

Kolmogorov-Arnold Networks for Financial Market Prediction and Analysis

A dissertation submitted for

COMP8800: Project and Dissertation

as part of the program: MSc Artificial Intelligence

supervised by
Dr Frank Wang

UNIVERSITY OF KENT

8,166 words

September, 2024

UNIVERSITY OF KENT

ACCESS TO A MASTER'S DEGREE OR POSTGRADUATE DIPLOMA DISSERTATION

In accordance with the Regulations, I hereby confirm that I shall permit general access to my dissertation at the discretion of the University Librarian. I agree that copies of my dissertation may be made for Libraries and research workers on the understanding that no publication in any form is made of the contents without my permission.

Notes for Candidates: by submitting your dissertation, you agree with the following:

- 1 Where the examiners consider the dissertation to be of distinction standard, one copy may be deposited in the University Library and/or uploaded into Moodle as an example of good dissertation for future students.
- 2 If a copy is sent to the Library, it becomes the property of the University Library. The copyright in its contents remains with the candidate. A duplicated sheet is pasted into the front of every thesis or dissertation deposited in the Library. The wording on the sheet is:

"I undertake not to use any matter contained in this thesis for publication in any form without the prior knowledge of the author."

Every reader of the dissertation must sign and date this sheet.

- 3 The University has the right to publish the title of the dissertation and the abstract and to authorise others to do so.

.....
SIGNATURE

.....
DATE

.....
FULL NAME

(Please print, underlining surname, in order to assist the cataloguing of theses deposited in the Library.)

CERTIFICATE ON SUBMISSION OF DISSERTATION

I certify that:

- 1 I have read the University Degree Regulations under which this submission is made;
- 2 In so far as the dissertation involves any collaborative research, the extent of this collaboration has been clearly indicated; and that any material which has been previously presented and accepted for the award of an academic qualification at this University or elsewhere has also been clearly identified in the dissertation.

.....
SIGNATURE

.....
DATE

This form should be completed and included in the submission of your dissertation.

Abstract

This project investigates the application of Kolmogorov-Arnold Networks (KANs) and their variants in time series prediction and partial differential equation (PDE) learning for option pricing models. Kolmogorov-Arnold Networks, inspired by the Kolmogorov-Arnold Representation Theorem, utilize learnable parameterized B-splines for their representation functions. KANs have been applied to various domains, including data fitting, PDE solving, hyperspectral image classification, and time-series analysis and forecasting. Building on Temporal Convolutional Networks, I developed the novel Temporal Convolutional Kolmogorov-Arnold Network (TCKAN) by implementing causal dilated KAN convolution, inspired by the efficient-kan library. The project aims to investigate the performance of this novel variant and compare KAN layers with traditional MLP layers in terms of their overall effectiveness. Three architectures—TCKAN, KANTransformer, and standard KAN—are compared with their non-KAN counterparts, including Temporal Convolutional Networks (TCNs), Transformers, and Multi-Layer Perceptrons (MLPs), for time series prediction using AAPL stock and ETT data. Additionally, KAN and MLP models are applied to PDE learning for option pricing, specifically focusing on the Black-Scholes model using Physics-Informed Neural Networks (PINNs). This project aims to evaluate the effectiveness of KANs and derived architectures in time series forecasting and in KAN versus MLP performance for PINN-based option price modelling using the Black-Scholes partial differential equation.

Contents

1	Introduction	3
2	Background and Literature Review	5
2.1	Multi-Layer Perceptrons	5
2.1.1	Overview and Architecture of MLPs	5
2.1.2	Training MLPs	6
2.1.3	Backpropagation	7
2.1.4	Universal Approximation Theorem	8
2.1.5	Applications	9
2.2	Kolmogorov-Arnold Networks (KAN)	9
2.2.1	Kolmogorov-Arnold Representation Theorem	9
2.2.2	Kolmogorov-Arnold Network	10
2.2.3	B-Splines in KANs	12
2.2.4	Comparison of pykan and efficient-kan	15
2.2.5	Convolutional KANs	17
2.2.6	Applications	18
2.3	Time-Series Prediction	18
2.3.1	Introduction	18
2.3.2	Models and Applications	19
2.4	Temporal Convolutional Networks (TCN)	19
2.4.1	Introduction	19
2.4.2	Applications	20
2.5	Transformers	21
2.6	Options	22
2.6.1	Introduction	22
2.6.2	Black-Scholes Model	23
2.7	Physics-Informed Neural Networks (PINNs)	24
2.7.1	Introduction	24
2.7.2	Applications	24
3	Methodology, Experimental Setup and Model Architectures	25
3.1	Methodology and Experimental Setup	25

3.2	Multilayer Perceptron (MLP)	26
3.3	Kolmogorov-Arnold Network (KAN)	26
3.4	Temporal Convolutional Networks (TCN)	26
3.5	TCKAN: Novel Hybrid KAN and TCN Model	27
3.6	Transformer and KANTransformer	28
3.7	PINNs for Black-Scholes PDE	29
4	Data Preparation	30
4.1	ETTM Data and AAPL Stock and AAPL Stock Options Data	30
4.2	Preprocessing	31
5	Results and Discussion	32
5.1	Performances of Models on Time-Series Prediction	32
5.1.1	Results	32
5.1.2	Analysis and Discussion	34
5.2	Option Pricing PINN Models Performances	35
5.2.1	Results, Analysis and Discussion	35
6	Conclusion	36
6.1	Summary and Contributions	36
6.2	Future Work	36
7	Appendices	38
7.1	Code Snippets	38

Chapter 1

Introduction

The objective of this work is to explore and evaluate the performance of Kolmogorov-Arnold Networks (KANs) and their variants in the context of time series prediction and partial differential equation (PDE) learning. Specifically, this study focuses on financial data from AAPL stock, energy consumption data from the ETT dataset across various forecast horizons, and AAPL stock option data for PDE learning. The analysis compares KANs and their variants against their non-KAN counterparts. Temporal Convolutional Networks (TCNs), Temporal Convolutional Kolmogorov-Arnold Networks (TCKANs), Transformers, KANTransformers, KANs and Multi-Layer Perceptrons (MLPs) are all compared.

Chapter 2 provides the necessary background and literature review, starting with an overview of Multi-Layer Perceptrons (MLPs), covering their architecture, training methods, backpropagation, the Universal Approximation Theorem, and their various applications. This section establishes a baseline understanding of traditional neural networks.

In Section 2.2, we delve into Kolmogorov-Arnold Networks (KANs), discussing the Kolmogorov-Arnold Representation Theorem, the architecture of KANs, and enhancements like B-Splines and convolutional layers within KANs. Comparisons between pykan and efficient-kan implementations are also provided, alongside the introduction of Convolutional KANs (TCKAN) and their applications.

Section 2.3 focuses on Time-Series Prediction, where relevant models and applications are introduced, forming the basis for the comparative analysis of KAN-based and non-KAN-based models. This section sets the stage for the key comparisons in this study: TCN vs. TCKAN, Transformer vs. KANTransformer, and MLP vs. KAN, all applied to AAPL stock and ETT energy data.

In Section 2.4, Temporal Convolutional Networks (TCNs) are explored, particularly their architecture and effectiveness in time series forecasting. This provides a direct comparison point with TCKANs.

Section 2.5 discusses Transformers, another leading architecture in sequence modelling, and compares their performance with that of KANTransformers in the time series prediction tasks.

Options and the Black-Scholes model are briefly introduced in Section 2.6 to contextualize the financial aspects of the study. This section is particularly relevant to the application of PINNs for learning PDEs from AAPL stock option data.

Section 2.7 introduces Physics-Informed Neural Networks (PINNs) and their role in solving PDEs, such as those encountered in financial modelling. Here, MLP and KAN models are applied to PINN-based learning of PDEs, with a comparative analysis of their performance.

Chapter 3 will detail the specific models used, explaining the architecture and configurations for both KAN and non-KAN variants. Chapter 4 covers the data preparation processes, and Chapter 5 presents the results of the experiments, providing insights into the effectiveness of KANs and their variants in time series prediction and PDE learning. This exploration aims to determine the viability of KANs as robust models for both time series forecasting and PDE-based financial modelling, contributing to the broader understanding of neural network applications in these domains.

Chapter 2

Background and Literature Review

2.1 Multi-Layer Perceptrons

2.1.1 Overview and Architecture of MLPs

Multi-Layer Perceptrons (MLPs) [25], often referred to as feed-forward neural networks, and whose layers are known as dense layers or linear layers, in tensorflow and pytorch respectively, are one of the oldest and most foundational architectures within machine learning and artificial intelligence [15,25]. Comprising multiple layers of neurons, where each neuron in one layer is connected to every neuron in the next, MLPs transform input data through a series of linear transformations and non-linear activations. MLPs are traditionally used in supervised learning, where they act as universal function approximators. Using a dataset, they are trained by updating the weights or connections to minimise a cost function, which quantifies the divergence between the networks output and desired output.

An artificial neuron or perceptron is a crude mathematical approximation of a biological neuron. The neurons output is determined by a weighted sum of its inputs plus a bias or threshold term and its activation function. The i th neuron in an arbitrary layer of a MLP takes as inputs all the neurons from the previous layer.

$$a_i = \sigma\left(\sum_j w_{ij}x_j\right) \quad (2.1)$$

Each layer may be treated as a vector, where each element is the output of a single neuron in that layer. And as a single neuron is connected to all neurons

in the next layer, the connections or weights are represented as a matrix. For an arbitrary layer l the forward pass is as follows, with $\mathbf{a}^{(0)} = \mathbf{x}$.

$$\mathbf{a}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \quad (2.2)$$

Where activation output vector from the previous layer $\mathbf{a}^{(l-1)}$ is linearly transformed by weight matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$, giving intermediate pre-activation vector $\mathbf{z}^{(l)}$. A non-linear activation function σ is then applied element wise to $\mathbf{z}^{(l)}$ giving $\mathbf{a}^{(l)} \in \mathbb{R}^{n_l}$.

An MLP then contains L layers not including the input with $l \in [1, L] \cap \mathbb{Z}$, $L - 1$ hidden layers, and $L + 1$ layers including input layer. The input $\mathbf{x} = \mathbf{a}^{(0)} \in \mathbb{R}^{n_0}$ is passed through the L layers iteratively giving the output vector $\mathbf{a}^{(L)}$.

Hence, MLPs can be expressed as a series of compositions of weight matrices and element-wise non-linearities. The weight matrices are zero-indexed in this expression.

$$\text{MLP}(\mathbf{x}) = (\mathbf{W}^{(L-1)} \circ \sigma \circ \mathbf{W}^{(L-2)} \circ \dots \circ \mathbf{W}^{(1)} \circ \sigma \circ \mathbf{W}^{(0)})\mathbf{x} \quad (2.3)$$

2.1.2 Training MLPs

For a supervised learning task, we use a labelled dataset to train the MLP [25], $D = \{(\mathbf{X}_i, \mathbf{Y}_i) \mid \mathbf{X}_i = \mathbf{x} \in \mathbf{X}, \mathbf{Y}_i = \mathbf{y} \in \mathbf{Y}, i = 1, \dots, N\}$, where N is the size of the dataset and \mathbf{x} and \mathbf{y} are the i th input and label of the dataset. This training dataset is often derived from larger dataset, where testing and validation sets are also taken for evaluation.

The loss or cost function for a single data point i is denoted as $C_i(\mathbf{a}^{(L)}, \mathbf{y})$. The total cost over all the data points is $C(\mathbf{a}^{(L)}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^N C_i(\mathbf{a}^{(L)}, \mathbf{y})$. The objective then for training, is to minimise the cost function with respect to the parameters of the network.

$$\min_{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}, \forall l} C(\mathbf{a}^{(L)}, \mathbf{y}) \quad (2.4)$$

As solving for all parameter derivatives equal to zero is impractical due to high dimensionality and non-convex nature of the loss function, gradient-based optimization methods are used instead to iteratively minimize the loss.

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial C}{\partial \mathbf{W}^{(l)}} \quad (2.5)$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial C}{\partial \mathbf{b}^{(l)}} \quad (2.6)$$

This is done for all layers, with derivatives taken according to denominator layout, and with C being either the cost for a single data point (stochastic gradient descent), an average cost over a small subset of the whole dataset (mini-batch gradient descent) or over the whole dataset.

2.1.3 Backpropagation

Calculating the gradient for parameters in the final layer is simple, but not so for previous layers [25]. In order to compute parameter derivatives in layers $L - 1$ and lower, it is necessary, as the derivative is a linear operator, to sum the derivatives over all paths through neurons of outer layers. The backpropagation algorithm [25] takes this into account and efficiently computes the gradient of the loss with respect to the parameters, by iteratively traversing backwards through the layers of the network, backpropagating error. This is also known as the backward pass. The derivation is as follows, first using indices and then vectors and matrices.

For the final layer ($l = L$):

$$\frac{\partial C}{\partial W_{ij}^{(L)}} = \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial W_{ij}^{(L)}} \quad (2.7)$$

For the penultimate layer ($l = L - 1$) using j and k as indexes to the $L - 1$ and $L - 2$ neuron layers:

$$\frac{\partial C}{\partial W_{ij}^{(L-1)}} = \left(\sum_i \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}} \right) \frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial z_j^{(L-1)}}{\partial W_{jk}^{(L-1)}} \quad (2.8)$$

Now for a general layer l , where k , i and j are the indexes for the next, current and previous layers respectively. We now generalise the summation term in the parentheses above ($l = L - 1$) for any $l \in [1, L - 1] \cap \mathbb{Z}$:

$$\frac{\partial C}{\partial a_i^{(l)}} = \sum_k \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}} = \sum_k \frac{\partial C}{\partial a_k^{(l+1)}} \sigma'(z_k^{(l+1)}) W_{ki}^{(l)} \quad (2.9)$$

And for any l :

$$\frac{\partial C}{\partial a_i^{(l)}} = \begin{cases} \sum_k \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}} & \text{if } l < L \\ \frac{\partial C}{\partial a_i^{(l)}} & \text{if } l = L \end{cases} \quad (2.10)$$

We now define δ :

$$\delta_i^{(l)} = \frac{\partial C}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \quad (2.11)$$

For any l we have the gradient for the weights and biases:

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l)} \quad \frac{\partial C}{\partial b_i^{(l)}} = \delta_i^{(l)} \quad (2.12)$$

From these equations the canonical backpropagation algorithm using matrices and vectors follows. The delta vectors are defined for the last layer and all previous layers, where \odot is the Hadamard product.

$$\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}^{(L)}} C \odot \sigma'(\mathbf{z}^{(L)}) \quad (2.13)$$

$$\boldsymbol{\delta}^{(l)} = ((\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)}) \quad (2.14)$$

The delta vector updates iteratively as you traverse backwards through the layers, using the most proximate outer layer's delta vector and weight matrix, this defines a recursive algorithm. Using the delta vector of the current layer l , the gradients for the weights and biases expressed in linear algebra operations are:

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \mathbf{a}^{(l)T} \quad \frac{\partial C}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \quad (2.15)$$

Using gradient descent [8] according to (2.5) and (2.6) will iteratively minimise the cost function with respect to the parameters (2.4), effectively approximating the function whose range and domain is inputs and labels of the training dataset pair D , and this causes the function to generalise to samples outside of that set as the MLP function is continuous.

2.1.4 Universal Approximation Theorem

The Universal Approximation Theorem [9] states that an MLP with at least one hidden layer and a non-linear activation function can approximate any continuous function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ on a closed and bounded subset of \mathbb{R}^n to any desired degree of accuracy, given a sufficient number of neurons. Specifically, for any continuous function $f(x)$ and any $\epsilon > 0$, there exists a neural network $\hat{f}(x)$ such that:

$$|f(x) - \hat{f}(x)| < \epsilon \quad \text{for all } x \text{ in the subset.} \quad (2.16)$$

Where for an MLP with one hidden layer, the function $\hat{f}(x)$ can be expressed as:

$$\hat{f}(x) = \sum_{j=1}^m v_j \sigma \left(\sum_{i=1}^n w_{ij} x_i + b_j \right) + c \quad (2.17)$$

where x is the input vector, w_{ij} are the hidden layer weights, b_j and c are biases, σ is the activation function, and v_j are the output layer weights.

2.1.5 Applications

MLPs [25] are fundamental to deep learning and serve as the bedrock for more complex architectures like Convolutional Neural Networks (CNNs) and Transformers. They are widely used in tasks such as classification, regression, and anomaly detection, providing the basic framework for feature extraction and pattern recognition [15] in diverse applications.

2.2 Kolmogorov-Arnold Networks (KAN)

2.2.1 Kolmogorov-Arnold Representation Theorem

The Kolmogorov-Arnold Representation Theorem [1, 13] is a foundational result in the theory of multivariate functions, which asserts that any continuous multivariate function can be decomposed into a finite sum of continuous functions of a single variable.

Let $f: [0, 1]^n \rightarrow \mathbb{R}$ be a continuous function. The Kolmogorov-Arnold Representation Theorem states that there exist continuous functions $\phi_i: \mathbb{R} \rightarrow \mathbb{R}$ and continuous functions $\psi_{ij}: [0, 1] \rightarrow \mathbb{R}$ for $i = 1, 2, \dots, 2n + 1$ and $j = 1, 2, \dots, n$ such that:

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{2n+1} \phi_i \left(\sum_{j=1}^n \psi_{ij}(x_j) \right) \quad (2.18)$$

This theorem shows that any multivariate function f can be expressed as a finite sum of univariate functions applied to linear combinations of the input variables.

2.2.2 Kolmogorov-Arnold Network

Noticing the double sum in (2.18) one can express the Kolmogorov-Arnold representation theorem as a matrix multiplication of the input with two function matrices [19]: a row vector $\Phi \in (\mathbb{R}^\mathbb{R})^{1 \times 2n+1}$ and a matrix $\Psi \in ([0, 1]^\mathbb{R})^{2n+1 \times n}$ whose elements are univariate functions (2.18). The function $f : [0, 1]^n \rightarrow \mathbb{R}$ can be represented as follows.

$$f(\mathbf{x}) = [\phi_1(\cdot) \ \cdots \ \phi_{2n+1}(\cdot)] \begin{bmatrix} \psi_{1,1}(\cdot) & \cdots & \psi_{1,n}(\cdot) \\ \vdots & \ddots & \vdots \\ \psi_{2n+1,1}(\cdot) & \cdots & \psi_{2n+1,n}(\cdot) \end{bmatrix} \mathbf{x} \quad (2.19)$$

If Φ contained multiple rows f would become a vector function \mathbf{f} . Although this imposes a constraint that each output f_i shares the inner representation function matrix Ψ which potentially limits the ability of each f_i to fully represent itself according the Kolmogorov-Arnold representation theorem, it allows one to draw an analogy between operations of a weight matrix and a non-linear activation function in MLPs (2.3) to a function matrix.

This leads to the definition of a KAN layer [19] (2.20) for an arbitrary layer $l \in [0, L - 1] \cap \mathbb{Z}$ in a network of L layers.

$$\mathbf{z}^{(l+1)} = \underbrace{\begin{bmatrix} \phi_{1,1}^{(l)}(\cdot) & \phi_{1,2}^{(l)}(\cdot) & \cdots & \phi_{1,n_l}^{(l)}(\cdot) \\ \phi_{2,1}^{(l)}(\cdot) & \phi_{2,2}^{(l)}(\cdot) & \cdots & \phi_{2,n_l}^{(l)}(\cdot) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{n_{l+1},1}^{(l)}(\cdot) & \phi_{n_{l+1},2}^{(l)}(\cdot) & \cdots & \phi_{n_{l+1},n_l}^{(l)}(\cdot) \end{bmatrix}}_{\Phi^{(l)}} \mathbf{z}^{(l)} \quad (2.20)$$

Where $\mathbf{z}^{(l)}$ and $\mathbf{z}^{(l+1)}$ are pre and post-activation values of the KAN layer $\Phi^{(l)}$ respectively with $\mathbf{z}^{(0)} = \mathbf{x}$, and where the dimension of $\mathbf{z}^{(l)}$ is denoted as n_l .

A general Kolmogorov-Arnold Network [19] is a composition of these L layers. It's shape is represented by an integer array: $[n_0, n_1, \dots, n_L]$. The output of a KAN (2.21) given an input vector $\mathbf{x} \in \mathbb{R}^{n_0}$ is as follows:

$$\text{KAN}(\mathbf{x}) = (\Phi^{(L-1)} \circ \Phi^{(L-2)} \circ \dots \circ \Phi^{(1)} \circ \Phi^{(0)})\mathbf{x} \quad (2.21)$$

In the canonical formulation of the Kolmogorov-Arnold Network each function

$\phi(x)$ (2.22) is expressed as a linear combination of a SiLU basis function $b(x)$ (2.23) and a spline function $spline(x)$ (2.24) that is a parametrised B-spline [7].

$$\phi(x) = w_b b(x) + w_s spline(x) \quad (2.22)$$

$$b(x) = SiLU(x) = x\sigma(x) = \frac{x}{1 - e^{-x}} \quad (2.23)$$

$$spline(x) = \sum_i c_i B_i(x) \quad (2.24)$$

Where w_b , w_s and each c_i are trainable parameters, and each $B_i(x)$ is a B-spline basis, all of a certain order, whose support is on the intervals defined by the B-spline's grid or knot vector.

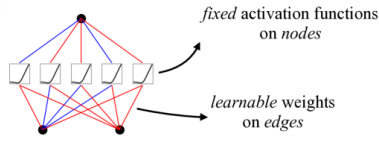
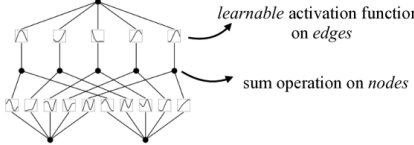
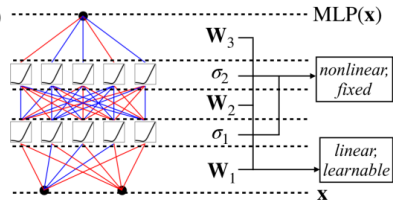
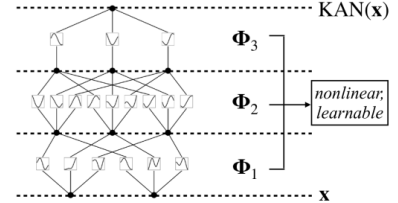
Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(e)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a)  fixed activation functions on nodes learnable weights on edges	(b)  learnable activation functions on edges sum operation on nodes
Formula (Deep)	$MLP(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$KAN(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	(c)  MLP(x) \mathbf{W}_3 σ_2 \mathbf{W}_2 σ_1 \mathbf{W}_1 \mathbf{x} nonlinear, fixed linear, learnable	(d)  KAN(x) Φ_3 Φ_2 Φ_1 \mathbf{x} nonlinear, learnable

Figure 2.1: MLPs vs KANs

A Kolmogorov-Arnold Network is hence a series of applications of function matrices to an input where each element of a function matrix is a unique learnable activation function. The Kolmogorov-Arnold representation theorem showed that only true multivariate function is that of addition as every other function can be written using univariate functions and summation. The parameters w_s and w_b , therefore, determine the KANs share various similarities and differences to MLPs, as shown in Figure 2.1.

2.2.3 B-Splines in KANs

B-splines [7], or basis splines, are piecewise polynomial functions of a certain order defined over a specified set of intervals, known as a knot vector or grid. The knot vector is a non-decreasing monotonic sequence $\mathbf{T} = [t_0, t_1, \dots, t_m]$, that defines the parametric intervals of the B-spline and therefore (along with the order) the support of each of the B-spline basis functions, dictating the placement and smoothness of each of the spline's polynomial segments. A B-spline of order k , determines the degree $p = k - 1$ of all of the polynomial bases. The support of a function is the interval of its domain where the range is non-zero and is shown for a B-spline basis below.

$$B_{i,k}(t) = \begin{cases} \text{non-zero} & \text{if } t_i \leq t < t_{i+k}, \\ 0 & \text{otherwise.} \end{cases} \quad (2.25)$$

Where a B-spline basis of order k can be generated by the Cox-de Boor [7] recursion formula.

$$B_{i,1}(t) = \begin{cases} 1 & \text{if } t_i \leq t < t_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \quad (2.26)$$

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} B_{i+1,k-1} \quad (2.27)$$

A B-spline curve \mathbf{C} is a linear combination of control points $\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_n$, where $\mathbf{P}_i \in \mathbb{R}^D$ and B-spline bases $B_{0,k}, B_{1,k}, \dots, B_{n,k}$, where $B_{i,k} \in \mathbb{R}$.

$$\mathbf{C}(t) = \sum_{i=0}^n \mathbf{P}_i B_{i,k}(t) \quad (2.28)$$

This defines a D -dimensional curve where the control points locally shape the curve's geometry pulling the curve towards them as they effectively weight the B-spline basis polynomials. The curve is typically contained within the convex hull of the control points except in certain configurations.

The B-spline must satisfy $m = n + p + 1 = n + k$ and the order k can be no more than the number of control points $n + 1$. If each consecutive knot is separated by the same distance h , such that $h = t_{i+1} - t_i$, the knot vector and b-spline are known as uniform. In a uniform setup, B-splines are continuously differentiable up to the derivative of degree $p - 1$.

In any point in the domain defined by the knot vector there are at most $k = p + 1$ non-zero B-spline basis functions. The internal knots of a B-spline are knots $[t_p, \dots, t_{m-p}]$, this is the region (open interval) of full support, where there are k non-zero basis functions defined over the region (if there are p knots repeated either side of internal knots, known as the end or boundary knots). Hence in an open (not a closed curve) uniform b-spline, the domain is often defined to be only this region of internal knots, as it has full support, the KAN paper uses this definition. In certain cases the boundary knots either side may be repetitions of the same values of the first and last interval knots respectively, such that there are k repeated values either side, in order to clamp the curve, such that it ends on the first and last control points respectively. The KAN paper does not use this and keeps uniform spacing throughout the knot vector.

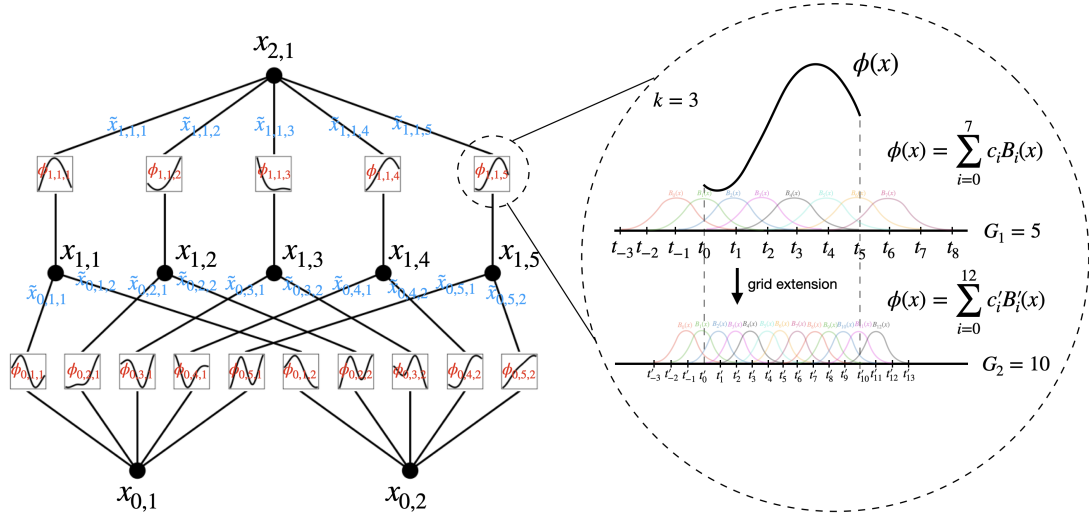


Figure 2.2: B-Splines in KANs [19]

In KANs, the control points lie in \mathbb{R} , and so the range of C (2.28) is in \mathbb{R} . This leads to the definition in (2.24), where all $B_i(x)$ are of some spline order k not shown with input x . As the control points are learnable parameters, you get a learnable piecewise polynomial activation function.

In the KAN paper and corresponding `pykan` [11] code and derived source codes like `efficient-kan` [4], the definitions are slightly different. The degree of the b-spline polynomials is called the spline order in the paper and are denoted k or `spline_order` in the paper and code respectively. The length of the internal knot

vector is called the grid, denoted as G and `grid_size`. The `grid_range` is the domain and is by default $[-1, 1]$. The uniform spacing h is calculated by dividing the `grid_range` interval by G .

The actual knot vector is called `grid` and is a PyTorch tensor with `grid_size + 2*spline_order + 1` elements, separated by increments of h around 0, which is the open uniform spline definition. The number of control points and hence basis functions is `grid_size + spline_order`.

The KAN paper [19] has its own approximation theorem, which uses results from B-spline theory, called KAT:

Suppose f can be represented as $f(\mathbf{x}) = (\Phi_{L-1} \circ \Phi_{L-2} \circ \dots \circ \Phi_1 \circ \Phi_0)\mathbf{x}$. Where all ϕ functions are $(k+1)$ -times continuously differentiable. Then there exists a constant C depending on f and its representation, such that we have the following approximation bound in terms of the grid size G :

There exist k -th order B-spline functions $\Phi_{l,i,j}^G$ such that for any $0 \leq m \leq k$, we have the bound:

$$\|f - (\Phi_{L-1}^G \circ \Phi_{L-2}^G \circ \dots \circ \Phi_1^G \circ \Phi_0^G)\mathbf{x}\|_{C^m} \leq CG^{-k-1+m}. \quad (2.29)$$

Where the C^m -norm measures the largest magnitude of derivatives up to order m inclusive:

$$\|g\|_{C^m} = \max_{|\beta| \leq m} \sup_{\mathbf{x} \in [0,1]^n} |D^\beta g(\mathbf{x})|. \quad (2.30)$$

This essentially states in simple terms, that the KAN approximation error with B-splines is bounded, and the approximation gets better as grid size and spline order increase, as the maximum of the error (L_∞ norm = C^0 -norm) decreases as G increases. However the error still depends on the representation.

In the proof of this theorem, they define the residue of a certain layer R_l as the difference between a KAN representation composed of B-spline approximated KAN layers from $L-1$ to $l+1$ and another from $L-1$ to l . They note that $\|R_l\|_{C^m} = CG^{-k+1+m}$. You can then express the whole approximation error as the sum of these residues.

$$f - (\Phi_{L-1}^G \circ \Phi_{L-2}^G \circ \dots \circ \Phi_1^G \circ \Phi_0^G)\mathbf{x} = R_{L-1} + R_{L-2} + \dots + R_1 + R_0 \quad (2.31)$$

Which means, according to the KAN paper [19], that the KAN can approximate functions with a residue rate independent of the dimension, as it depends on number of layers instead, which beats the curse of dimensionality in a sense. However the constant C is dependent on representation, which depends on dimensionality.

2.2.4 Comparison of `pykan` and `efficient-kan`

The GitHub repo `efficient-kan` is a fork of the original KAN paper repo `pykan`, that has improved performance. It does this by reformulating the tensor operations in such a way without having to expand out all intermediary variables to perform different activation functions, increasing speed.

For the comparison, the relevant sections are the classes `KANLayer` in `pykan`, and `KANLinear` in `efficient-kan`, which define a single KAN layer Φ . The input and output parameters of the layer are given names `in_dim` and `out_dim` respectively for `pykan`, and `in_features` and `out_features` for `efficient-kan`, they are equivalent and will be used interchangeably with the words themselves.

In `pykan`, the `forward` method of `KANLayer` defines the forward pass. In this method, the input `x` is a tensor of shape `(batch, in_dim)`, it is passed element-wise through function `self.base_fun` which is $b(x)$ in (2.23) to give `base`.

`x` is also passed through a function `coef2curve`, which also takes in the `grid` and the order `k`. `grid` is of shape `(input dimension, G+2k)`. This function passes `x` through an intermediate function `B.batch`. `B.batch` recursively computes the $G+k$ b-spline basis (2.25) using the Cox-de Boor recursion formula (2.27), for each input dimension resulting in an output returned as `b_splines` of shape `(batch, in_dim, G+k)`. `b_splines` is then `.einsummed` with a coefficient parameter tensor `coeff` of shape `(in_dim, out_dim, G+k)`, representing the b-spline coefficients `c` in (2.24) for each element of Φ . The operation is `.einsum("ijk,jlk->ijl", b_splines, coeff)`, meaning for each `batch`, `in_dim` and `out_dim` dimension, it performs a weighted sum over dimension $G+k$ between the b-spline bases functions vector of that `in_dim` and the parameters `c` of that `in_dim`. This gives an output returned from `coef2curve` as `y` of shape `(batch, in_dim, out_dim)`, and means each `out_dim` row shares the same `(in_dim, G+2k)` b-spline bases functions vector, which is like function matrix multiplication according to (2.20), before they are summed with the coefficient tensor `coeff`'s $G+k$ dimension.

The `scale_base` parameter representing all w_b of Φ (2.22) of shape `(in_dim, out_dim)` is multiplied element-wise by a `base` which is broadcasted over the `out_dim` dimension, effectively applying each row to the input vector element-wise, which follows (2.20), and then multiplying the entire matrix element-wise by `scale_base`. The `scale_spline` parameter representing all w_s of Φ (2.22) of shape `(in_dim, out_dim)` is multiplied element-wise by a `y` over the entire `batch` dimension. These two values are summed to give you an intermediary of shape `(batch, in_dim, out_dim)` which for each batch, each element represents (2.22).

Finally the intermediary is summed over the `input` dimension, giving you the output vector according to (2.20).

In `efficient-kan` this operation is done more efficiently. In class `KANLinear`, the method `b_splines` is equivalent to `B_batch` in `pykan`, `self.base_fun` is equivalent to `self.base_activation`, `base_weight` is `scale_base` but with shape `(out_dim, in_dim)`, `spline_weight` is `coeff` but with shape `(out_dim, in_dim, G+k)`. `self.scaled_spline_weight` is `spline_scalar (scale_spline)` element wise multiplied by `spline_weight`. In the forward method, the input `x` is of shape `(batch, in_dim)` like before, however the subsequent operations are different. It uses `F.linear`, which is highly optimised, to multiply `self.base_activation(x)` by `self.base_weight` to give `base_output`, which is a very similar operation to `pykan`.

The main difference is in the computation of the *spline*(x) functions `spline_output`. `b_splines(x)` is computed giving shape `(batch, in_dim, G+k)`, but it is `.viewed` in a way that flattens the shape to `(batch, in_dim * G+k)`. This `.view` flattening operation is also done to `self.scaled_spline_weight` giving shape `(out_dim, in_dim * G+k)`. These two values are matrix multiplied using `F.linear`, giving `spline_output` of shape `(batch, out_dim)`. `base_output` and `spline_output` are added together to give the batched final output vector (2.20). The flattening and `F.linear` operations, are an equivalent operation to the one in `pykan`, but much faster, as it uses `F.linear` and does not expand out the tensor, but does the multiplication in one go.

Remember that for any layer output $z_i = \sum_j \phi_{i,j}(x_j) = \sum_j \sum_k c_{i,j,k} B_k(x_j)$ (k is an index not spline order), and ϕ represents the $w_s \text{spline}$ part of the function as the bias function $w_b b$ is added later. So we can express the `efficient-kan` operation to show that is equivalent to (2.20) and (2.24) as follows.

$$\beta = [B_1(x_1) \quad \dots \quad B_k(x_1) \quad \dots \quad B_1(x_n) \quad \dots \quad B_k(x_n)]^T \quad (2.32)$$

A single row of flattened `spline_weight` looks like this:

$$\mathbf{C}_{i,\cdot} = [c_{i,1,1} \quad \dots \quad c_{i,1,(G+k)} \quad \dots \quad c_{i,\text{in_dim},1} \quad \dots \quad c_{i,\text{in_dim},(G+k)}] \quad (2.33)$$

From this we can see:

$$\mathbf{C}_{i,\cdot} \beta = \sum_j \sum_k c_{i,j,k} B_k(x_j) = \sum_j \phi_{i,j}(x_j) = z_i \quad (2.34)$$

The operation is equivalent to (2.20) and uses the pytorch optimised `F.linear` function.

2.2.5 Convolutional KANs

Traditional 2D convolution [15], which is the most common operation in Convolutional Neural Networks (CNNs), is defined as follows, for a single filter of shape $M \times N \times C_{in}$.

$$\mathbf{Y}_{i,j} = (\mathbf{X} * \mathbf{K})_{i,j} = \sum_{c=1}^{C_{in}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{X}_{c,i+m,j+n} \cdot \mathbf{K}_{c,m,n} \quad (2.35)$$

Each channel of the input is a 2D array. An example of the first channel \mathbf{X}_1 :

$$\mathbf{X}_1 = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix} \quad (2.36)$$

The kernel is a smaller array that is slid across the input array. And the output is an array resulting from the dot product of the kernel (filter) and the input at each position. An example of the first channel of the kernel:

$$\mathbf{K}_1 = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix} \quad (2.37)$$

Taking inspiration from the KAN layer (2.20), we can replace the weights in the kernel with functions to be applied to some input. KAN Convolution [5] can then be defined as follows:

$$\mathbf{Y}_{i,j} = (\mathbf{X} * \Phi^\kappa)_{i,j} = \sum_{c=1}^{C_{in}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \phi_{c,m,n}(\mathbf{X}_{c,i+m,j+n}) \quad (2.38)$$

With an example of the first channel of the KAN Convolutional kernel as:

$$\Phi_1^\kappa = \begin{pmatrix} \phi_{1,1}(\cdot) & \phi_{1,2}(\cdot) \\ \phi_{1,2}(\cdot) & \phi_{2,2}(\cdot) \end{pmatrix} \quad (2.39)$$

1D, 3D and higher dimensional KAN convolutions can be defined analogously to traditional convolution straightforwardly by changing spatial dimensions of input and kernel.

2.2.6 Applications

KANs are a very recent architecture, only being released in the last year [19] applied in domains such as data fitting, solving partial differential equations [18], hyperspectral image classification [10], and time series analysis [32]. With its distinct architecture, KANs show potential promise in outperforming conventional deep learning models in these fields.

2.3 Time-Series Prediction

2.3.1 Introduction

In time series prediction [22], the aim is to predict future values based on historical data. We have a time-series dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$. Where each $\mathbf{x}_i \in \mathbb{R}^D$. In the most general case of multi-feature lookahead to multi-feature horizon forecasting the objective is predict (horizon window) h future values $\{\mathbf{x}_{t+1}, \mathbf{x}_{t+2}, \dots, \mathbf{x}_{t+h}\}$ using the most recent (lookback window) p values $\{\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-p+1}\}$.

This is done by modelling a function f to forecast a sequence of h future values $\{\hat{\mathbf{x}}_{t+1}, \hat{\mathbf{x}}_{t+2}, \dots, \hat{\mathbf{x}}_{t+h}\}$ using the most recent p observations:

$$\hat{\mathbf{x}}_{t+1}, \hat{\mathbf{x}}_{t+2}, \dots, \hat{\mathbf{x}}_{t+h} = f(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-p+1}; \theta) \quad (2.40)$$

Such that the predictions most closely resemble the future values as possible. The dataset is often restructured into $\mathbf{X} = \{\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_m\}$ where $\mathbf{X}_t = \{\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-p+1}\}$, so that you can train the model using batches of \mathbf{X}_t , so that you can train the function $f(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-p+1}; \theta) = f(\mathbf{X}_t; \theta)$ in parallel without breaking causality, where each batch is $\{\mathbf{X}_i, \dots, \mathbf{X}_{i+\text{batch_size}-1}\} \subseteq \mathbf{X}$. Analogously $\mathbf{Y} = \{\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_f\}$ where $\mathbf{Y}_t = \{\mathbf{x}_{t+1}, \mathbf{x}_{t+2}, \dots, \mathbf{x}_{t+h}\}$.

The training objective is the following minimisation problem:

$$\min_{\theta} C(f(\mathbf{X}), \mathbf{Y}) \quad (2.41)$$

Which means minimising the cost function C , which is often something like mean squared error, by finding the best set of parameters θ .

Multi-feature lookahead to multi-feature horizon forecasting, often requires a large

complex model for accurate predictions. A simpler forecasting setup is multi-feature lookback to single-feature horizon forecasting, where the only difference is the forecast vector for a time t , $\mathbf{Y}_t = \{x_{t+1}, x_{t+2}, \dots, x_{t+h}\}$, where $x_i \in \mathbb{R}$, is a single chosen feature instead of all of the features where previously $\mathbf{x}_i \in \mathbb{R}^D$.

2.3.2 Models and Applications

Time series models [21] are widely used in forecasting across finance, weather prediction, sales, and supply chain management. Common models include ARIMA for univariate series with autocorrelations. ARIMA uses differencing to remove non-stationarity. LSTM networks, a type of RNN, are effective for complex, non-linear data, especially in capturing long-term dependencies. Transformers have also been applied to time-series showing success in very long sequence tasks, with models like the Informer. State space models like MAMBA have also shown success. N-Beats is another deep learning based model that shows great performance. Temporal Convolutional Networks (TCNs) [14, 17] offer stability and performance for sequences with long-range dependencies. Prophet, by Facebook, is used in business applications for handling seasonality, outliers, and missing data.

2.4 Temporal Convolutional Networks (TCN)

2.4.1 Introduction

Temporal Convolutional Networks (TCNs) [14, 17, 28] are specialized neural networks for sequential data, offering a fully convolutional architecture that processes sequences in parallel. It uses causal convolutions, ensuring only past inputs are considered, and dilated convolutions, which extend the receptive field at each layer, capturing long-range dependencies efficiently. TCNs were introduced in WaveNet which is a model that generates waveforms.

Dilated 1-dimensional convolution for a single filter and a dilation factor d , using the time-series data setup defined in (2.40) is expressed as:

$$(\mathbf{O}_t)_i = (\mathbf{X}_t *_d \mathbf{K})_i = \sum_{c=1}^{C_{in}} \sum_{m=0}^{K-1} (\mathbf{X}_t)_{c,i+d*m} \cdot \mathbf{K}_{c,m} \quad (2.42)$$

It is made a causal convolution by left-padding the input data by $(K - 1) * d$, where K is kernel length over time-steps, so that the convolutions can stack in a way that tends to all past inputs and does not produce any future timesteps.

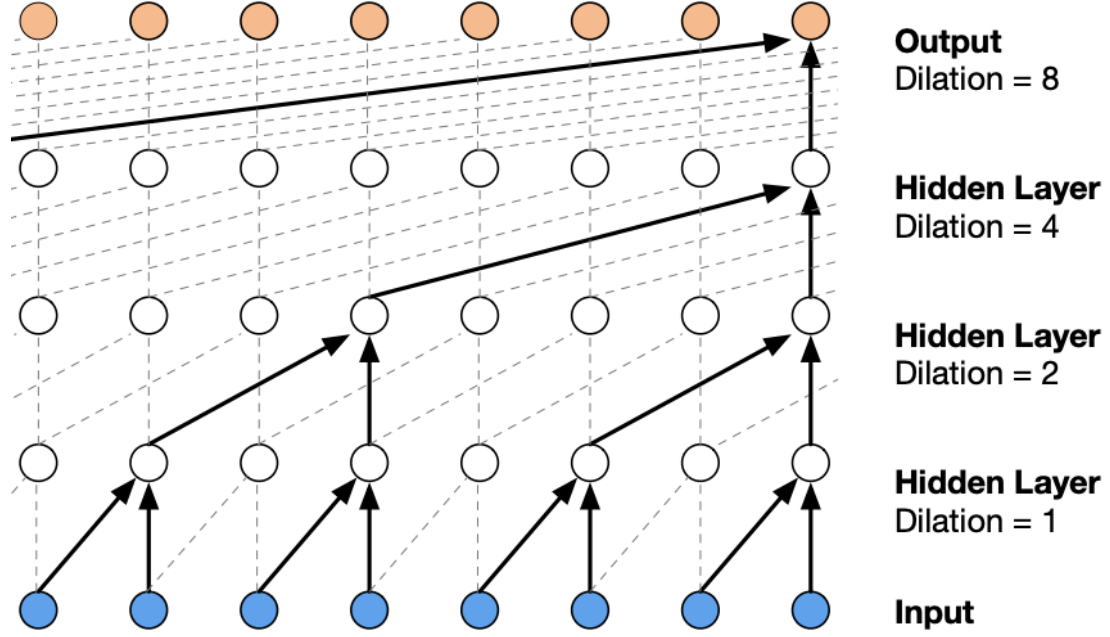


Figure 2.3: Dilated causal convolution in TCNs

Future time-steps are produced after this TCN encoding process.

A simple TCN encoder layer output (for one filter) is hence:

$$\mathbf{O}_t^{(l+1)} = \mathbf{O}_t^{(l)} *_{2^{l-1}} \mathbf{K} \quad (2.43)$$

Showing that the dilation of the causal convolution increases exponentially every layer according to 2^{l-1} , allowing it to incrementally capture longer range and more agglomerated dependencies and features, without a large model complexity.

2.4.2 Applications

Temporal Convolutional Networks (TCNs) have been effectively applied in various domains, including sequence modelling, time series forecasting, and NLP tasks. Notable applications include speech enhancement and separation as demonstrated by [14] and [2] in their respective papers on sequence modelling. TCNs have also been used in predictive maintenance for industrial equipment, and in financial time series forecasting [6]. Their ability to capture long-range dependencies with stability has made TCNs a robust alternative to RNNs and LSTMs in these

applications.

2.5 Transformers

Transformers are a type of neural network architecture introduced in the paper "Attention is All You Need" [29]. Unlike traditional models that rely on recurrent or convolutional layers, Transformers use multi-headed attention mechanisms to process input data in parallel, allowing them to capture long-range dependencies more efficiently. They are particularly effective in natural language processing tasks such as translation, summarization, and language modelling, powering models like BERT, GPT, and T5. Transformers have revolutionized NLP by enabling state-of-the-art performance on a wide range of tasks, thanks to their scalability and ability to handle large datasets.

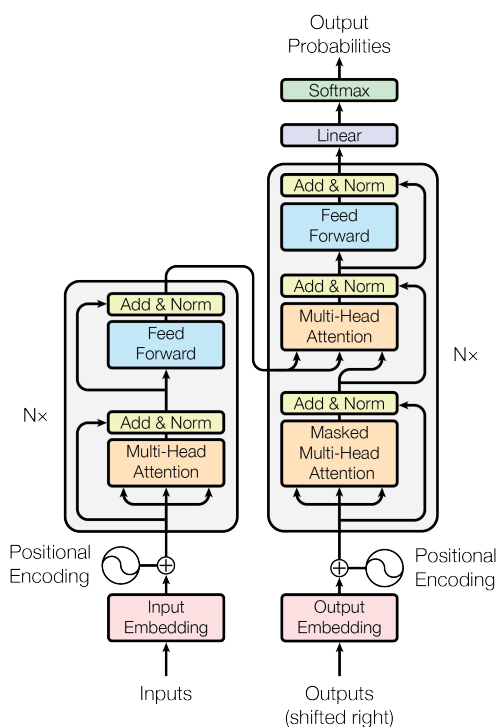


Figure 2.4: Transformer Encoder-Decoder Model

In Transformer encoders inputs are like those described in the section (2.40), although they are usually sequences of vector encoded words for NLP tasks. The inputs are embedded by an MLP which increases the size of the feature dimension to allow the handling of more complicated relationships. After embedding the

positionally encoded input is added to it, this is necessary as transformers treat the input as a bag of values, with no sequential ordering.

The power of transformers is in multi-headed dot-product self-attention, where for each head, the transformer first projects the input sequence into a sequence key query and value vectors and calculates the dot-product self attention. Dot-product attention computes how well a given query aligns with all keys in the sequence by taking their dot products. It then scales, applies softmax to get attention weights, and uses these weights to combine the corresponding value vectors, producing a context vector focused on relevant information.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \quad (2.44)$$

A single encoder layer is as follows:

$$Q = \mathbf{X}\mathbf{W}^Q \quad K = \mathbf{X}\mathbf{W}^K \quad V = \mathbf{X}\mathbf{W}^V \quad (2.45)$$

$$\text{head}_i = \text{Attention}(Q\mathbf{W}_i^Q, K\mathbf{W}_i^K, V\mathbf{W}_i^V) \quad (2.46)$$

$$O = \text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)\mathbf{W}^O \quad (2.47)$$

$$U = \text{LayerNorm}(\mathbf{X} + O) \quad (2.48)$$

$$Z' = \text{PFFN}(U) = \text{ReLU}(\mathbf{W}_1 U + \mathbf{1}\mathbf{b}_1^T)\mathbf{W}_2 + \mathbf{1}\mathbf{b}_2^T \quad Z = \text{LayerNorm}(U + Z') \quad (2.49)$$

Where the weight dimensions are: $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$, $W_i^K, W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, $W_i \in \mathbb{R}^{d_{\text{model}}}$.

2.6 Options

2.6.1 Introduction

An option in finance is a derivative contract giving the buyer the right, but not the obligation, to buy (call) or sell (put) an underlying asset at a predetermined price (strike price K) either before or at the expiration date (American option), or only at the expiration date (European option).

The payoff for a call option at expiration T is:

$$\text{Call Payoff} = \max(S_T - K, 0)$$

For a put option, the payoff is:

$$\text{Put Payoff} = \max(K - S_T, 0)$$

Where S_T is the asset price at expiration. The value of these options prior to expiration can be modeled using the Black-Scholes formula for European options, which is based on the asset's current price S_0 , strike price K , time to expiration $T - t$, risk-free rate r , and volatility σ .

2.6.2 Black-Scholes Model

The Black-Scholes [3] Partial Differential Equation (PDE) models the price of European options and is given by:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0 \quad (2.50)$$

where $V(S, t)$ is the option price as a function of the underlying asset price S and time to expiration t . Here, σ represents the volatility of the underlying asset, r is the risk-free interest rate, and $\frac{\partial V}{\partial t}$, $\frac{\partial V}{\partial S}$, and $\frac{\partial^2 V}{\partial S^2}$ are the partial derivatives of the option price with respect to time, and first and second order partial derivatives with respect to the underlying asset price, respectively. The equation must be solved with specific boundary conditions.

The boundary conditions for a European call option are:

$$V(S, 0) = \max(S - K, 0) \quad \text{and} \quad V(0, t) = 0 \quad (2.51)$$

For large S , the boundary condition is:

$$V(S, t) \approx S - Ke^{-r(T-t)} \quad (2.52)$$

The boundary conditions for a European put option are:

$$V(S, 0) = \max(K - S, 0) \quad \text{and} \quad V(0, t) = Ke^{-r(T-t)} \quad (2.53)$$

For large S , the boundary condition is:

$$V(S, t) \approx 0 \quad (2.54)$$

2.7 Physics-Informed Neural Networks (PINNs)

2.7.1 Introduction

Physics-Informed Neural Networks [24] (PINNs) solve partial differential equations (PDEs) by incorporating the PDE directly into the neural network's loss function. Given a general PDE of the form $\mathcal{L}(u) = 0$, where \mathcal{L} is some general partial differential operator and $u(x)$ is the solution, PINNs approximate $u(x)$ by a neural network $u_\theta(x)$ with parameters θ . The loss function is:

$$\text{Loss} = \lambda_1 \cdot \text{MSE}_{\text{data}} + \lambda_2 \cdot \text{MSE}_{\text{PDE}} + \lambda_3 \cdot \text{MSE}_{\text{BC}} \quad (2.55)$$

where MSE_{data} measures the error on observed data, and $\text{MSE}_{\text{PDE}} = \|\mathcal{L}(u_\theta(x))\|^2$ enforces the solution conforming to the PDE, and MSE_{BC} which is the sum of squares of boundary condition residuals, causing enforcement of the boundary conditions. PINNs use automatic differentiation to compute $\mathcal{L}(u_\theta(x))$, allowing the network to learn the solution by minimizing the combined loss. This approach effectively integrates data with physical laws, enabling the solution of complex PDEs across various domains.

2.7.2 Applications

Physics-Informed Neural Networks (PINNs) [24] solve PDEs by integrating physical laws [23] into neural networks. They are used in fluid dynamics, material stress analysis, and inverse problems like parameter estimation, providing accurate models even with limited data.

Chapter 3

Methodology, Experimental Setup and Model Architectures

3.1 Methodology and Experimental Setup

There are two main separate experiments that are conducted: one for time-series prediction, where all of the model architectures listed in the chapter are compared, and an option pricing model comparison, using PINNs for Black-Scholes PDE learning, comparing the KAN and MLP. For both experiments the deep learning python library `PyTorch` is used.

For the time-series prediction experiments, the following models were compared: MLP, KAN, TCN, TCKAN, Transformer and KANTransformer. Other models like state space models and recurrent models like LSTMs and RNNs were not considered due to time constraints in creating equivalent KAN variants and training. The objective is to determine and compare performance of the models, to see if Kolmogorov-Arnold layers with learnable B-spline functions, give better prediction metrics than their canonical counterpart models and compare all models. All training and evaluation of the models was done with a sequence length or look-back window of $p = 48$, a learning rate $\eta = 0.001$ for the Adam optimiser [12], a batch size of 32 and 100 epochs of training with early stopping. Training was done over the datasets: AAPL stock, ETTm1 and ETTm2. For each model and dataset the training and testing was done with the following set of forecast horizons $h = \{1, 12, 24, 48\}$ and such that they only predict a single feature of the dataset upto h , using all the features in the lookback window. This is the multi-feature lookback to single-feature horizon forecasting mentioned in Chapter 2.

For the PINN option pricing model experiments, the models under evaluation

are the KAN and MLP, with the same hyperparameters, a batch size of 32, a learning rate $\eta = 0.001$ for the Adam optimiser and 100 epochs of training with early stopping.

3.2 Multilayer Perceptron (MLP)

The architecture of the multilayer perceptron used was a flattened input layer of the number of dataset features f multiplied by the look back window p , 2 hidden layers of 256 and an output layer of length of the forecast horizon h , giving a layer dimension list of $[f * p, 256, 256, h]$.

3.3 Kolmogorov-Arnold Network (KAN)

For the KAN model, the `efficient-kan` implementation is used, the grid size is constant throughout, and all is setup according to the `efficient-kan` part of Subsection 2.2.4. The architecture of the KAN has the same dimensions in its architecture as the MLP for comparison purposes. Its dimension list is also $[f * p, 256, 256, h]$.

3.4 Temporal Convolutional Networks (TCN)

The architecture of the Temporal Convolutional network is a bit more involved than described in Section 2.4, as TCN blocks like those used in WaveNet and for sequence modelling and time-series tasks, often have alterations like repeated convolutions, masking and residual connections.

The particular architecture used is as follows:

A single TCN block or layer consists of two causal convolutions at a dilation of $d = 2^{l-1}$ where l is the current layer or block. The first dilated causal convolution takes input of shape `(in_channels, sequence_length)`, where `in_channels` is the same as the feature dimension, and `sequence_length` is the lookback period dimension which is the dimension that is convolved across in all of the convolutions. It uses `out_channels` number of filters to effectively upscale the feature dimension in its output allowing more temporal relationships to be considered. The second dilated causal convolution takes the input of `out_channels` channels and gives an output of `out_channels` channels, keeping the same channel/feature dimensions. After each dilated casual convolution, there is a dropout of 0.2, and a ReLU activation. The input to the TCN block is also added to output at the end before the

output is ReLU'd. If the `input_channels` \neq `output_channels`, the input is passed through a simple 1d convolutional layer with `output_channels` number of filters and a kernel size $K = 1$ so as to minimally effect input, to upscale or downscale the channel dimensions, so the residual connection can be added. For the dilated causal convolution the kernel sizes are default $K = 2$ for each.

The entire TCN is 3 of these TCN blocks and an operation at the end, with the pairs of (`in_channels`, `output_channels`) for each block as follows: $(f, 256), (256, 128), (128, 128)$. After the last TCN block the last timestep of the output, represented the agglomerated information and representation of all timesteps from the increasingly dilated casual convolutions, is passed through with a single `nn.Linear` (MLP with 0 hidden layers), with input and output dimensions of $(128, h)$. The channel dimension list for the TCN is then $[f, 256, 128, 128, h]$, where there is also a temporal dimension that is initially `sequence_length` that is convolved over until it is lost before the final MLP layer.

3.5 TCKAN: Novel Hybrid KAN and TCN Model

I propose a novel architecture, the Temporal Convolutional Kolmogorov-Arnold Network, which uses causal dilated Kolmogorov-Arnold convolution instead canonical causal dilated convolution, and a KAN instead of the MLP, with the rest of the architecture the same. In causal dilated Kolmogorov-Arnold convolution, the operation is the same as (2.42), except instead using Kolmogorov Arnold kernels Φ^κ according to (2.38) (convolution is 1-dimensional over time).

The Dilated Kolmogorov-Arnold Convolution equation for a single filter is:

$$(\mathbf{O}_t)_i = (\mathbf{X}_t *_d \Phi^\kappa)_i = \sum_{c=1}^{C_{in}} \sum_{m=0}^{K-1} \phi_{c,m}((\mathbf{X}_t)_{c,i+d*m}) \quad (3.1)$$

and is made causal with the left padding described in Subsection 2.4.1 . The method implementation of this in my code is called `KANConv1d` and is inspired by `efficient-kan`. Like `efficient-kan` the computation of the bias or base function $w_b b$ (2.23) and $w_s spline$ are done separately and added at the end as convolution is linear in the filters $\mathbf{X}_t *_d (\mathbf{b} + \mathbf{s}) = \mathbf{X}_t *_d \mathbf{b} + \mathbf{X}_t *_d \mathbf{s}$, where \mathbf{b} is the base function part and \mathbf{s} is the spline part of KAN kernel Φ^κ . Convoluting with a filter of functions that are applied is not a natural operation in pytorch so it is reformulated.

For $\mathbf{X}_t *_d \mathbf{b}$ notice each element of \mathbf{b} , $b_i = w_b b(\cdot)$, so $\mathbf{b} = \mathbf{w}_b \odot b(\cdot)$. So we can apply $b(\cdot)$ as it is the same for each position of the kernel (no matter the

channel) to the input first. So the operation becomes $\mathbf{X}_t *_d \mathbf{b} = b(\mathbf{X}_t) *_d \mathbf{w}_b$.

For $\mathbf{X}_t *_d \mathbf{s}$ the operation is a bit more involved, but is analogous to how it is handled in `efficient-kan`. The `b_splines` function of `efficient-kan`, copied into my `KANConv1d` class, computes the b-spline bases using (2.27) for each feature/channel and returns a shape of `(batch, in_features, sequence_length, G+k)`. This is then `.viewed` into the shape `(batch, in_features * G+k, sequence_length)`. This is done because for convolution in `pytorch` the channel dimension must be the second dimension, and the flattening is done as β for one of the timesteps according to (2.32), so that each basis for each feature is computed and flattened, for each timestep. The coefficients for a single feature/channel are the c_i of $\sum_i c_i B_i(x)$ (2.24) are put in a flattened array for every feature/channel as $\mathbf{C}_{i\cdot}$ in (2.33), and this is done implicitly in `PyTorch` by expressing setting the 1-dimensional convolutional layer as `nn.Conv1d(in_channels=(in_features * (G+k)), ...)`, which means there will be, $f \times (G + k)$ channels and therefore weights for each element of the kernel. A row $\mathbf{C}_{i\cdot}$ can now represent an element of the kernel.

The convolution can be reformulated (where ϕ is just the spline part):

$$\sum_{m=0}^{K-1} \beta((\mathbf{X}_t)_{i+d*m}) \cdot \mathbf{C}_m = \sum_{c=1}^{C_{in}} \sum_{m=0}^{K-1} \phi_{c,m}((\mathbf{X}_t)_{c,i+d*m}) = (\mathbf{X}_t *_d \mathbf{s})_i \quad (3.2)$$

Showing the equivalence of the operations (there is no w_s in this case as it is absorbed into the c_i). The two spline parts are added at the end completing the `KANConv1d` implementation. There is also an option for causal padding which is default `causal=False`.

The rest of the architecture is the same as the TCN, but with the upscaling done with `KANConv1d` and the final `nn.Linear` layer replaced with `KANLinear`. There is also a `LayerNorm` over features dimension after the two dilated causal convolutions. The dimension list is also $[f, 256, 128, 128, h]$.

3.6 Transformer and KANTransformer

The transformer encoder forward pass is identical to the encoder described in Section 2.5 except for a dropout of 0.1 before each `LayerNorm`. In the experiments the following hyperparameters are used for the model: a embedding dimension $d_{\text{model}} = 128$, 8 heads for the MultiHead attention, and the position-wise feedforward dimensions have a hidden layer of size 128. There are 4 stacked encoder layers. The last timestep of the encoder is passed (dimension d_{model}) into `nn.Linear` to

give an output of dimension h .

For the KANTransformer these hyperparameters are the same, and so is the model architecture except each `nn.Linear` is replaced with a `KANLinear`.

3.7 PINNs for Black-Scholes PDE

For this experiment we compare MLPs and KANs for the option pricing model that uses PINN loss function. We use the Black Scholes PDE (2.50) as a residual, the summed boundary conditions residuals and the data error residual in our loss function.

$$\text{Loss} = \lambda_1 \cdot \text{MSE}_{\text{data}} + \lambda_2 \cdot \text{MSE}_{\text{PDE}} + \lambda_3 \cdot \text{MSE}_{\text{BC}} \quad (3.3)$$

Our MLP/KAN model is a function $V(S, t)$, where t is time to expiration, V is option price and S is price of underlying. The loss function method uses PyTorch's `autograd.grad` to compute the necessary derivatives for the loss. The λ parameters are all 1. The data error residual is simply the mean squared error of the models output V and the actual option price data. The option price data used is AAPL stock options.

Chapter 4

Data Preparation

4.1 ETTm Data and AAPL Stock and AAPL Stock Options Data

The ETT (Electricity Transformer Temperature) dataset is a publicly available dataset widely used for benchmarking time series forecasting models. It was used in the Informer paper [35], and a variety of other papers for benchmarking the performance of time-series models. It consists of long-term multivariate time series data related to the electricity consumption and transformer temperatures recorded by a Chinese power supply company. The dataset is available in several variants based on the time granularity of the data. ETTh1 and ETTh2 are hourly-sampled data. ETTm1 and ETTm2 are 15-minute interval sampled data.

The ETTm1 and ETTm2 data are used for the forecasting models in this project. The data is retrieved from NIXTLA which is an open source time series ecosystem, using their `datasetforecasts` pip package. The ETT datasets are in the `long_horizon` module of `datasetsforecast`, and ETTm1 and ETTm2 are downloaded using `long_horizon.LongHorizon.load`.

The downloaded data comes with endogenous and exogenous variables, which is feature engineered data provided by NIXTLA. Only the endogenous variables are used. The data has multiple features that are each tagged by a `unique_id`. For the both datasets the `unique_id`'s and hence features are 'HUFL', 'HULL', 'LUFL', 'LULL', 'MUFL', 'MULL', 'OT', where the 4 character `unique_id`'s are different levels of power consumption/usage levels for different loads for the transformers, and 'OT' is the oil temperature, which is the target used for forecasting. The DataFrame is transformed using pandas's `.pivot` to turn features from 'unique_id's into columns.

AAPL stock data is also used for the time-series forecasting, with the last 5 years downloaded using `yfinance` with method `yfinance.download("AAPL, period="5y")`. The features for the AAPL stock are 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', which are the stock's daily opening, maximum, minimum, closing, adjusted closing prices and volume. The target used is the 'Close' price.

For the PINN option pricing experiments, AAPL stock option data is retrieved first by using `yfinance.Ticker("AAPL").options`, to get expiration dates and then this is looped over to get the option chain for each, `yfinance.Ticker("AAPL").option_chain(expir`. The `.calls` property is accessed to get the call options. The data is looped through to get option price, underlying price, time to expiration, strike price and implied volatility.

4.2 Preprocessing

For the ETTm datasets the data is already normalised by NIXTLA to between -1 and 1. For the AAPL stock data, the mean is subtracted and then its divided by the standard deviation, then the data is minmax scaled to -1 and 1. For the AAPL stock option data no normalisation is used.

Chapter 5

Results and Discussion

5.1 Performances of Models on Time-Series Prediction

5.1.1 Results

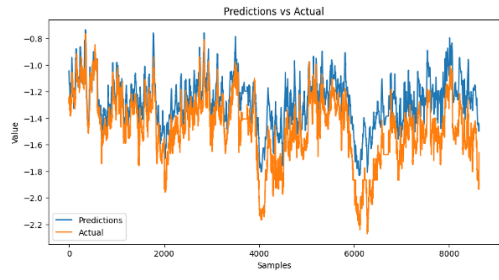
The mean square error and mean absolute percentage error results for each dataset, model and set of forecasting horizons are shown in the table below:

Data	FH	MLP		KAN		TCN		TCKAN		Transformer		KANTransformer	
		MSE	MAPE	MSE	MAPE	MSE	MAPE	MSE	MAPE	MSE	MAPE	MSE	MAPE
AAPL Stock	1	0.0124	7.3916	0.0162	16.9614	0.0011	5.0561	0.0347	18.6266	0.0296	18.6499	0.0493	24.4500
	12	0.0342	12.0139	0.0281	22.8748	0.0091	13.2520	0.0535	25.9718	0.0662	27.5846	0.1964	67.1046
	24	0.0985	19.1886	0.0288	23.0675	0.0261	21.0062	0.0639	28.8669	0.0404	23.0636	0.1194	43.5702
	48	0.3854	31.8671	0.0239	20.1719	0.0737	32.7801	0.0789	33.5295	0.0627	32.0978	0.1308	52.4660
ETTM1	1	0.0695	15.2678	0.0616	7.6880	0.0109	6.1350	0.0020	2.8318	–	–	–	–
	12	0.1337	21.4892	0.2760	21.7738	0.0659	15.3915	0.0623	14.6607	–	–	–	–
	24	0.1576	23.3858	1.5039	31.5698	0.0726	15.4706	0.1619	23.1924	–	–	–	–
	48	0.4641	37.8475	0.4641	33.4480	0.2304	27.3779	0.4027	36.6891	–	–	–	–
ETTM2	1	0.0130	8.8466	0.1882	18.8683	76.7480	106.1547	–	–	–	–	–	–
	12	0.0916	28.7356	0.2251	33.9296	96.8810	116.1427	–	–	–	–	–	–
	24	0.1705	42.7802	2.3131	93.6540	123.1123	136.2179	–	–	–	–	–	–
	48	0.5272	64.7698	1.1406	78.5234	180.0000	156.2179	–	–	–	–	–	–

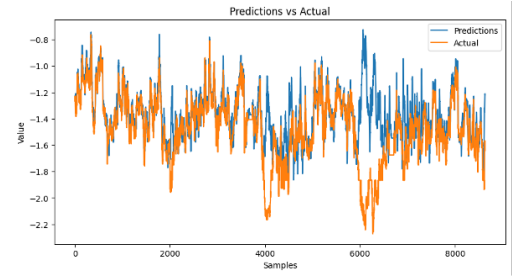
Table 5.1: Time-Series Forecasting Results

Where the values are bolded when a KAN model variant outperforms its non-KAN model counterpart. Due to time constraints in training not all values for all combinations are present and omitted values are denoted with ‘–’.

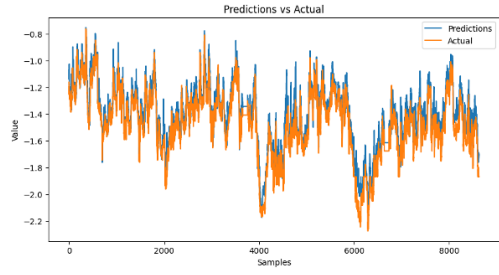
The actual target vs predicted target data is shown below for MLP, KAN, TCN and TCKAN models on the ETTm1 dataset, where the first data point of the forecast horizon is plotted.



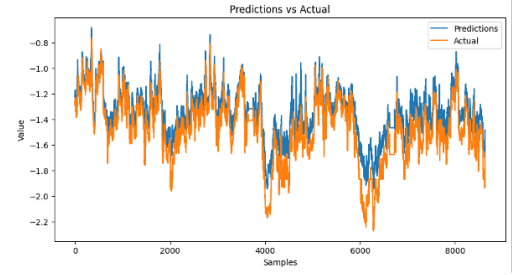
(a) MLP with horizon $f = 12$



(b) KAN with horizon $f = 1$



(c) TCN with horizon $f = 24$



(d) TCKAN with horizon $f = 12$

Figure 5.1: ETTm1 dataset predictions for different models and forecast horizons

For the TCKAN with $f = 12$ the forecast horizons are plotted for intervals of data points.

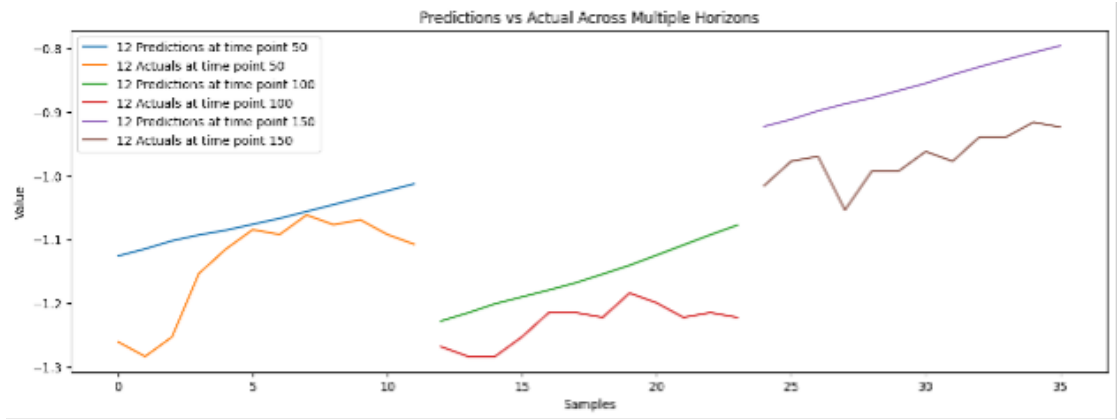


Figure 5.2: TCKAN $f = 12$ horizon predictions at point 50, 100, 150.

The actual target vs predicted target for MLP, KAN, TCN and TCKAN on the AAPL stock with $f = 1$ are shown below.

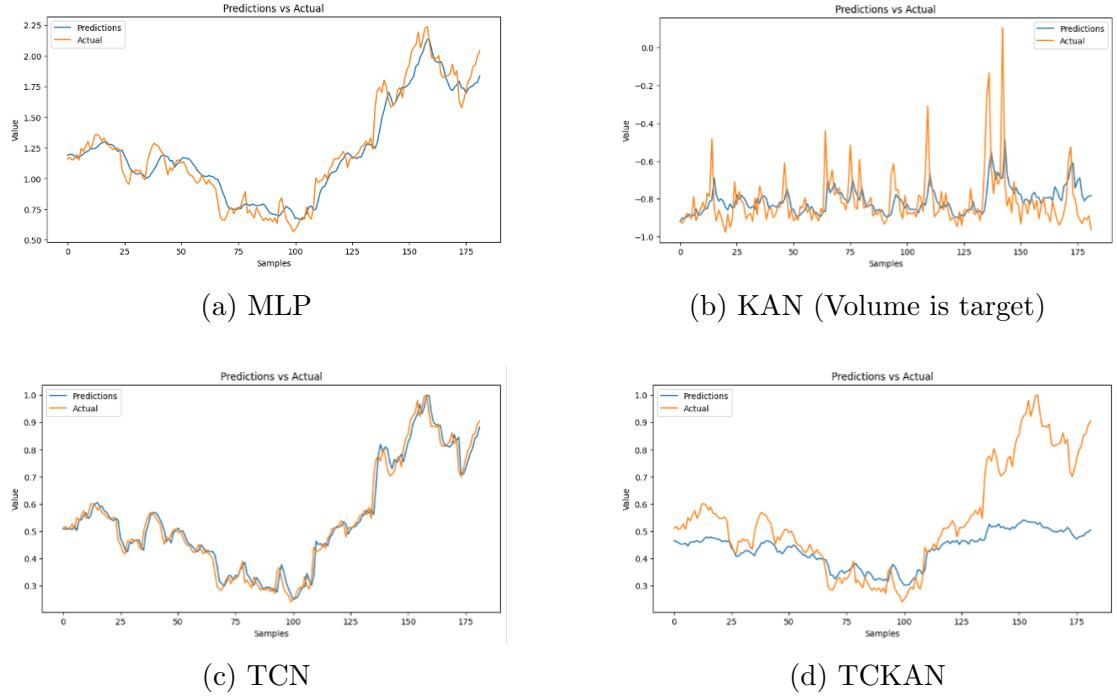


Figure 5.3: AAPL stock dataset predictions

5.1.2 Analysis and Discussion

The analysis reveals that KAN models demonstrate selective advantages over their non-KAN counterparts. Specifically, KAN outperforms MLP in AAPL Stock data for longer forecasting horizons ($f = 12$, $f = 24$, $f = 48$) in terms of Test Loss (MSE), although it does not surpass MLP in Mean Absolute Percentage Error (MAE). For the ETTm1 dataset, KAN outperforms MLP at $f = 1$ in both MSE and MAE, and also shows superiority in MAE at $f = 48$.

In comparing TCKAN to TCN, TCKAN shows clear advantages only in the ETTm1 dataset for the shorter horizons ($f = 1$ and $f = 12$), where it outperforms TCN in both MSE and MAE. However, TCKAN does not exhibit any performance gains over TCN for AAPL Stock.

KANTransformer does not outperform the Transformer model in any scenario, indicating that the KAN mechanism does not provide a benefit in the Transformer-based architecture for this data.

Overall, while the KAN models show specific improvements in certain scenarios, particularly for the ETTm1 dataset in short-term forecasting, the traditional

models (MLP, TCN, Transformer) generally perform better across most of the datasets and forecasting horizons.

More investigations are required, as KANs and KAN variants are very recent. Architectural improvements should be considered as when data passes through layers, it can sometimes go out of the interval specified by the spline grid.

In totality, across datasets and models the TCN generally provides the best performance, where on the AAPL stock it achieves the lowest MSE and MAE across almost all forecasting horizons.

5.2 Option Pricing PINN Models Performances

5.2.1 Results, Analysis and Discussion

For the option pricing model the training loss and option price predictions versus actual option price are plotted below for the KAN model. The results for the MLP

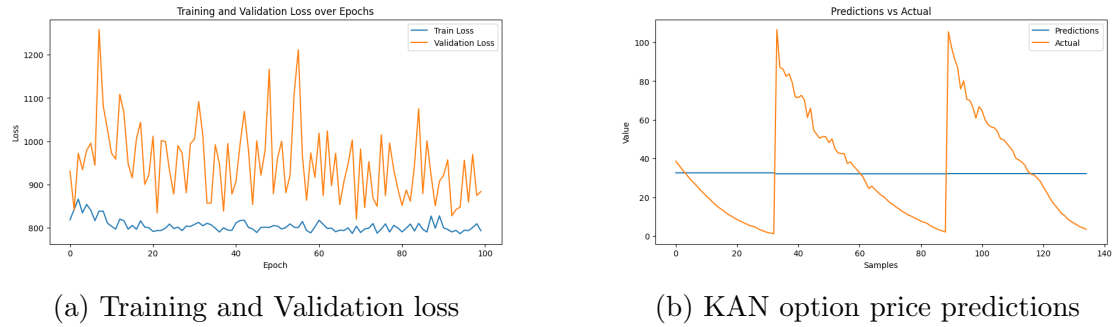


Figure 5.4: KAN option model results

were very similar, with this horizontal prediction, which indicates some problem with the implementation, which could possibly be due to the boundary conditions in the loss or data preparation issues.

Chapter 6

Conclusion

6.1 Summary and Contributions

This project evaluated the Kolmogorov-Arnold Network (KAN), a KAN Transformer variant, and the novel TCKAN architecture alongside the non-KAN counterpart models like MLP, TCN, and Transformer in the application of time-series forecasting across various datasets and forecasting horizons. While TCKAN showed some advantages, particularly in short-term forecasting on the ETTm1 dataset, traditional models such as TCN and MLP generally performed better overall. TCKAN outperformed TCN in specific scenarios, but it did not consistently offer improvements across other datasets, indicating that the architecture may require further refinement. KAN-based models showed selective improvements, especially in terms of MSE for longer forecasting horizons in AAPL Stock data, but these gains were not consistent across all metrics and datasets. The KANTransformer, in particular, did not outperform the standard Transformer. This work contributes to the exploration of new architectures like TCKAN. The findings suggest that while there are some promising, these models need further development to achieve broader applicability and consistent performance across different forecasting tasks.

In the option pricing model using PINN neural networks, the results are inconclusive, though the theory is sound and more time needs to be committed to practical implementation.

6.2 Future Work

There are various avenues for future work, the most prescient is to extend the set of results so that it covers all commonly used datasets and benchmarks for time-series prediction so a more exhaustive evaluation can be made. NIXTLA provides

a variety of other datasets, and using state of the art models, a more thorough comparison can be made between all the models used in this paper, and also with specific alterations to the state of the art models like trading out linear layers with KAN layers and using KAN convolution.

In terms of the option-pricing models, more time must be devoted to the implementation of the models and data preparation, so that options can be priced with good accuracy, however it is theoretically sound.

After these immediate concerns are addressed I propose an investigation into an implementation a hybrid time-series option-pricing model, using principles from time-series forecasting, such as having a lookback window of length p on the various inputs to the option pricing model and the option itself, such that it takes previous values of V, S, t , etc, and tries to forecast a window of option prices V , whose length is specified by a horizon h , while implementing the Black-Scholes PDE into the loss using principles from Physics-Informed Neural Networks. The exact implementation details must be thought through further, such as the type of neural network architecture of the model.

Chapter 7

Appendices

7.1 Code Snippets

```
[ ] '''
NXLong horizon datasets:

'ETTh1', 'ETTh2',
'ETTm1', 'ETTm2',
'ECL', 'Exchange',
'TrafficL', 'Weather', 'ILI'.
'''

# Nixtla Forecast Datasets
# Exchange - good
# ECL - bad
# ETTh1 - good
# ETTh2 - good
# ETTm1 - good
# ETTm2 - good
# TrafficL - bad
# Weather - good
# ILI - bad

DATASET = 'ETTm2'
data_tuple = long_horizon.LongHorizon.load("./", DATASET, True)

# data_tuple = long_horizon.LongHorizon.load("./", 'ETTh1', True)
# data_tuple = long_horizon.LongHorizon.load("./", 'ETTh2', True)
# data_tuple = long_horizon.LongHorizon.load("./", 'ETTm1', True)
# data_tuple = long_horizon.LongHorizon.load("./", 'ETTm2', True)
# data_tuple = long_horizon.LongHorizon.load("./", 'Exchange', True)
# data_tuple = long_horizon.LongHorizon.load("./", 'ECL', True)

en, ex, _ = data_tuple # endogenous variable and exogenous
```

100% | 314M/314M [00:07<00:00, 44.6MiB/s]

```
[ ] # tsdata = pd.concat([ex, en.drop(columns=['ds', 'unique_id'])], axis=1)
# uniques = np.unique(tsdata[["unique_id"]].values)
# print(uniques)
# for unique in uniques:
#     series = tsdata[tsdata["unique_id"] == unique]
#     print(len(series))
#     print(series.head())

uniques = np.unique(en[["unique_id"]].values)
print(uniques)
# for unique in uniques:
#     variable = en[en["unique_id"] == unique]
#     print(len(variable))
#     print(variable.head())

data = en.pivot(index='ds', columns='unique_id', values='y')
data.columns.name = None # Remove the automatic naming of the 'unique_id' dimension
data = data.rename_axis(None, axis=1) # Remove the name for the columns index

print(data.head())
```

```
[ ] ['HUFL' 'HULL' 'LUFL' 'LUFL' 'MUFL' 'MULL' 'OT']
      HUFL      HULL      LUFL      LULL      MUFL \
ds
2016-07-01 00:00:00 -0.041413  0.040104  0.695804  0.434430 -0.599211
2016-07-01 00:15:00 -0.185467 -0.214450  0.434685  0.428168 -0.658068
2016-07-01 00:30:00 -0.257495 -0.378215  0.698168  0.428168 -0.728020
2016-07-01 00:45:00 -0.577510 -0.669474  0.698168  0.423087 -0.834580
2016-07-01 01:00:00 -0.385501 -0.469220  0.698168  0.423087 -0.753474

      MULL      OT
ds
2016-07-01 00:00:00 -0.393536  1.018032
2016-07-01 00:15:00 -0.659338  0.980124
2016-07-01 00:30:00 -0.978366  0.904223
2016-07-01 00:45:00 -0.606111  0.885226
2016-07-01 01:00:00 -0.996218  0.885226
```

```
[ ] import yfinance as yf
# import pandas as pd

# Define the ticker symbol for Apple
ticker_symbol = 'AAPL'

# Fetch historical market data for the last 5 years
data = yf.download(ticker_symbol, period="5y")
print("\n")

# Display the first few rows of the data
data.reset_index(inplace=True)
data = (data - data.mean()) / data.std()

data = 2 * (data - data.min()) / (data.max() - data.min()) - 1
print(data.head())

column_names = data.columns

uniques = [col for col in column_names]

print(uniques)
```

39

```
[ ] [*****100%*****] 1 of 1 completed

Date      Open      High      Low      Close  Adj Close  Volume
2019-09-03 -1.000000 -1.000000 -1.000000 -1.000000 -1.000000 -0.721495
2019-09-04 -0.994699 -0.993261 -0.991485 -0.990485 -0.990862 -0.738091
2019-09-05 -0.984936 -0.981157 -0.979976 -0.979334 -0.980154 -0.644157
2019-09-06 -0.979391 -0.979944 -0.977230 -0.979389 -0.980206 -0.734629
2019-09-09 -0.977255 -0.974499 -0.981185 -0.976908 -0.977823 -0.576659
['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']
```

Figure 7.1: Datasets

```

# ALL EXCEPT STOCK
# TARGET = uniques[-1]
# AAPL STOCK
TARGET = uniques[3]

X = torch.tensor(data[uniques].values, dtype=torch.float32) # features
y = torch.tensor(data[TARGET].values, dtype=torch.float32) # target

# FORECAST LENGTHS: 1, 12, 24, 48

SEQUENCE_LENGTH = 48

FORECAST_LENGTH = 1

VARIATES = X.shape[1]
print(VARIATES)

class print_shape(nn.Module):
    def __init__(self):
        super(print_shape, self).__init__()

    def forward(self, x):
        print(x.shape)
        return x

# model = LSTMModel(VARIATES, 128, FORECAST_LENGTH, 2) #
# model = MLP(VARIATES+SEQUENCE_LENGTH, 256, FORECAST_LENGTH)
# model = TCNModel(VARIATES, FORECAST_LENGTH, [256, 128, 128], kernel_size=2, dropout=0.2)
# model = TransformerTimeSeriesModel(VARIATES, FORECAST_LENGTH)

# model = nn.Sequential(nn.Flatten(), KAN([VARIATES+SEQUENCE_LENGTH, 256, 256, FORECAST_LENGTH]))

model = TCKAN(SEQUENCE_LENGTH, VARIATES, FORECAST_LENGTH, [256, 128, 128], kernel_size=2, dropout=0.2)
# model = KANTransformerTimeSeriesModel(VARIATES, FORECAST_LENGTH)

model = model.to(device)

# -----
# Create Sequences
#
# Single Target Forecast: [ex_t-p+1, ..., ex_t] and [y_t-p+1, ..., y_t] to predict y_t+1
#
# Multi Target Forecast: [ex_t-p+1, ..., ex_t] and [y_t-p+1, ..., y_t+f-1] to predict [y_t+1, ..., y_t+f]
#
# -----

def create_sequences(X, y, seq_length=SEQUENCE_LENGTH, forecast_length=FORECAST_LENGTH): # SINGLE TARGET, SINGLE STEP FORECAST
    xs, ys = [], []
    for i in range(seq_length, len(X)-forecast_length):
        xs.append(X[i-seq_length+1: i+1])
        ys.append(y[i+1])
    return torch.stack(xs), torch.stack(ys)

def create_sequences(X, y, seq_length=SEQUENCE_LENGTH, forecast_length=FORECAST_LENGTH): # SINGLE TARGET, MULTI STEP FORECAST
    xs, ys = [], []
    for i in range(seq_length, len(X)-forecast_length):
        xs.append(X[i-seq_length+1: i+1])
        ys.append(y[i+1 : i+forecast_length+1])
    return torch.stack(xs), torch.stack(ys)

# Apply function to create sequences
X_seq, y_seq = create_sequences(X, y, SEQUENCE_LENGTH, FORECAST_LENGTH)

print(X_seq.shape)
print(y_seq.shape)

## Check shapes
# print("Shape of X_seq:", X_seq.shape) # (num_samples, seq_length, num_features)
# print("Shape of y_seq:", y_seq.shape) # (num_samples, 1)

## Example output
# print("X_seq:", X_seq[:2]) # Print first 2 sequences
# print("y_seq:", y_seq[:2]) # Print corresponding y values

# print("X:", X[seq_length:seq_length+2])
# print("y:", y[seq_length:seq_length+2])

# Time series split hyperparams from Nixtla
# test_size = getattr(long_horizon, DATASET).test_size
# val_size = getattr(long_horizon, DATASET).val_size
# train_size = len(X_seq) - test_size - val_size

# print(test_size/len(X_seq))
# print(val_size/len(X_seq))
# print(train_size/len(X_seq))

# Time series split 0.7:0.15:0.15
train_size = int(0.7 * len(X_seq))
val_size = int(0.15 * len(X_seq))
test_size = len(X_seq) - train_size - val_size

X_train = X_seq[:train_size]
X_val = X_seq[train_size:train_size + val_size]
X_test = X_seq[train_size + val_size:]

y_train = y_seq[:train_size]
y_val = y_seq[train_size:train_size + val_size]
y_test = y_seq[train_size + val_size:]

```

Figure 7.2: Time-series creation

```

train_data = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_data, batch_size=32, shuffle=False)

val_data = TensorDataset(X_val, y_val)
val_loader = DataLoader(val_data, batch_size=32, shuffle=False)

test_data = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)

LEARNING_RATE = 1e-3

early_stopping = EarlyStopping(patience=100, verbose=False)

optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
# scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1) # Reduce learning rate by a factor of 0.1 every 10 epochs
criterion = nn.MSELoss()

train_losses = []
val_losses = []

# Training loop
epochs = 100

for epoch in range(epochs):
    model.train()
    train_loss = 0.0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_loader)
    train_losses.append(train_loss)

    # Validation loop
    model.eval()
    val_loss = 0.0

    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(val_loader):
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = criterion(output, target)
            val_loss += loss.item()

    val_loss /= len(val_loader)
    val_losses.append(val_loss)

    early_stopping(val_loss, model)

    if early_stopping.early_stop:
        print("Early stopping")
        break

    print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")

model.load_state_dict(torch.load('checkpoint.pt'))

```

Figure 7.3: Training loop

```

class CausalConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, dilation):
        super(CausalConv1d, self).__init__()
        self.padding = (kernel_size - 1) * dilation
        self.conv = nn.Conv1d(in_channels, out_channels, kernel_size, stride=stride, padding=0, dilation=dilation)

    def forward(self, x):
        # x = F.pad(x, (self.padding, 0))
        return self.conv(F.pad(x, (self.padding, 0)))

class TemporalConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, dilation, dropout=0.2):
        super(TemporalConvBlock, self).__init__()
        # padding = (kernel_size - 1) * dilation
        # self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size, stride=stride, padding=padding, dilation=dilation)
        # self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size, stride=stride, padding=padding, dilation=dilation)
        self.conv1 = CausalConv1d(in_channels, out_channels, kernel_size, stride=stride, dilation=dilation)
        self.conv2 = CausalConv1d(out_channels, out_channels, kernel_size, stride=stride, dilation=dilation)
        self.dropout = nn.Dropout(dropout)
        self.net = nn.Sequential(self.conv1, nn.ReLU(), self.dropout, self.conv2, nn.ReLU(), self.dropout)
        self.downsample = nn.Conv1d(in_channels, out_channels, 1) if in_channels != out_channels else None
        self.relu = nn.ReLU()

    def forward(self, x):
        # print("TCN Block Iteration\n")
        # print("x shape: ", x.shape)
        out = self.net(x)
        res = x if self.downsample is None else self.downsample(x)
        # print("Out shape: ", out.shape)
        # print("Res shape: ", res.shape)
        return self.relu(out + res)

class TemporalConvNet(nn.Module):
    def __init__(self, num_inputs, num_channels, kernel_size=2, dropout=0.2):
        super(TemporalConvNet, self).__init__()
        layers = []
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i-1]
            out_channels = num_channels[i]
            layers += [TemporalConvBlock(in_channels, out_channels, kernel_size, stride=1, dilation=dilation_size,
                                         dropout=dropout)]

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

class TCNModel(nn.Module):
    def __init__(self, input_size, output_size, num_channels, kernel_size=2, dropout=0.2):
        super(TCNModel, self).__init__()
        self.tcn = TemporalConvNet(input_size, num_channels, kernel_size=kernel_size, dropout=dropout)
        self.linear = nn.Linear(num_channels[-1], output_size)

    def forward(self, x):
        # print("TCN Model \n")
        # print("x shape: ", x.shape)
        x = x.transpose(1, 2) # B x T x F -> B x F x T
        # conv1d needs B x F x T in order to convolve across time
        y1 = self.tcn(x)
        o = self.linear(y1[:, :, -1]) # selects last timestep
        return o

```

Figure 7.4: TCN Implementation

```

class KANConv1d(nn.Module):
    def __init__(self, sequence_length, in_features, out_features, kernel_size=3, grid_size=5, spline_order=3, padding=0, dilation=1, stride=1,
                 base_activation=torch.nn.SiLU,
                 grid_range=[-1, 1], causal=False
    ):
        super(KANConv1d, self).__init__()

        self.sequence_length = sequence_length
        self.kernel_size = kernel_size
        self.dilation = dilation
        # self.padding = (kernel_size - 1) * dilation
        self.padding = padding

        self.grid_size = grid_size
        self.spline_order = spline_order

        self.stride=stride

        self.in_features = in_features
        self.out_features = out_features
        self.grid_size = grid_size
        self.spline_order = spline_order

        h = (grid_range[1] - grid_range[0]) / grid_size
        grid = (
            torch.arange(-spline_order, grid_size + spline_order + 1) * h
            + grid_range[0]
        )
        .expand(in_features, -1)
        .unsqueeze(1) # Add a dimension for sequence_length
        .expand(in_features, sequence_length, -1)
        .contiguous()
        )
        self.register_buffer("grid", grid)

        # no stand alone scale spline

        # m = grid_size + 2*spline_order + 1 is length of knot vector, our number of bases is given by n = m-spline_order-1
        # number of bases n = grid_size + spline_order

        self.causal = causal

        if self.causal:
            self.padding = 0

        self.base_conv = nn.Conv1d(in_features, out_features, kernel_size, dilation=dilation, padding=self.padding, stride=self.stride)
        self.spline_conv = nn.Conv1d( (self.grid_size + self.spline_order) * in_features, out_features, kernel_size, dilation=dilation, padding=self.padding, stride=self.stride)

        nn.init.kaiming_uniform_(self.base_conv.weight, nonlinearity='linear')
        nn.init.kaiming_uniform_(self.spline_conv.weight, nonlinearity='linear')

        self.base_activation = base_activation()

```

Figure 7.5: KANConv1d class defintion

```

def forward(self, x: torch.Tensor): # FORWARD
    # print("x shape: ", x.shape)

    # Expects B x F x T
    # x = x.transpose(1, 2) # B x T x F -> B x F x T
    # print("x shape: ", x.shape)

    assert x.size(1) == self.in_features
    original_shape = x.shape

    base_act = self.base_activation(x)

    padded_base_act = F.pad(base_act, ((self.kernel_size-1)*self.dilation, 0))
    # print("Padded x shape: ", padded_x.shape)
    base_input = padded_base_act if self.causal else base_act

    base_output = self.base_conv(base_input)

    # print("B-Spline shape: ", self.b_splines(x).shape)
    # print("B-Spline View shape: ", self.b_splines(x).view(x.size(0), self.in_features, -1).shape)
    # print("Spline Weight shape: ", self.spline_conv.weight.shape)
    # print("Spline Weight View shape: ", self.spline_conv.weight.view(self.out_features, self.in_features, -1).shape)

    '''
    B-Spline shape: torch.Size([32, 7, 5, 8])
    B-Spline View shape: torch.Size([32, 7, 40])
    Spline Weight shape: torch.Size([64, 7, 3, 8])
    Spline Weight View shape: torch.Size([64, 7, 24])
    '''

    spline_bases = self.b_splines(x).view(x.size(0), self.in_features * (self.grid_size + self.spline_order), -1)
    spline_bases_padded = F.pad(spline_bases, ((self.kernel_size-1)*self.dilation, 0))

    spline_input = spline_bases_padded if self.causal else spline_bases

    spline_output = self.spline_conv(spline_input)

    output = base_output + spline_output # B x F x T
    # print("Output shape: ", output.shape) # torch.Size([32, 64, 5])
    # output = output.transpose(1, 2)
    # print("Output shape: ", output.shape)
    return output

```

Figure 7.6: KANConv1d forward pass

```

def b_splines(self, x: torch.Tensor):
    """
    Compute the B-spline bases for the given input tensor.

    Args:
        x (torch.Tensor): Input tensor of shape (batch_size, in_features, sequence_length).

    Returns:
        torch.Tensor: B-spline bases tensor of shape (batch_size, in_features, sequence_length, grid_size + spline_order).
    """
    assert x.dim() == 3 and x.size(1) == self.in_features

    grid: torch.Tensor = (
        self.grid
    ) # (in_features, grid_size + 2 * spline_order + 1)
    x = x.unsqueeze(-1)

    # bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).to(x.dtype)
    bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).to(x.dtype)
    # print(bases.shape)
    # print(grid.shape)

    for k in range(1, self.spline_order + 1):
        bases = (
            (x - grid[:, :-1])
            / (grid[:, k:-1] - grid[:, :-1])
            * bases[:, :-1]
        ) + (
            (grid[:, k + 1 :] - x)
            / (grid[:, k + 1 :] - grid[:, 1:(-k)])
            * bases[:, 1:]
        )

    assert bases.size() == (
        x.size(0),
        self.in_features,
        self.sequence_length,
        self.grid_size + self.spline_order,
    )
    return bases.contiguous()

```

Figure 7.7: KANConv1d B-splines method


```

class FeaturesLayerNorm(nn.Module):
    def __init__(self, normalized_shape):
        super(FeaturesLayerNorm, self).__init__()
        self.layer_norm = nn.LayerNorm(normalized_shape)

    def forward(self, x):
        x = x.transpose(1, 2) # B x F x T -> B x T x F
        x = self.layer_norm(x)
        return x.transpose(1, 2) # B x T x F -> B x F x T

class TemporalKANConvBlock(nn.Module):
    def __init__(self, sequence_length, in_channels, out_channels, kernel_size, stride, dilation, dropout=0.2):
        super(TemporalKANConvBlock, self).__init__()
        # padding = (kernel_size - 1) * dilation
        # self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size, stride=stride, padding=padding, dilation=dilation)
        # self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size, stride=stride, padding=padding, dilation=dilation)
        self.conv1 = KANConv1d(sequence_length, in_channels, out_channels, kernel_size, stride=stride, dilation=dilation, causal=True)
        self.conv2 = KANConv1d(sequence_length, out_channels, out_channels, kernel_size, stride=stride, dilation=dilation, causal=True)
        self.layer_norm = FeaturesLayerNorm(out_channels)
        self.dropout = nn.Dropout(dropout)
        self.net = nn.Sequential(self.conv1, self.layer_norm, self.dropout, self.conv2, self.layer_norm, self.dropout)
        # up/downsampling - sends in_channels into ka function, sums them, does this operation separately out channel times
        self.downsample = KANConv1d(sequence_length, in_channels, out_channels, kernel_size=1) if in_channels != out_channels else None
        # self.relu = nn.ReLU()

    def forward(self, x):
        # print("TCN Block Iteration\n")
        # print("x shape: ", x.shape)
        out = self.net(x)
        res = x if self.downsample is None else self.layer_norm(self.downsample(x))
        # print("Out shape: ", out.shape)
        # print("Res shape: ", res.shape)
        return out + res

class TemporalKANConvNet(nn.Module):
    def __init__(self, sequence_length, num_inputs, num_channels, kernel_size=2, dropout=0.2):
        super(TemporalKANConvNet, self).__init__()
        layers = []
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i-1]
            out_channels = num_channels[i]
            layers += [TemporalKANConvBlock(sequence_length, in_channels, out_channels, kernel_size, stride=1, dilation=dilation_size,
                                            dropout=dropout)]

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

class TCKAN(nn.Module):
    def __init__(self, sequence_length, input_size, output_size, num_channels, kernel_size=2, dropout=0.2):
        super(TCKAN, self).__init__()

        self.tcn = TemporalKANConvNet(sequence_length, input_size, num_channels, kernel_size=kernel_size, dropout=dropout)
        self.linear = KANLinear(num_channels[-1], output_size)

    def forward(self, x):
        # print("TCN Model \n")
        # print("x shape: ", x.shape)
        x = x.transpose(1, 2) # B x T x F -> B x F x T
        # conv1d needs B x F x T in order to convolve across time
        y1 = self.tcn(x)
        o = self.linear(y1[:, :, -1]) # selects last timestep
        return o

```

Figure 7.8: TCKAN Implementation

```

class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadSelfAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.out_linear = nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size = x.size(0)

        # Linear projections for query, key, and value
        Q = self.q_linear(x) # [batch_size, seq_len, d_model]
        K = self.k_linear(x) # [batch_size, seq_len, d_model]
        V = self.v_linear(x) # [batch_size, seq_len, d_model]

        # Split into multiple heads and then reshape to perform scaled dot-product attention
        Q = Q.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn_weights = F.softmax(scores, dim=-1)
        attn_output = torch.matmul(attn_weights, V) # [batch_size, num_heads, seq_len, d_k]

        # Concatenate heads and pass through the final linear layer
        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)
        output = self.out_linear(attn_output) # [batch_size, seq_len, d_model]

        return output

class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout=0.1):
        super(FeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, dim_feedforward)
        self.fc2 = nn.Linear(dim_feedforward, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward=128, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn = MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward = FeedForward(d_model, dim_feedforward, dropout)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, src):
        # Self-attention layer with residual connection and layer normalization
        attn_output = self.self_attn(src)
        src = src + self.dropout1(attn_output)
        src = self.norm1(src)

        # Feed-forward layer with residual connection and layer normalization
        ff_output = self.feed_forward(src)
        src = src + self.dropout2(ff_output)
        src = self.norm2(src)

        return src

# Encoder-Only Transformer Model
class TransformerTimeSeriesModel(nn.Module):
    def __init__(self, input_size, output_size, d_model=128, num_heads=8, num_encoder_layers=4, dim_feedforward=128, dropout=0.1):
        super(TransformerTimeSeriesModel, self).__init__()

        self.d_model = d_model
        self.input_embedding = nn.Linear(input_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, dropout)

        # Stack of custom encoder layers
        self.encoder_layers = nn.ModuleList([TransformerEncoderLayer(d_model, num_heads, dim_feedforward, dropout) for _ in range(num_encoder_layers)])

        # Final output projection
        self.fc_out = nn.Linear(d_model, output_size)

    def forward(self, src):
        # Embed the input sequence and add positional encoding
        src = self.input_embedding(src) * torch.sqrt(torch.tensor(self.d_model, dtype=torch.float32))
        src = self.positional_encoding(src)

        # Pass through the stack of encoder layers
        for encoder_layer in self.encoder_layers:
            src = encoder_layer(src)

        # Final output projection to the desired output size
        output = self.fc_out(src[:, -1, :]) # Use the last time step's encoded state for prediction

        return output

```

Figure 7.9: Transformer Implementation

```

class KANMultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(KANMultiHeadSelfAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.q_linear = KANLinear(d_model, d_model)
        self.k_linear = KANLinear(d_model, d_model)
        self.v_linear = KANLinear(d_model, d_model)
        self.out_linear = KANLinear(d_model, d_model)

    def forward(self, x):
        batch_size = x.size(0)

        # Linear projections for query, key, and value
        Q = self.q_linear(x) # [batch_size, seq_len, d_model]
        K = self.k_linear(x) # [batch_size, seq_len, d_model]
        V = self.v_linear(x) # [batch_size, seq_len, d_model]

        # Split into multiple heads and then reshape to perform scaled dot-product attention
        Q = Q.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn_weights = F.softmax(scores, dim=-1)
        attn_output = torch.matmul(attn_weights, V) # [batch_size, num_heads, seq_len, d_k]

        # Concatenate heads and pass through the final linear layer
        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)
        output = self.out_linear(attn_output) # [batch_size, seq_len, d_model]

        return output

class KANFeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward=128, dropout=0.1):
        super(KANFeedForward, self).__init__()
        self.fc1 = KANLinear(d_model, dim_feedforward)
        self.fc2 = KANLinear(dim_feedforward, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

class KANTransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward, dropout=0.1):
        super(KANTransformerEncoderLayer, self).__init__()
        self.self_attn = MultiHeadSelfAttention(d_model, num_heads)
        self.feed_forward = FeedForward(d_model, dim_feedforward, dropout)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, src):
        # Self-attention layer with residual connection and layer normalization
        attn_output = self.self_attn(src)
        src = src + self.dropout1(attn_output)
        src = self.norm1(src)

        # Feed-forward layer with residual connection and layer normalization
        ff_output = self.feed_forward(src)
        src = src + self.dropout2(ff_output)
        src = self.norm2(src)

        return src

# Encoder-Only Transformer Model
class KANTransformerTimeSeriesModel(nn.Module):
    def __init__(self, input_size, output_size, d_model=128, num_heads=8, num_encoder_layers=4, dim_feedforward=128, dropout=0.1):
        super(KANTransformerTimeSeriesModel, self).__init__()

        self.d_model = d_model
        self.input_embedding = KANLinear(input_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, dropout)

        # Stack of custom encoder layers
        self.encoder_layers = nn.ModuleList([TransformerEncoderLayer(d_model, num_heads, dim_feedforward, dropout) for _ in range(num_encoder_layers)])

        # Final output projection
        self.fc_out = KANLinear(d_model, output_size)

    def forward(self, src):
        # Embed the input sequence and add positional encoding
        src = self.input_embedding(src) * torch.sqrt(torch.tensor(self.d_model, dtype=torch.float32))
        src = self.positional_encoding(src)

        # Pass through the stack of encoder layers
        for encoder_layer in self.encoder_layers:
            src = encoder_layer(src)

        # Final output projection to the desired output size
        output = self.fc_out(src[:, -1, :]) # Use the last time step's encoder state for prediction

        return output

```

Figure 7.10: KANTransformer Implmeentation

```

import pandas as pd
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd
import torch.functional as F
from torch.utils.data import DataLoader, TensorDataset

ticker = yf.Ticker("AAPL")

underlying_price = ticker.history(period='1d')['Close'].iloc[-1]

expiration_dates = ticker.options

S_data = []
t_data = []
V_data = []
K_data = []
# T_data = []
IV_data = []

# Loop over expiration dates to gather data
for expiry in expiration_dates:
    options_chain = ticker.option_chain(expiry)
    calls = options_chain.calls

    for idx, row in calls.iterrows():
        if (pd.isna(underlying_price) or pd.isna(row['bid']) or pd.isna(row['ask']) or pd.isna(row['lastPrice']) or
            pd.isna(row['strike']) or pd.isna(row['impliedVolatility']) or pd.isna((pd.to_datetime(expiry) - pd.to_datetime('today')).days / 365.0) ):
            continue # Skip to the next row

        if row['bid'] == 0 or row['ask'] == 0:
            continue # Skip to the next row

        S_data.append(underlying_price) # Replace with the correct stock price if needed
        t_data.append((pd.to_datetime(expiry) - pd.to_datetime('today')).days / 365.0) # Time to expiration in years
        # V_data.append((row['bid'] + row['ask'])/2) # midpoint
        V_data.append((row['lastPrice'])/2) # midpoint
        K_data.append(row['strike']) # Strike price
        # T_data.append(expiry)
        IV_data.append(row['impliedVolatility']) # Implied volatilities

# Convert lists to tensors for model training
S_tensor = torch.tensor(S_data, dtype=torch.float32).unsqueeze(1)
t_tensor = torch.tensor(t_data, dtype=torch.float32).unsqueeze(1)
V_tensor = torch.tensor(V_data, dtype=torch.float32).unsqueeze(1)
K_tensor = torch.tensor(K_data, dtype=torch.float32).unsqueeze(1)
IV_tensor = torch.tensor(IV_data, dtype=torch.float32).unsqueeze(1)
# T_tensor = torch.tensor(T_data, dtype=torch.float32).unsqueeze(1)

total_size = len(S_tensor)
train_size = int(0.7 * total_size)
val_size = int(0.15 * total_size)
test_size = total_size - train_size - val_size # Ensures the rest goes to the test set

S_train, S_val, S_test = S_tensor[:train_size], S_tensor[train_size:train_size + val_size], S_tensor[train_size + val_size:]
t_train, t_val, t_test = t_tensor[:train_size], t_tensor[train_size:train_size + val_size], t_tensor[train_size + val_size:]
V_train, V_val, V_test = V_tensor[:train_size], V_tensor[train_size:train_size + val_size], V_tensor[train_size + val_size:]
K_train, K_val, K_test = K_tensor[:train_size], K_tensor[train_size:train_size + val_size], K_tensor[train_size + val_size:]
IV_train, IV_val, IV_test = IV_tensor[:train_size], IV_tensor[train_size:train_size + val_size], IV_tensor[train_size + val_size:]

```

Figure 7.11: Option model data loading

```

def black_scholes_pde_loss(model, V_target, S, t, K, r, sigma, option_type="call"):
    # Concat S and t
    r = torch.full_like(S, r)
    inputs = torch.cat((S, t), dim=1)
    inputs_at_T = torch.cat((S, torch.zeros_like(t)), dim=1)
    inputs_at_zero_underlying = torch.cat((torch.zeros_like(S), t), dim=1)
    S_large = 100000*K
    inputs_at_inf_underlying = torch.cat((S_large, t), dim=1)

    V = model(inputs)
    V_at_T = model(inputs_at_T)
    V_at_zero_underlying = model(inputs_at_zero_underlying)
    V_at_inf_underlying = model(inputs_at_inf_underlying)

    # Compute derivatives using autograd
    V_t = autograd.grad(V, t, grad_outputs=torch.ones_like(V), create_graph=True)[0]
    V_S = autograd.grad(V, S, grad_outputs=torch.ones_like(V), create_graph=True)[0]
    V_SS = autograd.grad(V_S, S, grad_outputs=torch.ones_like(V), create_graph=True)[0]

    # Black-Scholes PDE loss
    pde_loss = torch.mean((V_t + 0.5 * sigma**2 * S**2 * V_SS + r * S * V_S - r * V)**2)

    # if option_type == "call":
    #     bc1 = torch.maximum(S - K, torch.zeros_like(S)) # Final condition at t=T for call option
    #     bc2 = torch.zeros_like(t)
    #     bc3 = S_large - K * torch.exp(-r * (t))

    # else:
    #     bc1 = torch.maximum(K - S, torch.zeros_like(S)) # For put option
    #     bc2 = K * torch.exp(-r * (t))
    #     bc3 = torch.zeros_like(t)

    # bc_loss = torch.mean((V_at_T - bc1)**2) + torch.mean((V_at_zero_underlying - bc2)**2) + torch.mean((V_at_inf_underlying - bc3)**2)
    bc_loss = 0

    data_loss = torch.mean((V - V_target)**2)

    return (pde_loss + bc_loss + data_loss), pde_loss, bc_loss, data_loss, V

```

Figure 7.12: Black Scholes PDE PINN Loss Implementation

```

num_epochs = 100
learning_rate = 1e-3

# model = nn.Sequential(nn.Linear(2, 64), nn.ReLU(), nn.Linear(64,64), nn.ReLU(), nn.Linear(64, 1), nn.ReLU())
model = KAN([2,64,64,1])
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# S_train, t_train, V_train, K_train, IV_train tensors

# Create a TensorDataset and DataLoader for batching
train_dataset = TensorDataset(S_train, t_train, V_train, K_train, IV_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True) # Adjust batch_size as needed

val_dataset = TensorDataset(S_val, t_val, V_val, K_val, IV_val)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=True) # Adjust batch_size as needed

r = 4.519/100 # 1 year treasury bill

train_losses = []
val_losses = []

train_mse_losses = []
val_mse_losses = []

train_pde_losses = []
val_pde_losses = []

# Training loop
for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    train_mse_loss = 0
    train_pde_loss = 0

    for batch in train_loader:
        S_batch, t_batch, V_target_batch, K_batch, IV_batch = batch

        # Extract S and t from inputs
        # S_batch = inputs_batch[:, 0].unsqueeze(1)
        # t_batch = inputs_batch[:, 1].unsqueeze(1)
        S_batch.requires_grad_(True)
        t_batch.requires_grad_(True)

        optimizer.zero_grad()

        # Compute the loss using the fixed risk-free rate
        loss, pde_loss, bc_loss, mse_loss, pred = black_scholes_pde_loss(model, V_target_batch, S_batch, t_batch, K_batch, r, IV_batch)
        loss.backward()
        # nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        train_loss += loss.item()
        train_pde_loss += pde_loss.item()
        train_mse_loss += mse_loss.item()

    train_loss /= len(train_loader)
    train_losses.append(train_loss)
    train_pde_loss /= len(train_loader)
    train_pde_losses.append(train_pde_loss)
    train_mse_loss /= len(train_loader)
    train_mse_losses.append(train_mse_loss)

    # Validation loop
    model.eval()
    val_loss = 0.0
    val_mse_loss = 0.0
    val_pde_loss = 0.0

    for batch in val_loader:
        S_batch, t_batch, V_target_batch, K_batch, IV_batch = batch
        S_batch.requires_grad_(True)
        t_batch.requires_grad_(True)

        optimizer.zero_grad()

        # Compute the loss using the fixed risk-free rate
        loss, pde_loss, mse_loss, pred = black_scholes_pde_loss(model, V_target_batch, S_batch, t_batch, K_batch, r, IV_batch)
        loss.backward()
        # nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        val_loss += loss.item()
        val_pde_loss += pde_loss.item()
        val_mse_loss += mse_loss.item()

    val_loss /= len(val_loader)
    val_losses.append(val_loss)
    val_pde_loss /= len(val_loader)
    val_pde_losses.append(val_pde_loss)
    val_mse_loss /= len(val_loader)
    val_mse_losses.append(val_mse_loss)

print(f'Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss:.6f}, Validation Loss: {val_loss:.6f}')

```

Figure 7.13: Options model training loop

Bibliography

- [1] Vladimir I. Arnold and Andrey N. Kolmogorov. On functions of three variables. *Doklady Akademii Nauk SSSR*, 114:679–681, 1957.
- [2] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.
- [3] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.
- [4] Blealtan. Efficient-kan. <https://github.com/Blealtan/efficient-kan>, 2024.
- [5] Alexander Dylan Bodner, Antonio Santiago Tepsich, Jack Natan Spolski, and Santiago Pourteau. Convolutional kolmogorov-arnold networks, 2024.
- [6] Wei Dai, Yuan An, and Wen Long. Price change prediction of ultra high frequency financial data based on temporal convolutional network. *arXiv preprint arXiv:2107.00261*, 2021.
- [7] Carl de Boor. *A Practical Guide to Splines*. Springer-Verlag, New York, NY, USA, 1978.
- [8] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.
- [9] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feed-forward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
- [10] Yuntian Hou and Di Zhang. A Comprehensive Survey on Kolmogorov Arnold Networks (KAN). 7 2024.
- [11] KindXiaoming. Pykan. <https://github.com/KindXiaoming/pykan>, 2024.

- [12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [13] Andrey N. Kolmogorov. On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables. *Doklady Akademii Nauk SSSR*, 108:179–182, 1956. English translation: *Amer. Math. Soc. Transl.*, 17 (1961), 369–373.
- [14] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 156–165, 2017.
- [15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [16] J. Lee, T. Wang, and S. Liu. A comprehensive survey on kolmogorov arnold networks (kan). *arXiv preprint arXiv:2407.11075*, 2023. Accessed: 2024-09-01.
- [17] Bryan Lim and Stefan Zohren. Temporal convolutional networks for time series forecasting. *arXiv preprint arXiv:2002.09203*, 2021.
- [18] Ziming Liu, Pingchuan Ma, Yixuan Wang, Wojciech Matusik, and Max Tegmark. Kan 2.0: Kolmogorov-arnold networks meet science, 2024.
- [19] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks, 2024.
- [20] Valeriy Lobanov, Nikita Firsov, Evgeny Myasnikov, Roman Khabibullin, and Artem Nikonorov. Hyperkan: Kolmogorov-arnold networks make hyperspectral image classifiers smarter, 2024.
- [21] Francisco J. Ordóñez and Daniel Roggen. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1):115, 2016.
- [22] Yao Qin, Dongjin Song, Haifeng Chen, Wei Cheng, Guofei Jiang, and Garrison W Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 2627–2633. AAAI Press, 2017.

- [23] Maziar Raissi, Paris Perdikaris, Nazanin Ahmadi, and George Em Karniadakis. Physics-informed neural networks and extensions. *arXiv preprint arXiv:2408.16806*, 2024.
- [24] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.
- [25] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [26] Seyd Teymoor Seydi. Unveiling the power of wavelets: A wavelet-based kolmogorov-arnold network for hyperspectral image classification, 2024.
- [27] Cristian J. Vaca-Rubio, Luis Blanco, Roberto Pereira, and Màrius Caus. Kolmogorov-arnold networks (kans) for time series analysis, 2024.
- [28] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.
- [29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.
- [30] Yanheng Wang, Xiaohan Yu, Yongsheng Gao, Jianjun Sha, Jian Wang, Lianru Gao, Yonggang Zhang, and Xianhui Rong. Spectralkan: Kolmogorov-arnold network for hyperspectral images change detection, 2024.
- [31] Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. Transformers in time series: A survey, 2023.
- [32] Kunpeng Xu, Lifei Chen, and Shengrui Wang. Kolmogorov-arnold networks for time series: Bridging predictive power and interpretability, 2024.
- [33] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. Dilated residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 472–480, 2017.
- [34] Yanping Zheng, Yu Zhang, Yudong Zheng, Ping Huang, and Liuyi Wang. A fully convolutional neural network for time series classification. *arXiv preprint arXiv:1611.06455*, 2017.

- [35] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115, 2021.