MACHINE LEARNING IN A NUTSHELL

THIMO PREIS[1]

18th March 2020

1 thimo.preis@posteo.de

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

TODO LIST

1 INTRODUCTION

1.1 INTRO TO AI

This is about considering the relationship between AI and ML.

1. AI includes machine learning, but machine learning does't fully define AI.
2. The main point of confusion between learning and intelligence is that people assume that simply because a machine gets better at its job (*learning*) it is also aware (*intelligence*). Nothing supports this view of machine learning. The same phenomenon occurs when people assume that a computer is purposely causing problems for them. The computer can not assign emotions and therefore acts only upon the input provided and the instruction contained within an application to process that input. A true AI will eventually occur when computers can finally emulate the clever combination used by nature
   a) *Genetics*:
      Slow learning from one generation to the next.
   b) *Teaching*:
      Fast learning from organized sources.
   c) *Exploration*:
      Spontaneous learning through media and interactions with others.
3. ML is only part of what a system requires to become an AI. The ML portion of the picture enables an AI to perform these tasks:
   a) Adapt to new circumstances that the original developer did not envision
   b) Detect patterns in all sorts of data sources
   c) Create new behaviours based on the recognized patters
   d) Make decisions based on the success or failure of the behaviours.

   The use of algorithms to manipulate data is the centrepiece of ML. To prove successful, a ML sessions must use an appropriate algorithm to achieve a desired result. In addition, the data must lend itself to analysis using the desired algorithm, or it requires a careful preparation by scientists.
   AI encompasses many other disciplines to simulate the thought process successfully. In addition to ML, AI normally includes

a) Natural language processing: The act of allowing language input and putting it into a form that a computer can use.

b) Natural language understanding: The act of deciphering the language in order to act upon the meaning it provides.

c) Knowledge representation: The ability to store information that makes fast access possible.

d) Planning (in the form of goal seeking): The ability to use stored information to draw conclusions in near real time (almost at the moment it happens, but with a slight delay so short that only computer notices).

e) Robotics: The ability to act upon requests from a user in some physical form

1.1.1 *On governance/policy*

As scientists continue to work with a technology and turn hypotheses into theories, the technology becomes related more to engineering than science. As the rules governing a technology become clear, groups of experts work together to define these rules in written form. The result is *specifications* (a set of rules that everyone agrees upon). Eventually, implementations of the specifications become *standards* that a governing body, such as the IEEE (Institute of Electrical and Electronics Engineers) or a combination of the ISO/IEC (International Organization for Standardization/international Electrotechnical Comission), manages. AI and ML have both been around long enough to create specifications, but you currently won't find any standards for either technology.

1.1.2 *ML*

1.1.2.1 *What is ML – the schools of thought*

Statistics and ML have a lot in common and statistics represents one of the five *tribes* (schools of thought) that make ML feasible. The five tribes are

1. Symbolists:
   The origin of this tribe is in logic and philosophy. This group relies on inverse deduction to solve problems.

2. Connectionists:
   The origin of this tribe is neuroscience. This group relies on backpropagation to solve problems.
   This tribe strives to reproduce the brain's functions using silicon instead of neurons. Essentially, each of the neurones (created as an algorithm that models the real-world counterpart) solves a

small piece of the problem, and the use of many neurons in parallel solves the problem as a whole.
The use of *backpropagation*, or backward propagation of errors, seeks to determine the conditions under which errors are removed from networks built to resemble the human neurons by changing the *weights* (how much a particular input figures into the result) and *biases* (which features are selected) of the network. The goal is to continue changing the weights and biases until such time as the actual output matches the target output. At this point, the artificial neuron fires and passes its solution along to the next neuron in line. The solution created by just one neuron is only part of the whole solution. Each neuron passes information to the net neuron in line until the group of neurons creates a final output.

3. Evolutionaries:
   The origin of this tribe is in evolutionary biology. This group relies on genetic programming to solve problems.
   The evolutionaries rely on the principles of evolution to solve problems. In other words, this strategy is based on the survival of the fittest (removing any solutions that don't match the desired output). A fitness funciton determines the viability of each function in solving a problem.
   Using a tree structure, the solution method looks for the best solution based on function output. The winner of each level of evolution gets to build the next-level functions. The idea is that the next level will get closer to solving the problem but may not solve it completely, which means that another level is needed. This particular tribes relies heavily on recursion and languages that strongly support recursion to solve problems. An interesting output of this strategy has been algorithms that evolve: One generation of algorithms actually builds the next generation.

4. Bayesians:
   The origin of this tribe is in statistics. This group relies on probabilistic inference to solve problems.
   The Bayesians use various statistical methods to solve problems. Given that statistical methods can create more than one apparently correct solution, the choice of a function becomes one of determining which function has the highest probability of succeeding. For example, when using these techniques, you can accept a set of symptoms as input and decide the probability that a particular disease will result from the symptoms as output. Given that multiple diseases have the same symptoms, the probability is important because a user will see some in which a lower probability output is actually the correct output for a given circumstance.
   Ultimately, this tribe supports the idea of never quite trusting

any hypothesis (a result that someone has given you) completely without seeing the evidence used to make it (the input the other person used to make the hypothesis). Analyzing the evidence proves or disproves the hypothesis that it supports. Consequently, it is not possible to determine which disease someone has until you test all the symptoms. One of the most recognizable outputs from this tribe is the spam filter.

5. Analogizers:
   The origin of this tribe is in psychology. This group relies on kernel machines to solve problems.
   The analogyzers use kernel machines to recognize patterns in data. By recognizing the pattern of one set of inputs and comparing it to the pattern of a known output, you can create a problem solution. The goal is to use similarity to determine the best solution to a problem. It is the kind of reasoning that determines that using a particular solution worked in a given circumstance at some previous time; therefore, using that solution for a similar set of circumstances should also work. One of the most recognizable outputs from this tribe is recommender systems like in Amazon.

The ultimate goal of ML is to combine the technologies and strategies embraced by the five tribes to create a single algorithm (the *master algorithm*) that can learn anything.
ML for dummies is following Bayesian approach.

1.1.2.2 *What means training ?*

> In machine learning you have inputs and you know the desired result. However, you do not know what function to apply to create the desired result. Training provides a learner algorithm with all sorts of examples of the desired inputs and results expected from those inputs. The learner then uses this input to create a function. In other words, training is the process whereby the learner algorithm maps a flexible function to the data. The output is typically the probability of a certain class or a numeric value.
> Note that we are basically talking about the function of a narrow AI, specific to one problem.

The secret to ML is generalization, the goal is to generalize the output function so that it works on data beyond the training set.
To create this generalized function, the learner algorithm relies on just three components:

1. Representation:
   The learner algorithm creates a *model*, which is a function that

will produce a given result for specific inputs. The representation is a set of models that a learner algorithm can learn. In other words, the learner algorithm must create a model that will produce the desired results from the input data. If the learner algorithm can't perform this task, it can't learn from the data and the data is outside the hypothesis space of the learner algorithm. Part of the representation is to discover which *features* (data elements within the data source) to use for the learning process.

2. Evaluation:
   The learner can create more than one model. However, it does not know the difference between good and bad models. An evaluation function determines which of the models works best in creating a desired result from a set of inputs. The evaluation function scores the models because more than one model could provide the required results.

3. Optimization:
   At some point, the training process produces a set of models that can generally output the right result for a given set of inputs. At this point, the training process searches through these models to determine which one works best. The best model is then output as the result of the training process.

1.1.2.3 *Intriccacies with*

1. Cleaning the data also lends a certain amount of artistics quality to the result. The cleaned dataset used by one scientist for ML tasks may not precisely match the cleaned datasets used by another.

2. When working in a ML environment, you also have the problem of input data to consider. For example, the microphone found in one smartphone won't produce precisely the same input data that a microphone in another smartphone will. The characteristics of the microphones differ, yet the result of interpreting the vocal commands provided by the user must remain the same.

1.1.3 *Big Data*

1.2 MATH BASICS

1.2.1 *Different possible types of learning*

1.2.1.1 *Supervised learning*

Supervised learning occurs when an algorithm learns from example data and associated target responses that can consist of numeric values or

string labels, such as classes or tags, in order to later predict the correct response when posed with new examples. The supervised approach is indeed similar to human learning under the supervision of a teacher. The teacher provides good examples for the student to memorize, and the student then derives general rules from these specific examples.
You need to distinguish between regression problems, whose target is a numeric value, and classification problems, whose target is a qualitative variable, such as a class or tag. E.g. a regression task determines the average prices of houses in the Boston area, and classification tasks distinguishes between kinds of iris flowers based on their sepal and petal measures.

Definition

Supervised learning concerns learning from labelled data (for example, a collection of pictures labeled as *containing a cat* or *not containing a cat*). Commonsupervised learning tasks include classification and regression

1.2.1.2 *Unsupervised learning*

Unsupervised learning occurs when an algorithm learns from plain examples without any associated response, leaving to the algorithm to determine the data patterns on its own. This type of algorithm tends to restructure the data into something else, such as new features that may represent a class or a new series of uncorrelated values. They are quite useful in providing humans with insights into the meaning of data and new useful inputs to supervised ML algorithms. As a kind of learning, it resembles the methods humans use to figure out that certain objects or events are from the same class, such as by observing the degree of similarity between objects. Some recommendation systems that you find on the web in the form of marketing automation are based on this type of learning. The marketing automation algorithm derives its suggestions from what you have bought in the past. The recommendations are based on an estimation of what group of customers you resemble the most and then inferring your likely preferences based on that group.

Definition

Unsupervised learning is concerned with finding patterns and structure in unlabelled data. Examples of unsupervised learning include clustering, dimensionality reduction, and generative modelling.

1.2.1.3 *Reinforcement learning*

Reinforcement learning occurs when you present the algorithm with examples that lack labels, as in unsupervised learning. However, you can accompany an example with positive or negative feedback according to the solution the algorithm proposes. Reinforcement learning is connected to applications for which the algorithm must make decisions (so the product is prescriptive, not just descriptive, as in unsupervised learning), and the decisions bear consequences. In the human world, it is just like learning by trial and error. Errors help you learn because they have a penalty added.
An interesting example occurs when computers learn to play video games by themselves. In this case, an application presents the algorithm with examples of specific situations. The application lets the algorithm know the outcome of actions it takes, and learning occurs while trying to avoid what it discovers to be dangerous and to pursue survival. The program is initially clumsy and unskilled but steadily improves with training until it becomes a champion.

Definition

In reinforcement learning, an agent learns by interacting with an environment and changing its behaviour to maximize its reward. For example, a robot can be trained to navigate in a complex environment b assigning a high reward to actions that help the robot reach a desired destination.

1.2.1.4 *On the learning process*

Even though supervised learning is the most popular and frequently used, all ML algorithms respond to the same logic. The central idea is that you can represent reality using a mathematical function that the algorithm does not know in advance but can guess after having seen some data. You can express reality and all its challenging complexity in terms of unknown mathematical functions that ML algorithms find and make advantageous. This concept it the core idea for all kinds of ML algorithms.

The objective of a supervised classifier is to assign a class to an example after having examined some characteristics of the example itself. Such characteristics are called *features*, and they can be both quantitative (numeric values) or qualitative (string labels). To assign classes correctly, the classifier must first examine a certain number of known examples closely (examples that already have a class assigned to them), each one accompanied by the same kinds of features as the examples that do not have classes. The training phase involves observation of many ex-

amples by the classifier that helps it learn so that it can provide an answer in terms of a class when it sees an example without a class later.

Example

To give an idea of what happens in the training process, imagine a child learning to distinguish trees from other objects. Before the child can do so in an independent fashion, a teacher presents the child with a certain number of tree images, complete with all the facts that make a tree distinguishable from other objects of the world. Such facts could be features such as its material(wood), its parts (trunk, branches, leaves or needles, roots), and location (planted into the soil). The child produces an idea of what a tree looks like by contrasting the display of tree features with the images of other different objects, such as pieces of furniture that are made of wood but do not share other characteristics with a tree.

A ML classifier works the same. It builds its cognitive capabilities by creating a mathematical formulation, called a *target function*, that includes all the given features ion a way that creates a function that can distinguish one class from another. Assume a target function, which can express the characteristics of a tree, to exist. In such a case, a ML classifier can look for its representation as a replica or as an approximation (a different function that works alike). Being able to express such a target function is the representation capability of the classifier.
The representation process takes place via *mapping*, where you discover the construction of a function by observing its outputs. A successful mapping in ML is similar to a child internalizing the idea of an object. She understands the abstract rules derived from the facts of the world in an effective way so that when she sees a tree she immediately recognizes it.
The set of all the potential functions that the learning algorithm can figure out is called the *hypothesis space*. We call the resulting classifier with all its set parameters a *hypothesis*. The hypothesis space must contain all the parameter variants of all the ML algorithms that you want to try to map to an unknown function when solving a classification problem. Different algorithms can have different hypothesis spaces. What really matters is that the hypothesis space contains the target function (or its approximation, which is a different but similar function).

Definition

In ML, someone has to provide the right learning algorithms, supply some nonlernable parameters (called *hyper-parameters*), choose a set of examples to learn from, and select the features that accompany the examples. Just as a child can't always learn to distinguish between right and wrong if left alone in the world, so ML algorithms need human beings to learn successfully.

Note that noise in real-world data is the norm. Many extraneous factors and errors that occur when recording data distort the values of the features. A good ML algorithm should distinguish the signals that can map back to the target function from extraneous noise.

1.2.2 *Cost function*

The driving force behind optimization in ML is the response from a function internal to the algorithm, called the *cost function*. You may see other terms used in some contexts, such as *loss function, objective function, scoring function,* or, *error function,* but the cost function is an evaluation function that measures how well the ML algorithm maps the target function that it is striving to guess. In addition, a cost function determines how well a ML algorithm performs in a supervised prediction or an unsupervised optimization problem.
The evaluation function works by comparing the algorithm predictions against the actual outcome recorded from the real world. Comparing a prediction against its real value using a cost function determines the algorithm's error level, keeping errors low is optimal. The cost function transmits what is actually important and meaningful for your purposes to the learning algorithm.

Example

When the problem is to predict who will likely become ill from a certain disease, you prize algorithms that can score a high probability of singling out people who have the same characteristics and actually did become ill later. Based on the severity of the illness, you may also prefer that the algorithm wrongly chooses some people who don't get ill after all rather than miss the people who actually do get ill.

When an algorithm uses a cost function directly in the optimization process, the cost function is used internally. Given that algorithms are set to work with certain cost functions, the optimization objective may differ from your desired objective. In such a case, you measure the results using an external cost function that, for clarity of terminology, you call an *error function* or *loss function* (if it has to be minimized) or a *scoring function* (if it has to be maximized).

With respect to your target, a good practice is to define the cost function that works the best in solving your problem, and then to figure out which algorithms work best in optimizing it to define the hypothesis space you want to test. When you work with algorithms that don't allow the cost function you want, you can still indirectly influence their optimization process by fixing their hyper-parameters and selecting your input features with respect to your cost function. Finally, when you've gathered all the algorithm results, you evaluate them by using your chosen cost function and then decide on the final hypothesis with the best result from your chosen error function.

> Deciding on the cost function is a really important and fundamental task because it determines how the algorithm behaves after learning and how it handles the problem you want to solve. Never rely on default options, but always ask yourself what you want to achieve using ML and check what cost function can best represent the achievement.

If you need to pick a cost function, ML explanations introduce a range of error functions for regression and classification, comprising root mean squared errors, log loss, accuracy, precision, recall, and area under the curve.

1.2.2.1 *An example: The gradient descent algorithm*

Gradient descent works out a solution by starting from a random solution when given a set of parameters (a data matrix made of features and a response). It then proceeds in various iterations using the feedback from the cost function, thus changing its parameters with values that gradually improve the initial random solution and lower the error. Even though the optimization may take a large number of iterations before reaching a good mapping, it relies on changes that improve the response cost function most (lower error) during each iteration. There can be local minima where the process may get stuck and cannot continue its descent.

You can visualize the optimization process as a walk in high mountains, with the parameters being the different paths to descend to the valley. A gradient descent optimization occurs at each step. At each iteration, the algorithm chooses the path that reduces error the most, regardless of the direction taken. The idea is that if steps aren't too large (causing the algorithm to jump over the target), always following the most downward direction will result in finding the lowest place. Unfortunately, this result does not always occurs because the algorithm can arrive at intermediate valleys, creating the illusion that it has reached the target. However, in most cases, gradient descent leads the ML algorithm to discover the right hypothesis for successfully mapping the problem. Given the optimization process's random initialization , run-

ning the optimization many times is good practice. This means trying different sequences of descending paths and not getting stuck in the same local minimum.
When working with repeated updates of its parameters base on mini-batches and single examples, the gradient descent takes the name *stochastic gradient descent*.

ML boils down to an optimization problem in which you look for a global minimum given a certain cost function.

1.2.3 *Data processing*

When operating with data within the limits of the computer's memory (RAM), you are working in *core memory*. Algorithms that work with core memory are called *batch algorithms*.
If the data set is too big to fit into the standard memory of a single computer you could the following.

1. Subsample:
   Data is reshaped by a selection of cases (and sometimes even features) based on statistical sampling into a more manageable, yet reduced, data matrix. Clearly, reducing the amount of data can't always provide exactly the same results as when globally analysing it. A successful subsampling must correctly use statistical sampling, by employing random or stratified sample drawings.

Definition

In *random sampling*, you create a sample by randomly choosing the examples that appear as part of the sample. The larger the sample, the more likely he sample will resemble the original structure and variety of data, but even with few drawn examples, the results are often acceptable, both in terms of representation of the original data and for ML purposes.

Definition

In *stratified sampling*, you control the final distribution of the target variable or of certain features in data that you deem critical for successfully replicating the characteristics of your complete data. A classic example is to draw a sample in a classroom made up of different proportions of males and females in order to guess the average height. If females are, on average, shorter than and in smaller proportion to males, you want to draw a sample that repliates the same proportion in order to obtain a reliable estimate of the average height. If you sample only males by mistake, you'll overestimate the average height. Using prior insight with sampling (such as knowing that gender can matter in height guessing) helps a lot in obtaining samples that are suitable for ML.

After you choose a sampling strategy, you have to draw a subsample of enough examples, given your memory limitations, to represent the variety of data. Data with high dimensionality, characterized by many cases and many features, is more difficult to subsample because it needs a much larger sample, which may not even fit into your core memory.

2. Network parallelism:
   Split data into multiple computers that are connected in a network. Each computer handles part of the data for optimization. You cannot split all ML algorithms into separable processes.

3. Rely on out-of-core algorithms, which work by keeping data on the storage device and feeding it in chunks into computer memory for processing. The feeding process is called *streaming*. Because the chunks are smaller than the core memory, the algorithm can handle them properly and use them for updating the ML algorithm optimization. After the update, the system discards them in favour of new chunks, which the algorithm uses for learning. This process goes on repetitively until there are no more chunks. Chunks can be small, and the process ic called *mini-batching*, or they can even be constituted by just a single example, called *online learning*.

2 SUPERVISED AND UNSUPERVISED LEARNING

We limit our focus to supervised and unsupervised learning if not specified otherwise, reinforcement learning will be treated later on.

Solve problem with bold greek letters

2.1 MORE FORMAL INTRODUCTION INTO THE IDEA OF MACHINE LEARNING

2.1.1 *Problem set-up and recipe*

Problems in ML typically involve inference about complex systems where we do not know the exact form of the mathematical model that describes the system. It is therefore not uncommon to have multiple candidate models that need to be compared.

2.1.1.1 *Ingredients*

Many problems in ML and data science start with the same ingredients. The first ingredient is the dataset $\mathcal{D} = (\mathbf{X}, \mathbf{y})$ where $\mathbf{X}$ is a matrix of independent variables and $\mathbf{y}$ is a vector of dependent variables. The second is the model $f(\mathbf{x}; \boldsymbol{\theta})$, which is a function $f : \mathbf{x} \rightarrow y$ of the *parameters* $\boldsymbol{\theta}$. That is, f is a function used to predict an output from a vector of input variables.

Using polynomial models (i.e. polynomials of different order) in polynomial regressions, we can think of each term in the polynomial as a 'feature' (i.e. a,b are features for $f_1(x) = ax^2 + bx$) in our model, then increasing the order of the polynomial we fit increases the number of features.

Model class

To make predictions, we will consider a family of functions $f_\alpha(x, \boldsymbol{\theta}_\alpha)$ that depend on some parameters $\boldsymbol{\theta}_\alpha$. These functions represent the *model class* that we are using to model the data and make predictions. Note that we choose the model class without knowing the function $f(x)$. The $f_\alpha(x; \boldsymbol{\theta}_\alpha)$ encode the *features* we choose to represent the data. Different models (e.g. $\alpha = 1, 2, 3$) can contain different number of parameters, then the models have different *model complexity*.

The final ingredient is the *cost function* $C(\mathbf{y}, f(\mathbf{X}; \boldsymbol{\theta}))$ that allow us to judge how well the model performs on the observations $\mathbf{y}$. The model is fit by finding the value of $\boldsymbol{\theta}$ that minimizes the cost function. For example, one commonly used cost function is the squared error. Minimizing the squared error cost function is known as the method of least squares, and is typically appropriate for experiments with Gaussian measurement errors.

2.1.1.2 *Recipe*

1. The *first step* in the analysis is to *randomly* divide the dataset $\mathcal{D}$ into two mutually exclusive groups $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ called the training and test sets. The fact that this must be the first step should be heavily emphasized - performing some analysis (such as using the data to select important variables) before partitioning the data is a common pitfall that can lead to incorrect conclusions. Typically, the majority of the data are partitioned into the training set (e.g. 90%) with the remainder going into the test set.

   Cross evaluation

   Therefore, to learn the parameters $\boldsymbol{\theta}_\alpha$, we will train our models on a *training dataset* and then test the effectiveness of the model on a **different** dataset, the *test dataset*.

2. The model is fit by minimizing the cost function using only the data in the training set $\hat{\boldsymbol{\theta}} = \arg\min_\theta\{C(\mathbf{y}_{train}, f(\mathbf{X}_{train}; \boldsymbol{\theta}))\}$.

3. Finally, the performance of the model is evaluated by computing the cost function using the test set $C(\mathbf{y}_{test}, f(\mathbf{X}_{test}; \hat{\boldsymbol{\theta}}))$.

2.1.2 *Performance evaluation*

2.1.2.1 *Ingredients for performance evaluation*

Measure for evaluating performance

The value of the cost function for the best fit model on the training set is called the *in-sample error*

$$E_{in} = C(\mathbf{y}_{train}, f(\mathbf{X}_{train}; \boldsymbol{\theta})) \tag{2.1}$$

and the value of the cost function on the test set is called the *out-of-sample error*

$$E_{out} = C(\mathbf{y}_{test}, f(\mathbf{X}_{test}; \boldsymbol{\theta})). \tag{2.2}$$

One of the most important observations we can make is that **the out-of-sample error is almost always greater than the in-sample error**

$$E_{out} \geq E_{in}. \tag{2.3}$$

Comparison of candidate models is usually done by using E_{out}. The model that minimizes this out-of-sample error is chosen as the best model (i.e. model selection).

Note that once we select the best model on the basis of its performance on E_{out}, the real-world performance of the winning model should be expected to be slightly worse because the test data was now used in the fitting procedure.
Splitting the data into mutually exclusive training and test sets provides an unbiased estimate for the predictive performance of the model - this is known as *cross-validation*.

Pitfalls for performance evaluation

It may be at first surprising that the model that has the lowest out-of-sample error E_{out} usually *does not* have the lowest in-sample Error E_{in}. Therefore, if our goal is to obtain a model that is useful for prediction, we may not want to choose the model that provides the best explanation for the current observations. At first glance, the observation that the model providing the best explanation for the current dataset probably will not provide the best explanation for future datasets is very counter-intuitive.
Moreover, the discrepancy between E_{in} and E_{ou} becomes more and more important, as the complexity of our data, and the models we use to make predictions, grows. As the number of parameters in the model increases, we are forced to work in high-dimensional spaces. The 'curse of dimensionality' ensures that many phenomena that are absent or rare in low-dimensional spaces become generic.

A comment on the difference between the in-and out-of-sample errors: There is a fundamental difference between minimizing the in-sample error and minimizing the out-of-sample error. The underlying reason for this is that the training data may not be representative of the full data distribution. From a Bayesian point of view, as David MacKay likes to repeat: *We can't make predictions without making assumptions*. Thus, it is sensible to introduce priors that reflect the fact that we are likely to be undersampled (especially in high dimensions).

2.1.2.2 *Another mathematical measure for performance evaluation*

R^2 coefficient of determination

The model performance (in-sample and out-of-sample) can be evaluated using the so-called *coefficient of determination*, which is given by:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left| y_i^{true} - y_i^{pred} \right|^2}{\sum_{i=1}^{n} \left| y_i^{true} - \frac{1}{n} \sum_{i=1}^{n} y_i^{pred} \right|^2}. \tag{2.4}$$

Optimal performance is $R^2 = 1$, but it can also be negative. A constant model that always predicts the expected value of y, $\langle y^{true} \rangle$, disregarding the input features, would get a R^2 score of 0.

2.1.2.3 *How to effectively do the performance evaluation*

It turns out that for complicated models studied in ML, predicting and fitting are very different things.
Models that give the best fit to existing data do not necessarily make the best predictions even for simple tasks. At small sample sizes, noise can create fluctuations in the data that look like genuine patterns. Simple models (like a linear function) cannot represent complicated patterns in the data, so they are forced to ignore the fluctuations and to focus on the larger trends.

Overfitting

Complex models with many parameters can capture both the global trends and noise-generated patterns at the same time. In this case, the model can be tricked into thinking that the noise encodes real information. This problem is called *overfitting* and leads to a steep drop-off in predictive performance.
We can guard against overfitting in two ways:

1. We can use less expensive models with fewer parameters, or
2. we can collect more data so that the likelihood that the noise appears patterned decreases.

The relative degree of overfitting: This information is contained in the difference in accuracy of our model on the training and test datasets.

Bias-Variance tradeoff

The *bias-variance* tradeoff is used in our countermeasures against overfitting. What is it ?
When the amount of training data is limited, one can often get better predictive power performance by using a less expressive model rather than the more complex model. The simpler model has more 'bias' but is less dependent on the particular realization of the training dataset, i.e. less 'variance'. Therefore, even though the correct model is guaranteed to have better predictive performance for an infinite amount of training data (less bias), the training errors stemming from finite-size sampling (variance) can cause simpler models to outperform the more complex model when sampling is limited.

The bias-variance tradeoff is one of the key concepts in ML and therefore discussed quantitatively in more detail in 2.1.2.3 and qualitatively in 2.2.2.1.
These two concepts are now discussed in more detail, for that we introduce another quantity and then look explicitly at the components of our theory causing problems for different complexity regimes.

Bias

The bias represents the best our model could do if we had an infinite amount of training data to beat down sampling noise. The bias is a property of the kind of functions, or model class, we are using to approximate $f(x)$. In general, the more complex the model class we use, the smaller the bias. However, we do not generally have an infinite amount of data. For this reason, to get best predictive power it is better to minimize the out-of-sample error, E_{out}, rather than the bias.

[1] How can we get a better idea of what is the true object to minimize to get bets results ?
The Bias-Variance tradeoff is implicitly encoded in the in-and out-of-sample errors. We will therefore draw from statistical learning theory in the following to get a better understanding of what it is we ought to be doing to achieve best practices.

1 This stems from the the law of large number. This is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer to the expected value as more trials are performed

2.1.2.4 *Where do the insights about how to do best practices come from ?*

The out-of-sample error will decrease with the number of data points. As the number of data points gets large, the sampling noise decreases and the training data set becomes more representative of the true distribution from which the data is drawn. For this reason, in the infinite data limit, the in-sample and out-of-sample error must approach the same value, which is called the 'bias' of our model.

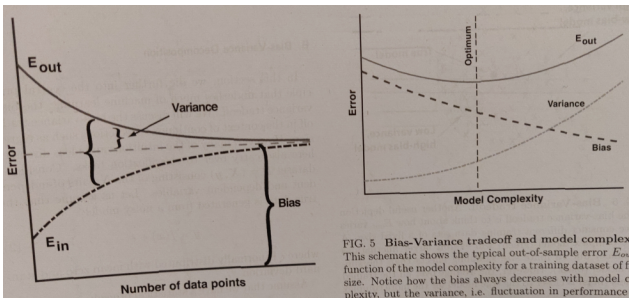

Figure 2.1

Compare the behaviour of both errors with increasing number of data points, left graph of 2.1, and the out-of-sample error with increasing model complexity, right graph of 2.1. As for the left graph, we can look at the difference between the generalization (E_{out}) and training error (E_{in}), i.e. the big curly bracket representing $|E_{in} - E_{out}|$. It measures how well our in-sample error reflects the out-of-sample error, and measures how much worse we would do on a new data set compared to our training data. For this reason, the difference between these errors is precisely the quantity that measures the difference between fitting and predicting. *Models with a large difference between the in-sample and out-of-sample errors are said to* ***overfit*** *the data.*

> One of the lessons of statistical learning theory is that it is not enough to simply minimize the training error, because the out-of-sample error can still be large.

Considering the right graph, model complexity is, in many cases, related to the number of parameters we are using to approximate the true function $f(x)$. If we consider a training dataset of a fixed size, E_{out} will be a non-monotonic function of the model complexity, and is generally minimized for models with *intermediate* complexity. The underlying reason for this is that, even though using a more complicated model always reduces the bias, at some point the model becomes too complex for the amount of training data and the generalization error becomes large due to high variance.

Thus, to minimize E_{out} and maximize our predictive power, it may be more suitable to use a more biased model with small variance than a less-biased model with large variance. This is is, again, the *bias-variance* tradeoff we introduced above.

Another way to understand this tradeoff is the following. Due to sampling noise from having finite size data sets, the learned models will differ for each choice of training sets. In general, more complex models need a larger amount of training data. For this reason, the fluctuations in the learned models (variance) will be much larger for the more complex model than the simpler model. However, if we consider the asymptotic performance as we increase the size of the training set (the bias), it is clear that the complex model will eventually perform better than the simpler model. Thus, depending on the amount of training data, it may **be more favourable to use a less complex, high-bias model to make predictions**.

2.2 STATISTICS

Statistical modelling revolves around estimation or prediction.

2.2.1 *Difference between estimation and prediction*

Note first and foremost that techniques in ML tend to be more focused on prediction rather than estimation, which means that we will mostly treat prediction problems in this compendium. This is for example because an artificially intelligent agent needs to be able to recognize objects in its surroundings and predict the behaviour of its environment in order to make informed choices.

2.2.1.1 *Contrasting the two*

Estimation and prediction problems can be cast into a common conceptual framework. In both cases, we choose some observable quantity $\mathbf{x}$ of the system we are studying (e.g. an interference pattern) that is related to some parameters $\boldsymbol{\theta}$ (e.g. the speed of light) of a model $p(\mathbf{x}|\boldsymbol{\theta})$ that describes the probability of observing $\mathbf{x}$ given $\boldsymbol{\theta}$.
Now we perform an experiment to obtain a dataset $\mathbf{X}$ and use these data to fit the model. Typically, 'fitting' the model involves finding $\hat{\boldsymbol{\theta}}$ that provides the best explanation for the data. In the case when 'fitting' refers to the method of least squares, the estimated parameters maximize the probability of observing the data (i.e., $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}}\{p(\mathbf{X}|\boldsymbol{\theta})\}$).

Estimation vs Prediction

Estimation problems are concerned with the accuracy of $\hat{\boldsymbol{\theta}}$, whereas *prediction problems* are concerned with the ability of the model to predict new observations (i.e., the accuracy of $p(\mathbf{x}|\hat{\boldsymbol{\theta}})$. Although the goals of estimation and prediction are related, they often lead to different approaches.

2.2.2 *Mathematical motivation to presented key ideas*

We begin with an unknown function $y = f(x)$ and fix a *hypothesis set* $\mathcal{H}$ consisting of all functions we are willing to consider, defined also on the domain of f . This set may be uncountably infinite (e.g. if there are real-valued parameters to fit). The choice of which functions to include in $\mathcal{H}$ usually depends on our intuition about the problem of interest. The function $f(x)$ produces a set of pairs $(x_i, y_i), i = 1...N$, which serve as the observable data. Our goal is to select a function from the hypothesis set $h \in \mathcal{H}$ that approximates $f(x)$ as best as possible, namely, we would like to find $h \in \mathcal{H}$ such that $h \approx f$ in some strict mathematical sense which we specify below.

If this is possible, we say that we *learned* $f(x)$.

But if the function $f(x)$ can, in principle, take any value on *unobserved inputs*, how is it possible to learn in any meaningful sense ?
The answer is that learning is possible in the restricted sense that the fitted model will probably perform approximately as well on new data as it did on the training data. How can we evaluate the performance of our model then ?
As discussed in 2.1.2, once an appropriate error function E is chosen for the problem under consideration (e.g. sum of squared errors in linear regression), we can define the in-and out-of-sample error to then minimize the out-of-sample error, with the variance-bias tradeoff in mind, to achieve robust predictions.

2.2.2.1 *Bias-Variance Decomposition for one Classifier*

Here we give a mathematical motivation to the bias-variance tradeoff discussed in 2.1.2.3.
Consider a dataset $\mathcal{D} = (\mathbf{x}, \mathbf{y})$ consisting of the N pairs of independent and dependent variables. Let us assume that the true data is generated from a noisy model

$$y = f(x) + \epsilon \tag{2.5}$$

where ϵ is normally distributed with mean zero and standard deviation σ_ϵ, i.e. the 'noise'. Assume that we have a statistical procedure

(e.g. least- squares regression) for forming a predictor $f(\mathbf{x};\hat{\boldsymbol{\theta}})$ that gives the prediction of our model for a new data point $\mathbf{x}$. This estimator is chosen by minimizing a cost function which we take to be the squared error

$$C(\mathbf{y},f(\mathbf{x};\boldsymbol{\theta})) = \sum_i (y_i - f(x_i;\boldsymbol{\theta}))^2. \tag{2.6}$$

Therefore, the estimates for the parameters

$$\hat{\boldsymbol{\theta}} = \arg\min_{\theta} C(\mathbf{y},f(\mathbf{x};\boldsymbol{\theta})) \tag{2.7}$$

are a function of the dataset, $\mathcal{D}$. We would obtain a different error $C(\mathbf{y}_j,f(\mathbf{x}_j;\hat{\boldsymbol{\theta}}_{\mathcal{D}_j}))$ for each dataset $\mathcal{D}_j = (\mathbf{y}_j,\mathbf{x}_j)$ in a universe of possible datasets obtained by drawing N samples from the true data distribution. We denote an expectation value over all of these datasets as $\mathbb{E}_{\mathcal{D}}$.

Errors

Combining these expressions, we see that the expected *out-of-sample error*

$$E_{out} := \mathbb{E}_{\mathcal{D},\epsilon}[C(\mathbf{y},f(\mathbf{x};\hat{\boldsymbol{\theta}}_{\mathcal{D}}))], \tag{2.8}$$

can be decomposed as

$$E_{out} = \text{Bias}^2 + \text{Var} + \text{Noise}, \tag{2.9}$$

with

$$\text{Noise} = \sum_i \sigma_\epsilon^2,\ \text{Var} = \sum_i \mathbb{E}_{\mathcal{D}}[(f(\mathbf{x}_i;\hat{\boldsymbol{\theta}}_{\mathcal{D}}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x}_i;\hat{\boldsymbol{\theta}}_{\mathcal{D}})])^2],$$
$$\text{Bias}^2 = \sum_i (f(\mathbf{x}_i) - \mathbb{E}_{\mathcal{D}}[f((\mathbf{x}_i;(\hat{\boldsymbol{\theta}}_{\mathcal{D}})])^2.$$

The variance measures how much our estimator fluctuates due to finite-sample effects and the bias measures the deviation of the expectation value of our estimator (i.e. the asymptotic value of our estimator in the infinite data limit) from the true value.

This gives us a mathematical tradeoff of the concepts discussed in 2.1.2 and in particular in 2.1.

Bias-variance trade-off

The bias-variance tradeoff summarizes the fundamental tension in machine learning, particularly supervised learning, between the complexity of a model and the amount of training data needed to train it. Since data is often limited, in practice it is often useful to use a less-complex model with higher bias – a model whose asymptotic performance is worse than another model – because it is easier to train and less sensitive to sampling noise arising from having a finite-sized training dataset (smaller variance).

2.2.2.2 *Bias-Variance Decomposition for Ensembles*

We will discuss the bias-variance tradeoff in the context of continuous predictions such as regression 2.6. However, many of the intuitions and ideas discussed here also carry over to classification tasks, cf. 2.7. Consider a data set consisting of data $\mathbf{X}_{\mathcal{L}} = \{(y_j, \mathbf{x}_j), j = 1 \dots n\}$ as above. Assume that we have a statistical procedure (e.g. OLS 2.35)) for forming a predictor $\hat{g}_{\mathcal{L}}(\mathbf{x})$ that gives the prediction of our model for a new data point $\mathbf{x}$ given that we trained the model using a dataset $\mathcal{L}$. We will now, however, consider an ensemble of estimators $\{\hat{g}_{\mathcal{L}}(\mathbf{x}_i)\}$. Given a dataset $\mathbf{X}_{\mathcal{L}}$ and hyper-parameters $\boldsymbol{\theta}$ that parametrize members of our ensemble, we will consider a procedure that deterministically generates a model $\hat{g}_{\mathcal{L}}(\mathbf{x}_i, \boldsymbol{\theta})$ given $\mathbf{X}_{\mathcal{L}}$ and $\boldsymbol{\theta}$. We assume that the $\boldsymbol{\theta}$ includes some random parameters that introduce stochasticity into our ensemble (e.g. an initial condition for stochastic gradient descent or a random subset of features or data points used for training.) Concretely, with a given dataset $\mathcal{L}$ one has a learning algorithm $\mathcal{A}$ that generates a model $\mathcal{A}(\boldsymbol{\theta}, \mathcal{L})$ based on a deterministic procedure which introduced stochasticity through $\boldsymbol{\theta}$ in its execution on dataset $\mathcal{L}$.
We will be concerned with the expected prediction error of the *aggregate ensemble predictor*

$$\hat{g}^{\mathcal{A}}_{\mathcal{L}}(\mathbf{x}_i, \{\boldsymbol{\theta}\}) = \frac{1}{M} \sum_{n=1}^{M} \hat{g}_{\mathcal{L}}(\mathbf{x}_i, \boldsymbol{\theta}_m). \tag{2.10}$$

For future reference, let us define the mean, variance and covariance (i.e. the connected correlation function in the language of physics), and

the normalized correlation coefficient of a single randomized model $\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_m)$ as

$$\begin{aligned}
\mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_m)] &= \mu_{\mathcal{L},\boldsymbol{\theta}_m}(\mathbf{x}) \\
\mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_m)^2] - \mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta})]^2 &= \sigma^2_{\mathcal{L},\boldsymbol{\theta}_m}(\mathbf{x}) \\
\mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_m)\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_{m'})] - \mathbb{E}_{\boldsymbol{\theta}}[\hat{g}_{\mathcal{L}}(\mathbf{x}, \boldsymbol{\theta}_m)]^2 &= C_{\mathcal{L},\boldsymbol{\theta}_m,\boldsymbol{\theta}_{m'}}(\mathbf{x}) \\
\rho(\mathbf{x}) &= \frac{C_{\mathcal{L},\boldsymbol{\theta}_m,\boldsymbol{\theta}_{m'}}(\mathbf{x})}{\sigma^2_{\mathcal{L},\boldsymbol{\theta}}}.
\end{aligned} \tag{2.11}$$

Note that the expectation $\mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[\cdot]$ is computed over the joint distribution of $\mathcal{L}$ and $\boldsymbol{\theta}_m$. Also by definition, we assume $m \neq m'$.
Now, we can, as in 2.2.2.1, define the expected generalization (out-of-sample) error for the ensemble

$$\begin{aligned}
\mathbb{E}_{\mathcal{L},\boldsymbol{\theta}_m}[C(\mathbf{X}, \hat{g}^A_{\mathcal{L}}(\mathbf{x}))] &= \mathbb{E}_{\mathcal{L},\epsilon,\boldsymbol{\theta}}\left[\sum_i (\mathbf{y}_i - \hat{g}^A_{\mathcal{L}}(\mathbf{x}_i, \{\boldsymbol{\theta}\}))^2\right] \\
&= Bias^2(\mathbf{x}_i) + Var(\mathbf{x}_i) + \sum_i \sigma^2_{\epsilon_i},
\end{aligned} \tag{2.12}$$

where one defines the bias of an aggregate predictor

$$Bias^2(\mathbf{x}) \equiv \left(f(\mathbf{x}) - \mathbb{E}_{\mathcal{L},\boldsymbol{\theta}}[\hat{g}^A_{\mathcal{L}}(\mathbf{x}, \{\boldsymbol{\theta}\})]\right)^2 \tag{2.13}$$

and the variance as

$$Var(\mathbf{x}) = \mathbb{E}_{\mathcal{L},\boldsymbol{\theta}}\left[(\hat{g}^A_{\mathcal{L}}(\mathbf{x}, \{\boldsymbol{\theta}\}) - \mathbb{E}_{\mathcal{L},\boldsymbol{\theta}}[\hat{g}^A_{\mathcal{L}}(\mathbf{x}, \{\boldsymbol{\theta}\})])^2\right].$$

So far the calculation for ensembles is almost identical to that of a single estimator, compare 2.2.2.1.

Crux of Bias-Variance for Ensembles

However, since the aggregate estimator is a sum of estimators, its variance implicitly depends on the correlations between the individual estimators in the ensemble. One finds

$$Var(\mathbf{x}) = \rho(\mathbf{x})\sigma^2_{\mathcal{L},\boldsymbol{\theta}} + \frac{1 - \rho(\mathbf{x})}{M}\sigma^2_{\mathcal{L},\boldsymbol{\theta}}. \tag{2.14}$$

This 2.14 is the key to understanding the power of random ensembles. Notice that by using large ensembles ($M \to \infty$), we can significantly reduce the variance, and for completely random ensembles where the models are uncorrelated ($\rho(\mathbf{x}) = 0$), maximally suppresses the variance !
Thus, reducing the aggregate predictor beats down fluctuations due to finite-sample effects. They key, as 2.14 indicates, is to **decorrelate the models as much as possible while still using a very large ensemble.**

What does this do to the bias ?

One can be worried that this comes at the expense of a very large bias. This turns out not to be the case. When models in the ensemble are completely random, the bias of the aggregate predictor is just the expected bias of a single model

$$Bias^2(\mathbf{x}) = (f(\mathbf{x}) - \mu_{\mathcal{L},\boldsymbol{\theta}})^2.$$

Thus, for a random ensemble one can always add more models without increasing the bias.

Therefore, the correlation between models that constitute the ensemble is the key property here. It is important for two reasons. First, holding the ensemble size fixed, averaging the predictions of correlated models reduces the variance less than averaging uncorrelated models. Second, in some cases, correlations between models within an ensemble can result in an *increase* in bias, offsetting any potential reduction in variance gained from ensemble averaging.
This will be discussed in the context of bagging 2.8.2.1.

2.2.3 *Mathematical set-up of every problem ever*

As described quantitatively in 2.1.1, every problem in ML can be designed via a recipe which is presented now qualitatively.

2.2.3.1 *Set-up of the notation*

Suppose we are given a dataset with n samples $\mathcal{D} = \{(y_i, \mathbf{x}^{(i)})\}_{i=1}^n$, where $\mathbf{x}^{(i)}$ is the i-th observation vector while y_i is its corresponding (scalar) response. We assume that every sample has p *features*, namely,

$$\mathbf{x}^{(i)} \in \mathbb{R}^p.$$

Let f be the true function/model that generated these samples via

$$y_i = f(\mathbf{x}^{(i)}, \mathbf{w}_{true}) + \epsilon_i,$$

where $\mathbf{w}_{true} \in \mathbb{R}^p$ is a *parameter* vector and ϵ_i is some i.i.d. white noise with zero mean and finite variance. Conventionally, we cast all samples in an $n \times p$ matrix, the *design matrix*

$$\mathbf{X} \in \mathbb{R}^{n \times p}$$

with the rows

$$\mathbf{X}_{i,:} = \mathbf{x}^{(i)} \in \mathbb{R}^p, \quad i = 1, \dots, n$$

being *observations* and the columns

$$\mathbf{X}_{:,j} \in \mathbb{R}^n, \quad j = 1, \dots, p$$

being measured *features*(i.e. feature predictors).
Bear in mind that this function f is never known to us explicitly, though in practice we usually presume its functional form.

Example

For example, in *linear regression*, we assume

$$y_i = f(\mathbf{x}^{(i)}; \mathbf{w}_{true}) + \epsilon_i = \mathbf{w}_{true}^T \mathbf{x}^{(i)} + \epsilon_i$$

for some unknown but fixed $\mathbf{w}_{true} \in \mathbb{R}^p$.

2.2.3.2 *Set-up of the problem*

We want to find a function g with parameters $\mathbf{w}$ fit to the data $\mathcal{D}$ that can best approximate f. This is done when we have found a $\hat{\mathbf{w}}$ such that $g(\mathbf{x}; \hat{\mathbf{w}})$ yields our best estimate of f. Now we can use this g to make predictions about the response y_0 for a new data point $\mathbf{x}_0$.
Therefore:
Fitting a given set of samples $(y_i, \mathbf{x}_i)$ means relating the independent variables $\mathbf{x}_i$ to their responses y_i. The next step is to assume some *models* that might explain the measurements and measuring their performance, i.e. finding the function g such that

$$y_i = g(\mathbf{x}_i; \mathbf{w}).$$

We call $g(\mathbf{x}^{(i)}, \mathbf{w})$ the *predictor*. Recall that the optimal choice of predictor depends on, among many other things, the functions used to fit the data and the underlying noise level. We then try to minimize the errors made in explaing the given set of measurements based on our model g by tuning the parameters $\mathbf{w}$. To do so, we need to first define the error function (formally called the *loss function*) that characterizes the deviation of our prediction from the actual response

2.2.3.3 *On statistical language*

As described above, a ML or statistics problem is defined via a optimization (minimization) problem of an error function, the solution to this is given by some estimator

$$\hat{\mathbf{w}}_{Problem} = \arg\min_{\mathbf{w} \in \mathbb{R}^p} C(\mathbf{X}, g(\mathbf{w}, \mathbf{y})). \tag{2.15}$$

The bias (or bias function) of an estimator is the difference between this estimator's expected value and the true value of the parameter being estimated.

2.2.4 *Language of optimization problems*

2.2.4.1 *Convexity*

Recall that a set $C \subset \mathbb{R}^n$ is called *convex* if any $x, y \in C$ and $t \in [0, 1]$,

$$tx + (1 - t)y \in C.$$

In other words, every line segments joining x, y lies entirely in C .
A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *convex* if its domain $\mathrm{dom}(f)$ is a convex set and for any $x, y \in \mathrm{dom}(f)$ and $t \in [0, 1]$,

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y).$$

In other words, the function lies below the line segment joining $f(x)$ and $f(y)$. This function f is called *strictly convex* if this inequality holds strictly for $x \neq y$ and $t \in (0, 1)$.

Why is convexity important?
For convex functions, any local minimizer is a global minimizer. Algorithmically, this means that in the minimization (optimization) procedure, as long as we're "going down the hill" and agree to stop when we can't go any further, then we've hit the global minimum. In addition to this, there's a menagerie of beautiful theory regarding convex duality and optimality, which gives us a way of understanding the solutions even before solving the problem itself.

2.3 OVERVIEW OF BAYESIAN INFERENCE

Bayesian inference provides a set of principles and procedures for learning from data and for describing uncertainty.

2.3.1 *Language*

Likelihood function

The *likelihood function*

$$p(\mathbf{X}|\boldsymbol{\theta}) \tag{2.16}$$

describes the probability of observing a dataset $\mathbf{X}$ for a given value of the unknown parameters $\boldsymbol{\theta}$. It is a function of the parameters $\boldsymbol{\theta}$ with the data $\mathbf{X}$ held fixed. Furthermore, it is determined by the model and the measurement noise.

Prior distribution

The *prior distribution* $p(\boldsymbol{\theta})$ describes any knowledge that we have about the parameters before we collect the data.

Priors

There are two general classes of priors:

1. The *uninformative prior*:
   We choose this one if we do not have any specialized knowledge about $\boldsymbol{\theta}$ before we look at the data.

2. The *informative prior*:
   If we have prior knowledge, we choose an informative prior that accurately reflects the knowledge that we have about $\boldsymbol{\theta}$. This one is commonly employed in ML:

There is another not so important prior, the *hierarchical prior*. This describes the process of choosing a hyperparameter by defining a prior distribution for it (via an uninformative prior) and then averaging the posterior distribution over all choices of the hyperparameter.

Using an informative prior tends to decrease the variance of the posterior distribution while, potentially, increasing its bias. It is beneficial if the decrease in variance is larger than the increase in bias. In high-dimensional problems, it is reasonable to assume that many of the parameters will not be strongly relevant. Therefore, many of the parameters of the model will be zero or close to zero. We can express this belief using two commonly used priors.

Two informative priors commonly employed

The *Gaussian prior* is used to express the assumption that many of the parameters will be small

$$p(\boldsymbol{\theta}|\lambda) = \prod_j \sqrt{\frac{\lambda}{2\pi}} e^{-\lambda\theta_j^2}, \tag{2.17}$$

where λ is a *hyperparameter*. The *Laplace prior* is used to express the assumption that many of the parameters will be zero

$$p(\boldsymbol{\theta}|\lambda) = \prod_j \frac{\lambda}{2} e^{-\lambda|\theta_j|}. \tag{2.18}$$

The hyperparameter can either be chosen via a hierarchical prior employing MCMC or by simply finding a good value of λ using an optimization procedure (see linear regression).

Indeed, a Gaussian prior is the *conjugate prior* that gives a Gaussian posterior. For a given likelihood, conjugacy guarantees the preservation of prior distribution at the posterior level.

Example

For example, for a Gaussian (Geometric) likelihood with a Gaussian (Beta) prior, the posterior distribution is still a Gaussian (Beta) distribution.

hyperparameters

A *hyperparameter* or *nuisance variable* is a parameter whose value is set before the learning process begins. By contrast, the values of other *parameters* are derived via training.

2.3.1.1 *Connecting statistics and bayesian framework*

To connect a statistical model to the Bayesian framework, we often write the model as

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mu(\mathbf{x}), \sigma^2(\mathbf{x})). \tag{2.19}$$

In other words, our model is defined by a conditional probability that depends not only on data $\mathbf{x}$ but on some model parameters $\boldsymbol{\theta}$.

Example

For example, if the mean is a linear function of $\mathbf{x}$ given by $\mu = \mathbf{x}^T\mathbf{w}$, and the variance is fixed $\sigma^2(\mathbf{x}) = \sigma^2$, then $\boldsymbol{\theta} = (\mathbf{w}, \sigma^2)$.

2.3.2 *Calculations*

Bayes' rule

The *posterior distribution* $p(\boldsymbol{\theta}|\mathbf{X})$ describes our knowledge about the unknown parameter $\boldsymbol{\theta}$ after observing the data $\mathbf{X}$. It is given via Bayes' rule

$$p(\boldsymbol{\theta}|\mathbf{X}) = \frac{p(\mathbf{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{\int \mathrm{d}\boldsymbol{\theta}' p(\mathbf{X}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')}. \tag{2.20}$$

The denominator is difficult in practice such that one normally uses Markov Chain Monte Carlo (MCMC) to draw random samples from $p(\boldsymbol{\theta}|\mathbf{X})$.

2.3.3 *Estimation*

2.3.3.1 *Maximum Likelihood estimation*

In statistics, many problems rely on estimation of some parameters of interest.

Example

For example, suppose we are given the height data of 20 junior students from a regional high school, but what we are interested in is the average height of all high school juniors in the whole country. It is conceivable that the data we are given are not representative of the student population as a whole.

It is therefore necessary to devise a systematic way to perform reliable estimation. Here we present the *maximum likelihood estimation* (MLE), and show that MLE for $\boldsymbol{\theta}$ is the one that minimizes the mean squared error (MSE) used in 2.35.

MLE

Many common statistical procedures such as least-square fitting can be cast as MLE. In MLE, one chooses the parameters $\hat{\boldsymbol{\theta}}$ that maximizes the likelihood (or equivalently the log-likelihood since log is a monotonic function) of the observed data $\mathbf{X}$ (or equivalently $\mathcal{D}$):

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{X}|\boldsymbol{\theta}). \tag{2.21}$$

In MLE we therefore choose the parameters that maximize the probability of seeing the observed data given our generative model.

Using the assumption that samples are i.i.d., we can write the *log-likelihood* as

$$l(\boldsymbol{\theta}) \equiv \log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p(y_i|\mathbf{x}^{(i)}, \boldsymbol{\theta}). \tag{2.22}$$

Note that the conditional dependence of the response variable y_i on the independent variable $\mathbf{x}^{(i)}$ in the likelihood function is made explicit since, in most applications, the observed value of data, y_i, is predicted based on $\mathbf{x}^{(i)}$ using a model that is assumed to be a probability distribution that depends on unknown parameters $\boldsymbol{\theta}$. This distribution, when endowed with $\boldsymbol{\theta}$, can, as we hope, potentially explain our prediction on y_i. By definition, such distribution is the likelihood function 2.16. We stress that this notation does *not* imply $\mathbf{x}^{(i)}$ is unknown, it is still part of the observed data!

Error function

The negative of the log-likelihood gives us the error function

$$E(\boldsymbol{\theta}) = -l(\boldsymbol{\theta}). \tag{2.23}$$

2.3.3.2 *Maximum-a-Posteriori estimation*

2.21 provides the tools for computing the posterior distribution $p(\boldsymbol{\theta}|\mathbf{X})$, which uses probability as a framework for expressing our knowledge about the parameters $\boldsymbol{\theta}$. In most cases, however, we need to summarize our knowledge and pick a single 'best' value for the parameters. In principle, the specific value of the parameters should be chosen to maximize a utility function. In practice, however, we usually use on of two choices:

1. The posterior mean, or *Bayes estimate*

$$\langle \boldsymbol{\theta} \rangle = \int \mathrm{d}\boldsymbol{\theta}\boldsymbol{\theta}\, p(\boldsymbol{\theta}|\mathbf{X}), \tag{2.24}$$

   or

2. the posterior mode, also called the *maximum-a-posteriori* (MAP) estimate

$$\hat{\boldsymbol{\theta}}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\mathbf{X}). \tag{2.25}$$

   This estimate gives the parameters at which the posterior probability distribution is peaked

While the Bayes estimate minimizes the mean-squared error, MAP estimate is often used instead because it is easier to compute.

2.3.3.3 *Out-of-Bag estimators*

For ensemble methods, especially random forest 2.8.3, there exists another object for estimation, the *out-of-bag (OOB) estimates*. This is discussed in more detail in 2.8.3.

2.3.4 *Bayesian view on regularization*

As discussed in 2.1.2, *we can't make predictions without making assumptions*. Thus, it is sensible to introduce priors that reflect the fact that we are likely to be undersampled (especially in high dimensions).
We can supplement an error function with a regularizer that prevents overfitting. From a Bayesian point of view, the regularization can be thought of as a prior on parameters. Minimizing the combined in-sample error + regularization terms is the same as the *Maximum a posteriori probability* (MAP) estimate in Bayesian regression 2.25. Note that in a

true Bayesian approach, we should not use the mode of the posterior but the average over all possible choices of parameters weighted by their posterior probability. In practice, this is often not done (for computational and practical reasons).

2.3.4.1 *MAP estimator in the context of regularization*

Instead of maximizing the the log-likelihood 2.22 to obtain a good estimator 2.21, let us maximize the log posterior $\log p(\boldsymbol{\theta}|\mathcal{D})$, which we can get via Bayes' rule 2.20. Then, the MAP estimator becomes

$$\hat{\boldsymbol{\theta}}_{MAP} \equiv \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \tag{2.26}$$

A common choice for the prior $p(\boldsymbol{\theta})$ is the Gaussian distribution. Consider the Gaussian prior 2.17 with zero mean and variance τ^2, namely

$$p(\mathbf{w}) = \prod_j \mathcal{N}(w_j|0, \tau^2).$$

Then, we can recast the MAP estimator into

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{MAP} &= \arg\max_{\boldsymbol{\theta}} \left[-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}^{(i)})^2 - \frac{1}{2\tau^2} \sum_{j=1}^{n} w_j^2 \right] \\ &= \arg\max_{\boldsymbol{\theta}} \left[-\frac{1}{2\sigma^2} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 - \frac{1}{2\tau^2} \|\mathbf{w}\|_2^2 \right]. \end{aligned} \tag{2.27}$$

This is used later on to cast linear regression into bayesian language, 2.6.3.

2.4 MATHEMATICAL TOOLS

2.4.1 *Sampling methods*

2.4.1.1 *Empirical Bootstrapping*

Empirical bootstrapping is a method to enlarge a sparse dataset, we use it in bagging, cf. 2.8.2.1.
Suppose we are given a finite set of n data points $\mathcal{D} = \{X_1, \dots, X_n\}$ as training samples and our job is to construct measures of confidence for our sample estimates (e.g. the confidence interval or mean-squared error of sample median estimator). To do so, one first samples n points **with replacement** from $\mathcal{D}$ to get a new set $\mathcal{D}^{*(1)} = \{X_1^{*(1)}, \dots, X_n^{*(1)}\}$. called a **bootstrap sample**, which possibly contains repetitive elements. Then we repeat the same procedure to get in total B such sets:

$$\mathcal{D}^{*(1)}, \dots, \mathcal{D}^{*(B)}.$$

The next step is to use these B bootstrap sets to get the **bootstrap estimate** of the quantity of interest. For example, let $M_n^{*(k)}$ =Median($\mathcal{D}^{*(k)}$)

be the sample median of bootstrap data $\mathcal{D}^{*(k)}$. Then we can construct the variance of the distribution of bootstrap medians as

$$\hat{Var}_B(M_n) = \frac{1}{B-1}\sum_{k=1}^{B}\left[M_n^{*(k)} - \bar{M}_n^*\right]^2 = \sigma^2 \overset{n\to\infty}{\longrightarrow} \frac{1}{n} \tag{2.28}$$

where

$$\bar{M}_n^* = \frac{1}{B}\sum_{k=1}^{B} M_n^{*(k)}$$

is the mean of the median of all bootstrap samples.
For $n \to \infty$, it was shown that the distribution of the bootstrap estimate will be a Gaussian centred around $\hat{M}_n(\mathcal{D})$ =Median$(X_1, \ldots, X_n)$ with standard deviation proportional to $1/\sqrt{n}$. This mean that the bootstrap distribution $\hat{M}_n^* - \hat{M}_n$ approximates fairly well the sampling distribution $\hat{M}_n - M$ from which we obtain the training data $\mathcal{D}$. Note that m is the median based on which the true distribution $\mathcal{D}$ is generated. In other words, if we plot the histogram of $\{M_n^{*(k)}\}_{k=1}^B$, we will see that in the large n limit it can be well fitted by a Gaussian which sharp peaks at $\hat{M}_n(\mathcal{D})$ and vanishing variance whose definition is given by 2.28.

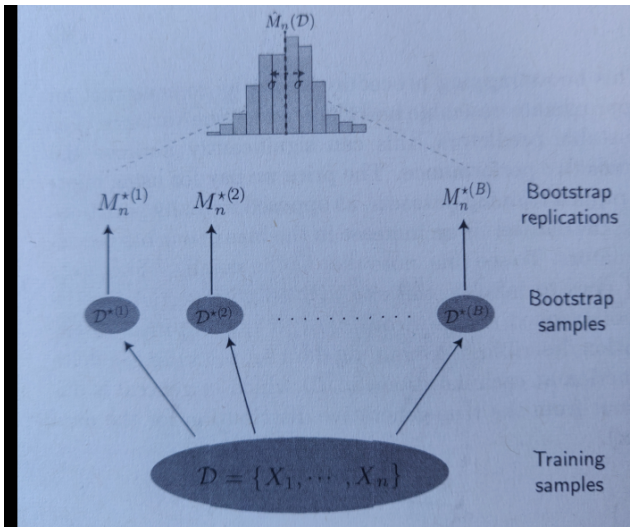

Figure 2.2

Figure 2.2 shows the procedure of empirical bootstrapping.

Bootstrapping - Summary

The goal is to assess the accuracy of a statistical quantity of interest, which in the main text is illustrated as the sample median $\hat{M}_m(\mathcal{D})$. We start from a given dataset $\mathcal{D}$ and bootstrap B size n datasets $\mathcal{D}^{*(1)}, \dots, \mathcal{D}^{*(B)}$ called the bootstrap samples. Then we compute the statistical quantity of interest on these bootstrap samples to get the median $M_n^{*(k)}$, for $k = 1, \dots, B$. These are then used to evaluate the accuracy of $\hat{M}_n(\mathcal{D})$. It can be shown that in the $n \to \infty$ limit the distribution of $M_n^{*(k)}$ would be a Gaussian centred around $\hat{M}_n(\mathcal{D})$ with variance σ^2 defined by 2.28.

2.4.2 *Algorithms*

2.4.2.1 *Decision Trees*

Decision trees are high-variance, weak classifiers that can be easily randomized, and as such, are ideally suited for ensemble-based methods. They are employed for random forests, cf. 2.8.3.
A decision tree uses a series of questions to hierarchically partition the the data. Each branch of the decision tree consists of a question that splits the data into smaller subsets with the leaves (end points) of the tree corresponding to the ultimate partitions of the data. When using decision trees for classification, the goal is to construct trees such that the partitions are informative about the class label. It is clear that more complex decision trees lead to finer partitions that give improved performance on the training set. However, this **generally leads to overfitting**, limiting the out-of-sample performance. For this reason, in practice almost all decision trees use some form of regularization (e.g. maximum depth for the tree) to control complexity and reduce overfitting. Decision trees also have extremely high variance, and are often extremely sensitive to many details of the training data. This is not surprising since decision trees are learned by partitioning the training data. Therefore, **individual decision trees are weak classifiers.**

However, these same properties make them ideal for incorporation in an ensemble method.

2.5 GRADIENT DESCENT

2.5.1 *Simple gradient descent*

As always in the context of ML, we want to minimize the cost function $E(\boldsymbol{\theta}) = C(\mathbf{x}, g(\boldsymbol{\theta}))$.

Gradient descent

The simplest gradient descent (GD) algorithm is characterized by the following *update rule* for the parameters $\boldsymbol{\theta}$. Initialize the parameters to some value $\boldsymbol{\theta}_0$ and iteratively update the parameters according to the equation

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{2.29}$$

where we have introduced the *learning rate* η_t (one hyperparameter of the model), that controls how big a step we should take in the direction of the gradient at time step t.

For sufficiently small choice of η_t, this method will converge to a *local minimum* of the cost function, however this is computationally expensive. In practice, one usually specifies a 'schedule' that decreases η_t at long times (common schedules include power law and exponential decay in time).[2] The simple GD has the following limitations

1. GD finds local minima of the cost function.
   Because in ML we are often dealing with extremely rugged landscapes with many local minima, this can lead to poor performance.
2. Gradients are computationally expensive to calculate for large datasets.
3. GD is very sensitive to choices of the learning rates.
   Ideally, we would 'adaptively' choose the learning rates to match the landscape.
4. GD treats all directions in parameter space uniformly.
5. GD is sensitive to initial conditions.
6. GD can take exponential time to escape saddle points, even with random initialization..

These limitiations lead to generalized GD methods which form the backbone of much of modern DL and NN.

2 Note that Newton's method is a first-order approximation of GD method, which is not practical as it is a computationally expensive algorithm. However, Newton's method automatically adjusts the step size so that one takes larger steps in flat directions with small curvature and smaller steps in steep directions with large curavture. This gives an intuition of how to modify GD methods to get better results.

2.5.2 *Modified gradient descent*

2.5.2.1 *Stochastic gradient descent (SGD) with mini-batches*

SGD

In SGD, we replace the actual gradient over the full data at each gradient descent step by an approximation to the gradient computed using a minibatch. This introduces stochasticity and decreases the chance that our fitting algorithm gets stuck in isolated local minima, as you cycle over all minibatches one at a time.
The update rule is

$$\mathbf{v}_t = \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t. \tag{2.30}$$

2.5.2.2 *Algorithm gradient descent with momentum*

It has been argues that first-order methods (with appropriate initial conditions) can perform comparable to more expensive second-order methods, especially in the context of comples DL models.

GDM

Introduce a 'momentum' term into SGD which serves as a memory of the direction we are moving in parameter space. This helps the GD algorithm to gain speed in directions with persistent but small gradients even in the presence of stochasticity, while suppressing oscillations in high-curvature directions. The update rules is

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}_t), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t, \tag{2.31}$$

where we have introduced a momentum parameter $\gamma \in [0, 1]$.

NAG

A final widely used variant of gradient descent with momentum is called the Nesterov accelerated gradient (NAG). In NAG, rather than calculating the gradient at the current position, one calculates the gradient at the position momentum will carry us to at time $t + 1$, namely, $\theta_t - \gamma v_{t-1}$. Thus, the update becomes

$$\mathbf{v}_t = \gamma \mathbf{v}_t + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t - \gamma \mathbf{v}_{t-1}), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t. \tag{2.32}$$

2.5.2.3 *Methods that use the second moment of the gradient*

We would like to adaptively change the step size to match the landscape. This can be accomplished by tracking not only the gradient, but also the second moment of the gradient[3]

3 Similar but avoiding the Hessian, which encodes local curvatures via second derivatives, as in Newton's method.

Root-mean-square propagation - RMSprop

In addition to keeping a running average of the first moment of the gradient, we also keep track of the second moment denoted by $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. The update rule is

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}),\ \mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2,\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \quad (2.33)$$

where β controls the averaging time of the second moment and is typically taken to be about $\beta = 0.9$, η_t is typically chosen to be 10^{-3}, and $\epsilon \propto 10^{-8}$ is a small regularization constant to prevent divergencies. It is clear from this formula that the learning rate is reduced in directions where the gradient is consistently large.

ADAM

In ADAM, we keep a running average of both the first and second moment of the gradient and use this information to adaptively change the learning rate for different parameters. In addition to keeping a running average of the second moments of the gradient (i.e $\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t], \mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$), ADAM performs an additional bias correction to account for the fact that we are estimating the first two moments of the gradient using a running average (denoted by the hat in the update rule). The update rule is given by (where multiplication and division are once again understood to be element-wise operations)

$$\begin{aligned} &\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}), \quad \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \\ &\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2, \quad \hat{\mathbf{m}}_t = \frac{\mathbf{m}}{1-(\beta_1)^t}, \\ &\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-(\beta_2)^t}, \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}, \end{aligned} \quad (2.34)$$

where β_1 and β_2 set the memory lifetimes of the first and second moment (typically $\beta_{1,2} = \{0.9, 0.99\}$ respectively, ϵ, η_t are same as in RMSprop).

The learning rates for RMSprop and ADAM can be set significantly higher than other methods due to their adaptive step sizes. For this reason, ADAM and RMSprop tend to be much quicker at navigating the landscape than simple momentum based methods.[4]

4 Note that in some cases trajectories might not end up at the global minimum. This kind of landscape structure is generic in high-dimensional spaces where saddle points proliferate.

2.5.3 *Practical tips for using GD*

Employ these tips for getting the best performance from GD based algorithms, especially in the context of deep neural networks (DNN)

1. Randomize the data when making mini-batches.
   Otherwise, the GD method can fit spurious correlations resulting from the order in which data is presented.

2. Transform your inputs.
   One simple trick for minimizing problems in difficult landscapes is to standardize the data by subtracting the mean and normalizing the variance of input variables. Whenever possible, also decorrelate the inputs.

3. Monitor the out-of-sample performance.
   Always monitor the performance of your model on a validation set (a small portion of the training data that is held out of the training process to serve as a proxy for the test set). If the validation error starts increasing then the model is beginning to overfit. Terminate the learning process. This *early stopping* significantly improves performance in many settings.

4. Adaptive optimization methods do not always have good generalization.
   Recent studies have shown that adaptive methods such as ADAM, RMSprop , and AdaGrad tend to have poor generalization to SGD or SGD with momentum, particularly in the high-dimensional limit (i.e. the number of parameters exceeds the number of data points). Although it is not clear at this state why sophisticated methods (e.g. ADAM, RMSprop, AdaGrad) perform so well in training DNN such as generative adversarial networks (GANs), simpler procedures like properly-tuned plain SGD may work equally well or better in some applications.

2.6 LINEAR REGRESSION

2.6.1 *Least-square regressions -frequentist*

We'll consider ordinary least squares regression problem in which the "error function" is defined as the square from the deviation of our linear predictor to the true response.

OLS

Ordinary least squares linear regression (OLS) is defined as the minimization of the L_2 norm of the difference between the response y_i and the predictor $g(\mathbf{x}^{(i)}; \mathbf{w}) = \mathbf{w}^T\mathbf{x}^{(i)}$:

$$\min_{\mathbf{w}\in\mathbb{R}^p} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \min_{\mathbf{w}\in\mathbb{R}^p} \sum_{i=1}^{n} (\mathbf{w}^T\mathbf{x}^{(i)} - y_i)^2. \tag{2.35}$$

Where 2.35 is the minimization of the loss function of OLS.
We are looking to find the parameters $\mathbf{w}$ which minimize the L_2 error. Geometrically speaking, the predictor function g defines a hyperplane in $\mathbb{R}^p$. Minimizing the least squares error is therefore equivalent to minimizing the sum of all projections (i.e. residuals) for all points $\mathbf{x}^{(i)}$ to this hyperplane. If $\text{rank}(\mathbf{X}) = p$, namely, the feature predictors $\mathbf{X}_{:,1}, \dots, \mathbf{X}_{:,p}$ are linearly independent, then there exists unique solution to this problem, which we denote as $\hat{\mathbf{w}}_{LS}$:

$$\hat{\mathbf{w}}_{LS} = \arg\min_{\mathbf{w}\in\mathbb{R}^p} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}, \tag{2.36}$$

where we have assumed that $\mathbf{X}^T\mathbf{X}$ is inertible, which is often the case when $n \gg p$ (i.e. method works if number of data points exceeds number of features). The best fit of our data $\mathbf{X}$ is

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}_{LS} = \underbrace{\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T}_{\equiv P_{\mathbf{X}}}, \tag{2.37}$$

where $P_{\mathbf{X}}$ is the projection matrix which acts on $\mathbf{y}$ and projects it onto the column space of $\mathbf{X}$, which is spanned by the predictors $\mathbf{X}_{:,1}, \dots, \mathbf{X}_{:,p}$.

Note how we found the optimal solution $\hat{\mathbf{w}}_{LS}$ in one shot, without doing any sort of iterative optimization.
What are the errors of OLS such that we can evaluate its performance according to 2.1.2 ?[5]
We find that the average *generalization error* for this method to be

$$|\bar{E}_{in} - \bar{E}_{out}| = \sigma^2 \left|(1 - \frac{p}{n}) - (1 + \frac{p}{n})\right| = 2\sigma^2\frac{p}{n}. \tag{2.38}$$

This implies: If we have $p \gg n$ (i.e. high-dimensional data), the generalization error is extremely large, meaning the model is not learning. Even when we have $p \approx n$, we might still not learn well due to the intrinsic noise σ^2.

5 As we have seen, the difference between learning and fitting lies in the prediction on 'unseen' data.

One way to ameliorate this is to use regularization. We will mainly focus on two forms of regularization which are introduced in the following:
the first one employs an L_2 penalty and is called *Ridge regression*, while the second uses an L_1 penalty and is called *LASSO*.

2.6.2 *Regularized Least-square regressions-frequentist*

Due to poor generalization, regularization is necessary, in particular in the *high-dimensional limit* ($p \gg n$). Regularization typically leads to better generalization.

Idea behind regularization

Due to the equivalence between the constrained and penalized form of regularized regression, we can regard the regularized regression problem as an un-regularized problem but on a *constrained set of parameters*. Since the size of the allowed parameter space (e.g. $\mathbf{w} \in \mathbb{R}^p$ when un-regularized vs. $\mathbf{w} \in C \subset \mathbb{R}^p$ when regularized) is roughly a proxy for model complexity, solving the regularized problem is in effect **solving the un-regularized problem with a smaller model complexity class**. This implies that we are **less likely to overfit**.

Why is that so ?
Let's say you are a young Physics student taking a laboratory class where the goal of the experiment is to measure the behaviour of several different pendula and use that to predict the formula (i.e. model) that determines the period of oscillation. In you investigation you would probably recod many things in an effort to give yourself the best possible chance of determining the unknown relationship, perhaps writing down the temperature of the room, any air currents, if the table were vibrating, etc. What you have done is create a high-dimensional dataset for yourself. However you actually possess an even higher-dimensional dataset than you probably would admit to yourself, e.g. whether it is Alice's birthday or whether you found a penny this morning, but you almost assuredly haven't written these down in your notebook. Why not ? The reason is because *you entered the classroom with strongly held* ***prior beliefs*** *that none of those things affect the physics which takes place in that room*. What is serving you here is the *intuition* that probably only a few things matter in the physics of pendula. Hence again you are approaching the experiment with prior beliefs about how many features you will need to pay attention to in order to predict what will happen when you swing an unknown pendulum.
The point is that we live in a high-dimensional world of information and while we have good intuition about what to write down in our notebook for well-known problems, often in the field of ML we cannot

say with any confidence a priori *what* the small list of things to write down will be, but we can at least **use regularization to help us enforce that the list not be too long** so that we don't end up predicting that the period of a pendulum depends on Bob having a cold on Wednesdays.

2.6.2.1 *Ridge-Regression*

Ridge-Regression

We now add to the least squares loss function a *regularizer* defined as as the L_2 norm of the parameter vector we wish to optimize over. In Ridge-Regression, the regularization penalty is taken to be the L_2-norm of the parameters

$$E_{Ridge} = \lambda \|\mathbf{w}\|_2^2 = \lambda \mathbf{w}^T \mathbf{w} = \lambda \sum_{\gamma=1}^{p} w_\gamma w_\gamma. \tag{2.39}$$

Thus, the model is fit by minimizing the sum of the in-sample error and the regularization term

$$\hat{\mathbf{w}}_{Ridge}(\lambda) = \arg\min_{\mathbf{w}\in\mathbb{R}^p} \left(\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 \right) \tag{2.40}$$

Notice that the parameter λ controls how much we weigh the fit and regularization term. The solution/estimate is found by differentiating w.r.t $\mathbf{w}$

$$\hat{\mathbf{w}}_{Ridge}(\lambda) = \frac{\hat{\mathbf{w}}_{LS}}{1+\lambda} \tag{2.41}$$

where the equality via 2.36 holds for orthogonal $\mathbf{X}$. This implies that the ridge estimate is merely the least squares estimate scaled by a factor $(1+\lambda)^{-1}$. This problem is equivalent to the following *constrained* optimization problem

$$\hat{\mathbf{w}}_{Ridge}(t)) \arg\min_{\mathbf{w}\in\mathbb{R}^p : \|\mathbf{w}\|_2^2 \le t} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 2. \tag{2.42}$$

Thus, by adding a regularization term, $\lambda\|\mathbf{w}\|_2^2$, to our least squares loss function, we are effectively constraining the magnitude of the parameter vector learned from the data. The solution

One can furthermore derive a relation (via singular value decomposition (SVD) on X) between the fitted vector $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}_{Ridge}$ and the prediction made by least squares linear regression

$$\hat{y}_{Ridge} = \mathbf{X}\hat{\mathbf{w}}_{Ridge} \le \mathbf{X}\hat{\mathbf{y}} \equiv \hat{\mathbf{y}}_{LS}. \tag{2.43}$$

It is clear that in order to compute the fitted vector $\hat{\mathbf{y}}$, both Ridge and least squares linear regression have to project $\mathbf{y}$ to the column space

of $\mathbf{X}$. The only difference is that Ridge regression further shrinks each basis component j by a factor $d_j^2/(d_j^2+\lambda)$, where $d_1 \geq d_2 \geq \dots d_p \geq 0$ are the singular values of $\mathbf{X}$.

> It is considered good practice to always check the performance for the given model and data as a function of λ.

If increasing λ simply degrades performance, we are most likely not undersampled.

2.6.2.2 *LASSO and Sparse Regression*

LASSO

Now we add an L_1 regularization penalty, conventionally called 'least absolute shrinkage and selection operator' (LASSO). The penalty is the L_1-norm of the parameters (sum of absolute values of parameters)

$$E_{LASSO} = \lambda \|\mathbf{w}\|_1 = \lambda \sum_{\gamma=1}^{p} \left|\mathbf{w}_\gamma\right| \tag{2.44}$$

LASSO in the penalized form is defined by the following regularized regression problem

$$\hat{\mathbf{w}}_{LASSO}(\lambda) = \arg\min_{\mathbf{w}\in\mathbb{R}^p} \|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1. \tag{2.45}$$

Obtaining a solution is not that easy, assuming $\mathbf{X}$ to be orthogonal one finds a so-called threshold function (compare 2.3)

$$\hat{w}_j^{LASSO}(\lambda) = \text{sign}(\hat{w}_j^{LS})\left[\left|\hat{w}_j^{LS}\right| - \lambda\right]_+ \tag{2.46}$$

where $(x)_+$ denotes the positive part of x and $\hat{w}_j^{LS}$ is the jth component of 2.36.
In general, LASSO tends, in contrast to Ridge, to given sparse solutions, meaning many components of $\hat{\mathbf{w}}_{LASSO}$ are zero. Looking at the behaviour of weights of LASSO and Ridge depending on the regularization parameter, one observes that LASSO, unlike Ridge, sets feature weights to zero leading to sparsity.

On the hyperparameter

The regularization parameter λ affects the weights (features) the model (LASSO, Ridge) learns on a data set.

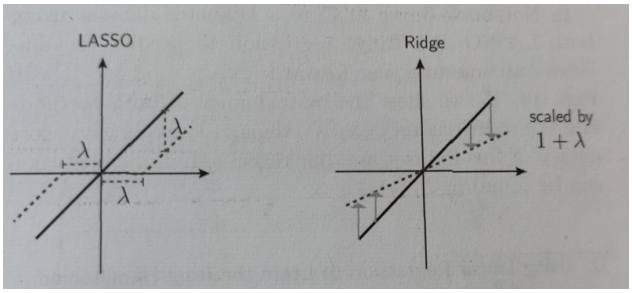

Figure 2.3

2.6.2.3 *A note on LASSO and Ridge*

Note that both LASSO and Ridge regression are convex in $\mathbf{w}$. What's more, Ridge is actually a strictly convex problem (assuming $\lambda > 0$) due to presence of L_2 penality. In fact, this is always true regardless of $\mathbf{X}$ and so the ridge regression solution is always well-defined.
In contrast, LASSO is not always strictly convex and hence by convexity theory, it need not have a unique solution. The LASSO solution is unique under general conditions, for example, when $\mathbf{X}$ has columns in general position. To mitigate this, one can define a modified problem called the elastic net such that the function we want to minimize is always strictly convex:

$$\min_{\mathbf{w}\in\mathbb{R}^p} \|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1 + \delta\|\mathbf{w}\|_2^2$$

where $\lambda, \delta \geq 0$ are regularization parameters. Now aside from uniqueness of the solution, the elastic net combines some of the desirable properties (e.g. prediction) of ridge regression with the sparsity properties of the LASSO.

2.6.2.4 *Another note being a general comment on regularization*

Comparing the performance of OLS, LASSO and Ridge for the $1D$ Ising model via the coefficient of determination 2.4 by looking at the parameter space of the hyperparameter λ, one can draw the following conclusions.
Choosing whether to use Ridge or LASSO regression turns out to be similar to fixing gauge degrees of freedom.

Picking a regularization scheme

Different regularization schemes can lead to learning equivalent models but in different 'gauges'. Any information we have about the symmetry of the unknown model that generated the data should be reflected in the definition of the model and the choice of regularization.

2.6.2.5 *A general perspective on regularizers*

On the hyperparameter

The **hyperparameter** λ involved in e.g. LASSO and Ridge is usually predetermined, which means that it is not part of the regression process. Our learning performance and solution depends strongly on λ, thus it is vital to choose it properly. As discussed in 2.3.1, one approach is to assume an *uninformative prior* on the hyper-parameters, $p(\lambda)$, and average the posterior over all choices of λ following this distribution. However, this comes with a large computational cost. Therefore, is is simpler to choose the regularization parameter through some optimization procedure.

2.6.3 *Bayesian formulation of linear regression*

In the formal statistical treatment of regression, the goal is to estimate the *conditional expectation* of the depndent variable given the value of the independent variable (sometimes called the covariate). To connect linear regression to the Bayesian framework, we use 2.19. In combination with 2.22, we get

$$l(\boldsymbol{\theta}) = -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}^{(i)})^2 - \frac{n}{2}\log(2\pi\sigma^2) = -\frac{1}{2\sigma^2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + const.$$

By comparing this with 2.36, it is clear that performing least squares is the same as maximizing the log-likelihood of this model.

2.6.3.1 *Regularization*

What about adding regularization?

The MAP estimate 2.25 corresponds to regularized linear regression, where the choice of prior determines the type of regularization.

The equivalence between MAP estimation with a Gaussian prior and Ridge regression is established by comparing 2.27 and Ridge regression 2.41 with $\lambda \equiv \sigma^2/\tau^2$. An analogous derivation holds for LASSO .

todo ?

2.6.4 *Outlook from linear regression*

Linear regression can be applied to model non-linear relationships between input and response. This can be done by replacing the input $\mathbf{x}$ with some non-linear function $\phi(\mathbf{x})$. Note that doing so preserves the linearity as a function of the parameters $\mathbf{w}$, since the model is defined by their inner product $\phi^T(\mathbf{x})\mathbf{w}$. This method is known as *basis function expansion*.[6]

2.7 LOGISTIC REGRESSION

So far we have focused on learning from datasets for which there is a 'continuous' output. However, a wide variety of problems, such as *classification*, are concerned with outcomes taking the form of discrete variables (i.e. *categories*).

Example

For example, we may want to detect if there is a cat or a dog in an image.

We will now introduce *logistic regression* which deals with binary, dichotomous outcomes (.e.g. True or False, Success or Failure,etc.).

2.7.1 *Mathematical set-up*

2.7.1.1 *Binary classification*

Throughout this section, we consider the case where the dependent variables $y_i \in \mathbb{Z}$ are discrete and only take values from $m = 0, \dots, M-1$ (which enumerate the M classes). The goal is to predict output classes from the design matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ made of n samples, each of which bears p features. The primary goal is to identify the classes to which new unseen samples belong.

2.7.1.2 *Multi-class classification*

For multi-class classification, we can not only look at binary classification (in which the labels are dichotomous variables). One approach is to treat the label as a vector $\mathbf{y}_i \in \mathbb{Z}_2^M$, namely a binary string of length M with only one component of y_i being 1 and the rest zero.

Example

For example, $\mathbf{y}_i = (1, 0, \dots, 0)$ means data the sample $\mathbf{x}_i$ belongs to class 1.

6 Look into ML for physicists review for references.

2.7.2 *Classifiers*

Given $\mathbf{x}_i$, the classifier returns the probability of being in category m. The following perceptron is an example of *hard* classification, each datapoint is assigned to a category (i.e. $y_i = 0$ or $y_i = 1$). A *soft* classifier on the other hand gives the probability of a given category as an output. The classifiers do this by using threshold functions, which are discussed in the following 2.7.2.1.
In many cases, it is favourable to work with a soft classifier.

2.7.2.1 *On threshold function functions*

A threshold function is a function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(x)$ (a single binary value). One simple way to get a discrete output is to have sign (or threshold) functions that map the output of a linear regressor to $\{0, 1\}$. The following are possible:

1. Step functions (perceptrons) given by the *sign* function

$$\sigma(s_i) = \text{sign}(s_i) = \left\{ \begin{array}{ll} 1 & \text{if } s_i \geq 0 \\ 0 & \text{otherwise} \end{array} \right\} \tag{2.47}$$

   are employed for hard classification.

2. The *logistic* (or *sigmoid*) function (i.e. Fermi functions)

$$\sigma(s) = \frac{1}{1 + e^{-s}}, \quad 1 - \sigma(s) = \sigma(-s). \tag{2.48}$$

3. The hyperbolic tangent

   Put definition here

$$\tanh(z) \tag{2.49}$$

4. Rectified linear units (ReLUs)
5. Leaky rectified linear unity (leaky ReLUs)
6. Exponential linear units (ELUs)

2.7.3 *Perceptron Learning Algorithm (PLA)*

Before delving into the details of logistic regression, it is helpful to consider a slightly simpler classifier.

Perceptron crude idea

Consider a linear classifier that categorizes examples using a weighted linear-combination of the features and an additive offset:

$$s_i = \mathbf{x}_i^T \mathbf{w} + b_0 \equiv \mathbb{x}_i^T \mathbb{w}, \tag{2.50}$$

where we use the short-hand notation $\mathbb{x}_i = (1, \mathbf{x}_i)$ and $\mathbb{w} = (b_0, \mathbf{w})$. This function takes values on the entire real axis. IN the case of logistic regression, however, the labels y_i are discrete variables. Using the sign function 2.47 to create discrete outputs via hard classification, we obtain the *perceptron* model.
In the modern sense, the perceptron is an algorithm for learning a binary classifier called a *threshold function*, see 2.7.2.1.

2.7.3.1 *The algorithm*

Suppose that we're given a set of N observations each bearing p features, $\mathbf{x}_n = (x_1^{(n)}, \cdots, x_p^{(n)}) \in \mathbb{R}^p$, $n = 1, \cdots, N$. The goal of binary classification is to relate these observations to their corresponding binary label $y_n \in \{+1, -1\}$. Concretely, this amounts to finding a function $h : \mathbb{R}^p \to \{+1, -1\}$ such that $h(\mathbf{x}_n)$ is ideally the same as y_n.

Perceptron

A *perceptron* accomplishes this feat by utilizing a set of weights $\mathbb{w} = (w_0, w_1, \cdots, w_d) \in \mathbb{R}^{p+1}$ to construct h so that labelling is done through

$$h(\mathbb{x}_n) = \text{sign}\left(w_0 + \sum_{i=1}^{p} w_i x_i^{(n)}\right) = \text{sign}\,(\mathbb{w}^T \mathbb{x}_n), \tag{2.51}$$

where $\mathbb{x}_n = (1, x_1^{(n)}, \cdots, x_p^{(n)}) = (1, \mathbf{x}_n)$. The perceptron can be viewed as the zero-temperature limit of the logistic regression where the sigmoid (Fermi-function) becomes a step function.

PLA begins with randomized weights. It then selects a point from the training set at random. If this point, say, $\mathbb{x}_n$, is misclassified, namely, $y_n \neq \text{sign}\,(\mathbb{w}^T \mathbb{x}_n)$, weights are updated according to

$$\mathbb{w} \leftarrow \mathbb{w} + y_n \mathbb{x}_n \tag{2.52}$$

Otherwise, $\mathbb{w}$ is preserved and PLA moves on to select another point. This procedure continues until a specified threshold is met, after which PLA outputs h. It is clear that PLA is an online algorithm since it does not treat all available data at the same time. Instead, it learns the weights as it progress along data points in the training set one-by-one. The

update rule is built on the intuition that whenever a mistake is encountered, it corrects its weights by moving towards the right direction.

2.7.4 *Definition of logistic regression - Bayesian*

This is a binary classification problem, see 2.7.1.1. Here we define logistic regression and discuss the minimization of its corresponding cost function (the *cross entropy*).

Logistic regression

Logistic regression is the canonical example of a soft classifier. In logistic regression, the probability that a data point $\mathbf{x}_i$ belongs to a category $y_i = \{0, 1\}$ is given by

$$P(y_i = 1|\mathbf{x}_i, \boldsymbol{\theta}) = \frac{1}{1 + e^{-\mathbb{x}_i^T \boldsymbol{\theta}}} = \sigma(\mathbb{x}_i^T \mathbb{w})$$
$$P(y_i = 0|\mathbf{x}_i, \boldsymbol{\theta}) = 1 - P(y_i = 1|\mathbf{x}_i, \boldsymbol{\theta})$$

where $\boldsymbol{\theta} = \mathbb{w}$ are the weights we wish to learn from the data. The cost function of logistic regression, the *cross entropy*, is found to be

$$C(\mathbb{w}) = \sum_{i=1}^{n} \left[-y_i \log \sigma(\mathbb{x}_i^T \mathbb{w}) - (1 - y_i) \log \left(1 - \sigma(\mathbb{x}_i^T \mathbb{w})\right) \right]. \tag{2.53}$$

In practice, we usually implement the cross-entropy (like in linear regression) with additional regularization terms, usually L_1 and L_2 regularization.

Minimizing the cross entropy

The cross entropy is a convex function of the weights $\mathbb{w}$ and, therefore, any local minimizer is a global minimizer. Minimizing this cost function leads to

$$0 = \nabla C(\mathbb{w}) = \sum_{i=1}^{n} \left[\sigma(\mathbb{x}_i^T \mathbb{w}) - y_i \right] \mathbb{x}_i. \tag{2.54}$$

In words, the gradient points in the sum of training example directions weighted by the difference between the true label and the probability of predicting that label. Equation 2.54 defines a transcendental equation for $\mathbb{w}$, the solution of which, unlike linear regression, cannot be written in a closed form. For this reason, one must use numerical methods such as 2.5 to solve this optimization problem.

[7] In the notebooks, one looks at two examples to train a logistic regressor to classify binary data. We call *one-hot* a group of bits where only one bit is 1 and all others 0, i.e. representing states: $|1\rangle = (1, 0, \dots, 0), |2\rangle = (0, 1, 0, \dots, 0)$.

2.7.4.1 *On the 2d Ising example*

Given an Ising state, we would like to classify whether it belongs to the ordered or the disordered phase, without any additional information other than the spin configuration itself. This categorical ML problem is well suited for logistic regression, and will thus consist of recognizing whether a given state is ordered by looking at its bit configurations. Notice that, for the purposes of logistic regression, the $2D$ spin state of the Ising model will be flattened out to a $1D$ array, so it will not be possible to learn information about the structure of the contiguous ordered $2D$ domains. SUch information can be incorporated using deep convolutional neural networks.
We use both ordered and disordered states to train the logistic regressor and, once the supervised training procedure is complete, we will evaluate the performance of our classification model on unseen ordered, disordered, and near-critical states. Here, we deploy the *liblinear* routine (the default for Scikit' s logistic regression) and stochastic gradient descent to optimize the logistic regression cost function with L_2 regularization. We define the accuracy of the classifier as the percentage of correctly classified data points. Comparing the accuracy on the training and test data, we can study the degree of overfitting.
Similar to the linear regression examples, we find that there exists a sweet spot for the SGD regularization strength λ that results in optimal performance of the logistic regressor.

2.7.4.2 *SUSY*

Here we will use logistic regression in an attempt to find the relative probability that an event is from a signal or a background. Is a good classification problem with noisy data, how to discriminate against the background ? Some signal events look background-like, and some background events look signal-like to our discriminator.

2.7.5 *SoftMax regression*

In this section we generalize logistic regression to the case of multiple categories which is called *SoftMax regression*. This is a multi-class classification problem 2.7.1.2.
For general categorical data, y can then take on M values so that $y \in$

[7] However, as a word of caution, note there is a generic instability in the MLE procedure for linearly separable data)

$\{0, 1, \dots, M-1\}$. For each datapoint i, define a vector $y_{im} \equiv [\mathbf{y}_i]_m$, whichs refers to the m'-th component of vector $\mathbf{y}_i$, called a *one-hot* vector, such that

$$y_{im} = \left\{ \begin{array}{ll} 1, & \text{if } y_i = m \\ 0, & \text{otherwise.} \end{array} \right\} \tag{2.55}$$

SoftMax regression

The probability of $\mathbf{x}_i$ being in class m' is given by

$$\hat{y}_{im(\mathbb{w})=p(yi=m'|\mathbb{x}_i;\mathbb{w})\equiv} P(y_{im'} = 1|\mathbf{x}_i, \{\mathbb{w}_k\}_{k=0}^{M-1}) = \frac{e^{-\mathbb{x}_i^T \mathbb{w}_{m'}}}{\sum_{m=0}^{M-1} e^{-\mathbb{x}_i^T \mathbb{w}_m}}, \tag{2.56}$$

where $y_{im'} \equiv [\mathbf{y}_i]_{m'}$ refers to the m'-th component of vector $\mathbf{y}_i$. This is known as the *SoftMax* function. The generalized cost function reads

$$C(\mathbb{w}) = -\sum_{i=1}^{n} \sum_{m=0}^{M-1} [y_{im} \log P(y_{im} = 1|\mathbf{x}_i, \mathbb{w}_m) \, . \tag{2.57}$$

$$+(1 - y_{im}) \log (1 - P(y_{im} = 1|\mathbf{x}_i, \mathbb{w}_m))] \, , \tag{2.58}$$

which reduces to the cross entropy 2.53 for $M = 1$, i.e. for only two possible classes.

2.8 ENSEMBLE METHODS - ON COMBINING MODELS

2.8.1 *Introduction*

Ensemble methods combine predictions from multiple, often weak, statistical models to improve predictive performance. Ensemble methods, such as random forests, and boosted gradient trees, such as XGBoost, undergird many of the winning entries in data science competitions such as Kaggle, especially on structured datasets. Note that Neural Networks generally perform better than ensemble methods on unstructured data, images and audio. Even in the context of NN, it is common to combine predictions from multiple neural networks to increase performance on tough image classification tasks.

2.8.1.1 *Motivation*

We will give an overview of ensemble methods and provide rules of thumb for when and why they work.
On one hand, the idea of training multiple models and then using a weighted sum of the predictions of all these models is very natural. On

the other hand, one can also imagine that the ensemble predictions can be much worse than the predictions from each of the individual models that constitute the ensemble, especially when pooling reinforces weak but correlated deficiencies in each of the individual predictors. Thus, it is important to understand when we expect ensemble methods to work.
As we saw in 2.2.2.2, the key to determining when ensemble methods work is the degree of correlation between the models in the ensemble.

2.8.1.2 *Benefits of Ensemble Methods before diving in*

Three distinct shortcomings that are fixed by ensemble methods are: statistical, computational, and representational.
The first reason is statistical. When the learning set is too small, a learning algorithm can typically find several models in the hypothesis space $\mathcal{H}$ that all give the same performance on the training data. Provided their predictions are uncorrelated, averaging several models reduces the risk of choosing the wrong hypothesis. The second reason is computational. Many learning algorithms rely on some greedy assumption or local search that may get stuck in local optima. As such, an ensemble made of individual models built from many different starting points may provide a better approximation of the true unknown function than any of the single models. Finally, the third reason is representational. In most cases, for a learning set of finite size, the true function cannot be represented by any of the candidate models in $\mathcal{H}$. By combining several models in an ensemble, it may be possible to expand the space of representable functions and to better model the true function.

The increase in representational power of ensembles comes from the fact that it is more advantageous to combine a group of simple hypotheses than to utilize a single arbitrary linear classifier. This of course comes with the price of introducing more parameters to our learning procedure. But if the problem itself can never be learned through a simple hypothesis, then there is no reason to avoid applying a more complex model. Since ensemble methods reduce the variance and are often easier to train than a single complex model, they are a powerful way of increasing representational power (also called expressivity in the ML literature).

How should we construct ensembles ?

1. Try to randomize ensemble construction as much as possible to reduce the correlations between predictors in the ensemble. This ensures that our variance will be reduced while minimizing an increase in bias due to correlated errors.

2. The ensembles will work best for procedures where the error of the predictor is dominated by the variance and not the bias. Thus, these methods are especially well suited for unstable procedures whose results are sensitive to small changes in the training dataset.

3. Although the discussion above was derived in the context of continuous predictors such as regression, the basic intuition behind ensembles applies equally well to classification tasks. Using an ensemble allows one to reduce the variance by averaging the result of many independent classifiers. As with regression, this procedure works best for unstable predictors for which error are dominated by variance due to finite sampling rather than bias.

2.8.2 *Aggregate predictor methods - Bagging and Boosting*

A powerful approach is the idea to build a strong predictor by combing many weaker classifiers from different models. In bagging, the contribution of all predictors is weighted equally in the bagged (aggregate) predictor 2.59. However, in principle, there are myriad ways to combine different predictors. In some problems one might prefero to use an autocratic approach that emphasizes the best predictors, while in others it might be better to opt for more 'democratic' ways as is done in bagging, compare 2.8.2.1. In boosting on the other hand, one associates weights to the different weak classifiers in order to amplify the contribution from the most robust classifiers, compare 2.8.2.2.

2.8.2.1 *Bagging*

BAGGing, or Bootstrap AGGregation, is one of the most widely employed and simplest ensemble-inspired methods. Bagging is effective on 'unstable' learning algorithms where small changes in the training set result in large changes in predictions.
Imagine we have a very large dataset $\mathcal{L}$ that we could partition into M smaller data sets which we label $\{\mathcal{L}_1, \dots, \mathcal{L}_m\}$. If each partition is sufficiently large to learn a predictor, we can create an ensemble aggregate predictor composed of predictors trained on each subset of the data. For continuous predictors like regression, this is just the average of all the individual predictors

$$\hat{g}^A_{\mathcal{L}}(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} g_{\mathcal{L}_i}(\mathbf{x}). \tag{2.59}$$

For classification tasks where each predictor predicts a class label $j \in \{1, \dots, J\}$, this is just a majority vote of all the predictors,

$$\hat{g}^A_{\mathcal{L}}(\mathbf{x}) = \arg\max_j \sum_{i=1}^{M} I[g_{\mathcal{L}_i}(\mathbf{x}) = j], \tag{2.60}$$

where $I[\,]$ is an indicator function that is equal to one if $g_{\mathcal{L}_i}(\mathbf{x}) = j$ and zero otherwise.

This can significantly reduce the variance without increasing the bias.

While simple and intuitive, this form of aggregation clearly works only when we have enough data in each partitioned set $\mathcal{L}_i$. To see this, one can consider the extreme limit where $\mathcal{L}_i$ contains exactly one point. In this case, the bases hypothesis $g_{\mathcal{L}_i}(\mathbf{x})$ (e.g. linear regressor) becomes extremely poor and the procedure above fails. One way to circumvent this shortcoming is to resort to *empirical bootstrapping*, which is treated in more detail in 2.4.1.1. The idea of empirical bootstrapping is to use sampling with replacement to create new 'bootstrapped' datasets $\{L_1^{BS}, \ldots, L_M^{BS}\}$ from our original dataset $\mathcal{L}$. These bootstrapped datasets share many points, but due to the sampling with replacement, are all somewhat different from each other.

Bootstrapped Bagging

In the bagging procedure, we create an aggregate estimator by replacing the M independent datasets by the M bootstrapped estimators

$$\hat{g}_{\mathcal{L}}^{BS}(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} g_{\mathcal{L}_i^{BS}}(\mathbf{x}), \tag{2.61}$$

and

$$\hat{g}_{\mathcal{L}}^{BS}(\mathbf{x}) = \arg\max_j \sum_{i=1}^{M} I[g_{\mathcal{L}_i^{BS}}(\mathbf{x}) = j]. \tag{2.62}$$

This bootstrapping procedure allows us to construct an approximate ensemble and thus reduce the variance. For unstable predictors, this can significantly improve the predictive performance. The price we pay for using bootstrapped training datasets, as opposed to really partitioning the dataset, is an increase in the bias of our bagged estimators.

Limitations of Bagging

When the procedure is unstable, the prediction error is dominated by the variance and one can exploit the aggregation component of bagging to reduce the prediction error. In contrast, for a stable procedure the accuracy is limited by the bias introduced by using bootstrapped datasets. This means that there is an instability-to-stability transition point beyond which bagging stops improving our prediction.

For more complicated datasets, how can we choose the right hyperparameters?

Out-of-bag estimate and out-of-bag prediction error

We can actually make use of one of the most important and interesting features of ensemble methods that employ Bagging: *out-of-bag* (OOB) estimates. Whenever we bag data, since we are drawing samples with replacement, we can ask how well our classifiers do on data points that are not used in the training. This is the *out-of-bag prediction error* and plays a similar role to cross-validation error in other ML methods. Since this is the best proxy for out-of-sample prediction, we choose hyperparameters to minimize the out-of-bag error.

This is listed in 2.3.3.3 for reference, will have to expand on it.

2.8.2.2 *Boosting*

Boosting

In boosting, an ensemble of weak classifiers $\{g_k(\mathbf{x})\}$ is combined into an aggregate, boosted classifier. However, unlike bagging 2.8.2.1, each classifier is associated with a weight α_k that indicates how much it contributes to the aggregate classifier

$$g_A(\mathbf{x}) = \sum_{k=1}^{M} \alpha_k g_k(\mathbf{x}), \tag{2.63}$$

where $\sum_k \alpha_k = 1$. Boosting, like all ensemble methods, works best when we combine simple, high-variance classifiers into a more complex whole.

There are different ideas of how the boosting itself works, for now we only discuss *adaptive boosting* or AdaBoost. The basic idea is to form the aggregate classifier in an iterative process. Importantly, at each iteration we reweight the error function to 'highlight' data points where the aggregate classifier performs poorly (so that in the next round the procedure puts more emphasis on making those right.) In this way, we can successively ensure that our classifier has good performance over the whole dataset.
How does AdaBoost work ?
Suppose that we are given a data set $\mathcal{L} = \{(\mathbf{x}_i, y_i), i = 1, \dots, N\}$ where $\mathbf{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y} = \{+1, -1\}$. Our objective is to find an optimal hypothesis/classifier $g : \mathcal{X} \to \mathcal{Y}$ to classify the data. Let $\mathcal{H} = \{g : \mathcal{X} \to \mathcal{Y}\}$ be the family of classifiers available in our ensemble. In the AdaBoost setting, we are concerned with the classifiers that perform somehow better than 'tossing a fair coin'. This means that for each classifier, the

family $\mathcal{H}$ can predict y_i correctly at least half of the time.
We construct the boosted classifier as follows:

1. Initialize

$$w_{t=1}(\mathbf{x}_n) = \frac{1}{N}, \quad n = 1, \dots, N.$$

2. For $t = 1, \dots, T$ (desired termination step) do:
   a) Select a hypothesis $g_t \in \mathcal{H}$ that minimizes the weighted error

$$\epsilon_t = \sum_{i=1}^{N} w_t(\mathbf{x}_i) \mathbb{I}(g_(\mathbf{x}_i) \neq y_i). \tag{2.64}$$

   b) Let $\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$ update the weight for each data $\mathbf{x}_n$ by

$$w_{t+1}(\mathbf{x}_n) \leftarrow w_t(\mathbf{x}_n) \frac{\exp[-\alpha_t y_n g_t(\mathbf{x}_n)]}{Z_t},$$

   where $Z_t = \sum_{n=1}^{N} w_t(\mathbf{x}_n) e^{-\alpha_t y_n g_t(\mathbf{x}_n)}$ ensures all weights add up to unity.
   c) Output

$$g_A(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t g_t(\mathbf{x}) \right).$$

2.8.3 *Random Forests*

We now review one of the most widely used and versatile algorithms in data science and ML, *Random Forests* (RF).

Random Forests

Random Forests is an ensemble method widely deployed for complex classification tasks. A random forest is composed of a family of (randomized) tree-based classifier decision trees, cf. 2.4.2.1.

In order to create an ensemble of decision trees, we must introduce a randomization procedure. The power of ensembles to reduce variance only manifests when randomness reduces correlations between the classifiers within the ensemble, as discussed in 2.8.1.1. Randomness is usually introduced into random forests in one of three distinct ways.

1. Use bagging (cf. 2.8.2.1) and simply '*bag*' the decision trees by training each decision tree on a different bootstrapped dataset. Strictly speaking, this procedure dos not constitute a random forest but rather a bagged decision tree.

2. Only use a different random subset of the featues at each split in the tree. This *feature bagging* is the distinguishing characteristic of random forests. Using feature bagging reduces correlations between decision trees that can arise when only a few features are strongly predictive of the class label.

3. Extremized random forests (ERF) combine ordinary and feature bagging with an extreme randomization procedure where splitting is done randomly instead of using optimality criteria. Even though this reduces the predictive power of each individual decision tree, it still often improves the predictive power of the ensemble because it dramatically reduces correlations between members and prevents overfitting.

Why are they cool

RFs are largely immune to overfitting problems even as the number of estimators in the ensemble becomes large. By random selection of input features, random forest improves the variance reduction of bagging by reducing the correlation between the trees without dramatic increase of variance.

There are two main hyper-parameters that will be important in practice for the performance of the algorithm and the degree to which it overfits/underfits: the number of estimators in the ensemble and the depth of the trees used.

2.8.4 *Gradient Boosted Trees and XGBoost*

The basic idea of gradient-boosted trees is to use intuition from boosting (cf. 2.8.2.2) and gradient descent (cf. 2.5) to construct ensembles of decision trees (for decision trees, see 2.4.2.1). Like in boosting, the ensembles are created by iteratively adding new decision trees to the ensembles.

Gradient Boosted Trees - Essence

In gradient boosted trees, one critical component is a cost function that measures the performance of our ensemble. At each step, we compute the gradient of the cost function w.r.t. the predicted value of the ensemble and add trees that move us in the direction of the gradient.

Of course, this requires a clever way of mapping gradients to decision trees, this is done within XGBoost (Extreme Gradient Boosting) 2.8.4.1

2.8.4.1 *XGBoost*

What follow is a rough sketch of the XGBoost algorithm.
Our starting point is a clever parametrization of decision trees. Here,

we use notation where the decision tree makes continuous predictions (regression trees), though this can also be generalized to classification tasks. We parametrize a decision tree j, denoted as $g_j(\mathbf{x})$ with T leaves by two quantities: a function $q : \mathbf{x} \in \mathbb{R}^d \to \{1, 2, \dots, T\}$ that maps each data point to one of the leaves of the tree and a weight vector $\mathbf{w} \in \mathbb{R}^T$ that assigns a predicted value to each leaf. In other words, the j-th decision tree's prediction for the data point $\mathbf{x}_i$ is simply $g_j(\mathbf{x}_i) = w_{q(\mathbf{x}_i)}$. In addition to a parametrization of decision trees, we also have to specify a cost function which measures predictions. The prediction of our ensemble for a datapoint $(y_i, \mathbf{x}_i)$ is given by

$$\hat{y}_i = g_A(\mathbf{x}_i) = \sum_{j=1}^{M} g_j(\mathbf{x}_i), \quad g_j \in \mathcal{F}$$

where M is the number of members of the ensemble, and $\mathcal{F} = \{g(\mathbf{x}) = w_{q(\mathbf{x})}\}$ is the space of trees. As discussed for 2.4.2.1, we introduce a regularization term into the cost function to prevent overfitting.

Cost function XGBoost

The cost function is composed of two terms, a term that measures the goodness of predictions on each datapoint, $l_i(y_i, \hat{y}_i)$, which is assumed to be differentiable and convex, and for each tree in the ensemble, a regularization term $\Omega(g_j)$ that does not depend on the data

$$C(\mathbf{X}, g_A) = \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{j=1}^{M} \Omega(g_j), \tag{2.65}$$

where the index i runs over data points and the index j runs over decision trees in our ensemble. In XGBoost, the regularization function is chosen to be

$$\Omega(g) = \gamma T + \frac{\lambda}{2} \|\mathbf{w}\|_2^2, \tag{2.66}$$

with γ and λ regularization parameters that must be chosen appropriately. Notice that this regularization penalizes both large weights on the leaves (similar to L^2-regularization) and having large partitions with many leaves.

As in boosting, we form the ensemble iteratively. For this reason, we define a family of predictors $\hat{y}_i^{(t)}$ as

$$\hat{y}_i^{(t)} = \sum_{j=1}^{t} g_j(\mathbf{x}_i) = \hat{y}_i^{(t-1)} + g_t(\mathbf{x}_i). \tag{2.67}$$

Note that by definition $y_i^{(M)} = g_A(\mathbf{x}_i)$.

The central idea is that for large t, each decision tree is a small perturbation to the predictor (of order $1/T$) and hence we can perform a Taylor expansion on our loss function to second order

$$C_t = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + g_t(\mathbf{x}_i)) + \Omega(g_t) \approx C_{t-1} + \Delta C_t, \quad (2.68)$$

where

$$\Delta C_t = a_i g_t(\mathbf{x}_i) + \frac{1}{2} b_i g_t(\mathbf{x}_i)^2 + \Omega(g_t).$$

We then choose the t-th decision tree g_t to minimize ΔC_t, one finds

$$\Delta C_t^{opt} = -\frac{1}{2} \sum_{j=1}^{T} \frac{A_j^2}{B_j + \lambda} + \gamma T, \quad (2.69)$$

where $B_j = \sum_{i \in I_j} b_i, A_j = \sum_{i \in I_j} a_i$, with the set of points that get mapped to leaf j $I_j = \{i | q_t(\mathbf{x}_i) = j\}$.

It is clear that ΔC_t^{opt} measures the in-sample performance of g_t and we should find the decision tree that minimizes this value. In principle, one could enumerate all possible trees over the data and find the tree that minimizes ΔC_t^{opt}. However, in practice this is impossible. Instead, an approximate greedy algorithm is run that optimizes one level of the tree at a time by trying to find optimal splits of the data. This leads to a tree that is a good local minimum of ΔC_t^{opt} which is then added to the ensemble.

As a note about the sketch above, in practice additional regularization such as *shrinkage* and *feature subsampling* is used. In addition, there are many numerical and technical tricks used for the approximate algorithm and how to find splits of the data that give good decision trees.

Fscore

One nice feature of ensemble methods such as XGBoost is that they automatically allow us to calculate feature scores (*Fscores*) that rank the importance of various features for classification. The higher the Fscore, the more important the feature for classification.

2.9 AN INTRODUCTION TO FEED-FORWARD DEEP NEURAL NETWORKS (DNNS)

Neural networks are one of the most powerful and widely-used supervised learning techniques. Deep Neural Networks (DNNs) got rebranded as 'Deep Learning' and have become the workhorse technique for many image and speech recognition based ML tasks. The large-scale industrial deplyoment of DNNs has given rise to a number of high-level libraries and packages (Caffe, Keras, Pytorch, Tensorflow, etc.).
Conceptually, it is helpful to divide NNs into four categories

1. General purpose NNs for supervised learning,
2. NNs designed specifically for image processing, the most prominent example of this class being Convolutional NNs (CNNs),
3. NNs for sequential data such as Recurrent NNs (RNNs), and
4. NNs for unsupervised learning such as Deep Boltzmann Machines.

This section deals with the first two categories.[8] Results for modern NNs are largely empirical and heuristic and lack a firm theoretical footing. Therefore, in this review we (for now) only give the fundamentals.

2.9.1 *Neural Network Basics*

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. Neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear 2.6 and logistic regression 2.7 and soft-max methods 2.7.5.

2.9.1.1 *The basic building block: neurons*

Neurons

The basic unit of a neural net is a stylized *neuron* i that takes a vector of d input features $\mathbf{x} = (x_1, x_2, \dots, x_d)$ and produces a scalar output $a_i(\mathbf{x})$.

A neural network consists of many such neurons stacked into layers, with the output of one layer serving as the input for the next. The first layer in the neural net is called the *input layer*, the middle layers are often called *hidden layers*, and the final layer is called the *output layer*. The exact function a_i varies depending on the type of non-linearity used in the NN. However, in essentially all cases a_i can be decomposed

8 Though increasingly important for many applications such as audio and speech recognition, for now we omit a discussion of sequential data and RNNs. Look at Chris Olah's blog for an introduction to RNNs and LSTM networks.

into a linear operation that weights the relative importance of the various inputs, and a non-linear transformation $\sigma_i(z)$ which is usually the same for all neurons[9], compare 2.4.

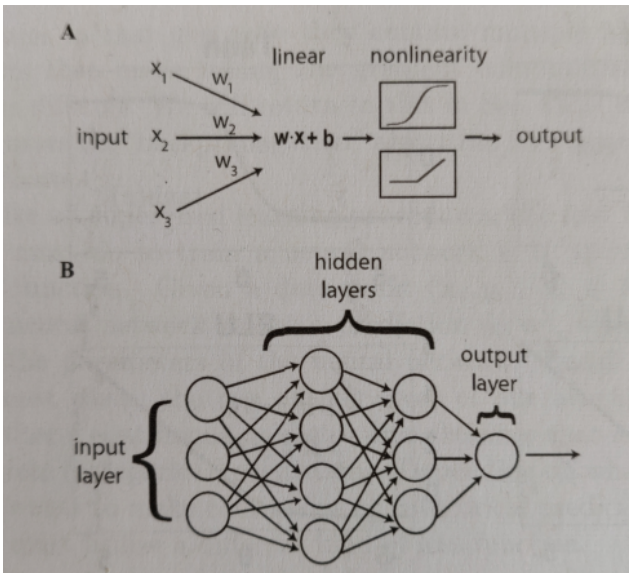

Figure 2.4: *A) Neurons consist of a linear transformation that weights the importance of various inputs, followed by a non-linear activation function. B) Network architecture.*

The linear transformation in almost all NNs takes the form of a dot product with a set of neuron-specific weights $\mathbf{w}^{(i)} = (w_1^{(i)}, w_2^{(i)}, \dots, w_d^{(i)})$ followed by re-centring with a neuron-specific bias $b^{(i)}$:

$$z^{(i)} = \mathbf{w}^{(i)} \cdot \mathbf{x} + b^{(i)} = \mathbb{x}^T \cdot \mathbb{w}^{(i)}, \tag{2.70}$$

where $\mathbb{x} = (1, \mathbf{x})$ and $\mathbb{w}^{(i)} = (b^{(i)}, \mathbf{w}^{(i)})$.

9 I.e. a threshhold function which squishes the sum of all a_i into a number between 0 and 1, cf. 2.7.2.1.

Choosing the right non-linearity

In terms of $z^{(i)}$ and the non-linear function $\sigma_i(z)$, we can write the full input-output function as

$$a_i(\mathbf{x}) = \sigma_i(z^{(i)}). \tag{2.71}$$

Historical choices of nonlinearities include step-functions (perceptrons), sigmoids (i.e. Fermi functions), and the hyperbolic tangent, cf. 2.7.2.1. More recently, it has become more common to use rectified linear units (ReLUs), leaky rectified linear units (leaky ReLUs), and exponential linear units (ELUs). Different choice of non-linearities lead to different computational and training properties for neurons. The underlying reason for this is that we train neural nets using gd based methods, cf. 2.5, that require us to take derivatives of the neural input-output function with respect to the weights $\mathbf{w}^{(i)}$ and the bias $b^{(i)}$.

Notice that the derivatives of the aforementioned non-linearities $\sigma(z)$ have very different properties. The derivative of the perceptron is zero everywhere except where the input is zero. This discontinuous behaviour makes it impossible to train perceptrons using gradient descent. For this reason, until recently the most popular choice of non-linearity was the tanh function or a sigmoid/Fermi function. However, this choice of non-linearity has a major drawback. When the input weights become large, as they often do in training, the activation functions saturates and the derivative of the output w.r.t. the weights tends to zero since $\partial\sigma/\partial z \to 0$ for $z \gg 1$. Such *vanishing gradients* are a feature of any saturating function, making it harder to train deep networks. In contrast, for a non-saturating function such as ReLUs or ELUs, the gradients stay finite even for large inputs.

2.9.1.2 *Layering neurons to build deep networks: network architecture*

The basic idea of all NNs is to layer neurons in a hierarchical fashion, the general structure of which is known as the *network architecture*, compare 2.4. In the simplest feed-forward networks, each neuron in the *input layer* of the neurons takes the inputs $\mathbb{x}$ and produces an output $a_i(\mathbb{x})$ that depends on its current weights, see 2.71. The outputs of the input layer are then treated as the inputs to the next *hidden layer*. This is usually repeated several times until one reaches the top or *output layer*.

Output layer and general architecture

The output layer is almost always a simple classifier of the form discussed before: a logistic regression 2.7 or soft-max function 2.7.5 in the case of categorical data (i.e. discrete labels) or a linear regression 2.6 layer in the case of continuous outputs. Thus, the whole neural network can be though of as a complicated nonlinear transformation of the inputs $\mathbb{x}$ into an output $\hat{y}$ that depends on the weights and biases of all the neurons in the input, hidden, and output layers.

The use of hidden layers greatly expands the representation power (*expressivity*) of a NN when compared with a simple soft-max or linear regression network, this is exemplified by the following theorem.

Universal approximation theorem

A NN with a single hidden layer can approximate any continuous multi-input/multi-output function with arbitrary accuracy.

What does this mean and where does this come from ?
Note that by increasing the number of hidden neurons we can improve the approximation. Suppose we're given a function $f(x)$ which we'd like to compute to within some desired accuracy $\epsilon > 0$. The guarantee is that by using enough hidden neurons we can always find a neural network whose output $g(x)$ satisfies

$$|g(x) - f(x)| < \epsilon,$$

for all inputs x. The approximation of the function works in the following way like a Riemann sum. Basically every pair of neurons in the hidden layer can be combined, via a combination of both weights, into a step function (or a sigmoid function which becomes a step function for large inputs). To be more precise, you can make every activation function into a step function by a clever combination of weights and bias if the activation function satisfies the following properties:
We do need to assume that $s(z)$ is well-defined as$z \to \infty$ and $z \to -\infty$. These two limits are the two values taken on by our step function. We also need to assume that these limits are different from one another. If they weren't, there'd be no step, simply a flat graph! But provided the activation function $s(z)$ satisfies these properties, neurons based on such an activation function are universal for computation (i.e. they are able to approximate any function). Why this is is described in the following.
Having N pairs of hidden layer neuron pairs gives you N step function, which you can arrange beside each other, or overlapping. This is basically the approximation of the function. You arrange step functions in such a way that they give you the function in a small section of the interval. Therefore, basically neuron pairs in the hidden layer provide

support functions as in analysis, which are step functions approximating the true function for a small part of the interval. Therefore, by simply increasing the number of hidden layer neurons, you increase the number of step functions and decrease their width, resulting in a finer approximation of the function.
This makes use of the weighted combination $\sum_j w_j a_j$ output from the hidden neurons, however the function is given by the output layer, i.e. by the output $\sigma(\sum_j w_j a_j + b)$ where b is the bias on the output neuron. How do you make the transition ?
The solution is to design a neural network whose hidden layer has a weighted output given by $\sigma^{-1} \circ f(x)$, such that the overall output will be a very good approximation of $\sigma \circ \sigma^{-1} \circ f(x) = f(x)$.
If you have more inputs for one neuron, i.e. a many-input function $f(x_1, x_2, \dots)$, you get more weights, because every input x_i has a weight w_i associated with it which determines how much this input variable should contribute in this particular neuron, where every neuron also has one specific bias associated with it.
For two input variables:
You can again create step function but this time in three dimensions (i.e. $x_1, x_2, g(x_1, x_2)$), which are called tower functions. These are constructed by introducing a second hidden layer. A pair of Neurons in the first hidden layer giving us a step-function in one direction can be combined with a pair of neurons giving us a step function in another direction (and so on) to create a tower function by a clever combination of heights of the two neuron pairs and bias of the neuron in the second hidden layer both of these pairs are connected to. Then, you can create any 3 or m dimensional function by stacking tower functions together. We can also make them as thin as we like, and whatever height we like. As a result, we can ensure that the weighted output from the second hidden layer approximates any desired function of two variables. In particular, by making the weighted output from the second hidden layer a good approximation to $\sigma^{-1} \circ f$, we ensure the output from our network will be a good approximation to any desired function, f.
For m input variables you need to have m neuron pairs in the first hidden layer to feed into one respective neuron in the second hidden layer, where the former create all the step functions in the different dimensions by clever combination of parameters and the latter gives the output tower function in the given dimension.
For vector-valued functions, you simply compute the components of the vector separately via the above method and then put them into a vector.

Conclusion

We saw that we can approximate any function with just one hidden layer. Given this, you might wonder why we would ever be interested in deep networks, i.e., networks with many hidden layers. Can't we simply replace those networks with shallow, single hidden layer networks?
While in principle that's possible, there are good practical reasons to use deep networks. Deep networks have a hierarchical structure which makes them particularly well adapted to learn the hierarchies of knowledge that seem to be useful in solving real-world problems. Put more concretely, when attacking problems such as image recognition, it helps to use a system that understands not just individual pixels, but also increasingly more complex concepts: from edges to simple geometric shapes, all the way up through complex, multi-object scenes. Deep networks do a better job than shallow networks at learning such hierarchies of knowledge.

2.9.1.3 *Further on network architecture: How many layers is adequate for a problem ?*

Deep architectures are favourable for learning. Increasing the number of layers increases the number of parameters and hence the representational power of NNs. Indeed, recent numerical experiments suggest that as long as the number of parameters is larger than the number of data points, certain classes of NNs can fit arbitrarily labelled random noise samples. This suggests that large NNs of the kind used in practice can express highly complex function. Adding hidden layers is also thought to allow neural nets to learn more complex features from the data. Work on convolutional networks suggests that the first few layers of a neural network learn simple, 'low-level' features that are then combined into higher-level, more abstract features in the deeper layers.

Choosing the exact network architecture for a NN remains an art that requires extensive numerical experimentation and intuition, and is often times problem-specific. Both the number of hidden layers and the number of neurons in each layer can affect the performance of a NN.

Rule of thumb

A general rule of thumb that seems to be emerging is that the number of parameters in the NN should be large enough to prevent underfitting.

Empirically, the best architecture for a problem depends on the task, the amount and type of data that is available, and the computational

resources at one's disposal. Certain architectures are easier to train, while others might be better at capturing complicated dependencies in the data and learning relevant input features. Finally, there are architectures beyond simple deep, feed-forward NNs. For example, modern NNs for image segmentation often incorporate 'skip connections' that skip layers of NN. This allows information to directly propagate to a hidden or output layer, bypassing intermediate layers and often improving performance.

2.9.2 *Training deep networks*

Training procedure

The training procedure is the same as we used for training simpler supervised learning algorithms such as logistic and linear regression, compare 2.1.1:
Construct a cost/loss function and then use gradient descent to minimize the cost function and find the optimal weights and biases.

NNs differ from these simpler supervised procedures in that generally they contain multiple hidden layers that make taking the gradient computationally more difficult, this will be discussed in more detail via the *backpropagation* algorithm for computing gradients in 2.9.3.

Like all supervised learning procedures, the first thing one must do to train a NN is to specify a loss function. Given a data point $(\mathbb{x}_i, y_i)$, $\mathbb{x}_i \in \mathbb{R}^{d+1}$, the NN makes a prediction $\hat{y}_i(\mathbb{w})$, where $\mathbb{w}$ are the parameters of the NN. Recall that in most cases, the top output layer of our NN is either a continuous predictor or a classifier that makes discrete (categorical) predictions. Depending on whether one wants to make continuous or categorical predictions, one must utilize a different kind of loss function.

1. For continuous data, the loss functions that are commonly used to train NNs are identical to those used in linear regression:
   a) One is the mean squared error
   $$E(\mathbb{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i(\mathbb{w}))^2, \tag{2.72}$$
   where n is the number of data points,
   b) and the mean-absolute error (i.e. L_1 norm)
   $$E(\mathbb{w}) = \frac{1}{n}\sum_{i}|y_i - \hat{y}_i(\mathbb{w})|. \tag{2.73}$$
   The full cost function often includes terms that implement regularization (e.g. L_1 or L_2 regularizers).

2. For categorical data, the most commonly used loss function is the cross-entropy (2.53 and 2.57), since the output layer is often taken to be a logistic classifier for binary data with two types of labels, or a soft-max classifier if there are more than two types of labels. As usual, these loss functions are often supplemented by aditional terms that implement regularization.

Having defined an architecture and a cost function, we must now train the model. Similar to other supervised learning methods, we make use of gradient-descend based methods 2.5 to optimize the cost function. Recall that the basic idea of GD is to update the parameters $\mathbb{w}$ to move in the direction of the gradient of the cost function $\nabla_{\mathbb{w}} E(\mathbb{w})$.[10]

2.9.3 *The Backpropagation algorithm*

The training procedure of a NN requires us to be able to calculate the derivative of the cost function w.r.t. all the parameters of the NN (the weights and biases of all the neurons in the input, hidden, and visible layers). The backpropagation algorithm is a clever procedure that exploits the layered structure of NNs to more efficiently compute gradients.

We will assume that there are L layers in our network with $l = 1, \dots, L$ indexing the layer. Denote by w^l_{jk} the weight for the connection from the k-th neuron in layer $l-1$ to the j-th neuron in layer l We denote the bias of this neuron by b^l_j. By construction, in a feed-forward NN the activation a^l_j of the j-th neuron in the l-th layer can be related to the activities of the neurons in the layer $l-1$ by the equation

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) = \sigma(z^l_j). \tag{2.74}$$

Define the error Δ^L_j of the j-th neuron in the L-th layer as the change in cost function w.r.t. the weighted input z^L_j

$$\Delta^L_j = \frac{\partial E}{\partial z^L_j}, \tag{2.75}$$

where E is the cost function which depends directly on the activities of the output layer a^L_j and indirectly on all the activities of neurons in lower layers iteratively via 2.74. Equation 2.75 therefore asks the question of how much the cost function changes when we change the weighted sum. Equally, the following three equations ask the question of how the cost function changes w.r.t. a change in the bias, weight and

10 Most modern NN packages, such as Keras, allow the user to specify which of the optimizers discussed in 2.5 they would like to use in order to train the NN. Depending on the architecture, data, and computation resources, different optimizers may work better on the problem, though vanilla SGD 2.5.2.1 is a good first choice.

activation respectively. Equation 2.75 together with the following three equations

$$\Delta_j^l = \frac{\partial^E}{\partial z} = \frac{\partial E}{\partial b_j^l}, \tag{2.76}$$

$$\Delta_j^l = \sum_k \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{z_j^l} = \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l), \tag{2.77}$$

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}. \tag{2.78}$$

define the four backpropagation equations relating the gradients of the activations of various neuron a_j^l, the weighted inputs $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$, and the errors Δ_j^l. These equations can be combined into a simple, computationally efficient algorithm to calculate th gradient w.r.t. all parameters. As all of them are asking the question of how the error/-cost function is affected by a change in one of the parameters, they are a means of quantifying how to most easily minimize the cost function, compare figure 2.5. This is what the backpropagation algorithm is doing. It combines the average amount of change of the error function w.r.t. one of the parameters into the gradient vector

$$\vec{\nabla} C = (w_1, w_2, \dots, w_{1000}, b_1, \dots, b_{400}).$$

The optimized gradient of the cost function is determined via the backpropagation algorithm by choosing the direction of change in this huge parameter space.

The Backpropagation algorithm

1. *Activation at input layer*:
   Calculate the activation a_j^1 of all the neurons in the input layer.

2. *Feedforward*:
   Starting with the first layer, exploit the feed-forward architecture through 2.74 to compute z^l and a^l for each subsequent layer.

3. *Error at top layer*:
   Calculate the error of the top layer using 2.75. This requires to know the expression for the derivative of both the cost function $E(\mathbb{W}) = E(\mathbf{a}^L)$ and the activation function $\sigma(z)$.

4. *'Backpropagate' the error*:
   Use 2.77 to propagate the error backwards and calculate Δ_j^l for all layers.

5. *Calculate gradient*:
   Use equations 2.76 and 2.78 to calculate $\frac{\partial E}{\partial b_j^l}$ and $\frac{\partial E}{\partial w_{jk}^l}$.

The algorithm consists of a forward pass from the bottom layer to the top layer where one calculates the weighted inputs and activations of all the neurons. One then *backpropagates* the error starting with the top layer down to the input layer and uses these errors to calculate the desired gradients. This description makes clear the incredible utility and computational efficiency of the backgpropagation algorithm. We can calculate all the derivatives using a single 'forward' and 'backward' pass of the NN. This computational efficiency is crucial since we must calculate the gradient w.r.t. all parameters of the NN at each step of gradient descent.[11]

2.9.3.1 *What can go wrong with backpropagation ?*

A problem that occurs in deep networks, which transmit information through many layers, is that gradients can vanish or explode. This is known as the *problem of vanishing or exploding gradients*. Especially pronounced in NNs that try to capture long-range dependencies, such as RNN for sequential data. Consider the eigenvalues (or singular values) of the weight matrices w_{jk}^l. In order for the gradients to be finite for deep networks, we need these eigenvalues to stay near unity even

11 These basic ideas also underlie almost all modern automatic differentiation packages such as Autograd (Pytorch).

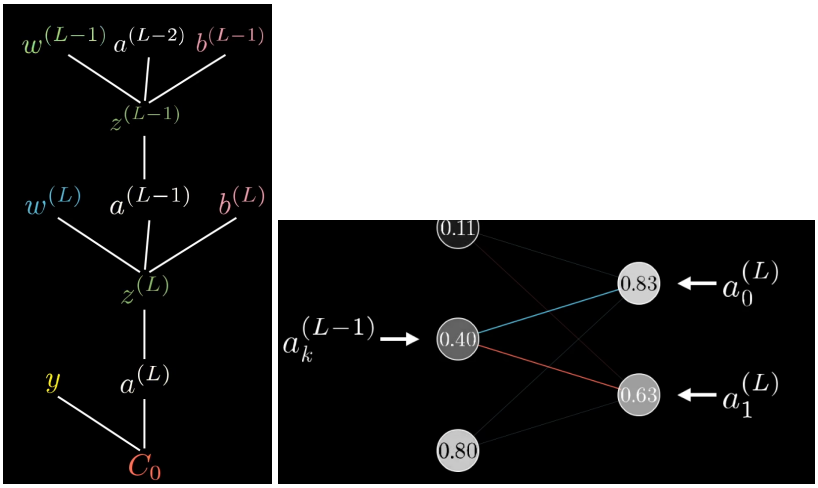

Figure 2.5: Left :*Backpropagation for a DNN with one neuron per layer.* Right: *L-th layer is introduced by multiple activations in the previous layer if we have more than one neuron.*

after many gradient descent steps.[12] Proper initialization and regularization schems such as *gradient clipping* (cutting-off gradients with very large values), and *batch normalization* also help mitigate this problem.
Summarizing Backpropagation:
Backpropagation is the algorithm for determining how a single training example would nudge all weights and biases of the NN - not in terms of whether they should go up or down but in terms of what relative proportions to those changes cause the *most rapid decrease of the cost function*. A true GD step would involve doing this for all your tens of thousands of training examples and averaging the desired changes that you get (i.e. average change to weight w_{143}). This however is computationally slow, such that you randomly subdivide the data into mini-batches and compute each step w.r.t. a mini-batch. Repeatedly going through all the mini-batches making these adjustments to the weights, you will converge towards a local minimum.

2.9.4 *Regularizing neural networks and other practical considerations*

DNNs, like all supervised learning algorithms, must navigate the bias-variance tradeoff 2.1.2.3. Regularization techniques play an important role in ensuring that DNNs generalize well to new data. In addition to special regularization techniques (Batch Normalization, Dropout), large DNNs seem especially well-suited to implicit regularization that already takes place in SGD 2.5.2.1. The implicit stochasticity and local nature of SGD often prevent overfitting of spurious correlations in the

12 In modern feed-forward and ReLU NNs, this is achieved by initializing the weights for the gradient descent in clever ways and using non-linearities that do not saturate, such as ReLUs (recall that for saturating functions, $\sigma' \to 0$, which will cause the gradient to vanish).

training data, especially when combined with techniques such as Early Stopping.

2.9.4.1 *Implicit regularization using SGD: Initialization, hyper-parameter tuning, and Early Stopping*

Implicit regularization using SGD: initialization, hyper-parameter tuning, and Early Stopping.
The most commonly employed and effective optimizer for training NN is SGD. SGD acts as an implicit regularizer by introducing stochasticity (from the use of mini-batches) that prevents overfitting.

1. In order to achieve good performance, it is important that the weight initialization is chosen randomly, in order to break any leftover symmetries. One common choice is drawing the weights from a Gaussian centred around zero with some variance that scales inversely with number of inputs to the neuron.

   > Since SGD is a local procedure, as networks gets deeper, *choosing a good weight initialization becomes increasingly important* to ensure that the gradients are well-behaved.

   It is important to experiment with different variances, the NN is extremely sensitive to the choice of variance.

2. The second important thing is to appropriately choose the learning rate or step-size by searching over five logarithmic grid points. If the best performance occurs at the edge of the grid, repeat this procedure until the optimal learning rate is in the middle of the grid parameters.

3. Finally, it is common to centre or whiten the input data (as in linreg 2.6 and logreg 2.7).

Another important form of regularization that is often employed in practice is *Early Stopping*.

Early Stopping

The idea of Early Stopping is to divide the training data into two portions, the dataset we train on, and a smaller *validation set* that serves as a proxy for out-of-sample performance of the test set (i.e. cross-validation 2.1.1, note however that you also split test test into test and play set, where you perform hyperparameter grid search on the play set). As we train the model, we plot both the training error and the validation error (i.e. E_{in}, E_{out}). We expect the training error to continuously decrease during training ($E_{in,t} \leq E_{in,t-1}$). However, the validation error will eventually increase due to overfitting. The basic idea of Early Stopping is to halt the training procedure when the validation error starts to rise. This Early Stopping procedure ensures that we stop the training and avoid fitting sample specific features in the data.

2.9.4.2 *Dropout*

Dropout

The basic idea of Dropout is to prevent overfitting by reducing spurious correlations between neurons within the network by introducing a randomization procedure similar to that underlying ensemble models such as Bagging 2.8.2.1. Dropout prevents problems of correlation and computational cost by randomly dropping out neurons (along with their connections) from the NN during each step of the training. Typically, for each mini-batch in the GD step, a neuron is dropped from the NN with a probability p. The GD step is then performed only on the weights of the 'thinned' network of individual predictors. Since during training, on average weights are only present a fraction p of the time, predictions are made by reweighing the weights by p: $\mathbb{w}_{test} = p\mathbb{w}_{train}$. The learned weights can be viewed as some 'average' weight over all possible thinned NNs, this is similar to Bagging and reduces the variance.

2.9.4.3 *Batch Normalization*

The basic inspiration is that training NNs works best when the inputs are centred around zero w.r.t. the bias. The reason for this is that it prevents neurons from saturating and gradients from vanishing in deep nets.

Batch Normalization

The idea of Batch Normalization is to introduce new 'BatchNorm' layers that standardize the inputs by the mean and variance of the mini-batch. Consider a layer l with d neurons whose inputs are $(z_1^l, \dots, z_d^l)$. We standardize each dimension so that

$$z_k^l \to \hat{z}_k^l = \frac{z_k^l - \mathbb{E}[z_k^l]}{\sqrt{Var[z_k^l]}}, \tag{2.79}$$

where the mean and variance are taken over all samples in the mini-batch. One furthermore introduces two new parameters γ_k^l, β_k^l for each neuron that can additionally shift and scale the normalized input

$$\hat{z}_k^l \to \gamma_k^l \hat{z}_k^l + \beta_k^l. \tag{2.80}$$

These two equations 2.79, 2.80 are like adding new extra layers $\hat{z}$ in the deep net architecture.

Hence, the new parameters γ_k^l and β_k^l can be learned just like the weights and biases using backpropagation (since this is just an extra layer for the chain rule). We initiliaze the NN so that at the beginning of training the inputs are being standardized. Backpropagation then adjusts γ, β during training.
In practice, Batch Normalization considerably improves the learning speed by preventing gradients from vanishing. The randomness introduced by the mini-batches seems to shape Batch Normalization as a power regularization tool by introducing additional randomness into the training procedure.

2.9.5 *Deep neural networks in practice*

Different packages:

1. In Tensorflow one constructs data flowgraphs, the nodes of which represent mathematical operations, while the edges encode mutlidimensional tensors (data arrays). A DNN can then be thought of as a graph with a particular architecture and connectivity.

2. PyTorch offers libraries for automatic differentiation of tensors at GPU speed. As we discussed above, manipulating NNs boils down to fast array multiplication and contraction operations and, therefore, the **torch.nn** library often does the job.

3. Keras is a high-level package which does simple jobs in a few lines of code, but does not give you much control for more complicated processes.

2.9.6 *Recipe DNNs*

Constructing a Deep Neural Network to solve supervised ML problems is a multiple-stage process. Quite generally, one can identify the key steps as follows:

1. Collect and pre-process the data.
2. Define the model and its architecture
3. Choose the optimizer and the cost function
4. Train the model
5. Evaluate and *study*[13] the model performance on the *unseen* test data
6. Use the validation data to adjust the hyperparameters (and, if necessary, network architecture) to optimize performance for the specific dataset.

A few comments:

1. While we treat step 1 above as consisting mainly of loading and reshaping a dataset prepared ahead of time, we emphasize that obtaining a sufficient amount of data is a typical challenge in many applications. Oftentimes insufficient data serves as a major bottleneck on the ultimate performance of DNNs. In such cases one can consider *data augmentation*, i.e. distorting data samples from the existing dataset in some way to enhance size of the dataset. Obviously, if one knows how to do this, one already has partial information about the important features in the data.
2. How do you determine the split ratio into training and validation dataset ?

   Rule of thumb

   The more classification categories there are in the task, the closer the sizes of the training and test datasets should be ($\rightarrow$ 50/50) in order to prevent overfitting.

   Once the size of the training set is fixed, it is common to reserve 20% of it for validation , which is used to fine-tune the hyperparameters of the model.
3. Also related to preprocessing is the standardization of the dataset. As discussed in backpropagation 2.9.3, the problem of vanishing and exploding gradients come up when values of the data differ by orders of magnitude. Two approaches are possible:

13 I.e. early stopping

a) All data should be mean-centred, i.e. from every data point we subtract the mean of the entire dataset.

b) Rescale the data - for which there are two ways:

   i. If the data is approximately normally distributed one can rescale by the standard deviation.

   ii. Otherwise, it is typically rescaled by the maximum absolute value so the rescaled data lies within the interval $[-1, 1]$ (i.e. .reshape$(-1, 1)$). Rescaling ensures that the weights of the DNN are of a similar order of magnitude, compare Batch Normalization 2.9.4.3.

4. How to choose the right hyperparameters to start training the model ?
   The optimal learning rate is often an order of magnitude lower than the smallest learning rate that blows up the loss. It is usually a good idea to play with a small enough fraction of the training data to get a rough feeling about the correct hyperparameter regimes, the usefulness of the DNN architecture, and to debug the code. The size of this small 'play set' should be such that training on it can be done fast and in real time to allow to quickly adjust the hyperparameters. A typical strategy of exploring the hyperparameter landscape is to use grid searches.

Whereas it is always possible to view Steps $1 - 5$ as generic and independent of the particular problem we are trying to solve, it is only when these steps are put together in Step 6 that the real benefit of DL is revelaed.
The optimal choice of network architecture 2.9.1.2,2.9.1.3, cost function 2.9.2, and optimizer is determined by the properties of the training and test datasets, which are usually revealed when we try to improve the model 2.9.4.

2.10 CONVOLUTIONAL NEURAL NETWORKS (CNNS)

2.10.1 *Symmetries*

One of the core lessons of physics is that we should **explot symmetries and invariances when analyzing physical systems.** Likey physical systems, many datasets and supervised learning tasks also possess additional symmetries and structure.

Example

For instance, consider a supervised learning task where we want to label images from some dataset as being pictures of cats or not. Our statistical procedure must first learn features associated with cats. Because a cat is a physical object, we know that these features are likely to be local (groups of neighbouring pixels in the $2D$ image corresponding to whiskers, tails, etc.). We also know that the cat can be anywhere in the image. Thus, it does not really matter where in the picture these features occur (though relative positions of features likely do matter). *This is a manifestation of translational invariance that is built into our supervised learning task.*

The all-to-all coupled NNs in 2.9 fail to exploit this additional structure. CNNs take advantage of this additional structure (locality and translational invariance).

2.10.2 *The structure of convolutional neural networks*

A CNN is a translationally invariant NN that respects locality of the input data[14].

2.10.2.1 *Architecture*

There are two kinds of basic layers that make up a CNN:

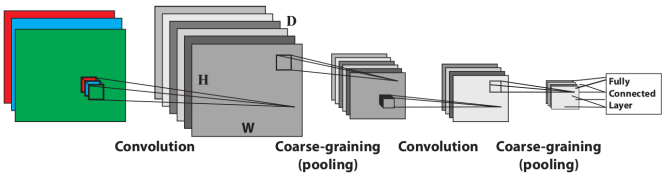

Figure 2.6: Architecture of a CNN: *The neurons in a CNN are arranged in three dimension: height (H), width (W), and depth (D). For the input layer, the depth corresponds to the number of channels (in this case 3 for RGB images). Neurons in the convolutional layers calculate the convolution of the image with a local spatial filter (e.g. 3×3 pixel grid, times 3 channels for first layer) at a given location in the spatial (W, H)-plane. The depth D of the convolutional layer corresponds to the number of filters used in the convolutional layer. Neurons at the same depth correspond to the same filter. Neurons in the convolutional layer mix inputs at different depths but preserve the spatial location. Pooling layers perform a spatial coarse graining (pooling step) at each depth to give a smaller height and width while preserving the depth. The convolutional and pooling layers are followed by a fully connected layer and classifier (not shown).*

14 Excellent course in Andrej Karpathy and Fei-Fei Li.

1. A convolution layer that computes the convolution of the input with a bank of filters[15],

2. and pooling layers that coarse-grain the input while maintaining locality and spatial structure, compare fig. 2.6.

For two-dimensional data, a layer l is characterized by three numbers: height H_l, width W_l, and depth D_l[16] The height and width correspond to the sizes of the two-dimensional spatial (W_l, H_l)-plane (in neurons), and the depth D_l (marked by the different colours in 2.6)-to the number of filters in that layer. All neurons corresponding to a particular filter have the same parameters (i.e. shared weights and bias).
In general, we will be concerned with local spatial filters (often called a *receptive field* in analogy with neuroscience) that take as inputs a small spatial patch of the previous layer at all depths. For instance, a square filter os fize F is a three-dimensional array of size $F \times F \times D_{l-1}$. The convolution consists of running this filter over all locations in the spatial plane.

Example

To demonstrate how this works in practice, let us consider the simple example consisting of a one-dimensional input of depth 1, shown in fig 2.7. In this case, a filter of size $F \times 1 \times 1$ can be specified by a vector of weights w of length F.

The stride, S, encodes by how many neurons we translate the filter by when performing the convolution. In addition, it is common to pad the input with P zeroes (c.f. 2.7).

15 As a mathematical operation, see this practical guide to :Image kernels.

16 The depth D_l is often called 'number of channels', to distinguish it from the depth of the NN itself, i.e. the total number of layers (which can be convolutional, pooling or fully-connected), c.f. 2.6.

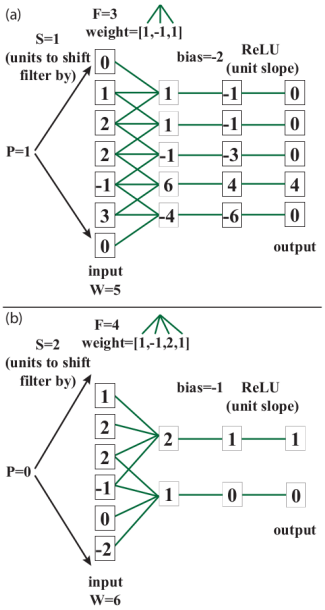

Figure 2.7: Two examples to illustrate a one-dimensional convolutional layer with ReLU nonlinearity:*Convolutional layer for a spatial filter of size F for a one-dimensional input of width W with stride S and padding P followed by a ReLU non-linearity.*

3 IDEAS FOR PROBLEMS TO WORK ON

3.1 PHYSICS PROBLEMS

3.1.1 *Cosmology*

1. Maybe write a classification logistic regressor, or deep CNN to classify where dark matter halos have their boundary. It is very difficult to define a boundary of such a spread out object with confidence, maybe find criteria via classification ML technique ?

3.1.2 *QFT*