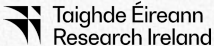
# Introduction to Machine Learning and Neural Networks

Tobias Rossmann

Ollscoil na Gaillimhe
University of Galway

MathLearn — June 2026




# Propaganda & disclaimer

- We're a group of pure mathematicians with an interest in machine learning. We don't do research in numerical analysis, statistics, or machine learning.
- We have been experimenting with and using machine learning for some time. We're happy to share what we learned.
- While there's a vast body of literature on the subject, this mathematician struggles with most texts on mathematical topics that weren't written by mathematicians. :)

> *"Mathematicians are like Frenchmen: whatever you say to them they translate into their own language and forthwith it is something entirely different."*
>
> — *Johann Wolfgang von Goethe*

# MathLearn material

For the timetable, lecture notes, Jupyter notebooks, please visit
https://math-learn.github.io/

# Goals of this lecture

- **Demystify machine learning (ML):**
  ML is essentially numerical optimisation over parameterised families of functions.
- **A mathematician's dictionary:**
  Decoding ML buzzwords, e.g. backpropagation, features, hyperparameters, learning rate, loss functions, etc. by expressing them in mathematical terms.
- **Introduction to neural networks:**
  Neural networks are diagrams of affine functions followed by so-called activation functions.

# Categorising machine learning

Machine learning can be categorised along two axes.

## Model architecture

These typically use some form of artificial neural networks. Important subclasses:

- Linear models
- Multi-Layer Perceptrons (MLPs)
- Convolutional Neural Networks (CNNs)
- Graph Neural Networks (GNNs)
- Transformers

## Learning paradigm

- Supervised learning: find a function that approximates pairs of data points
- Unsupervised learning: find patterns, dimensionality reduction
- Reinforcement learning: train an agent to interact with a dynamic environment in a way that maximises a reward

We'll start at the beginning: the "perceptron" (Rosenblatt, 1958).

# The perceptron

- Fix a dimension $d$. A **perceptron** is a function

$$f_{w,b}\colon \mathbf{R}^d \to \mathbf{R}, \quad x \mapsto H(w \cdot x + b),$$

where $w \in \mathbf{R}^d$ is a vector of **weights**, $b \in \mathbf{R}$ is the **bias**, and

$$H(z) = \begin{cases} 1, & \text{if } z \geqslant 0, \\ 0, & \text{otherwise.} \end{cases}$$

is the **Heaviside step function**.
That is, a perceptron is an affine function followed by a particular (non-linear and discontinuous) **activation function**.

# The perceptron

- The collection of all perceptrons (for fixed $\mathrm{d}$) is a **model (class)**.
- The weights and bias together form the **parameters** of the model.
  Think: the weights and bias are knobs that the machine can turn to achieve some goal
- The dimension $\mathrm{d}$ is an example of a **hyperparameter**: values that affect the model design and learning process, but which are not modified during the learning process.

# The perceptron

- Let $(x_1, y_1), \ldots, (x_N, y_N)$ with $x_i \in \mathbf{R}^d$ and $y_i \in \{0, 1\}$.
  The points $x_1, \ldots, x_N$ are divided into two classes ($y_i = 0$ and $y_i = 1$).
  The $x_i$ are our **training examples**. We call $y_i$ the **target** or **label** of $x_i$.
- For $x \in \mathbf{R}^d$, each choice of $w$ and $b$ produces a **predicted class** $f_{w,b}(x)$ of $x$.

**Question**

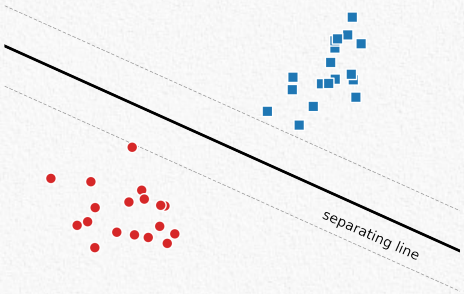
Can we find $(w, b)$ with $f_{w,b}(x_i) = y_i$ for $i = 1, \ldots, N$?

**Exercise**

This is possible if and only if the two classes of points are **linearly separable**, i.e. if and only if they lie on opposite sides of an affine hyperplane.

**Question**

When the classes are linearly separable, how can we find $(w, b)$ such that our perceptron correctly predicts classes?



# The perceptron rule

We choose another hyperparameter: the **learning rate** $\eta > 0$. We then **train** our model by adjusting model parameters as follows:

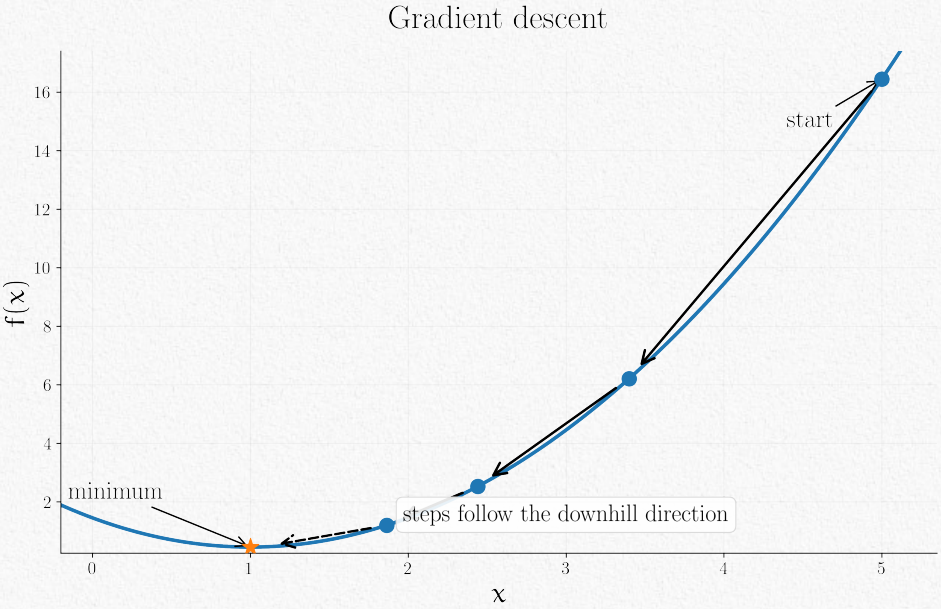
- Initialise $w$ and $b$ to be some (small) random numbers.
- Repeatedly update the parameters $(w, b)$ using the following **learning rule**:
  - For each training example $x_i$, compute the predicted class $\hat{y}_i := f_{w,b}(x_i)$.
  - If $\hat{y}_i = y_i$ for $i = 1, \ldots, N$, then we stop.
  - Otherwise, we update $w \leftarrow w + \eta \sum_{i=1}^{N} (y_i - \hat{y}_i) x_i$ and $b \leftarrow b + \eta \sum_{i=1}^{N} (y_i - \hat{y}_i)$.

**Theorem**

*If the two classes of points are linearly separable, then the above algorithm will, after finitely many steps, construct a perceptron which correctly recognises the two classes.*

**Question**

Where does this learning rule come from?


Gradient descent
f(x)
x
start
minimum
steps follow the downhill direction
16
14
12
10
8
6
4
2
0
1
2
3
4
5

# Reminder: gradient descent

- Let $f\colon \mathbf{R}^d \to \mathbf{R}$ be (sufficiently) smooth. Recall that the **gradient** of $f$ at $x \in \mathbf{R}^d$ is

$$\nabla f(x) = \left( \frac{\partial f(x)}{\partial x_1}, \dots \right).$$

- Locally $\nabla f(x)$ is the direction of steepest ascent and $-\nabla f(x)$ is the direction of steepest descent.
- We can try to approximate a local minimum by repeated application of the rule

$$x \leftarrow x - \eta \nabla f(x)$$

  for $\eta > 0$.
- The role of $\eta$ is critical! If it's too small, then it may take very long for us to get close to a local optimum. On the other hand, if $\eta$ is too big, we might jump around.

# Deriving the perceptron rule via gradient descent: idea

- Write $g_{w,b}(x) = w \cdot x + b$. Hence, $f_{w,b}(x) = H(g_{w,b}(x))$.
- We can't (or shouldn't) derive the perceptron learning rule through a literal application of gradient descent: the Heaviside function is discontinuous at zero, with zero derivative away from zero!
- What we *can* do is consider the (squared) Euclidean error on the raw affine output $g_{w,b}(x)$ and apply gradient descent to this. This provides a heuristic bridge between gradient descent and the perceptron rule.

skip

# Deriving the perceptron rule via gradient descent: details

- Write $\tilde{y}_i = g_{w,b}(x_i)$. We'll use this as a proxy for $\hat{y} = H(\tilde{y})$.
- Write

$$L(w, b) := \sum_{i=1}^{N} (y_i - \tilde{y}_i)^2.$$

Then $\frac{\partial L(w,b)}{\partial w_j} = -2 \sum_{i=1}^{N} (y_i - \tilde{y}_i)(x_i)_j$ and $\frac{\partial L(w,b)}{\partial b} = -2 \sum_{i=1}^{N} (y_i - \tilde{y}_i)$.

# Deriving the perceptron rule via gradient descent: details

- Hence, repeated gradient descent along all training examples gives the rule

$$w \leftarrow w + \eta \sum_{i=1}^{N} (y_i - \tilde{y}_i) x_i, \quad b \leftarrow b + \eta \sum_{i=1}^{N} (y_i - \tilde{y}_i).$$

  We're off by a factor of 2, but this can be absorbed by $\eta$.

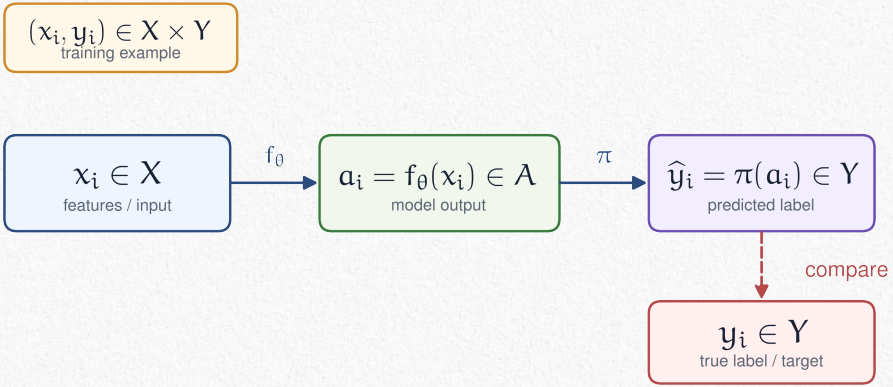
- The classical perceptron rule replaces the raw residual $y_i - \tilde{y}_i$ by the classification residual $y_i - \hat{y}_i$.
- An alternative approach would be to use a different type of activation function.

# Supervised learning

- We're given a set of **features** $X$, a set $Y$ of **targets** or **labels**, and a set $A$ of **model outputs**.
- Let $(x_1, y_1), \ldots, (x_N, y_N)$ be a finite collection of data points in $X \times Y$ sampled from an unknown process or distribution. These points form our **training set**.
- The features and model outputs are typically vectors ($X \subset \mathbf{R}^n$, $A \subset \mathbf{R}^m$), the labels might be vectors, but they could also e.g. be elements of some discrete set of **class labels**.
- A **model (class)** is a parameterised family of functions $(f_\theta \colon X \to A)_{\theta \in \Theta}$. The parameters include so-called **weights** and **biases**.
- We're assuming that we're also giving a **prediction map** $\pi \colon A \to Y$ that translates model outputs to labels.

  Often, $A = Y$ and $\pi = \mathrm{id}$ is the identity function. On the other hand, the perceptron uses $\pi = H$.
- If the $y_i$ are vectors (and $\pi = \mathrm{id}$), then supervised learning is known as **regression**.

# Supervised learning: one training example through the model




$f_\theta : X \to A$ is chosen from a parameterised model class; $\pi : A \to Y$ translates model outputs into labels.

# Supervised learning

- In our applications, we'd usually like to apply our trained model to *unseen data* outside of the training set. However, good performance on training data does not imply good performance on unseen data. This is the **generalisation problem**.
- Having somehow chosen a model class (and prediction map), our goal is to find $\theta$ such that $\pi(f_\theta(x_i)) \approx y_i$ for $i = 1, \dots, N$. More precisely, we seek to find $\theta$ minimising the quantity (called **empirical risk** or **training objective**)

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i).$$

  This process of numerical optimisation is called **training**.
- **Model** can mean *either* the model class $(f_\theta)_{\theta \in \Theta}$ *or* a fitted model $f_\theta$.
- The **(per-example) loss function** $\ell \colon A \times Y \to \mathbf{R}$ measures the distance between the true labels $y_i$ and our model outputs $f_\theta(x_i)$.
- *Warning.* In the literature, "loss function" can sometimes refer to either $\ell$ or $J$.

# Supervised learning

- **Training** is the process of adjusting model parameters based on data. The goal is to reduce prediction errors as measured by the loss function.
- If $Y = \{1, \ldots, K\}$, then a data point $(x_i, y_i)$ attaches the **class label** $y_i$ to feature $x_i$. Predicting class labels is called **classification**.
  Examples:
  - Binary classification = properties. For instance, given the adjacency matrix of a graph, decide if it is connected.
  - Multiclass classification: given a plane curve, classify its singularity type (smooth, node, cusp, ...)

# Generalising perceptrons?!

- We saw that a perceptron

$$f_{w,b}\colon \mathbf{R}^d \to \mathbf{R}, \quad x \mapsto H(w \cdot x + b)$$

  can be trained to correctly identify binary classes that are linearly separable.
- The raw affine output $g_{w,b}(x) = w \cdot x + b$ is just an affine (scalar-valued) function. It is natural to generalise this and consider vector-valued affine functions $\mathbf{R}^d \to \mathbf{R}^e, x \mapsto Ax + b$ for a matrix $A \in M_{e\times d}(\mathbf{R})$ and vector $b \in \mathbf{R}^e$. Finding an affine function that best approximates a collection of data points $(x_1, y_1), \ldots, (x_N, y_N)$ is **linear regression**. This is basic linear algebra.

  (Statisticians make a big fuss about this ... )
- In this way, the perceptron becomes a simple tool for classifying sets *and* for approximating functions.
- **Multi-layer perceptrons (MLPs)** chain together affine maps and suitable activation functions.

# Artificial neurons

- Let $w \in \mathbf{R}^d$, $b \in \mathbf{R}$, and let $\sigma\colon \mathbf{R} \to \mathbf{R}$.
- An **artificial neuron** with $d$ inputs and activation function $\sigma$ is a function $\phi\colon \mathbf{R}^d \to \mathbf{R}$ of the form

$$\phi(x) = \sigma(w \cdot x + b).$$

- If $\sigma$ is the Heaviside step function, we recover a perceptron.
- Given $\sigma$, we abuse notation and use the same symbol to denote the function $\mathbf{R}^n \to \mathbf{R}^n$ that applies $\sigma$ in each coordinate.
- A sequence $(\phi_1, \ldots, \phi_e)$ of artificial neurons with $d$ inputs and (common) activation function $\sigma$ is really the same as a function $\Phi\colon \mathbf{R}^d \to \mathbf{R}^e$ of the form

$$\Phi(x) = \sigma(Wx + c)$$

for $W \in M_{e\times d}(\mathbf{R})$ and $c \in \mathbf{R}^e$.

# Multi-layer perceptrons (MLPs)

- MLPs are also referred to as *fully connected feedforward neural networks*.
- The hyperparameters of an MLP are its **depth** $L \geqslant 1$, the **layer widths** $d_0, \dots, d_L \geqslant 1$, and the **activation functions** $\sigma_1, \dots, \sigma_L \colon \mathbf{R} \to \mathbf{R}$.
- The parameters are a weight matrix $W_\ell \in \mathrm{M}_{d_\ell \times d_{\ell-1}}(\mathbf{R})$ and bias $b_\ell \in \mathbf{R}^{d_\ell}$ for $\ell = 1, \dots, L$. We write $\theta = (W_\ell, b_\ell)_{\ell=1,\dots,L}$.
- Our MLP computes a function

$$T_\theta = T_{L,\theta} \circ \cdots \circ T_{1,\theta} \colon \mathbf{R}^{d_0} \xrightarrow{T_{1,\theta}} \mathbf{R}^{d_1} \to \cdots \xrightarrow{T_{L,\theta}} \mathbf{R}^{d_L}$$

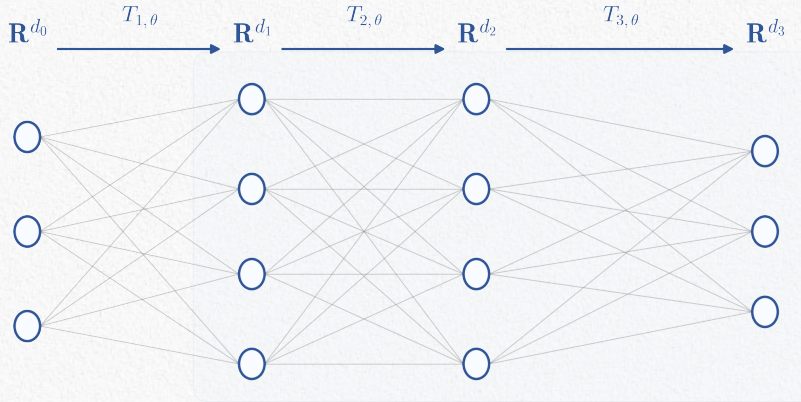
  given by

$$T_{\ell,\theta} \colon \mathbf{R}^{d_{\ell-1}} \to \mathbf{R}^{d_\ell}, \quad x \mapsto \sigma_\ell(W_\ell x + b_\ell).$$

  *Note.* We allow the final activation function to be vector-valued: $\sigma_L \colon \mathbf{R}^{d_L} \to \mathbf{R}^{d_L}$.

- Let $T_{0,\theta} \colon \mathbf{R}^{d_0} \to \mathbf{R}^{d_0}$ be the identity function. Then $T_{0,\theta}, \dots, T_{L,\theta}$ are the **layers** of our MLP. The **input layer** is $T_{0,\theta}$. We call $T_{1,\theta}, \dots, T_{L,\theta}$ the **learned layers**. The **hidden layers** are $T_{1,\theta}, \dots, T_{L-1,\theta}$.

# Multi-layer perceptrons (MLPs): illustration




$L = 3, \qquad T_\theta = T_{3,\theta} \circ T_{2,\theta} \circ T_{1,\theta}, \qquad T_{\ell,\theta}(x) = \sigma_\ell(W_\ell x + b_\ell)$

# Multi-layer perceptrons (MLPs)

- A perceptron ($L = 1$) doesn't have hidden layers. **Deep learning** refers to the case of more than one hidden layer. Excellent branding!
- In practice, one usually uses the same activation function $\sigma = \sigma_1 = \cdots = \sigma_{L-1}$ for all layers *except for the last*. The final activation function is special ... it crucially depends on what we want our MLP to do for us.

# Universal approximation: overview

**Question**

What functions can be computed by MLPs?

- If all activation functions are affine linear, then our MLP itself computes an affine linear function, which is very restrictive.
- There is a considerable body of literature on **Universal Approximation Theorems (UATs)**. These characterise classes of functions that can be approximated by specific types of MLPs (and more general neural networks).
  The first UAT is due to Cybenko (1989).

**Theorem (Leshno, Lin, Pinkus, Schocken 1993 (informal version))**

*Let $\sigma\colon \mathbf{R} \to \mathbf{R}$ be "close to continuous", but not a polynomial.*
*Then MLPs with one hidden layer and intermediate activation function $\sigma$ can approximate continuous functions on compact subsets of $\mathbf{R}^n$ with any precision.*

skip

# Universal approximation: details

- Let $\sigma\colon \mathbf{R} \to \mathbf{R}$ be a function satisfying the following properties:
  - $\sigma$ is Lebesgue measurable.
  - $\sigma$ is (essentially) bounded on every compact subset of $\mathbf{R}$.
  - The closure of the set of points of discontinuity of $\sigma$ has Lebesgue measure zero.
- Main example of $\sigma$: a piecewise continuous function

**Theorem (Leshno, Lin, Pinkus, Schocken 1993)**

*The following are equivalent:*

1. *For every $n \geqslant 1$, every compact set $K \subset \mathbf{R}^n$, every continuous function $f\colon K \to \mathbf{R}$, and every $\varepsilon > 0$, there exist $N \geqslant 1$, $W \in M_{N\times n}(\mathbf{R})$, $b, c \in \mathbf{R}^N$ such that for all $x \in K$, we have*
$$|c \cdot \sigma(Wx + b) - f(x)| < \varepsilon.$$
2. *$\sigma$ is not a polynomial.*

# Universal approximation

- This is an *existence* result. It doesn't tell us how to actually train an MLP (i.e. how to tune MLP parameters) to approximate a specific function.
- Training will (essentially) be based on *gradient descent*, similar to how we derived the perceptron rule.
- Using gradient descent requires two ingredients:
  - Specific (mostly) differentiable choices of activation functions.
  - Some way of measuring the difference between model outputs and actual truths. This is what *loss functions* do.

# Activation functions

- For function approximation, the final activation function is often just the identity.
- Historically, *sigmoids* were popular activation functions. A **sigmoid** is a bounded increasing function $\mathbf{R} \to \mathbf{R}$ such as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  with range $(0, 1)$.
- The most popular intermediate activation function is the **rectified linear unit** $\mathrm{ReLU}(x) = \max(0, x)$. Very cheap to compute and differentiate (away from zero)!

# General classification problems and softmax

- Let $x_1, \dots, x_N \in \mathbf{R}^n$. Suppose that we are given a finite set $\mathcal{Y}$ consisting of $K$ **class labels** and that each $x_i$ is labelled $y_i \in \mathcal{Y}$. Write $Y = \{1, \dots, K\}$.
- Our goal is to train MLPs to predict class labels. Choose an MLP architecture giving rise to functions $T_\theta \colon \mathbf{R}^n \to \mathbf{R}^K$.
- We convert the raw outputs of $T_\theta$ (called **logits**) into probabilities using **softmax**:

$$\operatorname{softmax}(z_1, \dots, z_K) = \left( \frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}, \dots, \frac{e^{z_K}}{\sum_{k=1}^{K} e^{z_k}} \right).$$

- Write $\operatorname{softmax}(T_\theta(x)) = (p_\theta(y = 1 \mid x), \dots, p_\theta(y = K \mid x))$. We interpret this as a vector of conditional probabilities.
- To predict the class label of $x \in \mathbf{R}^n$, pick an entry with the largest probability:

$$\hat{y} = \arg\max_{a \in \mathcal{Y}} p_\theta(y = a \mid x).$$

# Loss functions and backpropagation

- Recall: given a model class $(f_\theta \colon X \to A)_{\theta \in \Theta}$ and training data $(x_1, y_1), \ldots, (x_N, y_N) \in X \times Y$, the (per-example) loss function $\ell \colon A \times Y \to \mathbf{R}$ measures how close the model outputs are to the actual labels.
- Training a model means minimising the average loss

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

  across our training examples.
- Using gradient descent, this means picking an initial parameter $\theta_0$ and updating

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t),$$

  where $\eta > 0$ is the learning rate.

# Loss functions and backpropagation

- Note that

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(f_\theta(x_i), y_i).$$

- In case of MLPs (and neural networks), $f_\theta$ is an explicit composition of affine functions and activation functions. We can compute $\nabla J(\theta)$ using the chain rule.
- **Backpropagation** (Rumelhart 1985) is an algorithm for differentiating $J$ that cleverely avoids repetitions and is used by ML libraries.

# Stochastic gradient descent

- In practice, for large training sets, computing

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

  can be very expensive.
- For $B \subset \{1, \dots, N\}$, let

$$J_B(\theta) = \frac{1}{|B|} \sum_{i \in B} \ell(f_\theta(x_i), y_i).$$

- Instead of differentiating $J$, during each update cycle, one typically chooses a small random **mini-batch** $B$ and updates

$$\theta_{t+1} = \theta_t - \eta \nabla J_B(\theta_t).$$

  This is called **stochastic gradient descent (SGD)**.

# Square loss and beyond

- For function prediction, the standard loss function is the **square loss**

$$\ell(u, v) = \|u - v\|_2^2,$$

  where $\|u\|_2 = \sqrt{u_1^2 + \ldots + u_n^2}$ is the Euclidean norm of $u \in \mathbf{R}^n$.
- Because the per-example loss function $\ell$ and the empirical loss $J$ are traditionally confused, this loss function is often called **mean squared error (MSE)**.
- Square loss heavily penalises large errors, but it is also highly sensitive to outliers.
- One lesson from our ML experiments is that choosing a good loss function is an art!

# Training a model using SGD: an overview

- Suppose we're given a (large) set of data points $(x_1, y_1), \ldots, (x_M, y_M)$.
- Choose the size $N \leqslant M$ of the training set. After (randomly) permuting data points, we may assume that $(x_1, y_1), \ldots, (x_N, y_N)$ form our training set.
- Choose a model architecture (e.g. MLP) and loss function according to the task.
- Choose hyperparameters (e.g. depth and width, activation functions).
- Choose a mini-batch size $B \leqslant N$.
- Randomly initialise the parameters $\theta$.
- Repeat until happy:
  - Do the following $\lceil N/B \rceil$ times:
    - Choose a random mini-batch of size $B$ from the training data.
    - Perform SGD on the mini-batch and update $\theta$.

If we pretend that our random mini-batches are disjoint, then each iteration of the outer loop passes through the entire training set once. This is called an **epoch**.

# Training, validation, test

- We usually split data into:
  - a training set
  - a validation set
  - a test set
- The training set is used to update parameters.
- The validation set is used to tune hyperparameters and to detect overfitting.
- The test set is used at the very end to estimate performance on unseen data.

# Cross entropy loss

- Consider a classification problem with set of class labels $Y = \{1, \ldots, K\}$ and training points $(x_1, y_1), \ldots, (x_N, y_N)$.
- Let

$$\Delta_K^\circ = \left\{ (p_1, \ldots, p_K) \in \mathbf{R}^K : 0 < p_k < 1 \text{ for all } k \text{ and } \sum_{k=1}^{K} p_k = 1 \right\}$$

  be the interior of the standard $K$-simplex.
- We choose an MLP (or neural network) architecture defining functions $f_\theta \colon \mathbf{R}^n \to \Delta_K^\circ$; the final activation function is $\mathrm{softmax}$.
- Let $z_\theta(x) = (z_{1,\theta}(x), \ldots, z_{K,\theta}(x)) \in \mathbf{R}^K$ denote the model output / logits prior to the final $\mathrm{softmax}$; hence, $f_\theta(x) = \mathrm{softmax}(z_\theta(x))$.
- As before, write $f_\theta(x) = (p_\theta(y = 1 \mid x), \ldots, p_\theta(y = K \mid x))$.

# Cross entropy loss

- Given the input $x_i$, our model assigns the probability $p_\theta(y = y_i \mid x_i)$ to $x_i$ being labelled using the correct class $k$. We'd like to tune $\theta$ to bring this probability close to $1$.
- Under the model and conditional independence assumptions, the likelihood of the observed labels is $\prod_{i=1}^{N} p_\theta(y = y_i \mid x_i)$.
- Maximising this quantity is equivalent to minimising its negative logarithm. Dividing by $N$, we obtain

$$-\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(y = y_i \mid x_i),$$

which is the empirical risk associated with the loss function $\ell\colon \Delta_K^\circ \times Y \to \mathbf{R}$ given by

$$\ell(p, k) = -\log(p_k).$$

# Cross entropy loss

- The loss function $\ell(\mathrm{p}, \mathrm{k}) = -\log(\mathrm{p}_\mathrm{k})$ goes by several names: **cross-entropy loss**, **log loss**, **logistic loss**. This is the standard loss function for classification tasks.
- In practice, cross entropy loss is computed directly on the logits $z_\theta(\mathrm{x})$ without applying $\mathrm{softmax}$.
- For future reference: in PyTorch, `CrossEntropyLoss` expects logits, not softmax probabilities.

# Overfitting

*"Past performance is no indicator of future results."*
*(used by various financial regulators)*

- Just as in real life, one can learn the wrong lessons from experience.
- **Overfitting** occurs when a model performs well on training data but poorly on unseen data. This means that a trained model is too specifically tailored to the training data, whereas the real goal is to find rules that generalise well to new data.
- Mathematically, this happens when the empirical loss is small on training data, but the test/validation loss is large.
- This can happen for a number of reasons, e.g. a training set that is too small, a model that is too flexible, training periods that are too long, ...

# Optimisers

- Gradient descent and SGD aren't our only choices when it comes to minimising our loss function.
- One popular variant is to perform SGD with **momentum**. Here, we update

$$\theta_{t+1} = \theta_t - \eta \nu_t,$$

  where $\nu_t = \mu \nu_{t-1} + \nabla J(\theta_t)$. This is meant to smooth out noisy gradients.
- A more sophisticated optimisation algorithm is **Adam** (adaptive moment estimation).
  Adam is often a good default for experimentation.

# Normalisation

- Before training begins, it is common to **normalise** (or **standardise**) the training examples. This is not usually required (or performed automatically) by software libraries, but often needed for good and reliable performance.
- Suppose that $x_1, \ldots, x_N \in \mathbf{R}^n$ are our training examples. For each coordinate $j = 1, \ldots, n$, compute the mean

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{ij}$$

and standard deviation

$$\sigma_j = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_{ij} - \mu_j)^2}.$$

- After normalisation, we then consider

$$\tilde{x}_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}.$$

# Normalisation

- Input is typically normalised.
- In case of regression problems, normalisation is also common for outputs. In that case, model predictions will need to be unnormalised.
- *Important.* The normalisation parameters ($\mu_j$, $\sigma_j$) are computed only from the training set. The same affine transformation is applied to validation/test data.

# Practical comments

Popular software libraries for machine learning include:

- scikit-learn
- TensorFlow (with or without Keras)
- JAX
- PyTorch

These provide tools for:

- models
- automatic differentiation
- optimisation
- GPU acceleration

We'll use PyTorch for the practical parts.