Lecture 1a: Approximation and expressivity of NNs

Chris Budd and Aengus Roberts[1]

[1]University of Bath

CWI, October 2025

Some papers/books to look at

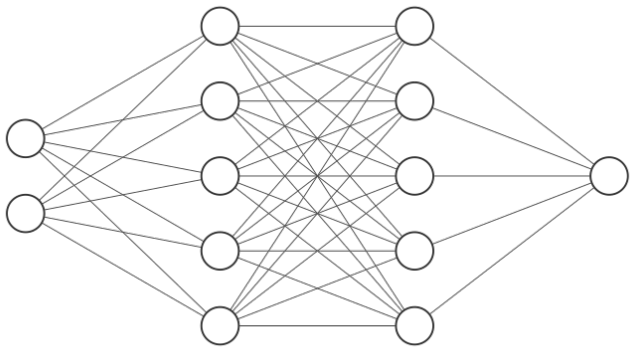
- Grohs and Kutyniok, *Mathematical aspects of deep learning*, (2022)
- Korolev, *Approximation properties of two-layer neural networks with values in a Banach space*, https://people.bath.ac.uk/td314/m4dlworkshop/YuryKorolev.pdf
- Appela, B et. al. *Equidistribution-based training of Free Knot Splines and ReLU Neural Networks*
- López *Breaking the curse of dimensionality with Barron spaces* https://dcn.nat.fau.eu/breaking-the-curse-of-dimensionality-with-barron-spaces/

Overview

All of Scientific Machine Learning is based on Neural Networks of some architecture. Regardless of the architecture these all have one thing in common

A Deep Neural Net (NN) of width W and depth L gives a nonlinear functional approximation to $u(\mathbf{x})$ with input $x \in R^n$ (or even $x \in H^1$) given by

$$y(\mathbf{x}) = DNN(\mathbf{x}) \in R^m$$



Big Question

To understand the use of NN's to solve (for example) differential equations, we must first understand how well they can approximate a function

A NN of width W, depth L, and activation functions σ with parameter set θ describes a set of functions. These lie in a nonlinear space Y.

Suppose that we have a function $u(x)$ that we want to approximate. Have two questions

1. Expressivity: What is the best approximation $y^* \in Y$ that we can find for u:
2. Trainability: Can we train the NN, i.e. find the optimal θ^*
   - At all?
   - In a reasonable computational time?

We have good answers to Question 1, and few answers to Question 2!

$y(\mathbf{x})$ is constructed via a combination of linear transformations and nonlinear/semi-linear activation functions.

Example: Shallow 1D neural net

$$y(\mathbf{x}) = \sum_{i=0}^{W-1} c_i \sigma(a_i \mathbf{x} + \mathbf{b}_i)$$

Often take

$$\sigma(z) = \text{ReLU}(z) \equiv z_+,$$

Then

$$y(\mathbf{x}) = \sum_{i=0}^{W-1} c_i (a_i \mathbf{x} + \mathbf{b}_i)_+$$

In 1-dimension this is a **piece-wise linear function** with N **free knots** at

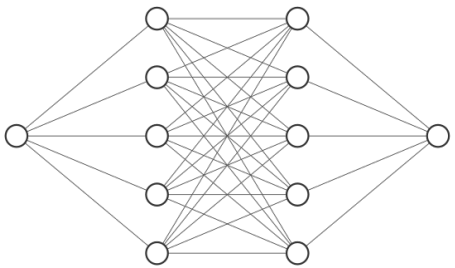
$$k_i = -b_i/a_i.$$

Expressivity 1

Convergence of a NN relies on understanding both approximation and training

A feed-forward Deep Neural Network (DNN) can be 'in principle' trained to approximate a target function $u(x)$

$$y_{j+1} = \Sigma_{i=0}^{W-1} c_{i,j}\, \sigma(a_{i,j} y_j + b_{i,j}) \quad j = 1, \cdots, L \quad y_0 \equiv x, \quad y(x) = y_L$$



Expressivity 2

There are a lot of results on the **theoretical expressivity** of a DNN [Grohs and Kutyniok, Mathematical aspects of deep learning, (2022)].

Universal approximation theorem, [Hornick et. al., 1989] *If* $u(\mathbf{x})$ *is a* continuous function *with* $\mathbf{x} \in K \subset R^d$ *(compact), and* σ *a continous activation function. Then for any* $\epsilon > 0$ *there exists a (shallow) neural network* $y(\mathbf{x}, \theta)$ *such that*

$$\|u(\mathbf{x}) - y(\mathbf{x})\|_\infty < \epsilon.$$

Expressivity 3

Theorem [Yarotsky (2017)]

Let

$$F_{n,d} = \{u \in W^{n,\infty}([0,1]^d) : \|u\|_{W^{n,\infty}} \leq 1\}$$

For any d, n and $\epsilon \in (0,1)$ there is a ReLU network architecture that

1. *Is capable of expressing any function from $F_{d,n}$ with error ϵ*
2. *Has depth*

$$\textit{depth:} \quad L < c(\log(1/\epsilon) + 1) \quad \textit{width:} \quad W < c\epsilon^{-d/n}(\log(1/\epsilon) + 1)$$

for some constant $c(d,n)$.

Implies exponential rates of convergence with increasing depth L. Note the importance of the dimension d

Can get very precise estimates which overcome problems of dimensionality using Barron Spaces. See the references

But how well do we do in practice?

Other approximation methods: Linear splines

Express $u(x)$ as:

$$u(x) = \sum_{i=0}^{N} a_i \, \phi_i(x).$$

Here $\phi_i(x)$ are **known functions**. eg. B-splines (Finite element), Fourier modes, Wavelets etc.

Advantages: Linear in a_i, Convex optimisation problem, easy to use, guaranteed error bounds

Disadvantages: Poor expressivity, slow convergence as N increases, results depend on the form of the function $u(x)$ (hard to approximate rapidly changing functions), Curse of dimensionality

Example: Direct Learned Univariate NN Function Approximation

For a learned function $y(x)$ approximate target function $u(x)$ by minimising the 'usual' loss function L over parameters θ

$$\min_{\theta} \; L(\theta) \equiv \sum_{k=1}^{M} |y(X_k) - u(X_k)|^2$$

Use the shallow ReLU network

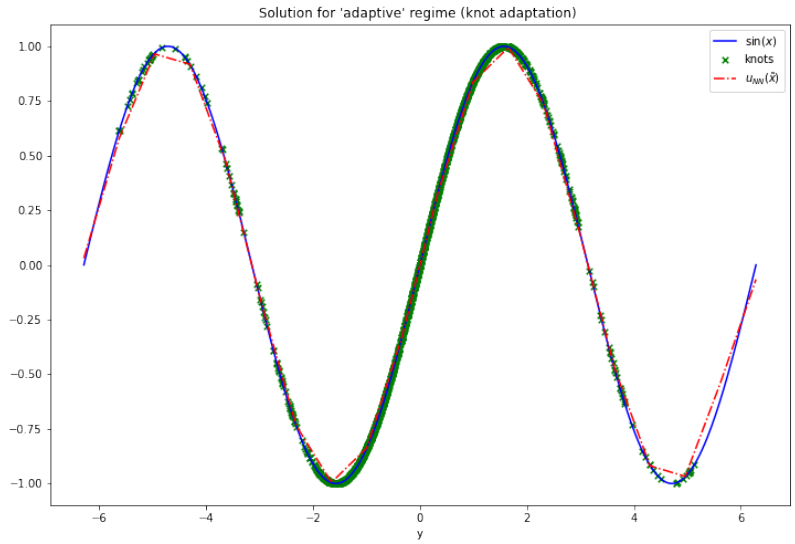
$$y(x) = \sum_{j=0}^{W-1} c_j (a_j x + b_j)_+, \quad \theta = [\mathbf{a}, \mathbf{b}, \mathbf{c}].$$

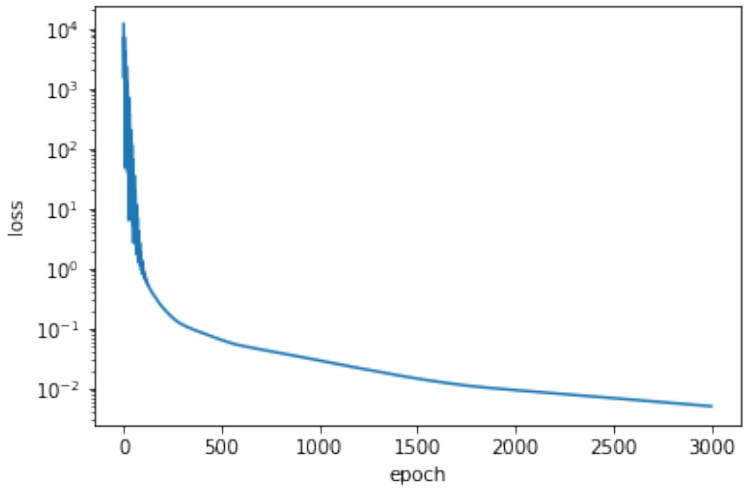
Find the optimal set of coefficients a, b, c

Use an ADAM SGD (over the quadrature points) optimiser

Approximation of: $u(x) = \sin(x)$ using ReLU network

Uniform quadrature points:

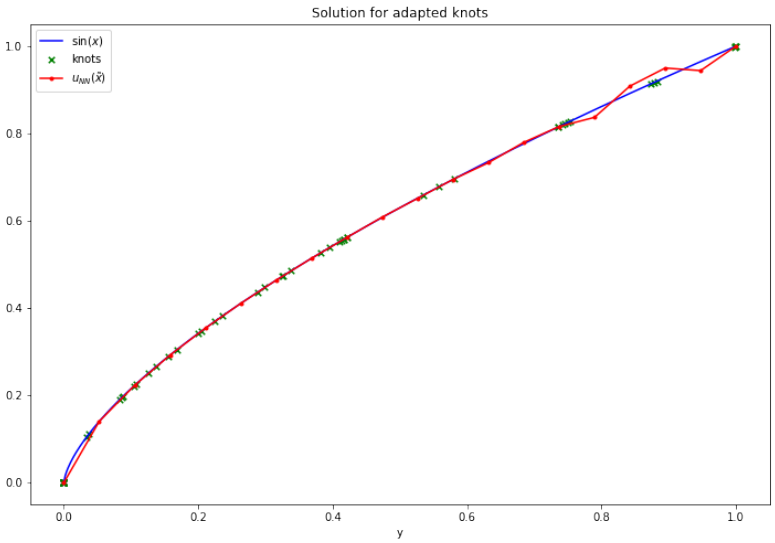


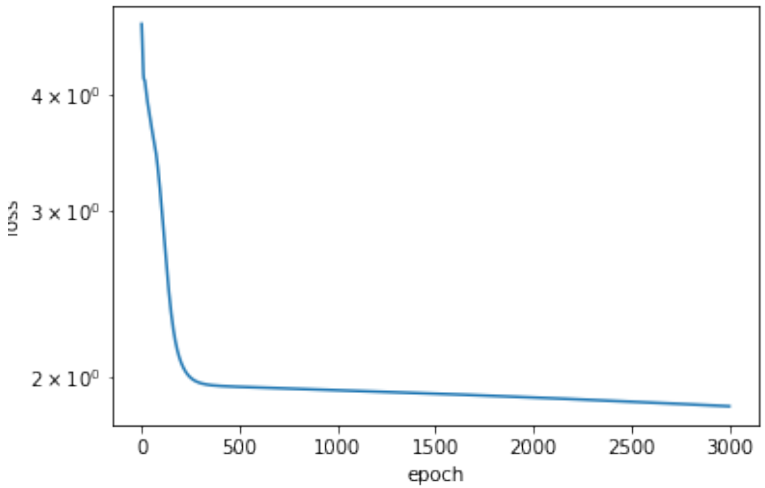


Results poor. Depend crucially on the starting values. Even then poor.

Approximation of: $u(x) = x^{2/3}$

Uniform quadrature points:





Results even worse! Depend crucially on the starting values. Even then very bad!

Problems with:

- the loss function,
- training,
- and conditioning,

of the ReLU NN.

REASON: Trying to train a_i, b_i, c_i together, whereas they play very different role in the approximation.

This leads to a very non-convex and ill conditioned optimisation problem. Either very slow convergence to the optimum approximation, or convergance to a (very) sub-optimal solution

RESOLUTION:

- **First** solve the *nonlinear* problem of finding nearly optimal (knots) a_i, b_i first
- **Then** Solve the *nearly linear but ill-conditioned* problem of finding the (weights) c_i next by *pre-conditioning the system*
- **Iterate** if needed (it's not needed)

Compare with other univariate approximation methods:

- Linear spline

$$y(x) = \sum_{i=0}^{N} w_i \, \phi_i(x - X_i), \quad X_i \quad \text{fixed}$$

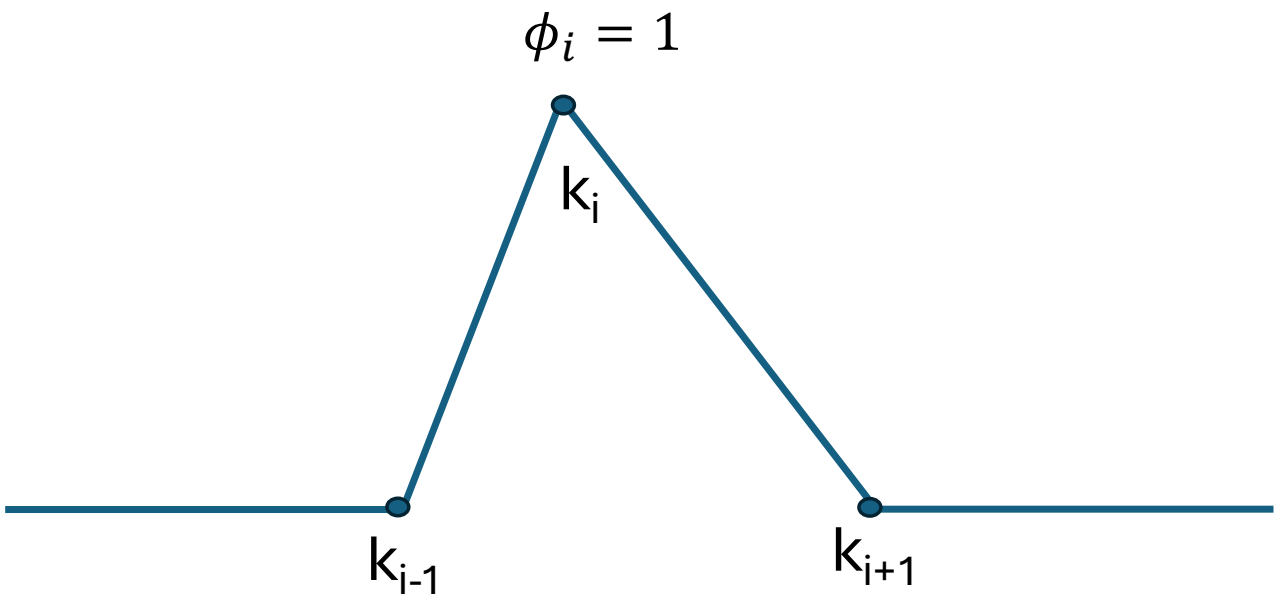
$$w_i \quad \text{to be determined} \quad \text{Linear}$$

- Free Knot Spline (FKS)

$$y(x) = \sum_{i=0}^{N} w_i \, \phi_i(x - k_i).$$

Knots k_i also to be determined Nonlinear

Basic linear spline on free knots



Comparison between approximation methods

Linear spline

- Less expressive (lower accuracy for a given N)
- Non-adaptive (problems with singular functions)
- Not Equivariant (fixed mesh imposes a-priori structure)
- Good error estimates
- **Linear space:** Essential to the use of the FE method

FKS/NN

- Very expressive (high accuracy for a given N)
- Self-adaptive (can work for singular functions, adaptive FE)
- Equivariant (eg. to Galilean transformations, scalings)
- Good error estimates
- **Nonlinear space:** Addition of two FKS with N knots is not a FKS with N knots

ReLU NN and Free Knot Splines (FKS)

[deVore] Any 1D ReLU network of width W and depth L is formally equivalent to a piecewise linear free knot spline (FKS) with N free knots k_i, where

$$N \leq W^L,$$

$$y(x) = \sum_{i=0}^{N} w_i \, \phi_i(x - k_i).$$

Note usually $N \ll W^L$.

Proof ReLU $\implies$ FKS

The combination of two piecewise linear functions is a piecewise linear function. Therefore a ReLU network is a piecewise linear function with knot points that depend in a subtle way on the coefficiensts of the network.

Proof FKS $\Longrightarrow$ ReLU

The linear spline functions $\phi_i(z)$ can be expressed as a combination of ReLU functions with coefficients *nonlinear functions* of k_i

$$\phi_i(z) = \frac{ReLu(z-k_{i-1})}{k_i - k_{i-1}} - \frac{(k_{i+1}-k_{i-1})\,ReLu(z-k_i)}{(k_{i+1}-k_i)(k_i-k_{i-1})} + \frac{ReLu(z-k_{i+1})}{k_{i+1}-k_i}.$$

Therefore

$$\sum w_i\,\phi_i(x-k_i) = \sum c_i\,ReLu(x-k_i)$$

$$c_i = \frac{w_{i-1}}{k_i - k_{i-1}} - \frac{(k_{i+1}-k_{i-1})w_i}{(k_{i+1}-k_i)(k_i-k_{i-1})} + \frac{w_{i+1}}{k_{i+1}-k_i} \approx \frac{k_{i+1}-k_{i-1}}{2} w_i''.$$

SO .. in principle a ReLU network has the same expressivity as a FKS and vice-versa

But this often does not seem to be the case in practice when training the network

We will explore

- The theoretical and practical expressivity of the FKS
  Correct loss function and train knots then weights
- Whether the ReLU network can be **trained stabily and efficiently** to give the same result as the FKS! Pre-conditioning
- Extensions to deep networks and higher dimensions

Analysis of a FKS

Can compare ReLU expressivity results to those of

- Using a **free knot linear spline** (FKS):

$$y(x) = \sum_{i=0}^{N} w_i \, \phi_i(x - k_i).$$

  Optimise over $\theta = [\mathbf{w}, \mathbf{k}]$

- Using an **interpolating free knot linear spline** (IFKS):

$$y(x) \equiv \Pi_1 u(x) = \sum_{i=0}^{N} u(k_i) \, \phi_i(x - k_i).$$

  Optimise over $\theta = \mathbf{k}$

Expressivity of a (I)FKS

The expressivity of a (I)FKS can be very good. Full theory [see deVore (1998),(2022)] uses results from Besov spaces.

In summary: for both regular (eg. $\sin(x)$) and quite irregular target functions $u(x)$ (eg. $x^{2/3}$) expect to see

$$\|u - y_{FKS}\|_p < C_{FKS}/N^2, \quad p = 2, \quad \infty$$

$$\|u - y_{IFKS}\|_p < C_{IFKS}/N^2, \quad p = 2, \quad \infty$$

With $C_{FKS} < C_{IFKS}$ but with very similar knot locations

A little theory on linear spline IFKS

If we know $u''(x)$ we can develop a theory for the best knot location for an IFKS (and hence for a FKS). [Huang + Russell (2010)]. Using the local interpolation over the knot intervals we have:

$$\|u - y_{IFKS}\|_2^2 = \frac{1}{120}\sum_i (k_{i+1} - k_i)^5 (u''(\xi_i))^2, \quad \xi_i \in (k_i, k_{i+1}).$$

$$\|u - y_{IFKS}\|_\infty = \frac{1}{4}\max_i (k_{i+1} - k_i)^2 |u''(\xi_i)|, \quad \xi_i \in (k_i, k_{i+1}).$$

Equidistribution

It can be shown [de Boor, Huang + Russell] that the *best knot location* for the IFKS **equidistributes** $\rho_{i+1/2}$ given by:

$$\rho_{i+1/2} = (k_{i+1} - k_i)(u'')^{2/5}, \quad p = 2$$

or

$$\rho_{i+1/2} = (k_{i+1} - k_i)(u'')^{1/2}, \quad p = \infty$$

Motivates the equidistribution loss function to train the **knots** of the IFKS:

$$L_E(\mathbf{k}) = \sum(\rho_{i+1/2} - \sigma)^2, \quad \sigma = mean(\rho).$$

Some analytical results

Can sometimes solve the equidistribution equations by quadrature (and a bit of care at the singularity). Useful for comparison with the numerical computations.

Set $x = x(\xi)$, with $x \in [0,1], \xi \in [0,1]$ and set

$$k_0 = 0, k_N = 1, \quad k_i = x(i/N).$$

Then

$$\rho_{i+1/2} \approx \frac{1}{N} x_\xi \left(u''\right)^{2/5} = \sigma \quad (*)$$

Solve (*) and apply the boundary conditions to find x and σ

Example: $u(x) = x^{2/3}$.

$$u_{xx} = -(2/9)x^{-4/3} \implies \frac{1}{N}(2/9)^{2/5} x_\xi \, x^{-8/15} = \sigma.$$

Solve and apply the BC to give *optimal knots*

$$x = \xi^{15/7}, \quad k_i = (i/N)^{15/7}.$$

With optimal L_2^2 error:

$$L_2^2 = \sum_{i=0}^{N-1} \frac{1}{120} \frac{1}{N^5} x_\xi^5 (u'')^2 = \frac{1}{120N^5} \sum_{i=0}^{N-1} \left(\frac{15}{7}\right)^5 \frac{4}{81} = \frac{1.859 \times 10^{-2}}{N^4}.$$

NOTE The L_2^2 error for a PWL interpolant with *uniform knots* has:

$$L_2^2 = \frac{C}{N^{7/3}}.$$

Comparison of the splines for $x^{2/3}$

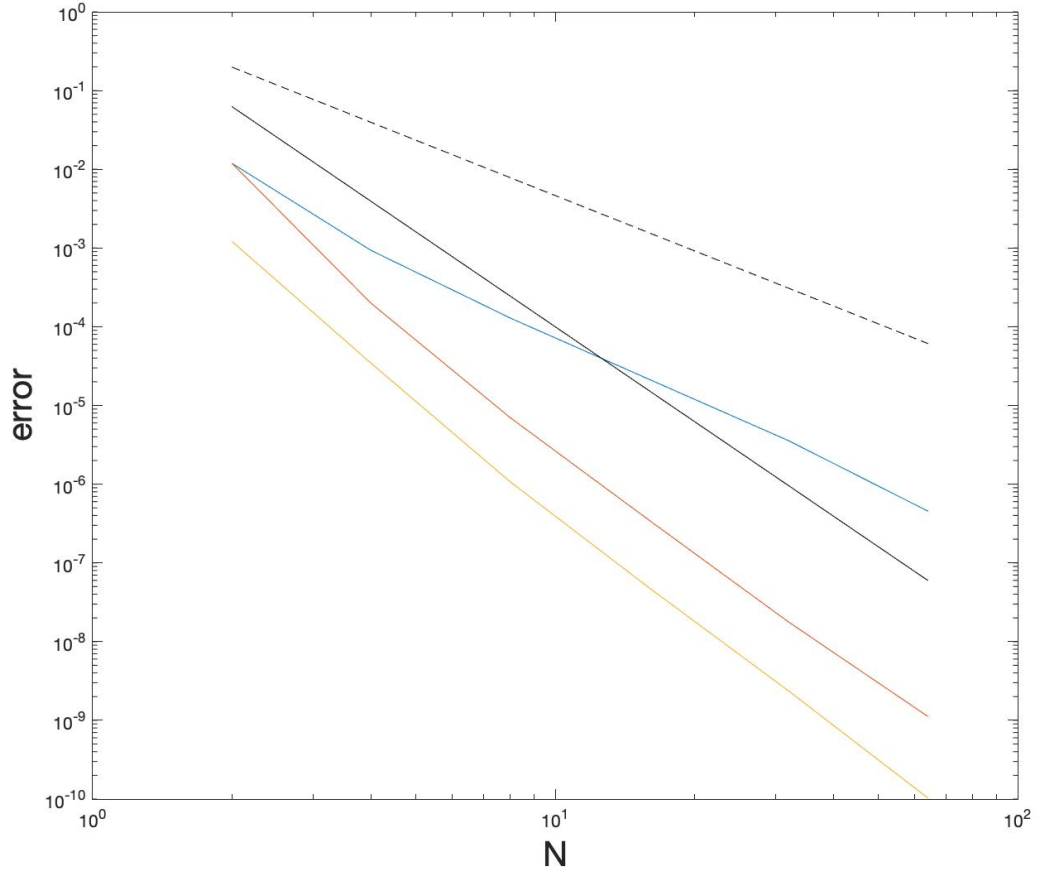


Figure: (Red) Optimal IFKS, (Yellow) Optimal FKS, (Blue) Uniform interpolating spline, (Black) $1/N^4$, (Black dashed) $1/N^{7/3}$

Approximation form for $u(x) = x^{2/3}$ (singularity at $x = 0$)

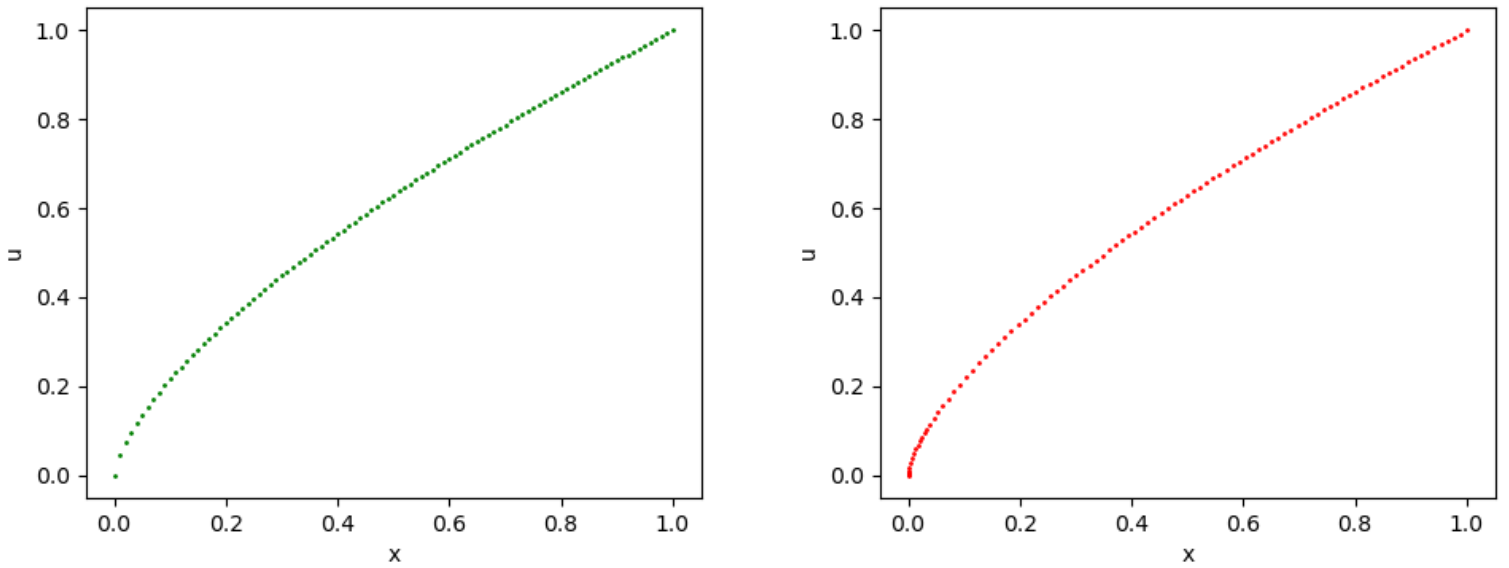


Figure: Comparison between uniform and optimal spline for $N = 100$. Note that the optimal knots cluster towards $x = 0$, where the solution exhibits a singular behaviour.

Two-level training for a general FKS

Find knots a_i, b_i then the weights c_i

- Construct a *mixed* loss function

$$L = \lambda L_2^2(\theta) + (1-\lambda) L_E(\mathbf{k}).$$

- From a set of uniform knots $k_i = i/N$ set $\lambda = 0.1$ and train to find optimal IFKS knots and y_{IFKS}
- Set $\lambda = 0.9$ and use Adam to optimise y_{FKS} from y_{IFKS} by finding optimal *weights*
  Almost linear and well-conditioned as IFKS knots close to optimal FKS knots

Results : $W = 30$ training the knots

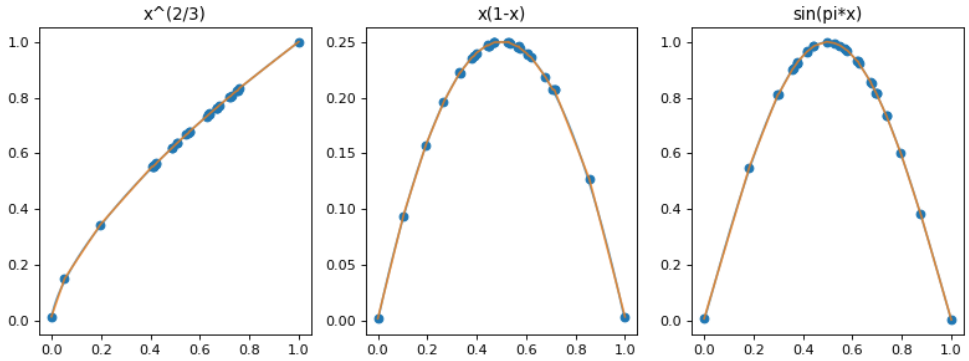


Figure: L_2^2 Loss function

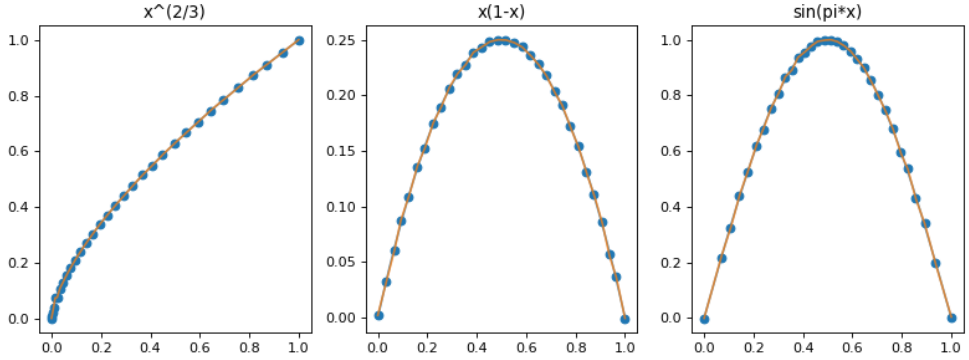


Figure: Mixed loss function

Results $W = 30$: Knot evolution during training

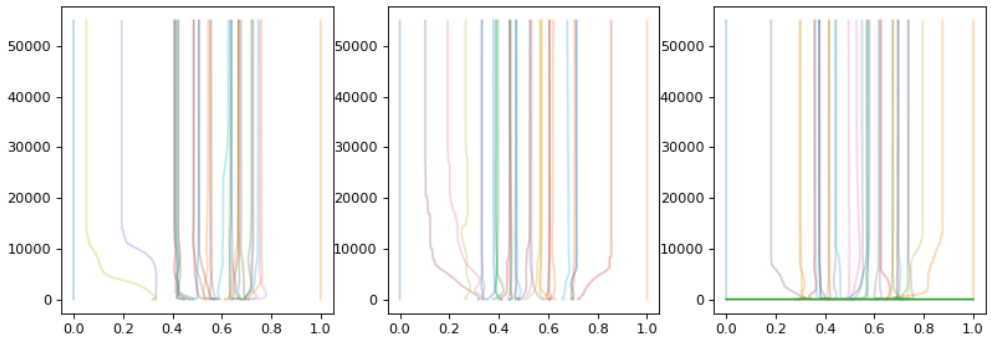


Figure: L_2^2 Loss function. Note knot coalescence and irregularity

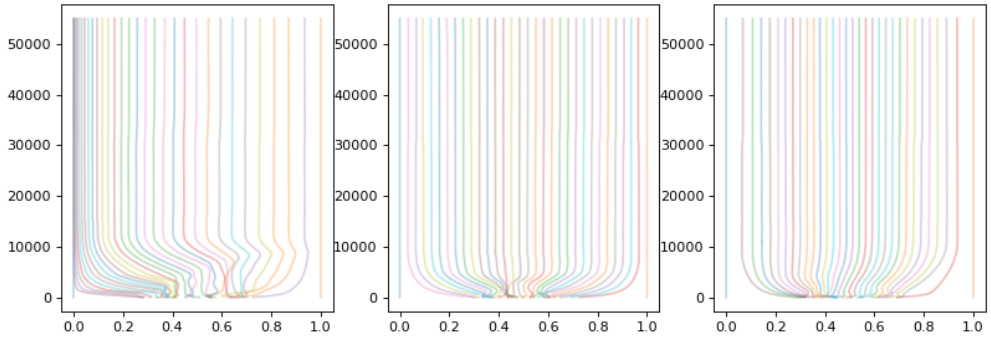


Figure: Mixed loss function

Training a shallow ReLU NN

Motivated by this we can modify the training of a *shallow* ReLU NN.

- Take shallow ReNN with parameters $\theta = [\mathbf{a}, \mathbf{b}, \mathbf{c}]$
- Compute the implicit knot locations $\mathbf{k}$
- Construct a *mixed* loss function

$$L = \lambda L_2^2(\theta) + (1-\lambda) L_E(\mathbf{k}).$$

- Use two-layer training as above to find the **knots** and then the **weights/coefficients**

This improves the training results for the ReLU NN knot location, but there are still strong ill-conditioning problems with finding the ReLU weights c_i

Comparison between the ReLU NN and the FKS training

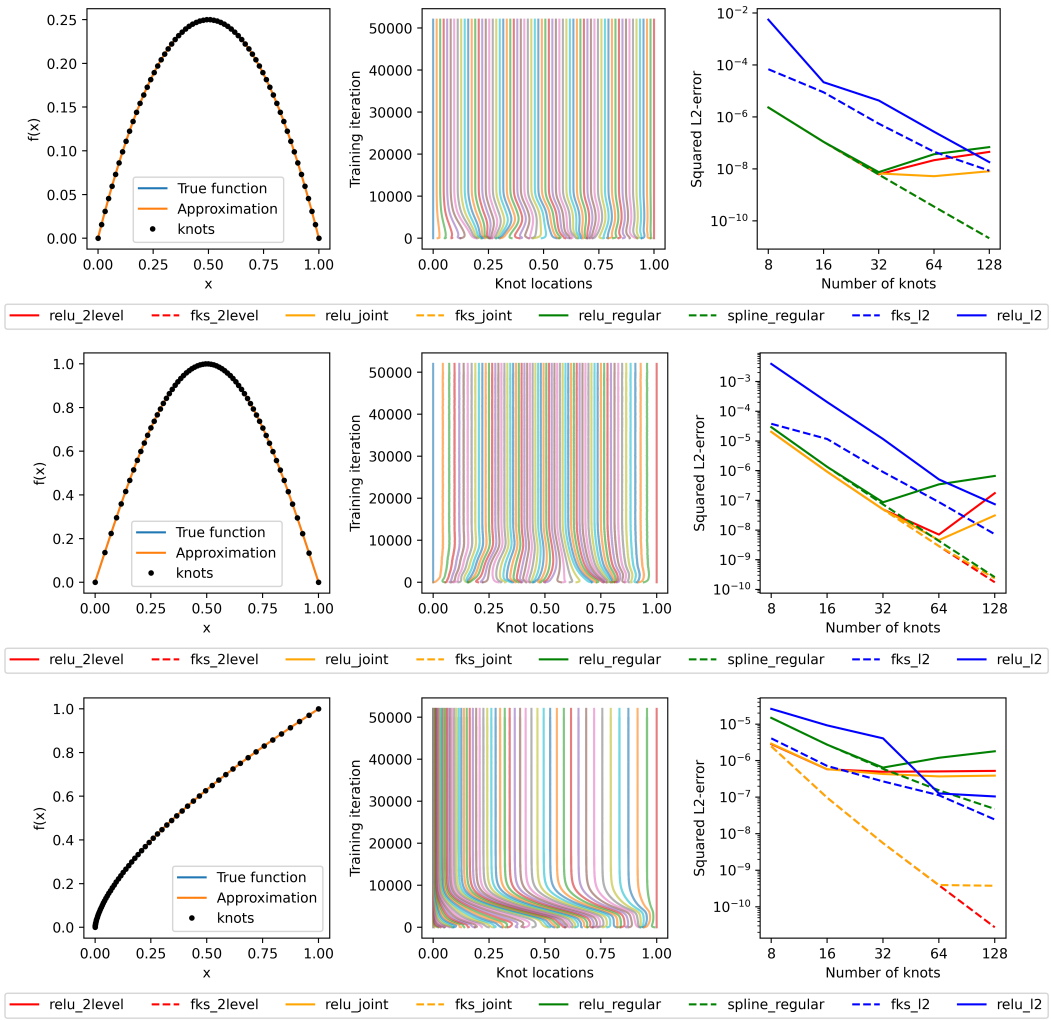


Figure: $u(x) = x(1-x), \sin(\pi x), x^{2/3},$

Comparison between the ReLU NN and the FKS II

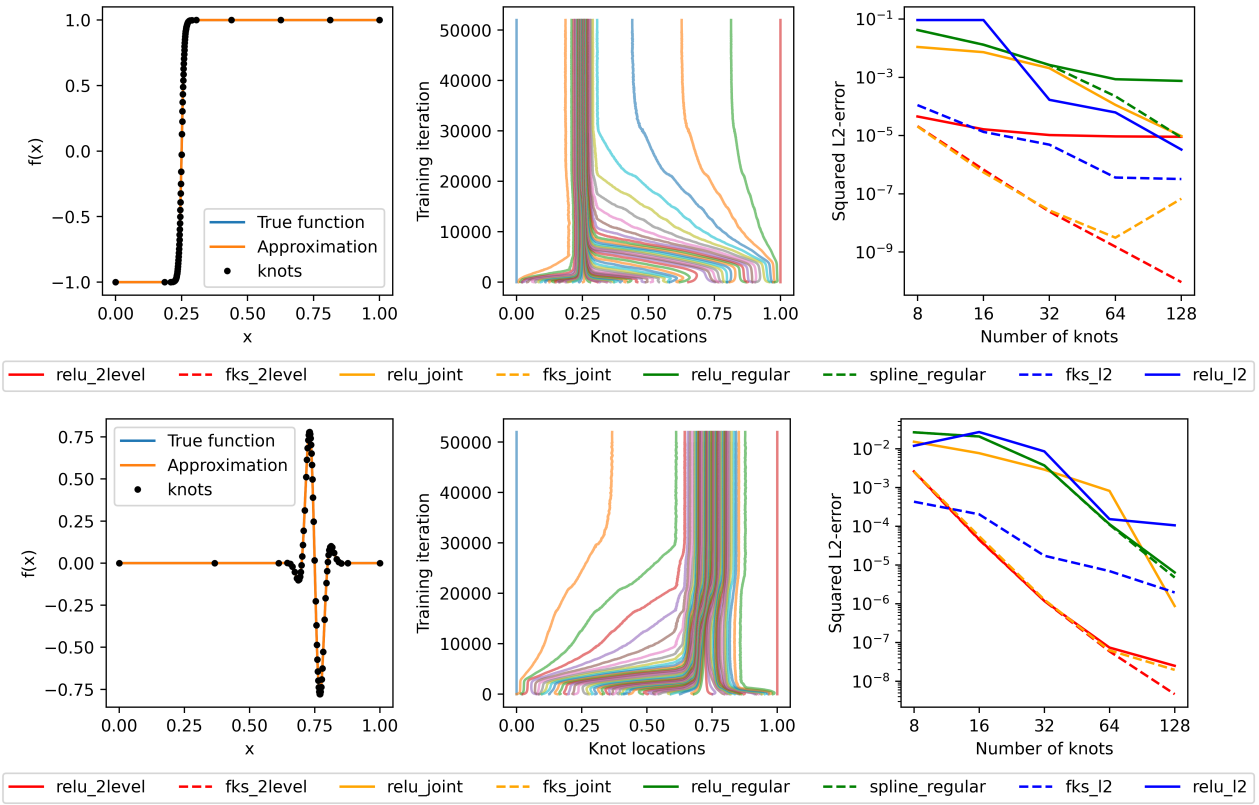


Figure: $u(x) = \tanh(100(x - 1/4), \exp(500(x - 3/4)^2)\sin(20\pi x)$.

It is clear that the FKS significantly outperforms the ReLU NN

Timing and ill-conditioning

WHY: The training of the ReLU NN network is **much slower** than for the FKS due to the ill conditioning of the ReLU coefficients c_i (high condition number of the matrices) compared to the weights w_i of the FKS.

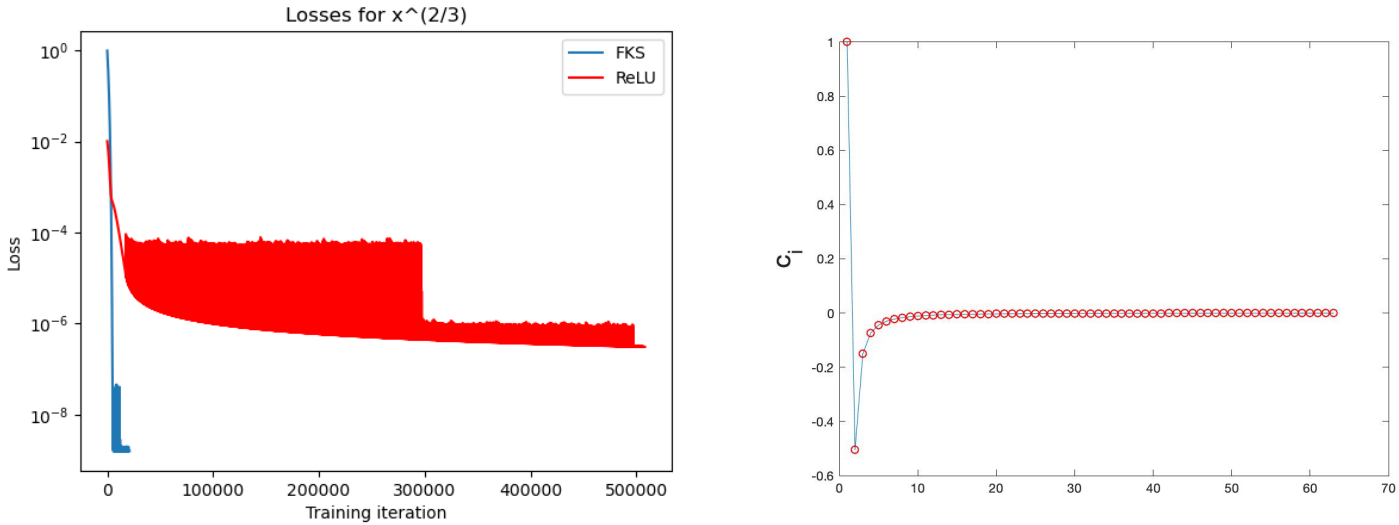


Figure: Two-level training ReLU NN and FKS for $u(x) = x^{2/3}$ (left) timings (right) ReLU coefficients.

Conditioning and pre-conditioning

Can establish the following

Theorem *Given known N knots, the condition number of the normal equation matrices solving for the weights is*

$$\mathcal{O}(1) \quad \text{for FKS architectures}$$

$$\mathcal{O}(N^4) \quad \text{for ReLU architectures}$$

Conclusion: To get optimal expressivity for ReLU training

- Train the knots
- Pre-condition the ReLU NN to have same architecture as the FKS.

Then get NN expressivity equivalent to that of the FKS

Other activation functions

The L_2^2 error of directly training a shallow network with the tanh activation function, without preconditioning, to approximate the test functions $u_i(x)$.

N	u_1	u_2	u_3	u_4	u_5	P
8	4.6×10^{-9}	1.70×10^{-6}	3.55×10^{-4}	1.08×10^{-3}	1.77×10^{-2}	25
16	1.56×10^{-9}	1.20×10^{-8}	1.62×10^{-7}	8.01×10^{-4}	1.53×10^{-2}	49
32	1.21×10^{-8}	1.41×10^{-8}	1.50×10^{-7}	6.17×10^{-4}	1.64×10^{-2}	97
64	5.25×10^{-9}	8.88×10^{-9}	2.15×10^{-7}	5.54×10^{-4}	1.77×10^{-2}	193
128	2.21×10^{-8}	8.00×10^{-9}	2.15×10^{-7}	4.40×10^{-4}	1.90×10^{-2}	385

Error is better than the ReLU NN as the system is much better conditioned. But still not as good as the two-layer trained FKS

Deep networks

Consider 2-layer neural network, with ReLU activation, width $W = 5$ and 46 trainable parameters.

	u_1	u_2	u_3	u_4	u_5
L_2^2 error	4.2×10^{-3}	5.47×10^{-2}	3.45×10^{-2}	1.8×10^{-1}	1.85×10^{-2}
Variance	9.71×10^{-5}	1.42×10^{-2}	1.35×10^{-2}	5.40×10^{-2}	1.09×10^{-4}

Compare with FKS with $N = 20$ knots and 42 trainable parameters.

FKS error of 10^{-6} or less on each of these problems.

Higher dimensions

A 2D ReLU is a piecewise-linear function of (x, y). However, the mesh on which it is defined is very irregular. This makes any comparison very hard.

Note that this mesh would in general be poor for a finite element or similar approximation!