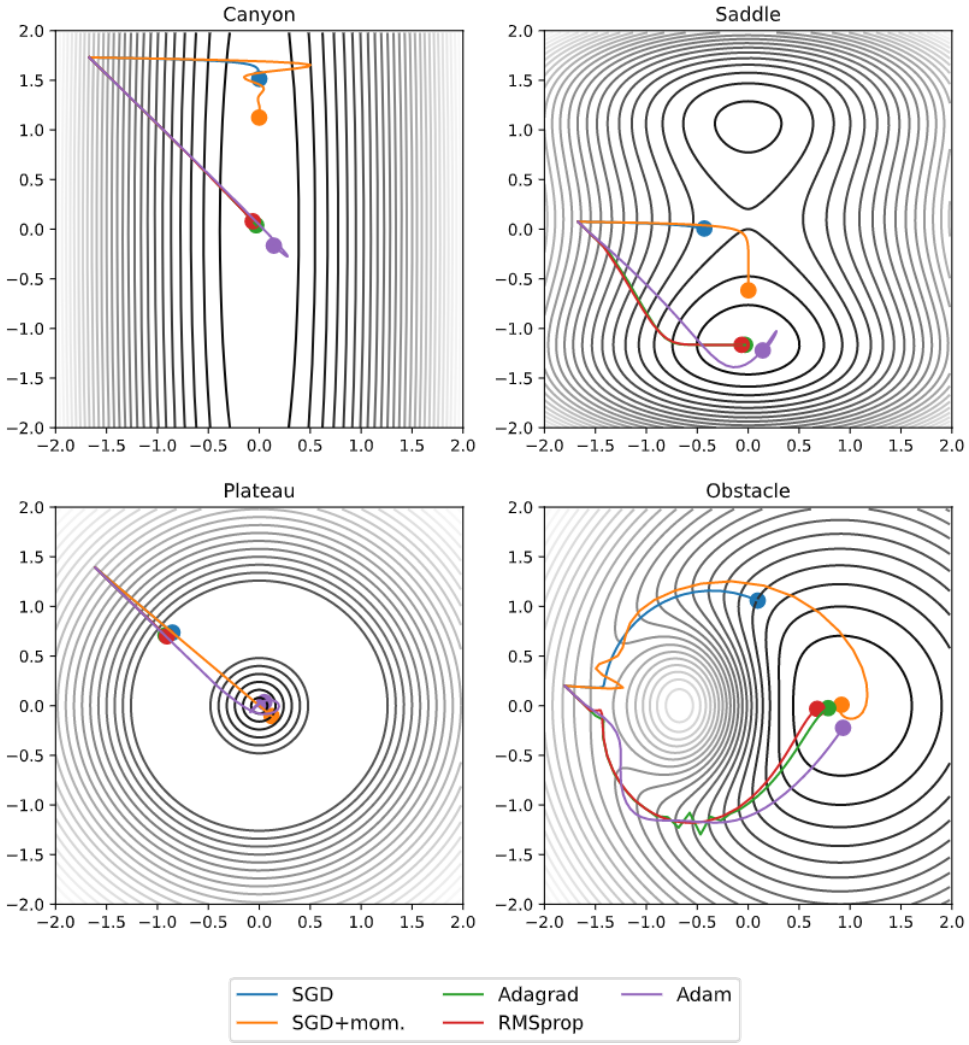


Figure 2.12: Comparing 5 SGD variants in 4 different situations for a number of steps. In the *canyon* situation we have one parameters that has a large effect on the loss and one parameter that has little effect on the loss. The *saddle* situation has a prominent saddle point. The *plateau* has a large flat section with a vanishing gradient that needs to be traversed to get to the minimum. The final situation has an *obstacle* the algorithm has to go around to reach the minimum. Dark level sets indicate lower function values. This figure was generated with `SGDDemonstration.ipynb`.

Chapter 3

Equivariance

There are many applications where we want the neural network to have certain symmetries, such as in Fig. 3.1. Most applications come with natural symmetries. It might be rotation-translation invariance for a medical diagnosis application that detects tumors in X-ray images or time invariance for a weather forecasting system.

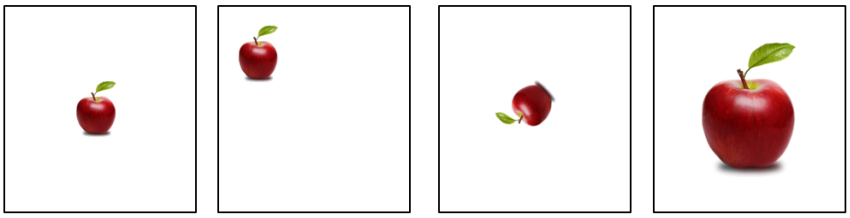


Figure 3.1: We would like a classification network to be invariant under translation, rotation and scaling. We could train our network with translated, rotated and scaled versions of our original data and hope the invariance gets encoded that way. But this would drastically increase the training time and gives no guarantee of success. Preferable would be designing the network in such a way that it is intrinsically invariant.

The desired symmetry might not be an actual invariance in that the output is not expected to remain the same but instead transform in some manner similar to the input. For example: in an image enhancement application the output image is expected to rotate and translate along with the input image. Some authors use **invariance** to denote both cases, we will use the term **equivariance** (as in: *transforms with*) to refer to both and use invariance for the special case where the output should stay the same.

Expressing many of these symmetries on discrete domains is awkward, indeed rotations and translations of an image are not even well defined unless the translation is on grid or the rotation is in 90° increments. So instead we will be working in the continuous setting and only discretizing when it becomes time to implement our ideas. For images, instead of representing them as elements of $\mathbb{R}^{H\times W}$ we represent them as elements of $C_c^\infty(\mathbb{R}^2)$, i.e. smooth functions with compact support.

Our eventual practical goal for this chapter is building a type of CNN that is not only translation

equivariant but also rotation equivariant. But being mathematicians we want to be general and develop a theory that allows for other transformations as well. This brings us to the theory of Lie groups, which are essentially continuous transformation groups, something we will make precise in this chapter.

Lie group theory plays an important role in many disparate fields of study such as geometric image analysis, computer vision, particle physics, financial mathematics, robotics, etc. In the next sections we will build up a general equivariance framework based on Lie groups. The payoff of this theoretical work will be a general recipe for building a neural network that is equivariant with respect to an arbitrary transformation group: a so called Group Equivariant CNN (Cohen, Geiger, and Weiler 2020; Cohen and Welling 2016), or **G-CNN** for short.

3.1 Manifolds

We start with the basic object we will be working with: the manifold. We are accustomed to doing analysis on $\mathbb{R}^n$ but we also know that there are non-Euclidean spaces of importance. Classic example is the unit circle S^1, which is distinctly non-Euclidean but still admits derivatives, integrals, PDEs, etc. When working with an object such as S^1 we usually do it indirectly by parameterizing it some way, such as with an angle $\theta \in [0, 2\pi)$, so that at least locally it resembles $\mathbb{R}$. We can generalize this and consider all spaces that we can, at least locally, identify with a subset of $\mathbb{R}^n$.

3.1.1 Characterization

The proper way to introduce manifolds is starting with a set and then adding several layers of structure, as is done in Lee (2010, 2013). After going through that laborious process we would then see a characterization that is a more practical tool for constructing manifolds. In this course we will skip straight to that characterization in the form of the following lemma.

Lemma 3.1 (Smooth manifold chart lemma)**.** Let M be a set and suppose we are given a collection of subsets $\{U_\alpha\}_{\alpha \in I}$ of M for some index set I, together with maps $\varphi_\alpha : U_\alpha \to \mathbb{R}^n$. Then M together with $\{(U_\alpha, \varphi_\alpha)\}_{\alpha \in I}$ gives a smooth n-dimensional manifold if the following conditions are satisfied.

(i) For all $\alpha \in I$, φ_α is a bijection between U_α and an open subset of $\mathbb{R}^n$.

(ii) For each $\alpha, \beta \in I$, the sets $\varphi_\alpha\left(U_\alpha \cap U_\beta\right)$ and $\varphi_\beta\left(U_\alpha \cap U_\beta\right)$ are open in $\mathbb{R}^n$.

(iii) When $U_\alpha \cap U_\beta \neq \varnothing$ then the map $\tau_{\alpha,\beta} := \varphi_\beta \circ \varphi_\alpha^{-1} : \varphi_\alpha(U_\alpha \cap U_\beta) \to \varphi_\beta(U_\alpha \cap U_\beta)$ is smooth.

(iv) There exists an (at most) countably infinite $J \subset I$ so that $\bigcup_{\alpha \in J} U_\alpha = M$, i.e. there exists a countable cover of M.

(v) Whenever p_1 and p_2 are distinct point in M, there exist U_α and U_β (not necessarily distinct) so that there exist disjoint sets $V \subset U_\alpha$ and $W \subset U_\beta$, with $\varphi_\alpha(V)$ and $\varphi_\beta(W)$ open in $\mathbb{R}^n$, with $p_1 \in V$ and $p_2 \in W$.

We say that each pair $(U_\alpha, \varphi_\alpha)$ is a (smooth) **chart** and that the set $\{(U_\alpha, \varphi_\alpha)\}_{\alpha \in I}$ is a (smooth) **atlas** of M. The maps $\tau_{\alpha,\beta} := \varphi_\beta \circ \varphi_\alpha^{-1}$ are called the atlas' **transition maps**. Since the transition maps are from $\mathbb{R}^n$ to $\mathbb{R}^n$ we know exactly what it means for these maps to be smooth. Charts

that have smooth transition maps both ways are said to be **compatible**. This construction is illustrated in Fig. 3.2, for details and proof see Lee (2013, Ch. 1).

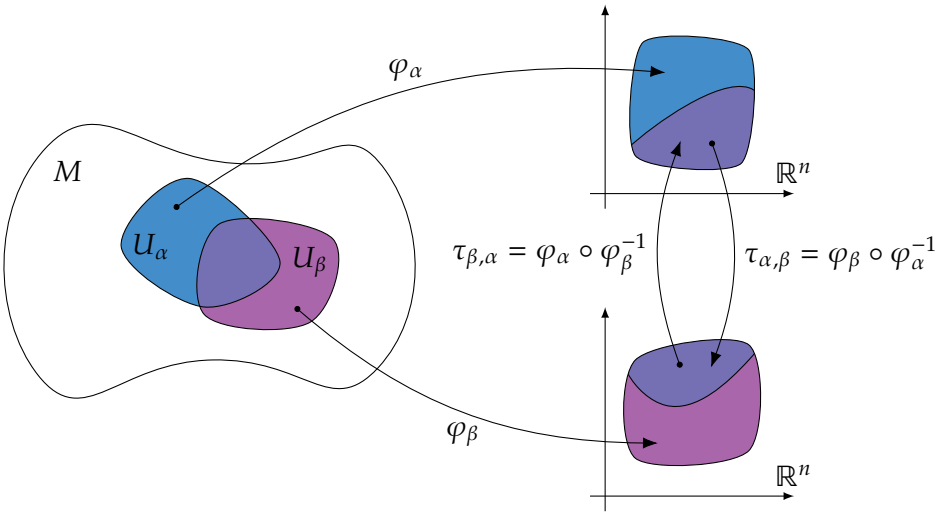


Figure 3.2: A manifold M with two charts $(U_\alpha, \varphi_\alpha)$ and (U_β, φ_β) and their transitions maps $\tau_{\alpha,\beta}$ and $\tau_{\beta,\alpha}$.

Remark 3.2 (The manifold is not the atlas). In Lemma 3.1 we say that the set M along with an atlas $\{(U_\alpha, \varphi_\alpha)\}_{\alpha \in I}$ **gives** a smooth manifold rather than stating that it **is** a smooth manifold. This is because we could construct another atlas compatible with the first but the set together with the new atlas would still describe the same manifold. In example 3.5 this distinction will be clear as we can see that we can construct any number of atlases describing the same manifold.
We say two atlases are compatible if the transition maps between charts of the two atlases are also smooth, if two atlases are compatible they describe the same manifold. Formally we could define a smooth manifold as the equivalence class of all atlases per Lemma 3.1 under the equivalence relation of the atlases being compatible.
We could also consider the union of all possible compatible atlases, called the **maximal atlas** as defining the manifold.

Remark 3.3 (Hausdorff space). Item *(v)* of Lemma 3.1 guarantees that a manifold is a Hausdorff space, this simply means that for any two distinct points p_1 and p_2 we can find neighborhoods of both that are disjoint (a neighborhood in the manifold being a preimage of a neighborhood in a chart). This may seem redundant but one can construct counterexamples that satisfy *(i)-(iv)* but not *(v)*. The Hausdorff property is needed for limits to be unique and manifolds are required to be Hausdorff for that reason.

We define what the **open subsets** of the manifold are by looking at their images in $\mathbb{R}^n$. A subset $V \subset M$ is open if $\varphi_\alpha(V \cap U_\alpha)$ is open in $\mathbb{R}^n$ for all $\alpha \in I$. In other words: we let the atlas and the standard topology on $\mathbb{R}^n$ define the topology of the manifold. With this definition we could rephrase item *(v)* of Lemma 3.1 as: for any two distinct points of M there exist **neighborhoods** (i.e. open subsets containing the point in question) of said two points that are disjoint.

Of course if $(U_\alpha, \varphi_\alpha)$ is a chart then if $V \subset U_\alpha$ is open then $(V, \varphi_\alpha|_V)$ is also a chart.

Remark 3.4 (Open sets and continuous charts). Defining the open sets of M as the preimages of open subsets of $\mathbb{R}^n$ makes the charts continuous maps by definition. It might be the case that when we are constructing a manifold we do not start with M being just a set. For example, in many cases we start with M being a subset of $\mathbb{R}^n$, in that case we already know what the open subsets of M are (i.e. the topology of M is already known). If we already decided what the open subsets of M are going to be then we need to make sure that the two notions of 'open' coincide. This requires that the charts are continuous maps by construction instead of them being continuous by definition.

Lemma 3.1 is a characterization so it works the other way around as well. If M is a smooth manifold then it is always possible to produce a smooth atlas, i.e. a set $\{(U_\alpha, \varphi_\alpha)\}_{\alpha\in I}$ that satisfies all the conditions of the lemma.

Example 3.5 (The unit circle)**.** Let S^1 be the unit circle in $\mathbb{R}^2$, i.e. $S^1 = \left\{(x,y) \,\middle|\, x^2+y^2=1\right\}$. Then we can construct two smooth charts that cover S^1:

$$\begin{aligned} U_1 &= S^1 \setminus \{(-1,0)\}\,, \qquad \varphi_1(x,y) : U_1 \to (-\pi,\pi), \\ U_2 &= S^1 \setminus \{(1,0)\}\,, \qquad \varphi_2(x,y) : U_2 \to (0,2\pi). \end{aligned}$$

Where φ_1 gives the angle between the x-axis and (x,y) measured from $-\pi$ to π and φ_2 gives that same angle measured from 0 to 2π, as illustrated in Figure 3.3.

Clearly $U_1 \cup U_2 = S^1$ and we see that $\varphi_1(U_1) = (-\pi,\pi)$ and $\varphi_2(U_2) = (0,2\pi)$, so these two charts form an atlas of S^1. Now we have to check the transition maps, note that $U_1 \cap U_2 = S^1 \setminus \{(-1,0),(1,0)\}$ which consists of two disjoint components, so the transition map from the first to the second chart would have as domain $(-\pi,0)\cup(0,\pi)$ and be defined as:

$$\tau_{1,2}(\theta) = \varphi_2 \circ \varphi_1^{-1}(\theta) = \begin{cases} \theta & \text{if } \theta \in (0,\pi), \\ \theta + 2\pi & \text{if } \theta \in (-\pi,0), \end{cases}$$

which is smooth on its domain (the discontinuity at $\theta = 0$ is not part of the domain). The inverted transition map is similarly smooth. The whole construction is visualized in Fig. 3.3.

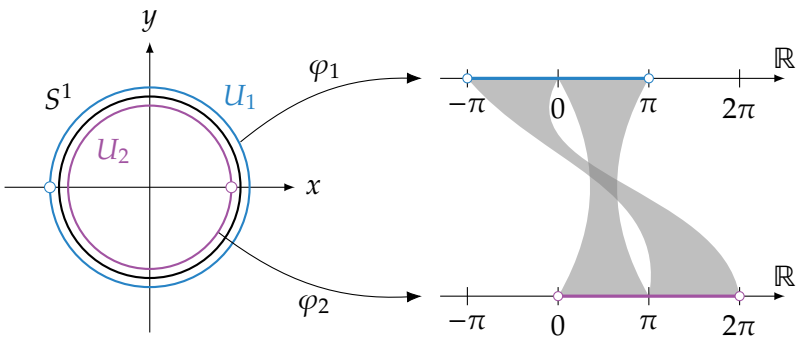


Figure 3.3: A smooth atlas on S^1 with the transition maps illustrated in grey.

3.1.2 Smooth Maps

An important reason for introducing smooth manifolds is being able to talk about smooth maps. While the terms **map** and **function** are technically interchangeable we will use the term

function for maps whose co-domain is $\mathbb{R}$ or $\mathbb{R}^k$ and use **map** for more general maps between manifolds.

Definition 3.6 (Smooth map)**.** Let M and N be two smooth manifolds and let $F : M \to N$ be any map. We say F is a **smooth map** if for every $p \in M$ there exists a smooth chart (U, φ) of M that contains p and a smooth chart (V, ψ) of N that contains $F(p)$ so that $F(U) \subseteq V$ and the map $\psi \circ F \circ \varphi^{-1}$ is smooth from $\varphi(U)$ to $\psi(F(U)) \subseteq \psi(V)$.

Due to the smooth structure (i.e. all of its possible atlases) that a smooth manifold is equipped with this definition is independent of the choice of charts, the smoothness of transition maps ensures this key property. The definition makes it clear that we can only talk about the smoothness of maps if the manifolds in question have smooth structures, so when we say $F : M \to N$ is a smooth map we imply that M and N are smooth manifolds even if we do not specify that fact explicitly. The construction from the definition is illustrated in Fig. 3.4.

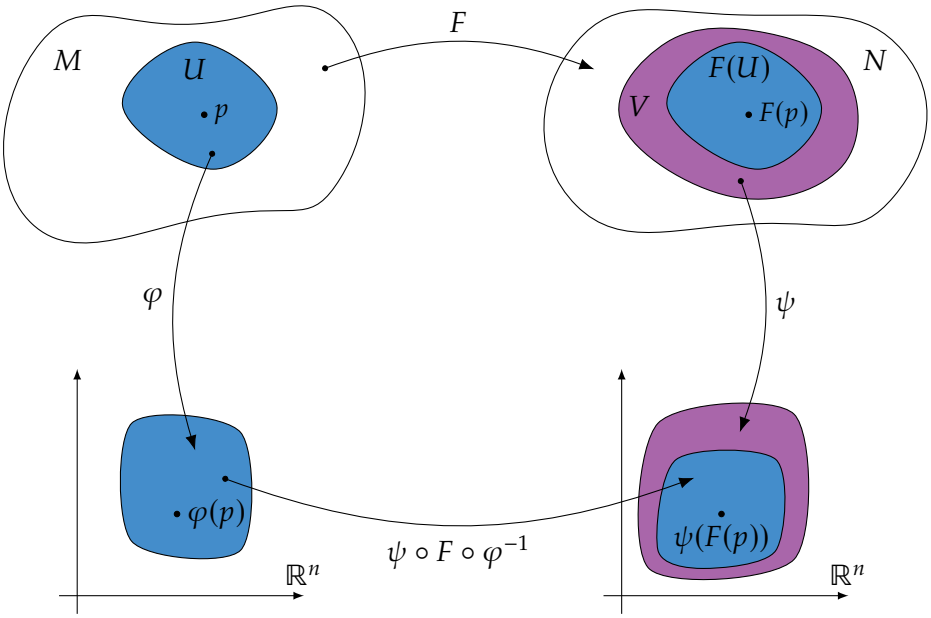


Figure 3.4: A smooth map F between two manifolds and its representation $\psi \circ F \circ \varphi^{-1}$ between chart co-domains.

With **smooth function** we now just mean a smooth map from M to $\mathbb{R}$ or $\mathbb{R}^k$. Another special case for which we reserve its own term is that of a **smooth curve**: a smooth map from $\mathbb{R}$ (or some interval of $\mathbb{R}$) to another smooth manifold.

Example 3.7. The map $F : \mathbb{R} \to S^1$ given by $F(\theta) := (\cos\theta, \sin\theta)$ is a smooth map from the manifold $\mathbb{R}$ to the manifold S^1.

Definition 3.8 (Diffeomorphism)**.** Let M and N be smooth manifolds, a smooth bijective map from M to N that has a smooth inverse is called a **diffeomorphism**. Two manifolds between which a diffeomorphism exists are said to be **diffeomorphic**.

Example 3.9. The unit circle S^1 is not diffeomorphic with $\mathbb{R}$ since there exists no continuous bijection between the two.

The unit circle is however diffeomorphic with the group $SO(2)$ (i.e. the group of orthogonal 2×2 matrices with determinant 1) via the identification

$$S^1 \ni (x, y) \leftrightarrow \begin{bmatrix} x & -y \\ y & x \end{bmatrix} \in SO(2),$$

usually parametrized with $\theta \in \mathbb{R}/(2\pi\mathbb{Z})$ as

$$S^1 \ni (\cos\theta, \sin\theta) \leftrightarrow \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \in SO(2).$$

3.2 Lie Groups

3.2.1 Basic Definitions

We assume that the transformations we are interested in form Lie groups. Classical groups such as the general and special linear groups (the groups of matrices that are invertible resp. have determinant 1), the orthogonal group (the group of orthogonal matrices), etc. are examples of Lie groups.

Definition 3.10. A **Lie group** G is a smooth manifold so that G is also an algebraic group given by two smooth maps, one for the group product (also called multiplication):

$$G \times G \to G, \quad (g_1, g_2) \mapsto g_1 g_2,$$

and one for inversion:

$$G \to G, \qquad g \mapsto g^{-1}.$$

Recall that being a group, a Lie group has the following properties.

- **Closure**: $\forall g_1, g_2 \in G : g_1 g_2 \in G$.
- **Associativity**: $\forall g_1, g_2, g_3 \in G : (g_1 g_2) g_3 = g_1 (g_2 g_3)$.
- **Unit element**: $\exists e \in G, \forall g \in G : eg = ge = g$ and this element e is unique. We use the traditional e to denote the unit element, which derives from the German **Einselement**.
- **Inverse**: $\forall g \in G \; \exists g^{-1} \in G : g g^{-1} = g^{-1} g = e$.

We should emphasize that a group need not be commutative, indeed the particular Lie groups we are most interested in are not commutative and have group elements g_1, g_2 for which $g_1 g_2 \neq g_2 g_1$.

For $g \in G$, we denote by $L_g : G \to G, L_g(h) = gh$ the **left multiplication** by g and by $R_g : G \to G$, $R_g(h) = hg$ the **right multiplication** by g. Left and right multiplication are also sometimes called left and right translation.

Example 3.11. $\mathbb{R}^n$ is a (commutative) Lie group under vector addition $(\boldsymbol{x}, \boldsymbol{y}) \mapsto \boldsymbol{x} + \boldsymbol{y}$ and negation $\boldsymbol{x} \mapsto -\boldsymbol{x}$.

Example 3.12 (Multiplicative group of positive real numbers)**.** $\mathbb{R}_{>0}$ is a (commutative) Lie group under multiplication $(x, y) \mapsto xy$ and inversion $x \mapsto 1/x$.

Example 3.13 (General linear group)**.** The **general linear group** of degree n, $GL(n)$, is the group of all invertible $n \times n$ matrices. The group product is the matrix product.

Example 3.14 (Special orthogonal group)**.** The **special orthogonal group** of degree n, $SO(n)$, is the subgroup of $GL(n)$ of all matrices with determinant 1. It is the group of rotations in n dimensions. For $SO(2)$ we write the matrices in terms of the angle of the rotation $\theta \in \mathbb{R}/(2\pi\mathbb{Z})$ as

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{3.1}$$

with the group law given by $R(\theta_1)R(\theta_2) = R(\theta_1 + \theta_2)$.

Example 3.15 (Special Euclidean group)**.** The **special Euclidean group** of degree n, $SE(n)$, is the group of rotations and translations in n dimensions. As a set it equals $\mathbb{R}^n \times SO(n)$ with the group law given by:

$$(\boldsymbol{x}_1, R_1)\,(\boldsymbol{x}_2, R_2) = (\boldsymbol{x}_1 + R_1\boldsymbol{x}_2,\ R_1R_2)\,. \tag{3.2}$$

The group law is not the direct product (i.e. not $(\boldsymbol{x}_1 + \boldsymbol{x}_2,\ R_1R_2)$) but the second group affects the law of the first group. We call this a **semidirect product**, to emphasize this difference we write $SE(n) = \mathbb{R}^n \rtimes SO(n)$.

Since we want to design a neural network that takes in 2D images and is rotation-translation equivariant, the Lie group $SE(2)$ is of special interest to us. In the $SE(2)$ case we can either represent elements as $(\boldsymbol{x}, R(\theta)) \in \mathbb{R}^2 \times SO(2)$ with $R(\theta)$ the rotation matrix from (3.1), or as $(\boldsymbol{x}, \theta) \in \mathbb{R}^2 \times [0, 2\pi)$. In the latter case the group law can be written as

$$(\boldsymbol{x}_1, \theta_1)(\boldsymbol{x}_2, \theta_2) = (\boldsymbol{x}_1 + R(\theta_1)\boldsymbol{x}_2,\ \theta_1 + \theta_2 \bmod 2\pi)\,.$$

3.2.2 Lie Subgroups

Algebraic groups can have subgroups (think of the many subgroups of the general linear group). Lie groups also have subgroups but those subgroups are not automatically Lie groups themselves. Let G be a Lie group, then $H \subset G$ is a **Lie subgroup** of G if

(i) H is a subgroup of the group G,

(ii) H is an immersed submanifold of the manifold G,

(iii) the group operations on H are smooth.

We have not seen what an immersed submanifold is and this is beyond the scope of this course. However the following theorem provides an easier way of identifying most Lie subgroups.

Theorem 3.16 (Cartan's closed subgroup theorem)**.** Any subgroup of a Lie group that is closed (as a set) is a Lie subgroup.

See Lee (2013, Ch. 7) for proof and details.

Not all Lie subgroups are closed but the ones we are interested in all are. Consequently Theorem 3.16 is our go-to method for proving whether a subgroup is a Lie group.

Example 3.17. Every linear subspace of $\mathbb{R}^n$ is a (closed) Lie subgroup under vector addition.

Example 3.18. The groups $\mathbb{R}^n \times \{0\}$ and $\{\mathbf{0}\} \times SO(n)$ are (closed) Lie subgroups of $SE(n)$.

3.2.3 Group Actions

The most important use of Lie groups in manifold theory involves the action of a Lie group on a manifold.

Definition 3.19 (Group action)**.** If G is a group and M a set then a **left action** of G on M is a map $G \times M \to M$ written as $(g, p) \mapsto g \cdot p$ that satisfies:

$$\begin{aligned} g_2 \cdot (g_1 \cdot p) &= (g_2 g_1) \cdot p \qquad && \forall g_1, g_2 \in G, p \in M, \\ e \cdot p &= p && \forall p \in M. \end{aligned} \tag{3.3}$$

A **right action** is defined similarly as a map $M \times G \to M$ that satisfies:

$$\begin{aligned} (p \cdot g_1) \cdot g_2 &= p \cdot (g_1 g_2) \qquad && \forall g_1, g_2 \in G, p \in M, \\ p \cdot e &= p && \forall p \in M. \end{aligned}$$

If both G and M are manifolds and the map is smooth in both inputs G and M then we say the action is a **smooth action**.

We will focus exclusively on left actions since their group law (3.3) has the property that group multiplication corresponds to map composition. In any case a right action can always be converted to a left action by defining $g \cdot p := p \cdot g^{-1}$, or vice versa for turning a left action into a right action.

In our setting G is always a Lie group and M always a manifold and we will only be considering smooth actions.

Sometimes it is convenient to label an action, say $\rho : G \times M \to M$. The action of a group element g on a point p can then be written equivalently as $g \cdot p \equiv \rho(g, p) \equiv \rho_g(p)$.

If ρ is a smooth action then for all $g \in G$, $\rho_g : M \to M$ is a diffeomorphism since $\rho_{g^{-1}}$ is a smooth inverse.

A group action on a manifold induces an action on any function space on that manifold in a straightforward manner. Let X be a function space on M, such as $C^k(M)$ or $L^p(M)$, then the mapping $\rho^X : G \times X \to X$, defined by

$$\left(\rho^X(g, f)\right)(p) = f(g^{-1} \cdot p) = f(\rho(g^{-1}, p)), \tag{3.4}$$

for all $f \in X$ and $g \in G$ is a (left) action. We can verify that with

$$\begin{aligned} &\rho^X\left(g_2, \rho^X(g_1, f)\right)(p) \\ &= \rho^X(g_1, f)(g_2^{-1} \cdot p) \\ &= f\left(g_1^{-1} \cdot (g_2^{-1} \cdot p)\right) \\ &= f\left((g_1^{-1} g_2^{-1}) \cdot p\right) \\ &= f\left((g_2 g_1)^{-1} \cdot p\right) \\ &= \rho^X(g_2 g_1, f)(p). \end{aligned}$$

This action on function spaces has the additional property that it is linear in the second argument. We call actions with this property representations of the Lie groups.

Definition 3.20 (Lie group representation)**.** Let G be a Lie group and V a vector space (finite dimensional or not) then $\nu : G \to \mathrm{Aut}(V)$ is a **representation** of G if it is a smooth homomorphism, i.e. is smooth and

$$\nu(g_1 g_2) = \nu(g_1) \circ \nu(g_2) \qquad \forall g_1, g_2 \in G.$$

Recall that the automorphism group $\mathrm{Aut}(V)$ is the group of invertible linear transformations of V.

It follows from the definition that a representation ν also has the following properties:

$$\nu(e) = \mathrm{id}_V \quad \text{and} \quad \nu(g^{-1}) = \nu(g)^{-1}.$$

Since we usually only have one group action per manifold, and so one corresponding representation on a given function space we can overload the meaning of the "$\cdot$" symbol and use the following equivalent notations:

$$g \cdot f := \rho_g^X(f) := \rho^X(g, f).$$

This is how we are going to be modeling transformation acting on our data. In our rotation-translation case f would be an input image on $\mathbb{R}^2$ and $g \in SE(2)$ would be a rotation-translation acting on the image.

3.2.4 Equivariant Maps and Operators

Suppose G is a Lie group and M and N are smooth manifolds with smooth (left) actions ρ^M and ρ^N. Then we can consider maps $F : M \to N$ that are **equivariant** with respect to those group actions, i.e.

$$F(\rho^M(g, p)) = \rho^N(g, F(p))$$

for all $g \in G$ and $p \in M$, or more concisely:

$$F(g \cdot p) = g \cdot F(p).$$

Equivalently, F is equivariant if the following diagram commutes for each $g \in G$:

$$\begin{array}{ccc} M & \xrightarrow{F} & N \\ {\scriptstyle \rho_g^M}\downarrow & & \downarrow{\scriptstyle \rho_g^N} \\ M & \xrightarrow{F} & N. \end{array}$$

This idea extends naturally to operators between function spaces on those manifolds. Let X be a function space on M and Y a function space on N equipped with the corresponding representations ρ^X and ρ^Y per (3.4). Then an operator $A : X \to Y$ is equivariant if

$$A \circ \rho_g^X = \rho_g^Y \circ A, \qquad \forall g \in G. \tag{3.5}$$

Or in words: for every group element doing the corresponding transform on the input space X and then applying the operator A gives the same results as first applying the operator A and then performing the transform corresponding to the group element on the output space Y.

Our goal in the continuous setting is designing our neural network as an equivariant operator that satisfies (3.5).

3.2.5 Homogeneous Spaces

While Lie groups represent the transformations we are interested in, homogeneous spaces are the spaces our data will live on and on which the Lie groups will act.

Definition 3.21 (Homogeneous space)**.** A smooth manifold M is a **homogeneous space** of a Lie group G if there exists a smooth (left) action $\rho : G \times M \to M$ that is **transitive**, i.e.

$$\forall p_1, p_2 \in M \; \exists g \in G : \rho(g, p_1) = g \cdot p_1 = p_2.$$

The elements of M are called the points of the homogeneous space and G is called the motion group or the fundamental group of the homogeneous space. The transitive property can be reformulated as: for every point in M there is a g that takes us to any other point in M.

Observe that G is a homogeneous space of itself, called the **principal homogeneous space**. The group action is just the left multiplication, i.e. $\rho_g = L_g$.

On the other end we have the **trivial homogeneous space** consisting of a single element $\{0\}$, which is a homogeneous space of every Lie group under the identity action $\rho(g, 0) = 0$.

Remark 3.22 (Zero-dimensional manifolds). You may wonder whether $\{0\}$ is a manifold. In fact all (at most) countable sets S are 0-dimensional manifolds. Assign each point $p \in S$ its own unique chart $\varphi_p : \{p\} \to \mathbb{R}^0 = \{0\}$. Since these chart domains do not overlap the charts are trivially smoothly compatible and form a unique smooth atlas.

For each $p \in M$, the **stabilizer** or the **isotropy group** of p is the subset G_p of G (also denoted by $\mathrm{Stab}_G(p)$) that fixes p:

$$G_p := \mathrm{Stab}_G(p) := \{g \in G \mid g \cdot p = \rho(g, p) = p\}. \tag{3.6}$$

If we have $g_1, g_2 \in G_p$ then $(g_1 g_2) \cdot p = g_1 \cdot (g_2 \cdot p) = g_1 \cdot p = p$. So $g_1 g_2 \in G_p$, meaning that G_p is a subgroup of G. Moreover since the group action is smooth it follows that if $(g_n)_{n\in\mathbb{N}}$ is a sequence in G_p with $\lim_{n\to\infty} g_n = g \in G$ then

$$\rho(g, p) = \rho\left(\lim_{n\to\infty} g_n, p\right) = \lim_{n\to\infty} \rho(g_n, p) = p.$$

From which we conclude that $g \in G_p$ and so G_p is closed and consequently by Theorem 3.16, G_p is a Lie subgroup of G for all $p \in M$.

When we pick a **reference element** $p_0 \in M$, we can define the subset $G_{p_0,p} \subset G$ of all group elements that map p_0 to p:

$$G_{p_0,p} := \{g \in G \mid g \cdot p_0 = p\}. \tag{3.7}$$

Note that this is generally not a subgroup, except for $G_{p_0,p_0} = G_{p_0}$. If we have two group elements that map p_0 to the same p, i.e. $g_1, g_2 \in G_{p_0,p}$ then

$$g_1 \cdot p_0 = g_2 \cdot p_0 \quad \Leftrightarrow \quad g_1^{-1} g_2 \cdot p_0 = p_0 \quad \Leftrightarrow \quad g_1^{-1} g_2 \in G_{p_0}. \tag{3.8}$$

This condition imposes an equivalence relation on G, which we can quotient out as follows:

$$G/G_{p_0} := \left\{s \subset G \,\middle|\, \forall g_1, g_2 \in s : g_1^{-1} g_2 \in G_{p_0}\right\} = \left\{G_{p_0,p} \,\middle|\, \forall p \in M\right\}.$$

There is a straightforward isomorphism between M and G/G_{p_0} given by $p \mapsto G_{p_0,p}$ and $G_{p_0,p} \mapsto G_{p_0,p} \cdot p_0$.

From which we can conclude that all homogeneous spaces are isomorphic to a Lie group quotient G/H for some closed Lie subgroup H of G. For this reason many authors blur the line between a homogeneous space and its corresponding group quotient and effectively equate a point of the homogeneous space with its corresponding equivalence class in the group, i.e. $p \equiv G_{p_0,p}$ after fixing a $p_0 \in M$. This leads to concise notation such as $g \in p \Leftrightarrow g \cdot p_0 = p$ and the dropping of the '$\cdot$' notation since if p is seen as a subset of G then $g \cdot p \equiv gp$.

3.3 Linear Operators

Now let us look at how we can start putting together an artificial neuron in our new setting. We have an input manifold M and an output manifold N that are both homogeneous spaces of a Lie group G. Our input data is a function on M, say $f \in X = B(M)$ and we are expected to output a function on N, say an element of $Y = B(N)$. Recall that the set of bounded functions $B(M)$ is a Banach space under the supremum norm (a.k.a. the ∞-norm or the uniform norm) given by $\|f\|_\infty := \sup_{p\in M} |f(p)|$.

The first part of a discrete artificial neuron was a linear operator $A : \mathbb{R}^n \to \mathbb{R}^m$, given by:

$$(A\boldsymbol{x})_i = \sum_j (A)_{ij} x_i.$$

Its analogue in the continuous setting is an **integral operator** $A : \mathbb{R}^M \to \mathbb{R}^N$ of the form:

$$(Af)(q) = \int_M k_A(p,q)\, f(p)\, dp, \tag{3.9}$$

where the function $k_A : M \times N \to \mathbb{R}$ is called the operator's **kernel**.

> *Remark* 3.23 (Measurable functions). Technically for the Lebesgue integral in (3.9) to exist the integrand needs to be measurable. We will not be dealing with non-measurable functions and you may assume that when we say function we mean measurable function. If the concept of measurable functions is new to you, you may ignore the issue.

In this framework, instead of training the matrix A, we will train the kernel k_A. In practice we cannot train a continuous function so training the kernel will come down to either training a discretization or training the parameters of some parameterization of k_A.

Now we still need to specify how we are going to integrate on a homogeneous space to make progress.

3.3.1 Integration

Integration on $\mathbb{R}^n$ has the desirable property that it is translation invariant: for all $\boldsymbol{y} \in \mathbb{R}^n$ and integrable functions $f : \mathbb{R}^n \to \mathbb{R}$ we have

$$\int_{\mathbb{R}^n} f(\boldsymbol{x} - \boldsymbol{y})\, d\boldsymbol{x} = \int_{\mathbb{R}^n} f(\boldsymbol{x})\, d\boldsymbol{x}. \tag{3.10}$$

Ideally we would want integration on a homogeneous space M of a Lie group G to behave similarly, namely for all $g \in G$ we would like:

$$\int_M (g \cdot f)(p)\, d\mu_M(p) := \int_M f(g^{-1} \cdot p)\, d\mu_M(p) = \int_M f(p)\, d\mu_M(p), \tag{3.11}$$

for some Radon measure μ_M on M.

Remark 3.24 (Measures). Recall that measures are the generalization of concepts such as length, volume, mass, probability etc. A measure assigns a non-negative real number to subsets of a space in such a way that it behaves similarly to the aforementioned concepts. A Radon measure on a Hausdorff topological space is a measure that plays well with the topology of the space (defined for open and closed sets, finite on compact sets, etc.). The Lebesgue measure is the translation invariant Radon measure on $\mathbb{R}^n$ and coincides with our less general notion of the length/area/volume of subsets of $\mathbb{R}^n$. Integration on $\mathbb{R}^n$ such as in (3.10) implicitly uses the Lebesgue measure and so is translation invariant. For a comprehensive introduction to measure theory see Tao (2011). For the purpose of this course it is sufficient to think about a measure as measuring the volume of a subset.

This imposes a condition on the measure μ_M, namely: for all measurable subsets S of M and $g \in G$ we require

$$\mu_M(g \cdot S) = \mu_M(S). \tag{3.12}$$

In other words we would need a (non-zero) group invariant measure to get the desired integral. These G-invariant measures, or just **invariant measures**, do not always exist. In some cases we can still obtain a **covariant measure**, which is a measure that satisfies

$$\mu(g \cdot S) = \chi(g)\,\mu(S), \tag{3.13}$$

where $\chi : G \to \mathbb{R}^+$ is a character of G.

Definition 3.25 (Character)**.** A multiplicative character or linear character or simply **character** of a Lie group G is a continuous homomorphism from the group to the multiplicative group of positive real numbers, i.e. $\chi : G \to \mathbb{R}_{>0}$ so that:

$$\chi(g_1 g_2) = \chi(g_1)\,\chi(g_2) \qquad \forall g_1, g_2 \in G.$$

The function χ needs to be a character since by (3.13) we have:

$$\chi(g_1 g_2)\,\mu(S) = \mu(g_1 g_2 \cdot S) = \mu(g_1 \cdot (g_2 \cdot S)) = \chi(g_1)\,\mu(g_2 \cdot S) = \chi(g_1)\,\chi(g_2)\,\mu(S),$$

for all $g_1, g_2 \in G$ and all measurable $S \subset M$.

If we integrate with respect to a G-invariant measure we say we have a G-invariant integral, or just an **invariant integral**, if the measure is covariant with a character χ we say we have a χ-covariant integral or just **covariant integral**.

Definition 3.26 (Covariant integral)**.** Let M be a homogeneous space of a Lie group G, we say the integral $\int_M \ldots dp$ (using some Radon measure on M) is **covariant** with respect to G if there exists a character χ_M of G so that

$$\int_M (g \cdot f)(p)\; dp = \chi_M(g) \int_M f(p)\; dp$$

for all $g \in G$ and all $f : M \to \mathbb{R}$ for which the integral exists. In the special case that $\chi_M \equiv 1$ we say the integral is **invariant**.

Remark 3.27 (Abuse of notation). Integration is always with respect to some measure. If we are integrating with respect to the measure μ then for the sake of completeness we should

write

$$\int_M \dots \, d\mu(p).$$

But since we only ever consider one measure per space we integrate over and for the sake of brevity we abbreviate $dp \equiv d\mu(p)$.

If the homogeneous space is G itself then an invariant measure is called the (left) **Haar measure** on G (named after the Hungarian mathematician Alfréd Haar). We can say *the* Haar measure since Haar measures are unique up to multiplication with a constant and always exist (see Federer 2014, Ch. 2.7). Hence when integrating on the group itself we can always have a Haar measure μ_G so that the following equality holds

$$\int_G (h \cdot f)(g) \, dg = \int_G f(g) \, dg \qquad \forall h \in G, \tag{3.14}$$

where we abbreviated $dg := d\mu_G(g)$. We also call this invariant integral on the group the (left) **Haar integral**.

Not all homogeneous spaces admit a covariant integral but those in which we are interested all do. Going forward we will assume that all homogeneous spaces that we consider admit a covariant integral and that we can always use the equality from Definition 3.26.

Example 3.28 ($G = SE(2)$ and $M = \mathbb{R}^2$)**.** In the case we are most interested in, namely $G = SE(2)$ and $M = \mathbb{R}^2$, we are fortunate that the Lebesgue measure on $\mathbb{R}^2$ is invariant with respect to G. This is intuitively easy to understand: the area of a subset of $\mathbb{R}^2$ is invariant under both translation and rotation.

Example 3.29 (Haar measure on $SE(2)$)**.** The Haar measure on $SE(2)$ also conveniently coincides with the Lebesgue measure on $\mathbb{R}^2 \times [0, 2\pi)$ when using the parameterization from Example 3.15. Indeed, let $g = (\boldsymbol{x}_1, \theta_1)$ and $h = (\boldsymbol{x}_2, \theta_2)$ then:

$$\int_{\mathbb{R}^2} \int_0^{2\pi} ((\boldsymbol{x}_1, \theta_1) \cdot f)(\boldsymbol{x}_2, \theta_2) \, d\theta_2 \, d\boldsymbol{x}_2 = \int_{\mathbb{R}^2} \int_0^{2\pi} f\left((\boldsymbol{x}_1, \theta_1)^{-1}(\boldsymbol{x}_2, \theta_2)\right) d\theta_2 \, d\boldsymbol{x}_2.$$

When we change variables to $(\boldsymbol{x}_3, \theta_3) = (\boldsymbol{x}_1, \theta_1)^{-1}(\boldsymbol{x}_2, \theta_2)$ we obtain the following Jacobian matrix:

$$\frac{\partial(x_2^1, x_2^2, \theta_2)}{\partial(x_3^1, x_3^2, \theta_3)} = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

which has determinant 1. Consequently, the Haar integral (up to a multiplicative constant) on $SE(2)$ can be calculated as:

$$\int_{SE(2)} f(g) \, dg = \int_{\mathbb{R}^2} \int_0^{2\pi} f(\boldsymbol{x}, \theta) \, d\theta \, d\boldsymbol{x}. \tag{3.15}$$

3.3.2 Equivariant Linear Operators

Of course the objective of this chapter is building equivariant operators, so when is an integral operator (3.9) equivariant? Equivariance means that

$$A(g \cdot f) = g \cdot (Af)$$

for all $g \in G$ and $f \in B(M)$ or equivalently

$$g^{-1} \cdot A(g \cdot f) = Af. \tag{3.16}$$

This extra condition on A will naturally impose some restrictions on the kernel of the operator as the following lemma shows.

Lemma 3.30 (Equivariant linear operators)**.** Let M and N be homogeneous spaces of a Lie group G so that M admits a covariant integral with character χ_M. Let A be an integral operator (3.9) from $C(M) \cap B(M)$ to $C(N) \cap B(N)$ with a kernel $k_A \in C(M \times N)$. Then

$$A(g \cdot f) = g \cdot (Af)$$

for all $g \in G$ and $f \in C(M) \cap B(M)$ if and only if

$$\chi_M(g)\, k_A(g \cdot p, g \cdot q) = k_A(p, q) \tag{3.17}$$

for all $g \in G$, $p \in M$ and $q \in N$.
Moreover A is bounded (and so continuous) in the supremum norm if

$$\sup_{q \in N} \int_M |k_A(p, q)|\, dp < \infty. \tag{3.18}$$

Proof.
"$\Rightarrow$"

Assuming A to be equivariant, take an arbitrary $g \in G$ and $f \in C(M) \cap B(M)$ and substitute the definition of the group representation and A in (3.16) to find

$$\int_M k_A(p, g \cdot q)\, f(g^{-1} \cdot p)\, dp = \int_M k_A(p, q)\, f(p)\, dp \tag{3.19}$$

for all $q \in N$.

Fix $q \in N$ and let $F(p) := k_A(g \cdot p, g \cdot q) f(p)$ then observe that

$$(g \cdot F)(p) = k_A(g \cdot g^{-1} \cdot p, g \cdot q) f(g^{-1} \cdot p) = k_A(p, g \cdot q)\, f(g^{-1} \cdot p),$$

which is the left integrand from (3.19). Since we have assumed covariant integration we use Definition 3.26 and have

$$\int_M (g \cdot F)(p)\, dp = \chi_M(g) \int_M F(p)\, dp.$$

Applying this to (3.19) we find

$$\chi_M(g) \int_M k_A(g \cdot p, g \cdot q)\, f(p)\, dp = \int_M k_A(p, q)\, f(p)\, dp. \tag{3.20}$$

Since f was arbitrary and $p \mapsto k_A(p, q)$ continuous it follows that

$$\chi_M(g)\, k_A(g \cdot p, g \cdot q) = k_A(p, q)$$

for all $p \in M$.

"$\Leftarrow$"

Assuming $\chi_M(g)\, k_A(g \cdot p, g \cdot q) = k_A(p, q)$ for all $g \in G$, $p \in M$ and $q \in N$ then (3.20) follows for any choice of $f \in C(M) \cap B(M)$, $g \in G$ and $q \in N$. Substituting the covariant integral the other way yields (3.19), which implies (3.16) since $q \in N$ is arbitrary. The function f and group element g were also chosen arbitrarily so (3.16) follows for all $f \in C(M) \cap B(M)$ and $g \in G$.

Boundedness of A follows from

$$\begin{aligned}
\|Af\|_\infty &= \sup_{q \in N} \left| \int_M k_A(p, q)\, f(p)\, dp \right| \\
&\leq \sup_{q \in N} \int_M |k_A(p, q)|\, |f(p)|\, dp \\
&\leq \|f\|_\infty \cdot \sup_{q \in N} \int_M |k_A(p, q)|\, dp \\
&\overset{(3.18)}{<} \infty.
\end{aligned}$$

□

The condition on the kernel (3.18) is partially redundant with the symmetry requirement as the following lemma shows.

Lemma 3.31. In the same setting as Lemma 3.30. If the kernel $k_A \in C(M \times N)$ satisfies the symmetry (3.17) and condition (3.18) then

$$\|k_A(\,\cdot\,, q_1)\|_{L^1(M)} = \|k_A(\,\cdot\,, q_2)\|_{L^1(M)}$$

for all $q_1, q_2 \in N$.

Proof. Since N is a homogeneous space then for all $q_1, q_2 \in N$ there exists a $g \in G$ so that $q_1 = g \cdot q_2$, then

$$\begin{aligned}
\int_M |k_A(p, q_1)|\, dp &= \int_M |k_A(p, g \cdot q_2)|\, dp \\
&= \int_M \left| k_A(g \cdot g^{-1} \cdot p, g \cdot q_2) \right| dp \\
\overset{(3.17)}{=} \frac{1}{\chi_M(g)} &\int_M \left| k_A(g^{-1} \cdot p, q_2) \right| dp \\
\overset{\text{(Def. 3.26)}}{=} \frac{\chi_M(g)}{\chi_M(g)} &\int_M |k_A(p, q_2)|\, dp \\
&= \int_M |k_A(p, q_2)|\, dp.
\end{aligned}$$

□

The condition on the kernel from Lemma 3.30 can be exploited to express it as a function on M instead of $M \times N$. If we fix a $q_0 \in N$ and for all $q \in N$ we choose a $g_q \in G_{q_0, q}$ (i.e. so that $g_q \cdot q_0 = q$) then by (3.17) we have

$$k_A(p, q) = \chi_M(g_q^{-1})\, k_A(g_q^{-1} \cdot p, g_q^{-1} \cdot q) = \chi_M(g_q^{-1})\, k_A(g_q^{-1} \cdot p, q_0),$$

which fixes the second input of k_A. Consequently we could contain all the information of our kernel in a function that exists only on M as $\kappa_A(p) := k_A(p, q_0)$. This reduced kernel κ_A still has some restrictions placed on it for the resulting operator to be equivariant, as the following theorem makes precise.

Theorem 3.32 (Equivariant linear operators)**.** Let M and N be homogeneous spaces of a Lie group G so that M admits a covariant integral with respect to a character χ_M of G. Fix a $q_0 \in N$ and let $\kappa_A \in C(M) \cap L^1(M)$ be **compatible**, i.e. have the property that

$$\forall h \in G_{q_0} : h \cdot \kappa_A = \chi_M(h)\, \kappa_A. \tag{3.21}$$

Then the operator A defined by

$$(Af)(q) := \frac{1}{\chi_M(g_q)} \int_M (g_q \cdot \kappa_A)(p)\, f(p)\, dp$$

where for all $q \in N$ we can choose any g_q so that $g_q \cdot q_0 = q$, is a **well defined bounded linear operator** from $C(M) \cap B(M)$ to $C(N) \cap B(N)$ that is **equivariant** with respect to G.
Conversely every equivariant integral operator with a kernel $k_A \in C(M \times N)$ and with $k_A(\cdot, q) \in L^1(M)$ for some $q \in N$ is of this form.

Proof.
"$\Rightarrow$"

Assuming we have a $\kappa_A \in C(M) \cap L^1(M)$ that satisfies (3.21). Define $k_A \in C(M \times N)$ by

$$k_A(p, q) := \frac{1}{\chi_M(g_q)} (g_q \cdot \kappa_A)(p).$$

Then k_A is well defined since it does not depend on the choice of g_q for a given $q \in N$. If g'_q is another group element with $g_q \cdot q_0 = q$ then there exists a $h \in G_{q_0}$ so that $g'_q = g_q h$, we can check k_A is invariant under choice of $h \in G_{q_0}$:

$$\frac{1}{\chi_M(g_q h)} (g_q \cdot h \cdot \kappa_A)(p) = \frac{\chi_M(h)}{\chi_M(g_q)\chi_M(h)} (g_q \cdot \kappa_A)(p) = \frac{1}{\chi_M(g_q)} (g_q \cdot \kappa_A)(p).$$

The kernel k_A also satisfies the symmetry requirement (3.17) from Lemma 3.30:

$$\begin{aligned}
\chi_M(g)\, k_A(g \cdot p, g \cdot q) &= \chi_M(g) \frac{1}{\chi_M(g_{(g \cdot q)})} (g_{(g \cdot q)} \cdot \kappa_A)(g \cdot p) \\
&= \chi_M(g) \frac{1}{\chi_M(g g_q)} (g \cdot g_q \cdot \kappa_A)(g \cdot p) \\
&= \frac{\chi_M(g)}{\chi_M(g)\chi_M(g_q)} (g_q \cdot \kappa_A)(g^{-1} g \cdot p) \\
&= \frac{1}{\chi_M(g_q)} (g_q \cdot \kappa_A)(p) \\
&= k_A(p, q).
\end{aligned}$$

By Lemma 3.31 we have

$$\sup_{q \in N} \int_M |k_A(p, q)|\, dp = \|k_A(\cdot, q_0)\|_{L^1(M)} = \|\kappa_A\|_{L^1(M)} < \infty.$$

Consequently, A also satisfies (3.18) and is a bounded equivariant linear operator per Lemma 3.30.

"$\Leftarrow$"

Assuming we have an equivariant linear operator A with kernel $k_A \in C(M \times N)$ then we pick a fixed $q_0 \in N$ and define $\kappa_A \in C(M)$

$$\kappa_A(p) := k_A(p, q_0).$$

This reduced kernel κ_A satisfies the compatibility condition (3.21) since if $h \in G_{q_0}$ then

$$\begin{aligned}(h \cdot \kappa_A)(p) &= k_A(h^{-1} \cdot p, q_0) \\ &= k_A(h^{-1} \cdot p, h^{-1} \cdot q_0) \\ &= \chi_M(h)\, k_A(p, q_0) \\ &= \chi_M(h)\, \kappa_A(p).\end{aligned}$$

Since we required $k_A(\,\cdot\,, q) \in L^1(M)$ for some $q \in N$, we apply Lemma 3.31 to find

$$\|\kappa_A\|_{L^1(M)} = \|k_A(\,\cdot\,, q_0)\|_{L^1(M)} = \|k_A(\,\cdot\,, q)\|_{L^1(M)} < \infty.$$

□

Theorem 3.32 is the at the core of group equivariant CNNs since it allows us to generalize the familiar convolution operation present in CNNs to general linear operators that are equivariant with respect to a group of choice.

Example 3.33 (Group convolution)**.** Let $G = M = N$ be some Lie group. A Lie group always admits a Haar integral, so we have a trivial character $\chi = 1$. As reference element we obviously choose the unit element e, though any group element would do. Then $G_g = \{e\}$ and $G_{e,g} = \{g\}$ are both trivial. Hence we have no symmetry condition on the kernel. Any $\kappa_A \in C(G) \cap L^1(G)$ defines a linear operator $A : C(G) \cap B(G) \to C(G) \cap B(G)$ by

$$(Af)(h) = \int_G (h \cdot \kappa_A)(g)\, f(g)\, dg = \int_G \kappa_A(h^{-1}g)\, f(g)\, dg$$

We also call this operation **group cross-correlation** and denote it as

$$(\kappa \star_G f)(h) := \int_G (h \cdot \kappa)(g)\, f(g)\, dg.$$

As in the familiar $\mathbb{R}^n$ setting, group cross-correlation is closely related to **group convolution**, which is defined as

$$(\check{\kappa} *_G f)(h) := \int_G \check{\kappa}(g^{-1}h)\, f(g)\, dg.$$

We leave relating the two kernels κ and $\check{\kappa}$ as an exercise: when is $\kappa \star_G f = \check{\kappa} *_G f$?

As in the $\mathbb{R}^n$ case, when we talk about group convolution we mean both group cross-correlation and group convolution since they are interchangeable.

Example 3.34 (Rotation-translation equivariance in $\mathbb{R}^2$)**.** Let $G = SE(2) = \mathbb{R}^2 \rtimes SO(2)$ and $M = N = \mathbb{R}^2$. The Lebesgue measure on $\mathbb{R}^2$ is rotation-translation invariant so we have a G-invariant integral on $\mathbb{R}^2$. Choose $\boldsymbol{y}_0 = \mathbf{0}$ as the reference element then $G_{\boldsymbol{y}_0} = \{(\mathbf{0}, R(\theta)) \in G \mid \theta \in [0, 2\pi)\}$ is the stabilizer of $\boldsymbol{y}_0$. A kernel κ_A on $\mathbb{R}^2$ is then compatible if

$$(\mathbf{0}, R(\theta)) \cdot \kappa_A = \kappa_A \qquad \forall \theta \in [0, 2\pi),$$

i.e. κ_A needs to be radially symmetric. Now, we could have figured that out without building up the whole equivariance framework. But the next section will show how we can use the equivariance framework to step over the severe restriction that is imposed on the allowable kernels here.

3.4 Building a Rotation-translation Equivariant CNN

How do we now use this framework to construct a rotation-translation invariant network for images? From Example 3.34 we know that we do not have a lot of freedom in training a rotation-translation equivariant operator from $\mathbb{R}^2$ to $\mathbb{R}^2$. We can buy ourselves a lot more freedom by making the first linear operation in our network one that maps functions on $\mathbb{R}^2$ to functions on $SE(2)$. In the context of image analysis the process of transforming an image to a higher dimensional representation is called **lifting**. Therefore we will call the first layer in our network the **lifting layer**. Once we are operating on the group we have much more freedom since group convolution is equivariant. Just like in a conventional CNN we can have a series of **group convolution layers** that make up the bulk of our network. Of course we might not want our final product to live on the group, we might want to go back to $\mathbb{R}^2$, but we also have a recipe for that. Since going from the 3 dimensional space $SE(2)$ to the 2 dimensional space $\mathbb{R}^2$ is akin to projection we call a layer that does this a **projection layer**.

This three stage design is illustrated in Figure 3.5 for the retinal vessel segmentation application. For some more examples of the use of this type of network in medical imaging applications see Bekkers et al. (2018).

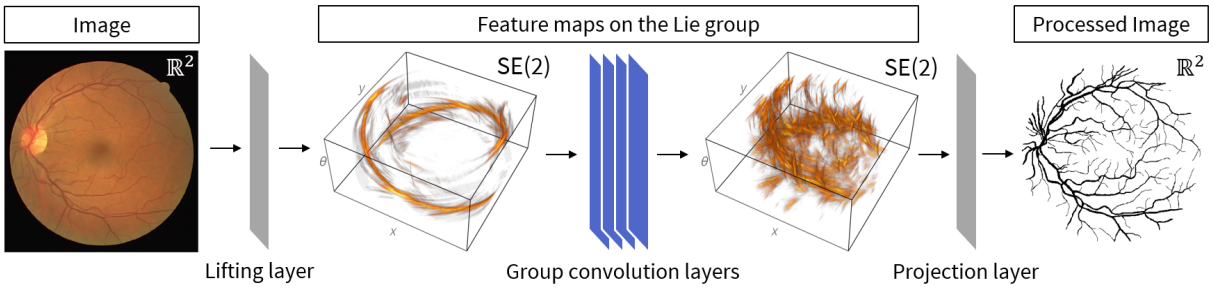


Figure 3.5: A G-CNN for retinal vessel segmentation that is rotation-translation equivariant. Lifting data to a higher dimensional space affords us more freedom in the kernels we train while maintaining equivariance.

3.4.1 Lifting Layer

Let $G = SE(2) \equiv \mathbb{R}^2 \rtimes [0, 2\pi)$ using the parameterization from Example 3.15. Let $M = \mathbb{R}^2$ and $N = G$. Choose $e = (\mathbf{0}, 0) \in N$ as the reference element then the stabilizer G_e is trivially $\{e\}$, so any kernel on $M = \mathbb{R}^2$ is compatible. The Lebesgue measure is rotation-translation equivariant so we have an invariant integral.

Let $n_0 \in \{1, 3\}$ be the number of input channels and denote the input functions as $f_j^{(0)} : \mathbb{R}^2 \to \mathbb{R}$ for j from 1 to n_0. Let us denote the number of desired feature maps in the first layers as n_1. Recall that there are two conventions for convolution layers: single channel and multi channel.

In the multi channel setup we associate with each output channel a number of kernels equal to the amount of input channels. Our parameters would be a set $\{\kappa_{ij}^{(1)}\}_{ij} \subset C(\mathbb{R}^2) \cap L^1(\mathbb{R}^2)$ of kernels and a set $\{b_i^{(1)}\}_i \subset \mathbb{R}$ of biases for i from 1 to n_1 and j from 1 to n_0. The calculation for output channel i is then given by

$$f_i^{(1)}(\boldsymbol{x}, \theta) = \sigma\left(\sum_{j=1}^{n_0} \int_{\mathbb{R}^2} \left((\boldsymbol{x}, \theta) \cdot \kappa_{ij}^{(1)}\right)(\boldsymbol{y})\, f_j^{(0)}(\boldsymbol{y})\, d\boldsymbol{y} + b_i^{(1)}\right),$$

for some choice of activation function σ.

In the single channel setup we associate a kernel with each input channel and then make linear combinations of the convolved input channels to generate output channels. Our parameters would then consist of a set of kernels $\{\kappa_j^{(1)}\}_j \subset C(\mathbb{R}^2) \cap L^1(\mathbb{R}^2)$ and a set of weights $\{a_{ij}^{(1)}\}_{ij} \subset \mathbb{R}$ and biases $\{b_i^{(1)}\}_i \subset \mathbb{R}$ for i from 1 to n_1 and j from 1 to n_0. The calculation for output channel i is then given by

$$f_i^{(1)}(\boldsymbol{x}, \theta) = \sigma\left(\sum_{j=1}^{n_0} a_{ij}^{(1)} \int_{\mathbb{R}^2} \left((\boldsymbol{x}, \theta) \cdot \kappa_j^{(1)}\right)(\boldsymbol{y})\, f_j^{(0)}(\boldsymbol{y})\, d\boldsymbol{y} + b_i^{(1)}\right).$$

In either case the actual lifting happens by translating and rotating the kernel over the image, a particular translation and rotation gives us a particular scalar value at the corresponding location in $SE(2)$.

Remark 3.35 (Orientation score transform). If you followed the course *Differential Geometry for Image Processing (2MMA70)* this will seem very familiar to you. Indeed this an **orientation score transform** except we do not design the wavelet filter (the kernel) ourselves but leave it up to the network to train.

3.4.2 Group Convolution Layer

We already saw in Example 3.33 how to do convolution on the group itself. On the Lie group we always have an invariant integral (the left Haar integral) and the symmetry requirement on the kernel is trivial so we have no restrictions to take into account for training the kernel (unlike for the case $\mathbb{R}^2 \to \mathbb{R}^2$). In layer $\ell \in \mathbb{N}$ with $n_{\ell-1}$ input channels and n_ℓ output channels the calculation for output channel i is given (for the single channel setup) by

$$f_i^{(\ell)} = \sigma\left(\sum_{j=1}^{n_{\ell-1}} a_{ij}^{(\ell)} \left(\kappa_j^{(\ell)} \star_G f_j^{(\ell-1)}\right) + b_i^{(\ell)}\right)$$

for all $i \in \{1, \ldots, n_\ell\}$ or

$$f_i^{(\ell)}(\boldsymbol{x}, \theta) = \sigma\left(\sum_{j=1}^{n_\ell} a_{ij}^{(\ell)} \int_{\mathbb{R}^2} \int_0^{2\pi} \left((\boldsymbol{x}, \theta) \cdot \kappa_j^{(\ell)}\right)(\boldsymbol{y}, \alpha)\, f_j^{(\ell-1)}(\boldsymbol{y}, \alpha)\, d\alpha\, d\boldsymbol{y} + b_i^{(\ell)}\right),$$

for some choice of activation function σ. Here the kernels $\kappa_j^{(\ell)} \in C(G) \times L^1(G)$, the weights $a_{ij}^{(\ell)} \in \mathbb{R}$ and the biases $b_i^{(\ell)} \in \mathbb{R}$ are the trainable parameters. Deducing the formula for the multi channel setup we leave as an exercise.

Group convolution layers can be stacked sequentially just like normal convolution layers in a CNN to make up the heart of a G-CNN, see Figure 3.5.

3.4.3 Projection

The desired output of our network is likely not a feature map on the group or some other higher dimensional homogeneous space. So at some point we have to transition away from them.

In traditional CNNs used for classification we saw that at some point we flattened our multi-dimensional array by 'forgetting' the spatial dimensions. Once we have discretized, flattening is of course also a viable approach for a G-CNN when the goal is classification. However we might not want to throw away our spatial structure, if the goal of the network is to transform its input in some way then we want to go back to our original input space (like in the example in Figure 3.5).

Applying our equivariance framework again to the case $G = M = SE(2)$ and $N = \mathbb{R}^2$. Choose $\mathbf{0} \in N$ as the reference element then its stabilizer is the subgroup of just rotations. So to construct an equivariant linear operator from $SE(2)$ to $\mathbb{R}^2$ requires a kernel κ on $SE(2)$ that satisfies

$$(\mathbf{0}, \beta) \cdot \kappa = \kappa \quad \Leftrightarrow \quad \kappa(\boldsymbol{x}, \theta) = \kappa\left(R(-\beta)\boldsymbol{x}, \theta - \beta\right) \qquad \forall \beta, \theta \in [0, 2\pi), \boldsymbol{x} \in \mathbb{R}^2,$$

where $R(-\theta)$ is the rotation matrix over $-\theta$. Consequently we can reduce the trainable (unrestricted) part of the kernel κ to a 2 dimensional slice:

$$\kappa(\boldsymbol{x}, \theta) = \kappa\left(R(-\theta)\boldsymbol{x}, 0\right).$$

A kernel like this gives us the desired equivariant linear operator and with a set of them we can construct a layer in the same fashion as the lifting and group convolution layer.

In practice this type of operator with trainable kernel is not what is used for projection from $SE(2)$ to $\mathbb{R}^2$. Instead the much simpler (and non-trainable) integration over the θ axis is used, let $f \in B(SE(2))$ then the operator $P : C(SE(2)) \cap B(SE(2)) \to C(\mathbb{R}^2) \cap B(\mathbb{R}^2)$ given by

$$(Pf)(\boldsymbol{x}) := \int_0^{2\pi} f(\boldsymbol{x}, \theta)\, d\theta, \tag{3.22}$$

is a bounded linear operator that is rotation-translation equivariant.

> *Remark* 3.36. Note that the projection operator (3.22) is one of our equivariant linear operators if we take the kernel to be $\kappa_P(\boldsymbol{x}, \theta) = \delta(\boldsymbol{x})$ where δ is the Dirac delta on $\mathbb{R}^2$. The Dirac delta is not a function in $C(G) \cap L^1(G)$ but we can take a sequence in $C(G) \cap L^1(G)$ that has κ_P as the limit in the weak sense, such as a sequence of narrowing Gaussians.

A common alternative to integrating over the orientation axis is taking the maximum over that axis:

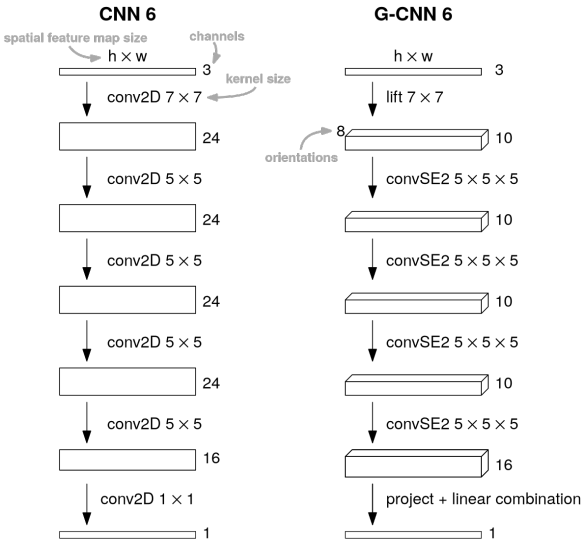
$$\left(P_{\max} f\right)(\boldsymbol{x}) := \max_{\theta \in [0, 2\pi)} f(\boldsymbol{x}, \theta). \tag{3.23}$$

This is not a linear operator but it is rotation-translation equivariant, we will revisit this projection operator later.

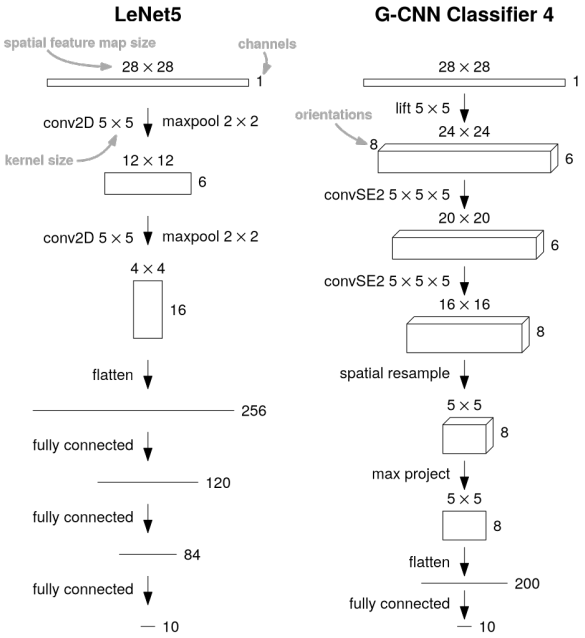
After we have once again obtained feature maps on $\mathbb{R}^2$ we can proceed to our desired output format in the same way as we would do with a classic CNN. Either we forget the spatial dimensions and transition to a fully connected network for classification applications or we take a linear combination of the obtained 2D feature maps to generate an output image such as in Figure 3.5 and Bekkers et al. (2018).

3.4.4 Discretization

To implement our developed G-CNN in practice we will need to switch to a discretized setting. For our specific case of an $SE(2)$ G-CNN the lifting layer typically uses kernels of size 5×5 to 7×7. We usually choose the number of discrete orientations to be 8, so an input of $\mathbb{R}^{H \times W}$ would be lifted to $\mathbb{R}^{8 \times H \times W}$, this may seem low but empirically this is around the sweet-spot between performance and memory usage/computation time. The group convolution layers usually employ $5 \times 5 \times 5$ kernels. In both cases we need to sample the kernel off-grid to be able to rotate them, for that we almost always use linear interpolation. Example G-CNN implementations are illustrated in Figure 3.6 for both segmentation and classification, examples for various medical applications can be found in Bekkers et al. (2018).



(a) A traditional CNN (left) versus a rotation-translation equivariant G-CNN (right) for segmenting a $h \times w$ color image. Figure 3.5 shows an application of this type.



(b) A traditional CNN (left) versus a rotation-translation equivariant G-CNN (right) for classifying 28×28 grayscale images into 10 classes. Digit classification falls into this category.

Figure 3.6: Example $SE(2)$ G-CNNs architectures versus similar traditional CNN architectures. The shapes show the size of the data tensors at each stage in the network.

As a general rule of thumb in deep learning we discretize as coarsely as we can get away with. Increasing the size of the kernels or the number of orientations does increase performance but nowhere near proportional to the increase in memory and computation time this causes. Keeping coarse kernels and increasing the depth of the network is a better way of spending a given memory/time budget.

Remark 3.37 (Linear interpolation alternatives). Higher order polynomial interpolation methods are highly undesirable on such coarse grids, as the oscillations can make the network behave erratically. More advanced interpolation techniques have been proposed, see for example Bekkers (2021), but the added computationally complexity can be a drawback. Just as with discretization, the rule of thumb for interpolation is: as coarsely as you can get away

$\mathbf{I}$ with.

3.5 Tropical Operators

We previously generalized the idea of a convolution to that of equivariant linear operators in a Lie group setting, essentially generalizing the domain of our functions. But we can go further by looking at the codomain and generalizing what we mean by 'linear'.

Linear and non-linear are generally thought of as an absolute dichotomy. Another way of thinking about a linear function or operator is as a map that preserves the algebraic properties of the field of real numbers. But could we not look at maps that preserve the structure of other algebraic structures rather than the field of real numbers?

We could in principle develop this idea using any algebraic structure, but to be useful in the context of neural networks we want the underlying set to be numeric and the operations to be able to be performed by a computer.

Additionally, restricting ourselves to fields would be very limiting, a more congenial algebraic structure is the semiring. Semirings turn up in many places, indeed the first mathematical structure we learn about, the set of natural numbers $\mathbb{N}$, is a semiring. Semirings arise in many areas of mathematics such as functional analysis, topology, graphs, combinatorics, optimization, etc. See Golan (1999) for an extensive survey of the applications of semirings.

3.5.1 Semirings

A semiring is an algebraic structure in which we can add and multiply elements, but in which neither subtraction nor division are necessarily possible.

Definition 3.38 (Semiring)**.** A **semiring** is a set R equipped with two binary operations $\oplus$ and $\odot$, called addition and multiplication, such that

(i) addition and multiplication are associative,

(ii) addition is commutative,

(iii) addition has an identity element $\mathbb{0}$,

(iv) multiplication has an identity element $\mathbb{1}$,

(v) multiplication distributes over addition:

$$a \odot (b \oplus c) = a \odot b \oplus a \odot c,$$
$$(a \oplus b) \odot c = a \odot c \oplus b \odot c,$$

(vi) multiplication by $\mathbb{0}$ annihilates:

$$\mathbb{0} \odot a = a \odot \mathbb{0} = \mathbb{0}.$$

Additionally if $a \oplus a = a$ we say the semiring is **idempotent** and if $a \odot b = b \odot a$ we say the semiring is **commutative** or **abelian**.

Just like with standard multiplication it is conventional to abbreviate $ab \equiv a \odot b$ if there can be no confusion. We also let $\odot$ take precedence over $\oplus$, i.e. $a \odot b \oplus c = (a \odot b) \oplus c$.

Remark 3.39 (Rig). Some works refer to semirings as **rigs**, from rings 'without negatives', hence the missing 'n'.

Example 3.40 (Real linear semiring)**.** The real numbers form a commutative semiring $(\mathbb{R}, +, \cdot)$ under standard addition and multiplication. Naturally all fields and rings are also semirings.

Example 3.41 (Viterbi semiring)**.** The Viterbi semiring is given by the unit interval $[0, 1]$ and the operations

$$a \oplus b := \max\{a, b\},$$
$$a \odot b := ab.$$

The additive unit is 0 and multiplicative unit is 1. This semiring is both idempotent and commutative. Note that there exists no element $(-a)$ so that $a \oplus (-a) = \mathbb{0} = 0$ and no element a^{-1} so that $a \odot a^{-1} = \mathbb{1} = 1$. So neither subtraction nor division is possible in this semiring.

Example 3.42 (Log semiring)**.** The log semiring is given by the set $\mathbb{R} \cup \{-\infty, +\infty\}$ and the operations

$$a \oplus b := \log\left(e^a + e^b\right),$$
$$a \odot b := a + b.$$

The additive unit is $-\infty$ and the multiplicative unit is 0.

The central object of study in linear algebra is the vector space over a field (usually $\mathbb{R}$ or $\mathbb{C}$). The analogue to the vector space in our generalized setting is the semimodule.

Definition 3.43 (Semimodule)**.** Let $(R, \oplus, \odot)$ be a semiring. A left R-semimodule or **semimodule** over R is a commutative monoid $(M, +)$ with additive identity 0_M and a map $R \times M \to M$ denoted by $(a, m) \mapsto am$, called **scalar multiplication**, so that for all $a, b \in R$ and $m, m' \in M$ the following hold:

(i) $(a \odot b)m = a(bm)$,

(ii) $a(m + m') = am + am'$,

(iii) $(a \oplus b)m = am + bm$,

(iv) $\mathbb{1}m = m$,

(v) $a0_M = 0_M = \mathbb{0}m$.

Naturally every semiring is a semimodule over itself just like every field is a vector space over itself.

Example 3.44. Let R be a semiring and S a non-empty set, then R^S (the set of all functions $S \to R$) is a semimodule with addition and scalar multiplication defined element-wise: let $f, f' \in R^S$ and $r \in R$ then

$$(f \oplus f')(s) := f(s) \oplus f'(s) \quad \text{and} \quad (r \odot f)(s) := r \odot f(s)$$

for all $s \in S$. Here we used $\oplus$ and $\odot$ for the operations in the semimodule as well since they correspond with the $\oplus$ and $\odot$ operations in the semiring. The additive identity of the semimodule is the constant function $s \mapsto \mathbb{0}$. This generalizes the prototypical vector space $\mathbb{R}^n$ to the semimodule R^n using the notational convention $n \equiv \{1, \ldots, n\}$. For non-finite S we get the generalization of function vector spaces to function semimodules.

Example 3.45. Let $R = ([0,1], \max, \cdot)$ be the Viterbi semiring from Example 3.41 and let S be a manifold. Let $f, f' : S \to [0,1]$ and define addition and scalar multiplication as

$$(f + f')(s) := \max\{f(s), f'(s)\} \quad \text{and} \quad (rf)(s) := rf(s),$$

then the functions from S to $[0,1]$ form a semimodule.

Now we can formulate a generalization of linear maps in the form of semiring homomorphisms.

Definition 3.46 (Semimodule homomorphism)**.** Let R be a semiring and let X and Y be semimodules over R. Then a map $A : X \to Y$ is an **R-homomorphic map** or an **R-homomorphism** if for all $a, b \in R$ and $f, f' \in X$:

$$A(af + bf') = a(Af) + b(Af'),$$

where on the left the addition and scalar multiplication happen in X and on the right the addition and scalar multiplication happen in Y.

Just like with the definition of linearity the single condition above is equivalent to the following two conditions:

$$A(af) = a(Af') \quad \text{and} \quad A(f + f') = Af + Af' \qquad \forall a \in R,\ f, f' \in X.$$

Under this definition we can understand linear as meaning homomorphic with respect to the real linear semigroup $(\mathbb{R}, +, \cdot)$. But now, instead of just considering linear maps we can pick another semigroup and consider homomorphisms with respect to this semigroup. This allows us to construct equivariant semimodule homomorphic operators in the same fashion as we did with equivariant linear operators in section 3.3.2. We will develop such a class of equivariant operators for a particular choice of semiring.

3.5.2 Tropical Semiring

Definition 3.47 (Tropical semiring)**.** The max tropical semiring or max-plus algebra or simply **tropical semiring** is the semiring $\mathbb{R}_{\max} := (\mathbb{R} \cup \{-\infty\}, \oplus, \odot)$, with the operations

$$a \oplus b := \max\{a, b\},$$
$$a \odot b := a + b.$$

Where the additive identity is $\mathbb{0} := -\infty$ and the multiplicative identity is $\mathbb{1} := 0$.

Since $a \oplus a = \max\{a, a\} = a$ and $a \odot b = a + b = b + a = b \odot a$ the tropical semiring is both idempotent and commutative. By definition we set $a + (-\infty) := -\infty$ so that we satisfy the annihilation property $a \odot \mathbb{0} = \mathbb{0} \odot a = \mathbb{0}$.

> *Remark* 3.48. The tropical semiring can alternatively be defined as $(\mathbb{R} \cup \{+\infty\}, \min, +)$, which is then also called the min tropical semiring or min-plus algebra. But we observe that one is isomorphic to the other via negation so we make a style choice and go with the max version.

Example 3.49 (ReLU)**.** Recall that the rectified linear unit is defined as $x \mapsto \{x, 0\}$ for $x \in \mathbb{R}$. In the tropical setting we can write this as $x \mapsto x \oplus 0$ for $x \in \mathbb{R}_{\max}$, hence the ReLU may not be a linear or affine function but it is **tropically affine**. So we can think about a typical ReLU neural network as really alternating operations from two distinct semirings.

We based our construction of equivariant linear operators on integration. When you consider an integral as a limit of a sum of products we can see how we could define a type of integration with respect to another semigroup.

Recall that Riemannian integration is defined in terms of Darboux sums over ever smaller partitions of the underlying space. Let M be a manifold with some Radon measure μ and $(R, \oplus, \odot)$ a semiring. Let P be a partition of M and define $\mu(P) := \sup_{S\in P} \mu(S)$. Then we can generalize a Darboux sum

$$\lim_{\mu(P)\to 0} \sum_{S\in P} f(p_S) \cdot \mu(S),$$

as:

$$\lim_{\mu(P)\to 0} \bigoplus_{S\in P} f(p_S) \odot \mu(S), \tag{3.24}$$

where $p_S \in M$ is any arbitrary point in the partition element S and assuming that for all $\varepsilon > 0$ we can find a partition P so that $\mu(P) < \varepsilon$. Filling in the tropical semiring operations we obtain:

$$\lim_{\mu(P)\to 0} \max_{S\in P} (f(p_S) + \mu(S))$$

($\mu(S) \to 0$ since $\mu(P) \to 0$)

$$= \lim_{\mu(P)\to 0} \max_{S\in P} f(p_S) \tag{3.25}$$

(Assuming f is such that the limit exists and is unique)

$$= \sup_{p\in M} f(p).$$

Lebesgue integration can be generalized similarly. This would yield the same $\sup_{p\in M} f(p)$ formula but for all measurable functions that are bounded from above rather then all the functions for which the Darboux sum has a unique limit. We will not detail the Lebesgue construction (see Kolokoltsov and Maslov (1997) for that) but proceed with the $\sup_{p\in M} f(p)$ formula as our definition of tropical integral.

Remark 3.50. Classic example of a function that is Lebesgue integrable but not Riemann integrable is the indicator function of the rational numbers $\mathbb{1}_{\mathbb{Q}}$. The limit (3.25) does not exist depending on our choice of points p_S but in the Lebesgue sense we simply get $\sup_{x\in\mathbb{R}} \mathbb{1}_{\mathbb{Q}}(x) = 1$ as expected.

Definition 3.51. Let M be a manifold, then we define the set of measurable $\mathbb{R}_{\max}$-valued functions that are bounded from above as

$$BA(M) := BA(M, \mathbb{R}_{\max}) := \left\{ f \in \mathbb{R}_{\max}^M \,\middle|\, \sup_{p\in M} f(p) < \infty \text{ and } f \text{ is measurable} \right\}.$$

Additionally if f is not identical to $-\infty$ everywhere we say f is **proper**.

Clearly BA(M) is a tropical semimodule (i.e. a semimodule with respect to the tropical semiring) under pointwise addition and multiplication:

$$(f \oplus f')(p) := \max\{f(p), f'(p)\} \quad \text{and} \quad (a \odot f)(p) := a + f(p),$$

for all $a \in \mathbb{R}_{\max}$, $f, f' \in BA(M)$ and all $p \in M$.

The functions in BA(M) are exactly those for which the tropical integral exists.

Definition 3.52 (Tropical integral)**.** Let M be a manifold then we call the mapping $BA(M) \to \mathbb{R}_{\max}$ defined by

$$f \mapsto \sup_{p \in M} f(p)$$

for $f \in BA(M)$ the **tropical integral** over M.

We can easily check that the tropical integral is a **tropical map** (i.e. a semimodule homomorphism with respect to the tropical semiring) from $BA(M)$ to $\mathbb{R}_{\max}$ in the same way that the (linear) integral is a linear map from the integrable functions to $\mathbb{R}$. For all $a, b \in \mathbb{R}_{\max}$ and $f, f' \in BA(M)$ we have:

$$\sup_{p \in M}(a \odot f \oplus b \odot f')(p) = a \odot \left(\sup_{p \in M} f(p)\right) \oplus b \odot \left(\sup_{p \in M} f'(p)\right).$$

Let M be a homogeneous space of the Lie group G, let '$\cdot$' denote both the action of G on M and the corresponding representation on functions on M. Since this representation does not affect the codomain of the functions it does not change the supremum, hence the tropical integral is always invariant:

$$\sup_{p \in M} (g \cdot f)(p) = \sup_{p \in M} f(p).$$

So now we have developed an alternative notion of linearity and an alternative to the (linear) invariant integral. We will use these elements to construct a new type of equivariant operator in the same manner as we did before with linear operators.

3.5.3 Equivariant Tropical Operators

The starting point for our equivariant linear operators was the integral operator from (3.9). We obtain the tropical analogue simply by replacing the linear semiring operations:

$$(Af)(q) := \int_M k_A(p,q) \cdot f(p)\, dp,$$

with the tropical semiring operations:

$$(Tf)(q) := \sup_{p \in M} k_T(p,q) + f(p). \tag{3.26}$$

We can see that if $k_T \in BA(M \times N)$ and $f \in BA(M)$ then $Tf \in BA(N)$. It is straightforward to verify that T is a tropical operator from $BA(N)$ to $BA(N)$.

Now we can proceed in the exact same way as in section 3.3.2 to find out when T is equivariant.

Lemma 3.53 (Equivariant tropical integral operators)**.** Let M and N be homogeneous spaces of a Lie group G. Let T be a tropical integral operator (3.26) from $BA(M)$ to $BA(N)$ with a kernel $k_T \in BA(M \times N)$. Then

$$T(g \cdot f) = g \cdot (Tf)$$

for all $g \in G$ and $f \in BA(M)$ if and only if

$$k_T(g \cdot p, g \cdot q) = k_T(p, q) \tag{3.27}$$

for all $g \in G$, $p \in M$ and $q \in N$.

Proof. First we show that $Tf \in BA(N)$:

$$\begin{aligned}\sup_{q \in N} (Tf)(q) &= \sup_{q \in N} \left(\sup_{p \in M} k_T(p, q) + f(p) \right) \\ &\leq \left(\sup_{(p,q) \in M \times N} k_T(p, q) \right) + \left(\sup_{p \in M} f(p) \right) \\ &< \infty,\end{aligned}$$

since both k_T and f are bounded from above.

"$\Rightarrow$"

Assuming T to be equivariant, take an arbitrary $g \in G$ and $f \in BA(M)$ and substitute the definition of the group representation and T in

$$g^{-1} \cdot T(g \cdot f) = Tf$$

to find

$$\sup_{p \in M} k_T(p, g \cdot q) + f(g^{-1} \cdot p) = \sup_{p \in M} k_T(p, q) + f(p) \tag{3.28}$$

for all $q \in N$. Since the tropical integral is invariant under domain transformation we can substitute p with $g \cdot p$ on the left without changing the result:

$$\sup_{p \in M} k_T(g \cdot p, g \cdot q) + f(p) = \sup_{p \in M} k_T(p, q) + f(p)$$

for all $q \in N$. The equality only holds for all f if

$$k_T(g \cdot p, g \cdot q) = k_T(p, q) \tag{3.29}$$

for all $p \in M$, $q \in N$ and $g \in G$.

"$\Leftarrow$"

Assuming $k_T(g \cdot p, g \cdot q) = k_T(p, q)$ for all $g \in G$, $p \in M$ and $q \in N$ then (3.29) follows for any choice of $f \in BA(M)$, $g \in G$ and $q \in N$. Substituting the invariant tropical integral the other way yields (3.28), which implies the equivariance

$$g^{-1} \cdot T(g \cdot f) = Tf$$

since $q \in N$ is arbitrary. The function f and group element g were also chosen arbitrarily so equivariance follows for all $f \in BA(M)$ and $g \in G$. □

Now we can use the same strategy as we did in the linear case and use the symmetry (3.27) to fix the second argument of the kernel k_T. Pick a $q_0 \in N$ then for all $q \in N$ there exists at least one $g_q \in G_{q_0}$, consequently

$$k_T(p,q) = k_T(p, g_q \cdot q_0) = k_T(g_q \cdot g_q^{-1} \cdot p, g_q \cdot q_0) = k_T(g_q^{-1} \cdot p, q_0)$$

for all $p \in M$ and $q \in N$. From which we can define a reduced kernel $\kappa_T \in BA(M)$ as

$$\kappa_T(p) := k_A(p, q_0)$$

that still allows us to recover the full kernel by

$$k_T(p,q) = k_T(g_q^{-1} \cdot p, q_0) = \kappa_T(g_q^{-1} \cdot p).$$

This whole construction in the end gives us the same type of operator as in Theorem 3.32 but with the linear operations switched for tropical operations:

$$(Af)(q) := \int_M (g_q \cdot \kappa_A)(p) \cdot f(p)\, dp,$$

$$(Tf)(q) := \sup_{p \in M} (g_q \cdot \kappa_T)(p) + f(p).$$

We summarize this result in the following theorem.

Theorem 3.54 (Equivariant tropical operators)**.** Let M and N be homogeneous spaces of a Lie group G. Fix a $q_0 \in N$ and let $\kappa_T \in BA(M)$ be **compatible**, i.e

$$\forall h \in G_{q_0} : h \cdot \kappa_A = \kappa_A.$$

Then the operator T defined by

$$(Tf)(q) := \sup_{p \in M} (g_q \cdot \kappa_T)(p) + f(p)$$

where for all $q \in N$ we choose any g_q so that $g_q \cdot q_0 = q$, is a **well defined tropical operator** from $BA(M)$ to $BA(N)$ that is **equivariant** with respect to G.
Conversely every tropical integral operator (3.26) with a kernel in $BA(M \times N)$ that is equivariant is of this form.

Proof. (sketch)

"$\Rightarrow$"

Assume we have a $\kappa_T \in BA(M)$ that is compatible. Define

$$k_T(p,q) := (g_q \cdot \kappa_T)(p)$$

we can check that k_T is well defined and satisfies the requirements of Lemma 3.53 because of the compatibility condition on κ_T.

"$\Leftarrow$"

Assume we have a tropical integral operator T, then by Lemma 3.53 we have a kernel $k_T \in BA(M \times N)$ that satisfies (3.27). Fix a $q_0 \in N$ and define

$$\kappa_T(p) := k_T(p, q_0),$$

then we can check that κ_T satisfies the compatibility condition. □

Example 3.55 (Max pooling)**.** Let $G = (\mathbb{R}^n, +)$ be the translation group and $M = N = \mathbb{R}^n$. Pick a subset $S \subset \mathbb{R}^n$ and define

$$\kappa_T(p) := \begin{cases} 0 & \text{if } p \in S, \\ -\infty & \text{elsewhere.} \end{cases}$$

Then the corresponding operator T equals

$$\begin{aligned}(Tf)(\boldsymbol{y}) &= \sup_{\boldsymbol{x}\in\mathbb{R}^n} (\boldsymbol{y} \cdot \kappa_T)(\boldsymbol{x}) + f(\boldsymbol{x}) \\ &= \sup_{\boldsymbol{x}\in\mathbb{R}^n} \kappa_T(\boldsymbol{x} - \boldsymbol{y}) + f(\boldsymbol{x}) \\ &= \sup_{\boldsymbol{x}\in\boldsymbol{y}+S} f(\boldsymbol{x}),\end{aligned}$$

so at each point $\boldsymbol{y}$ the output equals the supremum of the input in the subset $\boldsymbol{y} + S$. Think of this a continous version the shift-invariant max pooling operation usually seen in CNNs. In this light we can see that max pooling is a tropical operator.

Example 3.56 (Tropical convolution)**.** Let $G = M = N$ be some Lie group, then we call the equivariant tropical operation **tropical convolution** or **morphological convolution**, which we denote as

$$(\kappa \square_G f)(h) := \sup_{g\in G} (h \cdot \kappa)(g) + f(g).$$

The name morphological convolution comes from the field of grayscale morphology where these types of operators have previously been used for image processing applications, see Wikipedia (2021a). An example use of these types of operations in neural networks can be found in Smets et al. (2021).

Example 3.57 (Pointwise ReLU)**.** Let $G = M = N$ be some Lie group and let $f \in BA(M)$. Define a kernel

$$\kappa_T(g) := \begin{cases} 0 & \text{if } g = e, \\ -\sup_{h\in G} f(h) & \text{else.} \end{cases}$$

Then applying the corresponding operator to f gives

$$\begin{aligned}(Tf)(h) &= \sup_{g\in G}(h \cdot \kappa_T)(g) + f(g) \\ &= \max\left\{f(h), \sup_{g\in G}\left(-\sup_{z\in G} f(z)\right) + f(g)\right\} \\ &= \max\{f(h), 0\} \\ &= \mathrm{ReLU}(f(h)).\end{aligned}$$

We have not said anything about the boundedness of T in Theorem 3.54 since it is not clear what metric or norm we should consider on the function space BA. In the following lemma we detail some reasonable conditions under which T will be bounded in the supremum norm.

Lemma 3.58. In the setting of Theorem 3.54: if $f \in B(M)$ and $\sup_{p\in M} \kappa_T(p) = 0$ then $Tf \in B(N)$ and T is a bounded operator from $B(M)$ to $B(N)$ in the supremum norm.

Proof.

$$\begin{aligned}
\|Tf\|_\infty &= \sup_{q\in N} \left| \sup_{p\in M} (g_q \cdot \kappa_T)(p) + f(p) \right| \\
&\leq \sup_{q\in N} \left| \sup_{p\in M} (g_q \cdot \kappa_T)(p) + \sup_{p\in M} f(p) \right| \\
&= \sup_{q\in N} \left| \sup_{p\in M} \kappa_T(p) + \sup_{p\in M} f(p) \right| \\
&= \left| \sup_{p\in M} \kappa_T(p) + \sup_{p\in M} f(p) \right| \\
&= \left| \sup_{p\in M} f(p) \right| \\
&\leq \sup_{p\in M} |f(p)| \\
&= \| f \|_\infty .
\end{aligned}$$

□

We have seen how we can use semirings, and the tropical semiring in particular, to develop classes of equivariant operators other than linear. From the examples we saw we can conclude that many operators currently in use in neural networks (particularly the ReLU and max pooling activation functions) are special cases of tropical or tropically affine operators and fit well inside the equivariant semimodule homomorphism framework.

Bibliography

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (2018). **Automatic differentiation in machine learning: a survey**. arXiv: 1502.05767 [cs.SC] (page 38).

Bekkers, Erik J (2021). **B-Spline CNNs on Lie Groups**. arXiv: 1909.12057 [cs.LG] (page 68).

Bekkers, Erik J, Maxime W Lafarge, Mitko Veta, Koen AJ Eppenhof, Josien PW Pluim, and Remco Duits (2018). "Roto-translation covariant convolutional networks for medical image analysis". In: **International Conference on Medical Image Computing and Computer-Assisted Intervention**. Springer, pp. 440–448. URL: https://arxiv.org/abs/1804.03393 (pages 65, 67, 68).

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (2020). **Language Models are Few-Shot Learners**. arXiv: 2005.14165 [cs.CL] (page 25).

Cohen, Taco, Mario Geiger, and Maurice Weiler (2020). **A General Theory of Equivariant CNNs on Homogeneous Spaces**. arXiv: 1811.02017 [cs.LG] (page 49).

Cohen, Taco and Max Welling (2016). "Group Equivariant Convolutional Networks". In: **International Conference on Machine Learning**. PMLR, pp. 2990–2999 (page 49).

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: **2009 IEEE conference on computer vision and pattern recognition**. Ieee, pp. 248–255 (page 5).

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization." In: **Journal of Machine Learning Research** 12.7 (page 44).

Dumoulin, Vincent and Francesco Visin (2018). **A guide to convolution arithmetic for deep learning**. arXiv: 1603.07285 [stat.ML] (page 30).

Federer, Herbert (2014). **Geometric Measure Theory**. Springer. ISBN: 978-3-540-60656-7. DOI: 10.1007/978-3-642-62010-2 (page 60).

Fukushima, Kunihiko (1987). "Neural network model for selective attention in visual pattern recognition and associative recall". In: **Applied Optics** 26.23, pp. 4985–4992 (page 5).

Glorot, Xavier and Yoshua Bengio (May 2010). "Understanding the difficulty of training deep feedforward neural networks". In: **Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics**. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html (page 28).

Golan, Jonathan S. (1999). **Semirings and their Applications**. Dordrecht: Springer Netherlands. ISBN: 978-90-481-5252-0. DOI: 10.1007/978-94-015-9333-5. URL: http://link.springer.com/10.1007/978-94-015-9333-5 (page 69).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). **Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification**. arXiv: 1502.01852 [cs.CV] (page 29).

Ivakhnenko, Aleksei Grigorevich and Valentin Grigorevich Lapa (1966). **Cybernetic predicting devices**. Tech. rep. PURDUE UNIV LAFAYETTE IND SCHOOL OF ELECTRICAL ENGINEERING (page 5).

Kingma, Diederik P. and Jimmy Ba (2017). **Adam: A Method for Stochastic Optimization**. arXiv: 1412.6980 [cs.LG] (pages 45, 46).

Kolokoltsov, Vassili N. and Victor P. Maslov (1997). **Idempotent Analysis and Its Applications**. Dordrecht: Springer Netherlands. ISBN: 978-90-481-4834-9. DOI: 10.1007/978-94-015-8901-7. URL: http://link.springer.com/10.1007/978-94-015-8901-7 (page 72).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: **Advances in Neural Information Processing Systems**. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (page 5).

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel (1989). "Backpropagation applied to handwritten zip code recognition". In: **Neural computation** 1.4, pp. 541–551 (page 5).

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). "Gradient-based learning applied to document recognition". In: **Proceedings of the IEEE** 86.11, pp. 2278–2324 (page 37).

Lee, John M. (2010). **Introduction to Topological Manifolds**. 2nd ed. Graduate Texts in Mathematics. Springer. ISBN: 978-1-4419-7939-1. DOI: 10.1007/978-1-4419-7940-7 (page 49).

Lee, John M. (2013). **Introduction to Smooth Manifolds**. 2nd ed. Graduate Texts in Mathematics. Springer. ISBN: 978-1-4419-9981-8. DOI: 10.1007/978-1-4419-9982-5_1 (pages 49, 50, 54).

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: **The bulletin of mathematical biophysics** 5.4, pp. 115–133 (page 5).

Oh, Kyoung-Su and Keechul Jung (2004). "GPU implementation of neural networks". In: **Pattern Recognition** 37.6, pp. 1311–1314 (page 5).

Smets, Bart, Jim Portegies, Erik Bekkers, and Remco Duits (2021). **PDE-based Group Equivariant Convolutional Neural Networks**. arXiv: 2001.09046 [cs.LG] (page 76).

Tao, Terence (2011). **An Introduction to Measure Theory**. Vol. 126. Graduate Studies in Mathematics. American Mathematical Society. isbn: 978-1-4704-6640-4. doi: 10.1090/gsm/126 (page 59).

Tieleman, Tijmen, Geoffrey Hinton, et al. (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: **Coursera: Neural networks for machine learning** 4.2, pp. 26–31 (page 45).

Wikipedia (Sept. 26, 2021a). **Mathematical morphology**. Page Version ID: 1046500059. url: https://en.wikipedia.org/w/index.php?title=Mathematical_morphology&oldid=1046500059 (page 76).

Wikipedia (Dec. 16, 2021b). **Types of artificial neural networks**. Page Version ID: 1053596085. url: https://en.wikipedia.org/w/index.php?title=Types_of_artificial_neural_networks&oldid=1053596085 (pages 20, 21).