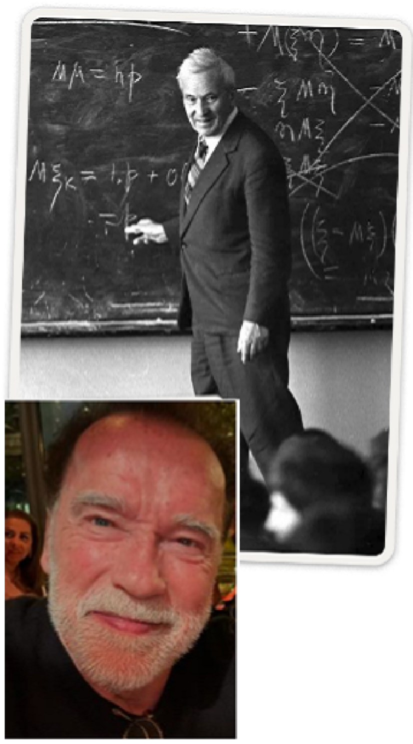
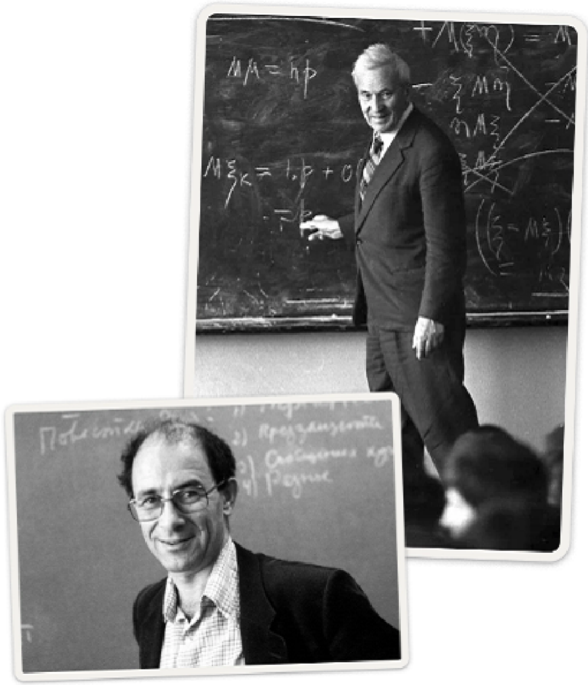
t

Kolmogorov-Arnold Neural Network

Who are they?



Really? Perhaps another Arnold?!



1. A.N. Kolmogorov, 'On the representation of continuous functions of several variables as superpositions of continuous functions of a smaller number of variables', *Dokl. Akad. Nauk SSSR* **108**:2 (1956), 179–182 (in Russian). (See No. 55.)
2. V.I. Arnol'd, 'On the representation of continuous functions of three variables as superpositions of continuous functions of two variables', *Dokl. Akad. Nauk SSSR* **114**:4 (1957), 679–681 (in Russian).

Kolmogorov- Arnold Representation Theorem

The Kolmogorov-Arnold representation theorem, which states that any continuous function of several variables can be represented as a finite superposition of continuous functions of a single variable and addition.

Example 1

```
In[∘]:= f[x_, y_] = x / y;
Φ1[u_] = Exp[u];
ϕ1,1[u_] = Log[u];
ϕ1,2[u_] = -Log[u];
f[x, y] == Φ1[ϕ1,1[x] + ϕ1,2[y]]
Out[∘]=
True
```

Example 2

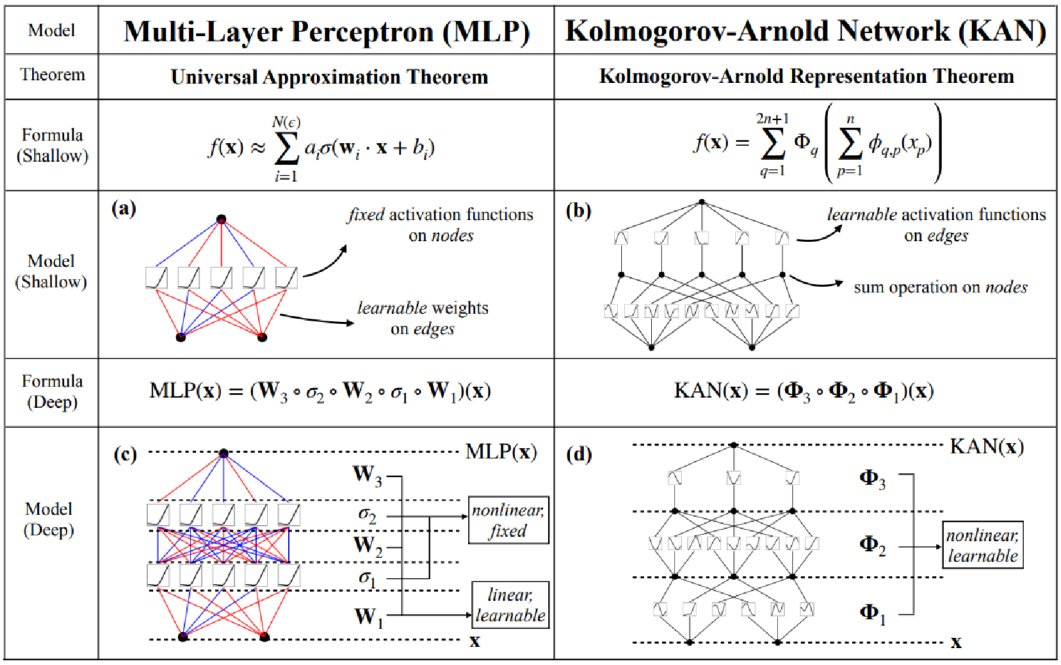
```
In[∘]:= f[x_, y_] = x^y;
Φ1[u_] = Exp[Exp[u]];
ϕ1,1[u_] = Log[Log[u]];
ϕ1,2[u_] = Log[u];
f[x, y] == Φ1[ϕ1,1[x] + ϕ1,2[y]]
Out[∘]=
True
```

Example 3

```
In[∘]:= f[x_, y_, z_] = x^y + Sin[y / z];
Φ1[u_] = Exp[Exp[u]];
ϕ1,1[u_] = Log[Log[u]];
ϕ1,2[u_] = Log[u];
ϕ1,3[u_] = 0;
Φ2[u_] = Sin[Exp[u]];
ϕ2,1[u_] = 0;
ϕ2,2[u_] = Log[u];
ϕ2,3[u_] = -Log[u];
f[x, y, z] == Φ1[ϕ1,1[x] + ϕ1,2[y] + ϕ1,3[z]] + Φ2[ϕ2,1[x] + ϕ2,2[y] + ϕ2,3[z]]
Out[∘]=
True
```

Kolmogorov- Arnold Neural Network

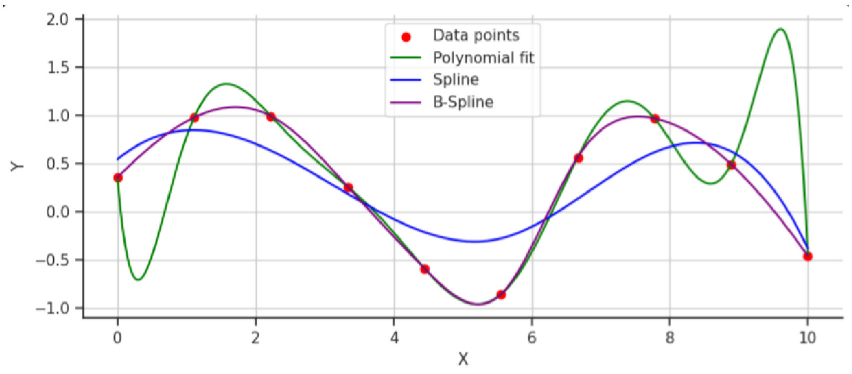
In traditional neural networks, one form of which is called a multilayer perceptron (left), each synapse learns a number called weight, and each neurons applies a simple function to the sum of its inputs. In the new Kolmogorov-Arnold architecture (right) each synapse learns function, and the neurons sum the outputs of those functions



Basis function

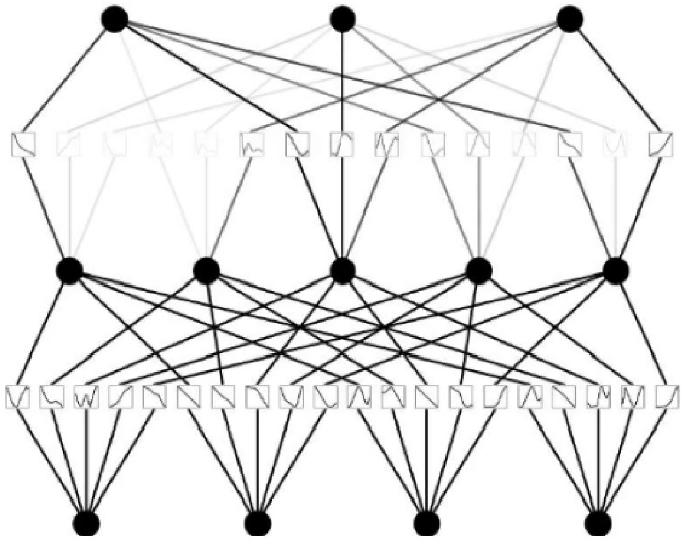
The second ingredient they introduce is the idea of representing the functions in each layer by B-splines. Some of the main claims about KANs are that they can approximate functions more efficiently (using less parameters/memory) than multilayer perceptrons (MLPs), and that they are interpretable.

Instead of using fixed activation functions at each neuron, KANs employ learnable activation functions on the edges themselves. These activation functions are constructed using basis functions, like B-splines.



Why B-Spline?

https://daniel-bethell.co.uk/posts/kan/



Structure of KAN

Implementation

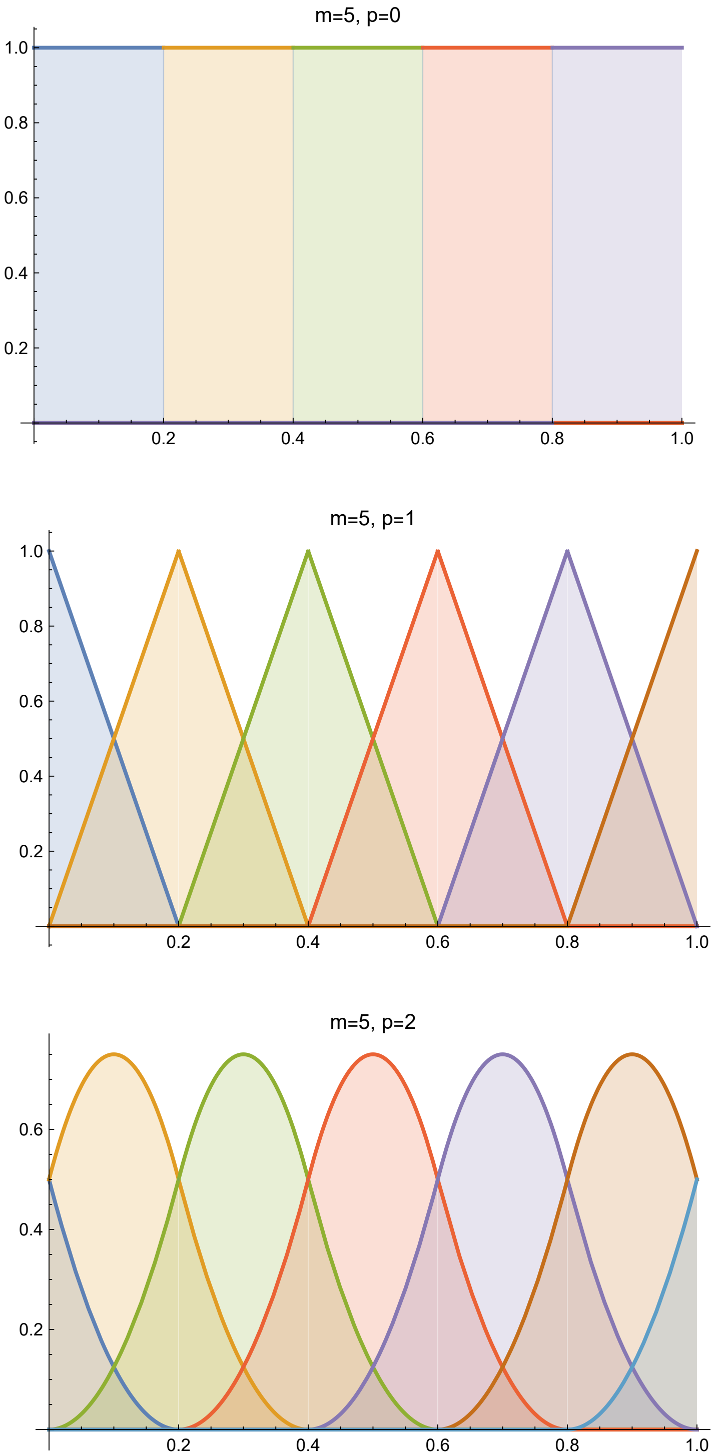
The Wolfram Language has an inbuilt function for B-Splines, however I was not able to use it inside a ElementwiseLayer, and therefore had to implement it myself. Here I define B-spline basis functions over the interval [0,1], with m being the number of intervals, i the index of the basis function ,and p the polynomial order of the B-spline:

```
In[∘]:= B[i_, 0, m_][t_] := Boole[i (1 / m) ≤ t < (i + 1) (1 / m) ]
B[i_, -1, m_][t_] := 0
B[i_, p_, m_][t_] := Simplify[Boole[i (1 / m) ≤ t < (i + 1 + p) (1 / m) ]
   ((t - i / m) / (p / m) B[i, p - 1, m][t] + ((i + 1 + p) / m - t) / (p / m) B[i + 1, p - 1, m][t])]
```

Here are examples of how the basis functions look for polynomial orders 0-2. Note that the sum of the basis functions are 1 over the entire domain .

```
In[∘]:= m = 5;
Table[B[i, 0, m][t], {i, 0, m - 1}];
Table[B[i, 1, m][t], {i, -1, m - 1}];
Table[B[i, 2, m][t], {i, -2, m - 1}];
Row[{
  Plot[%%%, {t, 0, 1}, PlotRange → All, ImageSize → Medium,
   PlotLabel → "m=" <> ToString[m] <> ", p=0", Filling → Axis],
  Plot[%%, {t, 0, 1}, PlotRange → All, ImageSize → Medium,
   PlotLabel → "m=" <> ToString[m] <> ", p=1", Filling → Axis],
  Plot[%, {t, 0, 1}, PlotRange → All, ImageSize → Medium,
   PlotLabel → "m=" <> ToString[m] <> ", p=2", Filling → Axis]
 }]
```

Out[∘]=



Note that the basis functions are simply shifted functions of each other. Therefore, instead of applying different basis functions to the same input, we may instead apply the same input to shifted versions of the input. Based on this, we can implement a KAN layer as follows:

```
In[•]:= KANlayer[inputs_, outputs_, m_, p_] := NetGraph[
   <|
    "repeat" → ReplicateLayer[{outputs, m + p}],
    "shift" →
     NetArrayLayer["Array" → Table[j / m, {i, outputs}, {j, -p, m - 1}, {k, inputs}]],
    "thread" → ThreadingLayer[#1 - #2 &],
    "spline" →
     ElementwiseLayer@Function[t, Evaluate[Simplify`PWToUnitStep[B[0, p, m][t]]]],
    "alpha" → NetArrayLayer["Array" → 0.1 RandomReal[{-1, 1}, {outputs, m + p, inputs}]],
    "times" → ThreadingLayer[Times],
    "sum" → AggregationLayer[Total, -2 ;; -1]
    |>,
   {NetPort["Input"] → "repeat",
    {"repeat", "shift"} → "thread" → "spline",
    {"spline", "alpha"} → "times" → "sum" → NetPort["Output"]
   },
   "Input" → inputs,
   LearningRateMultipliers → {"shift" → 0, "alpha" → 2}
  ]
```

Here is an example:

```
In[•]:= KANlayer[3, 1, 10, 2]
Out[•]=
NetGraph[ Input port:   vector (size: 3)
          Output port:  vector (size: 1) ]
```

Here are some functions to extract the learnt spline functions in each layer and plot them:

```
In[•]:= getSpline[α_, m_, p_] := Simplify[Sum[α[[i + p + 1]] × B[0, p, m][t - i / m], {i, -p, m - 1}]]
KANsplines[kan_] := Module[{arrays, basefunctions, p, m},
  arrays = NetExtract[kan, {All, "alpha", "Array"}];
  arrays = Map[Normal@Transpose[#, {2, 3, 1}] &, arrays];
  basefunctions = NetExtract[kan, {All, "spline", "Function"}];
  TableForm@Table[
    p = Cases[basefunctions[[i]], Power[_, n_] :> n, {0, Infinity}][[1]];
    m = Dimensions[arrays[[i]]][[-1]] - p;
    Row[{"Layer " <> ToString[i], MatrixForm@Map[Plot[getSpline[#, m, p],
           {t, 0, 1}, Frame → True, FrameTicks → None] &, arrays[[i]], {2}]}]
    , {i, Length[arrays]}]
 ]
```

Examples

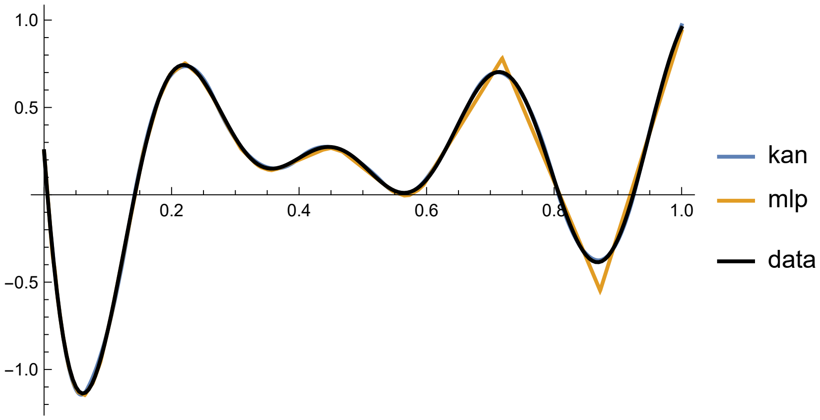
1D function fitting

In[ ]:=
```
x = Table[{i}, {i, 0, 1, 0.005}];
y = (1 - Sin[20 x] (x^2 - 5 x + 1 / (x + 0.1))) / 4;

kan = NetTrain[NetChain[{KANlayer[1, 1, 15, 2]}], x → y]
mlp = NetTrain[NetChain[{50, Ramp, 50, Ramp, 1}], x → y];

Show[{
  Plot[{kan[{x}], mlp[{x}]}, {x, 0, 1}, PlotLegends → {"kan", "mlp"}],
  ListLinePlot[Join[x, y, 2], PlotStyle → Black, PlotLegends → {"data"}]
 }]
```

Out[ ]=

NetChain[Input port: vector (size: 1) Output port: vector (size: 1)]

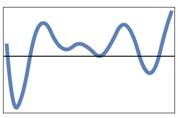
Out[ ]=



Note how the MLP struggles to learn the function.
Note also that for this 1D case, the final output is simply the single spline learnt by the KAN.

In[ ]:=
```
KANsplines[kan]
```

Out[ ]//TableForm=

Layer 1 ()

In[∘]:=
```
TableForm[{{"MLP", "KAN"}, {Information[mlp], Information[kan]}}]
```
Out[∘]//TableForm=

MLP

Net Information	
Layers Count	5
Arrays Count	6
Shared Arrays Count	0
Input Port Names	{Input}
Output Port Names	{Output}
Arrays Total Element Count	2701
Arrays Total Size	10.804 kB

KAN

Net Information	
Layers Count	7
Arrays Count	2
Shared Arrays Count	0
Input Port Names	{Input}
Output Port Names	{Output}
Arrays Total Element Count	34
Arrays Total Size	136 B

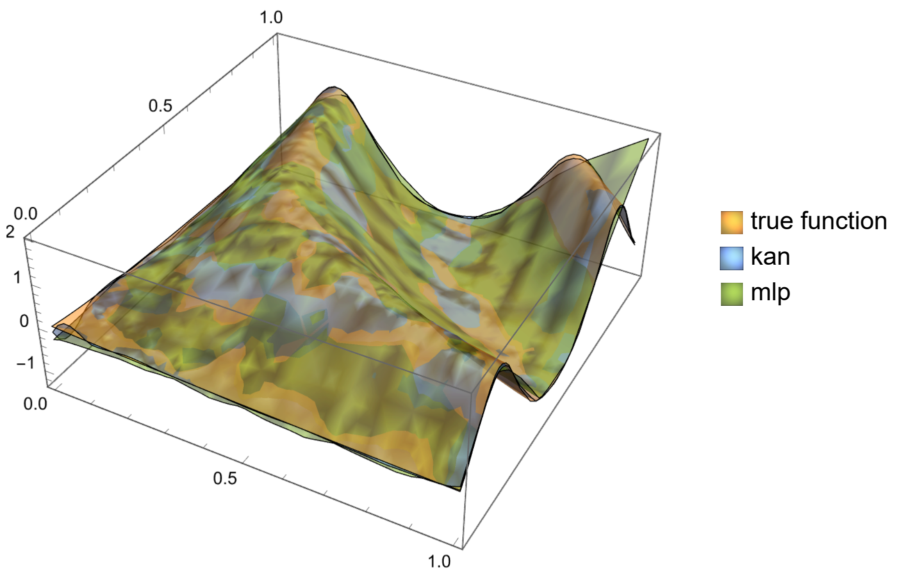
2D function fitting

In[∘]:=
```
f[x1_, x2_] = Exp[Sin[3 Pi x1 x2]] + 1 / Sqrt[(x1 + 3 x2 + 1)] - 2;
x = RandomReal[{0, 1}, {1000, 2}];
y = Map[{f @@ #} &, x];
```

In[∘]:=
```
Short[x]
```
Out[∘]//Short=
```
{{0.533683, <<20>>}, <<998>>, {<<20>>, <<20>>}}
```

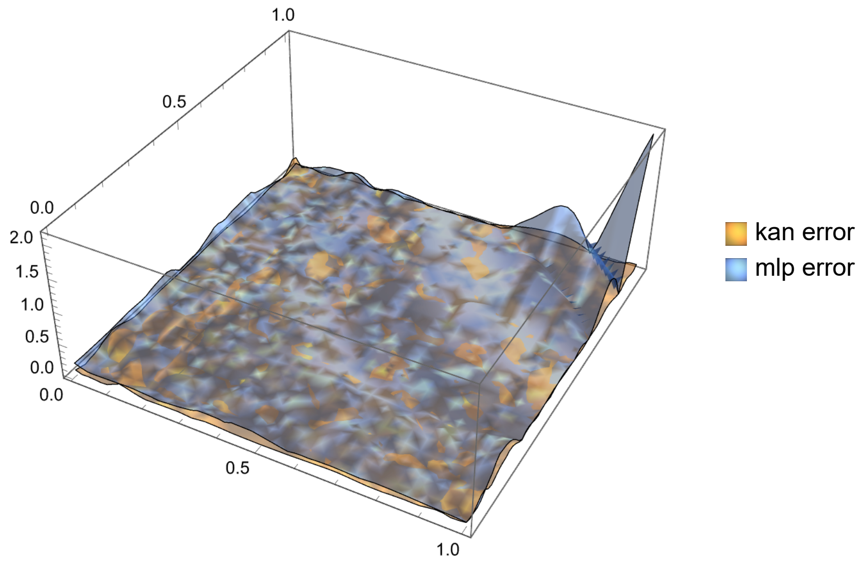
In[∘]:=
```
Short[y]
```
Out[∘]//Short=
```
{{0.646693}, <<998>>, {<<19>>}}
```

```
In[•]:= f[x1_, x2_] = Exp[Sin[3 Pi x1 x2]] + 1 / Sqrt[(x1 + 3 x2 + 1)] - 2;
        x = RandomReal[{0, 1}, {1000, 2}];
        y = Map[{f @@ #} &, x];

        kan = NetTrain[NetChain[{KANlayer[2, 5, 10, 2], KANlayer[5, 1, 10, 2]}], x → y];
        mlp = NetTrain[NetChain[{100, Ramp, 1}], x → y];

        Plot3D[{f[x1, x2], kan[{x1, x2}], mlp[{x1, x2}]}, {x1, 0, 1}, {x2, 0, 1}, Mesh → None,
         PlotStyle → Opacity[0.5], PlotLegends → {"true function", "kan", "mlp"}]
Out[•]=
```



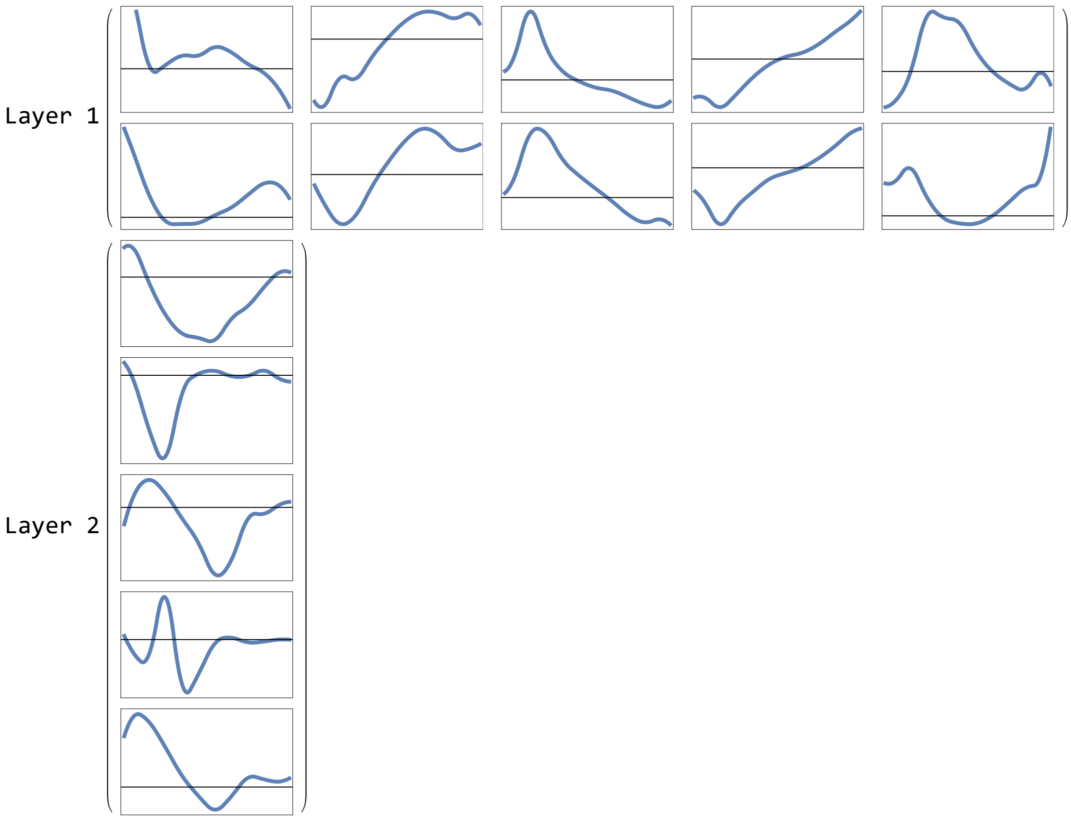
In[•]:=
```
Plot3D[{Abs[kan[{x1, x2}] - f[x1, x2]], Abs[mlp[{x1, x2}] - f[x1, x2]]},
 {x1, 0, 1}, {x2, 0, 1}, Mesh → None, PlotStyle → Opacity[0.5],
 PlotLegends → {"kan error", "mlp error"}, PlotRange → All]
```

Out[•]=



In[•]:=
```
KANsplines[kan]
```

Out[•]//TableForm=



In[∘]:=
```
TableForm[{{"MLP", "KAN"}, {Information[mlp], Information[kan]}}]
```

Out[∘]//TableForm=

MLP

Net Information	
Layers Count	3
Arrays Count	4
Shared Arrays Count	0
Input Port Names	{Input}
Output Port Names	{Output}
Arrays Total Element Count	401
Arrays Total Size	1.604 kB

KAN

Net Information	
Layers Count	14
Arrays Count	4
Shared Arrays Count	0
Input Port Names	{Input}
Output Port Names	{Output}
Arrays Total Element Count	360
Arrays Total Size	1.44 kB

Classification

https://www.kaggle.com/datasets/jeffheaton/iris-computer-vision?select=iris-setosa

In[∘]:=
```
iris = ResourceData[ResourceObject["Sample Data: Fisher's Irises"]];
```

```
In[∘]:= Take[iris, 55]
Out[∘]=
```

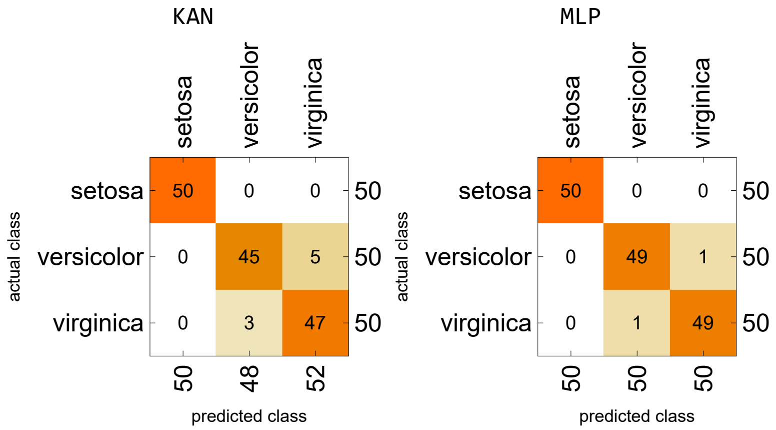
Species	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	5.7 cm	4.4 cm	1.5 cm	0.4 cm
setosa	5.4 cm	3.9 cm	1.3 cm	0.4 cm
setosa	5.1 cm	3.5 cm	1.4 cm	0.3 cm
setosa	5.7 cm	3.8 cm	1.7 cm	0.3 cm
setosa	5.1 cm	3.8 cm	1.5 cm	0.3 cm
setosa	5.4 cm	3.4 cm	1.7 cm	0.2 cm
setosa	5.1 cm	3.7 cm	1.5 cm	0.4 cm
setosa	4.6 cm	3.6 cm	1. cm	0.2 cm
setosa	5.1 cm	3.3 cm	1.7 cm	0.5 cm
setosa	4.8 cm	3.4 cm	1.9 cm	0.2 cm
setosa	5. cm	3. cm	1.6 cm	0.2 cm
setosa	5. cm	3.4 cm	1.6 cm	0.4 cm
setosa	5.2 cm	3.5 cm	1.5 cm	0.2 cm
setosa	5.2 cm	3.4 cm	1.4 cm	0.2 cm
setosa	4.7 cm	3.2 cm	1.6 cm	0.2 cm
setosa	4.8 cm	3.1 cm	1.6 cm	0.2 cm
setosa	5.4 cm	3.4 cm	1.5 cm	0.4 cm
setosa	5.2 cm	4.1 cm	1.5 cm	0.1 cm
setosa	5.5 cm	4.2 cm	1.4 cm	0.2 cm
setosa	4.9 cm	3.1 cm	1.5 cm	0.2 cm

rows 16–35 of **55**

```
In[∘]:=
x = Map[QuantityMagnitude, Normal[Values[iris[All, 2 ;; -1]]], {2}];
dec = NetDecoder[{"Class", {"setosa", "versicolor", "virginica"}}];
y = Normal[iris[All, 1]];

In[∘]:= kan = NetTrain[NetChain[{KANlayer[4, 5, 5, 3],
          KANlayer[5, 3, 5, 3], SoftmaxLayer[]}, "Output" → dec], x → y];
mlp = NetTrain[NetChain[{5, Ramp, 3, SoftmaxLayer[]}, "Output" → dec], x → y];
```

```
In[]:= Grid[{{"KAN", "MLP"}, {
    ClassifierMeasurements[kan, Thread[x → y]]["ConfusionMatrixPlot"],
    ClassifierMeasurements[mlp, Thread[x → y]]["ConfusionMatrixPlot"]
   }}]
```

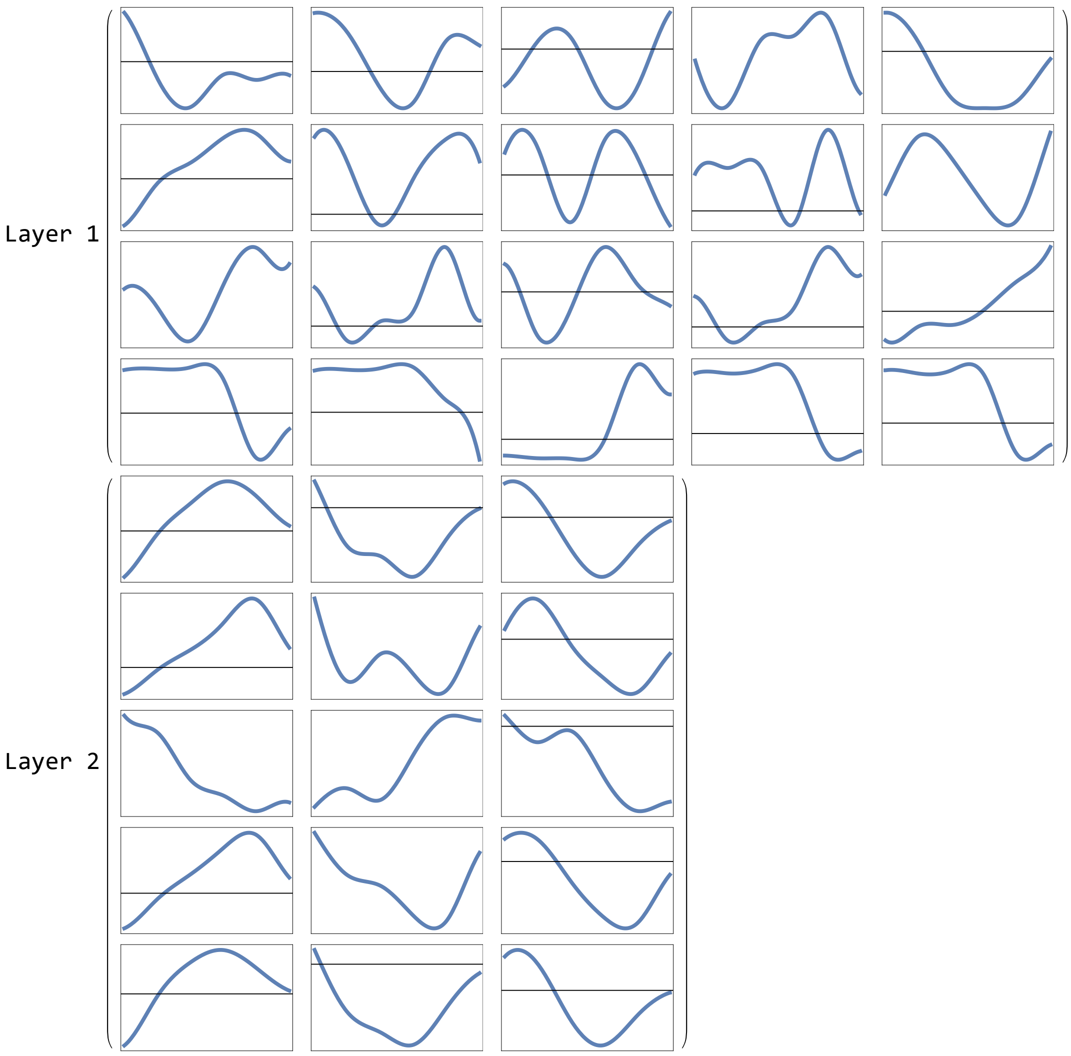
Out[]=



Note that the MLP is better than the KAN here, but the KAN is not “extremely bad”.

```
In[]:= KANsplines[NetTake[kan, {1, 2}]]
```

Out[]//TableForm=



In[∘]:=
```
TableForm[{{"MLP", "KAN"}, {Information[mlp], Information[kan]}}]
```

Out[∘]//TableForm=

MLP		KAN	
Net Information		Net Information	
Layers Count	4	Layers Count	15
Arrays Count	4	Arrays Count	4
Shared Arrays Count	0	Shared Arrays Count	0
Input Port Names	{Input}	Input Port Names	{Input}
Output Port Names	{Output}	Output Port Names	{Output}
Arrays Total Element Count	43	Arrays Total Element Count	560
Arrays Total Size	172 B	Arrays Total Size	2.24 kB

Time Series Forecasting

In[∘]:=
```
zz = Import["/Volumes/MyUSB/MackeyGlass_t17.txt", "Data"];
```

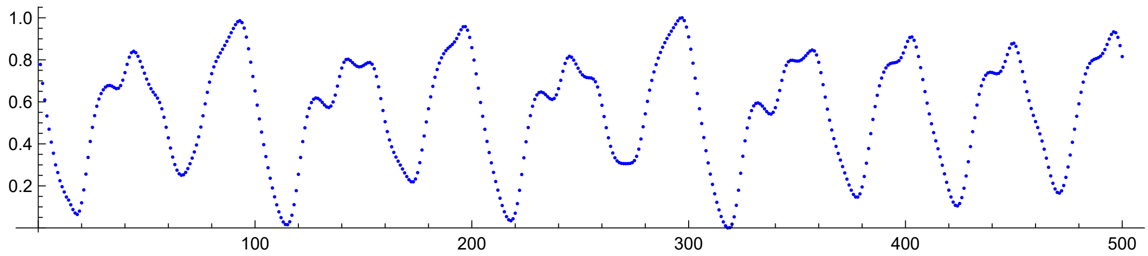
In[∘]:=
```
val = Take[zz // Flatten, 500];
```

In[∘]:= $\alpha = \frac{1}{\text{Max[val]} - \text{Min[val]}}; \beta = -\alpha\,\text{Min[val]};$

In[∘]:=
```
vaL = Map[(α # + β) + 0 RandomReal[{-0.05, 0.05}] &, val];
```

In[∘]:=
```
pipa0 = ListPlot[vaL, AspectRatio → 0.2, PlotStyle → {Blue, PointSize → 0.003}]
```

Out[∘]=



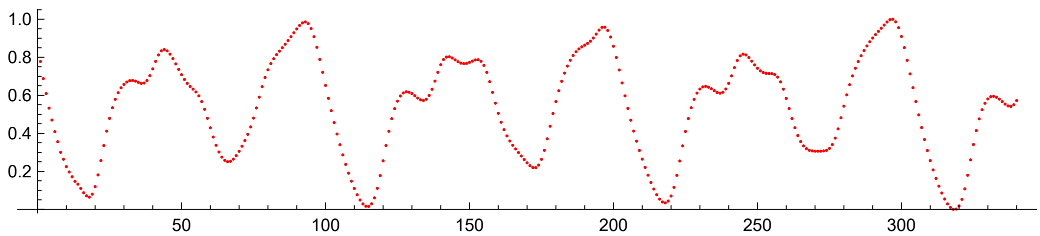
The total time series

Learning process

In[∘]:=
```
zz = Take[vaL, 340];
```

In[∘]:=
```
pipa01 = ListPlot[zz, AspectRatio → 0.2, PlotStyle → {Red, PointSize → 0.003}]
```

Out[∘]=



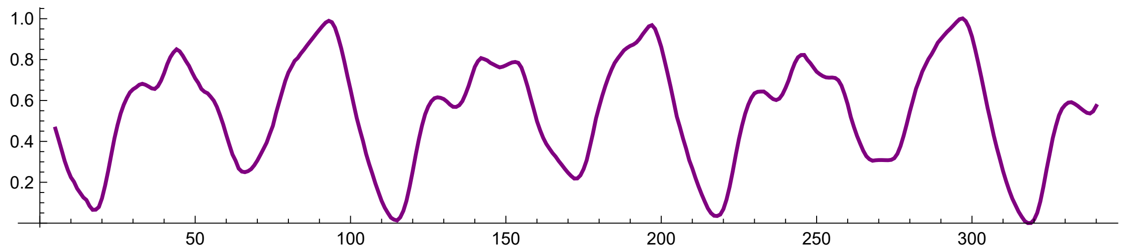
The learning set 0-340

In[∘]:=
```
x = Range[336]; y = Range[336];
```

In[∘]:=
```
Do[x[[i]] = {zz[[i]], zz[[i + 1]], zz[[i]] + 2, zz[[i + 3]]}, {i, 1, 336}]
```

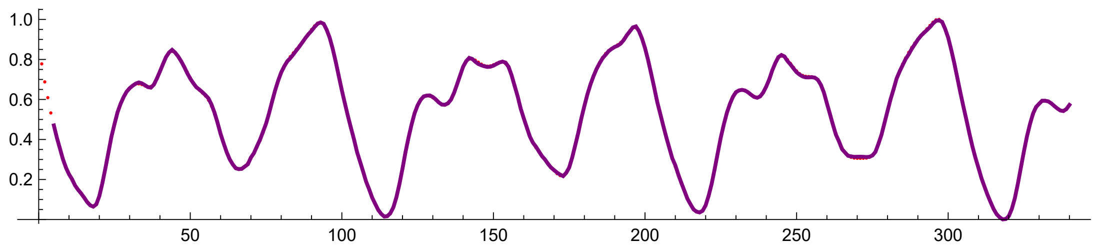
In[∘]:=
```
Do[y[[i]] = {zz[[i + 4]]}, {i, 1, 336}]
```

```
In[∘]:= Length[y]
Out[∘]=
        336

In[∘]:= y⟦10⟧
Out[∘]=
        {0.132355}

In[∘]:= x⟦10⟧
Out[∘]=
        {0.224668, 0.196172, 2.22467, 0.147083}

In[∘]:= s = 9;

In[∘]:= {vaL⟦1 + s⟧, vaL⟦2 + s⟧, vaL⟦3 + s⟧, vaL⟦4 + s⟧, vaL⟦5 + s⟧}
Out[∘]=
        {0.224668, 0.196172, 0.171077, 0.147083, 0.132355}

In[∘]:= kan = NetTrain[NetChain[{KANlayer[4, 5, 10, 2], KANlayer[5, 1, 10, 2]}], x → y];

In[∘]:= kan[x⟦s + 1⟧]
Out[∘]=
        {0.126176}

In[∘]:= KANY = Map[({# + 5, kan[x⟦# + 1⟧]} // Flatten) &, Range[0, 335]];

In[∘]:= KANY⟦10⟧
Out[∘]=
        {14, 0.126176}

In[∘]:= pipa1 = ListPlot[KANY, PlotStyle → Purple, AspectRatio → 0.2, Joined → True]
Out[∘]=
```



The learning set time series reproduced by KAN

```
In[∘]:= Show[pipa01, pipa1]
Out[∘]=
```



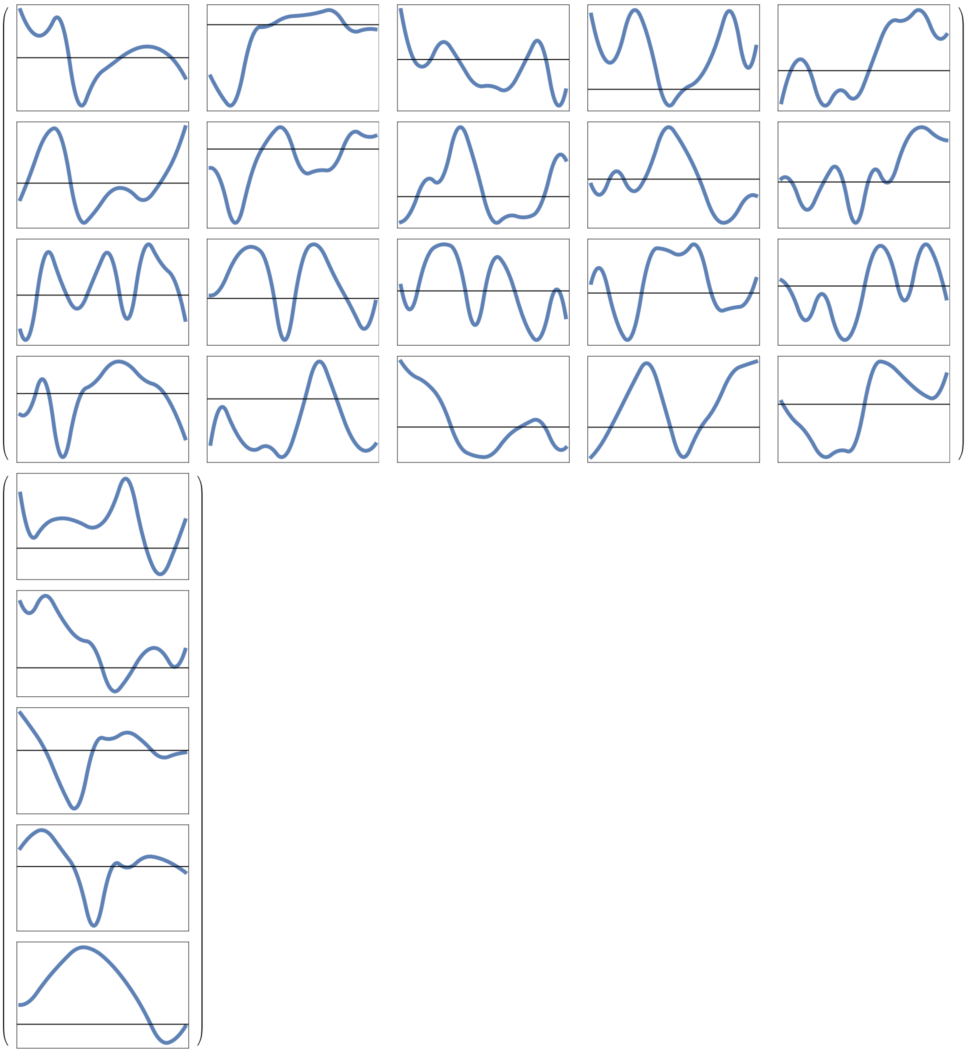
The learning set time series and its reproduction by KAN

In[∘]:= **KANsplines[kan]**

Out[∘]//TableForm=

Layer 1

Layer 2



In[∘]:= **Information[kan]**

Out[∘]=

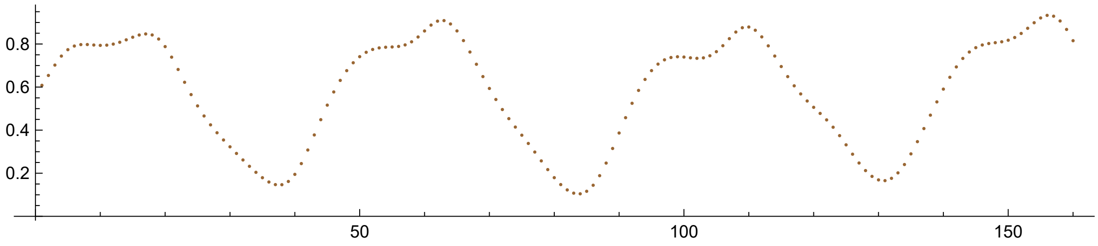
Net Information	
Layers Count	14
Arrays Count	4
Shared Arrays Count	0
Input Port Names	{Input}
Output Port Names	{Output}
Arrays Total Element Count	600
Arrays Total Size	2.4 kB

Testing process

In[∘]:= **zz = Take[vaL, {341, 500}];**

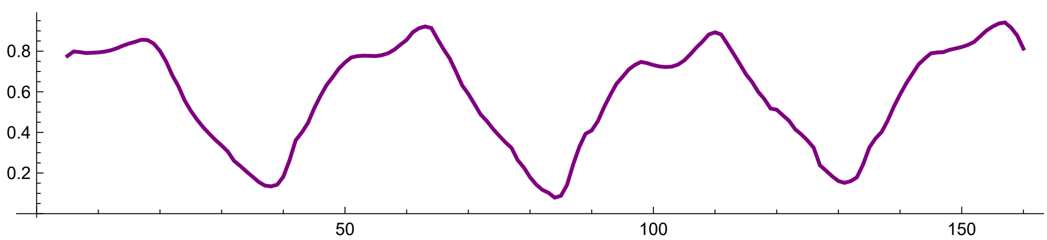
In[◦]:= pipa02 = ListPlot[zz, AspectRatio → 0.2, PlotStyle → {Brown, PointSize → 0.003}]

Out[◦]=



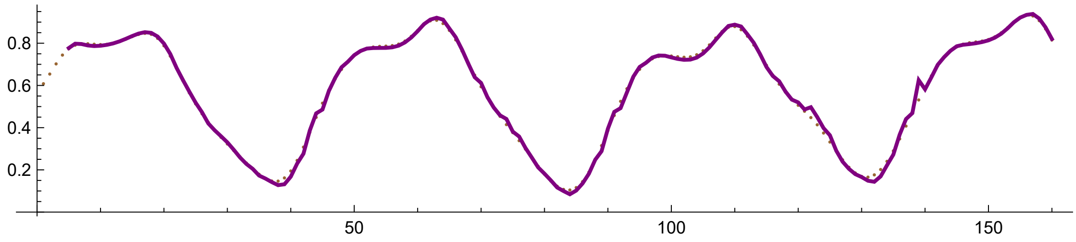
The testing set 340-500

```
In[◦]:= x = Range[156]; y = Range[156];

In[◦]:= Do[x⟦i⟧ = {zz⟦i⟧, zz⟦i + 1⟧, zz⟦i⟧ + 2, zz⟦i + 3⟧}, {i, 1, 156}]

In[◦]:= Do[y⟦i⟧ = {zz⟦i + 4⟧}, {i, 1, 156}]

In[◦]:= Length[y]

Out[◦]=
156

In[◦]:= y⟦10⟧

Out[◦]=
{0.819421}

In[◦]:= x⟦10⟧

Out[◦]=
{0.793647, 0.794709, 2.79365, 0.80814}

In[◦]:= s = 9;

In[◦]:= {vaL⟦341 + s⟧, vaL⟦342 + s⟧, vaL⟦343 + s⟧, vaL⟦344 + s⟧, vaL⟦345 + s⟧}

Out[◦]=
{0.793647, 0.794709, 0.799553, 0.80814, 0.819421}

In[◦]:= kan[x⟦s + 1⟧]

Out[◦]=
{0.826943}

In[◦]:= KANY = Map[({# + 5, kan[x⟦# + 1⟧]} // Flatten) &, Range[0, 155]];

In[◦]:= KANY⟦10⟧

Out[◦]=
{14, 0.826943}

In[◦]:= pipa2 = ListPlot[KANY, PlotStyle → Purple, AspectRatio → 0.2, Joined → True]

Out[◦]=
```



The testing set time series reproduced by KAN

```
In[]:= Show[pipa02, pipa2]
Out[]=
```



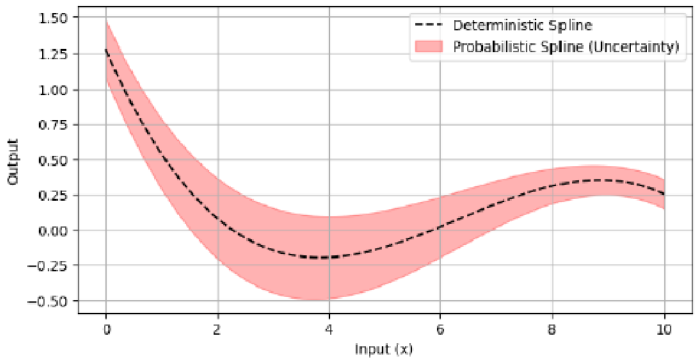
The testing set time series and its reproduction by KAN

Variants

KAN with Wavelets

KAN with Radial Bases

Bayesian KAN



Deterministic vs Probabilistic Spline Functions.

Python

pykan

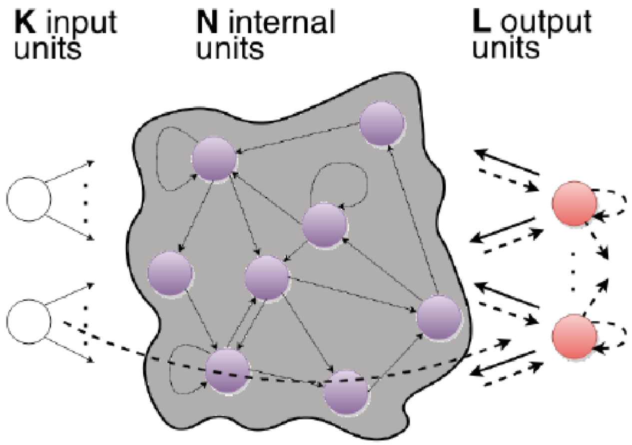
Further New Directions

Reservoir Neural Network

Liquid Neural Network

Echo State Neural Network (ESN)

Common Feature



General structure of an ESN, where dashed arrows represents possible optional connections.

Sources

https://arxiv.org/abs/2404.19756

Kolmogorov-Arnold networks (KANs) in Wolfram language
by Andreas Hafver
Wolfram Community, STAFF PICKS, August 23, 2024
https://community.wolfram.com/groups/-/m/t/3254225