Multi-Class Image Classification

Machine Learning & Deep Learning

CNN

Index:

1) Data Collection
2) Data Cleaning
3) Prepare for Model training
4) Scaling
5) Machine Learning

   i) Train Test Split

   ii) Model Training

   iii) Evaluation the Model

6) Hyperparameter Tuning

7) Deep Learning - CNN (Convolution Neural Network)

i) CNN with PyTorch

ii) Import Libraries

iii) Prepare Data For PyTorch - i] Reshape - RGB 3D to GrayScale 1D

ii] Train Test Split

iii] Convert to PyTorch Tensors

iv] Create Data Loader

iv) Define the CNN Model

v) Instantiate Model, Loss Function, Optimizer

vi) Train the CNN with Early Stop (Epoch)

vi) Evaluate the CNN

8) Data Augmentation

9) Deeper CNN - Complex CNN

10) Classification Report (Confusion Matrix)

11) AUC - ROC Curve

12) Save Model

    i) Full CNN PyTorch Model Save (architect + Weight)

    ii) Save Only CNN PyTorch Model Parameters

13) Prediction Done Both - Full Model + Model Parameters

14) Streamlit Using Model Parameters Based

Data Collection

```python
import glob
from PIL import Image
```

[87] Python

Data Collection

```python
airplane = glob.glob("airplane/*png")
automobile = glob.glob("automobile/*png")
bird = glob.glob("bird/*png")
cat = glob.glob("cat/*png")
deer = glob.glob("deer/*png")
dog = glob.glob("dog/*png")
frog = glob.glob("frog/*png")
horse = glob.glob("horse/*png")
ship = glob.glob("ship/*png")
truck = glob.glob("truck/*png")
```

[88] Python

```python
print(len(airplane))
print(len(automobile))
print(len(bird))
print(len(cat))
print(len(deer))
print(len(dog))
print(len(frog))
print(len(horse))
print(len(ship))
print(len(truck))
```

9]

Python

```
5000
5000
5000
5000
5000
5000
5000
5000
5000
5000
```

Data Cleaning

```python
img = Image.open(airplane[2])
```

[90] Python

```python
img
```

[91] Python

...

Converted to Grey Scale RGB has 3 filters but Grey Scale has 1 filter

```python
img = img.convert('L')
```

[92] Python

```python
img
```

[93] Python

...

convert greay scale to array

```python
import numpy as np
```

Python

```python
pixels = np.array(img)
```

Python

```python
type(pixels)
```

Python

```
numpy.ndarray
```

```python
pixels.shape
```

Python

```
(32, 32)
```

[98]

```python
pixels
```

Python

```
array([[ 49,  50,  51, ...,  62,  61,  61],
       [ 49,  50,  52, ...,  64,  63,  62],
       [ 52,  53,  54, ...,  67,  66,  65],
       ...,
       [179, 180, 176, ..., 198, 186, 181],
       [176, 184, 173, ..., 170, 165, 158],
       [164, 168, 168, ..., 154, 152, 151]], dtype=uint8)
```

```python
data = []
labels = []
```

Python

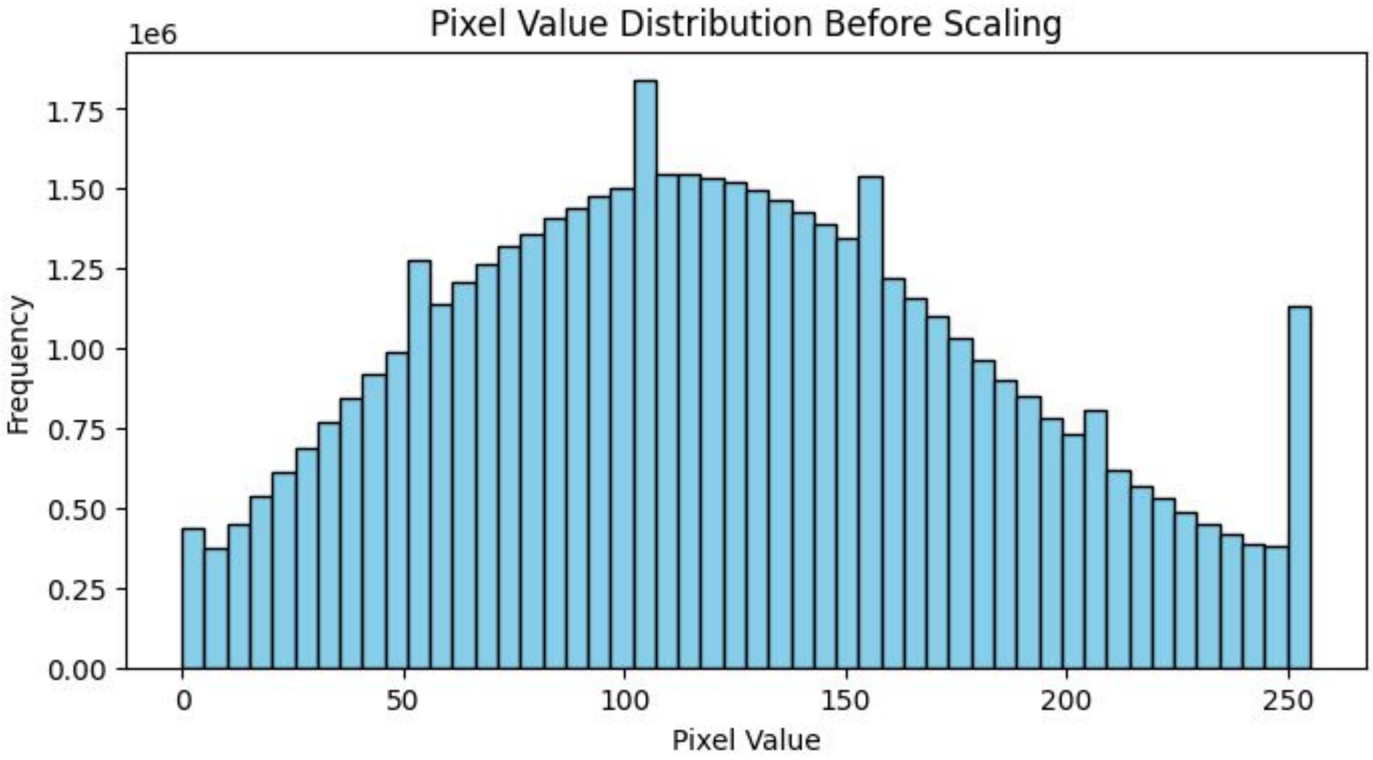
Need to flatten to array for 3d to 1d convert for model training

```python
# airplane images
for img_path in airplane:
    img = Image.open(img_path).convert('L')
    img = img.resize((32, 32))  # Resize for consistency, if needed
    pixels = np.array(img).flatten()
    data.append(pixels)
    labels.append(0)  # Label for airplane class
```

Python

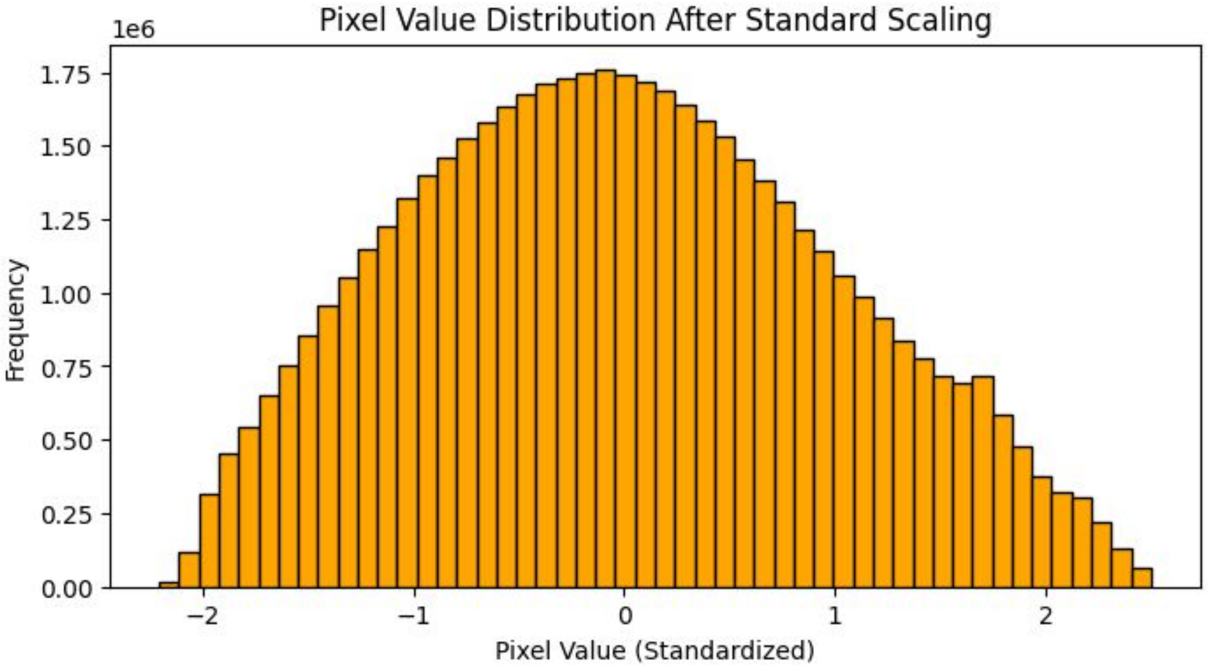
```python
#automobile images
for img_path in automobile:
    img = Image.open(img_path).convert('L')
    img = img.resize((32, 32))  # Resize for consistency, if needed
    pixels = np.array(img).flatten()
    data.append(pixels)
    labels.append(1)  # Label for automobile class
```

Python


Pixel Value Distribution Before Scaling
1e6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Frequency
0
50
100
150
200
250
Pixel Value

What Happens After Using StandardScaler?

StandardScaler standardizes features by removing the mean and scaling to unit variance. After scaling, the pixel values will have a mean close to 0 and a standard deviation close to 1. The distribution will be centered around 0, but the shape will remain similar to the original.



Save scaling model

```python
import joblib

# Save the fitted scaler to a file
joblib.dump(scaler_std, 'standard_scaler_model.pkl')
```

Python

```
['standard_scaler_model.pkl']
```

i) Train Test Split

```python
from sklearn.model_selection import train_test_split

# Use standardized data for ML
X_train, X_test, y_train, y_test = train_test_split(x_std_scaled, y, test_size=0.2, random_state=42)

print('Train shape:', X_train.shape, y_train.shape)
print('Test shape:', X_test.shape, y_test.shape)
```

58] Python

```
Train shape: (40000, 1024) (40000,)
Test shape: (10000, 1024) (10000,)
```

1024 mean by 32 x 32 pixel

This is multi class classification model am selecting Random Forest Ensamble method is type of multiple decision tress, Its Manage Noise Data

```python
# Train a Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Initialize and train the model
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rf_clf.fit(X_train, y_train)

# Predict on the test set
y_pred = rf_clf.predict(X_test)
```

9]

Python

Evaluation the Model

```python
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Random Forest Test Accuracy: {accuracy:.4f}')
print('Classification Report:')
print(classification_report(y_test, y_pred))
```

[70] Python

```
... Random Forest Test Accuracy: 0.3977
Classification Report:
              precision    recall  f1-score   support

           1       0.43      0.39      0.41      1009
           2       0.44      0.46      0.45      1034
           3       0.35      0.34      0.34       985
           4       0.31      0.24      0.27      1008
           5       0.31      0.35      0.33       986
           6       0.39      0.32      0.35      1030
           7       0.40      0.42      0.41      1021
           8       0.43      0.41      0.42       977
           9       0.46      0.54      0.50       955
          10       0.43      0.51      0.47       995

    accuracy                           0.40     10000
   macro avg       0.39      0.40      0.39     10000
weighted avg       0.39      0.40      0.39     10000
```

Almost Class is Balanced

##

Try Hyperparameter tuning

Using Grid Search CV For Random Forest and Cross Validation For 3 folds

Try Hyperparameter tuning

Using Grid Search CV For Random Forest and Cross Validation For 3 folds

Hyperparameter Tuning with GridSearchCV

Let's use GridSearchCV to find the best hyperparameters for the Random Forest model.

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
Best Parameters: {'max_depth': 30, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 200}
Best Cross-Validation Accuracy: 0.41192486989315874
Test Accuracy (GridSearchCV): 0.4025
Classification Report (GridSearchCV):
              precision    recall  f1-score   support

           1       0.45      0.38      0.41      1009
           2       0.45      0.45      0.45      1034
           3       0.36      0.33      0.34       985
           4       0.33      0.23      0.27      1008
           5       0.32      0.35      0.33       986
           6       0.38      0.32      0.35      1030
           7       0.40      0.45      0.42      1021
           8       0.44      0.44      0.44       977
           9       0.44      0.55      0.49       955
          10       0.43      0.54      0.48       995

    accuracy                           0.40     10000
   macro avg       0.40      0.40      0.40     10000
weighted avg       0.40      0.40      0.40     10000
```

No Major Changes happens after Using Hyperparameter Tuning

XGBoost Classifier

Let's try another ensemble method: XGBoost, which is often effective for structured/tabular data.

But No Changes

Now Try Deep Learning Technique

Multiconnect Neural Network

CNN - Convolution Neural Network

Working of Convolution Layer + Relu , Pooling Layer , Fully Flatten Layer (Convert Multi Dimension to 1 Dimension)

Convolution Layers take

1. Loopy Pattern
2. Vertical Line
3. Diagonal Line

More Hidden Layers Can Analyse Image Deeply

Step-by-Step CNN with PyTorch

This section demonstrates how to build, train, and evaluate a simple Convolutional Neural Network (CNN) for image classification using PyTorch.

X = np.array(data).reshape(-1, 1, 32, 32).astype(np.float32)

Converts your list of image data (data) into a NumPy array.

.reshape(-1, 1, 32, 32) changes the shape to (number of samples, 1, 32, 32):

-1 lets NumPy automatically determine the number of samples.

1 is the number of channels (for grayscale images).

32, 32 is the image height and width.

.astype(np.float32) ensures the data type is 32-bit float, which is required for most deep learning frameworks like PyTorch.

y = np.array(labels).astype(np.int64)

Converts your list of labels (labels) into a NumPy array.

.astype(np.int64) ensures the labels are 64-bit integers, which is the standard type for class labels in PyTorch.

Summary:

These lines prepare your image data and labels in the correct format and data type for training a convolutional neural network (CNN) using PyTorch.

kernel_size=3 means each filter is a 3x3 window.

Kernel size = size of the filter (e.g., 3x3 pixels)

The number of output channels (e.g., 16) means you get 16 feature maps, one for each filter.

1. Batch normalization is a technique used in deep learning to make training faster and more stable. Here's what it does:

Normalizes the output of a layer (like a convolutional layer) so that the mean is close to 0 and the standard deviation is close to 1, for each mini-batch during training.

Reduces internal covariate shift, which means it helps keep the distribution of activations more consistent as the network trains.

Allows higher learning rates and can speed up convergence.

Acts as a regularizer, sometimes reducing the need for dropout.

In your CNN, batch normalization is applied after each convolutional layer. This helps the network learn better and can improve both training speed and final accuracy.

- `class SimpleCNN(nn.Module)`: Defines a new neural network class that inherits from PyTorch's base `nn.Module`.
- `__init__`: The constructor sets up the layers:
  - `self.conv1`: 2D convolutional layer with 1 input channel (grayscale), 16 output channels, 3x3 kernel, and padding=1.
  - `self.bn1`: Batch normalization layer for the first convolutional layer.
  - `self.pool`: Max pooling layer with 2x2 window, reduces spatial size by half.
  - `self.conv2`: Second convolutional layer, takes 16 input channels, outputs 32, 3x3 kernel, padding=1.
  - `self.bn2`: Batch normalization layer for the second convolutional layer.
  - `self.fc1`: Fully connected (dense) layer, input size is 3288 (after two poolings on 32x32 input), outpu size is 128.
  - `self.fc2`: Output layer, 128 inputs to 10 outputs (for 10 classes).
  - `self.relu`: ReLU activation function.
  - `self.dropout`: Dropout layer for regularization (25% dropout rate).
- `forward(self, x)`: Defines the forward pass (how data flows through the network):
  1. Pass input through first conv layer, apply ReLU and batch normalization.
  2. Apply max pooling.
  3. Pass through second conv layer, apply ReLU and batch normalization.
  4. Apply max pooling again.
  5. Flatten the output to a vector.
  6. Pass through first fully connected layer, apply ReLU and dropout.
  7. Pass through output layer to get class scores.

This architecture is suitable for small grayscale images (like 32x32) and is a common starting point for image classification tasks.

** 2D convolutional layer with 1 input channel (grayscale), 16 output channels, 3x3 kernel, and padding=1.

Yes, that's correct! In a 2D convolutional layer like nn.Conv2d(1, 16, kernel_size=3, padding=1), the 16 output channels mean the layer uses 16 different filters (also called kernels).

Each filter learns to detect a different feature or pattern in the input image, such as:

Loopy or circular shapes

Vertical lines

Horizontal lines

Diagonal lines

Edges, corners, or other textures

After the convolution, you get 16 separate "feature maps" (one for each filter), which can be visualized as 16 different 2D arrays (or "boxes") showing where each filter detected its pattern in the image. These feature maps are then passed to the next layer for further processing.

So, your understanding is correct: 16 output channels = 16 filters, each learning to detect different visual features!

Training with Early Stopping

This cell adds early stopping to the CNN training loop. Training will stop if the validation loss does not improve for a set number of epochs (patience).

```
Epoch 11/50, Train Loss: 0.8141, Val Loss: 0.9935
Epoch 11/50, Train Loss: 0.8141, Val Loss: 0.9935
Epoch 12/50, Train Loss: 0.7863, Val Loss: 1.0013
Epoch 12/50, Train Loss: 0.7863, Val Loss: 1.0013
Epoch 13/50, Train Loss: 0.7630, Val Loss: 0.9714
Early stopping at epoch 13
Epoch 13/50, Train Loss: 0.7630, Val Loss: 0.9714
Early stopping at epoch 13
```

```
Test Accuracy: 0.6613
```

##

Little Bit Accuracy Increased after Using Batch Normalization

##How to Increase CNN Test Accuracy##

1. Data Augmentation

Add random transformations (rotation, flip, crop, etc.) to your training images to help the model generalize better.

2. Increase Model Complexity

Add more convolutional layers, increase the number of filters, or add more neurons to fully connected layers.

3. Train for More Epochs

If you are not overfitting, try increasing the number of epochs or patience for early stopping.

4. Tune Learning Rate

Try different learning rates (e.g., 0.0005, 0.0001) for the optimizer.

5. Batch Normalization

Add batch normalization layers after convolutions to stabilize and speed up training.

6. Regularization

Adjust dropout rate (e.g., 0.3 or 0.4) or add L2 regularization to the optimizer.

7. Use Pretrained Models

For small datasets, try transfer learning with a pretrained model (e.g., ResNet, VGG) using PyTorch’s torchvision.

8. Check Data Quality

Ensure your images and labels are correct and balanced across classes.

9. Hyperparameter Tuning

Experiment with batch size, optimizer type (SGD, RMSprop), and network architecture.

You can further add confusion matrix, class-wise accuracy, or visualize predictions as needed.

Data Augmentation for Improved Generalization

Let's use torchvision transforms to apply data augmentation to the training images. This can help the model generalize better and improve accuracy.

```
Test Accuracy with Data Augmentation: 0.1008
```

```
########################################################################################
```

After Using Data Augmentation Accuracy was droped heavely

Increasing Model Complexity: Deeper CNN

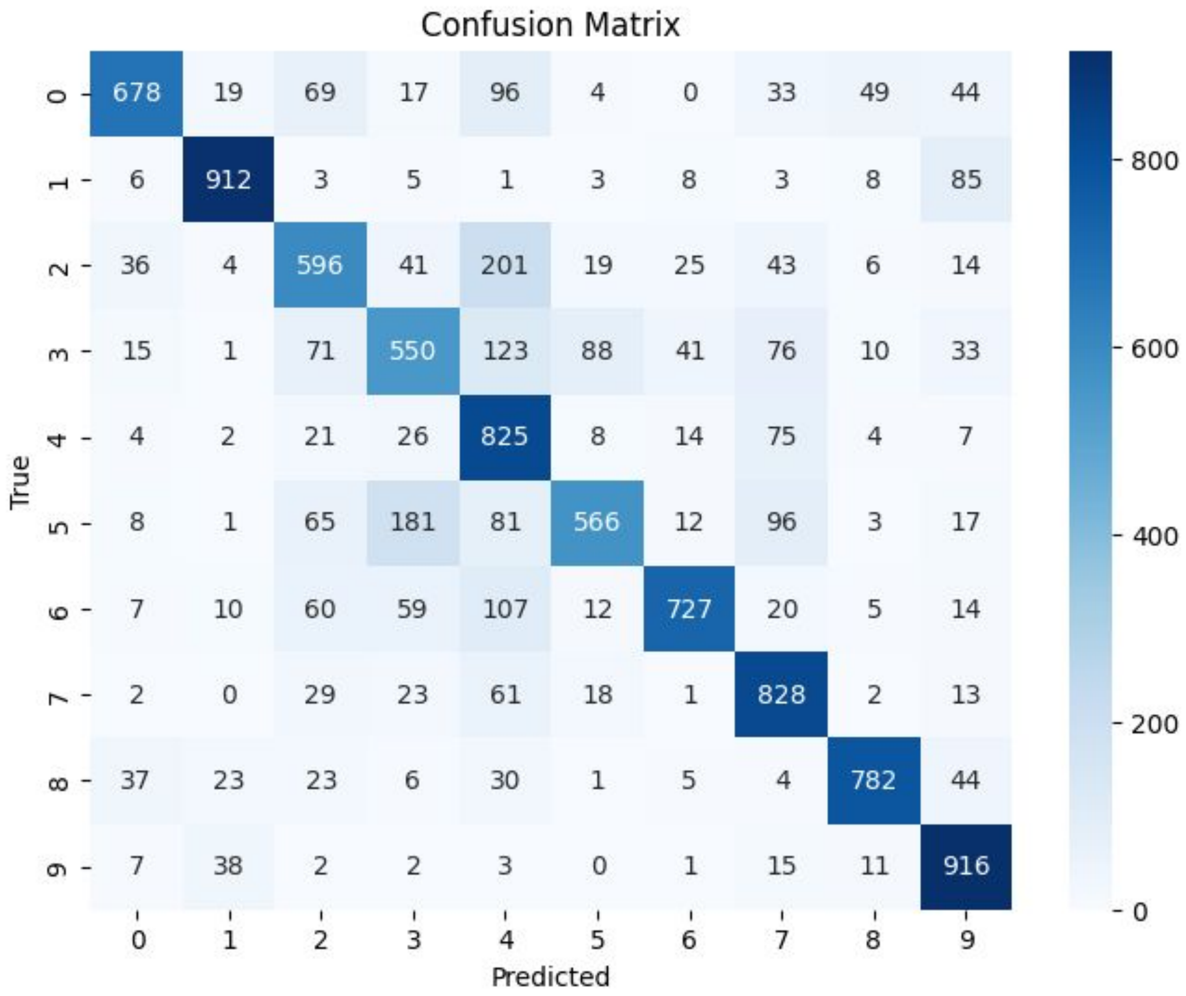
Let's increase the model complexity by adding more convolutional layers and filters to the CNN. This can help the model learn more complex features from the images.

```
Epoch 12/50, Train Loss: 1.0230, Val Loss: 0.8813
Epoch 13/50, Train Loss: 0.9957, Val Loss: 0.8562
Epoch 13/50, Train Loss: 0.9957, Val Loss: 0.8562
...
Epoch 26/50, Train Loss: 0.8583, Val Loss: 0.7582
Early stopping at epoch 26
Epoch 26/50, Train Loss: 0.8583, Val Loss: 0.7582
Early stopping at epoch 26
```

```
Test Accuracy (Complex CNN): 0.7380
```

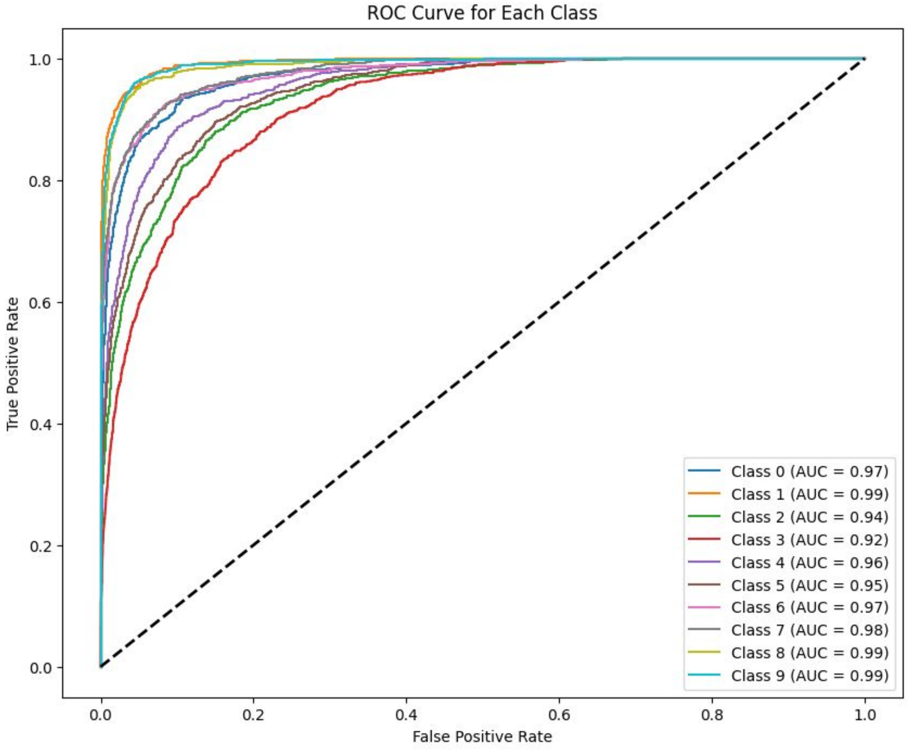
Classification Report:

	precision	recall	f1-score	support
0	0.85	0.67	0.75	1009
1	0.90	0.88	0.89	1034
2	0.63	0.61	0.62	985
3	0.60	0.55	0.57	1008
4	0.54	0.84	0.66	986
5	0.79	0.55	0.65	1030
6	0.87	0.71	0.78	1021
7	0.69	0.85	0.76	977
8	0.89	0.82	0.85	955
9	0.77	0.92	0.84	995
accuracy			0.74	10000
macro avg	0.75	0.74	0.74	10000
weighted avg	0.76	0.74	0.74	10000


Confusion Matrix
True
Predicted
0 1 2 3 4 5 6 7 8 9
678 19 69 17 96 4 0 33 49 44
6 912 3 5 1 3 8 3 8 85
36 4 596 41 201 19 25 43 6 14
15 1 71 550 123 88 41 76 10 33
4 2 21 26 825 8 14 75 4 7
8 1 65 181 81 566 12 96 3 17
7 10 60 59 107 12 727 20 5 14
2 0 29 23 61 18 1 828 2 13
37 23 23 6 30 1 5 4 782 44
7 38 2 2 3 0 1 15 11 916
0
200
400
600
800

ROC Curve, AUC Score, and Threshold Selection for Complex CNN

Plot ROC curves and calculate AUC for each class. Also, show how to adjust the threshold for a specific class.



```
Macro-average AUC: 0.9662
Micro-average AUC: 0.9676
Best threshold for class 0: 0.043 (Youden's J statistic)
```

- The ROC curve and AUC are plotted for each class.
- Macro and micro average AUC scores are printed.
- The best threshold for class 0 (using Youden's J statistic) is shown. You can adjust the threshold for othe classes similarly.

- The ROC curve and AUC are plotted for each class.
- Macro and micro average AUC scores are printed.
- The best threshold for class 0 (using Youden's J statistic) is shown. You can adjust the threshold for other classes similarly.

J = True Positive Rate (TPR) - False Positive Rate (FPR)

How is it taken?

For each possible threshold, you calculate TPR and FPR.

You compute J = TPR - FPR for each threshold.

The threshold with the highest J value is considered optimal, as it maximizes the difference between true positive rate and false positive rate.

Auc is 0.96 is great Because of Low False Positive Rate so its reduce false Alaram

Method	Save Command	Load Command	Notes
Entire model	`torch.save(model, "model.pth")`	`torch.load("model.pth")`	Easy but not flexible
Only model parameters	`torch.save(model.state_dict())`	`model.load_state_dict(torch.load())`	Recommended and more robust

Load the Entire Saved Model for Prediction

You can load the entire model (architecture + weights) using torch.load if you saved it with torch.save(model, ...).

Option 1: Load only the weights (recommended and safest)

✅ Solution 1 (Safe Way — Only if You Trust the Model File) Use the weights_only=False flag explicitly, like this:

This tells PyTorch to trust the file and load the full model (including custom classes).

✅ Use this only if:

You saved this model yourself.

You trust the source 100%.

- Both methods will allow you to make predictions.
- Loading only the model parameters (option 2) is recommended for most use cases.

Streamlit Application

Digit Classifier using CNN (PyTorch + state_dict)

Upload a grayscale image (32x32)

Drag and drop file here

Limit 200MB per file • PNG, JPG, JPEG

Browse files

download (2).jpg 6.7KB

The use_column_width parameter has been deprecated and will be removed in a future release. Please utilize the use_container_width parameter instead.

Uploaded Image

Predicted Class: **6 - Frog**

Thank You