GTU Department of Computer Engineering

CSE 655

Homework 4 - Part 2

Report

Emirkan Burak Yılmaz

254201001054

Contents

1. Introduction

This report presents the implementation of a simplified Kolmogorov–Arnold Network (KAN), a neural network architecture that offers an alternative to the traditional Multi-Layer Perceptron (MLP) by introducing learnable activation functions. The implemented version deviates from the original KAN design by replacing B-spline-based activations with piecewise linear functions.

2. Algorithmic Details

The simplified KAN architecture consists of three core components:

- **LinearSpline**: A learnable piecewise linear activation module.
- **KANLayer**: A single layer in the KAN, where each connection is equipped with its own spline function.
- **KAN**: The full network, composed of multiple stacked KANLayers.

Each of these modules is detailed below.

2.1. LinearSpline

At the heart of the KAN architecture lies the LinearSpline module (Figure 1). This component defines a trainable piecewise linear activation function, constructed by interpolating between a fixed set of input-domain knot points (knot_x) and a learnable set of output values (knot_y). The linear interpolation ensures differentiability, enabling gradient-based optimization.

In contrast to the original KAN implementation, which blends the spline output with a SiLU (Sigmoid Linear Unit) activation using learnable scalar weights, this simplified version removes the SiLU component entirely. This decision was made based on experimental results, which showed that excluding the SiLU activation led to improved performance and generalization, while also reducing model complexity by eliminating weights for SiLU and spline components.

To maintain numerical stability, inputs to the spline function are clamped within the valid knot range, ensuring all values are safely interpolated.

```python
class LinearSpline(nn.Module):
    """
    A learnable piecewise linear activation function (linear spline).
    Knots are trainable and define the shape of the activation.
    """
    def __init__(self, num_knots=5, init_scale=0.1, input_range=3.0):
        super().__init__()
        self.num_knots = num_knots
        self.input_range = input_range

        # Linear spline parameters
        self.register_buffer("knot_x", torch.linspace(-input_range, input_range, num_knots))
        self.knot_y = nn.Parameter(torch.randn(num_knots) * init_scale)

    def forward(self, x):
        # Clamp input to valid range
        x_clamped = torch.clamp(x, -self.input_range, self.input_range)

        # Compute spline component
        segments = torch.searchsorted(self.knot_x, x_clamped) - 1
        segments = segments.clamp(0, self.num_knots - 2)

        x0 = self.knot_x[segments]
        x1 = self.knot_x[segments + 1]
        y0 = self.knot_y[segments]
        y1 = self.knot_y[segments + 1]

        alpha = (x_clamped - x0) / (x1 - x0 + 1e-6)
        spline_out = y0 + alpha * (y1 - y0)

        return spline_out
```

Figure 1: The LinearSpline module implements a learnable piecewise linear activation function combined with SiLU

2.2. KANLayer

The KANLayer replaces a standard fully connected (dense) layer with a novel mechanism (Figure 2). Instead of computing dot products between inputs and weights, it processes each input-output pair with a dedicated LinearSpline. For an input dimension dimension d_{in} and output dimension d_{out}, the layer $d_{in} \times d_{out}$ independent spline functions.

During the forward pass, each output neuron aggregates the contributions from all input neurons, each transformed through its own spline activation. This design allows the network to learn highly flexible and input-specific non-linear transformations, significantly enhancing its expressive power compared to classical activations.

```python
class KANLayer(nn.Module):
    """
    A single KAN layer with learnable piecewise linear activations.
    Replaces a traditional linear layer.
    """
    def __init__(self, input_dim, output_dim, num_knots=5):
        super().__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim

        # Each edge has its own spline activation
        self.splines = nn.ModuleList([
            LinearSpline(num_knots) for _ in range(input_dim * output_dim)
        ])

    def forward(self, x):
        batch_size = x.shape[0]
        out = torch.zeros(batch_size, self.output_dim).to(x.device)

        # Apply each spline to its corresponding input
        for j in range(self.output_dim):
            for i in range(self.input_dim):
                spline_idx = i * self.output_dim + j
                out[:, j] += self.splines[spline_idx](x[:, i])

        return out
```

Figure 2: The KANLayer module replaces fully connected weights with dedicated spline functions for each connection

2.3. KAN

The full KAN model is constructed by stacking multiple KANLayer modules in a feedforward fashion, with the number of hidden layers treated as a configurable hyperparameter (Figure 3). The model maintains architectural flexibility, allowing for controlled depth in experiments.

```python
class KAN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim=1, num_hidden_layers=2, num_knots=5):
        super().__init__()
        self.layers = nn.ModuleList()

        # First layer
        self.layers.append(KANLayer(input_dim, hidden_dim, num_knots))

        # Hidden layers
        for _ in range(num_hidden_layers - 1):
            self.layers.append(KANLayer(hidden_dim, hidden_dim, num_knots))

        # Output layer
        self.layers.append(KANLayer(hidden_dim, output_dim, num_knots))

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

Figure 3: The KAN model consists of stacked KANLayers

3. Experimental Results

Two synthetic regression tasks were designed to evaluate and compare the performance of the proposed KAN model with a standard MLP baseline (Figure 4). In each experiment, datasets were generated based on predefined nonlinear functions, and both KAN and MLP models were trained for 2000 epochs using the same optimization and evaluation procedures. A 60/20/20 split was used for training, validation, and test sets, respectively. Gaussian noise was added to simulate real-world imperfections.

```python
class MLP(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=20, output_dim=1, num_hidden_layers=2):
        super().__init__()
        layers = []

        # First layer
        layers.append(nn.Linear(input_dim, hidden_dim))
        layers.append(nn.ReLU())

        # Hidden layers
        for _ in range(num_hidden_layers - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.ReLU())

        # Output layer
        layers.append(nn.Linear(hidden_dim, output_dim))

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

Figure 4: Standard feed-forward multi-layer perceptron with ReLU activation function

3.1. Experiment 1

$$y = sin(x_1) + cos(x_2) + x_1^2 - x_2^2$$

The first experiment involves the approximation of function containing both periodic and polynomial components.

Model Configuration:

- Input Dimension: 2
- Hidden Dimension: 5
- Number of Hidden Layers: 1
- Number of Knots (KAN): 10

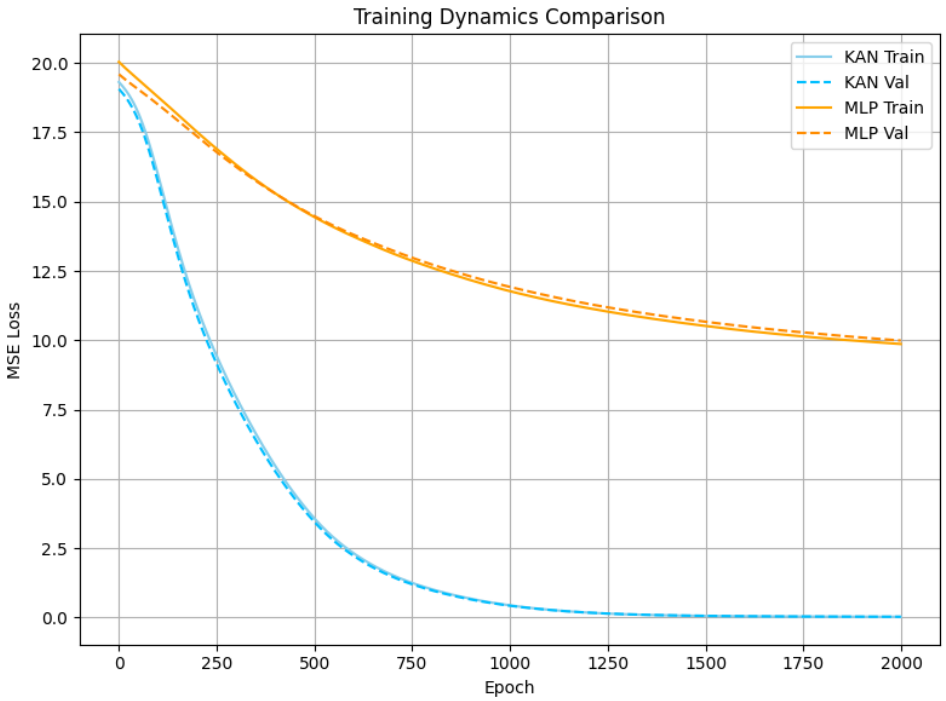

Figure 5: Training and validation loss curves for experiment 1

Figure 5 shows the training and validation loss curves, illustrating how the KAN converges to a much lower error in smaller number of epochs compared to the MLP.

Table 1: KAN vs. MLP performance for experiment 1

Metric	**KAN**	**MLP**
Parameters	150	21
Training Time (s)	36.26	2.39
Train MSE	0.0256	9.8650
Val MSE	0.0257	9.9901
Test MSE	0.0266	9.8581

As shown in the results presented in Table 1, KAN significantly outperformed MLP in terms of MSE on all data splits, despite having more parameters (150 vs. 21) and a longer training time (36.26s vs. 2.39s). While the MLP offers a lightweight baseline with fixed activations, it struggled to match the performance of KAN. The learnable spline-based activations in KAN provided greater flexibility,

enabling the model to better capture the structure of the target function and achieve much lower mean squared error.

Figures 6 and 7 visualize the learned piecewise linear activation functions from the first and last (second) KAN layers respectively, demonstrating the diverse and adaptive nonlinearities learned by individual connections.

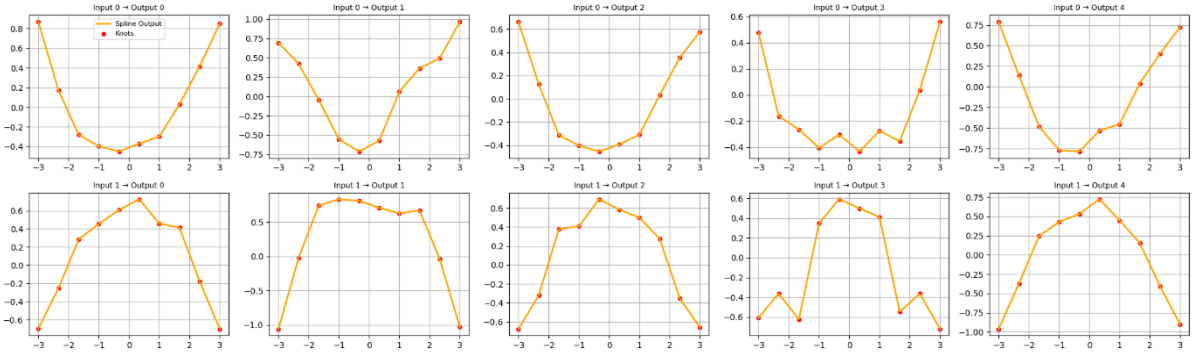

Figure 6: Visualization of piecewise linear activation functions for the first layer for experiment 1

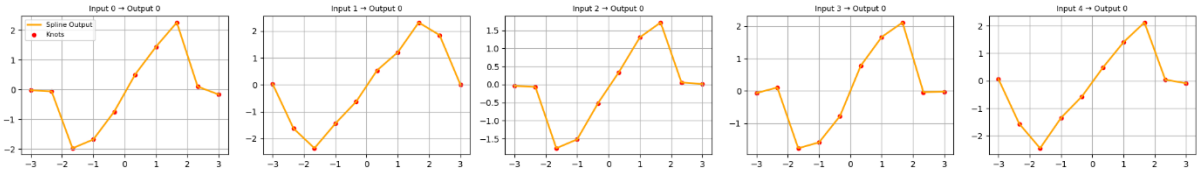

Figure 7: Visualization of piecewise linear activation functions for the second layer for experiment 1

3.2. Experiment 2

$$y = \exp\left(\sin(x_1^2 + x_2^2) + \sin(x_3^2 + x_4^2)\right)$$

The second experiment is significantly more complex, involving higher-dimensional input and a deeply nested nonlinear function. This function poses a considerable challenge due to the interaction of multiple nonlinear transformations, including squared inputs, nested sine functions, and an exponential output.

Model Configuration:

- Input Dimension: 4
- Hidden Dimension: 5
- Number of Hidden Layers: 2
- Number of Knots (KAN): 10

Figure 8 presents the training and validation loss curves, showing that KAN converges to lower error levels than MLP.

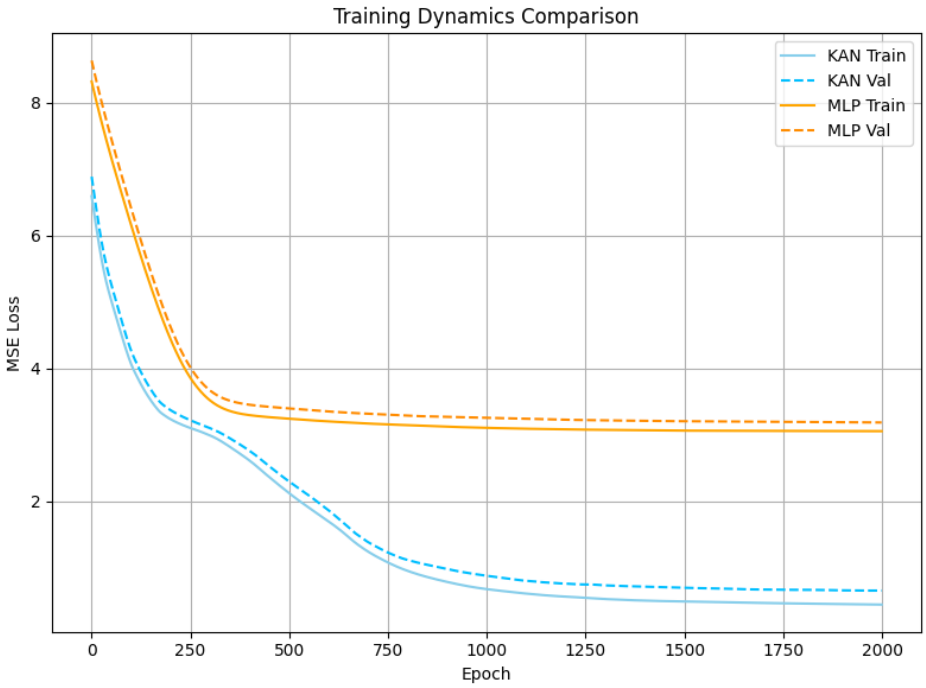

Figure 8: Training and validation loss curves for experiment 2

Table 2: KAN vs. MLP performance for experiment 2

Metric	**KAN**	**MLP**
Parameters	500	61
Training Time (s)	107.16	3.31
Train MSE	0.4406	3.0540
Val MSE	0.6516	3.1848
Test MSE	0.7341	3.3156

In Table 2, this performance gap remains consistent in a more complex task. KAN again outperforms MLP with lower error across training, validation, and test sets, though at the cost of increased parameter count (500 vs. 61) and longer training time (107.16s vs. 3.31s). Despite both models being relatively compact, KAN's ability to learn activation functions for each input-output connection allows it to more effectively model the complex function, demonstrating the value of adaptive activations in function approximation tasks.

Figures 9 and 10 show the visualization of learned piecewise linear activation functions in the first and last (third) KAN layers respectively, highlighting the diverse shapes that emerge to capture the nonlinearities in the data.

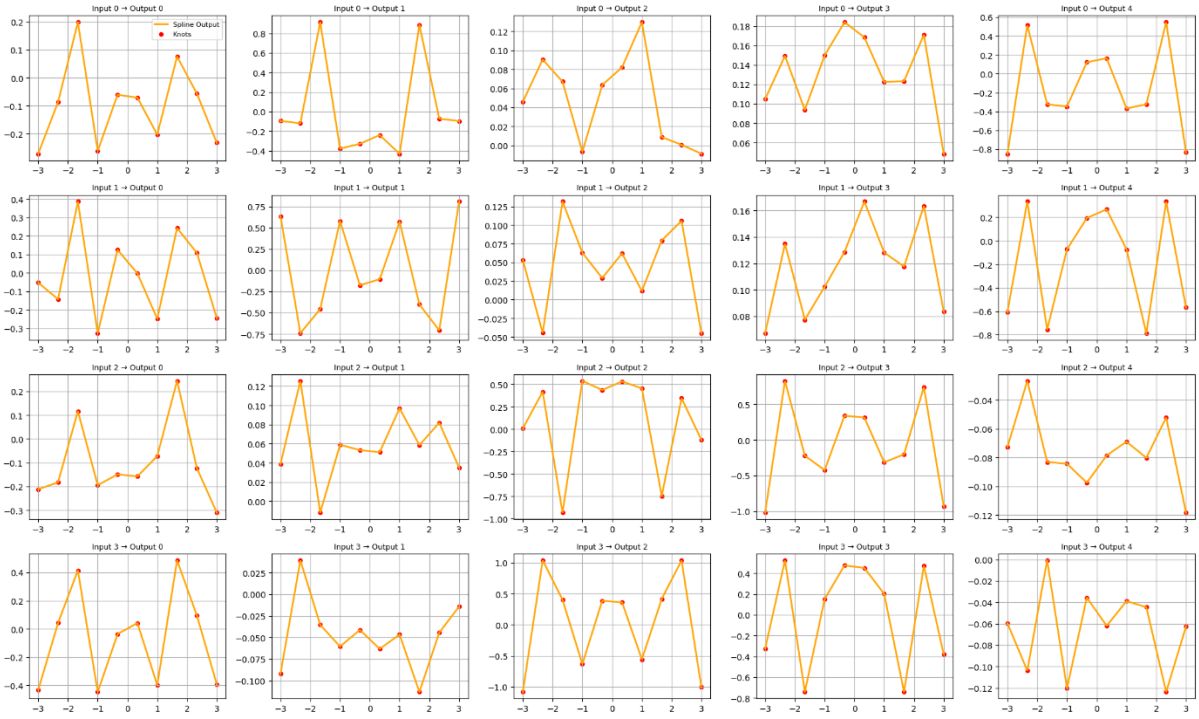

Figure 9: Visualization of piecewise linear activation functions for the first layer for experiment 2

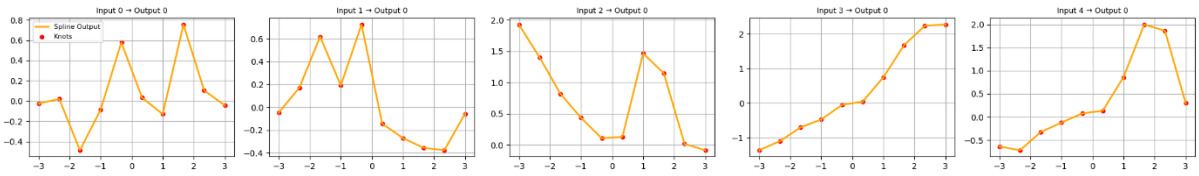

Figure 10: Visualization of piecewise linear activation functions for the last (third) layer for experiment 2

4. Conclusion

This report presented the implementation of a simplified Kolmogorov–Arnold Network (KAN) architecture, utilizing piecewise linear activation functions instead of B-splines. The adapted KAN model was evaluated on two function approximation tasks of increasing complexity and compared against a standard Multi-Layer Perceptron (MLP) baseline.

Experimental results demonstrated that the KAN architecture consistently achieved significantly lower mean squared error (MSE) across training, validation, and test sets. Despite involving more parameters and longer training durations than the MLP, the KAN models showed superior accuracy and function approximation capability. This performance is attributed to KAN's use of learnable, connection-specific activation functions, which provide enhanced representational flexibility compared to fixed activation functions used in MLPs.

It is important to note that the current implementation represents a basic version of KAN, employing linear splines in place of the B-splines used in the original formulation. While this simplification improves interpretability and implementation ease, the original B-spline-based activations are expected to offer greater expressiveness and potentially improved performance.

These results highlight the potential of KAN architectures as powerful alternatives to traditional feedforward networks, particularly in tasks requiring fine-grained function approximation. Future work may involve extending the implementation with B-splines, incorporating regularization techniques, and applying the model to higher-dimensional and real-world datasets.