Solving Inverse Problems with Deep Learning

–

Theoretical introduction to Neural Networks

Andreas Hauptmann

Jesus College, Cambridge

20 September 2023

Outline

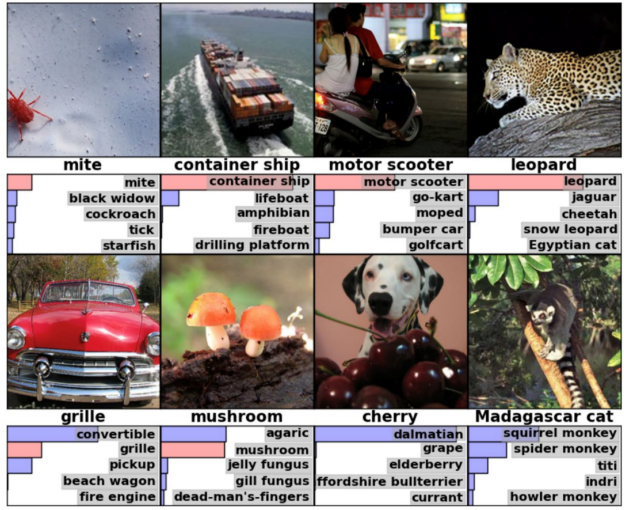
Outline

Outline

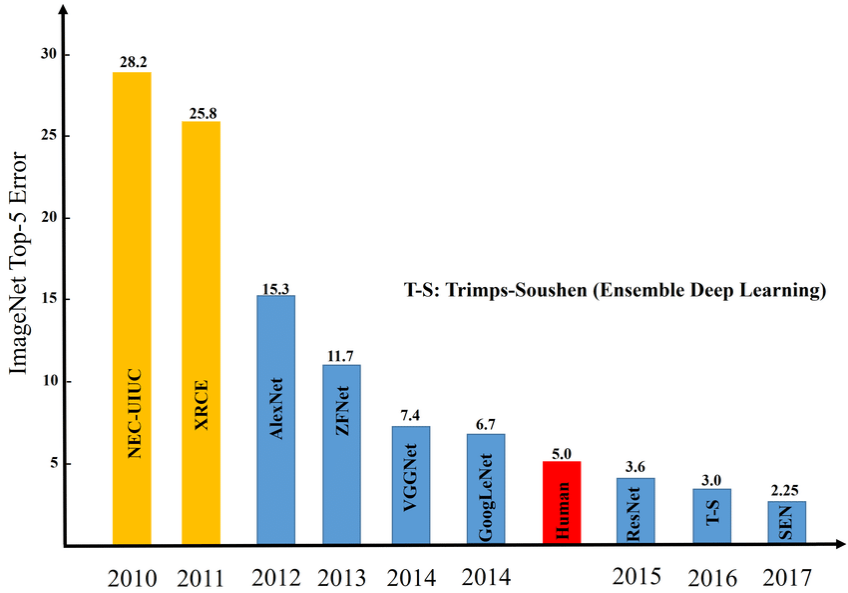
Standard computer vision problem: classification (ImageNet)

IMAGENET

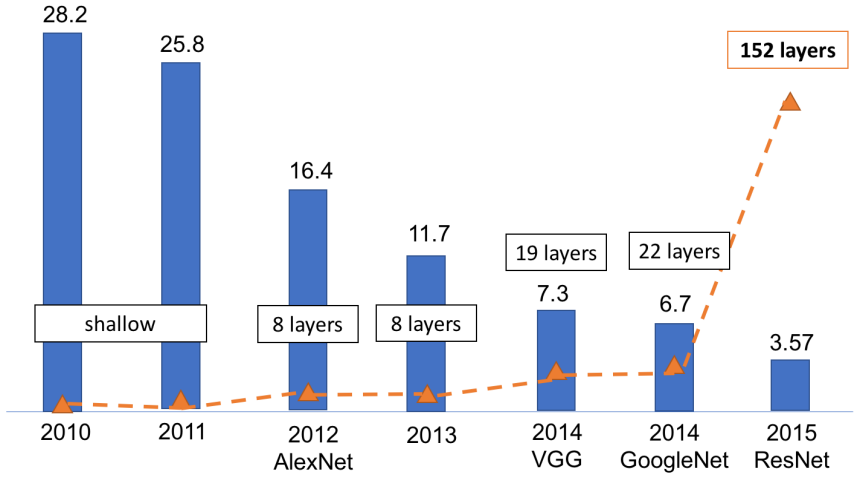
- 1,000 object classes (categories).
- Images:
  - 1.2 M train
  - 100k test.



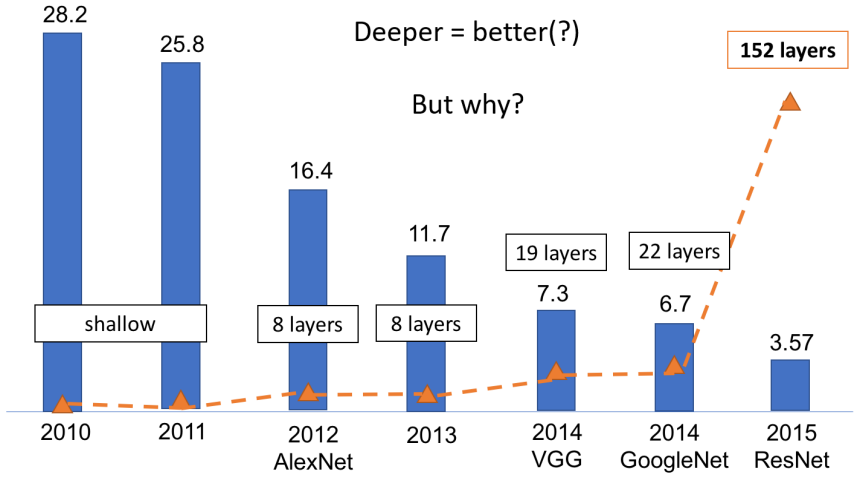
Performance over the years



Performance over the years: Number of layers



Performance over the years: Number of layers



Foreword: What are we trying to learn?

- Many tasks in signal processing involve approximating a function/operator from example data.
- A special case of this is solving an inverse problem, which can be seen as finding an appropriate approximation to the inverse of the forward operator.
- We view training a neural network as attempting to approximate an unknown function/operator using example data (training data).

Note: Much of theory of (deep) neural networks deal with its usage for classification whereas inverse problems often deal with regression in function spaces, not classification. Thus, there is not that much theory that is directly applicable to inverse problems.

The inverse problem: The Why and What

- We assume there is an underlying mapping that can approximate the inverse, i.e., $\Lambda : g \mapsto f$ such that $Af \approx g$.
- The task of a neural network is then to find this mapping $\Lambda\colon\ Y \to X$ from pairs of measurement data g and ground truth signals f.
  Remember: the inversion operator for an ill-posed inverse problem is (usually) discontinuous and a regularised version of the inverse mapping would be desired.
- We assume in the following that a ground-truth $\Lambda\colon\ Y \to X$ exists, but is unknown.
- We consider here a fully supervised setting, that means we have access to pairs of $\{g, f\} \in Y \times X$ with $Af = g$ either on the whole space or suitable subspaces.
  This is of course often not the case in practice.

Today: What are we trying to do and can we?

The inverse problem: The Why and What

- We assume there is an underlying mapping that can approximate the inverse, i.e., $\Lambda : g \mapsto f$ such that $Af \approx g$.
- The task of a neural network is then to find this mapping $\Lambda\colon\ Y \to X$ from pairs of measurement data g and ground truth signals f.
  Remember: the inversion operator for an ill-posed inverse problem is (usually) discontinuous and a regularised version of the inverse mapping would be desired.
- We assume in the following that a ground-truth $\Lambda\colon\ Y \to X$ exists, but is unknown.
- We consider here a fully supervised setting, that means we have access to pairs of $\{g, f\} \in Y \times X$ with $Af = g$ either on the whole space or suitable subspaces.
  This is of course often not the case in practice.

But How? That will be (my) next lectures

Outline

The observation model

Given elements $v \in V$ and $u \in U$ related through the data model

$$u = \Lambda_{\text{true}}(v) + e \quad \text{for some unknown error } e \in U, \tag{2.1}$$

where $\Lambda_{\text{true}} \colon V \to U$.

- Λ_{true} is only known through observed example data $\Rightarrow$ approximate Λ_{true}
- Example data consists of pairs $\{v^{(i)}, u^{(i)}\} \subset V \times U$ (supervised training data) generated by the data model:

$$u^{(i)} = \Lambda_{\text{true}}(v^{(i)}) + e^{(i)} \quad \text{for some unknown error } e^{(i)} \in U. \tag{2.2}$$

Our task: seek an approximator for Λ_{true} in some sub-class $\mathbb{L}$ of measurable U-valued mappings defined on V.

The training problem: Empirical risk minimisation

Λ_{true} is not necessarily contained in $\mathbb{L}$, but each element in $\mathbb{L}$ is a candidate approximator. We can find an approximator as follows:

Training: Using the training data to find an 'optimal' approximation within $\mathbb{L}$.

Optimality: Introduce a pre-defined loss $L_U \colon U \times U \to \mathbb{R}$ that quantifies the approximation error.

Empirical risk: Quantifies the average approximation error over the entire training data, $\Phi \colon \mathbb{L} \to \mathbb{R}$ defined as

$$\Phi(\Lambda) = \frac{1}{N} \sum_{i=1}^{N} L_U(\Lambda(v^{(i)}), u^{(i)}) \quad \text{for } \Lambda \in \mathbb{L}. \tag{2.3}$$

Training problem: To seek the approximator that minimises the empirical risk, i.e., to minimise $\Lambda \mapsto \Phi(\Lambda)$ over $\mathbb{L}$.

Parameterise the mapping: Neural Networks

- For computational feasibility, it is common to consider parametrised approximator classes, i.e., $\mathbb{L} := \{\Lambda_\theta\}_{\theta\in\Theta}$ where the parameter θ specifies the mapping. (Compare: Finite elements parametrise the solution of PDEs by basis functions)
- We will loosely refer to the parametrised map $\Lambda_\theta \colon V \to U$ for a some parameters $\theta \in \Theta$ as a *neural network*.
- Then, the above training problem can be rephrased as finding the approximator Λ_{θ^*} where θ^* solves

$$\theta^* \in \arg\min_{\theta\in\Theta} \Phi(\Lambda_\theta) = \arg\min_{\theta\in\Theta} \frac{1}{N}\sum_{i=1}^{N} L_U(\Lambda_\theta(v^{(i)}), u^{(i)}) \tag{2.4}$$

$\Rightarrow$ The above corresponds to the case where we have finite training data available and limited capacity due the class of possible approximators $\mathbb{L}$.

Evaluating the neural network: Risk

When deploying the trained neural network for prediction we are rather interested in how well does the found approximator do on unseen data.:

- We consider the average error that the trained approximator has over all possible data.

  (*Risk*) Introduce $\widehat{\Phi}\colon \mathbb{L} \to \mathbb{R}$ defined as

$$\widehat{\Phi}(\Lambda) = \int_{V\times U} L_U\big(\Lambda(v), u\big)\,\mathrm{d}\mu(v,u) \tag{2.5}$$

  where μ is a normalised measure on $V \times U$.

- One could similarly seek the approximator $\Lambda_{\widehat{\theta}}\colon V \to U$ where

$$\widehat{\theta} \in \operatorname*{arg\,min}_{\theta\in\Theta} \widehat{\Phi}(\Lambda_\theta). \tag{2.6}$$

$\Rightarrow$ This corresponds to the ideal case of considering all possible data while having a limited capacity $\Lambda_\theta \in \mathbb{L}$.

Outline

Risk vs. empirical risk

The risk $\widehat{\Phi}(\Lambda_{\theta^*}) \in \mathbb{R}$ of θ^* found by empirical risk minimisation in eq. (2.4) is now a quantitative measure for how well a trained approximator performs on unseen data.

- This brings us to one of the central problems in training a neural network approximator, which is to ensure good performance on all data (risk $\widehat{\Phi}$) while having access only to limited data (empirical risk Φ).
- This in particular important when applying neural networks to inverse problems: is the available training data sufficient enough to approximate a reliable reconstruction operator that performs well also when applied to new unseen data?

Risk separation

Bearing in mind the above, an approach is therefore to separate the risk of a trained network into components that can either be estimated or computed.
$\Rightarrow$ We express the risk as:

$$\widehat{\Phi}(\Lambda_{\theta^*}) = \underbrace{\widehat{\Phi}(\Lambda_{\theta^*}) - \Phi(\Lambda_{\theta^*})}_{\text{Generalisation}} + \underbrace{\Phi(\Lambda_{\theta^*}) - \widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Training}} + \underbrace{\widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Approximation}}. \tag{2.7}$$

$$\widehat{\Phi}(\Lambda_{\theta^*}) = \underbrace{\widehat{\Phi}(\Lambda_{\theta^*}) - \Phi(\Lambda_{\theta^*})}_{\text{Generalisation}} + \underbrace{\Phi(\Lambda_{\theta^*}) - \widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Training}} + \underbrace{\widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Approximation}}.$$

Separation of risk and errors

Approximation: The term $\widehat{\Phi}(\Lambda_{\widehat{\theta}})$ measures how well the ideal approximator $\Lambda_{\widehat{\theta}}$ does for all possible samples, i.e, it measures the best possible approximation provided by the class $\mathbb{L}$.

Training: The term $\Phi(\Lambda_{\theta^*}) - \widehat{\Phi}(\Lambda_{\widehat{\theta}})$ signifies the difference between the best possible $\widehat{\Phi}(\Lambda_{\widehat{\theta}})$ given infinite data and the achievable through training with finite data $\Phi(\Lambda_{\theta^*})$.

Generalisation: The final term measures the difference between evaluating the trained model Λ_{θ^*} on finite data $\Phi(\Lambda_{\theta^*})$ versus evaluating on infinite data $\Phi(\Lambda_{\theta^*})$.

Takeaways so far

$$\widehat{\Phi}(\Lambda_{\theta^*}) = \underbrace{\widehat{\Phi}(\Lambda_{\theta^*}) - \Phi(\Lambda_{\theta^*})}_{\text{Generalisation}} + \underbrace{\Phi(\Lambda_{\theta^*}) - \widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Training}} + \underbrace{\widehat{\Phi}(\Lambda_{\widehat{\theta}})}_{\text{Approximation}}.$$

- We can see from the risk separation, that when available data would approach infinity and generalisation and training error approach zero, the risk is still limited by the approximation quality of the class $\mathbb{L}$.
- It should be also noted, that in practice a training error approaching zero is not desired when the given data $v \in V$ contains noise. This is particularly important for inverse problems (need for regularisation).
  $\Rightarrow$ non-perfect training error can be seen as a way to regularise the mapping Λ_{θ^*}.

Outline

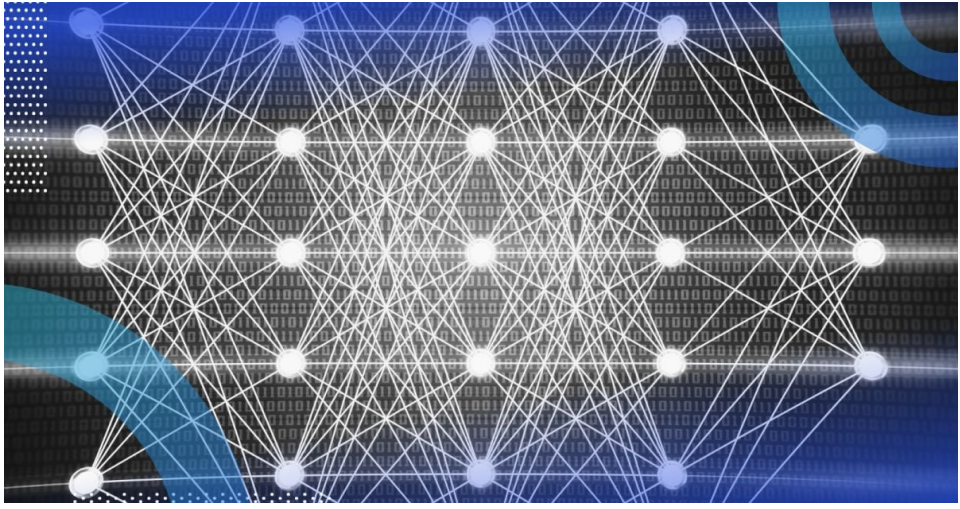
Outline

Neural networks as function approximators

A common interpretation of neural networks is that of parametrised nonlinear function approximator. $\Rightarrow$ what kind of functions are we aiming to represent and how to formulate a neural network to achieve this task?

- The classic formulation of a neural network in the finite dimensional setting is given as functions from $\mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$.
- The basic building blocks of a neural network follow the simple principle of composing (affine) linear transformations, i.e., matrix multiplication and vector additions, with fixed pointwise nonlinear functions.
- A simple parametrisation can be found to represent general classes of nonlinear functions.

Neural Networks: The machine learning view



Shallow networks: I

Basic shallow network: Consider the mapping for $x \in \mathbb{R}^d$,

$$x \mapsto \sum_{j=1}^{m} a_j \sigma(w_j^T x + b_j) \in \mathbb{R}. \tag{3.1}$$

- $\{(a_j, w_j, b_j)\}_{j=1}^m$ are the *trainable parameters* (network parameters): varying them defines the function class/mapping properties.

Shallow networks: I

Basic shallow network: Consider the mapping for $x \in \mathbb{R}^d$,

$$x \mapsto \sum_{j=1}^{m} a_j \sigma(w_j^T x + b_j) \in \mathbb{R}.$$

- $\{(a_j, w_j, b_j)\}_{j=1}^m$ are the *trainable parameters* (network parameters): varying them defines the function class/mapping properties.
- σ is the *nonlinearity/activation*. Typical choices:
  ReLU $z \mapsto \max\{0, z\}$
  sigmoid $z \mapsto \frac{1}{1+\exp(-z)}$
  Note: σ always acts point-wise from $\mathbb{R} \to \mathbb{R}$. (Also when applied to a vector)

Shallow networks: II

Basic shallow network: Consider the mapping for $x \in \mathbb{R}^d$,

$$x \mapsto \sum_{j=1}^{m} a_j \sigma(w_j^T x + b_j) \in \mathbb{R}.$$

- We can think of this as a two layer neural network of width m: we have a hidden layer of m nodes, where the jth node computes $x \mapsto \sigma(w_j^T x + b_j)$.
- Define weight matrix $W \in \mathbb{R}^{m \times d}$ and bias vector $b \in \mathbb{R}^m$ with $W_{j,\cdot} = w_j^T$ (rows) and b_j as above.

Shallow networks: II

Basic shallow network: Consider the mapping for $x \in \mathbb{R}^d$,

$$x \mapsto \sum_{j=1}^{m} a_j \sigma(w_j^T x + b_j) \in \mathbb{R}.$$

- We can think of this as a two layer neural network of width m: we have a hidden layer of m nodes, where the jth node computes $x \mapsto \sigma(w_j^T x + b_j)$.
- Define weight matrix $W \in \mathbb{R}^{m \times d}$ and bias vector $b \in \mathbb{R}^m$ with $W_{j,\cdot} = w_j^T$ (rows) and b_j as above.
  $\Rightarrow$ The first layer then computes

$$h := \sigma(Wx + b) \in \mathbb{R}^m$$

(Remember: $\sigma : \mathbb{R} \to \mathbb{R}$ acts point/coordinate-wise).
The second layer computes the output $h \mapsto a^T h$.

Shallow Neural Networks: Formal definition

Definition 3.1 (Shallow neural network)

Given input vector $\mathbf{v} \in \mathbb{R}^d$, weight matrix $\mathrm{W} \in \mathbb{R}^{m \times d}$, bias vector $\mathrm{b} \in \mathbb{R}^m$, weight vector $\mathbf{a} \in \mathbb{R}^m$ and pointwise nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$. Then the parameters are given as $\theta = \{\mathrm{W}, \mathrm{b}, \mathbf{a}\}$ and we define a shallow neural network by

$$\Lambda_\theta(\mathbf{v}) = \mathbf{a}^{\mathrm{T}} \sigma(\mathrm{W}\,\mathbf{v} + \mathrm{b}). \tag{3.2}$$

Deep Neural Networks: Composition of layers

- A more efficient parameterisation of a neural network can be achieved with more layers $\Rightarrow$ This leads to the notion of deep neural networks as composition of multiple layers of the form $h := \sigma(Wx + b)$
- This composition follows the same construction principle of affine linear mappings followed by a pointwise nonlinear function in a layered structure.
- We obtain a basic L-layer deep neural network as

$$\begin{aligned} \Lambda_\theta(\mathbf{v}) &= \mathbf{a}^{\mathrm{T}}(h^{(L)} \circ h^{(L-1)} \circ \cdots \circ h^{(2)} \circ h^{(1)})(\mathbf{v}) \\ &= \mathbf{a}^{\mathrm{T}}\sigma_L(\mathrm{W}^{(L)}\,\sigma^{(L-1)}(\cdots \mathrm{W}^{(2)}\,\sigma_1(\mathrm{W}^{(1)}\,\mathbf{v} + \mathrm{b}^{(1)}) + \mathrm{b}^{(2)} \cdots) + \mathrm{b}^{(L)}). \end{aligned} \tag{3.3}$$

- The network parameters are $\theta = \{\mathbf{a}, \mathrm{W}^{(1)}, b^{(1)}, \mathrm{W}^{(2)}, b^{(2)}, \ldots, \mathrm{W}^{(L)}, b^{(L)}\}$.

Some observations

- If all weight matrices W have width m then the total number of learnable parameters is consequently $|\Theta| = L(md + m) + m$. In contrast, the shallow network had $|\Theta| = md + 2m$.
- On the first sight this is more than in the shallow case, but we will see that deep networks we do not need large width m compared to shallow network (for same representation power)
- The formulation of a deep neural network leaves a lot of freedom: number of layers, width of each layer (dimensions of the weight matrices), as well as choice of nonlinearities.
- Additionally, one can enforce certain structures on the weight matrices, such as sparsity or certain linear transformations such as convolutions. Sometimes, biases are dropped in the above formulation or only used for a subset of layers. We will discuss specific building blocks in more detail next lecture.

A short note on the continuous setting: Infinite width networks

The majority of literature in neural network theory deals with the discrete setting. In recent years there has been increasing interest in studying neural networks in the continuous setting. Despite the growing interest, there are varying notions of a continuous neural networks.

$\Rightarrow$ What happens if we let the width go to infinity?

- This question goes back to Barron 1993 by connections to Fourier Analysis.
- One common way proposed by Weinan E 2019: let $\Lambda : \mathbb{R}^d \to \mathbb{R}$ we then define the infinite width equivalent by

$$\Lambda(\mathbf{v}) = \int_{\Theta} a\sigma(\mathbf{w}^{\mathrm{T}}\mathbf{v} + b) \, \mathrm{d}\mu(a, \mathbf{w}, b).$$

  Here $\Theta = \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}$ and μ is a measure on Θ.
- The infinite width is represented in the integral, compare to the finite

$$\Lambda_\theta(\mathbf{v}) = \sum_{i=1}^{m} a_i \sigma(\mathbf{w}_i^{\mathrm{T}}\mathbf{v} + b_i).$$

Outline

Motivation

- The aim to use neural networks is to approximate complicated and possibly unknown mappings between input v and output pairs u.
- To formalise this, we assumed that the mapping Λ to be approximated lies in some function class $\mathbb{L}$, which can be understood as a certain class of architectures.
- We would want to find an element $\Lambda \in \mathbb{L}$ which simultaneously has small risk $\widehat{\Phi}(\Lambda)$ (small error when applied to test data) and small complexity (few parameters).
- First, we concentrate on the aspect of the approximation quality that can be achieved within the class $\mathbb{L}$. Then we can shortly examine if we can achieve the same approximation quality with fewer parameters by constructing deeper networks.

Approximation of continuous functions

The classical approach to examine the approximation quality is to set a natural baseline with good approximation quality. For instance, one could compare against all continuous functions:

- Given $\Gamma \in C^0$ is continuous, then one can consider how well continuous functions are approximated by $\Lambda \in \mathbb{L}$, i.e., to bound

$$\|\Lambda - \Gamma\|_\infty := \sup_{v \in V} \left|\Lambda(v) - \Gamma(v)\right| \text{ for all } \Gamma \in C^0.$$

$\Rightarrow$ We want to ensure that the worst case approximation within the class $\mathbb{L}$ is still sufficiently good.

Universal approximation

This is exactly what the well known classic result on universal approximation examines.

- We consider the case of a shallow neural network as introduced in eq. (3.2):

$$\Lambda_\theta(\mathbf{v}) = \mathbf{a}^{\mathrm{T}}\sigma(\mathrm{W}\,\mathbf{v} + \mathrm{b})$$

- A shallow neural network with sufficient representational power (width) can approximate any given continuous function to a given accuracy.

Let us first define an universal approximator:

Definition 4.1

A class of functions $\mathbb{L}$ is a universal approximator over a compact set $S \subset V$ if for every continuous function $\Gamma \in C^0$ and target accuracy $\epsilon > 0$, there exists $\Lambda \in \mathbb{L}$ with

$$\|\Lambda - \Gamma\|_\infty = \sup_{v \in S} |\Lambda(v) - \Gamma(v)| \leq \epsilon.$$

That means: We can approximate a given continuous functions to any desirable accuracy

Shallow neural networks are universal approximator

We define the class of shallow neural networks of width m by

$$\mathbb{L}_{\sigma,m,d} := \left\{ v \mapsto \mathbf{a}^T \sigma(Wx + \mathbf{b}) : W \in \mathbb{R}^{m \times d},\ \mathbf{a}, \mathbf{b} \in \mathbb{R}^m, \sigma : \mathbb{R} \to \mathbb{R} \right\}$$

and the union over arbitrary wide but finite width networks

$$\mathbb{L}_{\sigma,d} := \bigcup_{m \geq 1} \mathbb{L}_{\sigma,m,d}.$$

Then the classic universal approximation theorem can be stated simply as follows.

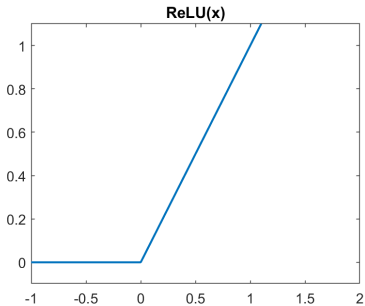
Theorem 4.2 (Hornik, Stinchcombe, and White 1989)

For $\sigma : \mathbb{R} \to \mathbb{R}$ continuous and not a polynomoial, $\mathbb{L}_{\sigma,d}$ is a universal approximator.

Proof: Verify conditions of Stone-Weierstrass approximation theorem

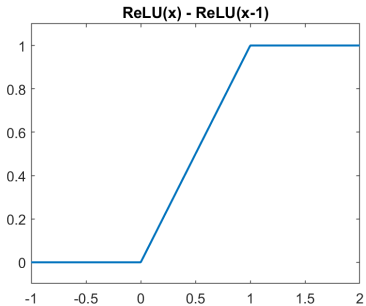
Intuition: Approximation by piecewise linear functions

- The universal approximation property can be intuitively seen in the one-dimensional case, as we can construct piecewise linear functions by ReLU networks.
- This leads to a similar approximation as used by the Riemann/Trapezoidal rule for integrals: approximate any continuous function, with sufficiently many nodes up to the desired accuracy.



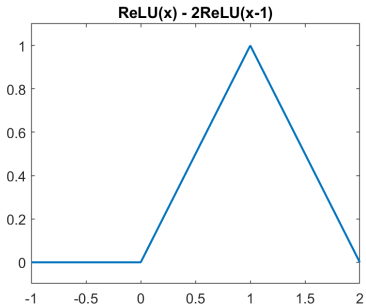
Intuition: Approximation by piecewise linear functions

- The universal approximation property can be intuitively seen in the one-dimensional case, as we can construct piecewise linear functions by ReLU networks.
- This leads to a similar approximation as used by the Riemann/Trapezoidal rule for integrals: approximate any continuous function, with sufficiently many nodes up to the desired accuracy.



Intuition: Approximation by piecewise linear functions

- The universal approximation property can be intuitively seen in the one-dimensional case, as we can construct piecewise linear functions by ReLU networks.
- This leads to a similar approximation as used by the Riemann/Trapezoidal rule for integrals: approximate any continuous function, with sufficiently many nodes up to the desired accuracy.



Curse of dimensionality

Continuous functions and curse of dimensionality

For every $\Gamma \in C^0$ and any given ε there exists an $\Lambda \in \mathbb{L}_{\sigma,d}$ with width m large enough that

$$\|\Lambda - \Gamma\|_\infty < \varepsilon.$$

The problem with this estimate is that it only states the possibility to approximate any continuous function accurately, but not how wide the neural network has to be. In fact, the width of a shallow neural networks is governed exponentially with the estimate $m = \Omega(1/\varepsilon^d)$, i.e., the width m is bounded below by $1/\varepsilon^d$ asymptotically.
This is referred to as the *curse of dimensionality*.

Curse of dimensionality: Remarks

- Remember, d is the input dimension of the discrete data and if we think about images in inverse problems, the needed width of a network would quickly exceed sensible limits.
- Shallow networks do not see much application in inverse problems and imaging applications in general.
- Fortunately, we can overcome this by adding more layers to the network.
- Note, that the constructions here only consider networks from $\mathbb{R}^d \to \mathbb{R}$. But can be shown to generalise to multiple (output) dimensions.

Benefit of deep networks

Shallow networks are sufficient to approximate any continuous function, but the needed width for practical sized problems is unfeasibly large and by that the complexity of models, which in turn makes training more difficult due to large parameter search space.

$\Rightarrow$ Deep Neural networks have shown empirically that performance can be improved when going deeper, improving representational power while simultaneously reducing complexity.

This is based on the realisation that deep ReLU networks are efficient in generating piecewise affine pieces: shallow networks generate one piecewise affine per weight (neuron), deep networks exponentially increase piecewise affine parts per layer.

Piecewise linear parts: Shallow vs. Deep

In short each piecwise affine part in a shallow net needs two nodes, in deep networks of width m and L layers we get m^L nodes.

A recursive sawtooth function

We consider the specific saw function:

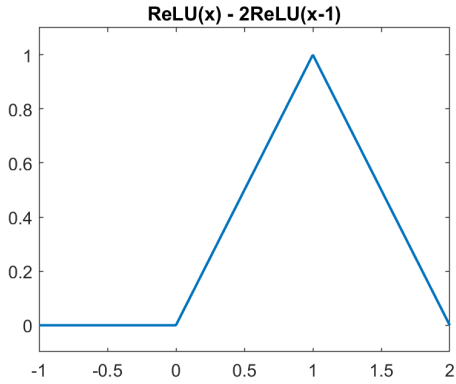
$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$

- We can efficiently represent t by a neural network of width 2 using ReLU activations.
- The n-fold composition $t_L(x) := t \circ \cdots \circ t$ produces a saw function with 2^L spikes.
- In particular, t_L admits 2^L affine linear pieces, which we can represent with a L-layer width 2 network, i.e., a total of $2L$ neurons and $4L + 2L$ weights (matrix + bias).

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$

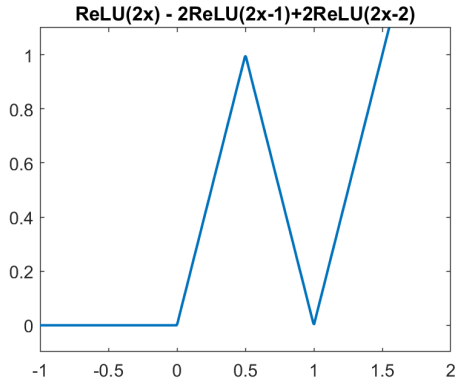


Weights: 2 nodes$\times$3 parameters

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$

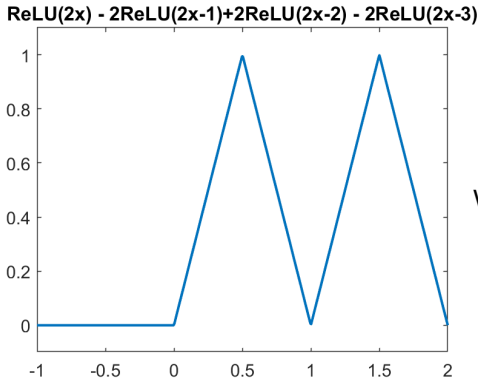


Weights: 3 nodes$\times$3 parameters

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$

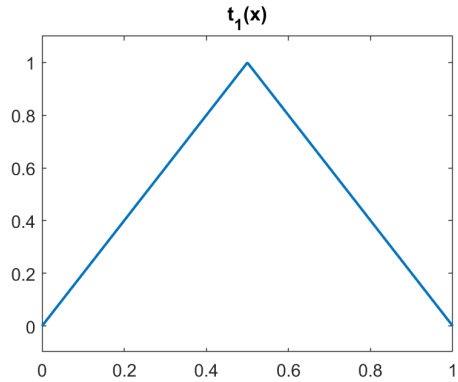


Weights: 4 nodes$\times$3 parameters

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \leq x \leq \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \leq x \leq 1. \end{cases}$$



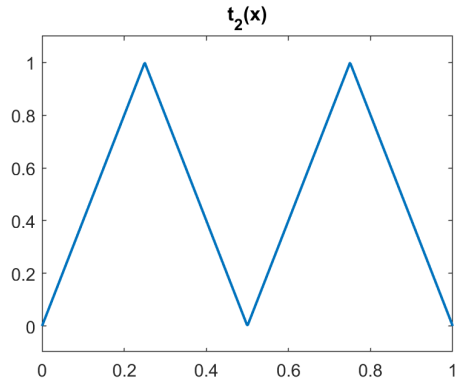
Shallow: 2 nodes$\times$3 parameters

Deep: 2 nodes$\times$3 parameters

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$



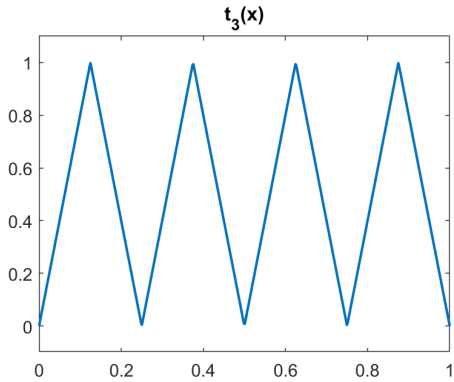
Shallow: 4 nodes$\times$3 parameters $= 12$

Deep: 2 layers $\times$ $(2 \text{ nodes})^2$ $+$ 2 layers $\times$ 2 bias $= 12$

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$



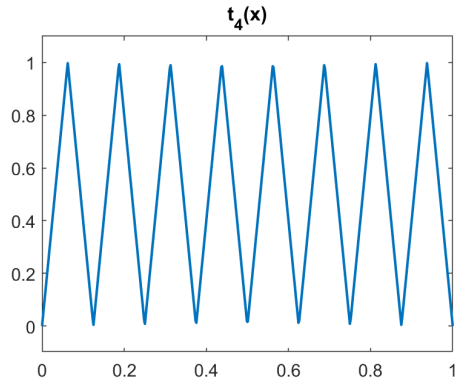
Shallow: 8 nodes$\times$3 parameters $= 24$

Deep: 3 layers $\times$ $(2 \text{ nodes})^2$ $+$ 3 layers $\times$ 2 bias $= 18$

A recursive sawtooth function: Illustration

We consider the specific saw function:

$$t(x) = \begin{cases} 2x & \text{for } 0 \le x \le \frac{1}{2} \\ 2(1-x) & \text{for } \frac{1}{2} \le x \le 1. \end{cases}$$



Shallow: 16 nodes$\times$3 parameters $= 48$

Deep: 4 layers $\times$ (2 nodes)^2 $+$ 4 layers $\times$ 2 bias $= 24$

Deep networks are exponentially more efficient

To prove the effectiveness, the formal result states that t_{L^2+2} can not be approximated by shallow L-layer networks, unless they have exponential size, i.e., more than 2^L nodes:

Theorem 4.3 (Telgarsky 2015, 2016)

Let $L \geq 2$, then $\Gamma = t_{L^2+2}$ is a ReLU *network with $2L^2 + 4$ nodes and $L^2 + 2$ layers. Let Λ be a* ReLU *network with $\leq 2^L$ nodes and $\leq L$ layers, then it can not approximate Γ with bound*

$$\int_{[0,1]} \left|\Lambda(v) - \Gamma(v)\right| \mathrm{d}v \geq \frac{1}{32}.$$

$\Rightarrow$ Deep ReLU networks are exponentially more efficient in approximating affine linear functions.

Summary approximation

Main takeaway

We have shown that neural networks are universal approximator, i.e., they can approximate continuous functions to any given accuracy.
$\Rightarrow$ Our underlying function needs to be well approximable by continuous functions.

- Shallow networks would be sufficient, but need exponential width with input dimension.
- Deep networks offer more representative power with fewer parameters

Outline

Training the network: Optimisation

We have discussed how well we can approximate certain functions with a given class and in particular that deep networks are more efficient in representing general classes with less weights: We still need to find this approximation

$\Rightarrow$ which leads to the optimisation problem, referred to as *training task*:

- Consider now fixed architectures, that is we have a class $\mathbb{L}$ with a fixed parameter set Θ. That is, we set here again $\mathbb{L} := \{\Lambda_\theta\}_{\theta \in \Theta}$.
- The goal is to find $\theta \in \Theta$ through optimisation
- In contrast to allowing for varying parameterisations of variable width, as considered in the approximation section.

Recall: Empirical risk minimisation

Recall: the total risk does not only depend on the approximation error, but also on how well we can train the network weights $\rightarrow$ We have only access to the empirical risk.

- Given a training set of pairs $\{v^{(i)}, u^{(i)}\}_{i=1}^{N}$.
- Find the optimal parameter θ^* that minimises the empirical risk Φ for fixed architectures $\Lambda_\theta \colon V \rightarrow U$ in the class $\mathbb{L}$:

$$\theta^* = \arg\min_{\theta \in \Theta} \Phi(\Lambda_\theta) = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} L_U\big(\Lambda_\theta(v^{(i)}), u^{(i)}\big). \tag{5.1}$$

Remember: Ideally we would like to find a minimiser over the full risk $\widehat{\Phi}$, i.e., over infinite data pairs to provide the best available approximation error. However, due to finite data we only have access to the empirical risk Φ which additionally limits approximation properties of the found estimator Λ_{θ^*}. We refer to this discrepancy as the *training error*.

Gradient descent

- For neural networks this optimisation is widely performed by first-order methods and in particular variations of gradient descent, where the gradient is computed with respect to the parameters.
- A gradient descent scheme for eq. (5.1) is given as

$$\theta_{i+1} := \theta_i - \eta \nabla_\theta \Phi(\Lambda_\theta(\cdot)),$$

- $\eta > 0$ is a step-length, often referred to as the *learning rate*.
- We need to compute the gradient $\nabla_\theta \Phi(\Lambda_\theta(\cdot))$ with respect to all samples in the training data $\{v^{(i)}, u^{(i)}\}_{i=1}^N$.

Stochastic gradient descent

- Computing the full gradient is computationally intractable for large amounts of data. (large sets of high resolution CT scans)
- Empirical evidence suggests that basic gradient descent tend to get stuck in sub-optimal local minima.
- Consequently, we consider stochastic variants, we compute updates as

$$\theta_{i+1} := \theta_i - \eta g_i,$$

  where g_i represents a descent direction with respect to only a subset of the data.
- The samples (training pairs) over which the gradient is computed are chosen randomly.
- Under suitable random sampling one can then show that in expectation we indeed obtain the gradient at the i-th iteration

$$\mathbb{E}[g_i|\theta_i] = \nabla_\theta \Phi(\Lambda_{\theta_i}).$$

- In practice, to compute the directions g_i so-called *batches* of several training pairs are chosen, instead of single samples.

A usual convergence result

The usual convergence results we obtain for stochastic gradient descent provide convergence bounds in expectation. In the simplest case assuming differentiability (in the nonlinearity) and convex Φ, we obtain a convergence rate to the optimal parameter θ^* as

$$\mathbb{E}\big[\Phi(\Lambda_{\overline{\theta}})\big] - \Phi(\Lambda_{\theta^*}) \leq \frac{1}{\sqrt{T}} \text{ where } \overline{\theta} = \frac{1}{T}\sum_{i=1}^{T} \theta_i \tag{5.2}$$

- The convergence rate follows $1/\sqrt{T}$ with number of iterations. This is not particularly encouraging, but computation of the direction g_i is assumed to be fast.
- Keep in mind that we do not expect to compute the best approximator possible (only access to the empirical risk), and there is no need to iterate until convergence in practice.

Some remarks on generalisation

Generalisation is difficult to study, there are a few attempts but mostly limited to classification problems:

- Margin maximisation and implicit bias - assumes separable data (no regression problems).
- Rademacher complexity measures generalisation bounds depending on network size (with very pessimistic bounds)

Exception: The work (de Hoop, Lassas, Wong, 2022) does provide a result on approximating nonlinear operators in inverse problems with neural networks and provides (pessimistic) complexity dependent bounds.
$\Rightarrow$ Most generalisation results are empirical, but show good results on regression tasks.