# Building blocks of modern machine learning: (self-)attention and diffusion models

Joscha Diehl

June 5, 2023

## Contents

For coding tutorials we will work with `tensorflow`, just for the sake of concreteness (and since several groups in Greifswald are using it). But the concepts taught are of course independent of the particular framework. Moreover, currently available frameworks (`pytorch` and `JAX/Flax` are other examples) are so similar, that switching between them is very easy.

Initial remarks:

- We *will* get our hands dirty, and implement (again and again) the mechanisms we talk about. We will not be too interested in the (very successful) large pipelines; instead we will prefer to probe and take apart small models.
- The machine learning content will be interspersed with various kinds of rigorous mathematics. It is good to want to understand all of this in detail, and I am happy to answer all questions. Just note that sometimes it pays off to take quoted results, momentarily, for granted. (This will probably happen mostly in the section on diffusion models.)
- Text in blue is generated with the help of GPT4 (with possible minor edits by me). If meaningful, a link to the prompt is given.

**Notation:**

- $\mathbb{R}$ for the set of real numbers, elements are usually written in standard text (e.g. $x, y, ..$)
- $\mathbb{R}_{>0}$ for the set of positive real numbers
- $\mathbb{N} = \{0, 1, \dots\}$ for the set of natural numbers
- $\mathbb{N}_{\geq 1} = \{1, \dots\}$ for the set of positive natural numbers
- $\mathbb{Z}$ for the set of integers
- $\mathbb{R}^d$ for the set of $d$-dimensional vectors, elements are usually written in bold (e.g. $\mathbf{x}, \mathbf{y}, ..$)
- $|x|$ for the absolute value of $x \in \mathbb{R}$
- $||\mathbf{x}|| := \sqrt{\sum_{i=1} x_i^2}$ for the Euclidean norm of $\mathbf{x} \in \mathbb{R}^d$.

# 1 Neural networks

A **neural network** is nothing but a parametrized function

$$f_\theta : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}},$$

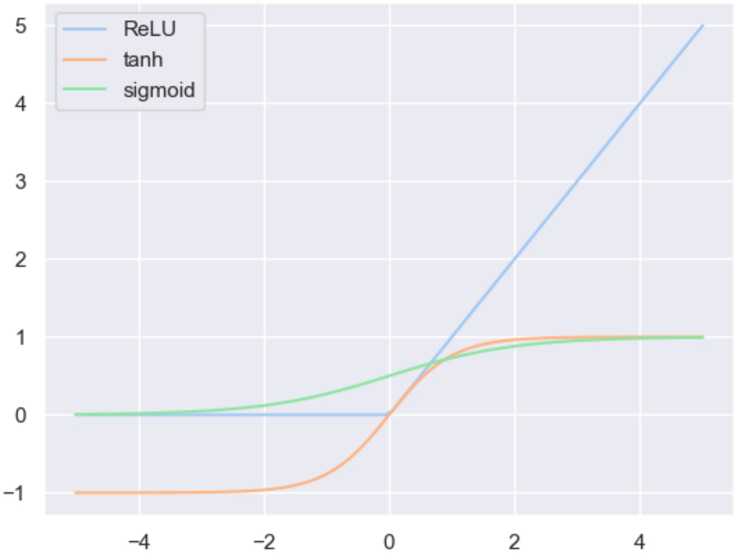
where $\theta \in \mathbb{R}^p$ are the **parameters** or **weights** of the network.

To really warrant the name "neural network" it is usually built by **composing** very simple functions. Important ingredients are

- **affine maps** [1] for $A \in \mathbb{R}^{n\times m}$ (also called a **kernel**), $\mathbf{b} \in \mathbb{R}^n$ (also called a **bias term**), this is a map

$$\begin{aligned} \mathbb{R}^m &\to \mathbb{R}^n \\ \mathbf{x} &\mapsto A\mathbf{x} + \mathbf{b}. \end{aligned}$$

- **activation functions**; this is any kind of nonlinear, one-dimensional, function, e.g.



  – (rectifier linear unit)

$$\begin{aligned} \mathrm{ReLU} : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto 1_{[0,\infty)}(x) \cdot x = \max\{0, x\}. \end{aligned}$$

[1] Often misnamed **linear maps**; they are linear only if the bias term is zero.

-
$$\tanh : \mathbb{R} \to \mathbb{R}.$$

-
$$\begin{aligned}\text{sigmoid} : \mathbb{R} &\to \mathbb{R}\\ x &\mapsto \frac{1}{1+e^{-x}}.\end{aligned}$$

-
$$\begin{aligned}\mathsf{GeLU} : \mathbb{R} &\to \mathbb{R}\\ x &\mapsto x\mathbb{P}_{\mathcal{N}(0,1)}[X \leq x] = x\frac{1}{2}\left(1+\operatorname{erf}(\frac{x}{\sqrt{2}})\right).\end{aligned}$$

-
$$\begin{aligned}\mathsf{ELU} : \mathbb{R} &\to \mathbb{R}\\ x &\mapsto \begin{cases} x & x > 0\\ \exp(x) - 1 & x \leq 0.\end{cases}\end{aligned}$$

**Remark 1.1.** *We will mostly use the ReLU function.*[2]

*Pros:*

- *Empirical evidence. ("Deep CNNs with ReLUs train several times faster than their equivalents with tanh units." [KSH17])*
- *"creating sparse representations with true zeros" [GBB11]*
- *"folding" / changing the topology of input data [NZL20]*
- *(restricted) homogeneity: for all $\alpha > 0, x \in \mathbb{R}$,*
$$\mathrm{ReLU}(\alpha x) = \alpha\,\mathrm{ReLU}(x)$$

*Cons:*

- *If all units are non-active (i.e. the input is smaller than $0$), gradient descent will not work. ("dead neurons")*
- *Discontinuous derivative at $0$. This is usually* not *a problem (since one never "hits" zero).*
- ReLU *is unbounded which can lead to instabilities and/or overconfidence. [HAB19]*

[2] Other activation functions are in use, in particular in the last layer; more on this later.

We now build an example of a neural network. First, for $A \in \mathbb{R}^{n\times m}, \mathbf{b} \in \mathbb{R}^n$, (so here, $\theta = (A, \mathbf{b})$) define

$$\begin{aligned}\mathrm{Dense}'_{A,\mathbf{b}} : \mathbb{R}^m &\to \mathbb{R}^n\\ \mathbf{x} &\mapsto A\mathbf{x} + \mathbf{b}.\end{aligned}$$

Problem: if we stack this layer, for example with $A_1 \in \mathbb{R}^{n_1\times m}, \mathbf{b}_1 \in \mathbb{R}^{n_1}, A_2 \in \mathbb{R}^{n\times n_1}, \mathbf{b}_2 \in \mathbb{R}^n$, we get

$$\begin{aligned}\mathrm{Dense}'_{A_2,\mathbf{b}_2} \circ \mathrm{Dense}'_{A_1,\mathbf{b}_1} : \mathbb{R}^m &\to \mathbb{R}^n\\ \mathbf{x} &\mapsto A_2A_1\mathbf{x} + A_2\mathbf{b}_1 + \mathbf{b}_2,\end{aligned}$$

which is again an affine map! **We need to introduce some non-linearity to make compositions of layers interesting.** Define

$$\begin{aligned}\mathrm{Dense}_{A,\mathbf{b}} := \mathrm{Dense}_{A,\mathbf{b};\mathrm{ReLU}} : \mathbb{R}^m &\to \mathbb{R}^n\\ \mathbf{x} &\mapsto \mathrm{ReLU}(A\mathbf{x} + \mathbf{b}).\end{aligned} \tag{1}$$

Here, ReLU is applied *entrywise*. Spelled out in coordinates this is,

$$\mathrm{Dense}_{A,\mathbf{b}}(\mathbf{x}) = \begin{pmatrix} \mathrm{ReLU}\left(\sum_{j=1}^m A_{1j}x_j + b_1\right)\\ \mathrm{ReLU}\left(\sum_{j=1}^m A_{2j}x_j + b_2\right)\\ \cdots\\ \mathrm{ReLU}\left(\sum_{j=1}^m A_{nj}x_j + b_n\right)\end{pmatrix}.$$

Now we can stack, to get interesting functions.

**Example 1.2.** $d_{\mathrm{in}} = 2, d_1 = 5, d_{\mathrm{out}} = 1$, *Note the annoying fact that in figures, the input is usually on the left, and the output on the right, whereas in mathematical formulas we usually write the input on the right.*

*https://github.com/diehlj/2023-building-blocks-lecture/blob/master/examples/nn251.py*

Of course we can use more layers,

$$\mathrm{Dense}_{A_n,\mathbf{b}_n} \circ \ldots \circ \mathrm{Dense}_{A_1,\mathbf{b}_1}.$$

The number $n$ is also called the **depth** of such a network. The last (outermost) layer, $\mathrm{Dense}_{A_n,\mathbf{b}_n}$, is also called **output layer**. It often uses a different activation function, e.g. the sigmoid function or the identity function. The other layers, $\mathrm{Dense}_{A_i,\mathbf{b}_i}$, $i = 1, \ldots, n-1$, are **hidden layers**. The **depth** of such a network is $n - 1$.

The **width** of a layer $\mathrm{Dense}_{A,b}$ is the dimension of the codomain. The **width** of a network is the maximal width of its layers.

Since the matrices $A$ are arbitrary, in a graphic representation like in Figure 1 all nodes in neighboring layers are connected. Such neural networks are said to be **fully connected** and are also called **multilayer perceptrons**.

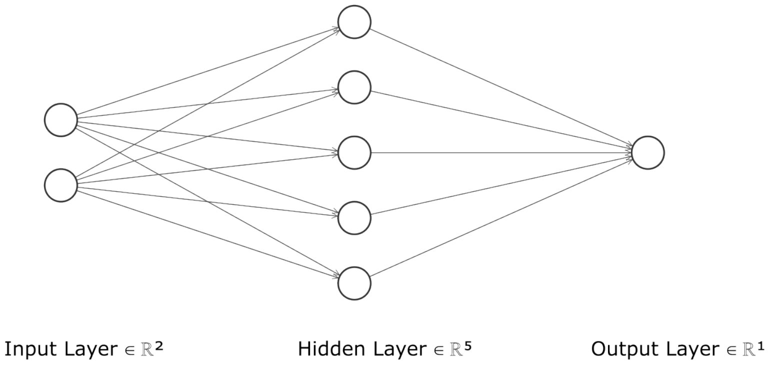


Figure 1: *A neural network with* $d_{\text{in}} = 2, d_{inner} = 5, d_{\text{out}} = 1$*.*

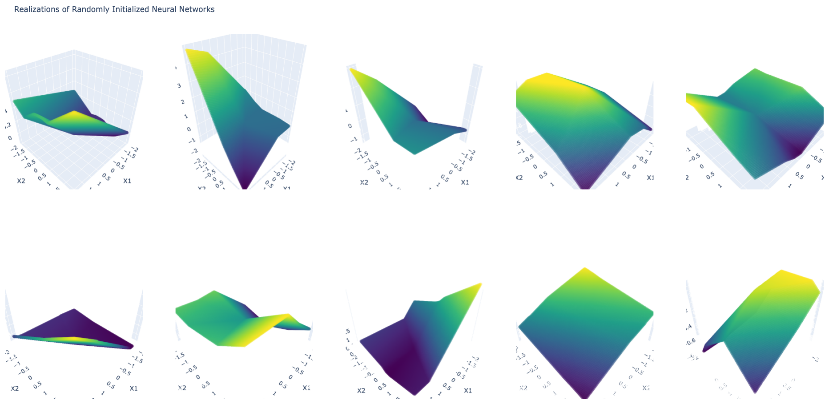


Figure 2: *Example realizations of a neural network with* $d_{\text{in}} = 2, d_{inner} = 5, d_{\text{out}} = 1$*.*

## 1.1 Expressiveness

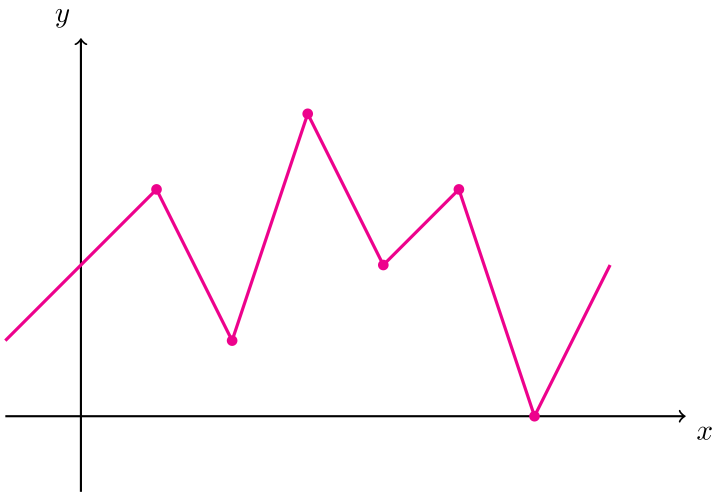
Let us consider the following network,

$$\text{Dense}_{A_2,\mathbf{b}_2;\text{id}} \circ \text{Dense}_{A_1,\mathbf{b}_1;\text{ReLU}} : \mathbb{R} \to \mathbb{R}, \tag{2}$$

with one hidden layer (with ReLU activation) and one output layer (with identity activation).

What functions can this network compute?

**Lemma 1.3.** *A continuous function* $f : \mathbb{R} \to \mathbb{R}$ *is* ***piecewise linear*** *if there exist*



*$t_0 < t_1 < \cdots < t_p$ and linear maps $L_0, \ldots, L_p, L_{p+1}$ such that*

$$f(x) = \begin{cases} L_0(x), & x \in (-\infty, t_0] \\ L_\ell(x), & x \in [t_{\ell-1}, t_\ell] \\ L_{p+1}(x), & x \in [t_p, +\infty). \end{cases}$$

*For every such $f$ there exists a neural network (2) (with one hidden layer with ReLU activation and one linear output layer) that is equal to $f$. In other words, there exist $n$, $a^{(1)}, a^{(2)}, b_i^{(1)}, b^{(2)} \in \mathbb{R}$ such that*

$$f(x) = \sum_{i=1}^{n} a_i^{(2)} \operatorname{ReLU}(a_i^{(1)} x + b_i^{(1)}) + b^{(2)}.$$

**Remark 1.4.** *This is* not *true for neural networks with input dimension 2, i.e. there are piecewise linear functions $\mathbb{R}^2 \to \mathbb{R}$ that cannot be represented by a neural network with one hidden layer (ReLU activation) and one output layer (identity activation). See [Fou22, Theorem 24.1].*

*Proof.* Write

$$L_\ell(x) = \lambda_\ell x + c_\ell.$$

Then

$$f(x) = \lambda_0 \, (x \wedge t_0) + c_0 + \sum_{\ell=1}^{p-1} \lambda_\ell \, \big((0 \vee (x - t_\ell)) \wedge (t_{\ell+1} - t_\ell)\big) + \lambda_p \, \big(0 \vee (x - t_p)\big) \, .$$

Note that this uses the fact that $L_{\ell-1}(t_{\ell-1}) = L_\ell(t_{\ell-1}), \ell = 1, \ldots, p+1$.

Then, using

$$\lambda_\ell \big((0 \vee (x - t_\ell)) \wedge (t_{\ell+1} - t_\ell)\big) = \lambda_\ell \,\mathrm{ReLU}(x - t_\ell) - \lambda_\ell \,\mathrm{ReLU}(x - t_{\ell+1}),$$

we get that this is equal to

$$\begin{aligned} &\lambda_0 x + c_0 + \sum_{\ell=1}^{p+1} (\lambda_\ell - \lambda_{\ell-1}) \,\mathrm{ReLU}(x - t_\ell) \\ &\quad = \mathrm{ReLU}(\lambda_0 x) - \mathrm{ReLU}(-\lambda_0 x) + c_0 + \sum_{\ell=1}^{i-1} (\lambda_\ell - \lambda_{\ell-1}) \,\mathrm{ReLU}(x - t_\ell), \end{aligned}$$

as desired. □

As a corollary we obtain a special case of the following theorem.

**Theorem 1.5** (Universal approximation theorem; arbitrary width; ReLU activation)**.**[3] *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous function. Then the functions*

$$\mathrm{Dense}_{A_2,\mathbf{b}_2;\mathrm{id}} \circ \mathrm{Dense}_{A_1,\mathbf{b}_1;\sigma},$$

*are dense in the space of continuous functions on compact sets (under the supremum norm) if and only if $\sigma$ is* not *a polynomial function.*

*This means that for all continuous functions $g : \mathbb{R}^{d_{\mathrm{in}}} \to \mathbb{R}^{d_{\mathrm{out}}}$, every $\epsilon > 0$, and every compact set $K \subset \mathbb{R}^{d_{\mathrm{in}}}$, there exist $d_{\mathrm{hidden}} \in \mathbb{N}_{\geq 1}$ and weights $A_1 \in \mathbb{R}^{d_{\mathrm{hidden}} \times d_{\mathrm{in}}}, A_2 \in \mathbb{R}^{d_{\mathrm{out}} \times d_{\mathrm{hidden}}}, \mathbf{b}_1 \in \mathbb{R}^{d_{\mathrm{hidden}}}, \mathbf{b}_2 \in \mathbb{R}^{d_{\mathrm{out}}}$ such that*

$$\sup_{x \in K} ||g(x) - \mathrm{Dense}_{A_2,\mathbf{b}_2;\mathrm{id}} \circ \mathrm{Dense}_{A_1,\mathbf{b}_1;\sigma}(x)|| < \epsilon.$$

Bis hier: Lecture 3. 20.4.23

*Proof.* ⇒: If $\sigma$ is a polynomial of order $N$, then the output of the neural network is also a polynomial of order at most $N$. Polynomials of bounded degree are not dense in continuous functions.

⇐: We only show the case $\sigma = \mathrm{ReLU}$; the case of arbitrary $\sigma$ is shown in [Fou22, Theorem 25.1].

First $d_{\mathrm{in}} = d_{\mathrm{out}} = 1$. By Lemma 1.3 it remains to show that piecewise linear functions are dense in the space of continuous functions on $K$ for every compact set $K \subset \mathbb{R}$. Without loss of generality we can take $K = [0, 1]$.

Let $g$ be a continuous function on $[0, 1]$. By the Heinor-Cantor theorem [Lan12, Proposition 3.11], $g$ is *uniformly continuous*, that is

$$\forall \epsilon > 0 : \exists \delta > 0 : \forall x, y \in [0, 1] : |x - y| < \delta \Rightarrow |g(x) - g(y)| < \epsilon.$$

[3] References: 1D: [Fou22, Theorem 24.1], general: [Fou22, Theorem 25.1]

Now, for a given $\epsilon > 0$ pick such a $\delta > 0$. Possibly making it smaller, let $\delta = T/N$ for some $N \in \mathbb{N}$. Let $L$ be the continuous function that is linear on the intervals $[t_{\ell-1}, t_\ell]$ for $\ell = 1, \ldots, N$ and satisfies

$$L(t_\ell) = g(t_\ell), \quad \ell = 0, \ldots, N.$$

Let $x \in [0,1]$ be given. Then we have $x \in [t_{\ell-1}, t_\ell]$ for some $\ell \in \{0, \ldots, N\}$ and

$$\begin{aligned}
|g(x) - L(x)| &= |g(x) - g(t_\ell)| + |g(t_\ell) - L(t_\ell)| + |L(t_\ell) - L(x)| \\
&\leq |g(x) - g(t_\ell)| + |g(t_\ell) - L(t_\ell)| + |L(t_\ell) - L(x)| \\
&\leq \epsilon + 0 + \frac{|g(t_\ell) - g(t_{\ell-1})|}{\delta} |x - t_{\ell-1}| \\
&\leq \epsilon + |g(t_\ell) - g(t_{\ell-1})| \leq 2\epsilon.
\end{aligned}$$

Hence

$$\sup_{x \in [0,1]} |g(x) - L(x)| \leq 2\epsilon.$$

Now, for arbitrary $d_{\text{in}}$ (and still $d_{\text{out}} = 1$, $\sigma = \text{ReLU}$). Consider the subspace of $C(K, \mathbb{R})$ given by

$$\mathcal{A} := \operatorname{span}_{\mathbb{R}} \{\exp(\langle \nu, \cdot \rangle) : \nu \in \mathbb{R}^{d_{\text{in}}}\}.$$

It is a subalgebra, since

$$\exp(\langle \nu, \cdot \rangle) \exp(\langle \mu, \cdot \rangle) = \exp(\langle \nu + \mu, \cdot \rangle).$$

It contains the constant function $1 = \exp(\langle 0, \cdot \rangle)$. It separates points, since for all $x, y \in K$, there exists $\nu \in \mathbb{R}^{d_{\text{in}}}$ such that

$$\langle \nu, x \rangle \neq \langle \nu, y \rangle.$$

By Theorem 1.6, for $g \in C(K, \mathbb{R})$ and there exist an integer $k$ and $\nu_i \in \mathbb{R}^{d_{\text{in}}}, \gamma_i \in \mathbb{R}$, $i = 1, \ldots, k$, such that

$$|g(x) - \sum_{i=1}^{k} \gamma_i \exp(\langle \nu_i, x \rangle)| < \epsilon, \quad \forall x \in K.$$

Now, for every $i$,

$$t \mapsto \gamma_i \exp(t),$$

is a continuous function and we only evaluate it in the compactum

$$K_i := \big\{ \langle \nu_i, x \rangle : x \in K \big\} \subset \mathbb{R}.$$

By above result in the one-dimensional case, there exist integers $r_i, i = 1, \dots, k$ and $a_\ell^{(i)}, b_\ell^{(i)}, c_\ell^{(i)} \in \mathbb{R}$, $\ell = 1, \dots, r_i$, $i = 1, \dots, k$, such that

$$\sup_{t \in K_i} |\gamma_i \exp(t) - \left( \sum_{\ell=1}^{r_i} c_\ell^{(i)} \operatorname{ReLU}(a_\ell^{(i)} t + b_\ell^{(i)}) + d^{(i)} \right) | < \frac{\epsilon}{k}, \quad i = 1, \dots, k.$$

Then for $x \in K$,

$$\begin{aligned} &|g(x) - \left( \sum_{i=1}^{k} \sum_{\ell=1}^{r_i} c_\ell^{(i)} \operatorname{ReLU}(a_\ell^{(i)} t + b_\ell^{(i)}) + d^{(i)} \right) | \\ &\quad \leq |g(x) - \sum_{i=1}^{k} \gamma_i \exp(\langle \nu_i, x \rangle)| \\ &\qquad + \sum_{i=1}^{k} |\gamma_i \exp(\langle \nu_i, x \rangle) - \left( \sum_{\ell=1}^{r_i} c_\ell^{(i)} \operatorname{ReLU}(a_\ell^{(i)} \langle \nu_i, x \rangle + b_\ell^{(i)}) + d^{(i)} \right) | \\ &\quad \leq 2\epsilon. \end{aligned}$$

□

**Theorem 1.6** (Stone-Weierstrass theorem)**.** *Let $K \subset \mathbb{R}^d$ be a compact set and $C(K, \mathbb{R})$ the $\mathbb{R}$-algebra of continuous, real-valued functions on $K$. Let $\mathcal{A}$ be a subalgebra of $C(K, \mathbb{R})$, satisfying*

1. *$\mathcal{A}$ is* ***nowhere vanishing****, that is for all $x \in K$ there is $\phi \in \mathcal{A}$ such that*

$$\phi(x) \neq 0.$$

2. *$\mathcal{A}$ is* ***separating points****, that is for all $x, y \in K$ there is $\phi \in \mathcal{A}$ such that*

$$\phi(x) \neq \phi(y).$$

*Then: $\mathcal{A}$ is dense in $C(K, \mathbb{R})$ with norm given by the supremum norm.*

*Proof.* See `https://diehlj.github.io/stone-weierstrass/stone-weierstrass.html` for a proof following [BD81].

Other proofs: [Fou22, Theorem E.3], [Loo13, p.9]. □

So, why not just work with single layer neural networks then?

> "In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error." [GBC16].

Some results in this direction:

**Theorem 1.7.**

- *(Universal approximation theorem; arbitrary depth) [HS17] Width* $d_{\text{in}} + 1$ *and arbitrary depth, neural networks with ReLU activation are dense in the space of continuous functions on a compactum* $K \subset \mathbb{R}^{d_{\text{in}}}$*.*
- *A FFN with ReLU activation has number of affine regions of the order ([MPCB14])*

$$\mathcal{O}(\binom{\mathsf{width}}{d_{\text{in}}}^{d_{\text{in}}(\mathsf{depth}-1)} \mathsf{width}^{d_{\text{in}}})$$

- *[Tel16] For all* $L \geq 1$ *there is a depth* $\mathcal{O}(L^2)$ *FFN* $f$ *with* $\mathcal{O}(L^2)$ *nodes such that for all depth* $\mathcal{O}(L)$ *FFNs* $g$ *with* $\mathcal{O}(2L^2)$ *nodes we have*

$$\int_0^1 \left|f(x) - g(x)\right| \,\mathrm{d}x > \frac{1}{32}.$$

So, deep feed forward networks are what we should work with? **Usually not.** Often it pays to posit some **inductive bias**, usually affecting the network architecture. Starting from FFNs this usually means putting restrictions on the weight matrices.

Bis hier: Lecture 4. 24.4.23

- Convolutional Neural Networks (CNNs): Local and translation-invariant feature learning due to convolutional layers with smaller shared weight matrices (kernels), reducing the number of parameters and focusing on spatial hierarchies.[4]

  Consider a 1-dimensional convolutional layer, with *stride* 1, kernel $a \in \mathbb{R}^3$ of size 3, no bias, no *padding* and ReLU activation. The input is any $x \in \mathbb{R}^{d_{\text{in}}}$ and the output is

$$\begin{pmatrix} \mathrm{ReLU}(a_1x_1 + a_2x_2 + a_3x_3) \\ \mathrm{ReLU}(a_1x_2 + a_2x_3 + a_3x_4) \\ \vdots \\ \mathrm{ReLU}(a_1x_{d_{\text{in}}-2} + a_2x_{d_{\text{in}}-1} + a_3x_{d_{\text{in}}}) \end{pmatrix} \in \mathbb{R}^{d_{\text{in}}-2}.$$

  We can realize this as a dense layer with a very structured weight matrix, namely

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & 0 & \cdots & 0 \\ 0 & a_1 & a_2 & a_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & a_1 & a_2 & a_3 \end{pmatrix} \in \mathbb{R}^{d_{\text{in}}-2\times d_{\text{in}}}.$$

  The fact that the (small) kernel is shared across the input (i.e. is re-used again and again) is called **weight sharing**.

[4] GPT4-prompt: https://sharegpt.com/c/4htnBbg

See https://adamharley.com/nn_vis/cnn/2d.html for a visualization of a 2-dimensional convolutional layer (and a $2d$ FFN for comparison: https://adamharley.com/nn_vis/mlp/2d.html).

- Recurrent Neural Networks (RNNs): Sequence processing and long-range dependency learning through looping connections, enabling internal state maintenance for tasks like natural language processing and time-series analysis.
- Graph Neural Networks (GNNs): Efficient graph-structured data handling by structuring weight matrices to aggregate information from neighboring nodes, learning meaningful node representations.

Weight sharing, prominently featured in CNNs, offers multiple benefits, such as reduced memory requirements, faster training, and increased generalization due to fewer parameters.

One can also more fundamentally modify the architecture, using layers that are not of the form form (1), i.e. of the form "affine map, then entrywise nonlinearity". We will see that attention layers are such an example.

## 1.2 Tensors

Data is usually stored in **tensors**: scalar values (0-dimensional tensors), vectors (1-dimensional tensors), matrices (2-dimensional tensors), ...

In this course we consider tensors just as *multi-dimensional arrays* of numbers. To be precise, a tensor of **order** $n$ is an element $T \in \mathbb{R}^{d_1 \times \ldots \times d_n}$. For $i_1 \in \{1, \ldots, d_1\}$, ..., $i_n \in \{1, \ldots, d_n\}$, we write

$$T_{i_1,\ldots,i_n} \in \mathbb{R},$$

for the entry of $T$ at position $(i_1, \ldots, i_n)$. $i_k$ is said to index the $k$-th **axis** of $T$. The **shape** of $T$ is the tuple $(d_1, \ldots, d_n)$.

**Einstein summation** is a convenient notation to express tensor operations. For example the matrix-vector product, $A\mathbf{v}$ results in a vector (tensor of order 1)

$$\begin{aligned}(A\mathbf{v})_i &= \sum_{j=1}^{d} A_{i,j} v_j \\ &= A_{i,j} v_j.\end{aligned}$$

In Einstein's notation, a summation is implicitly applied to every index (here, just the index $j$) that appears at least twice in the expression.

Other example:

- Matrix-matrix product:

$$A_{i,k} B_{k,j} = (AB)_{i,j}.$$

- Matrix trace:

$$A_{i,i} = \mathrm{Tr}(A).$$

- Matrix-matrix product, with a different order:

$$A_{i,k}B_{j,k} = (AB^\top)_{i,j}.$$

- A bigger example:

$$A_{i,k,\ell}B_{k,\ell,j} = \sum_{k,\ell} A_{i,k,\ell}B_{k,\ell,j}.$$

All deep learning frameworks provide a method to compute the Einstein summation of a tensor expression. In tensorflow this is the function `tf.einsum`.

The **tensor product** of tensors $S \in \mathbb{R}^{d_1\times\ldots\times d_n}$ and $T \in \mathbb{R}^{d_{n+1}\times\ldots\times d_{n+m}}$ is defined as

$$S \otimes T \in \mathbb{R}^{d_1\times\ldots\times d_n\times d_{n+1}\times\ldots\times d_{n+m}}.$$

where

$$(S\otimes T)_{i_1,\ldots,i_n,j_1,\ldots,j_m} = S_{i_1,\ldots,i_n}T_{j_1,\ldots,j_m}.$$

The tensor product is associative, but not commutative.

A **pure** tensor is a tensor $T$ that can be represented as the tensor product of a set of vectors,

$$\mathcal{T} = \mathbf{v}_1 \otimes \mathbf{v}_2 \otimes \cdots \otimes \mathbf{v}_n,$$

with $\mathbf{v}_i \in \mathbb{R}^{d_i}$.

The **rank**[5] of a tensor $T$ is the minimal $r$ such that $T$ is the sum of $r$ pure tensors, i.e. the minimal $r$ such that there exixts $\mathbf{u}_i^{(j)}$

$$T = \sum_{i=1}^{r} \mathbf{u}_i^{(1)} \otimes \mathbf{u}_i^{(2)} \otimes \cdots \otimes \mathbf{u}_i^{(n)}.$$

Note that pure tensors have rank one.

Bis hier: Lecture 5. 27.4.23

## 1.3 Learning

The usual task in (supervised) machine learning is the prediction of a value given some input.

- **Classification**: we want to predict a discrete label of the input. (e.g. `cat` or `dog` for an image)

[5] Sometimes people denote the *order* of a tensor with rank. This is *not* good use of nomenclature.

- **Regression**: we want to approximate a, usually real-valued and continuous, function. Classical example: linear regression.

In **classification**, a collection $(\mathbf{x}_i^{\mathsf{train}}, y_i^{\mathsf{train}}) \in \mathbb{R}^{d_{\text{in}}} \times [n]$, $1 \leq i \leq N_{\mathsf{train}}$, of examples (training data) is given, where $\mathbf{x}_i^{\mathsf{train}}$ is the input, say an image, and $y_i$ is the label, say 1 (`cat`) or 2 (`dog`). We want to construct a function

$$\hat{f} : \mathbb{R}^{d_{\text{in}}} \to [n],$$

that does a 'good job' at predicting the labels.

First, we aim for a function that does well on the training data. To measure 'goodness' we use a **loss function**

$$\mathsf{loss}' : [n] \times [n] \to \mathbb{R}$$

that measures the distance between the predicted label and the true label. For reasons that will become clear, we usually let the network predict a *probability distribution over the labels*, i.e.

$$\hat{f}(\mathbf{x}) \in \mathbb{R}^n,$$

with $\sum_{i=1}^n \hat{f}(\mathbf{x})_i = 1$ and $\hat{f}(\mathbf{x})_i \geq 0$. Then

$$\mathsf{loss} : \mathbb{R}^n \times [n] \to \mathbb{R}.$$

Define the **training loss** as

$$\mathsf{loss}_{\text{train}}(f) := \frac{1}{N_{\mathsf{train}}} \sum_{i=1}^{N_{\mathsf{train}}} \mathsf{loss}(f(\mathbf{x}_i^{\mathsf{train}}), y_i^{\mathsf{train}}).$$

We want to find a function $\hat{f}$ that minimizes the training loss. This is called **training** the network. If $f_\theta$ is our parametrized function, this amounts to

$$\theta^* \in \operatorname{argmin}_\theta \mathsf{loss}_{\text{train}}(f_\theta).$$

We usually do this via **gradient descent** (later). This will find (if we are lucky) a parameter $\hat{\theta}$, such that $\hat{f} := f_{\hat{\theta}}$ that does well on the training data. However, this is not enough. We want a function $\hat{f}$ that does well on **unseen** data. This is called **generalization**. We can measure generalization by evaluating the function $\hat{f}$ on a **test set**, a collection $(\mathbf{x}_i^{\mathsf{test}}, y_i^{\mathsf{test}}) \in \mathbb{R}^{d_{\text{in}}} \times [n]$, $1 \leq i \leq N_{\mathsf{test}}$, of examples. The **test loss** is defined as

$$\mathsf{loss}_{\mathsf{test}}(f) := \frac{1}{N_{\mathsf{test}}} \sum_{i=1}^{N_{\mathsf{test}}} \mathsf{loss}(f(\mathbf{x}_i^{\mathsf{test}}), y_i^{\mathsf{test}}).$$

**Remark 1.8.** *If the function $\hat{f}$ does well on the training data, but not on the test data, we say that the network **overfits** the training data. This is a common problem and many ways to combat it exist.*

In **regression** we aim to approximate an (unknown) function

$$\phi : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}.$$

Again, we are given training data $(\mathbf{x}_i^{\mathsf{train}}, \mathbf{y}_i^{\mathsf{train}}) \in \mathbb{R}^{d_{\text{in}}} \times \mathbb{R}^{d_{\text{out}}}$, where we think of $\mathbf{y}_i \approx \phi(\mathbf{x}_i)$. We want to construct a function

$$\hat{f} : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}},$$

that minimizes some loss on the data (first, on the training data, but really we are interested in doing well on test data).

$$\mathsf{loss}_{\mathsf{train}}(f) := \frac{1}{N_{\mathsf{train}}} \sum_{i=1}^{N_{\mathsf{train}}} \mathsf{loss}(f(\mathbf{x}_i^{\mathsf{train}}), \mathbf{y}_i),$$

where

$$\mathsf{loss} : \mathbb{R}^{d_{\text{out}}} \times \mathbb{R}^{d_{\text{out}}} \to \mathbb{R}.$$

The usual example is

$$\mathsf{loss}(\mathbf{v}, \mathbf{w}) := \frac{1}{d_{\text{out}}} \sum_{j=1}^{d_{\text{out}}} |v_j - w_j|^p,$$

for $p = 1$ (**mean absolute error**) or $p = 2$ (**mean squared error**).

## 1.4 Objective functions / Loss functions for classification

As we have seen, in the context of classification we shall need a loss function $\mathsf{loss} : \mathbb{R}^n \times [n] \to \mathbb{R}$. We can think of this as a function $\mathsf{loss} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ where we interpret the second argument as a one-hot vector

$$\mathsf{onehot}(i) = \mathbf{e}_i \in \mathbb{R}^n.$$

Interpreted as a probability measure, $\mathsf{onehot}(i)$ is the probability distribution with probability 1 on the $i$-th element and 0 elsewhere.

So, we are looking for a loss function

$$\mathsf{loss} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R},$$

or more specificially

$$\mathsf{loss} : \mathcal{P}_n \times \mathcal{P}_n \to \mathbb{R},$$

where $\mathcal{P}$ is the set of probability distributions on $[n]$.

Bis hier: Lecture 6. 4.5.23

Possible distances for discrete probability measures:

- $L^p$ **distances**; for $p \geq 1$,

$$d_{L^p}(P,Q) := \left( \sum_{i=1}^{n} |P(i) - Q(i)|^p \right)^{1/p}.$$

Note: this is an actual metric.

- **total-variation distance**;

$$d_{\mathsf{total}}(P,Q) := \sup_{A \subset [n]} |P(A) - Q(A)| = \frac{1}{2} d_{L^1}(P,Q).$$

Hence: this is an actual metric.

*Proof of the equality.* First, for any $A$ we have

$$P(A) - Q(A) = 1 - P(A^c) - (1 - Q(A^c)), = Q(A^c) - P(A^c).$$

Hence

$$|P(A) - Q(A)| = |Q(A^c) - P(A^c)|,$$

and hence

$$\begin{aligned} |P(A) - Q(A)| &= \frac{1}{2} \big( |P(A) - Q(A)| + |Q(A^c) - P(A^c)| \big) \\ &\leq \frac{1}{2} \sum_{i=1}^{n} |P(i) - Q(i)|. \end{aligned}$$

This shows

$$\sup_{A \subset [n]} |P(A) - Q(A)| \leq \frac{1}{2} \sum_{i=1}^{n} |P(i) - Q(i)|.$$

To show equality, define

$$A := \{ i \mid P(i) \geq Q(i) \}.$$

Then

$$\begin{aligned} |P(A) - Q(A)| &= P(A) - Q(A) = \sum_{i \in A} P(i) - Q(i) \\ |P(A^c) - Q(A^c)| &= Q(A^c) - P(A^c) = \sum_{i \in A^c} Q(i) - P(i). \end{aligned}$$

Hence

$$|P(A) - Q(A)| = \frac{1}{2} \big( |P(A) - Q(A)| + |P(A^c) - Q(A^c)| \big) = \frac{1}{2} \sum_{i} |P(i) - Q(i)|.$$

□

■ **Kullback-Leibler divergence**[6]

$$D_{\mathrm{KL}}(\mathbf{p},\mathbf{q}) := D_{\mathrm{KL}}(\mathbf{p}||\mathbf{q}) := \sum_{i=1}^{n} \mathbf{p}_i \log\left(\frac{\mathbf{p}_i}{\mathbf{q}_i}\right),$$

where $0\log(0/0) := 0$. Or, as probability measures,

$$D_{\mathrm{KL}}(P,Q) = D_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P}\left[\log\frac{P(x)}{Q(x)}\right].$$

$D_{\mathrm{KL}}$ is *not* a metric.

Interpretation: the Kullback-Leibler divergence is the expected number of extra bits needed to encode a sample from **p** when using a code optimized for **q** instead of a code optimized for **p**.

Properties:

–
$$D_{\mathrm{KL}}(P||Q) = +\infty \Leftrightarrow \exists i : P(i) > 0, Q(i) = 0.$$

–
$$D_{\mathrm{KL}}(P||Q) \geq 0, \quad D_{\mathrm{KL}}(P||Q) = 0 \iff P = Q.$$

– (Convexity)
$$D_{\mathrm{KL}}\left(\lambda P + (1-\lambda)P'||\lambda Q + (1-\lambda)Q'\right) \leq \lambda D_{\mathrm{KL}}\left(P||Q\right) + (1-\lambda)D_{\mathrm{KL}}\left(P||Q\right)$$

*Proof.*

1. Immediate.
2. First, the function

$$\phi(x) := x\log(x),$$

is convex on $(0,\infty)$. Indeed,

$$\phi'(x) = \log(x) + 1, \qquad \phi''(x) = \frac{1}{x} + 1 > 0.$$

Assume that $D_{\mathrm{KL}}(P||Q) \neq +\infty$, i.e. for all $i$ we have $P(i) > 0 \Rightarrow Q(i) > 0$. Then, using Jensen's inequality

$$\sum_i p_i \log\left(\frac{p_i}{q_i}\right) = \sum_i q_i \frac{p_i}{q_i}\log\left(\frac{p_i}{q_i}\right) \geq \phi\left(\sum_i q_i\frac{p_i}{q_i}\right) = \phi\left(\sum_i p_i\right) = \phi(1) = 0.$$

[6] On nomenclature: a **divergence** in statistics is a binary map on a (statistical) manifold, satisfying certain properties.

□

We used the following lemma:

**Lemma 1.9** (Jensen's inequality)**.** *Let $\phi : \mathbb{R}^p \to \mathbb{R}$ be* ***convex****, that is, for every* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^p, \lambda \in [0,1]$,

$$\phi(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda\phi(\mathbf{x}) + (1-\lambda)\phi(\mathbf{y}).$$

*Let* $p_i \in [0,1], i = 1, \dots, n, \mathbf{x}_i \in \mathbb{R}^p$ *with* $\sum_i p_i = 1$. *Then*

$$\phi\left(\sum_i p_i x_i\right) \leq \sum_i p_i \phi(x_i).$$

*Proof.* Induction on $n$. For $n = 1$ it is trivially true.

Let it be true for $n$. If $p_{n+1} = 1$ there is nothing to show. Assume $p_{n+1} < 1$. Then

$$\begin{aligned}
\phi\left(\sum_{i=1}^{n+1} p_i x_i\right) &= \phi\left((1-p_{n+1})\sum_{i=1}^{n} \frac{p_i}{1-p_{n+1}} x_i + p_{n+1}x_{n+1}\right) \\
&\leq (1-p_n)\phi\left(\sum_{i=1}^{n} \frac{p_i}{1-p_{n+1}} x_i\right) + p_{n+1}\phi\left(x_{n+1}\right) \\
&\leq (1-p_n)\sum_{i=1}^{n} \frac{p_i}{1-p_{n+1}} \phi\left(x_i\right) + p_{n+1}\phi\left(x_{n+1}\right) \\
&= \sum_{i=1}^{n+1} p_i \phi\left(x_i\right).
\end{aligned}$$

□

- **cross-entropy**;

$$\mathrm{CE}(\mathbf{p}, \mathbf{q}) := -\sum_{i=1}^{n} \mathbf{p}_i \log(\mathbf{q}_i) = D_{\mathrm{KL}}(\mathbf{p}||\mathbf{q}) - \sum_{i=1}^{n} \mathbf{p}_i \log(\mathbf{p}_i).$$

This is *not* a metric.

Properties: From the properties of the KL divergence we get

1. $\mathrm{CE}(\mathbf{p}, \mathbf{q}) \geq -\sum_i p_i \log(p_i) \geq 0$.
2. $\mathrm{CE}(\mathbf{p}, \mathbf{q}) = -\sum_i p_i \log(p_i) \iff \mathbf{p} = \mathbf{q}$.

Interpretation: the cross-entropy is the expected number of bits needed to encode a sample from $P$ using a code $Q$.

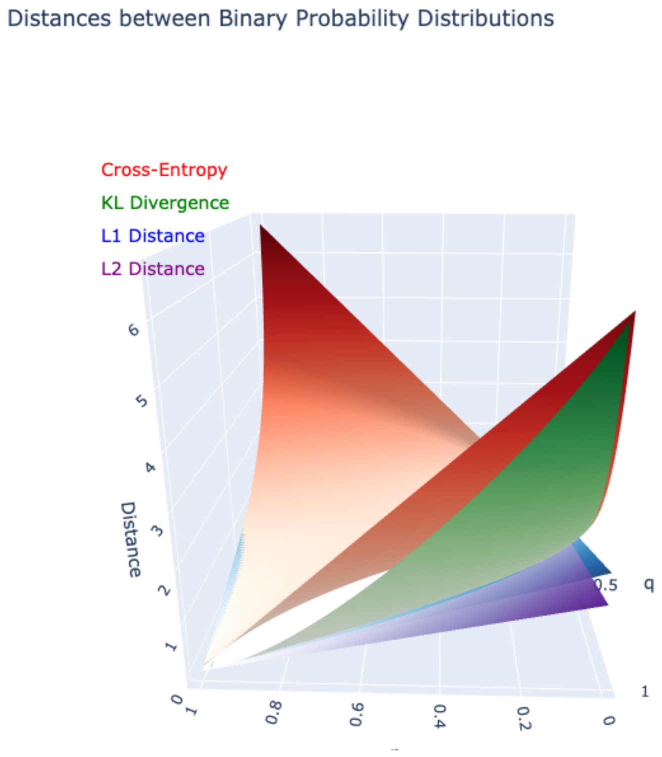


Figure 3: *Comparing several distances on binary probability distributions.*

**Remark 1.10.** *By the first point, in a classification task, it ’only makes sense’ to plug the one-hot vector, which is the target, in the $P$-slot. (If $Q$ is one-hot, then the only case for which $\mathrm{D}_{\mathrm{KL}}(P, Q)$ is finite, is $P = Q$.)*

*Then*

$$D_{\mathrm{KL}}(P||Q) = \sum_i p_i \log(p_i) + \mathrm{CE}(P, Q).$$

*Hence, for a fixed $P$, optimizing the cross-entropy $\mathrm{CE}(P, Q)$ is equivalent to optimizing the KL-divergence $\mathrm{D}_{\mathrm{KL}}(P||Q)$ (over $Q$). If $\mathbf{p}$ is hot on position $\hat{\imath}$ then*

$$\mathrm{CE}(\mathbf{p}, \mathbf{q}) = -\log(\mathbf{q}_{\hat{\imath}}).$$

## 1.5 (Stochastic) gradient descent

An abstract optimization problem is of the form[7]

$$\text{argmin}_\theta C(\theta) := \{\theta \in \mathbb{R}^p : C(\theta) \leq C(\theta') \text{ for all } \theta' \in \mathbb{R}^p\},$$

for some **objective** or **cost** function $C : \mathbb{R}^p \to \mathbb{R}$. A simple approach to solving it is: start with some initial parameters $\theta_0$, determine in which direction in parameter space we should move to decrease the cost, and then move in that direction.

We know that, infinitesimally, the negative of the **gradient**

$$\nabla_\theta C(\theta) = \begin{bmatrix} \partial\theta_1 C \\ \vdots \\ \partial\theta_p C \end{bmatrix},$$

points in the direction of steepest descent. Indeed, for a vector $\mathbf{x} \in \mathbb{R}^p, \mathbf{x} \neq 0$, and for every $\mathbf{y} \in \mathbb{R}^p$ with $||\mathbf{y}||= 1$, by the Cauchy-Schwarz inequality we have

$$\langle \mathbf{y}, \mathbf{x} \rangle \geq -||\mathbf{y}||||\mathbf{x}||= -||\mathbf{x}||= \langle \mathbf{x}, -\frac{\mathbf{x}}{||\mathbf{x}||} \rangle.$$

This yields the method of **gradient descent**:

$$\theta_{t+1} := \theta_t - \eta \nabla_\theta C(\theta_t).$$

Here $\eta$ is the **learning rate**, which is a hyperparameter usually of the order of $10^{-3}$ or $10^{-4}$.

We will apply gradient descent in settings where there are absolutely no theoretical guarantees (to date) of convergence. In more restricted settings, however, we can prove that gradient descent converges.

**Theorem 1.11.** *Assume that $C : \mathbb{R}^p \to \mathbb{R}$ is **convex**, Assume that $\min C$ is attained, i.e. $\text{argmin}\, C \neq \emptyset$. Assume also that $C$ is continuously differentiable and that $\nabla C$ is Lipschitz continuous with Lipschitz constant $L$. If the learning rate $\eta$ is chosen smaller than $\frac{1}{L}$, then* [8]

$$C(\theta_t) - \min C \leq \frac{||\theta_0 - \theta^*||^2}{2\eta t}.$$

*In particular,* from any starting point*, gradient descent converges to a global minimum.*

For the proof, see the appendix.

There are a plethora of variants of vanilla gradient descent. A classical modification is the introduction of **momentum**. The updates then look as follows. For fixed $m \in (0, 1]$ [9]

[7] Note that argmin is a *set*. We will abuse notation and write argmin also for an *element* of this set.

[8] References: [GG23, Theorem 3.4]

and $\eta \in (0, \infty)$,

$$\begin{aligned} v_{t+1} &= m \cdot v_t - \gamma \cdot \nabla_\theta C(\theta_t) \\ \theta_{t+1} &= \theta_t + v_t, \end{aligned}$$

or with a different parametrization, $\beta \in (0, 1)$,

$$\begin{aligned} v_{t+1} &= \beta \cdot v_t + (1 - \beta) \cdot \nabla_\theta C(\theta_t) \\ \theta_{t+1} &= \theta_t - \eta \cdot v_t. \end{aligned}$$

A more recent modification is **Adam**. Let [10]

$$g_t := \nabla C(\theta_t),$$

fix some initial $x_0 \in \mathbb{R}^p$. Fix $\beta_1, \beta_2 \in [0, 1), \eta, \epsilon > 0$. Consider the "moment updates"

$$v_t := \beta_1 v_{t-1} + (1 - \beta_1) g_t, \qquad s_t := \beta_2 s_{t-1} + (1 - \beta_2) g_t^2,$$

Define the "bias-corrected moments" as

$$\hat{v}_t := \frac{v_t}{1 - (\beta_1)^t}, \qquad \hat{s}_t := \frac{s_t}{1 - (\beta_2)^t}.$$

Then, the final update is

$$\theta_{t+1} := \theta_t - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \hat{v}_t.$$

## 1.6 Various other ingredients (Regularization, standardization, normalization, skip connections, ..)

### Dropout

**Dropout** is a regularization technique used in neural networks to prevent overfitting. The technique involves randomly ”dropping out” (i.e., temporarily removing) a number of output features of the layer *during training*. For each training sample, each neuron has a probability of being temporarily dropped out, meaning it will not contribute to the forward pass nor participate in backpropagation. This means that the model is forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

Here's how you might implement a dropout layer in a neural network using TensorFlow:

[9] References: https://distill.pub/2017/momentum/, https://www.cs.mcgill.ca/~siamak/COMP551/slides/7-gradient-descent_short.pdf

[10] References: [KB], [Cal20, Section 4.7]

```python
import tensorflow as tf

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

In this example, the `Dropout` layer will randomly set 20% of the input units to 0 at each update during training time, which helps prevent overfitting.

### Skip connections

**Skip connections**, also known as **shortcut connections** or **residual connections**, are a type of architecture used in neural networks that allow for the direct propagation of information from earlier layers to later layers without going through all the intermediate layers.

In a traditional feedforward neural network, information from one layer is passed to the next layer, and so on. This means that information from the first layer must pass through all intermediate layers before it reaches the final layer.

However, in very deep networks, this approach can cause two significant problems: *vanishing gradients* and *representational bottlenecks*. The vanishing gradient problem occurs when gradients are backpropagated through the network and become increasingly small, making it difficult for the network to learn. Representational bottlenecks occur when the dimensionality of the layers is not enough to encode the necessary information about the input.

Skip connections address these issues by allowing information to bypass some layers. This is typically achieved by adding the output of an earlier layer to the output of a later layer, a concept known as identity shortcut connection introduced in the ResNet model (Residual Network). This direct path has the effect of enabling the backpropagation of gradients through the network without the risk of them vanishing, as they can be directly propagated back through the shortcut connections.

Skip connections have another beneficial effect: they encourage the network to learn residual mappings. A residual mapping is the difference (or residual) between the input and the desired output. The idea is that it should be easier to optimize the network to produce a residual of zero (i.e., to produce an output identical to the input) than to optimize the network to produce any particular non-zero residual. This can significantly improve the network's learning capability, especially in the case of very deep networks.

An example from `https://github.com/keras-team/keras/blob/v2.12.0/keras/applications/resnet.py`:

```
# Shortcut is either identity or 1x1 conv
if conv_shortcut:
  shortcut = layers.Conv2D(
      4 * filters, 1, strides=stride, name=name + "_0_conv"
  )(x)
  shortcut = layers.BatchNormalization(
      axis=bn_axis, epsilon=1.001e-5, name=name + "_0_bn"
  )(shortcut)
else:
  shortcut = x

# The main path
x = layers.Conv2D(filters, 1, strides=stride, name=name + "
    _1_conv")(x)
...
x = layers.BatchNormalization(
  axis=bn_axis, epsilon=1.001e-5, name=name + "_3_bn"
)(x)

# Add shortcut value to main path
x = layers.Add(name=name + "_add")([shortcut, x])
x = layers.Activation("relu", name=name + "_out")(x)
```

### Batch normalization

Each dimension of the non-batch dimensions is normalized (over the batch dimension).[11] For a order-1 tensor $\mathbf{x}$,

$$\begin{aligned} \mathsf{mean}(\mathbf{x}) &:= \frac{1}{|\mathbf{x}|}\sum_{i=1}^{|\mathbf{x}|} \mathbf{x}_i \\ \mathsf{var}(\mathbf{x}) &:= \frac{1}{|\mathbf{x}|}\sum_{i=1}^{|\mathbf{x}|} (\mathbf{x}_i - \mathsf{mean}(\mathbf{x}))^2. \end{aligned}$$

In the example of data format $\mathbb{R}^{|n|} \otimes \mathbb{R}^{|c|}$,

$$\begin{aligned} \mathsf{BatchNormalization} : \mathbb{R}^{|b|} \otimes \mathbb{R}^{|n|} \otimes \mathbb{R}^{|c|} &\to \mathbb{R}^{|b|} \otimes \mathbb{R}^{|n|} \otimes \mathbb{R}^{|c|} \\ \mathsf{BatchNormalization}(T)_{b,n,c} &:= \frac{T_{b,n,c} - \mathsf{mean}(T_{\bullet,n,c})}{\sqrt{\mathsf{var}(T_{\bullet,n,c})} + \varepsilon} \end{aligned}$$

Here, $\varepsilon > 0$ is a small (fixed) number, added for numerical stability.

[11] References: [IS15], [GBC16, Section 8.7.1]

This is at training time. At test time the mean and variance are replaced by running averages collected at training.

### Layer normalization

Similar to batch normalization, but now normalizing a fixed batched, i.e. [12]

$$\mathsf{LayerNormalization}(T)_{b,n,c} := \gamma_{n,c} \frac{T_{b,n,c} - \mathsf{mean}(T_{b,\bullet,\bullet})}{\sqrt{\mathsf{var}(T_{b,\bullet,\bullet})} + \varepsilon}. + \beta_{n,c}$$

Here, $\gamma, \beta$ are learned parameters. Unlike batch normalization, layer normalization performs exactly the same computation at training and test times.

[12] References: [BKH16]

# 2 Attention

## 2.1 Softmax

[13]

Given a vector $\mathbf{x} \in \mathbb{R}^n$, the softmax function is defined as

$$\operatorname{softmax} \mathbf{x} := \frac{1}{\sum_{j=1}^n \exp(x_j)} \begin{pmatrix} \exp(x_1) \\ \exp(x_2) \\ \dots \\ \exp(x_n) \end{pmatrix}$$

**Lemma 2.1.** *Let $\mathbf{x} \in \mathbb{R}^n$.*

*1.*

$$(\operatorname{softmax} \mathbf{x})_i \in (0, 1] \qquad \forall i \in [n].$$

*2.*

$$\sum_{i=1}^n (\operatorname{softmax} \mathbf{x})_i = 1.$$

*3. For all $c \in \mathbb{R}$*

$$\operatorname{softmax}(\mathbf{x} + c) = \operatorname{softmax}(\mathbf{x}).$$

*4. Assume that $\mathbf{x}$ has a unique maximal value at position $\hat{\imath}$. Then*

$$\operatorname{softmax}(\lambda \mathbf{x}) \to_{\lambda \to \infty} \mathbf{e}_{\hat{\imath}}.$$

**Remark 2.2.** *Point 1.,2. imply that the softmax yields a* probability measure *on $[n]$. Point 4. shows that this probability measure is concentrated at the maximum position of $\mathbf{x}$. Hence, the name* softargmax *would be more fitting.*

*Point 3. shows that* softmax *only sees the* relative *size of the inputs.*

*Proof.*

1. This follows from the fact that $\exp : \mathbb{R} \to (0, \infty)$.
2. $$\sum_{i=1}^n (\operatorname{softmax} \mathbf{x})_i = \sum_{i=1}^n \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)} = \frac{\sum_{i=1}^n \exp(x_i)}{\sum_{j=1}^n \exp(x_j)} = 1.$$
3. Direct calculation.

[13] References: [VSP+17]

4. For $i \neq \hat{\imath}$ we have

$$x_i - x_{\hat{\imath}} < 0.$$

Hence

$$\frac{\exp(\lambda x_i)}{\sum_{j=1}^n \exp(\lambda x_j)} \leq \frac{\exp(\lambda x_i)}{\exp(\lambda x_{\hat{\imath}})} = \exp(\lambda(x_i - x_{\hat{\imath}})) \xrightarrow[\lambda\to\infty]{} 0.$$

Hence

$$\operatorname{softmax}(\lambda \mathbf{x})_i \xrightarrow[\lambda\to\infty]{} 0.$$

By ii. we then get

$$\operatorname{softmax}(\lambda \mathbf{x})_{\hat{\imath}} \xrightarrow[\lambda\to\infty]{} 1.$$

□

Bis hier: Lecture 7. 8.5.23

We shall need the softmax function applied to only certain axis of larger tensors, mostly the last axis. For example, for $T \in \mathbb{R}^{b\times n\times c}$,

$$\operatorname{softmax}_{\mathsf{axis}=-1}(T)_{b,n,c} := \frac{\exp(T_{b,n,c})}{\sum_{c'=1}^{c} \exp(T_{b,n,c'})}$$

## 2.2 Attention

**Example 2.3.**

***How to model a database/lookup table as a differential function?***

*Let there be $n$ many "values" $\mathbf{v}_i \in \mathbb{R}^{|k|}, i = 1, \ldots, n$, for example stored in a python array. Given an integer $i = 1, \ldots, n$ we can of course pick out $\mathbf{v}_i$, by indexing the array. How can we turn this (discrete) operation into a differentiable one?*

*We assign to each value $\mathbf{v}_i$ a "key", which we - in this example - choose to be $\mathbf{e}_i \in \mathbb{R}^n$ the $i$-th standard basis vector.*

*PICTURE IN LECTURE*

*We can now "query" these keys with a vector $q \in \mathbb{R}^n$,*

$$\langle q, \mathbf{e}_i\rangle$$

*For example, with $q = \mathbf{e}_3$,*

$$\langle q, \mathbf{e}_i\rangle = \delta_{3,i}.$$

*It then makes sense to form a linear combination of the values, depending on how well the corresponding key has matched:*

$$\mathsf{lookup}(q) := \sum_{i=1}^{n} \langle q, \mathbf{e}_i \rangle \ \mathbf{v}_i.$$

*For example,*

$$\mathsf{lookup}(\mathbf{e}_3) = \mathbf{v}_3, \quad \mathsf{lookup}(\mathbf{e}_3 + 2\mathbf{e}_5) = \mathbf{v}_3 + 2\mathbf{v}_5.$$

*Note that* lookup *is a differentiable function.*

*Several observations on*

- lookup *is linear in each of its argument,*
- *The coefficients $\langle q, \mathbf{e}_i \rangle$ can take any value in $\mathbb{R}$.*

*It is often beneficial to restrict the coefficients to the interval $[0, 1]$ and moreover to make them sum to $1$, i.e. have them be a probability distribution.*

*As we have seen, this can be achieved by the softmax function. This leads to*

$$\mathsf{lookup}(q) := \sum_{i=1}^{n} \operatorname{softmax}\left( (\langle q, \mathbf{e}_j \rangle)_{j=1,\dots,n} \right)_i \ \mathbf{v}_i.$$

We now define the general attention layer, with learnable weights.

Given

$$Q \in \mathbb{R}^{n_q \times d_q}, K \in \mathbb{R}^{n_k \times d_k}, V \in \mathbb{R}^{n_k \times d_v}$$

and

$$W^Q \in \mathbb{R}^{d_q \times k}, W^K \in \mathbb{R}^{d_k \times k}, W^V \in \mathbb{R}^{d_v \times v},$$

define

$$\mathsf{Attention}_{W^Q, W^K, W^V}(Q, K, V) := \operatorname{softmax}_{\mathsf{axis}=-1}\left( (QW^Q)(KW^K)^\top \right) VW^V.$$

Note that attention is *not* of the form (1), since the nonlinear function (the softmax) does *not* act entrywise.

If $Q = V = K$, then this layer is also called a **self-attention** layer.

The canonical example comes from sequential data $X \in \mathbb{R}^{n \times d}$, where the first dimension is thought of as time, and $W^Q \in \mathbb{R}^{d \times k}, W^K \in \mathbb{R}^{d \times k}, W^V \in \mathbb{R}^{d \times v}$. Then

$$\mathsf{Attention}_{W^Q, W^K, W^V}(X, X, X) \in \mathbb{R}^{n \times v},$$

which can again be fed into another self-attention layer (now with $v$ taking the role of $d$ ..).

**Remark 2.4.** *Note that the calculation of*

$$\mathsf{Attention}_{W^Q,W^K,W^V}(X,X,X),$$

*has complexity $\mathcal{O}(n^2)$. This is the **quadratic-cost problem of self-attention**. It prohibits the use of (vanilla) self-attention for long sequences. We will see solutions to this problem in* **??**.

In practice, a slight modification called Multi Head Attention is used, for some $|h|\geq 1$, and $A_{\mathsf{projection}}, \mathbf{b}_{\mathsf{projection}}$ of appropriate dimensions:

$$\mathsf{MultiHeadAttention}(Q,K,V) := A_{\mathsf{projection}} \cdot \begin{pmatrix} \mathsf{Attention}_{W_1^Q,W_1^K,W_1^V}(Q,K,V) \\ \cdots \\ \mathsf{Attention}_{W_{|h|}^Q,W_{|h|}^K,W_{|h|}^V}(Q,K,V) \end{pmatrix} + \mathbf{b}_{\mathsf{projection}}$$

## References

[BD81] Bruno Brosowski and Frank Deutsch. An elementary proof of the stone-weierstrass theorem. *Proceedings of the American Mathematical Society*, 81(1):89–92, 1981.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[Cal20] Ovidiu Calin. *Deep learning architectures.* Springer, 2020.

[Fou22] Simon Foucart. *Mathematical Pictures at a Data Science Exhibition.* Cambridge University Press, 2022.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[GG23] Guillaume Garrigos and Robert M Gower. Handbook of convergence theorems for (stochastic) gradient methods. *arXiv preprint arXiv:2301.11235*, 2023.

[HAB19] Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 41–50, 2019.

[HS17] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[KB] DP Kingma and JL Ba. Adam: a method for stochastic optimization. proceedings of the 3rd international conference on learning representations, iclr 2015—conference track proceedings. international conference on learning representations, iclr; 2015. *CA, USA*.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[Lan12] Serge Lang. *Real and functional analysis*, volume 142. Springer Science & Business Media, 2012.

[Loo13] Lynn H Loomis. *Introduction to abstract harmonic analysis*. Courier Corporation, 2013.

[MPCB14] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

[NZL20] Gregory Naitzat, Andrey Zhitnikov, and Lek-Heng Lim. Topology of deep neural networks. *The Journal of Machine Learning Research*, 21(1):7503–7542, 2020.

[Tel16] Matus Telgarsky. Benefits of depth in neural networks. In *Conference on learning theory*, pages 1517–1539. PMLR, 2016.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.