Askitas, Nikos

Working Paper

A Hands-On Machine Learning Primer for Social Scientists: Math, Algorithms and Code

CESifo Working Paper, No. 11353

Provided in Cooperation with:
Ifo Institute – Leibniz Institute for Economic Research at the University of Munich

Suggested Citation: Askitas, Nikos (2024) : A Hands-On Machine Learning Primer for Social Scientists: Math, Algorithms and Code, CESifo Working Paper, No. 11353, CESifo GmbH, Munich

This Version is available at:
https://hdl.handle.net/10419/305595

CESifo WORKING PAPERS

11353
2024

September 2024

A Hands-On Machine Learning Primer for Social Scientists: Math, Algorithms and Code

Nikos Askitas

CESifo

Impressum:

CESifo Working Papers
ISSN 2364-1428 (electronic version)
Publisher and distributor: Munich Society for the Promotion of Economic Research - CESifo GmbH
The international platform of Ludwigs-Maximilians University's Center for Economic Studies and the ifo Institute
Poschingerstr. 5, 81679 Munich, Germany
Telephone +49 (0)89 2180-2740, Telefax +49 (0)89 2180-17845, email office@cesifo.de
Editor: Clemens Fuest
https://www.cesifo.org/en/wp
An electronic version of the paper may be downloaded
· from the SSRN website: www.SSRN.com
· from the RePEc website: www.RePEc.org
· from the CESifo website: https://www.cesifo.org/en/wp

CESifo *Working Paper No. 11353*

A Hands-On Machine Learning Primer for Social Scientists: Math, Algorithms and Code

Abstract

This paper addresses the steep learning curve in Machine Learning faced by non-computer scientists, particularly social scientists, stemming from the absence of a primer on its fundamental principles. I adopt a pedagogical strategy inspired by the adage ”once you understand OLS, you can work your way up to any other estimator,” and apply it to Machine Learning. Focusing on a single-hidden-layer artificial neural network, the paper discusses its mathematical underpinnings, including the pivotal Universal Approximation Theorem—an essential ”existence theorem”. The exposition extends to the algorithmic exploration of solutions, specifically through “feed forward” and “back-propagation”, and rounds up with the practical implementation in Python. The objective of this primer is to equip readers with a solid elementary comprehension of first principles and fire some trailblazers to the forefront of AI and causal machine learning.

JEL-Codes: C010, C870, C000, C600.

Keywords: machine learning, deep learning, supervised learning, artificial neural network, perceptron, Python, keras, tensorflow, universal approximation theorem.

Nikos Askitas
Forschungsinstitut zur Zukunft der Arbeit GmbH
IZA, Bonn / Germany
Askitas@iza.org

Based on part of a course given at the University of Luxembourg in 2023. For the code in Section 3 I used ChatGPT3.5 for speed which I adapted to my needs. Readers who want to start with Python will find a wealth of knowledge online or might work through a hands-on course like e.g. Effective Programming Practices for Economists but are also encouraged to use OpenAI’s ChatGPT (or Google’s gemini) as a tutor. To streamline using ChatGPT while coding you might use askgpt. A Python module which works a bit like Github copilot but cheaper. Jupyter notebooks with some code related to this paper is deposited at the IDSC Dataverse.

A version of this paper was presented at the research retreat of IZA - Institute of Labor Economics. I would like to thank the participants for comments and in particular Andrew Oswald for in addition taking the time to read an earlier draft.

1 Introduction

Machine Learning (ML) has made significant strides due to the confluence of enhanced computing power, refined algorithms, and progress in its mathematical underpinnings[1]. Machine Learning techniques are used widely in life and empirical research[2][3]: speech and image recognition, natural language tasks (i.e. translations, chat bots etc), medical diagnosis, a wide range of prediction tasks, recommendation systems (from e-commerce, to the labor market i.e. jobs boards etc), autonomous vehicles, filtering of all kinds (e.g. email spam detection, fraud detection etc), in stock market trading, fraud detection. In causal machine learning it is used to remedy the situation in which a "treatment" comes from observational data and may or may not be randomly assigned. In that case we might use machine learning to control for covariates without needing to know the functional form of their effect on treatment selection[4].

However, the absence of a foundational primer tailored to non-computer scientists, particularly social scientists, results in a steeper than necessary learning curve. This challenge is compounded by field-specific jargon and a disconnection between mathematical concepts, algorithms, and code. This paper seeks to address this gap. In the spirit of the adage "once you understand OLS, you can work your way up to any other estimator," I contend that comprehending artificial (feed-forward) neural networks (ANN) serves as a gateway to understanding the entire spectrum of modern machine learning, including the latest advancements in large language models and even more cutting edge developments like quantum machine learning.

For social scientists, especially economists, who do not "just" concern themselves with prediction in a business context but whose work might have more profound societal or policy

[1] This diversity of contributing factors in return might inhibit adoption for many.

[2] For example a good place to start on machine learning for social scientist and text to data approaches might be Grimmer et al. [7].

[3] An emerging "market" with pre-trained, off-the-shelf models where researchers might shop for models: https://huggingface.co/. First signs of a trend among young economics with job market papers.

[4] A good place to start on causal machine learning might be Huber [9].

impact understanding the foundations of Machine Learning from first principles is a moral obligation and a *sine qua non* both when they add it to their research toolkit as well as when they consume it (e.g. Generative AI) as an end-user or provide guidance to the general public and the press in order to fend off the useless, unfounded palaver, on the topic, flooding us.

This paper will discuss Universal Approximation Theorems, existence theorems which constitute the mathematical underpinnings of ANNs and imply that they can indeed be trained to approximate any function with mild restrictions imposed on it (e.g. continuous with compact support). Additionally it will discuss the algorithmic side of searching for solutions when training models (i.e. looking for suitable model parameters) and provide practical implementations using `Python` code with `keras`[5] and `tensorflow`[6].

I use an informal exposition style whose purpose is to explain the basic principles and supply the reader with the necessary intuition for a surefooted entry into the wonderful world of ML. The aim is that a reader will understand the mathematical foundation of ANNs as well as write code to implement them but also, equipped with a solid theoretical foundation, be able to interpret algorithm behavior. By doing so, the paper aims at guiding the reader from foundational knowledge to the frontiers of AI and (causal) machine learning. Put differently this paper has a very modest goal: to create a solid foundation on first principles for a social scientist who wishes to add Machine Learning to their research toolkit. A reader will then be able to appreciate more involved ML accounts for economists such as Athey and Imbens [3] or Mullainathan and Spiess [13].

This paper is organised as follows: Section 2 discusses the mechanics of feed forward ANNs using visual representations and sets language and terminology. Section 3 discusses a universal approximation theorem which implies that feed-forward ANNs with just a single hidden layer can approximate any well behaved function at the expense of adding neurons to the layer. Functions need to satisfy mild requirements such as being continuous and having

[5]https://keras.io/
[6]https://www.tensorflow.org/

compact support, certainly overlapping with data generating mechanisms a social scientist might encounter. I also sketch the proof of the theorem in a special case. In Section 4 I approximate the XOR function (a tiny labelled dataset) with an ANN in `Python` and `keras` and give the reader the play by play of the code. In the last Section 5 I give a helicopter view of the wonderful world of ML and some trailblazers that might help readers zero in on regions of ML suitable to their own interests. Appendix 5 contains `Python` code for the illustrations of Section 3 which is generated by ChatGPT and adopted to suit my needs.

2 What can ANNs do for a social scientist?

Mathematically, Artificial neural networks can be used to approximate certain *"well behaved functions"*

$$f\colon \mathbb{R}^m \supset X \to \mathbb{R}^n.$$

Such functions need to be continuous and have compact support X. In Social Science such functions occur whenever we have a number m of covariates that might predict (determine, be conjectured to *"explain" or simply correlate with* etc.) an n-dimensional outcome. Whenever then we have a number of observations in a dataset we assume that the underlying data generating mechanism is continuous with compact support. Such an assumption is in a significant number of cases plausible.

Typically when $n = 1$ we speak of an outcome that we need to predict, explain etc and such outcomes might be unemployment rate, housing prices, economic sentiment, educational attainment etc. The covariates might be survey answers, administrative data or high dimensional vectors that represent free text. In traditional econometrics a social scientist will run some flavour of OLS *"controlling for covariates"* to demonstrate some covariates play a more important role than others. Whenever we don't have a theory or the dimension m is huge we might want to approach the problem in a way more agnostic about what the functional form of the underlying data generating mechanism is and in these cases an ANN is the tool of choice[7]. Figure 1 is an example of an ANN which could be used to approximate a function f when $m = 3$ and $n = 1$. Reading the neural network from left to right we speak of a 3-dimensional *"input layer"* (x_1, x_2, x_3), a 1-dimensional *"output layer"* (y) and a single 5-dimensional *"hidden layer"* $(h_1, \ldots, h_5)$.

In the context of neural networks one speaks of *"training"* a model which basically means using the observations (x_1, x_2, x_3, y) to find weights w_{ij}, w_i which produce outputs $\hat{y}$ so that

[7]This is in particular the case of causal machine learning. Whenever we have a treatment from observational data and not e.g. a proper RCT and want to control for covariates that may confound the treatment assignment but don't know the functional form of how they do so machine learning is the best that we can do.

some measure of error based on the differences of y and $\hat{y}$ is as small as possible.

We then speak of *"supervised learning"* and we say that such learning can take place whenever we have a *"labelled"* dataset[8]. In this example the dataset consists of the data points (x_1, x_2, x_3) and the labels are the y's. In contrast to the case of an OLS where the parameters of the model are found analytically finding the parameters in the case of an ANN is not possible analytically. We say that the ANN in Figure 1 has *"width"* 5 (the number of *"nodes"* or *"neurons"* of the hidden layer). Neural networks may have more than one hidden layers. The number of such hidden layers is the *"depth"* of the ANN, hence *"deep learning"*. The example of Figure 1 has depth 1 while Figure 2 shows an example of depth 2. A neural network of depth zero is just linear regression.

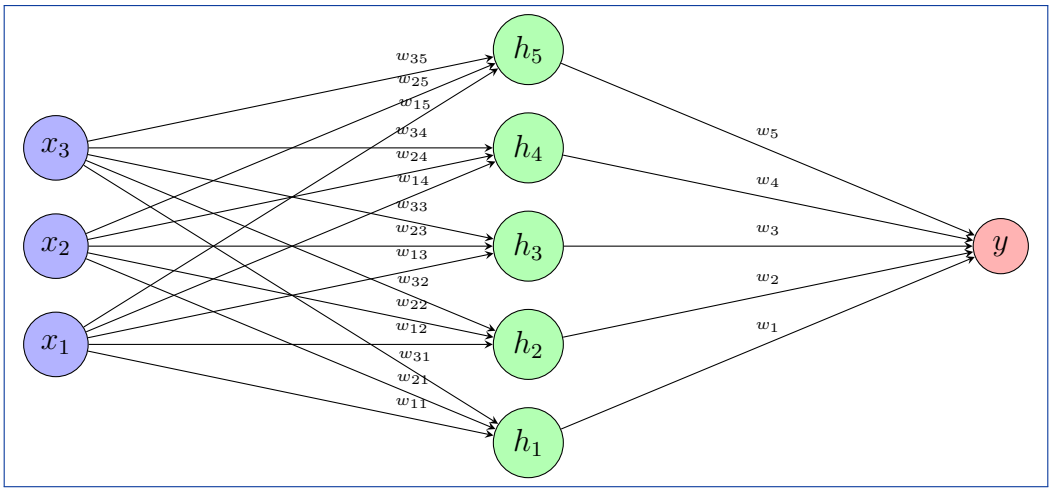


Figure 1: An ANN with one input layer with three nodes, one hidden layer with five nodes and one output layer with one single node.

The input data is entered at the input layer and *"feeds forward"* from left to right. At each layer each node/neuron receives a linear combination of the data and weights coming from all neurons of the previous layer that point to it with a weighted edge/synapsis and adds a constant of its own called the *"bias"* and subsequently applies an *activation function*

[8]So we approximate a function whose form we don't know calling elements in its domain "the data" and their images "labels". This paper focuses on supervised learning. The unsupervised type of (machine) learning techniques are used whenever we have data but no function i.e. no labels. Such is the case of a corpus of documents for example which might be clustered agnostically by e.g. a k-means algorithm or on which topic analysis is done using an LDA algorithm (Latent Dirichlet Allocation).

σ assigned to the layer. For example in Figure 2 the hidden node $h_{1,k}$ receives:

$$\sum_{i=1}^{3} w_{i,k} x_i,$$

it then adds its bias $b_{1,k}$ applies the layer's activation function σ_1 and then emits (*"fires", "outputs"*):

$$x_{1,k} = \sigma_1(\sum_{i=1}^{3} w_{i,k} x_i + b_{1,k}), \text{ for } k = 1, 2, 3, 4, 5.$$

Now the hidden layer $h_{2,s}$ receives:

$$\sum_{i=1}^{5} v_{i,s} x_{1,i},$$

adds ist bias $b_{2,s}$, applies the activation functions of its layer σ_2 and emits (*"fires"* or outputs):

$$x_{2,s} = \sigma_2(\sum_{i=1}^{5} v_{i,s} x_{1,i} + b_{2,s}).$$

Finally the output node receives:

$$\sum_{i=1} 4 u_i x_{2,i},$$

and after adding its own bias b and applying its activation function σ it outputs:

$$\sigma(\sum_{i=1}^{4} u_i x_{2,i} + b).$$

This concludes this section in which we formally introduced all the parameters of which an ANN is made up, the language and jargon that might be used in this context and how input feeds forward through an ANN to produce an output. The logical next question is why on earth would such a thing have a chance at helping us approximate observed but elusive

functions? The next section is dedicated to answering this question.

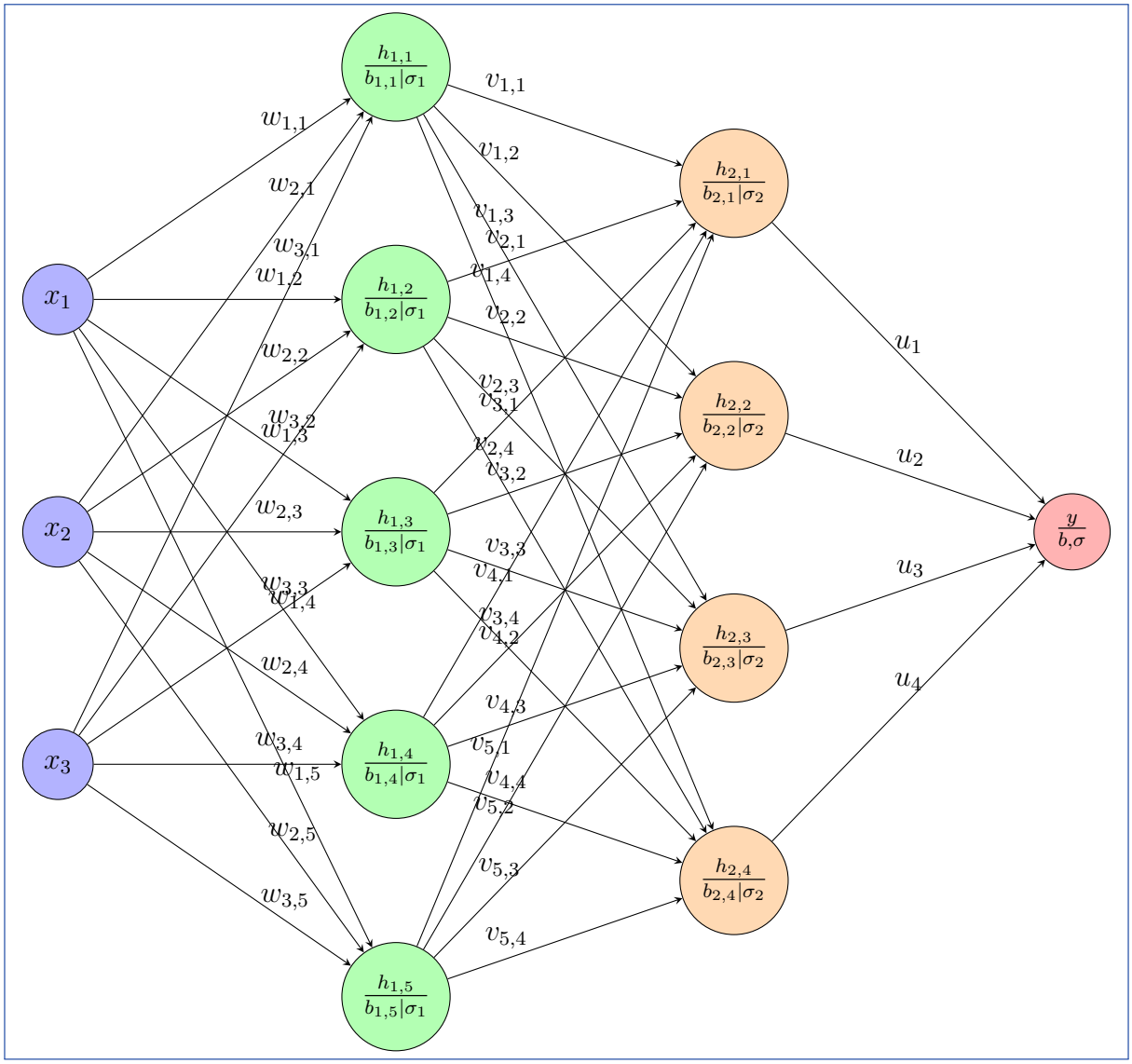


Figure 2: An ANN with one input layer with three nodes and one output layer with one single node. Its depth is 2 i.e. it has two hidden layers: one hidden layer with five nodes and one with four nodes. This examples is fully labelled with all ingredients that built an ANN.

3 Universal Approximators

A universal approximation theorem is an existence theorem which states that certain well behaved functions may be approximated by a certain form of simply built functions. The original proof of the one that relates to neural networks appears in [6]. In the language of [6] the paper shows that *"finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function"*. More formally, with less mathematical lingo and in more generality our universal approximation theorem states that:

Universal Approximation Theorem [1]. *Let* $C(X, \mathbb{R}^m)$ *be the set of continuous functions from a subset* $X \subset \mathbb{R}^n$ *to* $\mathbb{R}^m$ *and* $\sigma \in C(\mathbb{R}, \mathbb{R})$. *Then* σ *is not polynomial iff for every* $n \in \mathbb{N}$, $m \in \mathbb{N}$, *compact* $K \subseteq \mathbb{R}^n, f \in C(K, \mathbb{R}^m), \varepsilon > 0$ *there exist* $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ *such that if* $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$ *then* $\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$, *where* $(\sigma \circ x)_i = \sigma(x_i)$.

The theorem says that "well behaved functions" (i.e. continuous with compact support) can be arbitrarily (i.e. up to ϵ) approximated by an affine transformation (i.e. $A \cdot x + b$) followed by a point wise application of any non-polynomial function (i.e. σ) followed by a linear transformation (i.e. C). It also says that when σ is polynomial this is not true[9]. In other words it can be restated as saying that:

> Any continuous function $f \colon \mathbb{R}^m \supset X \to \mathbb{R}^n$, with compact support X can be approximated, at any desired precision, by a neural network of one single hidden layer if we are willing to add enough nodes to it (k).

This is a beautiful and powerful theorem which, if we are willing to believe it, guarantees the existence of a neural network (i.e. parameters like weights, activation functions, biases

[9]For the mathematically minded reader who might remember their elementary mathematical analysis this does not contradict the Stone–Weierstrass theorem which states that: "well behaved" functions have arbitrarily precise polynomial approximations.

and width) which approximates any function leaving us with the "simple task" of finding it. We then thus enter the world of Machines (computers) which learn their parameters (algorithms).

Before moving to the next section where we will start looking into the search for such neural networks, we will sketch a proof of the theorem in [6] in order to achieve that the reader is at ease with believing the theorem by understanding the flavor of the ingredients that go into a proof. The reader who feels comfortable without the proof can skip the rest of this section. In what follows we sketch how we could construct an arbitrarily close approximation of a function $f : [0, 1] \to [0, 1]$ by means of a neural network using the sigmoid as an activation function:

$$\sigma(x) = \frac{1}{1 + e^{-sx}},$$

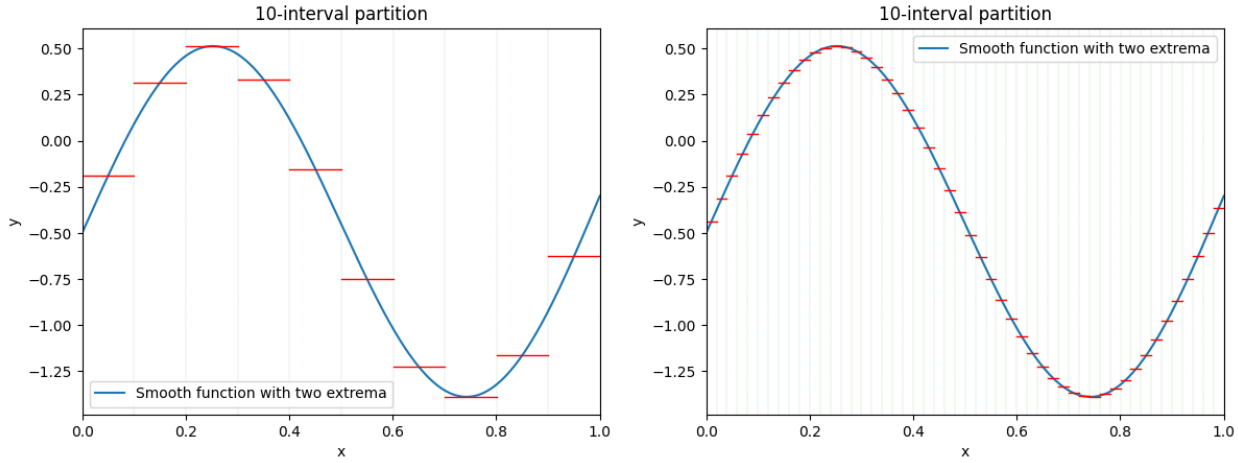
Here the parameter s is the steepness and determines how sharp we change from values near 0 to values near 1. A function defined on the unit interval with constant support on some subinterval $[a, b] \subset [0, 1]$ will be called a bump function.

By the Mean Value Theorem for Intervals it is obvious that we can approximate any continuous function f to any desired prediction by a linear combination of a finite number of bump functions by partitioning the unit interval in subintervals and defining appropriate bump functions on them. The finer the partition the better the approximation. Figure 3 demonstrates such approximation by bump functions and the code that generated the example is in Listing 4.

It remains to show that we can approximate any bump function by using linear combinations of Sigmoid functions. The reader might play with the Python code in Listing 5 to produce output like in Figure 4 and convince themselves that we can apply linear transformations and the Sigmoid to get a step function of any kind. The sketch is only an existence construction and it is not concerned with bounding the number of nodes of the hidden layer. In practice we can do better. Now using the python code in the Listing 6 we can convince ourselves that we can produce arbitrarily close approximations of any bump function (see

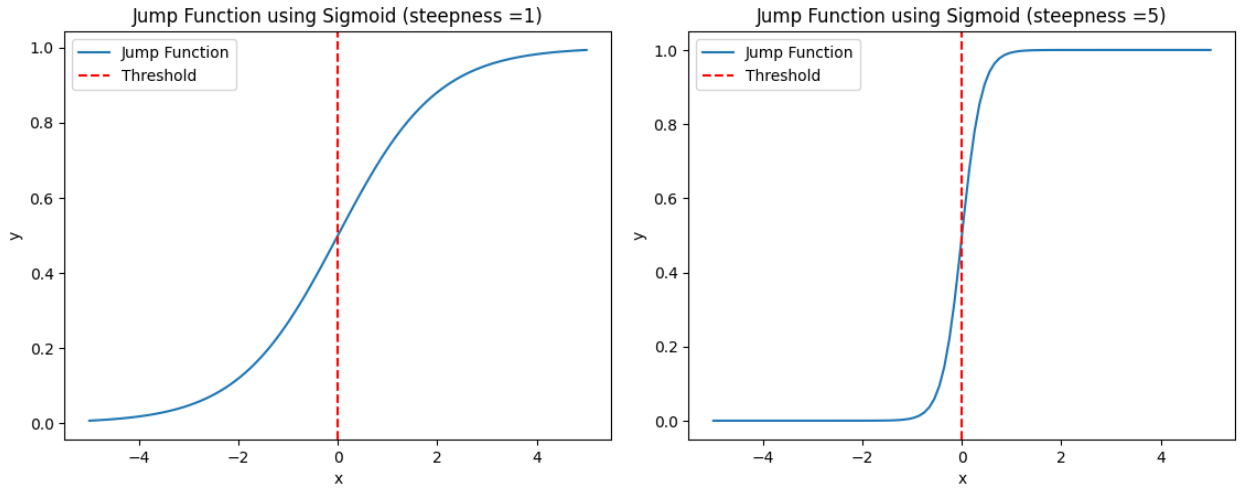
Figure 5). This concludes the sketch of the proof.

I leave it as an exercise to the reader to produce a sketch of the proof of the statement that using the ReLU functions works as well. A hint on how to proceed is that any continuous function with compact support can be approximated arbitrarily closely by a piece-wise linear function.



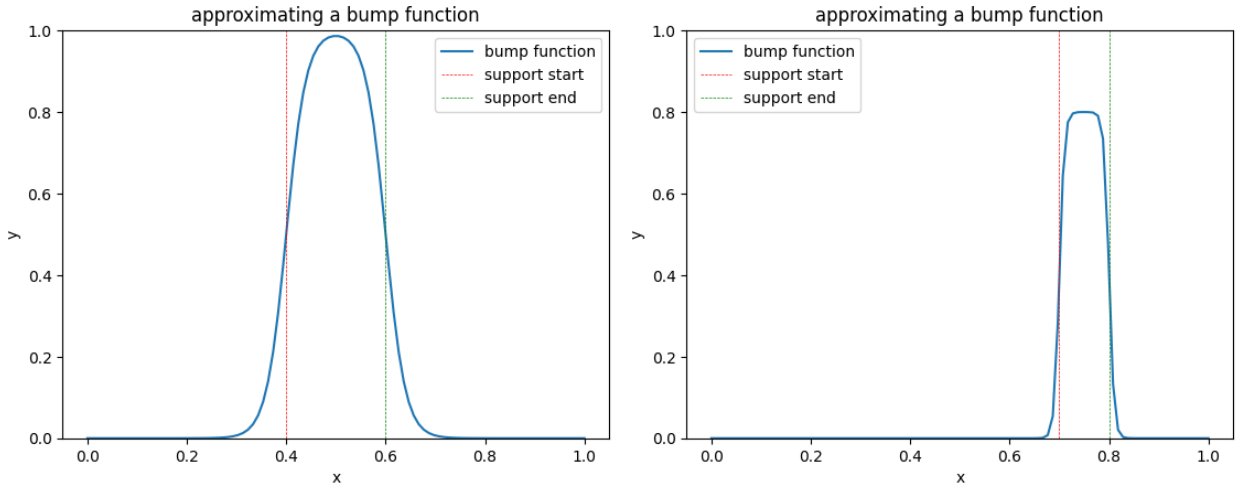
(a) *num_segments* = 10 in Listing 4 (b) *num_segments* = 50 in Listing 4

Figure 3: Example of approximating a function by bump functions.



(a) Sigmoid with steepness 1 (b) Sigmoid with steepness 5

Figure 4: Sigmoids produced with code in Listing 5.



(a) In Listing 6: $bump\_start = .4$, $bump\_end = .6$, $steepness = 200$, $jump\_height = .8$.

(b) In Listing 6: $bump\_start = .4$, $bump\_end = .6$, $steepness = 50$, $jump\_height = 1$.

Figure 5: A composition of linear transformation and Sigmoids approximates any bump function with any support and any height arbitrarily close.

The take-away from this section is that we have seen a simple demonstration in a concrete case that we can approximate any continuous function with compact support with a neural network of a single hidden layer at the expense of increasing its width. A number of other papers starting with Lu et al. [11] in 2017 show that neural networks with bounded width and arbitrary depth are also universal approximators of a large class of functions. The class of functions in [11] are Lebesgue-integrable functions $f : \mathbb{R}^n \to \mathbb{R}$ with respect to the L^1 distance, the depth is bound by $n + 4$ and the activation function is the so-called ReLU function (rectified linear unit) defined by: $\sigma(x) = \frac{x+|x|}{2}$. A number of remarkable, more recent generalisations beyond our scope are also available and can be discovered starting from [1].

We close this section with a remarkable result in Maiorov and Pinkus [12] which shows that an analytic, strictly monotone, sigmoidal activation function exists with which we can approximate arbitrarily well any continuous function on any compact domain by a multilayer feedforward perceptron (ANN) of depth 2 with hidden layers of width $3n$ and $6n + 3$.

4 Training a neural network

Armed with an existence theorem which assures us that any continuous function with compact support can be arbitrarily closely approximated by some ANN we will now use the Python modules `keras` and `tensorflow` to approximate the so-called XOR function (exclusive or) defined on the set $\{(0,0),(0,1),(1,0),(1,1)\}$ as follows: $XOR((0,0)) = XOR((1,1)) = 0$ and $XOR((0,1)) = XOR((1,0)) = 1$. The function is called "exclusive or" because it can be thought of as the truth table of the statement "either A or B but not both" for two statements A and B. The domain of the XOR function are then all possible pairs of truth values for these statements.

This innocent looking function is considered as having contributed to the first AI winter (a period of reduced funding for AI following a hype period) in 1973[10]. It can be easily proven by elementary means that an ANN of depth 1 with polynomial activation function cannot approximate the XOR function. The geometric way to think about this is that there is no straight line on the plane which separates the two vertices of the unit square on one diagonal $\{(0,0),(1,1)\}$ from those on the other $\{(0,1),(1,0)\}$[11]. We will leverage this toy example to familiarize ourselves with training a neural network for supervised learning. This very simple example is no different than anything a social scientist might encounter which falls within the application domain of ANN. From a social scientist's perspective we are dealing with a dataset of three variables and four observations:

A very simple dataset

X	**Y**	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

with covariates X, Y predicting XOR or in machine learning lingo we have a labelled

[10] Start with [2] to dig into some of the history of the first AI winter.

[11] Increasing the depth makes it possible.

(XOR) dataset (X, Y). Everything we will do will apply to any other dataset with any number of covariates, labels and number of observations e.g. vectors representing words in a document with labels equal to binary sentiment or vectors representing a number of lagged values of a time series and labels representing the unlagged value (autoregressive model), or (the case of generative AI) properly encoded sequences of letters (sentences) of length n from a corpus of documents and the character that succeeds them. To make this point in what follows, for clarity and language, I supply the broad strokes of the mechanics in some real cases on how to get labelled data before we proceed with our toy example.

Autoregression: Given a time series Q_t for $t = 1, \ldots, n$ an autoregressive model is based on the belief that if the state of the world prior to time $t = t_0$ affects Q_{t_0} then since Q_t for all $t < t_0$ are part of that state the corresponding values Q_t should also affect Q_{t_0}. Typically economists, utilising not all but some number of lagged values, will estimate linear models. Assuming linearity however is a very restrictive approach since the functional form of the way Q_t's for $t < t_0$ affect Q_{t_0} is unknown. With machine learning we know that if a function exists then we can find it. So we might fix e.g. 7 lagged values to predict the eighth in which case our labelled dataset looks like this:

Time Series prepared for training an ANN

Lagged	**Unlagged**
$[\mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_3, \mathrm{t}_4, \mathrm{t}_5, \mathrm{t}_6, \mathrm{t}_7]$	t_8
$[\mathrm{t}_2, \mathrm{t}_3, \mathrm{t}_4, \mathrm{t}_5, \mathrm{t}_6, \mathrm{t}_7, \mathrm{t}_8]$	t_9
$[\mathrm{t}_3, \mathrm{t}_4, \mathrm{t}_5, \mathrm{t}_6, \mathrm{t}_7, \mathrm{t}_8, \mathrm{t}_9]$	t_{10}
$\ldots$	$\ldots$
$[\mathrm{t}_{\mathrm{n}-7}, \mathrm{t}_{\mathrm{n}-6}, \mathrm{t}_{\mathrm{n}-5}, \mathrm{t}_{\mathrm{n}-4}, \mathrm{t}_{\mathrm{n}-3}, \mathrm{t}_{\mathrm{n}-2}, \mathrm{t}_{\mathrm{n}-1}]$	t_n

Now we have a labelled dataset which looks like our toy example and would be amenable to an ANN approach.

Large Language Models: Lets say you want to train an ANN chatbot to respond in Lewis Carroll style. One approach would be to take "Alice in the Wonderland" and anything else he wrote and work as with the time series example. Any n characters occurring in the book

are in the domain of our function and they are mapped to the character that follows them so the data we use to train our ANN are these sequences of n characters and the labels are the characters that follows them. An example of what the title would look like is given in the table below for $n = 4$ without the encoding. The first four variables are the dataset the fifth one is our label:

1	**2**	**3**	**4**	**5**
A	l	i	c	e
l	i	c	e	
i	c	e		i
c	e		i	n
e		i	n	
	i	n		t
i	n		t	h
n		t	h	e
	t	h	e	
t	h	e		W
h	e		W	o
e		W	o	n
	W	o	n	d
W	o	n	d	e
o	n	d	e	r
n	d	e	r	l
d	e	r	l	a
e	r	l	a	n
r	l	a	n	d

Sentiment from text: A collection of documents with some thematic relation such as presidential speeches, or book reviews, or paper abstracts are called a document corpus. If we are dealing with, say, movie reviews they might be labelled as positive or negative in which case the task is to train an ANN to predict the sentiment from the text. Lets say we have a corpus with two documents:

Movie reviews with binary sentiment

Review	**Sentiment**
"this movie is some good shit"	1
"this movie bored me out of my skull"	0

First we need to form the "feature space" which is a vector space in which each document is a (sparse) vector. The feature space is essentially the union of all words from all documents in the corpus. You might drop short words like e.g. of length three or less (so drop: is, me, out, of, my) and do a stemming like drop plurals, ing's etc (so in our example bored might become bore). Such treatment may often be referred to as preprocessing. Now the corpus looks as follows:

Movie reviews: after some preprocessing

Review	**Sentiment**
"this movie some good shit"	1
"this movie bore skull"	0

Finally we tokenize the documents so documents become arrays of features (stems):

Tokenized movie reviews

Review	**Sentiment**
[this, movie, some, good, shit]	1
[this, movie, bore, skull]	0

Now take the union of all features from all documents: {this, movie, bore, skull, some, good, shit} and order them in some arbitrary but fixed way e.g. [this, movie, bore, skull, some, good, shit]. The cardinality of the set of features (i.e. 7) is the dimension of our feature space which is nothing but $\mathbb{R}^7$. Now our (vectorized) documents look like this:

Vectorised movie reviews

Review	**Sentiment**
[1, 1, 0, 0, 1, 1, 1]	1
[1, 1, 1, 1, 0, 0, 0]	0

In encoding our documents we might just enter 1 or 0 depending on the presence or a word or not, this is called one-hot encoding. There are other encodings like frequency encodings etc. Now our labelled data looks very much like our XOR example.

Getting back to our toy example, looking for a single-hidden-layer neural network to approximate XOR will basically be an empirical task. Now we are in the world of searching for a solution we know exists so we need to empirically try several things. For example we need to try several widths and activation functions. For each width and activation function we are looking for a number of parameters that do the job. For example in Figure 2 we are looking for $w_{i,j}, v_{i,j}, b_{i,j}$ a total of 48 parameters[12].

How would we go about searching? There is only one way: randomly initialise the parameters and find a way to improve them: send all your data through the ANN (this is called forward-propagation), then measure the error between what the network predicts and the actual labels. Then go back and adjust the weights so as to reduce the error (this is called back-propagation). Then rinse and start over by feeding the data forward, measuring the error, adjusting the weights etc. When to stop? When you are not making progress any more.

In propagating forward and backward there are decisions to make. Do I forward all of my data at once or in batches? How large should the batches be? In adjusting the weights in a direction that reduces the error do I take small or large steps (the learning rate). Finally what measurement of the error do I use? Luckily all of that is easily done with the Python module `keras` a user-friendly interface to `Tensorflow` which does the heavy lifting of back-propagation (linear algebra, chain rule etc)[13].

The mathematics of updating the weights all at once is essentially an application of the chain rule: starting from right to left successively adjust each model parameter p by $p - r\frac{\partial E}{\partial p}$ where r is the learning rate and E is the prediction error. Computationally at the heart of how Tensorflow does back-propagation is the "computational graph" which is a type of directed acyclic graph that turns computations into data and is fundamental

[12] 15 weights from the input layer to the first hidden layer + 5 biases at the first hidden layer + 20 weights from the first to the second hidden layer + 4 biases at the second hidden layer + 4 weights from the second hidden layer to the output layer

[13] The reader interested in the gory details can read Chapter 2 of Chollet [5], a wonderful introduction to Deep Learning with Keras which in addition to code and examples includes some very enjoyable and informative prose accessible also as an audiobook.

for deep learning. It is a computational tool to compute derivatives of arbitrarily complex functions build by operations like addition and multiplications, activation functions and function compositions. As you back-propagate further to the left you apply chain rules to express $\frac{\partial E}{\partial p}$ for every parameter p in terms of other parameters q further to the right: $\frac{\partial E}{\partial q} \cdot \frac{\partial q}{\partial p}$. Tensorflow does all of that all at once[14].

The code that does all that is in Listing 1 and a visual representation of the ANN it builds is in Figure 6. Lets discuss the play by play of the code. Lines 1-5 are just module imports, line 7 is a numpy array with the dataset, line 8 an array with our labels. Line 10 just instantiates our neural network. Line 11 adds our hidden layer with 16 nodes and at the same time declares that the input layer is two-dimensional. It is a dense layer in the sense that the network is "fully connected" i.e. any two nodes of two successive layers are connected with a weighted edge. The activation function of the layer is the sigmoid function[15] . Line 12 adds the output layer of dimension 1 (our labels). Its activation function is the hyperbolic tangent. Line 13 instantiates the Adam optimiser[16] our chosen method of stochastic gradient descent and declares the learning rate. Line 14 compiles our model by declaring our loss function[17] and our metric[18]. In Line 15 we fit the model i.e. train the network. The arguments of `model.fit` besides our data and labels says that we will run 100 iterations[19], sending the data through the current state of the network in batches of one data point. Line 16 makes prediction and line 17 prints them. Lines 19-21 predict the support of the XOR function

[14]Mathematically minded readers might ask "why not user polynomials instead of neural networks since the Stone-Weierstrass theorem is also a suitable existence theorem?". The answer is that while the two classes of functions (neural networks and polynomials) are equally "expressive" i.e. are able to approximate well behaved functions equally well they differ in terms of their computational tractability. In other words representing (existence) and learning functions (searching) are two different things. A recent ([10]) attempt to use the Kolmogorov-Arnold representation theorem (*"multivariare functions can be written as sums of compositions of univariate functions"*) has shown considerable success in producing so-called KANs i.e. Kolmogorov-Arnold Networks. The univariate functions in play are splines whose parameters are then trainable resulting in KANs with nice technical properties which as the authors of [10] state *"are nothing more than combinations of splines and multi-layer preceptrons"* like the ones this paper describes.

[15]Keras activation functions

[16]Keras optimizers

[17]Keras Losses

[18]Keras metrics

[19]In each epoch all of our data gets fed forward in the batches we declare and for each batch model-parameters will be adjusted by back-propagation.

again and line 24 saves the weights of the model in an array w which is used in Listing 3 to write down the resulting model analytically as a Python method while Figure 6 depicts the function visually with placing the elements of the array w as weights[20].

```
1  import numpy as np
2  import keras
3  from keras.models import Sequential
4  from keras.layers import Dense
5  from keras.backend import clear_session
6
7  training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
8  target_data = np.array([[0],[1],[1],[0]], "float32")
9
10 model = Sequential()
11 model.add(Dense(16, input_dim=2, name ="hidden",activation='sigmoid'))
12 model.add(Dense(1, name="output",activation='tanh'))
13 optimizer = keras.optimizers.legacy.Adam(learning_rate=0.1)
14 model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['accuracy'])
15 model.fit(training_data, target_data, epochs=100,batch_size=1, verbose=2)
16 prediction = model.predict(training_data).round()
17 print("prediction",prediction)
18
19 support = np.array([[1,0],[0,1]], "float32")
20 support_prediction=model.predict(support).round()
21 print("support_prediction",support_prediction)
22
23 #to use for writing the model down analytically in Python
24 w=model.get_weights()
```

Listing 1: Approximating the XOR function using an artificial neural network

The output of running this code might look like in Listing 2 which represents a successful training session. The reader is encouraged to play with various model configuration-parameters several times (optimizers, activation layers, width, depth, loss functions, metrics, learning rates etc). The search for suitable weights will not alway be successful and training

[20]There are many many more configuration parameters in keras. One worth mentioning is the layer weight regulariser also referred to as Lasso regularisation. Specified as e.g. `kernel_regularizer=l1(0.01)` or `kernel_regularizer=l2(0.01)` within a layer it modifies the chosen loss function by adding the L1 (L2) norm of all the layer's weights multiplied by 0.01. This is useful for reducing the number of relevant features in a model.

might complete sooner or later. This is due to the fact that we randomly initialise our parameters and in doing so we might be trapped in a local extremum or start closer or further to a solution[21].

```
Epoch 1/100
4/4 - 0s - loss: 1.0583 - accuracy: 0.2500 - 173ms/epoch - 43ms/step
Epoch 2/100
4/4 - 0s - loss: 0.4475 - accuracy: 0.5000 - 19ms/epoch - 5ms/step
...
Epoch 34/100
4/4 - 0s - loss: 0.3075 - accuracy: 0.7500 - 19ms/epoch - 5ms/step
...
Epoch 48/100
4/4 - 0s - loss: 0.1158 - accuracy: 1.0000 - 18ms/epoch - 5ms/step
...
Epoch 100/100
4/4 - 0s - loss: 5.3166e-04 - accuracy: 1.0000 - 18ms/epoch - 4ms/step
1/1 [==============================] - 0s 27ms/step

Data = [[0. 0.] [0. 1.] [1. 0.] [1. 1.]]
Labels = [[0.] [1.] [1.] [0.]]
Predictions = [[0.] [1.] [1.] [0.]]
```

Listing 2: A successful training session with the code in Listing 1

Running these should go quickly on a modern laptop and can be enjoyed with a glass of wine on the sofa after work. I would approach this as follows. Go down to width 2. Try several activation functions and number of epochs. Do you ever produce convergence? Increase the width, repeat and so one. Save the weights you find. Are they always the same up to permutation of the neurons within the hidden-layer? Do some choices of activation functions work better than others? Are there initial values that trap the search? These lines of code and this example can teach a reader a lot and provide important intuition.

Listing 3 is Python code which implements the weights we found in an analytic manner. Being able to understand the analytic form of the model we train might be more important in social science than in business especially if our work might have policy relevance.

More Python code with keras which demonstrates saving a model, loading a model or working with free GPUs and TPUs on Google Colab can be found in the accompanying

[21]For a fun investigation of the error space of an ANN of depth 2 trained to learn the XOR function the reader might enjoy Bland [4].

code in the IDSC Dataverse, together with a complete example of sentiment analysis using a movie reviews dataset. In the next section we will conclude with the outlook of where a reader might go from here and in which way this paper understands itself as the start of a reader's journey into Machine Learning.

```
def XOR(x,y):
    import numpy as np

    def relu(x):
        return max(0.0, x)

    def sig(x):
        return 1/(1 + np.exp(-x))


    result = np.vectorize(np.tanh)(np.dot(
                (np.vectorize(sig)(
                                    np.dot(np.column_stack((w[0][0], w[0][1])),
                                            np.array([ [x],[y]])
                                          ) + np.array(w[1]).reshape(-1, 1)
                                  )).T ,w[2]
                ))
    print(result)
    return(result[0][0].round())

output = XOR(1,0)
print(output)
```

Listing 3: Writing down the estimated function analytically

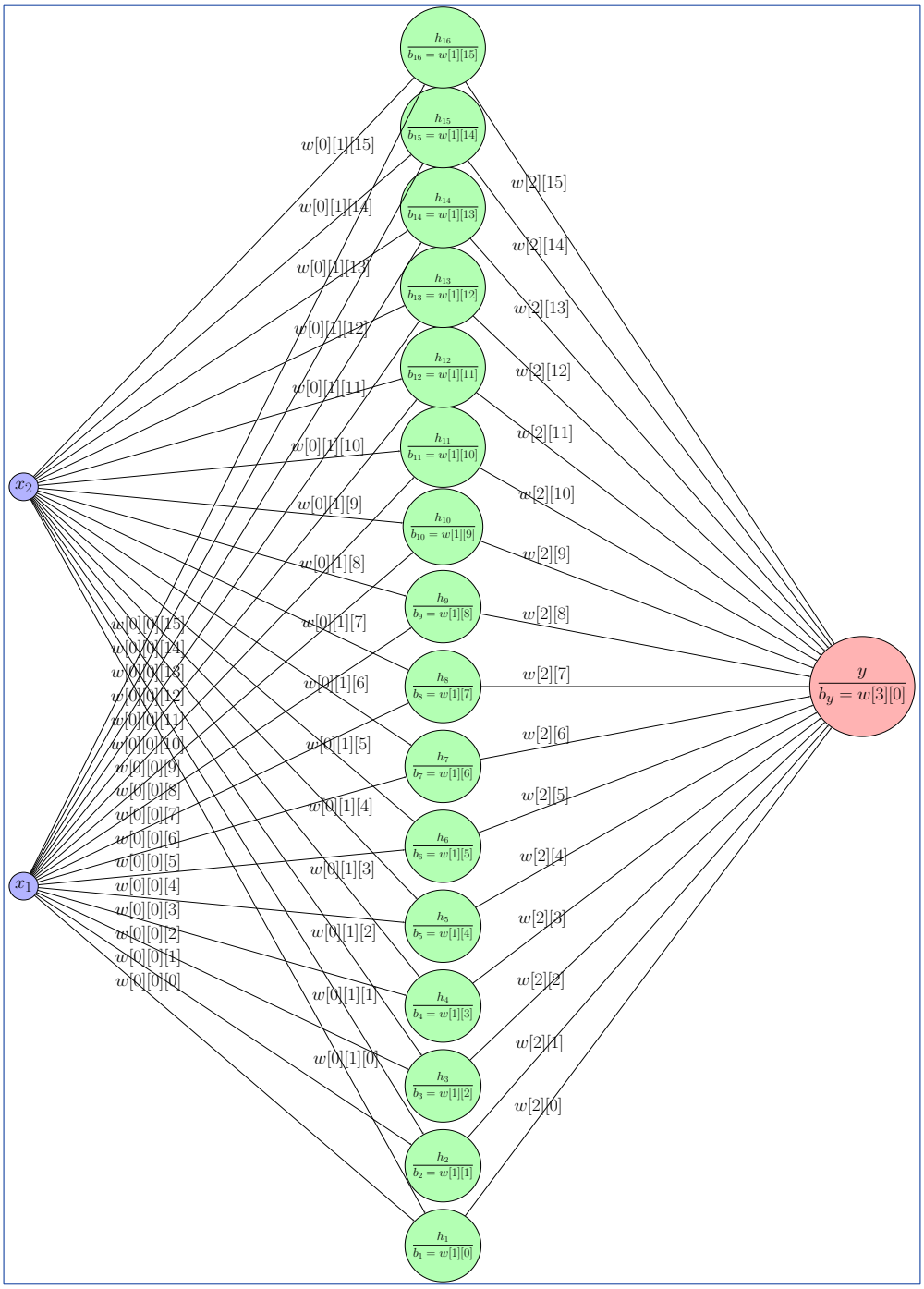


Figure 6: An ANN with a 2-dimensional input layer, one hidden layer of width 16 and one output layer with one node representing our XOR keras model. The weights of our pre-trained model are in an array w of length 4 and plotted at the right place of the figure. The hidden layer and the output layer have Sigmoid and hyperbolic tangent as activation functions.

5 Where to go from here

I hope to have given an elementary overview of the smallest amount of information a researcher might need in order to enter the world of Machine Learning on a sure foot and to have demonstrated that at least as far as a basic, vanilla flavored neural network is concerned we need nothing but elementary mathematics and that if we are willing to blackbox the computational heavy lifting inside Tensorflow then with the use of keras entering the world of ML is a manageable task achieved in just 24 lines of Python code. The rest of the vastness of Machine Learning all the way to the very latest developments in e.g Transformers architecture (Vaswani et al. [14]) or Mamba (Gu and Dao [8]) can be thought of as a series of improvements on previously known techniques in order to remedy various issues (e.g. performance) and failures (e.g. vanishing gradient)[22] when applied to different data contexts at different scales.

In that sense, for example, in the case of large language models where input data consists of all string sequences of a certain length occurring in a large corpus of texts and the labels are the succeeding character, Mamba was created in 2023 to address Transformers' computational inefficiency on long sequences. Transformers itself introduced in 2017 a so-called attention mechanism to improve on memory requirements of prior so-called Recurrent Networks. The latter are one of two types of networks in which, unlike feed-forward networks (such as the ones we are concerned with in this paper) where the data flows only from left to right, data is allowed to flow in both directions so that output from a node might be allowed to influence its input at a later time during the training. The other kind of bi-directional networks are the so-called Convolutional Neural Networks distinguished from the Recurrent type by having a so-called finite impulse. Finally, wrapping up this small excursion into the evolution of ANNs, Convolutional Neural Networks were created to address issues of vanishing or exploding gradients during back-propagation in fully connected neural networks. These networks are suitable for sequential data e.g. pictures or time-series and apply certain

[22] A good place to start digging in is: https://en.wikipedia.org/wiki/Machine_learning.

filters to the data to reduce the number of weights that need to be estimated when training on e.g. pictures[23]. Such filters can be thought of as e.g. aggregating daily data to monthly data in time-series analysis and can be found in modern graphic design software.

The breadth of elements involved in ML becomes an impediment for entry. These elements include mathematical and algorithmic understanding as well as programming skills. This paper provided a basic understanding of these. There is one more practical obstacle which must also be overcome that we did not deal with here, namely, the ability to deal with the vagaries of installing and troubleshooting the necessary Python packages. Although anyone serious about adding ML to their toolkit should learn to maintain their own programming environment one way to avoid having to do so at the beginning might be to resort to Google Colab (where one can always come back for access to GPUs and TPUs). Some basics can be picked up from this course: Effective Programming Practices for Economists and of course you can always google errors and issues with your environement and leverage the experience of others.

[23]For example if you are analyzing medical imaging e.g. x-rays in digital form you might have an image of 600 pixels by 600 pixels containing numbers from 0 to 255 expressing how bright the pixel is. In that case if e.g. you are training a model to identify e.g. a malicious tumor of some kind your input layer will be of dimension $600^2 = 360000$. In a fully connected network whose first hidden layer has 1000 nodes this adds $36 \cdot 10^7 + 1000$ weights and biases just between input layer and first hidden layer.

References

[1] Universal approximation theorem. https://en.wikipedia.org/wiki/Universal_approximation_theorem. Accessed: January-2024.

[2] Perceptrons (book). https://en.wikipedia.org/wiki/Perceptrons_(book). Accessed: January-2024.

[3] S. Athey and G. W. Imbens. Machine learning methods that economists should know about. *Annual Review of Economics*, 11:685–725, 2019.

[4] R. Bland. Learning XOR: exploring the space of a classic problem. Technical report, University of Stirling, Department of Computing Science and Mathematics, 06 2010.

[5] F. Chollet. *Deep Learning with Python*. Manning Publications, 2017. ISBN-13: 978-1617294433.

[6] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[7] J. Grimmer, M. E. Roberts, and B. M. Stewart. *Text as data: A new framework for machine learning and the social sciences*. Princeton University Press, 2022.

[8] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[9] M. Huber. *Causal analysis: Impact evaluation and Causal Machine Learning with applications in R*. MIT Press, 2023.

[10] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark. Kan: Kolmogorov-arnold networks, 2024.

[11] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/32cbf687880eb1674a07bf717761dd3a-Paper.pdf.

[12] V. Maiorov and A. Pinkus. Lower bounds for approximation by mlp neural networks. *Neurocomputing*, 25(1):81–91, 1999. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(98)00111-8. URL https://www.sciencedirect.com/science/article/pii/S0925231298001118.

[13] S. Mullainathan and J. Spiess. Machine learning: an applied econometric approach. *Journal of Economic Perspectives*, 31(2):87–106, 2017.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Appendix A. Python code

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate a smooth function with one local maximum and one local minimum
def generate_smooth_function_with_extrema(num_segments=10):
    x = np.linspace(0, 1, 100 * num_segments)
    y = np.sin(2 * np.pi * x) + 0.2 * x**2 - 0.5
    return x, y

# Plot the smooth function with one local maximum and one local minimum
x_values, y_values = generate_smooth_function_with_extrema()
plt.plot(x_values, y_values, label='Smooth function with two extrema')
# Draw vertical lines at each point of the partition and at x=1
for i in range(len(x_values) // 100):
    a = x_values[100 * (i - 1)] if i > 0 else 0
    b = x_values[100 * i]
    x_mid = (a + b) / 2
    y_mid = np.interp(x_mid, x_values, y_values)
    plt.axvline(x=x_values[100 * i], color='g', linestyle='--', linewidth=.1)
    plt.axhline(y=y_mid, xmin=a, xmax=b, color='r', linestyle='-', linewidth=1)

# Add a vertical line at x=1
plt.axvline(x=1, color='g', linestyle='--', linewidth=.1 )
# Add a horizontal line in the last partition interval
myx=(len(x_values)-1) // 100
print(myx)
a=x_values[100*myx]
b=x_values[len(x_values)-1]
x_mid = (a + b) / 2
y_mid = np.interp(x_mid, x_values, y_values)
plt.axhline(y=y_mid, xmin=a, xmax=b, color='r', linestyle='-', linewidth=1)
plt.xlim(xmin=0,xmax=1)
plt.xlabel('x')
plt.ylabel('y')
plt.title(f"{num_segments}-interval partition")
plt.legend()
plt.show()
```

Listing 4: Python code showing bump functions approximate an example function.

```python
import numpy as np
import matplotlib.pyplot as plt

def jump_function(x, threshold, jump_from, jump_to, steepness=1):
    return jump_from + (jump_to - jump_from) / (1 + np.exp(-steepness * (x - threshold))
    )

threshold_value = 0  # Point where the jump occurs
jump_from_value = 0  # Value before the jump
jump_to_value = 1    # Value after the jump
# Generate x values
x_values = np.linspace(-5, 5, 100)
# Generate y values using the jump function
y_values = jump_function(x_values, threshold_value, jump_from_value, jump_to_value)
# Plot the jump function
plt.plot(x_values, y_values, label='Jump Function')
plt.axvline(x=threshold_value, color='r', linestyle='--', label='Threshold')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Jump Function using Sigmoid')
plt.show()
```

Listing 5: Python code for a jump function.

```python
import numpy as np
import matplotlib.pyplot as plt

def jump_function(x, threshold, jump_from, jump_to, steepness):
    return jump_from + (jump_to - jump_from) / (1 + np.exp(-steepness * (x - threshold))
    )

def bump_function(x, bump_start, bump_end, jump_height, steepness):
    return jump_height*(jump_function(x, bump_start, 0, 1, steepness)  \
           -  jump_function(x, bump_end, 0, 1, steepness))

# Example usage
jump_height = .8
steepness = 200
bump_start = 0.7
bump_end = 0.8


# Generate x values
x_values = np.linspace(0, 1, 100)

# Generate y values using the bump function
y_values = bump_function(x_values, bump_start, bump_end, jump_height, steepness)

# Plot the bump function
plt.plot(x_values, y_values, label='bump function')
plt.axvline(x=bump_start, color='r', linestyle='--', label='support start',linewidth=.5)
plt.axvline(x=bump_end, color='g', linestyle='--', label='support end',linewidth=.5)
plt.ylim(ymin=0,ymax=1)
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('approximating a bump function')
plt.show()
```

Listing 6: Python code for a bump function