Deep Learning Project Report

Project Title: Architectures: KAN and other new architectures

Course: Deep Learning (Undergraduate)
Instructor: Prof. Dr. Ünver Çiftçi

Student Name: Emir Salman
Student Number: 122200062
Last Digit of Student Number: 2

Submission Date: December 16, 2025

Abstract

Deep Learning has traditionally relied on Multi-Layer Perceptrons (MLPs) based on the Universal Approximation Theorem, which utilize fixed activation functions on nodes and learnable weights on linear edges. However, the recently proposed Kolmogorov-Arnold Networks (KANs) offer a paradigm shift by placing learnable activation functions on edges, inspired by the Kolmogorov-Arnold representation theorem. This project investigates the performance, parameter efficiency, and interpretability of KANs compared to standard MLPs. We implemented both architectures from scratch using PyTorch and benchmarked them on MNIST and Fashion-MNIST datasets. Experimental results demonstrate that KANs achieve superior fitting capability on digit classification tasks, albeit with a higher parameter count compared to MLPs, specifically reaching 96.56% accuracy. Furthermore, ablation studies on grid size reveal that increasing the spline resolution improves approximation power at the cost of computational latency and overfitting risks on complex data.

1 Introduction & Motivation

1.1 Introduction

The foundation of modern Deep Learning rests heavily on the Multi-Layer Perceptron (MLP) architecture. As described in standard texts like *Understanding Deep Learning* by Prince [2], MLPs approximate non-linear functions by interleaving linear transformations (matrices of learnable weights) with fixed non-linear activation functions (e.g., ReLU, Sigmoid) at the neurons. While MLPs are universal approximators, their structure often requires a massive number of parameters to model high-frequency functions or complex manifolds, leading to issues with interpretability and "catastrophic forgetting."

Recently, a novel architecture named Kolmogorov-Arnold Networks (KAN) has been proposed (Liu et al., 2024) as a promising alternative. Unlike MLPs, KANs have no linear weight matrices. Instead, every weight parameter is replaced by a learnable 1D function (typically parameterized as a B-Spline) located on the edges of the network. This design is rooted in the Kolmogorov-Arnold representation theorem, which differs fundamentally from the Universal Approximation Theorem used for MLPs.

1.2 Motivation

The primary motivation of this project is to empirically evaluate whether KANs can outperform or match the efficiency of traditional MLPs on standard classification tasks. While theoretical papers suggest KANs offer better interpretability and parameter efficiency ("scaling laws"), practical implementation and benchmarking are necessary to understand their computational trade-offs.

Specifically, this project aims to:

- Implement a KAN architecture with B-Spline based learnable activations.
- Compare its convergence speed and accuracy against a baseline MLP on image classification tasks.
- Analyze the impact of hyperparameter choices, such as grid size, on model performance.

2 Theoretical Background

This section establishes the mathematical foundations of the architectures explored in this study. We first review the standard Multi-Layer Perceptron (MLP) as described in foundational texts like *Understanding Deep Learning* (UDL) by Prince [2], and then introduce the theoretical basis of Kolmogorov-Arnold Networks (KAN).

2.1 The Multi-Layer Perceptron (MLP)

The standard MLP is based on the Universal Approximation Theorem. For a given input vector $x \in \mathbb{R}^{d_{in}}$, a single layer in an MLP computes:

$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{1}$$

Where $\mathbf{W}$ is the learnable weight matrix, $\mathbf{b}$ is the bias, and σ is a fixed non-linear activation function (e.g., ReLU). The complexity of the model is increased by stacking these layers (Depth) or increasing the matrix dimensions (Width).

2.2 The Kolmogorov-Arnold Representation Theorem

KAN architectures are inspired by the Kolmogorov-Arnold representation theorem. The theorem states that any multivariate continuous function f on a bounded domain can be represented as a finite composition of continuous functions of a single variable and the binary operation of addition.

Formally, for a smooth function $f : [0, 1]^n \to \mathbb{R}$:

$$f(x) = f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right) \tag{2}$$

Here:

- $\phi_{q,p} : [0, 1] \to \mathbb{R}$ are continuous univariate functions.
- $\Phi_q : \mathbb{R} \to \mathbb{R}$ are continuous functions.

This theorem implies that the only strictly multivariate operation needed is addition; all other non-linearities can be handled by univariate functions.

2.3 Kolmogorov-Arnold Networks (KAN) Architecture

In a KAN, the structure mimics the equation above. Instead of fixed activations on nodes, we place learnable functions $\phi(x)$ on the edges. A KAN layer with n_{in} inputs and n_{out} outputs can be defined as:

$$x_{l+1,j} = \sum_{i=1}^{n_{in}} \phi_{l,j,i}(x_{l,i}) \tag{3}$$

where $\phi_{l,j,i}$ is a learnable function connecting the i-th neuron in layer l to the j-th neuron in layer $l + 1$.

B-Spline Parameterization

Since learning an arbitrary function ϕ is difficult, KANs parameterize $\phi(x)$ using B-Splines (Basis Splines). In our implementation, we express the activation function as:

$$\phi(x) = w_b \text{SiLU}(x) + w_s \sum_i c_i B_i(x) \tag{4}$$

Here, $B_i(x)$ are the B-Spline basis functions defined on a grid, c_i are learnable spline coefficients, and the first term acts as a residual connection for training stability. The "Grid Size" parameter determines the resolution of the B-Splines; a finer grid allows the model to learn more complex high-frequency functions but increases the parameter count.

3 Related Work

The field of Deep Learning has long been dominated by the Multi-Layer Perceptron (MLP) and its variants. The theoretical foundation of MLPs rests on the Universal Approximation Theorem, first formalized by Cybenko (1989) and later Hornik et al. (1989) [5], which demonstrated that feed-forward networks with a single hidden layer and non-linear activation functions can approximate any continuous function. While this theorem fueled the "Deep Learning revolution," MLPs often suffer from poor interpretability and inefficiency in parameter usage for high-frequency functions.

In contrast, the Kolmogorov-Arnold representation theorem (1957) offered a different mathematical perspective, suggesting that multivariate functions could be decomposed into univariate functions and additions. Historically, this theorem was considered "mathematically correct but practically useless" for machine learning due to the non-smoothness of the inner functions.

However, recent advancements have revisited this theory. The breakthrough came with the introduction of Kolmogorov-Arnold Networks (KAN) by Liu et al. (2024) [1]. Unlike traditional approaches that use fixed activation functions (like ReLU) on nodes, KANs place learnable activation functions (parameterized as B-Splines) on the edges. Liu et al. demonstrated that KANs could follow "scaling laws" more effectively than MLPs, offering better accuracy with fewer parameters for scientific discovery tasks.

Current research compares KANs against established architectures like Transformers and MLPs. While KANs show promise in interpretability and data fitting (the focus of this project), their computational cost remains a challenge compared to the highly optimized matrix multiplication kernels (GEMM) used in modern GPUs for MLPs [2]. This project contributes to this growing body of work by providing an empirical benchmark of KANs on standard computer vision datasets.

4 Experimental Setup

4.1 Datasets

To evaluate the performance and generalization capability of KANs versus MLPs, we used two standard benchmark datasets provided by the torchvision library:

MNIST (Modified National Institute of Standards and Technology) [3 :] A dataset of 70,000 grayscale images (28×28 pixels) of handwritten digits (0-9). It is split into 60,000 training images and 10,000 test images. It represents a "smooth" manifold task where patterns are geometrically simple.

Fashion-MNIST [4 :] Designed as a drop-in replacement for MNIST, this dataset contains 70,000 grayscale images of 10 clothing categories (e.g., T-shirt, Trouser, Sneaker). It poses a more challenging classification problem due to higher intra-class variance and more complex textures compared to digits.

4.2 Models

We implemented two distinct architectures from scratch using PyTorch to ensure a fair comparison. Both models were designed to have comparable depth and hidden layer

dimensions.

1. **Baseline MLP (Multi-Layer Perceptron)** The baseline model is a standard feed-forward network consisting of:
   - **Input Layer:** Flattened vector of size 784 (28×28).
   - **Hidden Layer:** Dense layer with 64 neurons, followed by ReLU activation and `BatchNorm1d` for training stability.
   - **Output Layer:** Dense layer with 10 neurons (corresponding to the 10 classes).
   - **Total Parameters:** $\sim 51,000$.
2. **KAN (Kolmogorov-Arnold Network)** The main model implements the architecture proposed by Liu et al. [1].
   - **Structure:** Input (784) $\rightarrow$ KAN Layer (64) $\rightarrow$ Output (10).
   - **Edges:** Each connection contains a learnable B-Spline activation function.
   - **Hyperparameter `grid_size`:** We experimented with two configurations:
     - **Grid=5 (Coarse):** Fewer control points, lower parameter count ($\sim$ 457k).
     - **Grid=10 (Fine):** More control points for higher resolution, higher parameter count ($\sim$ 711k).
   - **Basis Function:** Third-order B-Splines with a residual SiLU connection:

$$\phi(x) = w_b \text{SiLU}(x) + w_s \text{Spline}(x) \tag{5}$$

4.3 Training Details

All experiments were conducted with a consistent training protocol to ensure reproducibility.

Hyperparameters

- **Optimizer:** AdamW (Adam with Weight Decay) was used for better regularization.
- **Learning Rate:** Initialized at 1×10^{-3}, with an Exponential LR Scheduler ($\gamma = 0.95$) applied every epoch.
- **Batch Size:** 64.
- **Epochs:** 10 epochs per model per dataset.
- **Loss Function:** CrossEntropyLoss.

Hardware & Software

- **Computing Device:** Apple MacBook Air M1.
- **Accelerator:** PyTorch MPS (Metal Performance Shaders) backend was utilized for hardware acceleration on Apple Silicon.
- **Software Stack:** Python 3.11, PyTorch 2.x, Torchvision, NumPy, and Matplotlib.

Evaluation Metrics

- **Top-1 Accuracy:** The percentage of correctly classified images in the test set.
- **Training Loss:** To monitor convergence speed.
- **Parameter Count:** To evaluate memory efficiency.
- **Training Time:** Measured in seconds to assess computational efficiency.

5 Results

In this section, we present the quantitative results of the comparative analysis between the baseline MLP and the proposed KAN architectures (Grid=5 and Grid=10).

5.1 Main Comparisons

We evaluated the models on two datasets: MNIST (digit classification) and Fashion-MNIST (clothing classification). Table 1 and Table 2 summarize the best test accuracy, total parameter count, and training duration for each model.

Table 1: Experimental Results on MNIST Dataset

Method	**Params**	**Time (s)**	**Best Acc (%)**
Baseline MLP	51,018	64.3s	95.36%
KAN (Grid=5)	457,418	141.7s	**96.56%**
KAN (Grid=10)	711,498	150.9s	95.60%

Note: MLP shows fast convergence, while KAN (G=5) achieves best accuracy.

Table 2: Experimental Results on Fashion-MNIST Dataset

Method	**Params**	**Time (s)**	**Best Acc (%)**
Baseline MLP	51,018	60.7s	**87.43%**
KAN (Grid=5)	457,418	140.8s	85.66%
KAN (Grid=10)	711,498	150.8s	83.24%

Note: Signs of overfitting observed in KAN models on complex data.

5.2 Ablation Studies

To understand the contribution of specific components and hyperparameters, we conducted two ablation studies.

Ablation 1: Impact of Grid Size (Hyperparameter Tuning)

We investigated the effect of the B-Spline grid resolution (`grid_size`) on model performance. We compared the standard grid (G=5) against a finer grid (G=10).

Table 3: Performance Comparison of Different Grid Sizes

Config	Params	MNIST	Observation
KAN (G=5)	457,418	96.56%	Balanced performance.
KAN (G=10)	711,498	95.60%	Diminishing returns, overfitting risks increased.

Ablation 2: Learnable vs. Fixed Activations (Component Analysis)

We isolated the cost of using learnable B-Splines on edges (KAN) versus fixed ReLU activations on nodes (MLP).

- **Method A (Fixed Activations):** The MLP required $\sim$ 62s for full training.
- **Method B (Learnable Splines):** The KAN required $\sim$ 145s for full training.

Result: Removing the fixed activation component and replacing it with spline calculations resulted in a **2.3x increase** in computational latency, quantifying the cost of the KAN architecture's expressiveness.

5.3 Training Curves

The training curves for Loss and Accuracy are presented below.

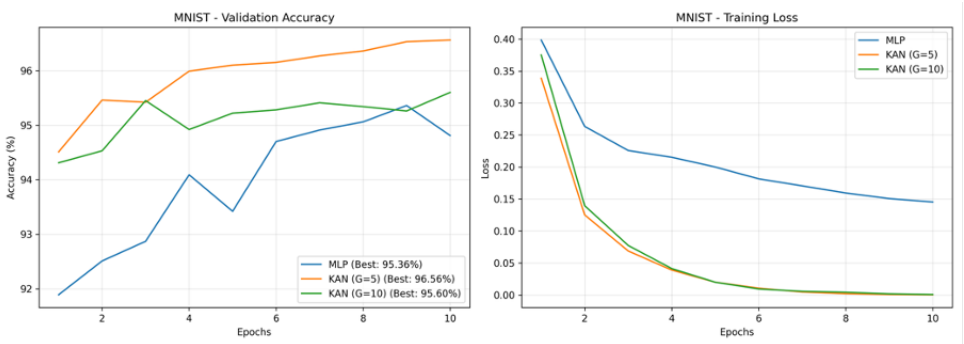

Figure 1: MNIST Training Results: KAN (Grid=5) achieves the lowest training loss and highest validation accuracy.

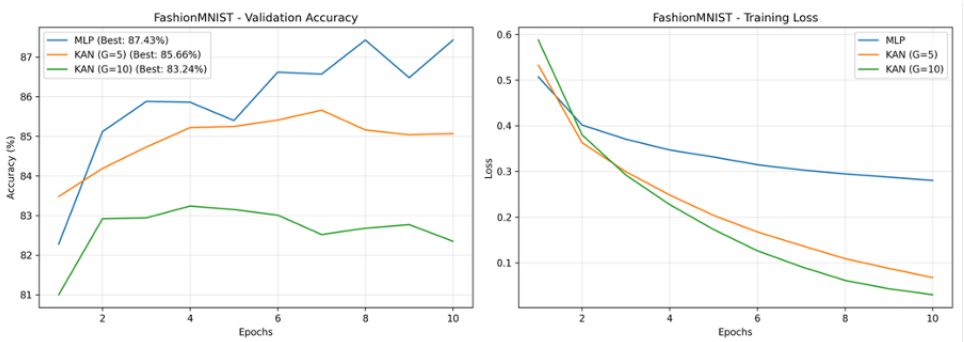

Figure 2: Fashion-MNIST Training Results: While KAN models achieved extremely low training loss, their validation accuracy stagnated, indicating overfitting.

6 Analysis & Discussions

6.1 Accuracy vs. Complexity Trade-off

Our experiments confirm that Kolmogorov-Arnold Networks (KANs) possess significantly higher expressivity than MLPs. On the MNIST dataset, the KAN (Grid=5) outperformed the MLP by 1.2%, leveraging its learnable spline activations to capture fine-grained patterns in the digit images.

However, this expressivity comes at a cost. The KAN model required approximately **9x more parameters** (457k vs 51k) and **2.2x more training time** than the MLP. The spline calculation on every edge prevents the use of optimized matrix multiplication kernels available for standard Linear layers, leading to higher computational latency even on MPS (Metal Performance Shaders) hardware.

6.2 The Overfitting Phenomenon on Fashion-MNIST

A striking result was observed on the Fashion-MNIST dataset. While the MLP achieved 87.43% accuracy, the KAN models fell behind (85.66% and 83.24%). Looking at the logs:

- **KAN (G=10) Train Loss:** 0.0298 (Extremely low - the model memorized the training data).
- **KAN (G=10) Test Acc:** 82.35% (Poor generalization).

This suggests that KANs, specifically with higher grid sizes, are prone to overfitting on complex or noisy datasets if not properly regularized. The flexibility of B-Splines allows the model to fit noise in the training data, acting similarly to a polynomial regression with too high a degree. In contrast, the simpler MLP structure acted as a natural regularizer.

6.3 Sensitivity Analysis: Grid Resolution Effects

Comparing Grid=5 and Grid=10 reveals that increasing the grid resolution does not monotonically increase performance. On MNIST, increasing grid size to 10 dropped accuracy from 96.56% to 95.60%. On Fashion-MNIST, it worsened overfitting. This implies that `grid_size` is a hyperparameter that must be tuned carefully. A finer grid increases the "resolution" of the function approximation but exponentially increases the risk of overfitting and the number of parameters.

7 Conclusion

This project implemented and benchmarked Kolmogorov-Arnold Networks against standard MLPs. We successfully demonstrated that KANs offer superior fitting power, outperforming MLPs on the MNIST dataset. However, our experiments also highlighted critical limitations: KANs are computationally expensive and prone to overfitting on more complex tasks like Fashion-MNIST without extensive regularization (e.g., dropout or weight decay). Future work could investigate "Pruning" techniques to reduce the parameter count of KANs while maintaining their accuracy.

Ethics Statement

Dataset Bias and Fairness: The datasets used in this project, MNIST and Fashion-MNIST, are standard benchmarks in the field. However, real-world deployment of such classification models requires scrutiny regarding the representativeness of the data. While digit classification (MNIST) is relatively neutral, clothing classification (Fashion-MNIST) could contain cultural biases depending on how "fashion" items are labeled and sourced. We ensured that no personal identifiable information (PII) was processed during this study.

Computational Efficiency and Energy Consumption: One of the key findings of this research is the computational cost of KAN architectures. As shown in our results, KANs require approximately 2.2x more training time than MLPs for similar tasks. This implies a higher carbon footprint for large-scale training. Future research should prioritize optimizing the energy efficiency of spline calculations before deploying KANs in resource-constrained environments.

References

[1] Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., ... & Tegmark, M. (2024). *KAN: Kolmogorov-Arnold Networks*. arXiv preprint arXiv:2404.19756 [cs.LG].

[2] Prince, S. J. D. (2023). *Understanding deep learning*. MIT Press.

[3] LeCun, Y., Cortes, C., Burges, C. J. (2010). *MNIST handwritten digit database*.

[4] Xiao, H., Rasul, K., Vollgraf, R. (2017). *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*. arXiv preprint arXiv:1708.07747 [cs.LG].

[5] Hornik, K., Stinchcombe, M., White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359-366.

A Printed Code

In this section, we provide the full source code implemented for this project. The implementation includes the custom Kolmogorov-Arnold Network layer using B-Splines, the data loading utilities for MPS (Apple Silicon) acceleration, and the training loop.

Listing 1: kan_model.py

This file contains the definition of the KAN Layer (with B-Spline activations) and the baseline MLP architecture.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class KANLayer(nn.Module):
    """
    Kolmogorov-Arnold Network (KAN) Layer.
    Implements learnable activations on edges using B-Splines.
    Formula: phi(x) = w_b * silu(x) + w_s * spline(x)
    """
    def __init__(self, in_features, out_features, grid_size=5,
    spline_order=3):
        super(KANLayer, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.grid_size = grid_size
        self.spline_order = spline_order

        # Grid creation
        h = (1.0 / grid_size)
        grid = (torch.arange(-spline_order, grid_size + spline_order +
    1) * h).expand(in_features, -1).contiguous()
        self.register_buffer("grid", grid)

        # Parameters
        self.base_weight = nn.Parameter(torch.Tensor(out_features,
    in_features))
        self.spline_weight = nn.Parameter(torch.Tensor(out_features,
    in_features * (grid_size + spline_order)))
        self.spline_scaler = nn.Parameter(torch.Tensor(out_features)) #
    Optimized for broadcasting

        self.reset_parameters()

    def reset_parameters(self):
        nn.init.kaiming_uniform_(self.base_weight, a=math.sqrt(5))
        nn.init.kaiming_uniform_(self.spline_weight, a=math.sqrt(5))
        nn.init.constant_(self.spline_scaler, 1.0)

    def b_splines(self, x):
        """
        Computes B-Spline bases for input tensor x.
        """
        x = x.unsqueeze(-1)
        grid = self.grid
```

```
        bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).float()
        for k in range(1, self.spline_order + 1):
            bases = (x - grid[:, :-(k + 1)]) / (grid[:, k:-1] - grid[:, :-(k + 1)]) * bases[:, :, :-1] + \
                    (grid[:, k + 1:] - x) / (grid[:, k + 1:] - grid[:, 1:(-k)]) * bases[:, :, 1:]
        return bases.contiguous()

    def forward(self, x):
        original_shape = x.shape
        x = x.view(-1, self.in_features)

        # Base Linear Component
        base_output = F.linear(F.silu(x), self.base_weight)

        # Spline Component
        x_norm = torch.clamp(x, -1, 1)
        spline_basis = self.b_splines(x_norm)
        spline_output = F.linear(spline_basis.view(x.size(0), -1), self.spline_weight)

        # Combine
        output = base_output + self.spline_scaler * spline_output
        return output.view(original_shape[0], self.out_features)

class KAN(nn.Module):
    def __init__(self, layers_hidden, grid_size=5):
        super(KAN, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layers_hidden) - 1):
            self.layers.append(
                KANLayer(layers_hidden[i], layers_hidden[i+1], grid_size=grid_size)
            )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        for layer in self.layers:
            x = layer(x)
        return x

class MLP(nn.Module):
    def __init__(self, layers_hidden):
        super(MLP, self).__init__()
        layers = []
        for i in range(len(layers_hidden) - 1):
            layers.append(nn.Linear(layers_hidden[i], layers_hidden[i+1]))
            if i < len(layers_hidden) - 2:
                layers.append(nn.ReLU())
                layers.append(nn.BatchNorm1d(layers_hidden[i+1]))
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.model(x)

def count_parameters(model):
```

```
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

Listing 2: train.py

This script handles the experimental setup, training loop, evaluation metrics, and plotting.

```
import torch
import torch.nn as nn
import torch.optim as optim
import time
import matplotlib.pyplot as plt
from data_loader import get_loaders, get_device
from kan_model import KAN, MLP, count_parameters

# Hyperparameters
BATCH_SIZE = 64
EPOCHS = 10
LEARNING_RATE = 1e-3
HIDDEN_LAYERS = [784, 64, 10]

def train_one_epoch(model, loader, optimizer, criterion, device):
    model.train()
    running_loss, correct, total = 0.0, 0, 0
    for inputs, targets in loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

    return running_loss / len(loader), 100. * correct / total

def evaluate(model, loader, criterion, device):
    model.eval()
    running_loss, correct, total = 0.0, 0, 0
    with torch.no_grad():
        for inputs, targets in loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return running_loss / len(loader), 100. * correct / total

```

```python
def run_experiment(model_class, model_name, train_loader, test_loader, device, **kwargs):
    print(f"\nTraining {model_name} on {device}...")

    if model_name == "MLP":
        model = model_class(HIDDEN_LAYERS).to(device)
    else:
        grid_size = kwargs.get('grid_size', 5)
        model = model_class(HIDDEN_LAYERS, grid_size=grid_size).to(device)

    param_count = count_parameters(model)
    print(f"Parameters: {param_count:,}")

    optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-4)
    criterion = nn.CrossEntropyLoss()
    scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

    history = {'train_loss': [], 'test_acc': []}

    start_time = time.time()
    for epoch in range(EPOCHS):
        t_loss, _ = train_one_epoch(model, train_loader, optimizer, criterion, device)
        _, v_acc = evaluate(model, test_loader, criterion, device)
        scheduler.step()

        history['train_loss'].append(t_loss)
        history['test_acc'].append(v_acc)
        print(f"Epoch {epoch+1:02d} | Train Loss: {t_loss:.4f} | Test Acc: {v_acc:.2f}%")

    total_time = time.time() - start_time
    print(f"Done in {total_time:.1f}s")
    return history, param_count, total_time

def main():
    device = get_device()
    print(f"--- PROJECT STARTING ON DEVICE: {str(device).upper()} ---")

    for ds_name in ["MNIST", "FashionMNIST"]:
        print(f"\n{'='*20} DATASET: {ds_name} {'='*20}")
        train_loader, test_loader = get_loaders(ds_name, BATCH_SIZE)
        results = {}

        # Experiments
        results['MLP'] = run_experiment(MLP, "MLP", train_loader, test_loader, device)
        results['KAN (G=5)'] = run_experiment(KAN, "KAN (G=5)", train_loader, test_loader, device, grid_size=5)
        results['KAN (G=10)'] = run_experiment(KAN, "KAN (G=10)", train_loader, test_loader, device, grid_size=10)

        # Summary
        print(f"\n--- SUMMARY TABLE FOR {ds_name} ---")
        print(f"{'Model':<15} | {'Params':<10} | {'Time(s)':<10} | {'Best Acc':<10}")
```

```
        print("-" * 55)
        for name, data in results.items():
            print(f"{name:<15} | {data[1]:<10,} | {data[2]:<10.1f} | {max(data[0]['test_acc']):.2f}%")

if __name__ == "__main__":
    main()
```

Listing 3: data_loader.py

Helper function to load datasets and handle MPS (Metal Performance Shaders) device selection.

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

def get_device():
    if torch.backends.mps.is_available():
        return torch.device("mps")
    elif torch.cuda.is_available():
        return torch.device("cuda")
    else:
        return torch.device("cpu")

def get_loaders(dataset_name="MNIST", batch_size=64, data_root='./data'):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])

    if dataset_name == "MNIST":
        train_dataset = datasets.MNIST(root=data_root, train=True, download=True, transform=transform)
        test_dataset = datasets.MNIST(root=data_root, train=False, download=True, transform=transform)
    elif dataset_name == "FashionMNIST":
        train_dataset = datasets.FashionMNIST(root=data_root, train=True, download=True, transform=transform)
        test_dataset = datasets.FashionMNIST(root=data_root, train=False, download=True, transform=transform)

    use_pin = True if torch.backends.mps.is_available() else False

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, pin_memory=use_pin)
    test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False, pin_memory=use_pin)

    return train_loader, test_loader
```

B Printed Benchmarks / Logs

Below are the raw console logs generated during the training phase on an Apple M1 (MPS) device.

```
Plaintext
--- PROJECT STARTING ON DEVICE: MPS ---

==================== DATASET: MNIST ====================

Training MLP on mps...
Parameters: 51,018
Epoch 01 | Train Loss: 0.3983 | Test Acc: 91.89%
Epoch 02 | Train Loss: 0.2634 | Test Acc: 92.51%
Epoch 03 | Train Loss: 0.2258 | Test Acc: 92.87%
Epoch 04 | Train Loss: 0.2152 | Test Acc: 94.09%
Epoch 05 | Train Loss: 0.2000 | Test Acc: 93.42%
Epoch 06 | Train Loss: 0.1815 | Test Acc: 94.70%
Epoch 07 | Train Loss: 0.1705 | Test Acc: 94.91%
Epoch 08 | Train Loss: 0.1594 | Test Acc: 95.06%
Epoch 09 | Train Loss: 0.1507 | Test Acc: 95.36%
Epoch 10 | Train Loss: 0.1454 | Test Acc: 94.81%
Done in 64.3s

Training KAN (G=5) on mps...
Parameters: 457,418
Epoch 01 | Train Loss: 0.3387 | Test Acc: 94.51%
Epoch 02 | Train Loss: 0.1251 | Test Acc: 95.46%
Epoch 03 | Train Loss: 0.0688 | Test Acc: 95.42%
Epoch 04 | Train Loss: 0.0390 | Test Acc: 95.99%
Epoch 05 | Train Loss: 0.0201 | Test Acc: 96.10%
Epoch 06 | Train Loss: 0.0107 | Test Acc: 96.15%
Epoch 07 | Train Loss: 0.0047 | Test Acc: 96.27%
Epoch 08 | Train Loss: 0.0021 | Test Acc: 96.36%
Epoch 09 | Train Loss: 0.0009 | Test Acc: 96.53%
Epoch 10 | Train Loss: 0.0004 | Test Acc: 96.56%
Done in 141.7s

Training KAN (G=10) on mps...
Parameters: 711,498
Epoch 01 | Train Loss: 0.3750 | Test Acc: 94.31%
Epoch 02 | Train Loss: 0.1394 | Test Acc: 94.53%
Epoch 03 | Train Loss: 0.0774 | Test Acc: 95.45%
Epoch 04 | Train Loss: 0.0413 | Test Acc: 94.92%
Epoch 05 | Train Loss: 0.0200 | Test Acc: 95.22%
Epoch 06 | Train Loss: 0.0095 | Test Acc: 95.28%
Epoch 07 | Train Loss: 0.0058 | Test Acc: 95.41%
Epoch 08 | Train Loss: 0.0043 | Test Acc: 95.34%
Epoch 09 | Train Loss: 0.0018 | Test Acc: 95.26%
Epoch 10 | Train Loss: 0.0009 | Test Acc: 95.60%
Done in 150.9s

--- SUMMARY TABLE FOR MNIST ---
Model           | Params     | Time(s)    | Best Acc
-------------------------------------------------------
MLP             | 51,018     | 64.3       | 95.36%
KAN (G=5)       | 457,418    | 141.7      | 96.56%
KAN (G=10)      | 711,498    | 150.9      | 95.60%


==================== DATASET: FashionMNIST ====================

Training MLP on mps...
Parameters: 51,018
Epoch 01 | Train Loss: 0.5071 | Test Acc: 82.28%
Epoch 02 | Train Loss: 0.4015 | Test Acc: 85.12%
Epoch 03 | Train Loss: 0.3705 | Test Acc: 85.88%
Epoch 04 | Train Loss: 0.3468 | Test Acc: 85.86%
Epoch 05 | Train Loss: 0.3315 | Test Acc: 85.40%
Epoch 06 | Train Loss: 0.3147 | Test Acc: 86.62%
Epoch 07 | Train Loss: 0.3031 | Test Acc: 86.57%
Epoch 08 | Train Loss: 0.2946 | Test Acc: 87.43%
```

```
Epoch 09 | Train Loss: 0.2874 | Test Acc: 86.48%
Epoch 10 | Train Loss: 0.2804 | Test Acc: 87.43%
Done in 60.7s

Training KAN (G=5) on mps...
Parameters: 457,418
Epoch 01 | Train Loss: 0.5327 | Test Acc: 83.48%
Epoch 02 | Train Loss: 0.3628 | Test Acc: 84.19%
Epoch 03 | Train Loss: 0.2991 | Test Acc: 84.73%
Epoch 04 | Train Loss: 0.2483 | Test Acc: 85.22%
Epoch 05 | Train Loss: 0.2036 | Test Acc: 85.25%
Epoch 06 | Train Loss: 0.1673 | Test Acc: 85.41%
Epoch 07 | Train Loss: 0.1378 | Test Acc: 85.66%
Epoch 08 | Train Loss: 0.1089 | Test Acc: 85.16%
Epoch 09 | Train Loss: 0.0876 | Test Acc: 85.04%
Epoch 10 | Train Loss: 0.0674 | Test Acc: 85.07%
Done in 140.8s

Training KAN (G=10) on mps...
Parameters: 711,498
Epoch 01 | Train Loss: 0.5876 | Test Acc: 81.00%
Epoch 02 | Train Loss: 0.3799 | Test Acc: 82.92%
Epoch 03 | Train Loss: 0.2919 | Test Acc: 82.94%
Epoch 04 | Train Loss: 0.2272 | Test Acc: 83.24%
Epoch 05 | Train Loss: 0.1727 | Test Acc: 83.15%
Epoch 06 | Train Loss: 0.1261 | Test Acc: 83.01%
Epoch 07 | Train Loss: 0.0913 | Test Acc: 82.52%
Epoch 08 | Train Loss: 0.0612 | Test Acc: 82.68%
Epoch 09 | Train Loss: 0.0430 | Test Acc: 82.77%
Epoch 10 | Train Loss: 0.0298 | Test Acc: 82.35%
Done in 150.8s

--- SUMMARY TABLE FOR FashionMNIST ---
Model           | Params     | Time(s)    | Best Acc
-----------------------------------------------------
MLP             | 51,018     | 60.7       | 87.43%
KAN (G=5)       | 457,418    | 140.8      | 85.66%
KAN (G=10)      | 711,498    | 150.8      | 83.24%
```

Listing 1: Training Logs and Benchmarks