Chapter 2

Theoretical Background

DEEP LEARNING, *a class of machine learning (ML) algorithms based on neural networks (NNs)*, 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 (CNN)*, 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 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 minimum 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 or 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*.

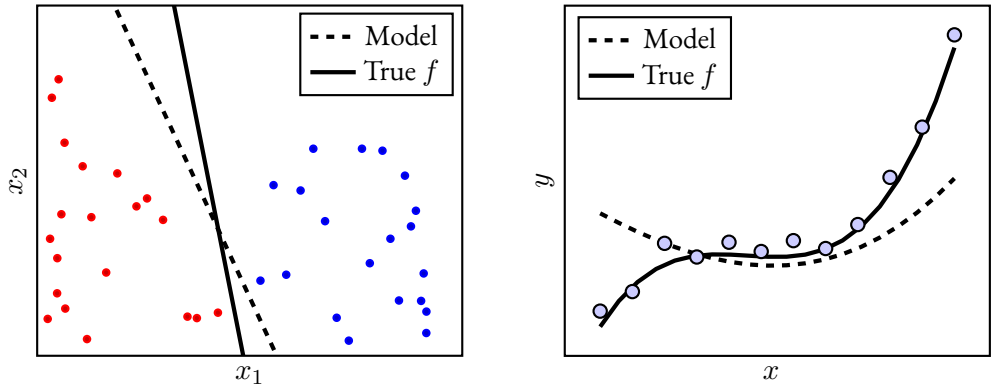
Definition 2.3 (Supervised learning). *The experience comes in the form* $\mathcal{D}_{train} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$*, where* $(\boldsymbol{x}_i, \boldsymbol{y}_i) \sim p(\boldsymbol{x}, \boldsymbol{y})$ *and* $\boldsymbol{y}_i$ *is the output aka label of the* i*-th training instance.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

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 X, an output space Y and we are interested in learning a function $\hat{h}\colon X \to Y$, called the *hypothesis*, which produces an output $\boldsymbol{y} \in Y$ given an input $\boldsymbol{x} \in X$. At our disposal we have a collection of input-output pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, forming the ***training set*** $\mathcal{D}_{\text{train}}$, 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***:

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

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 we don't have access to infinite data.

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*:

$$\hat{y} = \beta_0 + \beta x \tag{2.6}$$

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.:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 \tag{2.7}$$

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. n.d.).

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:

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

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:

$$\hat{y} = \beta_0 + \sum_{i=1}^{k} \beta_i x^i \tag{2.11}$$

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

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

Lets 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* β_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 $\beta_i = 0$. This leave us with a very simple model—only the *bias* β_0 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 the section 2.2.3.

Before moving on, it is worth to add that both the regularization and the optimizer can effect 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

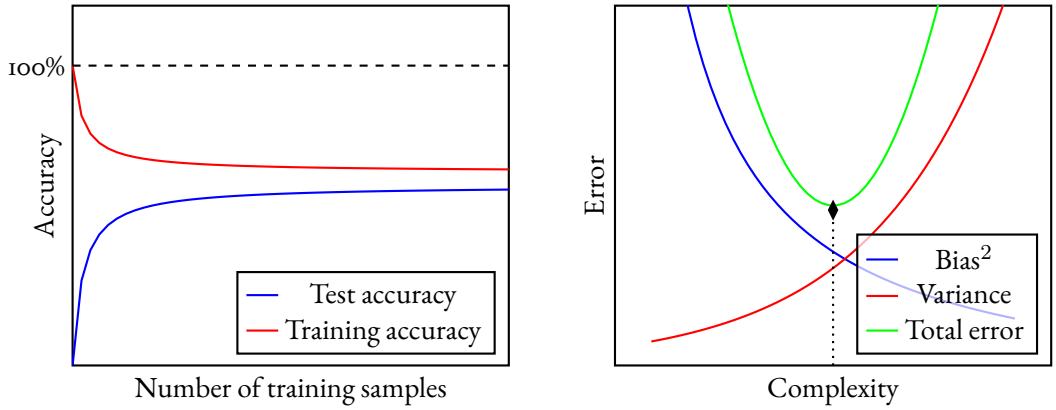
Suppose that we have trained our learner, and finally we get our model, as stated by Equation 2.5. *How can we assess its performance?* Remember, we can't calculate the generalization loss, since we are

[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).



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

FIGURE 2.2: Relation between performance and experience.

not given an infinite amount of data. First of all, *we should not report the training loss, because it is 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 have 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, called ***validation set***

THEOREM 1 (Bias-variance decomposition). *From Equation 2.1 and assuming* $\epsilon \sim \mathcal{N}(0, 1)$*, the ex-*

[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.

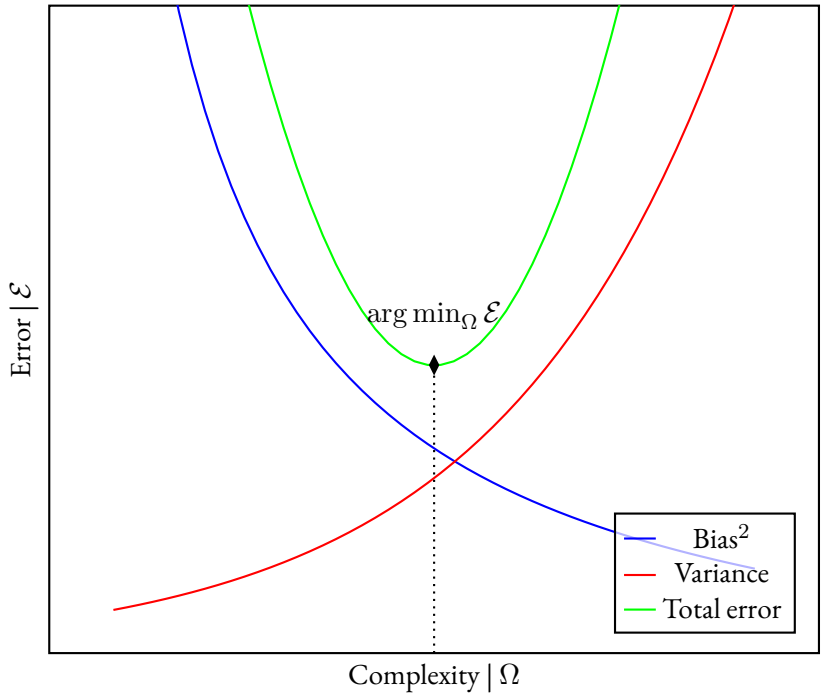


Figure 2.3: The bias-variance trade-off.

pected squared loss at $\boldsymbol{x}^*$, can be decomposed as following:

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

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 $\boldsymbol{x}^*$ in the test set.

2.2 Fundamentals of Deep Learning

2.2.1 Convolutional neural networks

2.2.2 Regularizing neural networks

2.2.3 Training neural networks

Algorithm 2: Batch, mini-batch and stochastic gradient descent algorithms

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

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

Algorithm 3: Momentum algorithm, needs citation

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

$\boldsymbol{\theta} \leftarrow$ random initialization;
while *stopping criterion not met* **do**
  $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$;
end
return *optimized parameters* $\boldsymbol{\theta}$

Algorithm 4: Nesterov momentum algorithm, needs citation

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

$\boldsymbol{\theta} \leftarrow$ random initialization;
while *stopping criterion not met* **do**
  $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta} + \beta \boldsymbol{m})$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$;
end
return *optimized parameters* $\boldsymbol{\theta}$

Algorithm 5: AdaGrad algorithm, needs citation

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

$\boldsymbol{\theta} \leftarrow$ random initialization;
while *stopping criterion not met* **do**
  $\boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \epsilon}$;
end
return *optimized parameters* $\boldsymbol{\theta}$

Algorithm 6: RMSProp algorithm, needs citation

Input: Model parameters $\boldsymbol{\theta}$, decay rate β, learning rate η, smoothing term ϵ

$\boldsymbol{\theta} \leftarrow$ random initialization;
while *stopping criterion not met* **do**
  $\boldsymbol{s} \leftarrow \boldsymbol{s} + (1-\beta)\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \epsilon}$;
end
return *optimized parameters* $\boldsymbol{\theta}$

Algorithm 7: Adam algorithm, needs citation

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**
  $\boldsymbol{m} \leftarrow \beta_1\boldsymbol{m} - (1-\beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$;
  $\boldsymbol{s} \leftarrow \beta_2\boldsymbol{s} + (1-\beta_2)\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$;
  $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t}$;
  $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^t}$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$;
end
return *optimized parameters* $\boldsymbol{\theta}$

Algorithm 8: Backpropagation algorithm, needs citation

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

`/* Forward pass */`
for $i = 1$ **to** N **do**
  Compute u_i as a function of $\mathrm{Pa}_{\mathcal{G}}(u_i)$;
end
$u_N = 1$;
`/* Backward pass */`
for $i = N-1$ **to** 1 **do**
  $\bar{u}_i = \sum_{j \in \mathrm{Ch}_{\mathcal{G}}(u_i)} \bar{u}_j \frac{\partial u_j}{\partial u_i}$;
end
return *derivatives* $\bar{u}_i$

[a] Any ordering such that parents come before children.