MACHINE LEARNING IN A NUTSHELL

THIMO PREIS[1]

3rd 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.

2.1.2.2 *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 patters 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.

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.

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.

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

to get a better understanding of what it is we ought to be doing to achieve best practices.
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.

2.1.3 *title*

2.2 BASICS OF STATISTICAL LEARNING

Statistical modeling 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 concered 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 *Language*

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. 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 two distinct performance measures of interest. The in-sample error, E in , and the out-of-sample or generalization error, E out

2.2.2.1 *Set-up*

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

$$y = f(x) + \epsilon \tag{2.4}$$

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

$$C(\mathbf{y}, f(\mathbf{X}; \boldsymbol{\theta})) = \sum_i (y_i - f(x_i; \boldsymbol{\theta}))^2. \tag{2.5}$$

Therefore, the estimates for the parameters

$$\hat{\boldsymbol{\theta}} = \arg\min_\theta C(\mathbf{y}, f(\mathbf{X}; \boldsymbol{\theta})) \tag{2.6}$$

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*

$$E_{out} := \mathbb{E}_{\mathcal{D},\epsilon}[C(\mathbf{y}, f(\mathbf{X}; \hat{\boldsymbol{\theta}}_{\mathcal{D}}))], \tag{2.7}$$

can be decomposed as

$$E_{out} = \text{Bias}^2 + \text{Var} + \text{Noise}, \tag{2.8}$$

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.

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).
Thus, to minimize E_{out} and maximize our predic- tive power, it may be more suitable to use a more bi-

2.3 GRADIENT DESCENT

2.3.1 *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

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{2.9}$$

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).[1] 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.

1 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.

2.3.2 *Modified gradient descent*

2.3.2.1 *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

$$\mathbf{v}_t = \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t. \tag{2.10}$$

2.3.2.2 *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

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}_t), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t, \tag{2.11}$$

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

$$\mathbf{v}_t = \gamma \mathbf{v}_t + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t - \gamma \mathbf{v}_{t-1}), \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t. \tag{2.12}$$

2.3.2.3 *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[2]

2 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

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}),\ \mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2,\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \quad (2.13)$$

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)

$$\begin{aligned} &\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}), \quad \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \\ &\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2, \quad \hat{\mathbf{m}}_t = \frac{\mathbf{m}}{1-(\beta_1)^t}, \\ &\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-(\beta_2)^t}, \quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}, \end{aligned} \quad (2.14)$$

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.[3]

3 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.

2.3.3 *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.

2.4 OVERVIEW OF BAYESIAN INFERENCE

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

2.4.1 *Language*

Likelihood function

The *likelihood function* $p(\mathbf{X}|\boldsymbol{\theta})$ 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 reasonabe 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

$$p(\boldsymbol{\theta}|\lambda) = \prod_j \sqrt{\frac{\lambda}{2\pi}} e^{-\lambda\theta_j^2}, \tag{2.15}$$

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

$$p(\boldsymbol{\theta}|\lambda) = \prod_j \frac{\lambda}{2} e^{-\lambda|\theta_j|}. \tag{2.16}$$

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

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.

2.4.2 *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

$$p(\boldsymbol{\theta}|\mathbf{X}) = \frac{p(\mathbf{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{\int \mathrm{d}\boldsymbol{\theta}' p(\mathbf{X}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')}. \tag{2.17}$$

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

Maximum Likelihood Estimation (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 maxmimize the likelihood (or equivalently the log-likelihood since log is a monotonic function) of the observed data:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{X}|\boldsymbol{\theta}). \tag{2.18}$$

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