Chapter 3

Training

As introduced in § 1.1, training a model consists of minimizing a loss $\mathscr{L}(w)$ which reflects the performance of the predictor $f(\cdot; w)$ on a training set $\mathscr{D}$.

Since models are usually extremely complex, and their performance is directly related to how well the loss is minimized, this minimization is a key challenge, which involves both computational and mathematical difficulties.

3.1 Losses

The example of the mean squared error from Equation 1.1 is a standard loss for predicting a continuous value.

For density modeling, the standard loss is the likelihood of the data. If $f(x;w)$ is to be interpreted as a normalized log-probability or log-density, the loss is the opposite of the sum of its values over training samples, which corresponds to the likelihood of the data-set.

Cross-entropy

For classification, the usual strategy is that the output of the model is a vector with one component $f(x;w)_y$ per class y, interpreted as the logarithm of a non-normalized probability, or logit.

With X the input signal and Y the class to predict, we can then compute from f an estimate of the posterior probabilities:

$$\hat{P}(Y = y \mid X = x) = \frac{\exp f(x;w)_y}{\sum_z \exp f(x;w)_z}.$$

This expression is generally called the softmax, or more adequately, the softargmax, of the logits.

To be consistent with this interpretation, the model should be trained to maximize the probability of the true classes, hence to minimize the cross-entropy, expressed as:

$$\begin{aligned}\mathscr{L}_{\text{ce}}(w) &= -\frac{1}{N}\sum_{n=1}^{N}\log \hat{P}(Y = y_n \mid X = x_n) \\ &= \frac{1}{N}\sum_{n=1}^{N}\underbrace{-\log \frac{\exp f(x_n;w)_{y_n}}{\sum_z \exp f(x_n;w)_z}}_{L_{\text{ce}}(f(x_n;w),y_n)}.\end{aligned}$$

Contrastive loss

In certain setups, even though the value to be predicted is continuous, the supervision takes the form of ranking constraints. The typical domain where this is the case is metric learning, where the objective is to learn a measure of distance between samples such that a sample x_a from a certain semantic class is closer to any sample x_b of the same class than to any sample x_c from another class. For instance, x_a and x_b can be two pictures of a certain person, and x_c a picture of someone else.

The standard approach for such cases is to minimize a contrastive loss, in that case, for instance, the sum over triplets (x_a, x_b, x_c), such

that $y_a = y_b \neq y_c$, of

$$\max(0, 1 - f(x_a, x_c; w) + f(x_a, x_b; w)).$$

This quantity will be strictly positive unless $f(x_a, x_c; w) \geq 1 + f(x_a, x_b; w)$.

Engineering the loss

Usually, the loss minimized during training is not the actual quantity one wants to optimize ultimately, but a proxy for which finding the best model parameters is easier. For instance, cross-entropy is the standard loss for classification, even though the actual performance measure is a classification error rate, because the latter has no informative gradient, a key requirement as we will see in § 3.3.

It is also possible to add terms to the loss that depend on the trainable parameters of the model themselves to favor certain configurations.

The weight decay regularization, for instance, consists of adding to the loss a term proportional to the sum of the squared parameters. This can be interpreted as having a Gaussian Bayesian prior on the parameters, which favors smaller values and thereby reduces the influence of the data. This degrades performance on the train-

ing set, but reduces the gap between the performance in training and that on new, unseen data.

3.2 Autoregressive models

A key class of methods, particularly for dealing with discrete sequences in natural language processing and computer vision, are the autoregressive models,

The chain rule for probabilities

Such models put to use the chain rule from probability theory:

$$
\begin{aligned}
&P(X_1 = x_1, X_2 = x_2, \ldots, X_T = x_T) = \\
&\quad P(X_1 = x_1) \\
&\times P(X_2 = x_2 \mid X_1 = x_1) \\
&\quad \ldots \\
&\times P(X_T = x_T \mid X_1 = x_1, \ldots, X_{T-1} = x_{T-1}).
\end{aligned}
$$

Although this decomposition is valid for a random sequence of any type, it is particularly efficient when the signal of interest is a sequence of tokens from a finite vocabulary $\{1, \ldots K\}$.

With the convention that the additional token $\emptyset$ stands for an "unknown" quantity, we can represent the event $\{X_1 = x_1, \ldots, X_t = x_t\}$ as the vector $(x_1, \ldots, x_t, \emptyset, \ldots, \emptyset)$.

Then, a model

$$f : \{\emptyset, 1, \ldots, K\}^T \to \mathbb{R}^K$$

which, given such an input, computes a vector l_t of K logits corresponding to

$$\hat{P}(X_t \mid X_1 = x_1, \ldots, X_{t-1} = x_{t-1}),$$

allows to sample one token given the previous ones.

The chain rule ensures that by sampling T tokens x_t, one at a time given the previously sampled $x_1, \ldots, x_{t-1}$, we get a sequence that follows the joint distribution. This is an autoregressive generative model.

Training such a model can be done by minimizing the sum across training sequences and time steps of the cross-entropy loss

$$L_{\text{ce}}\big(f(x_1, \ldots, x_{t-1}, \emptyset, \ldots, \emptyset; w), x_t\big),$$

which is formally equivalent to maximizing the likelihood of the true x_ts.

The value that is classically monitored is not the cross-entropy itself, but the perplexity, which is defined as the exponential of the cross-entropy. It corresponds to the number of values of a uniform distribution with the same entropy, which is generally more interpretable.

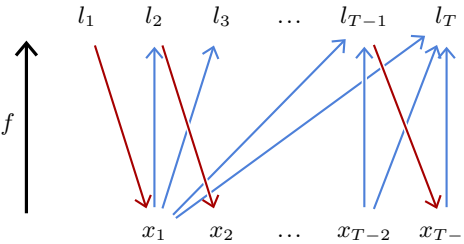

Figure 3.1: *An autoregressive model f, is causal if a time step x_t of the input sequence modulates the predicted logits l_s only if $s > t$, as depicted by the blue arrows. This allows computing the distributions at all the time steps in one pass during training. During sampling, however, the l_t and x_t are computed sequentially, the latter sampled with the former, as depicted by the red arrows.*

Causal models

The training procedure we just described requires a different input for each t, and the bulk of the computation done for $t < t'$ is repeated for t'. This is extremely inefficient since T is often of the order of hundreds or thousands.

The standard strategy to address this issue is to design a model f that predicts all the vectors of logits $l_1, \dots, l_T$ at once, that is:

$$f : \{1, \dots, K\}^T \to \mathbb{R}^{T \times K},$$

but with a computational structure such that the computed logits l_t for x_t depend only on the input values $x_1, \ldots, x_{t-1}$.

Such a model is called causal, since it corresponds, in the case of temporal series, to not letting the future influence the past, as illustrated in Figure 3.1.

The consequence is that the output at every position is the one that would be obtained if the input were only available up to before that position. During training, it allows one to compute the output for a full sequence and to maximize the predicted probabilities of all the tokens of that same sequence, which again boils down to minimizing the sum of the per-token cross-entropy.

Note that, for the sake of simplicity, we have defined f as operating on sequences of a fixed length T. However, models used in practice, such as the transformers we will see in § 5.3, are able to process sequences of arbitrary length.

Tokenizer

One important technical detail when dealing with natural languages is that the representation as tokens can be done in multiple ways, ranging from the finest granularity of individual symbols

to entire words. The conversion to and from the token representation is carried out by a separate algorithm called a tokenizer.

A standard method is the Byte Pair Encoding (BPE) [Sennrich et al., 2015] that constructs tokens by hierarchically merging groups of characters, trying to get tokens that represent fragments of words of various lengths but of similar frequencies, allocating tokens to long frequent fragments as well as to rare individual symbols.

3.3 Gradient descent

Except in specific cases like the linear regression we saw in § 1.2, the optimal parameters w^* do not have a closed-form expression. In the general case, the tool of choice to minimize a function is gradient descent. It starts by initializing the parameters with a random w_0, and then improves this estimate by iterating gradient steps, each consisting of computing the gradient of the loss with respect to the parameters, and subtracting a fraction of it:

$$w_{n+1} = w_n - \eta \nabla \mathscr{L}_{|w}(w_n). \tag{3.1}$$

This procedure corresponds to moving the current estimate a bit in the direction that locally decreases $\mathscr{L}(w)$ maximally, as illustrated in Figure 3.2.

Learning rate

The hyper-parameter η is called the learning rate. It is a positive value that modulates how quickly the minimization is done, and must be chosen carefully.

If it is too small, the optimization will be slow at best, and may be trapped in a local minimum early. If it is too large, the optimization may

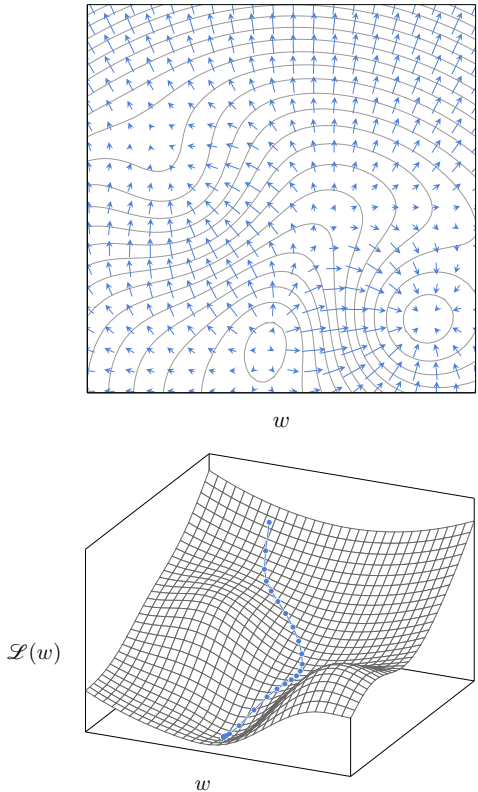

Figure 3.2: *At every point w, the gradient $\nabla\mathscr{L}_{|w}(w)$ is in the direction that maximizes the increase of $\mathscr{L}$, orthogonal to the level curves (top). The gradient descent minimizes $\mathscr{L}(w)$ iteratively by subtracting a fraction of the gradient at every step, resulting in a trajectory that follows the steepest descent (bottom).*

bounce around a good minimum and never descend into it. As we will see in § 3.6, it can depend on the iteration number n.

Stochastic Gradient Descent

All the losses used in practice can be expressed as an average of a loss per small group of samples, or per sample such as:

$$\mathscr{L}(w) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(w),$$

where $\ell_n(w) = L(f(x_n; w), y_n)$ for some L, and the gradient is then:

$$\nabla\mathscr{L}_{|w}(w) = \frac{1}{N}\sum_{n=1}^{N} \nabla\ell_{n|w}(w). \qquad (3.2)$$

The resulting gradient descent would compute exactly the sum in Equation 3.2, which is usually computationally heavy, and then update the parameters according to Equation 3.1. However, under reasonable assumptions of exchangeability, for instance, if the samples have been properly shuffled, any partial sum of Equation 3.2 is an unbiased estimator of the full sum, albeit noisy. So, updating the parameters from partial sums corresponds to doing more gradient steps

for the same computational budget, with noisier estimates of the gradient. Due to the redundancy in the data, this happens to be a far more efficient strategy.

We saw in § 2.1 that processing a batch of samples small enough to fit in the computing device's memory is generally as fast as processing a single one. Hence, the standard approach is to split the full set $\mathscr{D}$ into batches, and to update the parameters from the estimate of the gradient computed from each. This is called mini-batch stochastic gradient descent, or stochastic gradient descent (SGD) for short.

It is important to note that this process is extremely gradual, and that the number of mini-batches and gradient steps are typically of the order of several million.

As with many algorithms, intuition breaks down in high dimensions, and although it may seem that this procedure would be easily trapped in a local minimum, in reality, due to the number of parameters, the design of the models, and the stochasticity of the data, its efficiency is far greater than one might expect.

Plenty of variations of this standard strategy have been proposed. The most popular one is

Adam [Kingma and Ba, 2014], which keeps running estimates of the mean and variance of each component of the gradient, and normalizes them automatically, avoiding scaling issues and different training speeds in different parts of a model.

3.4 Backpropagation

Using gradient descent requires a technical means to compute $\nabla \ell_{|w}(w)$ where $\ell = L(f(x;w);y)$. Given that f and L are both compositions of standard tensor operations, as for any mathematical expression, the chain rule from differential calculus allows us to get an expression of it.

For the sake of making notation lighter, we will not specify at which point gradients are computed, since the context makes it clear.

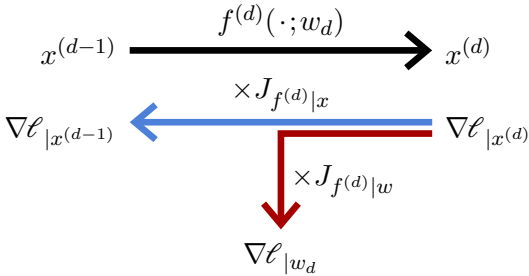

Figure 3.3: *Given a model $f = f^{(D)} \circ \cdots \circ f^{(1)}$, the forward pass computes the outputs $x^{(d)}$ of the $f^{(d)}$ in order (top, black). The backward pass computes the gradients of the loss with respect to the activations $x^{(d)}$ (bottom, blue) and the parameters w_d (bottom, red) backward by multiplying them by the Jacobians.*

Forward and backward passes

Consider the simple case of a composition of mappings:

$$f = f^{(D)} \circ f^{(D-1)} \circ \cdots \circ f^{(1)}.$$

The output of $f(x; w)$ can be computed by starting with $x^{(0)} = x$ and applying iteratively:

$$x^{(d)} = f^{(d)}\Big(x^{(d-1)}; w_d\Big),$$

with $x^{(D)}$ as the final value.

The individual scalar values of these intermediate results $x^{(d)}$ are traditionally called activations in reference to neuron activations, the value D is the depth of the model, the individual mappings $f^{(d)}$ are referred to as layers, as we will see in § 4.1, and their sequential evaluation is the forward pass (see Figure 3.3, top).

Conversely, the gradient $\nabla \ell_{|x^{(d-1)}}$ of the loss with respect to the output $x^{(d-1)}$ of $f^{(d-1)}$ is the product of the gradient $\nabla \ell_{|x^{(d)}}$ with respect to the output of $f^{(d)}$ multiplied by the Jacobian $J_{f^{(d-1)}|x}$ of $f^{(d-1)}$ with respect to its variable x. Thus, the gradients with respect to the outputs of all the $f^{(d)}$s can be computed recursively backward, starting with $\nabla \ell_{|x^{(D)}} = \nabla L_{|x}$.

And the gradient that we are interested in for training, that is $\nabla \ell_{|w_d}$, is the gradient with respect to the output of $f^{(d)}$ multiplied by the Jacobian $J_{f^{(d)}|w}$ of $f^{(d)}$ with respect to the parameters.

This iterative computation of the gradients with respect to the intermediate activations, combined with that of the gradients with respect to the layers' parameters, is the backward pass (see Figure 3.3, bottom). The combination of this computation with the procedure of gradient descent is called backpropagation.

In practice, the implementation details of the forward and backward passes are hidden from programmers. Deep learning frameworks are able to automatically construct the sequence of operations to compute gradients.

A particularly convenient algorithm is Autograd [Baydin et al., 2015], which tracks tensor operations and builds, on the fly, the combination of operators for gradients. Thanks to this, a piece of imperative programming that manipulates tensors can automatically compute the gradient of any quantity with respect to any other.

Resource usage

Regarding the computational cost, as we will see, the bulk of the computation goes into linear operations, each requiring one matrix product for the forward pass and two for the products by the Jacobians for the backward pass, making the latter roughly twice as costly as the former.

The memory requirement during inference is roughly equal to that of the most demanding individual layer. For training, however, the backward pass requires keeping the activations computed during the forward pass to compute the Jacobians, which results in a memory usage that grows proportionally to the model's depth. Techniques exist to trade the memory usage for computation by either relying on reversible layers [Gomez et al., 2017], or using checkpointing, which consists of storing activations for some layers only and recomputing the others on the fly with partial forward passes during the backward pass [Chen et al., 2016].

Vanishing gradient

A key historical issue when training a large network is that when the gradient propagates backwards through an operator, it may be scaled by a

multiplicative factor, and consequently decrease or increase exponentially when it traverses many layers. A standard method to prevent it from exploding is gradient norm clipping, which consists of re-scaling the gradient to set its norm to a fixed threshold if it is above it [Pascanu et al., 2013].

When the gradient decreases exponentially, this is called the vanishing gradient, and it may make the training impossible, or, in its milder form, cause different parts of the model to be updated at different speeds, degrading their co-adaptation [Glorot and Bengio, 2010].

As we will see in Chapter 4, multiple techniques have been developed to prevent this from happening, reflecting a change in perspective that was crucial to the success of deep-learning: instead of trying to improve generic optimization methods, the effort shifted to engineering the models themselves to make them optimizable.

3.5 The value of depth

As the term "deep learning" indicates, useful models are generally compositions of long series of mappings. Training them with gradient descent results in a sophisticated co-adaptation of the mappings, even though this procedure is gradual and local.

We can illustrate this behavior with a simple model $\mathbb{R}^2 \to \mathbb{R}^2$ that combines eight layers, each multiplying its input by a 2×2 matrix and applying Tanh per component, with a final linear classifier. This is a simplified version of the standard Multi-Layer Perceptron that we will see in § 5.1.

If we train this model with SGD and cross-entropy on a toy binary classification task (Figure 3.4, top left), the matrices co-adapt to deform the space until the classification is correct, which implies that the data have been made linearly separable before the final affine operation (Figure 3.4, bottom right).

Such an example gives a glimpse of what a deep model can achieve; however, it is partially misleading due to the low dimension of both the signal to process and the internal representations. Everything is kept in 2D here for the sake of

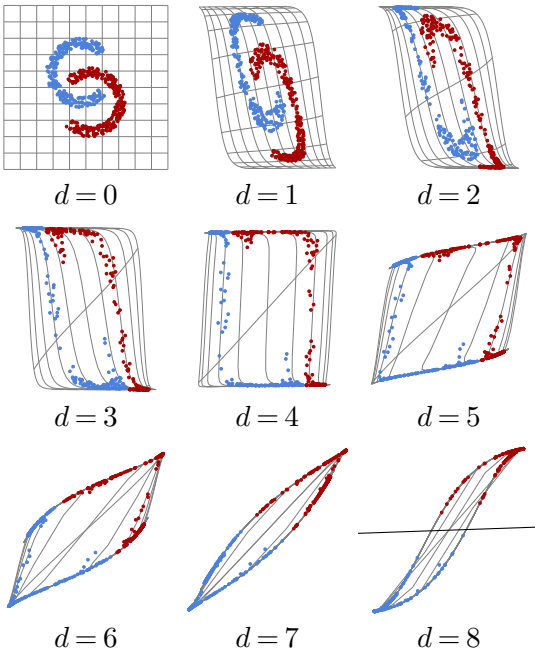

Figure 3.4: *Each plot shows the deformation of the space and the resulting positioning of the training points in $\mathbb{R}^2$ after d layers of processing, starting with the input to the model itself (top left). The oblique line in the last plot (bottom right) shows the final affine decision.*

visualization, while real models take advantage of representations in high dimensions, which, in particular, facilitates the optimization by providing many degrees of freedom.

Empirical evidence accumulated over twenty years demonstrates that state-of-the-art performance across application domains necessitates models with tens of layers, such as residual networks (see § 5.2) or Transformers (see § 5.3).

Theoretical results show that, for a fixed computational budget or number of parameters, increasing the depth leads to a greater complexity of the resulting mapping [Telgarsky, 2016].

3.6 Training protocols

Training a deep network requires defining a protocol to make the most of computation and data, and to ensure that performance will be good on new data.

As we saw in § 1.3, the performance on the training samples may be misleading, so in the simplest setup one needs at least two sets of samples: one is a training set, used to optimize the model parameters, and the other is a test set, to evaluate the performance of the trained model.

Additionally, there are usually hyper-parameters to adapt, in particular, those related to the model architecture, the learning rate, and the regularization terms in the loss. In that case, one needs a validation set that is disjoint from both the training and test sets to assess the best configuration.

The full training is usually decomposed into epochs, each of which corresponds to going through all the training examples once. The usual dynamic of the losses is that the training loss decreases as long as the optimization runs, while the validation loss may reach a minimum after a certain number of epochs and then start to increase, reflecting an overfitting regime, as

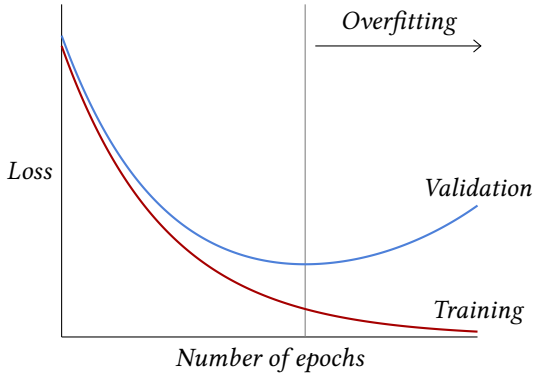

Figure 3.5: *As training progresses, a model's performance is usually monitored through losses. The training loss is the one driving the optimization process and goes down, while the validation loss is estimated on an other set of examples to assess the overfitting of the model. Overfitting appears when the model starts to take into account random structures specific to the training set at hand, resulting in the validation loss starting to increase.*

introduced in § 1.3 and illustrated in Figure 3.5.

Paradoxically, although they should suffer from severe overfitting due to their capacity, large models usually continue to improve as training progresses. This may be due to the inductive bias of the model becoming the main driver of optimization when performance is near perfect

on the training set [Belkin et al., 2018].

An important design choice is the learning rate schedule during training, that is, the specification of the value of the learning rate at each iteration of the gradient descent. The general policy is that the learning rate should be initially large to avoid having the optimization being trapped in a bad local minimum early, and that it should get smaller so that the optimized parameter values do not bounce around and reach a good minimum in a narrow valley of the loss landscape.

The training of very large models may take months on thousands of powerful GPUs and have a financial cost of several million dollars. At this scale, the training may involve many manual interventions, informed, in particular, by the dynamics of the loss evolution.

Fine-tuning

It is often beneficial to adapt an already trained model to a new task, referred to as a downstream task.

It can be because the amount of data for the original task is plentiful, while they are limited for the downstream task, and the two tasks share enough similarities that statistical struc-

tures learned for the first provide a good inductive bias for the second. It can also be to limit the training cost by reusing the patterns encoded in an existing model.

Adapting a pre-trained model to a specific task is achieved with fine-tuning, which is a standard training procedure for the downstream task, but which starts from the pre-trained model instead of using a random initialization.

This is the main strategy for most computer vision applications which generally use a model pre-trained for classification on ImageNet [Deng et al., 2009] (see § 6.3 and § 6.4), and it is also how purely generative pre-trained Large Language Models are re-purposed as assistant-like models, able to produce interactive dialogues (see § 7.1).

We come back to techniques to cope with limited resources in inference and for fine-tuning in Chapter 8.

3.7 The benefits of scale

There is an accumulation of empirical results showing that performance, for instance, estimated through the loss on test data, improves with the amount of data according to remarkable scaling laws, as long as the model size increases correspondingly [Kaplan et al., 2020] (see Figure 3.6).

Benefiting from these scaling laws in the multi-billion sample regime is possible in part thanks to the structure of deep models which can be scaled up arbitrarily, as we will see, by increasing the number of layers or feature dimensions. But it is also made possible by the distributed nature of the computation they implement, and by the stochastic gradient descent, which requires only a fraction of the data at a time and can operate with datasets whose size is orders of magnitude greater than that of the computing device's memory. This has resulted in an exponential growth of the models, as illustrated in Figure 3.7.

Typical vision models have 10–100 million trainable parameters and require 10^{18}–10^{19} FLOPs for training [He et al., 2015; Sevilla et al., 2022]. Language models have from 100 million to hundreds of billions of trainable parameters and re-

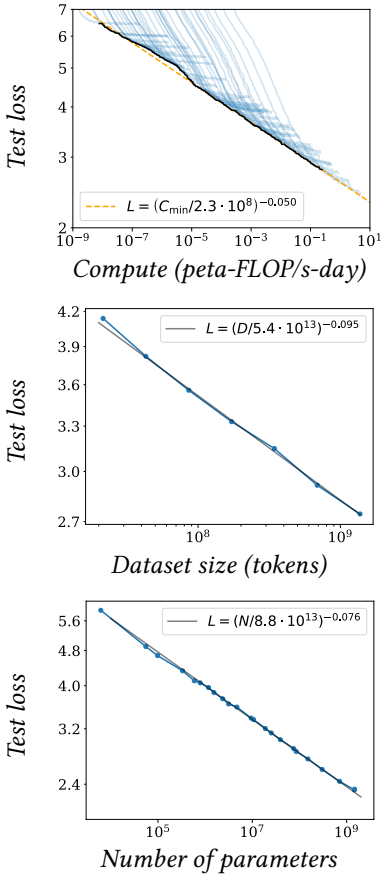

Figure 3.6: *Test loss of a language model vs. the amount of computation in petaflop/s-day, the dataset size in tokens, that is fragments of words, and the model size in parameters [Kaplan et al., 2020].*

Dataset	Year	Nb. of images	Size
ImageNet	2012	1.2M	150Gb
Cityscape	2016	25K	60Gb
LAION-5B	2022	5.8B	240Tb

Dataset	Year	Nb. of books	Size
WMT-18-de-en	2018	14M	8Gb
The Pile	2020	1.6B	825Gb
OSCAR	2020	12B	6Tb

Table 3.1: *Some examples of publicly available datasets. The equivalent number of books is an indicative estimate for 250 pages of 2000 characters per book.*

quire 10^{20}–10^{23} FLOPs for training [Devlin et al., 2018; Brown et al., 2020; Chowdhery et al., 2022; Sevilla et al., 2022]. These latter models require machines with multiple high-end GPUs.

Training these large models is impossible using datasets with a detailed ground-truth costly to produce, which can only be of moderate size. Instead, it is done with datasets automatically produced by combining data available on the internet with minimal curation, if any. These sets may combine multiple modalities, such as text and images from web pages, or sound and images from videos, which can be used for large-scale supervised training.

As of 2024, the most powerful models are the

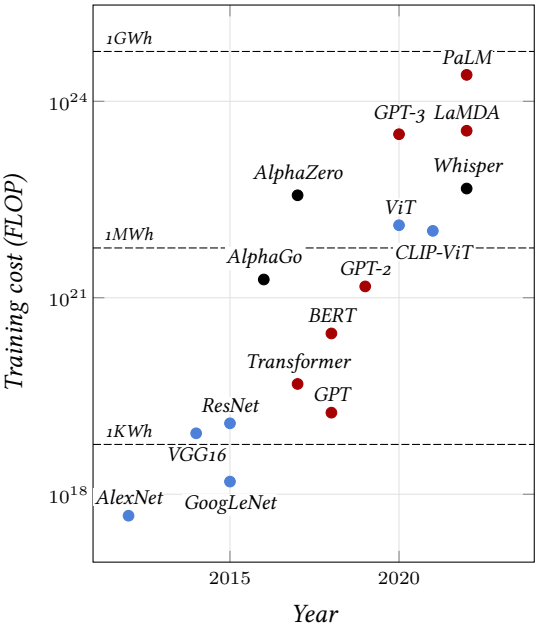

Figure 3.7: *Training costs in number of FLOP of some landmark models [Sevilla et al., 2023]. The colors indicate the domains of application: Computer Vision (blue), Natural Language Processing (red), or other (black). The dashed lines correspond to the energy consumption using A100s SXM in 16-bit precision. For reference, the total electricity consumption in the US in 2021 was* 3920 *TWh.*

so-called Large Language Models (LLMs), which we will see in § 5.3 and § 7.1, trained on extremely large text datasets (see Table 3.1).

Part II

Deep Models