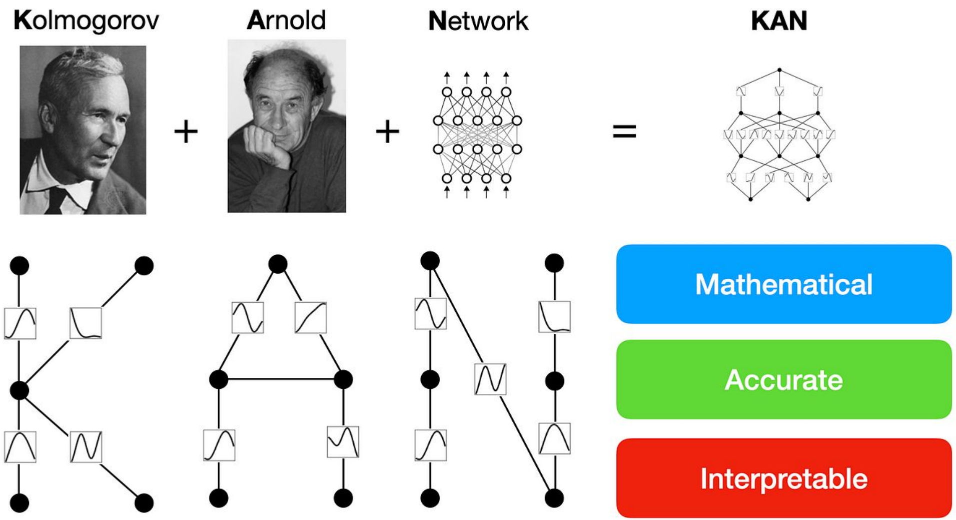


Kolmogorov Arnold Networks

Nidhi, Tejasvini, Vinil, Yash

Introduction

- We explore a new neural network design proposed by [1] which is inspired by the **Kolmogorov-Arnold Representation Theorem.**
- **Core concept:** Replace traditional edge weights and fixed activation functions on nodes with **learnable edge functions** and **simple summation on nodes.**
- **Benefits:**
  - **Improved accuracy:** Models complex functions with smaller networks.
  - **Better interpretability:** Clearer understanding of model behavior.

[1] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark, "KAN: Kolmogorov-Arnold Networks," arXiv, 2024. [Online]. Available: https://arxiv.org/abs/2404.19756

Gaps in NN Design and Why KAN can fill it?

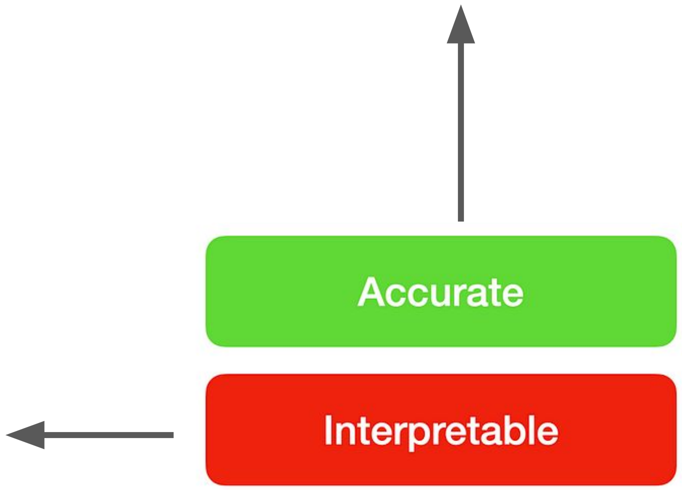
Challenges with Multi-layer perceptron(MLP) based Neural Networks(NNs):

- MLP–NNs → Blackbox: we don’t know how model made decision
- MLP–NNs → Huge Number of Parameters for deep models

So, **Gaps** in NN design: **less interpretability** and **more training parameters**

We need answer:	We need AI to be:
Why AI made certain medical decisions?	Trustable and Transparent AI
Does AI have racial/gender bias?	Ethical and Fair AI
Why did the AI approve/reject credit applications?	Regulation Compliant AI
How did the AI make the investment decision?	Reliable AI
Why did the AI driven autonomous-car brake?	Safe AI

We also want less training parameters

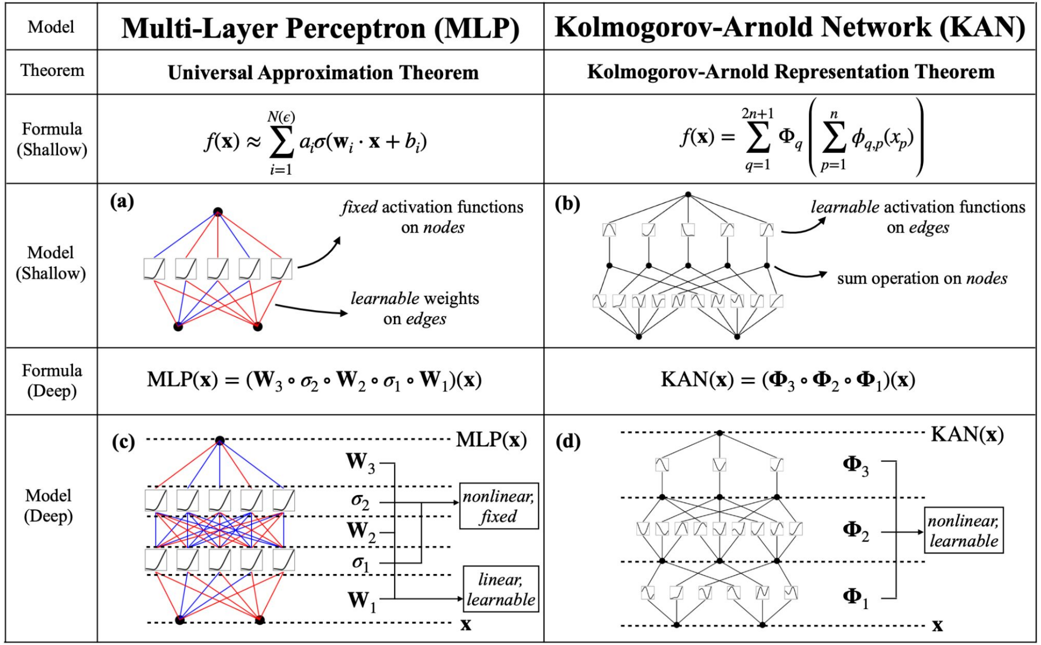


Objectives of our work

1. Study and analyse KANs
2. Study KAN variations like KAN-CNN
3. Build simple KAN models from pykan library

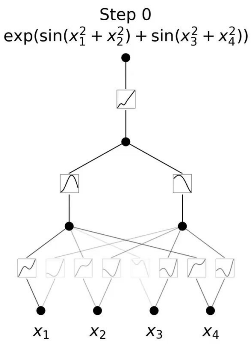
4. ***Case Study on application of CNN-KAN on MedMnist Dataset(PathMNIST)***

KAN vs MLP Architecture



Key concepts

1. Function Approximation Theorems
2. Bezier Curves
3. B-Splines
4. Training KANs



[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

Universal Approximation Theorem for MLPs

In simple words, it states that a neural network with just one hidden layer containing a finite number of neurons can approximate **ANY** continuous function to a reasonable accuracy on a compact subset of $\mathbb{R}^n$, given suitable activation functions.

Mathematically speaking, for any continuous function f and $\epsilon > 0$, there always exists a neural network $\hat{f}$ such that:

$$|f(x) - \hat{f}(x)| < \epsilon$$

Kolmogorov Arnold Representation Theorem for KANs

More formally, the Kolmogorov-Arnold representation theorem asserts that any multivariate continuous function can be represented as the composition of a **finite** number of continuous functions of a single variable.

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

$$y = F(x_1, x_2, x_3, \cdots, x_n)$$

Univariate functions

$$\phi_1(x_1) + \phi_2(x_2) + \phi_3(x_3) + \cdots + \phi_n(x_n)$$

$$\boxed{\psi}\left(\sum_{i=1}^{n} \phi_i(x_i) \right)$$

$$\sum_{j=1}^{m} \psi_j\left(\sum_{i=1}^{n} \phi_{ij}(x_i)\right)$$

Multivariate continuous function

Composition of univariate functions

$$F(x_1, x_2, x_3, \cdots, x_n) = \sum_{j=1}^{m} \psi_j\left(\sum_{i=1}^{n} \phi_{ij}(x_i)\right)$$

Multivariate continuous function

Composition of univariate functions

$$\begin{aligned} F(x_1, x_2, x_3, \cdots, x_n) = & \ \psi_1(\phi_{11}(x_1) + \phi_{21}(x_2) + \cdots + \phi_{n1}(x_n)) \\ & + \psi_2(\phi_{12}(x_1) + \phi_{22}(x_2) + \cdots + \phi_{n2}(x_n)) \\ & \vdots \\ & + \psi_m(\phi_{1m}(x_1) + \phi_{2m}(x_2) + \cdots + \phi_{nm}(x_n)) \end{aligned}$$

Here's a simple toy example:

Multivariate continuous function

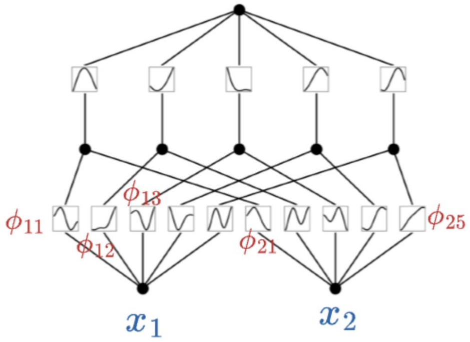
$$F(x, y) = xy$$

Composition of univariate functions

$$F(x, y) = \underbrace{\boxed{exp}}_{\psi}\left(\underbrace{\boxed{log(x)}}_{\phi_x} + \underbrace{\boxed{log(y)}}_{\phi_y}\right)$$

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

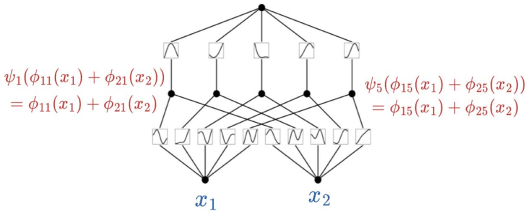
At first:



- The input x_1 is passed through univariate functions $(\phi_{11}, \phi_{12}, \cdots, \phi_{15})$ to get $(\phi_{11}(x_1), \phi_{12}(x_1), \cdots, \phi_{15}(x_1))$.
- The input x_2 is passed through univariate functions $(\phi_{21}, \phi_{22}, \cdots, \phi_{25})$ to get $(\phi_{21}(x_2), \phi_{22}(x_2), \cdots, \phi_{25}(x_2))$.

Next, the two corresponding outputs are aggregated (summed), so the ψ function in this case is just the identity operation ($\psi(z) = z$):

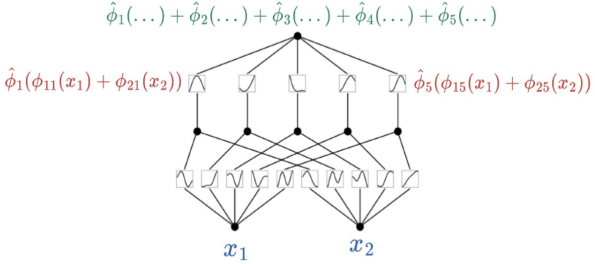
Next, the two corresponding outputs are aggregated (summed), so the ψ function in this case is just the identity operation ($\psi(z) = z$):



This forms one KAN layer.

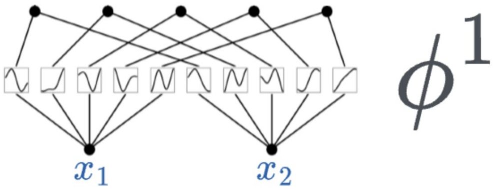
Next, we pass the above output through one more KAN layer.

So the above output is first passed through the ϕ function of the next layer and summed to get the final output:



[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

All we are doing in a single KAN layer is taking the input $(x_1, x_2, \cdots, x_n)$ and applying a transformation ϕ to it.



ϕ^1 denotes the transformation in the first layer

Thus, the transformation matrix ϕ^1 (corresponding to the first layer) can be represented as follows:

$$\phi^1 = \begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{12} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix}$$

In the above matrix:

- n denotes the number of inputs.
- m denotes the number of output nodes in that layer.
- **[IMPORTANT] The individual entries are not numbers, they are univariate functions.** For instance:
  - ϕ_{11} could be $2x^2 - 3x + 4$.
  - ϕ_{12} could be $4x^3 + 5x^2 + x - 2$.
  - and so on...

So to generate a transformation, all we have to do is take the input vector and pass it through the corresponding functions in the above transformation matrix:

$$\underbrace{z^1}_{\text{Output of first layer}} = \overbrace{\begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{12} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix}}^{\text{Transformation matrix } \phi^1} * \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input } X}$$

This will result in the following vector:

$$z^1 = \begin{bmatrix} \overbrace{\boxed{\phi_{11}(x_1)}}^{\text{Function applied to } x_1} + \phi_{12}(x_2) + \cdots + \overbrace{\boxed{\phi_{1n}(x_n)}}^{\text{Function applied to } x_n} \\ \phi_{21}(x_1) + \phi_{22}(x_2) + \cdots + \phi_{2n}(x_n) \\ \vdots \\ \phi_{m1}(x_1) + \phi_{m2}(x_2) + \cdots + \phi_{mn}(x_n) \end{bmatrix}$$

The above is the output of the first layer, which is then passed through the next layer for another function transformation.

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

Thus, the entire KAN network can be condensed into one formula as follows:

$$KAN(x) = \phi^L \left(\phi^{L-1} \left(\ \ldots \ \left(\phi^2 \left(\phi^1(x) \right) \right) \right) \right)$$

Where:

- x denotes the input vector
- ϕ^k denotes the function transformation matrix of layer k.
- $KAN(x)$ is the output of the KAN network.

The above formulation can appear quite similar to what we do in neural networks:

$$NN(x) = \theta^L \left(\sigma(\theta^{L-1} \left(\ \ldots \ \left(\sigma(\theta^2 \left(\sigma(\theta^1(x)) \right) \right) \right) \right) \right)$$

The only difference is that the parameters θ^i are linear transformations, and σ denotes the activation function used for non-linearity, and it is the same activation function across all layers.

In the case of KANs, the matrices ϕ^k themselves are non-linear transformation matrices, and each univariate function can be quite different.

$2x^2 - 3x + 4$

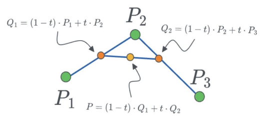
$$\phi^1 = \begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{12} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix}$$

$4x^3 + 5x^2 + x - 2$

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

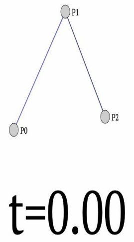
Bezier curves and B-Splines

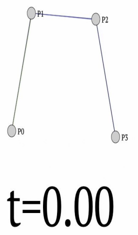
The final equation for the position of point P is given by:

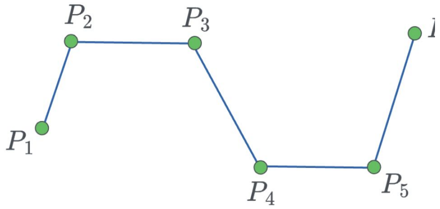


$$\boxed{B(t) = (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3,\ t\in[0,1]}$$

To obtain this path, substitute all values in terms of P_1, P_2, and P_3.





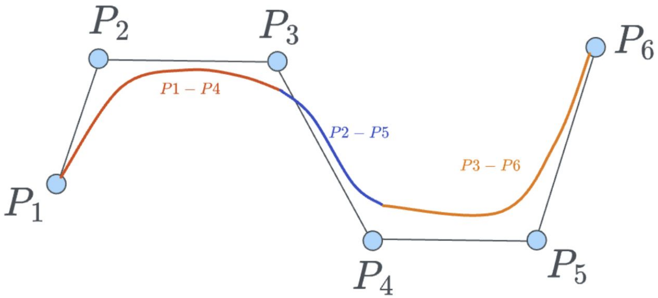


Given that we have 6 points, we can generate a bezier curve of degree 5. That's always an option. However, as discussed above, this is still computationally expensive and not desired.

Instead, we can create curves of smaller degrees (say, 3), and then conne them.

For instance, a full B-spline can be created as follows:

- Some part of it can come from a curve of degree 3 from points (P_1, P_2, P_3, P_4)
- Some part of it can come from a curve of degree 3 from points (P_2, P_3, P_4, P_5)
- Some part of it can come from a curve of degree 3 from points (P_3, P_4, P_5, P_6)



Note: In this diagram, the individual Bezier curves don't appear to be connected that well, but in reality, the final curve is smooth.

When we have n control pints (6 in the diagram above), and we create k degree polynomial Bezier curves, we get $(n-k)$ Bezier curves in the final Bsplines.

Similar to what we saw in Bezier curves, the final B-spline curve is represented as a linear combination of the points P_i:

$$S(t) = \sum_{i=0}^{n} \boxed{P_i} \cdot \boxed{N_{i,k}}$$

control points

Basis function

- P_i: Control points that define the shape of the curve.
- $N_{i,k}(t)$: B-spline basis functions of degree k associated with each control point P_i, and they are similar to what we saw earlier in the case of Bezier curves and are **fixed**.

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

Training KANs

So here's the core idea of KANs:

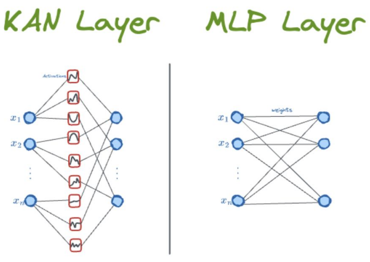
Let's make the positions of control points learnable in the activation function so that the model is free to learn any arbitrary shape activation function that fits the data best.

[2] A. Dasgupta, "A beginner-friendly introduction to Kolmogorov-Arnold Networks (KAN)," Daily Dose of Data Science, 2023. [Online]. Available: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/#:~:text=In%20simple%20words%2C%20it%20states,Network%20Structure%3A%20Single%20hidden%20layer.

KAN vs MLP: Parameter Count, Performance, Interpretability, Run-time and drawbacks

Parameter Count

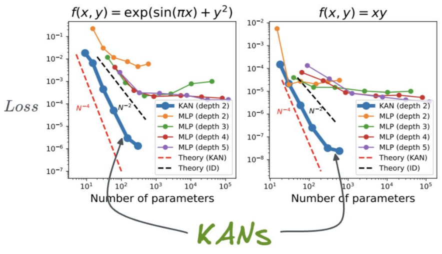
- The final parameter count comes out to be as follows (including all layers):



$\sim N^2L(G+k)$ $\quad$ N^2L

While MLPs appear to be more efficient than KANs, a point to note is that based on their experiments, KANs usually don't require as much large N as MLPs do. This saves parameters while also achieving better generalization.

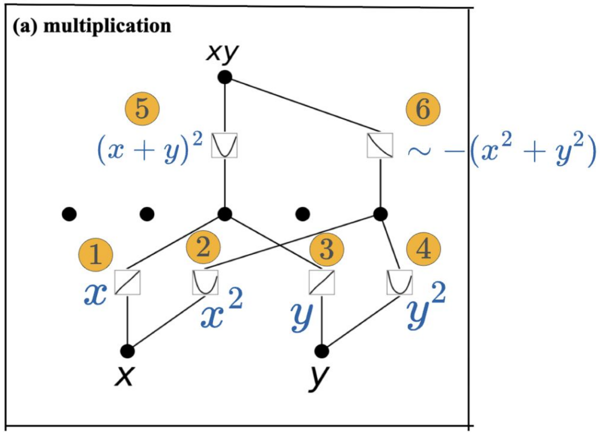
Performance



- In both plots,
  - KANs consistently outperform MLPs, achieving significantly lower test loss across a range of parameter, and at much lower network depth (number of layers).
  - KANs demonstrate superior efficiency, with steeper declines in loss, particularly noticeable with fewer parameters.
  - MLP's performance almost stagnates with increasing the number of parameters.
- The theoretical lines, N^{-4} for KAN and N^{-2} for ideal models (ID), show that KANs closely follow their expected theoretical performance.

Interpretability

For instance, consider the KAN network below, which learns $f(x, y) = xy$.



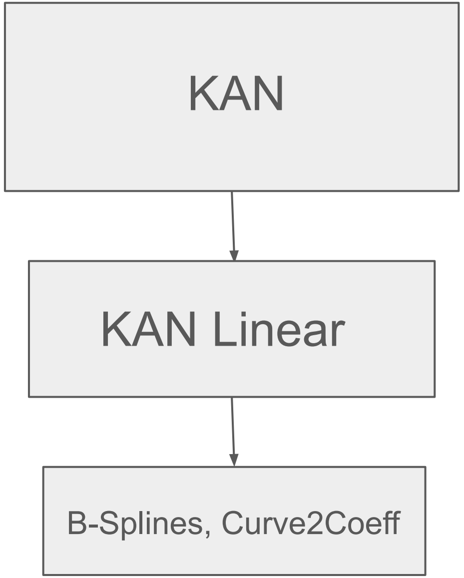
Biggest Drawback of KANs → Long and slow training

KAN Code

```python
class KAN(torch.nn.Module):
    def __init__(
        self,
        layers_hidden,
        grid_size=20, #was 5
        spline_order=5, #Was 3
        scale_noise=0.1,
        scale_base=1.0,
        scale_spline=1.0,
        base_activation=torch.nn.SiLU,
        grid_eps=0.03,
        grid_range=[-4, 4], #Was -1,1
    ):
        super(KAN, self).__init__()
        self.grid_size = grid_size
        self.spline_order = spline_order

        self.layers = torch.nn.ModuleList()
        for in_features, out_features in zip(layers_hidden, layers_hidden[1:]):
            self.layers.append(
                KANLinear(
                    in_features,
                    out_features,
                    grid_size=grid_size,
                    spline_order=spline_order,
                    scale_noise=scale_noise,
                    scale_base=scale_base,
                    scale_spline=scale_spline,
                    base_activation=base_activation,
                    grid_eps=grid_eps,
                    grid_range=grid_range,
                )
            )

    def forward(self, x: torch.Tensor, update_grid=False):
        for layer in self.layers:
            if update_grid:
                layer.update_grid(x)
            x = layer(x)
        return x

    def regularization_loss(self, regularize_activation=1.0, regularize_entropy=1.0):
        return sum(
            layer.regularization_loss(regularize_activation, regularize_entropy)
            for layer in self.layers
        )
```

```python
class KANLinear(torch.nn.Module):
    def __init__(
        self,
        in_features,
        out_features,
        grid_size=5,
        spline_order=3,
        scale_noise=0.1,
        scale_base=1.0,
        scale_spline=1.0,
        enable_standalone_scale_spline=True,
        base_activation=torch.nn.SiLU,
        grid_eps=0.02,
        grid_range=[-1, 1],
    ):
        super(KANLinear, self).__init__()
        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)
            .contiguous()
        )
        self.register_buffer("grid", grid)

        self.base_weight = torch.nn.Parameter(torch.Tensor(out_features, in_features))
        self.spline_weight = torch.nn.Parameter(
            torch.Tensor(out_features, in_features, grid_size + spline_order)
        )
        if enable_standalone_scale_spline:
            self.spline_scaler = torch.nn.Parameter(
                torch.Tensor(out_features, in_features)
            )

        self.scale_noise = scale_noise
        self.scale_base = scale_base
        self.scale_spline = scale_spline
        self.enable_standalone_scale_spline = enable_standalone_scale_spline
        self.base_activation = base_activation()
        self.grid_eps = grid_eps

        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.kaiming_uniform_(self.base_weight, a=math.sqrt(5) * self.scale_base)
        with torch.no_grad():
            noise = (
                (
                    torch.rand(self.grid_size + 1, self.in_features, self.out_features)
                    - 1 / 2
                )
                * self.scale_noise
                / self.grid_size
            )
            self.spline_weight.data.copy_(
                (self.scale_spline if not self.enable_standalone_scale_spline else 1.0)
                * self.curve2coeff(
                    self.grid.T[self.spline_order : -self.spline_order],
                    noise,
                )
            )
            if self.enable_standalone_scale_spline:
                # torch.nn.init.constant_(self.spline_scaler, self.scale_spline)
                torch.nn.init.kaiming_uniform_(self.spline_scaler, a=math.sqrt(5) * self.scale_spline)

    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).

        Returns:
            torch.Tensor: B-spline bases tensor of shape (batch_size, in_features, grid_size + spline_order).
        """
        assert x.dim() == 2 and x.size(1) == self.in_features

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

KAN Code



Create a module list of KANLinear layers
For each input data:

- For each KANLinear layer:
  - If `update_grid` is True, update grid positions
  - Calculate linear part:
    - Apply activation function to input
    - Multiply by base weight
  - Calculate spline part:
    - Calculate B-spline basis functions for input
    - Multiply by scaled spline weight
  - Add linear and spline parts
- Calculate regularization loss

Return final output and regularization loss

MLP:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	200,960
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 10)	1,290

Total params: 235,146 (918.54 KB)
Trainable params: 235,146 (918.54 KB)
Non-trainable params: 0 (0.00 B)

Some Important Hyper-parameters:

```
class KAN(torch.nn.Module):
    def __init__(
        self,
        layers_hidden,
        grid_size=20, #was 5
        spline_order=5, #Was 3
        scale_noise=0.1,
        scale_base=1.0,
        scale_spline=1.0,
        base_activation=torch.nn.SiLU,
        grid_eps=0.03,
        grid_range=[-4, 4], #Was -1,1
    ):
```

```
# Define hyperparameters
batch_size = 64 #Was 64
epochs = 6  # It was 10
lr = 0.001  # It was 0.001
```

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
# Initialize KAN model
model = KAN([784, 256, 128, 10]).to(device)
```

Introduction to MedMNIST

What is MedMNIST?

- A collection of standardized biomedical image datasets for machine learning research.
- Designed for tasks like classification, regression, and segmentation in the medical domain.

Key Features:

- Over **10 preprocessed datasets** for various medical tasks.
- Balanced, small-sized, and low-resolution (28x28 or 32x32 pixels) for accessibility.
- Ideal for lightweight ML models and quick prototyping.

Applications:

- Medical image analysis for pathology, dermatology, ophthalmology, and more.

MedMNIST Subsets Used

Table 1: Overview of Selected Datasets in MedMNIST2D

Dataset	Data Modality	Tasks (Classes/Labels)	# Samples	Training / Validation / Test
PathMNIST	Colon Pathology	Multi-Class (9)	107,180	89,996 / 10,004 / 7,180
PneumoniaMNIST	Chest X-Ray	Binary-Class (2)	5,856	4,708 / 524 / 624
RetinaMNIST	Fundus Camera	Ordinal Regression (5)	1,600	1,080 / 120 / 400
BreastMNIST	Breast Ultrasound	Binary-Class (2)	780	546 / 78 / 156
BloodMNIST	Blood Cell Microscope	Multi-Class (8)	17,092	11,959 / 1,712 / 3,421
OrganCMNIST	Abdominal CT	Multi-Class (11)	23,583	12,975 / 2,392 / 8,216

CNN Structure used:

Layer 1: nn.Conv2d(1, 16, kernel_size=3)

- **Kernel size**: 3x3
- 16 Feature maps made
- After this layer, 16 f-maps and the feature map size is 26x26.

Layer 2: nn.Conv2d(16, 16, kernel_size=3) + nn.MaxPool2d(kernel_size=2, stride=2)

- After this layer, 16 f-maps and the feature map size is 12x12.

Layer 3: nn.Conv2d(16, 64, kernel_size=3)

- After this layer, 64 f-maps the feature map size is 10x10.

Layer 4: nn.Conv2d(64, 64, kernel_size=3)

- After this layer, 64 f-maps and the feature map size is 8x8.

Layer 5: nn.Conv2d(64, 64, kernel_size=3, padding=1) + nn.MaxPool2d(kernel_size=2, stride=2)

- After this layer, 64 maps and the feature map size is 4x4.

```python
# CNN model for MedMNIST with KANLinear
class CNNKAN(nn.Module):
    def __init__(self):
        super(CNNKAN, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3),
            nn.BatchNorm2d(16),
            nn.ReLU())

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size=3),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer3 = nn.Sequential(
            nn.Conv2d(16, 64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU())

        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU())

        self.layer5 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.fc = nn.Sequential(
            KANLinear(64 * 4 * 4, 128),
            nn.ReLU(),
            KANLinear(128, 128),
            nn.ReLU(),
            KANLinear(128, 9))

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Fully Connected Dense Network used:

- 3 Layered dense network

Layer 1:

- 64*4*4 in-features
- 128 out - features

Layer 2:

- 128 in-features
- 128 out-features

Layer 3:

- 128 in-features
- 9 out-features (classes) ***(varies with different MedMNIST datasets based on number of output classes)***

```
self.fc = nn.Sequential(
    KANLinear(64 * 4 * 4, 128),
    nn.ReLU(),
    KANLinear(128, 128),
    nn.ReLU(),
    KANLinear(128, 9))
```

Metrics Used:

F1 SCORE:

The F1 score is the **harmonic mean of Precision and Recall**. It combines these two metrics into single number that balances both concerns:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

AUC_ROC:

The **AUC** is the area under the ROC curve. It summarizes the ROC curve into a single value, ranging from 0 to 1.

The higher the AUC, the better the model's performance at distinguishing between positive and negative classes.

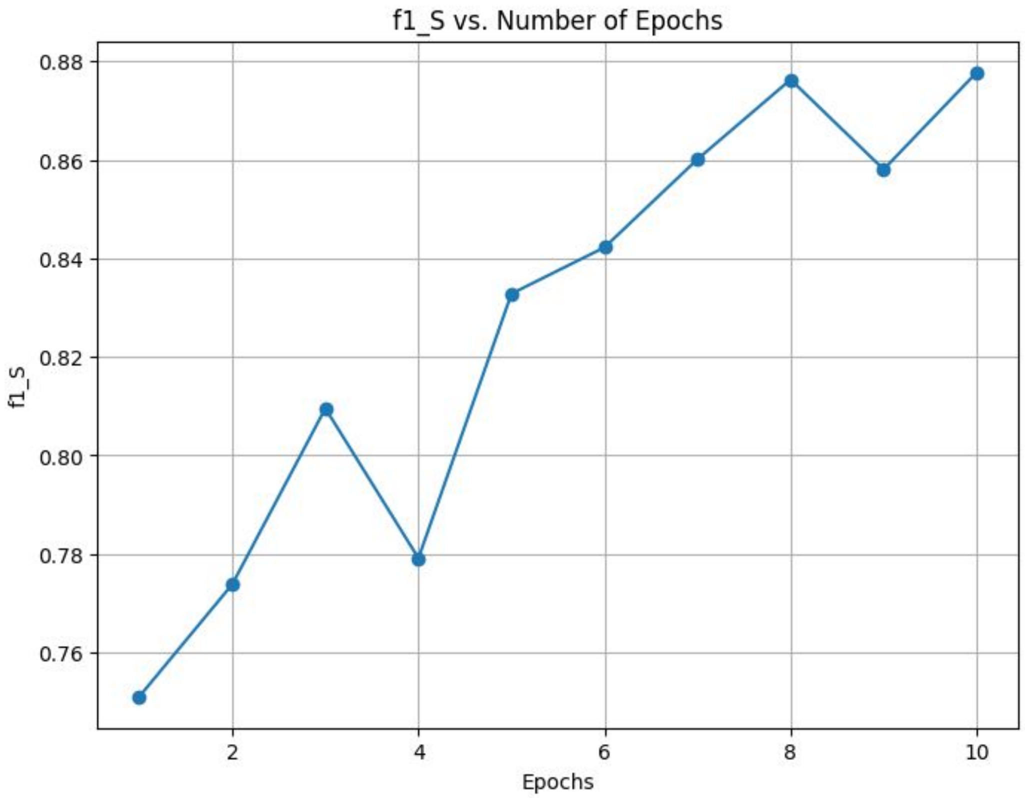
Range: The value of AUC ranges from 0 to 1:

- **1:** Perfect model.
- **0.5:** Random guess.
- **< 0.5:** Worse than random (often due to a problem with the model).

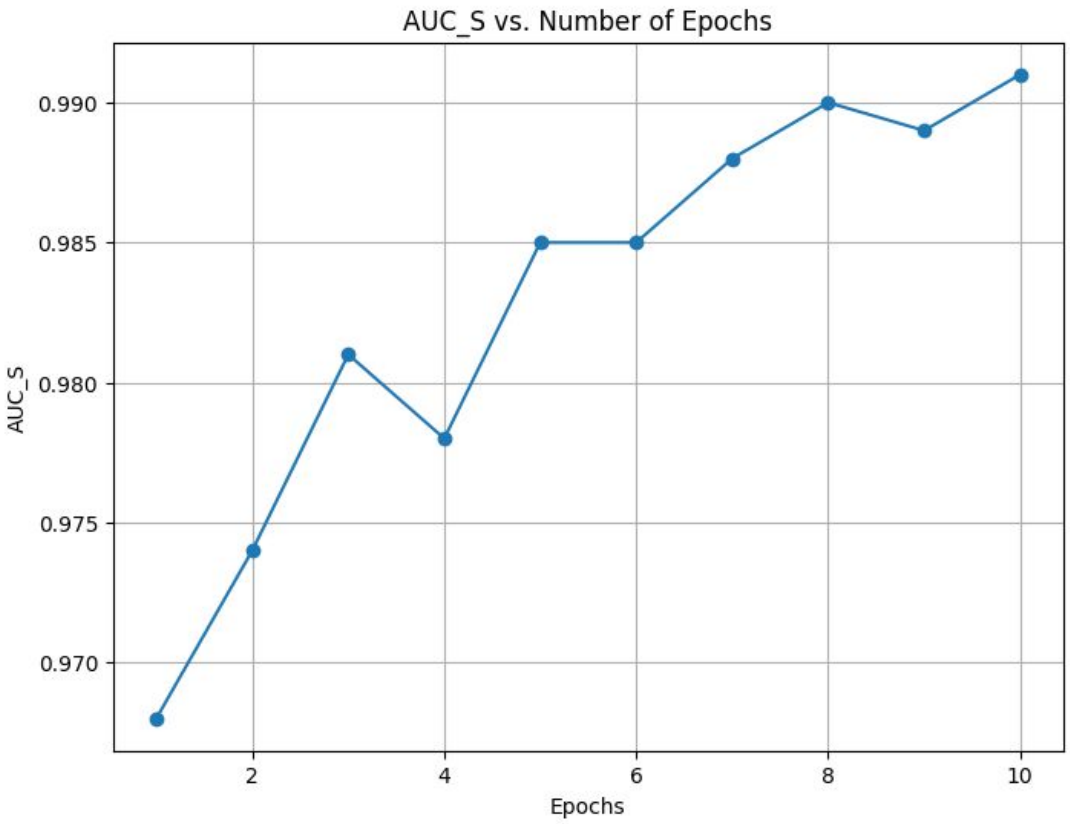
Accuracy:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

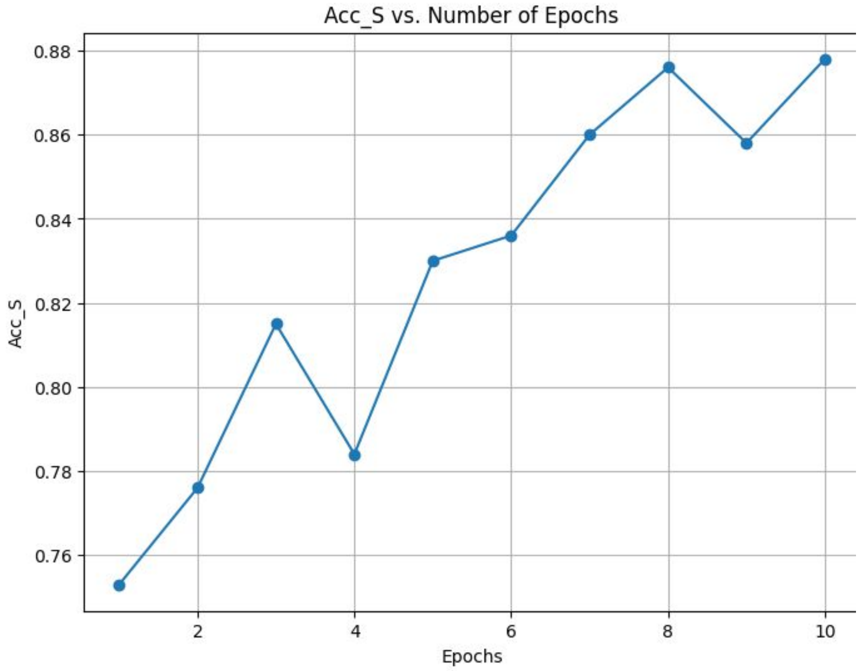
F-1 Score vs Epochs

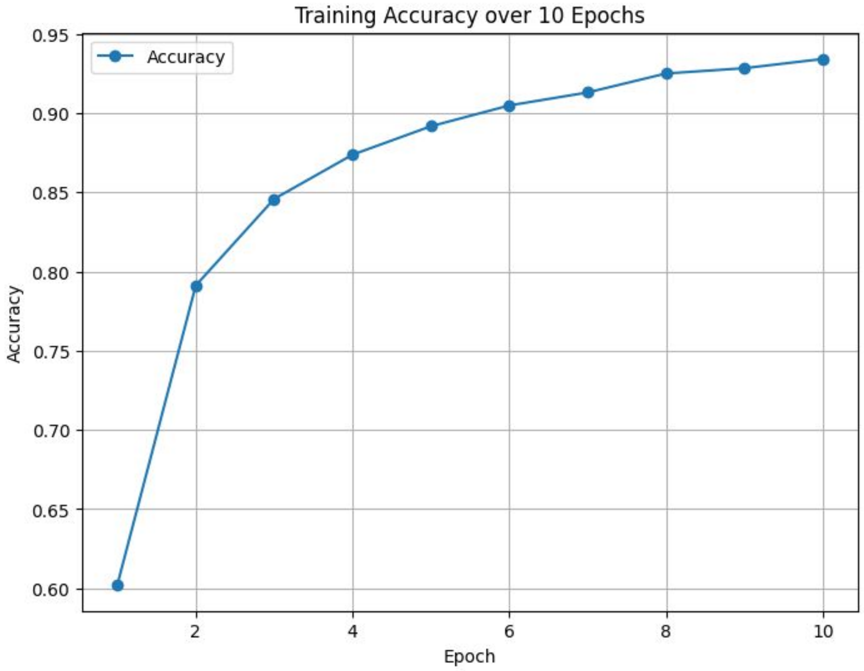


AUC-ROC vs Epochs:

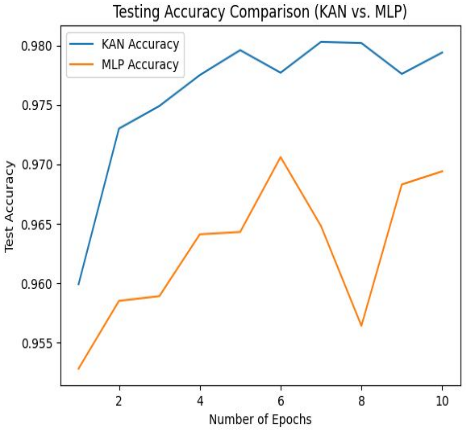


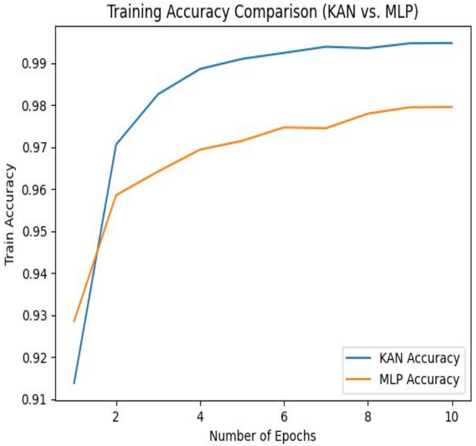
Accuracy vs Epochs:

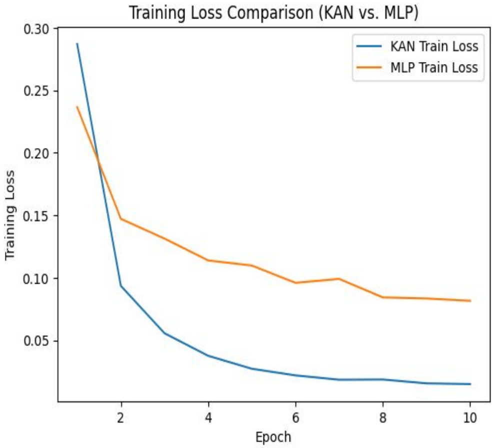


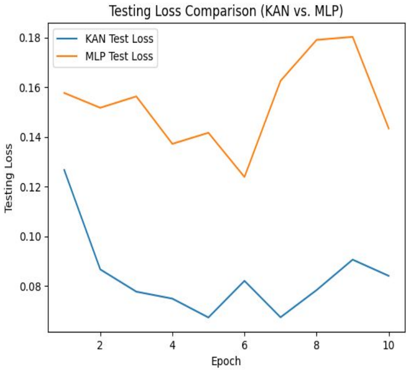


RESULTS*(on MNIST DATASET)*









RESULTS*(on MedMNIST: PathMNIST)*

Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	PathMNIST AUC	ACC
ResNet-18 (28)[10]	0.983	0.907
ResNet-18 (224)[10]	0.989	0.909
ResNet-50 (28)[10]	**0.990**	**0.911**
ResNet-50 (224)[10]	0.989	0.892
auto-sklearn[11]	0.934	0.716
AutoKeras[12]	0.959	0.834
Google AutoML Vision	0.944	0.728

		PathMNIST	
	Epochs	**AUC**	**ACC**
	3	0.94	0.627
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.966	0.724
5,3		0.96	0.735
3,5		0.929	9.666
5,5		0.951	0.671

RESULTS*(on MedMNIST: PneumoniaMNIST)*

Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	PneumoniaMNIST AUC	PneumoniaMNIST ACC
ResNet-18 (28)[10]	0.944	0.854
ResNet-18 (224)[10]	0.956	0.864
ResNet-50 (28)[10]	0.948	0.854
ResNet-50 (224)[10]	0.962	0.884
auto-sklearn[11]	0.942	0.855
AutoKeras[12]	0.947	0.878
Google AutoML Vision	**0.991**	**0.946**

		PneumoniaMNIST	
	Epochs	**AUC**	**ACC**
	3	0.945	0.772
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.938	0.838
5,3		0.945	0.811
3,5		0.939	0.822
5,5		0.936	0.83

RESULTS*(on MedMNIST: RetinaMNIST)*

Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	RetinaMNIST AUC	RetinaMNIST ACC
ResNet-18 (28)[10]	0.717	0.524
ResNet-18 (224)[10]	0.710	0.493
ResNet-50 (28)[10]	0.726	0.528
ResNet-50 (224)[10]	0.716	0.511
auto-sklearn[11]	0.690	0.515
AutoKeras[12]	0.719	0.503
Google AutoML Vision	**0.750**	**0.531**

		RetinaMNIST	
	Epochs	**AUC**	**ACC**
	3	0.522	0.395
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.509	0.155
5,3		0.53	0.435
3,5		0.534	0.435
5,5		0.472	0.435

RESULTS*(on MedMNIST: BreastMNIST)*

Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	BreastMNIST AUC	BreastMNIST ACC
ResNet-18 (28)[10]	0.901	**0.863**
ResNet-18 (224)[10]	0.891	0.833
ResNet-50 (28)[10]	0.857	0.812
ResNet-50 (224)[10]	0.866	0.842
auto-sklearn[11]	0.836	0.803
AutoKeras[12]	0.871	0.831
Google AutoML Vision	**0.919**	0.861

		BreastMNIST	
	Epochs	**AUC**	**ACC**
	3	0.462	0.619
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.628	0.269
5,3		0.458	0.731
3,5		0.351	0.269
5,5		0.436	0.731

RESULTS*(on MedMNIST: BloodMNIST)*

Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	BloodMNIST AUC	BloodMNIST ACC
ResNet-18 (28)[10]	**0.998**	0.958
ResNet-18 (224)[10]	**0.998**	0.963
ResNet-50 (28)[10]	0.997	0.956
ResNet-50 (224)[10]	0.997	0.950
auto-sklearn[11]	0.984	0.878
AutoKeras[12]	**0.998**	0.961
Google AutoML Vision	**0.998**	**0.966**

		BloodMNIST	
	Epochs	**AUC**	**ACC**
	3	0.94	0.707
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.935	0.702
5,3		0.949	0.736
3,5		0.933	0.673
5,5		0.945	0.736

RESULTS*(on MedMNIST: OrganCMNIST)*

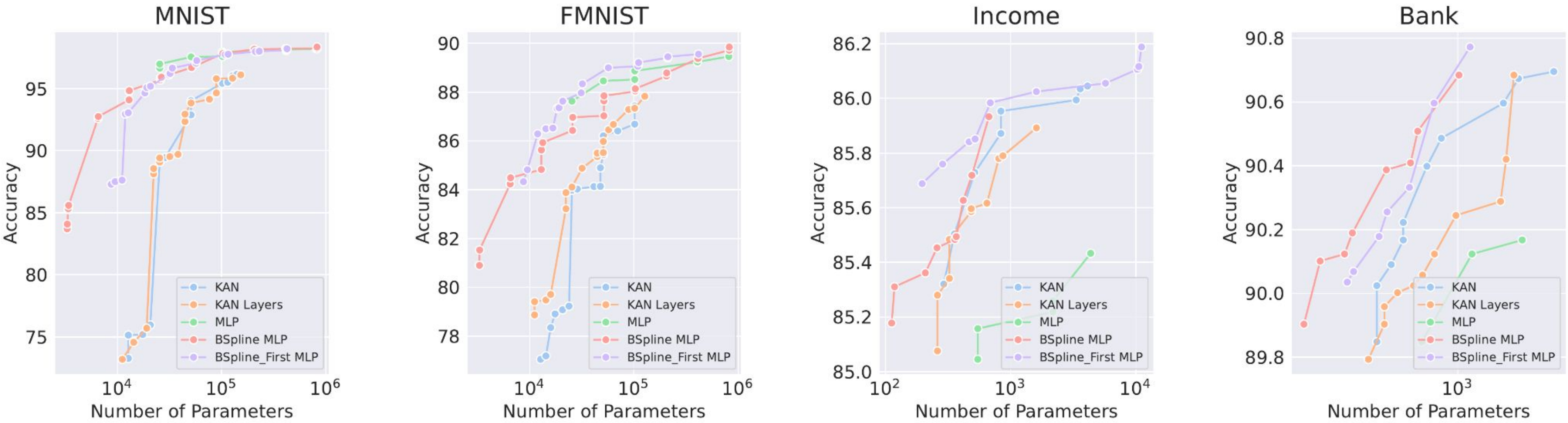
Benchmark on each dataset of MedMNIST2D in metrics of AUC and ACC.

Methods	OrganCMNIST AUC	OrganCMNIST ACC
ResNet-18 (28)[10]	0.992	0.900
ResNet-18 (224)[10]	**0.994**	**0.920**
ResNet-50 (28)[10]	0.992	0.905
ResNet-50 (224)[10]	0.993	0.911
auto-sklearn[11]	0.976	0.829
AutoKeras[12]	0.990	0.879
Google AutoML Vision	0.988	0.877

		OrganCMNIST	
	Epochs	**AUC**	**ACC**
	3	0.877	0.943
(Grid Size, Spline Order, Epochs = 3)			
3,3		0.867	0.43
5,3		0.882	0.362
3,5		0.871	0.483
5,5		0.882	0.396

Observations and Outcomes

By ablating the network architecture of KAN and MLP, the primary advantage of KAN lies in the use of B-Spline functions. Replacing the activation functions in MLP with B-Spline allows MLP to outperform KAN on datasets where KANN previously had the upper hand.



Img Src: [2407.16674 (arxiv.org)]

Future Work

1. Temporal KANs
2. Adversarial KANs