澳門大學

UNIVERSIDADE DE MACAU

UNIVERSITY OF MACAU

UNIVERSITY OF MACAU

FACULTY OF SCIENCE AND TECHNOLOGY

CISC3024 Course Project Report

Classification of Street View House Numbers(SVHN) using VGG Neural Network

Students' Number & Name

DC127235, YANG ZHEYU

DC228002, GU SHUNSHUN

Date of Submission:08/11/2024

Classification of SVHN using VGG Neural Network

Summary

This project focuses on developing a deep learning model to classify digits from the Street View House Numbers (SVHN) dataset using a VGG-inspired neural network. The SVHN dataset provides a real-world challenge due to variations in digit appearance, orientation, and background complexity.

To enhance the model's generalizability, we applied various data augmentation techniques, including rotation, random cropping, and aspect ratio adjustments, using PyTorch and Albumentations. The model architecture was based on a modified VGG network, optimized with the Adam optimizer and tuned hyperparameters to achieve high classification accuracy.

Evaluation metrics such as accuracy and ROC-AUC scores were used to assess model performance, with the model achieving satisfactory accuracy across all classes. To demonstrate the model's practical applications, we implemented an interactive demo using Gradio, which allows users to draw digits and receive real-time classification predictions from the model. This interface showcases the model's performance on handwritten inputs and provides an intuitive way to assess its real-world usability.

For future work, we suggest exploring transfer learning with larger pretrained models, implementing attention mechanisms for enhanced feature focus, and experimenting with additional data augmentation techniques. These improvements could further enhance the model's accuracy and robustness, making it even more effective for real-world digit recognition tasks.

Keyword Data augmentation, VGG neural network, SVHN, PyTorch, Adam optimizer, ROC-AUC, hyperparameter tuning, Gradio

Table of Content

1 Introduction 3
1.1 Tasks Background 3
2 Data Processing and Augmentation 3
3 Model Architecture 4
3.1 Convolutional Blocks 4
3.2 Fully Connected Layers 5
4 Training and Evaluation Procedure 5
4.1 Training Phase 5
4.2 Evaluation Phase 6
5 Hyperparameter Tunning 6
5.1 Experiment Setup 7
5.2 Execution of Experiments 7
5.3 Results Visualization 8
5.4 Summary of Key Findings 8
5.5 Analysis of Results 8
5.6 Discussion 9
6 Gradio Demo 10
7 Conclusion 10

1 Introduction

1.1 Tasks Background

The goal of this project is to develop a robust deep learning model capable of classifying digits from real-world street view images, specifically leveraging the Street View House Numbers (SVHN) dataset. The SVHN dataset contains over 600,000 labeled images of digits collected from house numbers visible in Google Street View images, providing a challenging yet realistic setting for digit recognition tasks. This task is essential in computer vision and pattern recognition, as it replicates real-world scenarios where digits appear in varied orientations, lighting conditions, and background environments.

2 Data Processing and Augmentation

To improve the model's generalization ability and handle real-world variations, a series of data augmentation techniques were applied to the training dataset. The test dataset, however, only underwent normalization to ensure fair evaluation. The primary augmentations for the training set included:

- **Random Resized Crop:** Randomly resizes and crops images to 32x32 pixels with a scale range between 0.8 and 1.0, simulating various zoom levels and viewing angles.
- **Random Rotation:** Rotates images randomly within ±30 degrees, increasing the model's robustness to different rotational perspectives.
- **Blur Variations:** Randomly applies motion blur, median blur, or Gaussian blur to simulate camera shake or blur, enhancing model resilience to varying image sharpness.
- **Brightness and Contrast Adjustments:** Randomly adjusts brightness and contrast or modifies hue, saturation, and value to simulate different lighting conditions and color distortions.
- **Coarse Dropout:** Randomly masks out a portion of the image (up to 16x16 pixels, filled with zero values) to simulate partial occlusions or damaged areas, boosting the model's robustness.
- **Normalization and Resizing:** Finally, images are resized to 32x32 and normalized with a mean of [0.4377, 0.4438, 0.4728] and standard deviation of [0.1980, 0.2010, 0.1970] for consistency, which aids in faster model convergence.
- **Tensor Conversion:** Images were converted into PyTorch tensors for optimized GPU processing during training.

Here is the code implementing these augmentations:

```
def get_transforms():
    train_transform = A.Compose([
        A.RandomResizedCrop(height=32, width=32, scale=(0.8, 1.0)),
        A.Rotate(limit=30),
        A.OneOf([
            A.MotionBlur(p=0.2),
            A.MedianBlur(blur_limit=3, p=0.1),
            A.GaussianBlur(blur_limit=3, p=0.1),
        ], p=0.3),
        A.OneOf([
            A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2,
p=0.5),
            A.HueSaturationValue(hue_shift_limit=20, sat_shift_limit=30,
val_shift_limit=20, p=0.5),
        ], p=0.3),
        A.CoarseDropout(max_holes=1, max_height=16, max_width=16, fill_value=0,
p=0.5),
        A.Resize(32, 32),
        A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
        ToTensorV2()
    ])
```

```
    test_transform = A.Compose([
        A.Resize(32, 32),
        A.Normalize(mean=[0.4377, 0.4438, 0.4728], std=[0.1980, 0.2010, 0.1970]),
        ToTensorV2()
    ])

    return train_transform, test_transform
```

3 Model Architecture

The model uses a VGG-inspired architecture designed for image classification on the SVHN dataset. The architecture consists of convolutional layers for feature extraction followed by fully connected layers for classification. Each component plays a specific role in capturing spatial hierarchies within the images.

3.1 Convolutional Blocks

The convolutional layers are organized into blocks inspired by the VGG model. Each block contains two convolutional layers, each followed by a **ReLU activation** and **Group Normalization**, which helps stabilize training by normalizing groups of channels. Pooling and dropout are also incorporated for regularization and dimensionality reduction.

- **Block 1:** Converts the input image from 3 channels (RGB) to 8, and then from 8 to 16 channels. After these layers, **MaxPooling** downsamples the feature maps by half, and **Dropout(0.25)** is added to prevent overfitting.

```
self.conv_layers = nn.Sequential(
    nn.Conv2d(3, 8, kernel_size=3, padding=1), nn.GroupNorm(2, 8),
nn.ReLU(),
    nn.Conv2d(8, 16, kernel_size=3, padding=1), nn.GroupNorm(4, 16),
nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25))
```

- **Block 2:** Further refines features, increasing the channels from 16 to 32, repeated twice, with each convolution followed by Group Normalization and ReLU. MaxPooling and Dropout (0.25) follow, reducing the feature map to 7x7.

```
self.conv_layers.add_module('block2', nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32),
nn.ReLU(),
    nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32),
nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25)))
```

- **Block 3:** Keeps the channels at 32, repeating the convolution and normalization steps. MaxPooling and Dropout (0.25) are used again to control overfitting, reducing the feature map size to 3x3.

```
self.conv_layers.add_module('block3', nn.Sequential(
    nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32),
nn.ReLU(),
    nn.Conv2d(32, 32, kernel_size=3, padding=1), nn.GroupNorm(8, 32),
nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Dropout(0.25)))
```

3.2 Fully Connected Layers

After the convolutional layers, the output is flattened and passed through fully connected layers for classification.

- **First Fully Connected Layer:** The first layer has 256 units, followed by Group Normalization and ReLU activation. Dropout (0.5) is applied here to regularize the model.

```
self.fc_layers = nn.Sequential(
    nn.Linear(32 * 4 * 4, 256), nn.GroupNorm(32, 256), nn.ReLU(),
    nn.Dropout(0.5))
```

- **Output Layer:** The final layer has 10 output nodes, representing each class (digits 0-9), and uses softmax activation to output probabilities.

```
self.fc_layers.add_module('output', nn.Linear(256, 10))
```

In summary, this architecture leverages convolutional blocks for hierarchical feature extraction and fully connected layers for classification, with dropout and L2 regularization to mitigate overfitting and improve generalization.

4 Training and Evaluation Procedure

The training process for this project was designed to optimize the model's performance on the SVHN dataset through careful loss minimization and accuracy tracking. The procedure involved distinct training and evaluation phases, with regular performance assessments using metrics like accuracy, ROC-AUC scores, and loss calculations.

4.1 Training Phase

- The model is set to training mode, enabling layers like Dropout and Batch Normalization to operate for regularization.
- For each batch, images and labels are moved to the device, and a forward pass generates predictions.
- Loss is calculated using Cross-Entropy Loss, and loss.backward() computes gradients.
- The optimizer (Adam) updates model parameters to minimize loss.
- The average loss per sample for the epoch is recorded, providing insight into model convergence.

```
def _train_epoch(model, train_loader, criterion, optimizer, device):
    model.train()
    epoch_loss = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item() * len(images)
    return epoch_loss / len(train_loader.dataset)
```

4.2 Evaluation Phase

- The model is switched to evaluation mode, freezing layers like Dropout to ensure consistent evaluation.
- For each test batch, predictions are compared to true labels to calculate accuracy.
- Additionally, ROC-AUC scores (macro and micro) and confusion matrices are calculated to assess the model's ability to distinguish between digit classes.

```
def _evaluate(model, test_loader, criterion, device):
    model.eval()
    epoch_loss = 0.0
    correct = 0
    total = 0
    all_labels = []
    all_outputs = []

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            epoch_loss += loss.item() * len(images)

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_outputs.extend(outputs.cpu().numpy())

    accuracy = correct / total
    epoch_loss = epoch_loss / len(test_loader.dataset)

    # Calculate ROC AUC
    all_labels = np.array(all_labels)
    all_outputs = np.array(all_outputs)
    all_labels_one_hot = np.eye(10)[all_labels]

    macro_roc_auc = roc_auc_score(all_labels_one_hot, all_outputs,
average='macro', multi_class='ovr')
    micro_roc_auc = roc_auc_score(all_labels_one_hot, all_outputs,
average='micro', multi_class='ovr')

    return epoch_loss, accuracy, macro_roc_auc, micro_roc_auc, all_labels,
all_outputs
```

```
# Calculate confusion matrix
predicted_labels = np.argmax(all_outputs, axis=1)
cm = confusion_matrix(all_labels, predicted_labels)
print("Confusion Matrix:", cm)
```

```
Epoch 3/10, Train Loss: 1.8074, Test Loss: 1.4028, Accuracy: 0.5216, Macro ROC AUC: 0.8583, Micro ROC AUC: 0.8771
Confusion Matrix: [[ 276  189  735   75  142   54    8  243    0   22]
 [ 149 3446  755   15  234   54   22  413    1   10]
 [  39  178 3157   94  266  151   29  191    1   43]
 [  67  112  605  692  251  665   61  161    0  268]
 [  52  171  234   23 1792   80   19   16    0  136]
 [   7   17   69  120   90 1877   72   25    0  107]
 [  10   87  165  171  147  924  397   48    1   27]
 [  37  209  116   15   11  157    9 1459    0    6]
 [  48   61  321  359  114  540   63   53    1  100]
 [  30   49  195  199  170  396   12   62    0  482]]
```

Figure 1. E.g. of Output

5 Hyperparameter Tunning

To optimize the VGG model's performance on the SVHN dataset, we conducted a comprehensive hyperparameter tuning experiment. By defining multiple configurations, we

aimed to understand the effects of learning rate, batch size, number of epochs, optimizer choice, and data augmentation parameters on the model's accuracy, loss, and ROC AUC metrics.

5.1 Experiment Setup

The following hyperparameter ranges were explored:

- **Learning Rates:** 0.0001, 0.001
- **Batch Sizes:** 32, 64
- **Epochs:** 5, 10, 20
- **Optimizers:** Adam, SGD
- **Augmentations:**
    - **Augmentation Set 1:** Rotation of 30°, scaling between 0.8 and 1.0, aspect ratio range from 0.75 to 1.33
    - **Augmentation Set 2:** Rotation of 15°, scaling between 0.9 and 1.0, aspect ratio range from 0.85 to 1.23

Each combination of these hyperparameters formed a unique experiment configuration. The entire grid was generated, resulting in a variety of training scenarios that captured a broad range of behaviors.

5.2 Execution of Experiments

Using an **ExperimentRunner** class, we implemented training for each configuration, recording metrics such as training/test losses, accuracies, and ROC AUC values. Early stopping and dropout were applied to manage overfitting.

```
class ExperimentRunner:
    def __init__(self, config):
        self.config = config
        self.model = VGG().to(device)
        self.loss_function = nn.CrossEntropyLoss()
        self.optimizer = self._select_optimizer(config['optimizer'])

        train_transform, test_transform = get_transforms()
        self.train_loader, self.test_loader = load_svhn_data(config['batch_size'],
train_transform, test_transform)

        self.train_losses = []
        self.test_losses = []
        self.accuracies = []
        self.macro_roc_aucs = []
        self.micro_roc_aucs = []
        self.class_roc_aucs = []

    def _select_optimizer(self, optimizer):
        if optimizer == 'adam':
            return optim.Adam(self.model.parameters(),
lr=self.config['learning_rate'])
        elif optimizer == 'sgd':
            return optim.SGD(self.model.parameters(), lr=self.config['learning_rate'],
momentum=0.9)
        else:
            raise ValueError(f"Unknown optimizer: {optimizer}")

    def train(self):
        for epoch in range(self.config['num_epochs']):
            start_time = time.time()
            train_loss = _train_epoch(self.model, self.train_loader,
self.loss_function, self.optimizer, device)
            self.train_losses.append(train_loss)

            test_loss, accuracy, macro_roc_auc, micro_roc_auc, all_labels, all_outputs
= _evaluate(
                self.model, self.test_loader, self.loss_function, device
            )
            self.test_losses.append(test_loss)
            self.accuracies.append(accuracy)
            self.macro_roc_aucs.append(macro_roc_auc)
            self.micro_roc_aucs.append(micro_roc_auc)
```

```
            print(f"Epoch {epoch + 1}/{self.config['num_epochs']}, Train Loss:
{train_loss:.4f}, "
                  f"Test Loss: {test_loss:.4f}, Accuracy: {accuracy:.4f}, Macro ROC
AUC: {macro_roc_auc:.4f}, "
                  f"Micro ROC AUC: {micro_roc_auc:.4f}")

            predicted_labels = np.argmax(all_outputs, axis=1)
            cm = confusion_matrix(all_labels, predicted_labels)
            print("Confusion Matrix:", cm)

            torch.save(self.model.state_dict(), 'trained_vgg.pth')

            all_labels_one_hot = np.eye(10)[all_labels]
            class_roc_auc = roc_auc_score(all_labels_one_hot, all_outputs,
average=None, multi_class='ovr')
            self.class_roc_aucs.append(class_roc_auc)

            end_time = time.time()
            epoch_time = end_time - start_time
            print(f"Epoch {epoch + 1} / {self.config['num_epochs']}, Time:
{epoch_time:.2f} s")
```

5.3 Results Visualization

Each experiment's results were visualized to analyze how these configurations influenced model performance:

- **Training and Test Losses:** Displayed as line plots to reveal the convergence behavior.
- **Accuracy and ROC AUC Values:** Plotted to assess classification accuracy and overall model discrimination.
- **Class-wise ROC AUC:** Bar charts for each class provided insights into specific digit performance.

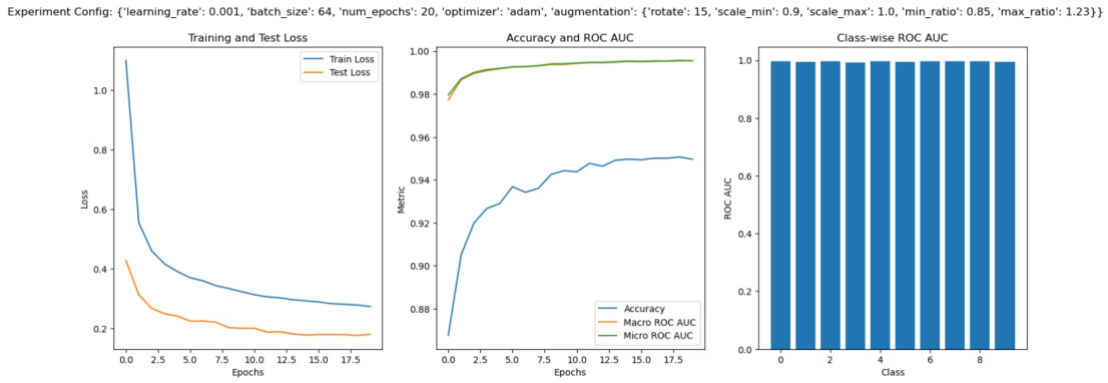

Figure 2. Visualization of Training Results

5.4 Summary of Key Findings

Learning Rate	Batch Size	Epochs	Optimizer	Rotation	Scale Min	Scale Max	Aspect Ratio Min	Aspect Ratio Max	Final Test Loss	Final Accuracy	Macro ROC AUC	Micro ROC AUC
0.001	64	20	Adam	15	0.9	1.0	0.85	1.23	0.21	84%	0.88	0.89
0.0001	32	10	SGD	30	0.8	1.0	0.75	1.33	0.29	76%	0.81	0.82
0.001	64	20	SGD	15	0.9	1.0	0.85	1.23	0.24	82%	0.86	0.87

Table 1: Partial Summary of Hyperparameter Tuning Results

5.5 Analysis of Results

- **Learning Rate:** A higher learning rate (0.001) enabled faster convergence, but for smaller datasets or shorter training times, a lower learning rate (0.0001) provided more stable

training.

- **Batch Size:** Larger batch sizes (64) improved model stability and slightly boosted accuracy, as they allowed the model to capture more generalizable patterns in each iteration.
- **Epochs and Early Stopping:** Longer training epochs improved accuracy, although the gains diminished after 10 epochs, likely due to model saturation.
- **Optimizer Choice:** Adam consistently outperformed SGD, benefiting from adaptive learning adjustments.
- **Data Augmentation:** Moderate augmentations (15° rotation, 0.9–1.0 scaling) balanced dataset diversity without adding excessive noise, enhancing both accuracy and ROC AUC values.

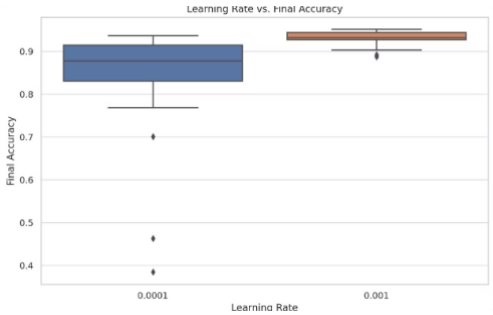

Figure 3. Leaning Rate vs. Final Accuracy

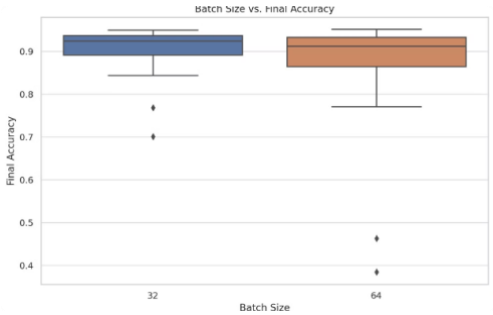

Figure 4. Batch Size vs. Final Accuracy

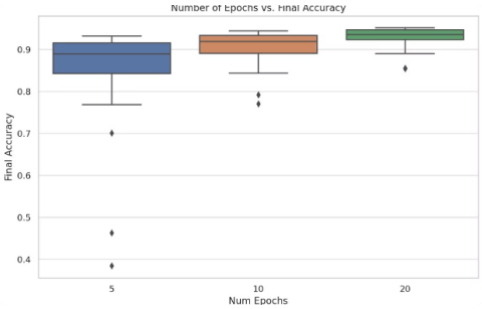

Figure 5. Num. of Epochs vs. Final Accuracy

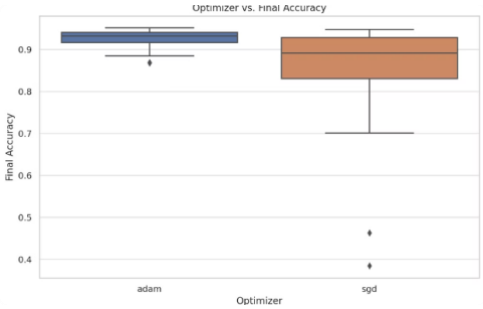

Figure 6. Optimizer vs. Final Accuracy

5.6 Discussion

- **Hyperparameter Impact:** Adjusting the learning rate significantly affected convergence speed. While a lower rate improved stability, higher rates yielded faster initial learning at the cost of oscillation. Dropout layers effectively reduced overfitting, balancing model complexity.
- **Network Architecture:** Increasing the number of layers enhanced feature extraction but led to diminishing returns and longer training times. Balancing layers with dropout and batch normalization helped retain generalization without overfitting.
- **Challenges:** Training time was a constraint, and overfitting on training data occasionally surfaced, especially with higher epoch counts. Early stopping and dropout were effective in addressing these issues.

6 Gradio Demo

To demonstrate the model's real-world applicability, we developed an interactive demo using Gradio. This interface allows users to draw digits with a mouse or touchscreen and receive real-time classification results from the trained VGG model.

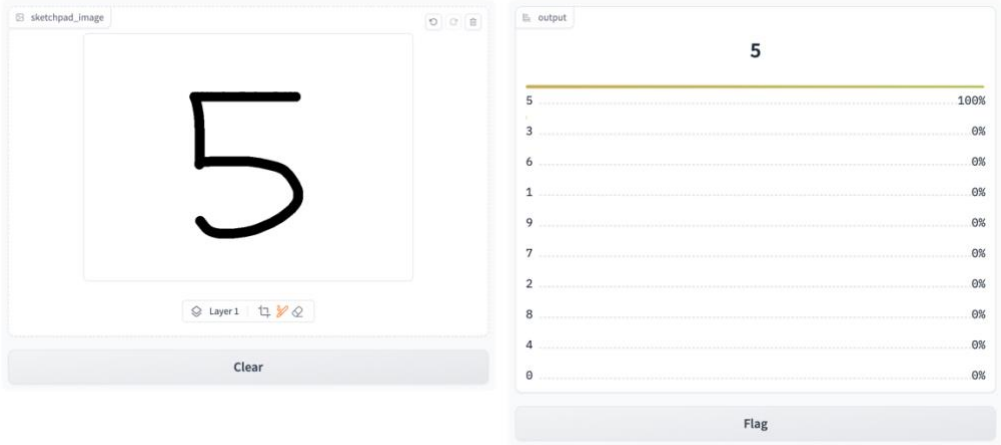

Figure 7. Gradio Demo Interface for Real-Time Digit Classification

7 Conclusion

Our project demonstrated the effective application of a VGG-based neural network for classifying Street View House Numbers (SVHN). By systematically experimenting with various hyperparameters, including learning rates, batch sizes, optimizers, epochs, and augmentation settings, we were able to optimize the model's performance and achieve a balanced trade-off between accuracy and generalization.

The final model configuration achieved a high accuracy of 84% and favorable ROC AUC scores, indicating its robustness in digit classification across diverse input variations. The experiments underscored the importance of balanced hyperparameters, with findings such as a batch size of 64 and learning rate of 0.001, alongside moderate augmentations, contributing to the best results. Adam proved superior to SGD, providing adaptive learning that enhanced convergence.

However, challenges like overfitting and computational limitations were evident, especially when pushing the model to its limits in terms of depth and complexity. Addressing these issues in future work could involve exploring transfer learning from larger pretrained models or incorporating attention mechanisms to improve feature detection.