Introduction to the Mathematics of Deep Learning

Davide Murari

Department of Applied Mathematics and Theoretical Physics
University of Cambridge

davidemurari.com/cism

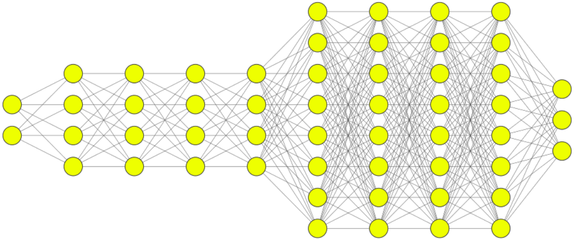
dm2011@cam.ac.uk

Outline

The building blocks of neural networks

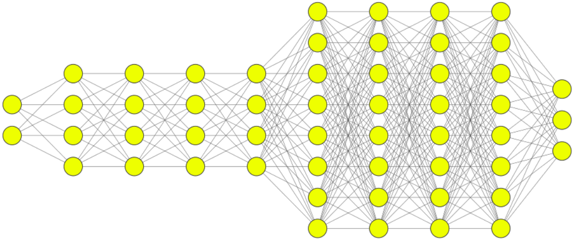
What is a neural network mathematically

- Neural networks are typically visualised as something like this



What is a neural network mathematically

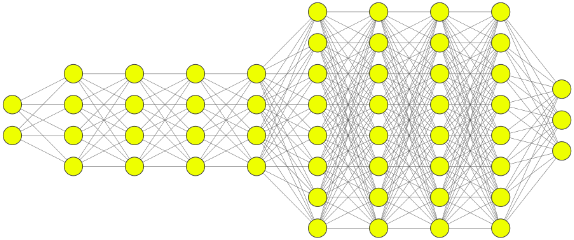
- Neural networks are typically visualised as something like this



- Mathematically, a neural network (NN) is a **parametric map** $\mathcal{N}_\theta : \mathbb{R}^c \to \mathbb{R}^d$, usually defined by composing L functions, called **layers**, as $\mathcal{N}_\theta = F_{\theta_L} \circ \ldots \circ F_{\theta_1}$, $F_{\theta_i} : \mathbb{R}^{c_i} \to \mathbb{R}^{c_{i+1}}$, $c_1 = c$, $c_{L+1} = d$. Each component of each layer is called **neuron**.

What is a neural network mathematically

- Neural networks are typically visualised as something like this



- Mathematically, a neural network (NN) is a **parametric map** $\mathcal{N}_\theta : \mathbb{R}^c \to \mathbb{R}^d$, usually defined by composing L functions, called **layers**, as $\mathcal{N}_\theta = F_{\theta_L} \circ \ldots \circ F_{\theta_1}$, $F_{\theta_i} : \mathbb{R}^{c_i} \to \mathbb{R}^{c_{i+1}}$, $c_1 = c$, $c_{L+1} = d$. Each component of each layer is called **neuron**.

- The parametrisation strategy behind $\mathcal{N}_\theta$ is defined by the so-called **neural network architecture**.

The simplest type of layer

- It is common practice to define layers by alternating affine maps with non-linear functions applied entrywise:

$$F_{\theta_i}(\mathbf{x}) = \Sigma \circ A_i(\mathbf{x}),\ \Sigma(\mathbf{x}) := \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_{c_{i+1}}) \end{bmatrix}, \tag{1}$$

$$A_i : \mathbb{R}^{c_i} \to \mathbb{R}^{c_{i+1}},\ \Sigma : \mathbb{R}^{c_{i+1}} \to \mathbb{R}^{c_{i+1}},\ \sigma : \mathbb{R} \to \mathbb{R}.$$

We will use σ both for the scalar function and for the vector function.

The simplest type of layer

- It is common practice to define layers by alternating affine maps with non-linear functions applied entrywise:

$$F_{\theta_i}(\mathbf{x}) = \Sigma \circ A_i(\mathbf{x}),\ \Sigma(\mathbf{x}) := \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_{c_{i+1}}) \end{bmatrix}, \tag{1}$$

$$A_i : \mathbb{R}^{c_i} \to \mathbb{R}^{c_{i+1}},\ \Sigma : \mathbb{R}^{c_{i+1}} \to \mathbb{R}^{c_{i+1}},\ \sigma : \mathbb{R} \to \mathbb{R}.$$

  We will use σ both for the scalar function and for the vector function.

- σ is called **activation function**.

The simplest type of layer

- It is common practice to define layers by alternating affine maps with non-linear functions applied entrywise:

$$F_{\theta_i}(\mathbf{x}) = \Sigma \circ A_i(\mathbf{x}), \;\; \Sigma(\mathbf{x}) := \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_{c_{i+1}}) \end{bmatrix}, \tag{1}$$

$$A_i : \mathbb{R}^{c_i} \to \mathbb{R}^{c_{i+1}},\; \Sigma : \mathbb{R}^{c_{i+1}} \to \mathbb{R}^{c_{i+1}},\; \sigma : \mathbb{R} \to \mathbb{R}.$$

  We will use σ both for the scalar function and for the vector function.

- σ is called **activation function**.

- Depending on how A_i is defined, we can get different types of neural networks, such as Fully Connected Networks, Convolutional Networks, Graph Neural Networks, and more.

- We can modify (1) to get architectures such as ResNets, U-Nets, Transformers, and more.

Deep VS Shallow Networks

- A NN $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^c$ is **shallow** if it has a **single hidden layer**, so generally this means that it can be written as

$$\mathcal{N}_\theta(\mathbf{x}) = A_1\sigma(A_0\mathbf{x} + \mathbf{b}),\ A_0 \in \mathbb{R}^{h\times d}\ A_1 \in \mathbb{R}^{d\times h},\ \mathbf{b} \in \mathbb{R}^h,\ h \in \mathbb{N}.$$

- $\mathcal{N}_\theta$ is **deep** if it is not shallow, so if it has $L > 1$ layers. If the layers are defined as seen below, this means that

$$\mathcal{N}_\theta(\mathbf{x}) = A_L \circ \sigma \circ A_{L-1} \circ \ldots \circ A_1 \circ \sigma \circ A_0(\mathbf{x}). \tag{2}$$

- If the affine layers are defined by unconstrained/dense matrices, we call $\mathcal{N}_\theta$ in (2) a **Multi-Layer Perceptron** (**MLP**).

Finding the weights of a NN

- The weights θ of $\mathcal{N}_\theta$ can be found by solving a suitable optimisation problem. This optimisation process is called **network training**.

Finding the weights of a NN

- The weights θ of $\mathcal{N}_\theta$ can be found by solving a suitable optimisation problem. This optimisation process is called **network training**.
- The **loss function** to minimise is defined thanks to the data one has available, or thanks to properties we would like the approximation to satisfy.

Finding the weights of a NN

- The weights θ of $\mathcal{N}_\theta$ can be found by solving a suitable optimisation problem. This optimisation process is called **network training**.
- The **loss function** to minimise is defined thanks to the data one has available, or thanks to properties we would like the approximation to satisfy.
- After minimising the loss function, we hopefully have a good set of parameters θ^* and we can use $\mathcal{N}_{\theta^*}$ to make new predictions, for unseen inputs.

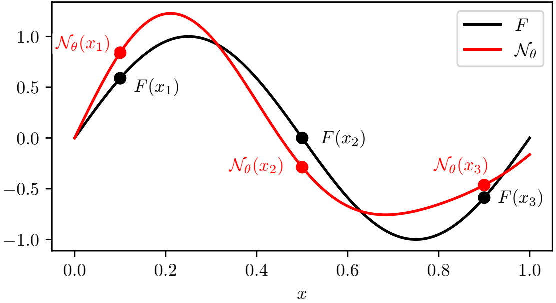
Finding the weights of a NN

- The weights θ of $\mathcal{N}_\theta$ can be found by solving a suitable optimisation problem. This optimisation process is called **network training**.
- The **loss function** to minimise is defined thanks to the data one has available, or thanks to properties we would like the approximation to satisfy.
- After minimising the loss function, we hopefully have a good set of parameters θ^* and we can use $\mathcal{N}_{\theta^*}$ to make new predictions, for unseen inputs.
- We now briefly describe the common loss functions used for **regression tasks**, and **classification tasks**.

Mean-Squared Error Loss Function for regression

Given the dataset $\{(\mathbf{x}_i, \mathbf{y}_i = F(\mathbf{x}_i))\}_{i=1}^N$, $x_1, \dots, x_N \in \Omega \subset \mathbb{R}^d$, to approximate $F : \mathbb{R}^d \to \mathbb{R}^c$ over Ω with a neural network $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^c$, we can minimise the **Mean-Squared Error Loss function** defined as

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^N \|\mathcal{N}_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2 .$$



Cross-Entropy Loss Function for classification

- **Dataset:** $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \Omega \subset \mathbb{R}^d$, and $y_i \in \mathcal{Y} := \{1, \dots, K\}$. y_i is the **class index** (the label) of $\mathbf{x}_i$; e.g. $y_i = 3$ means $\mathbf{x}_i$ belongs to the third class (e.g. cats).

Cross-Entropy Loss Function for classification

- **Dataset:** $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \Omega \subset \mathbb{R}^d$, and $y_i \in \mathcal{Y} := \{1, \dots, K\}$. y_i is the **class index** (the label) of $\mathbf{x}_i$; e.g. $y_i = 3$ means $\mathbf{x}_i$ belongs to the third class (e.g. cats).
- **Target function:** $F : \mathbb{R}^d \to \mathcal{Y}$. For convenience, define its one-hot vector $e(y_i) \in \{0,1\}^K$ with $[e(y_i)]_k = \delta_{y_i,k}$, $k = 1, \dots, K$, $i = 1, \dots, N$.

Cross-Entropy Loss Function for classification

- **Dataset:** $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \Omega \subset \mathbb{R}^d$, and $y_i \in \mathcal{Y} := \{1, \dots, K\}$. y_i is the **class index** (the label) of $\mathbf{x}_i$; e.g. $y_i = 3$ means $\mathbf{x}_i$ belongs to the third class (e.g. cats).
- **Target function:** $F : \mathbb{R}^d \to \mathcal{Y}$. For convenience, define its one-hot vector $e(y_i) \in \{0,1\}^K$ with $[e(y_i)]_k = \delta_{y_i,k}$, $k = 1, ..., K$, $i = 1, ..., N$.
- **Approximation strategy:** We approximate F with a neural network $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^K$ producing logits $\mathcal{N}_\theta(\mathbf{x}) \in \mathbb{R}^K$ and class probabilities via the $\mathrm{softmax}$ function:

$$p_\theta(\mathbf{x}) = \mathrm{softmax}(\mathcal{N}_\theta(\mathbf{x})),\ [p_\theta(\mathbf{x})]_k = \frac{\exp([\mathcal{N}_\theta(\mathbf{x})]_k)}{\sum_{j=1}^K \exp([\mathcal{N}_\theta(\mathbf{x})]_j)} \geq 0,\ \sum_{k=1}^K [p_\theta(\mathbf{x})]_k = 1.$$

Cross-Entropy Loss Function for classification

- **Dataset:** $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, $\mathbf{x}_i \in \Omega \subset \mathbb{R}^d$, and $y_i \in \mathcal{Y} := \{1, \ldots, K\}$. y_i is the **class index** (the label) of $\mathbf{x}_i$; e.g. $y_i = 3$ means $\mathbf{x}_i$ belongs to the third class (e.g. cats).

- **Target function:** $F : \mathbb{R}^d \to \mathcal{Y}$. For convenience, define its one-hot vector $e(y_i) \in \{0, 1\}^K$ with $[e(y_i)]_k = \delta_{y_i,k}$, $k = 1, ..., K$, $i = 1, ..., N$.

- **Approximation strategy:** We approximate F with a neural network $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^K$ producing logits $\mathcal{N}_\theta(\mathbf{x}) \in \mathbb{R}^K$ and class probabilities via the $\mathrm{softmax}$ function:

$$p_\theta(\mathbf{x}) = \mathrm{softmax}(\mathcal{N}_\theta(\mathbf{x})), \; [p_\theta(\mathbf{x})]_k = \frac{\exp([\mathcal{N}_\theta(\mathbf{x})]_k)}{\sum_{j=1}^K \exp([\mathcal{N}_\theta(\mathbf{x})]_j)} \geq 0, \; \sum_{k=1}^K [p_\theta(\mathbf{x})]_k = 1.$$

- The (multi-class) **Cross-Entropy Loss** is

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K [e(y_i)]_k \log[p_\theta(\mathbf{x}_i)]_k$$

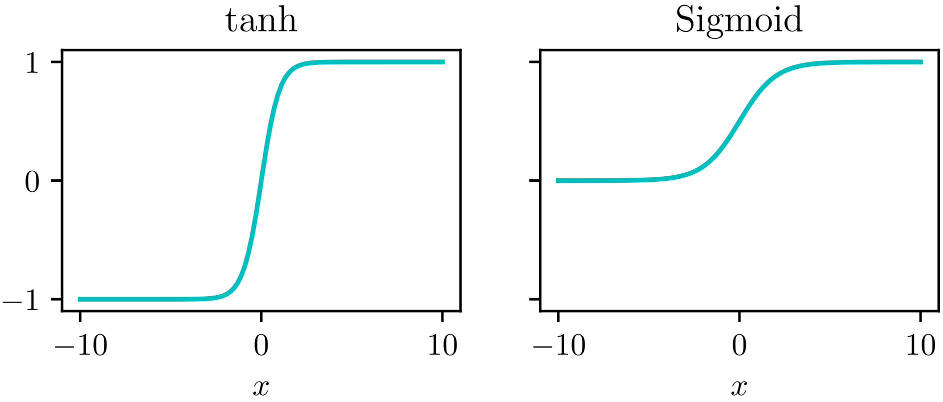
Activation functions

Main properties of the most popular activation functions

- In principle, most functions $\sigma : \mathbb{R} \to \mathbb{R}$ can be used as an activation function.
- Furthermore, one could also change the activation function from neuron to neuron and layer to layer. This is not so common, though.
- Most (but not all) of the commonly used activation functions satisfy the following properties:
  1. **Non-linear**
  2. **Not polynomials**
  3. Non-decreasing
  4. Lipschitz continuous

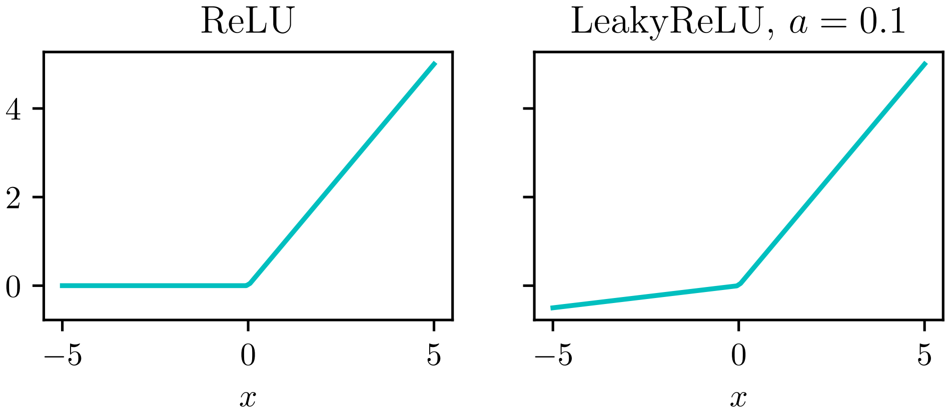
Activations with bounded range

- An important class of activation functions, called **sigmoidal** have a bounded range, i.e., $\sigma(\mathbb{R})$ is bounded. Examples are $\sigma(x) = \tanh(x)$ and $\sigma(x) = 1/(1 + e^{-x})$. These functions are said to **saturate**, which could be a problem for gradient stability.



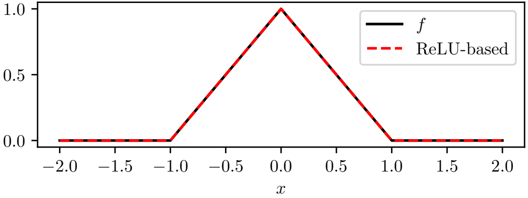
Activations with unbounded range

- There are many other activation functions with unbounded range, such as the popular $\sigma(x) = \mathrm{ReLU}(x) = \max\{x, 0\}$ and $\sigma(x) = \mathrm{LeakyReLU}(x) = \max\{x, ax\}$, $a \in (0, 1)$. ReLU is flat in half of the line, and this could also lead to gradient instabilities.



Some functions representable with ReLU

- $x = \mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$, $x^p = \mathrm{ReLU}^p(x) + (-1)^p\mathrm{ReLU}(-x)$, $p \in \mathbb{N}$,
- $|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$,
- $\max\{x, y\} = x + \mathrm{ReLU}(y - x) = y + \mathrm{ReLU}(x - y)$
- $\min\{x, y\} = x - \mathrm{ReLU}(x - y) = y - \mathrm{ReLU}(y - x)$
- Hat functions such as $f(x) = \max\{0, |1 - |x||\}$ can also be represented: $f(x) = \mathrm{ReLU}(x - 1) - 2\mathrm{ReLU}(x) + \mathrm{ReLU}(x + 1)$



How do we train neural networks?

The need for first-order optimisation methods

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

The need for first-order optimisation methods

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D},\ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...),\ k = 1, ..., \text{epochs}.$$

The need for first-order optimisation methods

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D},\ \ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...),\ k = 1, ..., \text{epochs}.$$

- When dealing with neural network training, we generally need to solve very high-dimensional problems, since $\theta \in \mathbb{R}^p$ with p usually large. For example, GPT-1 has 117 million parameters.

The need for first-order optimisation methods

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D}, \ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...), \ k = 1, ..., \text{epochs}.$$

- When dealing with neural network training, we generally need to solve very high-dimensional problems, since $\theta \in \mathbb{R}^p$ with p usually large. For example, GPT-1 has 117 million parameters.

- This implies that we need to use first-order algorithms, i.e., methods where T only depends on the gradient of $\mathcal{L}$, and not on its higher-order derivatives.

Some properties of functions fundamental in optimisation

Let $F : \mathbb{R}^d \to \mathbb{R}$ be a continuously differentiable function.

- **Lipschitz continuity:** F is L-Lipschitz continuous if and only if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $|F(\mathbf{y}) - F(\mathbf{x})| \leq L\|\mathbf{y} - \mathbf{x}\|_2$,

Some properties of functions fundamental in optimisation

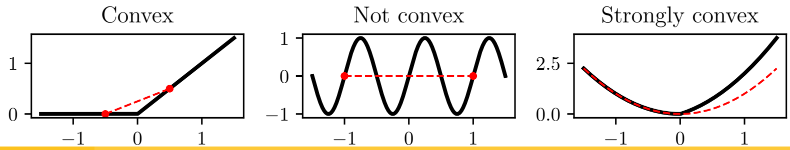
Let $F : \mathbb{R}^d \to \mathbb{R}$ be a continuously differentiable function.

- **Lipschitz continuity:** F is L-Lipschitz continuous if and only if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $|F(\mathbf{y}) - F(\mathbf{x})| \leq L\|\mathbf{y} - \mathbf{x}\|_2$,
- **L-smoothness:** F is L-smooth if and only if ∇F is L-Lipschitz, i.e. for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $\|\nabla F(\mathbf{y}) - \nabla F(\mathbf{x})\|_2 \leq L\|\mathbf{y} - \mathbf{x}\|_2$,

Some properties of functions fundamental in optimisation

Let $F : \mathbb{R}^d \to \mathbb{R}$ be a continuously differentiable function.

- **Lipschitz continuity:** F is L-Lipschitz continuous if and only if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $|F(\mathbf{y}) - F(\mathbf{x})| \leq L\|\mathbf{y} - \mathbf{x}\|_2$,
- **L-smoothness:** F is L-smooth if and only if ∇F is L-Lipschitz, i.e. for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $\|\nabla F(\mathbf{y}) - \nabla F(\mathbf{x})\|_2 \leq L\|\mathbf{y} - \mathbf{x}\|_2$,
- **Convexity:** F is convex if and only if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $(\nabla F(\mathbf{y}) - \nabla F(\mathbf{x}))^\top (\mathbf{y} - \mathbf{x}) \geq 0$,
- **Strong Convexity:** Let $\mu > 0$. F is μ-strongly convex if and only if for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $(\nabla F(\mathbf{y}) - \nabla F(\mathbf{x}))^\top (\mathbf{y} - \mathbf{x}) \geq \mu\|\mathbf{y} - \mathbf{x}\|_2^2 = \mu(\mathbf{y} - \mathbf{x})^\top (\mathbf{y} - \mathbf{x})$,



Gradient descent: The algorithm and its convergence properties

$$\theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k)) := \theta_k - \tau\nabla\mathcal{L}(\theta_k)$$

Gradient descent: The algorithm and its convergence properties

$$\theta_{k+1} = T(\theta_k, \nabla \mathcal{L}(\theta_k)) := \theta_k - \tau \nabla \mathcal{L}(\theta_k)$$

Assume that $\mathcal{L} : \mathbb{R}^p \to \mathbb{R}$ is μ-strongly convex, continuously differentiable, and L-smooth. Let $\theta^* = \arg\min_{\theta \in \mathbb{R}^p} \mathcal{L}(\theta)$. Assume $0 < \tau \leq 2/(\mu + L)$. Then

$$\|\theta_k - \theta_*\|_2^2 \leq \gamma^k \|\theta_0 - \theta_*\|_2^2, \ \gamma = \left(1 - 2\tau \frac{\mu L}{\mu + L}\right) \in (0, 1).$$

The contraction factor γ is minimised at $\tau^* = 2/(L + \mu)$, where

$$\gamma^* = \left(\frac{L - \mu}{L + \mu}\right)^2 = \left(\frac{\kappa - 1}{\kappa + 1}\right)^2, \ \kappa = \frac{L}{\mu} \text{(condition number of the problem)}.$$

For a proof, see Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*.

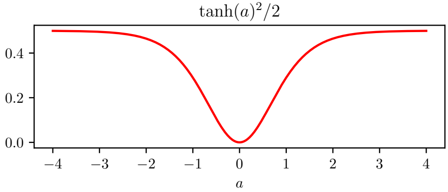
The loss function is usually not convex

The loss of neural networks is generally not convex. Consider $\mathcal{N}_\theta : \mathbb{R} \to \mathbb{R}$ defined as

$$\mathcal{N}_\theta(x) = \tanh(ax), \;\; \theta = a \in \mathbb{R}.$$

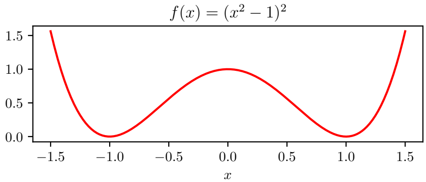
It is easy to see that the loss function below is not convex in this case

$$\mathcal{L}(a) = \frac{1}{2}\left(\mathcal{N}_\theta(1) - 0\right)^2 = \frac{1}{2}\tanh(a)^2.$$



Convergence properties for non-convex objectives

- The lack of convexity generally leads to several equivalent local minima.



- This complicates the convergence analysis of the optimisers. These lectures will not cover these aspects, but here are a couple of relevant references:
  - Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. "A Convergence Theory for Deep Learning via Over-Parameterization". In: *International Conference on Machine Learning*. Vol. 451. 2018,
  - Simon Du et al. "Gradient Descent Finds Global Minima of Deep Neural Networks". In: *International conference on machine learning*. PMLR. 2019, pp. 1675–1685.

Vanishing gradients

Backpropagation: how do we compute the gradients?

```
prediction = model(input) #Forward propagation
loss = criterion(prediction,target) #Compute the mean squared error
loss.backward() #Backpropagation
```

Backpropagation: how do we compute the gradients?

```
prediction = model(input) #Forward propagation
loss = criterion(prediction,target) #Compute the mean squared error
loss.backward() #Backpropagation
```

Let us focus on a data point $(\mathbf{x}_n, \mathbf{y}_n) \in \mathbb{R}^d \times \mathbb{R}^c$, and consider the network $\mathcal{N}_\theta = F_{\theta_L} \circ \cdots \circ F_{\theta_1}$. Define

$$\mathbf{x}^1 = \mathbf{x}_n, \qquad \mathbf{x}^{j+1} = F_{\theta_j}(\mathbf{x}^j),\ j = 1, \dots, L, \qquad \widehat{\mathbf{y}}_n := \mathbf{x}^{L+1}.$$

Define $\mathcal{L}_n := \|\widehat{\mathbf{y}}_n - \mathbf{y}_n\|_2^2/2$. Assume for simplicity that all the weights $\theta_1, ..., \theta_L$ are vectors. Set

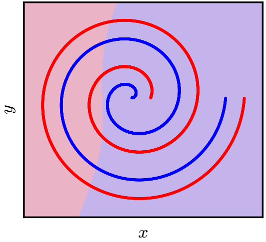
$$\mathbf{g}^{L+1} := \nabla_{\mathbf{x}^{L+1}} \mathcal{L}_n = \mathbf{x}^{L+1} - \mathbf{y}_n, \qquad \mathbf{g}^j := \nabla_{\mathbf{x}^j} \mathcal{L}_n = \left(J_{\mathbf{x}^j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \mathbf{g}^{j+1}, \quad j = L, \dots, 1,$$

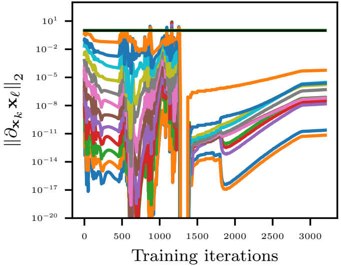
where J denotes a Jacobian.

Gradients (per sample)

$$\nabla_{\theta_j} \mathcal{L}_n = \left(J_{\theta_j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \nabla_{\mathbf{x}^j} \mathcal{L}_n = \left(J_{\theta_j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \mathbf{g}^{j+1}, \qquad j = L, \dots, 1.$$

Thus, the Backpropagation algorithm is just the chain rule organised to reuse Jacobian–vector products.





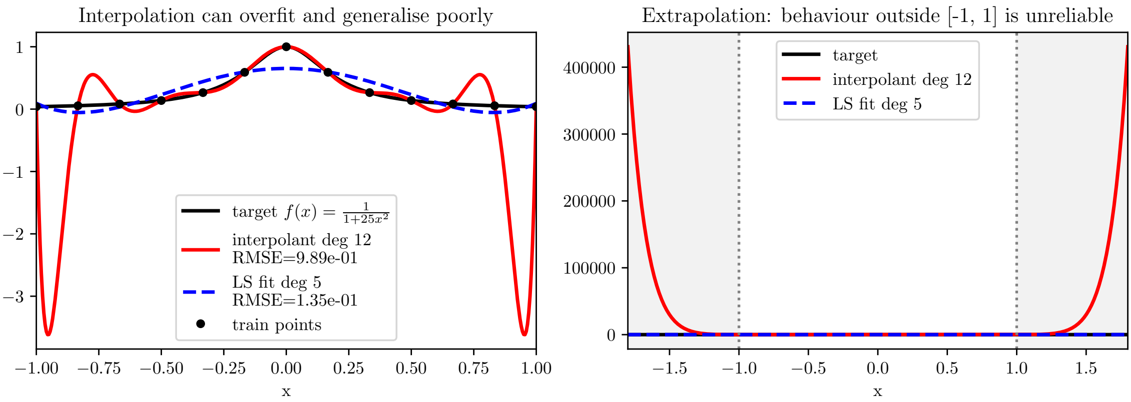
By repeated application of the chain rule, we can see that

$$\boxed{\|\nabla_{\theta_j}\mathcal{L}_n\|_2 \;\leq\; \|J_{\theta_j}F_{\theta_j}(\mathbf{x}^j)\|_2 \left(\prod_{\ell=j+1}^{L}\left\|J_{\mathbf{x}^\ell}F_{\theta_\ell}(\mathbf{x}^\ell)\right\|_2\right)\|\nabla_{\mathbf{x}^{L+1}}\mathcal{L}_n\|}$$

- If $\|J_{\mathbf{x}}F_{\theta_\ell}(\mathbf{x})\|_2 \leq \rho < 1$ (e.g. $\mathrm{Lip}(\sigma) \leq 1$ and $\|A_\ell\|_2 \leq \rho$), then $\|\nabla_{\theta_j}\mathcal{L}_n\|_2 \lesssim \rho^{L-j}$ $\Rightarrow$ **vanishing gradients**, and we can not meaningfully update the weights.

Interpolation, Generalisation, and Extrapolation

A visual understanding (the Runge function)



Improving generalisation and extrapolation in neural networks

Generalisation = Performance on i.i.d. test data from the same distribution of the training set.
Extrapolation = Performance outside the training regime/support.

Improving generalisation (in-distribution)

- **Data**: augmentation or synthetic data.
- **Explicit regularisation**: weight decay (ℓ_2), dropout, early stopping.
- **Smoothness & stability**: Jacobian/Lipschitz penalties, spectral/weight norm constraints, batch/weight norm.
- **Architecture**: residual connections, or normalisation layers.

Improving extrapolation (out-of-distribution)

- **Inductive biases**: symmetry/equivariance, invariances.
- **Physical/structural constraints**: physics-informed losses or hard constraints, Symplectic/Hamiltonian Networks, Monotone/Convex layers, stability/Lipschitz control.

Universal Approximation Theorems

What is a universal approximation theorem?

- One of the reasons behind the popularity of neural networks is their incredible flexibility and ability to approximate complicated and interesting functions.

What is a universal approximation theorem?

- One of the reasons behind the popularity of neural networks is their incredible flexibility and ability to approximate complicated and interesting functions.
- The Weierstrass approximation theorem states that every continuous function defined on a closed interval $[a, b]$ can be uniformly approximated as closely as desired by a polynomial function. Can we do something similar for neural networks?

What is a universal approximation theorem?

- One of the reasons behind the popularity of neural networks is their incredible flexibility and ability to approximate complicated and interesting functions.
- The Weierstrass approximation theorem states that every continuous function defined on a closed interval $[a, b]$ can be uniformly approximated as closely as desired by a polynomial function. Can we do something similar for neural networks?
- Analogous results for neural networks are called universal approximation theorems, and we now see two of them.

Shallow Neural Networks

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be an activation function, and consider the set of neural networks

$$\mathcal{F}_{\sigma,d} = \left\{ \mathbb{R}^d \ni \mathbf{x} \mapsto \mathcal{N}_\theta(\mathbf{x}) = \mathbf{a}^\top \sigma(A\mathbf{x} + \mathbf{b}) \in \mathbb{R} : \; A \in \mathbb{R}^{h\times d}, \, \mathbf{a}, \mathbf{b} \in \mathbb{R}^h, \, h \in \mathbb{N} \right\}.$$

Universal approximation theorem for shallow networks

Let $d \in \mathbb{N}$, and σ be a continuous function which is not a polynomial. Then for every $\Omega \subset \mathbb{R}^d$ compact, for every $\varepsilon > 0$, and for every continuous function $f : \Omega \to \mathbb{R}$, there is a network $\mathcal{N}_\theta \in \mathcal{F}_{\sigma,d}$ such that

$$\max_{x\in\Omega} |f(\mathbf{x}) - \mathcal{N}_\theta(\mathbf{x})| < \varepsilon.$$

This and several more such results can be found in Allan Pinkus. "Approximation theory of the MLP model in neural networks". In: *Acta numerica* 8 (1999), pp. 143–195.

Deep Neural Networks

There are several such results also for deep networks, see for example Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

Let us consider the set of networks

$$\mathcal{F}_{\sigma,d} = \left\{ A_L \circ \sigma \circ \ldots \circ A_1 \circ \sigma \circ A_0 : \mathbb{R}^d \to \mathbb{R} : L \in \mathbb{N},\ A_\ell \text{ affine}, \ell = 1, \ldots, L \right\}.$$

A simple and constructive result that we can prove for Deep ReLU networks is the following:

Representation of piecewise affine functions

Let $\sigma = \mathrm{ReLU}$. Any continuous pieceiwise affine (CPA) function $f : \mathbb{R}^d \to \mathbb{R}$ belongs to $\mathcal{F}_{\sigma,d}$.

Part I of the proof

- If $g : \mathbb{R}^d \to \mathbb{R}$ is convex and CPA, then $g(x) = \max_{m=1,\dots,M} \{\mathbf{a}_m^\top \mathbf{x} + b_m\}$ for some $\{(\mathbf{a}_m, b_m) \in \mathbb{R}^d \times \mathbb{R} : m = 1, \dots, M\}$.

Part I of the proof

- If $g : \mathbb{R}^d \to \mathbb{R}$ is convex and CPA, then $g(x) = \max_{m=1,\dots,M}\{\mathbf{a}_m^\top \mathbf{x} + b_m\}$ for some $\{(\mathbf{a}_m, b_m) \in \mathbb{R}^d \times \mathbb{R} : m = 1, \dots, M\}$.
- The function $(u, v) \mapsto \max(u, v) = f(u, v)$ belongs to $\mathcal{F}_{\sigma,2}$:

$$
\begin{aligned}
f(u, v) = \mathrm{ReLU}(u - v) + v &= \mathrm{ReLU}(u - v) + \mathrm{ReLU}(v) - \mathrm{ReLU}(-v) \\
&= \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^\top \mathrm{ReLU}\left(\begin{bmatrix} 1 & -1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right).
\end{aligned}
$$

This extends to the function $\mathbb{R}^M \ni \mathbf{u} \mapsto \max\{u_1, \dots, u_M\}$ since $\max\{a, b, c\} = \max\{\max\{a, b\}, c\}$ for $a, b, c \in \mathbb{R}$.

Part II of the proof

- By defining

$$\mathbf{u} = A\mathbf{x} + \mathbf{b} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} + b_1 \\ \vdots \\ \mathbf{a}_M^\top \mathbf{x} + b_M \end{bmatrix},$$

we see that $g \in \mathcal{F}_{\sigma,d}$.

[1] Prove that this is true as an exercise.

Part II of the proof

- By defining

$$\mathbf{u} = A\mathbf{x} + \mathbf{b} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} + b_1 \\ \vdots \\ \mathbf{a}_M^\top \mathbf{x} + b_M \end{bmatrix},$$

  we see that $g \in \mathcal{F}_{\sigma,d}$.

- Every CPA $f : \mathbb{R}^d \to \mathbb{R}$ can be written as $f = g - h$ where $g, h : \mathbb{R}^d \to \mathbb{R}$ are convex CPA.

[1] Prove that this is true as an exercise.

Part II of the proof

- By defining
$$\mathbf{u} = A\mathbf{x} + \mathbf{b} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} + b_1 \\ \vdots \\ \mathbf{a}_M^\top \mathbf{x} + b_M \end{bmatrix},$$
we see that $g \in \mathcal{F}_{\sigma,d}$.

- Every CPA $f : \mathbb{R}^d \to \mathbb{R}$ can be written as $f = g - h$ where $g, h : \mathbb{R}^d \to \mathbb{R}$ are convex CPA.

- Assume without loss of generality that g and h can be represented with the same number of layers[1]. We can then conclude that since $g, h \in \mathcal{F}_{\sigma,d}$ also $f \in \mathcal{F}_{\sigma,d}$. This is because
$$f(x) = g(x) - h(x) = \begin{bmatrix} 1 & -1 \end{bmatrix}^\top \begin{bmatrix} g(x) \\ h(x) \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}^\top \begin{bmatrix} A_L^g & 0 \\ 0 & A_L^h \end{bmatrix} \circ \sigma \ldots \circ \sigma \circ \begin{bmatrix} A_0^g(x) \\ A_0^h(x) \end{bmatrix},$$
and running in parallel two ReLU networks in $\mathcal{F}_{\sigma,d}$ maintains us inside $\mathcal{F}_{\sigma,d}$.

[1] Prove that this is true as an exercise.

Some of the most popular architectures

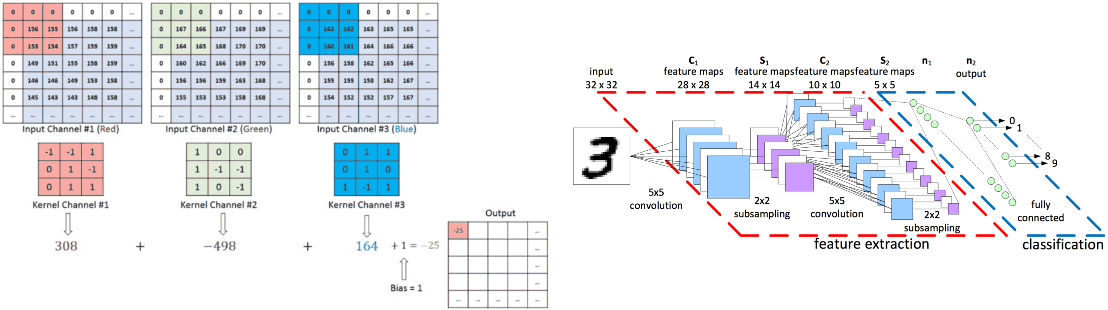


Figure 1: Source: `https://pylessons.com/CNN-tutorial-introduction`.

They allow to represent finite differences discretisations, see Zichao Long et al. "PDE-Net: Learning PDEs from Data". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3208–3216.

Autoencoders

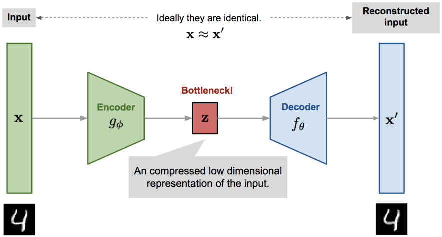
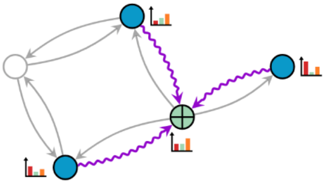


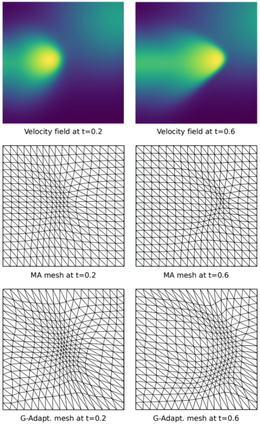
Figure 2: Source: `https://lilianweng.github.io/posts/2018-08-12-vae/`.

They can be seen as a non-linear version of the truncated Singular Value Decomposition $A \approx U\Sigma V^\top \in \mathbb{R}^{d\times d}$, $U, V \in \mathbb{R}^{d\times r}$, $\Sigma \in \mathbb{R}^{r\times r}$, $r \ll d$. Used for Reduced Order Modelling, and data-driven modelling, see, e.g., Kathleen Champion et al. "Data-driven discovery of coordinates and governing equations". In: *Proceedings of the National Academy of Sciences* 116.45 (2019), pp. 22445–22451.

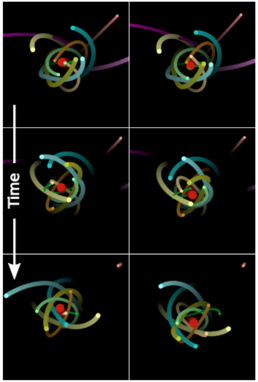
Graph Neural Networks



(a) Source: `https://pytorch-geometric.readthedocs.io/`.



(b) Rowbottom et al., “G-Adaptivity: optimised graph-based mesh relocation for finite element methods”.



(c) Battaglia et al., “Interaction networks for learning about objects, relations and physics”.

Recurrent Neural Networks / Transformers

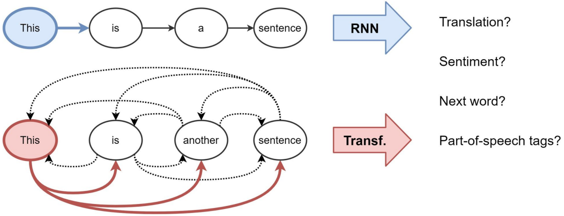


Figure 4: Source: `https://thegradient.pub/transformers-are-graph-neural-networks/`.

Transformers are still hard to describe mathematically, but a promising interpretation relates them with dynamical systems (as we will do with ResNets, and as it can be done for RNNs as well), and interprets them as interacting particle systems, see, e.g., Borjan Geshkovski et al. "A mathematical perspective on transformers". In: *Bulletin of the American Mathematical Society* 62.3 (2025), pp. 427–479.

APPENDIX

Some historical background on Neural Networks

- The research in this area began with Frank **Rosenblatt**, who developed the **Perceptron**, attempting to replicate the functioning of biological neurons (1957).

Some historical background on Neural Networks

- The research in this area began with Frank **Rosenblatt**, who developed the **Perceptron**, attempting to replicate the functioning of biological neurons (1957).
- Apart from a few exciting developments like **Hopfield Neural Networks**, this research direction seemed less promising in the 1970s and 1980s, especially after the publication of the book "Perceptrons" (M. **Minsky** and S. **Papert**, 1969).

Some historical background on Neural Networks

- The research in this area began with Frank **Rosenblatt**, who developed the **Perceptron**, attempting to replicate the functioning of biological neurons (1957).
- Apart from a few exciting developments like **Hopfield Neural Networks**, this research direction seemed less promising in the 1970s and 1980s, especially after the publication of the book "Perceptrons" (M. **Minsky** and S. **Papert**, 1969).
- **Rumelhart, Hinton, and Williams** in 1986 published an experimental analysis of the **backpropagation algorithm**, still used nowadays to train neural networks.

Some historical background on Neural Networks

- The research in this area began with Frank **Rosenblatt**, who developed the **Perceptron**, attempting to replicate the functioning of biological neurons (1957).
- Apart from a few exciting developments like **Hopfield Neural Networks**, this research direction seemed less promising in the 1970s and 1980s, especially after the publication of the book "Perceptrons" (M. **Minsky** and S. **Papert**, 1969).
- **Rumelhart, Hinton, and Williams** in 1986 published an experimental analysis of the **backpropagation algorithm**, still used nowadays to train neural networks.
- Neural networks found their real traction when computing resources, like **graphics cards**, improved their efficiency.

Some historical background on Neural Networks

- The research in this area began with Frank **Rosenblatt**, who developed the **Perceptron**, attempting to replicate the functioning of biological neurons (1957).
- Apart from a few exciting developments like **Hopfield Neural Networks**, this research direction seemed less promising in the 1970s and 1980s, especially after the publication of the book "Perceptrons" (M. **Minsky** and S. **Papert**, 1969).
- **Rumelhart, Hinton, and Williams** in 1986 published an experimental analysis of the **backpropagation algorithm**, still used nowadays to train neural networks.
- Neural networks found their real traction when computing resources, like **graphics cards**, improved their efficiency.
- Our **mathematical understanding** of why neural networks are so effective in many areas is **still lacking**. A lot of mathematicians are now working on the **Mathematics of Deep Learning** to try to understand these models better.