Three different Mixture of Experts for comparing with MoEBius

This notebook demonstrates how to train three different models for a mixture of experts : 1. A simple mixture of regressions model with fixed proportions. 2. A sparse mixture of experts using ℓ_1 regularization. 3. A sparse mixure of experts using winner take all gating net

The simulated data follows the second model, where the target variable is generated as a mixture of three different regression models.

Mixture of regressions with fixed proportions

- The problem is to predict the target variable, which follows a mixture of three different regression models, where each observation belongs to one of three distinct regression regimes.

- The target variable is generated as follows:
  - Each observation $x \in \mathbb{R}^{10}$ is drawn from one of three different multivariate normal distributions with distinct mean vectors.

  - The corresponding target value y is computed using one of three linear functions that depend on a subset of four features among the ten.

  - The model for y is given by:

    $$y = w_k^\top x_{[1:4]} + \epsilon, \quad \text{where } k \in \{1, 2, 3\}$$

    where w_k represents the regression coefficients for the (k)-th expert, (x_{[1:4]}) denotes the first four features of (x), and ϵ is a Gaussian noise term.

- The training data consists of 300 samples generated from the above process.

- The test data consists of 300 samples generated from the same distribution.
- The evaluation metric for this competition is the **mean squared error (MSE)**.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.decomposition import PCA

# Define a dense linear layer without sparsity
class DenseLinear(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)  # Standard linear layer

    def forward(self, x):
        return self.linear(x)  # Apply linear transformation

# Define the Mixture of Regressions (MoR) model without sparsity
class MixtureOfRegressions(nn.Module):
    def __init__(self, input_dim, output_dim, num_experts=3):
        super().__init__()
        self.num_experts = num_experts  # Number of experts
        self.gate_weights = nn.Parameter(torch.ones(num_experts) / num_experts)  # Fixed prop

        # Create a list of dense linear experts
        self.experts = nn.ModuleList([DenseLinear(input_dim, output_dim) for _ in range(num_e

    def forward(self, x):
        expert_outputs = torch.stack([expert(x) for expert in self.experts], dim=1)  # Compu
        output = torch.sum(self.gate_weights.unsqueeze(0).unsqueeze(-1) * expert_outputs, di
        return output
```

Function to generate synthetic data

Simple function to generate synthetic data, which is a mixture of three different regression models.

```
# Function to generate synthetic training data with three different regression models
def generate_data(n_samples=300, input_dim=10):

    # Define three different mean vectors for normal distributions
    means = [torch.randn(input_dim) * 2 for _ in range(3)]
    cov = torch.eye(input_dim)  # Identity covariance for simplicity

    # Define linear relationships for each expert with different relevant features
    feature_indices = [torch.randperm(input_dim)[:4] for _ in range(3)]  # Select different
    weights = [torch.randn(4, 1) for _ in range(3)]  # Weights for selected features

    # Assign samples to experts
    assignments = torch.randint(0, 3, (n_samples,))

    X = torch.stack([torch.mv(cov, torch.randn(input_dim)) + means[assignments[i]] for i in
    y = torch.stack([torch.matmul(X[i, feature_indices[assignments[i]]], weights[assignments

    return X, y, assignments
```

Function to visualize the data

```
# Function to visualize data using 3D PCA
def visualize_data(X, y, assignments):
    pca = PCA(n_components=2)
    X_reduced = pca.fit_transform(X.numpy())
    y = y.numpy().flatten()

    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')
    scatter = ax.scatter(X_reduced[:, 0], X_reduced[:, 1], y, c=assignments.numpy(), cmap='v

    ax.set_xlabel("Principal Component 1")
    ax.set_ylabel("Principal Component 2")
    ax.set_zlabel("Target Variable y")
    ax.set_title("3D PCA Projection of the Data")
    fig.colorbar(scatter, label="Expert Assignment")
    plt.show()
```

Training Process

The training process involves the following steps:

1. **Data Generation**: Synthetic training data is generated using a mixture of three different regression models. Each sample is assigned to one of the three experts, and the corresponding target value is computed using the linear function of the assigned expert.
2. **Model Initialization**: The Mixture of Regressions (MoR) model is initialized with three experts. Each expert is a dense linear layer, and the gate weights are fixed to ensure equal contribution from each expert.
3. **Training Loop**: The model is trained using the Adam optimizer and Mean Squared Error (MSE) loss function. The training loop runs for a specified number of epochs, and in each epoch, the following steps are performed:
   - Reset gradients.
   - Perform a forward pass to compute the model output.
   - Compute the loss by comparing the model's output with the true target values.
   - Perform backpropagation to compute the gradients.
   - Update the model parameters using the optimizer.
4. **Loss Monitoring**: The training loss is monitored and printed every 100 epochs to track the model's performance.
5. **Visualization**: After training, the training loss is plotted to visualize the loss over epochs. Additionally, the learned weights of each expert are printed and visualized using a heatmap to understand the feature importance for each expert.
6. **Validation**: The model is evaluated on a validation dataset to select the best penalty parameter (lambda) for the Lasso model. A grid search is performed over a range of lambda values, and the lambda that gives the best performance on the validation dataset is selected.

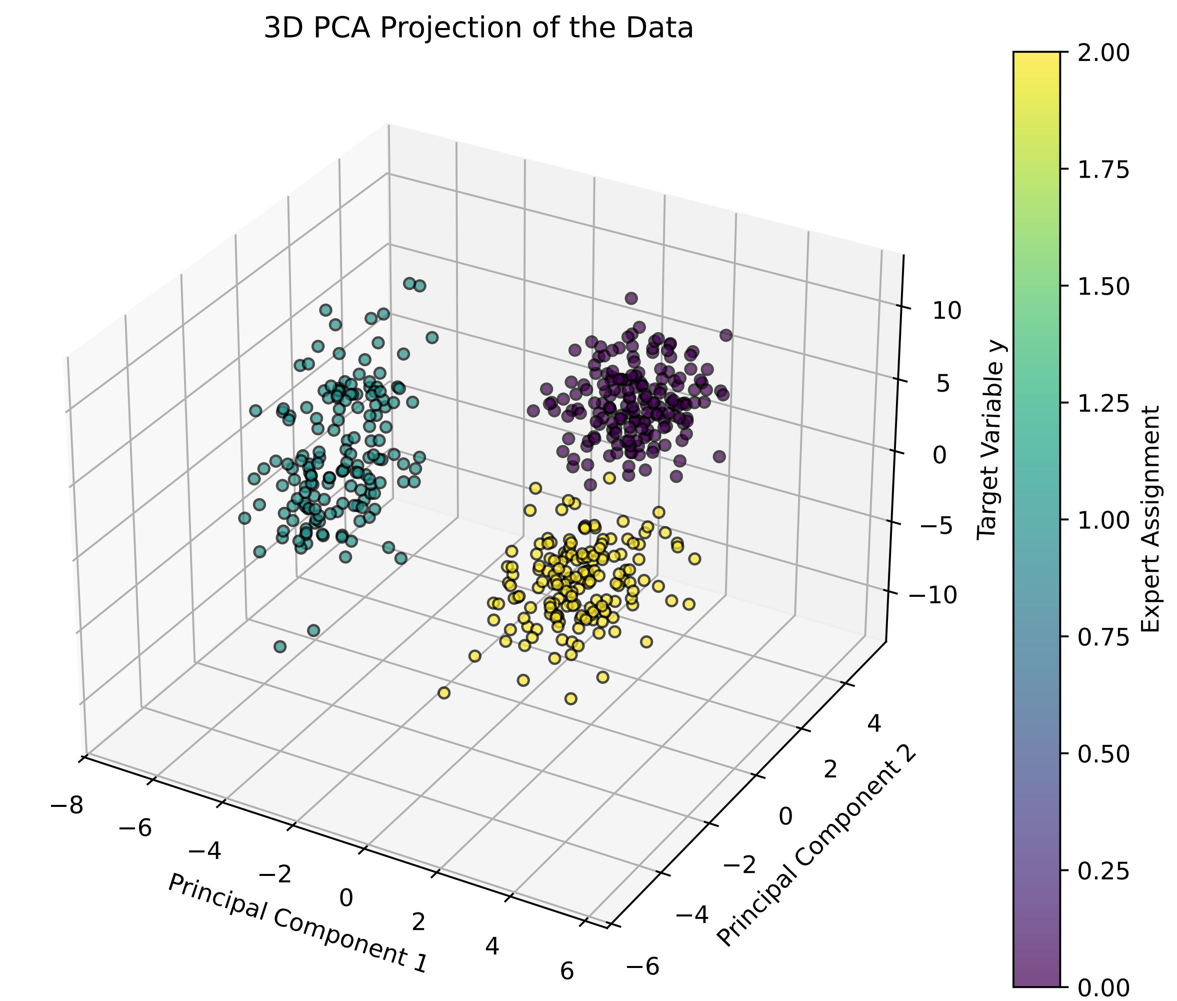
The training process aims to minimize the MSE loss and learn the optimal weights for each expert to accurately predict the target values.

```python
# Function to plot training loss
def plot_training_loss(losses):
    plt.figure(figsize=(8, 6))
    plt.plot(losses, label="Training Loss", color='b')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Loss Over Training Epochs")
    plt.legend()
    plt.show()
```

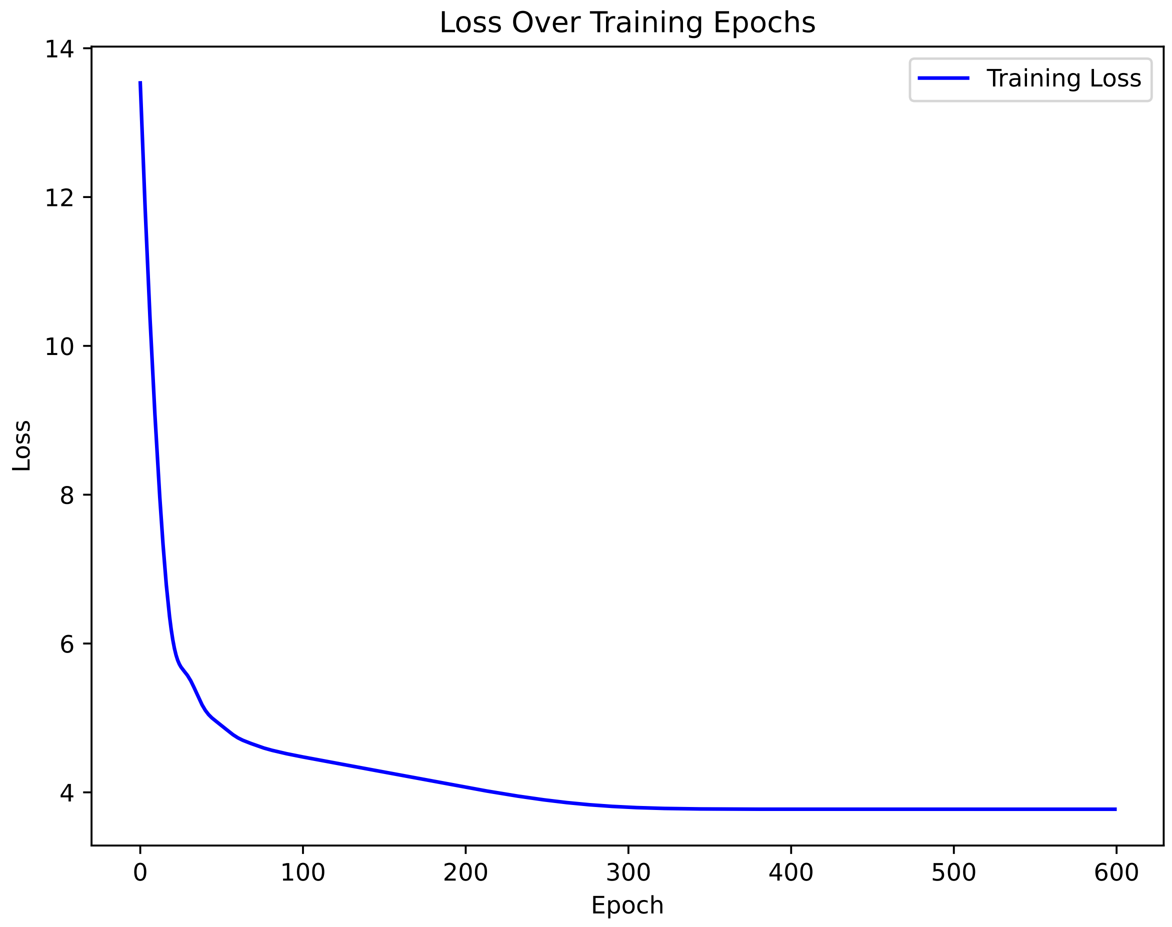
```
# Function to plot expert weights
def plot_expert_weights(model):
    # Print learned feature importance
    for i, expert in enumerate(model.experts):
        print(f"Expert {i} weights:", expert.linear.weight.data)

    # Plot expert weights in a matrix
    weights_matrix = np.array([expert.linear.weight.detach().numpy().flatten() for expert in
    plt.figure(figsize=(8, 6))
    sns.heatmap(weights_matrix, annot=True, cmap='coolwarm', xticklabels=[f'Feature {i}' for
    plt.xlabel("Input Features")
    plt.ylabel("Experts")
    plt.title("Expert Weights Heatmap")
    plt.show()
```

```
criterion = nn.MSELoss()
# Function to train the model
def train_model(X_train, y_train, model, epochs=600, lr=0.01):
    optimizer = optim.Adam(model.parameters(), lr=lr)  # Adam optimizer
    criterion = nn.MSELoss()  # Mean Squared Error loss

    losses = []
    for epoch in range(epochs):
        optimizer.zero_grad()  # Reset gradients
        outputs = model(X_train)  # Forward pass
        loss = criterion(outputs, y_train)  # Compute loss (MSE only)
        loss.backward()  # Backpropagation
        optimizer.step()  # Update model parameters
        losses.append(loss.item())

        # Print loss every 100 epochs
        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

    # Plot training loss
    plot_training_loss(losses)

    # Plot expert weights
    plot_expert_weights(model)


# Generate data
```

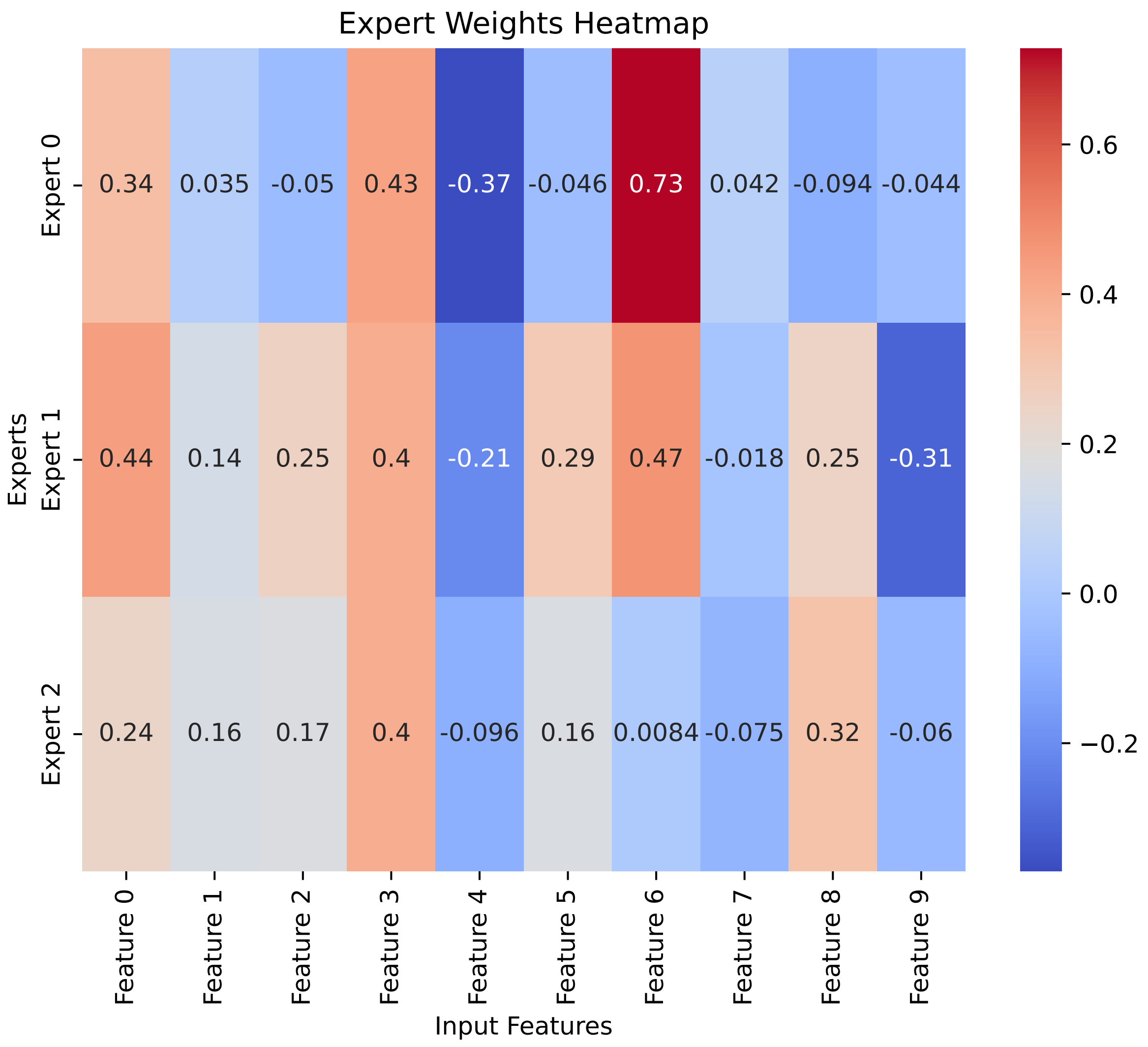
```
torch.manual_seed(42)  # Set seed for reproducibility
X, y, assignments = generate_data(n_samples=1500, input_dim=10)
# Split data into training, validation, and test sets
X_train, y_train = X[:500], y[:500]
X_val, y_val = X[500:1000], y[500:1000]
X_test, y_test = X[1000:], y[1000:]



# Visualize data
visualize_data(X_train, y_train, assignments[:500])



# Initialize the model
model = MixtureOfRegressions(input_dim=10, output_dim=1, num_experts=3)

# Train the model
train_model(X_train, y_train, model)
```



```
Epoch 0, Loss: 13.5339
Epoch 100, Loss: 4.4752
Epoch 200, Loss: 4.0680
Epoch 300, Loss: 3.7996
Epoch 400, Loss: 3.7732
Epoch 500, Loss: 3.7731
```

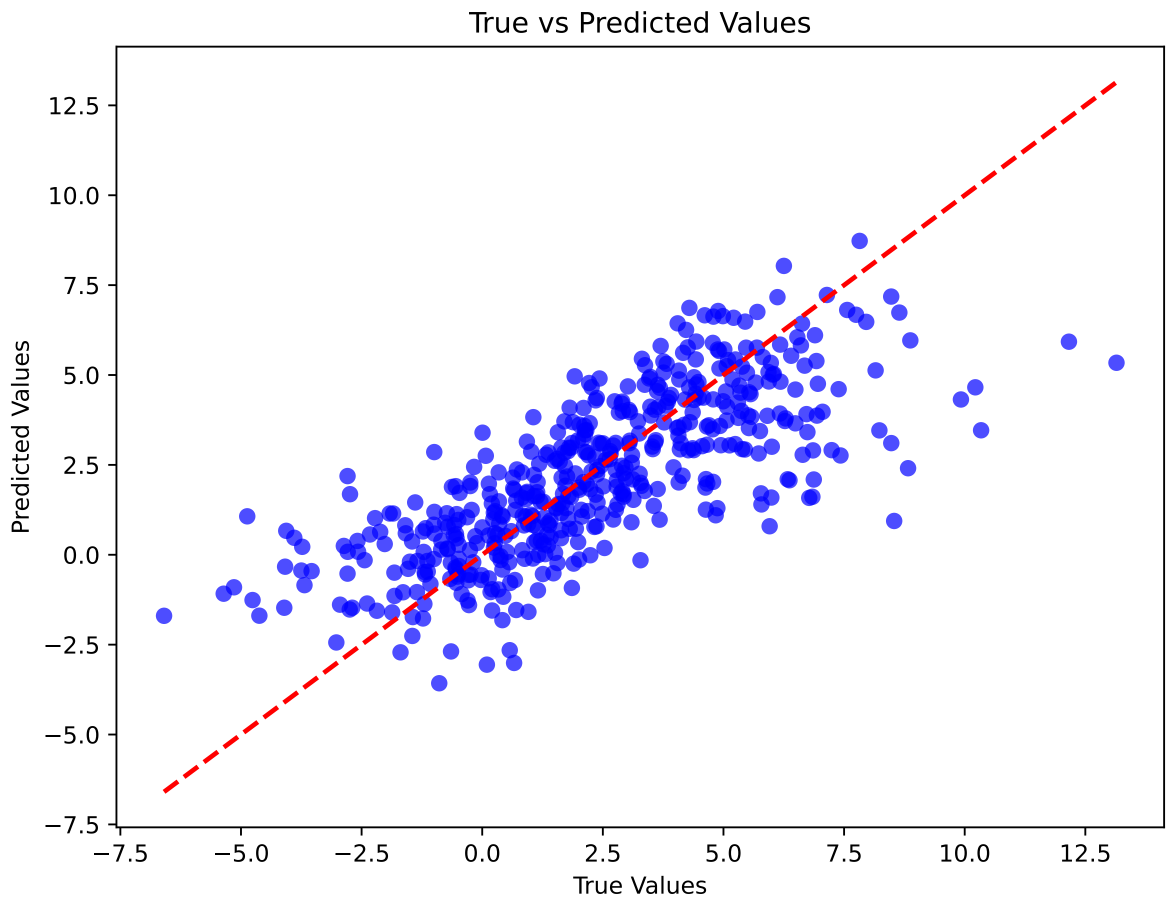


```
Expert 0 weights: tensor([[ 0.3390,  0.0354, -0.0498,  0.4312, -0.3712, -0.0458,  0.7284,  0
         -0.0944, -0.0445]])
Expert 1 weights: tensor([[ 0.4399,  0.1378,  0.2521,  0.3951, -0.2113,  0.2908,  0.4701, -0
          0.2498, -0.3107]])
Expert 2 weights: tensor([[ 0.2395,  0.1570,  0.1675,  0.3963, -0.0956,  0.1635,  0.0084, -0
          0.3213, -0.0603]])
```



```
# Evaluate the model on the test data
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    test_outputs = model(X_test)
    test_loss = criterion(test_outputs, y_test)

print(f"Test Loss: {test_loss.item():.4f}")

# Plot the true vs predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test.numpy(), test_outputs.numpy(), alpha=0.7, color='b')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("True vs Predicted Values")
plt.show()
```

```
Test Loss: 3.7904
```



Simple Sparse Mixture of Experts

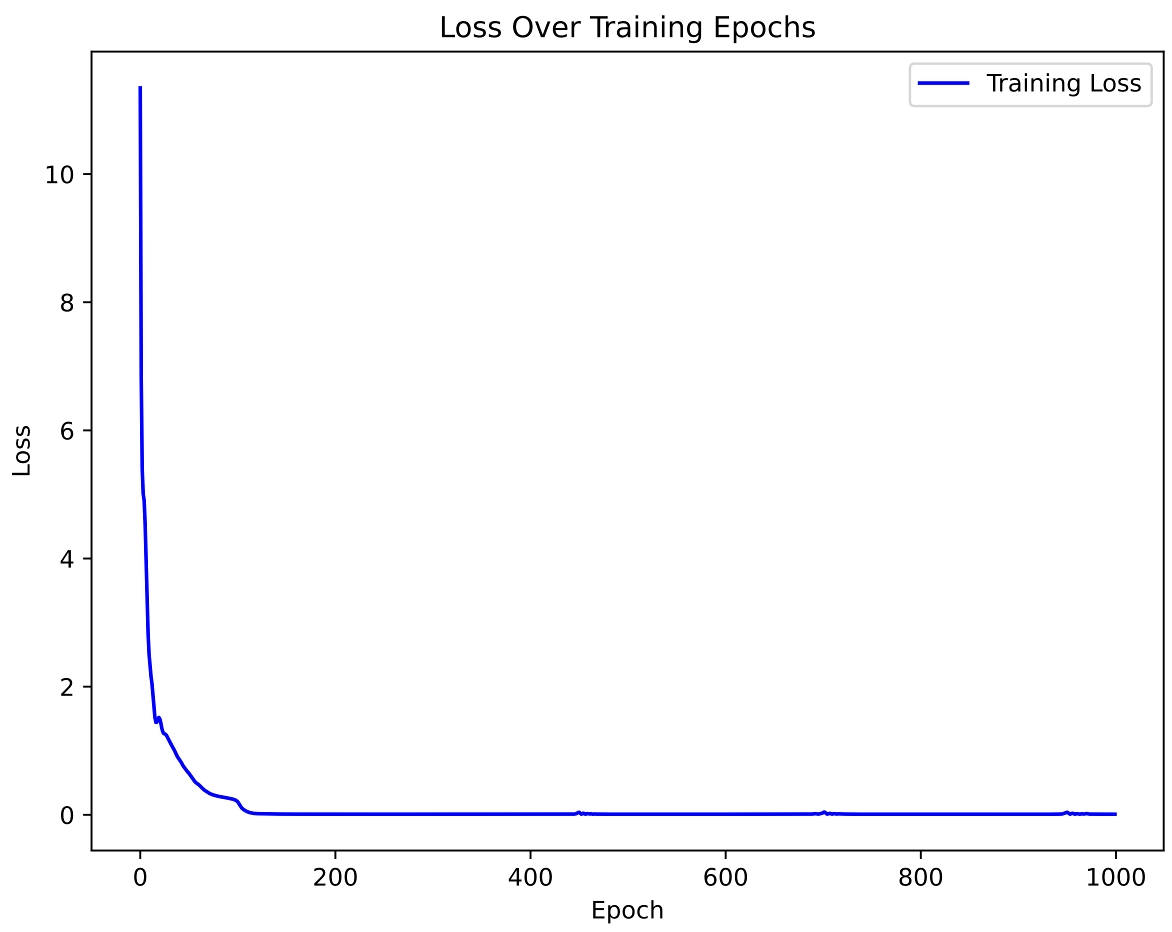
This is a simple implementation of a sparse mixture of experts model. The model is trained using a simple gradient descent algorithm on a simple synthetic dataset. The model is then used to predict the output of the test dataset.

```python
# Define a sparse linear layer with L1 regularization
class SparseLinear(nn.Module):
    def __init__(self, input_dim, output_dim, l1_lambda=10):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)  # Standard linear layer
        self.l1_lambda = l1_lambda  # L1 regularization coefficient

    def forward(self, x):
        return self.linear(x)  # Apply linear transformation

    def l1_loss(self):
        # Compute L1 loss (sum of absolute values of weights)
        return self.l1_lambda * torch.norm(self.linear.weight, p=1)

# Define the Sparse Mixture of Experts (MoE) model
class SparseMoE(nn.Module):
    def __init__(self, input_dim, output_dim, num_experts=3, l1_lambda=0.01):
        super().__init__()
        self.num_experts = num_experts  # Number of experts
        self.gate = nn.Linear(input_dim, num_experts)  # Gating network (determines expert we

        # Create a list of sparse linear experts
        self.experts = nn.ModuleList([SparseLinear(input_dim, output_dim, l1_lambda) for _ in

    def forward(self, x):
        gate_scores = torch.softmax(self.gate(x), dim=1)  # Compute softmax-based gating weig
        expert_outputs = torch.stack([expert(x) for expert in self.experts], dim=1)  # Compu
        output = torch.sum(gate_scores.unsqueeze(-1) * expert_outputs, dim=1)  # Weighted sum
        return output

    def l1_loss(self):
        # Sum L1 losses of all experts
        return sum(expert.l1_loss() for expert in self.experts)
```

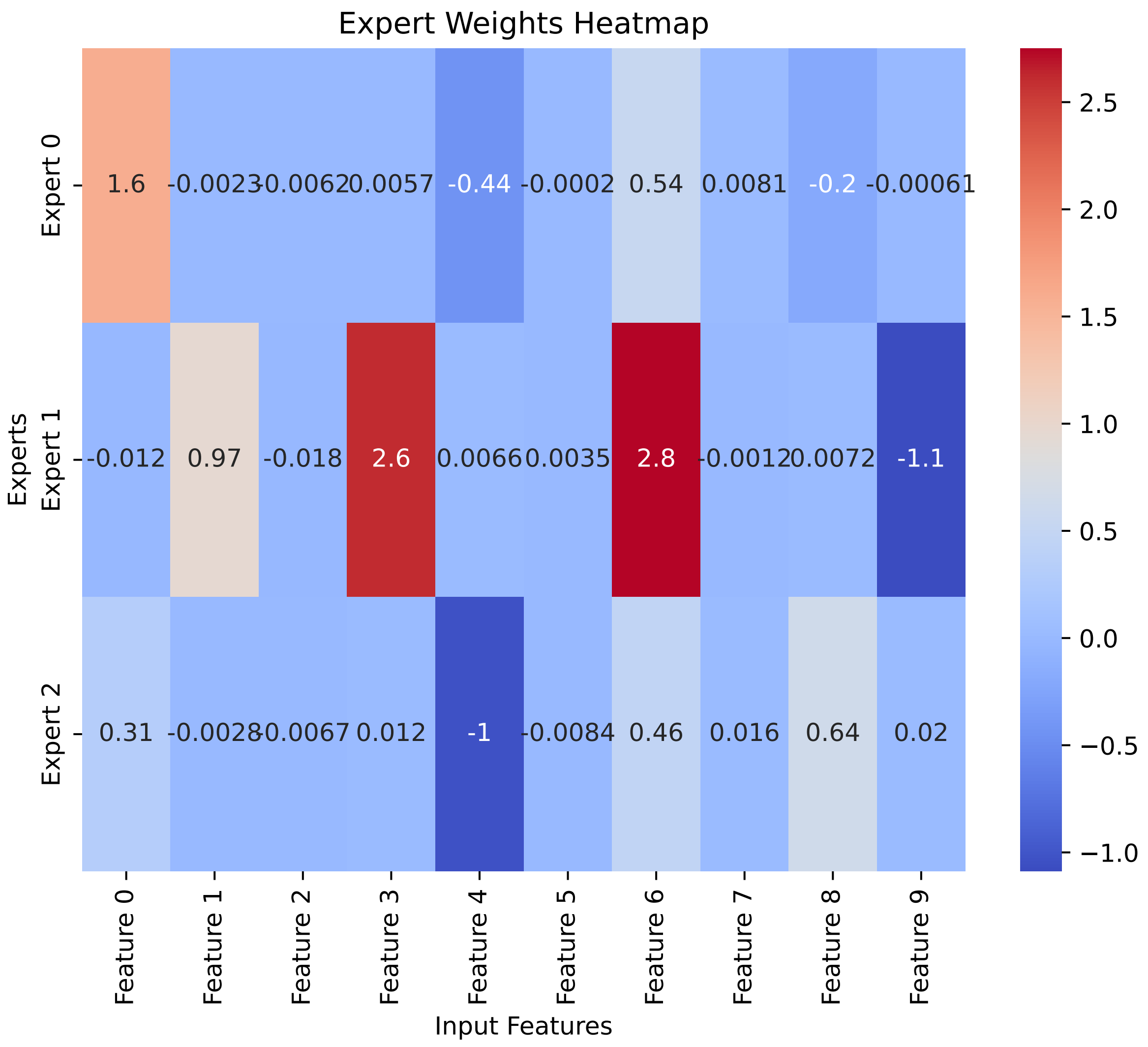
```python
# Initialize the model
model = SparseMoE(input_dim=10, output_dim=1, num_experts=3)

# Train the model
train_model(X_train, y_train, model, epochs=1000, lr=0.1)

# Evaluate the model on the test data
model.eval()  # Set the model to evaluation mode
```

```python
with torch.no_grad():
    test_outputs = model(X_test)
    test_loss = criterion(test_outputs, y_test)

print(f"Test Loss: {test_loss.item():.4f}")

# Plot the true vs predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test.numpy(), test_outputs.numpy(), alpha=0.7, color='b')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("True vs Predicted Values")
plt.show()
```

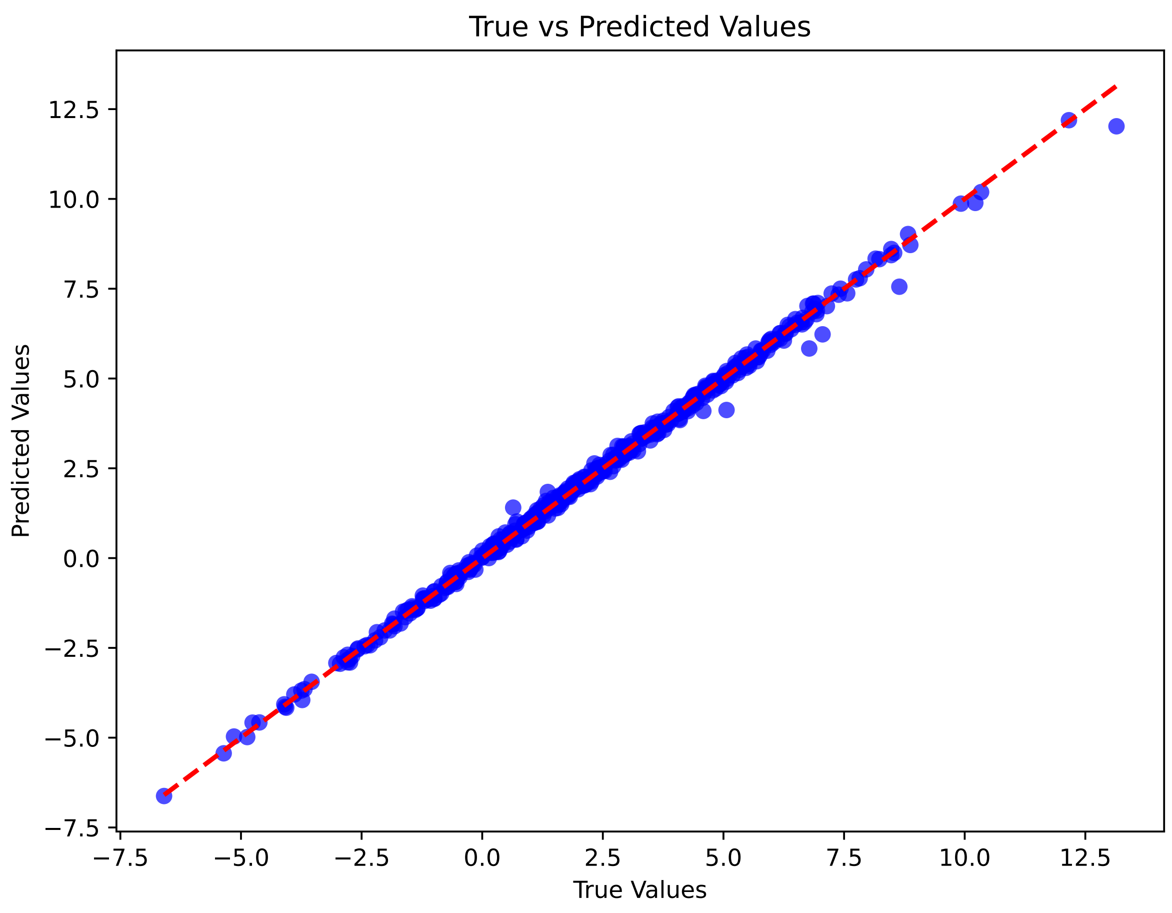
```
Epoch 0, Loss: 11.3449
Epoch 100, Loss: 0.2015
Epoch 200, Loss: 0.0099
Epoch 300, Loss: 0.0095
Epoch 400, Loss: 0.0094
Epoch 500, Loss: 0.0093
Epoch 600, Loss: 0.0092
Epoch 700, Loss: 0.0357
Epoch 800, Loss: 0.0091
Epoch 900, Loss: 0.0091
```



```
Expert 0 weights: tensor([[ 1.5888e+00, -2.2763e-03, -6.2341e-03,  5.7183e-03, -4.3521e-01,
         -2.0141e-04,  5.4162e-01,  8.0671e-03, -2.0207e-01, -6.1031e-04]])
Expert 1 weights: tensor([[-1.1793e-02,  9.6587e-01, -1.7848e-02,  2.6096e+00,  6.6091e-03,
          3.4504e-03,  2.7510e+00, -1.1868e-03,  7.2092e-03, -1.0882e+00]])
Expert 2 weights: tensor([[ 0.3144, -0.0028, -0.0067,  0.0118, -1.0390, -0.0084,  0.4592,  0
          0.6424,  0.0197]])
```



```
Test Loss: 0.0235
```



Selection of the penalty parameter in the Lasso model

The penalty parameter in the Lasso model is selected using a validation dataset. The penalty parameter is selected using a grid search over a range of values. The penalty parameter that gives the best performance on the validation dataset is selected.

```
lambdas = [0.001, 0.01, 0.1, 1, 10]  # Liste de valeurs candidates
best_lambda = None
best_loss = float("inf")

# Recherche du meilleur lambda
for lmbd in lambdas:
    model = SparseMoE(input_dim=10, output_dim=1, num_experts=3, l1_lambda=lmbd)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    criterion = nn.MSELoss()  # ou BCEWithLogitsLoss pour classification
    # Entraînement rapide sur les données
    for epoch in range(100):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train) + model.l1_loss()
        loss.backward()
        optimizer.step()

    # Évaluation sur l'ensemble de validation
    with torch.no_grad():
        val_outputs = model(X_val)
        val_loss = criterion(val_outputs, y_val)

    if val_loss < best_loss:
        best_loss = val_loss
        best_lambda = lmbd

print(f"Meilleur lambda sélectionné : {best_lambda}")
```

```
Meilleur lambda sélectionné : 0.01
```

Sparse Mixture of Experts with sparse gating network

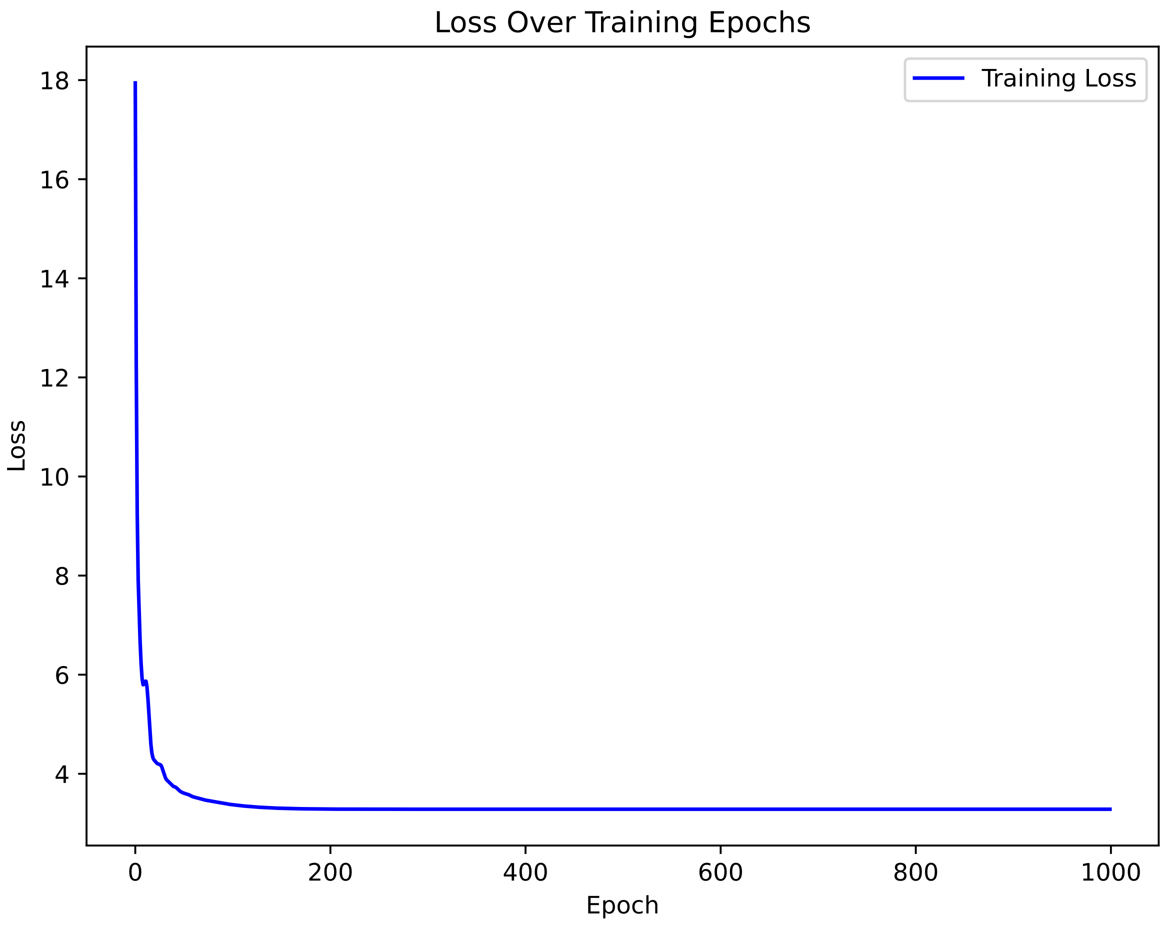
This is a simple implementation of a sparse mixture of experts model with a sparse gating network. The sparse gating network follows a winner-takes-all strategy, meaning that only the best expert (the one with the highest gating score) is selected for each input. The model is trained using a simple gradient descent algorithm on a synthetic dataset. The model is then used to predict the output of the test dataset.

```
# Define a sparse linear layer with L1 regularization
class SparseLinear(nn.Module):
    def __init__(self, input_dim, output_dim, l1_lambda=10):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)  # Standard linear layer
        self.l1_lambda = l1_lambda  # L1 regularization coefficient

    def forward(self, x):
        return self.linear(x)  # Apply linear transformation

    def l1_loss(self):
        # Compute L1 loss (sum of absolute values of weights)
        return self.l1_lambda * torch.norm(self.linear.weight, p=1)

# Define the Sparse Mixture of Experts (MoE) model with Hard Gating (Winner-Takes-All)
class SparseMoE_WTA(nn.Module):
    def __init__(self, input_dim, output_dim, num_experts=3, l1_lambda=0.1):
        super().__init__()
        self.num_experts = num_experts  # Number of experts
        self.gate = nn.Linear(input_dim, num_experts)  # Gating network (determines expert se

        # Create a list of sparse linear experts
        self.experts = nn.ModuleList([SparseLinear(input_dim, output_dim, l1_lambda) for _ in

    def forward(self, x):
        gate_scores = self.gate(x)  # Compute gating scores
        selected_expert_idx = torch.argmax(gate_scores, dim=1)  # Select the best expert (ha

        # Collect outputs from the selected expert only
        output = torch.zeros(x.shape[0], self.experts[0].linear.out_features, device=x.devic
        for i in range(self.num_experts):
            mask = (selected_expert_idx == i).unsqueeze(1).float()
            output += mask * self.experts[i](x)

        return output

    def l1_loss(self):
        # Sum L1 losses of all experts
        return sum(expert.l1_loss() for expert in self.experts)
```

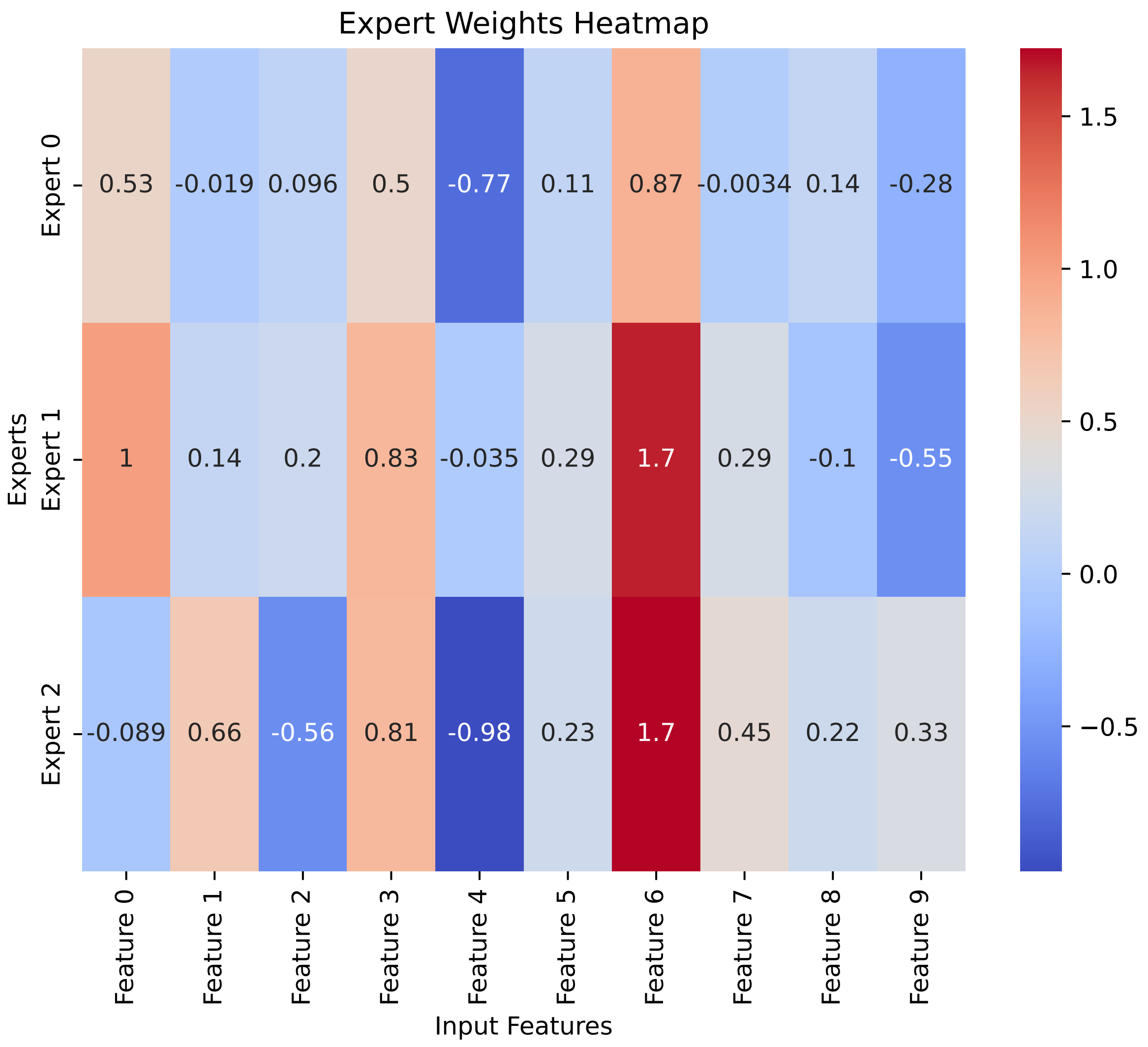
```
# Initialize the model
model = SparseMoE_WTA(input_dim=10, output_dim=1, num_experts=3)

# Train the model
train_model(X_train, y_train, model, epochs=1000, lr=0.1)

# Evaluate the model on the test data
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    test_outputs = model(X_test)
    test_loss = criterion(test_outputs, y_test)

print(f"Test Loss: {test_loss.item():.4f}")
```

```python
# Plot the true vs predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test.numpy(), test_outputs.numpy(), alpha=0.7, color='b')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("True vs Predicted Values")
plt.show()
```

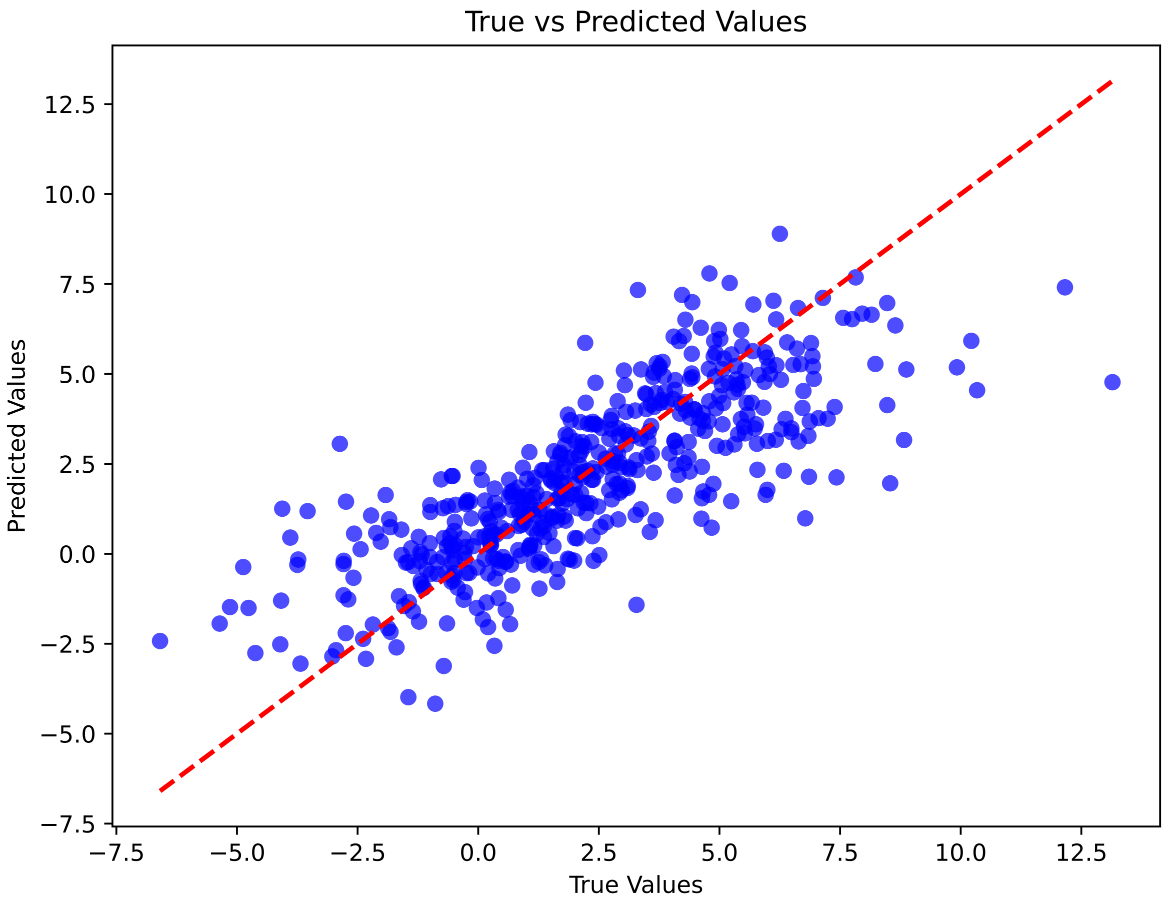
```
Epoch 0, Loss: 17.9430
Epoch 100, Loss: 3.3748
Epoch 200, Loss: 3.2876
Epoch 300, Loss: 3.2851
Epoch 400, Loss: 3.2850
Epoch 500, Loss: 3.2850
Epoch 600, Loss: 3.2850
Epoch 700, Loss: 3.2850
Epoch 800, Loss: 3.2850
Epoch 900, Loss: 3.2854
```



```
Expert 0 weights: tensor([[ 0.5287, -0.0189,  0.0959,  0.5004, -0.7705,  0.1103,  0.8743, -0
          0.1364, -0.2768]])
Expert 1 weights: tensor([[ 1.0133,  0.1373,  0.2031,  0.8287, -0.0354,  0.2889,  1.6537,  0
         -0.1021, -0.5508]])
Expert 2 weights: tensor([[-0.0892,  0.6594, -0.5551,  0.8139, -0.9753,  0.2325,  1.7226,  0
          0.2153,  0.3315]])
```



```
Test Loss: 3.2859
```


True vs Predicted Values
Predicted Values
True Values
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5