Section 4: How AI Learns From Data

4.1 What is Learning?

Recall that AI models are nothing more than very sophisticated functions with three critical components: a numerical input (encoding data such as image or text), a numerical output (encoding a label or response), and a collection of parameters that determine how the input is transformed into the output.

These parameters must ensure that the system outputs match (as much as possible) their corresponding labels across all training data. **Learning** is the process of adjusting the parameters to achieve this goal.

A Concrete Example

Imagine an AI system that takes as input a vector $\mathbf{x}$ with two variables representing a person's height and weight:

$$\mathbf{x} = \begin{bmatrix} \mathrm{x}_1 \\ \mathrm{x}_2 \end{bmatrix} \begin{array}{l} \leftarrow \text{height} \\ \leftarrow \text{weight.} \end{array}$$

Suppose the system produces an output using the function

$$f(\mathbf{x}) = \mathrm{w}_1\mathrm{x}_1 + \mathrm{w}_2\mathrm{x}_2,$$

where w_1 and w_2 are adjustable parameters that control how height and weight influence the output.

We want this system to predict diabetes risk as a function of height and weight. We thus provide it example data consisting of height–weight measurements for four individuals:

$$\mathbf{x}_1 = \begin{bmatrix} 177 \\ 150 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 150 \\ 170 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 160 \\ 140 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 165 \\ 120 \end{bmatrix},$$

along with their corresponding known risk values:

$$\mathrm{y}_1 = 1/4, \qquad \mathrm{y}_2 = 5/6, \qquad \mathrm{y}_3 = 1/8, \qquad \mathrm{y}_4 = 1/9.$$

The goal of learning is to choose the parameters w_1 and w_2 so that the function $f(\mathbf{x})$ produces outputs that match these known risk values as closely as possible for all individuals at the same time.

4.2 The Key Ingredients

For the rest of this course we will use $\mathbf{x}$ to denote **input** to the AI system. Recall that $\mathbf{x}$ is simply a vector containing numbers that may take any value. These numbers encode the input data, which in turn may be an image, text, sound, or any other type of data.

Since an AI system is just a function, we will use $f(\mathbf{x})$ do denote the **output** of the system. This function can take many forms. It may be as simple as in the example above, or as complex as modern AI models. For our purposes, this distinction does not matter. All we need to know is that when an input $\mathbf{x}$ is provided to the system, it produces an output $f(\mathbf{x})$. Like a vending machine. At this stage, we are not concerned with how this output is computed internally, only with the relationship between inputs and outputs.

We will use $\mathbf{W}$ to denote the **parameters** of the function, often called *weights*. These parameters control how the input is transformed into the output internally. Depending on the complexity of the model, $\mathbf{W}$ may consist of a single number, a vector, a matrix, or even a collection of matrices. Regardless of their form, all of these parameters play the same role: they determine the behavior of the function and, therefore, the behavior of the AI system.

Recall that the parameters must be adjusted to make the system produce the desired outputs. This is achieved through **training data**, that is, a collection of known input-output pairs $(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})$. Here $\mathbf{x}_1, \dots, \mathbf{x}_\mathrm{N}$ denote the inputs, each one a numerical encoding of a piece of data; $\mathbf{y}_1, \dots, \mathbf{y}_\mathrm{N}$ denote the corresponding *known* outputs, also known as **response**. Recall that each response $\mathbf{y}_\mathrm{i}$ may consist of a single number (label) or a vector encoding a more sophisticated response, such as an image. This is the case, for example, when you ask a system like ChatGPT to generate an image.

At this point we have all the ingredients we need to understand how AI systems learn.

4.3 The Training Process

The process of feeding data to the AI so that it can adjust its parameters is often called **training**, and can be seen as an ongoing dialogue between a human and the machine:

Human: Here is an input $\mathbf{x}_\mathrm{i}$ (e.g., an image of a dog). What is your output?

Machine: My output is $f(\mathbf{x}_\mathrm{i})$ (e.g., '1', corresponding to the label of a cat).

Human: The correct output is $\mathbf{y}_\mathrm{i}$ (e.g., '2', corresponding to the label of a dog). Adjust your parameters so that $f(\mathbf{x}_\mathrm{i})$ is closer to $\mathbf{y}_\mathrm{i}$.

This exchange is repeated until the machine adjusts its parameters sufficiently well so that its output $f(\mathbf{x}_\mathrm{i})$ becomes *close enough* to the response $\mathbf{y}_\mathrm{i}$ for every training pair.

4.4 The Loss Function

To determine when the system's output $f(\mathbf{x}_\mathrm{i})$ is *close enough* to the response $\mathbf{y}_\mathrm{i}$ we use what is commonly known as **loss function**. The loss is a function of the parameters, denoted by $\ell(\mathbf{W})$, and it measures how far $f(\mathbf{x}_\mathrm{i})$ is from $\mathbf{y}_\mathrm{i}$. In other words, the loss quantifies the error the system makes on the training data. Learning then consists of adjusting the parameters to reduce this loss.

There are many ways to quantify such error. Common examples include:

- **Mean Squared Error.** It measures the square of the difference between the output and the response:

$$\ell(\mathbf{W}) = \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \| f(\mathbf{x}_\mathrm{i}) - \mathbf{y}_\mathrm{i} \|_2^2.$$

Recall that given two vectors $f(\mathbf{x})$ and $\mathbf{y}$, their 2-norm distance is given by the squared root of the sum of squared differences of their entries:

$$\|f(\mathbf{x}) - \mathbf{y}\|_2^2 \ = \ \sqrt{\sum_{\mathrm{j}} \left(f_{\mathrm{j}}(\mathbf{x}) - \mathrm{y}_{\mathrm{j}}\right)^2}.$$

- **Absolute Error.** It measures the absolute difference between the output and the response:

  $$\ell(\mathbf{W}) = \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \|f(\mathbf{x}_{\mathrm{i}}) - \mathbf{y}_{\mathrm{i}}\|_1,$$

  where similarly:

  $$\|f(\mathbf{x}) - \mathbf{y}\|_1 \ = \ \sum_{\mathrm{j}} \left|f_{\mathrm{j}}(\mathbf{x}) - \mathrm{y}_{\mathrm{j}}\right|.$$

  This loss is sometimes easier to interpret, as it corresponds directly to how far off the prediction is.

- **Cross-entropy.** It is a classification loss used when the response is a category or label. Omitting some technical details, the cross-entropy for two categories looks like:

  $$\ell(\mathbf{W}) = -\frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \Big(\mathrm{y}_{\mathrm{i}} \log f(\mathbf{x}_{\mathrm{i}}) + (1 - \mathrm{y}_{\mathrm{i}}) \log\left(1 - f(\mathbf{x}_{\mathrm{i}})\right)\Big).$$

  It penalizes confident wrong predictions more heavily than uncertain ones, encouraging the system not only to be correct but also to be appropriately confident.

Some loss functions are better suited to certain tasks than others, because they emphasize different features or patterns in the data. The choice of loss function therefore depends on what we care about and what kind of behavior we want the system to learn.

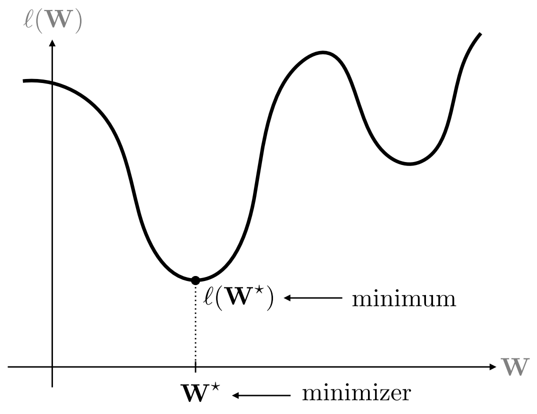
4.5 Minimizing the Loss

Once we have the training data (pairs $(\mathbf{x}_{\mathrm{i}}, \mathbf{y}_{\mathrm{i}})$) and specified the AI model (function f with parameters $\mathbf{W}$) and the loss function $\ell(\mathbf{W})$, all that remains is to adapt the parameters $\mathbf{W}$ so that the model's output $f(\mathbf{x}_{\mathrm{i}})$ differs with $\mathbf{y}_{\mathrm{i}}$ as little as possible. From a mathematical perspective, this simply means finding the parameter values that minimize the loss:

$$\min_{\mathbf{W}} \ \ell(\mathbf{W}).$$

This is exactly the kind of task studied in the field of **optimization**. In its simplest form, optimization asks the following question: given a function $\ell(\mathbf{W})$, how can we find the parameter values $\mathbf{W}^\star$ that make this function as small as possible? In other words, optimization aims to find the *optimal* parameter values $\mathbf{W}^\star$ that satisfy

$$\ell(\mathbf{W}^\star) \ \leq \ \ell(\mathbf{W}) \qquad \text{for any choice of } \mathbf{W}.$$



The Easy Case

If ℓ is *convex* (u-shaped) and *simple* enough, $\mathbf{W}^\star$ can be determined using our elemental calculus recipe:

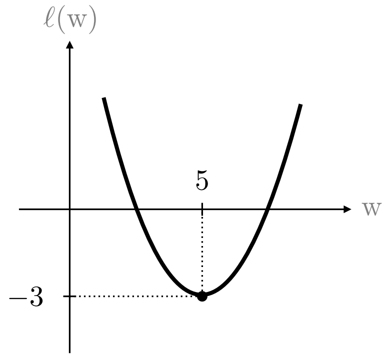
1. Take derivative of $\ell(\mathbf{W})$
2. Set the derivative to zero, and solve for the minimizer.

For example, consider the scalar function $\ell(\mathrm{w}) = (\mathrm{w}-5)^2-3$. We can follow our recipe to find its minimizer:

1. The derivative of ℓ is given by $\ell'(\mathrm{w}) = 2(\mathrm{w}-5)$.
2. Setting the derivative to zero and solving for w we obtain:

$$\begin{aligned} 2(\mathrm{w}-5) &= 0 \\ \mathrm{w} &= 5. \end{aligned}$$

Since ℓ is concave, we conclude that its minimizer is $\mathrm{w}^\star = 5$, and its minimum is $\ell(\mathrm{w}^\star) = -3$:



Matrix Derivatives

In general, ℓ will not always be a function as simple as in our previous example. In fact, in most AI tasks, ℓ will be a complex multivariate function in matrix form, for example:

$$\ell(\mathbf{W}) = (\mathbf{y} - \mathbf{W}\mathbf{x})^\mathsf{T}(\mathbf{y} - \mathbf{W}\mathbf{x}),$$

If we want to optimize $\ell(\mathbf{W})$, we need to take the derivative with respect to the matrix $\mathbf{W}$.

To learn more about how to take derivatives w.r.t. vectors and matrices I recommend taking a look at *Old and new matrix algebra useful for statistics* by Thomas P. Minka, which shows how to take derivatives of some matrix functions.

Gradient Descent

Some functions are too complex to solve for $\mathbf{W}$ in step 2. For example, consider the following function that describes the likelihood of i.i.d. Bernoulli samples:

$$\ell(\mathbf{w}) = \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{y_i} \log\left(\frac{1}{1+e^{-\mathbf{w}^\mathsf{T}\mathbf{x}_\mathrm{i}}}\right) + (1-\mathrm{y_i})\log\left(1 - \frac{1}{1+e^{-\mathbf{w}^\mathsf{T}\mathbf{x}_\mathrm{i}}}\right).$$

Its gradient is given by:

$$\ell'(\mathbf{w}) = \sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\mathrm{y_i} - \frac{1}{1+e^{-\mathbf{w}^\mathsf{T}\mathbf{x}_\mathrm{i}}}\right)\mathbf{x}_\mathrm{i}.$$

If we set this to zero, nobody knows how to solve for $\mathbf{w}$.

For cases where our basic calculus recipe does not work, we will use one of the most elemental tools of optimization: **gradient descent**.

The main idea is this: we have a function $\ell(\mathrm{w})$. We want to find its minimum. We cannot solve for it directly using the derivative trick. However, we can *test* the value of ℓ for different values of w. For example, we can check what is $\ell(0)$, then maybe $\ell(1)$, then maybe $\ell(-1)$, then maybe $\ell(1.5)$, and so on, until we find the minimizer. Of course, depending on the domain of ℓ, there could be infinitely many options, so testing them all would be infeasible.

As the name suggests, the main idea of gradient ascent is to test some initial value w_0 (for example 0), and iteratively use the gradient (another name for derivative) to determine which value of w to test next, such that each new value w produces a higher value for ℓ, until we find the minimum. The main intuition is that the gradient $\ell'(\mathrm{w})$ tells us the slope of ℓ at w. If this slope is positive, then we know that ℓ is increasing, and we should try a lower value of w, say $\mathrm{w_{t+1}} = \mathrm{w_t} - \eta$, where η is often referred to as *step-size*. If the slope is negative, then we know that ℓ is decreasing, and we should try a larger value of w, say $\mathrm{w_{t+1}} = \mathrm{w_t} + \eta$ (see Figure 4.1 to build some intuition).

The same insight extends to multivariable functions. If ℓ is a function of a matrix $\mathbf{W}$, then $\ell'(\mathbf{W})$ gives the slope of ℓ in each of the entries of $\mathbf{W}$. In general, gradient descent can be summarized as follows:

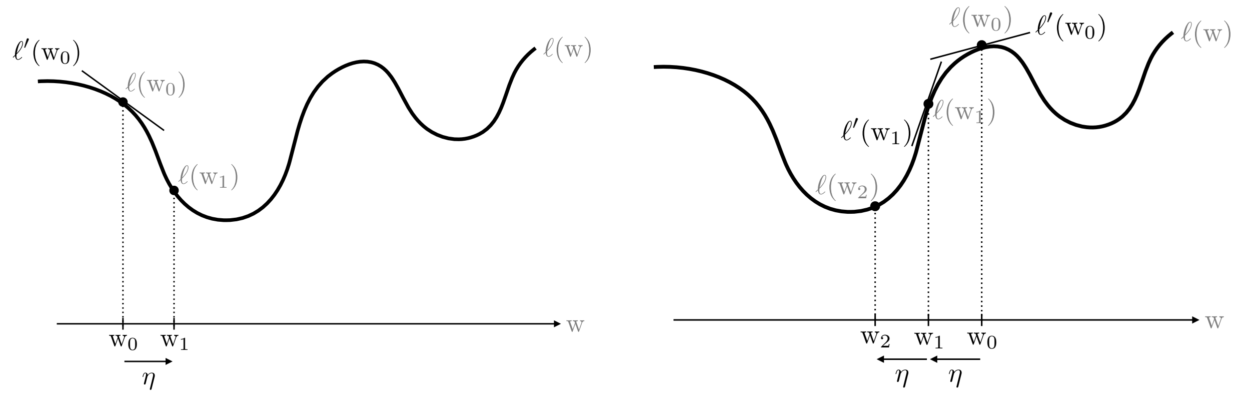


Figure 4.1: Start at some point w_0. If the gradient is positive (left figure), try a larger value of w, say $w_1 = w_0 + \eta$. If the gradient is negative (right figure), try a smaller value of w, say $w_1 = w_0 + \eta$. Repeat this until convergence.

Algorithm 1: Gradient Descent

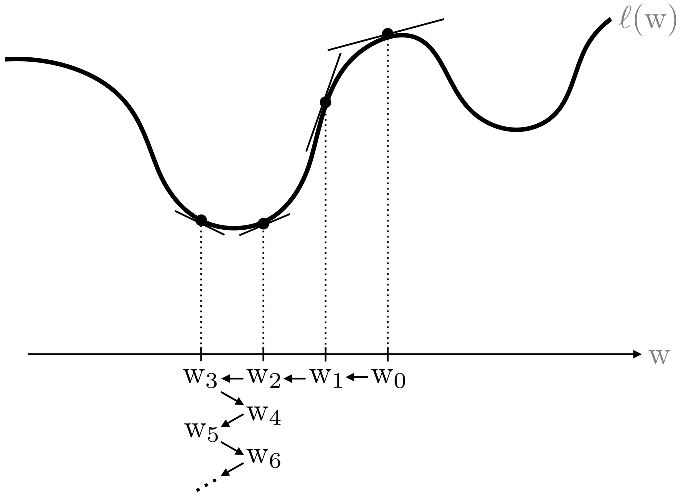
Input: Function ℓ, step-size parameter $\eta > 0$.
Initialize $\mathbf{W}_0$. For example, $\mathbf{W}_0 = \mathbf{0}$.
Repeat until convergence: $\mathbf{W}_{t+1} \; = \; \mathbf{W}_t - \eta \; \ell'(\mathbf{W}_t)$.
Output: $\mathbf{W}^\star = \mathbf{W}_t$.

Hyperparameters: the Step-size η

The keen reader will be wondering what happens if we move too far. In our example of Figure 4.1, we could run into an infinite loop, where

$$\begin{aligned} w_2 &= w_4 = w_6 = w_8 = \cdots \\ w_3 &= w_5 = w_7 = w_9 = \cdots, \end{aligned}$$

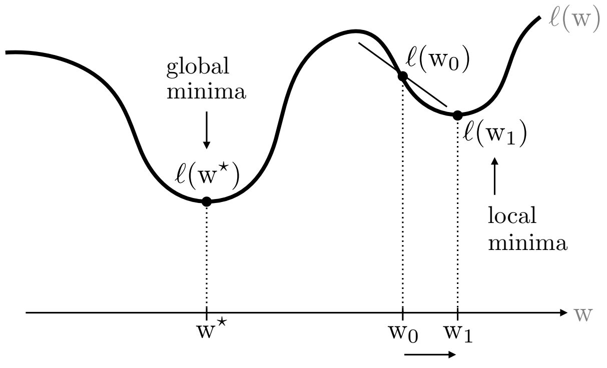
without ever achieving $w^\star$, as depicted below:



How would you solve this?

Initialization

The keen reader will also be wondering what happens if we start at the wrong place, as depicted below:



In cases like these we could run into a so-called *local minima*, that is, a point that is smaller than all other points in its vicinity, but not necessarily the minimum over the whole range of ℓ. In the figure above, w_1 is a local minimizer.

How would you solve this?

Maximization

How would things change if you wanted to maximize, rather than minimize?

Maximization

How would things change if you wanted to maximize, rather than minimize?

4.6 Training in Practice

Luckily for us, there already exist numerous software packages that do all the training for us, such as PyTorch or TensorFlow. All we need to do is:

- Provide the training data.
- Specify the model, that is, the function f that we want to use.
- Specify the loss function.
- Specify the optimization method (e.g., stochastic gradient descent) and its parameters (e.g., step size).

Naturally, there are several technical details that we need to worry about, such as cleaning normalizing, and splitting the data into training, validation, and test sets. We will discuss each of these topics in the upcoming sections.