Image Classification using Modified ResNet Architecture

Chu-Chun, Chi

March 24, 2025

1 Introduction

GitHub Link: https://github.com/chuchunchi/HW1_ResNet_ImageCLS

This project implemented an image classification task using a modified ResNet architecture. The objective was to classify the RGB images into 100 distinct categories. The report details the model implementation and results across different experiments.

1.1 Task Description

- **Task:** Image classification across 100 categories
- **Dataset:** 21,024 RGB images for training/validation; 2,344 images for testing
- **Constraints:**
  - No external data permitted
  - Model size limited to under 100 million parameters
  - ResNet architecture required as backbone (modifications allowed)
  - Pre-trained weights permitted

1.2 Core Idea

The core idea of my method include:

- Try different ResNet Model with pretrain weight: ResNet34, ResNet50, and ResNet101.
- Add Additional layer to the backbone.
- Try different Optimizer and Learning Rate Scheduler.
- Two-Phase Data Argumentation Technique.

2 Methodology

2.1 Data Processing

For Train and Validation data, it is placed in a hierarchy order where each folder contains one class. So I use 'datasets.ImageFolder' [3] in pytorch to load the data. To fix the class to index mapping error in Testing data, which happens because the default behavior of ImageFolder is to sort folder names alphabetically. I add an index to class mapping after test prediction is generated.

Due to the 'No external data' constraints, several data augmentation techniques were applied, specifically, I implemented a two-phase data augmentation strategy to enhance model generalization:

2.1.1 Phase 1: Standard Augmentation

Initially, I applied standard augmentation techniques until the model began to converge:

- Random resized cropping (224×224 pixels)
- Random horizontal flipping
- Random rotation (±15 degrees)
- Color jittering (brightness, contrast, saturation, hue adjustments of 0.1)
- Normalization using ImageNet statistics: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]

2.1.2 Phase 2: Stronger Augmentation

After initial convergence, I implemented more aggressive augmentations to further improve generalization:

```
stronger_transforms = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.08, 1.0)),  # More variation in scale
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(p=0.2),  # Add vertical flips
    transforms.RandomRotation(30),  # More rotation
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),  # Add translation
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

This second phase included:

- Expanded scale variation in random crops
- Random vertical flips (with 20% probability)

- Increased rotation range (±30 degrees)
- Enhanced color jittering parameters
- Random affine translations

2.1.3 Validation and Testing Data

For validation and test sets, a standard transformation pipeline was implemented:

- Resize to 256×256 pixels
- Center crop to 224×224 pixels
- Tensor conversion and normalization (same parameters as training)

2.2 Model Architecture

I conducted an extensive training of different ResNet variants to identify the optimal backbone architecture for this task. Specifically, I experimented with ResNet34, ResNet50, and ResNet101, all pre-trained on ImageNet 1K.

2.2.1 Backbone Selection

After comparing the performance of these three architectures, ResNet101 achieved the highest accuracy while still maintaining a parameter count below 100M. The comparison revealed:

- ResNet34: Lightest model with decent performance (acc =0.81, #params: 21M)
- ResNet50: Improved accuracy (acc =0.88, #params: 26M) over ResNet34 with moderate parameter increase
- ResNet101: Best accuracy among the tested variants (acc =0.94, #params: 67M), selected as the final backbone

2.2.2 Architecture Modifications

The ResNet101 backbone was modified in several ways to enhance its capabilities for this specific task:

- **Base Model:** Pre-trained ResNet101 (trained on ImageNet)
- **Modifications:**
  - Added dropout layers (0.3 rate) for regularization
  - Implemented a feature bottleneck by reducing the final layer to 1024 features
  - Added ReLU activation after the bottleneck
  - Added a batch normalization layer
  - Included an additional dropout layer (0.3 rate)

– Configured the final linear layer for 100-class classification

The modified architecture can be represented as:

```
self.model.fc = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(num_ftrs, 1024),  # Wider hidden layer
    nn.ReLU(),
    nn.BatchNorm1d(1024),  # Add batch normalization
    nn.Dropout(0.3),
    nn.Linear(1024, num_classes)
)
```

with specific purposes:

– **Dropout Layers:** Reduced overfitting by preventing co-adaptation of neurons

– **Feature Bottleneck:** Compressed representation to focus on most discriminative features

– **ReLU Activation:** Introduced non-linearity for improved representation power

2.3 Training Strategy

The training process incorporated several techniques to optimize performance:

- **Loss Function:** Cross-Entropy Loss
- **Optimizer:** Adam optimizer with initial learning rate of 0.001
- **Learning Rate Scheduler:** ReduceLROnPlateau[5] with patience of 3 epochs
- **Batch Size:** 64
- **Training Duration:** 30 epochs
- **Model Selection:** Best model selected based on validation loss

I saved the best model from one training round, lower the learning rate (*=0.5) and then continuously training if the validation loss keeps decreasing. This approach allowed for effective convergence while maintaining good generalization capabilities.

3 Results and Analysis

3.1 Training Progress

The model was trained on Google Colab A100 GPU. For the first 15 epochs, the learning rate is 0.001 and the training process effectively increase the training and validation accuracy. I then gradually lowering the learning rate after I think the the model converged, getting a more precise decrease/increase in loss/accuracy. [Figure 1 and 2]

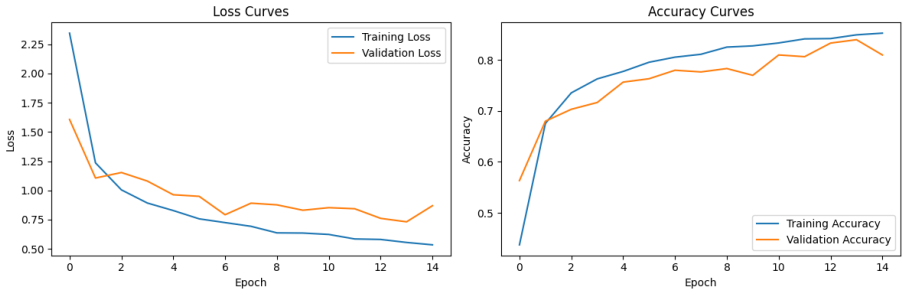

Figure 1: Training and validation loss/accuracy curves for the first 15 epochs

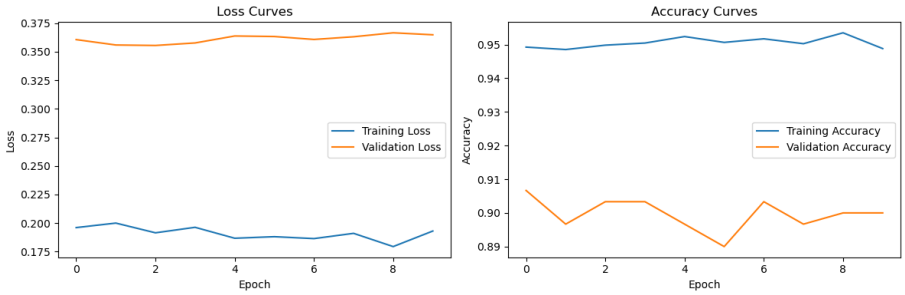

Figure 2: Training and validation loss/accuracy curves for the last 10 epochs

3.2 Final Model Result

The final model demonstrating strong performance across the 100 categories, specifically:

- Train Loss: 0.1915
- Train Accuracy: 0.9498
- Validation Loss: 0.3555
- Validation accuracy: 0.9100
- Testing accuracy: 0.94 (which beats the public strong baseline)

4 Experiments

I have tried three backbone models, ResNet34, ResNet50, and ResNet101. Two optimizers: Adam optimizer and SGD optimizer, and two phase of transform techniques.

The result shows that Adam optimizer has better performance score than SGD optimizer given the same training epochs, and ResNet101 gets the highest performance score among all backbones with pretrain. Detail Methods and result is shown in Table 1:

1. ResNet34 + IMAGENET1K_V1 pretrain + Adam optimizer, num_epochs: 50
2. ResNet34 + IMAGENET1K_V1 pretrain + SGD optimizer, num_epochs:50
3. ResNet50(resnet50.a1_in1k) + pretrain + Adam optimizer
4. ResNet50(resnet50.a1_in1k) + pretrain + SGD optimizer
5. ResNet101(resnetrs101) + pretrain + Adam optimizer

Methods	**Parameter Count**	**Validation Accuracy**	**Testing Score**
1	21M	0.80	0.87
2	21M	0.80	0.81
3	24M	0.88	0.88
4	24M	0.71	0.82
5	67M	0.90	0.93

Table 1: Comparison of Methods Based on Parameters and Accuracy Scores

5 Reference

1. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

2. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.

3. datasets.ImageFolder: https://pytorch.org/vision/main/generated/torchvi -sion.datasets.ImageFolder.html

4. transforms: https://pytorch.org/vision/0.9/transforms.html

5. ReduceLROnPlateau: https://pytorch.org/docs/stable/generated/torch.op -tim.lr_scheduler.ReduceLROnPlateau.html