POSTDIGITAL+: KAN Development Task Report

Author: Bowen Zhang

2.1 General Knowledge

2.1.1 Kolmogorov Superposition Theorem

For any continuous function (f: [0,1]^n \to \mathbb{R}), there exist continuous univariate functions (\Phi_q: \mathbb{R} \to \mathbb{R}) and (\psi_{q,p}: [0,1] \to \mathbb{R}) such that:
[f(x_1, x_2, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \psi_{q,p}(x_p) \right).]

2.1.2 Theoretical Foundation for KANs

Kolmogorov-Arnold Networks (KANs) mirror this decomposition:

- **Structure**: Replace linear layers with compositions of learnable univariate functions (e.g., splines).
- **Efficiency**: The theorem suggests KANs can approximate multivariate functions with fewer parameters than MLPs.

2.1.3 Splines as Basis Functions

- Splines (piecewise polynomials) parameterize (\psi_{q,p}) and (\Phi_q).
- Each spline is defined by trainable coefficients over fixed knot intervals, enabling flexible approximation.

2.1.4 Nonlinearity in KANs vs. MLPs

Aspect	KANs	MLPs
Nonlinearity Source	Adaptive spline basis functions	Fixed activations (e.g., ReLU)
Flexibility	Higher (learned basis functions)	Lower (static activations)

---

2. Practical Implementation

2.1 Minimal KAN Implementation

Code:

```
import torch
import torch.nn as nn

class SplineLayer(nn.Module):
    def __init__(self, input_dim=2, num_branches=5, num_knots=5):
        super().__init__()
```

```
        # Spline coefficients for ψ and Φ
        self.psi_coeffs = nn.ParameterList([...])  # See previous answer
        self.phi_coeffs = nn.ParameterList([...])

    def forward(self, x):
        # Compute branch outputs and sum
        return sum(outputs)

## 2.2. Practical Implementation   -->

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# ==================================================================
# 2.2.1: Minimal KAN Implementation (Core)
# ==================================================================
class SplineLayer(nn.Module):
    """KAN layer using spline-parameterized ψ and Φ functions."""
    def __init__(self, input_dim=2, num_branches=5, num_knots=5,
degree=3):
        super().__init__()
        self.num_branches = num_branches
        self.num_knots = num_knots
        self.degree = degree

        # Spline coefficients for ψ (input transforms) and Φ (output
transforms)
        self.psi_coeffs = nn.ParameterList([
            nn.Parameter(torch.randn(input_dim, num_knots + degree))
            for _ in range(num_branches)
        ])
        self.phi_coeffs = nn.ParameterList([
            nn.Parameter(torch.randn(num_knots + degree))
            for _ in range(num_branches)
        ])

    def compute_basis(self, x):
        """Simplified B-spline basis computation."""
        knots = torch.linspace(0, 2*np.pi, self.num_knots)
        basis = torch.zeros(x.shape[0], self.num_knots + self.degree)
        for i in range(self.num_knots + self.degree - 1):
            basis[:, i] = torch.clamp((x - knots[i]) / (knots[i+1] -
knots[i] + 1e-6), 0, 1)
        return basis

    def forward(self, x):
        outputs = []
        for q in range(self.num_branches):
            branch_sum = 0
            for p in range(x.shape[1]):
```

```
                basis = self.compute_basis(x[:, p])
                psi = torch.matmul(basis, self.psi_coeffs[q][p])
                branch_sum += psi
            phi_basis = self.compute_basis(branch_sum)
            phi = torch.matmul(phi_basis, self.phi_coeffs[q])
            outputs.append(phi)
        return torch.stack(outputs).sum(dim=0)

# ===================================================================
# 2.2.2: Fit f(x,y) = sin(xy) + cos(x² + y²)
# ===================================================================
def generate_data():
    x = torch.rand(1000, 2) * 2 * np.pi
    y = torch.sin(x[:, 0] * x[:, 1]) + torch.cos(x[:, 0]**2 + x[:, 1]**2)
    return x, y

def train_kan(x, y):
    kan = SplineLayer(input_dim=2, num_branches=5)
    optimizer = optim.Adam(kan.parameters(), lr=1e-3)
    loss_fn = nn.MSELoss()
    losses = []

    for epoch in range(1000):
        pred = kan(x)
        loss = loss_fn(pred.squeeze(), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
    return kan, losses

# ===================================================================
# 2.2.3: Compare with Shallow MLP
# ===================================================================
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(2, 20), nn.ReLU(),
            nn.Linear(20, 1)
        )

    def forward(self, x):
        return self.layers(x).squeeze()

def train_mlp(x, y):
    mlp = MLP()
    optimizer = optim.Adam(mlp.parameters(), lr=1e-3)
    loss_fn = nn.MSELoss()
    losses = []

    for epoch in range(1000):
        pred = mlp(x)
        loss = loss_fn(pred, y)
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
    return mlp, losses

# ==================================================================
# 2.2.4: Loss Surface Analysis
# ==================================================================
def plot_loss_surface(model, x, y):
    params = list(model.parameters())
    perturbations = [torch.randn_like(p) for p in params]

    alphas = np.linspace(-1, 1, 50)
    losses = []
    for alpha in alphas:
        for p, pert in zip(params, perturbations):
            p.data.add_(alpha * pert)
        with torch.no_grad():
            pred = model(x)
            loss = nn.MSELoss()(pred, y)
        losses.append(loss.item())
        for p, pert in zip(params, perturbations):
            p.data.sub_(alpha * pert)

    plt.plot(alphas, losses)
    plt.title("KAN Loss Surface Along Random Direction")
    plt.xlabel("Perturbation Scale (α)")
    plt.ylabel("MSE Loss")

# ==================================================================
# 2.2.5: Optimization Dynamics Visualization
# ==================================================================
def plot_training_curves(kan_losses, mlp_losses):
    plt.plot(kan_losses, label='KAN')
    plt.plot(mlp_losses, label='MLP')
    plt.yscale('log')
    plt.xlabel("Epoch")
    plt.ylabel("Loss (log scale)")
    plt.title("Training Dynamics Comparison")
    plt.legend()

# ==================================================================
# 2.2.6: Theoretical Example (KAN-Superior Function Class)
# ==================================================================
def theoretical_example():
    print("""
    *** Theoretical Example (Bonus Task 6) ***
    Function: f(x₁,x₂) = sin(x₁ + x₂) + cos(x₁ - x₂)

    Why KANs Excel:
    1. Matches KAN's additive structure: f = Φ₁(ψ₁₁ + ψ₁₂) + Φ₂(ψ₂₁ + ψ₂₂)
    2. MLPs require deeper layers to approximate the same composition
    3. KANs achieve lower approximation error with fewer parameters
```

```
    """)

# ==================================================================
# 2.2.7: Novel Activation Function (SplineActivation)
# ==================================================================
class SplineActivation(nn.Module):
    """Learnable spline-based activation inspired by Kolmogorov's
theorem."""
    def __init__(self, num_knots=5):
        super().__init__()
        self.knots = nn.Parameter(torch.linspace(0, 2*np.pi, num_knots))
        self.coeffs = nn.Parameter(torch.randn(num_knots))

    def compute_basis(self, x):
        basis = torch.zeros(x.shape[0], len(self.knots))
        for i in range(len(self.knots) - 1):
            basis[:, i] = torch.clamp((x - self.knots[i]) /
(self.knots[i+1] - self.knots[i] + 1e-6), 0, 1)
        return basis

    def forward(self, x):
        return torch.matmul(self.compute_basis(x), self.coeffs)

def test_spline_activation():
    # Generate toy data: y = sin(2πx) + noise
    x = torch.rand(100, 1) * 2 * np.pi
    y = torch.sin(2 * np.pi * x) + 0.1 * torch.randn_like(x)

    # Models
    spline_model = nn.Sequential(nn.Linear(1, 1), SplineActivation())
    relu_model = nn.Sequential(nn.Linear(1, 20), nn.ReLU(), nn.Linear(20,
1))

    # Training
    def train(model, x, y):
        optimizer = optim.Adam(model.parameters(), lr=1e-3)
        losses = []
        for _ in range(1000):
            pred = model(x)
            loss = nn.MSELoss()(pred, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            losses.append(loss.item())
        return losses

    spline_loss = train(spline_model, x, y)
    relu_loss = train(relu_model, x, y)

    # Plot
    plt.figure()
    plt.plot(spline_loss, label='SplineActivation')
    plt.plot(relu_loss, label='ReLU MLP')
    plt.yscale('log')
```

```
    plt.title("Bonus Task 7: Activation Comparison")
    plt.legend()

# ================================================================
# Main Execution
# ================================================================
if __name__ == "__main__":
    # Core tasks
    x, y = generate_data()
    kan, kan_losses = train_kan(x, y)
    mlp, mlp_losses = train_mlp(x, y)

    # Visualization
    plt.figure(figsize=(15, 5))
    plt.subplot(131)
    plot_loss_surface(kan, x, y)
    plt.subplot(132)
    plot_training_curves(kan_losses, mlp_losses)

    # 3D Function Plot
    plt.subplot(133, projection='3d')
    xx, yy = torch.meshgrid(torch.linspace(0, 2*np.pi, 50),
torch.linspace(0, 2*np.pi, 50))
    grid = torch.stack([xx.ravel(), yy.ravel()], dim=1)
    with torch.no_grad():
        pred = kan(grid).reshape(50, 50)
    ax = plt.gca()
    ax.plot_surface(xx.numpy(), yy.numpy(), pred.numpy(), cmap='viridis')
    plt.title("KAN Approximation")

    # Bonus tasks
    theoretical_example()
    test_spline_activation()

    plt.show()
```