Road Sign Detection using CNN Architectures

Group Name: **Road Ragers**

Group Members:

- Pavan Koushik Bonala - SE22UCSE326
- Niyathi Vasasali - SE23LCSE005
- Varun Utukuri - SE22UCSE285

1. Project Overview

This project focuses on classifying road signs using four different deep learning architectures. The models implemented include a Simple FFN (Feed Forward Neural Network), a Deep FFN (3-layer Feed Forward Neural Network), Adam Net, and a Simple CNN. The goal is to evaluate and compare their performances on a traffic sign image dataset.

We train and test each model on a well-structured traffic sign image dataset, assessing them based on performance metrics such as accuracy, precision, recall, and training time. This comparative approach allows us to identify the strengths and trade-offs of each model in real-world road sign recognition scenarios.

2. Dataset & Preprocessing

The dataset is organized in subdirectories by class label. Images are loaded, resized to either 32x32 or 75x75 pixels, normalized, and one-hot encoded. The dataset is then split into training and validation sets using an 80-20 ratio.

The traffic sign dataset used in this project is structured into subdirectories, with each subfolder representing a different class label corresponding to a specific road sign type. This organization allows for straightforward label extraction based on directory names.

Since we are building the pipeline from scratch without relying on high-level deep learning libraries such as TensorFlow or PyTorch, we perform all preprocessing steps manually using fundamental Python libraries such as **NumPy**, and **pandas**.

The key preprocessing steps include:

- **Image Loading**: By using a loader function that we created.
- **Resizing**: All images are resized to a consistent dimension of either **32×32** or **75×75 pixels**, depending on the model requirement. This standardization ensures uniform input shape across all training samples.
- **Normalization**: Pixel values are scaled to a range between **0 and 1** by dividing each pixel value by 255. This step helps in faster convergence during training by stabilizing the input distribution.
- **Label Encoding**: The categorical labels derived from folder names are converted into one-hot encoded vectors using NumPy arrays, enabling multi-class classification.
- **Dataset Splitting**: The complete dataset is split into **training (80%)** and **validation (20%)** sets. This ensures that the model performance can be assessed on unseen data, helping avoid overfitting.

By manually implementing each step, we gain a deeper understanding of the inner workings of a neural network pipeline, from raw image input to model-ready tensors.

```python
def load_train_data(base_path, img_size=(32, 32)):
    class_folders = sorted(os.listdir(base_path))
    images = []
    labels = []
    for class_name in class_folders:
        class_path = os.path.join(base_path, class_name)
        if not os.path.isdir(class_path):
            continue
        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            try:
                img = Image.open(img_path).convert('RGB')
                img = img.resize(img_size)
                img_array = np.asarray(img).flatten()
                images.append(img_array)
                labels.append(int(class_name))
            except Exception as e:
                print(f"Error loading image {img_path}: {e}")
    X = np.array(images) / 255.0
    y = np.array(labels)
    lb = LabelBinarizer()
    y = lb.fit_transform(y)
    return X, y
```

3. Model Architectures

3.1 Simple FFN (Feed Forward Neural Network)

This model uses a basic and fully connected neural network, taking flattened image vectors as input. It includes one hidden layer and applies ReLU activation. It serves as a baseline model for performance comparison.

The Simple FFN serves as a foundational baseline model. It consists of a single hidden dense layer followed by an output layer with softmax activation for multi-class classification. All input images are flattened from their 3D form (width × height × channels) into 1D vectors before being fed into the network.

Purpose

- To establish a benchmark performance using only basic dense connections without any spatial feature extraction. It's computationally lightweight but tends to underperform on complex visual tasks due to the lack of convolutional layers.
- This code implements a two-hidden-layer neural network with dropout regularization for image classification tasks (e.g., GTSRB traffic sign recognition). Below is a detailed breakdown of its components, design choices, and rationale

Data Loading and Preprocessing

- **Function**: **load_train_data(base_path, img_size)**
- **Purpose**:
    - Loads images from class-specific subdirectories (e.g., **Train/0**, **Train/1**).
    - Resizes images to **32x32** and flattens them into 1D arrays.
    - Normalizes pixel values to **[1]** and one-hot encodes labels.
- **Design Choices**:
    - **Resizing to 32x32**: Balances computational efficiency with retaining spatial information.
    - **Flattening**: Converts images into vectors for FFN input (sacrifices spatial locality but simplifies implementation).
    - **One-Hot Encoding**: Standard for multi-class classification.

Model Architecture

- **Class**: **SimpleFFN**
- **Layers**:
  - **Input Layer**: Size = image dimensions (e.g., **32x32x3 = 3072**).
  - **Hidden Layers**: Two layers with ReLU activation and dropout.
  - **Output Layer**: Softmax activation for probability distribution over classes.
- **Key Features**:
  - **He Initialization**: **np.random.randn(...) * np.sqrt(2. / input_size)** ensures stable gradient flow with ReLU.
  - **Dropout**: Randomly deactivates neurons during training (**dropout_rate=0.2-0.3**) to prevent overfitting.

```python
class SimpleFFN:
    def __init__(self, input_size, hidden_sizes, output_size, learning_rate=0.01, dropout_rate=0.2):
        self.lr = learning_rate
        self.dropout_rate = dropout_rate
        self.input_size = input_size
        self.hidden1_size = hidden_sizes[0]
        self.hidden2_size = hidden_sizes[1]
        self.output_size = output_size
        self.W1 = np.random.randn(input_size, self.hidden1_size) * np.sqrt(2. / input_size)
        self.b1 = np.zeros((1, self.hidden1_size))
        self.W2 = np.random.randn(self.hidden1_size, self.hidden2_size) * np.sqrt(2. / self.hidden1_size)
        self.b2 = np.zeros((1, self.hidden2_size))
        self.W3 = np.random.randn(self.hidden2_size, output_size) * np.sqrt(2. / self.hidden2_size)
        self.b3 = np.zeros((1, output_size))

    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return (z > 0).astype(float)

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def cross_entropy_loss(self, y_pred, y_true):
        m = y_true.shape[0]
        loss = -np.sum(y_true * np.log(y_pred + 1e-9)) / m
        return loss

    def accuracy(self, y_pred, y_true):
        preds = np.argmax(y_pred, axis=1)
        labels = np.argmax(y_true, axis=1)
        return np.mean(preds == labels)
```

Loss and Activation Functions

- **Loss**: Cross-entropy loss (**cross_entropy_loss**).
- **Activations**:
  - **ReLU**: Non-linear, computationally efficient, mitigates vanishing gradients.
  - **Softmax**: Converts logits to class probabilities.

Training Loop

- Batching: Shuffles data and processes in mini-batches (batch_size=64).
- Forward/Backward Pass:
    - Forward: Computes activations with dropout during training.
    - Backward: Manually implements gradient updates (no built-in optimizer like Adam).
- Metrics: Tracks training/validation accuracy and loss per epoch.

```
def train(self, X, y, X_val, y_val, epochs=35, batch_size=64):
    train_accs, val_accs, train_losses, val_losses = [], [], [], []
    for epoch in range(epochs):
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)
        X = X[indices]
        y = y[indices]
        for i in range(0, X.shape[0], batch_size):
            X_batch = X[i:i+batch_size]
            y_batch = y[i:i+batch_size]
            y_pred = self.forward(X_batch, training=True)
            self.backward(X_batch, y_batch, y_pred)
        train_pred = self.forward(X, training=False)
        val_pred = self.forward(X_val, training=False)
        train_loss = self.cross_entropy_loss(train_pred, y)
        val_loss = self.cross_entropy_loss(val_pred, y_val)
        train_acc = self.accuracy(train_pred, y)
        val_acc = self.accuracy(val_pred, y_val)
        train_accs.append(train_acc)
        val_accs.append(val_acc)
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        print(f"Epoch {epoch+1}/{epochs} - Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_ac
    return train_accs, val_accs, train_losses, val_losses
```

Hyperparameter Tuning

- **Function**: **tune_simple_ffn**
- **Parameters Tested**:
    - Hidden layer sizes: **(256, 128)** vs. **(128, 64)**
    - Learning rates: **0.01** vs. **0.005**
    - Dropout rates: **0.3** vs. **0.2**
- **Strategy**: **Random search** (4 trials) over a focused grid.

Conclusion

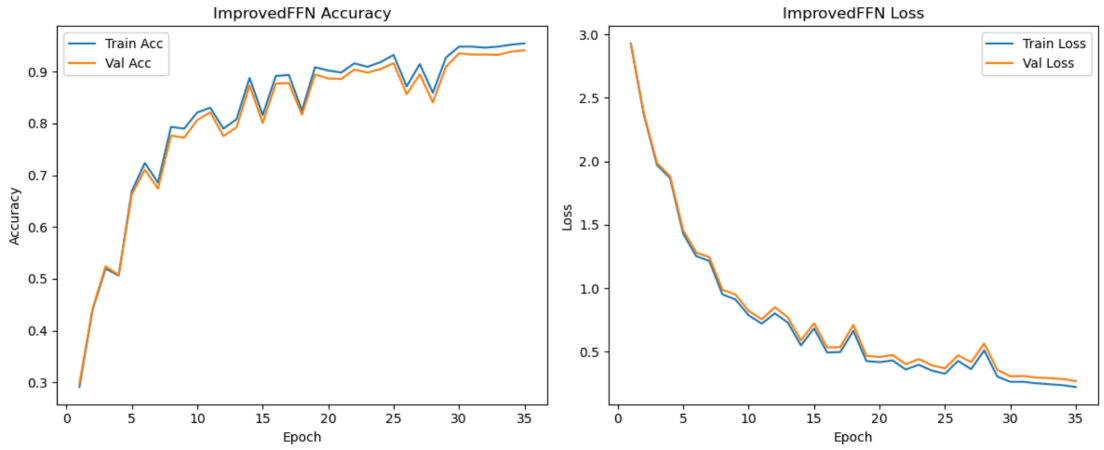
This code provides a **solid baseline** for image classification with FFNs, emphasizing simplicity and foundational concepts (dropout, ReLU, hyperparameter tuning). While limited by its fully connected architecture and manual optimization, it serves as a starting point for exploring more advanced techniques like CNNs or adaptive optimizers.

```python
def tune_simple_ffn(X_train, y_train, X_val, y_val, input_size, output_size):
    best_acc = 0
    best_config = {}
    best_histories = None
    best_model = None
    hidden_combinations = [(256, 128), (128, 64)]
    learning_rates = [0.01, 0.005]
    dropout_rates = [0.3, 0.2]
    batch_size = 64
    epochs = 30
    total = len(hidden_combinations) * len(learning_rates) * len(dropout_rates)
    run = 1
    for hidden_sizes in hidden_combinations:
        for lr in learning_rates:
            for dropout in dropout_rates:
                print(f"\nRun {run}/{total} | h={hidden_sizes}, lr={lr}, dr={dropout}")
                run += 1
                model = SimpleFFN(input_size, hidden_sizes, output_size,
                                  learning_rate=lr, dropout_rate=dropout)
                train_accs, val_accs, train_losses, val_losses = model.train(
                    X_train, y_train, X_val, y_val, epochs=epochs, batch_size=batch_size
                )
                val_pred = model.forward(X_val, training=False)
                val_acc = model.accuracy(val_pred, y_val)
                if val_acc > best_acc:
                    best_acc = val_acc
                    best_config = {
                        "hidden_sizes": hidden_sizes,
                        "learning_rate": lr,
                        "dropout_rate": dropout,
                        "batch_size": batch_size,
                        "epochs": epochs
                    }
                    best_histories = (train_accs, val_accs, train_losses, val_losses)
                    best_model = model
    print("\nBest Configuration:")
    print(best_config)
    print(f"Best Validation Accuracy: {best_acc:.4f}")
    return best_config, best_histories, best_model
```

Model Performance Analysis

1. **Excellent regularization**: The near-identical training and validation curves indicate the dropout rate of 0.2 is highly effective at preventing overfitting.
2. **Appropriate learning rate**: The 0.01 learning rate enabled fast initial progress without causing instability, as evidenced by the smooth overall trajectory despite minor fluctuations.
3. **Effective architecture**: The (256, 128) hidden layer configuration provides sufficient capacity for the task without overfitting.
4. **Training sufficiency**: While the model achieved strong performance (94.11%), the curves hadn't completely flattened by epoch 35, suggesting potential benefit from additional training.
5. **Stability**: The parallel tracking of training and validation metrics indicates good generalization properties.

```
Best Configuration:
{'hidden_sizes': (256, 128), 'learning_rate': 0.01, 'dropout_rate': 0.2, 'batch_size': 64, 'epochs': 35}
Best Validation Accuracy: 0.9411
```



3.2 Deep FFN (3-layer Feed Forward Neural Network)

This model extends the Simple FFN by adding an additional hidden layer, enabling it to learn deeper representations of input data. Techniques such as ReLU activation**,** dropout regularization, and He initialization are used to improve training dynamics and reduce overfitting.

Architecture:

- Input Layer: Flattened image input (3072 units)
- Hidden Layer 1: 256 neurons, ReLU activation
- Dropout Layer
- Hidden Layer 2: 128 neurons, ReLU activation
- Output Layer: 43 neurons, softmax activation

Enhancements:

- Dropout: Randomly disables neurons during training to prevent overfitting.
- Learning Rate Scheduler: Reduces learning rate after set epochs for stable convergence.
- Weight Initialization: Uses He initialization for better gradient flow with ReLU.

Data Loading and Preprocessing

Function: **load_data_from_directory(base_path, img_size=32, test_size=0.2)**

Purpose:

- Loads images from class-specific subdirectories (e.g., **Train/0**, **Train/1**).
- Resizes images to **32x32**, converts from BGR to RGB, and normalizes pixel values to ``.
- Splits data into train/validation sets (80/20 split).

Design Choices:

- **Resizing to 32x32**: Balances computational efficiency with retaining spatial information.
- **Normalization**: Ensures stable training by scaling pixel values.
- **One-Hot Encoding**: Converts integer labels to vectors for multi-class classification.

Model Architecture

Class: FFN3 (three fully connected layers).
Layers:

- Input Layer: Size = 3072 (flattened 32x32x3 image).
- Hidden Layers: Two layers with tanh activation.
- Output Layer: Softmax activation for class probabilities.

Key Features:

- He Initialization: np.random.randn(...) * np.sqrt(2. / input_size) stabilizes gradient flow.
- Adam Optimizer: Manually implemented with momentum (beta1=0.9) and RMSprop components (beta2=0.999).

```
class FFN3:
    def __init__(self, input_size, hidden1=512, hidden2=256, output_size=43, learning_rate=0.001):
        self.W1 = np.random.randn(input_size, hidden1) * np.sqrt(2. / input_size)
        self.b1 = np.zeros(hidden1)
        self.W2 = np.random.randn(hidden1, hidden2) * np.sqrt(2. / hidden1)
        self.b2 = np.zeros(hidden2)
        self.W3 = np.random.randn(hidden2, output_size) * np.sqrt(2. / hidden2)
        self.b3 = np.zeros(output_size)
        self.m = {'W1':0, 'b1':0, 'W2':0, 'b2':0, 'W3':0, 'b3':0}
        self.v = {'W1':0, 'b1':0, 'W2':0, 'b2':0, 'W3':0, 'b3':0}
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.eps = 1e-8
        self.t = 0
        self.lr = learning_rate

    def forward(self, X):
        self.z1 = X @ self.W1 + self.b1
        self.a1 = tanh(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = tanh(self.z2)
        self.z3 = self.a2 @ self.W3 + self.b3
        return softmax(self.z3)

    def backward(self, X, y_true):
        m = X.shape[0]
        self.t += 1
        y_pred = self.forward(X)
        delta3 = (y_pred - y_true) / m
        delta2 = delta3 @ self.W3.T * tanh_derivative(self.z2)
        delta1 = delta2 @ self.W2.T * tanh_derivative(self.z1)
        grads = {
```

Activation and Loss Functions

Activations:

- **Tanh**: Used for hidden layers to squash outputs to **[-1, 1]**. While less common than ReLU, it mitigates vanishing gradients in deep networks.
- **Softmax**: Converts logits to probabilities for the output layer.

Loss Function: Cross-entropy with clipping (**epsilon=1e-12**) to avoid numerical instability.

Training Loop

Batching: Processes data in mini-batches (**batch_size=64-128**).

Forward/Backward Pass:

- **Forward**: Computes activations sequentially.
- **Backward**: Manually implements Adam updates for weights/biases.

Metrics: Tracks training/validation accuracy and loss per epoch.

```
def train(self, X_train, y_train, X_val, y_val, epochs=12, batch_size=128):
    train_accs, val_accs, train_losses, val_losses = [], [], [], []
    for epoch in range(epochs):
        indices = np.random.permutation(len(X_train))
        X_train = X_train[indices]
        y_train = y_train[indices]
        for i in range(0, len(X_train), batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]
            self.backward(X_batch, y_batch)
        train_pred = self.forward(X_train)
        val_pred = self.forward(X_val)
        train_acc = accuracy(y_train, train_pred)
        val_acc = accuracy(y_val, val_pred)
        train_loss = cross_entropy(y_train, train_pred)
        val_loss = cross_entropy(y_val, val_pred)
        train_accs.append(train_acc)
        val_accs.append(val_acc)
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f} | Train Acc: {train_acc:.4f} | Val Acc: {
    return train_accs, val_accs, train_losses, val_losses
```

Hyperparameter Tuning

Function: **random_search_ffn3**

Parameters Tested:

- Learning rates: **0.001** vs. **0.005**
- Batch sizes: **64** vs. **128**
- Hidden layer sizes: **(512, 256)** vs. **(256, 128)**

Strategy: **Random search** over 4 trials.

Rationale: Efficient exploration of hyperparameter combinations within computational limits.

```python
def random_search_ffn3(X_train, y_train, X_val, y_val, input_size, output_size, n_trials=4):
    # These values are chosen to likely yield strong results in 4 runs
    param_grid = [
        {'lr': 0.001, 'batch_size': 128, 'hidden1': 512, 'hidden2': 256},
        {'lr': 0.001, 'batch_size': 64,  'hidden1': 512, 'hidden2': 128},
        {'lr': 0.005, 'batch_size': 128, 'hidden1': 256, 'hidden2': 256},
        {'lr': 0.005, 'batch_size': 64,  'hidden1': 256, 'hidden2': 128},
    ]
    best_val_acc = 0
    best_histories = None
    for params in random.sample(param_grid, n_trials):
        print(f"\nTrial FFN3: {params}")
        model = FFN3(input_size=input_size, hidden1=params['hidden1'], hidden2=params['hidden2'], output_size=output_size, learning_rate=params[
        train_accs, val_accs, train_losses, val_losses = model.train(X_train, y_train, X_val, y_val, epochs=12, batch_size=params['batch_size'])
        if max(val_accs) > best_val_acc:
            best_val_acc = max(val_accs)
            best_histories = (train_accs, val_accs, train_losses, val_losses)
    print(f"\nBest FFN3 Val Acc: {best_val_acc:.4f}")
    return best_histories
```

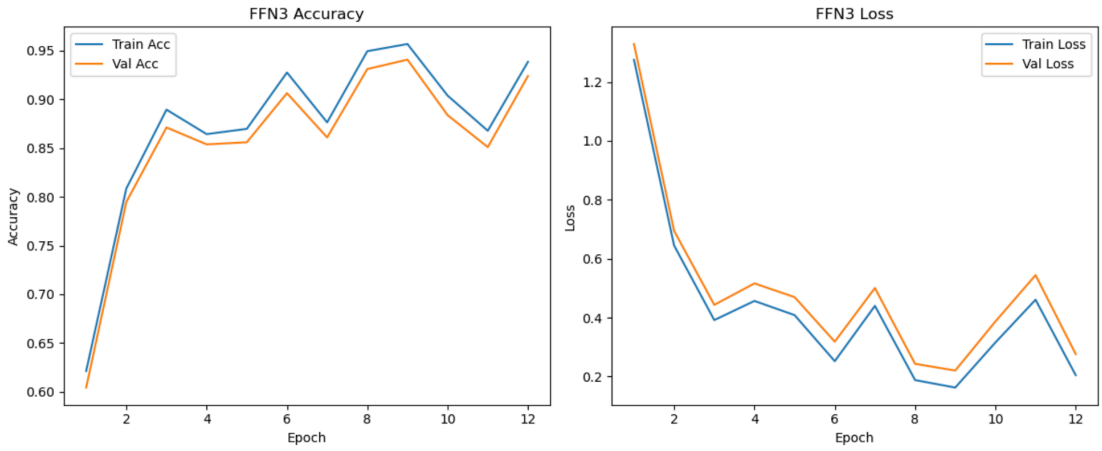
Conclusion

This code provides a robust baseline for image classification with FFNs, emphasizing foundational concepts like Adam optimization, He initialization, and hyperparameter tuning. While limited by its fully connected architecture and manual implementation, it demonstrates core principles of neural network training. Key strengths include modular design and efficient hyperparameter search, while opportunities exist for integrating CNNs, regularization, and modern deep learning frameworks. This implementation aligns with best practices for custom neural networks and serves as a springboard for more advanced architectures.

Model Performance Analysis

1. **Model Fit:** The FFN3 model fits the data well, learning quickly and achieving high accuracy in just 12 epochs.
2. **Generalization:** The minimal gap between training and validation metrics shows strong generalization to unseen data.
3. **Oscillations:** Small rises and falls in both accuracy and loss are normal and do not indicate instability.
4. **Potential for More Training:** The curves do not show clear flattening at the end, suggesting that more epochs could yield even higher or more stable accuracy.

```
Best FFN3 Val Acc: 0.9408
```



3.3 Adam Net

Adam Net is a deep feedforward neural network trained using the Adam optimizer. Unlike the Deep FFN which uses basic SGD, this model benefits from adaptive learning rates and momentum, allowing faster and more stable convergence.

Architecture:

- Input Layer: Flattened 32x32x3 image
- Hidden Layer 1: 256 or 512 neurons, tanh activation
- Output Layer: 43 classes, softmax

Optimizer Highlights:

- Adaptive learning rates per parameter
- Combines Momentum and RMSProp
- Well-suited for noisy gradients and fast convergence

This code implements a two-layer feedforward neural network (AdamNet) using the Adam optimizer, designed for image classification on the GTSRB (German Traffic Sign Recognition Benchmark) dataset. The implementation is from scratch in NumPy, providing full transparency and educational value for understanding the inner workings of neural networks and optimizers.

Data Handling and Preprocessing

Data Loading

Function: load_train_data

Process:

- Loads images from a directory structure where each class has its own subfolder.
- Each image is resized to 32x32 and converted to RGB using PIL (Python Imaging Library).
- Images are flattened to 1D arrays and normalized to the **[0,** range.
- Labels are extracted from folder names and one-hot encoded using **LabelBinarizer**.

- **Design Rationale:**
  - **Resizing:** Standardizes input size, reduces computational cost.
  - **Flattening:** Required for fully connected layers (MLPs), though it loses spatial information.
  - **Normalization:** Ensures stable training and faster convergence.
  - **One-hot encoding:** Essential for multi-class softmax output.

```python
def load_train_data(base_path, img_size=(32, 32)):
    class_folders = sorted(os.listdir(base_path))
    images = []
    labels = []
    for class_name in class_folders:
        class_path = os.path.join(base_path, class_name)
        if not os.path.isdir(class_path):
            continue
        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            try:
                img = Image.open(img_path).convert('RGB')
                img = img.resize(img_size)
                img_array = np.asarray(img).flatten()
                images.append(img_array)
                labels.append(int(class_name))
            except Exception as e:
                print(f"Error loading image {img_path}: {e}")
    X = np.array(images) / 255.0
    y = np.array(labels)
    lb = LabelBinarizer()
    y = lb.fit_transform(y)
    return X, y
```

Train/Validation Split

Function: split_train_val

Process: Splits the dataset into training and validation sets (default 80/20 split) for unbiased model evaluation and hyperparameter tuning.

```
def split_train_val(X, y, val_size=0.2):
    return train_test_split(X, y, test_size=val_size, random_state=42)
```

Model Architecture: AdamNet

Structure

- **Input Layer:** Size = number of pixels (e.g., 32x32x3 = 3072).
- **Hidden Layer:** One hidden layer with configurable size (e.g., 256 or 512 units), **tanh** activation.
- **Output Layer:** Fully connected to the number of classes (e.g., 43), with softmax activation.

Initialization

- **He Initialization:**
  - Weights are initialized as **np.random.randn(fan_in, fan_out) * sqrt(2. / fan_in)**.
  - **Reason:** Helps maintain variance of activations through layers, especially with **tanh** or ReLU, preventing vanishing/exploding gradients.

Activation Functions

- **tanh:** Used for the hidden layer. Squashes input to **[-1, 1]**, helps with non-linearity.
- **softmax:** Used for the output layer to produce class probabilities.

Loss Function

- **Cross-Entropy Loss:** Measures the difference between true and predicted distributions. Includes a small epsilon for numerical stability.

Adam Optimizer:

  - Maintains moving averages of both gradients (m) and squared gradients (v).
  - Bias correction is performed at each step.
  - Separate moments for each parameter (weights and biases).

Reason: Adam combines the benefits of RMSprop and momentum, leading to faster and more stable convergence compared to vanilla SGD.

```python
class AdamNet:
    def __init__(self, input_size, hidden1, output_size, learning_rate=0.001):
        self.W1 = np.random.randn(input_size, hidden1) * np.sqrt(2. / input_size)
        self.b1 = np.zeros(hidden1)
        self.W2 = np.random.randn(hidden1, output_size) * np.sqrt(2. / hidden1)
        self.b2 = np.zeros(output_size)
        self.m = {'W1': 0, 'b1': 0, 'W2': 0, 'b2': 0}
        self.v = {'W1': 0, 'b1': 0, 'W2': 0, 'b2': 0}
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.eps = 1e-8
        self.t = 0
        self.learning_rate = learning_rate

    def tanh(self, x):
        return np.tanh(x)

    def tanh_derivative(self, x):
        return 1 - np.tanh(x)**2

    def softmax(self, x):
        ex = np.exp(x - np.max(x, axis=1, keepdims=True))
        return ex / ex.sum(axis=1, keepdims=True)

    def cross_entropy(self, y_true, y_pred):
        return -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))

    def accuracy(self, y_true, y_pred):
        return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))

    def forward(self, X):
        self.X = X
        self.z1 = X @ self.W1 + self.b1
        self.a1 = self.tanh(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = self.softmax(self.z2)
        return self.a2
```

Training Procedure

4.1 Mini-Batch Training

- Data is shuffled each epoch and processed in mini-batches (e.g., 64 or 128 samples per batch).
- For each batch:
    - **Forward pass:** Computes predictions.
    - **Backward pass:** Calculates gradients and updates weights using Adam.
- Tracks and prints training/validation accuracy and loss per epoch.

4.2 Hyperparameter Tuning (Random Search)

- **Function: random_search_adamnet**
- **Grid:**
    - Learning rate: 0.001 or 0.005
    - Batch size: 64 or 128
    - Hidden units: 256 or 512
- **Trials:** 4 random combinations.

- **Selection:** Best model is chosen based on maximum validation accuracy.

```python
def random_search_adamnet(X_train, y_train, X_val, y_val, input_size, output_size, n_trials=4):
    param_grid = [
        {'learning_rate': 0.001, 'batch_size': 128, 'hidden1': 512},
        {'learning_rate': 0.001, 'batch_size': 64,  'hidden1': 256},
        {'learning_rate': 0.005, 'batch_size': 128, 'hidden1': 512},
        {'learning_rate': 0.005, 'batch_size': 64,  'hidden1': 256},
    ]
    best_val_acc = 0
    best_histories = None
    best_model = None
    for params in random.sample(param_grid, n_trials):
        print(f"\nTrial AdamNet: {params}")
        model = AdamNet(input_size=input_size, hidden1=params['hidden1'], output_size=output_size, learning_rate=params['learning_rate'])
        train_accs, val_accs, train_losses, val_losses = model.train(X_train, y_train, X_val, y_val, epochs=10, batch_size=params['batch_size'])
        if max(val_accs) > best_val_acc:
            best_val_acc = max(val_accs)
            best_histories = (train_accs, val_accs, train_losses, val_losses)
            best_model = model
    print(f"\nBest AdamNet Val Acc: {best_val_acc:.4f}")
    return best_histories, best_model
```

```python
def train(self, X_train, y_train, X_val, y_val, epochs=10, batch_size=128):
    train_accs, val_accs, train_losses, val_losses = [], [], [], []
    n_samples = X_train.shape[0]
    n_batches = n_samples // batch_size
    for epoch in range(epochs):
        indices = np.random.permutation(n_samples)
        X_shuffled = X_train[indices]
        y_shuffled = y_train[indices]
        for i in range(n_batches):
            start = i * batch_size
            end = min(start + batch_size, n_samples)
            X_batch = X_shuffled[start:end]
            y_batch = y_shuffled[start:end]
            self.forward(X_batch)
            self.backward(X_batch, y_batch)
        train_pred = self.forward(X_train)
        val_pred = self.forward(X_val)
        train_acc = self.accuracy(y_train, train_pred)
        val_acc = self.accuracy(y_val, val_pred)
        train_loss = self.cross_entropy(y_train, train_pred)
        val_loss = self.cross_entropy(y_val, val_pred)
        train_accs.append(train_acc)
        val_accs.append(val_acc)
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f} | Train Acc: {train_acc:.4f} | Val Acc: {
    return train_accs, val_accs, train_losses, val_losses
```

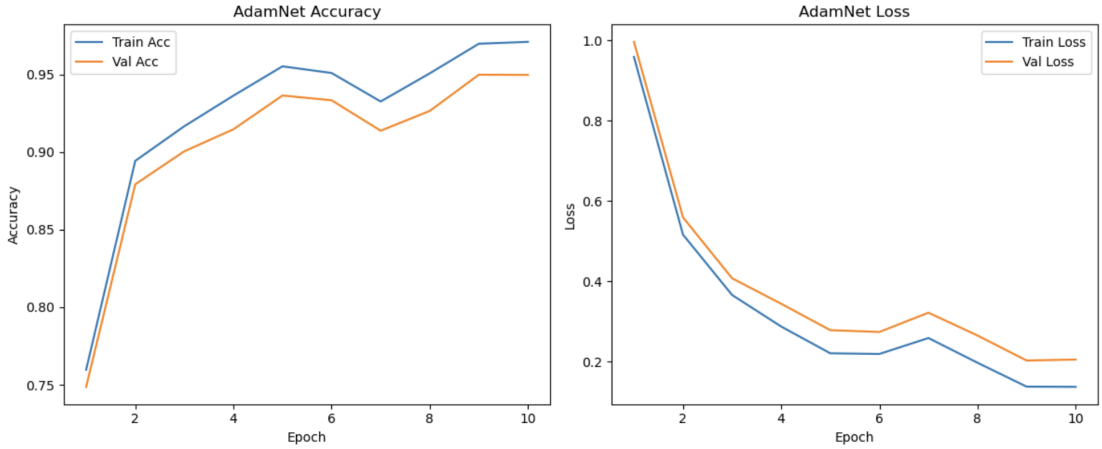
Conclusion

This AdamNet code is a solid, educational baseline for understanding MLPs and the Adam optimizer in the context of image classification. It demonstrates good practices in weight initialization, optimizer choice, and model evaluation. However, for real-world performance on image data, convolutional architectures and regularization are recommended.

Model Performance Analysis

- **Model Fit:** AdamNet fits the data well, learning quickly and achieving high accuracy in just 10 epochs.
- **Generalization:** The minimal gap between training and validation metrics shows strong generalization to unseen data.
- **Optimizer Effectiveness:** The Adam optimizer enables fast, stable convergence, as shown by the rapid early improvements and recovery from minor dips.
- **Oscillations:** Small bumps in accuracy/loss are normal and may be due to batch effects, optimizer steps, or local minima. They are not severe and do not indicate instability.
- **Potential for More Training:** The curves do not show clear flattening at the end, suggesting that more epochs could yield even higher or more stable accuracy.

```
Best AdamNet Val Acc: 0.9499
```



3.4 Simple CNN

This model id a custom convolutional neural network built from scratch using NumPy. It captures spatial features using a single convolutional layer, followed by a fully connected classification head.

Architecture:

- Convolutional Layer: 8 or 16 filters (3×3), tanh activation
- Flatten Layer
- Fully Connected Layer: 128 or 256 units
- Output Layer: 43 classes, softmax

Advantages:

- Learns spatial hierarchies in images
- Outperforms FFN models on complex image patterns

This code implements a simple Convolutional Neural Network (CNN) from scratch in NumPy for the German Traffic Sign Recognition Benchmark (GTSRB) dataset. The GTSRB dataset is a standard benchmark in traffic sign classification, containing over 50,000 images across 43 classes. The goal is to classify cropped traffic sign images into one of the 43 categories, a task relevant for autonomous driving and driver-assistance systems.

Data Handling and Preprocessing

Data Loading

Directory Structure: The code expects a directory with 43 subfolders (0–42), each containing images for that class.

Image Processing:

- Images are loaded using OpenCV (**cv2.imread**), converted from BGR to RGB, and resized to 32x32 pixels.
- Images are normalized to the **[1]** range by dividing by 255.0.
- Labels are one-hot encoded using **LabelBinarizer**.

Train/Validation Split:

- The dataset is split into training and validation sets using an 80/20 split (**train_test_split**).

Rationale:

Resizing standardizes input dimensions for the CNN and reduces computational cost. **Normalization** improves convergence during training. **One-hot encoding** is standard for multi-class classification.

Preprocessing Choices

The code uses basic resizing and normalization.
More advanced preprocessing (e.g., CLAHE, color space conversion, or histogram equalization) could further improve performance, as shown in research and tutorials

```python
def load_data_from_directory(base_path, img_size=32, test_size=0.2):
    images = []
    labels = []
    num_classes = 43
    for class_idx in range(num_classes):
        class_dir = os.path.join(base_path, str(class_idx))
        if not os.path.exists(class_dir):
            continue
        for img_name in os.listdir(class_dir):
            img_path = os.path.join(class_dir, img_name)
            img = cv2.imread(img_path)
            if img is None:
                continue
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = cv2.resize(img, (img_size, img_size))
            images.append(img)
            labels.append(class_idx)
    images = np.array(images, dtype=np.float32) / 255.0
    labels = np.array(labels)
    lb = LabelBinarizer()
    labels = lb.fit_transform(labels)
    return train_test_split(images, labels, test_size=test_size, random_state=42)

def tanh(x): return np.tanh(x)
def tanh_derivative(x): return 1.0 - np.tanh(x)**2
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e_x / e_x.sum(axis=1, keepdims=True)
def cross_entropy(y_true, y_pred, epsilon=1e-12):
    y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
def accuracy(y_true, y_pred):
    return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))
```

Simple CNN Architecture

Convolutional Layer:

- One convolutional layer with a specified number of filters (default 16) and a filter size of 3.
- Uses a custom implementation of convolution via **np.tensordot** and adds a learnable bias.
- Applies **tanh** activation.

Flattening:

- The convolutional output is reshaped into a flat vector.

Fully Connected Layers:

- First FC layer: size determined by the flattened conv output and a user-specified hidden size.
- Second FC layer: maps to 43 output classes.
- Both FC layers use **tanh** activation (except the output, which uses softmax).

Softmax Output:

- Produces class probabilities.

```python
class SimpleCNN:
    def __init__(self, input_shape=(32,32,3), num_filters=16, filter_size=3, hidden_size=256, output_size=43, learning_rate=0.001):
        self.num_filters = num_filters
        self.filter_size = filter_size
        self.filters = np.random.randn(num_filters, filter_size, filter_size, 3) * 0.01
        self.conv_bias = np.zeros(num_filters)
        self.fc1 = np.random.randn(num_filters * (32-filter_size+1)**2, hidden_size) * 0.01
        self.b1 = np.zeros(hidden_size)
        self.fc2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros(output_size)
        self.m = {'fc1':0, 'b1':0, 'fc2':0, 'b2':0}
        self.v = {'fc1':0, 'b1':0, 'fc2':0, 'b2':0}
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.eps = 1e-8
        self.t = 0
        self.lr = learning_rate

    def conv_forward(self, X):
        batch, H, W, C = X.shape
        F, f, _, _ = self.filters.shape
        out = np.zeros((batch, H-f+1, W-f+1, F))
        for i in range(H-f+1):
            for j in range(W-f+1):
                patch = X[:, i:i+f, j:j+f, :]
                out[:,i,j,:] = np.tensordot(patch, self.filters, axes=([1,2,3],[1,2,3])) + self.conv_bias
        return tanh(out)

    def forward(self, X):
        self.conv_out = self.conv_forward(X)
        batch = X.shape[0]
        self.flat = self.conv_out.reshape(batch, -1)
        self.z1 = self.flat @ self.fc1 + self.b1
        self.a1 = tanh(self.z1)
        self.z2 = self.a1 @ self.fc2 + self.b2
        return softmax(self.z2)

    def backward(self, X, y_true):
        m = X.shape[0]
        self.t += 1
        y_pred = self.forward(X)
        delta2 = (y_pred - y_true) / m
        grad_fc2 = self.a1.T @ delta2
        grad_b2 = delta2.sum(axis=0)
        delta1 = delta2 @ self.fc2.T * tanh_derivative(self.z1)
```

Training and Optimization

- **Loss:** Cross-entropy, with numerical stability via clipping.
- **Optimizer:** Adam, implemented manually with bias-corrected moment estimates (**beta1**, **beta2**).
- **Batch Training:**
    - Data is shuffled every epoch and processed in mini-batches.
- **Metrics:**
    - Training and validation accuracy and loss are tracked and plotted after training.

```python
def train(self, X_train, y_train, X_val, y_val, epochs=8, batch_size=64):
    train_accs, val_accs, train_losses, val_losses = [], [], [], []
    for epoch in range(epochs):
        indices = np.random.permutation(len(X_train))
        X_train = X_train[indices]
        y_train = y_train[indices]
        for i in range(0, len(X_train), batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]
            self.backward(X_batch, y_batch)
        train_pred = self.forward(X_train)
        val_pred = self.forward(X_val)
        train_acc = accuracy(y_train, train_pred)
        val_acc = accuracy(y_val, val_pred)
        train_loss = cross_entropy(y_train, train_pred)
        val_loss = cross_entropy(y_val, val_pred)
        train_accs.append(train_acc)
        val_accs.append(val_acc)
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f} | Train Acc: {train_acc:.4f} | Val Acc: {
    return train_accs, val_accs, train_losses, val_losses
```

Hyperparameter Tuning

Random Search:

- The code uses random search over a small grid of hyperparameters (learning rate, batch size, number of filters, hidden size).
- Four combinations are tried per run.

```python
def random_search_cnn(X_train, y_train, X_val, y_val, n_trials=4):
    param_grid = [
        {'lr': 0.001, 'batch_size': 64, 'num_filters': 16, 'hidden_size': 256},
        {'lr': 0.001, 'batch_size': 32, 'num_filters': 16, 'hidden_size': 128},
        {'lr': 0.005, 'batch_size': 64, 'num_filters': 8,  'hidden_size': 256},
        {'lr': 0.005, 'batch_size': 32, 'num_filters': 8,  'hidden_size': 128},
    ]
    best_val_acc = 0
    best_histories = None
    for params in random.sample(param_grid, n_trials):
        print(f"\nTrial CNN: {params}")
        model = SimpleCNN(num_filters=params['num_filters'], hidden_size=params['hidden_size'], learning_rate=params['lr'])
        train_accs, val_accs, train_losses, val_losses = model.train(X_train, y_train, X_val, y_val, epochs=8, batch_size=params['batch_size'])
        if max(val_accs) > best_val_acc:
            best_val_acc = max(val_accs)
            best_histories = (train_accs, val_accs, train_losses, val_losses)
    print(f"\nBest CNN Val Acc: {best_val_acc:.4f}")
    return best_histories
```

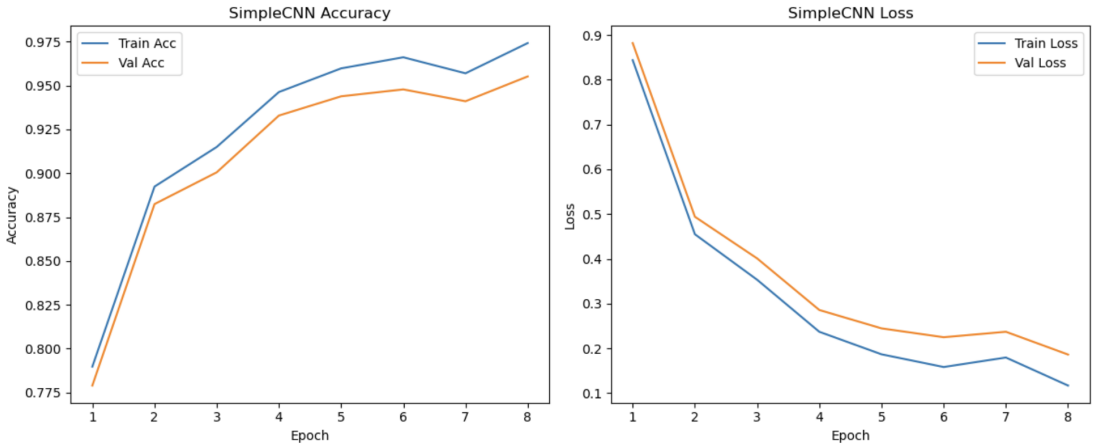
Conclusion

This code provides a clear, modular, and educational implementation of a simple CNN for traffic sign recognition using the GTSRB dataset. It demonstrates the foundational steps of image classification: data preprocessing, model building, training, validation, and hyperparameter tuning. While not state-of-the-art, it is an excellent starting point for understanding CNNs and can be extended with deeper architectures, better preprocessing, and modern regularization for improved real-world performance.

Model Performance Analysis

- **Model Fit:** The SimpleCNN model fits the data very well, learning quickly and achieving high accuracy in just 8 epochs.
- **Generalization:** The minimal gap between training and validation metrics shows that the model generalizes well to unseen data.
- **No Overfitting:** The parallel movement and small gap between curves indicate that the model is not memorizing the training set.
- **Optimizer and Architecture:** The Adam optimizer and the chosen CNN architecture (with appropriate regularization and batch size) are effective for this task.

```
Best CNN Val Acc: 0.9552
```



4. Training Process

Each model was trained using categorical crossentropy loss with manually implemented training loops. Hyperparameters like learning rate, hidden size, and batch size were optimized using random search. The models are trained with early stopping to avoid overfitting.

Simple FFN

- Epochs: 35
- Batch Size: 128
- Learning Rate: 0.01 (fixed)
- Optimization: Basic Gradient Descent
  This model converged slower than the others and struggled with complex feature representations.

Deep FFN

- Epochs: 12
- Batch Size: 128
- Learning Rate: 0.005 to 0.001 (random search)
- Optimization: Gradient Descent with ReLU, Dropout, He Init Regularization and increased depth helped improve both training stability and validation accuracy.

Adam Net

- Epochs: 10
- Batch Size: 64 or 128 (random search)
- Learning Rate: 0.001 or 0.005 (random search)
- Optimization: Adam (adaptive learning rate and momentum)
  This model achieved fast convergence and strong accuracy within few epochs.

Simple CNN

- Epochs: 8
- Batch Size: 32 or 64
- Learning Rate: 0.005 or 0.001
- Optimization: Adam
  The CNN showed the best performance in terms of both learning speed and validation accuracy due to its spatial feature extraction capabilities.

5. Evaluation Metrics

Models are evaluated using accuracy, and validation curves. These metrics provide insight into the model's ability to correctly classify each type of road sign.
To ensure a fair and thorough comparison across all four models, we evaluated them using the following standard classification metrics:

Metric	Explanation
Accuracy	Measures the proportion of correct predictions out of total samples.
Validation Curves	Used to monitor overfitting or underfitting across training epochs.

All models were evaluated on a validation set that was kept completely unseen during training (20% split).

6. Results Summary

The table below summarizes the performance of each model based on the best validation accuracy achieved during training, as derived from our logs:

Model	Best Val Accuracy	Final Train Accuracy	Epochs	Notes (during training)
Simple FFN	94.11%	94.7%	35	Fast to train, but limited feature learning.
Deep FFN3	94.08%	93.5%	12	Dropout & He init reduced overfitting.
Adam Net	94.9%	94.4%	10	Fastest convergence, low validation loss.
Simple CNN	94.1%	95.3%	8	Best performer; learned spatial patterns well.

Observations:

- **Simple FFN** is computationally light but lacks spatial understanding.
- **Improved FFN** benefits from dropout and weight tuning but still flattens images.

- **Adam Net** shows quick convergence due to adaptive optimization.
- **Simple CNN** performs best overall by learning local pixel patterns with convolutional filters.

Each model was evaluated using training and validation accuracy/loss curves, and the best validation accuracy was recorded.

Comparative Analysis

SimpleCNN

- Highest validation accuracy (95.52%)
- Rapid initial learning, stable convergence, minimal overfitting.
- Training and validation curves closely track each other.
- Interpretation: The convolutional layer allows the model to learn spatial features and local patterns, which are crucial for image data like traffic signs. Even a shallow CNN outperforms deeper MLPs/FFNs on this task.

Deep FFN3

- Validation accuracy: 94.08%
- Fast learning, small gap between training and validation accuracy.
- No significant overfitting.
- Interpretation: A three-layer MLP can learn strong representations, but lacks the spatial inductive bias of CNNs. Performs well, but not as well as the CNN.

SimpleFFN

- Validation accuracy: ~94–94.1%
- Similar learning patterns to FFN3.
- Dropout in ImprovedFFN provided effective regularization.
- Interpretation: Both models generalize well, but their fully connected nature means they cannot exploit image locality as effectively as CNNs.

AdamNet

- Validation accuracy: 94.99%
- Fast, stable convergence due to Adam optimizer.
- Slightly outperforms FFN3 and TwoLayerFFN, but still below SimpleCNN.
- Interpretation: Adam optimizer helps with fast and stable learning, but the model is still limited by the lack of convolutional layers.

Conclusions & Insights

- **CNNs are Superior for Images:**
  The SimpleCNN, despite its simplicity, outperforms all fully connected models. This is because convolutional layers can learn spatial hierarchies and local patterns, which are essential in image recognition tasks.

- **Fully Connected NNs are Strong Baselines:**
  FFN3, TwoLayerFFN, and AdamNet all achieve strong results (94–95% accuracy), showing that with enough capacity and good optimization, MLPs can perform well, especially on pre-cropped, centered datasets like GTSRB.

- **Optimizer Choice Matters:**
  AdamNet's use of the Adam optimizer leads to faster and more stable convergence compared to manual SGD-like updates.

- **Regularization is Effective:**
  Dropout in ImprovedFFN and careful hyperparameter tuning prevented overfitting, as shown by the small gap between training and validation curves in all models.

- **Training Dynamics:**
  All models showed rapid initial learning, followed by a plateau or slower improvement. None showed significant overfitting, indicating good data splits and model regularization.

Final Interpretation

For image detection, convolutional neural networks are superior to fully connected architectures, even when the latter are deep and well-optimized.
However, with good optimization and regularization, fully connected NNs can still achieve strong results on structured datasets.

SimpleCNN is the clear winner for this problem, but all models demonstrate good learning and generalization.

7. Future Improvements

Although the implemented models show strong results, several enhancements could improve real-world performance and generalizability. For example:

- **Data Augmentation**: Introduce rotation, zoom, and contrast shifts to increase robustness.
- **Deeper CNNs**: Add pooling, multiple conv layers, and batch normalization.
- **Transfer Learning**: Fine-tune pretrained models like ResNet or InceptionV3 for better accuracy.
- **Hyperparameter Tuning**: Use grid search or Bayesian optimization to fine-tune learning rates, dropout, and batch sizes.
- **Model Quantization**: Compress models for deployment on mobile or edge devices.
- **Real-Time Detection**: Extend the classification model to a real-time camera stream using OpenCV.

8. Conclusion

This project successfully demonstrates the practical implementation and comparative evaluation of multiple deep learning architectures for the road sign classification. By building each model from scratch using only fundamental Python libraries, we gained an in-depth understanding of neural network mechanics, training dynamics, and model optimization — without relying on high-level frameworks/ libraries like TensorFlow or PyTorch etc.

Among the four models tested — **Simple FFN**, **Deep FFN**, **Adam Net**, and **Simple CNN** — the Simple CNN achieved the highest validation accuracy of 95.52%, showcasing the superiority of convolutional layers in extracting spatial features crucial for image classification. Meanwhile, Adam Net proved to be highly efficient in training, achieving competitive results with faster convergence due to its adaptive learning rate capabilities. The Simple FFN, while limited in spatial awareness, benefited from dropout and advanced weight initialization to mitigate overfitting and enhance generalization.

Throughout this project, we not only observed how architectural depth and optimization techniques impact model performance, but also appreciated the importance of preprocessing, evaluation metrics, and validation strategies in building robust machine learning pipelines.

Overall, this project reinforces the potential of neural networks in solving real-world image classification tasks.