CS163 Solo Project Report

Tai Pham Nguyen Anh

August 2024

The program is designed to visualize Kolmogorov-Arnold Networks (KANs), a type of neural network with learnable activation functions on edges rather than nodes. It provides tools for loading and training KAN models, visualizing their architecture, and displaying the unique B-spline activation functions that define the network's behavior. Users can interact with the visualization to adjust parameters and observe the impact on the network's output in real time. The program aims to enhance understanding and interpretation of KANs by offering a clear graphical representation and interactive manipulation of network components.

1 Kolmogorov-Arnold Networks (KANs)

Kolmogorov-Arnold Networks (KANs) are inspired by the Kolmogorov-Arnold representation theorem and offer an alternative to Multi-Layer Perceptrons (MLPs). Unlike MLPs that use fixed activation functions on nodes, KANs implement learnable activation functions on edges. This distinction leads to the following characteristics:

- **Learnable Activation Functions:** KANs replace traditional weight parameters with univariate functions parameterized as splines. These functions are learnable, allowing the network to adapt the transformations it applies to the inputs dynamically.
- **No Linear Weights:** In contrast to MLPs, which separate linear transformations and non-linear activations, KANs integrate these into learnable functions. Thus, KANs do not employ linear weight matrices, relying solely on the functional transformations defined at each edge.
- **Flexible Architecture:** KANs generalize the Kolmogorov-Arnold representation to support networks of arbitrary depth and width. This allows them to effectively represent complex functions and enhances their ability to generalize across different tasks.
- **Improved Interpretability and Accuracy:** Due to the learnable nature of the activation functions, KANs offer greater interpretability. They

have demonstrated higher accuracy in function fitting tasks and possess faster neural scaling laws compared to MLPs.

The fundamental operation of a KAN can be described as a composition of layers, each defined by a matrix of univariate functions:

$$KAN(x) = (\Phi_{L-1} \circ \Phi_{L-2} \circ \ldots \circ \Phi_1 \circ \Phi_0)(x),$$

where each Φ_l represents a layer of learnable functions. This architecture allows KANs to efficiently capture and model the compositional structure of functions, making them a powerful tool for both AI and scientific applications.

How Kolmogorov-Arnold Networks (KANs) Work

Kolmogorov-Arnold Networks (KANs) are a novel class of neural networks inspired by the Kolmogorov-Arnold representation theorem, which states that any multivariate continuous function can be represented as a finite composition of continuous univariate functions and the addition operation. Unlike traditional Multi-Layer Perceptrons (MLPs), which use fixed activation functions on their nodes (neurons), KANs leverage learnable activation functions on their edges (weights), represented using spline functions. This unique architecture offers enhanced flexibility and interpretability, making KANs particularly suitable for tasks requiring high accuracy and human-understandable representations.

1. Network Architecture

A KAN consists of multiple layers, each defined by a set of nodes and learnable activation functions on the edges. Formally, let the input layer be represented as a vector $\mathbf{x} \in R^n$, and the output layer as $\mathbf{y} \in R^m$. The KAN is structured as a series of transformations:

$$\mathbf{y} = \Phi_L \circ \Phi_{L-1} \circ \cdots \circ \Phi_1(\mathbf{x})$$

where Φ_l denotes the transformation applied by the l-th layer.

Each layer l is composed of n_l nodes and is connected to the subsequent layer by edges, each of which carries a learnable univariate spline function φ_{ij}. Here, φ_{ij} represents the function mapping from the i-th node in layer l to the j-th node in layer $l+1$.

2. Layer Functionality

In a traditional neural network, each layer computes an affine transformation followed by a non-linear activation. In KANs, each layer l computes its output $\mathbf{z}^{(l)}$ as:

$$z_j^{(l+1)} = \sum_{i=1}^{n_l} \varphi_{ij}(z_i^{(l)})$$

where $z_i^{(l)}$ is the output of the i-th node in layer l, and φ_{ij} is a learnable spline function that maps the input from node i in layer l to node j in layer $l+1$.

3. Spline Functions as Learnable Parameters

The activation functions φ_{ij} in KANs are represented as splines, which are piecewise polynomial functions defined over a series of control points. Let $\mathbf{t} = \{t_0, t_1, \ldots, t_K\}$ be a set of K control points, then a spline function $\varphi_{ij}(x)$ can be expressed as:

$$\varphi_{ij}(x) = \sum_{k=0}^{K} c_k B_k(x)$$

where c_k are the coefficients (learnable parameters), and $B_k(x)$ are the basis functions, typically chosen as B-splines. B-splines are defined such that:

$$B_k(x) = \begin{cases} 1 & \text{if } x \in [t_k, t_{k+1}] \\ 0 & \text{otherwise} \end{cases}$$

These splines allow the network to learn complex non-linear functions efficiently by adjusting the coefficients c_k during training.

4. Training Process

The training of KANs is similar to that of traditional neural networks, relying on backpropagation to minimize a loss function. However, instead of learning scalar weights, KANs learn the parameters of the spline functions.

- **Forward Pass**: In the forward pass, the network computes the output by passing the input $\mathbf{x}$ through each layer, applying the spline functions on the edges to compute the activations:

$$\mathbf{z}^{(l+1)} = \Phi_l(\mathbf{z}^{(l)}) = \left\{ \sum_{i=1}^{n_l} \varphi_{ij}(z_i^{(l)}) \right\}_{j=1}^{n_{l+1}}$$

- **Loss Computation**: The loss $\mathcal{L}$ is computed between the network's output $\mathbf{y}$ and the true labels $\mathbf{y}_{\text{true}}$. A common choice for the loss function is the mean squared error (MSE):

$$\mathcal{L} = \frac{1}{M}\sum_{k=1}^{M}(y_k - y_{\text{true},k})^2$$

where M is the number of training examples.

- **Backward Pass and Parameter Update**: During the backward pass, gradients of the loss function with respect to the spline parameters c_k are computed using the chain rule. These gradients are then used to update the spline parameters:

$$c_k \leftarrow c_k - \eta\frac{\partial \mathcal{L}}{\partial c_k}$$

where η is the learning rate.

5. Grid Extension Technique

To enhance accuracy, KANs use a grid extension technique where the grid size of the spline control points is gradually increased during training. This approach allows the network to start with a coarse approximation and refine it over time, improving the model's capacity without a significant increase in training time.

- Initially, splines are defined over a small number of control points $\mathbf{t} = \{t_0, t_1, \dots, t_{G_0}\}$.
- As training progresses, the grid is extended to $\mathbf{t} = \{t_0, t_1, \dots, t_{G_t}\}$, where $G_t > G_0$. The new control points are inserted based on the distribution of data points or using predefined rules.

This fine-graining approach allows the network to adjust its complexity dynamically and capture finer details of the target function.

6. Interpretability and Visualization

KANs offer enhanced interpretability compared to traditional neural networks. Since each edge represents a univariate function, visualizing these functions provides insights into how the network processes information. This interpretability is particularly valuable in scientific applications, where understanding the underlying model behavior is crucial.

2 Functional Data Structure

The simple and from-scratch implementation of Kolmogorov-Arnold Network is implemented in C++. The network have the following capabilities:

- train(): This function will train the model, given the data, number of epochs, and sizes of each epochs, and learning rate.
- predict(): This function will predict the output, after being trained.
- initialize: This will construct the network with the specified number of layers, number of grid points for each edge and the order of splines for each edge.
- forward(): This function do the forward pass through each layer in the network layer-by-layer then returns the output.
- backward(): This function do the backward propagation after the forward pass to adjust the scalars, biases and coefficients of each splines function on each layer.

3 Structure of the Program

`KAN.h` and `KAN.cpp`

- `KAN` **Class**: This is the main class representing a Kolmogorov-Arnold Network. It encapsulates the architecture and functionality of the KAN, including initialization, training, and inference.
- **Functions**:
  - `train()`: Handles the training of the KAN model on a given dataset.
  - `evaluate()`: Evaluates the model performance on test data.
  - `loadParameters()`: Loads saved parameters from a file to restore a trained model.
  - `saveParameters()`: Saves the current model parameters to a file for future use.
  - `forward()`: Implements the forward pass through the network layers, computing the output for a given input.
- **Variables**:
  - `layers`: Stores the layers of the network.
  - `learningRate`: The rate at which the model updates its weights during training.

`KANLayer.h` and `KANLayer.cpp`

- `KANLayer` **Class**: Represents a single layer within the KAN. Each layer performs transformations using univariate functions, which are learnable.
- **Functions**:

  - `compute()`: Computes the output of the layer given an input, using its activation functions.
  - `initialize()`: Sets up the initial state of the layer, including activation functions.

- **Variables**:
  - `activations`: The list of activation functions used in the layer.
  - `inputSize` and `outputSize`: Define the dimensions of the input and output for the layer.

`spline.h` and `spline.cpp`

- **Spline Interpolation**: These files handle the definition and usage of spline functions, which are used as learnable activation functions in the KAN. Spline interpolation allows for smooth, adjustable curves to be fitted, making the network flexible in learning complex functions.
- **Functions**:
  - `computeSpline()`: Evaluates the spline function for a given input value.
  - `initializeSpline()`: Initializes the parameters of the spline function.

`utils.h` and `utils.cpp`

- **Utility Functions**: Contains various helper functions used throughout the project, such as activation functions and numerical utilities.
- **Functions**:
  - `sigmoid()`: Implements the sigmoid activation function.
  - `silu()`: A variant of the SiLU (Sigmoid Linear Unit) activation function.
  - `linspace()`: Generates a linearly spaced vector between two values, used for defining grid points.
- **Variables**:
  - `grid`: A global variable defining the grid used for interpolation and other computations.

`EngineSupport.h`

- **Support Functions and Constants**: Provides additional support functions, macros, and constants needed for the KAN operations and visualization. This file might include definitions that facilitate the interaction between the core KAN logic and the frontend.

`main.cpp`

- **Main Entry Point**: This is the main application file that sets up the user interface, handles user inputs, and initiates the KAN operations. It integrates with a visualization library to show the results.
- **Functions**:
  - `DrawTextBoxWithInput()`: Handles drawing a text box and processing user input.
  - `VisualizePage::show()`: Manages the visualization of KAN operations, including play, pause, and navigation functionalities.
  - `func()`: Defines the function to be modeled by the KAN.
  - `randomFloat()`: Generates a random float number within a specified range.
- **Variables**:
  - `state`: Tracks the current state of the visualization (e.g., playing, paused).
  - `play_button`, `pause_button`: Manage the state of the play and pause actions.

How the Project Works

1. **Initialization**: The project initializes the KAN with a specified number of layers and nodes. Each layer is defined with learnable spline-based activation functions.
2. **Training**: The network is trained using a dataset to learn the underlying function. The training process involves adjusting the parameters of the spline functions to minimize the error between the predicted output and the actual data.
3. **Evaluation**: After training, the KAN model is evaluated on test data to determine its accuracy and performance. This involves computing the output using the trained model and comparing it to the expected results.
4. **Visualization**: The frontend component visualizes the results, showing how the input is transformed through each layer of the network. Users can interact with the visualization, pause, play, and navigate through different states of the network during training and evaluation.
5. **User Interaction**: The application allows users to input parameters, control the training process, and observe the impact of different settings on the network's performance and visualization.

4 User's Manual

1. Clone my repository.

```
git clone <repository_url>
```

2. Compile the project by running the following command in your terminal. This command will compile the necessary source files and link them with the required libraries:

```
g++ main.cpp ./KAN/KAN.cpp ./KAN/KANLayer.cpp ./KAN/spline.cpp ./KAN/utils.cpp
    -o main.exe -std=c++20 -Llib -Iinclude -lraylib -lwinmm -lgdi32 -Wno-enum-compa
```

5 Commited List

6 Github Link

7 Link to Demo Video