MLP SHAP Dashboard

Interactive Neural Network Visualization & Explanation Tool

Overview

1. Interactive Neural Network Visualization
2. SHAP (SHapley Additive exPlanations) Integration
3. Real-time Model Performance Analysis
4. Customizable Architecture & Parameters
5. Multiple Dataset Support

Key Features

- 🧠 Interactive Neural Network Architecture
- 📊 Real-time Performance Metrics
- 🎯 SHAP Value Explanations
- 🔄 Dynamic Parameter Tuning
- 📈 Live Training Visualization

Code Structure

```
# Main application structure
app/
├── app.py           # Main Streamlit application
├── utils.py         # Visualization utilities
├── train_models.py # Model training functions
└── pages/
    ├── 1_🧠_Neural_Network_Visualization.py
    └── 2_📊_SHAP_Analysis.py
```

Dataset Loading & Preprocessing (1/2)

```python
import streamlit as st
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer, load_iris, load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def load_dataset(name):
    """
    Load and preprocess selected dataset.

    Args:
        name: Name of the dataset to load

    Returns:
        Preprocessed train/test split data
    """
    # Load selected dataset
    if name == "Breast Cancer":
        data = load_breast_cancer(as_frame=True)
    elif name == "Iris":
        data = load_iris(as_frame=True)
    elif name == "Wine":
        data = load_wine(as_frame=True)
```

Dataset Loading & Preprocessing (2/2)

```
    # Extract features and target
    X = data.data
    y = data.target

    # Store feature names for later use
    st.session_state['feature_names'] = X.columns.tolist()
    st.session_state['target_names'] = data.target_names.tolist()

    # Standardize features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Split data into train/test sets
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y,
        test_size=0.2,
        random_state=42,
        stratify=y
    )

    return X_train, X_test, y_train, y_test, data
```

Neural Network Architecture (1/3)

```python
from sklearn.neural_network import MLPClassifier

# Model Configuration
def create_model(hidden_layers, activation, learning_rate, max_iter):
    """
    Create and configure MLPClassifier model.

    Args:
        hidden_layers: Tuple of integers for nodes in each hidden layer
        activation: Activation function ('relu', 'tanh', or 'logistic')
        learning_rate: Initial learning rate
        max_iter: Maximum number of iterations

    Returns:
        Configured MLPClassifier model
    """
    return MLPClassifier(
        hidden_layer_sizes=hidden_layers,
        activation=activation,
        learning_rate_init=learning_rate,
        max_iter=max_iter,
        random_state=42,
        early_stopping=True,
        validation_fraction=0.1,
        n_iter_no_change=10
    )
```

Neural Network Architecture (2/3)

```python
# Parameter Configuration UI
def configure_network():
    """Configure neural network architecture through Streamlit UI."""
    st.sidebar.header("Neural Network Architecture")

    # Number of hidden layers
    num_hidden_layers = st.sidebar.slider(
        "Number of Hidden Layers",
        1, 3, 1,
        help="More layers can learn more complex patterns"
    )

    # Nodes per layer
    hidden_layers = []
    for i in range(num_hidden_layers):
        nodes = st.sidebar.slider(
            f"Nodes in Hidden Layer {i+1}",
            5, 200, 50,
            step=5,
            help=f"Layer {i+1} capacity"
        )
        hidden_layers.append(nodes)
```

Neural Network Architecture (3/3)

```
# Training parameters
st.sidebar.header("Training Parameters")

activation = st.sidebar.selectbox(
    "Activation Function",
    ["relu", "tanh", "logistic"],
    help="Choose activation function"
)

learning_rate = st.sidebar.slider(
    "Learning Rate",
    0.0001, 0.1, 0.001,
    step=0.0005,
    format="%.4f",
    help="Step size for gradient descent"
)

max_iter = st.sidebar.slider(
    "Max Iterations",
    100, 1000, 300,
    step=100,
    help="Maximum training iterations"
)

return hidden_layers, activation, learning_rate, max_iter
```

Network Visualization (1/4)

```python
import graphviz

def visualize_neural_network(num_features, hidden_layers,
                             num_classes, model=None,
                             sample_input=None,
                             activations=None):
    """
    Generate interactive neural network visualization.

    Args:
        num_features: Number of input features
        hidden_layers: List of integers for nodes in each hidden layer
        num_classes: Number of output classes
        model: Trained MLPClassifier model (optional)
        sample_input: Input sample for visualization (optional)
        activations: List of layer activations (optional)
    """
    # Create digraph
    dot = graphviz.Digraph('neural_network')
    dot.attr(rankdir='LR')  # Left to right layout

    # Node styling
    dot.attr('node', shape='circle', style='filled')
```

Network Visualization (2/4)

```
# Colors for different layers
colors = {
    'input': '#A8E6CF',   # Light green
    'hidden': '#FFD3B6',  # Light orange
    'output': '#FF8B94'   # Light red
}

# Function to get node color based on activation
def get_activation_color(activation_value):
    if activation_value > 0.7:
        return '#FF0000'  # Strong activation (red)
    elif activation_value > 0.3:
        return '#FFA500'  # Medium activation (orange)
    else:
        return '#CCCCCC'  # Weak activation (gray)

# Add input layer nodes
for i in range(num_features):
    label = f'x{i+1}'
    if sample_input is not None:
        label += f'\n{sample_input[i]:.2f}'
    fillcolor = colors['input']
    if activations is not None and sample_input is not None:
        fillcolor = get_activation_color(abs(sample_input[i]))
    dot.node(f'i{i}', label, fillcolor=fillcolor)
```

Network Visualization (3/4)

```
    # Add hidden layer nodes
    for l, layer_size in enumerate(hidden_layers, 1):
        for i in range(layer_size):
            label = f'h{l}_{i}'
            fillcolor = colors['hidden']
            if activations is not None and len(activations) >= l:
                try:
                    activation_value = activations[l-1].flatten()[i]
                    fillcolor = get_activation_color(abs(activation_value))
                    label += f'\n{activation_value:.2f}'
                except (IndexError, AttributeError):
                    pass
            dot.node(f'h{l}_{i}', label, fillcolor=fillcolor)

    # Add output layer nodes
    for i in range(num_classes):
        label = f'y{i+1}'
        fillcolor = colors['output']
        if activations is not None and len(activations) > 0:
            try:
                activation_value = activations[-1].flatten()[i]
                fillcolor = get_activation_color(abs(activation_value))
                label += f'\n{activation_value:.2f}'
            except (IndexError, AttributeError):
                pass
        dot.node(f'o{i}', label, fillcolor=fillcolor)
```

Network Visualization (4/4)

```
    # Add edges with weights if model is provided
    if model is not None and hasattr(model, 'coefs_'):
        weights = model.coefs_
        max_weight = max(abs(w).max() for w in weights)

        # Helper function for weight colors
        def get_color_for_weight(weight, vmin=-1, vmax=1):
            norm_weight = (weight - vmin) / (vmax - vmin)
            if weight > 0:
                return f"#{int(norm_weight * 255):02x}{int(norm_weight * 200):02x}{int(norm_weight * 200):02x}"
            else:
                return f"#{int(-norm_weight * 200):02x}{int(-norm_weight * 200):02x}{int(-norm_weight * 255):02x}"

        # Add edges for all layers
        for l in range(len(weights)):
            layer_weights = weights[l]
            start_nodes = range(layer_weights.shape[0])
            end_nodes = range(layer_weights.shape[1])

            for i in start_nodes:
                for j in end_nodes:
                    try:
                        weight = layer_weights[i, j]
                        color = get_color_for_weight(weight, -max_weight, max_weight)
                        width = str(0.1 + 2.0 * abs(weight) / max_weight)

                        # Determine node names based on layer
                        if l == 0:
                            start, end = f'i{i}', f'h1_{j}'
                        elif l == len(weights) - 1:
                            start = f'h{l}_{i}'
                            end = f'o{j}'
                        else:
                            start = f'h{l}_{i}'
                            end = f'h{l+1}_{j}'

                        dot.edge(start, end, color=color, penwidth=width)
                    except IndexError:
                        continue

    return dot
```

SHAP Integration (1/3)

```python
import shap

def calculate_shap_values(model, X, X_background=None):
    """
    Calculate SHAP values for model predictions.

    Args:
        model: Trained MLPClassifier model
        X: Features to explain
        X_background: Background dataset for explainer

    Returns:
        SHAP values and explainer object
    """
    # Create background dataset if not provided
    if X_background is None:
        X_background = shap.sample(X, 100)

    # Create explainer
    explainer = shap.KernelExplainer(
        model.predict_proba,
        X_background,
        link="logit"
    )

    # Calculate SHAP values
    shap_values = explainer.shap_values(X)

    return shap_values, explainer
```

SHAP Integration (2/3)

```
def plot_shap_summary(shap_values, feature_names, class_names):
    """Generate and display SHAP summary plots."""

    st.subheader("SHAP Summary Plots")

    # For each class
    for i, class_shap in enumerate(shap_values):
        st.write(f"Class: {class_names[i]}")

        # Summary plot
        fig, ax = plt.subplots(figsize=(10, 6))
        shap.summary_plot(
            class_shap,
            feature_names=feature_names,
            show=False
        )
        st.pyplot(fig)
        plt.close()

        # Bar plot
        fig, ax = plt.subplots(figsize=(10, 6))
        shap.plots.bar(
            class_shap,
            feature_names=feature_names,
            show=False
        )
        st.pyplot(fig)
        plt.close()
```

SHAP Integration (3/3)

```
def plot_shap_waterfall(shap_values, feature_names,
                        sample_idx, class_idx):
    """Generate waterfall plot for specific prediction."""

    st.subheader(f"Detailed Analysis for Sample {sample_idx}")

    # Create waterfall plot
    fig, ax = plt.subplots(figsize=(10, 6))
    shap.plots.waterfall(
        shap_values[class_idx][sample_idx],
        feature_names=feature_names,
        show=False
    )
    st.pyplot(fig)
    plt.close()

    # Add force plot
    st_shap = st.container()
    with st_shap:
        shap.plots.force(
            shap_values[class_idx][sample_idx],
            feature_names=feature_names,
            matplotlib=False,
            show=False
        )
```

Performance Monitoring (1/2)

```
def monitor_performance(model, X_train, X_test,
                        y_train, y_test):
    """
    Monitor and display model performance metrics.

    Args:
        model: Trained MLPClassifier model
        X_train, X_test: Training and test features
        y_train, y_test: Training and test targets
    """
    # Calculate metrics
    train_acc = model.score(X_train, y_train)
    test_acc = model.score(X_test, y_test)

    # Training history
    loss_curve = model.loss_curve_
    validation_scores = model.validation_scores_
```

Performance Monitoring (2/2)

```
# Display metrics
col1, col2 = st.columns(2)

with col1:
    st.metric(
        "Training Accuracy",
        f"{train_acc:.2%}",
        f"{train_acc - 0.5:.2%} vs baseline"
    )

with col2:
    st.metric(
        "Testing Accuracy",
        f"{test_acc:.2%}",
        f"{test_acc - train_acc:.2%} vs training"
    )

# Plot learning curves
fig, ax = plt.subplots(figsize=(10, 6))
epochs = range(1, len(loss_curve) + 1)

ax.plot(epochs, loss_curve, 'b-', label='Training Loss')
if validation_scores:
    ax.plot(epochs, validation_scores, 'r-',
            label='Validation Score')

ax.set_xlabel('Epoch')
ax.set_ylabel('Loss / Score')
ax.legend()

st.pyplot(fig)
plt.close()
```

Activation Function Analysis (1/2)

```python
def suggest_activation_function(X, y):
    """
    Analyze dataset characteristics and suggest activation function.

    Args:
        X: Input features
        y: Target values

    Returns:
        dict: Suggested activation functions and explanations
    """
    # Calculate dataset characteristics
    n_samples, n_features = X.shape
    n_classes = len(np.unique(y))

    # Advanced data analysis
    has_negative = (X < 0).any()
    data_range = np.ptp(X, axis=0).mean()
    correlation = np.corrcoef(X.T)
    avg_correlation = np.abs(correlation - np.eye(n_features)).mean()
```

Activation Function Analysis (2/2)

```python
    # Additional metrics
    skewness = np.mean([np.abs(np.mean(X[:, i]))
                        for i in range(n_features)])
    sparsity = np.mean(X == 0)

    # Make suggestions
    suggestion = {
        'hidden_layer': None,
        'output_layer': 'softmax' if n_classes > 2 else 'sigmoid',
        'explanation': []
    }

    # Decision logic for hidden layer activation
    if has_negative and avg_correlation > 0.5:
        suggestion['hidden_layer'] = 'tanh'
        suggestion['explanation'].append(
            "High feature correlation with negative values"
        )
    elif data_range > 10 and skewness > 1.0:
        suggestion['hidden_layer'] = 'relu'
        suggestion['explanation'].append(
            "Large data range with high skewness"
        )
    elif sparsity > 0.5:
        suggestion['hidden_layer'] = 'relu'
        suggestion['explanation'].append(
            "Sparse data benefits from ReLU"
        )
    else:
        suggestion['hidden_layer'] = 'relu'
        suggestion['explanation'].append(
            "Default choice for stable training"
        )

    return suggestion
```

Best Practices & Usage

1. Start Simple
    - Single hidden layer
    - Moderate number of nodes
    - Default learning rate
2. Monitor & Adjust
    - Watch training metrics
    - Check for overfitting
    - Adjust parameters as needed
3. Interpret Results
    - Use SHAP values

Advanced Implementation Details

Error Handling

```python
def train_model_with_validation(model, X, y, validation_split=0.2):
    try:
        # Train with validation
        X_train, X_val, y_train, y_val = train_test_split(
            X, y, test_size=validation_split,
            stratify=y, random_state=42
        )

        # Training with early stopping
        best_loss = float('inf')
        patience = 10
        counter = 0

        for epoch in range(model.max_iter):
            model.partial_fit(X_train, y_train,
                              classes=np.unique(y))
            val_loss = log_loss(y_val,
                                model.predict_proba(X_val))

            # Early stopping check
            if val_loss < best_loss:
                best_loss = val_loss
                counter = 0
            else:
                counter += 1
                if counter >= patience:
                    print(f"Early stopping at epoch {epoch}")
                    break
```

Real-time Visualization Updates

```
def update_visualization(model, X_sample):
    # Calculate layer activations
    activations = []
    current_activation = X_sample

    for i, (weights, bias) in enumerate(zip(
            model.coefs_, model.intercepts_)):
        current_activation = np.dot(current_activation, weights) + bias

        # Apply activation function
        if model.activation == 'relu':
            current_activation = np.maximum(0, current_activation)
        elif model.activation == 'tanh':
            current_activation = np.tanh(current_activation)
        elif model.activation == 'logistic':
            current_activation = 1/(1 + np.exp(-current_activation))

        activations.append(current_activation)

    # Update network visualization
    return visualize_network_with_activations(
        model.coefs_,
        activations,
        feature_names=X.columns
    )
```

Interactive Feature Analysis

```
def analyze_feature_importance(model, X, feature_names):
    # Get feature importance from weights
    importance = np.zeros(len(feature_names))

    # Calculate importance based on first layer weights
    first_layer_weights = np.abs(model.coefs_[0])
    importance = np.mean(first_layer_weights, axis=1)

    # Create interactive plot
    fig = go.Figure()
    fig.add_trace(go.Bar(
        x=feature_names,
        y=importance,
        marker_color='rgb(158,202,225)',
        text=np.round(importance, 4),
        textposition='auto',
    ))

    fig.update_layout(
        title='Feature Importance Analysis',
        xaxis_title='Features',
        yaxis_title='Importance Score',
        showlegend=False
    )

    # Add interactive tooltips
    feature_descriptions = get_feature_descriptions()
    fig.update_traces(
        hovertemplate="<br>".join([
            "Feature: %{x}",
            "Importance: %{y:.4f}",
            "Description: %{customdata}"
        ]),
        customdata=feature_descriptions
    )

    return fig
```

Model Diagnostics

```python
def diagnose_model_performance(model, X, y):
    # Get predictions and probabilities
    y_pred = model.predict(X)
    y_prob = model.predict_proba(X)

    # Calculate metrics
    metrics = {
        'Accuracy': accuracy_score(y, y_pred),
        'Precision': precision_score(y, y_pred, average='weighted'),
        'Recall': recall_score(y, y_pred, average='weighted'),
        'F1': f1_score(y, y_pred, average='weighted'),
        'Log Loss': log_loss(y, y_prob)
    }

    # Learning curve analysis
    train_sizes, train_scores, val_scores = learning_curve(
        model, X, y, cv=5, n_jobs=-1,
        train_sizes=np.linspace(0.1, 1.0, 10)
    )

    # Plot learning curves
    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, np.mean(train_scores, axis=1),
             label='Training score')
    plt.plot(train_sizes, np.mean(val_scores, axis=1),
             label='Cross-validation score')
    plt.xlabel('Training examples')
    plt.ylabel('Score')
    plt.title('Learning Curves')
    plt.legend(loc='best')

    return metrics, plt.gcf()
```

Thank You!

Key Takeaways:

- Interactive neural network visualization
- Real-time performance monitoring
- Explainable AI with SHAP
- Customizable architecture
- Production-ready code examples