Chapter 2

Theoretical Background

Deep learning, *a class of machine learning algorithms based on neural networks*, has revolutionized the way we tackle a problem from a ML perspective and is one if not the most important factor for recent ML achievements. Solving complex tasks such image classification or language translation, that for years have bedevilled traditional ML algorithms, constitutes the signature of deep learning (DL). Admittedly, *the advent of a deep convolutional neural network*, the AlexNet (Krizhevsky et al. 2012) on September 30 of 2012, signified the "modern birthday" of this field. On this day, AlexNet not only won the ImageNet (Deng et al. 2009) Large Scale Visual Recognition Challenge (ILSVRC), but dominated it, achieving a top-5 accuracy of 85 %, surpassing the runner-up which achieved a top-5 accuracy of 75 %. AlexNet showed that neural networks (NNs) are not merely a pipe-dream, but they can be applied in real-world problems. It is worth to notice that ideas of NNs trace back to 1943, but it was until recently that these ideas got materialized. The reason for this recent breakthrough of DL (and ML) is twofold. First, the availability of large datasets—the era of big data—such as ImageNet. Second, the increase in computational power, mainly of GPUs for DL, accelerating the training of deep NNs and traditional ML algorithms.

2.1 Machine Learning Preliminaries

Since DL is a subfield of ML, it is necessary to familiarize with the later before diving into the former. In this section, the necessary theoretical background and jargon of ML is presented. Machine learning can be defined as *"the science and (art) of programming computers so they can learn from the data"* (Géron 2017). A more technical definition is the following:

Definition 2.1 (Machine learning, Mitchell 1997)**.** *A computer program is said to learn from experience **E** with respect to some class of tasks **T** and some performance measure **P**, if its performance on **T**, as measured by **P**, improves with experience **E**.*

For instance, a computer program that classifies emails into spam and non-spam (the task **T**), can improve its accuracy, i.e. the percentage of correctly classified emails (the performance **P**), through examples of spam and non-spam emails (the experience **E**). But in order to take advantage of the experience aka *data*, it must be written in such a way that *adapts to the patterns in the data*. Certainly, a *traditional spam filter can not learn from experience*, since the latter does not affect the classification rules of the former and as such, its performance. For a traditional spam filter to adapt to new patterns and perform better, it must change its hard-wired rules, but by then it will be a different program. In contrast, a *ML-based filter can adapt to new patterns, simply because it has been programmed to do so*.

In other words, *in traditional programming we write rules for solving* $\boldsymbol{T}$ *whereas* ***in ML we write rules to learn the rules*** *for solving* $\boldsymbol{T}$. This subtle but essential difference is what gives ML algorithms the ability to take advantage of the data.

2.1.1 Learning paradigms

Depending on the type of experience they are allowed to have during their *training phase* (Goodfellow et al. 2016), ML approaches are divided into three main *learning paradigms*: ***unsupervised learning***, ***supervised learning*** *and* ***reinforcement learning***. The following definitions are not by any means formal, but merely serve as an intuitive description of the different paradigms.

Definition 2.2 (Unsupervised learning)**.** *The experience comes in the form* $\mathcal{D}_{train} = \{\boldsymbol{x}_i\}$*, where* $\boldsymbol{x}_i \sim p(\boldsymbol{x})$ *is the input of the* i*-th training instance aka sample. In this paradigm we are interested in learning useful properties of the underlying structure captured by* $p(\boldsymbol{x})$ *or* $p(\boldsymbol{x})$ *itself.*

For example, suppose we are interested in generating images that look like Picasso paintings. In this case, the input is just the pixel values, i.e. $\boldsymbol{x} \in \mathbb{R}^{W \times H \times 3}$. The latter follow a distribution $p(\boldsymbol{x})$, so all we have to do is to train an unsupervised learning algorithm with many Picasso paintings to get a *model*, that is $\hat{p}(\boldsymbol{x})$. Assuming the estimation of the original distribution is good enough, new realistically looking paintings (with respect to original Picasso paintings) can be "drawn" by just sampling from $\hat{p}(\boldsymbol{x})$. In the ML parlance, this task is known as *generative modeling* while inputs are also called *features, predictors or descriptors*.

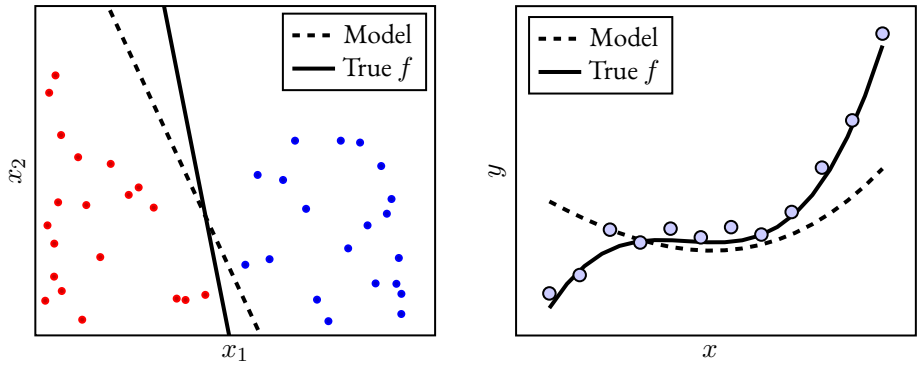
Definition 2.3 (Supervised learning)**.** *The experience comes in the form* $\mathcal{D}_{train} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$*, where* $(\boldsymbol{x}_i, \boldsymbol{y}_i) \sim p(\boldsymbol{x}, \boldsymbol{y})$ *and* $\boldsymbol{y}_i$ *is the output aka label of the* i*-th training instance. In this paradigm we are usually interested in learning a function* $f : \mathcal{X} \to \mathcal{Y}$*.*

This paradigm comes mainly under two flavors: *regression and classification*, which are schematically depicted in Figure 2.1. In regression the interest is in predicting a continuous value given an input, i.e. $y \in \mathbb{R}$, such as a molecular property given a mathematical representation of a molecule. In classification, the interest is to predict in which of k classes an input belongs to, i.e. $y \in \{1, \ldots, k\}$, such as predicting the breed of a dog image given the raw pixel values of the image. The term "supervised" is coined due to the "human supervision" the algorithm receives during its training phase, through the presence of the correct answer (the label) in the experience. In a sense, in this paradigm we "teach" the learning algorithm aka *learner*. It should be emphasized that the label is not constrained to be single-valued, but can also be multi-valued. In this case, one talks about *multi-label regression or classification* (Read et al. 2009).

A more exotic form of supervised learning is *conditional generative modelling*, where the interest is in estimating $p(\boldsymbol{x} \mid \boldsymbol{y})$. For example, one may want to build a model that generates images of a specific category or a *model that designs molecules/materials with tailored properties* (K. Kim et al. 2018; Yao et al. 2021; Gebauer et al. 2022). This is one approach of how ML can tackle the *inverse design problem* in chemistry.

Definition 2.4 (Reinforcement learning)**.** *The experience comes from the interaction of the learner, called agent in this context, with its environment. In other words, there is a feedback loop between the learner and its environment. In this paradigm we are interested in building an agent that can take suitable actions in a given situation.*

The agent observes its *environment*, selects and performs *actions* and gets *rewards or penalties* in



(A) Classification. **(B)** Regression.

FIGURE 2.1: Main tasks of supervised learning.

return. The goal is to learn an optimal strategy, called a *policy*, that *maximizes the long-term reward* (Géron 2017). A policy simply defines the action that the agent should choose in a given situation. In contrast to supervised learning, where the correct answers are provided to the learner, *in reinforcement learning the learner must find the optimal answers by trial and error* (Bishop 2007). Reinforcement learning techniques find application in fields such as gaming (AlphaGo is a well known example), robotics, autonomous driving and recently chemistry (H. Li et al. 2018; Gow et al. 2022). Since in the present thesis only supervised learning techniques were employed, the remaining of this chapter is devoted to this learning paradigm.

2.1.2 Formulating the problem of supervised learning

The general setting of supervised learning is as follows: we assume that there is some relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon} \tag{2.1}$$

and we want to estimate f from the data. The function f represents the *systematic information* that $\boldsymbol{x}$ gives about $\boldsymbol{y}$ while $\boldsymbol{\epsilon}$ is a *random error term* independent of $\boldsymbol{x}$ and with zero mean. More formally, we have an input space $\mathcal{X}$, an output space $\mathcal{Y}$ and we are interested in learning a function $\hat{h}\colon \mathcal{X} \to \mathcal{Y}$, called the *hypothesis*, which produces an output $\boldsymbol{y} \in \mathcal{Y}$ given an input $\boldsymbol{x} \in \mathcal{X}$. At our disposal we have a collection of input-output pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, forming the ***training set*** denoted as $\mathcal{D}_{\text{train}}$, with the pairs drawn i.i.d from $p(\boldsymbol{x}, \boldsymbol{y})$.

Ideally, we would like to learn a hypothesis that minimizes the ***generalization error or loss***:

$$\mathcal{L} := \int_{\mathcal{X}\times\mathcal{Y}} \ell(h(\boldsymbol{x}), \boldsymbol{y}) p(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{x} d\boldsymbol{y} \tag{2.2}$$

that is, the expected value of some *loss function* ℓ over all possible input-output pairs. A loss function just measures the discrepancy of the prediction $h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$ from the true value $\boldsymbol{y}$ and as such, the best hypothesis is the one that minimizes this integral. Obviously, it is impossible to evaluate the integral in EQUATION 2.2, since in general we don't know the joint probability distribution $p(\boldsymbol{x}, \boldsymbol{y})$.

The idea is to use the *training error or loss*:

$$\mathcal{L}_{\text{train}} \coloneqq \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} \ell(h(\boldsymbol{x}_i), \boldsymbol{y}_i) \tag{2.3}$$

as an approximation for the generalization loss, and *choose the hypothesis that minimizes the training loss*, a principle known as *empirical risk minimization*. In other words, to get a hypothesis aka *model* $\mathcal{M}$ from the data, we need to solve the following optimization problem:

$$\hat{h} \leftarrow \arg\min_{h \in \mathcal{H}} \mathcal{L}_{\text{train}} \tag{2.4}$$

which is achieved by just feeding the training data into the learning algorithm $\mathcal{A}$:

$$\mathcal{M} \leftarrow \mathcal{A}(\mathcal{D}_{\text{train}}) \tag{2.5}$$

2.1.3 Components of a learning algorithm

By breaking down Equation 2.4, i.e. the optimization problem the learner needs to solve, the components of a learner can be revealed. The latter is comprised of the following three "orthogonal" components: *a **hypothesis space**, a **loss function** and an **optimizer***. We now look into each of them individually and describe the contribution of each one to the solution of the optimization problem. For the ease of notation and clarity, in the remaining of this chapter we will stick to examples from simple (single-valued) regression and binary classification.

Definition 2.5 (Hypothesis space)**.** *The set of hypotheses (functions), denoted as $\mathcal{H}$, from which the learner is allowed to pick the solution of Equation 2.4.*

A simple example of a hypothesis space, is the one used in univariate *linear regression*:

$$\hat{y} = b + wx \tag{2.6}$$

where $\mathcal{H}$ contains all lines (or hyperplanes in the multivariate case) defined by Equation 2.6. Of course, one can get *a more expressive* hypothesis space, by including polynomial terms, e.g.:

$$\hat{y} = b + w_1 x + w_2 x^2 \tag{2.7}$$

The more expressive the hypothesis space, the larger the ***representational capacity*** of the learning algorithm. For a formal definition of representational capacity, the interested reader can look at *Vapnik-Chervonenkis Dimension* (Hastie et al. 2009).

Definition 2.6 (Loss function)**.** *A function that maps a prediction into a real number, which intuitively represents the quality of a candidate hypothesis.*

For example, a typical loss function used in regression is the *squared loss*:

$$\ell(\hat{y}, y) \coloneqq (\hat{y} - y)^2 \tag{2.8}$$

where $y, \hat{y} \in \mathbb{R}$. A typical loss function for binary classification is the *binary cross entropy loss*:

$$\ell(\hat{y}, y) \coloneqq y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y}) \tag{2.9}$$

where $y \in \{0, 1\}$, indicating the correct class, and $\hat{y} \in [0, 1]$ which corresponds to the predicted probability for class-1. Notice that in both cases the loss is minimum when the prediction is equal to the ground truth. For the cross entropy loss, if $y = 1$ is the correct class, then the model must predict $\hat{y} = 1$ for the loss to be minimized.

Usually, we are not only penalizing a hypothesis for its mispredictions, but also for its *complexity*. This is done in purpose, since a learner with a *very rich hypothesis space* can easily memorize the training set but fail to generalize well to new unseen examples. *Every modification that is made to a learner in order to reduce its generalization loss but not its training loss, is called* ***regularization*** (Goodfellow et al. 2016).

A common—but not the only—way to achieve that, is by including another penalty term called *regularization term or* ***regularizer***, denoted as $\mathcal{R}$, in EQUATION 2.3:

$$\mathcal{L}_{\text{train}} \coloneqq \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) + \lambda \mathcal{R} \tag{2.10}$$

The λ factor controls the strength of regularization and it is an ***hyperparameter***, i.e. a parameter that is not learned during training but *whose value is used to control the training phase.* In order to see how λ penalizes model complexity, assume we perform univariate polynomial regression of degree k:

$$\hat{y} = b + \sum_{i=1}^{k} w_i x^i \tag{2.11}$$

combining mean squared loss (MSL) and *lasso regularization* as training loss:

$$\mathcal{L}_{\text{train}} = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} \ell(\hat{y}_i - y_i)^2 + \lambda \sum_{i=1}^{k} |w_i| \tag{2.12}$$

Let's apply a very strong regularization by setting $\lambda \to \infty$ (in practice we set λ to a very large value) and observe what happens to the *weights* w_i. By setting $\lambda \to \infty$, the regularization term dominates the MSL and as such, the only way to minimize the training loss is by setting $w_i = 0$. This leave us with a very simple model—only the *bias* term b survives—which always predicts the mean value of y in the training set.

Applying a regularizer, is also useful when we need to select between two (or more) competing hypotheses that are equally good. For example, assuming two hypotheses achieve the same (unregularized) training loss, the inclusion of a regularization term help us decide between the two, *by favoring the simplest one*. This is reminiscent of the ***Occam's razor*** *aka* ***principle of parsimony***, which advocates that between two competing theories with equal explanatory power, one should prefer the one with the fewest assumptions.

DEFINITION 2.7 (Optimizer). *An algorithm that searches through $\mathcal{H}$ for the solution of EQUATION 2.4.*

Having defined the set of candidate models (the hypothesis space) and a measure that quantifies the quality of a given model (the loss function), all that is remaining is a tool to scan the hypothesis space and pick the model that minimizes the training loss (the optimizer). A naive approach is to check all hypotheses in $\mathcal{H}$ and then pick the one that achieves the lowest training loss. This approach can

work if $\mathcal{H}$ is finite, but obviously doesn't scale in the general case where $\mathcal{H}$ is infinite[1]. More efficient approaches are needed if we are aiming to solve EQUATION 2.4 in finite time.

One optimizer that is frequently used in ML and is the precursor of more refined ones, is ***gradient descent***. With this method, the exploration of hypothesis space[2] involves the following steps:

ALGORITHM 1: Gradient descent

```
θ ← random initialization;
while stopping criterion not met do
    θ ← θ − η∇_θ L_train(θ);
end
```

where η is a small number called the *learning rate*. Gradient descent is based on the idea that if a multivariate function is defined and differentiable at a point $\boldsymbol{x}$, then $f(\boldsymbol{x})$ *decreases fastest if one takes a small step from* $\boldsymbol{x}$ *in the direction of negative gradient at* $\boldsymbol{x}$, $-\nabla f(\boldsymbol{x})$.

The motivation becomes clear if we look at the differential of $f(\boldsymbol{x})$ in direction $\boldsymbol{u}$:

$$f(\boldsymbol{x} + \delta\boldsymbol{u}) - f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) \cdot \delta\boldsymbol{u} \tag{2.13}$$

EQUATION 2.13 says that this differential is minimized[3] when $\delta\boldsymbol{u}$ is anti-parallel to $\nabla f(\boldsymbol{x})$ and that is why we subtract the gradient in ALGORITHM 1, i.e. move in direction anti-parallel to the gradient. The fact that EQUATION 2.13 holds only locally (magnitude of $\delta\boldsymbol{u}$ must be small) explains why η must be a small number. It should be added that gradient descent can be trapped to (potential) local minima of the training loss and therefore fail to solve EQUATION 2.4. As it will be discussed later, this is not a problem, because *the ultimate purpose is to find a hypothesis that generalizes well, not necessarily the one that minimizes the training loss*[4]. Optimizers are discussed in further detail in SECTION 2.2.4.

Before moving on, it is worth to add that both the regularization and the optimizer have an effect on the "true" or ***effective capacity*** (Goodfellow et al. 2016) of the learner, *which might be less than the representational capacity of the hypothesis space*. For example a regularizer penalizes the complexity of an hypothesis, effectively "shrinking" the representational capacity of the hypothesis space. The effect of the optimizer can be understood by looking on its contribution to the solution of EQUATION 2.4. As described previously, the optimizer searches through the hypothesis space. If this "journey" is not long enough, then this "journey" is practically equivalent to a long "journey" in a shortened version of the original hypothesis space. In the rest of this chapter, by the term ***complexity*** *or capacity of a learner, we mean its effective capacity, which is affected by all its three components*.

2.1.4 Performance, complexity and experience

Suppose that we have trained our learner, and finally we get our model, as stated by EQUATION 2.5. *How can we assess its performance?* Remember, we can't calculate the generalization loss, since we are not given an infinite amount of data. First of all, *we should not report the training loss, because it is*

[1] It is not uncommon for $\mathcal{H}$ to be infinite. Even for simple learners like linear regression $\mathcal{H}$ is infinite, since there infinite lines defined by EQUATION 2.6.

[2] We have implicitly assumed that $\mathcal{H}$ can be parameterized, i.e. $\mathcal{H} = \{h(\boldsymbol{x}; \boldsymbol{\theta}) \mid \boldsymbol{\theta} \in \Theta\}$, where Θ denotes the parameter space, the set of all values the parameter $\boldsymbol{\theta}$ can take. This allows us to write the training loss as function of model parameters and optimize them with gradient descent.

[3] The right hand side of EQUATION 2.13 is a dot product.

[4] Remember we use the training loss (see EQUATION 2.3) as a proxy for the generalization loss (see EQUATION 2.2).

optimistically biased, as it is evaluated on the same data that has been trained on[5]. What we should is to collect new input-output pairs, forming the ***test set*** $\mathcal{D}_{\text{test}}$, and then *estimate the generalization loss* as following:

$$\mathcal{L}_{\text{test}} := \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{i \in \mathcal{D}_{\text{test}}} \ell(h(\boldsymbol{x}_i), \boldsymbol{y}_i) \tag{2.14}$$

The *test error or loss* is evaluated on new—unseen to the learner during the training phase—samples and as such, it is an *unbiased estimate* of the generalization loss. Usually, since many times is not even possible to collect new samples, *we split the initial dataset into training and test sets*.

The general recipe for building and evaluating the performance of a ML model has already been presented. What is missing is how we can improve its performance, or to put it differently, the factors that affect the quality of the returned model. There are two main factors that determine the performance of the model: *complexity and experience*. In general, the larger the experience—the training set— the better the performance, just like we humans perform improve on a task by keep practicing. With regards to the complexity, *learners of low complexity might fail to capture the patterns in the data*, meaning that the resulting model will fail to generalize. In contrast, *learners of higher complexity would be able to capture these patterns*, and as such return models of higher quality. However, *as the complexity of the learner keeps increasing, the latter is more sensitive to noise, i.e. there is a higher chance that the learner will simply memorize its experience* and as such, fail to generalize.

In other words, there is a trade-off between the complexity of the learner and its performance. The learner should be not too simple but also not too complex, in order to generalize well. This in turn implies that we need to find a way to "tune" the complexity. A common way to achieve that is by using another set of instances, known as the ***validation set***. We train learners of different complexity on the training set, evaluate their performance on the validation set, and then choose the learner that performs best on the validation set. The reason we use the validation set instead of the test set for tuning complexity, is to ensure that the performance estimation is unbiased. *The test set should not influence our decisions in any way*. After we have tuned the complexity, we can estimate the performance of the resulting model in the test set. Finally, it is a good practice to retrain the learner on the whole dataset—including validation and test sets—since more data result in models of higher quality.

Theorem 1 (Bias-variance decomposition, Bishop 2006)**.** *From Equation 2.1 and under the assumption that* $\epsilon \sim \mathcal{N}(0, 1)$*, the expected squared loss at* $\boldsymbol{x}^*$ *can be decomposed as following:*

$$\mathbb{E}\left[\left(y^* - \hat{f}(\boldsymbol{x}^*)\right)^2\right] = \left(f(\boldsymbol{x}^*) - \mathbb{E}\left[\hat{f}(\boldsymbol{x}^*)\right]\right)^2 + \mathbb{E}\left[\left(\hat{f}(\boldsymbol{x}^*) - \mathbb{E}\left[\hat{f}(\boldsymbol{x}^*)\right]\right)^2\right] + \sigma_\epsilon^2 \tag{2.15}$$

The expected squared loss refers to the average squared loss we would obtain by repeatedly estimating f *using different training sets, each tested at* $\boldsymbol{x}^*$*. The overall expected squared loss can be computed by averaging the left hand side of Equation 2.15 over all possible values* $\boldsymbol{x}^*$*.*

The trade-off between the complexity of the learner and its performance, is mathematically described in Theorem 1. Equation 2.15 states that the error of the learner can be decomposed into three terms: ***bias***, ***variance*** *and* ***irreducible error***. The bias (squared)—first term of Equation

[5] Intuitively, this is like assessing students' performance based on problems they have already seen before. They can easily achieve zero error, just by recalling their memory.

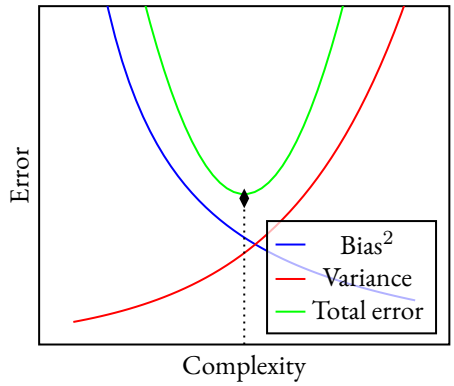


Figure 2.2: The bias-variance trade-off. For a given task, there is a "sweetspot" of complexity that minimizes the total error. Bias^2 and variance correspond to the first and second term of Equation 2.15, respectively.

2.15—refers to the error introduced by *approximating a real-world problem, which can be highly complicated, by a much simpler model* (Hastie et al. 2009; James et al. 2014). For instance, if the input-output relationship is highly nonlinearity, using linear regression to approximate f, will undoubtedly introduce some bias in the estimate of f. In contrast, if the input-output relationship is very close to linear, linear regression should be able to produce an accurate estimate of f. In general, more flexible learners, result in less bias (Hastie et al. 2009; James et al. 2014).

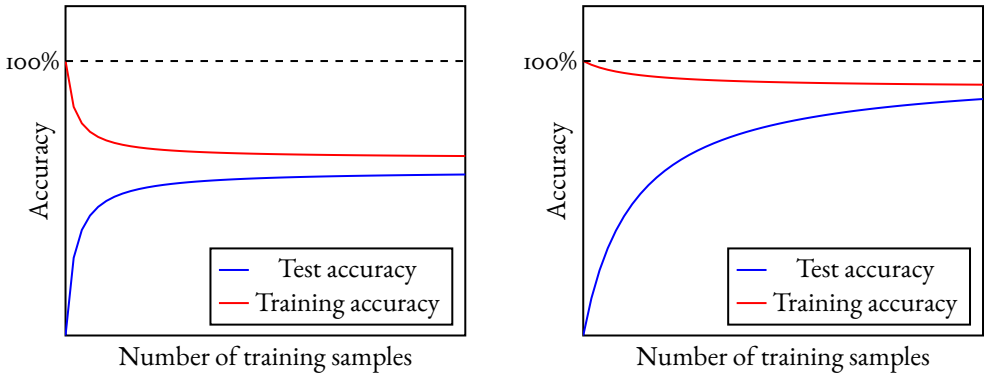
The variance—second term of Equation 2.15—refers to the *degree by which $\hat{f}$ would change if it was estimated by different training sets*. Since the training data are used to fit the learning algorithm, different training sets will result in a different estimate of f. Ideally, $\hat{f}$ should not exhibit too much variation between different training sets, since otherwise small changes in the training can result in large changes in $\hat{f}$. In that case, the learner essentially memorizes the training data. Generally, more flexible learners, result in higher variance (Hastie et al. 2009; James et al. 2014).

Lastly, the irreducible error—third term of Equation 2.15—refers to the *error caused by stochastic label noise*, as can be seen from Equation 2.1. A possible source for this noise, might be ommited features which are useful in predicting the output. It is called irreducible, because no matter how well we estimate f, even if we predict $\hat{y} = f(\boldsymbol{x})$, we can't reduce the error associated to the variability of ϵ. As stated in Section 2.1.2, this random error term is indepedent of $\boldsymbol{x}$, and as such, we have no control over it. The *bias-variance trade-off* is schematically depicted in Figure 2.2. Interested readers might also appreciate reading about *double descent* (Nakkiran et al. 2019), a phenomenon where increasing further the complexity of the learner, results in a new minimum (hence, the name).

Figure 2.3, shows the *learning curves* for learners of different complexity. A learning curve is a plot of the training and test performance[6] of the learner as function of its experience. First, let's look at the learning curve of the low complexity learner. The accuracy[7] starts out high on the training set, since with a small number of samples, the learner can fit them perfectly. However, by adding more training data, learner's training accuracy quickly drops due to learners inflexibility and inexpressivity.

[6] Usually, only the test performance is plotted.

[7] By accuracy we mean any performance metric where higher values are better, not necessarily the classification accuracy.



(A) Learning curve of learner with low complexity. **(B)** Learning curve of learner with high complexity.

FIGURE 2.3: Relation between performance and experience.

That is, it can't fit the patterns in the training data. On the other hand, test accuracy starts out very low since with very few training data, it is unlikely that the training set is a good representation of the underlying distribution $p(\boldsymbol{x}, \boldsymbol{y})$. In other words, it is unlikely that the learner will experience patterns in the training data, that will help it to generalize well. By increasing the training data, test accuracy increases but it never reaches a high value. This happens due to the learners inability to detect and exploit the patterns in the training data. In other words, the learner fails to generalize well not because its experience is low, but because it is biased. That is, it oversimplifies the problem and make strong assumptions that do not capture the complexity of the data.

Now let's consider the learning curve of the high complexity learner. Again, the training accuracy starts out high with a small amount of training data. However, in contrast to the previous case, as the number of training samples increases, the training accuracy remains high since the learner is flexible enough to learn the patterns in the training set, irrespective of its size. At some point, the training data becomes large enough that is a good representation of the underlying distribution $p(\boldsymbol{x}, \boldsymbol{y})$ and since the learner is very flexible, it can capture the true patterns in $p(\boldsymbol{x}, \boldsymbol{y})$, increasing the test accuracy. It is worth pointing out that the learning and complexity curves (see FIGURE 2.2), are just two slices of the same 3D plot: the plot of performance as function of experience and complexity.

2.2 Fundamentals of Deep Learning

Having covered the basic jargon and concepts of ML, we are now in a position to dive into DL. One might expect that DL is a very complex subfield of ML, given its astonishing results in complex tasks, but quite the opposite holds. Notably, DL shares similar ideas with reticular chemistry: *combining simple computational units, known as* ***neurons****, to achieve intelligent behavior*. And just like we can tune the properties of MOFs by judiciously selecting and combining their building blocks, we can design problem-specific *neural* ***architectures*** by reasonably arranging and connecting the neurons. In other words, both DL and reticular chemistry can be viewed as building with Legos.

Since the term "neuron" is admittedly a neuroscience term, one might wondering what is the relation between DL and the human brain. The neural perspective on DL is mainly motivated by the

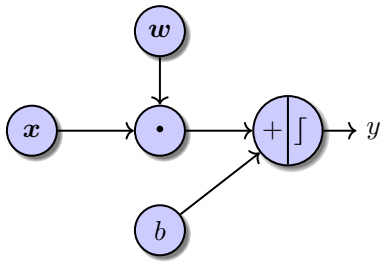


Figure 2.4: The perceptron.

following idea: *the brain is a proof by example that intelligent behavior is possible and as such, a straightforward approach to build an intelligent system is by reverse engineering the computational principles behind the brain and duplicating its functionality*. However, the term "deep learning" is not limited to this neuroscientific perspective. It appeals to a more general principle of learning *multiple levels of abstraction*, which is applicable in ML frameworks that are not necessary neurally inspired (Goodfellow et al. 2016).

Definition 2.8 (Deep learning)**.** *Class of machine learning algorithms inspired by brain organization, based on learning multiple levels of representation and abstraction. They achieve great power by learning to represent the world*[8] *as a nested hierarchy of concepts.*

Before exploring NNs we first need to understand how the neuron, the basic building block of NNs, works. *A neuron is nothing more than a device—**a simple computational unit**—that makes decisions by weighing up evidence* (Nielsen 2018). This sounds very similar to the way humans make decisions. For instance, suppose the weekend is coming up and your favorite singer has scheduled a concert near your city. In order to decide whether you should go to the concert or not, you weigh up different factors, such as weather conditions, ease of transportation (you don't own a car) and whether your boyfriend or girlfriend is willing to accompany you. This kind of decision-making can be described mathematically as following:

$$\text{decision} = \begin{cases} 1 & \text{if } \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \quad \boldsymbol{w}^\top \boldsymbol{x} := \sum_i w_i x_i \tag{2.16}$$

If this weighted sum plus the bias[9]—your willing to to go to the festival irrespective of the evidence—is greater than zero, then your decision is positive, otherwise negative.

The simple decision-making rule specified by Equation 2.16, which is known as the ***perceptron*** (Rosenblatt 1957), is schematically depicted in Figure 2.4. Essentially, the perceptron is a linear binary classifier (see Figure 2.1a). If we pay a little more attention to Equation 2.16, we can see that that the decision is basically an *application of a linear function*[10] *followed by a nonlinearity*. As such, we can rewrite Equation 2.16 as following:

$$y = \phi\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \tag{2.17}$$

[8] Hierarchy is deeply rooted in our world. Just think the hierarchy from subatomic particles to macroscopic objects.
[9] If you prefer the neuroscientific analogy, you can think of bias as how easy is for a neuron to "fire".
[10] Formally speaking it is an affine function. We can turn it into a linear by "absorbing" the bias term into the weights and adding 1 to the input vector, a procedure known as the *bias trick*.

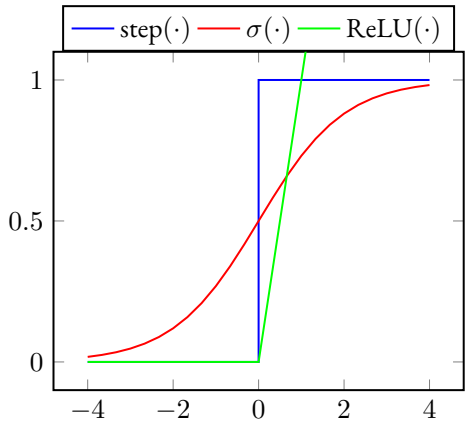


Figure 2.5: Examples of activation functions.

where $\phi(\cdot)$ is the nonlinear function aka ***activation function***.

In the perceptron, the activation function is the Heavyside step function but in modern NNs it has been substituted by functions such as the sigmoid, hyperbolic tangent and currently the rectified linear unit (ReLU) and its variants. Some activation functions are graphically shown in Figure 2.5. The reason that the step function isn't used anymore is that its derivative vanishes everywhere, which is problematic for gradient-based optimization methods that power the training of modern NNs. The ReLU function is defined as:

$$\mathrm{ReLU}(x) \coloneqq \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.18}$$

A common variant of ReLU is the leaky rectified linear unit (LeakyReLU) function which is defined as:

$$\mathrm{LeakyReLU}(x) \coloneqq \max(0, x) + a \min(0, x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{2.19}$$

where a is a small positive constant usually set to 0.01. It is worth to notice how simple the nonlinearities used in NNs are. For instance, ReLU, the most commonly used activation function these days, is just a piecewise linear function. This again highlights the fundamental idea behind DL: *building something complex by combining simple elements*.

2.2.1 Neural networks

Neural networks can be thought as ***collection of neurons*** *organized in layers* and can be represented as *computational graphs*.

Definition 2.9 (Graph)**.** *A set of objects in which some pairs of objects are in some sense "related". See Figure 2.7 for some types of graphs.*

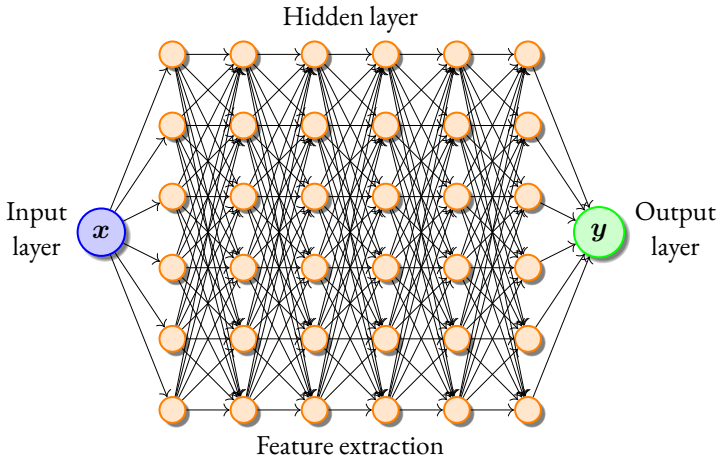
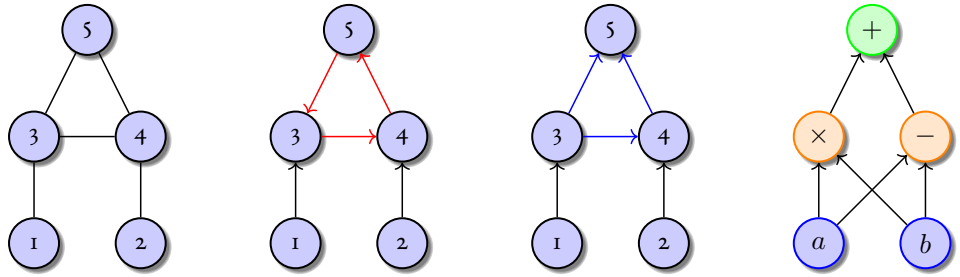


Figure 2.6: The multilayer perceptron. A typical example of a neural network.



(A) Undirected graph. **(B)** Directed cyclic graph. **(C)** Directed acyclic graph. **(D)** Computational graph.

Figure 2.7: Examples of graphs. In a directed graph the edges have direction. If at least one loop is present, they are called cyclic, otherwise acyclic.

Definition 2.10 (Computational graph). *A directed acyclic graph (DAG) where nodes correspond to operations or variables and edges show the data flow between the nodes.*

In general, the architecture of a NN can be broken down into the following three layers: ***input layer***, ***hidden layer*** *and* ***output layer***. Information flow starts from the input layer, passes through the hidden layer(s) and finally ends at the output layer. Neural networks with more than one hidden layer are classified as deep, and shallow otherwise. A typical architecture known as multilayer perceptron (MLP)[11] or fully connected neural network (FCNN) is presented in Figure 2.6. It should be emphasized that all the neurons in the hidden layers aka *hidden units* of the network perform exactly the same operation as that of the perceptron, described in Equation 2.17. As such all the *hidden units at a given layer make decisions based on the decisions of the previous layer*. Unsurprisingly, the kind of decisions made by the neurons depends solely on the problem and the data distribution at hand.

[11] The term MLP is kind of a misnomer, since the step function used originally in the perceptron is no longer used in modern NNs.

To understand better the purpose of the hidden layers and the functionality of a NN as a whole, let's consider the problem of image classification. This is by no means a trivial task, since we need to learn a mapping from a set of pixels to an object identity. Imagine for a moment you are blindfolded and you need to classify an image. The valid classes are: "person", "car" and "ship". Furthermore, assume that the correct class is "person". *Would you prefer to hear the sequence of pixel values or whether the image contains a face?* In other words, it is a lot easier to classify the content of an image if we know some *high-level features, i.e. a high-level description of the image*. *Neural networks extract such high-level features by exploiting the hierarchical structure of images*. A complex object like a "face" is defined in term of simpler ones, such as "eye" and "nose", which in turn are defined in terms of simpler ones and so on. This hierarchy allows the NN to solve the complex task of image classification by breaking it down into smaller sub-problems. The first layer learns to detect edges. The second layer combines the decisions of the first layer to detect corners. Subsequently, the third layer combines the decisions of the second layer to identify shapes like circles and squares and so on, until we reach the final layer which is able to detect high-level features such as objects or object parts. The deeper we are into the network—i.e. the closer to the output layer—the more abstract and task-specific the detected features become.

In a FCNN with N hidden layers, each hidden layer performs the following operation:

$$\boldsymbol{h}^l = \boldsymbol{\phi}\left(W^l \boldsymbol{h}^{l-1} + \boldsymbol{b}^l\right) \quad \text{where} \quad 1 \leq l \leq N \quad \text{and} \quad \boldsymbol{h}^0 := \boldsymbol{x} \tag{2.20}$$

which is just a matrix version of Equation 2.17 with the matrix W^l playing the role of the "synapses" between the neurons of the layers $l-1$ and l. Since the "stacking" of many hidden layers is equivalent to a huge composite function:

$$\boldsymbol{\tau}(\boldsymbol{x}) := \left(\boldsymbol{h}^l \circ \boldsymbol{h}^{l-1} \cdots \circ \boldsymbol{h}^1\right)(\boldsymbol{x}) \tag{2.21}$$

and the output layer is just a linear function of the last hidden layer, the output of the FCNN can be written as:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{\tau}(\boldsymbol{x}) + b \tag{2.22}$$

or in the general case of multi-valued output:

$$\hat{\boldsymbol{y}} = W\boldsymbol{\tau}(\boldsymbol{x}) + \boldsymbol{b} \tag{2.23}$$

In other words, *a linear model on top of the extracted features*. It should be emphasided that Equation 2.23 is not specific to FCNNs, but describes every type of NN used for classification and regression. Moreover, the use of activation function now becomes more clear: *the composition of many linear functions is just another linear function, which implies nonlinearities must be inserted between them, if we aim to learn a nonlinear relationship*. Equation 2.23 can also be understood in the following way: *a problem that is nonlinear—i.e. complex—in the original space, can become linear—i.e. simple—in a transformed space*. Figure 2.8 shows such an example, known as the XOR problem. The solution of Equation 2.23 essentially boils down to finding the right transformation funciton $\boldsymbol{\tau}(\boldsymbol{x})$. Please note that traditional ML algorithms like support vector machines (SVMs), also map the original space to a transformed space. However, they use a *fixed—i.e. not learnable during the training phase—mapping. Deep learning algorithms on the other hand learn this mapping during their training phase, considering both the problem and the data at hand*.

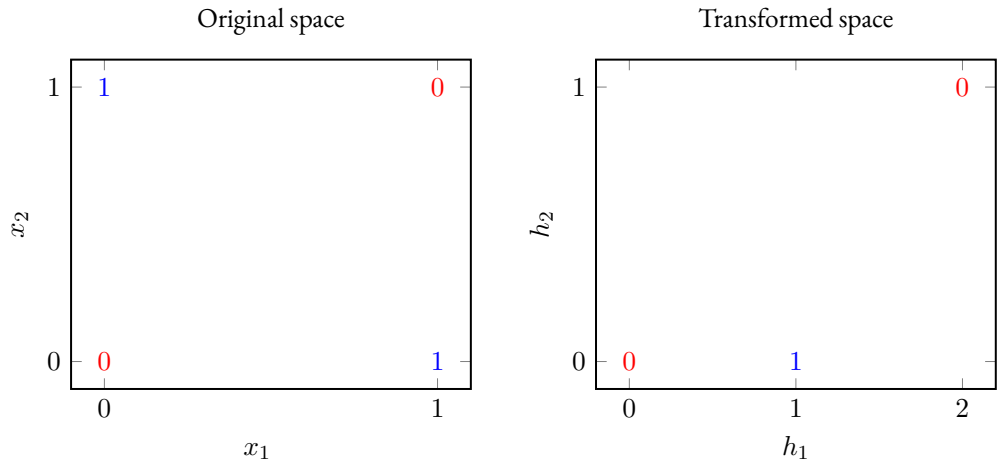


Figure 2.8: Solving the XOR problem. A linear classifier in the original space can't perfectly seperate the "ones" and "zeros". In contrast, if the points are projected into a new space, then they become linearly separable.

Theorem 2 (Universal approximation theorem, Hornik et al. 1989)**.** *A feedforward [12] network with a single hidden layer containing a finite number of neurons can approximate any continuous function.*

One interesting fact about NN is summarized in Theorem 2. An informal proof goes like this. The value of a function f at point x can be viewed as a "spike" or a "bump" at point x with height $f(x)$. If we put a "bump" at every point x we have essentially recovered the function f. The question now boils down to wheter a NN can create "bumps". The answer is affirmative, and a visual proof of the "bump" construction and Theorem 2 is provided by Nielsen 2018.

The universal approximation theorem implies that irrespective of the function we are trying to learn, a network with just one hidden layer and sufficient number of hidden units can represent this function. However, the theorem does not tell us two important things. First, the number of neurons required for the problem we are aiming to solve. Second, *whether we can learn the function at all* (Goodfellow et al. 2016). Learning the true function can fail since the optimizer is not guaranteed to pick the solution that minimizes the generalization loss. Remember it optimizes the training loss as a proxy for the generalization loss.

Theorem 3 (No free lunch theorem, Wolpert 1996)**.** *Averaged over all possible data generating distributions, every learner has the same error rate when predicting previously unobserved points.*

Finally, we close this section with Theorem 3. In essence, it states that *no learner is universally better than any other*. Please note that by "learner" we mean even "dummy" learners, such as random guessing. In other words, the more sophisticated learning algorithm we can conceive has the same

[12] In feedforward newtworks information flows from input to output. In contrast, feedback aka recurrent networks allow the information to travel in both directions by introducing loops. Computations derived from earlier input are fed back into the network, which gives them a kind of memory. This kind of NNs find application in natural language processing (NLP) tasks, such as text generation or classification. Note that although they contain loops, and as such they are not DAGs, we can convert them to DAGs by "unrolling" their computational graph (LeCun et al. 2015).

average performance (over all tasks) as random guessing. While THEOREM 3 seems unintuitive at first glance, it may be easier to understand it with an example. Suppose we see one sheep, and then expect to see another one. We could devise the following strategies (learning algorithms) for predicting the color of the second sheep:

$$\text{strategies} = \left\{ \begin{array}{ll} \text{Same color as the first} & \text{(white, white)} \\ \text{Different color than the first} & \text{(black, black)} \\ \text{Always black} & \text{(white, black)} \\ \text{Always white} & \text{(black, white)} \end{array} \right\} = \text{possible worlds} \tag{2.24}$$

Assuming that all possible worlds (data generating distributions) *are equally likely*, then each strategy has the same expected error: 50%. Fortunately, *in real-world this assumption breaks down*. For example, animals tend to be the same color, so the worlds where the first and the second sheep have different colors are unlikely. In this scenario, guessing the same color as the first is more likely to be correct. Every learning algorithm is equipped with an ***inductive bias***, *that is a set of assumptions about the underlying data distribution*[13]. Whether algorithm $\mathcal{A}_1$ will outperform $\mathcal{A}_2$ on problem $\mathcal{P}$, is just a matter of whose assumptions are better aligned with the structure of $\mathcal{P}$.

2.2.2 Regularizing neural networks

Deep NNs typically have hundreds of thousands, million, or even billion parameters. With such a huge number of parameters, they are capable of memorizing a huge variety of complex training sets. However, memorization is harmful from a generalization point of view. Therefore, it is necessary to employ regularization techniques when training deep NNs, especially when the size of the training set is relatively small. Besides adding penalty terms to the training loss as described in SECTION 2.1.3, common regularization techniques include: ***data augmentation*** *and* ***dropout***.

Data augmentation, as the name implies, is a technique to artificially increase the size of the data set. With this technique, each training instance is replicated many times with *each replicate being randomly distorted in such a way that the corresponding label is left unchanged*[14]. For example, in image classification the original images can undergo *geometric transformations* such as rotations, vertical or horizontal flips, small shifts and random crops. Such kind of transformations force the NN to be more tolerant to variations in orientation, position and size. Another way to augment the original images is by applying *color transformations*, such as changing the brightness or contrast of the image, increasing the NN's tolerance in different lighting conditions. Of course, we can further augment our training set by composing geometric and color transformations. The performance boost thanks to data augmentation can be understood in two ways: i). More data is better. ii). As introducing a useful inductive bias. That is, *we know a priori that the true function ought to be invariant in certain transformations, and the augmented images are a way of imposing this knowledge*.

[13] For example, the *composition of layers* in NNs provides a type of relational inductive bias: *hierarchical processing*. Another example of inductive bias is the linear relationship assumed in linear regression.

[14] We should be careful on how we augment the training set. For example, in character classification tasks there is difference between the characters "b" ↔ "d" and "6" ↔ "9". As such, horizontal flips and 180° rotations are not advisable for this task.

Dropout (Hinton et al. 2012; Srivastava et al. 2014) is a powerful and computational inexpensive method to regularize neural networks, which has proven to be extremely successful. Even the state-of-the-art architectures improved by 1–2 % when dropout was added to them. This may not sound like a lot, but if we consider a model that has already 95 % accuracy, a performance boost of 2 % amounts to 40 % drop in the error rate (going from 5 % to 3 %). Let's see how it works: at every training step the neurons of a hidden layer can be temporarily "dropped out" with probability p aka *dropout rate*, meaning that they will entirely ignored during this training step, but may become active again in the next training step. And that's it except for one technical detail. Dropout is applied only during training and as such, all neurons are active during testing. This means that during this phase a neuron will receive a different amount of input signal compared to its (average) input signal during training. For example, if the dropout rate is set to 0.5, then during testing a neuron will be connected to twice as many input neurons as it was during training (on average). To compensate for that, the neuron's input connections must be multiplied by 0.5 during testing. Otherwise, each neuron in the network will receive a total input signal roughly twice as large as what it was trained on, meaning that each neuron and the network as whole won't perform well. In general, during testing we need to multiply each input connection by the *keep probability* $(1 - p)$. Another way[15] to retain the same amount of input signal for both training and testing phases, is by just dividing each neuron's output with the keep probability, a technique known as *inverted dropout*.

It is quite surprising that this random "resurrection" of neurons improves the performance of the network. Dropout works because it *forces neurons to pay attention to all of their inputs, rather than relying exclusively on just a few of them, making them robust to the loss of any individual piece of evidence*. Or to put it differently, it strengthens them by forcing them to "live" in a stochastic handicap environment. Another way to understand dropout is the following. When different sets of neurons are "dropped out", it is like we are training different NNs. That is, the dropout procedure acts as an average of these different networks. The latter will overfit in different ways and so the net effect of dropout will hopefully mitigate this effect.

2.2.3 Convolutional neural networks

Convolutional neural networks are specialized NN architectures to process image-like data and in general, data which exhibit spatio-temporal relationships. They find application in tasks such as text, audio, video and image classification, semantic segmentation and object detection, just to name a few. As their name implies, CNNs make use of the ***convolution*** operation, so to understand the former we first need to understand the latter.

The convolution between the functions f and g, denoted as $f * g$, is defined as following:

$$(f * g)(t) \coloneqq \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \tag{2.25}$$

which is the integral of the product of f and g when g is reflected about the y-axis and shifted. In CNN terminology the first argument of the convolution operation is referred to as the *input* and the second one as the ***kernel*** *or* ***filter***. What is the interpretation? It is just a *moving inner product between*

[15] Note that these alternative are not perfectly equivalent, but in practice they work equally well.

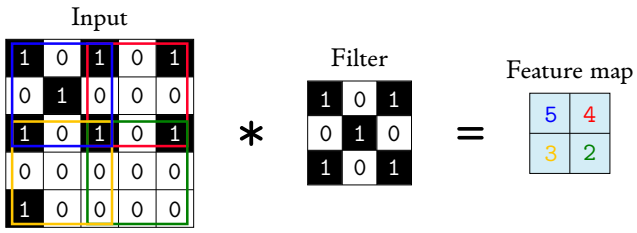


FIGURE 2.9: Convolution operation. Convolving a filter with an image can be seen as template matching. When a local image patch matches the filter—template to be matched—the output in the feature map is highly positive. Sliding of the filter over the image and recording the output of this template-patch similarity, produces the feature map.

the two functions. Recall that the inner product between two vectors $\boldsymbol{f}$ and $\boldsymbol{g}$ is defined as:

$$\boldsymbol{f} \cdot \boldsymbol{g} \coloneqq \sum_i f_i g_i \tag{2.26}$$

and can be viewed as *a measure of similarity* between $\boldsymbol{f}$ and $\boldsymbol{g}$. Conceptualizing functions as infinite-dimensional vectors and ignoring the reflection part of convolution[16] let us understand the latter as the inner products between f and shifted versions of g by t. If g represents a pattern—usually local—we are interested in to detect in the signal f, then the output of convolution known as ***feature map*** in the DL jargon, essentially tell us where (hence the term "map") the pattern described by g is located in the signal f.

Usually when we work with data on a computer, the index t will be discretized, meaning that it can take only integer values. Assuming that f and g are defined only for integer t, the discrete convolution is defined as following:

$$(f * g)(t) \coloneqq \sum_{\tau=-\infty}^{\infty} f(\tau) g(t-\tau) \tag{2.27}$$

In ML applications, the input and the filter are usually multidimensional arrays aka *tensors*. Because these two tensors must be explicitly stored separately, we treat the input and the kernel as functions that are zero everywhere except the finite set of points for which their values are stored (Goodfellow et al. 2016).

Please note that we can and often use convolutions over more than one axis at a time. A typical example is when the input is a 2D image. In that case, the convolution between the image I and the filter K is defined as:

$$(I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n) \tag{2.28}$$

and is schematically shown in FIGURE 2.9. The reason that the flipping of the filter is not necessary in ML applications, is simply because *the values of the filter are adapted, i.e. learned, during the training*

[16] As it will be latter discussed, whether the kernel is reflected or not, doesn't make a difference for ML applications.

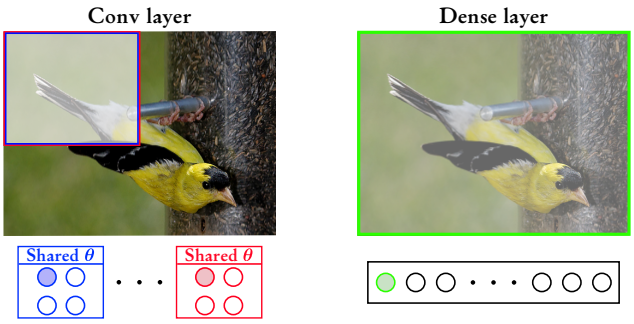


FIGURE 2.10: Neurons' arrangement in convolutional and dense layers. In convolutional layers, neurons are organized into groups and share the same parameters. The receptive field of each member is a small region of the input. In contrast, the receptive field of neurons in a dense layer is the whole input and there is no parameter sharing. Note that the receptive field of neurons incrementally increases as we transition to convolutional layers deeper in the network.

phase. Many DL frameworks instead of the convolution implement a related operation called *cross-correlation*, which is the same as convolution but without flipping the filter. We will stick to this convention for the rest of this section.

Now that the convolution operation has been presented, we can appreciate its contribution to the mechanics of a CNN. *Convolution introduces a beneficial inductive bias to the network, namely* ***sparse connections*** *and* ***parameter sharing***, as shown in FIGURE 2.10, and that's why CNNs outperform FCNNs in object recognition tasks. Sparse connections express the prior knowledge that closely placed pixels are related to each other or to put it differently, *local features such as edges are useful to understand images*. On the other hand, parameter sharing encodes the idea that *a feature detector that is useful in one part of the image is also useful in another part of the image*. Thanks to parameter sharing, once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a FCNN has learned to recognize a pattern in one location, it can recognize it only in that particular location (Géron 2017).

The basic building block of a CNN is the *convolutional layer* which contains many *learnable convolutional filters*, each of which is a template that determines whether a particular local pattern is present in an image. It should be emphasized that a feature map of a given layer combines all the feature maps of the previous layer or just the raw image (in the case of the 1-st convolutional layer). This means that if the layers $t-1$ and t contain n and m feature maps, respectively, then the layer t must learn $m \times n$ filters. By stacking many such layers a CNN extract features hierarchically, with the level of (feature) abstraction increasing the deeper we go into the network.

It is worth to mention that CNNs are essentially regularized FCNNs, meaning that the latter can learn to behave like the former. The catch is that they will probably need a great amount of training data. To understand why this is the case, let's examine the horizontal edge detection problem. The CNN can learn to detect horizontal edges *at any position* by adjusting the values for one of its filters to

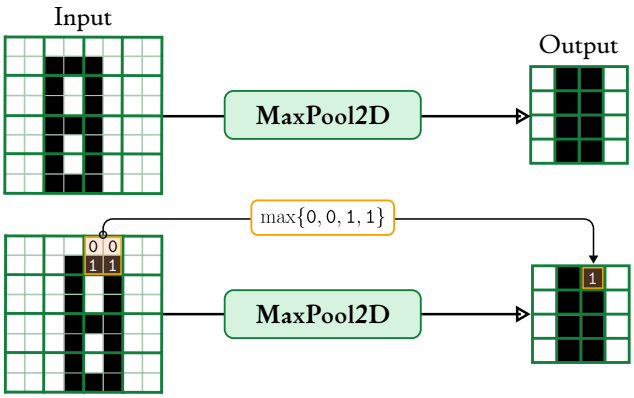


Figure 2.11: Max pooling operation. Small translations to the input (input B is just a shifted version of input A by one pixel to the right) produce the same output when passed through the max pooling layer, meaning that the latter introduces into the network some level of invariance to small translations.

the following ones[17]:

$$K_x^{\text{edge}} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.29}$$

What about the FCNN? Figure 2.9 gives the answer. A neuron in a fully connected aka *dense layer* can also learn to detect edges *by just zeroing the weights for its inputs except the small region where the edge needs to be detected*[18]. For this region, it must learn the weights specified by Equation 2.29. The problem is that the aforementioned zeroing of weights and the non-zero weights themselves, must be learned by many other neurons for horizontal edges to be detected at different positions, which is of course not guaranteed with limited amount of training data.

Besides convolutional layers, another typical of building block of CNNs are the ***pooling layers***. Their role is to downsample (reduce the resolution) in a parameter-free way the feature maps produced by convolutional layers. By downsampling in this manner, they reduce the memory-computational footprint of the CNN and also the number of parameters, thereby reducing the risk of overfitting (Géron 2017). A pooling layer takes as inputs the feature maps of the preceding convolutional layer and subsamples them by substituting the outputs in a small neighborhood of the feature map with a summary (Goodfellow et al. 2016). Figure 2.11 illustrates a common type of pooling, known as ***max pooling***, which uses the max function to compute the summary statistic. Another type of pooling is *average pooling*, which computes the summary statistic through averaging. Just like convolution, the idea of pooling generalizes to more than two dimensions.

[17] Recall that an "edge" is nothing else than a significant local change in the image intensity. Based on the formula of symmetric derivative: $\partial_x f \propto f(x+h) + 0 \cdot f(x) - f(x-h)$. In essence, by convolving an image with the filter specified in Equation 2.29, we calculate the gradient along the x-axis in a discretized fashion. Detecting vertical or adges along any direction follows the same idea. You can play with different filters here.

[18] We can view each neuron in a dense layer as performing convolution with a kernel size as large as the input.

2.2.4 Training neural networks

Training deep NNs might seem like a nightmare, given their enormous number of parameters. However, modern techniques make it possible to train very deep networks very efficiently. We will start this section by discussing some of the challenges one might face when training deep NNs and then move to describe the optimizers which take the hard job of finding model parameters that hopefully will generalize well. The training of NNs essentially boils down to the following two steps: i). Initialize model parameters ii). Update model parameters.

Therefore, the first question that must be answered is how the parameters must be initialized. First of all, it is important that *all weights are initialized randomly, otherwise training will fail*. For instance, suppose that all weights and biases are initialized to the same constant value, e.g. zero. *Then, all neurons in a given layer will be perfectly identical and thus, any update of the parameters will affect the in exactly the same way*. That is, despite having hundreds or thousands of neurons per layer, the model will end up having "duplicates" of a single neuron in each layer, or to put it differently, *it will act as if it had only one neuron per layer* (Géron 2017). On the contrary, the random initialization of weights *breaks the symmetry* and makes it possible to train a diverse team of neurons. What about the biases? Well, since the asymmetry breaking is already provided by the random initialization of the weights, it is possible and common to initialize the biases to be zero.

It is important that random initialization is performed carefully, otherwise we may end up with ***vanishing*** *or* ***exploding gradients***. Since we need the gradients with respect to the parameters in order to update them, if these gradients vanish—i.e. take very small values—then the learning will be very slow. On the other hand, if they explode—i.e. take very large values—then the training either stops due to NaNs or diverges. For simplicity, but without loss of generality consider a NN with N hidden layers and one neuron per layer. The gradient of the loss with respect to the layer l found k steps behind the final hidden layer, equals:

$$\frac{\partial \mathcal{L}_{\text{train}}}{\partial w^l} = \frac{\partial \mathcal{L}_{\text{train}}}{\partial y} \cdot \frac{\partial y}{\partial h^n} \cdot \left(\prod_{i=0}^{k} \frac{\partial h^{n-i}}{\partial h^{n-i-1}} \right) \cdot \frac{\partial h^{n-k}}{\partial w^{n-k}} \quad \text{where} \quad n-k=l \tag{2.30}$$

The origin of vanishing and exploding gradients is rooted in the middle term of Equation 2.30, which can be expanded to:

$$\prod_{i=0}^{k} \frac{\partial h^{n-i}}{\partial h^{n-i-1}} = \prod_{i=0}^{k} \phi'(z^{n-i}) w^{n-i} \tag{2.31}$$

The right hand side of Equation 2.31 essentially says that the *calculation of gradient involves a series of multiplications*. If the terms involved are all greater than one, then we end up with a very large gradient. In contrast, if all the terms are smaller than one, then the gradient vanishes[19]. These effects are more pronounced for earlier layers.

How can we avoid these problems? In essence, we need to find a way to *control the magnitude of this product*. First, notice that Equation 2.31 besides the values of weights involves also the derivative

[19] A toy example to understand the problem of unstable gradients is the following. First, replace the activation function $\phi(\cdot)$ with the identity function, i.e. $\phi(z) = z$, then set all weights to the same value w (either smaller or greater than one). In that case, the right hand side of Equation 2.31 reduces to w^k which for $w = 1.2$ and $k = 50$ is approximately equal to 9.1×10^4.

values of the activation function $\phi(\cdot)$. If these are smaller than one, then vanishing gradients are more likely to appear. This is the case when the sigmoid is used as activation function, since the maximum value of its derivative equals 0.25. For this reason, the sigmoid is no longer used as activation function in modern NN architectures. The hyperbolic tangent was used as a replacement but due to its derivative taking values extremely close to zero for large inputs, has given its position to ReLU and its variants. Nevertheless, even if we use ReLU the risk of unstable gradients remains due to the repeated multiplications of weights in Equation 2.31. An answer that elegantly accounts for all these subtleties was given by He et al. 2015, who proposed[20] an initialization scheme that mitigates the problem of unstable gradients, at least in the early stages of training. For a FCNN with ReLU as activation function this scheme suggests[21]:

$$W_{ij}^{l} \sim \mathcal{N}\left(0, \frac{2}{n^{l-1}}\right) \tag{2.32}$$

where n^l denotes the size (i.e. number of neurons) of layer l.

Although proper initialization can significantly reduce the risk of unstable gradients at the beginning of the training, it doesn't guarantee that they won't come back later on. The reason is as the training training phase proceeds, the parameters keep changing, so the initial control over the product in Equation 2.31 is lost. A techique called ***batch normalization*** (Ioffe et al. 2015) was proposed to address the problem of unstable gradients and its steps are summarized in Algorithm 2. A batch normalization layer is inserted between or after the activation function of each hidden layer and it contains two learnable parameters that allow the model to learn the optimal scale and mean of each of the layer's inputs.

Algorithm 2: Batch normalization

Input: Values of x_i over mini-batch $\mathcal{B} \subset \mathcal{D}_{\text{train}}$, learnable parameters γ and β

`/* Mini-batch mean */`

1 $\mu_{|\mathcal{B}|} = \frac{1}{\mathcal{B}} \sum_i x_i$;

`/* Mini-batch variance */`

2 $\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_i (x_i - \mu_{\mathcal{B}})^2$;

`/* Normalize */`

3 $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$;

`/* Scale and shift */`

4 $z_i \leftarrow \gamma \widehat{x}_i + \beta$;

5 **return** z_i *for all samples in mini-batch* $\mathcal{B}$

The authors demonstrated that the addition of batch normalization layers improved all NNs they experimented with, leading to significant improvements in the ImageNet classification task. Moreover, the vanishing gradients problem was significantly reduced, making even possible the use of tanh and sigmoid as activation functions. Adding batch normalization layers decreased the sensitivity to weight initialization and also allowed the use of higher learning rates, accelerating the learning process. Additionaly, since the mean and the variance are calculated over mini-batches, batch normalization

[20] Similar ideas were proposed by Glorot et al. 2010 five years earlier for NNs with tanh as activation function, when the latter was still a common choice.

[21] Similar scheme applies for CNNs and ReLU variants.

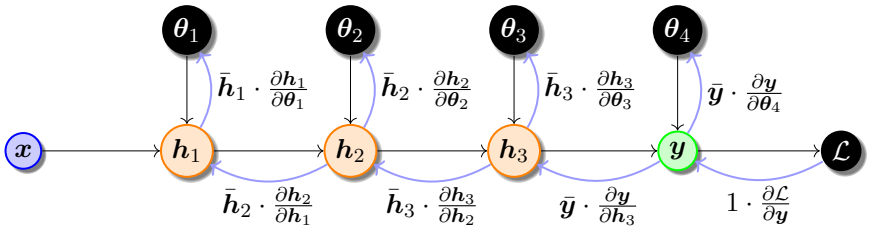


Figure 2.12: Illustration of back-propagation. The black and blue arrows show the information flow during the forward and backward pass, respectively. The notation $\bar{\boldsymbol{z}}$ denotes the partial derivative of the loss with respect to the node $\boldsymbol{z}$. That is, $\bar{\boldsymbol{z}} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$.

brings a regularization effect due to noisy estimates. It should be noted that during testing the mean and variance are obtained from a running average of the values seen during training.

Training NNs efficiently requires that the calculation of the gradient is extremely fast. A naive approach to calculate the gradient of the training loss with respect to model parameters is based on the definition of the gradient:

$$\frac{\partial \mathcal{L}}{\partial \theta_{ij}} \approx \frac{\mathcal{L}(\boldsymbol{\theta} + h\boldsymbol{e}_{ij}) - \mathcal{L}(\boldsymbol{\theta})}{h} \tag{2.33}$$

That is, first the training loss for a given parameter configuration $\boldsymbol{\theta}$ is calculated by performing a *forward pass* of our training data through the network. Then, we perturb each parameter θ_{ij} by a small amount, perform another forward pass and compare it with the initial loss. The main drawback of this approach is that we need to *perform as many forward passes as the number of model parameters*. Such computational overhead is of course prohibitive for training modern NNs.

Since a NN is ultimately a huge composite function, a more refined approach is to employ the tools of calculus and calculate the derivatives based on the *chain rule*. For a network with 3 hidden layers this amounts to the following calculations:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_4} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}_4} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_3} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_3} \cdot \frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{\theta}_3} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_3} \cdot \frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2} \cdot \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{\theta}_2} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}_3} \cdot \frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2} \cdot \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{h}_1} \cdot \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{\theta}_1}
\end{aligned} \tag{2.34}$$

From Equation 2.34 it is obvious that *some calculations are repeated*. As such, a faster calculation of the gradient is possible by avoiding these repetitions. This is the main idea behind the algorithm called ***back-propagation*** algorithm, often simply called ***backprop***. The procedure for a general computational graph is described in Algorithm 3 whereas a schematic representation for the aforementioned NN is provided in Figure 2.12. With this algorithm, a single forward and backward pass are enough to obtain the gradient.

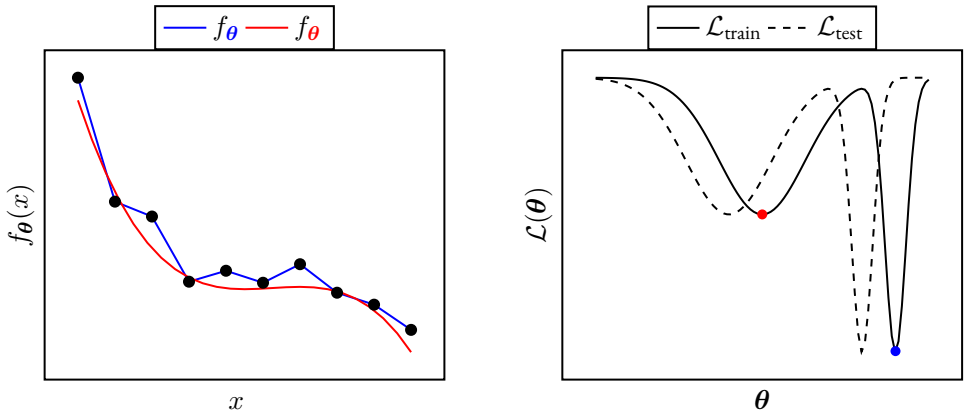


Figure 2.13: Machine learning $\neq$ optimization. From a purely optimization perspective, the blue solution is the ideal. In contrast, from a generalization point of view the red solution is preferred since it has lower error on new unseen samples.

Algorithm 3: Back-propagation (Rumelhart et al. 1986)

Input: Computational graph $\mathcal{G}$ where nodes u_i follow a topological ordering

`/* Forward pass */`

for $i = 1$ **to** N **do**
  Compute u_i as a function of $\text{Pa}_{\mathcal{G}}(u_i)$;
end
$u_N = 1$;

`/* Backward pass */`

for $i = N - 1$ **to** 1 **do**
  $\bar{u}_i = \sum_{j \in \text{Ch}_{\mathcal{G}}(u_i)} \bar{u}_j \partial_{u_j} u_i$;
end
return *derivatives* $\bar{u}_i$

In the rest of this section, the most common optimizers used for training NNs are presented. All of them are first-order iterative optimization methods, meaning that they can be trapped in local minima of the training loss landscape. However, this is not much of a problem since in ML *the interest is in finding parameters that generalize well, not necessarily the ones that perfectly fits the training data*. As shown in Figure 2.13 a local minimum might be a better choice than the global optimum. Moreover, flatter minima should be preferable because they are less sensitive to parameter perturbations or to put it differently, they are less tailored to the specifics of the training set at hand.

Since the training loss is a sum of individual losses (see Equation 2.3) and the gradient operator $\nabla(\cdot)$ is linear, the gradient of the training loss with respect to model parameters equals:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{train}} = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} \nabla_{\boldsymbol{\theta}} \ell_i(\boldsymbol{\theta}) \tag{2.35}$$

The problem is that *we need to compute all the individual gradients of the training samples*. Typical

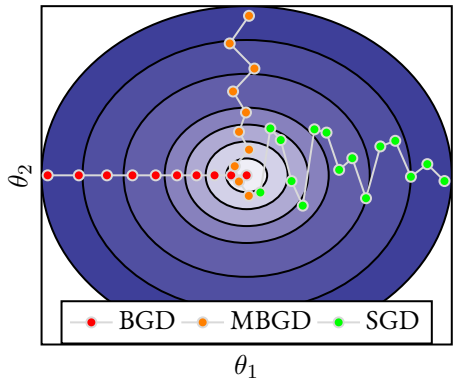


FIGURE 2.14: Variants of gradient descent.

datasets these days contain million training instances, hence it seems wasteful to perform all these calculations for just a single parameter update. Vanilla gradient descent variants come at the rescue by using a only a small subset—aka mini-batch in the DL jargon—of the training set. The insight of these algorithms is that *the gradient over the whole training set is an expectation and as such, it may be approximately estimated using only a small number of training samples*.

The mini-batches are usually drawn from the training set without replacement. These algorithms pass through the training samples until all the training data are used, at which point they start sampling from the full training set again. A single pass through the training set is called ***epoch*** and the number epochs is the most common criterion for stopping the traning of DL algorithms. Depending on the batch size $|\mathcal{B}|$, the gradient descent variants are classified as following[22]:

$$\text{variants} = \begin{cases} \text{SGD} & |\mathcal{B}| = 1 \\ \text{MBGD} & 1 < |\mathcal{B}| < |\mathcal{D}_{\text{train}}| \\ \text{BGD} & |\mathcal{B}| = |\mathcal{D}_{\text{train}}| \end{cases} \tag{2.36}$$

The decrease in the computational cost at each iteration comes at the expense of noisy updates as shown in FIGURE 2.14, meaning that these algorithms can't settle at the minimum. Although this is not a major problem, it can be mitigated by increasing the batch size at the last iterations or decrease the learning rate gradually (or both). Usually, the value of the batch size remains constant and only the learning rate changes through the training phase via a *learning rate scheduling* scheme. It should be added that the noisy gradient estimates might be beneficial since they can escape "bad" (e.g. sharp) local minima and saddle points. The aforementioed optimizers are described in ALGORITHM 4.

[22] Essentially, the batch gradient descent (BGD) algorithm is the vanilla gradient descent.

Algorithm 4: Batch, mini-batch and stochastic gradient descent (Bottou 1998)

Input: $\mathcal{D}_{\text{train}}$, loss function ℓ, model parameters $\boldsymbol{\theta}$, learning rate η, batch size $|\mathcal{B}|$

1 $\boldsymbol{\theta} \leftarrow$ random initialization;
2 **while** *stopping criterion not met* **do**
3 $\quad \mathcal{B} \leftarrow$ sample $|\mathcal{B}|$ datapoints from $\mathcal{D}_{\text{train}}$;
4 $\quad \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \ell_i(\boldsymbol{\theta})$;
5 $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$;
6 **end**
7 **return** *optimized parameters* $\boldsymbol{\theta}$

A common modification to the MBGD and SGD algorithms is the addition of a ***momentum*** term. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the previous steps to determine the next update. Momentum keeps track of the previous gradients via an exponentially decaying sum controlled by the hyperparameter $\beta \in (0, 1)$ which is typically set to 0.9. By averaging past gradients, their zig-zag directions (see Figure 2.14) effectively cancel out, leading to a smoother trajectory. Note that the value of β essentially controls how quickly the effect of the past gradients decay.

Algorithm 5: Momentum (Polyak 1964)

Input: Model parameters $\boldsymbol{\theta}$, momentum β, learning rate η

1 $\boldsymbol{\theta} \leftarrow$ random initialization;
2 **while** *stopping criterion not met* **do**
3 $\quad \boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$;
4 $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$;
5 **end**
6 **return** *optimized parameters* $\boldsymbol{\theta}$

Another commonly used optimizer for training NNs is the Adam optimizer, which is described in Algorithm 6. Besides the momentum term[23] $\boldsymbol{m}$ the update is controlled also by the term $\boldsymbol{s}$ which keeps track the square[24] of the past gradients. This term provides the means for using *adaptive learning rates* which can speed up convergence by pointing the resulting updates more towards the local minimum. The steps 5 and 6 are somewhat of a technical detail: since $\boldsymbol{m}$ and $\boldsymbol{s}$ are both initialized at $\mathbf{0}$, they will be biased towards $\mathbf{0}$ at the first iterations. These steps just boost $\boldsymbol{m}$ and $\boldsymbol{s}$ at the beginning of the training (Géron 2017). Typical values for the hyperparameters β_1 and β_2 are 0.9 and 0.999, respectively. Please note that the symbol $\odot$ represents the element-wise multiplication whereas $\oslash$ represents the element-wise division.

[23] Compared to momentum, Adam computes an exponentially decaying average rather than an exponentially decaying sum but these are actually equivalent except for a constant factor.

[24] That is, each s_i accumulates the squares of the partial derivative $\frac{\partial \mathcal{L}}{\partial \theta_i}$.

Algorithm 6: Adam (Kingma et al. 2017)

Input: Model parameters $\boldsymbol{\theta}$, learning rate η, smoothing term ϵ, momentum decay β_1, scaling decay β_2

$\boldsymbol{\theta} \leftarrow$ random initialization;
while *stopping criterion not met* **do**
  $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$;
  $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}} \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$;
  $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t}$;
  $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^t}$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$;
end
return *optimized parameters* $\boldsymbol{\theta}$