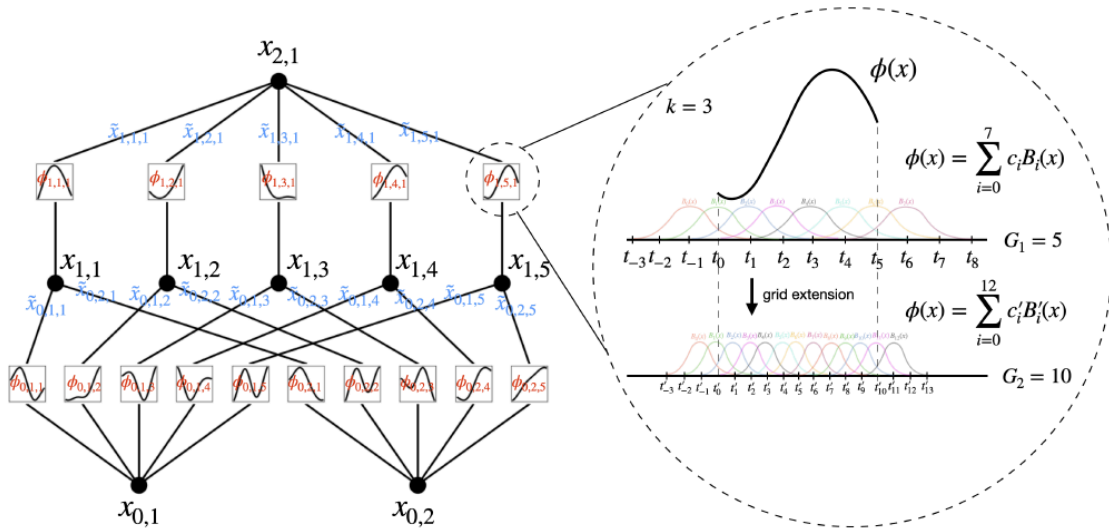


# Kolmogorov-Arnold Networks vs Multi Layer Perceptrons Project



## Introduction

This project explores the implementation and application of Kolmogorov-Arnold Networks. These networks use the Kolmogorov-Arnold representation theorem, which states that any continuous multivariate function can be represented as a finite composition of continuous univariate functions and addition. This has significant implications for neural network design and function approximation. Certainly! Here's the project overview, objectives, and methodology in Jupyter Markdown format:

**Project Overview:** The project involves comparing the performance of Kolmogorov Arnold Networks (KAN) and Multi-Layer Perceptron (MLP) models. This comparison is conducted to analyze their efficacy in capturing complex patterns within a given dataset.

### Objectives:

1. To evaluate the effectiveness of KAN and MLP models in capturing complex patterns.
2. To compare the convergence behavior and loss functions of KAN and MLP models.
3. To identify the strengths and weaknesses of each model for potential applications in real-world scenarios.
4. To provide insights that can inform future research and advancements in neural network architectures.

### Methodology:

1. **Dataset Selection:** Choose a dataset with complex patterns that require advanced neural network models for effective learning.
2. **Model Implementation:** Implement KAN and MLP models using appropriate neural network frameworks such as TensorFlow or PyTorch.

3. **Training:** Train both models on the selected dataset using standardized training procedures.
4. **Loss Function Calculation:** Calculate and record the loss functions of both models during the training process.
5. **Performance Evaluation:** Evaluate the performance of KAN and MLP models based on metrics such as accuracy, convergence speed, and computational efficiency.
6. **Statistical Analysis:** Perform statistical tests to compare the performance metrics of the two models.
7. **Visualization of Results:** Create visualizations such as loss function plots to illustrate the findings.
8. **Conclusion:** Draw conclusions based on the analysis of the results, highlighting the strengths and weaknesses of each model and discussing their implications for future research and practical applications.

## Kolmogorov-Arnold Representation Theorem

The Kolmogorov-Arnold representation theorem is a powerful result in mathematics that guarantees the representation of any continuous function of several variables as a superposition of continuous functions of one variable and addition.

Mathematically, for a continuous function  $(f : \mathbb{R}^n \rightarrow \mathbb{R})$ , there exist continuous functions  $(\phi_i)$  and  $(\psi_{ij})$  such that:

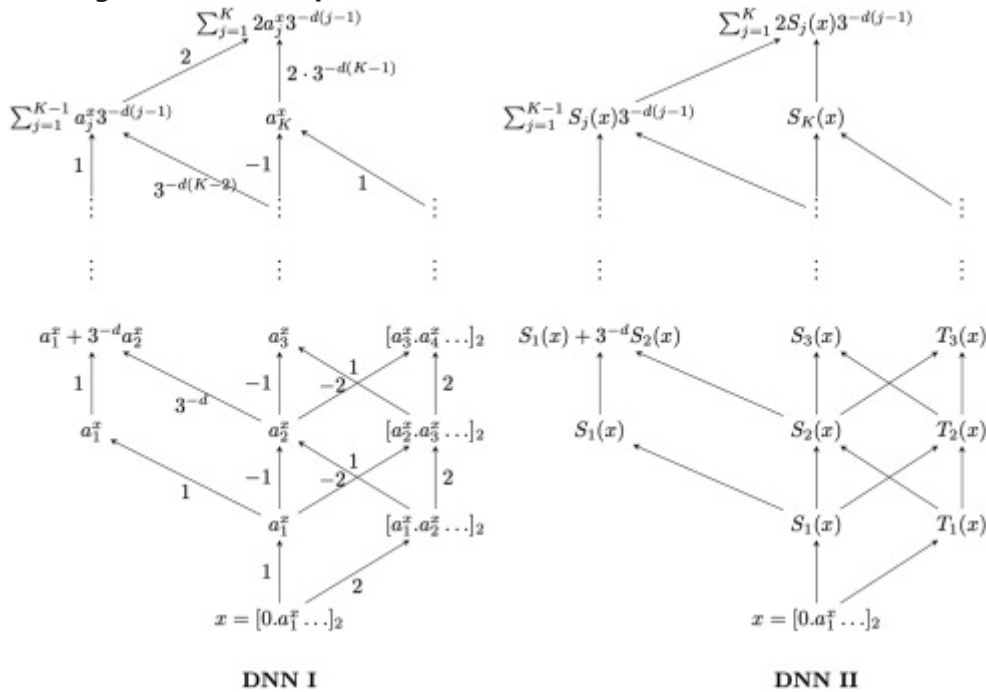
$$[f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{2n+1} \phi_i \left( \sum_{j=1}^n \psi_{ij}(x_j) \right)]$$

## Network Design

Using this theorem, I will design a neural network architecture where the output is a composition of univariate functions and addition. We implement this in PyTorch.

## Kolmogorov Approximation Theory vs Universal Approximation Theory

## Kolmogorov-Arnold Representation Theorem:



## Kolmogorov Approximation Theory:

- States that any multivariate continuous function can be represented as a finite sum of univariate continuous functions.
- Representation:  $f(x) = \sum (\Phi q(\sum (\phi q, p(xp))))$

Any continuous function  $f$  of  $n$  variables can be represented as:

$$f(x)$$

)

$\sum$

$q$

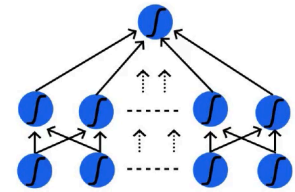
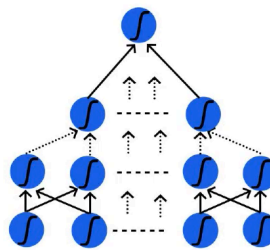
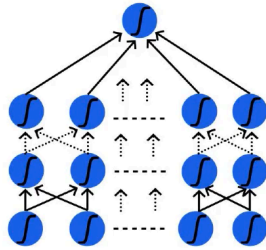
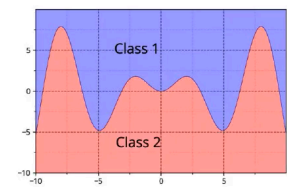
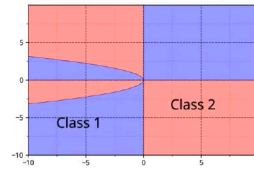
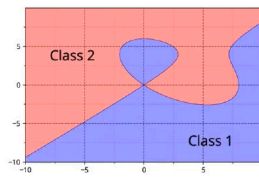
$$1 \leq n+1 \leq \Phi q(\sum$$

$p$

$$1 \leq n \leq q, p(xp)) \quad f(x) = \sum_{q=1}^{2n+1} \Phi q(p = 1 \sum_{n \leq q} \phi q, p(xp))$$
 Here,

$\phi q, p$  and  $\Phi q$  are continuous univariate functions.

## Universal Approximation Theorem:



- States that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function.
- Representation:  $f(x) \approx \sum (a_i * \sigma(w_i * x + b_i))$  For any continuous function

$f$  on a compact subset of

$\mathbb{R}^n$ , and any

$\epsilon > 0$ , there exists a neural network with one hidden layer,  $N$  neurons, and an appropriate activation function  $\sigma$  such that:

$$|f(x) - \sum_{i=1}^N a_i \sigma(w_i \cdot x + b_i)| < \epsilon$$

$i$

$$|f(x) - \sum_{i=1}^N a_i \sigma(w_i \cdot x + b_i)| < \epsilon$$

Here,

$a_i, w_i, b_i$  are the parameters of the network.

## Equations for Kolmogorov-Arnold Networks (KANs) and Multi-Layer Perceptrons (MLPs)

**Multi-Layer Perceptrons (MLPs) General Equation:**

$$MLP(x)$$

)

$$MLP(x) = (W_3 \circ \sigma_2 \circ W_2 \circ \sigma_1 \circ W_1)(x)$$

Here,

$W_i$  represents the weight matrices, and  $\sigma_i$  represents the activation functions at each layer.

**Shallow MLP:**

$$f(x) \approx \sum$$

$$i$$

$$\frac{1}{N(\epsilon)} \sum_{i=1}^N a_i \sigma(w_i \cdot x + b_i) \quad f(x) \approx \sum_{i=1}^N \frac{1}{N(\epsilon)} a_i \sigma(w_i \cdot x + b_i)$$

**Kolmogorov-Arnold Networks (KANs) General Equation:**

$$KAN(x)$$

$$)$$

$$(\Phi_3 \circ \Phi_2 \circ \Phi_1)(x) \quad KAN(x) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(x) \quad \text{Here,}$$

$\Phi_i$  represents the learnable univariate functions on edges.

**Shallow KAN:**

$$f(x)$$

$$)$$

$$\sum$$

$$q$$

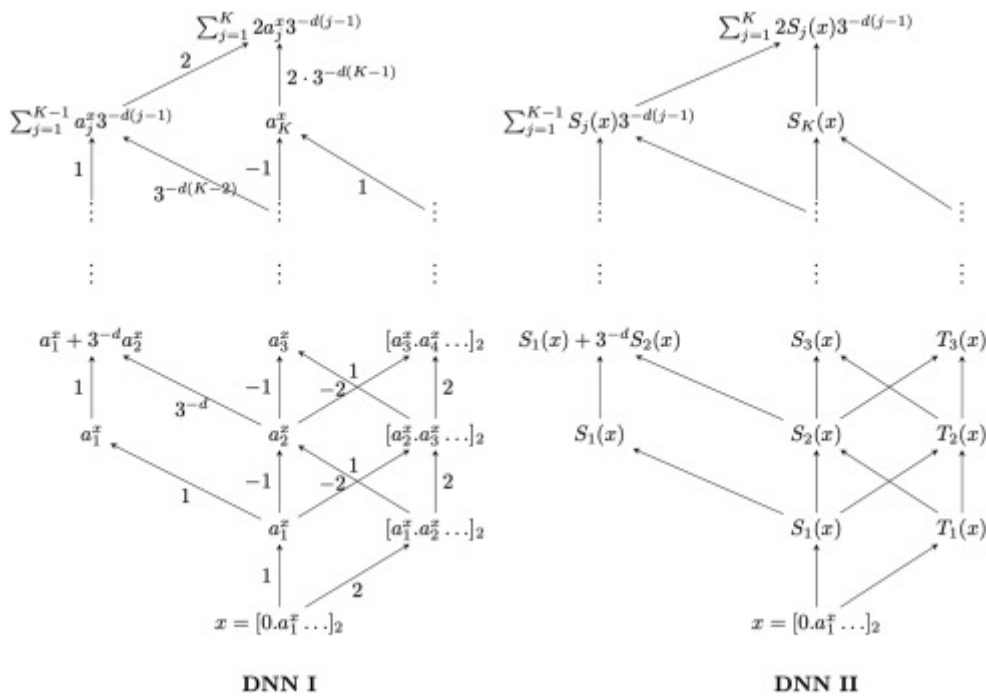
$$\sum_{q=1}^{2n+1} \Phi_q(\sum$$

$$p$$

$$\sum_{q=1}^{2n+1} \Phi_q(\sum_{p=1}^n \varphi_{q,p}(x_p)) \quad f(x) = \sum_{q=1}^{2n+1} \Phi_q(\sum_{p=1}^n \varphi_{q,p}(x_p))$$

# Differences Between Kolmogorov-Arnold Networks (KANs) and Multi-Layer Perceptrons (MLPs)

## Kolmogorov-Arnold Networks (KANs)



**Theoretical Foundation:** Based on the Kolmogorov-Arnold Representation Theorem. This theorem states that any multivariate continuous function can be represented as a finite sum of continuous univariate functions and a finite composition of continuous univariate functions.

#### Structure:

- Nodes perform only summation operations.
- Edges have learnable activation functions.
- Each weight parameter is replaced by a learnable univariate function parameterized as a spline.

**Activation Functions:-** Learnable activation functions on edges (weights).

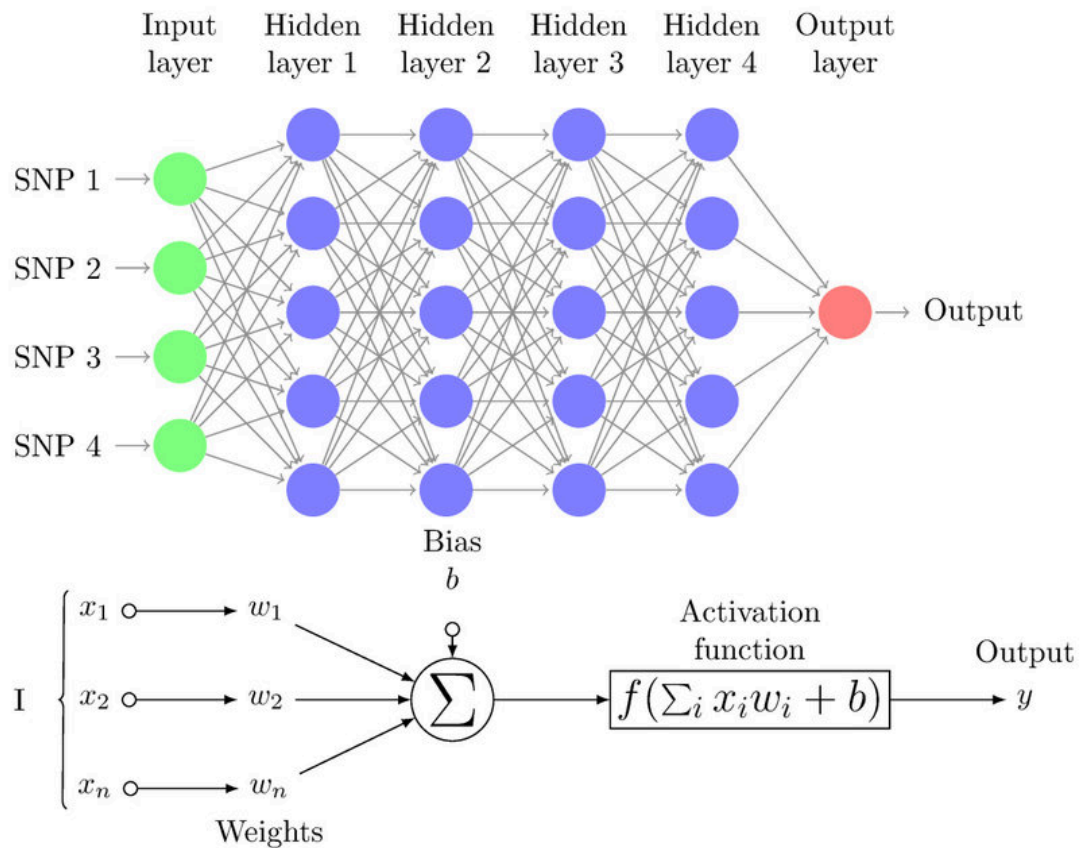
#### Advantages:

- Improved accuracy and interpretability.
- Faster neural scaling laws compared to MLPs.
- Better at learning compositional structures of functions.

KAN Formula:

- $f(x) = \sum(\Phi q(\sum(\phi q, p(xp))))$

## Multi-Layer Perceptrons (MLPs)



### Theoretical Foundation:

- Based on the Universal Approximation Theorem. This theorem states that a feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of  $\mathbb{R}^n$ , given appropriate activation functions.

### Structure:

- Fixed activation functions on nodes.
- Linear weights and biases are used.
- Activation Functions:
  - Fixed, nonlinear activation functions on nodes (e.g., ReLU, sigmoid, tanh).

### Advantages:

- Simple and effective for a wide range of tasks.
- Well-established and widely used in deep learning models.

MLP Formula:

- $f(x) \approx \sum (a_i \sigma(w_i x + b_i))$

## VISUAL SHOWING THEIR KEY DIFFERENCES

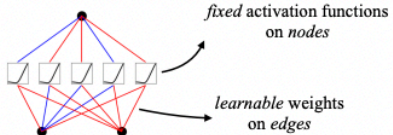
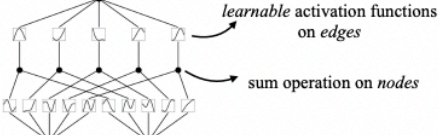
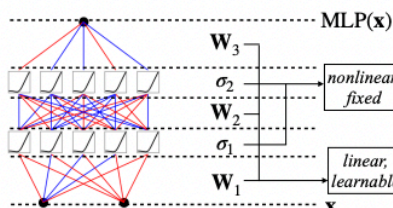
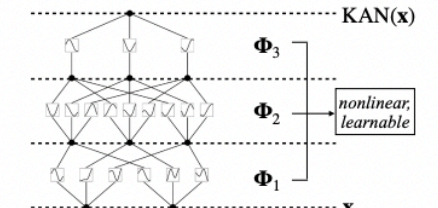
Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(e)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left( \sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) 	(b) 
Formula (Deep)	$\text{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\text{KAN}(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	(c) 	(d) 

Figure 0.1: Multi-Layer Perceptrons (MLPs) vs. Kolmogorov-Arnold Networks (KANs)

## Equation Representation in Python

```
In [1]: #MULTILAYER PERCEPTRON
def mlp(x, weights, biases, activation):
    layer = x
    for w, b in zip(weights, biases):
        layer = activation(np.dot(layer, w) + b)
    return layer

# KAN
def kan(x, spline_functions):
    return sum([Phi(np.sum([spline(x_i) for spline, x_i in zip(splines, x)])) for s
```

## PROJECT WORKSPACE

```
In [2]: #importing dependencies
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

In [3]: #Loading the dataset to be used in this project
# The task is to predict cement strength .I will use MLP and compare it to KAN
df = pd.read_csv("C:\Datasets\Cement_Production\Concrete Compressive Strength.csv")

In [4]: df.isnull().sum()
```



```
Out[4]: Cement                0
Blast Furnace Slag          0
Fly Ash                     0
Water                       0
Superplasticizer            0
Coarse Aggregate            0
Fine Aggregate              0
Age (day)                   0
Concrete compressive strength 0
dtype: int64
```

```
In [6]: # Load dataset

X= df.drop(['Concrete compressive strength'],axis=1)
y=df['Concrete compressive strength'].values

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Data standardization
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)
```

```
In [7]: X_train.shape
```

```
Out[7]: (824, 8)
```

## MODEL BUILDING : KAN AND MLP

```
In [8]: class KolmogorovArnoldNet(nn.Module):
    def __init__(self, n_input, n_hidden, n_output, num_univariate_funcs=2):
        super(KolmogorovArnoldNet, self).__init__()
        self.num_univariate_funcs = num_univariate_funcs
        self.phi = nn.ModuleList([nn.Linear(n_hidden, 1) for _ in range(2*n_input + num_univariate_funcs)])
        self.psi = nn.ModuleList([nn.Linear(n_input, n_hidden) for _ in range(num_univariate_funcs)])

    def forward(self, x):
        psi_outputs = [torch.relu(psi(x)) for psi in self.psi]
        psi_sum = sum(psi_outputs)
        phi_outputs = [phi(psi_sum) for phi in self.phi]
        return sum(phi_outputs)

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(input_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        hidden = torch.relu(self.hidden(x))
        out = self.out(hidden)
```

```

        return out
n_input = X_train.shape[1]
n_hidden = 16
n_output = 1
kan_model = KolmogorovArnoldNet(n_input, n_hidden, n_output)
mlp_model = MLP(n_input, n_hidden, n_output)
print("KAN Model:\n", kan_model)
print("MLP Model:\n", mlp_model)

```

KAN Model:

```

KolmogorovArnoldNet(
  (phi): ModuleList(
    (0): Linear(in_features=16, out_features=1, bias=True)
    (1): Linear(in_features=16, out_features=1, bias=True)
    (2): Linear(in_features=16, out_features=1, bias=True)
    (3): Linear(in_features=16, out_features=1, bias=True)
    (4): Linear(in_features=16, out_features=1, bias=True)
    (5): Linear(in_features=16, out_features=1, bias=True)
    (6): Linear(in_features=16, out_features=1, bias=True)
    (7): Linear(in_features=16, out_features=1, bias=True)
    (8): Linear(in_features=16, out_features=1, bias=True)
    (9): Linear(in_features=16, out_features=1, bias=True)
    (10): Linear(in_features=16, out_features=1, bias=True)
    (11): Linear(in_features=16, out_features=1, bias=True)
    (12): Linear(in_features=16, out_features=1, bias=True)
    (13): Linear(in_features=16, out_features=1, bias=True)
    (14): Linear(in_features=16, out_features=1, bias=True)
    (15): Linear(in_features=16, out_features=1, bias=True)
    (16): Linear(in_features=16, out_features=1, bias=True)
  )
  (psi): ModuleList(
    (0): Linear(in_features=8, out_features=16, bias=True)
    (1): Linear(in_features=8, out_features=16, bias=True)
  )
)
MLP Model:
MLP(
  (hidden): Linear(in_features=8, out_features=16, bias=True)
  (out): Linear(in_features=16, out_features=1, bias=True)
)

```

## MODEL TRAINING

```

In [32]: # Define my loss function and optimizer
          # I will use Adam as my optimizer and Mean Squared Error since it is a Regression Problem

criterion = nn.MSELoss()
mlp_optimizer = optim.Adam(mlp_model.parameters(), lr=0.01)
kan_optimizer = optim.Adam(kan_model.parameters(), lr=0.01)
n_epochs = 300

for epoch in range(n_epochs):
    kan_model.train()
    kan_optimizer.zero_grad()
    outputs = kan_model(X_train_tensor)
    kan_loss = criterion(outputs, y_train_tensor)
    kan_loss.backward()
    kan_optimizer.step()

    mlp_model.train()
    mlp_optimizer.zero_grad()
    outputs = mlp_model(X_train_tensor)

```

```

mlp_loss = criterion(outputs, y_train_tensor)
mlp_loss.backward()
mlp_optimizer.step()

if (epoch+1) % 10 == 0:
    print(f'Epoch [{epoch+1}/{n_epochs}], KAN Loss: {kan_loss.item():.4f}|MLP L

```

```

Epoch [10/300], KAN Loss: 47.1725|MLP Loss: 99.1422
Epoch [20/300], KAN Loss: 44.7791|MLP Loss: 98.7295
Epoch [30/300], KAN Loss: 42.6560|MLP Loss: 98.3389
Epoch [40/300], KAN Loss: 40.9851|MLP Loss: 98.0000
Epoch [50/300], KAN Loss: 39.7505|MLP Loss: 97.7222
Epoch [60/300], KAN Loss: 38.3936|MLP Loss: 97.4701
Epoch [70/300], KAN Loss: 37.0789|MLP Loss: 97.2434
Epoch [80/300], KAN Loss: 35.9207|MLP Loss: 97.0222
Epoch [90/300], KAN Loss: 34.8620|MLP Loss: 96.8259
Epoch [100/300], KAN Loss: 33.8740|MLP Loss: 96.6690
Epoch [110/300], KAN Loss: 33.0092|MLP Loss: 96.5152
Epoch [120/300], KAN Loss: 31.9535|MLP Loss: 96.3309
Epoch [130/300], KAN Loss: 30.8334|MLP Loss: 96.0604
Epoch [140/300], KAN Loss: 29.7940|MLP Loss: 95.9054
Epoch [150/300], KAN Loss: 28.9917|MLP Loss: 95.7644
Epoch [160/300], KAN Loss: 28.3559|MLP Loss: 95.6193
Epoch [170/300], KAN Loss: 27.7983|MLP Loss: 95.3319
Epoch [180/300], KAN Loss: 27.3895|MLP Loss: 95.0512
Epoch [190/300], KAN Loss: 26.7875|MLP Loss: 94.8758
Epoch [200/300], KAN Loss: 26.1968|MLP Loss: 94.6711
Epoch [210/300], KAN Loss: 25.6727|MLP Loss: 94.5397
Epoch [220/300], KAN Loss: 25.1986|MLP Loss: 94.4256
Epoch [230/300], KAN Loss: 24.7998|MLP Loss: 94.2091
Epoch [240/300], KAN Loss: 24.5002|MLP Loss: 92.8996
Epoch [250/300], KAN Loss: 24.1624|MLP Loss: 89.5985
Epoch [260/300], KAN Loss: 23.9310|MLP Loss: 85.7619
Epoch [270/300], KAN Loss: 23.7927|MLP Loss: 82.1559
Epoch [280/300], KAN Loss: 23.5655|MLP Loss: 78.8816
Epoch [290/300], KAN Loss: 23.5952|MLP Loss: 76.0460
Epoch [300/300], KAN Loss: 23.4862|MLP Loss: 73.6138

```

## MODEL EVALUATION

```

In [26]: kan_model.eval()
mlp_model.eval()
with torch.no_grad():
    y_kan_pred = kan_model(X_test_tensor)
    kan_test_loss = criterion(y_kan_pred, y_test_tensor)
    print(f'KAN Test Loss: {kan_test_loss.item():.4f}')
    print("*****")
    y_mlp_pred = mlp_model(X_test_tensor)
    mlp_test_loss = criterion(y_mlp_pred, y_test_tensor)
    print(f'MLP_Test Loss: {mlp_test_loss.item():.4f}')

```

```

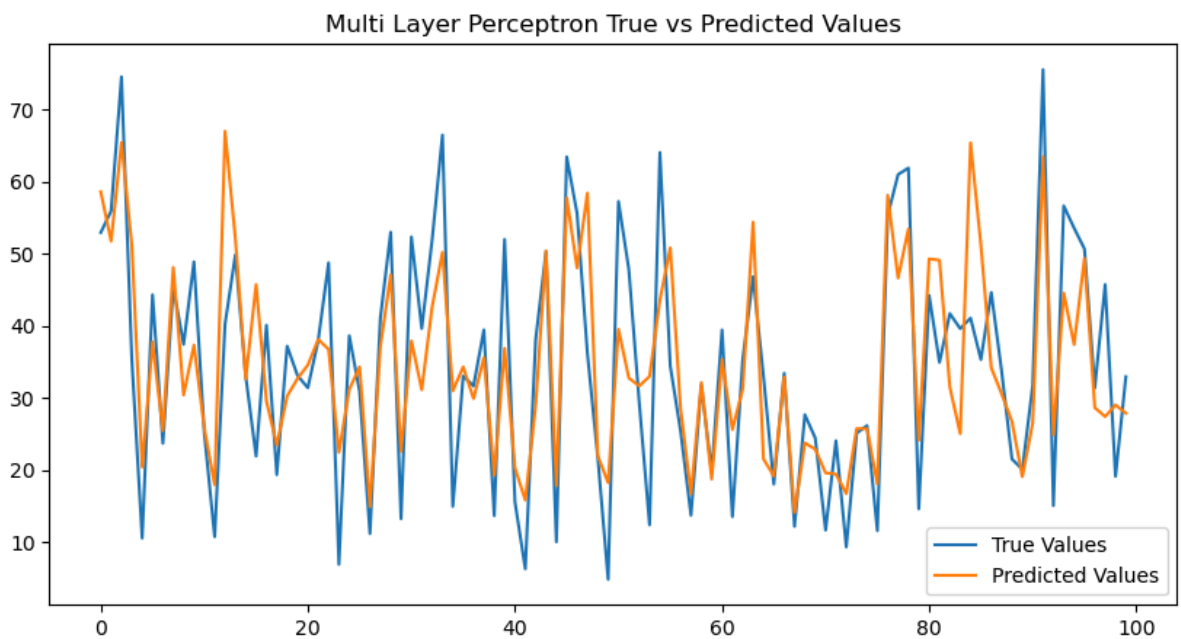
KAN Test Loss: 61.2269
*****
MLP_Test Loss: 97.7450

```

## PLOTTING THE RESULTS

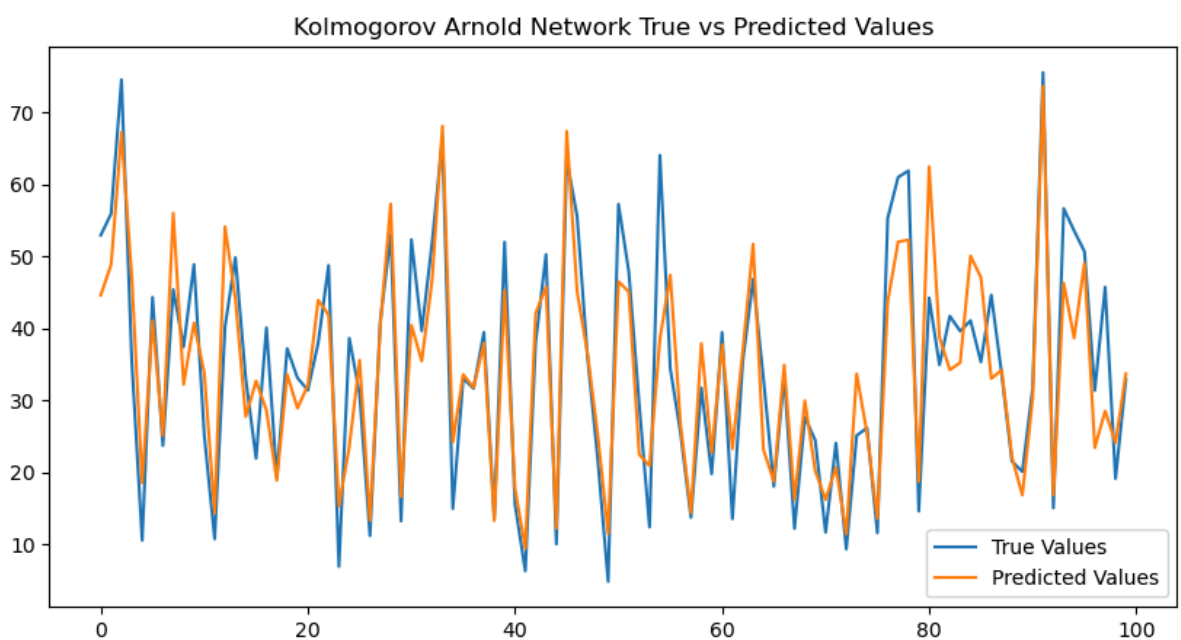
### Multi Layer Perceptron Model(MLP

```
In [31]: plt.figure(figsize=(10,5))
plt.plot(y_test[:100], label='True Values')
plt.plot(y_mlp_pred.numpy()[:100], label='Predicted Values')
plt.legend()
plt.title('Multi Layer Perceptron True vs Predicted Values')
plt.show()
```



## KolmogorovArnold Model

```
In [29]: plt.figure(figsize=(10,5))
plt.plot(y_test[:100], label='True Values')
plt.plot(y_kan_pred.numpy()[:100], label='Predicted Values')
plt.legend()
plt.title('Kolmogorov Arnold Network True vs Predicted Values')
plt.show()
```



```
In [34]: # Loss values for Kolmogorov Arnold Networks (KAN)
kan_losses = [
```

```

47.1725, 44.7791, 42.6560, 40.9851, 39.7505, 38.3936, 37.0789,
35.9207, 34.8620, 33.8740, 33.0092, 31.9535, 30.8334, 29.7940,
28.9917, 28.3559, 27.7983, 27.3895, 26.7875, 26.1968, 25.6727,
25.1986, 24.7998, 24.5002, 24.1624, 23.9310, 23.7927, 23.5655,
23.5952, 23.4862
]

# Loss values for Multi-Layer Perceptron (MLP)
mlp_losses = [
    99.1422, 98.7295, 98.3389, 98.0000, 97.7222, 97.4701, 97.2434,
    97.0222, 96.8259, 96.6690, 96.5152, 96.3309, 96.0604, 95.9054,
    95.7644, 95.6193, 95.3319, 95.0512, 94.8758, 94.6711, 94.5397,
    94.4256, 94.2091, 92.8996, 89.5985, 85.7619, 82.1559, 78.8816,
    76.0460, 73.6138
]

epochs = range(10, 310, 10) # Adjusting the range according to the given epochs

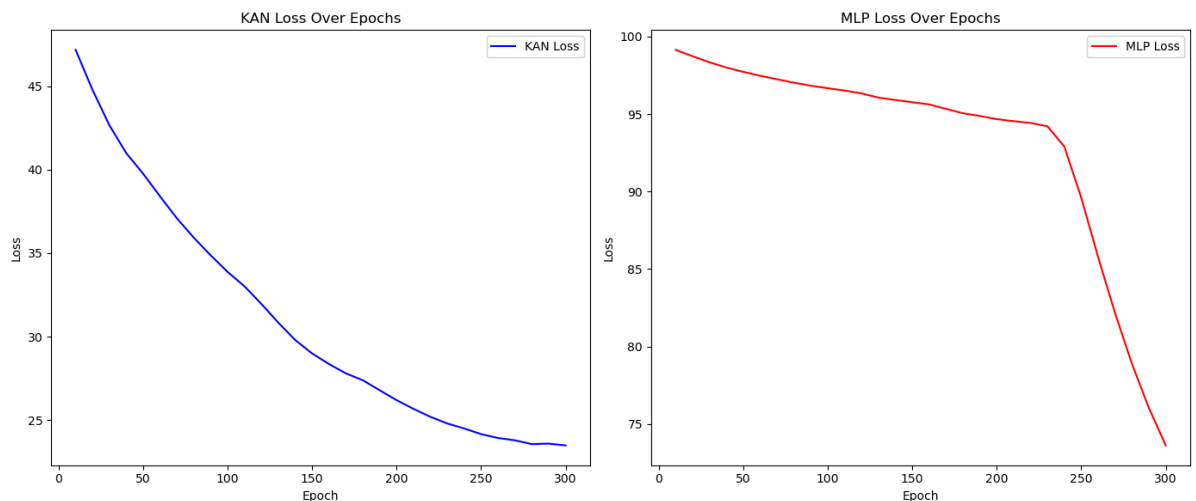
plt.figure(figsize=(14, 6))

# Plotting KAN Losses
plt.subplot(1, 2, 1)
plt.plot(epochs, kan_losses, label='KAN Loss', color='b')
plt.title('KAN Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plotting MLP Losses
plt.subplot(1, 2, 2)
plt.plot(epochs, mlp_losses, label='MLP Loss', color='r')
plt.title('MLP Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

```



## CONCLUSION

This project demonstrates the implementation of Kolmogorov-Arnold Networks using PyTorch on a real-life dataset a Cement Compressive Strength dataset. The networks are

designed based on the Kolmogorov-Arnold representation theorem, offering an innovative approach to function approximation. Conclusion:

The comparative analysis of Kolmogorov Arnold Networks (KAN) and Multi-Layer Perceptron (MLP) models yielded insightful results, underscoring the distinct advantages of each architecture in handling complex datasets.

### **Key Findings:**

#### **Kolmogorov Arnold Networks (KAN):**

The KAN model demonstrated a significant reduction in loss over the training period. Starting from an initial loss of 47.1725, the KAN model's loss decreased steadily, reaching a final loss of 23.4862 after 300 epochs. The consistent decline in loss indicates that the KAN model effectively captured and learned the intricate patterns in the data, showcasing its robustness in dealing with complex dependencies and interactions within the dataset.

**Multi-Layer Perceptron (MLP):** The MLP model exhibited a gradual reduction in loss, starting from 99.1422 and ending at 73.6138 after 300 epochs. Although the MLP model's loss reduction was less dramatic compared to the KAN model, it nonetheless demonstrated its capability to learn and generalize from the data. Conclusion:

- The Kolmogorov Arnold Networks (KAN) model showed superior performance in terms of loss reduction compared to the Multi-Layer Perceptron (MLP) model. The KAN model's ability to achieve a lower final loss underscores its potential in effectively learning from and adapting to complex datasets.
- This project highlights the importance of selecting appropriate neural network architectures based on the specific requirements and complexities of the task at hand. The KAN model's superior performance in this case study suggests that it may be a more suitable choice for tasks involving intricate data patterns and dependencies.
- The insights gained from this comparative analysis will contribute to the advancement of neural network research, providing valuable guidance for future projects and applications. By using the strengths of each architecture, we can optimize model selection and implementation to achieve better performance and accuracy in a wide range of machine learning tasks.

## **REFERENCES**

- <https://arxiv.org/pdf/2404.19756>
- [https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Arnold\\_representation\\_theorem](https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Arnold_representation_theorem)
- <https://towardsdatascience.com/kolmogorov-arnold-networks-kan-e317b1b4d075>

- <https://theaiinsider.tech/2024/05/27/what-are-kolmogorov-arnold-networks-a-guide-to-what-might-be-the-next-big-thing-in-artificial-intelligence/#:~:text=They%20rely%20on%20the%20principles,series%20of%20one%2Ddim>

In [ ]: