Deep coordinate networks can outperform 2D Shannon interpolation

A comparison of recent and classical methods on the task of image interpolation

Alex Mackay

June 13, 2023

Thesis submitted for the degree of Honours in Mathematical Sciences

The University of Adelaide
Faculty of Sciences, Engineering and Technology
School of Computer and Mathematical Sciences

Contents

Declaration of authorship

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint award of this degree.

I give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Alex Mackay

June 13, 2023

Acknowledgements

Firstly, I would like to thank my supervisors Dr Lewis Mitchell and Dr Simon Lucey for their support, knowledgeable advice, and insightful discussions throughout the past year.

I would also like to thank Dr Hemanth Saratchandran, who has been closely involved with the project and provided valuable mathematical consulting.

Finally, I would like to thank my partner, my friends, and my family, for their encouragement and support.

Abstract

A neural network can be trained to memorise samples from a single low-dimensional signal, such as an image. The resultant network provides a form of interpolation between the sampled coordinates.

Such "coordinate-MLPs" perform better when the input coordinates are first pre-processed by applying a positional encoding function. Previous work has shown that trigonometric and Gaussian functions work well for this task. These functions play a similar role in classical interpolation methods.

Coordinate-MLPs have been applied to a variety of tasks, but there is not yet a rigorous mathematical foundation for understanding their strengths and weaknesses.

Here we show that coordinate-MLPs can outperform 2D Shannon interpolation when applied to natural imagery, in a way that is closely related to the number of parameters required for each method.

The number of parameters required is, in turn, dependent on the dimensionality of the signal being processed. This link provides a possible explanation for why neural networks perform particularly well on high-dimensional data compared to other methods.

These results give some indications as to when coordinate networks may perform well, allowing us to find applications in which they could used effectively.

More broadly, a better understanding of the factors affecting neural network performance, such as dimensionality of data, may prove useful for determining which tasks are well-suited for machine learning methods, working towards a rigorous mathematical underpinning for deep learning.

Chapter 1

Introduction

Machine learning research has made great strides in the past decade, providing tools for solving many problems that were previously intractable.

In particular, the relatively new field of deep learning has shown that deep neural networks are excellent at finding patterns in high-dimensional data-sets. However, current deep learning methods are largely stochastic and heuristic in nature, and lack a comprehensive mathematical foundation characterising their behaviour and capabilities.

In stark contrast, classical interpolation is a mature field, with some key results dating back over a century. Many tasks involving low-dimensional signals have deterministic, optimal, closed-form solutions. In particular, Shannon's seminal paper *Communication in the presence of noise*[12] characterises how a class of 1D signals can be reconstructed perfectly given only a discrete set of samples.

Despite these differences, both fields provide valuable tools for the task of interpolating a signal, whether low- or high-dimensional. By applying methods from both fields to the same task, perhaps our solid understanding of classical interpolation can be used to give us a better understanding of deep learning.

In deep learning, the task of interpolation is not always entirely straightforward. The simplest deep learning architecture is the multilayer perceptron (MLP). When we apply one directly to the task of low-dimensional interpolation, performance is poor. The reason: an empirically observed phenomenon called "spectral bias". Roughly speaking, MLPs tend to learn each frequency component of a signal at a rate inversely proportional to the frequency. As a result they will effectively ignore very high frequencies, unless trained for an impractically long time.

In 2021 Mildenhall et al. published *NeRF: Representing scenes as neural radiance fields for view synthesis*[8], in which they use a "positional encoding" step to transform the high-frequency components of a 5D signal into lower-frequency components of a signal with many more dimensions. With this encoding, an MLP was able to learn the signal and achieve state-of-the-art results. Later works[13][15] have refined the positional encoding step and explored different encoding functions. An MLP preceded by a positional encoding is now known as a coordinate-MLP.

A recent paper by Zheng et al.[17], explores some new approaches to positional encoding. With these new ideas, coordinate-MLP and classical interpolation methods begin to converge. There are two parts to this.

Firstly, Zheng et al. use positional encodings composed of "shifted basis functions", analogous to a classical approach described by Unser[14], of using many shifted copies of a "generating function".

Secondly, Zheng et al. propose a "complex encoding" analogous to the classical method of tensor product splines. The resultant encodings are so good, in fact, that it is no longer necessary to follow them by a deep MLP. Instead, the encodings can be followed by single linear layer, which can be trained in a fraction of the time required for an MLP.

With these new positional encoding concepts, we can faithfully implement classical interpolation within the same architectural framework as deep coordinate-MLPs. With this common framework we can ensure that most factors are kept constant, allowing for a meaningful comparison of classical interpolation and deep learning methods.

In this thesis, we compare the performance of classical interpolation and coordinate-MLPs. Chapter 2 reviews classical interpolation, with a focus on Shannon interpolation, generating functions, and extension to multidimensional signals. Chapter 3 reviews neural networks, coordinate-MLPs, and the approaches introduced by Zheng et al.. Chapter 4 describes experiments that we performed on 1D and 2D signals, and what we might expect in higher dimensions. Finally, we present our conclusions in Chapter 5.

Chapter 2

Classical interpolation

Many phenomena can be modelled as **signals** — functions describing how some quantity varies over space, time, and perhaps other continuous dimensions. Some prototypical examples of signals include audio, radio waves, and electrical currents.

Not all signals involve a quantity that varies over time. In particular, Chapter 4 is largely concerned with modelling *images* as signals. In that case, the signal is a function that varies over two *spacial* dimensions (the coordinates of pixels on a screen).

In the real-world it is typically impossible to have perfect knowledge of any given signal of interest. Instead, we use some physical device to obtain a set of **samples**, which tell us the value of the signal at a discrete set of coordinates.

For many purposes we would like to know the signal value at *any* coordinate, not just those that we have samples for, so we would like to recover the original signal, or something like it. **Interpolation** is the task of reconstructing a signal, given samples from it. A signal reconstruction rarely matches the original signal exactly, but we can at least aim for a reasonable approximation.

The main applications of interpolation are in the field of signal processing, but it is an invaluable tool in any field that involves taking samples along some continuous dimensions.

2.1 Time signals, frequencies, and Shannon

We start with the conceptually simplest signals — time signals of infinite duration.

2.1.1 Signals

We model a signal as a function,

$$f : \mathbb{R} \to \mathbb{R}.$$

The signal takes $x \in \mathbb{R}$ as the input **coordinate**, and $y = f(x)$ is the output **value** of f at coordinate x. Signals are continuously-defined (as opposed to discrete), but they need not be continuous in the real analysis sense.

In practice, many signals have multiple channels. For example, audio is often recorded with two channels (left and right) and image files typically have three channels (red, green, and blue). A more general model of a signal might output a value in $\mathbb{R}^C$, where C is the number of channels. In many cases, a signal's channels are each processed independently, so treating them as entirely separate signals may be reasonable. For simplicity, we only consider single-channel signals.

2.1.2 Samples

We obtain information about a signal by **sampling** it. Before we do so, we need a set of **sample coordinates**.

Regular sampling is the most common and straightforward way to select coordinates. In this case the sample coordinates are evenly spaced, separated by the **sampling interval** $T \in \mathbb{R}_+$. Using $\mathbb{Z}$ as an indexing set, the set of sample coordinates is given by

$$\mathbf{x} = \{\, x_k = Tk \mid k \in \mathbb{Z} \,\}.$$

We then obtain a **sample value** $y_k = f(x_k)$ for each sample coordinate x_k. We assume that this process is flawless, so the sample values perfectly reflect the signal they are taken from. We denote the set of sample values as

$$\mathbf{y} = \{\, y_k = f(x_k) \mid k \in \mathbb{Z} \,\}.$$

It would be reasonable to consider errors introduced by noise and other factors, however that is outside of the scope of this thesis. Also out of scope is regular sampling on non-Cartesian grids, and nonuniform sampling — an interesting and relatively recent branch of sampling theory.

2.1.3 Signal frequencies

Given a signal $f : \mathbb{R} \to \mathbb{R}$, how can we characterise it? The most common way to describe a signal is in terms of the frequencies it contains.

The Fourier transform[5] of a signal f is

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\xi x} \, dx.$$

If the time x is measured in seconds, then the frequency ξ is in hertz. The result of the transform, $\hat{f}$, is sometimes called the frequency domain representation, while the original signal is the time domain representation.

The process can be reversed by using the inverse Fourier transform, given by

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{i2\pi\xi x} \, d\xi.$$

A signal is said to be **band-limited** to a frequency B if its frequency domain representation contains no frequencies above B. That is, $\hat{f}(\xi) = 0$ for all $|\xi| > B$.

2.1.4 Shannon interpolation

In 1949, Claude Shannon published *Communication in the presence of noise*[12], which brings together several concepts which were "common knowledge in the communication art", and puts them on a rigorous mathematical foundation.

The Nyquist-Shannon sampling theorem: If a function $f(x)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced less than $\frac{1}{2B}$ seconds apart.

In other words, if a signal is band-limited to a frequency B, then a sampling interval of $T < \frac{1}{2B}$ is sufficient to perfectly reconstruct the signal. The method of reconstruction is given by the **Whittaker-Shannon interpolation formula**:

$$\tilde{f}(x) = \sum_{k \in \mathbb{Z}} y_k \operatorname{sinc}\left(\frac{x}{T} - k\right)$$

where "sinc" is the normalised sinc function, defined as

$$\text{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & : x \neq 0 \\ 1 & : x = 0 \end{cases} .$$

2.2 Generalised classical interpolation

This section adapts work by Unser[14] which describes a general interpolation model that includes Shannon interpolation as a special case. But first, we impose some additional constraints on our signals.

Herein we require signals to be square-integrable, that is, $f \in L_2$. We also require that sequences of sample values be square-summable, that is, $\mathbf{y} \in \ell_2$.

Recall that the Hilbert space L_2 consists of all functions that are square-integrable, and the corresponding L_2-norm is

$$\|f\| = \sqrt{\int_{-\infty}^{+\infty} |f(x)|^2 dx}.$$

Similarly, ℓ_2 consists of all sequences that are square-summable, and the ℓ_2-norm is

$$\|\mathbf{y}\| = \sqrt{\sum_{i=1}^{\infty} |\mathbf{y}(i)|^2}.$$

2.2.1 Approximation spaces

An interpolation method takes as input a sequence of sample values $\mathbf{y} \in \ell_2$, and produces a **signal reconstruction** $\tilde{f} : \mathbb{R} \to \mathbb{R}$. We can model this as a surjective map $I : \ell_2 \to V$, where $V \subseteq L_2$ is the method's **approximation space**, consisting of all possible signal reconstructions it can produce (given appropriate sample values as inputs). We then have $\tilde{f} = I(\mathbf{y})$.

We would like $\tilde{f}$ to be as close as possible to f, as measured by the L_2-norm. In other words we want $\tilde{f}$ to equal $\underset{g \in V}{\text{argmin}} \|f - g\|$. This may not be possible in general, since the interpolation method does not have direct access to the original signal f.

Instead we require that $\tilde{f}(\mathbf{x})$ be as close as possible to $f(\mathbf{x})$, as measured by the ℓ_2-norm. This seems like a reasonable requirement — an interpolation method should choose from its approximation space the function that most closely matches the

sample values. This can be expressed as

$$\tilde{f} = I(\mathbf{y}) = \underset{g \in V}{\operatorname{argmin}} \|\mathbf{y} - g(\mathbf{x})\|. \tag{2.1}$$

This is a convenient way to describe the ideal behaviour of an interpolation method, given only its approximation space V. We will now consider some ways to choose the approximation space.

2.2.2 Basis functions

Often our signal reconstructions are comprised of linear combinations of a set of **basis functions** $\{\varphi_k\}_{k\in\mathbb{Z}}$. In this case the approximation space is given by $V = \operatorname{span}\{\varphi_k\}_{k\in\mathbb{Z}}$ and reconstructions will have the form

$$\tilde{f}(x) = \sum_{k\in\mathbb{Z}} w(k)\varphi_k(x) \tag{2.2}$$

where the sequence of coefficients $w \in \ell_2$ will depend on the sample values.

In the case of Shannon interpolation, our basis functions are $\varphi_k(x) = \operatorname{sinc}(\frac{x}{T} - k)$. These functions form an orthonormal basis for their approximation space. This orthonormality property is part of what makes Shannon interpolation so amenable to analysis and implementation.

In the more general case, we do not require that our basis functions be orthonormal, but only that they form a Riesz basis of V. This will be the case if there exists positive constants $A, B \in \mathbb{R}_+$ such that for all $w \in \ell_2$,

$$A\,\|w\|^2 \leq \left\| \sum_{k\in\mathbb{Z}} w(k)\varphi_k \right\|^2 \leq B\,\|w\|^2.$$

Among other consequences, this means our basis functions must be linearly independent.

2.2.3 Generating functions

In Unser's model[14], each basis function φ_k is a shifted copy of some **generating function** φ. Setting $\varphi_k(x) = \varphi(\frac{x}{T} - k)$ we get

$$\tilde{f}(x) = \sum_{k\in\mathbb{Z}} w(k)\varphi\left(\frac{x}{T} - k\right).$$

We would like our interpolation method to have the capability of approximating any signal as closely as desired, by making the sampling interval sufficiently small. This is equivalent to the **partition of unity** condition:

$$\sum_{k\in\mathbb{Z}} \varphi(x+k) = 1, \quad \forall x \in \mathbb{R}.$$

With the generating function $\varphi = \text{sinc}$, we get Shannon interpolation. This satisfies both the Riesz basis and partition of unity conditions.

Another class of functions satisfying these conditions are the centered cardinal **B-splines**. The B-spline of degree 0 is the **rectangle function**,

$$\beta^0(x) = \text{rect}(x) = \begin{cases} 1 & : |x| < \frac{1}{2} \\ \frac{1}{2} & : |x| = \frac{1}{2} \\ 0 & : |x| > \frac{1}{2} \end{cases}.$$

B-splines of higher degree are defined recursively by convolution with β^0:

$$\beta^n = \beta^0 * \beta^{n-1}.$$

For $n = 1$ this gives the **triangle function**:

$$\beta^1(x) = \text{tri}(x) = \max(0, 1 - |x|).$$

The B-spline of degree n is supported in the interval $[-\frac{n+1}{2}, \frac{n+1}{2}]$. As n increases, the splines flatten out and become closer and closer approximations of a Normal distribution with $\sigma^2 = \frac{n}{12}$. In particular, this gives

$$\beta^{12}(x) \approx \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}.$$

The B-splines are frequently used in practice for image processing, and number of common interpolation methods can be expressed by using B-splines as generating functions. With the generating function $\varphi = \beta^0 = \text{rect}$, we get nearest-neighbour interpolation. With $\varphi = \beta^1 = \text{tri}$, we get piecewise linear interpolation.

2.2.4 Determining coefficients

In equation (2.2) we specify that a signal reconstruction will be a linear combination of basis functions, but we do not give a way to determine the coefficient sequence w.

Typically each interpolation method will provide an expression for each $w(k)$, but we would like to solve this in the general case. Recall that we expect an interpolation method to match the sample points as closely as possible, as expressed in (2.1).

Ideally, the reconstructed signal will go exactly through the sample points, that is, $\mathbf{y} = \tilde{f}(\mathbf{x})$. For each $i \in \mathbb{Z}$ we have

$$\begin{aligned} y_i &= \sum_{k\in\mathbb{Z}} w(k)\varphi_k(x_i) \\ &= \cdots + w(-1)\varphi_{-1}(x_i) + w(0)\varphi_0(x_i) + w(1)\varphi_1(x_i) + \dots \quad . \end{aligned}$$

Considering every sample simultaneously we get:

$$\begin{aligned} &\dots \\ y_{-1} &= \cdots + w(-1)\varphi_{-1}(x_{-1}) + w(0)\varphi_0(x_{-1}) + w(1)\varphi_1(x_{-1}) + \dots \\ y_0 &= \cdots + w(-1)\varphi_{-1}(x_0) + w(0)\varphi_0(x_0) + w(1)\varphi_1(x_0) + \dots \\ y_1 &= \cdots + w(-1)\varphi_{-1}(x_1) + w(0)\varphi_0(x_1) + w(1)\varphi_1(x_1) + \dots \\ \dots &\quad . \end{aligned}$$

This is an infinite system of linear equations, where the only unknowns are the coefficients w_k.

This can be greatly simplified if our generating function is sinc, rect, or tri. In those cases it so happens that $\varphi(0) = 1$, and $\varphi(k) = 0$ for any non-zero $k \in \mathbb{Z}$. As a result we get $w(k) = y_k$, which is consistent with Shannon's interpolation formula, as we would expect.

2.3 Finite signals

In practice, the signals we are interested in are often finite in nature. Even if a signal is infinite, we may only be interested in a finite region of it.

To model either case we consider signals $f : \mathbb{R} \to \mathbb{R}$ that are zero everywhere except for within some compact region. Without loss of generality, we scale and shift the signal such that the compactly supported region is within $[0, 1]$, and all values outside of that region are zero. Similarly, for any signal reconstruction $\tilde{f}$, we will only be interested in its values within $[0, 1]$.

With this restriction, only finitely many sample values will be non-zero. The samples

outside of $[0, 1]$ convey no information, so we can disregard them. Instead assume that we have N sample coordinates with a sampling interval of $T = \frac{1}{N-1}$ such that $x_1 = 0$ and $x_N = 1$. The set of sample coordinates can be expressed as a vector $\mathbf{x} = (0, \frac{1}{N-1}, \ \ldots, \frac{N-2}{N-1}, 1)^T$. Similarly, the set of sample values is now a vector $\mathbf{y} = (f(x_1), f(x_2), \ \ldots, f(x_{N-1}), f(x_N))^T$.

If we use a generating function with compact support, then only finitely many of the shifted basis functions will have support that overlap with $[0, 1]$. The rest will contribute nothing to a signal reconstruction, so can be disregarded.

Even if our generating function does *not* have finite support (such as sinc), it may be reasonable to disregard shifted copies of it that are so far out of range as to have a negligible contribution.

In either case, assume that we are left with a **basis size** of K shifted basis functions. Equation (2.2) now becomes a finite sum:

$$\tilde{f}(x) = \sum_{k=1}^{K} w(k)\varphi_k(x)$$

The sequence of coefficients w can be expressed as a vector $\mathbf{w} \in \mathbb{R}^K$.

It will be convenient to apply all basis functions to all sample coordinates at once. To do this, we define $\Phi : \mathbb{R}^N \to M_{K \times N}$ such that

$$\Phi(\mathbf{x})_{rc} = \varphi_r(x_c).$$

2.3.1 Exact solutions

Suppose we want our reconstruction to match the sample points exactly. Then we have

$$\mathbf{y} = \tilde{f}(\mathbf{x})$$
$$\mathbf{y}^T = \mathbf{w}^T\Phi(\mathbf{x})$$

This is a linear equation and we can solve it uniquely if $\Phi(\mathbf{x})$ is invertible. This will be the case if the matrix is square and has full rank. It will be square if $N = K$, that is, the number of samples is equal to the basis size. It is guaranteed to be full rank since all the sample coordinates are distinct, and the basis functions form a Riesz basis so are linearly independent.

2.3.2 Approximate solutions

If an exact solution is not possible, we can still endeavour to match the sample points as closely as possible, as expressed in equation (2.1).

$$\underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y}^T - \mathbf{w}^T \Phi(\mathbf{x})\|$$

This is a linear least squares problem, and we will return to its solution in Chapter 3.

2.4 Multidimensional finite signals

We now consider multidimensional signals, and show how the concepts from univariate interpolation can be adapted in a fairly straightforward way.

A signal is now modelled as a function

$$f : \mathbb{R}^D \to \mathbb{R}$$

where $D \in \mathbb{Z}_+$ is the **dimensionality**, or **mode**, of the signal. The signal takes a vector $\mathbf{x} \in \mathbb{R}^D$ as the input **coordinate**, and $y = f(\mathbf{x})$ is the output value of f at coordinate $\mathbf{x}$.

We assume our signal is finite, with support limited to $[0, 1]^D$.

We have N^D sample coordinates forming a Cartesian lattice that extends to the corners of the region $[0, 1]^D$. The distance between neighbouring points is $T = \frac{1}{N-1}$, and each coordinate is of the form $\mathbf{x} = (Tn_1, Tn_2, \ \ldots, Tn_D) \in \mathbb{R}^D$ where $n_d \in \{0, 1, \ \ldots, N-1\} \subseteq \mathbb{Z}$ for each d.

The sample coordinates are arranged into a matrix $\mathbf{X} \in M_{D \times N^D}$, where each column is a sample coordinate vector. The order of the columns does not matter, so long as the sample values are given in the same order. The sample values are arranged as a vector $\mathbf{y} \in \mathbb{R}^{N^D}$ as before.

The univariate interpolation methods we have describe so far can be extended to the multivariate case by use of **tensor product splines**[6]. In the univariate case, our reconstructions were of the form

$$\tilde{f}(x) = \sum_{k \in \mathbb{Z}} w(k) \varphi_k(x).$$

In the multivariate case we can use the same set of basis functions by applying each of them to each component of a coordinate and then taking the product:

$$\tilde{f}(\mathbf{x}) = \sum_{\mathbf{k} \in \mathbb{Z}^D} w(\mathbf{k}) \prod_{d \in D} \varphi_{k_d}(x_d).$$

In effect we now have K^D basis functions, each of the form of a product

$$\varphi_{\mathbf{k}}(\mathbf{x}) = \varphi_{k_1}(x_1)\varphi_{k_2}(x_2) \dots \varphi_{k_D}(x_D).$$

As before there is a coefficient for each basis function, so together our coefficients form a vector $\mathbf{w} \in \mathbb{R}^{K^D}$.

With these changes, we can still use the solution methods given in section 2.3 for multidimensional signals, with only slight changes in terminology.

Chapter 3

Coordinate networks

An essential tool in the field of machine learning is the (artificial) **neural network**.

In this chapter we give a brief introduction to neural networks, with a focus on the simplest architectures and how they can be used for interpolation. We then explain how an extra preprocessing step allows **coordinate-MLPs** to achieve better performance. Finally, we examine how a similar architecture can be used to implement classical interpolation.

3.1 Interpolation with neural networks

Neural networks can be applied to many different tasks. A common example is image classification, in which single network is trained on thousands of images. It would be wise to put this example out of mind while reading this document. We are concerned instead with the task of interpolation of images (and other signals), in which a single network is trained on thousands of *pixels* taken from a *single* image.

The main reference for this section is the book *Deep Learning* by Goodfellow et al.[3].

3.1.1 Architecture

Neural networks are often presented as a graph like the one in Figure 3.1. A network consists of many **nodes** that are connected by directed edges that carry data.

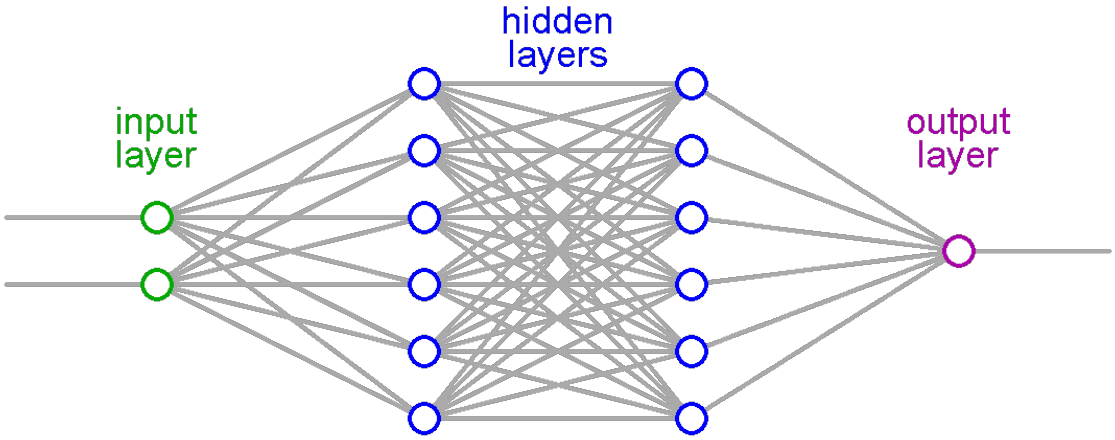


Figure 3.1: A diagram of a small neural network. The edges are unlabelled, but implied to be directed from left to right.

Nodes

There are three main types of nodes, with slightly differing behaviour. Data enters the network through **input nodes** and exits via **output nodes**. The rest of the nodes are called **neurons**. We will describe neurons first.

A neuron has multiple input edges, each of which carries some value $x_i \in \mathbb{R}$. Each neuron has a **weight** $w_i \in R$ associated with each of its input channels, and together they are used to calculate a weighted sum of the input values. The result is then passed through a non-linear **activation function** $a : \mathbb{R} \to \mathbb{R}$ and the final value is sent to each of the neuron's output edges.

A neuron with k input edges can be modelled as a function $n : \mathbb{R}^k \to \mathbb{R}$ defined by

$$n(\mathbf{x}) = a(\mathbf{w}^T\mathbf{x})$$

where $\mathbf{x} \in \mathbb{R}^k$ is a vector of the input values and $\mathbf{w} \in \mathbb{R}^k$ is a vector of the associated weights.

There are many possible choices for the non-linear activation function. One of the most popular, due to its simplicity and reasonable performance, is the **rectified linear unit (ReLU)**. The behaviour of ReLU is given by

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & : x \geq 0 \\ 0 & : x < 0 \end{cases}.$$

The behaviour of input and output nodes is simpler than that of neurons. An input

node simply makes available whatever values are fed into the network. An output node directly outputs a weighted sum, like a neuron but with the activation function omitted.

$$n_{out}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}.$$

In some architectures, the output nodes *do* have an activation function, which may be different from the neurons' activation function. We do not need to consider this case for our purposes.

Optionally, a neuron may have a **bias** value b, in which case the neuron function is $n(\mathbf{x}) = a(\mathbf{w}^T\mathbf{x} + b)$. This can equivalently be expressed as an additional input value that is constant, $x_{k+1} = 1$, and then the bias value can be treated as another component of the weight vector, $w_{k+1} = b$. In any case, we do not require bias weights for our analysis and experiments.

Layers

There are many ways nodes can be arranged in a neural network. We focus on the simplest architectures.

In a **feedforward neural network (FNN)**, the nodes are arranged in **layers**. The first layer is the **input layer**, the last is the **output layer**, and any layers between them are called **hidden layers**. Each node is connected to some number of nodes on the next layer, and data feeds strictly forwards from one layer to the next.

The count of nodes in a layer is called the layer's **width**. Each layer width can be chosen independently, but often all hidden layers are given the same width.

In a **fully connected** FNN, each node is connected to every node on the next layer. A **multilayer perceptron (MLP)** is a fully connected FNN with at least one hidden layer.

The **depth** of an FNN is determined by how many hidden layers it has. There is no universally agreed-upon definition of what qualifies as "deep learning", but most researchers agree that in order to be described as a deep network, an FNN must have at least two hidden layers[16]. Networks that do not meet this requirement are sometimes called **shallow** neural networks.

When describing the depth of a network, sometimes the input layer is not included in the total count of layers. Given this potential source of ambiguity, we will always

describe the depth of an FNN by specifying how many hidden layers it has.

Overall behaviour

The overall behaviour of an neural network, viewed as a function, is called its **implicit neural function (INF)**. In the case of fully connected FNNs, the INF can be expressed concisely in terms of vectors and matrices.

The output of each layer can be formulated as a vector with one component for each node, and the nodes in the next layer take linear combinations of this vector's components. We can express this as multiplication by a matrix containing the nodes' weights. Finally we apply our non-linear activation function, which we define to act component-wise on vectors.

As an example, consider the network depicted in figure 3.1. It has an input layer of width 2, two hidden layers of width 6, and an output layer of width 1. Our matrices of weights will be $W_1 \in M_{6\times 2}$, $W_2 \in M_{6\times 6}$, and $W_3 \in M_{1\times 6}$. The INF describing the overall behaviour is then $\tilde{f} : \mathbb{R}^2 \to \mathbb{R}$, given by

$$\tilde{f}(\mathbf{x}) = W_3 a(W_2 a(W_1 \mathbf{x}))$$

where $\mathbf{x} \in \mathbb{R}^2$ is the vector of input data.

It can be useful to treat the set of weights as additional parameters to be passed in. To this end we define another function, NN, such that

$$\tilde{f}(\mathbf{x}) = \mathrm{NN}(\mathbf{W}, \mathbf{x})$$

where $\mathbf{W}$ is some arrangement of all the weights into a single vector.

In the above example, the total number of weights is $6 \times 2 + 6 \times 6 + 1 \times 6 = 54$, so the function signature of NN would be $\mathrm{NN} : \mathbb{R}^{54} \times \mathbb{R}^2 \to \mathbb{R}$.

The above formulations in terms of linear algebra are not only a useful way to view neural networks, but also roughly how they are implemented in software.

3.1.2 Training

So far we have described networks as if they are static objects with fixed behaviour. This is partly true — for any given network, the NN function as described above *is* static, but we are free to change the weight vector $\mathbf{W}$ that we pass into it. Since the implicit neural function is defined as $\tilde{f}(\mathbf{x}) = \mathrm{NN}(\mathbf{W}, \mathbf{x})$, any change of $\mathbf{W}$ will

alter the INF. Through the process of **training** a neural network, the weight vector $\mathbf{W}$ is iteratively modified to yield an INF with whatever behaviour we desire.

Training data

A neural network is trained with the assistance of **training data**. For the task of interpolation, the training data consists of the sample coordinates $\mathbf{X}$ and sample values $\mathbf{y}$ as defined in Chapter 2. Each sample coordinate $\mathbf{x}_i$ will be input into the network. The result, $\mathrm{NN}(\mathbf{W}, \mathbf{x}_i)$, will be compared with the corresponding sample value y_i.

For tasks other than interpolation, elements of $\mathbf{X}$ are called **inputs** or **features** while elements of $\mathbf{y}$ are known as **outputs** or **labels**.

Our aim is to end training with an INF that closely matches the sample points used in training. That is, we want $\tilde{f}(\mathbf{x}_i) \approx y_i$ for each i. For notational convenience, we define $\tilde{f}$ (and other functions) to operate on multiple samples at once, although each sample is still processed independently. With this convention, our aim can be written as $\tilde{f}(\mathbf{X}) \approx \mathbf{y}$.

Loss functions

If we hope to achieve $\tilde{f}(\mathbf{X}) \approx \mathbf{y}$, we need a way of measuring how far off we are from this goal. We do this using a **loss function**.

We will use **mean square error (MSE)** as our loss function. It is given by

$$\mathrm{MSE}(\tilde{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} (\tilde{y}_i - y_i)^2$$

where N is the number of samples.

Note that MSE is related to the ℓ_2-norm by $\|\tilde{\mathbf{y}}, \mathbf{y}\|^2 = N \,\mathrm{MSE}(\tilde{\mathbf{y}}, \mathbf{y})$. Minimising one will result in also minimising the other, so we may use either notation as convenient.

Optimisers

An **optimiser** is an algorithm that updates the weights of a neural network, by making use of the training data and loss function.

The simplest optimiser is the **gradient descent** algorithm. Gradient descent involves calculating the directional derivative of the loss function evaluated at the

current value of the weight vector, and then adjusts the weights in the direction of steepest descent. In our case, this can be expressed as

$$\mathbf{W}' = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} \, \mathrm{MSE}(\mathrm{NN}(\mathbf{W}, \mathbf{X}), \mathbf{y})$$

where the **learning rate** ϵ determines how much the weights are adjusted in a single step. The network weights are replaced with the new weight vector $\mathbf{W}'$, and the process is repeated for many steps.

A single "step" as described above is called an **epoch** of training. For typical ML tasks, it may be impractical to evaluate the entire set of training data for every step. For this reason, among other others, training data is sometimes split into **mini-batches** and each optimiser step uses a single mini-batch. In this case, an epoch consists of however many steps are required to process the entire set of training data.

We will not be using mini-batches. This case is referred to as **full-batch** training.

There are many optimisers to choose from, most of which a based on gradient descent but improve upon it in some way. Our experiments in Chapter 4 use a popular optimisation algorithm called **Adam**[4].

In principle, the optimiser can be any algorithm for setting the weights. At the end of this chapter we consider some non-standard optimisers.

Overall behaviour

A theoretically ideal optimiser would find weights that achieve the global minimum value of $\mathrm{MSE}(\mathrm{NN}(\mathbf{W}, \mathbf{X}), \mathbf{y})$. In practice this is rarely achieved, and it is more likely to converge to a local minimum.

Fortunately, and perhaps surprisingly, it has been found that neural networks frequently converge to a local minimum that is reasonably close to the global minimum in terms of loss value. Thus we can approximately describe a trained neural network by

$$\begin{aligned} \tilde{\mathbf{W}} &\approx \underset{\mathbf{W}}{\mathrm{argmin}} \|\mathbf{y} - \mathrm{NN}(\mathbf{W}, \mathbf{X})\| \\ \tilde{f}(\mathbf{x}) &= \mathrm{NN}(\tilde{\mathbf{W}}, \mathbf{x}). \end{aligned}$$

If this equation is compared with equation (2.1), the link with classical interpolation begins to become apparent.

3.1.3 Hyperparameters and trainable parameters

We have described how the weight vector changes during training. This is, in fact, the only aspect of the network that changes during training. As a result, the weights are sometimes called **trainable parameters**.

By contrast, there are many parameters that allow us to specify the architecture of a neural network and the training process, but that do not change during training. These are called **hyperparameters**.

Some hyperparameters have a de facto default value that makes for a reasonable choice in many cases. A neural network using these default values is sometimes called a **vanilla** neural network.

Table 3.1 summarises the main hyperparameters and typical values.

Hyperparameter	Typical values
Direction of data flow	**feedforward**, recurrent
Connectivity	**fully connected**
Activation function	**ReLU**, various sigmoids
Hidden layer count	1–8
Hidden layer width	10^2–10^5
Input layer width	task-dependent
Output layer width	task-dependent
Loss function	**MSE**
Optimiser	**(stochastic) gradient descent**, Adam
Learning rate	**0.001**
Other optimiser parameters	optimiser-dependent
Training duration	10^3–10^6 epochs

Table 3.1: Summary of hyperparameters determining neural network architecture and training. "Vanilla" values are in bold.

3.1.4 Performance and spectral bias

A vanilla neural network can achieve good performance on a variety of tasks. Interpolation is not one of those tasks.

During training, a neural network will tend to learn some aspects of the training data faster than others. In particular, it has been empirically observed that lower frequency components are learned before higher frequency components, which are learned much more slowly. This phenomenon is called **spectral bias**[10].

The spectral bias of neural networks is a major issue if we wish to use them for interpolation. Most signals of interest have a significant amount of useful information

carried by their higher frequencies. In order for a vanilla neural network to learn those frequencies, it would need to be trained for many more epochs than is practical.

3.2 Coordinate-MLPs

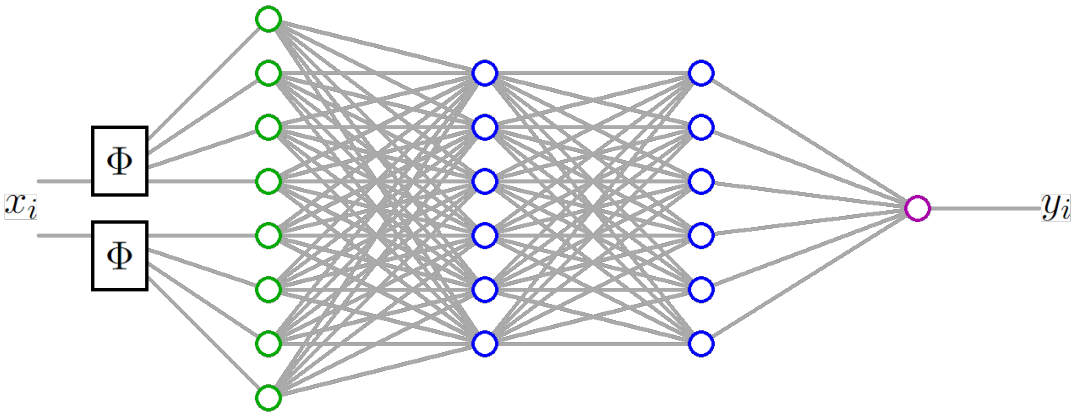


Figure 3.2: A diagram showing a simple coordinate-MLP.

To overcome spectral bias, we can apply some transformation to our input data such that high-frequency components are transformed into lower frequencies. The result can then be fed into a neural network as before.

Such a transformation is called a **positional encoding**. So long as the encoding is injective, no information is lost, so in principle the network can still extract whatever information it needs in order to successfully interpolate the signal.

A positional encoding followed by a multilayer perceptron is called a coordinate-based MLP, or simply a **coordinate-MLP**.

The main reference for this section is the paper *Trading positional complexity vs deepness in coordinate networks* by Zheng et al.[17].

3.2.1 Positional encoding

We choose some **positional encoding function** $\Phi_D : \mathbb{R}^D \to \mathbb{R}^m$, where $m \gg D$. By applying this function to a sample coordinate $\mathbf{x}$, we embed the coordinate in a higher dimensional space. The result is an **embedded coordinate** $\Phi_D(\mathbf{x})$. When applied to the entire set of sample coordinates, we define $\Phi_D(\mathbf{X})$ to act sample-wise on each sample coordinate.

There are many possible functions we could choose for Φ_D. We will restrict attention to those functions that can be described in terms of a simpler function Φ_1 that acts

on a single-dimensional coordinate. This restriction is convenient, but does rule out some potentially promising functions, such radial basis functions.

3.2.2 Encoding one dimension

Let us first consider a 1D positional encoding function, $\Phi_1 : \mathbb{R} \to \mathbb{R}^K$. Its output is a K-dimensional vector, and we can consider each component separately.

Let $\varphi_k : \mathbb{R} \to \mathbb{R}$ be the function for component k. Then we have

$$\Phi_1(x) = (\varphi_0(x), \varphi_1(x), \ldots, \varphi_{K-1}(x))^T.$$

We call $\{\varphi_0, \varphi_1, \ldots, \varphi_{K-1}\}$ our **1D basis functions**.

Positional encodings were first introduced in the paper *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*[8]. Their positional encoding uses basis functions consisting of trigonometric functions of different frequencies,

$$\Phi_1(x) = \left(\sin(2^0\pi x), \cos(2^0\pi x), \ \ldots, \sin(2^i\pi x), \cos(2^i\pi x), \ \ldots\right).$$

Trigonometric encoding has been in multiple subsequent works[13][15], and remains the most common encoding.

An alternative approach, explored by Zheng et al.[17], is to use shifted basis functions. With this approach, we only need choose a single **generating function**, $\varphi : \mathbb{R} \to \mathbb{R}$. Each basis function φ_k is then a copy of φ shifted by an amount proportional to k.

Our experiments in Chapter 4 use each of the following as generating functions.

- The sinc function.
- The rectangle function, which is the B-spline of degree 0.
- The triangle function, which is the B-spline of degree 1.
- A Gaussian function given by $\text{Gauss}(x) = \exp\left(-\frac{1}{2}\frac{x^2}{\sigma^2}\right)$, which is approximately the B-spline of degree 12.

Zheng et al.[17] consider the rectangle, triangle, and Gaussian as generating functions, among others. They examine the effect of changing the standard deviation of the Gaussian, and generally choose basis function widths and the variable K independently.

In contrast, we choose to make our basis function widths dependent on K, so as to be consistent with our treatment of classical interpolation in chapter 2. Given a generating function, our kth basis function will be

$$\varphi_k(x) = \varphi\left((K-1)x - k\right)$$

The result is a set of basis functions that are centered at 0, 1, and equally spaced in between. The basis functions are scaled such that their width is proportional to the intervals between them.

3.2.3 Encoding multiple dimensions

To construct a multidimensional encoding function Φ_D, we can apply a 1D encoding function Φ_1 to each coordinate component separately, and then combine the resulting vectors in some way. Zheng et al.[17] describe two approaches, which they call **simple encoding** and **complex encoding**.

They are equivalent in the case of 1D signals.

3.2.4 Simple encoding

We simply concatenate the results of applying Φ_1 to each coordinate component. Our overall encoding function $\Phi_D : \mathbb{R}^D \to \mathbb{R}^{DK}$ is given by

$$\begin{aligned}
\Phi_D\left((x_1, x_2, \ \ldots, x_D)\right) &= (\Phi_1(x_1), \Phi_1(x_2), \ \ldots, \Phi_1(x_D)) \\
&= (\varphi_0(x_1), \ \ldots, \varphi_{K-1}(x_1), \\
&\quad \varphi_0(x_2), \ \ldots, \varphi_{K-1}(x_2), \\
&\quad \vdots \\
&\quad \varphi_0(x_D), \ \ldots, \varphi_{K-1}(x_D))
\end{aligned}$$

This is the approach taken by most coordinate-MLP research to date.

3.2.5 Complex encoding

This encoding has a higher level of implementation complexity (it is unrelated to complex numbers). We take K^D different products, one for each combination of one component from each of the vectors $\Phi_1(x_1), \Phi_1(x_2), \ \ldots, \Phi_1(x_D)$.

This is easiest to understand and implement in the 2D case.

$$\Phi_2((x_1, x_2)) = \mathrm{vec}(\Phi_1(x_1)\Phi_1(x_2)^T)$$

The embedded coordinate components $\Phi_1(x_1)$ and $\Phi_1(x_2)^T$ are multiplied to produce a $K \times K$ matrix, which is then vectorised to flatten it into a K^2-dimensional vector.

The more general case of D dimensions can be formulated in terms of the Kronecker product.

$$\begin{aligned}\Phi_D((x_1, x_2, \ \ldots, x_D)) &= \bigotimes_{d=1}^{D} \Phi_1(x_d) \\ &= \Phi_1(x_1) \otimes \Phi_1(x_2) \otimes \cdots \otimes \Phi_1(x_D)\end{aligned}$$

This approach is equivalent to the method of tensor product splines, described in Chapter 2.

3.2.6 Overall behaviour

Positional encoding simply adds a preprocessing step to the inputs, after which an MLP is used without further modification. As a result, the training process can be described by

$$\begin{aligned}\tilde{\mathbf{W}} &\approx \underset{\mathbf{W}}{\mathrm{argmin}} \|\mathbf{y} - \mathrm{NN}(\mathbf{W}, \Phi(\mathbf{X}))\|_2 \\ \tilde{f}(\mathbf{x}) &= \mathrm{NN}(\tilde{\mathbf{W}}, \Phi(\mathbf{x}))\end{aligned}$$

The only change is the addition of Φ.

3.2.7 Hyperparameters

The addition of positional encoding results in more hyperparameters that must be configured. Table 3.2 lists the hyperparameters for a positional encoding with shifted basis functions.

	Positional encoding hyperparameters	Possible values
φ	Generating function	rect, tri, Gauss, sinc
K	Shift count	2 or more
	Encoding type	simple, complex

Table 3.2: Hyperparameters determining a shifted-basis positional encoding.

3.2.8 Performance and applications

The addition of a positional encoding leads to a dramatic improvement of the ability of MLPs to learn low-dimensional signals.

In the words of Tancik et al.[13], coordinate-MLPs *"have been used to represent images, volume density, occupancy, and signed distance, and have achieved state-of-the-art results across a variety of tasks such as shape representation, texture synthesis, shape inference from images, and novel view synthesis."*

As the above list suggests, signal interpolation is far from the only application of coordinate-MLPs.

3.3 Some terminology

A multilayer perceptron has, by definition, at least one hidden layer. We would like to consider a fully connected FNN with *zero* hidden layers. In this network the input layer is directly connected to the output layer, which simply produces a linear combination of the inputs. (We assume that the output layer does not have an activation function.)

A single linear layer is a somewhat degenerate architecture for a neural network, and the resultant network has relatively limited capabilities. So it is unsurprising that it is rarely considered or discussed in modern machine learning. For want of a more established term, we call such an architecture a **linear network (LN)**.

Zheng et al.[17] make productive use of a positional encoding followed by an LN, and we build upon their work. This architecture is technically not a coordinate-MLP, so instead we call it a **coordinate-LN**. We also use the umbrella term **coordinate networks** to encompass both coordinate-MLPs and coordinate-LNs.

3.4 Implementing classical interpolation

A coordinate-LN with a complex encoding consists of a positional encoding followed by a single linear layer. The implicit neural function is given by

$$\tilde{f}(\mathbf{x}) = \mathrm{NN}(\mathbf{W}, \Phi_D(\mathbf{x})) = \mathbf{W}_1 \Phi_D(\mathbf{x})$$

where the components of $\mathbf{W}$ are rearranged into a matrix $\mathbf{W}_1$ of dimensions $1 \times K^D$. Let us assume that $\mathbf{W}_1 = \mathbf{W}^T$. Then we have

$$\begin{aligned} \tilde{\mathbf{W}} &\approx \underset{\mathbf{W}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{W}^T \Phi_D(\mathbf{X})\|_2 \\ \tilde{f}(\mathbf{x}) &= \tilde{\mathbf{W}}^T \Phi_D(\mathbf{x}) \end{aligned} \tag{3.1}$$

This almost exactly describes classical interpolation as given in Chapter 2. The only difference is whether the equality in (3.1) is approximate or exact. The equality above is approximate since $\tilde{\mathbf{W}}$ is found by applying gradient descent (or another optimiser) for a limited number epochs, so may not achieve the global minimum. Zheng et al.[17] show that gradient descent converges quickly for a coordinate-LN, but also provide an alternative closed-form solution that requires even less computation.

Equation (3.1) is a linear least squares problem, so can be solved with the ordinary least squares method. The solution is given by

$$\tilde{\mathbf{W}} = \Phi_D(\mathbf{X})^+ \mathbf{y}.$$

where $\Phi_D(\mathbf{X})^+$ is the Moore-Penrose **pseudoinverse** of $\Phi_D(\mathbf{X})$. When a matrix A has linearly independent columns, the pseudoinverse is given by $A^+ = (A^T A)^{-1} A^T$. In other cases A^+ is still defined, but it does not always have a simple algebraic formula.

This method for finding $\tilde{\mathbf{W}}$ is effective but does not scale well. Zheng et al.[17] observe that a naive implementation has memory and time complexity $O(K^D N^D)$, which makes it impractical in many cases. They propose an elegant workaround by employing some properties of the Kronecker product. We use this workaround in the next chapter, where we compare the performance of coordinate-LNs and coordinate-MLPs.

Chapter 4

Comparison of classical interpolation and coordinate-MLPs

4.1 Motivation

In Section 3.4 we showed how classical interpolation can be implemented in the form of a coordinate-LN with an appropriate positional encoding. In a sense, coordinate-MLPs could be seen as classical interpolation with extra layers. This similarity suggests that a fruitful comparison is possible.

In this chapter we perform experiments and analysis with the aim of shedding light on the following questions:

- Can coordinate-MLPs outperform classical interpolation?
- How does this depend on the dimensionality of the signal?
- When do deep neural networks perform better than shallow ones?

We provide a framework for making fair comparisons by controlling for model complexity, describe our experimental results, and conclude with some recommendations for deep learning practitioners.

4.2 Model complexity

In general, the complexity of a model affects the size of its approximation space (defined in Section 2.2.1), which in turn affects its performance (whether measured by training error or test error). For a fair comparison, we want models to have roughly equal complexity.

For the interpolation methods we are considering, the model complexity is a function of several hyperparameters. A simple way to take all of these into account is to consider the total number of trainable parameters the model has. This is not an ideal measure of model complexity, but its relative simplicity gives us a straightforward way to compare models. We denote the parameter count by P.

We will be implementing classical interpolation in the form of coordinate-LNs with a complex positional encoding. In Section 2.4 we saw that the number of coefficients needed for multivariate classical interpolation is

$$P_{LN} = K^D,$$

where K is basis size in one dimension, and D is the dimensionality of the signal.

In a coordinate-MLP with a simple positional encoding and H hidden layers of width G, the widths of the layers are $[KD, G, \ \ldots, G, 1]$. There is a matrix of weights for each pair of consecutive layers. Summing the sizes of the matrices gives the number of parameters as

$$P_{MLP} = KDG + (H-1)G^2 + G.$$

For each of our experiments we will try many different parameter counts, and show how performance varies with P.

4.3 Architecture schema

For each coordinate-LN we test, we would like to compare it against a coordinate-MLP with a model complexity that is the same or simpler. In terms of parameter counts we want

$$\begin{aligned} P_{LN} &\geq P_{MLP} \\ K_{LN}^D &\geq K_{MLP}DG + (H-1)G^2 + G. \end{aligned}$$

We have distinguished between the values of K used by the coordinate-LN and the coordinate-MLP since they may not be the same.

Taking K_{LN}, D, and H as fixed, this constraint can still be satisfied in multiple ways by varying K_{MLP} and G. It would be reasonable to try multiple of these models, but we would like to restrict our focus further.

We choose to let $K = K_{LN} = K_{MLP}$, so that the models each use the same 1D basis size. We now have

$$\begin{aligned} K^D &\geq KDG + (H-1)G^2 + G \\ 0 &\geq (H-1)G^2 + (KD+1)G - K^D. \end{aligned}$$

This is a quadratic in terms of G (the only free variable). In addition, G needs to be an integer. Solving this, the maximum value G is allowed to take is

$$G = \left\lfloor \frac{1}{2(H-1)} \left(-(KD+1) + \sqrt{(KD+1)^2 + 4(H-1)K^D} \right) \right\rfloor.$$

We now have a systematic way to choose a coordinate-MLP architecture, given particular choices of D, K, and H.

4.4 Experiments

4.4.1 Dataset

Previous papers on coordinate-MLPs[13][17] use the DIV2K dataset[1] as a source of high-resolution images for testing. We use this same dataset so that our results may be somewhat comparable.

For our experiments on 1D signals, we used the Div2K validation dataset, selecting only those images that were 2040 pixels wide. For each of the resulting 89 images, we converted them to greyscale and took the central row of 2040 pixels. Some of them are shown in figure 4.1.

For our experiments on 2D signals, we took a subset of five images from the Div2K dataset, converted them to greyscale, and then cropped a central 512x512px region of each. The resulting five images can be seen in figure 4.2.

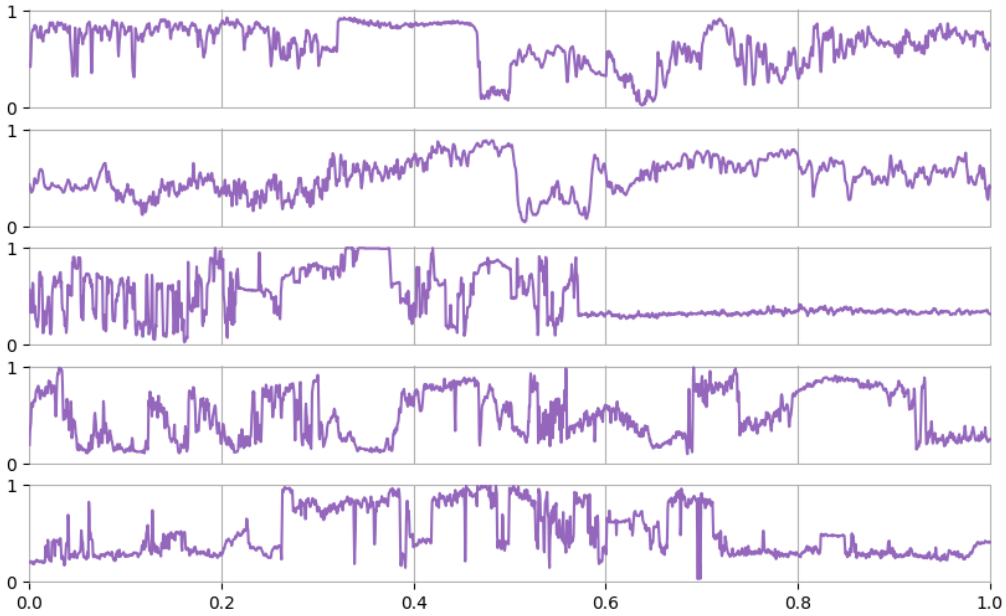


Figure 4.1: Some examples of 1D signals, each constructed from a row of pixels.

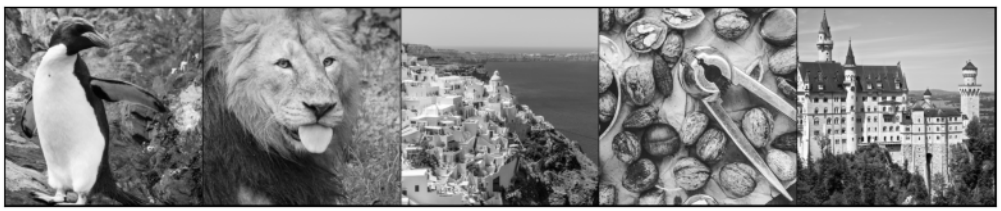


Figure 4.2: Our 2D experiments used these five images, each 512 by 512 pixels.

4.4.2 Experimental setup

In addition to controlling for model complexity, we kept as many other factors constant as possible, in order for our results to be meaningful.

Every experiment performed had the following setup in common:

1. Image data was preprocessed to create samples from a single-channel signal of appropriate dimensions. This was done by first converting colour images to greyscale, and then normalising sample coordinates and values to lie in the range $[0, 1]$.

2. A positional encoding function was applied to the sample coordinates, resulting in an embedding in a higher dimensional space.

3. The embedded coordinates were used as inputs to a network consisting of one or more linear layers (with no bias weights), separated by ReLU layers.

4. MSE was used as the loss function, and a training method was applied to update the network, using the embedded coordinates and sample values as training data.

5. Overall performance, expressed as **peak signal-to-noise ratio (PSNR)**, was based on the lowest training error reached during or after training. Since the signal values were normalised to the range $[0, 1]$, the PSNR could be calculated by $\text{PSNR} = 10 \log_{10}(\text{MSE}^{-1})$.

Classical interpolation was implemented with a coordinate-LN using these hyperparameters:

- We used a complex positional encoding, as described in Section 3.2.5.
- The hidden layer count, H, was zero.
- As the training method, we used the least-squares solution method described in Section 3.4.

For coordinate-MLPs we used these hyperparameters:

- We used a simple positional encoding, as described in Section 3.2.4.
- The hidden layer count, H, was one or more.
- We trained the network using Adam[4] as the optimiser, using full-batch training. The epoch count varied per experiment.

Our coordinate-LN experiments involve "training" by finding a closed-form least squares solution. This gives us results that are deterministic and are guaranteed to represent the best possible performance by those models.

In contrast, our coordinate-MLPs are initialised with random weights and trained with an optimiser that has no guarantee of finding the global minimum. It is possible that with different initial weights, or a better optimiser, the same coordinate-MLP architecture could perform much better.

With this in mind, our experiments focus on the *best-case* performance of each model. For coordinate-MLPs this means that we are free to re-run experiments with different random seeds, or train for a very large number of epochs. Out of all the runs performed with a particular architecture, we take the best result and use it as a proxy for the best-case performance achievable with that architecture.

4.4.3 Implementation details

Experiments were implemented using the Python ML library PyTorch[9]. This library makes use of the CUDA interface to allow the GPU to be utilised for many ML tasks.

All experiments were run on a consumer-grade desktop computer with an Intel Core i7 CPU, 16 GiB of RAM, and a NVIDIA GeForce GTX 1650 as the graphics card. Although the GPU provided a speed boost, its 4 GB of memory put some practical limitations on the types of experiments we could run.

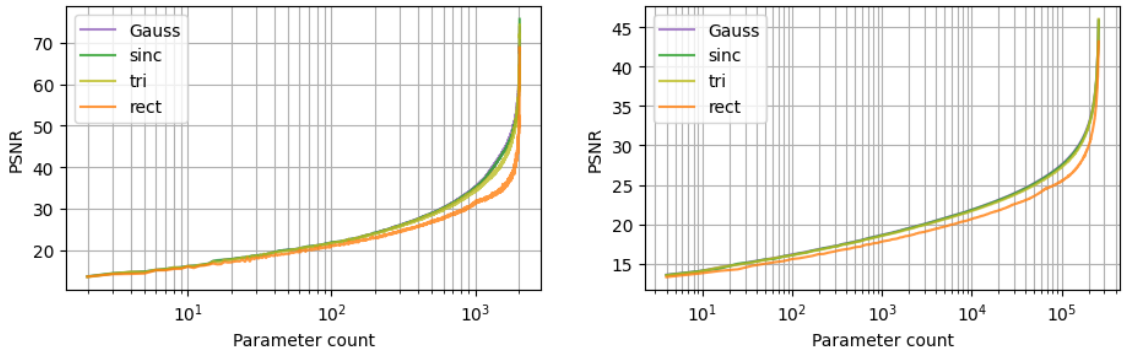
All of the code for this project can be found in our git repository[7].

4.4.4 Comparison of generating functions

To our knowledge, no previous work on coordinate-MLPs has considered the sinc function for a positional encoding. The closest we found was a recent preprint[11] that uses sinc as the activation function.

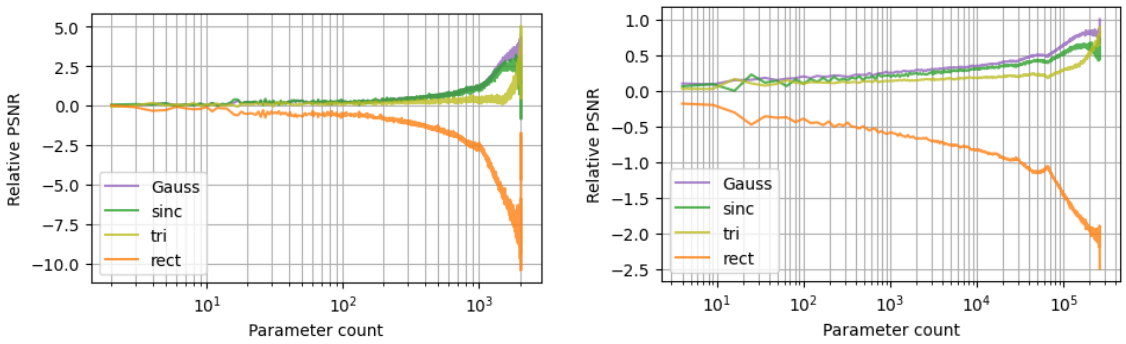
To validate our use of sinc as a generating function, we first compared it against the other generating functions describe in Section 3.2.2.

Figure 4.3 shows that for 1D and 2D signals and for each generating function, performance improves roughly monotonically as the number of parameters is increased. As P approaches the number of samples, the coordinate-LN approaches perfect memorisation regardless of the generating function used.



(a) Average PSNR over 89 1D signals. (b) Average PSNR over five 2D signals.

Figure 4.3: Generating function performance for 1D and 2D signals.



(a) Average PSNR over 89 1D signals. (b) Average PSNR over five 2D signals.

Figure 4.4: Generating function performance relative to their mean.

Figure 4.4 shows the same data, relative to the average of the four generating functions considered. In this figure it is easier to distinguish which performs best.

Recall from Section 2.2.3 that rect, tri, and Gauss correspond respectively to the B-splines of degree 0, 1, and 12. Their performance as generating functions reflects this ordering — B-splines of higher degree make for better approximations, but at the expense of being less well-localised. These results are also consistent those found in[17].

The performance of sinc is roughly on par with Gauss, but Gauss usually performs slightly better, especially for high values of P. We chose to use sinc in later experiments regardless, so as to allow comparison with Shannon interpolation.

4.4.5 Comparison of coordinate network depths

Figure 4.5: The image used for coordinate-MLP experiments.

Coordinate-MLPs can take a considerable time to converge, even with the aid of positional encoding. As such, it was infeasible to perform experiments on many different signals. Instead, we used the same image for all experiments, but gathered detailed results for that image. The image is shown in figure 4.5. The image depicts a coastal area with a mix of natural and artificial features. As such, we hope that it is fairly representative of a wider variety of images. Further work would be required to verify whether our results generalise to other images.

The previous section showed that sinc performs well as a generating function, at least for classical interpolation. Our coordinate network experiments all use sinc encodings — a complex encoding for coordinate-LNs, and a simple encoding for coordinate-MLPs.

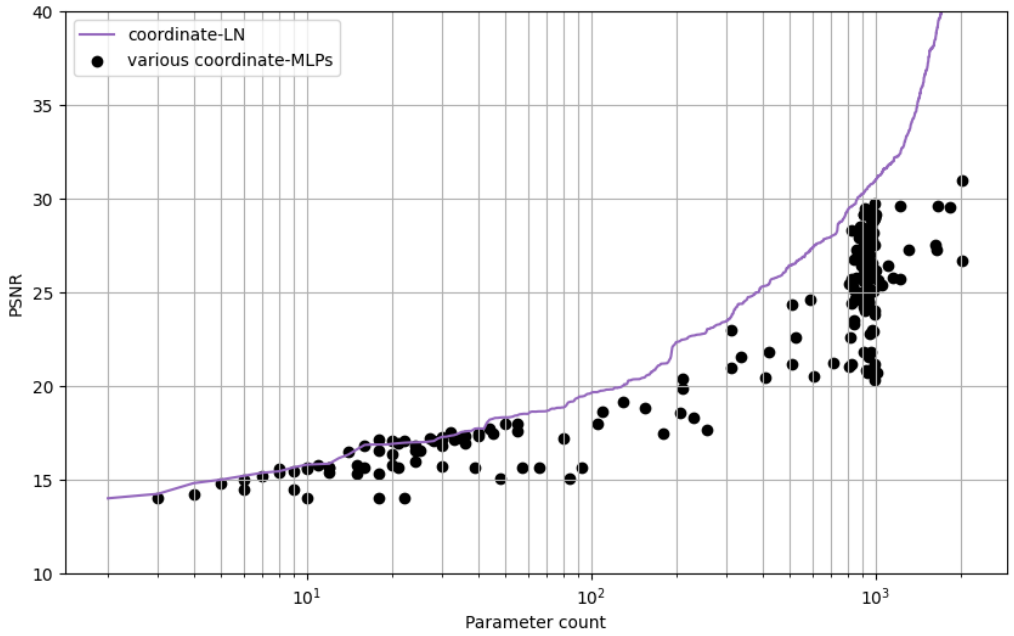


Figure 4.6: Performance of various coordinate-MLPs on a 1D signal. None significantly outperform a coordinate-LN implementing Shannon interpolation.

1D signal

For the 1D case, we tried a large number of different coordinate-MLP architectures, each trained for at least 10000 epochs. Figure 4.6 shows the results. No coordinate-MLP managed to outperform the coordinate-LNs by any significant margin.

2D signal

For the 2D case, we used the architecture schema described earlier. We tried every number of hidden layers from 1 to 8, and tried twelve different values of K.

For each combination, we ran three trials, each with a different seed for initialisation of the coordinate-MLP. Any trial that made no significant progress after 1000 epochs was cancelled and did not count towards the three trials. The remaining successful trials were trained for many epochs, until they appeared to have converged. Out of each set of three trials, we took the maximum PSNR achieved.

Figure 4.7 shows the results. For each value of K tested, at least one coordinate-MLP outperforms 2D Shannon interpolation. Figure 4.8 shows the same results, shifted to be relative to 2D Shannon interpolation.

Network deepness was only useful up to a point — networks with hidden layer counts of 5 and above were never the best-performing model. Figure 4.9 shows a subset

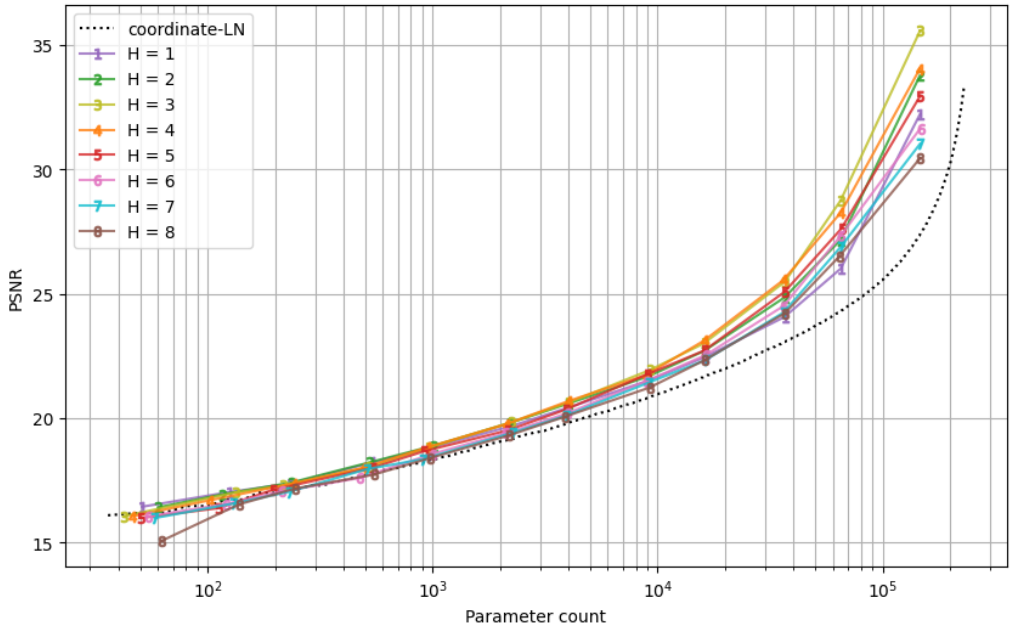


Figure 4.7: Performance of coordinate-MLPs with various hidden layer counts.

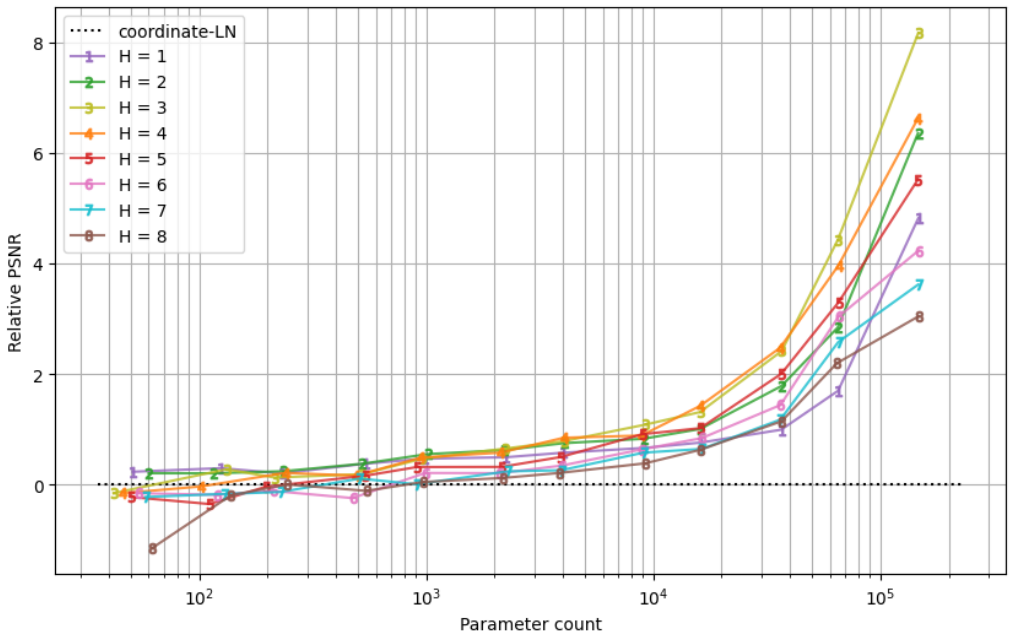


Figure 4.8: Relative performance of coordinate-MLPs with various hidden layer counts. PSNR is relative to the performance of the coordinate-LN.

of the data, plotted by hidden layer count. The best-performing hidden layer count seems to increase as K is increased, but this trend is somewhat inconclusive.

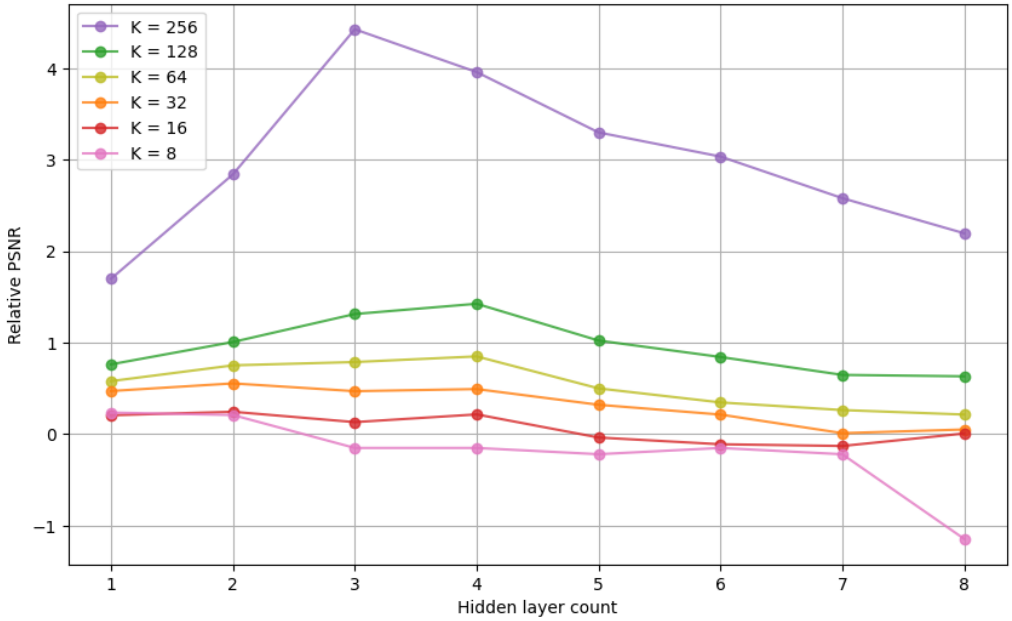


Figure 4.9: Relative performance of coordinate-MLPs for various values of K, plotted by hidden layer count.

4.5 Parameter counts in higher dimensions

Let us now examine how parameter counts depend on dimensionality.

Recall that for classical interpolation implemented with a coordinate-LN we have $P_{LN} = K^D$. The 1D basis count K is always at least 2, and usually much higher. The parameter count is exponential in the number of dimensions, so the classical approach can be prohibitively expensive even for a moderate number of dimensions.

In contrast, for a coordinate-MLP with a simple positional encoding we have $P_{MLP} = KDG + (H-1)G^2 + G$. This has several free variables, but we can reduce that by considering what sort of architectures are typical.

The width of the input layer is KD, and the hidden layer widths are G. A reasonable default choice would be to make the hidden layer width equal to the input layer width, so $G = KD$. In this case we get

$$\begin{aligned} P_{MLP} &= KD(KD) + (H-1)(KD)^2 + (KD) \\ &= HK^2D^2 + KD. \end{aligned}$$

This is quadratic in D. As the number of dimensions increases, P_{MLP} will increase, but not nearly as quickly as P_{LN}.

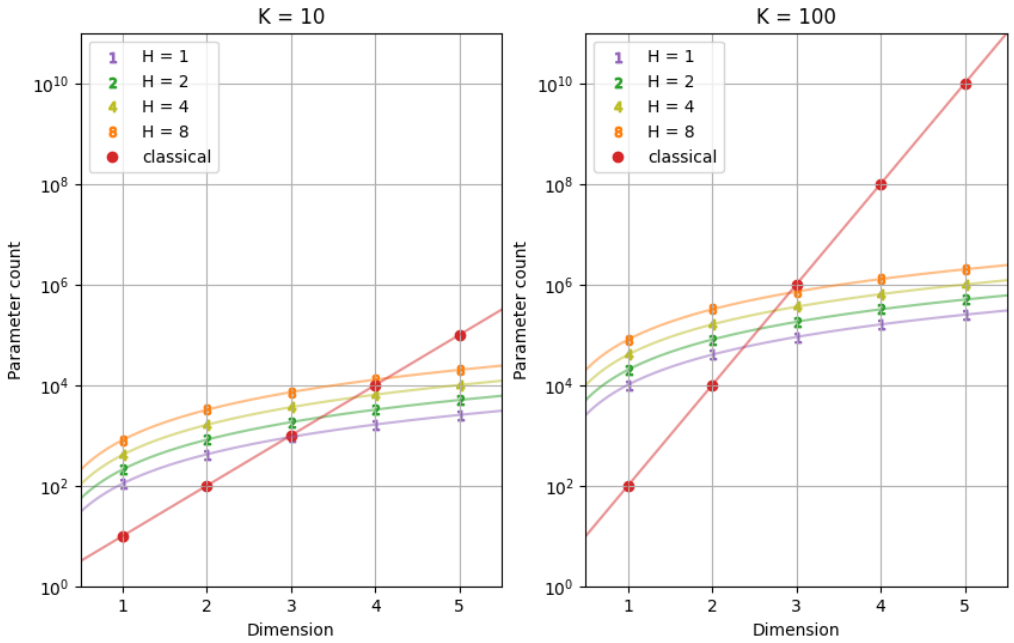


Figure 4.10: Parameter counts for classical interpolation and coordinate-MLPs with different numbers of hidden layers.

Figure 4.10 shows parameter count curves for two different values of K. For low dimensions, classical interpolation is cheaper in terms of parameters, while coordinate-MLPs are cheaper for higher dimensions. The exact crossover point depends on H and K.

Our experimental results show that, for equal parameter counts, coordinate-MLPs can outperform classical interpolation in two dimensions. Considered in combination with the parameter count curves in figure 4.10, we suspect that the same would be true in higher dimensions, and the performance difference will be more dramatic as the dimensionality is increased.

4.6 Limitations

We made a number of simplifications in order to make this project feasible. Some of these could potentially affect the validity of our results. Here we discuss the most major concerns, and how they might be overcome by future work.

Model complexity

We use parameter count as a convenient proxy for model complexity. In practice, not all parameters are equally important — to maintain a high PSNR, one parameter may require high precision, while another may be quantized to a high degree without any noticeable effect.

A better measure would be information content — how many bits are required to store all of the parameters? With a certain information budget, what is the best performance a particular model can achieve?

A recent paper[2] shows how coordinate-MLPs can be compressed by quantizing the weights of the network. A similar approach could be used to estimate information content, as follows:

1. For a given model, collate all the parameter values across multiple runs.
2. Use some quantization scheme to combine the values into some number of bins, to give a probability distribution over the values.
3. Calculate the Shannon entropy of the resulting discrete probability distribution, with $\sum_x -p(x) \log p(x)$.

Signal variety

All of our coordinate network experiments were based on signals from one particular image. Similar experiments should be repeated for other images. It would also be worth consider other sources of signals, higher dimensional signals, and artificially constructed signals.

Generalisation

We used training error as our measure of performance, with the assumption that training and test error would be similar for the low parameter counts we used. For more certainty in the results, it would be wise to repeat the experiments, but with some samples left out of training, to calculate test error.

Hyperparameter choices

Our neural networks all had hidden layers of equal width, determined by our particular architecture schema. Other architectures should be considered.

Chapter 5

Conclusions

The purpose of this thesis has been to compare classical and recent methods for signal interpolation. Chapters 2 and 3 gave brief introductions to these methods, with a focus on showing the commonalities between them.

Our experiments in Chapter 4 showed that sinc performs well as a generating function for a positional encoding. More significantly, we show that coordinate-MLPs can outperform classical methods in at least one case — interpolation of natural images. Our experiments were limited to a specific signal, generating function, and architecture schema. As such, we cannot draw strong conclusions, but instead our experiments should be seen as exploratory, showing some trends that may be worth investigating further.

Our main contribution is perhaps not our specific experiments, but the demonstration that these methods *can* be compared, with potentially insightful results. Future work could use a similar framework to produce a more robust set of results and more certainty in their conclusions.

In addition, our calculations in Chapter 4 show how model complexity, as measured by parameter count, is highly dependent on signal dimensionality. This seems like a promising avenue of investigation, that might help explain why deep learning performs well on high-dimensional data. We hope that future work will lead to a better understanding of deep neural networks, eventually providing level of mathematical certainty comparable with that of Shannon's famous results.

Bibliography

[1] Eirikur Agustsson and Radu Timofte. "NTIRE 2017 challenge on single image super-resolution: dataset and study". In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 2017, pp. 126–135.

[2] Emilien Dupont et al. *COIN: COmpression with Implicit Neural representations*. 2021. arXiv: 2103.03123 [eess.IV].

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A method for stochastic optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[5] Erwin Kreyszig. *Advanced Engineering Mathematics 9th Edition*. John Wiley & Sons, 2007.

[6] Cristian Constantin Lalescu. *Two hierarchies of spline interpolations. Practical algorithms for multivariate higher order splines*. 2009. arXiv: 0905.3564 [cs.NA].

[7] Alex Mackay. *Supplementary materials for my honours project*. URL: https://github.com/amackay/honours_project.

[8] Ben Mildenhall et al. "NeRF: Representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65.1 (2021), pp. 99–106.

[9] Adam Paszke et al. "PyTorch: An imperative style, high-performance deep learning library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[10] Nasim Rahaman et al. "On the spectral bias of neural networks". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 5301–5310.

[11] Sameera Ramasinghe et al. "On the effectiveness of neural priors in modeling dynamical systems". In: *arXiv preprint arXiv:2303.05728* (2023).

[12] Claude E Shannon. "Communication in the presence of noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21.

[13] Matthew Tancik et al. “Fourier features let networks learn high frequency functions in low dimensional domains”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 7537–7547.

[14] Michael Unser. “Sampling – 50 years after Shannon”. In: *Proceedings of the IEEE* 88.4 (2000), pp. 569–587.

[15] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[16] Wikipedia contributors. *Deep learning — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Deep_learning&oldid=1158747741. [Online; accessed 8-June-2023]. 2023.

[17] Jianqiao Zheng et al. “Trading positional complexity vs deepness in coordinate networks”. In: *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXVII*. Springer. 2022, pp. 144–160.