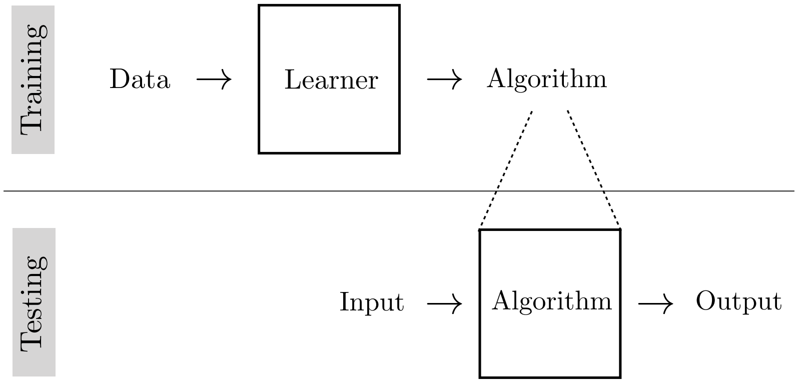
Chapter 11

Learning from examples

The goal of learning is to extract lessons from past experience in order to solve future problems. Typically, this involves searching for an **algorithm** that solves past instances of the problem. This algorithm can then be applied to future instances of the problem.

An **algorithm** is a formal mapping from inputs to outputs, e.g., from a problem statement to its solution.

Because learning is itself an algorithm, it can be understood as a meta-algorithm: an algorithm that outputs algorithms:



Learning usually consists of two phases: the **training** phase, where we search for an algorithm that performs well on past instances of the problem (training data), and the **testing** phase, where we deploy our learned algorithm to solve new instances of the problem.

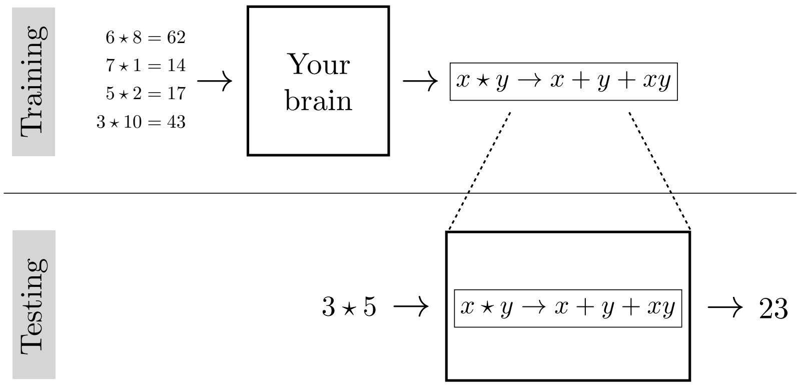
11.1 Learning from examples

Imagine you find an ancient mathematics text, with marvelous looking proofs, but there is a symbol you do not recognize, "$\star$". You see it being used here and there in equations, and you note down examples of its behavior:

$$\begin{aligned} 6 \star 8 &= 62 \\ 7 \star 1 &= 15 \\ 5 \star 2 &= 17 \\ 3 \star 10 &= 43 \end{aligned}$$

What do you think $\star$ represents? What function is it computing? Do you have it? Let's test your answer: what is the value of $3 \star 5$? (The answer is in the figure below.)

It may not seem like it, but you just performed learning! You *learned* what $\star$ does by looking at examples. In fact, here's what you did:



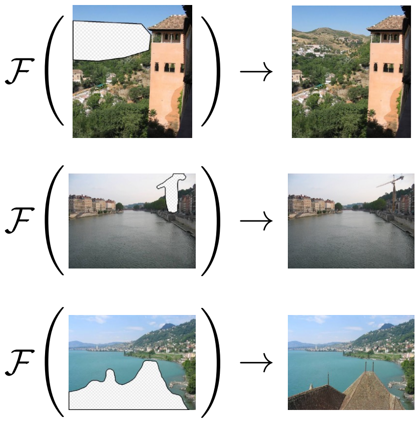
Nice job!

It turns out, we can learn almost anything from examples[1]. Remember that we are learning an *algorithm*, i.e. a computable mapping from inputs to outputs. A formal definition of *example*, in this context, is an {`input`, `output`} pair. The examples you were given for $\star$ consisted of four such pairs:

$$\{\texttt{input:}[6,8], \texttt{output:}62\}$$
$$\{\texttt{input:}[7,1], \texttt{output:}15\}$$
$$\{\texttt{input:}[5,2], \texttt{output:}17\}$$
$$\{\texttt{input:}[3,10], \texttt{output:}43\}$$

We were able to **model** the behavior of $\star$, on the examples we were given, with a simple algebraic equation. Let's try something rather more complicated. From the following three examples, can you figure out what the operator $\mathcal{F}$ does[2]:

Another name for this kind of learning is **fitting a model** to data.



You probably came up with something like "it fills in the missing pixels." That's exactly right, but it's sweeping a lot of details under the rug. Remember, we want to learn an *algorithm*, a procedure that is completely unambiguous. How exactly does $\mathcal{F}$ fill in the missing pixels?

It's hard to say in words. We may need a very complex algorithm to specify the answer, an algorithm so complex that we could never hope to write it out by hand. This is the point of machine learning. The machine writes the algorithm for us, but it can only do so if we give it many examples, not just these three.

[1] Not *exactly* anything. Some things are not learnable, such as non-computable functions.

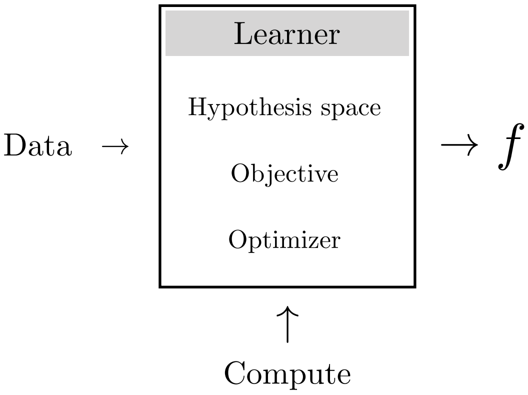
[2] This example is from [Hays and Efros2007]

11.2 Key ingredients

A learning algorithm consists of three key ingredients:

1. **Hypothesis space**: What is the set of possible mappings from inputs to outputs that we will we search over?
2. **Objective**: What does it mean for the learner to succeed, or, at least, to perform well?
3. **Optimizer**: *How*, exactly, do we search the model class for a specific mapping that maximizes the objective?

These three ingredients, when applied to large amounts of **data**, and run on sufficient hardware (**compute**) can do amazing things. We will focus on the learning algorithms in this chapter, but often the data and compute turn out to be more important.

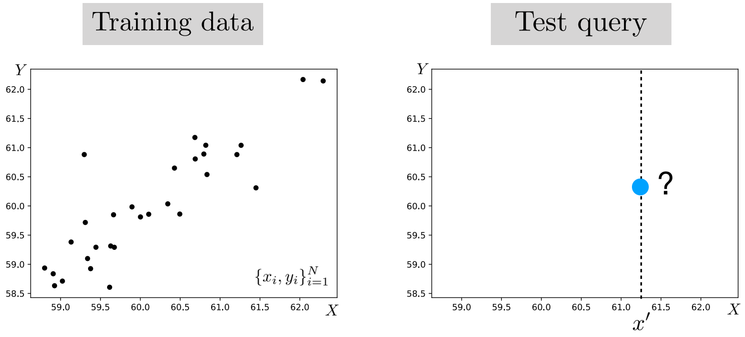


A learner outputs an algorithm, $f : X \to Y$, that maps inputs, $\mathbf{x} \in X$, to outputs, $\mathbf{y} \in Y$. Commonly, f is referred to as the learned "function".

11.2.1 Example: linear least squares regression

One of the simplest learning problems is known as "linear least squares regression". In this setting, we aim to model the relationship between two variables, x and y, with a line.

As a concrete example, let's imagine x represents the temperature on one day, and y represents the temperature on the following day. As before, we train (a.k.a. "fit") our model on many observed examples of {`temperature on day i`, `temperature on day i+1`} pairs, denoted as $\{x_i, y_i\}_{i=1}^N$. At test time, this model can be applied to predict the y value of a new query x':



Our *hypothesis space* is linear functions, i.e. the relationship between x and our predictions $\hat{y}$ of y has the form $\hat{y} = f_\theta(x) = \theta_1 x + \theta_0$. This hypothesis space is parametrized by a two scalars, $\theta_0, \theta_1 \in \mathbb{R}$, the intercept and slope of the line. We will use θ in general to refer to any parameters that are being learned. In this case we have $\theta = [\theta_0, \theta_1]$. Learning consists of finding the value of these parameters that maximizes the objective.

Our *objective* is that predictions should be near ground truth targets in a least squares sense, i.e. $(f_\theta(x_i) - y_i)^2$ should be small for all training examples $\{x_i, y_i\}_{i=1}^N$. $\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$ is referred to as the **loss function**, which is defined as the negative of the objective.

The full learning problem is:

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} (\theta_1 x_i + \theta_0 - y_i)^2. \tag{11.1}$$

We can choose any number of optimizers to solve this problem. A first idea might be "try a bunch of random values for θ and return the one that maximizes the objective." A better idea is to use math. From calculus, we know that at any maxima or minima of a function, $J(\theta)$, w.r.t. a variable θ, the derivative $\frac{\partial J(\theta)}{\partial \theta} = 0$. We are trying to find the minimum of the function:

$$J(\theta) = \sum_{i=1}^{N} (\theta_1 x_i + \theta_0 - y_i)^2. \tag{11.2}$$

This function can be rewritten as

$$J(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T (\mathbf{y} - \mathbf{X}\theta), \tag{11.3}$$

$$\mathbf{X} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \quad \theta = \begin{pmatrix} \theta_0 & \theta_1 \end{pmatrix}. \tag{11.4}$$

J is a **quadratic form**, which has a single global minimum where the derivative is zero, and no other points where the derivative is zero. Therefore, we can solve for the θ^* that minimizes J by finding the point where the derivative is zero. The derivative is:

$$\frac{\partial J(\theta)}{\partial \theta} = 2(\mathbf{X}^T \mathbf{X}\theta - \mathbf{X}^T \mathbf{y}). \tag{11.5}$$

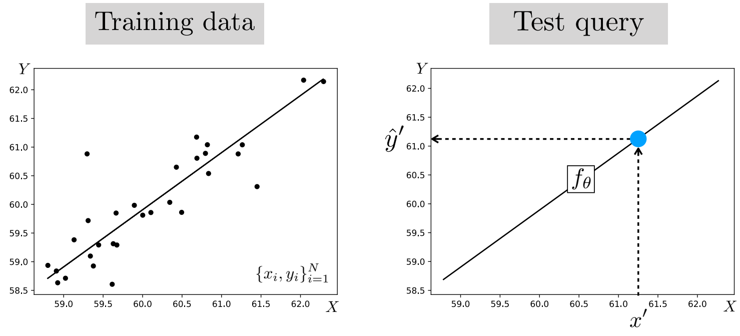
We set this to zero and solve for θ^*:

$$2(\mathbf{X}^T \mathbf{X}\theta^* - \mathbf{X}^T \mathbf{y}) = 0 \tag{11.6}$$

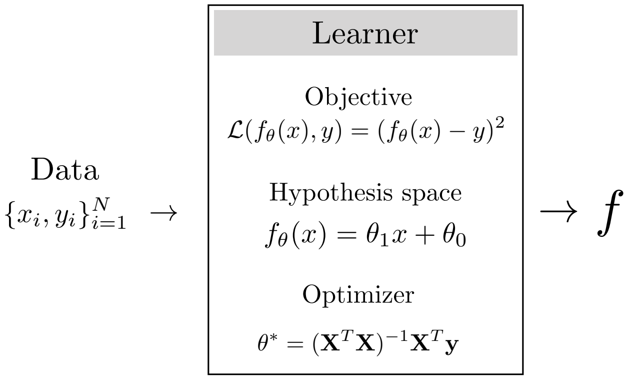
$$\mathbf{X}^T \mathbf{X}\theta^* = \mathbf{X}^T \mathbf{y} \tag{11.7}$$

$$\theta^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \tag{11.8}$$

θ^* defines the best fitting line to our data, and this line can be used to predict the y value of future observations of x:

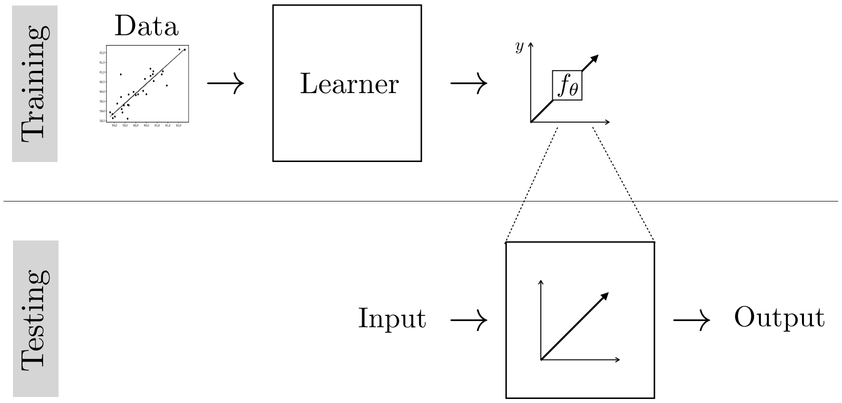


We can now summarize the entire linear least squares learning problem:


Data
$\{x_i, y_i\}_{i=1}^N$
Learner
Objective
$\mathcal{L}(f_\theta(x), y) = (f_\theta(x) - y)^2$
Hypothesis space
$f_\theta(x) = \theta_1 x + \theta_0$
Optimizer
$\theta^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$
f

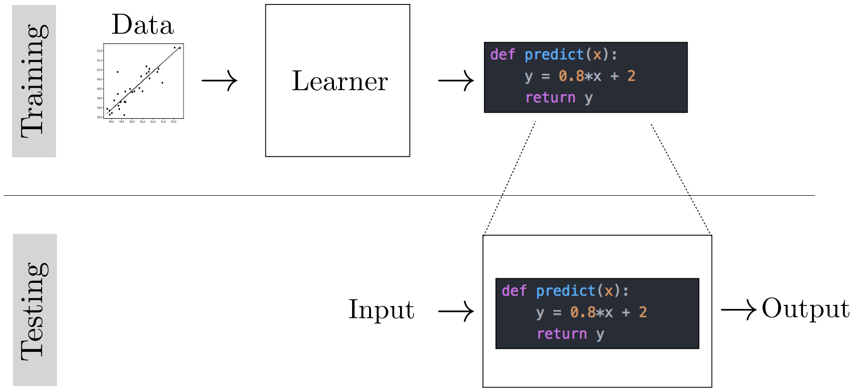
11.2.2 Example: program induction

At the other end of the spectrum we have what is known as "program induction", which is one of the broadest classes of learning algorithm. In this setting, our hypothesis space may be "all Python programs". Let's contrast linear least squares with Python program induction. Here's what linear least squares looks like:



The learned function is an algebraic expression that maps x to y. Learning consisting of searching over two scalar parameters, θ_0 and θ_1.

Here's Python program induction solving the same problem:



```
def predict(x):
    y = 0.8*x + 2
    return y
```

```
def predict(x):
    y = 0.8*x + 2
    return y
```

In this case, the learned function is a Python program that maps x to y. Learning consisted of searching over the space of all possible Python programs. Clearly that's a much harder search problem than just finding two scalars. In the next chapter, we will see some pitfalls of using too powerful a hypothesis space when a simpler one will do.

11.3 Empirical risk minimization

The three ingredients from the last section can be formalized using the framework of **empirical risk minimization** or **ERM**. This framework states the **supervised learning** problem as:

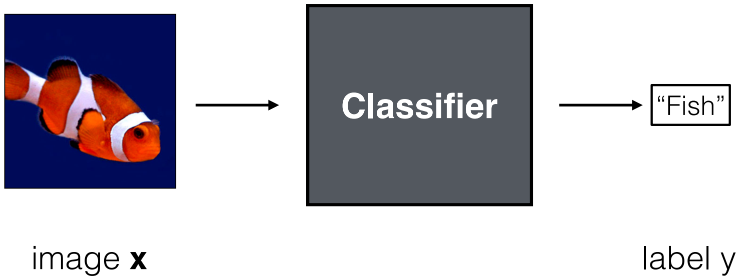
$$\arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}_i), \mathbf{y}_i), \quad \triangleleft \text{ ERM} \tag{11.9}$$

where $\mathcal{F}$ is the hypothesis space, $\mathcal{L}$ is the objective function (or loss), and $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{N}$ is the training data (example {`input`, `output`} pairs). f is the learned function.

Depending on the loss function, there is often an interpretation of ERM as doing maximum likelihood model fitting. For example, the least squares regression example we saw above can be interpreted as finding the maximum likelihood estimate of the parameters θ given the assumption that there truly exists a linear relationship between x and y but that training observations of y are corrupted by Gaussian noise (and we assume that training datapoints are independent and identically distributed, i.e. **iid**).

11.4 Case study #1: image classification

A common problem in computer vision is to recognize objects. This is a **classification** problem. Our input is an image $\mathbf{x}$, and our target output is a class label $\mathbf{y}$:



How should we formulate this task as a learning problem? The first question is: how do we even represent the input and output? Representing images is pretty straightforward, they are just arrays of numbers representing RGB colors: $\mathbf{x} \in \mathbb{R}^{H \times W \times 3}$, where H is image height and W is image width.

How can we represent class labels? It turns out a convenient representation is to let $\mathbf{y}$ be a K-dimensional vector, for K possible classes, with $y_k = 1$ if $\mathbf{y}$ represents class k, and 0 otherwise. So the learning problem is to map $f : \mathbb{R}^{H \times W \times 3} \rightarrow \mathbb{R}^{K}$. We will see why this representation makes sense shortly.

Next, we need to pick a loss function. Our first idea might be that we should minimize misclassifications. That would correspond to the so called **0-1 loss**:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = 1(\hat{\mathbf{y}} = \mathbf{y}). \tag{11.10}$$

Unfortunately, optimizing this loss is a discrete optimization problem, and it is NP-hard. Instead, people commonly use the **cross entropy loss**, which is continuous and differentiable (making it easier to optimize):

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k. \tag{11.11}$$

The way to think about this is $\hat{y}_k$ represents the probability we think the image is an image of class k. Under that interpretation, minimizing cross entropy maximizes the log likelihood of the ground truth observation $\mathbf{y}$ under our model's prediction $\hat{\mathbf{y}}$.

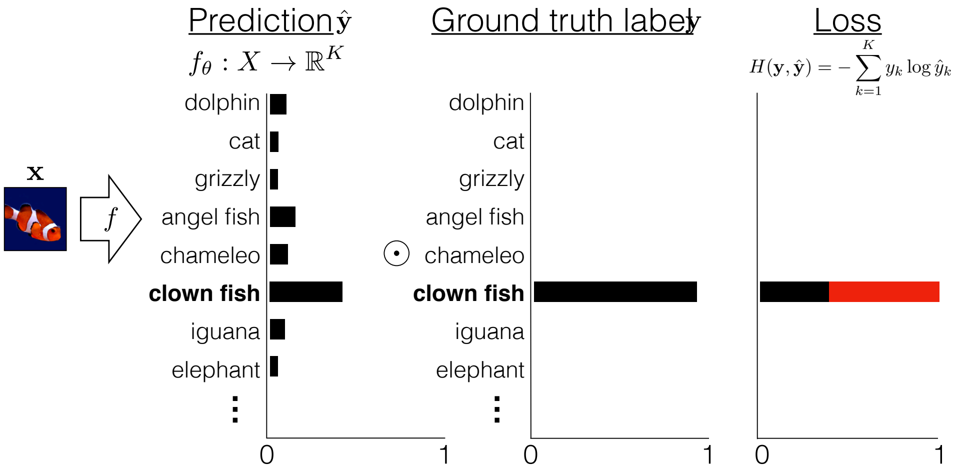
For that interpretation to be valid, we require that $\hat{\mathbf{y}}$ represents a probability mass function (**pmf**). A pmf is just a vector with elements in the range $[0, 1]$ that sums to 1. So, if our learned function outputs a vector $f_\theta(\mathbf{x}) \in \mathbb{R}^K$, then we can convert it to a pmf by "squashing" it into the range $[0, 1]$ and normalizing it to sum to 1. A popular way to squash is via the **softmax** function (this is a modeling choice, we could have used any function that squashes into a valid pmf, i.e. a nonnegative vector that sums to 1):

$$\mathbf{z} = f_\theta(\mathbf{x}) \tag{11.12}$$

$$\hat{\mathbf{y}} = \texttt{softmax}(\mathbf{z}) \tag{11.13}$$

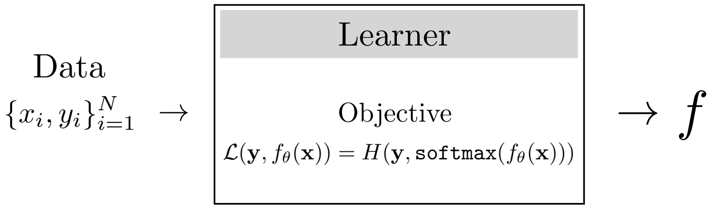
$$\hat{y}_j = \frac{e^{-z_j}}{\sum_{i=1}^{K} e^{-z_k}}. \tag{11.14}$$

Here's what the variables look like for processing one photo of a fish during training:



The prediction placed about 40% probability on the true class, "clown fish", so we are 60% off from an ideal prediction (indicated by the red bar; an ideal prediction would place 100% probability on the true class). Our loss is log −0.4.

The learning problem can be summarized as follows:


Data
$\{x_i, y_i\}_{i=1}^N$
$\rightarrow$
Learner
Objective
$\mathcal{L}(\mathbf{y}, f_\theta(\mathbf{x})) = H(\mathbf{y}, \texttt{softmax}(f_\theta(\mathbf{x})))$
$\rightarrow f$

Note that we have left the hypothesis space and optimizer unspecified. This is one of the reasons we conceptualized the learning problem into the three key ingredients described above: you can often develop them each in isolation, then mix and match.

11.5 Learning without examples

Even without examples, we can still learn. Instead, we can try to come up with an algorithm that optimizes for desirable *properties* of the input-output mapping, rather than coming up with a mapping that matches examples. This class of learners includes **unsupervised learning** and **reinforcement learning**.

In reinforcement learning, we suppose that we are given a **reward function** that explicitly measures the quality of the learned function's output. To be precise, a reward function is a mapping from outputs to scores: $r : Y \to \mathbb{R}$. The learner tries to come up with a function that maximizes rewards. Reinforcement learning also usually deals with the setting where the learned function gets to pick which training datapoints to observe next, rather than being provided them passively. An example is an agent that chooses which actions to take as it navigates a room, and this choice of action determines what data (view of the world) the agent will next observe. This dependence of data on learned function is one of the main reasons reinforcement learning is hard.

11.6 Learning to learn [Advanced topic]

"Learning to learn", also called **meta-learning**, is a special case of learning where the hypothesis space is learning algorithms.

Recall that learners train on past instances of a problem to produce an algorithm that can solve future instances of the problem. Suppose the problem is itself a simple learning problem: "find the least squares line fit to these data points." One way to train for this would be by example. Suppose we are given {`input`, `output`} exemplars:

$$\begin{aligned}
&\{\texttt{input:}\big(x:[1,2], y:[1,2]\big), &&\texttt{output:}\, y = x\}\\
&\{\texttt{input:}\big(x:[1,2], y:[2,4]\big), &&\texttt{output:}\, y = 2x\}\\
&\{\texttt{input:}\big(x:[1,2], y:[0.5,1]\big), &&\texttt{output:}\, y = \frac{x}{2}\}
\end{aligned}$$

These are examples of performing least squares regression. The learner should therefore learn to perform least squares regression. Since least squares regression is itself a learning algorithm, we can say that the learner learned to learn.

Notice that you can apply this idea recursively, constructing meta-meta-...-meta-learners. Humans perform at least three levels of this process, if not more: we have *evolved* to be *taught* in school how to *learn* quickly on our own.

Note that **evolution** is a learning algorithm, according to our present definition.

Chapter 12

The problem of generalization

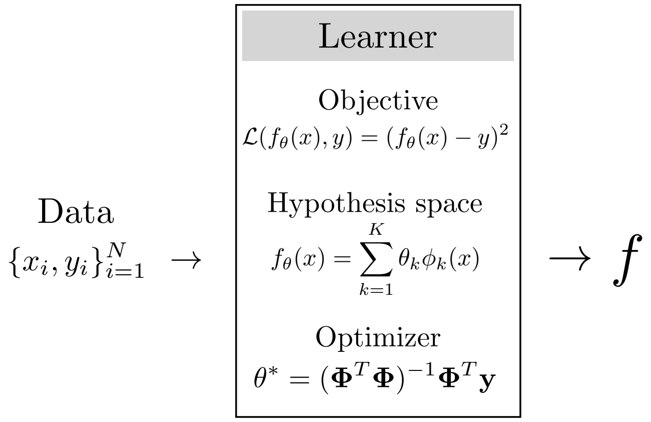
So far, we have described learning as an optimization problem: maximize an objective over the *training set*. But this is not our actual goal. Our goal is to maximize the objective over the *test set*. This is the key difference between learning and optimization. We do not have access to the test set, so we use optimization on the training set as a proxy for optimization on the test set.

Learning theory studies the settings under which optimization on the training set yields good results on the test set. In general it may not, since the test set may have different properties from the training set. When we fit to properties in the training data that do not exist in the test data we call this **overfitting**. When this happens, training performance will be high, but test performance can be very low, since what we learned about the training data does not **generalize** to the test data.

12.1 Underfitting and overfitting

A learner may perform poorly for one of two reasons: either it failed to optimize the objective on the training data, or it succeeded on the training data but in a way that does not generalize to the test setting. The former is called **underfitting** and we measure it with **approximation error**. The latter is called **overfitting** and we measure it with **generalization error**.

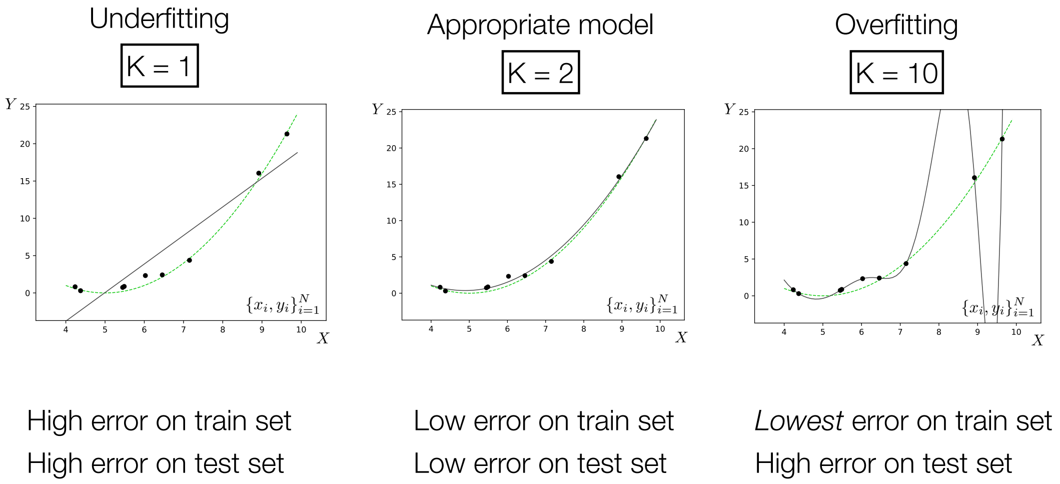
We can observer these two effects in least-squares polynomial regression:


Learner
Objective
$\mathcal{L}(f_\theta(x), y) = (f_\theta(x) - y)^2$
Hypothesis space
$f_\theta(x) = \sum_{k=1}^{K} \theta_k \phi_k(x)$
Optimizer
$\theta^* = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{y}$
Data
$\{x_i, y_i\}_{i=1}^N$ →
→ f

Notice that polynomial regression is the just linear regression over a particular set of basis functions $\mathbf{\Phi}$:

$$\mathbf{\Phi} = \begin{pmatrix} 1 & x_1 & x_1^2 & ... & x_1^K \\ 1 & x_2 & x_2^2 & ... & x_2^K \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & ... & x_N^K \end{pmatrix} \tag{12.1}$$

What happens as we increase the order of the polynomial K, i.e. we use $K + 1$ basis functions $x^0, \cdots, x^K$:



The black line is the model's fit. The green line is the true function that generated the data, i.e. the **data-generating process** was the green line plus Gaussian noise.

As we increase K we fit the data points better and better, but eventually start *overfitting*, where the model perfectly interpolates the data but deviates more and more from the true data-generating line.

Approximation error is the gap between the black line and the data points. Generalization error is the gap between the black line and the green line. Approximiation error goes down with increasing K but all we really care about is generalization error, which measures how well we will do on test queries that are newly sampled from the true data-generating process. Generalization error typically has a U-shaped function with respect to K: at first it is high because we are underfitting; gradually we fit the data better and better and eventually we overfit, with generalization error becoming high again.

12.2 Regularization

The above example suggests a kind of "Goldilocks principle". We should prefer hypotheses (functions f) that are sufficiently expressive to fit the data, but not so expressive that they can overfit the data.

Regularization refers to mechanisms for reducing **model capacity** so that we avoid overfitting. Model capacity measures the expressivity of the hypothesis space. Typically, regularizers are terms we add to the objective that prefer simple functions in the hypothesis space, all else being equal. They therefore embody the principle of **Occam's razor**.

The L_p norm of $\mathbf{x}$ is $(\sum_i |x_i|^p)^{\frac{1}{p}}$. The L_2 norm is the familiar least-squares objective.

One of the most common regularizers is to penalize the L_p norm of the parameters of our model, θ:

$$R(\theta) = \|\theta\|_p . \tag{12.2}$$

An especially common choice is $p = 2$, in which case the regularizer is called **Tikonov regularization** (a.k.a. **ridge regression**). In the context of neural nets, this regularizer is called **weight decay**. When $p = 1$, the regularizer, applied to regression problems, is called

LASSO regression. For any p, the effect is to encourage most parameters to be zero, or near zero. When most parameters are zero, the function takes on a degenerate form, i.e. a simpler form. For example, if we consider the quadractic hypothesis space $\theta_1 x + \theta_2 x^2$, then, if we use a strong L_p regularizer, and if a linear fit is almost perfect, then θ_2 will be forced to zero and the learned function will be linear, rather than quadratic. Again, we find that regularization is an embodiment of Occam's razor: when multiple functions can explain the data, prefer the simplest.

12.3 Regularizers as priors and the Bayesian Occam's razor [Advanced topic]

Most regularizers can be given probabilistic interpretations as priors on the hypothesis, whereas the original objective, e.g., least-squares, measures likelihood of the data given the hypothesis. These priors are not arbitrarily chosen. The notion of the **Bayesian Occam's razor** derives such priors by noting that more complex hypothesis spaces must cover more possible hypotheses, and therefore must assign less prior mass to any single hypothesis (the prior probability of all possible hypotheses in the hypothesis space must sum to one). This is why, probabilistically, simpler hypotheses are more likely to be true.

12.4 Rethinking generalization [Advanced topic]

A recent empirical finding is that deep nets tend not to overfit, even though they have many more "free" parameters than the number of datapoints they are fit to. Exactly why this happens is an ongoing topic of research [Zhang et al.2016]. But we should not be too surprised. Number of free parameters is just a rough proxy for model capacity (i.e., "the expressivity of hypothesis space"). A single parameter that has infinite numerical precision can parameterize an arbitrarily complexity function. Such a parameter defines a very expressive hypothesis space and will be capable of overfitting data. On the other hand, if we have a million parameters, but they are regularized so that almost all are zero, then these parameters may end up defining a simple class of functions, which does not overfit the data. So you can think of number of parameters as a rough estimate of model capacity, but it is not the full story. See [Belkin et al.2018] for more discussion on this point.

Bibliography

[Belkin et al.2018] Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2018. Reconciling modern machine learning and the bias-variance trade-off. *arXiv preprint arXiv:1812.11118.*

[Hays and Efros2007] Hays, James, and Alexei A Efros. 2007. Scene completion using millions of photographs. *ACM Transactions on Graphics (TOG)* 26 (3): 4.

[Zhang et al.2016] Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2016. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530.*